Isabelle/C

Frédéric Tuong Burkhart Wolff

September 13, 2023

LRI, CNRS, CentraleSupélec, Université Paris-Saclay
bât. 650 Ada Lovelace, 91405 Orsay, France
frederic.tuong@lri.fr burkhart.wolff@lri.fr

In case that you consider citing Isabelle/C, please refer to [23].

Contents

1 A Conceptual Description of the Isabelle/C Package 7
- 1.1 Introduction . . . 10
- 1.2 Background: PIDE and the Isabelle Document Model . . . 11
  - 1.2.1 The PIDE Document Model . . . 12
  - 1.2.2 Some Basics of PIDE Programming . . . 13
- 1.3 The C11 Parser Generation Process and Architecture . . . 14
  - 1.3.1 Generating the AST . . . 15
  - 1.3.2 Constructing a Lexer for C11 . . . 18
  - 1.3.3 Generating the Shift-Reduce Parser from the Grammar . . . 18
- 1.4 Isabelle/C: Syntax Tests and Experimental Results . . . 19
  - 1.4.1 Preprocessing Lexical Conventions: Comments and Newlines . . . 19
  - 1.4.2 Preprocessing Side-Effects: Antiquoting Directives vs. Pure Annotations . . . 20
  - 1.4.3 A Validation via the seL4 Test Suite . . . 20
- 1.5 Generic Semantic Annotations for C . . . 21
  - 1.5.1 Binding Space of Annotation Environment . . . 22
  - 1.5.2 Navigation for Annotation Commands . . . 23
  - 1.5.3 Defining Annotation Commands . . . 24
  - 1.5.4 Evaluation Order . . . 25
- 1.6 Semantic Back-Ends . . . 26
  - 1.6.1 A Simple Typed Memory Model: Clean . . . 26
  - 1.6.2 The Case of AutoCorres . . . 28
  - 1.6.3 Yet Another C Back-End: Securify . . . 29
- 1.7 Conclusions . . . 29
- 1.8 Appendix . . . 30
  - 1.8.1 Portability of Isabelle/C's Generated Files . . . 30
  - 1.8.2 A Zoom on Citadelle's HOL Generator . . . 30
  - 1.8.3 Shift-Reduce Forest of a Parsing Execution . . . 31

2 A Resume on Isabelle/C: Commands, Control Attributes and Programming Infrastructure 35
- 2.1 Syntax Commands for Isabelle/C . . . 35
  - 2.1.1 Toplevel Commands and Control Attributes . . . 35
  - 2.1.2 Predefined Annotation Commands inside the C context (C Annotation commands) . . . 37
  - 2.1.3 Inner Directive Commands . . . 39
- 2.2 Quick Start (for People More Familiar with C than Isabelle) . . . 39

2.3 Case Study: Mapping on the Parsed AST via C99 ASTs 40
2.3.1 Prerequisites 42
2.3.2 Structure of *Isabelle-C.C-Parser-Language* 42
2.3.3 Rewriting of AST node 44
2.4 Case Study: Mapping on the Parsed AST via the `C11_Ast_Lib` 47
2.5 Known Limitations, Troubleshooting 48
2.5.1 The Document Model of the Isabelle/PIDE (applying since at least Isabelle 2019) 48
2.5.2 Parsing Error versus Parsing Correctness 51
2.5.3 Exporting C Files to the File-System 53

3 Annex I: The Commented Sources of Isabelle/C 55
3.1 Core Language: An Abstract Syntax Tree Definition (C Language without Annotations) 55
3.1.1 Loading the Generated AST 55
3.1.2 Basic Aliases and Initialization of the Haskell Library 59
3.2 A General C11-AST iterator. 61
3.3 Core Language: Lexing Support, with Filtered Annotations (without Annotation Lexing) 74
3.4 Parsing Environment 104
3.5 Core Language: Parsing Support (C Language without Annotations) . . . 117
3.5.1 Parsing Library (Including Semantic Functions) 118
3.5.2 Miscellaneous 133
3.5.3 Loading the Grammar Simulator 134
3.5.4 Loading the Generated Grammar (SML signature) 150
3.5.5 Overloading Grammar Rules (Optional Part) 151
3.5.6 Loading the Generated Grammar (SML structure) 156
3.5.7 Grammar Initialization 156
3.6 Annotation Language: Parsing Combinator 161
3.7 Annotation Language: Command Parser Registration 188
3.8 Evaluation Scheduling 192
3.8.1 Evaluation Engine for the Core Language 193
3.8.2 Full Evaluation Engine (Core Language with Annotations) 202
3.9 Interface: Inner and Outer Commands 208
3.9.1 Parsing Entry-Point: Error and Acceptance Cases 208
3.9.2 Definitions of C11 Directives as C-commands 218
3.9.3 Definitions of C Annotation Commands 220
3.9.4 Definitions of Outer Classical Commands 228
3.9.5 Term-Cartouches for C Syntax 230
3.9.6 C-env related ML-Antiquotations as Programming Support 230
3.9.7 The Standard Store C11-AST's generated from C-commands . . . 231
3.10 Support for Document Preparation: Text-Antioquotations. 232

4 Appendix II: Examples for C11 Lexis and Syntax; and Isabelle/C User-Interaction 237
4.1 Regular C Code . . . 237
4.1.1 Comments, Keywords and Pragmas . . . 237
4.1.2 Scala/jEdit Latency on Multiple Bindings . . . 238
4.1.3 Lexing and Parsing Obfuscated Sources . . . 239
4.2 Experiments with `../../src_ext/parser_menhir` . . . 240
4.2.1 Expecting to succeed . . . 240
4.2.2 Expecting to fail . . . 240

5 Appendix III: Examples for the SML Interfaces to Generic and Specific C11 ASTs 243
5.1 Access to Main C11 AST Categories via the Standard Interface . . . 243
5.1.1 Queries on C11-Asts via the iterator . . . 244
5.1.2 A small compiler to Isabelle term's. . . . 244
5.2 Late-binding a Simplistic Post-Processor for ASTs and ENVs . . . 245
5.2.1 Definition of Core Data Structures . . . 245
5.2.2 Registering A Store-Function in `C_Module.Data_Accept.put` . . . 246
5.2.3 Registering an ML-Antiquotation with an Access-Function . . . 246
5.2.4 Accessing the underlying C11-AST's via the ML Interface. . . . 246
5.3 Example: A Possible Semantics for *#include* . . . 247
5.4 Defining a C-Annotation Commands Language . . . 249
5.4.1 C Code: Various Annotated Examples . . . 250
5.4.2 Example: An Annotated Sorting Algorithm . . . 253
5.5 C Code: Floats Exist . . . 255

6 Appendix IV : Examples for Annotation Navigation and Context Serialization 257
6.1 Setup of ML Antiquotations Displaying the Environment (For Debugging) 257
6.2 Introduction to C Annotations: Navigating in the Parsing Stack . . . 258
6.2.1 Basics . . . 258
6.2.2 Erroneous Annotations Treated as Regular C Comments . . . 260
6.2.3 Bottom-Up vs. Top-Down Evaluation . . . 261
6.2.4 Out of Bound Evaluation for Annotations . . . 262
6.3 Reporting of Positions and Contextual Update of Environment . . . 262
6.3.1 Reporting: *typedef*, *enum* . . . 262
6.3.2 Continuation Calculus with the C Environment: Presentation in ML . . . 264
6.3.3 Continuation Calculus with the C Environment: Presentation with Outer Commands . . . 264
6.3.4 Continuation Calculus with the C Environment: Deep-First Nesting vs Breadth-First Folding: Propagation of `C_Env.env_lang` . . . 265
6.3.5 Continuation Calculus with the C Environment: Deep-First Nesting vs Breadth-First Folding: Propagation of `C_Env.env_tree` . . . 266
6.3.6 Reporting: Scope of Recursive Functions . . . 266

6.3.7 Reporting: Extensions to Function Types, Array Types 267
6.4 General Isar Commands 268
6.5 Starting Parsing Rule 268
6.5.1 Basics 268
6.5.2 Embedding in Inner Terms 269
6.5.3 User Defined Setup of Syntax 269
6.5.4 Validity of Context for Annotations 269
6.6 Scopes of Inner and Outer Terms 270
6.7 Calculation in Directives 271
6.7.1 Annotation Command Classification 271
6.7.2 Generalizing ML Antiquotations with C Directives 272
6.8 Miscellaneous 274

7 Examples from the F-IDE Paper **277**
7.1 Setup 277
7.2 Defining Annotation Commands 278
7.3 Proofs inside C-Annotations 279
7.4 Scheduling the Effects on the Logical Context 279
7.5 As Summary: A Spaghetti Language — Bon Appétit! 279

1 A Conceptual Description of the Isabelle/C Package

Abstract

We present a framework for C code in C11 syntax deeply integrated into the Isabelle/PIDE development environment. Our framework provides an abstract interface for verification back-ends to be plugged-in independently. Thus, various techniques such as deductive program verification or white-box testing can be applied to the same source, which is part of an integrated PIDE document model. Semantic back-ends are free to choose the supported C fragment and its semantics. In particular, they can differ on the chosen memory model or the specification mechanism for framing conditions.

Our framework supports semantic annotations of C sources in the form of comments. Annotations serve to locally control back-end settings, and can express the term focus to which an annotation refers. Both the logical and the syntactic context are available when semantic annotations are evaluated. As a consequence, a formula in an annotation can refer both to HOL or C variables.

Our approach demonstrates the degree of maturity and expressive power the Isabelle/PIDE subsystem has achieved in recent years. Our integration technique employs Lex and Yacc style grammars to ensure efficient deterministic parsing. We present two case studies for the integration of (known) semantic back-ends in order to validate the design decisions for our back-end interface.

Keywords: User Interface, Integrated Development, Program Verification, Shallow Embedding

1.1 Introduction

Recent successes like the Microsoft Hypervisor project [18], the verified CompCert compiler [19] and the seL4 microkernel [15, 16] show that the verification of low-level systems code has become feasible. However, a closer look at the underlying verification engines VCC [8], or Isabelle/AutoCorres [10] show that the road is still bumpy: the empirical cost evaluation of the L4.verified project [16] reveals that a very substantial part of the overall effort of about one third of the 28 man years went into the development of libraries and the associated tool-chain. Accordingly, the project authors [16] express the hope that these overall investments will not have to be repeated for "similar projects".

In fact, none of these verifying compiler tool-chains capture all aspects of "real life" programming languages such as C. The variety of supported language fragments seem to contradict the assumption that we will all converge to one comprehensive tool-chain soon. There are so many different choices concerning memory models, non-standard control flow, and execution models that a generic framework is desirable: in which verified compilers, deductive verification, static analysis and test techniques (such as [14], [1]) can be developed and used inside the Isabelle platform as part of an integrated document.

In this paper we present Isabelle/C [1], a generic framework in spirit similar to Frama-C [7]. In contrast to the latter, Isabelle/C is deeply integrated into the Isabelle/PIDE document model [25]. Based on the C11 standard (ISO/IEC 9899:2011), Isabelle/C parses C11 code inside a rich IDE supporting static scoping. SML user-programmed extensions can benefit from the parallel evaluation techniques of Isabelle. The plug-in mechanism of Isabelle/C can integrate diverse semantic representations, including those already made available in Isabelle/HOL [20]: AutoCorres [10], IMP2 [17], Orca [3], or Clean [24]. A particular advantage of the overall approach compared to systems like Frama-C or VCC is that all these semantic theories are conservative extensions of HOL, hence no axiom-generators are used that produce the "background theory" and the verification conditions passed to automated provers. Isabelle/C provides a general infrastructure for semantic annotations specific for back-ends, i.e. modules that generate from the C source a set of definitions and derive automatically theorems over them. Last but not least, navigation features of annotations make the logical context explicit in which theorems and proofs are interpreted.

The heart of Isabelle/C, the new **C**‹ .. › command, is shown in Figure 1.1. Analogously to the existing **ML**‹ .. › command, it allows for editing C sources inside the ‹ .. › brackets, where C code is parsed on the fly in a "continuous check, continuous build" manner. A parsed source is coloured according to the usual conventions applying for Isabelle/HOL variables and keywords. A static scoping analysis makes the bindings inside the source explicit such that editing gestures like hovering and clicking may allow the user to reveal the defining variable occurrences and C type information (see yellow sub-box in the screenshot Figure 1.1). The C source may contain comments to set up semantic back-ends. Isabelle/C turns out to be sufficiently efficient for C sources such as the seL4 project.

[1] The current developer snapshot is provided in https://gitlri.lri.fr/ftuong/isabelle_c.

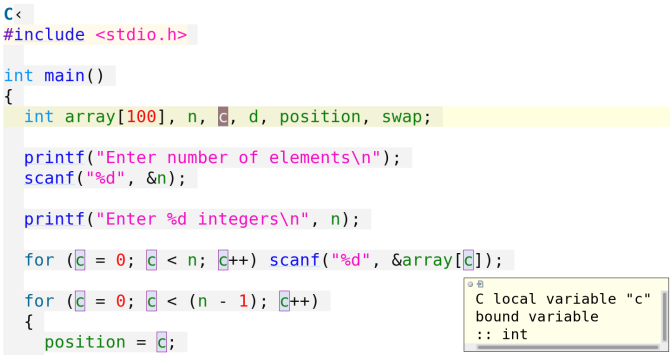


Figure 1.1: A C11 sample in Isabelle/jEdit

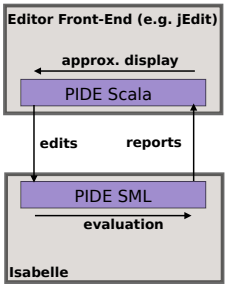


Figure 1.2: PIDE interaction

This paper proceeds as follows: in section 1.2, we briefly introduce Isabelle/PIDE and its document model, into which our framework is integrated. In section 1.3 and section 1.4, we discuss the build process and present some experimental results on the integrated parser. The handling of semantic annotations comments — a vital part for back-end developers — is discussed in section 1.5, while in section 1.6 we present some techniques to integrate back-ends into our framework at the hand of examples.

1.2 Background: PIDE and the Isabelle Document Model

The Isabelle system is based on a generic document model allowing for efficient, highly-parallelized evaluation and checking of its document content (cf. [2, 25, 26] for the fairly innovative technologies underlying the Isabelle architecture). These technologies allow for scaling up to fairly large documents: we have seen documents with 150 files be loaded in about 4 min, and individual files — like the x86 model generated from Antony Fox' L3 specs — have 80 kLoC and were loaded in about the same time.[2]

The PIDE (prover IDE) layer in Figure 1.2 consists of a part written in SML and another in Scala. Roughly speaking, PIDE implements "continuous build and continuous

[2]On a modern 6-core MacBook Pro with 32Gb memory, these loading times were counted *excluding* proof checking.

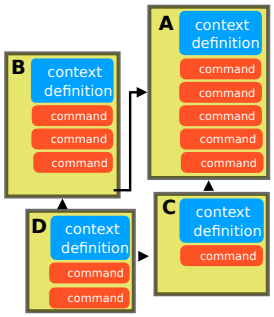


Figure 1.3: PIDE document model

```
theory C_Command
  imports C_Eval
  keywords "C" :: thy_decl
       and "C_file" :: thy_load
```

Figure 1.4: Header of a theory file in the Isabelle/C project

check" functionality over a textual albeit generic document model. It transforms user modifications of text elements in an instance of this model into increments — *edits* — and communicates them to the Isabelle system. The latter reacts by the creation of a multitude of light-weight reevaluation threads resulting in an asynchronous stream of *reports* containing *markup* that is used to annotate text elements in the editor front-end. For example, such markup is used to highlight variables or keywords with specific colours, to hyperlink bound variables to their defining occurrences, or to annotate type information to terms which become displayed by specific user gestures on demand (such as hovering). Note that PIDE is not an editor, it is the framework that coordinates these asynchronous information streams and optimizes their evaluation to a certain extent: outdated markup referring to modified text is dropped, and corresponding re-calculations are oriented to the user focus, for example. For PIDE, several editor applications have been developed, where Isabelle/jEdit (https://www.jedit.org) is the most commonly known. More experimental alternatives based on Eclipse or Visual Studio Code exist.

1.2.1 The PIDE Document Model

The document model foresees a number of atomic sub-documents (files), which are organized in the form of an acyclic graph in Figure 1.3. Such graphs can be grouped into sub-graphs called *sessions* which can be compiled to binaries in order to avoid long compilation times — Isabelle/C as such is a session. Sub-documents have a unique name (the mapping to file paths in an underlying file-system is done in an integrated build management). The primary format of atomic sub-documents is `.thy` (historically for "theory"), secondary formats can be `.sty`, `.tex`, `.c` or other sub-documents processed by Isabelle and listed in a configuration of the build system.

```
ML‹ val pos = @{here};
    val markup = Position.here pos;
    writeln ("And a link to the declaration\
             \ of 'here' is " ^ markup) ›
```

Figure 1.5: Referring to an editing source position with the ML antiquotation `@{here}`

A `.thy` file as in Figure 1.4 consists of a *context definition* and a body consisting of a sequence of *commands*. The context definition includes the sections **imports** and **keywords**. For example our context definition states that *C-Command* is the name of the sub-document depending on *C-Eval* which transitively includes the parser sources as (ML files) sub-documents, as well as the C environment and the infrastructure for defining C level annotations. *Keywords* like **C** or **C-file** must be declared before use.

For this work, it is vital that predefined commands allow for the dynamic creation of *user-defined* commands similarly to the definition of new functions in a shell interpreter. Semantically, commands are transition functions $\sigma \to \sigma$ where σ represents the system state called *logical context*. The logical context in interactive provers contains — among many other things — the declarations of types, constant symbols as well as the database with the definitions and established theorems. A command starts with a pre-declared keyword followed by the specific syntax of this command; an *evaluation* of a command parses the input till the next command, and transfers the parsed input to a transition function, which can be configured in a late binding table. Thus, the evaluation of the generic document model allows for user programmed extensions including IDE and document generation.

Note that the Isabelle platform supports multiple syntax embeddings, i.e. the possibility of nesting different language syntaxes inside the upper command syntax, using the ‹ .. › brackets (such parsing techniques will be exploited in section 1.5). Accordingly, these syntactic sub-contexts may be nested. In particular, in most of these sub-contexts, there may be a kind of semantic macro — called antiquotation and syntactically denoted in the format `@{name ‹.. ›}` — that has access to the underlying logical context. Similar to commands, user-defined antiquotations may be registered in a late-binding table. For example, the standard *term*-antiquotation in **ML**`‹ val t = @{term "3 +"} ›` parses the argument `"3 +"` with the Isabelle/HOL term parser, attempts to construct a λ-term in the internal term representation and to bind it to `t`; however, this fails (the plus operation is declared infix in logical context) and therefore the entire command fails.

1.2.2 Some Basics of PIDE Programming

A basic data-structure relevant for PIDE is *positions*; beyond the usual line and column information they can represent ranges, list of continuous ranges, and the name of the atomic sub-document in which they are contained. It is straightforward in Figure 1.5 to use the antiquotation `@{here}` to infer from the system lexer the actual position of the

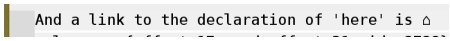


Figure 1.6: Output window showing the evaluation result of Figure 1.5 (the anchor of this hyperlink ⌂ targets at the position marked by `@{here}`)

antiquotation in the global document. The system converts the position to a markup representation (a string representation) and sends the result via `writeln` to the interface.

In return, the PIDE output window in Figure 1.6 shows the little house-like symbol ⌂, which is actually hyperlinked to the position of `@{here}`. The ML structures `Markup` and `Properties` represent the basic libraries for annotation data which is part of the protocol sent from Isabelle to the front-end. They are qualified as "quasi-abstract", which means they are intended to be an abstraction of the serialized, textual presentation of the protocol. A markup must be tagged with a unique id; this is done by the library `serial` function. Typical code for taking a string `cid` from the editing window, together with its position `pos`, and sending a specific markup referring to this in the editing window managed by PIDE looks like this:

```
ML‹ fun report_def_occur pos cid = Position.report pos (my_markup true cid (serial ()) pos) ›
```

Note that `my_markup` (not shown here) generates the layout attributes of the link and that the `true` flag is used for markup declaring `cid` as a defining occurrence, i.e. as *target* (rather than the *source*) in the hyperlink animation in PIDE.

1.3 The C11 Parser Generation Process and Architecture

Isabelle uses basically two parsing technologies:

1. Earley parsing [9] intensively used for mixfix-syntax denoting λ-terms in mathematical notation,

2. combinator parsing [12] typically used for high-level command syntax.

Both technologies offer the dynamic extensibility necessary for Isabelle as an interactive platform geared towards incremental development and sophisticated mathematical notations. However, since it is our goal to support *programming languages* in a fast parse-check-eval cycle inside an IDE, we opt for a Lex and Yacc deterministic grammar approach. It turns out the resulting automata based parser performs well enough for our purpose; the gain in performance is discussed in the next section.

In the following, we describe a novel technique for the construction and integration of this type of parser into the Isabelle platform. Since it is mostly relevant for integrators copying our process to similar languages such as JavaScript or Rust [3], users of the

[3] E.g. http://hackage.haskell.org/package/language-javascript or http://hackage.haskell.org/package/language-rust

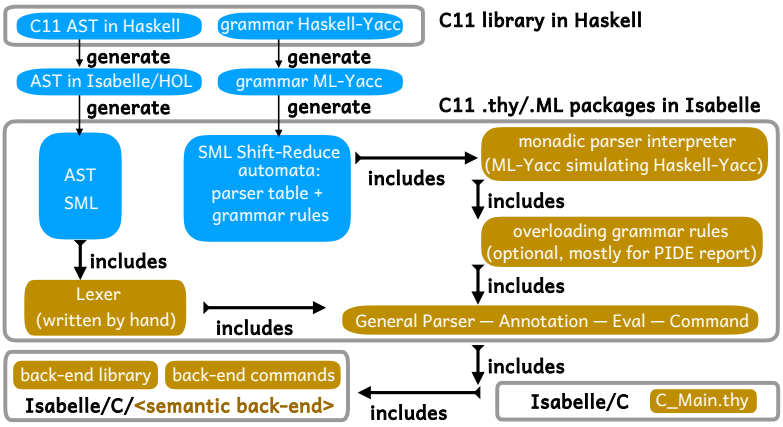


Figure 1.7: The architecture of Isabelle/C

Isabelle/C platform may skip this section: for them, the take-home message is that the overall generation process takes about 1 hour, the compilation of the generated files takes 15s, and that the generated files should be fairly portable to future Isabelle versions.[4]

We base our work on the C11 parsing library http://hackage.haskell.org/package/language-c implemented in Haskell by Huber, Chakravarty, Coutts and Felgenhauer; we particularly focus on its open-source Haskell Yacc grammar as our starting point. We would like to emphasize that this is somewhat arbitrary, our build process can be easily adapted to more recent versions when available.

The diagram in Figure 1.7 presents the architecture of Isabelle/C. The original Haskell library was not modified, it is presented in blue together with generated sources, in particular the final two blue boxes represent about 11 kLoC. In output, the glue code in brown constitutes the core implementation of Isabelle/C, amounting to 6 kLoC (without yet considering semantic back-ends).

1.3.1 Generating the AST

In the following, we refer to *languages* by $\mathcal{L}$, $\mathcal{I}$. The notation $\mathrm{AST}^{\mathcal{L}}_{\mathcal{I}}$ refers to abstract syntaxes for language $\mathcal{L}$ implemented in language $\mathcal{I}$. For example, we refer by $\mathrm{AST}^{\mathrm{C11}}_{\mathrm{ML}}$ to an AST implementation of C11 implemented in SML. Indices will be dropped when no confusion arises, or to highlight the fact that our approach is sufficiently generic.

For our case, we exploit that from a given Haskell source $\mathrm{AST}_{\mathrm{HS}}$, Haskabelle generates to a maximum extent an Isabelle/HOL theory. Via the Isabelle code generator, an $\mathrm{AST}_{\mathrm{ML}}$ can be obtained from a constructive $\mathrm{AST}_{\mathrm{HOL}}$ representation. Ultimately, the process to compile $\mathrm{AST}_{\mathrm{HS}}$ to $\mathrm{AST}_{\mathrm{ML}}$ is done only once at build time, it comprises:

[4] In our experiments, files generated with a version i of Isabelle natively work when loaded with a more recent version j, for at least any versions i and j satisfying $2016 \leq i \leq j \leq 2019$ (at the time of writing).

1. the generation of AST_{HOL} from AST_{HS}, represented as a collection of **datatype**,

2. the execution of the **datatype** theory for AST_{HOL} and checking of all their proofs,

3. the generation of an AST_{ML} from AST_{HOL}.

The choice on getting an Isabelle/HOL theory in the intermediate step allows us to make additional checks on the imported AST (e.g. non-emptiness of the imported types), as well as to use it as a first trusted basis point for further translations and reasoning activities (in subsection 1.4.3). However, relying on an intermediate AST_{HOL} in the above process is technically challenging for practical reasons. This is related to the prerequisites any HOL types must satisfy before being integrated in Isabelle: proofs of generated rules might take some time, or additional theorems must be proved before one can invoke the ML code generator on the types...

We present in the next descriptions below the main encountered corner cases, and how we solved them.

Maintenance of Haskabelle

The generation time for Step 1 involving Haskabelle is almost irrelevant (a few seconds), but first requires to significantly update Haskabelle from an abandoned Isabelle2016 distribution.[5]

Haskabelle is implemented in Haskell and mainly relies on a meta-programming library haskell-src-exts.[6] The design of this library drastically changed in 2016, making a new port of Haskabelle proportionally difficult. This sole reason has motivated the Isabelle developers of not maintaining anymore Haskabelle in the Isabelle distributions (it is discarded since Isabelle2016). The interested reader can find near our Isabelle/C source our renovated version of Haskabelle compatible with the latest official versions of Isabelle.

Limitation of Haskabelle on certain AST_{HS}

Since the support of Haskabelle on Haskell source is not complete, there would possibly remain Haskell construction features used by AST_{HS} left for an additional implementation task. For example, for the case of AST^{C11}, we have brought support for handling mutually recursive type declarations, particularly in the case where such type declarations are made by interleaving several Haskell keywords `type` and `data`. (The case of a mutually type declarations performed by only using several `data` was already supported.)

[5] See e.g. https://lists.cam.ac.uk/mailman/htdig/cl-isabelle-users/2016-December/msg00096.html and https://mailmanbroy.informatik.tu-muenchen.de/pipermail/isabelle-dev/2016-October/015677.html.

[6] https://hackage.haskell.org/package/haskell-src-exts

Resources needed to execute the generated HOL datatype theory

To our knowledge, our chosen AST^{C11} as importation has the most enormous size among any existing ASTs, e.g., ever published in the Isabelle AFP repository (more than 350 generated constants). Even if the generation with Haskabelle in Step 1 just takes a few seconds, Step 2 becomes much longer to execute: the generated **datatype**-theory correctly terminates only after 17 hours of elapsed time (or 70 hours of cpu time, with a gain factor of 4). Alternatively, one can skip the execution of proofs by activating the Isabelle *quick−and−dirty* option: this makes the resulting theory taking 40 min. A main reason for the resource issue is the 20 mutually recursive sub-definitions of AST_{HOL} (leading meanwhile to 6275 generated theorems). It is well known that large mutually recursive datatypes are impacting the resource performance, other projects encountered similar difficulties, see for instance https://lists.cam.ac.uk/pipermail/cl-isabelle-users/2016-March/msg00034.html and https://lists.cam.ac.uk/pipermail/cl-isabelle-users/2017-April/msg00000.html.

Of course, in the best scenario one could imagine the generator at Step 1 executing a more optimized algorithm to automatically remove any problematic mutual recursions as far as it is decidable (after the execution of Haskabelle, or possibly during its call). In general, we are tending towards generic solutions targeting modifications on the generator side (at Step 1), rather than modifying the AST or grammar sources by hand.

The extraction of HOL constructors versus the extraction of HOL constants

The code generator of Isabelle generates in ML all **datatype** constructors in a cartesian products form, while regular HOL **definition** constants are extracted as curried. Since an interoperable form is important for grammar composition in subsection 1.3.3; combining Haskabelle with Citadelle [22] in Step 1 allows for configuring Citadelle to curry all constructors, based on Haskabelle's output. For example, the last two definitions are the lines automatically added by Citadelle (for each **datatype** received from Haskabelle):

datatype 'a CStatement = *CLabel0 Ident 'a CStatement 'a CAttributes 'a*
| *CGoto0 Ident 'a*
definition CLabel = *CLabel0*
definition CGoto = *CGoto0*

Time to generate to SML

We conclude with Step 3 by using the Isabelle code generator. This takes about 20 min: its generation time is depending in a major part on respective theorems generated by the **datatype**-theory.

1.3.2 Constructing a Lexer for C11

We decided against the option of importing the equivalent Haskell lexer, as it is coming under-developed compared to the existing PIDE lexer library, natively supporting Unicode-like symbols (mostly for annotations). Using a more expressive position data-structure, our C lexer is also compatible with the native ML lexer regarding the handling of errors and backtracking (hence the perfect fit when nesting one language inside the other). Overall, the modifications essentially boil down to taking an extreme care of comments and directives which have intricate lexical conventions (see subsection 1.4.1).

1.3.3 Generating the Shift-Reduce Parser from the Grammar

In the original C11 library, together with AST_{HS}, there is a Yacc grammar file $\text{G}_{\text{HS-YACC}}$ included, which we intend to use to conduct the C parsing. However due to technical limitations of Haskabelle (and advanced Haskell constructs in the associated G_{HS}), we do not follow the same approach as subsection 1.3.1. Instead, an ultimate grammar G_{ML} is obtained by letting ML-Yacc participate in the generation process. In a nutshell, the overall grammar translation chain becomes: $\text{G}_{\text{HS-YACC}} \longrightarrow_{\text{HS}} \text{G}_{\text{ML-YACC}} \longrightarrow_{\text{ML}} \text{G}_{\text{ML}}$.

$\longrightarrow_{\text{HS}}$ is implemented by modifying the Haskell parser generator Happy, because Happy is already natively supporting the whole $\mathcal{L}_{\text{HS-YACC}}$. Due to the close connection between Happy and ML-Yacc, the translation is even almost linear. However cares must be taken while translating monadic rules of $\text{G}_{\text{HS-YACC}}$, as $\mathcal{L}_{\text{ML-YACC}}$ does not support such rules. Indeed, rules of $\mathcal{L}_{\text{HS-YACC}}$ can be either defined as being non-monadic (like $\mathcal{L}_{\text{ML-YACC}}$), or set to explicitly perform a monadic action.[7] In G^{C11}, monadic rules are particularly important for scoping analyses, or while building new informative AST nodes (in contrast to disambiguating non-monadic rules, see the difference between @ and & as used in section 1.5).[8] Note that $\longrightarrow_{\text{HS}}$ is also in charge of translating each rule code from Haskell to ML.

After executing $\longrightarrow_{\text{HS}}$, we obtain $\text{G}_{\text{ML-YACC}}$, but applying ML-Yacc $\longrightarrow_{\text{ML}}$ on $\text{G}_{\text{ML-YACC}}$ is not enough: even if the act of getting an efficient Shift-Reduce automaton G_{ML} is immediate, we needed to substantially modify the own grammar interpreter of ML-Yacc to implement all features of $\mathcal{L}_{\text{HS-YACC}}$ presented as used in $\text{G}_{\text{HS-YACC}}$.[9] Besides the grammar interpreter of ML-Yacc, we also modified $\longrightarrow_{\text{ML}}$ to generate additional meta-programming functions recording the types of all intermediate parsing values

[7] Syntactically, the difference relies in the presence of a particular flag before writing the rule code, namely `%`: https://www.haskell.org/happy/doc/html/sec-monads.html

[8] To add support of monadic rules in $\mathcal{L}_{\text{ML-YACC}}$, we adopt the same linguistic strategy as $\mathcal{L}_{\text{HS-YACC}}$: by letting the possibility to use a special string prefix in front of each rule code. Furthermore, to not modify the syntactic range of $\mathcal{L}_{\text{ML-YACC}}$, the string prefix can be directly written as *annotated* in rule code, for instance under the form of a special ML comment `(*%*)`.

[9] For example, for the case of monadic rules, this means to implement the necessary monadic optimization in the grammar interpreter. Globally, the default behavior of ML-Yacc is still preserved, except that we implemented the monadic support in ML-Yacc by overriding how certain native commands of ML-Yacc were functioning: for example `%arg` is now used to denote an implicit monadic state, and `%pure` to explicitly trigger the generation of monadic code, see https://www.cs.princeton.edu/~appel/modern/ml/ml-yacc/manual.html#section12.

```
C — ‹Nesting of comments› ‹
/* inside /* inside */ int a = "outside";
// inside // inside until end of line
int a = "outside";
/* inside
  // inside
inside
*/ int a = "outside";
// inside /* inside until end of line
int a = "outside";
›
```

Figure 1.8: Lexing convention for comments, illustrated in the CPP documentation

that might be ever present in the parser stack, see section 1.5. Later at parsing time, the necessary generated functions will be called to perform automatic cast operations (in subsection 1.5.4).

1.4 Isabelle/C: Syntax Tests and Experimental Results

The question arises, to what extent our construction provides a faithful parser for C11, and if Isabelle/C is sufficiently stable and robust to handle real world sources. A related question is the treatment of `cpp` preprocessing directives: while a minimal definition of the preprocessor is part of C standards since C99, practical implementations vary substantially. Moreover, `cpp` comes close to be Turing complete: recursive computations can be specified, but the expansion strategy bounds the number of unfolding. Therefore, a complete `cpp` reimplementation contradicts our objective to provide efficient IDE support inside Isabelle. Instead, we restrict ourselves to a common subset of macro expansions and encourage, whenever possible, Isabelle specific mechanisms such as user programmed C annotations. C sources depending critically on a specific `cpp` will have to be processed outside Isabelle.[10]

1.4.1 Preprocessing Lexical Conventions: Comments and Newlines

A very basic standard example taken from the GCC / CPP documentation [11] in Figure 1.8 shows the quite intricate mixing of comment styles that represents a challenge for our C lexer. A further complication is that it is allowed and common practice to use backslash-newlines $\boxed{\backslash\hookleftarrow}$ *anywhere* in C sources, be it inside comments, string denotations, or even regular C keywords like $i\boxed{\backslash\hookleftarrow}n\boxed{\backslash\hookleftarrow}t$ (see also Figure 1.12).

In fact, many C processing tools assume that all comments have already been removed via `cpp` before they start any processing. However, annotations in comments carry relevant information for back-ends as shown in section 1.5. Consequently, they must be explicitly represented in $\mathrm{AST}^{\mathrm{C11}}_{\mathrm{ML}}$, whereas the initial $\mathrm{AST}^{\mathrm{C11}}_{\mathrm{HS}}$ is not designed to carry such extra information. Annotations inside comments may again contain structured

[10] Isabelle/C has a particular option to activate (or not) an automated call to `cpp` before any in-depth treatment.

[11] https://gcc.gnu.org/onlinedocs/cpp/Initial-processing.html

information like programming code, formulas, and proofs, which implies the need for nested syntax. Fortunately, Isabelle is designed to manage multiple parsing layers with the technique of *cascade sources* [12] (see also Figure 1.10). We exploit this infrastructure to integrate back-end specific syntax and annotation semantics based on the parsing technologies available.

1.4.2 Preprocessing Side-Effects: Antiquoting Directives vs. Pure Annotations

For the mentioned reasons, we present how directives are classified and how some basic support can be implemented. As directives and preprocessing constitute a major chapter in the C standard, we only support very specific usages, but do not exclude statically typed alternatives to directives via user-programmed antiquotations. Still, in comparison with *comments* which can be safely removed without affecting the meaning of C code, *directives* are semantically relevant for compilation and evaluation.

1. Classical directives: *#define x TOKS* makes any incoming C identifier x be replaced by some *arbitrary* tokens *TOKS*, even when included via the *#include* directive.

2. Typed (pseudo-)directives as commands: It is easy to overload or implement a new *#define′* acting only on a decided subset of well-formed *TOKS*. There are actually no differences between Isabelle/C directives and Isabelle commands: both are internally of type $\sigma \to \sigma$ (see section 1.2).

3. Non-expanding annotations: Isabelle/C annotations /∗@ $\mathcal{L}_{\text{annot}}$ ∗/ or //@ $\mathcal{L}_{\text{annot}}$ can be freely intertwined between other tokens, even inside directives. In contrast to (antiquoting) directives and similarly as C comments, their designed intent is to not modify the surrounding parsing code. In section 1.5, we will address the issue of separation and interaction of plugged-in annotations, as well as the possibility of nesting annotations.

A limitation of Isabelle and its current document model is that there is no way for user programmed extensions to exploit implicit dependencies between sub-documents. Thus, a sub-document referred to via *#include <some-file>* will not lead to a reevaluation of a **C**‹ .. › command whenever modified. (The only workaround is to open all transitively required sub-documents *by hand*.)

1.4.3 A Validation via the seL4 Test Suite

The AutoCorres environment contains a C99 parser developed by Michael Norrish [15]. Besides a parser test-suite, there is the entire seL4 codebase (written in C99) which has been used for the code verification part of the seL4 project. While the parser in itself represents a component belonging to the trusted base of the environment, it is

[12] http://isabelle.in.tum.de/repos/isabelle/file/83774d669b51/src/Pure/General/source.ML

arguably the most tested parser for a semantically well-understood translation in a proof environment today.

It is therefore a valuable reference for a comparison test, especially since AST^{C99} and AST^{C11} are available in the same implementation language. From $\text{AST}^{\text{C11}}_{\text{HOL}}$ to $\text{AST}^{\text{C99}}_{\text{HOL}}$ we construct an abstraction function $C^{\downarrow}$. Its definition is rather straightforward but tedious and long: the main problem is the handling of C *struct*s where some thoughts have to be spent on the flattening of the recursive tree structure. A detailed description of $C^{\downarrow}$ is out of the scope of this paper; we would like to mention that it was 4 man-months of work due to the richness of AST^{C11}. This HOL function can be used for formal proofs over a preserving subset translation; however, so far, we only use it for code reflection into ML. As such, the abstraction function $C^{\downarrow}$ is at the heart of the AutoCorres integration into our framework described in subsection 1.6.2. Note that AST^{C99} seems to be already an abstraction compared to the C99 standard. This gives rise to a particular testing methodology: we can compile the test suites as well as the seL4 source files by both ML parsers $\text{PARSE}^{\text{C99}}_{stop}$ and $\text{PARSE}^{\text{C11}}_{report}$, abstract the output of the latter via $C^{\downarrow}$ and compare the results.

Our test establishes that both parsers agree on the entire seL4 codebase. However trying to compare the two parsers using other criteria is not possible, for example we had to limit ourselves to C programs written in a subset of C99. Fundamentally, the two parsers are achieving different tasks: the one of PARSE_{stop} is to just return a parsed AST. In contrast, PARSE_{report} intends to maximize markup reporting, irrespective of a final parsing success or failure, and reports are provided in parallel during its (monadic) parsing activity. Thus, in the former scenario, the full micro-kernel written in 26 kLoC can be parsed in 0.1s. In the latter, all reports we have thought helpful to implement are totally rendered before 20s. Applying $C^{\downarrow}$ takes 0.02 seconds, so our PARSE_{report} gives an average of 2s for a 2-3 kLoC source. By interweaving a source with proofs referring to the code elements, the responsiveness of PIDE should therefore be largely sufficient. In principle, there is an important potential for further optimizations by more incremental forms of reporting that follow the current focus of a user window. However we adopted the same strategy of reporting as **ML** and general Isabelle commands, for which incremental operations seem to be limited inside the atomic level of a command.

1.5 Generic Semantic Annotations for C

With respect to interaction with the underlying proof-engine, there are essentially two lines of thought in the field of deductive verification techniques:

1. either programs and specifications — i.e. the pre- and post-condition contracts — are clearly separated, or

2. the program is annotated with the specification, typically by using some form of formal comment.

```
C ‹
#define SQRT_UINT_MAX 65536
/*@ lemma uint_max_factor [simp]:
      "UINT_MAX = SQRT_UINT_MAX * SQRT_UINT_MAX - 1"
    by (clarsimp simp: UINT_MAX_def SQRT_UINT_MAX_def)
*/›
```

Figure 1.9: C11 code enriched with HOL proofs

Of course, it is possible to inject the essence of annotated specifications directly into proofs, e.g. by instantiating the *while* rule of the Hoare calculus by the needed invariant inside the proof script. The resulting clear separation of programs from proofs may be required by organisational structures in development projects. However, in many cases, modelling information may be interesting for programmers, too. Thus, having pre- and post-conditions locally in the source close to its point of relevance increases its maintainability. It became therefore common practice to design languages with annotations, i.e. structured comments *inside* a programming source. Examples are ACSL standardized by ANSI/ISO (see https://frama-c.com/download/acsl.pdf) or UML/OCL [4] for static analysis tools. When tackling with concurrency [21], other forms of annotation support are required, possibly in assembly code included in C11 code.[13] Isabelle/C supports both the inject-into-proof style and annotate-the-source style in its document model; while the former is kind of the default, we address in this section the necessary technical infrastructure for the latter.

1.5.1 Binding Space of Annotation Environment

Generally speaking, a generic annotation mechanism which is sufficiently expressive to capture idioms used in, e.g., Frama-C, Why3, or VCC is more problematic than one might think. Consider this:

```
for (int i = 0; i < n; i++) a+= a*i /*@ annotation */
```

To which part of the AST does the annotation refer? To i? $a{*}i$? The assignment? The loop? Some verification tools use prefix annotations (as in Why3 for procedure contracts), others even a kind of parenthesis of the form:

```
/*@ annotation_begin */ ... /*@ annotation_end */
```

The matter gets harder since the C environment — a table mapping C identifiers to their type and status — changes according to the reference point in the AST. This means that the context relevant to type-check an annotation such as /*@ *assert* ‹a > i› */ strongly differs depending on the annotation's position. And the matter gets even further complicated since Isabelle/C lives inside a proof environment; here, local theory development (rather than bold ad-hoc axiomatizations) is a major concern.

The desire for fast impact analysis resulting from changes may inspire one to annotate local proofs near directives, which is actually what is implemented in our Isabelle/C/Au-

[13] See for example the Securify project: http://securify.sce.ntu.edu.sg/

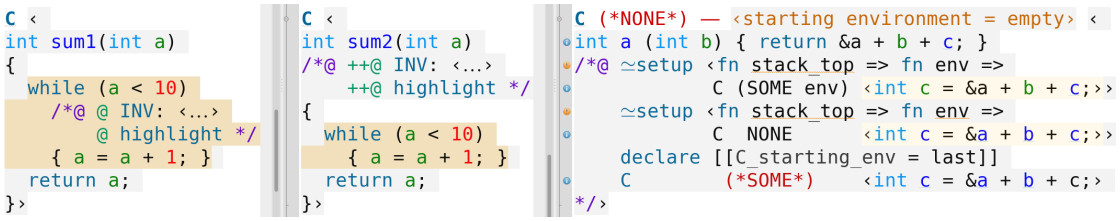


Figure 1.10: Advanced annotation programming

toCorres example (section 1.6), and shown again in Figure 1.9. In the example, the semantic back-end converts the `cpp` macro into a HOL *definition*, i.e. an extension of the underlying theory context by the conservative axiom $SQRT\text{-}UINT\text{-}MAX \equiv 65536::'a$ bound to the name *SQRT-UINT-MAX-def*. This information is used in the subsequent proof establishing a new theory context containing the lemma *uint-max-factor* configured to be used as rewrite rule whenever possible in future proofs. This local lemma establishes a relation of *SQRT-UINT-MAX* to the maximally representable number *UINT-MAX* for an unsigned integer according to the underlying memory model.

Obviously, the scheduling of these transformations of the underlying theory contexts is non-trivial.

1.5.2 Navigation for Annotation Commands

In order to overcome the problem of syntactic ambiguity of annotations, we slightly refine the syntax of semantic annotations by the concept of a navigation expression:

$\mathcal{L}_{\text{annot}} = \varnothing \mid$ `<navigation-expr>` `<annotation-command>` $\mathcal{L}_{\text{annot}}$

A `<navigation-expr>` string consists of a sequence of + symbols followed by a sequence consisting of @ or & symbols. It allows for navigating in the syntactic context, by advancing tokens with several +, or taking an ancestor AST node with several @ (or & which only targets monadic grammar rules). This corresponds to a combination of right-movements in the AST, and respectively parent-movements. This way, the "focus" of an `<annotation-command>` can be modified to denote any C fragment of interest.

As a relevant example for debugging, consider Figure 1.10. The annotation command **highlight** is a predefined Isabelle/C ML-library function that is interpreted as C annotation. Its code is implicitly parameterized by the syntactical context, represented by `stack_top` whose type is a subset of AST^{C11}, and the lexical environment `env` containing the lexical class of identifiers, scopes, positions and serials for markup. The navigation string before **highlight** particularly influences which `stack_top` value gets ultimately selected. The third screenshot in Figure 1.10 demonstrates the influence of the static environment: an Isabelle/C predefined command **≃setup** allows for "recursively" calling the C environment itself. This results in the export of definitions in the surrounding logical context, where the propagation effect may be controlled with options like *C-starting-env*. **≃setup** actually mimics standard Isabelle **setup** command, but extends it by

`stack_top` and `env` [14]. In the example, the first recursive call uses `env` allowing it to detect that b is a local parameter, while the second ignores it which results in a treatment as a free global variable. Note that bound global variables are not green but depicted in black.

1.5.3 Defining Annotation Commands

Extending the default configuration of commands, text and code antiquotations from the Isabelle platform to Isabelle/C is straightforward. For example, the central Isabelle command definition:

```
Outer_Syntax.command: K_cmd -> (σ -> σ) parser -> unit
```

establishes the dynamic binding between a command keyword K_{cmd} = **definition**|**lemma**|... and a parser, whose value is a system transition on σ. The `parser` type stems from the aforementioned parser combinator library: `'a parser = Token.T list -> 'a * Token.T list`.

Analogously, Isabelle/C provides an internal late-binding table for *annotation commands*:

```
C_Annotation.command : K_cmd -> (<navigation-expr> -> R_cmd c_parser) -> unit
C_Annotation.command': K_cmd -> (<navigation-expr> -> R_cmd c_parser) -> σ -> σ
C_Token.syntax': 'a parser -> 'a c_parser
```

where in this paper we define $R_{cmd} = \sigma$ `->` σ as above. It represents the registration of a chain of several consecutive actions.[15]

Since the type `c_parser` is isomorphic to `parser`, but accepting C tokens, one can use `C_Token.syntax'` to translate and carry the default Isar commands *inside* the **C**‹ .. › scope, such as **lemma** or **by**. Using ≃**setup**, one can even define an annotation command **C** taking a C code as argument, as the ML code of ≃**setup** has type α^{AST} `->` `env` `->` R_{cmd} (which is enough for calling `C_Annotation.command'` in the ML code). Here, whereas the type `env` is always the same, the type $\alpha^{\mathrm{AST}} \subseteq \mathrm{AST}^{\mathrm{C11}}$ varies depending on `<navigation-expr>` (see subsection 1.5.4).

Note, however, that the user experience of the IDE changes when nesting commands too deeply. In terms of error handling and failure treatment, there are some noteworthy

[14] cf. https://isabelle.in.tum.de/doc/isar-ref.pdf

[15] In some parallel work, we focus on running commands in native efficient speed with R_{cmd} = `(`K_{cmd}`*(`σ`->`σ`))list` [22]. Actually, σ has the internal Isabelle type `Toplevel.transition`, and `Toplevel.transition -> Toplevel.transition` has nothing to do with a hypothetical composition of commands, but only with a characterization of the backtracking effect for erroneous (proof) commands. This is one reason why in this definition of R_{cmd}, we used the type `list` to model a collection of elements. (Another reason is that the use of a functional type instead of a list seems to complicate the underlying scheduling implementation.) Note that for presentation purposes, we oversimplified the definition of K_{cmd}. Normally, the smooth execution of a list of commands requires to know how to optimize a given command, for mostly future-delaying pure-computational (proofs) execution (even if a command can always launch several computing threads). Such information is "encoded in its kind"; such as *thy-decl*, *thy-load*, *thy-defn*, etc., for example we should have defined K_{cmd} as: K_{cmd} = **definition** :: *thy-defn*|**lemma** :: *thy-goal-stmt*| ...

```
C ‹int _;
/*@ @ C ‹//@ C1 ‹int _; //@ @ ≃setup⇓ ‹@{C_def ⇑ C2}› \
                          @ C1  ‹//* C2 ‹int _;››   \
                          @ C1⇓ ‹//* C2 ‹int _;››    ››
    @ C ‹//* C2 ‹int _;›                              ›
      ≃setup ‹@{C_def ⇑ (* bottom-up *)  C1  }›
      ≃setup ‹@{C_def ⇓ (* top-down  *) "C1⇓"}›
*/›
```

Figure 1.11: Creating and nesting C annotations: the importance of evaluation order in annotations

implementation differences between the outermost commands and C annotation commands. Naturally, the PIDE toplevel has been optimized to maximize the error recovery and parallel execution. Inside a command, the possibilities to mimic this behaviour are somewhat limited. Whereas the failure of an outermost command must always be treated as an anomaly, in $\mathcal{L}_{\text{annot}}$, we are supposed to be editing in a comment space: an error should be reported but not too much interruptive, because a comment can contain for good reasons human narrative sentences. On the other hand, an error masking itself shyly as warning can be overlooked easily, so a form of per-case compromise has to be found to control the locally permissive set of errors. As a workaround useful during development and debugging, we offer a further pragma for a global annotation, namely $*$ (in complement to the violet @), that controls a switch between a strict and a permissive error handling for nested annotation commands:

```
<annotation-comment> = // [ * | @ ]* ℒannot
<annotation-comment> = /* [ * | @ ]* ℒannot */
```

1.5.4 Evaluation Order

We will now explain why positional languages are affecting the evaluation time of annotation commands in Figure 1.10. This requires a little zoom on how the parsing is actually executed.

The LALR parsing of our implemented C11 parser can be summarized as a sequence of alternations between Shift and Reduce actions. By definition of LALR, whereas a unique Shift action is performed for each C token read from left to right, some unlimited number of Reduce actions are happening between two Shifts. Internally, the parser manages a stack-like data-structure called α^{AST} `list` representing all already encountered Shift and Reduce actions (SR). A given α^{AST} `list` can be seen as a *forest of SR nodes*: all leafs are tagged with a Shift, and any other parent node is a Reduce node (see for example subsection 1.8.3). After a certain point in the parsing history, the top stack element α^{AST} (cast with the right type) is returned to ≃**setup**.

Since a SR-forest is a list of SR-trees, it is possible to go forward and backward at will in the actually unparsed SR-history, and execute a sequence of SR parsing steps only when needed. While every annotation command like ≃**setup** is by default attached to a closest previous Shift leaf, navigation expressions modify the attached node, making the presentation of α^{AST} referring to another term focus.

Instead of visiting the AST in the default bottom-up direction during parsing, it is possible to store the intermediate results, so that it can be revisited by using another direction strategy, for example top-down after parsing (where a parent node is executed before any of its children, and knows how they have been parsed thanks to α^{AST}). This enables commands to decide if they want to be executed during parsing, or after the full AST has been built. This gives rise to the implementation of different versions of annotation commands that are executed at different moments, relative to the parsing process. For example in Figure 1.11, the annotation command $\simeq$**setup** has been defined for being executed at bottom-up time, whereas the execution of the variant $\simeq$**setup**$\Downarrow$ happens at top-down time. In the above example, **C1** is a new command defined by `C_def`, a shorthand antiquotation for `C_Annotation.command'`. Since **C1** is meant to be executed during bottom-up time (during parsing), it is executed before **C2** is defined (which is directly after parsing).

Note that the C11 grammar has enough scoping structure for the full inference of the C environment `env` be at bottom-up time. In terms of efficiency, we use specific *static* rule wrappers having the potential of overloading default grammar rules (see Figure 1.7), to assign a wrapper to be always executed as soon as a Shift-Reduce rule node of interest is encountered. The advantage of this construction is that the wrappers are statically compiled, which results in a very efficient reporting of C type information.

1.6 Semantic Back-Ends

In this section, we briefly present two integrations of verification back-ends for C. We chose Clean [24] [16] used for program-based test generation [14], and AutoCorres [10], arguably the most developed deductive verification environment for machine-oriented C available at present.

Note that we were focusing on keeping modifications of integrated components minimal, particularly for the case of AutoCorres. Certain functionalities like position propagation of HOL terms in annotations, or "automatic" incremental declarations [17] may require internal revisions on the back-end side. This is out of the scope of this paper.

1.6.1 A Simple Typed Memory Model: Clean

Clean (pronounced as: "C lean" or "Céline" [selin]) is based on a simple, shallow-style execution model for an imperative target language. Importing Clean into a theory with its activated back-end proceeds as in Figure 1.12. For a deep presentation of Clean, the reader is referred to the published AFP document [24].

Finally, we will have a glance at the code for the registration of the annotation commands presented in Figure 1.13, and used in the example of Figure 1.12. Thanks to

[16] Stemming from recent HOL-TestGen distributions: https://www.brucker.ch/projects/hol-testgen/index.en.html

[17] https://github.com/seL4/l4v/blob/master/tools/autocorres/tests/examples/Incremental.thy

```
theory Prime imports Isabelle_C_Clean.Backend
                — ‹Clean back-end is now on›
begin
C ‹
//@ definition ‹prime_HOL (p :: nat) =                    \
          (1 < p ∧ (∀ n ∈ {2..<p}. ¬ n dvd p))›
#   define SQRT_UINT_MAX 65536
unsigned int k = 0;
unsigned int prime_C(unsigned int n) {
//@ pre_Clean  ‹C‹n› ≤ UINT_MAX›
//@ post_Clean ‹C‹prime_C(n)› ≠ 0 ⟷ prime_HOL C‹n››
    if (n < 2) return 0;
    for (unsigned i = 2; i < SQRT_UINT_MAX
                         && i * i <= n; i++) {
      if (n % i == 0) return 0;
      k++;
    }
    return 1;
}›
```

```
prime_C_core_def: "prime_C_core n ≡
  if_Clean ‹(n < 2) ›
then return 0 else skip;-
  ‹ i := 2 ›;-
  while_Clean ‹
i < SQRT_UINT_MAX ∧ i * i ≤ n›
    (if_Clean ‹n mod i = 0›
      then return 0 else skip;
     ‹k:=k+1›; assert ‹k≤UINT_MAX ›
     ‹i:=i+1›; assert ‹i≤UINT_MAX ›) ;-
  return 1"

prime_C_def: "prime_C n ≡
  block_Clean push_local_prime_C_state
              (is_prime_core n)
              pop_local_prime_C_state"
```

Figure 1.12: Activating the Isabelle/C/Clean back-end triggers the generation of theorems

```
ML ‹fun command keyword f =
  C_Annotation.command' keyword ""
    (C_Token.syntax'
      (Parse.token Parse.cartouche)
     >>> toplevel f)›
setup ‹command ("pre_Clean", ⌂) Spec
    #> command ("post_Clean", ⌂) End_spec
    #> command ("inv_Clean", ⌂) Invariant›
```

Figure 1.13: Registering new C annotation commands

```
theory Prime
  imports Isabelle_C_AutoCorres.Backend
begin
C ‹
  ⋮
theorem (in primeC) primeC'_correct:
  ‹{ λ _. n ≤ UINT_MAX } primeC' n
   { λ primeC' _. primeC' ≠ 0 ⟷ primeHOL n }!›
proof (rule validNF_assume_pre)
 assume 1: ‹n ≤ UINT_MAX›
 have   2: ‹n = 0 ∨ n = 1 ∨ n > 1› by linarith
 show ?thesis
  proof (insert 2, elim disjE)
   assume ‹n = 0›
   then show ?thesis
        by (clarsimp simp: primeC'_def, wp, auto)
  next
  ⋮
```

Figure 1.14: Isabelle/C/AutoCorres: integrating an HOL correctness proof as annotation *inside* C code

Isabelle/C's function `C_Annotation.command'`, the registration of user-defined annotations is very similar to the registration of ordinary commands in the Isabelle platform.

1.6.2 The Case of AutoCorres

The AutoCorres environment consists of a C99 parser, compiling to a deepish embedding of a generic imperative core programming language, over a refined machine word oriented memory model, and a translator of this presentation into a shallow language based on another Monad for non-deterministic computations. This translator has been described in [10, 27] in detail. However, the original use of AutoCorres implies a number of protocol rules to follow (e.g., calling the command **install-C-file** before the command **autocorres**), and is only loosely integrated into the Isabelle document model, which complicates the workflow substantially.

Our running example $prime_C$ for Isabelle/C/AutoCorres in Figure 1.14 basically differs in what the theory is importing in its header. Similarly to Clean, AutoCorres constructs a memory model and represents the program as a monadic operation on it. Actually, it generates even two presentations, one on a very precise word-level memory model taking aspects of the underlying processor architecture into account, and another one more abstract, then it automatically proves the correspondence in our concrete example. Both representations become the definitions *primeC-def* and *primeC'-def*. A Hoare-calculus plus a derived verification generator *wp* from the AutoCorres package leverage finally the correctness proof.

Note that the integration of AutoCorres crucially depends on the conversion $\text{AST}^{\text{C11}} \Rightarrow \text{AST}^{\text{C99}}$ of $C^{\downarrow}$ discussed in subsection 1.4.3. In particular, for the overall seL4 annotations **INVARIANT**, **INV**, **FNSPEC**, **RELSPEC**, **MODIFIES**, **DONT-TRANSLATE**, **AUXUPD**, **GHOSTUPD**, **SPEC**, **END–SPEC**, **CALLS**, and **OWNED-BY**, we have extended our implementation of $C^{\downarrow}$ in such a

way that the conversion places the information at the right position in the target AST. Obviously, this works even when navigation is used, as in Figure 1.10 left.

1.6.3 Yet Another C Back-End: Securify

In the Securify project at the Nanyang Technological University [11, 21], an extension of L4.verified is in active development which attempts to integrate concurrency and some form of non-coherent memory into the verification framework. This ongoing project is already profiting from the overall Isabelle/C framework, including the integrated parser $\text{PARSE}^{\text{C11}}_{report}$. Note, however, that the XtratuM micro-kernel [6] in study is using certain advance features of C11 not found in seL4 (including nested functions, or particular arrangement of assembly code). The project is actively working on developing dedicated semantic back-ends for respective features, to be separately or concurrently loaded with the AutoCorres one presented in subsection 1.6.2.

1.7 Conclusions

We presented Isabelle/C a novel, generic front-end for a deep integration of C11 code into the Isabelle/PIDE framework. Based on open-source Lex and Yacc style grammars, we presented a build process that constructs key components for this front-end: the lexer, the parser, and a framework for user-defined annotations including user-defined annotation commands. While the generation process is relatively long, the generated complete library can be loaded in a few seconds constructing an environment similar to the usual **ML** environment for Isabelle itself. 20 kLoC large C sources can be parsed and decorated in PIDE within seconds.

Our framework allows for the deep integration of the C source into a global document model in which literate programming style documentation, modelling as well as static program analysis and verification co-exist. In particular, information from the different tools realized as plugin in the Isabelle platform can flow freely, but based on a clean management of their semantic context and within a framework based on conservative theory development. This substantially increases the development agility of such type of sources and may be attractive to conventional developers, in particular when targeting formal certification [5].

Isabelle/C also forms a basis for future semantically well-understood combinations of back-ends based on different semantic interpretations: inside Isabelle, bridge lemmas can be derived that describe the precise conditions under which results from one back-end can be re-interpreted and used in another. Future tactic processes based on these bridge lemmas may open up novel ways for semantically safe tool combinations.

Acknowledgments. The authors warmly thank David Sanán and Yang Liu for encouraging the development and reuse of $C^{\downarrow}$, initially started in the Securify project [21] (http://securify.sce.ntu.edu.sg/).

1.8 Appendix

1.8.1 Portability of Isabelle/C's Generated Files

To evaluate the portability of the generated files in Figure 1.7, we recall that:

- the generation of AST^{C11} depends on Haskabelle, Citadelle and the Isabelle code generator,
- and the generated G_{ML} depends on the two respective ASTs of Happy and ML-Yacc.

As we assume the Isabelle code generator maintained by the Isabelle developers, and a substantial part of Citadelle already maintained in the Isabelle AFP, the discussion remains on the other tools used for generating AST^{C11} and G_{ML}. For the case of Happy and ML-Yacc, their own grammar languages have not significantly evolved in the last years, and are still maintained in https://github.com/simonmar/happy and https://github.com/MLton/mlton. For the case of Haskabelle, it has been dropped from the Isabelle distribution, because it is critically relying on haskell-src-exts http://hackage.haskell.org/package/haskell-src-exts, and the AST of haskell-src-exts received a drastic replacement from version 1.17.1 to 1.18.0.[18] Consequently, upgrading Haskabelle to use the new 1.18.0 represents an entire challenge. Hopefully, it is a task we already completed while integrating Haskabelle's output with Citadelle's input. Otherwise, we have not encountered any problems in upgrading Haskabelle from 1.18.0 to more recent versions (such as 1.20.1 which was straightforward).

1.8.2 A Zoom on Citadelle's HOL Generator

In the generation process of the parsers for Isabelle/C, the Citadelle system [22] plays a vital role. In particular, it is used to convert Haskell AST presentations into HOL AST presentations through an initial call to Haskabelle. However, the resulting output can be run natively in order to increase efficiency via Citadelle.

In more detail, combining Haskabelle and Citadelle is almost direct, as Haskabelle's output AST and Citadelle's input AST are already both subset models of a rich Isar/HOL interface, where the latter model is still up-to-date and maintained [22]. Note that one can configure Citadelle to generate an AST_{HOL} built with **old-datatype** commands instead of **datatype**. This results in a gain of 10 min on the shallow-reflected evaluation with the option *quick−and−dirty* activated.

In the following, we show how we proceeded: we implemented a command **Haskell-file**

[18] See http://hackage.haskell.org/package/haskell-src-exts-1.17.1 and http://hackage.haskell.org/package/haskell-src-exts-1.18.0. Actually, the default AST of haskell-src-exts was duplicated and enhanced in 2009 between 1.1.1 to 1.1.3 for a parsed source to carry additional position information in all its parsing nodes. Ultimately, the two ASTs was merged together while releasing version 1.18.0.

that accepts various options: [19]

Haskell-file **datatype-old try-import only-types concat-modules**
base-path $*HASKABELLE-HOME*/*ex*/*language*−*c*/*src*
[*Prelude* ⇀ *C-Model-init*, *Int*, *String*, *Option* ⇀ *C-Model-init*]

$*HASKABELLE-HOME*/*ex*/*language*−*c*/*src*/*Language*/*C*/*Syntax*/*AST.hs*

This construction has the advantage to allow immediate code value transformations obtained from Haskabelle: operationally, **Haskell-file** makes an implicit call to Haskabelle, translates the output of Haskabelle to HOL input, and lets the Citadelle framework execute the generated theorems and proofs in parallel. For the latter, we use an Isabelle version that has been especially modified for this purpose.[20]

On the one hand, the execution of the AST_{HOL} mostly consisting of **datatype** commands is not improving when being generated and executed by our Citadelle infrastructure, because the blocking point is located in just *one* problematic mutually recursive **datatype** command (which is not interacting with other commands). On the other hand, the question of advanced parallelisation techniques in the **datatype** package is actually opening an interesting topic of future experiments — i.e., whenever mutually recursive constructions could be expressed with *several* commands. The Citadelle infrastructure offers a number of mechanisms to address this task.

1.8.3 Shift-Reduce Forest of a Parsing Execution

By activating a specific option in Isabelle/C, one can dump the complete Shift-Reduce forest of a parsing execution. For example, the SR-forest associated to the left subwindow of Figure 1.10 is provided below:

```
REDUCE 0 X ⍓ ⍓ start_happy1 ( ( CTranslUnit, (CExtDecl, (CStat, (CExpr, unit) either) either) either ) either )
either)
 SHIFT ⍓ ⍓
 REDUCE 4 0 ⍓ ⍓ translation_unit (CTranslUnit)
  REDUCE 7 X ⍓ ⍓ ext_decl_list3 ( ( CExtDecl list ) Reversed)
   REDUCE 5 X ⍓ ⍓ ext_decl_list1 ( ( CExtDecl list ) Reversed)
   REDUCE 8 X ⍓ ⍓ external_declaration1 (CExtDecl)
    REDUCE 15 0 ⍓ ⍓ function_definition4 (CFunDef)
     REDUCE 127 X ⍓ ⍓ type_specifier1 (CDeclSpec list)
      REDUCE 147 X ⍓ ⍓ basic_type_specifier1 ( ( CDeclSpec list ) Reversed)
       REDUCE 133 0 ⍓ ⍓ basic_type_name4 (CTypeSpec)
        SHIFT ⍓ ⍓
     REDUCE 26 X ⍓ ⍓ function_declarator (CDeclr)
      REDUCE 260 X ⍓ ⍓ identifier_declarator1 (CDeclrR)
       REDUCE 262 X ⍓ ⍓ unary_identifier_declarator1 (CDeclrR)
        REDUCE 267 X ⍓ ⍓ postfix_identifier_declarator1 (CDeclrR)
```

[19] Note that **old-datatype** is a command, whereas **datatype-old** is an option provided to the command **Haskell-file**.

[20] The use of a modified Isabelle is just an optimization: the whole Citadelle framework can also be configured to run on a standard Isabelle version, albeit with a significantly higher execution time.

```
      REDUCE 272 0 ⌂ ⌂ paren_identifier_declarator1 (CDeclrR)
       SHIFT ⌂ ⌂
      REDUCE 312 0 ⌂ ⌂ postfixing_abstract_declarator2 ( ( CDeclrR -> CDeclrR ) )
       SHIFT ⌂ ⌂
       REDUCE 283 X ⌂ ⌂ parameter_type_list2 ( ( CDecl list * Bool ) )
        REDUCE 285 X ⌂ ⌂ parameter_list1 ( ( CDecl list ) Reversed)
         REDUCE 296 0 ⌂ ⌂ parameter_declaration10 (CDecl)
          REDUCE 127 X ⌂ ⌂ type_specifier1 (CDeclSpec list)
           REDUCE 147 X ⌂ ⌂ basic_type_specifier1 ( ( CDeclSpec list ) Reversed)
            REDUCE 133 0 ⌂ ⌂ basic_type_name4 (CTypeSpec)
             SHIFT ⌂ ⌂
          REDUCE 261 X ⌂ ⌂ identifier_declarator2 (CDeclrR)
           REDUCE 272 0 ⌂ ⌂ paren_identifier_declarator1 (CDeclrR)
            SHIFT ⌂ ⌂
          REDUCE 471 X ⌂ ⌂ attrs_opt1 (CAttr list)
       SHIFT ⌂ ⌂
    REDUCE 38 0 ⌂ ⌂ compound_statement1 (CStat)
     SHIFT ⌂ ⌂
     REDUCE 40 X ⌂ ⌂ enter_scope (unit)
     REDUCE 43 X ⌂ ⌂ block_item_list2 ( ( CBlockItem list ) Reversed)
      REDUCE 43 X ⌂ ⌂ block_item_list2 ( ( CBlockItem list ) Reversed)
       REDUCE 42 X ⌂ ⌂ block_item_list1 ( ( CBlockItem list ) Reversed)
       REDUCE 44 X ⌂ ⌂ block_item1 (CBlockItem)
        REDUCE 31 X ⌂ ⌂ statement5 (CStat)
         REDUCE 61 0 ⌂ ⌂ iteration_statement1 (CStat)
          SHIFT ⌂ ⌂
          SHIFT ⌂ ⌂
          REDUCE 452 X ⌂ ⌂ expression1 (CExpr)
           REDUCE 439 X ⌂ ⌂ assignment_expression1 (CExpr)
            REDUCE 436 X ⌂ ⌂ conditional_expression1 (CExpr)
             REDUCE 434 X ⌂ ⌂ logical_or_expression1 (CExpr)
              REDUCE 432 X ⌂ ⌂ logical_and_expression1 (CExpr)
               REDUCE 430 X ⌂ ⌂ inclusive_or_expression1 (CExpr)
                REDUCE 428 X ⌂ ⌂ exclusive_or_expression1 (CExpr)
                 REDUCE 426 X ⌂ ⌂ and_expression1 (CExpr)
                  REDUCE 423 X ⌂ ⌂ equality_expression1 (CExpr)
                   REDUCE 419 0 ⌂ ⌂ relational_expression2 (CExpr)
                    REDUCE 418 X ⌂ ⌂ relational_expression1 (CExpr)
                     REDUCE 415 X ⌂ ⌂ shift_expression1 (CExpr)
                      REDUCE 412 X ⌂ ⌂ additive_expression1 (CExpr)
                       REDUCE 408 X ⌂ ⌂ multiplicative_expression1 (CExpr)
                        REDUCE 406 X ⌂ ⌂ cast_expression1 (CExpr)
                         REDUCE 388 X ⌂ ⌂ unary_expression1 (CExpr)
                          REDUCE 376 X ⌂ ⌂ postfix_expression1 (CExpr)
                           REDUCE 360 0 ⌂ ⌂ primary_expression1 (CExpr)
                            SHIFT ⌂ ⌂
                    SHIFT ⌂ ⌂
                    REDUCE 415 X ⌂ ⌂ shift_expression1 (CExpr)
                     REDUCE 412 X ⌂ ⌂ additive_expression1 (CExpr)
                      REDUCE 408 X ⌂ ⌂ multiplicative_expression1 (CExpr)
                       REDUCE 406 X ⌂ ⌂ cast_expression1 (CExpr)
                        REDUCE 388 X ⌂ ⌂ unary_expression1 (CExpr)
                         REDUCE 376 X ⌂ ⌂ postfix_expression1 (CExpr)
                          REDUCE 361 X ⌂ ⌂ primary_expression2 (CExpr)
```

```
                    REDUCE 461 0 ⌂ ⌂ constant1 (CConst)
                     SHIFT ⌂ ⌂
      SHIFT ⌂ ⌂
      REDUCE 28 X ⌂ ⌂ statement2 (CStat)
       REDUCE 38 0 ⌂ ⌂ compound_statement1 (CStat)
        SHIFT ⌂ ⌂
        REDUCE 40 X ⌂ ⌂ enter_scope (unit)
        REDUCE 43 X ⌂ ⌂ block_item_list2 ( ( CBlockItem list ) Reversed)
         REDUCE 42 X ⌂ ⌂ block_item_list1 ( ( CBlockItem list ) Reversed)
         REDUCE 44 X ⌂ ⌂ block_item1 (CBlockItem)
          REDUCE 29 X ⌂ ⌂ statement3 (CStat)
           REDUCE 57 0 ⌂ ⌂ expression_statement2 (CStat)
            REDUCE 452 X ⌂ ⌂ expression1 (CExpr)
             REDUCE 440 0 ⌂ ⌂ assignment_expression2 (CExpr)
              REDUCE 388 X ⌂ ⌂ unary_expression1 (CExpr)
               REDUCE 376 X ⌂ ⌂ postfix_expression1 (CExpr)
                REDUCE 360 0 ⌂ ⌂ primary_expression1 (CExpr)
                 SHIFT ⌂ ⌂
              REDUCE 441 0 ⌂ ⌂ assignment_operator1 (CAssignOp Located)
               SHIFT ⌂ ⌂
              REDUCE 439 X ⌂ ⌂ assignment_expression1 (CExpr)
               REDUCE 436 X ⌂ ⌂ conditional_expression1 (CExpr)
                REDUCE 434 X ⌂ ⌂ logical_or_expression1 (CExpr)
                 REDUCE 432 X ⌂ ⌂ logical_and_expression1 (CExpr)
                  REDUCE 430 X ⌂ ⌂ inclusive_or_expression1 (CExpr)
                   REDUCE 428 X ⌂ ⌂ exclusive_or_expression1 (CExpr)
                    REDUCE 426 X ⌂ ⌂ and_expression1 (CExpr)
                     REDUCE 423 X ⌂ ⌂ equality_expression1 (CExpr)
                      REDUCE 418 X ⌂ ⌂ relational_expression1 (CExpr)
                       REDUCE 415 X ⌂ ⌂ shift_expression1 (CExpr)
                        REDUCE 413 0 ⌂ ⌂ additive_expression2 (CExpr)
                         REDUCE 412 X ⌂ ⌂ additive_expression1 (CExpr)
                          REDUCE 408 X ⌂ ⌂ multiplicative_expression1 (CExpr)
                           REDUCE 406 X ⌂ ⌂ cast_expression1 (CExpr)
                            REDUCE 388 X ⌂ ⌂ unary_expression1 (CExpr)
                             REDUCE 376 X ⌂ ⌂ postfix_expression1 (CExpr)
                              REDUCE 360 0 ⌂ ⌂ primary_expression1 (CExpr)
                               SHIFT ⌂ ⌂
                        SHIFT ⌂ ⌂
                        REDUCE 408 X ⌂ ⌂ multiplicative_expression1 (CExpr)
                         REDUCE 406 X ⌂ ⌂ cast_expression1 (CExpr)
                          REDUCE 388 X ⌂ ⌂ unary_expression1 (CExpr)
                           REDUCE 376 X ⌂ ⌂ postfix_expression1 (CExpr)
                            REDUCE 361 X ⌂ ⌂ primary_expression2 (CExpr)
                             REDUCE 461 0 ⌂ ⌂ constant1 (CConst)
                              SHIFT ⌂ ⌂
          SHIFT ⌂ ⌂
      REDUCE 41 X ⌂ ⌂ leave_scope (unit)
      SHIFT ⌂ ⌂
REDUCE 44 X ⌂ ⌂ block_item1 (CBlockItem)
 REDUCE 32 X ⌂ ⌂ statement6 (CStat)
  REDUCE 69 0 ⌂ ⌂ jump_statement5 (CStat)
   SHIFT ⌂ ⌂
   REDUCE 457 X ⌂ ⌂ expression_opt2 (CExpr Maybe)
```

```
          REDUCE 452 X ⌂ ⌂ expression1 (CExpr)
           REDUCE 439 X ⌂ ⌂ assignment_expression1 (CExpr)
            REDUCE 436 X ⌂ ⌂ conditional_expression1 (CExpr)
             REDUCE 434 X ⌂ ⌂ logical_or_expression1 (CExpr)
              REDUCE 432 X ⌂ ⌂ logical_and_expression1 (CExpr)
               REDUCE 430 X ⌂ ⌂ inclusive_or_expression1 (CExpr)
                REDUCE 428 X ⌂ ⌂ exclusive_or_expression1 (CExpr)
                 REDUCE 426 X ⌂ ⌂ and_expression1 (CExpr)
                  REDUCE 423 X ⌂ ⌂ equality_expression1 (CExpr)
                   REDUCE 418 X ⌂ ⌂ relational_expression1 (CExpr)
                    REDUCE 415 X ⌂ ⌂ shift_expression1 (CExpr)
                     REDUCE 412 X ⌂ ⌂ additive_expression1 (CExpr)
                      REDUCE 408 X ⌂ ⌂ multiplicative_expression1 (CExpr)
                       REDUCE 406 X ⌂ ⌂ cast_expression1 (CExpr)
                        REDUCE 388 X ⌂ ⌂ unary_expression1 (CExpr)
                         REDUCE 376 X ⌂ ⌂ postfix_expression1 (CExpr)
                          REDUCE 360 0 ⌂ ⌂ primary_expression1 (CExpr)
                           SHIFT ⌂ ⌂
        SHIFT ⌂ ⌂
     REDUCE 41 X ⌂ ⌂ leave_scope (unit)
     SHIFT ⌂ ⌂
VOID
```

2 A Resume on Isabelle/C: Commands, Control Attributes and Programming Infrastructure

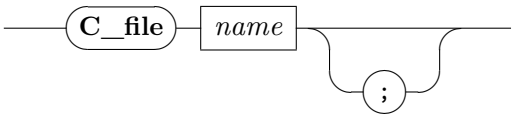
theory *C-Appendices*
imports *../examples/C2*
Isar-Ref.Base
begin

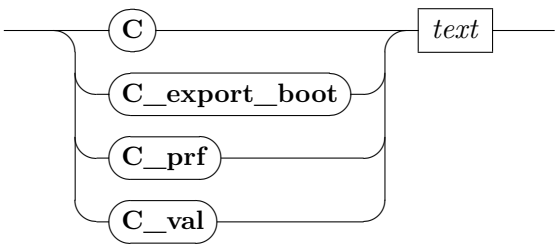
2.1 Syntax Commands for Isabelle/C

2.1.1 Toplevel Commands and Control Attributes

C-file	:	*local-theory* → *local-theory*
C	:	*local-theory* → *local-theory*
C-export-boot	:	*local-theory* → *local-theory*
C-prf	:	*proof* → *proof*
C-val	:	*any* →
C-export-file	:	*any* →

C-lexer-trace	:	*attribute*	default *false*
C-parser-trace	:	*attribute*	default *false*
C-ML-verbose	:	*attribute*	default *true*
C_{env0}	:	*attribute*	default *empty*
C_{rule0}	:	*attribute*	default *translation-unit*





C_export_file

C-file *name* reads the given C file, and let any attached semantic back-ends to proceed for further subsequent evaluation. Top-level C bindings are stored within the (global or local) theory context; the initial environment is set by default to be an empty one, or the one returned by a previous **C-file** (depending on attribute C_{env0}). The entry-point of the grammar taken as initial starting point of the parser is read from attribute C_{rule0} (see https://www.haskell.org/happy/doc/html/sec-directives.html#sec-parser-name). Multiple **C-file** commands may be used to build larger C projects if they are all written in a single theory file (existing parent theories are ignored, and not affecting the current working theory).

C is similar to **C-file**, but evaluates directly the given *text*. Top-level resulting bindings are stored within the (global or local) theory context.

C-export-boot is similar to **ML-export**, except that the code in input is understood as being processed by **C** instead of **ML**.

C-prf is similar to **ML-prf**, except that the code in input is understood as being processed by **C** instead of **ML**.

C-val is similar to **ML-val**, except that the code in input is understood as being processed by **C** instead of **ML**.

C-export-file is similar to **generate-file** *fic* = ‹*code*› **export-generated-files** *fic*, except that

- *code* refers to the dump of all existing previous C code in the current theory (parent theories are ignored),
- and any ML antiquotations in *code* are not analyzed by **generate-file** (in contrast with its default behavior).

C-lexer-trace indicates whether the list of C tokens associated to the source text should be output (that list is computed during the lexing phase).

C-parser-trace indicates whether the stack forest of Shift-Reduce node should be output (it is the final stack which is printed, i.e., the one taken as soon as the parsing terminates).

C-ML-verbose indicates whether nested **ML** commands are acting similarly as their default verbose configuration in top-level.

C_{env0} makes the start of a C command (e.g., **C-file**, **C**) initialized with the environment of the previous C command if existing.

C_{rule0} sets the root syntactic category in which the parser starts. C commands (e.g., **C-file**, **C**). Possible values are:

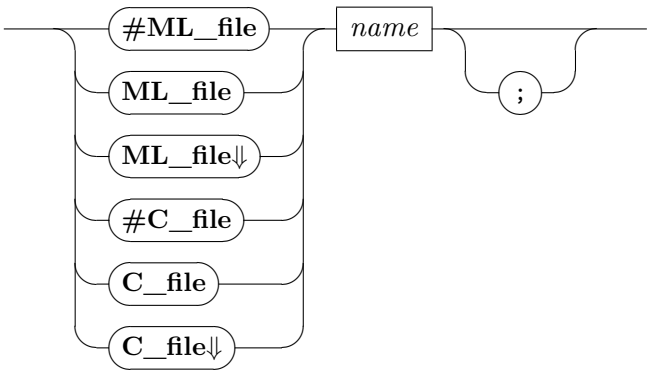
expression,

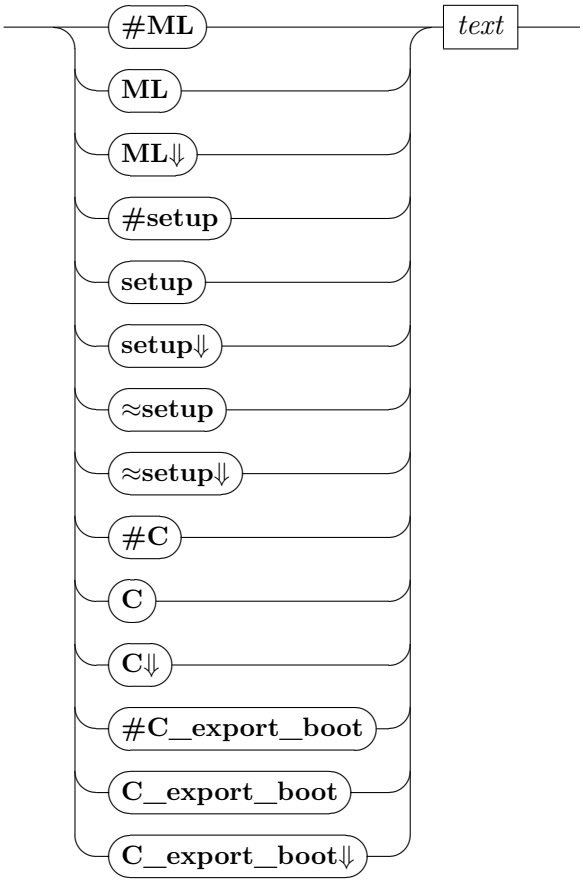
statement,

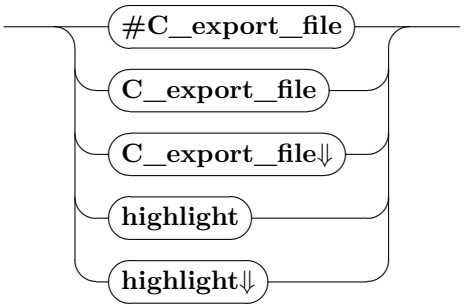
external-declaration and

translation-unit (the default)

2.1.2 Predefined Annotation Commands inside the C context (C Annotation commands)







ML-file, **C-file**, **ML**, **setup**, **C**, **C-export-boot**, and **C-export-file** behave similarly as the respective outer commands **ML-file**, **C-file**, **ML**, **setup**, **C**, **C-export-boot**, **C-export-file**.

≈setup ‹f'› has the same semantics as **setup** ‹f› whenever $\bigwedge$*stack top env*. f' *stack top env* = f. In particular, depending on where the annotation **≈setup** ‹f'› is located in the C code, the additional values *stack*, *top* and *env* can drastically vary, and then can be possibly used in the body of f' for implementing new interactive features

(e.g., in contrast to *f*, which by default does not have the possibility to directly use the information provided by *stack*, *top* and *env*).

highlight changes the background color of the C tokens pointed by the command.

#**ML-file**, #**C-file**, #**ML**, #**setup**, #**C**, #**C-export-boot**, and #**C-export-file** behave similarly as the respective (above inner) commands **ML-file**, **C-file**, **ML**, **setup**, **C**, **C-export-boot**, and **C-export-file** except that their evaluations happen as earliest as possible.

ML-file⇓, **C-file**⇓, **ML**⇓, **setup**⇓, ≈**setup**⇓, **C**⇓, **C-export-boot**⇓, **C-export-file**⇓, and **highlight**⇓ behave similarly as the respective (above inner) commands **ML-file**, **C-file**, **ML**, **setup**, ≈**setup**, **C**, **C-export-boot**, **C-export-file**, and **highlight** except that their evaluations happen as latest as possible.

2.1.3 Inner Directive Commands

Among the directives provided as predefined in Isabelle/C, we currently have: `#define _` and `#undef _`. In particular, for the case of `#define _`, rewrites are restricted to variable-form macros: support of functional macros is not yet provided.

In Isabelle/C, not-yet-defined directives (such as `#include _` or `#if #endif`, etc.) do not make the parsing fail, but are treated as "free variable commands".

2.2 Quick Start (for People More Familiar with C than Isabelle)

- Assuming we are working with Isabelle 2021 https://isabelle.in.tum.de/website-Isabelle2021/dist/Isabelle2021_linux.tar.gz, the shortest way to start programming in C is to open a new theory file with the shell-command:

  ```
  $ISABELLE_HOME/bin/isabelle jedit -d $AFP_HOME/thys Scratch.thy
  ```

  where `$ISABELLE_HOME` is the path of the above extracted Isabelle source, and `$AFP_HOME` is the downloaded content of https://foss.heptapod.net/isa-afp/afp-2021.[1]

- The next step is to copy this minimal content inside the newly opened window:

  ```
  theory Scratch imports Isabelle_C.C_Main begin C \<open>
  // C code
  \<close> end
  ```

[1] This folder particularly contains the Isabelle/C project, located in https://foss.heptapod.net/isa-afp/afp-2021/-/tree/branch/default/thys/Isabelle_C. To inspect the latest developper version, one can also replace `$AFP_HOME/thys` by the content downloaded from https://gitlab.lisn.upsaclay.fr/burkhart.wolff/Isabelle_C.

- *Quod Erat Demonstrandum!* This already enables the support of C code inside the special brackets "`\<open>\<close>`", now depicted as "‹›" for readability reasons.

Additionally, Isabelle/C comes with several functionalities that can be alternatively explored:

- To write theorems and proofs along with C code, the special C comment `/*@ (* Isabelle content *) */` can be used at any position where C comments are usually regularly allowed. At the time of writing, not yet all Isabelle commands can be written in C comments, and certain proof-solving-command combinations are also not yet implemented — manual registration of commands to retrieve some more or less native user-experience remains possible though. Generally, the kind of content one can write in C comments should be arbitrary. The exhaustive list of Isabelle commands is provided in the accompanying above archive, for example in `$ISABELLE_HOME/src/Doc/Isar_Ref` or https://isabelle.in.tum.de/doc/isar-ref.pdf.

- Instead of starting from scratch, any existing C files can also be opened with Isabelle/C, it suffices to replace:

  `C` ‹ /* *C* */ ›
  by
  `C_file` ‹~/*file.c*›

  Once done, one can press a CTRL-like key while hovering the mouse over the file name, then followed by a click on it to open a new window loading that file.

- After a `C` ‹ /* *C* */ › command, one has either the possibility to keep the content as such in the theory file, or use `C_export_file` to export all previous C content into a "real" C file.

  Note that since Isabelle2019, Isabelle/C uses a virtual file-system. This has the consequence, that some extra operations are needed to export a file generated into the virtual file-system of Isabelle into the "real" file-system. First, the `C_export_file` command needs to be activated, by putting the cursor on the command. This leads to the following message in the output window: `See theory exports "C/*/*.c"` (see Figure 2.1). By clicking on *theory exports* in this message, Isabelle opens a *File Browser* showing the content of the virtual file-system in the left window. Selecting and opening a generated file in the latter lets jEdit display it in a new buffer, which gives the possibility to export this file via "*File* → *Save As...*" into the real file-system.

2.3 Case Study: Mapping on the Parsed AST via C99 ASTs

In this section, we give a concrete example of a situation where one is interested to do some automated transformations on the parsed AST, such as changing the type of

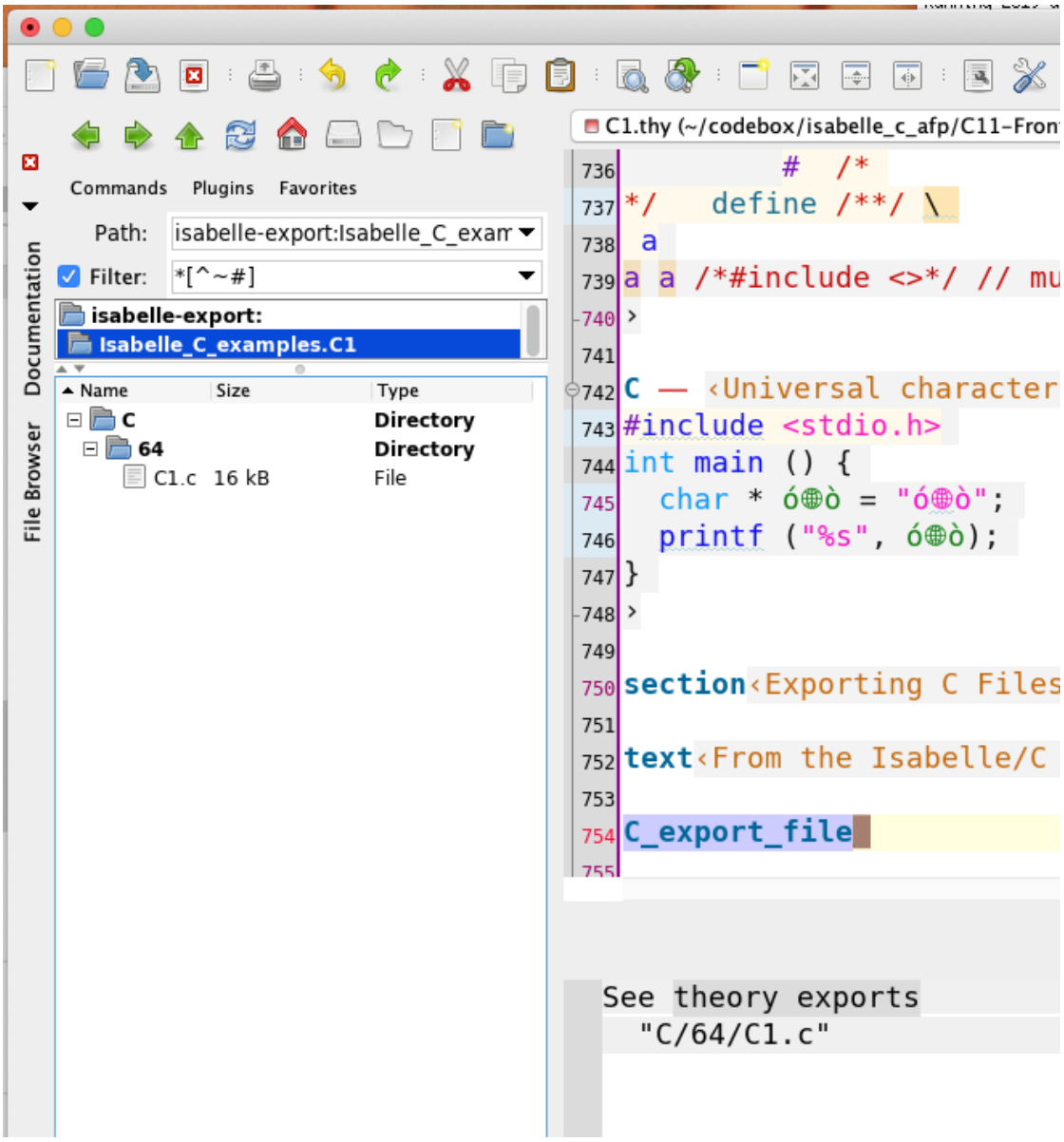


Figure 2.1: Making the File Browser Pointing to the Virtual File System

every encountered variables from `int _;` to `int _ [];`. The main theory of interest here is *Isabelle-C.C-Parser-Language*, where the C grammar is loaded, in contrast to *Isabelle-C.C-Lexer-Language* which is only dedicated to build a list of C tokens. As another example, *Isabelle-C.C-Parser-Language* also contains the portion of the code implementing the report to the user of various characteristics of encountered variables during parsing: if a variable is bound or free, or if the declaration of a variable is made in the global topmost space or locally declared in a function.

2.3.1 Prerequisites

Even if `../generated/c_grammar_fun.grm.sig` and `../generated/c_grammar_fun.grm.sml` are files written in ML syntax, we have actually modified `../../src_ext/mlton/lib/mlyacc-lib` in such a way that at run time, the overall loading and execution of *Isabelle-C.C-Parser-Language* will mimic all necessary features of the Haskell parser generator Happy [2], including any monadic interactions between the lexing (*Isabelle-C.C-Lexer-Language*) and parsing part (*Isabelle-C.C-Parser-Language*).

This is why in the remaining part, we will at least assume a mandatory familiarity with Happy (e.g., the reading of ML-Yacc's manual can happen later if wished [3]). In particular, we will use the term *rule code* to designate *a Haskell expression enclosed in braces* [4].

2.3.2 Structure of *Isabelle-C.C-Parser-Language*

In more detail, *Isabelle-C.C-Parser-Language* can be seen as being principally divided into two parts:

- a first part containing the implementation of `C_Grammar_Rule_Lib`, which provides the ML implementation library used by any rule code written in the C grammar https://github.com/visq/language-c/blob/master/src/Language/C/Parser/Parser.y (`../generated/c_grammar_fun.grm.sml`).

- a second part implementing `C_Grammar_Rule_Wrap`, providing one wrapping function for each rule code, for potentially complementing the rule code with an additional action to be executed after its call. The use of wrapping functions is very optional: by default, they are all assigned as identity functions.

The difference between `C_Grammar_Rule_Lib` and `C_Grammar_Rule_Wrap` relies in how often functions in the two structures are called: while building subtree pieces of the final AST, grammar rules are free to call any functions in `C_Grammar_Rule_Lib` for completing their respective tasks, but also free to not use `C_Grammar_Rule_Lib` at all. On the other hand, irrespective of the actions done by a rule code, the function associated to the rule

[2] https://www.haskell.org/happy/doc/html/index.html
[3] https://www.cs.princeton.edu/~appel/modern/ml/ml-yacc/manual.html
[4] https://www.haskell.org/happy/doc/html/sec-grammar.html

code in `C_Grammar_Rule_Wrap` is retrieved and always executed (but a visible side-effect will likely mostly happen whenever one has provided an implementation far different from `I`).

Because the grammar https://github.com/visq/language-c/blob/master/src/Language/C/Parser/Parser.y (`../generated/c_grammar_fun.grm.sml`) has been defined in such a way that computation of variable scopes are completely handled by functions in `C_Grammar_Rule_Lib` and not in rule code (which are just calling functions in `C_Grammar_Rule_Lib`), it is enough to overload functions in `C_Grammar_Rule_Lib` whenever it is wished to perform new actions depending on variable scopes, for example to do a specific PIDE report at the first time when a C variable is being declared. In particular, functions in `C_Grammar_Rule_Lib` are implemented in monadic style, making a subsequent modification on the parsing environment *Isabelle-C.C-Environment* possible (whenever appropriate) as this last is carried in the monadic state.

Fundamentally, this is feasible because the monadic environment fulfills the property of being always properly enriched with declared variable information at any time, because we assume

- working with a language where a used variable must be at most declared or redeclared somewhere before its actual usage,
- and using a parser scanning tokens uniquely, from left to right, in the same order as the execution of rule code actions.

Example

As illustration, `C_Grammar_Rule_Lib.markup_var o C_Ast.Left` is (implicitly) called by a rule code while a variable being declared is encountered. Later, a call to `C_Grammar_Rule_Lib.markup_var o C_Ast.Right` in `C_Grammar_Rule_Wrap` (actually, in `C_Grammar_Rule_Wrap_Overloading`) is made after the execution of another rule code to signal the position of a variable in use, together with the information retrieved from the environment of the position of where it is declared.

In more detail, the second argument of `C_Grammar_Rule_Lib.markup_var` is among other of the form: `Position.T * {global: bool}`, where particularly the field `#global : C_Env.markup_ident -> bool` of the record is informing `C_Grammar_Rule_Lib.markup_var` if the variable being reported (at either first declaration time, or first use time) is global or local (inside a function for instance). Because once declared, the property `#global : C_Env.markup_ident -> bool` of a variable does not change afterwards, it is enough to store that information in the monadic environment:

- **Storing the information at declaration time** The part deciding if a variable being declared is global or not is implemented in `C_Grammar_Rule_Lib.doDeclIdent` and `C_Grammar_Rule_Lib.doFuncParamDeclIdent`. The two functions come from

https://github.com/visq/language-c/blob/master/src/Language/C/Parser/Parser.y (so do any functions in `C_Grammar_Rule_Lib`). Ultimately, they are both calling `C_Grammar_Rule_Lib.markup_var o C_Ast.Left` at some point.

- **Retrieving the information at use time** `C_Grammar_Rule_Lib.markup_var o C_Ast.Right` is only called by `C_Grammar_Rule_Wrap.primary_expression1`, while treating a variable being already declared. In particular the second argument of `C_Grammar_Rule_Lib.markup_var` is just provided by what has been computed by the above point when the variable was declared (e.g., the globality versus locality information).

2.3.3 Rewriting of AST node

For the case of rewriting a specific AST node, from subtree *T1* to subtree *T2*, it is useful to zoom on the different parsing evaluation stages, as well as make precise when the evaluation of semantic back-ends are starting.

1. Whereas annotations in Isabelle/C code have the potential of carrying arbitrary ML code (as in *Isabelle-C.C2*), the moment when they are effectively evaluated will not be discussed here, because to closely follow the semantics of the language in embedding (so C), we suppose comments — comprising annotations — may not affect any parsed tokens living outside comments. So no matter when annotations are scheduled to be future evaluated in Isabelle/C, the design decision of Isabelle/C is to not let a code do directive-like side-effects in annotations, such as changing *T1* to *T2* inside annotations.

2. To our knowledge, the sole category of code having the capacity to affect incoming stream of tokens are directives, which are processed and evaluated before the "major" parsing step occurs. Since in Isabelle/C, directives are relying on ML code, changing an AST node from *T1* to *T2* can then be perfectly implemented in directives.

3. After the directive (pre)processing step, the main parsing happens. But since what are driving the parsing engine are principally rule code, this step means to execute `C_Grammar_Rule_Lib` and `C_Grammar_Rule_Wrap`, i.e., rules in `../generated/c_grammar_fun.grm.sml`.

4. Once the parsing finishes, we have a final AST value, which topmost root type entry-point constitutes the last node built before the grammar parser https://github.com/visq/language-c/blob/master/src/Language/C/Parser/Parser.y ever entered in a stop state. For the case of a stop acceptance state, that moment happens when we reach the first rule code building the type `C_Ast.CTranslUnit`, since there is only one possible node making the parsing stop, according to what is currently written in the C grammar. (For the case of a state stopped due to an error, it is the last successfully built value that is returned, but to simplify the

discussion, we will assume in the rest of the document the parser is taking in input a fully well-parsed C code.)

5. By *semantic back-ends*, we denote any kind of "relatively efficient" compiled code generating Isabelle/HOL theorems, proofs, definitions, and so with the potential of generally generating Isabelle packages. In our case, the input of semantic back-ends will be the type `C_Ast.CTranslUnit` (actually, whatever value provided by the above parser). But since our parser is written in monadic style, it is as well possible to give slightly more information to semantic back-ends, such as the last monadic computed state, so including the last state of the parsing environment.

Generally, semantic back-ends can be written in full ML starting from `C_Ast.CTranslUnit`, but to additionally support formalizing tasks requiring to start from an AST defined in Isabelle/HOL, we provide an equivalent AST in HOL in the project, such as the one obtained after loading https://gitlab.lisn.upsaclay.fr/frederictuong/isabelle_contrib/-/blob/master/Citadelle/doc/Meta_C_generated.thy. (In fact, the ML AST is just generated from the HOL one.)

Based on the above information, there are now several *equivalent* ways to proceed for the purpose of having an AST node be mapped from *T1* to *T2*. The next bullets providing several possible solutions to follow are particularly sorted in increasing action time.

- *Before even starting the Isabelle system.* A first approach would be to modify the C code in input, by adding a directive `#define _ _` performing the necessary rewrite.

- *Before even starting the Isabelle system.* As an alternative of changing the C code, one can modify https://github.com/visq/language-c/blob/master/src/Language/C/Parser/Parser.y by hand, by explicitly writing *T2* at the specific position of the rule code generating *T1*. However, this solution implies to re-generate `../generated/c_grammar_fun.grm.sml`.

- *At grammar loading time, while the source of Isabelle/C is still being processed.* Instead of modifying the grammar, it should be possible to first locate which rule code is building *T1*. Then it would remain to retrieve and modify the respective function of `C_Grammar_Rule_Wrap` executed after that rule code, by providing a replacement function to be put in `C_Grammar_Rule_Wrap_Overloading`. However, as a design decision, wrapping functions generated in `../generated/c_grammar_fun.grm.sml` have only been generated to affect monadic states, not AST values. This is to prevent an erroneous replacement of an end-user while parsing C code. (It is currently left open about whether this feature will be implemented in future versions of the parser...)

- *At directive setup time, before executing any* **C** *command of interest.* Since the behavior of directives can be dynamically modified, this solution amounts to change

the semantics of any wished directive, appearing enough earlier in the code. (But for the overall code be in the end mostly compatible with any other C preprocessors, the implementation change has to be somehow at least consistent with how a preprocessor is already expected to treat an initial C un(pre)processed code.) For example, the current semantics of `#undef _` depends on what has been registered in `C_Context.directive_update` (see *Isabelle-C.C-Command*).

- *After parsing and obtaining a constructive value.* Another solution consists in directly writing a mapping function acting on the full AST, so writing a ML function of type `C_Ast.CTranslUnit -> C_Ast.CTranslUnit` (or a respective HOL function) which has to act on every constructor of the AST (so in the worst case about hundred of constructors for the considered AST, i.e., whenever a node has to be not identically returned). However, as we have already implemented a conversion function from `C_Ast.CTranslUnit` (subset of C11) to a subset AST of C99, it might be useful to save some effort by starting from this conversion function, locate where *T1* is pattern-matched by the conversion function, and generate *T2* instead.

  As example, the conversion function `C_Ast.main` is particularly used to connect the C11 front-end to the entry-point of AutoCorres in `l4v/src/tools/c-parser/StrictCParser.ML`.

- *At semantic back-ends execution time.* The above points were dealing with the cases where modification actions were all occurring before getting a final `C_Ast.CTranslUnit` value. But this does not mean it is forbidden to make some slight adjustments once that resulting `C_Ast.CTranslUnit` value obtained. In particular, it is the tasks of semantic back-ends to precisely work with `C_Ast.CTranslUnit` as starting point, and possibly translate it to another different type. So letting a semantic back-end implement the mapping from *T1* to *T2* would mean here to first understand the back-end of interest's architecture, to see where the necessary minimal modifications must be made.

  By taking l4v as a back-end example, its integration with Isabelle/C first starts with translating `C_Ast.CTranslUnit` to l4v's default C99 AST. Then various analyses on the obtained AST are performed in https://github.com/seL4/l4v/tree/master/tools/c-parser (the reader interested in the details can start by further exploring the ML files loaded by https://github.com/seL4/l4v/blob/master/tools/c-parser/CTranslation.thy). In short, to implement the mapping from *T1* to *T2* in the back-end part, one can either:

  – modify the translation from `C_Ast.CTranslUnit` to C99,

  – or modify the necessary ML files of interests in the l4v project.

More generally, to better inspect the list of rule code really executed when a C code is parsed, it might be helpful to proceed as in *Isabelle-C.C2*, by activating **declare**[[*C-parser-trace*]]. Then, the output window will display the sequence of Shift Reduce actions associated to the **C** command of interest.

NOTE :the C99 library is part of the configuration with AutoCorres, which is currently only available for Isabelle2019.

2.4 Case Study: Mapping on the Parsed AST via the `C11_Ast_Lib`

ML‹*open C11-Ast-Lib*›

A simpler alternative for connecting Isabelle/C to a semantic backend is the use of the `C11_Ast_Lib`, an API for the C11 abstract syntax. Among a number of utilities, it provides a family of iterators (or: hylomorphisms, generalized fold operators, or whatever terminology you prefer). There is a fold-operator for each C11 Ast-category :

1. `fold_cArraySize: 'a -> (node_content->'a->'b->'b) -> 'a C_Ast.cArraySize -> 'b -> 'b`
2. `fold_cCompoundBlockItem: (node_content->'a->'b->'b) -> 'a C_Ast.cCompoundBlockItem->'b->'b`
3. `fold_cArraySize: 'a -> (node_content->'a->'b->'b) -> 'a C_Ast.cArraySize -> 'b -> 'b`
4. `fold_cDeclaration: (node_content->'a->'b->'b) -> 'a C_Ast.cDeclaration -> 'b -> 'b`
5. `fold_cExpression: (node_content->'a->'b->'b) -> 'a C_Ast.cExpression -> 'b -> 'b`
6. `fold_cStatement: (node_content->'a->'b->'b) -> 'a C_Ast.cStatement -> 'b -> 'b`
7. `fold_cExternalDeclaration: (node_content->'a->'b->'b) -> 'a C_Ast.cExternalDeclaration->'b->'b`
8. `fold_cTranslationUnit: (node_content->'a->'b->'b) -> 'a C_Ast.cTranslationUnit -> 'b -> 'b`
9. etc.

Here, `node_content` is a data-structure providing untyped and uniform information on which rule has been applied, and what kind of particular decoration appears in the C11-Ast.

This allows for a simple programming of queries like "get the list of identifiers" directly on the C11-Ast. Moreover, it is pretty straight-forward to program a compiler to λ-terms for a specific semantic interpretation in Isabelle/HOL. A simple example is here:

declare [[C_{rule0} = *expression*]]
C‹$a + b * c - a / b$›
ML‹*val ast-expr* = @{*C11-CExpr*}›

```
fun selectIdent0Binary (a as { tag, sub_tag, args }:C11_Ast_Lib.node_content)
                       (b:  C_Ast.nodeInfo )
                       (c : term list)=
    case tag of
      "Ident0" => (node_content_2_free a)::c
     |"CBinary0" => (case (drop_dark_matter sub_tag, c) of
                       ("CAddOp0",b::a::R) => (Const("Groups.plus_class.plus",dummyT)
$ a $ b :: R)
                     | ("CMulOp0",b::a::R) => (Const("Groups.times_class.times",dummyT)
$ a $ b :: R)
                     | ("CDivOp0",b::a::R) => (Const("Rings.divide_class.divide",dummyT)
$ a $ b :: R)
                     | ("CSubOp0",b::a::R) => (Const("Groups.minus_class.minus",dummyT)
$ a $ b :: R)
                     | _ => (writeln ("sub_tag all " ^sub_tag^" :>> "^
@{make_string} c);c ))
     | _ => c;


val S =  (C11_Ast_Lib.fold_cExpression selectIdent0Binary ast_expr []);
(* gives the (untyped) equivalent of : *)
val S' = @{term "a + b * c - a / b"};
```

This snippet is drawn from the C11-Example shown in Appendix III.

2.5 Known Limitations, Troubleshooting

2.5.1 The Document Model of the Isabelle/PIDE (applying since at least Isabelle 2019)

Introduction

Embedding C directives in C code is an act of common practice in numerous applications, as well as largely highlighted in the C standard. As an example of frequently encountered directives, *#include <some-file.c>* is used to insert the content of *some-file.c* at the place where it is written. In Isabelle/C, we can also write a C code containing directives like *#include*, and generally the PIDE reporting of directives is supported to a certain extent. Yet, the dynamic inclusion of arbitrary file with *#include* is hurting a certain technological barrier. This is related to how the document model of Isabelle 2019 is functioning, but also to the design decisions behind the implementation of **C**‹ .. ›. Thus, providing a complete semantic implementation of *#include* might be not

as evident as usual, if not more dangerous, i.e. "something requiring a manual intervention in the source of Isabelle 2019". In the next part, we show why in our current implementation of Isabelle/C there is no way for user programmed extensions to exploit implicit dependencies between sub-documents in pure ML: a sub-document referred to via *#include <some-file>* will not lead to a reevaluation of a **C**‹ .. › command whenever modified.

Embedding a language in Isabelle/PIDE

To clarify why the way a language being embedded in Isabelle is influencing the interaction between a future parser of the language with the Isabelle's document model, we recall the two "different" ways of embedding a language in Isabelle/PIDE.

At its most basic form, the general syntactic scope of an Isabelle/Isar document can be seen as being composed of two syntactic alternations of editing space: fragments of the inner syntax language, themselves part of the more general outer syntax (the inner syntax is implemented as an atomic entity of the outer language); see `~~/src/Doc/Isar_Ref/Outer_Syntax.thy`. So strictly speaking, when attempting to support a new language L in Isabelle, there is always the question of fine-grain estimating which subsets of L will be represented in the outer syntax part, and if it possibly remains any left subsets to be represented on the more inner (syntactic) part.

Generally, to answer this question, there are several criteria to consider:

- Are there any escaping symbols conflicting between L and the outer (syntax) language, including for example the ASCII `"` or `‘`?
- Is L a realistic language, i.e. more complex than any combinations of outer named tokens that can be ever covered in terms of expressivity power (where the list of outer named tokens is provided in `~~/src/Doc/Isar_Ref/Outer_Syntax.thy`)?
- Is it preferable of not altering the outer syntax language with too specific and challenging features of L? This is particularly true since in Isabelle 2019, there is no way of modifying the outer syntax without making the modifications irremediably happen on its source code.

For the above reasons, we have come up in Isabelle/C with the choice of making the full C language be supported inside the inner syntax allocated space. In particular, this has become all the more syntactically easy with the introduction of cartouches since Isabelle 2014.[5] However, for the case of the C language, certain C directives like *#include* are meant to heavily interact with external files. In particular, resources would be best utilized if we were taking advantage of the Isabelle's asynchronous document model

[5] Fortunately, parsing tokens of C do not strongly conflict with cartouche delimiter symbols. For example, it should not be ethically wrong in C to write an opening cartouche symbol (possibly in a C comment) without writing any closing cartouche symbol afterwards. However, we have not encountered such C code in our tested codebase, and it is a functionality implicitly rejected by the current parser of Isabelle/C, as it is relying on Isabelle 2019's parser combinator library for the lexing part.

for such interaction task. Unfortunately, the inner syntax space only has a minimum interaction with the document model, compared to the outer syntax one. Otherwise said, be it for experimenting the inner syntax layer and see how far it can deal with the document layer, or otherwise reimplementing parts of Isabelle/C in the outer syntax layer, the two solutions are conducting to do modifications in the Isabelle 2019 source code.

Note that the language embedding space of **C** closely resembles to how ML sources are delimited within a **ML** command. Additionally, in ML, one can use antiquotations to also refer to external files (particularly in formal comments). Still, the problem is still present in ML: referred files are not loaded in the document model.

Examples

- Commands declared as of type *thy-decl* in the theory header are scheduled to be executed once. Additionally, they are not tracking the content of file names provided in argument, so a change there will not trigger a reevaluation of the associated command. For example, even if the type of **ML-file** is not *thy-decl*, nothing prevents one to set it with *thy-decl* as type. In particular, by doing so, it is no more possible to CTRL-hover-click on the file name written after **ML-file**.

- To make a command C track the content of *file*, whenever the file is changing, setting C to be of type *thy-load* in the theory header is a first step, but not enough. To be effective, *file* must also be loaded, by either explicitly opening it, or clicking on its name after the command. Examples of commands in this situation requiring a preliminary one-time-click include: **external-file**, **bibtex-file**, **ML-file**. Internally, the click is bound to a Scala code invoking a request to make an asynchronous dependency to the newly opened document at ML side.

- In terms of recursivity, for the case of a chain of sub-documents of the form (a theory file containing: **C-file** ‹*file0.c*›) $\Longrightarrow$ (C file `file0.c` containing: `#include <file1.c>`) $\Longrightarrow$ (C file `file1.c` containing: `#include <file2.c>`) $\Longrightarrow$ (C file `file2.c` containing: `#include <file3.c>`), we ideally expect a modification in `file3.c` be taken into account in all ancestor files including the initial theory, provoking the associated command of the theory be reevaluated. However in C, directives resolving might be close to Turing-complete. For instance, one can also include files based on particular conditional situations: `#if _`

```
    #include <file1>
  #else
    #include <file2>
    #include <file3>
  #endif
```

- When a theory is depending on other theories (such as *Isabelle-C.C-Eval* depending on *Isabelle-C.C-Parser-Language* and *Isabelle-C.C-Parser-Annotation*), modifying the list of theories in importation automatically triggers what the user is expecting: for example, the newly added theories are dynamically imported, any change by another external editor makes everything consequently propagated.

  Following the internal implementation of the document model engine, we basically distinguish two phases of document importation: either at start-up time, or dynamically on user requests. Although the case of start-up time can be handled in pure ML side, the language dedicated to express which Isabelle theory files to import is less powerful than the close-to-Turing-completeness expressivity of C directives. On the other hand, the dynamic importation of files on user requests seems to be performed (at the time of writing) through a too high level ML protocol, mostly called from Scala side. Due to the fact that Isabelle/C is currently implemented in pure ML, a solution also in pure ML would thus sound more natural (although we are not excluding solutions interacting with Scala, as long as the resulting can be implemented in Isabelle, preferably outside of its own source).

2.5.2 Parsing Error versus Parsing Correctness

When trying to decide if the next parsing action is a Shift or Reduce action to perform, the grammar simulator `LALR_Parser_Eval.parse` can actually decide to do another action: ignore everything and immediately stop the simulation.

If the parser ever decides to stop, this can only be for two reasons:

- The parser is supposed to have correctly finished its parsing task, making it be in an acceptance state. As acceptance states are encoded in the grammar, it is easy to find if this information is correct, or if it has to be adjusted in more detail by inspecting https://github.com/visq/language-c/blob/master/src/Language/C/Parser/Parser.y (`../generated/c_grammar_fun.grm.sml`).

- The parser seems to be unable to correctly finish its parsing task. In this case, the user will see an error be explicitly raised by the prover IDE. However raising an error is just the default behavior of Isabelle/C: the decision to whether raise interruptive errors ultimately depends on how front-end commands are implemented (such as **C**, **C-file**, etc.). For instance, similarly as to how outer syntax commands are implemented, we can imagine a tool implementing a kind of partial parsing, analyzing the longest sequence of well-formed input, and discarding some strategic next set of faulty tokens with a well suited informative message, so that the parsing process could be maximally repeated on what is coming afterwards.

  Currently, the default behavior of Isabelle/C is to raise the error defined in `C_Module.err` at the very first opportunity [6]. The possible solutions to make the error disappear at the position the error is indicated can be detailed as follows:

[6] At the time of writing it is: *No matching grammar rule.*

– Modifying the C code in input would be a first solution whenever we suspect something is making it erroneous (and when we have a reason to believe that the grammar is behaving as it should).

– However, we could still get the above error in front of an input where one is usually expecting to see not causing a failure. In particular, there are several C features (such as C directives) explicitly left for semantic back-ends (pre-) processing, so in general not fully semantically processed at parsing time.

  For example, whereas the code `#define i int`

  `i a;` succeeds, replacing its first line with the directive `#include <file.c>` will not initially work, even if *file.c* contains `#define i int`, as the former directive has been left for semantic back-end treatment. One way of solving this would be to modify the C code in input for it to be already preprocessed (without directives, for example the C example of https://gitlab.lisn.upsaclay.fr/burkhart.wolff/Isabelle_C/-/blob/C/C11-BackEnds/AutoCorres_wrapper/examples/TestSEL4.thy is already provided as preprocessed). Another way would be adding a specific new semantic back-end implementing the automation of the preprocessing task (as done in https://gitlab.lisn.upsaclay.fr/burkhart.wolff/Isabelle_C/-/blob/C/C11-BackEnds/AutoCorres_wrapper/examples/IsPrime_linear_CCT.thy, where the back-end explicitly makes a call to *cpp* at run-time).

– Ultimately, modifying the grammar with new rules cancelling the exception would only work if the problem really relies on the grammar, as it was mentioned for the acceptance state.

In terms of parsing correctness, Isabelle/C provides at least two different parsers:

- a parser limited to C99/C11 code provided in `../../C11-FrontEnd` that can parse certain liberal extensions out of the C standard [7];
- and another parser accepting C99/C11/C18 code in https://gitlab.lisn.upsaclay.fr/burkhart.wolff/Isabelle_C/-/tree/C/C18-FrontEnd that is close to the C standard while focusing on resolving ambiguities of the standard [8] [13].

Note that the two parsers are not accepting/rejecting the same range of arbitrary C code. We have actually already encountered situations where an error is raised by one parser, while a success is happening with the other parser (and vice-versa). Consequently, in front of a C code, it can be a good recommendation to try out the parsing with all possible parsers of Isabelle/C. In any cases, a failure in one or several activated parsers might not be necessarily harmful: it might also indicate that a wrong parser has been selected, or a semantic back-end not yet supporting aspects of the C code being parsed.

[7] http://hackage.haskell.org/package/language-c
[8] https://github.com/jhjourdan/C11parser

2.5.3 Exporting C Files to the File-System

From the Isabelle/C side, the task is easy, just type:

C-export-file

... which does the trick and generates a file `C_Appendices.c`. But hold on — where is it? Well, Isabelle/C uses since version Isabelle2019 a virtual file-system. Exporting from it to the real file-system requires a few mouse-clicks (unfortunately).

So activating the command **C-export-file** leads to the output `See theory exports "C/*/C_Appendices.c"` (see Figure 2.1), and clicking on the highlighted `theory exports` lets Isabelle display a part of the virtual file-system (see subwidget left). Activating it in the subwidget lets jEdit open it as an editable file, which can be exported via "*File* → *Save As...*" into the real file-system.

end

3 Annex I: The Commented Sources of Isabelle/C

3.1 Core Language: An Abstract Syntax Tree Definition (C Language without Annotations)

theory *C-Ast*
  imports *Main*
begin

3.1.1 Loading the Generated AST

The abstract syntax tree of the C language considered in the Isabelle/C project is arbitrary, but it must already come with a grammar making the connection with a default AST, so that both the grammar and AST can be imported to SML.[1] The Haskell Language.C project fulfills this property: see for instance http://hackage.haskell.org/package/language-c and https://github.com/visq/language-c/blob/master/src/Language/C/Syntax/AST.hs, where its AST is being imported in the present theory file `C_Ast.thy`, whereas its grammar will be later in `C_Parser_Language.thy` (`C_Parser_Language.thy` depends on `C_Ast.thy`). The AST importation is based on a modified version of Haskabelle, which generates the C AST from Haskell to an ML file.

ML — `../generated/c_ast.ML` ‹
val fresh-ident0 =
  let val i = Synchronized.var counter for new identifier 38 in
    fn () => Int.toString (Synchronized.change-result i (fn i => (i, i + 1)))
  end
›

ML — `../generated/c_ast.ML` ‹
— https://gitlab.lisn.upsaclay.fr/burkhart.wolff/Isabelle_C/-/blob/C/Citadelle/src/compiler_generic/meta_isabelle/Printer_init.thy
structure CodeType = struct
  type mlInt = string
  type 'a mlMonad = 'a option
end

structure CodeConst = struct
  structure Monad = struct

[1] Additionally, the grammar and AST must both have a free licence — compatible with the Isabelle AFP, for them to be publishable there.

```
  val bind = fn
      NONE => (fn - => NONE)
    | SOME a => fn f => f a
  val return = SOME
 end

 structure Printf = struct
  local
   fun sprintf s l =
     case String.fields (fn #"%" => true | - => false) s of
       [] =>
     | [x] => x
     | x :: xs =>
         let fun aux acc l-pat l-s =
           case l-pat of
             [] => rev acc
           | x :: xs => aux (String.extract (x, 1, NONE) :: hd l-s :: acc) xs (tl l-s) in
         String.concat (x :: aux [] xs l)
   end
  in
   fun sprintf0 s-pat = s-pat
   fun sprintf1 s-pat s1 = sprintf s-pat [s1]
   fun sprintf2 s-pat s1 s2 = sprintf s-pat [s1, s2]
   fun sprintf3 s-pat s1 s2 s3 = sprintf s-pat [s1, s2, s3]
   fun sprintf4 s-pat s1 s2 s3 s4 = sprintf s-pat [s1, s2, s3, s4]
   fun sprintf5 s-pat s1 s2 s3 s4 s5 = sprintf s-pat [s1, s2, s3, s4, s5]
  end
 end

 structure String = struct
  val concat = String.concatWith
 end

 structure Sys = struct
  val isDirectory2 = SOME o File.is-dir o Path.explode handle ERROR - => K NONE
 end

 structure To = struct
  fun nat f = Int.toString o f
 end

 fun outFile1 - - = tap (fn - => warning not implemented) NONE
 fun outStand1 f = outFile1 f
end
›
```

ML-file ‹../generated/c-ast.ML›

Note that the purpose of `../generated` is to host any generated files of the Isabelle/C project. It contains among others:

- `../generated/c_ast.ML` representing the Abstract Syntax Tree of C, which has just been loaded.
- `../generated/c_grammar_fun.grm` is a generated file not used by the project, except for further generating `../generated/c_grammar_fun.grm.sig` and `../generated/c_grammar_fun.grm.sml`. Its physical presence in the directory is actually not necessary, but has been kept for informative documentation purposes. It represents the basis point of our SML grammar file, generated by an initial Haskell grammar file (namely https://github.com/visq/language-c/blob/master/src/Language/C/Parser/Parser.y) using a modified version of Happy.
- `../generated/c_grammar_fun.grm.sig` and `../generated/c_grammar_fun.grm.sml` are the two files generated from `../generated/c_grammar_fun.grm` with a modified version of ML-Yacc. This last comes from MLton source in `../../src_ext/mlton`, see for example `../../src_ext/mlton/mlyacc`.

For the case of `../generated/c_ast.ML`, it is actually not mandatory to have a "physical" representation of the file in `../generated`: it could be generated "on-the-fly" with **code-reflect** and immediately loaded: Citadelle has an option to choose between the two tasks [22].[2]

After loading the AST, it is possible in Citadelle to do various meta-programming renaming, such as the one depicted in the next command. Actually, its content is explicitly included here by hand since we decided to manually load the AST using the above **ML-file** command. (Otherwise, one can automatically execute the overall generation and renaming tasks in Citadelle without resorting to a manual copying-pasting.)

ML — `../generated/c_ast.ML` ‹
structure C-Ast =
struct
val Position = C-Ast.position val NoPosition = C-Ast.noPosition val BuiltinPosition = C-Ast.builtinPosition val InternalPosition = C-Ast.internalPosition val Name = C-Ast.name val OnlyPos = C-Ast.onlyPos val NodeInfo = C-Ast.nodeInfo val AnonymousRef = C-Ast.anonymousRef val NamedRef = C-Ast.namedRef val CChar = C-Ast.cChar val CChars = C-Ast.cChars val DecRepr = C-Ast.decRepr val HexRepr = C-Ast.hexRepr val OctalRepr = C-Ast.octalRepr val FlagUnsigned = C-Ast.flagUnsigned val FlagLong = C-Ast.flagLong val FlagLongLong = C-Ast.flagLongLong val FlagImag = C-Ast.flagImag val CFloat = C-Ast.cFloat val Flags = C-Ast.flags val CInteger = C-Ast.cInteger val CAssignOp = C-Ast.cAssignOp val CMulAssOp = C-Ast.cMulAssOp val CDivAssOp = C-Ast.cDivAssOp val CRmdAssOp = C-Ast.cRmdAssOp val CAddAssOp = C-Ast.cAddAssOp val CSubAssOp = C-Ast.cSubAssOp val CShlAssOp = C-Ast.cShlAssOp val CShrAssOp = C-Ast.cShrAssOp val CAndAssOp = C-Ast.cAndAssOp val CXorAssOp = C-Ast.cXorAssOp val COrAssOp = C-Ast.cOrAssOp val CMulOp = C-Ast.cMulOp val CDivOp = C-Ast.cDivOp val CRmdOp = C-Ast.cRmdOp val CAddOp = C-Ast.cAddOp val CSubOp = C-Ast.cSubOp val CShlOp = C-Ast.cShlOp val CShrOp = C-Ast.cShrOp val CLeOp = C-Ast.cLeOp val CGrOp

[2] https://gitlab.lisn.upsaclay.fr/frederictuong/isabelle_contrib/-/tree/master/Citadelle/src/compiler

= C-Ast.cGrOp val CLeqOp = C-Ast.cLeqOp val CGeqOp = C-Ast.cGeqOp val CEqOp = C-Ast.cEqOp val CNeqOp = C-Ast.cNeqOp val CAndOp = C-Ast.cAndOp val CXorOp = C-Ast.cXorOp val COrOp = C-Ast.cOrOp val CLndOp = C-Ast.cLndOp val CLorOp = C-Ast.cLorOp val CPreIncOp = C-Ast.cPreIncOp val CPreDecOp = C-Ast.cPreDecOp val CPostIncOp = C-Ast.cPostIncOp val CPostDecOp = C-Ast.cPostDecOp val CAdrOp = C-Ast.cAdrOp val CIndOp = C-Ast.cIndOp val CPlusOp = C-Ast.cPlusOp val CMinOp = C-Ast.cMinOp val CCompOp = C-Ast.cCompOp val CNegOp = C-Ast.cNegOp val CAuto = C-Ast.cAuto val CRegister = C-Ast.cRegister val CStatic = C-Ast.cStatic val CExtern = C-Ast.cExtern val CTypedef = C-Ast.cTypedef val CThread = C-Ast.cThread val CInlineQual = C-Ast.cInlineQual val CNoreturnQual = C-Ast.cNoreturnQual val CStructTag = C-Ast.cStructTag val CUnionTag = C-Ast.cUnionTag val CIntConst = C-Ast.cIntConst val CCharConst = C-Ast.cCharConst val CFloatConst = C-Ast.cFloatConst val CStrConst = C-Ast.cStrConst val CStrLit = C-Ast.cStrLit val CFunDef = C-Ast.cFunDef val CDecl = C-Ast.cDecl val CStaticAssert = C-Ast.cStaticAssert val CDeclr = C-Ast.cDeclr val CPtrDeclr = C-Ast.cPtrDeclr val CArrDeclr = C-Ast.cArrDeclr val CFunDeclr = C-Ast.cFunDeclr val CNoArrSize = C-Ast.cNoArrSize val CArrSize = C-Ast.cArrSize val CLabel = C-Ast.cLabel val CCase = C-Ast.cCase val CCases = C-Ast.cCases val CDefault = C-Ast.cDefault val CExpr = C-Ast.cExpr val CCompound = C-Ast.cCompound val CIf = C-Ast.cIf val CSwitch = C-Ast.cSwitch val CWhile = C-Ast.cWhile val CFor = C-Ast.cFor val CGoto = C-Ast.cGoto val CGotoPtr = C-Ast.cGotoPtr val CCont = C-Ast.cCont val CBreak = C-Ast.cBreak val CReturn = C-Ast.cReturn val CAsm = C-Ast.cAsm val CAsmStmt = C-Ast.cAsmStmt val CAsmOperand = C-Ast.cAsmOperand val CBlockStmt = C-Ast.cBlockStmt val CBlockDecl = C-Ast.cBlockDecl val CNestedFunDef = C-Ast.cNestedFunDef val CStorageSpec = C-Ast.cStorageSpec val CTypeSpec = C-Ast.cTypeSpec val CTypeQual = C-Ast.cTypeQual val CFunSpec = C-Ast.cFunSpec val CAlignSpec = C-Ast.cAlignSpec val CVoidType = C-Ast.cVoidType val CCharType = C-Ast.cCharType val CShortType = C-Ast.cShortType val CIntType = C-Ast.cIntType val CLongType = C-Ast.cLongType val CFloatType = C-Ast.cFloatType val CDoubleType = C-Ast.cDoubleType val CSignedType = C-Ast.cSignedType val CUnsigType = C-Ast.cUnsigType val CBoolType = C-Ast.cBoolType val CComplexType = C-Ast.cComplexType val CInt128Type = C-Ast.cInt128Type val CSUType = C-Ast.cSUType val CEnumType = C-Ast.cEnumType val CTypeDef = C-Ast.cTypeDef val CTypeOfExpr = C-Ast.cTypeOfExpr val CTypeOfType = C-Ast.cTypeOfType val CAtomicType = C-Ast.cAtomicType val CConstQual = C-Ast.cConstQual val CVolatQual = C-Ast.cVolatQual val CRestrQual = C-Ast.cRestrQual val CAtomicQual = C-Ast.cAtomicQual val CAttrQual = C-Ast.cAttrQual val CNullableQual = C-Ast.cNullableQual val CNonnullQual = C-Ast.cNonnullQual val CAlignAsType = C-Ast.cAlignAsType val CAlignAsExpr = C-Ast.cAlignAsExpr val CStruct = C-Ast.cStruct val CEnum = C-Ast.cEnum val CInitExpr = C-Ast.cInitExpr val CInitList = C-Ast.cInitList val CArrDesig = C-Ast.cArrDesig val CMemberDesig = C-Ast.cMemberDesig val CRangeDesig = C-Ast.cRangeDesig val CAttr = C-Ast.cAttr val CComma = C-Ast.cComma val CAssign = C-Ast.cAssign val CCond = C-Ast.cCond val CBinary = C-Ast.cBinary val CCast = C-Ast.cCast val CUnary = C-Ast.cUnary val CSizeofExpr = C-Ast.cSizeofExpr val CSizeofType = C-Ast.cSizeofType val CAlignofExpr = C-Ast.cAlignofExpr val CAlignofType = C-Ast.cAlignofType val CComplexReal = C-Ast.cComplexReal val CComplexImag = C-Ast.cComplexImag val CIndex = C-Ast.cIndex val CCall = C-Ast.cCall val CMember = C-Ast.cMember val CVar = C-Ast.cVar val CConst = C-Ast.cConst val CCompoundLit = C-Ast.cCompoundLit val CGenericSelection = C-Ast.cGenericSelection val CStatExpr = C-Ast.cStatExpr val CLabAddrExpr = C-Ast.cLabAddrExpr val CBuiltinExpr = C-Ast.cBuiltinExpr val CBuiltinVaArg = C-Ast.cBuiltinVaArg val CBuiltinOffsetOf = C-Ast.cBuiltinOffsetOf val CBuiltinTypesCompatible = C-Ast.cBuiltinTypesCompatible val CDe-

```
clExt = C-Ast.cDeclExt val CFDefExt = C-Ast.cFDefExt val CAsmExt = C-Ast.cAsmExt val
CTranslUnit = C-Ast.cTranslUnit
  open C-Ast
end
›
```

3.1.2 Basic Aliases and Initialization of the Haskell Library

ML — `../generated/c_ast.ML` ‹

```
structure C-Ast =
struct
type class-Pos = Position.T * Position.T
(**)
type NodeInfo = C-Ast.nodeInfo
type CStorageSpec = NodeInfo C-Ast.cStorageSpecifier
type CFunSpec = NodeInfo C-Ast.cFunctionSpecifier
type CConst = NodeInfo C-Ast.cConstant
type 'a CInitializerList = ('a C-Ast.cPartDesignator List.list * 'a C-Ast.cInitializer) List.list
type CTranslUnit = NodeInfo C-Ast.cTranslationUnit
type CExtDecl = NodeInfo C-Ast.cExternalDeclaration
type CFunDef = NodeInfo C-Ast.cFunctionDef
type CDecl = NodeInfo C-Ast.cDeclaration
type CDeclr = NodeInfo C-Ast.cDeclarator
type CDerivedDeclr = NodeInfo C-Ast.cDerivedDeclarator
type CArrSize = NodeInfo C-Ast.cArraySize
type CStat = NodeInfo C-Ast.cStatement
type CAsmStmt = NodeInfo C-Ast.cAssemblyStatement
type CAsmOperand = NodeInfo C-Ast.cAssemblyOperand
type CBlockItem = NodeInfo C-Ast.cCompoundBlockItem
type CDeclSpec = NodeInfo C-Ast.cDeclarationSpecifier
type CTypeSpec = NodeInfo C-Ast.cTypeSpecifier
type CTypeQual = NodeInfo C-Ast.cTypeQualifier
type CAlignSpec = NodeInfo C-Ast.cAlignmentSpecifier
type CStructUnion = NodeInfo C-Ast.cStructureUnion
type CEnum = NodeInfo C-Ast.cEnumeration
type CInit = NodeInfo C-Ast.cInitializer
type CInitList = NodeInfo CInitializerList
type CDesignator = NodeInfo C-Ast.cPartDesignator
type CAttr = NodeInfo C-Ast.cAttribute
type CExpr = NodeInfo C-Ast.cExpression
type CBuiltin = NodeInfo C-Ast.cBuiltinThing
type CStrLit = NodeInfo C-Ast.cStringLiteral
(**)
type ClangCVersion = C-Ast.clangCVersion
type Ident = C-Ast.ident
type Position = C-Ast.positiona
type PosLength = Position * int
type Name = C-Ast.namea
type Bool = bool
```

```
type CString = C-Ast.cString
type CChar = C-Ast.cChar
type CInteger = C-Ast.cInteger
type CFloat = C-Ast.cFloat
type CStructTag = C-Ast.cStructTag
type CUnaryOp = C-Ast.cUnaryOp
type 'a CStringLiteral = 'a C-Ast.cStringLiteral
type 'a CConstant = 'a C-Ast.cConstant
type ('a, 'b) Either = ('a, 'b) C-Ast.either
type CIntRepr = C-Ast.cIntRepr
type CIntFlag = C-Ast.cIntFlag
type CAssignOp = C-Ast.cAssignOp
type Comment = C-Ast.comment
(**)
type 'a Reversed = 'a
datatype CDeclrR = CDeclrR0 of C-Ast.ident C-Ast.optiona
                         * NodeInfo C-Ast.cDerivedDeclarator list Reversed
                         * NodeInfo C-Ast.cStringLiteral C-Ast.optiona
                         * NodeInfo C-Ast.cAttribute list
                         * NodeInfo
type 'a Maybe = 'a C-Ast.optiona
datatype 'a Located = Located of 'a * (Position * (Position * int))
(**)
fun CDeclrR ide l s a n = CDeclrR0 (ide, l, s, a, n)
val reverse = rev
val Nothing = C-Ast.None
val Just = C-Ast.Some
val False = false
val True = true
val Ident = C-Ast.Ident0
(**)
val CDecl-flat = fn l1 => C-Ast.CDecl l1 o map (fn (a, b, c) => ((a, b), c))
fun flat3 (a, b, c) = ((a, b), c)
fun maybe def f = fn C-Ast.None => def | C-Ast.Some x => f x
val Reversed = I
(**)
val From-string =
    C-Ast.SS-base
  o C-Ast.ST
  o implode
  o map (fn s => — prevent functions in ~~/src/HOL/String.thy of raising additional errors
(e.g., see the ML code associated to String.asciis-of-literal)
        if Symbol.is-char s then s
        else if Symbol.is-utf8 s then translate-string (fn c => \\ ^ string-of-int (ord c)) s
        else s)
  o Symbol.explode
val From-char-hd = hd o C-Ast.explode
(**)
val Namea = C-Ast.name
```

```
(**)
open C-Ast
fun flip f b a = f a b
open Basic-Library
end
›
```

3.2 A General C11-AST iterator.

ML‹

```
signature C11-AST-LIB =
  sig
    (* some general combinators *)
    val fold-either: ('a -> 'b -> 'c) -> ('d -> 'b -> 'c) -> ('a, 'd) C-Ast.either -> 'b ->
'c
    val fold-optiona: ('a -> 'b -> 'b) -> 'a C-Ast.optiona -> 'b -> 'b

    datatype data = data-bool of bool     | data-int of int
                  | data-string of string | data-absstring of string

    type node-content = { tag     : string,
                          sub-tag : string,
                          args    : data list }

    (* conversions of enumeration types to string codes *)
    val toString-cBinaryOp : C-Ast.cBinaryOp -> string
    val toString-cIntFlag  : C-Ast.cIntFlag -> string
    val toString-cIntRepr  : C-Ast.cIntRepr -> string
    val toString-cUnaryOp  : C-Ast.cUnaryOp -> string
    val toString-cAssignOp : C-Ast.cAssignOp -> string
    val toString-abr-string: C-Ast.abr-string -> string
    val toString-nodeinfo  : C-Ast.nodeInfo -> string


    (* a generic iterator collection over the entire C11 - AST. The lexical
       leaves of the AST's are parametric ('a). THe collecyot function g (see below)
       gets as additional parameter a string-key representing its term key
       (and sometimes local information such as enumeration type keys). *)
    (* Caveat : Assembly is currently not supported *)
    (* currently a special case since idents are not properly abstracted in the src files of the
       AST generation: *)
    val fold-ident: 'a -> (node-content -> 'a -> 'b -> 'c) -> C-Ast.ident -> 'b -> 'c
    (* the Leaf has to be delivered from the context, the internal non-parametric nodeinfo
       is currently ignored. HACK. *)

    val fold-cInteger: (node-content -> 'a -> 'b) -> C-Ast.cInteger -> 'a -> 'b
    val fold-cConstant: (node-content -> 'a -> 'b -> 'b) -> 'a C-Ast.cConstant -> 'b -> 'b
    val fold-cStringLiteral: (node-content -> 'a -> 'b -> 'b) -> 'a C-Ast.cStringLiteral ->
```

```
'b -> 'b


    val fold-cArraySize: 'a -> (node-content -> 'a -> 'b -> 'b) -> 'a C-Ast.cArraySize ->
'b -> 'b
    val fold-cAttribute: (node-content -> 'a -> 'b -> 'b) -> 'a C-Ast.cAttribute -> 'b -> 'b
    val fold-cBuiltinThing: (node-content -> 'a -> 'b -> 'b) -> 'a C-Ast.cBuiltinThing ->
'b -> 'b
    val fold-cCompoundBlockItem: (node-content -> 'a -> 'b -> 'b)
                                 -> 'a C-Ast.cCompoundBlockItem -> 'b -> 'b
    val fold-cDeclaration: (node-content -> 'a -> 'b -> 'b) -> 'a C-Ast.cDeclaration -> 'b
-> 'b
    val fold-cDeclarationSpecifier: (node-content -> 'a -> 'b -> 'b)
                                 -> 'a C-Ast.cDeclarationSpecifier -> 'b -> 'b
   val fold-cDeclarator: (node-content -> 'a -> 'b -> 'b) -> 'a C-Ast.cDeclarator -> 'b ->
'b
    val fold-cDerivedDeclarator: (node-content -> 'a -> 'b -> 'b)
                                 -> 'a C-Ast.cDerivedDeclarator -> 'b -> 'b
    val fold-cEnumeration: (node-content -> 'a -> 'b -> 'b) -> 'a C-Ast.cEnumeration ->
'b -> 'b
     val fold-cExpression: (node-content -> 'a -> 'b -> 'b) -> 'a C-Ast.cExpression -> 'b
-> 'b
    val fold-cInitializer: (node-content -> 'a -> 'b -> 'b) -> 'a C-Ast.cInitializer -> 'b ->
'b
    val fold-cPartDesignator: (node-content -> 'a -> 'b -> 'b) -> 'a C-Ast.cPartDesignator
-> 'b -> 'b
    val fold-cStatement: (node-content -> 'a -> 'b -> 'b) -> 'a C-Ast.cStatement -> 'b ->
'b
    val fold-cStructureUnion : (node-content -> 'a -> 'b -> 'b) -> 'a C-Ast.cStructureUnion
-> 'b -> 'b
    val fold-cTypeQualifier: (node-content -> 'a -> 'b -> 'b) -> 'a C-Ast.cTypeQualifier ->
'b -> 'b
    val fold-cTypeSpecifier: (node-content -> 'a -> 'b -> 'b) -> 'a C-Ast.cTypeSpecifier ->
'b -> 'b
         val fold-cExternalDeclaration: (node-content -> 'a -> 'b -> 'b) -> 'a
C-Ast.cExternalDeclaration -> 'b -> 'b
    val fold-cTranslationUnit: (node-content -> 'a -> 'b -> 'b) -> 'a C-Ast.cTranslationUnit
-> 'b -> 'b

  (* universal sum type : *)

    datatype 'a C11-Ast =
          mk-cInteger              of    C-Ast.cInteger
        | mk-cConstant             of 'a C-Ast.cConstant
        | mk-cStringLiteral        of 'a C-Ast.cStringLiteral
        | mk-cArraySize            of 'a C-Ast.cArraySize
        | mk-cAttribute            of 'a C-Ast.cAttribute
        | mk-cBuiltinThing         of 'a C-Ast.cBuiltinThing
        | mk-cCompoundBlockItem    of 'a C-Ast.cCompoundBlockItem
```

```
  | mk-cDeclaration           of 'a C-Ast.cDeclaration
  | mk-cDeclarationSpecifier of 'a C-Ast.cDeclarationSpecifier
  | mk-cDeclarator           of 'a C-Ast.cDeclarator
  | mk-cDerivedDeclarator    of 'a C-Ast.cDerivedDeclarator
  | mk-cEnumeration          of 'a C-Ast.cEnumeration
  | mk-cExpression           of 'a C-Ast.cExpression
  | mk-cInitializer          of 'a C-Ast.cInitializer
  | mk-cPartDesignator       of 'a C-Ast.cPartDesignator
  | mk-cStatement            of 'a C-Ast.cStatement
  | mk-cStructureUnion       of 'a C-Ast.cStructureUnion
  | mk-cTypeQualifier        of 'a C-Ast.cTypeQualifier
  | mk-cTypeSpecifier        of 'a C-Ast.cTypeSpecifier
  | mk-cStructTag            of C-Ast.cStructTag
  | mk-cUnaryOp              of C-Ast.cUnaryOp
  | mk-cAssignOp             of C-Ast.cAssignOp
  | mk-cBinaryOp             of C-Ast.cBinaryOp
  | mk-cIntFlag              of C-Ast.cIntFlag
  | mk-cIntRepr              of C-Ast.cIntRepr
  | mk-cExternalDeclaration  of 'a C-Ast.cExternalDeclaration
  | mk-cTranslationUnit      of 'a C-Ast.cTranslationUnit


 end


structure C11-Ast-Lib : C11-AST-LIB =
struct
local open C-Ast in

datatype data = data-bool of bool    | data-int of int
              | data-string of string | data-absstring of string

type node-content = { tag     : string,
                      sub-tag : string,
                      args    : data list }

fun TT s = { tag = s, sub-tag = , args = [] }
fun TTT s t = { tag = s, sub-tag = t, args = [] }
fun ({ tag,sub-tag,args} #>> S) = { tag = tag, sub-tag = sub-tag, args = args @ S }

fun fold-optiona - None st = st | fold-optiona g (Some a) st = g a st;
fun fold-either g1 - (Left a) st   = g1 a st
   |fold-either - g2 (Right a) st  = g2 a st

fun toString-cStructTag (X:C-Ast.cStructTag) = @{make-string} X

fun toString-cIntFlag  (X:C-Ast.cIntFlag)    = @{make-string} X
```

```
fun toString-cIntRepr  (X:C-Ast.cIntRepr)     = @{make-string} X

fun toString-cUnaryOp  (X:C-Ast.cUnaryOp)     = @{make-string} X

fun toString-cAssignOp (X:C-Ast.cAssignOp)    = @{make-string} X

fun toString-cBinaryOp (X:C-Ast.cBinaryOp)    = @{make-string} X

fun toString-cIntFlag  (X:C-Ast.cIntFlag)     = @{make-string} X

fun toString-cIntRepr  (X:C-Ast.cIntRepr)     = @{make-string} X

fun dark-matter x = XML.content-of (YXML.parse-body x)


fun toString-abr-string S = case  to-String-b-a-s-e S of
                              ST X => dark-matter X
                            | STa S => map (dark-matter o Int.toString) S
                                       |> String.concatWith ,
                                       |> enclose [ ]

fun toString-nodeinfo (NodeInfo0 (positiona, (positiona', i), namea)) =
    let val Position0 (i1,abrS,i2,i3) = positiona;
        val Position0 (i1',abrS',i2',i3') = positiona';
        val Name0 X = namea;
    in  <^Int.toString i1^: ^toString-abr-string abrS^:
        ^ Int.toString i2 ^:  ^ Int.toString i3 ^ :  ^
        Int.toString i1'^: ^toString-abr-string abrS'^:
        ^ Int.toString i2' ^:  ^ Int.toString i3' ^| ^ Int.toString i ^::
        ^ Int.toString X ^>
    end
   |toString-nodeinfo (OnlyPos0 (positiona, (positiona', i))) =
    let val Position0 (i1,abrS,i2,i3) = positiona;
        val Position0 (i1',abrS',i2',i3') = positiona';
    in  <^Int.toString i1^: ^toString-abr-string abrS^:
        ^ Int.toString i2 ^:  ^ Int.toString i3 ^ :  ^
        Int.toString i1'^: ^toString-abr-string abrS'^:
        ^ Int.toString i2' ^:  ^ Int.toString i3' ^| ^ Int.toString i ^>
    end;


fun toString-Chara (Chara(b1,b2,b3,b4,b5,b6,b7,b8)) =
            let val v1 = (b1 ? (K 0)) (128)
                val v2 = (b2 ? (K 0)) (64)
                val v3 = (b3 ? (K 0)) (32)
                val v4 = (b4 ? (K 0)) (16)
                val v5 = (b5 ? (K 0)) (8)
                val v6 = (b6 ? (K 0)) (4)
                val v7 = (b7 ? (K 0)) (2)
```

```
        val v8 = (b8 ? (K 0)) (1)
    in  String.implode[Char.chr(v1+v2+v3+v4+v5+v6+v7+v8)] end

(* could and should be done much more: change this on demand. *)
fun fold-cInteger g' (CInteger0 (i: int, r: cIntRepr, rfl:cIntFlag flags)) st =
                st |> g'(TT CInteger0
                        #>> [data-int i,
                             data-string (@{make-string} r),
                             data-string (@{make-string} rfl)])
fun fold-cChar   g' (CChar0(c : char, b:bool)) st          =
                st |> g' (TTCChar0
                         #>> [data-string (toString-Chara c),data-bool (b)])
  | fold-cChar   g' (CChars0(cs : char list, b:bool)) st   =
                let val cs' = cs |> map toString-Chara
                                 |> String.concat
                in  st |> g' (TTCChars0 #>> [data-string cs',data-bool b]) end
fun fold-cFloat  g' (CFloat0 (bstr: abr-string)) st        =
                st |> g' (TTCChars0#>> [data-string (@{make-string} bstr)])
fun fold-cString g' (CString0 (bstr: abr-string, b: bool)) st =
                st |> g' (TTCString0#>> [data-string (@{make-string} bstr), data-bool b])


fun fold-cConstant g (CIntConst0 (i: cInteger, a))  st = st |> fold-cInteger (fn x=>g x a) i
                                                           |> g (TTCIntConst0) a
  | fold-cConstant g (CCharConst0  (c : cChar, a))  st = st |> fold-cChar (fn x=>g x a) c
                                                           |> g (TTCCharConst0) a
  | fold-cConstant g (CFloatConst0 (f : cFloat, a)) st = st |> fold-cFloat (fn x=>g x a) f
                                                           |> g (TTCFloatConst0) a
  | fold-cConstant g (CStrConst0   (s : cString, a))st = st |> fold-cString (fn x=>g x a) s
                                                           |> g (TTCStrConst0) a

fun fold-ident a g (Ident0(bstr : abr-string, i : int, ni: nodeInfo (* hack !!! *))) st =
                st |> g (TT Ident0
                        #>> [data-string (@{make-string} bstr),
                             data-int i,
                             data-string (@{make-string} ni)
                            ]) a
(* |> fold-cString (fn x=>g x a)  *)
fun fold-cStringLiteral g (CStrLit0(cs:cString, a)) st =  st |> fold-cString (fn x=>g x a) cs
                                                           |> g (TTCStrLit0) a


fun  fold-cTypeSpecifier g (CAtomicType0 (decl : 'a cDeclaration, a)) st =
                              st |> fold-cDeclaration g decl |> g (TTCAtomicType0) a
   | fold-cTypeSpecifier g (CBoolType0 a)     st = st |> g (TTCBoolType0) a
   | fold-cTypeSpecifier g (CCharType0 a)     st = st |> g (TTCCharType0) a
   | fold-cTypeSpecifier g (CComplexType0 a)  st = st |> g (TTCComplexType0) a
   | fold-cTypeSpecifier g (CDoubleType0 a)   st = st |> g (TTCDoubleType0) a
   | fold-cTypeSpecifier g (CEnumType0(e: 'a cEnumeration, a)) st =
```

```
                                        st |> fold-cEnumeration g e
                                           |> g (TTCEnumType0) a
  | fold-cTypeSpecifier g (CFloatType0 a)     st = st |> g (TTCFloatType0) a
  | fold-cTypeSpecifier g (CInt128Type0 a)    st = st |> g (TTCInt128Type0) a
  | fold-cTypeSpecifier g (CIntType0 a)       st = st |> g (TTCIntType0) a
  | fold-cTypeSpecifier g (CLongType0 a)      st = st |> g (TTCLongType0) a
  | fold-cTypeSpecifier g (CSUType0 (su: 'a cStructureUnion, a))  st =
                                        st |> fold-cStructureUnion g su
                                           |> g (TTCSUType0) a
  | fold-cTypeSpecifier g (CShortType0 a)     st = st |> g (TTCShortType0) a
  | fold-cTypeSpecifier g (CSignedType0 a)    st = st |> g (TTCSignedType0) a
  | fold-cTypeSpecifier g (CTypeDef0 (id:ident, a)) st =
                                        st |>  fold-ident a g id
                                           |>  g (TTCTypeDef0) a
  | fold-cTypeSpecifier g (CTypeOfExpr0 (ex: 'a cExpression, a)) st =
                                        st |> fold-cExpression g ex
                                           |> g (TTCTypeOfExpr0) a
  | fold-cTypeSpecifier g (CTypeOfType0 (decl: 'a cDeclaration, a)) st =
                                        st |> fold-cDeclaration g decl
                                           |> g (TTCTypeOfType0) a
  | fold-cTypeSpecifier g (CUnsigType0 a)     st = st |> g (TTCUnsigType0) a
  | fold-cTypeSpecifier g (CVoidType0 a)      st = st |> g (TTCVoidType0)  a


and  fold-cTypeQualifier g (CAtomicQual0 a) st = g (TTCAtomicQual0) a st
  | fold-cTypeQualifier g (CAttrQual0 (CAttr0 (id,eL:'a cExpression list, a))) st =
                                        st |> fold-ident a g id
                                           |> fold(fold-cExpression g) eL
                                           |> g (TTCAttrQual0) a
  | fold-cTypeQualifier g (CConstQual0 a) st    = st |> g (TTCConstQual0) a
  | fold-cTypeQualifier g (CNonnullQual0 a) st  = st |> g (TTCNonnullQual0) a
  | fold-cTypeQualifier g (CNullableQual0 a) st = st |> g (TTCNullableQual0) a
  | fold-cTypeQualifier g (CRestrQual0 a) st    = st |> g (TTCRestrQual0) a
  | fold-cTypeQualifier g (CVolatQual0 a) st    = st |> g (TTCVolatQual0) a

and  fold-cStatement g (CLabel0(id:ident, s:'a cStatement,
                                aL: 'a cAttribute list, a)) st=
                                  st |> fold-ident a g id
                                     |> fold-cStatement g s
                                     |> fold(fold-cAttribute g) aL
                                     |> g (TTCLabel0) a
  | fold-cStatement g (CCase0(ex: 'a cExpression,
                              stmt: 'a cStatement, a)) st    =
                                  st |> fold-cExpression g ex
                                     |> fold-cStatement g stmt
                                     |> g (TTCCase0) a
  | fold-cStatement g (CCases0(ex1: 'a cExpression,
                               ex2: 'a cExpression,
                               stmt:'a cStatement, a)) st   =
```

```
                        st |> fold-cExpression g ex1
                           |> fold-cExpression g ex2
                           |> fold-cStatement g stmt
                           |> g (TTCCases0) a
| fold-cStatement g (CDefault0(stmt:'a cStatement, a)) st =
                        st |> fold-cStatement g stmt
                           |> g (TTCDefault0) a
| fold-cStatement g (CExpr0(ex-opt:'a cExpression optiona, a)) st =
                        st |> fold-optiona (fold-cExpression g) ex-opt
                           |> g (TTCExpr0) a
| fold-cStatement g (CCompound0(idS : ident list,
                         cbiS: 'a cCompoundBlockItem list, a)) st =
                        st |> fold(fold-ident a g) idS
                           |> fold(fold-cCompoundBlockItem g) cbiS
                           |> g (TTCCompound0) a
| fold-cStatement g (CIf0(ex1:'a cExpression,stmt: 'a cStatement,
                  stmt-opt: 'a cStatement optiona, a)) st =
                        st |> fold-cExpression g ex1
                           |> fold-cStatement g stmt
                           |> fold-optiona (fold-cStatement g) stmt-opt
                           |> g (TTCIf0) a
| fold-cStatement g (CSwitch0(ex1:'a cExpression,
                       stmt: 'a cStatement, a)) st =
                        st |> fold-cExpression g ex1
                           |> fold-cStatement g stmt
                           |> g (TTCSwitch0) a
| fold-cStatement g (CWhile0(ex1:'a cExpression,
                        stmt: 'a cStatement, b: bool, a)) st =
                        st |> fold-cExpression g ex1
                           |> fold-cStatement g stmt
                           |> g (TTCWhile0 #>> [data-bool b]) a
| fold-cStatement g (CFor0(ex0:('a cExpression optiona, 'a cDeclaration) either,
                      ex1-opt: 'a cExpression optiona,
                      ex2-opt: 'a cExpression optiona,
                      stmt: 'a cStatement, a)) st =
                        st |> fold-either (fold-optiona (fold-cExpression g))
                                          (fold-cDeclaration g) ex0
                           |> fold-optiona (fold-cExpression g) ex1-opt
                           |> fold-optiona (fold-cExpression g) ex2-opt
                           |> fold-cStatement g stmt
                           |> g (TTCFor0) a
| fold-cStatement g (CGoto0(id: ident, a)) st =
                        st |>  fold-ident a g id
                           |> g (TTCGoto0) a
| fold-cStatement g (CGotoPtr0(ex1:'a cExpression, a)) st =
                        st |> fold-cExpression g ex1 |> g (TTCGotoPtr0) a
| fold-cStatement g (CCont0 a)  st = st |> g (TTCCont0) a
| fold-cStatement g (CBreak0 a) st = st |> g (TTCBreak0) a
| fold-cStatement g (CReturn0 (ex:'a cExpression optiona,a)) st =
```

```
                                st |> fold-optiona (fold-cExpression g) ex |> g (TTCReturn0) a
  | fold-cStatement g (CAsm0(-: 'a cAssemblyStatement, a)) st =
                                (* assembly ignored so far *)
                                st |> g (TTCAsm0) a

and fold-cExpression g (CComma0 (eL:'a cExpression list, a)) st =
                                st |> fold(fold-cExpression g) eL |> g (TTCComma0) a
  | fold-cExpression g (CAssign0(aop:cAssignOp,
                                ex1:'a cExpression,
                                ex2:'a cExpression,a)) st =
                                 st |> fold-cExpression g ex1
                                    |> fold-cExpression g ex2
                                    |> g (TTTCAssign0 (toString-cAssignOp aop)) a
  | fold-cExpression g (CCond0(  ex1:'a cExpression,
                                ex2opt: 'a cExpression optiona, (* bescheuert ! Wieso option ?*)
                                ex3: 'a cExpression,a)) st =
                                 st |> fold-cExpression g ex1
                                    |> fold-optiona (fold-cExpression g) ex2opt
                                    |> fold-cExpression g ex3 |> g (TTCCond0) a
  | fold-cExpression g (CBinary0(bop: cBinaryOp, ex1: 'a cExpression,ex2: 'a cExpression, a))
st =
                                st |> fold-cExpression g ex1
                                   |> fold-cExpression g ex2
                                   |> g (TTTCBinary0(toString-cBinaryOp bop)) a
  | fold-cExpression g (CCast0(decl:'a cDeclaration, ex: 'a cExpression, a)) st =
                                st |> fold-cExpression g ex
                                   |> fold-cDeclaration g decl
                                   |> g (TTCCast0) a
  | fold-cExpression g (CUnary0(unop:cUnaryOp, ex: 'a cExpression, a)) st =
                                st |> fold-cExpression g ex
                                   |> g (TT(CUnary0 ^toString-cUnaryOp unop)) a
  | fold-cExpression g (CSizeofExpr0(ex:'a cExpression, a)) st =
                                st |> fold-cExpression g ex    |> g (TTCSizeofExpr0) a
  | fold-cExpression g (CSizeofType0(decl:'a cDeclaration,a)) st =
                                st |> fold-cDeclaration g decl |> g (TTCSizeofType0) a
  | fold-cExpression g (CAlignofExpr0(ex:'a cExpression, a)) st =
                                st |> fold-cExpression g ex    |> g (TTCAlignofExpr0) a
  | fold-cExpression g (CAlignofType0(decl:'a cDeclaration, a)) st =
                                st |> fold-cDeclaration g decl |> g (TTCAlignofType0) a
  | fold-cExpression g (CComplexReal0(ex:'a cExpression, a)) st =
                                st |> fold-cExpression g ex |> g (TTCComplexReal0) a
  | fold-cExpression g (CComplexImag0(ex:'a cExpression, a)) st =
                                st |> fold-cExpression g ex |> g (TTCComplexImag0) a
  | fold-cExpression g (CIndex0(ex1:'a cExpression, ex2: 'a cExpression, a)) st =
                                st |> fold-cExpression g ex1
                                   |> fold-cExpression g ex2
                                   |> g (TTCIndex0) a
  | fold-cExpression g (CCall0(ex:'a cExpression, argS: 'a cExpression list, a)) st =
                                st |> fold-cExpression g ex
```

```
                                |> fold (fold-cExpression g) argS
                                |> g (TTCCall0) a
  | fold-cExpression g (CMember0(ex:'a cExpression, id:ident, b, a)) st =
                              st |> fold-cExpression g ex
                                |> fold-ident a g id
                                |> g (TTCMember0#>> [data-bool b]) a
  | fold-cExpression g (CVar0(id:ident,a)) st = st |> fold-ident a g id |> g (TTCVar0) a
  | fold-cExpression g (CConst0(cc:'a cConstant)) st  = st |> fold-cConstant g cc
  | fold-cExpression g (CCompoundLit0(decl:'a cDeclaration,
                                  eqn: ('a cPartDesignator list * 'a cInitializer) list, a)) st =
                              st |> fold(fn(S,init) =>
                                      fn st => st |> fold(fold-cPartDesignator g) S
                                                  |> fold-cInitializer g init) eqn
                                |> fold-cDeclaration g decl
                                |> g (TTCCompoundLit0) a
  | fold-cExpression g (CGenericSelection0(ex:'a cExpression,
                                    eqn: ('a cDeclaration optiona*'a cExpression)list,a)) st =
                              st |> fold-cExpression g ex
                                |> fold (fn (d,ex) =>
                                       fn st => st |> fold-optiona (fold-cDeclaration g) d
                                                   |> fold-cExpression g ex) eqn
                                |> g (TTCGenericSelection0) a
  | fold-cExpression g (CStatExpr0(stmt: 'a cStatement,a)) st =
                              st |> fold-cStatement g stmt |> g (TTCStatExpr0) a
  | fold-cExpression g (CLabAddrExpr0(id:ident,a)) st =
                              st |> fold-ident a g id |> g (TTCLabAddrExpr0) a
  | fold-cExpression g (CBuiltinExpr0(X: 'a cBuiltinThing)) st = st |> fold-cBuiltinThing g X

and fold-cDeclaration g (CDecl0(dsS : 'a cDeclarationSpecifier list,
                                mkS: (('a cDeclarator optiona
                                     *'a cInitializer optiona)
                                    * 'a cExpression optiona) list,
                                a)) st =
                                st |> fold(fold-cDeclarationSpecifier g) dsS
                                   |> fold(fn ((d-o, init-o),ex-opt) =>
                                          fn st => st |> fold-optiona(fold-cDeclarator g) d-o
                                                      |> fold-optiona(fold-cInitializer g) init-o
                                                      |> fold-optiona(fold-cExpression g) ex-opt) mkS
                                  |> g (TTCDecl0) a
  | fold-cDeclaration g (CStaticAssert0(ex:'a cExpression, slit: 'a cStringLiteral, a)) st =
                              st |> fold-cExpression g ex
                                 |> fold-cStringLiteral g slit
                                 |> g (TTCStaticAssert0) a

and fold-cBuiltinThing g (CBuiltinVaArg0(ex:'a cExpression,decl: 'a cDeclaration,a)) st =
                              st |> fold-cExpression g ex
                                 |> fold-cDeclaration g decl
                                 |> g (TTCBuiltinVaArg0) a
  | fold-cBuiltinThing g (CBuiltinOffsetOf0(d: 'a cDeclaration, -: 'a cPartDesignator list, a)) st
```

```
=
                                st |> fold-cDeclaration g d
                                   |> g (TTCBuiltinOffsetOf0) a
  | fold-cBuiltinThing g (CBuiltinTypesCompatible0 (d1: 'a cDeclaration, d2: 'a cDeclaration,a))
st=
                                st |> fold-cDeclaration g d1
                                   |> fold-cDeclaration g d2
                                   |> g (TTCBuiltinTypesCompatible0) a

and  fold-cInitializer g (CInitExpr0(ex: 'a cExpression, a)) st =
                                st |> fold-cExpression g ex |> g (TTCInitExpr0) a
  | fold-cInitializer g (CInitList0 (mms: ('a cPartDesignator list * 'a cInitializer) list,a)) st =
                                st |> fold(fn (a,b) =>
                                        fn st => st|> fold(fold-cPartDesignator g) a
                                                   |> fold-cInitializer g b) mms
                                   |> g (TTCInitList0) a

and  fold-cPartDesignator g (CArrDesig0(ex: 'a cExpression, a)) st =
                                st |> fold-cExpression g ex |> g (TTCArrDesig0) a
  | fold-cPartDesignator g (CMemberDesig0(id: ident, a)) st =
                                st |> fold-ident a g id |> g (TTCMemberDesig0) a
  | fold-cPartDesignator g (CRangeDesig0(ex1: 'a cExpression, ex2: 'a cExpression, a)) st =
                                st |> fold-cExpression g ex1
                                   |> fold-cExpression g ex2
                                   |> g (TTCRangeDesig0) a

and fold-cAttribute g (CAttr0(id: ident, exS: 'a cExpression list, a)) st =
                                st |> fold-ident a g id
                                   |> fold(fold-cExpression g) exS
                                   |> g (TTCAttr0) a

and fold-cEnumeration g (CEnum0 (ident-opt: ident optiona,
                                 exS-opt: ((ident * 'a cExpression optiona) list) optiona,
                                 attrS: 'a cAttribute list, a)) st =
                                st |> fold-optiona(fold-ident a g) ident-opt
                                   |> fold-optiona(fold(
                                        fn (id,ex-o) =>
                                        fn st => st |> fold-ident a g id
                                                    |> fold-optiona (fold-cExpression g) ex-o))
                                                 exS-opt
                                   |> fold(fold-cAttribute g) attrS
                                   |> g (TTCEnum0) a



and fold-cArraySize a g (CNoArrSize0 (b: bool)) st =
                                st |> g (TT CNoArrSize0 #>> [data-bool b]) a
  | fold-cArraySize a g (CArrSize0 (b:bool, ex : 'a cExpression)) st =
                                st |> fold-cExpression g ex
```

```
                                |> g (TT CNoArrSize0 #>> [data-bool b]) a


and fold-cDerivedDeclarator g (CPtrDeclr0 (tqS: 'a cTypeQualifier list , a)) st =
                                st |> fold(fold-cTypeQualifier g) tqS
                                   |> g (TTCPtrDeclr0) a
  | fold-cDerivedDeclarator g (CArrDeclr0 (tqS:'a cTypeQualifier list, aS: 'a cArraySize,a)) st =

                                st |> fold(fold-cTypeQualifier g) tqS
                                   |> fold-cArraySize a g aS
                                   |> g (TTCArrDeclr0) a
  | fold-cDerivedDeclarator g (CFunDeclr0 (decl-alt: (ident list,
                                                      ('a cDeclaration list * bool)) either,
                                        aS: 'a cAttribute list, a)) st =
                                st |> fold-either
                                        (fold(fold-ident a g))
                                        (fn (declS,b) =>
                                         fn st => st |> fold (fold-cDeclaration g) declS
                                                     |> g (TTT CFunDeclr0decl-alt-Right
                                                               #>> [data-bool b]) a) decl-alt
                                   |> fold(fold-cAttribute g) aS
                                   |> g (TTCFunDeclr0) a

and fold-cDeclarationSpecifier g (CStorageSpec0(CAuto0 a)) st =
                                st |> g (TTTCStorageSpec0 CAuto0) a
   |fold-cDeclarationSpecifier g (CStorageSpec0(CRegister0 a)) st =
                                st |> g (TTTCStorageSpec0 CRegister0) a
   |fold-cDeclarationSpecifier g (CStorageSpec0(CStatic0 a)) st =
                                st |> g (TTTCStorageSpec0 CStatic0) a
   |fold-cDeclarationSpecifier g (CStorageSpec0(CExtern0 a)) st =
                                st |> g (TTTCStorageSpec0 CExtern0) a
   |fold-cDeclarationSpecifier g (CStorageSpec0(CTypedef0 a)) st =
                                st |> g (TTTCStorageSpec0 CTypedef0) a
   |fold-cDeclarationSpecifier g (CStorageSpec0(CThread0 a)) st =
                                st |> g (TTTCStorageSpec0 CThread0) a

   |fold-cDeclarationSpecifier g (CTypeSpec0(CVoidType0 a)) st =
                                st |> g (TTTCTypeSpec0CVoidType0) a
   |fold-cDeclarationSpecifier g (CTypeSpec0(CCharType0 a)) st =
                                st |> g (TTTCTypeSpec0CCharType0) a
   |fold-cDeclarationSpecifier g (CTypeSpec0(CShortType0 a)) st =
                                st |> g (TTTTCTypeSpec0CShortType0) a
   |fold-cDeclarationSpecifier g (CTypeSpec0(CIntType0 a)) st =
                                st |> g (TTTCTypeSpec0CIntType0) a
   |fold-cDeclarationSpecifier g (CTypeSpec0(CLongType0 a)) st =
                                st |> g (TTTCTypeSpec0CLongType0) a
   |fold-cDeclarationSpecifier g (CTypeSpec0(CFloatType0 a)) st =
                                st |> g (TTTCTypeSpec0CFloatType0) a
   |fold-cDeclarationSpecifier g (CTypeSpec0(CDoubleType0 a)) st =
```

```
                                st |> g (TTTCTypeSpec0CDoubleType0) a
  |fold-cDeclarationSpecifier g (CTypeSpec0(CSignedType0 a)) st =
                                st |> g (TTTCTypeSpec0CSignedType0) a
  |fold-cDeclarationSpecifier g (CTypeSpec0(CUnsigType0 a)) st =
                                st |> g (TTTCTypeSpec0CUnsigType0) a
  |fold-cDeclarationSpecifier g (CTypeSpec0(CBoolType0 a)) st =
                                st |> g (TTTCTypeSpec0CBoolType0) a

  |fold-cDeclarationSpecifier g (CTypeQual0(x: 'a cTypeQualifier)) st =
                                st |> fold-cTypeQualifier g x

  |fold-cDeclarationSpecifier g (CFunSpec0(CInlineQual0 a)) st =
                                st |> g (TTTCFunSpec0CInlineQual0) a
  |fold-cDeclarationSpecifier g (CFunSpec0(CNoreturnQual0 a)) st =
                                st |> g (TTTCFunSpec0CNoreturnQual0) a

  |fold-cDeclarationSpecifier g (CAlignSpec0(CAlignAsType0(decl,a))) st =
                                st |> fold-cDeclaration g decl
                                   |> g (TTTCAlignSpec0CAlignAsType0) a
  |fold-cDeclarationSpecifier g (CAlignSpec0(CAlignAsExpr0(ex,a))) st =
                                st |> fold-cExpression g ex
                                   |> g (TTTCAlignSpec0CAlignAsType0) a

and fold-cDeclarator g (CDeclr0(id-opt: ident optiona,
                                declS: 'a cDerivedDeclarator list,
                                sl-opt: 'a cStringLiteral optiona,
                                attrS: 'a cAttribute list, a)) st =
                                  st |> fold-optiona(fold-ident a g) id-opt
                                     |> fold (fold-cDerivedDeclarator g) declS
                                     |> fold-optiona(fold-cStringLiteral g) sl-opt
                                     |> fold(fold-cAttribute g) attrS
                                     |> g (TTCDeclr0) a

and fold-cFunctionDef g (CFunDef0(dspecS: 'a cDeclarationSpecifier list,
                                  dclr: 'a cDeclarator,
                                  declsS: 'a cDeclaration list,
                                  stmt: 'a cStatement, a)) st =
                                    st |> fold(fold-cDeclarationSpecifier g) dspecS
                                       |> fold-cDeclarator g dclr
                                       |> fold(fold-cDeclaration g) declsS
                                       |> fold-cStatement g stmt
                                       |> g (TTCFunDef0) a

and fold-cCompoundBlockItem g (CBlockStmt0 (stmt: 'a cStatement)) st =
                                  st |> fold-cStatement g stmt
  | fold-cCompoundBlockItem g (CBlockDecl0 (decl : 'a cDeclaration)) st =
                                  st |> fold-cDeclaration g decl
  | fold-cCompoundBlockItem g (CNestedFunDef0(fdef : 'a cFunctionDef)) st =
                                  st |> fold-cFunctionDef g fdef
```

```
and fold-cStructureUnion g (CStruct0(   ct : cStructTag, id-a: ident optiona,
                                        declS-opt : ('a cDeclaration list) optiona,
                                        aS: 'a cAttribute list, a)) st =
                                    st |> fold-optiona (fold-ident a g) id-a
                                       |> fold-optiona (fold(fold-cDeclaration g)) declS-opt
                                       |> fold(fold-cAttribute g) aS
                                       |> g (TTT CStruct0 (toString-cStructTag ct)) a
and fold-cExternalDeclaration g (CDeclExt0(cd : 'a cDeclaration)) st =
                                    st |> fold-cDeclaration g cd
  | fold-cExternalDeclaration g (CFDefExt0(fd : 'a cFunctionDef)) st =
                                    st |>  fold-cFunctionDef g fd
  | fold-cExternalDeclaration - (CAsmExt0( - : 'a cStringLiteral, - : 'a)) - = errorInline assembler
not supprted
and fold-cTranslationUnit g (CTranslUnit0 (ceL : 'a cExternalDeclaration list, a : 'a)) st =
                                    st |> fold(fold-cExternalDeclaration g) ceL
                                       |> g (TTCTranslUnit0) a


(* missing
  datatype 'a cTranslationUnit = CTranslUnit0 of 'a cExternalDeclaration list * 'a
*)


datatype 'a C11-Ast =
          mk-cInteger              of C-Ast.cInteger
        | mk-cStructTag            of C-Ast.cStructTag
        | mk-cUnaryOp              of C-Ast.cUnaryOp
        | mk-cAssignOp             of C-Ast.cAssignOp
        | mk-cBinaryOp             of C-Ast.cBinaryOp
        | mk-cIntFlag              of C-Ast.cIntFlag
        | mk-cIntRepr              of C-Ast.cIntRepr
        | mk-cConstant             of 'a C-Ast.cConstant
        | mk-cStringLiteral        of 'a C-Ast.cStringLiteral
        | mk-cArraySize            of 'a C-Ast.cArraySize
        | mk-cAttribute            of 'a C-Ast.cAttribute
        | mk-cBuiltinThing         of 'a C-Ast.cBuiltinThing
        | mk-cCompoundBlockItem    of 'a C-Ast.cCompoundBlockItem
        | mk-cDeclaration          of 'a C-Ast.cDeclaration
        | mk-cDeclarationSpecifier of 'a C-Ast.cDeclarationSpecifier
        | mk-cDeclarator           of 'a C-Ast.cDeclarator
        | mk-cDerivedDeclarator    of 'a C-Ast.cDerivedDeclarator
        | mk-cEnumeration          of 'a C-Ast.cEnumeration
        | mk-cExpression           of 'a C-Ast.cExpression
        | mk-cInitializer          of 'a C-Ast.cInitializer
        | mk-cPartDesignator       of 'a C-Ast.cPartDesignator
        | mk-cStatement            of 'a C-Ast.cStatement
        | mk-cStructureUnion       of 'a C-Ast.cStructureUnion
        | mk-cTypeQualifier        of 'a C-Ast.cTypeQualifier
```

```
  | mk-cTypeSpecifier        of 'a C-Ast.cTypeSpecifier
  | mk-cExternalDeclaration  of 'a C-Ast.cExternalDeclaration
  | mk-cTranslationUnit      of 'a C-Ast.cTranslationUnit


end

end (*struct *)
›
```

end

3.3 Core Language: Lexing Support, with Filtered Annotations (without Annotation Lexing)

theory *C-Lexer-Language*
imports *Main*
begin

The part implementing the C lexer might resemble to the implementation of the ML one, but the C syntax is much complex: for example, the preprocessing of directives is implemented with several parsing layers. Also, we will see that the way antiquotations are handled in C is slightly different than in ML (especially in the execution part).

Overall, the next ML structures presented here in this theory are all aligned with `~~/src/Pure/ROOT.ML`, and are thus accordingly sorted in the same order (except for `~~/src/Pure/ML/ml_options.ML` which is early included in the boot process).

This theory will stop at `~~/src/Pure/ML/ml_lex.ML`. It is basically situated in the phase 1 of the bootstrap process (of `~~/src/Pure/ROOT.ML`), i.e., the part dealing with how to get some C tokens from a raw string: `Position.T -> string -> token list`.

ML — `~~/src/Pure/General/scan.ML`

```
‹
structure C-Scan =
struct
datatype ('a, 'b) either = Left of 'a | Right of 'b

fun opt x = Scan.optional x [];
end
›
```

ML — `~~/src/Pure/General/symbol.ML`

```
‹
structure C-Symbol =
struct
fun is-ascii-quasi - = true
  | is-ascii-quasi $ = true
  | is-ascii-quasi - = false;
```

```
fun is-identletter s =
  Symbol.is-ascii-letter s orelse is-ascii-quasi s

fun is-ascii-oct s =
  Symbol.is-char s andalso Char.ord #0 <= ord s andalso ord s <= Char.ord #7;

fun is-ascii-digit1 s =
  Symbol.is-char s andalso Char.ord #1 <= ord s andalso ord s <= Char.ord #9;

fun is-ascii-letdig s =
  Symbol.is-ascii-letter s orelse Symbol.is-ascii-digit s orelse is-ascii-quasi s;

fun is-ascii-identifier s =
  size s > 0 andalso forall-string is-ascii-letdig s;

val ascii-blank-no-line =
  [ ([ ], NONE)
  , ([\t, \^K, \f], SOME Space symbol)
  , ([\194\160], SOME Non−standard space symbol) ]

fun is-ascii-blank-no-line s =
  exists (#1 #> exists (curry op = s)) ascii-blank-no-line
end
›
```

ML — `~~/src/Pure/General/position.ML` ‹

```
structure C-Position =
struct
type reports-text = Position.report-text list
end
›
```

ML — `~~/src/Pure/General/symbol_pos.ML`

‹

```
structure C-Basic-Symbol-Pos =   (∗not open by default∗)
struct
open Basic-Symbol-Pos;

fun one f = Scan.one (f o Symbol-Pos.symbol)
fun many f = Scan.many (f o Symbol-Pos.symbol)
fun many1 f = Scan.many1 (f o Symbol-Pos.symbol)
val one′ = Scan.single o one
fun scan-full !!! mem msg scan =
  scan −−| (Scan.ahead (one′ (not o mem)) || !!! msg Scan.fail)
fun this-string s =
  (fold (fn s0 => uncurry (fn acc => one (fn s1 => s0 = s1) >> (fn x => x :: acc)))
      (Symbol.explode s)
   o pair [])
```

```
  >> rev
val one-not-eof = Scan.one (Symbol.not-eof o #1)
fun unless-eof scan = Scan.unless scan one-not-eof >> single
val repeats-one-not-eof = Scan.repeats o unless-eof
val newline =   $$$ \n
            || $$$ \^M @@@ $$$ \n
            || $$$ \^M
val repeats-until-nl = repeats-one-not-eof newline
end

structure C-Symbol-Pos =
struct

(* basic scanners *)

val !!! = Symbol-Pos.!!!

fun !!!! text scan =
  let
    fun get-pos [] =  (end-of-input)
      | get-pos ((-, pos) :: -) = Position.here pos;

    fun err ((-, syms), msg) = fn () =>
      text () ^ get-pos syms ^
       Markup.markup Markup.no-report ( at  ^ Symbol.beginning 10 (map Symbol-Pos.symbol
syms)) ^
      (case msg of NONE =>  | SOME m => \n ^ m ());
  in Scan.!! err scan end;

val $$ = Symbol-Pos.$$

val $$$ = Symbol-Pos.$$$

val ~$$$ = Symbol-Pos.~$$$


(* scan string literals *)

local

val char-code =
  Scan.one (Symbol.is-ascii-digit o Symbol-Pos.symbol) --
  Scan.one (Symbol.is-ascii-digit o Symbol-Pos.symbol) --
  Scan.one (Symbol.is-ascii-digit o Symbol-Pos.symbol) :|--
  (fn (((a, pos), (b, -)), (c, -)) =>
    let val (n, -) = Library.read-int [a, b, c]
    in if n <= 255 then Scan.succeed [(chr n, pos)] else Scan.fail end);

fun scan-str q err-prefix stop =
```

```
  $$$ \\ |-- !!! (fn () => err-prefix ^ bad escape character in string)
    ($$$ q || $$$ \\ || char-code) ||
  Scan.unless stop
               (Scan.one (fn (s, -) => s <> q andalso s <> \\ andalso Symbol.not-eof s)) >>
single;

fun scan-strs q err-prefix err-suffix stop =
  Scan.ahead ($$ q) |--
    !!! (fn () => err-prefix ^ unclosed string literal within  ^ err-suffix)
      ((Symbol-Pos.scan-pos --| $$$ q)
       -- (Scan.repeats (scan-str q err-prefix stop) -- ($$$ q |-- Symbol-Pos.scan-pos)));

in

fun scan-string-qq-multi err-prefix stop = scan-strs \ err-prefix the comment delimiter stop;
fun scan-string-bq-multi err-prefix stop = scan-strs ‘ err-prefix the comment delimiter stop;
fun scan-string-qq-inline err-prefix =
  scan-strs \ err-prefix the same line C-Basic-Symbol-Pos.newline;
fun scan-string-bq-inline err-prefix =
  scan-strs ‘ err-prefix the same line C-Basic-Symbol-Pos.newline;

end;


(* nested text cartouches *)

fun scan-cartouche-depth stop =
  Scan.repeat1 (Scan.depend (fn (depth: int option) =>
    (case depth of
      SOME d =>
        $$ Symbol.open- >> pair (SOME (d + 1)) ||
          (if d > 0 then
            Scan.unless stop
                       (Scan.one (fn (s, -) => s <> Symbol.close andalso Symbol.not-eof s))
            >> pair depth ||
            $$ Symbol.close >> pair (if d = 1 then NONE else SOME (d - 1))
          else Scan.fail)
    | NONE => Scan.fail)));

fun scan-cartouche err-prefix err-suffix stop =
  Scan.ahead ($$ Symbol.open-) |--
    !!! (fn () => err-prefix ^ unclosed text cartouche within  ^ err-suffix)
      (Scan.provide is-none (SOME 0) (scan-cartouche-depth stop));

fun scan-cartouche-multi err-prefix stop =
  scan-cartouche err-prefix the comment delimiter stop;
fun scan-cartouche-inline err-prefix =
  scan-cartouche err-prefix the same line C-Basic-Symbol-Pos.newline;
```

```
(* C−style comments *)

local
val par-l = /
val par-r = /

val scan-body1 = $$$ * −−| Scan.ahead (~$$$ par-r)
val scan-body2 = Scan.one (fn (s, -) => s <> * andalso Symbol.not-eof s) >> single
val scan-cmt =
  Scan.depend (fn (d: int) => $$$ par-l @@@ $$$ * >> pair (d + 1)) ||
  Scan.depend (fn 0 => Scan.fail | d => $$$ * @@@ $$$ par-r >> pair (d − 1)) ||
  Scan.lift scan-body1 ||
  Scan.lift scan-body2;

val scan-cmts = Scan.pass 0 (Scan.repeats scan-cmt);

in

fun scan-comment err-prefix =
  Scan.ahead ($$ par-l −− $$ *) |−−
    !!! (fn () => err-prefix ^ unclosed comment)
      ($$$ par-l @@@ $$$ * @@@ scan-cmts @@@ $$$ * @@@ $$$ par-r)
  || $$$ / @@@ $$$ / @@@ C-Basic-Symbol-Pos.repeats-until-nl;

fun scan-comment-no-nest err-prefix =
  Scan.ahead ($$ par-l −− $$ *) |−−
    !!! (fn () => err-prefix ^ unclosed comment)
      ($$$ par-l @@@ $$$ * @@@ Scan.repeats (scan-body1 || scan-body2) @@@ $$$ * @@@ $$$
par-r)
  || $$$ / @@@ $$$ / @@@ C-Basic-Symbol-Pos.repeats-until-nl;

val recover-comment =
  $$$ par-l @@@ $$$ * @@@ Scan.repeats (scan-body1 || scan-body2);

end
end
›
```

ML — `~~/src/Pure/General/antiquote.ML`

```
‹
structure C-Antiquote =
struct

(* datatype antiquote *)

type antiq = { explicit: bool
             , start: Position.T
```

```
        , stop: Position.T option
        , range: Position.range
        , body: Symbol-Pos.T list
        , body-begin: Symbol-Pos.T list
        , body-end: Symbol-Pos.T list }

(* scan *)

open C-Basic-Symbol-Pos;

local

val err-prefix = Antiquotation lexical error: ;

val par-l = /
val par-r = /

val scan-body1 = $$$ * --| Scan.ahead (~$$$ par-r)
val scan-body1' = $$$ * @@@ $$$ par-r
val scan-body2 = Scan.one (fn (s, -) => s <> * andalso Symbol.not-eof s) >> single

val scan-antiq-body-multi =
  Scan.trace (C-Symbol-Pos.scan-string-qq-multi err-prefix scan-body1' ||
        C-Symbol-Pos.scan-string-bq-multi err-prefix scan-body1') >> #2 ||
  C-Symbol-Pos.scan-cartouche-multi err-prefix scan-body1' ||
  scan-body1 ||
  scan-body2;

val scan-antiq-body-multi-recover =
  scan-body1 ||
  scan-body2;

val scan-antiq-body-inline =
  Scan.trace (C-Symbol-Pos.scan-string-qq-inline err-prefix ||
        C-Symbol-Pos.scan-string-bq-inline err-prefix) >> #2 ||
  C-Symbol-Pos.scan-cartouche-inline err-prefix ||
  unless-eof newline;

val scan-antiq-body-inline-recover =
  unless-eof newline;

fun control-name sym = (case Symbol.decode sym of Symbol.Control name => name);

fun scan-antiq-multi scan =
  Symbol-Pos.scan-pos
  -- (Scan.trace ($$ par-l |-- $$ * |-- scan)
     -- Symbol-Pos.scan-pos
     -- Symbol-Pos.!!! (fn () => err-prefix ^ missing closing antiquotation)
                (Scan.repeats scan-antiq-body-multi
```

```
                    -- Symbol-Pos.scan-pos
                    -- ($$$ * @@@ $$$ par-r)
                    -- Symbol-Pos.scan-pos))

fun scan-antiq-multi-recover scan =
  Symbol-Pos.scan-pos
  -- ($$ par-l |-- $$ * |-- scan -- Symbol-Pos.scan-pos --
     (Scan.repeats scan-antiq-body-multi-recover
      -- Symbol-Pos.scan-pos -- ($$ * |-- $$ par-r |-- Symbol-Pos.scan-pos)))

fun scan-antiq-inline scan =
  (Symbol-Pos.scan-pos -- Scan.trace ($$ / |-- $$ / |-- scan)
  -- Symbol-Pos.scan-pos
  -- Scan.repeats scan-antiq-body-inline -- Symbol-Pos.scan-pos)

fun scan-antiq-inline-recover scan =
  (Symbol-Pos.scan-pos --| $$ / --| $$ / -- scan
  -- Symbol-Pos.scan-pos
  -- Scan.repeats scan-antiq-body-inline-recover -- Symbol-Pos.scan-pos)

in

val scan-control =
  Scan.option (Scan.one (Symbol.is-control o Symbol-Pos.symbol)) --
  Symbol-Pos.scan-cartouche err-prefix >>
    (fn (opt-control, body) =>
      let
        val (name, range) =
          (case opt-control of
              SOME (sym, pos) => ((control-name sym, pos), Symbol-Pos.range ((sym, pos) ::
body))
            | NONE => ((cartouche, #2 (hd body)), Symbol-Pos.range body));
      in {name = name, range = range, body = body} end) ||
  Scan.one (Symbol.is-control o Symbol-Pos.symbol) >>
    (fn (sym, pos) =>
      {name = (control-name sym, pos), range = Symbol-Pos.range [(sym, pos)], body = []});

val scan-antiq =
  scan-antiq-multi ($$$ @ >> K true || scan-body1 >> K false)
  >> (fn (pos1, (((explicit, body-begin), pos2), (((body, pos3), body-end), pos4))) =>
      {explicit = explicit,
       start = Position.range-position (pos1, pos2),
       stop = SOME (Position.range-position (pos3, pos4)),
       range = Position.range (pos1, pos4),
       body = body,
       body-begin = body-begin,
       body-end = body-end}) ||
  scan-antiq-inline ($$$ @ >> K true || $$$ * >> K false)
  >> (fn ((((pos1, (explicit, body-begin)), pos2), body), pos3) =>
```

```
      {explicit = explicit,
       start = Position.range-position (pos1, pos2),
       stop = NONE,
       range = Position.range (pos1, pos3),
       body = body,
       body-begin = body-begin,
       body-end = []})

val scan-antiq-recover =
  scan-antiq-multi-recover ($$$ @ >> K true || scan-body1 >> K false)
    >> (fn (-, ((explicit, -), -)) => explicit)
  ||
  scan-antiq-inline-recover ($$$ @ >> K true || $$$ * >> K false)
   >> (fn ((((-, explicit), -), -), -) => explicit)

end;

end;
›
```

ML — `~~/src/Pure/ML/ml_options.ML`

```
‹
structure C-Options =
struct

(* source trace *)

val lexer-trace = Attrib.setup-config-bool @{binding C-lexer-trace} (K false);
val parser-trace = Attrib.setup-config-bool @{binding C-parser-trace} (K false);
val ML-verbose = Attrib.setup-config-bool @{binding C-ML-verbose} (K true);
val starting-env = Attrib.setup-config-string @{binding C_env0} (K empty);
val starting-rule = Attrib.setup-config-string @{binding C_rule0} (K translation-unit);

end
›
```

ML — `~~/src/Pure/ML/ml_lex.ML`

```
‹
structure C-Lex =
struct

open C-Scan;
open C-Basic-Symbol-Pos;


(** keywords **)
```

val keywords =
[(,
),
[,
],
−>,
.,
!,
~,
++,
−−,
+,
−,
*,
/,
%,
&,
<<,
>>,
<,
<=,
>,
>=,
==,
!=,
^,
|,
&&,
||,
?,
:,
=,
+=,
−=,
*=,
/=,
%=,
&=,
^=,
|=,
<<=,
>>=,
,,
;,
{,
},
...,
(**)
-Alignas,

-Alignof,
--alignof,
alignof,
--alignof--,
--asm,
asm,
--asm--,
-Atomic,
--attribute,
--attribute--,
auto,
-Bool,
break,
--builtin-offsetof,
--builtin-types-compatible-p,
--builtin-va-arg,
case,
char,
-Complex,
--complex--,
--const,
const,
--const--,
continue,
default,
do,
double,
else,
enum,
--extension--,
extern,
float,
for,
-Generic,
goto,
if,
--imag,
--imag--,
--inline,
inline,
--inline--,
int,
--int128,
--label--,
long,
-Nonnull,
--nonnull,
-Noreturn,
-Nullable,

--nullable,
--real,
--real--,
register,
--restrict,
restrict,
--restrict--,
return,
short,
--signed,
signed,
--signed--,
sizeof,
static,
-Static-assert,
struct,
switch,
--thread,
-Thread-local,
typedef,
--typeof,
typeof,
--typeof--,
union,
unsigned,
void,
--volatile,
volatile,
--volatile--,
while];

val keywords2 =
[*--asm*,
asm,
--asm--,
extern];

val keywords3 =
[*-Bool*,
char,
double,
float,
int,
--int128,
long,
short,
--signed,
signed,
--signed--,

```
  unsigned,
  void];

val lexicon = Scan.make-lexicon (map raw-explode keywords);



(** tokens **)

(* datatype token *)

datatype token-kind-comment =
   Comment-formal of C-Antiquote.antiq
 | Comment-suspicious of (bool * string * ((Position.T * Markup.T) * string) list) option

datatype token-kind-encoding =
   Encoding-L
 | Encoding-default
 | Encoding-file of string (* error message *) option

type token-kind-string =
  token-kind-encoding
  * (Symbol-Pos.T, Position.range * int — exceeding Char.maxOrd) either list

datatype token-kind-int-repr = Repr-decimal
                             | Repr-hexadecimal
                             | Repr-octal

datatype token-kind-int-flag = Flag-unsigned
                             | Flag-long
                             | Flag-long-long
                             | Flag-imag

datatype token-kind =
  Keyword | Ident of (Symbol-Pos.T list, Symbol-Pos.T) either list | Type-ident | GnuC | ClangC
|
  (**)
  Char of token-kind-string |
  Integer of int * token-kind-int-repr * token-kind-int-flag list |
  Float of Symbol-Pos.T list |
  String of token-kind-string |
  File of token-kind-string |
  (**)
  Space of (string * Symbol-Pos.T) option list | Comment of token-kind-comment | Sharp of int
|
  (**)
  Unknown | Error of string * token-group | Directive of token-kind-directive | EOF

and token-kind-directive = Inline of token-group (* a not yet analyzed directive *)
```

```
                    | Include of token-group
                    | Define of token-group (* define *)
                            * token-group (* name *)
                            * token-group option (* functional arguments *)
                            * token-group (* rewrite body *)
                    | Undef of token-group (* name *)
                    | Cpp of token-group
                    | Conditional of token-group (* if *)
                                * token-group list (* elif *)
                                * token-group option (* else *)
                                * token-group (* endif *)

and token-group = Group1 of token list (* spaces and comments filtered from the directive *)
                        * token list (* directive: raw data *)
                | Group2 of token list (* spaces and comments filtered from the directive *)
                        * token list (* directive: function *)
                        * token list (* directive: arguments (same line) *)
                | Group3 of (  Position.range (* full directive (with blanks) *)
                             * token list (* spaces and comments filtered from the directive *)
                             * token list (* directive: function *)
                             * token list (* directive: arguments (same line) *))
                        * (Position.range * token list) (* C code or directive:
                                                           arguments (following lines) *)

and token = Token of Position.range * (token-kind * string);


(* position *)

fun set-range range (Token (-, x)) = Token (range, x);
fun range-of (Token (range, -)) = range;

val pos-of = #1 o range-of;
val end-pos-of = #2 o range-of;


(* stopper *)

fun mk-eof pos = Token ((pos, Position.none), (EOF, ));
val eof = mk-eof Position.none;

fun is-eof (Token (-, (EOF, -))) = true
  | is-eof - = false;

val stopper =
  Scan.stopper (fn [] => eof | toks => mk-eof (end-pos-of (List.last toks))) is-eof;

val one-not-eof = Scan.one (not o is-eof)
```

```
(* token content *)

fun kind-of (Token (-, (k, -))) = k;

val group-list-of = fn
   Inline g => [g]
 | Include g => [g]
 | Define (g1, g2, o-g3, g4) => flat [[g1], [g2], the-list o-g3, [g4]]
 | Undef g => [g]
 | Cpp g => [g]
 | Conditional (g1, gs2, o-g3, g4) => flat [[g1], gs2, the-list o-g3, [g4]]

fun content-of (Token (-, (-, x))) = x;
fun token-leq (tok, tok') = content-of tok <= content-of tok';

val directive-cmds = fn
   Inline (Group1 (-, - :: tok2 :: -)) => [tok2]
 | Include (Group2 (-, [-, tok2], -)) => [tok2]
 | Define (Group1 (-, [-, tok2]), -, -, -) => [tok2]
 | Undef (Group2 (-, [-, tok2], -)) => [tok2]
 | Conditional (c1, cs2, c3, c4) =>
     maps (fn Group3 ((-, -, [-, tok2], -), -) => [tok2]
            | - => error Only expecting Group3)
          (flat [[c1], cs2, the-list c3, [c4]])
 | - => []

fun is-keyword (Token (-, (Keyword, -))) = true
  | is-keyword - = false;

fun is-ident (Token (-, (Ident -, -))) = true
  | is-ident - = false;

fun is-integer (Token (-, (Integer -, -))) = true
  | is-integer - = false;

fun is-delimiter (Token (-, (Keyword, x))) = not (C-Symbol.is-ascii-identifier x)
  | is-delimiter - = false;

(* range *)

val range-list-of0 =
 fn [] => Position.no-range
  | toks as tok1 :: - => Position.range (pos-of tok1, end-pos-of (List.last toks))
    (* WARNING the use of:
      — fn content_of => fn pos_of => fn tok2 =>
            List.last (Symbol_Pos.explode (content_of tok2, pos_of tok2)) |->
Position.symbol
      would not return an accurate position if for example several
      backslash newlines are present in the symbol *)
```

```
fun range-list-of toks = (range-list-of0 toks, toks)
fun range-list-of' toks1 toks2 = (range-list-of0 toks1, toks2)

local
fun cmp-pos x2 x1 = case Position.distance-of (pos-of x2, pos-of x1) of SOME dist => dist <
0
                                                          | NONE => error cmp-pos

fun merge-pos xs = case xs of (xs1, []) => xs1
                            | ([], xs2) => xs2
                            | (x1 :: xs1, x2 :: xs2) =>
                                let val (x, xs) = if cmp-pos x2 x1 then (x1, (xs1, x2 :: xs2))
                                                                   else (x2, (x1 :: xs1, xs2))
                                in x :: merge-pos xs end
in
fun merge-blank toks-bl xs1 xs2 =
  let val cmp-tok2 = cmp-pos (List.last xs1)
  in ( range-list-of (merge-pos (xs1, filter cmp-tok2 toks-bl))
     , range-list-of (merge-pos (xs2, filter-out cmp-tok2 toks-bl)))
  end
end

val token-list-of =
  let fun merge-blank' toks-bl xs1 xs2 =
    let val ((-, l1), (-, l2)) = merge-blank toks-bl xs1 xs2
    in [l1, l2] end
  in group-list-of
    #> maps (fn
        Group1 (toks-bl, []) => [toks-bl]
      | Group1 (toks-bl, xs1) => merge-blank' toks-bl xs1 []
      | Group2 (toks-bl, xs1, xs2) => merge-blank' toks-bl xs1 xs2
      | Group3 ((-, toks-bl, xs1, xs2), (-, xs3)) => flat [merge-blank' toks-bl xs1 xs2, [xs3]])
    #> flat
  end

local

fun warn-utf8 s pos =
  Output.information (UTF-8 ^@{make-string} s ^ Position.here pos)

val warn-ident = app (fn Right (s, pos) => warn-utf8 s pos | - => ())

val warn-string =
  app (fn Left (s, pos) =>
          if Symbol.is-utf8 s then
            warn-utf8 s pos
          else if Symbol.is-printable s then
            ()
```

```
            else
              let val ord-s = ord s
              in
                Output.information (Not printable symbol
                                    ^ (if chr ord-s = s then @{make-string} (ord-s, s)
                                                        else @{make-string} s)
                                    ^ Position.here pos)
              end
          | Right ((pos1, -), n) =>
              Output.information
                (Out of the supported range (character number  ^ Int.toString n ^ )
                 ^ Position.here pos1))

val warn-space =
  app (fn SOME (msg, (s, pos)) =>
              Output.information (msg ^   ^ @{make-string} s ^ Position.here pos)
          | - => ())

fun warn-unknown pos = Output.information (Unknown symbol ^ Position.here pos)

val warn-directive =
  app (fn Token (-, (Error (msg, -), -)) => warning msg
          | Token ((pos, -), (Unknown, -)) => warn-unknown pos
          | - => ())

in
val warn = fn
    Token (-, (Ident l, -)) => warn-ident l
  | Token (-, (Char (-, l), -)) => warn-string l
  | Token (-, (String (-, l), -)) => warn-string l
  | Token (-, (File (-, l), -)) => warn-string l
  | Token (-, (Space l, -)) => warn-space l
  | Token ((pos, -), (Unknown, -)) => warn-unknown pos
  | Token (-, (Comment (Comment-suspicious (SOME (explicit, msg, -))), -)) =>
      (if explicit then warning else tracing) msg
  | Token (-, (Directive (kind as Conditional -), -)) => warn-directive (token-list-of kind)
  | Token (-, (Directive (Define (-, -, -, Group1 (-, toks4))), -)) => warn-directive toks4
  | Token (-, (Directive (Include (Group2 (-, -, toks))), -)) =>
    (case toks of
       [Token (-, (String -, -))] => ()
     | [Token (-, (File -, -))] => ()
     | - => Output.information
              (Expecting at least and at most one file
               ^ Position.here
                   (Position.range-position (pos-of (hd toks), end-pos-of (List.last toks)))))
  | - => ();
end

fun check-error tok =
```

```
  case kind-of tok of
    Error (msg, -) => [msg]
  | - => [];

(* markup *)

local

val token-kind-markup0 =
 fn Char - => (Markup.ML-char, )
  | Integer - => (Markup.ML-numeral, )
  | Float - => (Markup.ML-numeral, )
  | ClangC => (Markup.ML-numeral, )
  | String - => (Markup.ML-string, )
  | File - => (Markup.ML-string, )
  | Sharp - => (Markup.antiquote, )
  | Unknown => (Markup.intensify, )
  | Error (msg, -) => (Markup.bad (), msg)
  | - => (Markup.empty, );

fun token-report' escape-directive (tok as Token ((pos, -), (kind, x))) =
  if escape-directive andalso (is-keyword tok orelse is-ident tok) then
    [((pos, Markup.antiquote), )]
  else if is-keyword tok then
    let
      val (markup, txt) = if is-delimiter tok then (Markup.ML-delimiter, )
                  else if member (op =) keywords2 x then (Markup.ML-keyword2 |>
Markup.keyword-properties, )
                  else if member (op =) keywords3 x then (Markup.ML-keyword3 |>
Markup.keyword-properties, )
       else (Markup.ML-keyword1 |> Markup.keyword-properties, );
    in [((pos, markup), txt)] end
  else
    case kind of
     Directive (tokens as Inline -) =>
       ((pos, Markup.antiquoted), ) :: maps token-report0 (token-list-of tokens)
  | Directive (Include (Group2 (toks-bl, tok1 :: -, toks2))) =>
       ((pos, Markup.antiquoted), )
       :: flat [ maps token-report1 [tok1]
               , maps token-report0 toks2
               , maps token-report0 toks-bl ]
  | Directive
       (Define
        (Group1 (toks-bl1, tok1 :: -), Group1 (toks-bl2, -), toks3, Group1 (toks-bl4, toks4))) =>
       let val (toks-bl3, toks3) = case toks3 of SOME (Group1 x) => x | - => ([], [])
       in ((pos, Markup.antiquoted), )
         :: ((range-list-of0 toks4 |> #1, Markup.intensify), )
         :: flat [ maps token-report1 [tok1]
                 , maps token-report0 toks3
```

```
                , maps token-report0 toks4
                , maps token-report0 toks-bl1
                , maps token-report0 toks-bl2
                , map (fn tok => ((pos-of tok, Markup.antiquote), )) toks-bl3
                , maps token-report0 toks-bl4 ] end
  | Directive (Undef (Group2 (toks-bl, tok1 :: -, -))) =>
      ((pos, Markup.antiquoted), )
      :: flat [ maps token-report1 [tok1]
              , maps token-report0 toks-bl ]
  | Directive (Cpp (Group2 (toks-bl, toks1, toks2))) =>
      ((pos, Markup.antiquoted), )
      :: flat [ maps token-report1 toks1
              , maps token-report0 toks2
              , maps token-report0 toks-bl ]
  | Directive (Conditional (c1, cs2, c3, c4)) =>
      maps (fn Group3 (((pos, -), toks-bl, tok1 :: -, toks2), ((pos3, -), toks3)) =>
              ((pos, Markup.antiquoted), )
              :: ((pos3, Markup.intensify), )
              :: flat [ maps token-report1 [tok1]
                      , maps token-report0 toks2
                      , maps token-report0 toks3
                      , maps token-report0 toks-bl ]
            | - => error Only expecting Group3)
          (flat [[c1], cs2, the-list c3, [c4]])
  | Error (msg, Group2 (toks-bl, toks1, toks2)) =>
      ((range-list-of0 toks1 |> #1, Markup.bad ()), msg)
      :: ((pos, Markup.antiquoted), )
      :: flat [ maps token-report1 toks1
              , maps token-report0 toks2
              , maps token-report0 toks-bl ]
  | Error (msg, Group3 ((-, toks-bl, toks1, toks2), -)) =>
      ((range-list-of0 toks1 |> #1, Markup.bad ()), msg)
      :: ((pos, Markup.antiquoted), )
      :: flat [ maps token-report1 toks1
              , maps token-report0 toks2
              , maps token-report0 toks-bl ]
  | Comment (Comment-suspicious c) => ((pos, Markup.ML-comment), )
                                      :: (case c of NONE => [] | SOME (-, -, l) => l)
  | x => [let val (markup, txt) = token-kind-markup0 x in ((pos, markup), txt) end]

and token-report0 tok = token-report' false tok
and token-report1 tok = token-report' true tok

in
val token-report = token-report0
end;
```

```
(** scanners **)

val err-prefix = C lexical error: ;

fun !!! msg = Symbol-Pos.!!! (fn () => err-prefix ^ msg);

fun !!!! msg = C-Symbol-Pos.!!!! (fn () => err-prefix ^ msg);

val many1-blanks-no-line =
  Scan.repeat1
    (one C-Symbol.is-ascii-blank-no-line
     >> (fn (s, pos) =>
          List.find (#1 #> exists (curry op = s)) C-Symbol.ascii-blank-no-line
          |> the
          |> #2
          |> Option.map (rpair (s, pos))))

(* identifiers *)

local
fun left x = [Left x]
fun right x = [Right x]
in
val scan-ident-sym =
  let val hex = one' Symbol.is-ascii-hex
  in   one' C-Symbol.is-identletter >> left
    || $$$ \\ @@@ $$$ u @@@ hex @@@ hex @@@ hex @@@ hex >> left
    || $$$ \\ @@@ $$$ U @@@ hex @@@ hex @@@ hex @@@ hex @@@ hex @@@ hex @@@
hex @@@ hex >> left
    || one' Symbol.is-symbolic >> left
    || one' Symbol.is-control >> left
    || one Symbol.is-utf8 >> right
  end

val scan-ident =
      scan-ident-sym
  @@@ Scan.repeats (scan-ident-sym || one' Symbol.is-ascii-digit >> left);

val scan-ident' = scan-ident >> maps (fn Left s => s | Right c => [c]);
end

val keywords-ident =
  map-filter
    (fn s =>
        Source.of-list (Symbol-Pos.explode (s, Position.none))
      |> Source.source
           Symbol-Pos.stopper
           (Scan.bulk (scan-ident >> SOME || Scan.one (not o Symbol-Pos.is-eof) >> K NONE))
      |> Source.exhaust
```

```
    |> (fn [SOME -] => SOME s | - => NONE))
  keywords

(* numerals *)

fun read-bin s = #1 (read-radix-int 2 s)
fun read-oct s = #1 (read-radix-int 8 s)
fun read-dec s = #1 (read-int s)
val read-hex =
  let fun conv-ascii c1 c0 = String.str (Char.chr (Char.ord #9 + Char.ord c1 - Char.ord c0
+ 1))
  in map (fn s => let val c = String.sub (s, 0) in
                  if c >= #A andalso c <= #F then
                    conv-ascii c #A
                  else if c >= #a andalso c <= #f then
                    conv-ascii c #a
                  else s
                  end)
  #> read-radix-int 16
  #> #1
  end

local
val many-digit = many Symbol.is-ascii-digit
val many1-digit = many1 Symbol.is-ascii-digit
val many-hex = many Symbol.is-ascii-hex
val many1-hex = many1 Symbol.is-ascii-hex

val scan-suffix-ll = ($$$ l @@@ $$$ l || $$$ L @@@ $$$ L) >> K [Flag-long-long]
fun scan-suffix-gnu flag = ($$$ i || $$$ j) >> K [flag]
val scan-suffix-int =
  let val u = ($$$ u || $$$ U) >> K [Flag-unsigned]
      val l = ($$$ l || $$$ L) >> K [Flag-long] in
      u @@@ scan-suffix-ll
   || scan-suffix-ll @@@ opt u
   || u @@@ opt l
   || l @@@ opt u
  end

val scan-suffix-gnu-int0 = scan-suffix-gnu Flag-imag
val scan-suffix-gnu-int = scan-full !!!
                                (member (op =) (raw-explode uUlLij))
                                Invalid integer constant suffix
                                (    scan-suffix-int @@@ opt scan-suffix-gnu-int0
                                 || scan-suffix-gnu-int0 @@@ opt scan-suffix-int)

fun scan-intgnu x =
  x -- opt scan-suffix-gnu-int
  >> (fn ((s1', read, repr), l) => (read (map (Symbol-Pos.content o single) s1'), repr, l))
```

```
val scan-intoct = scan-intgnu ($$ 0 |--
                              scan-full
                                !!!
                                Symbol.is-ascii-digit
                                Invalid digit in octal constant
                                (Scan.max
                                  (fn ((xs2, -, -), (xs1, -, -)) => length xs2 < length xs1)
                                  (many C-Symbol.is-ascii-oct
                                     >> (fn xs => (xs, read-oct, Repr-octal)))
                                  (many (fn x => x = 0)
                                     >> (fn xs => (xs, read-dec, Repr-decimal)))))
val scan-intdec = scan-intgnu (one C-Symbol.is-ascii-digit1 -- many Symbol.is-ascii-digit
                              >> (fn (x, xs) => (x :: xs, read-dec, Repr-decimal)))
val scan-inthex = scan-intgnu (($$ 0 -- ($$ x || $$ X)) |-- many1-hex
                              >> (fn xs2 => (xs2, read-hex, Repr-hexadecimal)))

(**)

fun scan-signpart a A = ($$$ a || $$$ A) @@@ opt ($$$ + || $$$ -) @@@ many1-digit
val scan-exppart = scan-signpart e E

val scan-suffix-float = $$$ f || $$$ F || $$$ l || $$$ L
val scan-suffix-gnu-float0 = Scan.trace (scan-suffix-gnu ()) >> #2
val scan-suffix-gnu-float = scan-full !!!
                                    (member (op =) (raw-explode fFlLij))
                                    Invalid float constant suffix
                                    (   scan-suffix-float @@@ opt scan-suffix-gnu-float0
                                     || scan-suffix-gnu-float0 @@@ opt scan-suffix-float)

val scan-hex-pref = $$$ 0 @@@ $$$ x

val scan-hexmant = many-hex @@@ $$$ . @@@ many1-hex
                || many1-hex @@@ $$$ .
val scan-floatdec =
    (       (   many-digit @@@ $$$ . @@@ many1-digit
             || many1-digit @@@ $$$ .)
        @@@ opt scan-exppart
     || many1-digit @@@ scan-exppart)
  @@@ opt scan-suffix-gnu-float

val scan-floathex = scan-hex-pref @@@ (scan-hexmant || many1-hex)
                  @@@ scan-signpart p P @@@ opt scan-suffix-gnu-float
val scan-floatfail = scan-hex-pref @@@ scan-hexmant
in
val scan-int = scan-inthex
            || scan-intoct
            || scan-intdec
```

```
val recover-int =
    many1 (fn s => Symbol.is-ascii-hex s orelse member (op =) (raw-explode xXuUlLij) s)

val scan-float = scan-floatdec
            || scan-floathex
            || scan-floatfail @@@ !!! Hexadecimal floating constant requires an exponent
                                      Scan.fail

val scan-clangversion = many1-digit @@@ $$$ . @@@ many1-digit @@@ $$$ . @@@ many1-digit

end;


(* chars and strings *)

val scan-blanks1 = many1 Symbol.is-ascii-blank

local
val escape-char = [ (n, #\n)
                  , (t, #\t)
                  , (v, #\v)
                  , (b, #\b)
                  , (r, #\r)
                  , (f, #\f)
                  , (a, #\a)
                  , (e, #\^[)
                  , (E, #\^[)
                  , (\\, #\\)
                  , (?, #?)
                  , (', #')
                  , (\, #\) ]

val - = — printing a ML function translating code point from int -> string
 fn - =>
  app (fn (x0, x) => writeln ( |
                              ^string-of-int (Char.ord x)
                              ^ => \\\\\
                              ^(if exists (fn x1 => x0 = x1) [\, \\] then \\ ^x0 else x0)
                              ^\))
      escape-char

fun scan-escape s0 =
  let val oct = one' C-Symbol.is-ascii-oct
      val hex = one' Symbol.is-ascii-hex
      val sym-pos = Symbol-Pos.range #> Position.range-position
      fun chr' f l =
        let val x = f (map Symbol-Pos.content l)
            val l = flat l
        in [if x <= Char.maxOrd then Left (chr x, sym-pos l) else Right (Symbol-Pos.range l, x)]
```

```
end
      val read-oct' = chr' read-oct
      val read-hex' = chr' read-hex
  in one' (member (op =) (raw-explode (s0 ^ String.concat (map #1 escape-char))))
     >> (fn i =>
         [Left (( case AList.lookup (op =) escape-char (Symbol-Pos.content i) of
                    NONE => s0
                  | SOME c => String.str c)
                , sym-pos i)])
  || oct -- oct -- oct >> (fn ((o1, o2), o3) => read-oct' [o1, o2, o3])
  || oct -- oct >> (fn (o1, o2) => read-oct' [o1, o2])
  || oct >> (read-oct' o single)
  || $$ x |-- many1 Symbol.is-ascii-hex
     >> (read-hex' o map single)
  || $$ u |-- hex -- hex -- hex -- hex
     >> (fn (((x1, x2), x3), x4) => read-hex' [x1, x2, x3, x4])
  || $$ U |-- hex -- hex -- hex -- hex -- hex -- hex -- hex -- hex
     >> (fn (((((((x1, x2), x3), x4), x5), x6), x7), x8) =>
         read-hex' [x1, x2, x3, x4, x5, x6, x7, x8])
  end

fun scan-str s0 =
     Scan.unless newline
                 (Scan.one (fn (s, -) => Symbol.not-eof s andalso s <> s0 andalso s <> \\))
     >> (single o Left)
  || Scan.ahead newline |-- !!! bad newline Scan.fail
  || $$ \\ |-- !!! bad escape character (scan-escape s0);

fun scan-string0 s0 msg repeats =
  Scan.optional ($$ L >> K Encoding-L) Encoding-default --
    (Scan.ahead ($$ s0) |--
      !!! (unclosed ^ msg ^ literal)
        ($$ s0 |-- repeats (scan-str s0) --| $$ s0))

fun recover-string0 s0 repeats =
  opt ($$$ L) @@@ $$$ s0 @@@ repeats (Scan.permissive (Scan.trace (scan-str s0) >> #2));
in

val scan-char = scan-string0 ' char Scan.repeats1
val scan-string = scan-string0 \ string Scan.repeats
fun scan-string' src =
  case
    Source.source
      Symbol-Pos.stopper
      (Scan.recover (Scan.bulk (!!! bad input scan-string >> K NONE))
                    (fn msg => C-Basic-Symbol-Pos.one-not-eof >> K [SOME msg]))
      (Source.of-list src)
    |> Source.exhaust
  of
```

```
      [NONE] => NONE
    | [] => SOME Empty input
    | l => case map-filter I l of msg :: - => SOME msg
                               | - => SOME More than one string
val scan-file =
  let fun scan !!! s-l s-r =
    Scan.ahead ($$ s-l) |--
          !!!
          ($$ s-l
           |-- Scan.repeats
                 (Scan.unless newline
                              (Scan.one (fn (s, -) => Symbol.not-eof s andalso s <> s-r)
                               >> (single o Left)))
           --| $$ s-r)
  in
     Scan.trace (scan (!!! (unclosed file literal)) \ \)
       >> (fn (s, src) => String (Encoding-file (scan-string' src), s))
  || scan I — Due to conflicting symbols, raising Symbol_Pos.!!! here will not let a potential
legal "<" symbol be tried and parsed as a keyword.
          < > >> (fn s => File (Encoding-default, s))
  end

val recover-char = recover-string0 ' Scan.repeats1
val recover-string = recover-string0 \ Scan.repeats

end;

(* scan tokens *)

val check = fold (tap warn #> fold cons o check-error)

local

fun token k ss = Token (Symbol-Pos.range ss, (k, Symbol-Pos.content ss));
fun scan-token f1 f2 = Scan.trace f1 >> (fn (v, s) => token (f2 v) s)

val comments =
     Scan.recover
       (scan-token C-Antiquote.scan-antiq (Comment o Comment-formal))
       (fn msg => Scan.ahead C-Antiquote.scan-antiq-recover
                  -- C-Symbol-Pos.scan-comment-no-nest err-prefix
                  >> (fn (explicit, res) =>
                       token (Comment (Comment-suspicious (SOME (explicit, msg, [])))) res)
                || Scan.fail-with (fn - => fn - => msg))
  || C-Symbol-Pos.scan-comment-no-nest err-prefix >> token (Comment (Comment-suspicious
NONE))

fun scan-fragment blanks comments sharps non-blanks =
     non-blanks (scan-token scan-char Char)
```

```
  || non-blanks (scan-token scan-string String)
  || blanks
  || comments
  || non-blanks sharps
  || non-blanks (Scan.max token-leq (Scan.literal lexicon >> token Keyword)
                                    (   scan-clangversion >> token ClangC
                                     || scan-token scan-float Float
                                     || scan-token scan-int Integer
                                     || scan-token scan-ident Ident))
  || non-blanks (Scan.one (Symbol.is-printable o #1) >> single >> token Unknown)

(* scan tokens, directive part *)

val scan-sharp1 = $$$ #
val scan-sharp2 = $$$ # @@@ $$$ #

val scan-directive =
  let val f-filter = fn Token (-, (Space -, -)) => true
                      | Token (-, (Comment -, -)) => true
                      | Token (-, (Error -, -)) => true
                      | - => false
      val sharp1 = scan-sharp1 >> token (Sharp 1)
  in    (sharp1 >> single)
    @@@ Scan.repeat (   scan-token scan-file I
                     || scan-fragment (scan-token many1-blanks-no-line Space)
                                      comments
                                      (scan-sharp2 >> token (Sharp 2) || sharp1)
                                      I)
    >> (fn tokens => Inline (Group1 (filter f-filter tokens, filter-out f-filter tokens)))
  end

local
fun !!! text scan =
  let
    fun get-pos [] =  (end-of-input)
      | get-pos (t :: -) = Position.here (pos-of t);

    fun err (syms, msg) = fn () =>
      err-prefix ^ text ^ get-pos syms ^
      (case msg of NONE =>  | SOME m => \n ^ m ());
  in Scan.!! err scan end

val pos-here-of = Position.here o pos-of

fun one-directive f =
  Scan.one (fn Token (-, (Directive ( Inline (Group1 (-, Token (-, (Sharp 1, -))
                                                  :: Token (-, s)
                                                  :: -)))
                                    , -))
```

```
                => f s
            | - => false)

val get-cond = fn Token (pos, (Directive (Inline (Group1 (toks-bl, tok1 :: tok2 :: toks))), -)) =>
                 (fn t3 => Group3 ((pos, toks-bl, [tok1, tok2], toks), range-list-of t3))
             | - => error Inline directive expected

val one-start-cond = one-directive (fn (Keyword, if) => true
                                     | (Ident -, ifdef) => true
                                     | (Ident -, ifndef) => true
                                     | - => false)
val one-elif = one-directive (fn (Ident -, elif) => true | - => false)
val one-else = one-directive (fn (Keyword, else) => true | - => false)
val one-endif = one-directive (fn (Ident -, endif) => true | - => false)

val not-cond =
 Scan.unless
  (one-start-cond || one-elif || one-else || one-endif)
  (one-not-eof
   >>
    (fn Token (pos, ( Directive (Inline (Group1 ( toks-bl
                                                , (tok1 as Token (-, (Sharp -, -)))
                                                  :: (tok2 as Token (-, (Ident -, include)))
                                                  :: toks)))
                    , s)) =>
         Token (pos, ( case toks of [] =>
                         Error ( Expecting at least one file
                                 ^ Position.here (end-pos-of tok2)
                               , Group2 (toks-bl, [tok1, tok2], toks))
                     | - => Directive (Include (Group2 (toks-bl, [tok1, tok2], toks)))
                   , s))
     | Token (pos, ( Directive (Inline (Group1 ( toks-bl
                                                , (tok1 as Token (-, (Sharp -, -)))
                                                  :: (tok2 as Token (-, (Ident -, define)))
                                                  :: toks)))
                    , s)) =>
        let
         fun define tok3 toks =
          case
            case toks of
              (tok3' as Token (pos, (Keyword, ((*)*)))) :: toks =>
                if Position.offset-of (end-pos-of tok3) = Position.offset-of (pos-of tok3')
                then
                 let
                   fun right msg pos = Right (msg ^ Position.here pos)
                   fun right1 msg = right msg o #1
                   fun right2 msg = right msg o #2
                   fun take-prefix' toks-bl toks-acc pos =
                    fn
```

```
                (tok1 as Token (-, (Ident -, -)))
                :: (tok2 as Token (pos2, (Keyword, key)))
                :: toks =>
                  if key = ,
                  then take-prefix' (tok2 :: toks-bl) (tok1 :: toks-acc) pos2 toks
                  else if key = (*( *)) then
                    Left (rev (tok2 :: toks-bl), rev (tok1 :: toks-acc), toks)
                  else
                    right1 Expecting a colon delimiter or a closing parenthesis pos2
              | Token (pos1, (Ident -, -)) :: - =>
                  right2 Expecting a colon delimiter or a closing parenthesis pos1
              | (tok1 as Token (-, (Keyword, key1)))
                :: (tok2 as Token (pos2, (Keyword, key2)))
                :: toks =>
                  if key1 = ... then
                    if key2 = (*( *))
                    then Left (rev (tok2 :: toks-bl), rev (tok1 :: toks-acc), toks)
                    else right1 Expecting a closing parenthesis pos2
                  else right2 Expecting an identifier or the keyword '...' pos
              | - => right2 Expecting an identifier or the keyword '...' pos
            in case
                case toks of
                  (tok2 as Token (-, (Keyword, (*( *))))) :: toks => Left ([tok2], [], toks)
                | - => take-prefix' [] [] pos toks
               of Left (toks-bl, toks-acc, toks) =>
                    Left (SOME (Group1 (tok3' :: toks-bl, toks-acc)), Group1 ([], toks))
                | Right x => Right x
            end
           else Left (NONE, Group1 ([], tok3' :: toks))
      | - => Left (NONE, Group1 ([], toks))
    of Left (gr1, gr2) =>
         Directive (Define (Group1 (toks-bl, [tok1, tok2]), Group1 ([], [tok3]), gr1, gr2))
      | Right msg => Error (msg, Group2 (toks-bl, [tok1, tok2], tok3 :: toks))
   fun err () = Error ( Expecting at least one identifier ^ Position.here (end-pos-of tok2)
                      , Group2 (toks-bl, [tok1, tok2], toks))
  in
    Token (pos, ( case toks of
                    (tok3 as Token (-, (Ident -, -))) :: toks => define tok3 toks
                  | (tok3 as Token (-, (Keyword, cts))) :: toks =>
                      if exists (fn cts0 => cts = cts0) keywords-ident
                      then define tok3 toks
                      else err ()
                  | - => err ()
                , s))
  end
| Token (pos, ( Directive (Inline (Group1 ( toks-bl
                                          , (tok1 as Token (-, (Sharp -, -)))
                                            :: (tok2 as Token (-, (Ident -, undef)))
                                            :: toks)))
```

```
                , s)) =>
          Token (pos, ( let fun err () = Error ( Expecting at least and at most one identifier
                                               ^ Position.here (end-pos-of tok2)
                                             , Group2 (toks-bl, [tok1, tok2], toks))
                    in
                      case toks of
                        [Token (-, (Ident -, -))] =>
                          Directive (Undef (Group2 (toks-bl, [tok1, tok2], toks)))
                      | [Token (-, (Keyword, cts))] =>
                          if exists (fn cts0 => cts = cts0) keywords-ident
                          then Directive (Undef (Group2 (toks-bl, [tok1, tok2], toks)))
                          else err ()
                      | - => err ()
                    end
                , s))
      | Token (pos, ( Directive (Inline (Group1 ( toks-bl
                                                , (tok1 as Token (-, (Sharp -, -)))
                                                  :: (tok2 as Token (-, (Integer -, -)))
                                                  :: (tok3 as Token (-, (String -, -)))
                                                  :: toks)))
                , s)) =>
          Token (pos, ( if forall is-integer toks then
                          Directive (Cpp (Group2 (toks-bl, [tok1], tok2 :: tok3 :: toks)))
                        else Error ( Expecting an integer
                                   ^ Position.here (drop-prefix is-integer toks |> hd |> pos-of)
                                   , Group2 (toks-bl, [tok1], tok2 :: tok3 :: toks))
                , s))
      | x => x))

fun scan-cond xs = xs |>
  (one-start-cond -- scan-cond-list
   -- Scan.repeat (one-elif -- scan-cond-list)
   -- Scan.option (one-else -- scan-cond-list)
   -- Scan.recover one-endif
                  (fn msg =>
                    Scan.fail-with
                      (fn toks => fn () =>
                        case toks of
                          tok :: - => can be closed here ^ Position.here (pos-of tok)
                        | - => msg))
    >> (fn (((t-if, t-elif), t-else), t-endif) =>
          Token ( Position.no-range
                , ( Directive
                      (Conditional
                        let fun t-body x = x |-> get-cond
                        in
                        ( t-body t-if
                        , map t-body t-elif
                        , Option.map t-body t-else
```

```
                    , t-body (t-endif, []))
                    end)
                , ))))

and scan-cond-list xs = xs |> Scan.repeat (not-cond || scan-cond)

val scan-directive-cond0 =
      not-cond
  || Scan.ahead ( one-start-cond |-- scan-cond-list
                |-- Scan.repeat (one-elif -- scan-cond-list)
                |-- one-else --| scan-cond-list -- (one-elif || one-else))
     :-- (fn (tok1, tok2) => !!! ( directive ^ pos-here-of tok2
                                 ^ not expected after ^ pos-here-of tok1
                                 ^, detected at)
                                 Scan.fail)
     >> #2
  || (Scan.ahead one-start-cond |-- !!! unclosed directive scan-cond)
  || (Scan.ahead one-not-eof |-- !!! missing or ambiguous beginning of conditional Scan.fail)

fun scan-directive-recover msg =
      not-cond
  || one-not-eof >>
       (fn tok as Token (pos, (-, s)) => Token (pos, (Error (msg, get-cond tok []), s)))

in

val scan-directive-cond =
  Scan.recover
    (Scan.bulk scan-directive-cond0)
    (fn msg => scan-directive-recover msg >> single)

end

(* scan tokens, main *)

val scan-ml =
  Scan.depend
    let
      fun non-blanks st scan = scan >> pair st
      fun scan-frag st =
          scan-fragment (    scan-token (C-Basic-Symbol-Pos.newline >> K [NONE]) Space >>
pair true
                          || scan-token many1-blanks-no-line Space >> pair st)
                         (non-blanks st comments)
                         ((scan-sharp2 || scan-sharp1) >> token Keyword)
                         (non-blanks false)
    in
      fn true => scan-token scan-directive Directive >> pair false || scan-frag true
       | false => scan-frag false
```

```
    end;

fun recover msg =
 (recover-char ||
  recover-string ||
  Symbol-Pos.recover-cartouche ||
  C-Symbol-Pos.recover-comment ||
  recover-int ||
  one' Symbol.not-eof)
  >> token (Error (msg, Group1 ([], [])));

fun reader scan syms =
  let
    val termination =
      if null syms then []
      else
        let
          val pos1 = List.last syms |-> Position.symbol;
          val pos2 = Position.symbol Symbol.space pos1;
        in [Token (Position.range (pos1, pos2), (Space [NONE], Symbol.space))] end;

    val backslash1 =
        $$$ \\ @@@ many C-Symbol.is-ascii-blank-no-line @@@ C-Basic-Symbol-Pos.newline
    val backslash2 = Scan.one (not o Symbol-Pos.is-eof)

    val input0 =
      Source.of-list syms
      |> Source.source Symbol-Pos.stopper (Scan.bulk (backslash1 >> SOME || backslash2 >>
K NONE))
      |> Source.map-filter I
      |> Source.exhaust
      |> map (Symbol-Pos.range #> Position.range-position)

    val input1 =
      Source.of-list syms
       |> Source.source Symbol-Pos.stopper (Scan.bulk (backslash1 >> K NONE || backslash2
>> SOME))
      |> Source.map-filter I
      |> Source.source' true
                    Symbol-Pos.stopper
                    (Scan.recover (Scan.bulk (!!!! bad input scan))
                                  (fn msg => Scan.lift (recover msg) >> single))
      |> Source.source stopper scan-directive-cond
      |> Source.exhaust
      |> (fn input => input @ termination);

    val - = app (fn pos => Output.information (Backslash newline ^ Position.here pos)) input0
    val - = Position.reports-text (map (fn pos => ((pos, Markup.intensify), )) input0);
  in (input1, check input1)
```

```
end;

in

fun op @@ ((input1, f-error-lines1), (input2, f-error-lines2)) =
  (input1 @ input2, f-error-lines1 #> f-error-lines2)

val read-init = ([], I)

fun read text = (reader scan-ml o Symbol-Pos.explode) (text, Position.none);

fun read-source' {language, symbols} scan source =
  let
    val pos = Input.pos-of source;
    val - =
      if Position.is-reported-range pos
      then Position.report pos (language (Input.is-delimited source))
      else ();
  in
    Input.source-explode source
    |> not symbols ? maps (fn (s, p) => raw-explode s |> map (rpair p))
    |> reader scan
  end;

val read-source =
  read-source' { language =
                   Markup.language' {name = C, symbols = false, antiquotes = true}, symbols =
true}
              scan-ml;

end;

end;
›

end
```

3.4 Parsing Environment

```
theory C-Environment
  imports C-Lexer-Language C-Ast
begin
```

The environment comes in two parts: a basic core structure, and a (thin) layer of utilities.

```
ML‹
signature C-ENV =
  sig
    val namespace-enum: string
```

```
val namespace-tag: string
val namespace-typedef: string
type error-lines = string list

datatype stream-lang-state  = Stream-atomic
                            | Stream-ident of Position.range * string
                            | Stream-regular
                            | Stream-string of (Position.range * string) list
type ('a, 'b, 'c) stack-elem0 = 'a * ('b * 'c * 'c)
type 'a stream = ('a, C-Lex.token) C-Scan.either list
datatype 'a parse-status = Parsed of 'a | Previous-in-stack

eqtype markup-global

type markup-ident = {global         : markup-global,
                     params         : C-Ast.CDerivedDeclr list,
                     ret            : C-Ast.CDeclSpec list parse-status}
type 'a markup-store = Position.T list * serial * 'a

type var-table =    {idents         : markup-ident markup-store Symtab.table,
                     tyidents       : markup-global markup-store Symtab.table Symtab.table}

type env-directives = ( (string * Position.range
                           -> Context.generic
                             -> C-Lex.token list * Context.generic,
                         C-Lex.token list) C-Scan.either
                        * (string * Position.range -> Context.generic -> Context.generic))
                        markup-store
                         Symtab.table

type env-lang =     {env-directives : env-directives,
                     namesupply     : int,
                     scopes         : (C-Ast.ident option * var-table) list,
                     stream-ignored : C-Antiquote.antiq stream, var-table: var-table}

type env-tree =     {context        : Context.generic,
                     error-lines    : error-lines,
                     reports-text   : C-Position.reports-text}

type env-propagation-reduce    = int option
  type env-propagation-ctxt        = env-propagation-reduce -> Context.generic -> Con-
text.generic
type env-propagation-directive = env-propagation-reduce -> env-directives
                                 -> env-lang * env-tree -> env-lang * env-tree
datatype env-propagation-bottom-up = Exec-annotation of env-propagation-ctxt
                                   | Exec-directive of env-propagation-directive
datatype env-propagation = Bottom-up of env-propagation-bottom-up
                         | Top-down of env-propagation-ctxt
```

```
type rule-static = (env-tree -> env-lang * env-tree) option
type 'a rule-output0' = {output-env: rule-static, output-pos: 'a option, output-vacuous: bool}
type ('a, 'b, 'c) stack0 = ('a, 'b, 'c) stack-elem0 list
type rule-output = C-Ast.class-Pos rule-output0'
type eval-node = Position.range * env-propagation * env-directives * bool
type ('a, 'b, 'c) rule-reduce = int * ('a, 'b, 'c) stack0 * eval-node list list
type ('a, 'b, 'c) rule-reduce0 = (('a, 'b, 'c) stack0 * env-lang * eval-node) list
type ('a, 'b, 'c) rule-reduce' = int * bool * ('a, 'b, 'c) rule-reduce0
datatype ('a, 'b, 'c) rule-type = Reduce of rule-static * ('a, 'b, 'c) rule-reduce' | Shift | Void
type ('a, 'b, 'c) rule-ml = {rule-pos: 'c * 'c, rule-type: ('a, 'b, 'c) rule-type}
 type ('a, 'b, 'c) rule-output0 = eval-node list list * ('a, 'b, 'c) rule-reduce0 * ('c * 'c)
rule-output0'
datatype 'a tree = Tree of 'a * 'a tree list
 type ('a, 'b, 'c) stack' = ('a, 'b, 'c) stack0 * eval-node list list * ('c * 'c) list * ('a, 'b, 'c)
rule-ml tree list

datatype comment-style = Comment-directive | Comment-language

datatype eval-time =
    Never
  | Lexing of Position.range * (comment-style -> Context.generic -> Context.generic)
  | Parsing of (Symbol-Pos.T list * Symbol-Pos.T list) * eval-node

datatype antiq-language = Antiq-none of C-Lex.token
                        | Antiq-stack of C-Position.reports-text * eval-time

type 'a stream-hook = ('a list * Symbol-Pos.T list * eval-node) list list

type 'a T =  {env-lang    : env-lang,
              env-tree    : env-tree,
              rule-input  : C-Ast.class-Pos list * int,
              rule-output : rule-output,
              stream-hook: Symbol-Pos.T stream-hook,
              stream-hook-excess : int stream-hook,
              stream-lang : stream-lang-state * 'a stream}

val decode-positions: string -> Position.T list
val empty-env-lang: env-lang
val empty-env-tree: Context.generic -> env-tree
val empty-rule-output: rule-output
val encode-positions: Position.T list -> string
val get-scopes: env-lang -> (C-Ast.ident option * var-table) list
val make: env-lang -> 'a stream -> env-tree -> 'a T
val map-context: (Context.generic -> Context.generic)
                 -> {context: Context.generic, error-lines: 'c, reports-text: 'd}
                 -> {context: Context.generic, error-lines: 'c, reports-text: 'd}
(* why not just env-tree *)
val map-context': (Context.generic -> 'b * Context.generic)
                  -> {context: Context.generic, error-lines: 'd, reports-text: 'e}
```

```
                    -> 'b * {context: Context.generic, error-lines: 'd, reports-text: 'e}
    (* why not just env-tree *)
     val map-reports-text:(C-Position.reports-text -> C-Position.reports-text) -> env-tree ->
env-tree
    val map-error-lines: (error-lines -> error-lines)
                        -> {context: 'c, error-lines: error-lines, reports-text: 'd}
                        -> {context: 'c, error-lines: error-lines, reports-text: 'd}
                        (* why not just : env-tree *)
    val map-namesupply: (int -> int) -> env-lang -> env-lang
    val map-env-directives: (env-directives -> env-directives) -> env-lang -> env-lang
    val map-scopes     : ((C-Ast.ident option * var-table) list
                        -> (C-Ast.ident option * var-table) list)
                         -> env-lang -> env-lang
   val map-stream-ignored: (C-Antiquote.antiq stream->C-Antiquote.antiq stream) -> env-lang
-> env-lang
    val map-var-table: (var-table -> var-table) -> env-lang -> env-lang

    val map-env-lang      : (env-lang -> env-lang) -> 'a T -> 'a T
    val map-env-lang-tree : (env-lang -> env-tree -> env-lang * env-tree) -> 'a T -> 'a T
    val map-env-lang-tree': (env-lang -> env-tree -> 'c * (env-lang * env-tree)) -> 'a T ->
'c * 'a T
    val map-env-tree   : (env-tree -> env-tree) -> 'a T -> 'a T
    val map-env-tree'  : (env-tree -> 'b * env-tree) -> 'a T -> 'b * 'a T
    val map-rule-output: (rule-output -> rule-output) -> 'a T -> 'a T
     val map-stream-hook: (Symbol-Pos.T stream-hook -> Symbol-Pos.T stream-hook) -> 'a T
-> 'a T
    val map-stream-hook-excess: (int stream-hook -> int stream-hook) -> 'a T -> 'a T
    val map-rule-input : (C-Ast.class-Pos list * int -> C-Ast.class-Pos list * int) -> 'a T ->
'a T
    val map-stream-lang: (stream-lang-state*'a stream -> stream-lang-state*'a stream)-> 'a T
-> 'a T


   val map-output-env     : (rule-static -> rule-static) -> 'a rule-output0' -> 'a rule-output0'
    val map-output-pos     : ('a option -> 'a option) -> 'a rule-output0' -> 'a rule-output0'
    val map-output-vacuous : (bool -> bool) -> 'a rule-output0' -> 'a rule-output0'

     val map-idents: (markup-ident markup-store Symtab.table -> markup-ident markup-store
Symtab.table)
                  -> var-table -> var-table
    val map-tyidents: (markup-global markup-store Symtab.table Symtab.table
                      -> markup-global markup-store Symtab.table Symtab.table )
                      -> var-table -> var-table

    val string-of: env-lang -> string
  end

›
ML — ~~/src/Pure/context.ML ‹
```

```
structure C-Env : C-ENV = struct

type 'a markup-store = Position.T list * serial * 'a

type env-directives =
     (   ( string * Position.range -> Context.generic -> C-Lex.token list * Context.generic
         , C-Lex.token list)
       C-Scan.either
     * (string * Position.range -> Context.generic -> Context.generic))
   markup-store
  Symtab.table

(**)

datatype 'a parse-status = Parsed of 'a | Previous-in-stack

type markup-global = bool (*true: global*)

type markup-ident = { global : markup-global
                    , params : C-Ast.CDerivedDeclr list
                    , ret : C-Ast.CDeclSpec list parse-status }

type var-table = { tyidents : markup-global markup-store Symtab.table (*ident name*)
                                                          Symtab.table (*internal
                                                                         namespace*)
                 , idents : markup-ident markup-store Symtab.table (*ident name*) }

type 'antiq-language-list stream = ('antiq-language-list, C-Lex.token) C-Scan.either list

— Key entry point environment to the C language
type env-lang = { var-table : var-table — current active table in the scope
                , scopes : (C-Ast.ident option * var-table) list  — parent scope tables
                , namesupply : int
                , stream-ignored : C-Antiquote.antiq stream
                , env-directives : env-directives }
(* NOTE: The distinction between type variable or identifier can not be solely made
         during the lexing process.
         Another pass on the parsed tree is required. *)

type error-lines = string list

type env-tree = { context : Context.generic
                , reports-text : C-Position.reports-text
                , error-lines : error-lines }

type rule-static = (env-tree -> env-lang * env-tree) option

(**)
```

```
datatype comment-style = Comment-directive
                       | Comment-language

type env-propagation-reduce = int (*reduce rule number*) option (* NONE: shift action *)

type env-propagation-ctxt = env-propagation-reduce -> Context.generic -> Context.generic

type env-propagation-directive =
      env-propagation-reduce -> env-directives -> env-lang * env-tree -> env-lang * env-tree

datatype env-propagation-bottom-up = Exec-annotation of env-propagation-ctxt
                                   | Exec-directive of env-propagation-directive

datatype env-propagation = Bottom-up (*during parsing*) of env-propagation-bottom-up
                         | Top-down (*after parsing*) of env-propagation-ctxt

type eval-node = Position.range
               * env-propagation
               * env-directives
               * bool (* true: skip vacuous reduce rules *)

datatype eval-time = Lexing of Position.range
                           * (comment-style -> Context.generic -> Context.generic)
                   | Parsing of (Symbol-Pos.T list (* length = number of tokens to advance *)
                                 * Symbol-Pos.T list (* length = number of steps back in stack *))
                                 * eval-node
                   | Never (* to be manually treated by the semantic back-end, and analyzed there
*)

datatype antiq-language = Antiq-stack of C-Position.reports-text * eval-time
                        | Antiq-none of C-Lex.token
```

— One of the key element of the structure is `eval_time`, relevant for the generic annotation module.

```
(**)

type ('LrTable-state, 'a, 'Position-T) stack-elem0 = 'LrTable-state
                                                   * ('a * 'Position-T * 'Position-T)
type ('LrTable-state, 'a, 'Position-T) stack0 = ('LrTable-state, 'a, 'Position-T) stack-elem0 list

type ('LrTable-state, 'svalue0, 'pos) rule-reduce0 =
      (('LrTable-state, 'svalue0, 'pos) stack0 * env-lang * eval-node) list
type ('LrTable-state, 'svalue0, 'pos) rule-reduce =
      int * ('LrTable-state, 'svalue0, 'pos) stack0 * eval-node list list
type ('LrTable-state, 'svalue0, 'pos) rule-reduce' =
      int * bool (*vacuous*) * ('LrTable-state, 'svalue0, 'pos) rule-reduce0

datatype ('LrTable-state, 'svalue0, 'pos) rule-type =
```

```
                    Void
                  | Shift
                  | Reduce of rule-static * ('LrTable-state, 'svalue0, 'pos) rule-reduce'

type ('LrTable-state, 'svalue0, 'pos) rule-ml =
  { rule-pos : 'pos * 'pos
  , rule-type : ('LrTable-state, 'svalue0, 'pos) rule-type }

(**)

type 'class-Pos rule-output0' = { output-pos : 'class-Pos option
                                , output-vacuous : bool
                                , output-env : rule-static }

type ('LrTable-state, 'svalue0, 'pos) rule-output0 =
                                    eval-node list list (* delayed *)
                                  * ('LrTable-state, 'svalue0, 'pos) rule-reduce0 (* actual *)
                                  * ('pos * 'pos) rule-output0'

type rule-output = C-Ast.class-Pos rule-output0'

(**)

datatype stream-lang-state = Stream-ident of Position.range * string
                           | Stream-string of (Position.range * string) list
                           | Stream-atomic
                           | Stream-regular

type 'a stream-hook = ('a list * Symbol-Pos.T list * eval-node) list list

type 'a T =  {env-lang: env-lang,
              env-tree: env-tree,
              rule-input: C-Ast.class-Pos list * int,
              rule-output: rule-output,
              stream-hook: Symbol-Pos.T stream-hook,
              stream-hook-excess : int stream-hook,
              stream-lang: stream-lang-state * 'a stream}


type T' =  (C-Antiquote.antiq * antiq-language list) T

(**)

datatype 'a tree = Tree of 'a * 'a tree list

type ('LrTable-state, 'a, 'Position-T) stack' =
     ('LrTable-state, 'a, 'Position-T) stack0
   * eval-node list list
   * ('Position-T * 'Position-T) list
```

```
  * ('LrTable-state, 'a, 'Position-T) rule-ml tree list

(**)

fun map-env-lang f
      {env-lang, env-tree, rule-output, rule-input, stream-hook, stream-hook-excess, stream-lang}
=
                  {env-lang = f
                            env-lang, env-tree = env-tree, rule-output = rule-output,
                   rule-input = rule-input, stream-hook = stream-hook,
                   stream-hook-excess = stream-hook-excess, stream-lang = stream-lang}

fun map-env-tree f
      {env-lang, env-tree, rule-output, rule-input, stream-hook, stream-hook-excess, stream-lang}
=
                  {env-lang = env-lang, env-tree = f
                                                   env-tree, rule-output = rule-output,
                   rule-input = rule-input, stream-hook = stream-hook,
                   stream-hook-excess = stream-hook-excess, stream-lang = stream-lang}

fun map-rule-output f
      {env-lang, env-tree, rule-output, rule-input, stream-hook, stream-hook-excess, stream-lang}
=
                  {env-lang = env-lang, env-tree = env-tree, rule-output = f
                                                                           rule-output,
                   rule-input = rule-input, stream-hook = stream-hook,
                   stream-hook-excess = stream-hook-excess, stream-lang = stream-lang}

fun map-rule-input f
      {env-lang, env-tree, rule-output, rule-input, stream-hook, stream-hook-excess, stream-lang}
=
                  {env-lang = env-lang, env-tree = env-tree, rule-output = rule-output,
                   rule-input = f
                                rule-input, stream-hook = stream-hook,
                   stream-hook-excess = stream-hook-excess, stream-lang = stream-lang}

fun map-stream-hook f
      {env-lang, env-tree, rule-output, rule-input, stream-hook, stream-hook-excess, stream-lang}
=
                  {env-lang = env-lang, env-tree = env-tree, rule-output = rule-output,
                   rule-input = rule-input, stream-hook = f
                                                          stream-hook,
                   stream-hook-excess = stream-hook-excess, stream-lang = stream-lang}

fun map-stream-hook-excess f
      {env-lang, env-tree, rule-output, rule-input, stream-hook, stream-hook-excess, stream-lang}
=
                  {env-lang = env-lang, env-tree = env-tree, rule-output = rule-output,
                   rule-input = rule-input, stream-hook = stream-hook,
```

```
                stream-hook-excess = f
                                       stream-hook-excess, stream-lang = stream-lang}

fun map-stream-lang f
    {env-lang, env-tree, rule-output, rule-input, stream-hook, stream-hook-excess, stream-lang}
=
                {env-lang = env-lang, env-tree = env-tree, rule-output = rule-output,
                 rule-input = rule-input, stream-hook = stream-hook,
                 stream-hook-excess = stream-hook-excess, stream-lang = f
                                                                        stream-lang}

(**)

fun map-output-pos f {output-pos, output-vacuous, output-env} =
                {output-pos = f output-pos, output-vacuous = output-vacuous, output-env =
output-env}

fun map-output-vacuous f {output-pos, output-vacuous, output-env} =
                {output-pos = output-pos, output-vacuous = f output-vacuous, output-env =
output-env}

fun map-output-env f {output-pos, output-vacuous, output-env} =
                {output-pos = output-pos, output-vacuous = output-vacuous, output-env = f
output-env}

(**)

fun map-tyidents f {tyidents, idents} =
                                    {tyidents = f tyidents, idents = idents}

fun map-idents f {tyidents, idents} =
                                    {tyidents = tyidents, idents = f idents}

(**)

fun map-var-table f {var-table, scopes, namesupply, stream-ignored, env-directives} =
                                {var-table = f
                                                 var-table, scopes = scopes, namesupply = namesupply,
                                 stream-ignored = stream-ignored, env-directives = env-directives}

fun map-scopes f {var-table, scopes, namesupply, stream-ignored, env-directives} =
                                {var-table = var-table, scopes = f
                                                                    scopes, namesupply = namesupply,
                                 stream-ignored = stream-ignored, env-directives = env-directives}

fun map-namesupply f {var-table, scopes, namesupply, stream-ignored, env-directives} =
                                {var-table = var-table, scopes = scopes, namesupply = f
                                                                                      namesupply,
                                 stream-ignored = stream-ignored, env-directives = env-directives}
```

```
fun map-stream-ignored f {var-table, scopes, namesupply, stream-ignored, env-directives} =
                          {var-table = var-table, scopes = scopes, namesupply = namesupply,
                            stream-ignored = f
                                               stream-ignored, env-directives = env-directives}

fun map-env-directives f {var-table, scopes, namesupply, stream-ignored, env-directives} =
                          {var-table = var-table, scopes = scopes, namesupply = namesupply,
                            stream-ignored = stream-ignored, env-directives = f
                                                                 env-directives}

(**)

fun map-context f {context, reports-text, error-lines} =
                  {context = f context, reports-text = reports-text, error-lines = error-lines}

fun map-context' f {context, reports-text, error-lines} =
          let val (res, context) = f context
          in (res, {context = context, reports-text = reports-text, error-lines = error-lines})
          end

fun map-reports-text f {context, reports-text, error-lines} =
                  {context = context, reports-text = f reports-text, error-lines = error-lines}

fun map-error-lines f {context, reports-text, error-lines} =
                  {context = context, reports-text = reports-text, error-lines = f error-lines}

(**)

fun map-env-tree' f
    {env-lang, env-tree, rule-output, rule-input, stream-hook, stream-hook-excess, stream-lang}
=
          let val (res, env-tree) = f env-tree
          in (res, {env-lang = env-lang, env-tree = env-tree, rule-output = rule-output,
                    rule-input = rule-input, stream-hook = stream-hook,
                    stream-hook-excess = stream-hook-excess, stream-lang = stream-lang})
          end

fun map-env-lang-tree f
    {env-lang, env-tree, rule-output, rule-input, stream-hook, stream-hook-excess, stream-lang}
=
          let val (env-lang, env-tree) = f env-lang env-tree
          in {env-lang = env-lang, env-tree = env-tree, rule-output = rule-output,
              rule-input = rule-input, stream-hook = stream-hook,
              stream-hook-excess = stream-hook-excess, stream-lang = stream-lang}
          end

fun map-env-lang-tree' f
    {env-lang, env-tree, rule-output, rule-input, stream-hook, stream-hook-excess, stream-lang}
```

```
=
            let val (res, (env-lang, env-tree)) = f env-lang env-tree
            in (res, {env-lang = env-lang, env-tree = env-tree, rule-output = rule-output,
                    rule-input = rule-input, stream-hook = stream-hook,
                    stream-hook-excess = stream-hook-excess, stream-lang = stream-lang})
            end

(**)

fun get-scopes (t : env-lang) = #scopes t

(**)

val empty-env-lang : env-lang =
      {var-table = {tyidents = Symtab.make [], idents = Symtab.make []},
       scopes = [], namesupply = 0, stream-ignored = [],
       env-directives = Symtab.empty}
fun empty-env-tree context =
      {context = context, reports-text = [], error-lines = []}
val empty-rule-output : rule-output =
      {output-pos = NONE, output-vacuous = true, output-env = NONE}
fun make env-lang stream-lang env-tree =
     { env-lang = env-lang
     , env-tree = env-tree
     , rule-output = empty-rule-output
     , rule-input = ([], 0)
     , stream-hook = []
     , stream-hook-excess = []
     , stream-lang = ( Stream-regular
                 , map-filter (fn C-Scan.Right (C-Lex.Token (-, (C-Lex.Space -, -))) => NONE
                                | C-Scan.Right (C-Lex.Token (-, (C-Lex.Comment -, -))) =>
NONE
                                | C-Scan.Right tok => SOME (C-Scan.Right tok)
                                | C-Scan.Left antiq => SOME (C-Scan.Left antiq))
                              stream-lang) }
fun string-of (env-lang : env-lang) =
  let fun dest0 x f = x |> Symtab.dest |> map f
      fun dest {tyidents, idents} =
          (dest0 tyidents #1, dest0 idents (fn (i, (-,-,v)) =>
                                    (i, if #global v then global else local)))
  in make-string ( (var-table, dest (#var-table env-lang))
                 , (scopes, map (fn (id, i) =>
                                 ( Option.map (fn C-Ast.Ident0 (i, -, -) =>
                                                 C-Ast.meta-of-logic i)
                                              id
                                 , dest i))
                                (#scopes env-lang))
                 , (namesupply, #namesupply env-lang)
                 , (stream-ignored, #stream-ignored env-lang)) end
```

```
val namespace-typedef = typedef
val namespace-tag = tag
val namespace-enum = namespace-tag

(**)

val encode-positions =
    map (Position.dest
      #> (fn pos => ((#line pos, #offset pos, #end-offset pos),
       Properties.make-string Markup.fileN (#file (#props pos)) @
       Properties.make-string Markup.idN (#id (#props pos)))))
  #> let open XML.Encode in list (pair (triple int int int) properties) end
  #> YXML.string-of-body

val decode-positions =
    YXML.parse-body
  #> let open XML.Decode in list (pair (triple int int int) properties) end
  #> map ((fn ((line, offset, end-offset), props) =>
         {line = line, offset = offset, end-offset = end-offset, props = props})
        #> Position.of-props)

end
›
```

ML — `~~/src/Pure/context.ML` ‹

```
structure C-Env-Ext =
struct

local
fun map-tyidents' f = C-Env.map-var-table (C-Env.map-tyidents f)
fun map-tyidents f = C-Env.map-env-lang (map-tyidents' f)
in
fun map-tyidents-typedef f =
                  map-tyidents (Symtab.map-default (C-Env.namespace-typedef, Symtab.empty)
f)
fun map-tyidents-enum f = map-tyidents (Symtab.map-default (C-Env.namespace-enum,
Symtab.empty) f)
fun map-tyidents'-typedef f =
                  map-tyidents' (Symtab.map-default (C-Env.namespace-typedef, Symtab.empty)
f)
fun map-tyidents'-enum f = map-tyidents' (Symtab.map-default (C-Env.namespace-enum,
Symtab.empty) f)
end
fun map-idents' f = C-Env.map-var-table (C-Env.map-idents f)
fun map-idents f = C-Env.map-env-lang (map-idents' f)

(**)
```

```
fun map-var-table f = C-Env.map-env-lang (C-Env.map-var-table f)
fun map-scopes f = C-Env.map-env-lang (C-Env.map-scopes f)
fun map-namesupply f = C-Env.map-env-lang (C-Env.map-namesupply f)
fun map-stream-ignored f = C-Env.map-env-lang (C-Env.map-stream-ignored f)

(**)

local
fun get-tyidents' namespace (env-lang : C-Env.env-lang) =
  case Symtab.lookup (env-lang |> #var-table |> #tyidents) namespace of
    NONE => Symtab.empty
  | SOME t => t

fun get-tyidents namespace (t : 'a C-Env.T) = get-tyidents' namespace (#env-lang t)
in
fun get-tyidents-typedef env= get-tyidents C-Env.namespace-typedef env
fun get-tyidents-enum env = get-tyidents C-Env.namespace-enum env
fun get-tyidents'-typedef env = get-tyidents' C-Env.namespace-typedef env
fun get-tyidents'-enum env = get-tyidents' C-Env.namespace-enum env
end

fun get-idents (t: 'a C-Env.T) = #env-lang t |> #var-table |> #idents
fun get-idents' (env:C-Env.env-lang) = env |> #var-table |> #idents

(**)

fun get-var-table (t: 'a C-Env.T) = #env-lang t |> #var-table
fun get-scopes (t:'a C-Env.T) = #env-lang t |> #scopes
fun get-namesupply (t: 'a C-Env.T) = #env-lang t |> #namesupply

(**)

fun map-output-pos f = C-Env.map-rule-output (C-Env.map-output-pos f)
fun map-output-vacuous f = C-Env.map-rule-output (C-Env.map-output-vacuous f)
fun map-output-env f = C-Env.map-rule-output (C-Env.map-output-env f)

(**)

fun get-output-pos (t : 'a C-Env.T) = #rule-output t |> #output-pos

(**)

fun map-context f = C-Env.map-env-tree (C-Env.map-context f)
fun map-reports-text f = C-Env.map-env-tree (C-Env.map-reports-text f)

(**)

fun get-context (t : 'a C-Env.T) = #env-tree t |> #context
fun get-reports-text (t : 'a C-Env.T) = #env-tree t |> #reports-text
```

(**)

fun map-env-directives' f {var-table, scopes, namesupply, stream-ignored, env-directives} =
* let val (res, env-directives) = f env-directives*
* in (res, {var-table = var-table, scopes = scopes, namesupply = namesupply,*
*   stream-ignored = stream-ignored, env-directives = env-directives})*
* end*

(**)

fun map-stream-lang' f
*   {env-lang, env-tree, rule-output, rule-input, stream-hook, stream-hook-excess, stream-lang}*
=
* let val (res, stream-lang) = f stream-lang*
* in (res, {env-lang = env-lang, env-tree = env-tree, rule-output = rule-output,*
*   rule-input = rule-input, stream-hook = stream-hook,*
*   stream-hook-excess = stream-hook-excess, stream-lang = stream-lang})*
* end*

(**)

fun context-map (f : C-Env.env-tree -> C-Env.env-tree) =
* C-Env.empty-env-tree #> f #> #context*

*fun context-map' (f : C-Env.env-tree -> 'a * C-Env.env-tree) =*
* C-Env.empty-env-tree #> f #> apsnd #context*

(**)

fun list-lookup tab name = flat (map (fn (x, -, -) => x) (the-list (Symtab.lookup tab name)))

end

›

end

3.5 Core Language: Parsing Support (C Language without Annotations)

theory *C-Parser-Language*
  imports *C-Environment*
begin

As mentioned in *Isabelle-C.C-Ast*, Isabelle/C depends on certain external parsing libraries, such as `../../src_ext/mlton`, and more specifically `../../src_ext/mlton/lib/mlyacc-lib`. Actually, the sole theory making use of the files in `../../src_ext/`

`mlton/lib/mlyacc-lib` is the present `C_Parser_Language.thy`. (Any other remaining files in `../../src_ext/mlton` are not used by Isabelle/C, they come from the original repository of MLton: https://github.com/MLton/mlton.)

3.5.1 Parsing Library (Including Semantic Functions)

ML — `../generated/c_grammar_fun.grm.sml`

```
‹
signature C-GRAMMAR-RULE-LIB =
sig
  type arg = (C-Antiquote.antiq * C-Env.antiq-language list) C-Env.T
  type 'a monad = arg -> 'a * arg

  (* type aliases *)
  type class-Pos = C-Ast.class-Pos
  type reports-text0' = { markup : Markup.T, markup-body : string }
  type reports-text0 = reports-text0' list -> reports-text0' list
  type ('a, 'b) reports-base = ('a C-Env.markup-store * Position.T list,
                                Position.T list * 'a C-Env.markup-store option) C-Ast.either ->
                               Position.T list ->
                               string ->
                               'b ->
                               'b
   (**)
  type NodeInfo = C-Ast.nodeInfo
  type CStorageSpec = NodeInfo C-Ast.cStorageSpecifier
  type CFunSpec = NodeInfo C-Ast.cFunctionSpecifier
  type CConst = NodeInfo C-Ast.cConstant
  type 'a CInitializerList = ('a C-Ast.cPartDesignator List.list * 'a C-Ast.cInitializer) List.list
  type CTranslUnit = NodeInfo C-Ast.cTranslationUnit
  type CExtDecl = NodeInfo C-Ast.cExternalDeclaration
  type CFunDef = NodeInfo C-Ast.cFunctionDef
  type CDecl = NodeInfo C-Ast.cDeclaration
  type CDeclr = NodeInfo C-Ast.cDeclarator
  type CDerivedDeclr = NodeInfo C-Ast.cDerivedDeclarator
  type CArrSize = NodeInfo C-Ast.cArraySize
  type CStat = NodeInfo C-Ast.cStatement
  type CAsmStmt = NodeInfo C-Ast.cAssemblyStatement
  type CAsmOperand = NodeInfo C-Ast.cAssemblyOperand
  type CBlockItem = NodeInfo C-Ast.cCompoundBlockItem
  type CDeclSpec = NodeInfo C-Ast.cDeclarationSpecifier
  type CTypeSpec = NodeInfo C-Ast.cTypeSpecifier
  type CTypeQual = NodeInfo C-Ast.cTypeQualifier
  type CAlignSpec = NodeInfo C-Ast.cAlignmentSpecifier
  type CStructUnion = NodeInfo C-Ast.cStructureUnion
  type CEnum = NodeInfo C-Ast.cEnumeration
  type CInit = NodeInfo C-Ast.cInitializer
  type CInitList = NodeInfo CInitializerList
```

```
type CDesignator = NodeInfo C-Ast.cPartDesignator
type CAttr = NodeInfo C-Ast.cAttribute
type CExpr = NodeInfo C-Ast.cExpression
type CBuiltin = NodeInfo C-Ast.cBuiltinThing
type CStrLit = NodeInfo C-Ast.cStringLiteral
  (**)
type ClangCVersion = C-Ast.clangCVersion
type Ident = C-Ast.ident
type Position = C-Ast.positiona
type PosLength = Position * int
type Name = C-Ast.namea
type Bool = bool
type CString = C-Ast.cString
type CChar = C-Ast.cChar
type CInteger = C-Ast.cInteger
type CFloat = C-Ast.cFloat
type CStructTag = C-Ast.cStructTag
type CUnaryOp = C-Ast.cUnaryOp
type 'a CStringLiteral = 'a C-Ast.cStringLiteral
type 'a CConstant = 'a C-Ast.cConstant
type ('a, 'b) Either = ('a, 'b) C-Ast.either
type CIntRepr = C-Ast.cIntRepr
type CIntFlag = C-Ast.cIntFlag
type CAssignOp = C-Ast.cAssignOp
type Comment = C-Ast.comment
  (**)
type 'a Reversed = 'a C-Ast.Reversed
type CDeclrR = C-Ast.CDeclrR
type 'a Maybe = 'a C-Ast.optiona
type 'a Located = 'a C-Ast.Located
  (**)
structure List : sig val reverse : 'a list -> 'a list end

(* monadic operations *)
val return : 'a -> 'a monad
val bind : 'a monad -> ('a -> 'b monad) -> 'b monad
val bind' : 'b monad -> ('b -> unit monad) -> 'b monad
val >> : unit monad * 'a monad -> 'a monad

(* position reports *)
val markup-make : ('a -> reports-text0' option) ->
                  ('a -> 'b) ->
                  ('b option -> string) ->
                  ((Markup.T -> reports-text0) ->
                   bool ->
                   ('b, 'b option * string * reports-text0) C-Ast.either ->
                   reports-text0) ->
                  ('a, C-Position.reports-text) reports-base
val markup-tvar : (C-Env.markup-global, C-Position.reports-text) reports-base
```

```
val markup-var-enum : (C-Env.markup-global, C-Position.reports-text) reports-base
val markup-var : (C-Env.markup-ident, C-Position.reports-text) reports-base
val markup-var-bound : (C-Env.markup-ident, C-Position.reports-text) reports-base
val markup-var-improper : (C-Env.markup-ident, C-Position.reports-text) reports-base

(* Language.C.Data.RList *)
val empty : 'a list Reversed
val singleton : 'a -> 'a list Reversed
val snoc : 'a list Reversed -> 'a -> 'a list Reversed
val rappend : 'a list Reversed -> 'a list -> 'a list Reversed
val rappendr : 'a list Reversed -> 'a list Reversed -> 'a list Reversed
val rmap : ('a -> 'b) -> 'a list Reversed -> 'b list Reversed

(* Language.C.Data.Position *)
val posOf : 'a -> Position
val posOf' : bool -> class-Pos -> Position * int
val make-comment :
      Symbol-Pos.T list -> Symbol-Pos.T list -> Symbol-Pos.T list -> Position.range ->
Comment

(* Language.C.Data.Node *)
val mkNodeInfo' : Position -> PosLength -> Name -> NodeInfo
val decode : NodeInfo -> (class-Pos, string) Either
val decode-error' : NodeInfo -> Position.range

(* Language.C.Data.Ident *)
val quad : string list -> int
val ident-encode : string -> int
val ident-decode : int -> string
val mkIdent : Position * int -> string -> Name -> Ident
val internalIdent : string -> Ident

(* Language.C.Syntax.AST *)
val liftStrLit : 'a CStringLiteral -> 'a CConstant

(* Language.C.Syntax.Constants *)
val concatCStrings : CString list -> CString

(* Language.C.Parser.ParserMonad *)
val getNewName : Name monad
val shadowTypedef0'''' : string ->
                         Position.T list ->
                         C-Env.markup-ident ->
                         C-Env.env-lang ->
                         C-Env.env-tree ->
                         C-Env.env-lang * C-Env.env-tree
val shadowTypedef0' : C-Ast.CDeclSpec list C-Env.parse-status ->
                      bool ->
                      C-Ast.ident * C-Ast.CDerivedDeclr list ->
```

```
                    C-Env.env-lang ->
                    C-Env.env-tree ->
                    C-Env.env-lang * C-Env.env-tree
  val isTypeIdent : string -> arg -> bool
  val enterScope : unit monad
  val leaveScope : unit monad
  val getCurrentPosition : Position monad

  (* Language.C.Parser.Tokens *)
  val CTokCLit : CChar -> (CChar -> 'a) -> 'a
  val CTokILit : CInteger -> (CInteger -> 'a) -> 'a
  val CTokFLit : CFloat -> (CFloat -> 'a) -> 'a
  val CTokSLit : CString -> (CString -> 'a) -> 'a

  (* Language.C.Parser.Parser *)
  val reverseList : 'a list -> 'a list Reversed
  val L : 'a -> int -> 'a Located monad
  val unL : 'a Located -> 'a
  val withNodeInfo : int -> (NodeInfo -> 'a) -> 'a monad
  val withNodeInfo-CExtDecl : CExtDecl -> (NodeInfo -> 'a) -> 'a monad
  val withNodeInfo-CExpr : CExpr list Reversed -> (NodeInfo -> 'a) -> 'a monad
  val withLength : NodeInfo -> (NodeInfo -> 'a) -> 'a monad
  val reverseDeclr : CDeclrR -> CDeclr
  val withAttribute : int -> CAttr list -> (NodeInfo -> CDeclrR) -> CDeclrR monad
  val withAttributePF : int -> CAttr list -> (NodeInfo -> CDeclrR -> CDeclrR) ->
                        (CDeclrR -> CDeclrR) monad
  val appendObjAttrs : CAttr list -> CDeclr -> CDeclr
  val withAsmNameAttrs : CStrLit Maybe * CAttr list -> CDeclrR -> CDeclrR monad
  val appendDeclrAttrs : CAttr list -> CDeclrR -> CDeclrR
  val ptrDeclr : CDeclrR -> CTypeQual list -> NodeInfo -> CDeclrR
  val funDeclr : CDeclrR -> (Ident list, (CDecl list * Bool)) Either -> CAttr list -> NodeInfo
->
                CDeclrR
  val arrDeclr : CDeclrR -> CTypeQual list -> Bool -> Bool -> CExpr Maybe -> NodeInfo
-> CDeclrR
  val liftTypeQuals : CTypeQual list Reversed -> CDeclSpec list
  val liftCAttrs : CAttr list -> CDeclSpec list
  val addTrailingAttrs : CDeclSpec list Reversed -> CAttr list -> CDeclSpec list Reversed
  val emptyDeclr : CDeclrR
  val mkVarDeclr : Ident -> NodeInfo -> CDeclrR
  val doDeclIdent : CDeclSpec list -> CDeclrR -> unit monad
  val ident-of-decl : (Ident list, CDecl list * bool) C-Ast.either ->
                      (Ident * CDerivedDeclr list * CDeclSpec list) list
  val doFuncParamDeclIdent : CDeclr -> unit monad
end

structure C-Grammar-Rule-Lib : C-GRAMMAR-RULE-LIB =
struct
  open C-Ast
```

```
type arg = (C-Antiquote.antiq * C-Env.antiq-language list) C-Env.T
type 'a monad = arg -> 'a * arg

(**)
type reports-text0' = { markup : Markup.T, markup-body : string }
type reports-text0 = reports-text0' list -> reports-text0' list
type 'a reports-store = Position.T list * serial * 'a
type ('a, 'b) reports-base = ('a C-Env.markup-store * Position.T list,
                              Position.T list * 'a C-Env.markup-store option) C-Ast.either ->
                             Position.T list ->
                             string ->
                             'b ->
                             'b
fun markup-init markup = { markup = markup, markup-body =  }
val look-idents = C-Env-Ext.list-lookup o C-Env-Ext.get-idents
val look-idents' = C-Env-Ext.list-lookup o C-Env-Ext.get-idents'
val look-tyidents-typedef = C-Env-Ext.list-lookup o C-Env-Ext.get-tyidents-typedef
val look-tyidents'-typedef = C-Env-Ext.list-lookup o C-Env-Ext.get-tyidents'-typedef
val To-string0 = meta-of-logic
val ident-encode =
  Word8Vector.foldl (fn (w, acc) => Word8.toInt w + acc * 256) 0 o Byte.stringToBytes
fun ident-decode nb = radixpand (256, nb) |> map chr |> implode
fun reverse l = rev l

fun report - [] - = I
  | report markup ps x =
      let val ms = markup x
      in fold (fn p => fold (fn {markup, markup-body} => cons ((p, markup), markup-body))
ms) ps
      end

fun markup-make typing get-global desc report' data =
 report
 (fn name =>
  let
    val (def, ps, id, global, typing) =
      case data of
        Left ((ps, id, param), ps' as - :: -) =>
          ( true
          , ps
          , id
          , Right ( SOME (get-global param)
                  , Redefinition of " ^ quote name ^ " Position.here-list ps
              — Any positions provided here will be explicitly reported, which might not be the
desired effect. So we explicitly refer the reader to a separate tooltip.
                                    ^ " (more details in the command modifier tooltip)
                  , cons { markup = Markup.class-parameter
                         , markup-body = redefining this ^ Position.here-list ps' })
          , typing param)
```

```
      | Left ((ps, id, param), -) => (true, ps, id, Left (get-global param), typing param)
     | Right (-, SOME (ps, id, param)) => (false, ps, id, Left (get-global param), typing param)
      | Right (ps, -) => ( true
                         , ps
                         , serial ()
                         , Right (NONE, Undeclared ^ quote name ^ Position.here-list ps, I)
                         , NONE)
    fun markup-elem name = (name, (name, []): Markup.T)
    val (varN, var) = markup-elem (desc (case global of Left b => SOME b
                                                      | Right (SOME b, -, -) => SOME b
                                                      | - => NONE));
    val cons' = cons o markup-init
  in
   (cons' var
    #> report' cons' def global
    #> (case typing of NONE => I | SOME x => cons x))
     (map (markup-init o Position.make-entity-markup {def = def} id varN o pair name) ps)
  end)

fun markup-make' typing get-global desc report =
  markup-make
    typing
    get-global
    (fn global =>
      C ^ (case global of SOME true => global
                        | SOME false => local
                        | NONE => )
        ^ desc)
    (fn cons' => fn def =>
     fn Left b => report cons' def b
      | Right (b, msg, f) => tap (fn - => Output.information msg)
                        #> f
                        #> (case b of NONE => cons' Markup.free | SOME b => report cons'
def b))

fun markup-tvar0 desc =
  markup-make'
    (K NONE)
    I
    desc
    (fn cons' => fn def =>
     fn true => cons' (if def then Markup.free else Markup.ML-keyword3)
      | false => cons' Markup.skolem)

val markup-tvar = markup-tvar0 type variable
val markup-var-enum = markup-tvar0 enum tag

fun string-of-list f =
  (fn [] => NONE | [s] => SOME s | s => SOME ([ ^ String.concatWith ,  s ^ ]))
```

```
  o map f

val string-of-cDeclarationSpecifier =
  fn C-Ast.CStorageSpec0 - => storage
   | C-Ast.CTypeSpec0 t => (case t of
                              CVoidType0 - => void
                            | CCharType0 - => char
                            | CShortType0 - => short
                            | CIntType0 - => int
                            | CLongType0 - => long
                            | CFloatType0 - => float
                            | CDoubleType0 - => double
                            | CSignedType0 - => signed
                            | CUnsigType0 - => unsig
                            | CBoolType0 - => bool
                            | CComplexType0 - => complex
                            | CInt128Type0 - => int128
                            | CSUType0 - => SU
                            | CEnumType0 - => enum
                            | CTypeDef0 - => typedef
                            | CTypeOfExpr0 - => type-of-expr
                            | CTypeOfType0 - => type-of-type
                            | CAtomicType0 - => atomic)
   | C-Ast.CTypeQual0 - => type-qual
   | C-Ast.CFunSpec0 - => fun
   | C-Ast.CAlignSpec0 - => align

fun typing {params, ret, ...} =
  SOME
  { markup = Markup.typing
  , markup-body =
      case
       ( string-of-list
          (fn C-Ast.CPtrDeclr0 - => pointer
           | C-Ast.CArrDeclr0 - => array
           | C-Ast.CFunDeclr0 (C-Ast.Left -, -, -) => function [...] ->
           | C-Ast.CFunDeclr0 (C-Ast.Right (l-decl, -), -, -) =>
              function
             ^(String.concatWith
                 ->
                (map (fn CDecl0 ([decl], -, -) => string-of-cDeclarationSpecifier decl
                       | CDecl0 (l, -, -) => ( ^String.concatWith

                                                 (map string-of-cDeclarationSpecifier l)
                                               ^)
                       | CStaticAssert0 - => static-assert)
                     l-decl))
           ^ ->)
          params
```

```
    , case ret of C-Env.Previous-in-stack => SOME ...
                | C-Env.Parsed ret => string-of-list string-of-cDeclarationSpecifier ret)
   of (NONE, NONE) => let val - = warning markup-var: Not yet implemented in  end
    | (SOME params, NONE) => params
    | (NONE, SOME ret) => ret
    | (SOME params, SOME ret) => params ^   ^ ret }

val markup-var =
  markup-make'
    typing
    #global
    variable
    (fn cons' => fn def =>
    fn true => if def then cons' Markup.free else cons' Markup.delimiter (*explicit black color,
                                          otherwise the default color of constants might
                                          be automatically chosen (especially in term
                                          cartouches)*)
     | false => cons' Markup.bound)

val markup-var-bound =
  markup-make' typing #global variable (fn cons' => K (K (cons' Markup.bound)))

val markup-var-improper =
  markup-make' typing #global variable (fn cons' => K (K (cons' Markup.improper)))

(**)
val return = pair
fun bind f g = f #-> g
fun bind' f g = bind f (fn r => bind (g r) (fn () => return r))
fun a >> b = a #> b o #2
fun sequence- f = fn [] => return ()
                | x :: xs => f x >> sequence- f xs

(* Language.C.Data.RList *)
val empty = []
fun singleton x = [x]
fun snoc xs x = x :: xs
fun rappend xs ys = rev ys @ xs
fun rappendr xs ys = ys @ xs
val rmap = map
val viewr = fn [] => error viewr: empty RList
           | x :: xs => (xs, x)

(* Language.C.Data.Position *)
val nopos = NoPosition
fun posOf - = NoPosition
fun posOf' mk-range =
  (if mk-range then Position.range else I)
  #> (fn (pos1, pos2) =>
```

```
        let val {offset = offset, end-offset = end-offset, ...} = Position.dest pos1
        in ( Position offset (From-string (C-Env.encode-positions [pos1, pos2])) 0 0
           , end-offset - offset)
        end)
fun posOf'' node env =
  let val (stack, len) = #rule-input env
      val (mk-range, (pos1a, pos1b)) = case node of Left i => (true, nth stack (len - i - 1))
                                                  | Right range => (false, range)
      val (pos2a, pos2b) = nth stack 0
  in ( (posOf' mk-range (pos1a, pos1b) |> #1, posOf' true (pos2a, pos2b))
     , env |> C-Env-Ext.map-output-pos (K (SOME (pos1a, pos2b)))
           |> C-Env-Ext.map-output-vacuous (K false)) end
val posOf''' = posOf'' o Left
val internalPos = InternalPosition
fun make-comment body-begin body body-end range =
  Commenta ( posOf' false range |> #1
           , From-string (Symbol-Pos.implode (body-begin @ body @ body-end))
           , case body-end of [] => SingleLine | - => MultiLine)

(* Language.C.Data.Node *)
val undefNode = OnlyPos nopos (nopos, ~1)
fun mkNodeInfoOnlyPos pos = OnlyPos pos (nopos, ~1)
fun mkNodeInfo pos name = NodeInfo pos (nopos, ~1) name
val mkNodeInfo' = NodeInfo
val decode =
 (fn OnlyPos0 range => range
   | NodeInfo0 (pos1, (pos2, len2), -) => (pos1, (pos2, len2)))
 #> (fn (Position0 (-, s1, -, -), (Position0 (-, s2, -, -), -)) =>
        (case (C-Env.decode-positions (To-string0 s1), C-Env.decode-positions (To-string0 s2))
           of ([pos1, -], [-, pos2]) => Left (Position.range (pos1, pos2))
            | - => Right Expecting 2 elements)
     | - => Right Invalid position)
fun decode-error' x = case decode x of Left x => x | Right msg => error msg
fun decode-error x = Right (decode-error' x)
val nameOfNode = fn OnlyPos0 - => NONE
                  | NodeInfo0 (-, -, name) => SOME name

(* Language.C.Data.Ident *)
local
  val bits7 = Integer.pow 7 2
  val bits14 = Integer.pow 14 2
  val bits21 = Integer.pow 21 2
  val bits28 = Integer.pow 28 2
in
fun quad s = case s of
  [] => 0
| c1 :: [] => ord c1
| c1 :: c2 :: [] => ord c2 * bits7 + ord c1
| c1 :: c2 :: c3 :: [] => ord c3 * bits14 + ord c2 * bits7 + ord c1
```

```
  | c1 :: c2 :: c3 :: c4 :: s => ((ord c4 * bits21
                                  + ord c3 * bits14
                                  + ord c2 * bits7
                                  + ord c1)
                                 mod bits28)
                                + (quad s mod bits28)
  end

  local
    fun internalIdent0 pos s = Ident (From-string s, ident-encode s, pos)
  in
  fun mkIdent (pos, len) s name = internalIdent0 (mkNodeInfo' pos (pos, len) name) s
  val internalIdent = internalIdent0 (mkNodeInfoOnlyPos internalPos)
  end

  (* Language.C.Syntax.AST *)
  fun liftStrLit (CStrLit0 (str, at)) = CStrConst str at

  (* Language.C.Syntax.Constants *)
  fun concatCStrings cs =
        CString0 (flattena (map (fn CString0 (s,-) => s) cs), exists (fn CString0 (-, b) => b)
cs)

  (* Language.C.Parser.ParserMonad *)
  fun getNewName env =
    (Namea (C-Env-Ext.get-namesupply env), C-Env-Ext.map-namesupply (fn x => x + 1) env)

  fun addTypedef (Ident0 (-, i, node)) env =
    let val name = ident-decode i
        val pos1 = [decode-error' node |> #1]
        val data = (pos1, serial (), null (C-Env-Ext.get-scopes env))
    in ((), env |> C-Env-Ext.map-idents (Symtab.delete-safe name)
                |> C-Env-Ext.map-tyidents-typedef (Symtab.update (name, data))
                |> C-Env-Ext.map-reports-text
                     (markup-tvar
                       (Left (data, flat [ look-idents env name, look-tyidents-typedef env name ]))
                       pos1
                       name))
    end
  fun shadowTypedef0''' name pos data0 env-lang env-tree =
    let val data = (pos, serial (), data0)
        val update-id = Symtab.update (name, data)
    in ( env-lang |> C-Env-Ext.map-tyidents'-typedef (Symtab.delete-safe name)
                  |> C-Env-Ext.map-idents' update-id
       , update-id
       , env-tree
           |> C-Env.map-reports-text
                (markup-var (Left (data, flat [ look-idents' env-lang name
                                              , look-tyidents'-typedef env-lang name ]))
```

```
                        pos
                        name))
    end
  fun shadowTypedef0'''' name pos data0 env-lang env-tree =
    let val (env-lang, -, env-tree) = shadowTypedef0''' name pos data0 env-lang env-tree
    in ( env-lang, env-tree) end
  fun shadowTypedef0'' ret global (Ident0 (-, i, node), params) =
    shadowTypedef0''' (ident-decode i)
                      [decode-error' node |> #1]
                      {global = global, params = params, ret = ret}
  fun shadowTypedef0' ret global ident env-lang env-tree =
    let val (env-lang, -, env-tree) = shadowTypedef0'' ret global ident env-lang env-tree
    in (env-lang, env-tree) end
  fun shadowTypedef0 ret global f ident env =
    let val (update-id, env) =
          C-Env.map-env-lang-tree'
            (fn env-lang => fn env-tree =>
              let val (env-lang, update-id, env-tree) =
                        shadowTypedef0'' ret global ident env-lang env-tree
              in (update-id, (env-lang, env-tree)) end)
            env
    in ((), f update-id env) end
  fun shadowTypedef-fun ident env =
    shadowTypedef0 C-Env.Previous-in-stack
                   (case C-Env-Ext.get-scopes env of - :: [] => true | - => false)
                   (fn update-id =>
                    C-Env-Ext.map-scopes
                     (fn (NONE, x) :: xs => (SOME (fst ident), C-Env.map-idents update-id x) ::
xs
                       | (SOME -, -) :: - => error Not yet implemented
                       | [] => error Not expecting an empty scope))
                   ident
                   env
  fun shadowTypedef (i, params, ret) env =
    shadowTypedef0 (C-Env.Parsed ret) (List.null (C-Env-Ext.get-scopes env)) (K I) (i, params)
env
  fun isTypeIdent s0 = Symtab.exists (fn (s1, -) => s0 = s1) o C-Env-Ext.get-tyidents-typedef
  fun enterScope env =
    ((), C-Env-Ext.map-scopes (cons (NONE, C-Env-Ext.get-var-table env)) env)
  fun leaveScope env =
    case C-Env-Ext.get-scopes env of
      [] => error leaveScope: already in global scope
    | (-, var-table) :: scopes => ((), env |> C-Env-Ext.map-scopes (K scopes)
                                           |> C-Env-Ext.map-var-table (K var-table))
  val getCurrentPosition = return NoPosition

  (* Language.C.Parser.Tokens *)
  fun CTokCLit x f = x |> f
  fun CTokILit x f = x |> f
```

```
  fun CTokFLit x f = x |> f
  fun CTokSLit x f = x |> f

  (* Language.C.Parser.Parser *)
  fun reverseList x = rev x
  fun L a i = posOf''' i #>> curry Located a
  fun unL (Located (a, -)) = a
  fun withNodeInfo00 (pos1, (pos2, len2)) mkAttrNode name =
    return (mkAttrNode (NodeInfo pos1 (pos2, len2) name))
  fun withNodeInfo0 x = x |> bind getNewName oo withNodeInfo00
  fun withNodeInfo0' node mkAttrNode env = let val (range, env) = posOf'' node env
                                           in withNodeInfo0 range mkAttrNode env end
  fun withNodeInfo x = x |> withNodeInfo0' o Left
  fun withNodeInfo' x = x |> withNodeInfo0' o decode-error
  fun withNodeInfo-CExtDecl x = x |>
    withNodeInfo' o (fn CDeclExt0 (CDecl0 (-, -, node)) => node
                      | CDeclExt0 (CStaticAssert0 (-, -, node)) => node
                      | CFDefExt0 (CFunDef0 (-, -, -, -, node)) => node
                      | CAsmExt0 (-, node) => node)
  val get-node-CExpr =
    fn CComma0 (-, a) => a | CAssign0 (-, -, -, a) => a | CCond0 (-, -, -, a) => a |
    CBinary0 (-, -, -, a) => a | CCast0 (-, -, a) => a | CUnary0 (-, -, a) => a |
    CSizeofExpr0 (-, a) => a | CSizeofType0 (-, a) => a | CAlignofExpr0 (-, a) => a |
    CAlignofType0 (-, a) => a | CComplexReal0 (-, a) => a | CComplexImag0 (-, a) => a |
    CIndex0 (-, -, a) => a |
    CCall0 (-, -, a) => a | CMember0 (-, -, -, a) => a | CVar0 (-, a) => a | CConst0 c =>
(case c of
    CIntConst0 (-, a) => a | CCharConst0 (-, a) => a | CFloatConst0 (-, a) => a |
    CStrConst0 (-, a) => a) |
    CCompoundLit0 (-, -, a) => a | CGenericSelection0 (-, -, a) => a | CStatExpr0 (-, a) =>
a |
    CLabAddrExpr0 (-, a) => a | CBuiltinExpr0 cBuiltinThing => (case cBuiltinThing
     of CBuiltinVaArg0 (-, -, a) => a
     | CBuiltinOffsetOf0 (-, -, a) => a
     | CBuiltinTypesCompatible0 (-, -, a) => a)
  fun withNodeInfo-CExpr x = x |> withNodeInfo' o get-node-CExpr o hd
  fun withLength node mkAttrNode =
    bind (posOf'' (decode-error node)) (fn range =>
     withNodeInfo00 range mkAttrNode (case nameOfNode node of NONE => error nameOfNode
                                                            | SOME name => name))
  fun reverseDeclr (CDeclrR0 (ide, reversedDDs, asmname, cattrs, at)) =
                  CDeclr ide (rev reversedDDs) asmname cattrs at
  fun appendDeclrAttrs newAttrs (CDeclrR0 (ident, l, asmname, cattrs, at)) =
    case l of
      [] => CDeclrR ident empty asmname (cattrs @ newAttrs) at
    | x :: xs =>
      let
        val appendAttrs =
          fn CPtrDeclr0 (typeQuals, at) => CPtrDeclr (typeQuals @ map CAttrQual newAttrs)
```

```
at
          | CArrDeclr0 (typeQuals, arraySize, at) => CArrDeclr (typeQuals @ map CAttrQual
newAttrs)
                                                     arraySize
                                                     at
          | CFunDeclr0 (parameters, cattrs, at) => CFunDeclr parameters (cattrs @ newAttrs)
at
      in CDeclrR ident (appendAttrs x :: xs) asmname cattrs at end
  fun withAttribute node cattrs mkDeclrNode =
    bind (posOf''' node) (fn (pos, -) =>
    bind getNewName (fn name =>
        let val attrs = mkNodeInfo pos name
            val newDeclr = appendDeclrAttrs cattrs (mkDeclrNode attrs)
        in return newDeclr end))
  fun withAttributePF node cattrs mkDeclrCtor =
    bind (posOf''' node) (fn (pos, -) =>
    bind getNewName (fn name =>
        let val attrs = mkNodeInfo pos name
            val newDeclr = appendDeclrAttrs cattrs o mkDeclrCtor attrs
        in return newDeclr end))
  fun appendObjAttrs newAttrs (CDeclr0 (ident, indirections, asmname, cAttrs, at)) =
    CDeclr ident indirections asmname (cAttrs @ newAttrs) at
  fun appendObjAttrsR newAttrs (CDeclrR0 (ident, indirections, asmname, cAttrs, at)) =
    CDeclrR ident indirections asmname (cAttrs @ newAttrs) at
  fun setAsmName mAsmName (CDeclrR0 (ident, indirections, oldName, cattrs, at)) =
    case (case (mAsmName, oldName)
          of (None, None) => Right None
           | (None, oldname as Some -) => Right oldname
           | (newname as Some -, None) => Right newname
           | (Some n1, Some n2) => Left (n1, n2))
    of
      Left (n1, n2) => let fun showName (CStrLit0 (CString0 (s, -), -)) = To-string0 s
                       in error (Duplicate assembler name: ^showName n1 ^ ^showName n2) end
    | Right newName => return (CDeclrR ident indirections newName cattrs at)
  fun withAsmNameAttrs (mAsmName, newAttrs) declr =
        setAsmName mAsmName (appendObjAttrsR newAttrs declr)
  fun ptrDeclr (CDeclrR0 (ident, derivedDeclrs, asmname, cattrs, dat)) tyquals at =
    CDeclrR ident (snoc derivedDeclrs (CPtrDeclr tyquals at)) asmname cattrs dat
  fun funDeclr (CDeclrR0 (ident, derivedDeclrs, asmname, dcattrs, dat)) params cattrs at =
    CDeclrR ident (snoc derivedDeclrs (CFunDeclr params cattrs at)) asmname dcattrs dat
  fun arrDeclr (CDeclrR0 (ident, derivedDeclrs, asmname, cattrs, dat))
               tyquals
               var-sized
               static-size
               size-expr-opt
               at =
    CDeclrR ident
            (snoc
               derivedDeclrs
```

```
            (CArrDeclr tyquals (case size-expr-opt of
                                  Some e => CArrSize static-size e
                                | None => CNoArrSize var-sized) at))
          asmname
          cattrs
          dat
  val liftTypeQuals = map CTypeQual o reverse
  val liftCAttrs = map (CTypeQual o CAttrQual)
  fun addTrailingAttrs declspecs new-attrs =
    case viewr declspecs of
      (specs-init, CTypeSpec0 (CSUType0 (CStruct0 (tag, name, Some def, def-attrs, su-node),
node)))
      =>
        snoc
          specs-init
          (CTypeSpec (CSUType (CStruct tag name (Just def) (def-attrs @ new-attrs) su-node)
node))
    | (specs-init, CTypeSpec0 (CEnumType0 (CEnum0 (name, Some def, def-attrs, e-node),
node))) =>
        snoc
          specs-init
            (CTypeSpec (CEnumType (CEnum name (Just def) (def-attrs @ new-attrs) e-node)
node))
    | - => rappend declspecs (liftCAttrs new-attrs)
  val emptyDeclr = CDeclrR Nothing empty Nothing [] undefNode
  fun mkVarDeclr ident = CDeclrR (Some ident) empty Nothing []
  fun doDeclIdent declspecs (decl as CDeclrR0 (mIdent, -, -, -, -)) =
    case mIdent of
      None => return ()
    | Some ident =>
        if exists (fn CStorageSpec0 (CTypedef0 -) => true | - => false) declspecs
        then addTypedef ident
        else shadowTypedef ( ident
                           , case reverseDeclr decl of CDeclr0 (-, params, -, -, -) => params
                           , declspecs)

  val ident-of-decl =
    fn Left params => map (fn i => (i, [], [])) params
     | Right (params, -) =>
         maps (fn CDecl0 (ret, l, -) =>
                 maps (fn ((Some (CDeclr0 (Some mIdent, params, -, -, -)),-),-) =>
                           [(mIdent, params, ret)]
                        | - => [])
                      l
             | - => [])
           params

  local
  fun sequence-' f = sequence- f o ident-of-decl
```

```
val is-fun = fn CFunDeclr0 - => true | - => false
in
fun doFuncParamDeclIdent (CDeclr0 (mIdent0, param0, -, -, node0)) =
  let
    val (param-not-fun, param0') = chop-prefix (not o is-fun) param0
    val () =
      if null param-not-fun then ()
      else
        Output.information
          (Not a function
           ^ Position.here
             (decode-error' (case mIdent0 of None => node0
                                           | Some (Ident0 (-, -, node)) => node) |> #1))
    val (param-fun, param0') = chop-prefix is-fun param0'
  in
    (case mIdent0 of None => return ()
                   | Some mIdent0 => shadowTypedef-fun (mIdent0, param0))
    >>
    sequence- shadowTypedef
            (maps (fn CFunDeclr0 (params, -, -) => ident-of-decl params | - => []) param-fun)
    >>
    sequence-
      (fn CFunDeclr0 (params, -, -) =>
          C-Env.map-env-tree
            (pair Symtab.empty
             #> sequence-'
                (fn (Ident0 (-, i, node), params, ret) => fn (env-lang, env-tree) => pair ()
                  let
                    val name = ident-decode i
                    val pos = [decode-error' node |> #1]
                    val data = ( pos
                               , serial ()
                               , {global = false, params = params, ret = C-Env.Parsed ret})
                  in
                    ( env-lang |> Symtab.update (name, data)
                    , env-tree
                        |> C-Env.map-reports-text
                             (markup-var-improper
                               (Left (data, C-Env-Ext.list-lookup env-lang name))
                               pos
                               name))
                  end)
                params
             #> #2 o #2)
           #> pair ()
       | - => return ())
      param0'
  end
end
```

```
  (**)
  structure List = struct val reverse = rev end
end
›
```

3.5.2 Miscellaneous

ML — `~~/src/Pure/Thy/document_antiquotations.ML`

```
‹
structure ML-Document-Antiquotations =
struct

(* ML text *)

local

fun ml-text name ml =
    Document-Output.antiquotation-raw-embedded name (Scan.lift Parse.embedded-input —
TODO: enable reporting with Token.file as in Resources.parse_files)
    (fn ctxt => fn text =>
      let val file-content =
            Token.file-source
              (Resources.read-file
                (Resources.master-directory (Proof-Context.theory-of ctxt))
                (Path.explode (#1 (Input.source-content text)), Position.none))
          val - = (*TODO: avoid multiple file scanning*)
            ML-Context.eval-in (SOME ctxt) ML-Compiler.flags Position.none (* ← (optionally)
                                                                              disabling a potential
                                                                              double report*)
                                                                            (ml file-content)
      in file-content
        |> Input.source-explode
        |> Source.of-list
        |> Source.source
             Symbol-Pos.stopper
               (Scan.bulk (Symbol-Pos.scan-comment >> (C-Scan.Left o pair true)
                           || Scan.many1 (Symbol.is-ascii-blank o Symbol-Pos.symbol)
                                >> (C-Scan.Left o pair false)
                           || Scan.one (not o Symbol-Pos.is-eof) >> C-Scan.Right))
        |> Source.exhaust
        |> drop-prefix (fn C-Scan.Left - => true | - => false)
        |> drop-suffix (fn C-Scan.Left (false, -) => true | - => false)
        |> maps (fn C-Scan.Left (-, x) => x | C-Scan.Right x => [x])
        |> Symbol-Pos.implode
        |> enclose \n \n
        |> cartouche
        |> Document-Output.output-source ctxt
```

```
      |> Document-Output.isabelle ctxt
    end);

fun ml-enclose bg en source =
  ML-Lex.read bg @ ML-Lex.read-source source @ ML-Lex.read en;

in

val - = Theory.setup (ml-text binding‹ML-file› (ml-enclose  ));

end;

end;
›
```

3.5.3 Loading the Grammar Simulator

The parser consists of a generic module `../../src_ext/mlton/lib/mlyacc-lib/base.sig`, which interprets an automata-like format generated from ML-Yacc.

ML-file *../../src-ext/mlton/lib/mlyacc−lib/base.sig* — ‹

```
signature STREAM1 =
 sig type ('xa, 'xb) stream
     val streamify : ((('stack * 'stack-ml * 'stack-pos * 'stack-tree) * 'arg) -> 'a * (('stack *
'stack-ml * 'stack-pos * 'stack-tree) * 'arg)) -> 'arg -> ('a, (('stack * 'stack-ml * 'stack-pos *
'stack-tree) * 'arg)) stream * 'arg
     val cons : 'a * (('a, 'b) stream * 'b) -> ('a, 'b) stream * 'b
     val get : ('a, 'b) stream * 'b -> 'a * (('a, 'b) stream * 'b)
 end

signature STREAM2 =
 sig type 'xa stream
     val streamify : (unit -> 'a) -> 'a stream
     val cons : 'a * 'a stream -> 'a stream
     val get : 'a stream -> 'a * 'a stream
 end

(* LR-TABLE: signature for an LR Table.

   The list of actions and gotos passed to mkLrTable must be ordered by state
   number. The values for state 0 are the first in the list, the values for
    state 1 are next, etc.
*)

signature LR-TABLE =
    sig
        datatype ('a,'b) pairlist = EMPTY | PAIR of 'a * 'b * ('a,'b) pairlist
        datatype state = STATE of int
        datatype term = T of int
        datatype nonterm = NT of int
```

```
    datatype action = SHIFT of state
                    | REDUCE of int
                    | ACCEPT
                    | ERROR
    type table

    val numStates : table -> int
    val numRules : table -> int
    val describeActions : table -> state ->
                          (term,action) pairlist * action
    val describeGoto : table -> state -> (nonterm,state) pairlist
    val action : table -> state * term -> action
    val goto : table -> state * nonterm -> state
    val initialState : table -> state
    exception Goto of state * nonterm

    val mkLrTable : {actions : ((term,action) pairlist * action) array,
                     gotos : (nonterm,state) pairlist array,
                     numStates : int, numRules : int,
                     initialState : state} -> table
end

(* TOKEN: signature revealing the internal structure of a token. This signature
   TOKEN distinct from the signature {parser name}-TOKENS produced by ML-Yacc.
   The {parser name}-TOKENS structures contain some types and functions to
    construct tokens from values and positions.

   The representation of token was very carefully chosen here to allow the
   polymorphic parser to work without knowing the types of semantic values
   or line numbers.

   This has had an impact on the TOKENS structure produced by SML-Yacc, which
   is a structure parameter to lexer functors.  We would like to have some
   type 'a token which functions to construct tokens would create.  A
   constructor function for a integer token might be

        INT: int * 'a * 'a -> 'a token.

   This is not possible because we need to have tokens with the representation
   given below for the polymorphic parser.

   Thus our constructur functions for tokens have the form:

        INT: int * 'a * 'a -> (svalue,'a) token

   This in turn has had an impact on the signature that lexers for SML-Yacc
   must match and the types that a user must declare in the user declarations
   section of lexers.
*)
```

```
signature TOKEN =
    sig
        structure LALR-Table : LR-TABLE
        datatype ('a,'b) token = TOKEN of LALR-Table.term * ('a * 'b * 'b)
        val sameToken : ('a,'b) token * ('a,'b) token -> bool
    end

(* LR-PARSER: signature for a polymorphic LR parser *)

signature LR-PARSER1 =
    sig
        structure Stream : STREAM1
        structure LALR-Table : LR-TABLE
        structure Token : TOKEN

        sharing LALR-Table = Token.LALR-Table

        type ('-b, '-c) stack = (LALR-Table.state, '-b, '-c) C-Env.stack'

       type ('-b, '-c, 'arg1, 'arg2) lexer = (('arg1 -> '-b * 'arg1,'-c) Token.token, ('-b, '-c) stack
* 'arg1) Stream.stream * 'arg2

        val parse : {table : LALR-Table.table,
                     saction : int *
                               '-c *
                               (LALR-Table.state * ('-b * '-c * '-c)) list *
                               'arg
                               ->    LALR-Table.nonterm *
                                     (('arg -> '-b * 'arg) * '-c * '-c) *
                                     (LALR-Table.state * ('-b * '-c * '-c)) list,
                     void : 'arg -> '-b * 'arg,
                     void-position : '-c,
                     start : ('arg -> '-b * 'arg, '-c) Token.token,
                     accept : '-c * '-c -> ('-b, '-c) stack * 'arg -> 'user * 'arg,
                     reduce-init : (('-c * '-c) list * int) * 'arg -> 'arg,
                         reduce-get : (LALR-Table.state, '-b, '-c) C-Env.rule-reduce -> 'arg ->
(LALR-Table.state, '-b, '-c) C-Env.rule-output0 * 'arg,
                     ec : { is-keyword : LALR-Table.term -> bool,
                            noShift : LALR-Table.term -> bool,
                            preferred-change : (LALR-Table.term list * LALR-Table.term list) list,
                            errtermvalue : LALR-Table.term -> 'arg -> '-b * 'arg,
                            showTerminal : LALR-Table.term -> string,
                            terms: LALR-Table.term list,
                            error : '-c * '-c -> ('-b, '-c) stack * 'arg -> 'user * 'arg
                           },
                     lookahead : int  (* max amount of lookahead used in *)
                                 (* error correction *)
                    }
```

```
                -> ('-b, '-c, 'arg, 'arg) lexer
                -> ('-b, '-c, 'arg, 'user * 'arg) lexer
    end

signature LR-PARSER2 =
    sig
        structure Stream : STREAM2
        structure LALR-Table : LR-TABLE
        structure Token : TOKEN

        sharing LALR-Table = Token.LALR-Table

        exception ParseError

        val parse : {table : LALR-Table.table,
                     lexer : ('-b,'-c) Token.token Stream.stream,
                     arg: 'arg,
                     saction : int *
                               '-c *
                               (LALR-Table.state * ('-b * '-c * '-c)) list *
                               'arg
                               ->    LALR-Table.nonterm *
                                     ('-b * '-c * '-c) *
                                     (LALR-Table.state * ('-b * '-c * '-c)) list,
                     void : '-b,
                     ec : { is-keyword : LALR-Table.term -> bool,
                            noShift : LALR-Table.term -> bool,
                            preferred-change : (LALR-Table.term list * LALR-Table.term list) list,
                            errtermvalue : LALR-Table.term -> '-b,
                            showTerminal : LALR-Table.term -> string,
                            terms: LALR-Table.term list,
                            error : string * '-c * '-c -> unit
                          },
                     lookahead : int  (* max amount of lookahead used in *)
                                      (* error correction *)
                    }
                    -> '-b * ('-b,'-c) Token.token Stream.stream
    end

(* LEXER: a signature that most lexers produced for use with SML-Yacc's
   output will match.  The user is responsible for declaring type token,
   type pos, and type svalue in the LALR-Lex-Instance section of a lexer.

   Note that type token is abstract in the lexer.  This allows SML-Yacc to
   create a TOKENS signature for use with lexers produced by ML-Lex that
   treats the type token abstractly.  Lexers that are functors parametrized by
   a Tokens structure matching a TOKENS signature cannot examine the structure
   of tokens.
*)
```

```
signature LEXER =
  sig
    structure LALR-Lex-Instance :
      sig
        type ('a,'b) token
        type pos
        type svalue
      end
     val makeLexer : (int -> string)
                  -> unit
                                  -> (LALR-Lex-Instance.svalue, LALR-Lex-Instance.pos)
LALR-Lex-Instance.token
  end

(* ARG-LEXER: the %arg option of ML-Lex allows users to produce lexers which
   also take an argument before yielding a function from unit to a token
*)

signature ARG-LEXER1 =
  sig
    structure LALR-Lex-Instance :
      sig
        type ('a,'b) token
        type pos
        type arg
        type svalue0
        type svalue = arg -> svalue0 * arg
        type state
      end
                  type stack = (LALR-Lex-Instance.state, LALR-Lex-Instance.svalue0,
LALR-Lex-Instance.pos) C-Env.stack'
     val makeLexer : (stack * LALR-Lex-Instance.arg)
                                  -> (LALR-Lex-Instance.svalue, LALR-Lex-Instance.pos)
LALR-Lex-Instance.token
                   * (stack * LALR-Lex-Instance.arg)
  end

signature ARG-LEXER2 =
  sig
    structure LALR-Lex-Instance :
      sig
        type ('a,'b) token
        type pos
        type arg
        type svalue
      end
     val makeLexer : (int -> string)
                  -> LALR-Lex-Instance.arg
```

```
                    -> unit
                                        -> (LALR-Lex-Instance.svalue,LALR-Lex-Instance.pos)
LALR-Lex-Instance.token
  end

(* PARSER-DATA: the signature of ParserData structures in {parser name}LrValsFun
  produced by SML-Yacc. All such structures match this signature.

  The {parser name}LrValsFun produces a structure which contains all the values
  except for the lexer needed to call the polymorphic parser mentioned
  before.

*)

signature PARSER-DATA1 =
  sig
      (* the type of line numbers *)
      type pos

      (* the type of the user-supplied argument to the parser *)
      type arg

      (* the type of semantic values *)
      type svalue0
      type svalue = arg -> svalue0 * arg

      (* the intended type of the result of the parser. This value is
        produced by applying extract from the structure Actions to the
        final semantic value resultiing from a parse.
       *)
      type result

      structure LALR-Table : LR-TABLE
      structure Token : TOKEN
      sharing Token.LALR-Table = LALR-Table

      (* structure Actions contains the functions which mantain the
        semantic values stack in the parser. Void is used to provide
        a default value for the semantic stack.
       *)
      structure Actions :
        sig
          val actions : int * pos * (LALR-Table.state * (svalue0 * pos * pos)) list * arg
                      -> LALR-Table.nonterm * (svalue * pos * pos) * (LALR-Table.state *
(svalue0 * pos * pos)) list
          val void : svalue
          val extract : svalue0 -> result
        end
```

```
    (* structure EC contains information used to improve error
       recovery in an error−correcting parser *)
    structure EC :
      sig
        val is-keyword : LALR-Table.term -> bool
        val noShift : LALR-Table.term -> bool
        val preferred-change : (LALR-Table.term list * LALR-Table.term list) list
        val errtermvalue : LALR-Table.term -> svalue
        val showTerminal : LALR-Table.term -> string
        val terms: LALR-Table.term list
      end

    (* table is the LR table for the parser *)
    val table : LALR-Table.table
  end

signature PARSER-DATA2 =
  sig
    (* the type of line numbers *)
    type pos

    (* the type of the user−supplied argument to the parser *)
    type arg

    (* the type of semantic values *)
    type svalue

    (* the intended type of the result of the parser.  This value is
       produced by applying extract from the structure Actions to the
       final semantic value resultiing from a parse.
     *)
    type result

    structure LALR-Table : LR-TABLE
    structure Token : TOKEN
    sharing Token.LALR-Table = LALR-Table

    (* structure Actions contains the functions which mantain the
       semantic values stack in the parser.  Void is used to provide
       a default value for the semantic stack.
     *)
    structure Actions :
      sig
        val actions : int * pos * (LALR-Table.state * (svalue * pos * pos)) list * arg
                      -> LALR-Table.nonterm * (svalue * pos * pos) * (LALR-Table.state *
(svalue * pos * pos)) list
        val void : svalue
        val extract : svalue -> result
      end
```

```
    (* structure EC contains information used to improve error
       recovery in an error−correcting parser *)
    structure EC :
      sig
        val is-keyword : LALR-Table.term -> bool
        val noShift : LALR-Table.term -> bool
        val preferred-change : (LALR-Table.term list * LALR-Table.term list) list
        val errtermvalue : LALR-Table.term -> svalue
        val showTerminal : LALR-Table.term -> string
        val terms: LALR-Table.term list
      end

    (* table is the LR table for the parser *)
    val table : LALR-Table.table
  end

(* signature PARSER is the signature that most user parsers created by
   SML−Yacc will match.
*)

signature PARSER2 =
  sig
    structure Token : TOKEN
    structure Stream : STREAM2
    exception ParseError

    (* type pos is the type of line numbers *)
    type pos

    (* type result is the type of the result from the parser *)
    type result

    (* the type of the user−supplied argument to the parser *)
    type arg

    (* type svalue is the type of semantic values for the semantic value
       stack
     *)
    type svalue

    (* val makeLexer is used to create a stream of tokens for the parser *)
    val makeLexer : (int -> string)
                    -> (svalue, pos) Token.token Stream.stream

    (* val parse takes a stream of tokens and a function to print
       errors and returns a value of type result and a stream containing
       the unused tokens
     *)
```

```
      val parse : int * ((svalue, pos) Token.token Stream.stream) * (string * pos * pos -> unit)
* arg
                -> result * (svalue, pos) Token.token Stream.stream

      val sameToken : (svalue, pos) Token.token * (svalue,pos) Token.token
                   -> bool
    end

(* signature ARG-PARSER is the signature that will be matched by parsers whose
   lexer takes an additional argument.
*)

signature ARG-PARSER1 =
   sig
      structure Token : TOKEN
      structure Stream : STREAM1

      type arg
      type pos
      type svalue0
      type svalue = arg -> svalue0 * arg

      type stack = (Token.LALR-Table.state, svalue0, pos) C-Env.stack'

       type ('arg1, 'arg2) lexer = ((svalue, pos) Token.token, stack * 'arg1) Stream.stream *
'arg2

      val makeLexer : arg -> (arg, arg) lexer
      val parse :   int
                  * (pos * pos -> stack * arg -> 'user * arg)
                  * pos
                  * (svalue, pos) Token.token
                  * (pos * pos -> stack * arg -> 'user * arg)
                  * (((pos * pos) list * int) * arg -> arg)
                      * ((Token.LALR-Table.state, svalue0, pos) C-Env.rule-reduce -> arg ->
(Token.LALR-Table.state, svalue0, pos) C-Env.rule-output0 * arg)
                -> (arg, arg) lexer
                -> (arg, 'user * arg) lexer
      val sameToken : (svalue, pos) Token.token * (svalue, pos) Token.token -> bool
    end

signature ARG-PARSER2 =
   sig
      structure Token : TOKEN
      structure Stream : STREAM2
      exception ParseError

      type arg
      type pos
```

```
        type result
        type svalue

        val makeLexer : (int -> string) -> arg
                      -> (svalue, pos) Token.token Stream.stream
       val parse : int * ((svalue, pos) Token.token Stream.stream) * (string * pos * pos -> unit)
* arg
                    -> result * (svalue, pos) Token.token Stream.stream
        val sameToken : (svalue, pos) Token.token * (svalue,pos) Token.token
                      -> bool
    end
›
ML-file ../../src-ext/mlton/lib/mlyacc-lib/join.sml — ‹
functor Join2 (structure Lex : LEXER
               structure ParserData: PARSER-DATA2
               structure LrParser : LR-PARSER2
               sharing ParserData.LALR-Table = LrParser.LALR-Table
               sharing ParserData.Token = LrParser.Token
               sharing type Lex.LALR-Lex-Instance.svalue = ParserData.svalue
               sharing type Lex.LALR-Lex-Instance.pos = ParserData.pos
               sharing type Lex.LALR-Lex-Instance.token = ParserData.Token.token)
                : PARSER2 =
struct
    structure Token = ParserData.Token
    structure Stream = LrParser.Stream

    exception ParseError = LrParser.ParseError

    type arg = ParserData.arg
    type pos = ParserData.pos
    type result = ParserData.result
    type svalue = ParserData.svalue
    val makeLexer = LrParser.Stream.streamify o Lex.makeLexer
    val parse = fn (lookahead,lexer,error,arg) =>
       (fn (a,b) => (ParserData.Actions.extract a,b))
     (LrParser.parse {table = ParserData.table,
               lexer=lexer,
               lookahead=lookahead,
               saction = ParserData.Actions.actions,
               arg=arg,
               void= ParserData.Actions.void,
               ec = {is-keyword = ParserData.EC.is-keyword,
                     noShift = ParserData.EC.noShift,
                     preferred-change = ParserData.EC.preferred-change,
                     errtermvalue = ParserData.EC.errtermvalue,
                     error=error,
                     showTerminal = ParserData.EC.showTerminal,
                     terms = ParserData.EC.terms}}
     )
```

```
    val sameToken = Token.sameToken
end

(* functor JoinWithArg creates a variant of the parser structure produced
   above. In this case, the makeLexer take an additional argument before
   yielding a value of type unit -> (svalue,pos) token
 *)

functor LALR-Parser-Join(structure Lex : ARG-LEXER1
           structure ParserData: PARSER-DATA1
           structure LrParser : LR-PARSER1
           sharing ParserData.LALR-Table = LrParser.LALR-Table
           sharing ParserData.Token = LrParser.Token
           sharing type Lex.LALR-Lex-Instance.arg = ParserData.arg
           sharing type Lex.LALR-Lex-Instance.svalue0 = ParserData.svalue0
           sharing type Lex.LALR-Lex-Instance.pos = ParserData.pos
           sharing type Lex.LALR-Lex-Instance.token = ParserData.Token.token
           sharing type Lex.LALR-Lex-Instance.state = ParserData.Token.LALR-Table.state)
              : ARG-PARSER1 =
struct
    structure Token = ParserData.Token
    structure Stream = LrParser.Stream

    type arg = ParserData.arg
    type pos = ParserData.pos
    type svalue0 = ParserData.svalue0
    type svalue = arg -> svalue0 * arg

    type stack = (Token.LALR-Table.state, svalue0, pos) C-Env.stack'

    type ('arg1, 'arg2) lexer = ((svalue, pos) Token.token, stack * 'arg1) Stream.stream * 'arg2

    val makeLexer = LrParser.Stream.streamify Lex.makeLexer

    val parse = fn (lookahead, error, void-position, start, accept, reduce-init, reduce-get) =>
      LrParser.parse {table = ParserData.table,
                      lookahead = lookahead,
                      saction = ParserData.Actions.actions,
                      void = ParserData.Actions.void,
                      void-position = void-position,
                      start = start,
                      accept = accept,
                      reduce-init = reduce-init,
                      reduce-get = reduce-get,
                      ec = {is-keyword = ParserData.EC.is-keyword,
                            noShift = ParserData.EC.noShift,
                            preferred-change = ParserData.EC.preferred-change,
                            errtermvalue = ParserData.EC.errtermvalue,
                            error=error,
```

```
                        showTerminal = ParserData.EC.showTerminal,
                        terms = ParserData.EC.terms}}

    val sameToken = Token.sameToken
end

functor JoinWithArg2(structure Lex : ARG-LEXER2
             structure ParserData: PARSER-DATA2
             structure LrParser : LR-PARSER2
             sharing ParserData.LALR-Table = LrParser.LALR-Table
             sharing ParserData.Token = LrParser.Token
             sharing type Lex.LALR-Lex-Instance.arg = ParserData.arg
             sharing type Lex.LALR-Lex-Instance.svalue = ParserData.svalue
             sharing type Lex.LALR-Lex-Instance.pos = ParserData.pos
             sharing type Lex.LALR-Lex-Instance.token = ParserData.Token.token)
                : ARG-PARSER2 =
struct
    structure Token = ParserData.Token
    structure Stream = LrParser.Stream

    exception ParseError = LrParser.ParseError

    type arg = ParserData.arg
    type pos = ParserData.pos
    type result = ParserData.result
    type svalue = ParserData.svalue

    val makeLexer = LrParser.Stream.streamify oo Lex.makeLexer
    val parse = fn (lookahead,lexer,error,arg) =>
       (fn (a,b) => (ParserData.Actions.extract a,b))
     (LrParser.parse {table = ParserData.table,
              lexer=lexer,
              lookahead=lookahead,
              saction = ParserData.Actions.actions,
              arg=arg,
              void= ParserData.Actions.void,
              ec = {is-keyword = ParserData.EC.is-keyword,
                    noShift = ParserData.EC.noShift,
                    preferred-change = ParserData.EC.preferred-change,
                    errtermvalue = ParserData.EC.errtermvalue,
                    error=error,
                    showTerminal = ParserData.EC.showTerminal,
                    terms = ParserData.EC.terms}}
     )
    val sameToken = Token.sameToken
end;
›
ML-file ../../src-ext/mlton/lib/mlyacc-lib/lrtable.sml — ‹
structure LALR-Table : LR-TABLE =
```

```
struct
    val sub = Array.sub
    infix 9 sub
    datatype ('a,'b) pairlist = EMPTY
                              | PAIR of 'a * 'b * ('a,'b) pairlist
    datatype term = T of int
    datatype nonterm = NT of int
    datatype state = STATE of int
    datatype action = SHIFT of state
                    | REDUCE of int (* rulenum from grammar *)
                    | ACCEPT
                    | ERROR
    exception Goto of state * nonterm
    type table = {states: int, rules : int,initialState: state,
                  action: ((term,action) pairlist * action) array,
                  goto :  (nonterm,state) pairlist array}
    val numStates = fn ({states,...} : table) => states
    val numRules = fn ({rules,...} : table) => rules
    val describeActions =
      fn ({action,...} : table) =>
            fn (STATE s) => action sub s
    val describeGoto =
      fn ({goto,...} : table) =>
            fn (STATE s) => goto sub s
    fun findTerm (T term,row,default) =
        let fun find (PAIR (T key,data,r)) =
                  if key < term then find r
                  else if key=term then data
                  else default
              | find EMPTY = default
        in find row
        end
    fun findNonterm (NT nt,row) =
        let fun find (PAIR (NT key,data,r)) =
                  if key < nt then find r
                  else if key=nt then SOME data
                  else NONE
              | find EMPTY = NONE
        in find row
        end
    val action = fn ({action,...} : table) =>
          fn (STATE state,term) =>
            let val (row,default) = action sub state
            in findTerm(term,row,default)
            end
    val goto = fn ({goto,...} : table) =>
                  fn (a as (STATE state,nonterm)) =>
                    case findNonterm(nonterm,goto sub state)
                    of SOME state => state
```

```
                    | NONE => raise (Goto a)
        val initialState = fn ({initialState,...} : table) => initialState
        val mkLrTable = fn {actions,gotos,initialState,numStates,numRules} =>
            ({action=actions,goto=gotos,
              states=numStates,
              rules=numRules,
              initialState=initialState} : table)
end;
›
```

ML-file *../../src-ext/mlton/lib/mlyacc-lib/stream.sml* — ‹

```
structure Stream1 : STREAM1 =
struct
  datatype ('a, 'b) stream = Source of {buffer: 'a list, drain: 'b -> 'a * 'b}

  fun streamify drain = pair (Source {buffer = [], drain = drain})

  fun get (Source {buffer = [], drain}, info) =
        let val (x, info') = drain info
        in (x, (Source {buffer = [], drain = drain}, info')) end
    | get (Source {buffer = x :: buffer, drain}, info) =
            (x, (Source {buffer = buffer, drain = drain}, info))

  fun cons (x, (Source {buffer, drain}, info)) =
    (Source {buffer = x :: buffer, drain = drain}, info)
end;

structure Stream2 : STREAM2 =
struct
   open Unsynchronized

   datatype 'a str = EVAL of 'a * 'a str ref | UNEVAL of (unit->'a)

   type 'a stream = 'a str ref

   fun get(ref(EVAL t)) = t
     | get(s as ref(UNEVAL f)) =
            let val t = (f(), ref(UNEVAL f)) in s := EVAL t; t end

   fun streamify f = ref(UNEVAL f)
   fun cons(a,s) = ref(EVAL(a,s))

end;
›
```

ML-file *../../src-ext/mlton/lib/mlyacc-lib/parser1.sml* — ‹

```
structure LALR-Parser-Eval : LR-PARSER1 =
struct

structure LALR-Table = LALR-Table
structure Stream = Stream1
```

```
structure Token : TOKEN =
  struct
      structure LALR-Table = LALR-Table
      datatype ('a,'b) token = TOKEN of LALR-Table.term * ('a * 'b * 'b)
      val sameToken = fn (TOKEN (t,-),TOKEN(t',-)) => t=t'
  end


open LALR-Table
open Token

val DEBUG1 = false
exception ParseImpossible of int

type ('a,'b) stack0 = (state * ('a * 'b * 'b)) list

type ('-b, '-c) stack = (LALR-Table.state, '-b, '-c) C-Env.stack'

type ('-b, '-c, 'arg1, 'arg2) lexer = (('arg1 -> '-b * 'arg1,'-c) Token.token, ('-b, '-c) stack *
'arg1) Stream.stream * 'arg2

val showState = fn (STATE s) => STATE ^ Int.toString s

fun printStack(stack: ('a,'b) stack0, n: int) =
   case stack
     of (state, -) :: rest =>
          (writeln (          ^ Int.toString n ^ :  ^ showState state);
           printStack(rest, n+1)
          )
      | nil => ()

fun parse {table, saction, void, void-position, start, accept, reduce-init, reduce-get, ec =
{showTerminal, error, ...}, ...} =
  let fun empty-tree rule-pos rule-type =
        C-Env.Tree ({rule-pos = rule-pos, rule-type = rule-type}, [])

      fun prAction(stack as (state, -) :: -, (TOKEN (term,-),-), action) =
            (writeln Parse: state stack:;
             printStack(stack, 0);
             writeln(       state=
                    ^ showState state
                    ^  next=
                    ^ showTerminal term
                    ^  action=
                    ^ (case action
                         of SHIFT state => SHIFT  ^ (showState state)
                          | REDUCE i => REDUCE  ^ (Int.toString i)
                          | ERROR => ERROR
```

```
                    | ACCEPT => ACCEPT)))
      | prAction (-,-,-) = ()

    val action = LALR-Table.action table
    val goto = LALR-Table.goto table

    fun add-stack (value, stack-value) (ml, stack-ml) (pos, stack-pos) (tree, stack-tree) =
      (value :: stack-value, ml :: stack-ml, pos :: stack-pos, tree :: stack-tree)

    fun parseStep ( (token as TOKEN (terminal, (f-val,leftPos,rightPos)))
                  , (lexer, (((stack as (state,-) :: -), stack-ml, stack-pos, stack-tree), arg))) =
        let val nextAction = action (state, terminal)
            val - = if DEBUG1 then prAction(stack,(token, lexer),nextAction)
                    else ()
        in case nextAction
             of SHIFT s => (lexer, arg)
                         ||> (f-val #>> (fn value => add-stack ((s, (value, leftPos, rightPos)),
stack)
                                                              ([], stack-ml)
                                                              ((leftPos, rightPos), stack-pos)
                                                             (empty-tree (leftPos, rightPos) C-Env.Shift,
stack-tree)))
                         |> Stream.get
                         |> parseStep
              | REDUCE i =>
                (case saction (i, leftPos, stack, arg)
                  of (nonterm, (reduce-exec, p1, p2), stack' as (state, -) :: -) =>
                    let val dist = length stack - length stack'
                        val arg = reduce-init ((stack-pos, dist), arg)
                        val (value, arg) = reduce-exec arg
                        val goto0 = (goto (state, nonterm), (value, p1, p2))
                        val ((pre-ml, stack-ml), stack-pos, (l-tree, stack-tree)) =
                          ( chop dist stack-ml
                          , drop dist stack-pos
                          , chop dist stack-tree)
                           val ((ml-delayed, ml-actual, goto0'), arg) = reduce-get (i, goto0 :: stack',
pre-ml) arg
                        val pos = case #output-pos goto0' of NONE => (p1, p2) | SOME pos =>
pos
                    in ( add-stack
                           (goto0, stack')
                           (flat ml-delayed, stack-ml)
                           (pos, stack-pos)
                           ( C-Env.Tree ( { rule-pos = pos
                                          , rule-type = C-Env.Reduce (#output-env goto0', (i,
#output-vacuous goto0', ml-actual)) }
                                        , rev l-tree )
                           , stack-tree)
                       , arg) end
```

```
                | - => raise (ParseImpossible 197))
              |> (fn stack-arg => parseStep (token, (lexer, stack-arg)))
            | ERROR => (lexer, ((stack, stack-ml, stack-pos, stack-tree), arg))
                       |> Stream.cons o pair token
                       ||> error (leftPos, rightPos)
            | ACCEPT => (lexer, ((stack, stack-ml, stack-pos, stack-tree), arg))
                       |> Stream.cons o pair token
                       ||> accept (leftPos, rightPos)
        end
      | parseStep - = raise (ParseImpossible 204)
  in I
    ##> (fn arg => void arg
          |>> (fn void' => add-stack ((initialState table, (void', void-position, void-position)),
[])
                                     ([], [])
                                     ((void-position, void-position), [])
                                     (empty-tree (void-position, void-position) C-Env.Void, [])))
    #> pair start
    #> parseStep
end

end;
›
```

3.5.4 Loading the Generated Grammar (SML signature)

ML-file *../generated/c-grammar-fun.grm.sig*

ML — `../generated/c_grammar_fun.grm.sig`

```
‹
structure C-Grammar-Rule = struct
open C-Grammar-Rule

(* ast-generic is an untyped universe of (some) ast's with the specific lenses put ... get ... *)

type ast-generic = start-happy

val get-CExpr = start-happy4
val get-CStat = start-happy3
val get-CExtDecl = start-happy2
val get-CTranslUnit = start-happy1

fun put-CExpr (x : C-Grammar-Rule-Lib.CExpr) = Right (Right (Right (Left x))) :
ast-generic
fun put-CStat (x : C-Grammar-Rule-Lib.CStat) = Right (Right (Left x)) : ast-generic
fun put-CExtDecl (x : C-Grammar-Rule-Lib.CExtDecl) = Right (Left x) : ast-generic
fun put-CTranslUnit (x : C-Grammar-Rule-Lib.CTranslUnit) = Left x : ast-generic
end
```

›

3.5.5 Overloading Grammar Rules (Optional Part)

ML — `../generated/c_grammar_fun.grm.sml` ‹

```
structure C-Grammar-Rule-Wrap-Overloading = struct
open C-Grammar-Rule-Lib

fun update-env-bottom-up f x arg = ((), C-Env.map-env-lang-tree (f x) arg)
fun update-env-top-down f x =
  pair () ##> (fn arg => C-Env-Ext.map-output-env (K (SOME (f x (#env-lang arg)))) arg)

(*type variable (report bound)*)

val specifier3 : (CDeclSpec list) -> unit monad =
  update-env-bottom-up
    (fn l => fn env-lang => fn env-tree =>
      ( env-lang
      , fold
          let open C-Ast
          in fn CTypeSpec0 (CTypeDef0 (Ident0 (-, i, node), -)) =>
                let val name = ident-decode i
                    val pos1 = [decode-error' node |> #1]
                in
                  C-Env.map-reports-text
                    (markup-tvar
                    (Right (pos1, Symtab.lookup (C-Env-Ext.get-tyidents'-typedef env-lang) name))
                      pos1
                      name)
                end
              | - => I
          end
          l
          env-tree))
val declaration-specifier3 : (CDeclSpec list) -> unit monad = specifier3
val type-specifier3 : (CDeclSpec list) -> unit monad = specifier3


(*basic variable (report bound)*)

val primary-expression1 : (CExpr) -> unit monad =
  update-env-bottom-up
    (fn e => fn env-lang => fn env-tree =>
      ( env-lang
      , let open C-Ast
        in fn CVar0 (Ident0 (-, i, node), -) =>
              let val name = ident-decode i
                  val pos1 = [decode-error' node |> #1]
              in
```

```
            C-Env.map-reports-text
              (markup-var
                (Right (pos1, Symtab.lookup (C-Env-Ext.get-idents' env-lang) name))
                pos1
                name)
          end
        | - => I
    end
      e
      env-tree))


(*basic variable, parameter functions (report bound)*)

val declarator1 : (CDeclrR) -> unit monad =
  update-env-bottom-up
    (fn d => fn env-lang => fn env-tree =>
      ( env-lang
      , let open C-Ast
            fun markup markup-var params =
              pair Symtab.empty
              #> fold
                   (fn (Ident0 (-, i, node), params, ret) => fn (env-lang, env-tree) =>
                     let
                       val name = ident-decode i
                       val pos = [decode-error' node |> #1]
                       val data = ( pos
                                  , serial ()
                                  , {global = false, params = params, ret = C-Env.Parsed ret})
                     in
                       ( env-lang |> Symtab.update (name, data)
                       , env-tree
                           |> C-Env.map-reports-text
                                (markup-var (Left (data, C-Env-Ext.list-lookup env-lang name))
                                            pos
                                            name))
                     end)
                   (ident-of-decl params)
                #> #2
        in fn CDeclrR0 (-, param0, -, -, -) =>
          (case rev param0 of
             CFunDeclr0 (params, -, -) :: param0 =>
               pair param0 o markup markup-var-bound params
           | param0 => pair param0)
             #->
             fold
               (fn CFunDeclr0 (params, -, -) => markup markup-var-improper params
                 | - => I)
        end
```

```
        d
        env-tree))


(*old style syntax for functions (legacy feature)*)

val external-declaration1 : (CExtDecl) -> unit monad =
  update-env-bottom-up (fn f => fn env-lang => fn env-tree =>
    ( env-lang
    , let open C-Ast
      in fn CFDefExt0 (CFunDef0 (-, -, l, -, node)) =>
           if null l then
             I
           else
            tap (fn - => legacy-feature (Scope analysing for old function syntax not implemented
                                         ^ Position.here (decode-error' node |> #1)))
        | - => I
      end
       f
       env-tree))


(*(type) enum, struct, union (report define & report bound)*)

fun report-enum-bound i' node env-lang =
  let open C-Ast
      val name = ident-decode i'
      val pos1 = [decode-error' node |> #1]
  in
    C-Env.map-reports-text
      (markup-var-enum
       (Right (pos1, Symtab.lookup (C-Env-Ext.get-tyidents'-enum env-lang) name)) pos1 name)
  end

local
val look-tyidents'-enum = C-Env-Ext.list-lookup o C-Env-Ext.get-tyidents'-enum
val declaration : (CDecl) -> unit monad =
  update-env-bottom-up
    (fn decl => fn env-lang => fn env-tree =>
      let open C-Ast
      in
       fn CDecl0 (l, -, -) =>
          fold
            (fn CTypeSpec0 (CEnumType0 (CEnum0 (Some (Ident0 (-, i, node)), body, -, -), -))
=>
               (case body of
                  None => (fn (env-lang, env-tree) =>
                            (env-lang, report-enum-bound i node env-lang env-tree))
                | Some - =>
```

```
                    fn (env-lang, env-tree) =>
                     let val name = ident-decode i
                         val pos1 = [decode-error' node |> #1]
                         val data = (pos1, serial (), null (C-Env.get-scopes env-lang))
                     in
                       ( C-Env-Ext.map-tyidents'-enum (Symtab.update (name, data)) env-lang
                       , C-Env.map-reports-text
                           (markup-var-enum
                             (Left (data, look-tyidents'-enum env-lang name))
                             pos1
                             name)
                           env-tree)
                     end)
              | - => I)
            l
       | - => I
      end
        decl
        (env-lang, env-tree))
in
val declaration1 = declaration
val declaration2 = declaration
val declaration3 = declaration
end


(*(basic) enum, struct, union (report define)*)

local
val enumerator : ( ( Ident * CExpr Maybe ) ) -> unit monad =
  update-env-bottom-up
    (fn id => fn env-lang =>
      let open C-Ast
      in
        fn (ident as Ident0 (-, -, node), -) =>
          C-Grammar-Rule-Lib.shadowTypedef0'
            (C-Env.Parsed [CTypeSpec0 (CIntType0 node)])
            (null (C-Env.get-scopes env-lang))
            (ident, [])
            env-lang
      end
        id)
in
val enumerator1 = enumerator
val enumerator2 = enumerator
val enumerator3 = enumerator
val enumerator4 = enumerator
end
```

```
(*(type) enum, struct, union (report bound)*)

local
fun declaration-specifier env-lang =
  let open C-Ast
  in
    fold
      (fn CTypeSpec0 (CEnumType0 (CEnum0 (Some (Ident0 (-, i, node)), -, -, -), -)) =>
          report-enum-bound i node env-lang
        | - => I)
  end
in
val declaration-specifier2 : (CDeclSpec list) -> unit monad =
  update-env-bottom-up
    (fn d => fn env-lang => fn env-tree =>
      let open C-Ast
      in
      ( env-lang
      , env-tree |>
          (if exists (fn CStorageSpec0 (CTypedef0 -) => true | - => false) d then
             I
           else
             declaration-specifier env-lang d))
      end)

local
val f-definition : (CFunDef) -> unit monad =
  update-env-bottom-up
    (fn d => fn env-lang => fn env-tree =>
      ( env-lang
      , let open C-Ast
        in
         fn CFunDef0 (l, -, -, -, -) => declaration-specifier env-lang l
        end
          d
          env-tree))
in
val function-definition4 = f-definition
val nested-function-definition2 = f-definition
end

local
val parameter-type-list : ( ( CDecl list * Bool ) ) -> unit monad =
  update-env-bottom-up
    (fn d => fn env-lang => fn env-tree =>
      ( env-lang
      , let open C-Ast
        in
```

```
        #1 #> fold (fn CDecl0 (l, -, -) => declaration-specifier env-lang l | - => I)
       end
         d
         env-tree))
in
val parameter-type-list2 = parameter-type-list
val parameter-type-list3 = parameter-type-list
end
end
end
›
```

ML — `../generated/c_grammar_fun.grm.sml` ‹

```
structure C-Grammar-Rule-Wrap = struct
  open C-Grammar-Rule-Wrap
  open C-Grammar-Rule-Wrap-Overloading
end
›
```

3.5.6 Loading the Generated Grammar (SML structure)

ML-file *../generated/c-grammar-fun.grm.sml*

3.5.7 Grammar Initialization

Functor Application

ML — `../generated/c_grammar_fun.grm.sml` ‹

```
structure C-Grammar = C-Grammar-Fun (structure Token = LALR-Parser-Eval.Token)
›
```

Mapping Strings to Structured Tokens

ML — `../generated/c_grammar_fun.grm.sml` ‹

```
structure C-Grammar-Tokens =
struct
local open C-Grammar.Tokens in
  fun token-of-string
        error
        ty-ClangCVersion
        ty-cChar
        ty-cFloat
        ty-cInteger
        ty-cString
        ty-ident
        ty-string
        a1
        a2 =
    fn
      ( => x28 (ty-string, a1, a2)
```

```
| ) => x29 (ty-string, a1, a2)
| [ => x5b (ty-string, a1, a2)
| ] => x5d (ty-string, a1, a2)
| -> => x2d-x3e (ty-string, a1, a2)
| . => x2e (ty-string, a1, a2)
| ! => x21 (ty-string, a1, a2)
| ~ => x7e (ty-string, a1, a2)
| ++ => x2b-x2b (ty-string, a1, a2)
| -- => x2d-x2d (ty-string, a1, a2)
| + => x2b (ty-string, a1, a2)
| - => x2d (ty-string, a1, a2)
| * => x2a (ty-string, a1, a2)
| / => x2f (ty-string, a1, a2)
| % => x25 (ty-string, a1, a2)
| & => x26 (ty-string, a1, a2)
| << => x3c-x3c (ty-string, a1, a2)
| >> => x3e-x3e (ty-string, a1, a2)
| < => x3c (ty-string, a1, a2)
| <= => x3c-x3d (ty-string, a1, a2)
| > => x3e (ty-string, a1, a2)
| >= => x3e-x3d (ty-string, a1, a2)
| == => x3d-x3d (ty-string, a1, a2)
| != => x21-x3d (ty-string, a1, a2)
| ^ => x5e (ty-string, a1, a2)
| | => x7c (ty-string, a1, a2)
| && => x26-x26 (ty-string, a1, a2)
| || => x7c-x7c (ty-string, a1, a2)
| ? => x3f (ty-string, a1, a2)
| : => x3a (ty-string, a1, a2)
| = => x3d (ty-string, a1, a2)
| += => x2b-x3d (ty-string, a1, a2)
| -= => x2d-x3d (ty-string, a1, a2)
| *= => x2a-x3d (ty-string, a1, a2)
| /= => x2f-x3d (ty-string, a1, a2)
| %= => x25-x3d (ty-string, a1, a2)
| &= => x26-x3d (ty-string, a1, a2)
| ^= => x5e-x3d (ty-string, a1, a2)
| |= => x7c-x3d (ty-string, a1, a2)
| <<= => x3c-x3c-x3d (ty-string, a1, a2)
| >>= => x3e-x3e-x3d (ty-string, a1, a2)
| , => x2c (ty-string, a1, a2)
| ; => x3b (ty-string, a1, a2)
| { => x7b (ty-string, a1, a2)
| } => x7d (ty-string, a1, a2)
| ... => x2e-x2e-x2e (ty-string, a1, a2)
| x => let
val alignof = alignof (ty-string, a1, a2)
val alignas = alignas (ty-string, a1, a2)
val atomic = x5f-Atomic (ty-string, a1, a2)
```

val asm = asm (ty-string, a1, a2)
val auto = auto (ty-string, a1, a2)
val break = break (ty-string, a1, a2)
val bool = x5f-Bool (ty-string, a1, a2)
val case0 = case0 (ty-string, a1, a2)
val char = char (ty-string, a1, a2)
val const = const (ty-string, a1, a2)
val continue = continue (ty-string, a1, a2)
val complex = x5f-Complex (ty-string, a1, a2)
val default = default (ty-string, a1, a2)
val do0 = do0 (ty-string, a1, a2)
val double = double (ty-string, a1, a2)
val else0 = else0 (ty-string, a1, a2)
val enum = enum (ty-string, a1, a2)
val extern = extern (ty-string, a1, a2)
val float = float (ty-string, a1, a2)
val for0 = for0 (ty-string, a1, a2)
val generic = x5f-Generic (ty-string, a1, a2)
val goto = goto (ty-string, a1, a2)
val if0 = if0 (ty-string, a1, a2)
val inline = inline (ty-string, a1, a2)
val int = int (ty-string, a1, a2)
val int128 = x5f-x5f-int-x31-x32-x38 (ty-string, a1, a2)
val long = long (ty-string, a1, a2)
val label = x5f-x5f-label-x5f-x5f (ty-string, a1, a2)
val noreturn = x5f-Noreturn (ty-string, a1, a2)
val nullable = x5f-Nullable (ty-string, a1, a2)
val nonnull = x5f-Nonnull (ty-string, a1, a2)
val register = register (ty-string, a1, a2)
val restrict = restrict (ty-string, a1, a2)
val return0 = return0 (ty-string, a1, a2)
val short = short (ty-string, a1, a2)
val signed = signed (ty-string, a1, a2)
val sizeof = sizeof (ty-string, a1, a2)
val static = static (ty-string, a1, a2)
val staticassert = x5f-Static-assert (ty-string, a1, a2)
val struct0 = struct0 (ty-string, a1, a2)
val switch = switch (ty-string, a1, a2)
val typedef = typedef (ty-string, a1, a2)
val typeof = typeof (ty-string, a1, a2)
val thread = x5f-x5f-thread (ty-string, a1, a2)
val union = union (ty-string, a1, a2)
val unsigned = unsigned (ty-string, a1, a2)
val void = void (ty-string, a1, a2)
val volatile = volatile (ty-string, a1, a2)
val while0 = while0 (ty-string, a1, a2)
val cchar = cchar (ty-cChar, a1, a2)
val cint = cint (ty-cInteger, a1, a2)
val cfloat = cfloat (ty-cFloat, a1, a2)

```
val cstr = cstr (ty-cString, a1, a2)
val ident = ident (ty-ident, a1, a2)
val tyident = tyident (ty-ident, a1, a2)
val attribute = x5f-x5f-attribute-x5f-x5f (ty-string, a1, a2)
val extension = x5f-x5f-extension-x5f-x5f (ty-string, a1, a2)
val real = x5f-x5f-real-x5f-x5f (ty-string, a1, a2)
val imag = x5f-x5f-imag-x5f-x5f (ty-string, a1, a2)
val builtinvaarg = x5f-x5f-builtin-va-arg (ty-string, a1, a2)
val builtinoffsetof = x5f-x5f-builtin-offsetof (ty-string, a1, a2)
val builtintypescompatiblep = x5f-x5f-builtin-types-compatible-p (ty-string, a1, a2)
val clangcversion = clangcversion (ty-ClangCVersion, a1, a2)
in case x of
  -Alignas => alignas
| -Alignof => alignof
| --alignof => alignof
| alignof => alignof
| --alignof-- => alignof
| --asm => asm
| asm => asm
| --asm-- => asm
| -Atomic => atomic
| --attribute => attribute
| --attribute-- => attribute
| auto => auto
| -Bool => bool
| break => break
| --builtin-offsetof => builtinoffsetof
| --builtin-types-compatible-p => builtintypescompatiblep
| --builtin-va-arg => builtinvaarg
| case => case0
| char => char
| -Complex => complex
| --complex-- => complex
| --const => const
| const => const
| --const-- => const
| continue => continue
| default => default
| do => do0
| double => double
| else => else0
| enum => enum
| --extension-- => extension
| extern => extern
| float => float
| for => for0
| -Generic => generic
| goto => goto
| if => if0
```

```
    | --imag => imag
    | --imag-- => imag
    | --inline => inline
    | inline => inline
    | --inline-- => inline
    | int => int
    | --int128 => int128
    | --label-- => label
    | long => long
    | -Nonnull => nonnull
    | --nonnull => nonnull
    | -Noreturn => noreturn
    | -Nullable => nullable
    | --nullable => nullable
    | --real => real
    | --real-- => real
    | register => register
    | --restrict => restrict
    | restrict => restrict
    | --restrict-- => restrict
    | return => return0
    | short => short
    | --signed => signed
    | signed => signed
    | --signed-- => signed
    | sizeof => sizeof
    | static => static
    | -Static-assert => staticassert
    | struct => struct0
    | switch => switch
    | --thread => thread
    | -Thread-local => thread
    | typedef => typedef
    | --typeof => typeof
    | typeof => typeof
    | --typeof-- => typeof
    | union => union
    | unsigned => unsigned
    | void => void
    | --volatile => volatile
    | volatile => volatile
    | --volatile-- => volatile
    | while => while0
    | - => error
  end
end
end
›
```

end

3.6 Annotation Language: Parsing Combinator

theory *C-Lexer-Annotation*
imports *C-Lexer-Language*
begin

ML — `~~/src/Pure/Isar/keyword.ML`

```
‹
structure C-Keyword =
struct

(** keyword classification **)

(* kinds *)


val command-kinds =
  [Keyword.diag, Keyword.document-heading, Keyword.document-body, Keyword.document-raw,
                        Keyword.thy-begin, Keyword.thy-end, Keyword.thy-load, Keyword.thy-decl,
    Keyword.thy-decl-block, Keyword.thy-defn, Keyword.thy-stmt, Keyword.thy-goal,
                  Keyword.thy-goal-defn, Keyword.thy-goal-stmt, Keyword.qed, Keyword.qed-script,
          Keyword.qed-block, Keyword.qed-global, Keyword.prf-goal, Keyword.prf-block, Key-
word.next-block,
                                          Keyword.prf-open, Keyword.prf-close, Keyword.prf-chain,
    Keyword.prf-decl, Keyword.prf-asm, Keyword.prf-asm-goal, Keyword.prf-script,
                                          Keyword.prf-script-goal, Keyword.prf-script-asm-goal];


(* specifications *)

type spec = Keyword.spec;

type entry =
 {pos: Position.T,
  id: serial,
  kind: string,
  tags: string list};

fun check-spec pos ({kind, tags, ...}: spec) : entry =
  if not (member (op =) command-kinds kind) then
    error (Unknown annotation syntax keyword kind ^ quote kind)
  else {pos = pos, id = serial (), kind = kind, tags = tags};


(** keyword tables **)
```

```
(* type keywords *)

datatype keywords = Keywords of
 {minor: Scan.lexicon,
  major: Scan.lexicon,
  commands: entry Symtab.table};

fun minor-keywords (Keywords {minor, ...}) = minor;
fun major-keywords (Keywords {major, ...}) = major;

fun make-keywords (minor, major, commands) =
  Keywords {minor = minor, major = major, commands = commands};

fun map-keywords f (Keywords {minor, major, commands}) =
  make-keywords (f (minor, major, commands));


(* build keywords *)

val empty-keywords =
  make-keywords (Scan.empty-lexicon, Scan.empty-lexicon, Symtab.empty);

fun empty-keywords' minor =
  make-keywords (minor, Scan.empty-lexicon, Symtab.empty);

fun merge-keywords
  (Keywords {minor = minor1, major = major1, commands = commands1},
   Keywords {minor = minor2, major = major2, commands = commands2}) =
  make-keywords
   (Scan.merge-lexicons (minor1, minor2),
    Scan.merge-lexicons (major1, major2),
    Symtab.merge (K true) (commands1, commands2));

val add-keywords0 =
  fold
    (fn ((name, pos), force-minor, spec as {kind, ...}: spec) =>
      map-keywords (fn (minor, major, commands) =>
        let val extend = Scan.extend-lexicon (Symbol.explode name)
            fun update spec = Symtab.update (name, spec)
        in
          if force-minor then
            (extend minor, major, update (check-spec pos spec) commands)
          else if kind =  orelse kind = Keyword.before-command
                          orelse kind = Keyword.quasi-command then
            (extend minor, major, commands)
          else
            (minor, extend major, update (check-spec pos spec) commands)
        end));
```

```
val add-keywords = add-keywords0 o map (fn (cmd, spec) => (cmd, false, spec))
val add-keywords-minor = add-keywords0 o map (fn (cmd, spec) => (cmd, true, spec))


(* keyword status *)

fun is-command (Keywords {commands, ...}) = Symtab.defined commands;
fun dest-commands (Keywords {commands, ...}) = Symtab.keys commands;


(* command keywords *)

fun lookup-command (Keywords {commands, ...}) = Symtab.lookup commands;

fun command-markup keywords name =
  lookup-command keywords name
  |> Option.map (fn {pos, id, ...} =>
      Position.make-entity-markup {def = false} id Markup.command-keywordN (name, pos));




(* command categories *)

fun command-category ks =
  let
    val tab = Symtab.make-set ks;
    fun pred keywords name =
      (case lookup-command keywords name of
        NONE => false
      | SOME {kind, ...} => Symtab.defined tab kind);
  in pred end;




val is-theory-end = command-category [Keyword.thy-end];









val is-proof-asm = command-category [Keyword.prf-asm, Keyword.prf-asm-goal];
val is-improper = command-category [ Keyword.qed-script
                                   , Keyword.prf-script
                                   , Keyword.prf-script-goal
```

, *Keyword.prf-script-asm-goal*];

end;
›

Notes:

- The next structure contains a duplicated copy of the type `Token.T`, since it is not possible to set an arbitrary *slot* value in `Token`.
- Parsing priorities in C and HOL slightly differ, see for instance `Token.explode`.

ML — `~~/src/Pure/Isar/token.ML`

```
‹
structure C-Token =
struct

(** tokens **)

(* token kind *)

fun equiv-kind kind kind' =
  (case (kind, kind') of
    (Token.Control -, Token.Control -) => true
  | (Token.Error -, Token.Error -) => true
  | - => kind = kind');

val immediate-kinds' = fn Token.Command => 0
                        | Token.Keyword => 1
                        | Token.Ident => 2
                        | Token.Long-Ident => 3
                        | Token.Sym-Ident => 4
                        | Token.Var => 5
                        | Token.Type-Ident => 6
                        | Token.Type-Var => 7
                        | Token.Nat => 8
                        | Token.Float => 9
                        | Token.Space => 10
                        | - => ~1

val delimited-kind =
  (fn Token.String => true
    | Token.Alt-String => true
    | Token.Cartouche => true
    | Token.Control - => true
    | Token.Comment - => true
    | - => false);
```

```
(* datatype token *)

(*The value slot assigns an (optional) internal value to a token,
  usually as a side−effect of special scanner setup (see also
  args.ML). Note that an assignable ref designates an intermediate
  state of internalization −− it is NOT meant to persist.*)

datatype T = Token of (Symbol-Pos.text * Position.range) * (Token.kind * string) * slot

and slot =
  Slot |
  Value of value option |
  Assignable of value option Unsynchronized.ref

and value =
  Source of T list |
  Literal of bool * Markup.T |
  Name of Token.name-value * morphism |
  Typ of typ |
  Term of term |
  Fact of string option * thm list | (*optional name for dynamic fact, i.e. fact variable*)
  Attribute of morphism -> attribute |
  Declaration of Morphism.declaration |
  Files of Token.file Exn.result list |
  Output of XML.body option;

type src = T list;


(* position *)

fun pos-of (Token ((-, (pos, -)), -, -)) = pos;
fun end-pos-of (Token ((-, (-, pos)), -, -)) = pos;

fun adjust-offsets adjust (Token ((x, range), y, z)) =
  Token ((x, apply2 (Position.adjust-offsets adjust) range), y, z);


(* stopper *)

fun mk-eof pos = Token ((, (pos, Position.none)), (Token.EOF, ), Slot);
val eof = mk-eof Position.none;

fun is-eof (Token (-, (Token.EOF, -), -)) = true
  | is-eof - = false;

val not-eof = not o is-eof;
```

```
val stopper =
  Scan.stopper (fn [] => eof | toks => mk-eof (end-pos-of (List.last toks))) is-eof;


(* kind of token *)

fun kind-of (Token (-, (k, -), -)) = k;
fun is-kind k (Token (-, (k', -), -)) = equiv-kind k k';

fun get-control tok =
  (case kind-of tok of Token.Control control => SOME control | - => NONE);

val is-command = is-kind Token.Command;

fun keyword-with pred (Token (-, (Token.Keyword, x), -)) = pred x
  | keyword-with - - = false;

val is-command-modifier = keyword-with (fn x => x = private orelse x = qualified);

fun ident-with pred (Token (-, (Token.Ident, x), -)) = pred x
  | ident-with - - = false;

fun is-ignored (Token (-, (Token.Space, -), -)) = true
  | is-ignored (Token (-, (Token.Comment NONE, -), -)) = true
  | is-ignored - = false;

fun is-proper (Token (-, (Token.Space, -), -)) = false
  | is-proper (Token (-, (Token.Comment -, -), -)) = false
  | is-proper - = true;

fun is-comment (Token (-, (Token.Comment -, -), -)) = true
  | is-comment - = false;

fun is-informal-comment (Token (-, (Token.Comment NONE, -), -)) = true
  | is-informal-comment - = false;

fun is-formal-comment (Token (-, (Token.Comment (SOME -), -), -)) = true
  | is-formal-comment - = false;

fun is-document-marker (Token (-, (Token.Comment (SOME Comment.Marker), -), -)) = true
  | is-document-marker - = false;

fun is-begin-ignore (Token (-, (Token.Comment NONE, <), -)) = true
  | is-begin-ignore - = false;

fun is-end-ignore (Token (-, (Token.Comment NONE, >), -)) = true
  | is-end-ignore - = false;
```

```
fun is-error (Token (-, (Token.Error -, -), -)) = true
  | is-error - = false;

fun is-error' (Token (-, (Token.Error msg, -), -)) = SOME msg
  | is-error' - = NONE;

fun content-of (Token (-, (-, x), -)) = x;
fun content-of' (Token (-, (-, -), Value (SOME (Source l)))) =
                  map (fn Token ((-, (pos, -)), (-, x), -) => (x, pos)) l
  | content-of' - = [];

val is-stack1 = fn Token (-, (Token.Sym-Ident, -), Value (SOME (Source l))) =>
                     forall (fn tok => content-of tok = +) l
                | - => false;

val is-stack2 = fn Token (-, (Token.Sym-Ident, -), Value (SOME (Source l))) =>
                     forall (fn tok => content-of tok = @) l
                | - => false;

val is-stack3 = fn Token (-, (Token.Sym-Ident, -), Value (SOME (Source l))) =>
                     forall (fn tok => content-of tok = &) l
                | - => false;


(* blanks and newlines -- space tokens obey lines *)

fun is-space (Token (-, (Token.Space, -), -)) = true
  | is-space - = false;

fun is-blank (Token (-, (Token.Space, x), -)) = not (String.isSuffix \n x)
  | is-blank - = false;

fun is-newline (Token (-, (Token.Space, x), -)) = String.isSuffix \n x
  | is-newline - = false;


(* range of tokens *)

fun range-of (toks as tok :: -) =
      let val pos' = end-pos-of (List.last toks)
      in Position.range (pos-of tok, pos') end
  | range-of [] = Position.no-range;

val core-range-of =
  drop-prefix is-ignored #> drop-suffix is-ignored #> range-of;


(* token content *)
```

```
fun content-of (Token (-, (-, x), -)) = x;
fun source-of (Token ((source, -), -, -)) = source;

fun input-of (Token ((source, range), (kind, -), -)) =
  Input.source (delimited-kind kind) source range;

fun inner-syntax-of tok =
  let val x = content-of tok
  in if YXML.detect x then x else Syntax.implode-input (input-of tok) end;


(* markup reports *)

local

val token-kind-markup =
 fn Token.Var => (Markup.var, )
  | Token.Type-Ident => (Markup.tfree, )
  | Token.Type-Var => (Markup.tvar, )
  | Token.String => (Markup.string, )
  | Token.Alt-String => (Markup.alt-string, )
  | Token.Cartouche => (Markup.cartouche, )
  | Token.Control - => (Markup.cartouche, )
  | Token.Comment - => (Markup.ML-comment, )
  | Token.Error msg => (Markup.bad (), msg)
  | - => (Markup.empty, );

fun keyword-reports tok = map (fn markup => ((pos-of tok, markup), ));

fun command-markups keywords x =
  if C-Keyword.is-theory-end keywords x then [Markup.keyword2 |> Markup.keyword-properties]
  else
    (if C-Keyword.is-proof-asm keywords x then [Markup.keyword3]
     else if C-Keyword.is-improper keywords x then [Markup.keyword1, Markup.improper]
     else [Markup.keyword1])
    |> map Markup.command-properties;

fun keyword-markup (important, keyword) x =
  if important orelse Symbol.is-ascii-identifier x then keyword else Markup.delimiter;

fun command-minor-markups keywords x =
  if C-Keyword.is-theory-end keywords x then [Markup.keyword2 |> Markup.keyword-properties]
  else
    (if C-Keyword.is-proof-asm keywords x then [Markup.keyword3]
     else if C-Keyword.is-improper keywords x then [Markup.keyword1, Markup.improper]
     else if C-Keyword.is-command keywords x then [Markup.keyword1]
     else [keyword-markup (false, Markup.keyword2 |> Markup.keyword-properties) x]);

in
```

```
fun completion-report tok =
  if is-kind Token.Keyword tok
  then map (fn m => ((pos-of tok, m), )) (Completion.suppress-abbrevs (content-of tok))
  else [];

fun reports keywords tok =
  if is-command tok then
    keyword-reports tok (command-markups keywords (content-of tok))
  else if is-stack1 tok orelse is-stack2 tok orelse is-stack3 tok then
    keyword-reports tok [Markup.keyword2 |> Markup.keyword-properties]
  else if is-kind Token.Keyword tok then
    keyword-reports tok (command-minor-markups keywords (content-of tok))
  else
    let
      val pos = pos-of tok;
      val (m, text) = token-kind-markup (kind-of tok);
      val deleted = Symbol-Pos.explode-deleted (source-of tok, pos);
    in ((pos, m), text) :: map (fn p => ((p, Markup.delete), )) deleted end;

fun markups keywords = map (#2 o #1) o reports keywords;

end;


(* unparse *)

fun unparse' (Token ((source0, -), (kind, x), -)) =
  let
    val source =
     — We are computing a reverse function of Symbol_Pos.implode_range taking into account
consecutive Symbol.DEL symbols potentially appearing at the beginning, or at the end of the
string.
      case Symbol.explode source0 of
        x :: xs =>
          if x = Symbol.DEL then
            case rev xs of x' :: xs => if x' = Symbol.DEL then implode (rev xs) else source0
                         | - => source0
          else
           source0
      | - => source0
  in
    case kind of
      Token.String => Symbol-Pos.quote-string-qq source
    | Token.Alt-String => Symbol-Pos.quote-string-bq source
    | Token.Cartouche => cartouche source
    | Token.Control control => Symbol-Pos.content (Antiquote.control-symbols control)
    | Token.Comment NONE => enclose (* *) source
    | Token.EOF =>
```

```
    | - => x
  end;

fun text-of tok =
  let
    val k = Token.str-of-kind (kind-of tok);
    val ms = markups C-Keyword.empty-keywords tok;
    val s = unparse' tok;
  in
    if s =  then (k, )
    else if size s < 40 andalso not (exists-string (fn c => c = \n) s)
    then (k ^  ^ Markup.markups ms s, )
    else (k, Markup.markups ms s)
  end;



(** associated values **)

(* inlined file content *)

fun file-source (file: Token.file) =
  let
    val text = cat-lines (#lines file);
    val end-pos = Position.symbol-explode text (#pos file);
  in Input.source true text (Position.range (#pos file, end-pos)) end;

fun get-files (Token (-, -, Value (SOME (Files files)))) = files
  | get-files - = [];

fun put-files [] tok = tok
  | put-files files (Token (x, y, Slot)) = Token (x, y, Value (SOME (Files files)))
  | put-files - tok = raise Fail (Cannot put inlined files here ^ Position.here (pos-of tok));


(* access values *)



(* reports of value *)



(* name value *)



(* maxidx *)
```

```
(* fact values *)


(* transform *)


(* static binding *)

(*1st stage: initialize assignable slots*)
fun init-assignable tok =
  (case tok of
     Token (x, y, Slot) => Token (x, y, Assignable (Unsynchronized.ref NONE))
   | Token (-, -, Value -) => tok
   | Token (-, -, Assignable r) => (r := NONE; tok));

(*2nd stage: assign values as side-effect of scanning*)
fun assign v tok =
  (case tok of
     Token (x, y, Slot) => Token (x, y, Value v)
   | Token (-, -, Value -) => tok
   | Token (-, -, Assignable r) => (r := v; tok));

fun evaluate mk eval arg =
  let val x = eval arg in (assign (SOME (mk x)) arg; x) end;

(*3rd stage: static closure of final values*)
fun closure (Token (x, y, Assignable (Unsynchronized.ref v))) = Token (x, y, Value v)
  | closure tok = tok;


(* pretty *)


(* src *)




(** scanners **)

open Basic-Symbol-Pos;
```

```
val err-prefix = Annotation lexical error: ;

fun !!! msg = Symbol-Pos.!!! (fn () => err-prefix ^ msg);


(* scan stack *)

fun scan-stack is-stack = Scan.optional (Scan.one is-stack >> content-of') []


(* scan symbolic idents *)

val scan-symid =
  Scan.many1 (Symbol.is-symbolic-char o Symbol-Pos.symbol) ||
  Scan.one (Symbol.is-symbolic o Symbol-Pos.symbol) >> single;

fun is-symid str =
  (case try Symbol.explode str of
    SOME [s] => Symbol.is-symbolic s orelse Symbol.is-symbolic-char s
  | SOME ss => forall Symbol.is-symbolic-char ss
  | - => false);

fun ident-or-symbolic begin = false
  | ident-or-symbolic : = true
  | ident-or-symbolic :: = true
  | ident-or-symbolic s = Symbol-Pos.is-identifier s orelse is-symid s;


(* scan cartouche *)

val scan-cartouche =
  Symbol-Pos.scan-pos --
      ((Symbol-Pos.scan-cartouche err-prefix >> Symbol-Pos.cartouche-content) -- Sym-
bol-Pos.scan-pos);


(* scan space *)

fun space-symbol (s, -) = Symbol.is-blank s andalso s <> \n;

val scan-space =
  Scan.many1 space-symbol @@@ Scan.optional ($$$ \n) [] ||
  Scan.many space-symbol @@@ $$$ \n;


(* scan comment *)

val scan-comment =
```

```
  Symbol-Pos.scan-pos -- (Symbol-Pos.scan-comment-body err-prefix -- Symbol-Pos.scan-pos);


(** token sources **)

local

fun token-leq ((-, syms1), (-, syms2)) = length syms1 <= length syms2;

fun token k ss =
  Token ((Symbol-Pos.implode ss, Symbol-Pos.range ss), (k, Symbol-Pos.content ss), Slot);

fun token' (mk-value, k) ss =
  if mk-value then
    Token ( (Symbol-Pos.implode ss, Symbol-Pos.range ss)
          , (k, Symbol-Pos.content ss)
          , Value (SOME (Source (map (fn (s, pos) =>
                                       Token ((, (pos, Position.none)), (k, s), Slot))
                                     ss))))
  else
    token k ss;

fun token-t k = token' (true, k)

fun token-range k (pos1, (ss, pos2)) =
  Token (Symbol-Pos.implode-range (pos1, pos2) ss, (k, Symbol-Pos.content ss), Slot);

fun scan-token keywords = !!! bad input
  (Symbol-Pos.scan-string-qq err-prefix >> token-range Token.String ||
    Symbol-Pos.scan-string-bq err-prefix >> token-range Token.Alt-String ||
    scan-comment >> token-range (Token.Comment NONE) ||
    Comment.scan-outer >> (fn (k, ss) => token (Token.Comment (SOME k)) ss) ||
    scan-cartouche >> token-range Token.Cartouche ||
    Antiquote.scan-control err-prefix >> (fn control =>
      token (Token.Control control) (Antiquote.control-symbols control)) ||
    scan-space >> token Token.Space ||
    Scan.repeats1 ($$$ +) >> token-t Token.Sym-Ident ||
    Scan.repeats1 ($$$ @) >> token-t Token.Sym-Ident ||
    Scan.repeats1 ($$$ &) >> token-t Token.Sym-Ident ||
    (Scan.max token-leq
      (Scan.max token-leq
        (Scan.literal (C-Keyword.major-keywords keywords) >> pair Token.Command)
        (Scan.literal (C-Keyword.minor-keywords keywords) >> pair Token.Keyword))
      (Lexicon.scan-longid >> pair Token.Long-Ident ||
        Scan.max
          token-leq
          (C-Lex.scan-ident' >> pair Token.Ident)
          (Lexicon.scan-id >> pair Token.Ident) ||
```

```
        Lexicon.scan-var >> pair Token.Var ||
        Lexicon.scan-tid >> pair Token.Type-Ident ||
        Lexicon.scan-tvar >> pair Token.Type-Var ||
        Symbol-Pos.scan-float >> pair Token.Float ||
        Symbol-Pos.scan-nat >> pair Token.Nat ||
        scan-symid >> pair Token.Sym-Ident)) >> uncurry (token' o pair false));

fun recover msg =
  (Symbol-Pos.recover-string-qq ||
    Symbol-Pos.recover-string-bq ||
    Symbol-Pos.recover-cartouche ||
    Symbol-Pos.recover-comment ||
    Scan.one (Symbol.not-eof o Symbol-Pos.symbol) >> single)
  >> (single o token (Token.Error msg));

in

fun make-source keywords {strict} =
  let
    val scan-strict = Scan.bulk (scan-token keywords);
    val scan = if strict then scan-strict else Scan.recover scan-strict recover;
  in Source.source Symbol-Pos.stopper scan end;


end;


(* explode *)

fun tokenize keywords strict syms =
  Source.of-list syms |> make-source keywords strict |> Source.exhaust;

fun explode keywords pos text =
  Symbol-Pos.explode (text, pos) |> tokenize keywords {strict = false};

fun explode0 keywords = explode keywords Position.none;


(* print names in parsable form *)



(* make *)




(** parsers **)
```

```
type 'a parser = T list -> 'a * T list;
type 'a context-parser = Context.generic * T list -> 'a * (Context.generic * T list);


(* wrapped syntax *)

local
fun make src pos = Token.make src pos |> #1
fun make-default text pos = make ((~1, 0), text) pos
fun explode keywords pos text =
  case Token.explode keywords pos text of [tok] => tok
                                        | - => make-default text pos
in
fun syntax' f =
  I #> map
   (fn tok0 as Token ((source, (pos1, pos2)), (kind, x), -) =>
    if is-stack1 tok0 orelse is-stack2 tok0 orelse is-stack3 tok0 then
      make-default source pos1
    else if is-eof tok0 then
      Token.eof
    else if delimited-kind kind then
      explode Keyword.empty-keywords pos1 (unparse' tok0)
    else
      let
        val tok1 =
          explode
            ((case kind of
                Token.Keyword => Keyword.add-keywords [((x, Position.none), Keyword.no-spec)]
              | Token.Command => Keyword.add-keywords [( (x, Position.none)
                                                       , Keyword.command-spec
                                                           (Keyword.thy-decl, []))]
              | - => I)
               Keyword.empty-keywords)
            pos1
            source
      in
        if Token.kind-of tok1 = kind then
          tok1
        else
          make ( ( immediate-kinds' kind
                 , case Position.distance-of (pos1, pos2) of NONE => 0 | SOME i => i)
               , source)
               pos1
      end)
   #> f
   #> apsnd (map (fn tok => Token ( (Token.source-of tok, Token.range-of [tok])
                                 , (Token.kind-of tok, Token.content-of tok)
                                 , Slot)))
end
```

```
end;

type 'a c-parser = 'a C-Token.parser;
type 'a c-context-parser = 'a C-Token.context-parser;
›
```

ML — `~~/src/Pure/Isar/parse.ML`

```
‹
signature C-PARSE =
sig
  type T
  type src = T list
  type 'a parser = T list -> 'a * T list
  type 'a context-parser = Context.generic * T list -> 'a * (Context.generic * T list)
(**)
  val C-source: Input.source parser
  val star: string parser
(**)
  val group: (unit -> string) -> (T list -> 'a) -> T list -> 'a
  val !!! : (T list -> 'a) -> T list -> 'a
  val !!!! : (T list -> 'a) -> T list -> 'a
  val not-eof: T parser
  val token: 'a parser -> T parser
  val range: 'a parser -> ('a * Position.range) parser
  val position: 'a parser -> ('a * Position.T) parser
  val input: 'a parser -> Input.source parser
  val inner-syntax: 'a parser -> string parser
  val command: string parser
  val keyword: string parser
  val short-ident: string parser
  val long-ident: string parser
  val sym-ident: string parser
  val dots: string parser
  val minus: string parser
  val term-var: string parser
  val type-ident: string parser
  val type-var: string parser
  val number: string parser
  val float-number: string parser
  val string: string parser
  val string-position: (string * Position.T) parser
  val alt-string: string parser
  val cartouche: string parser
  val control: Antiquote.control parser
  val eof: string parser
  val command-name: string -> string parser
  val keyword-with: (string -> bool) -> string parser
```

val keyword-markup: *bool * Markup.T -> string -> string parser*
val keyword-improper: *string -> string parser*
val $$$: *string -> string parser*
val reserved: *string -> string parser*
val underscore: *string parser*
val maybe: *'a parser -> 'a option parser*
val maybe-position: *('a * Position.T) parser -> ('a option * Position.T) parser*
val opt-keyword: *string -> bool parser*
val opt-bang: *bool parser*
val begin: *string parser*
val opt-begin: *bool parser*
val nat: *int parser*
val int: *int parser*
val real: *real parser*
val enum-positions: *string -> 'a parser -> ('a list * Position.T list) parser*
val enum1-positions: *string -> 'a parser -> ('a list * Position.T list) parser*
val enum: *string -> 'a parser -> 'a list parser*
val enum1: *string -> 'a parser -> 'a list parser*
val and-list: *'a parser -> 'a list parser*
val and-list1: *'a parser -> 'a list parser*
val enum': *string -> 'a context-parser -> 'a list context-parser*
val enum1': *string -> 'a context-parser -> 'a list context-parser*
val and-list': *'a context-parser -> 'a list context-parser*
val and-list1': *'a context-parser -> 'a list context-parser*
val list: *'a parser -> 'a list parser*
val list1: *'a parser -> 'a list parser*
val name: *string parser*
val name-range: *(string * Position.range) parser*
val name-position: *(string * Position.T) parser*
val binding: *binding parser*
val embedded: *string parser*
val embedded-inner-syntax: *string parser*
val embedded-input: *Input.source parser*
val embedded-position: *(string * Position.T) parser*
val path-input: *Input.source parser*
val path: *string parser*
val path-binding: *(string * Position.T) parser*
val session-name: *(string * Position.T) parser*
val theory-name: *(string * Position.T) parser*
val liberal-name: *string parser*
val parname: *string parser*
val parbinding: *binding parser*
val class: *string parser*
val sort: *string parser*
val type-const: *string parser*
val arity: *(string * string list * string) parser*
val multi-arity: *(string list * string list * string) parser*
val type-args: *string list parser*
val type-args-constrained: *(string * string option) list parser*

```
  val typ: string parser
  val mixfix: mixfix parser
  val mixfix': mixfix parser
  val opt-mixfix: mixfix parser
  val opt-mixfix': mixfix parser
  val syntax-mode: Syntax.mode parser
  val where-: string parser
  val const-decl: (string * string * mixfix) parser
  val const-binding: (binding * string * mixfix) parser
  val params: (binding * string option * mixfix) list parser
  val vars: (binding * string option * mixfix) list parser
  val for-fixes: (binding * string option * mixfix) list parser
  val ML-source: Input.source parser
  val document-source: Input.source parser
  val document-marker: Input.source parser
  val const: string parser
  val term: string parser
  val prop: string parser
  val literal-fact: string parser
  val propp: (string * string list) parser
  val termp: (string * string list) parser
  val private: Position.T parser
  val qualified: Position.T parser
  val target: (string * Position.T) parser
  val opt-target: (string * Position.T) option parser
  val args: T list parser
  val args1: (string -> bool) -> T list parser
  val attribs: src list parser
  val opt-attribs: src list parser
  val thm-sel: Facts.interval list parser
  val thm: (Facts.ref * src list) parser
  val thms1: (Facts.ref * src list) list parser
  val options: ((string * Position.T) * (string * Position.T)) list parser
  val embedded-ml: ML-Lex.token Antiquote.antiquote list parser
end;

structure C-Parse: C-PARSE =
struct
type T = C-Token.T
type src = T list
type 'a parser = T list -> 'a * T list
type 'a context-parser = Context.generic * T list -> 'a * (Context.generic * T list)
structure Token =
struct
  open Token
  open C-Token
end

(** error handling **)
```

```
(* group atomic parsers (no cuts!) *)

fun group s scan = scan || Scan.fail-with
  (fn [] => (fn () => s () ^ expected,\nbut end-of-input was found)
    | tok :: - =>
        (fn () =>
          (case Token.text-of tok of
            (txt, ) =>
              s () ^ expected,\nbut  ^ txt ^ Position.here (Token.pos-of tok) ^
               was found
          | (txt1, txt2) =>
              s () ^ expected,\nbut  ^ txt1 ^ Position.here (Token.pos-of tok) ^
               was found:\n ^ txt2)));


(* cut *)

fun cut kind scan =
  let
    fun get-pos [] =  (end-of-input)
      | get-pos (tok :: -) = Position.here (Token.pos-of tok);

    fun err (toks, NONE) = (fn () => kind ^ get-pos toks)
      | err (toks, SOME msg) =
          (fn () =>
            let val s = msg () in
              if String.isPrefix kind s then s
              else kind ^ get-pos toks ^ :  ^ s
            end);
  in Scan.!! err scan end;

fun !!! scan = cut Annotation syntax error scan;
fun !!!! scan = cut Corrupted annotation syntax in presentation scan;



(** basic parsers **)

(* tokens *)

fun RESET-VALUE atom = (*required for all primitive parsers*)
  Scan.ahead (Scan.one (K true)) -- atom >> (fn (arg, x) => (Token.assign NONE arg; x));


val not-eof = RESET-VALUE (Scan.one Token.not-eof);

fun token atom = Scan.ahead not-eof --| atom;
```

```
fun range scan = (Scan.ahead not-eof >> (Token.range-of o single)) -- scan >> Library.swap;
fun position scan = (Scan.ahead not-eof >> Token.pos-of) -- scan >> Library.swap;
fun input atom = Scan.ahead atom |-- not-eof >> Token.input-of;
fun inner-syntax atom = Scan.ahead atom |-- not-eof >> Token.inner-syntax-of;

fun kind k =
  group (fn () => Token.str-of-kind k)
    (RESET-VALUE (Scan.one (Token.is-kind k) >> Token.content-of));

val command = kind Token.Command;
val keyword = kind Token.Keyword;
val short-ident = kind Token.Ident;
val long-ident = kind Token.Long-Ident;
val sym-ident = kind Token.Sym-Ident;
val term-var = kind Token.Var;
val type-ident = kind Token.Type-Ident;
val type-var = kind Token.Type-Var;
val number = kind Token.Nat;
val float-number = kind Token.Float;
val string = kind Token.String;
val alt-string = kind Token.Alt-String;
val cartouche = kind Token.Cartouche;
val control = token (kind Token.control-kind) >> (the o Token.get-control);
val eof = kind Token.EOF;

fun command-name x =
  group (fn () => Token.str-of-kind Token.Command ^ ^ quote x)
    (RESET-VALUE (Scan.one (fn tok => Token.is-command tok andalso Token.content-of tok
= x)))
  >> Token.content-of;

fun keyword-with pred = RESET-VALUE (Scan.one (Token.keyword-with pred) >> To-
ken.content-of);

fun keyword-markup markup x =
  group (fn () => Token.str-of-kind Token.Keyword ^ ^ quote x)
    (Scan.ahead not-eof -- keyword-with (fn y => x = y))
  >> (fn (tok, x) => (Token.assign (SOME (Token.Literal markup)) tok; x));

val keyword-improper = keyword-markup (true, Markup.improper);
val $$$ = keyword-markup (false, Markup.quasi-keyword);

fun reserved x =
  group (fn () => reserved identifier ^ quote x)
    (RESET-VALUE (Scan.one (Token.ident-with (fn y => x = y)) >> Token.content-of));

val dots = sym-ident :-- (fn ... => Scan.succeed () | - => Scan.fail) >> #1;

val minus = sym-ident :-- (fn - => Scan.succeed () | - => Scan.fail) >> #1;
```

```
val underscore = sym-ident :-- (fn - => Scan.succeed () | - => Scan.fail) >> #1;
fun maybe scan = underscore >> K NONE || scan >> SOME;
fun maybe-position scan = position (underscore >> K NONE) || scan >> apfst SOME;

val nat = number >> (#1 o Library.read-int o Symbol.explode);
val int = Scan.optional (minus >> K ~1) 1 -- nat >> op *;
val real = float-number >> Value.parse-real || int >> Real.fromInt;

fun opt-keyword s = Scan.optional ($$$ ( |-- !!! (($$$ s >> K true) --| $$$ ))) false;
val opt-bang = Scan.optional ($$$ ! >> K true) false;

val begin = $$$ begin;
val opt-begin = Scan.optional (begin >> K true) false;


(* enumerations *)

fun enum1-positions sep scan =
  scan -- Scan.repeat (position ($$$ sep) -- !!! scan) >>
    (fn (x, ys) => (x :: map #2 ys, map (#2 o #1) ys));
fun enum-positions sep scan =
  enum1-positions sep scan || Scan.succeed ([], []);

fun enum1 sep scan = scan ::: Scan.repeat ($$$ sep |-- !!! scan);
fun enum sep scan = enum1 sep scan || Scan.succeed [];

fun enum1' sep scan = scan ::: Scan.repeat (Scan.lift ($$$ sep) |-- scan);
fun enum' sep scan = enum1' sep scan || Scan.succeed [];

fun and-list1 scan = enum1 and scan;
fun and-list scan = enum and scan;

fun and-list1' scan = enum1' and scan;
fun and-list' scan = enum' and scan;

fun list1 scan = enum1 , scan;
fun list scan = enum , scan;


(* names and embedded content *)

val name =
  group (fn () => name)
    (short-ident || long-ident || sym-ident || number || string);

val name-range = input name >> Input.source-content-range;
val name-position = input name >> Input.source-content;
```

```
val string-position = input string >> Input.source-content;

val binding = name-position >> Binding.make;

val embedded =
  group (fn () => embedded content)
    (cartouche || string || short-ident || long-ident || sym-ident ||
      term-var || type-ident || type-var || number);

val embedded-inner-syntax = inner-syntax embedded;
val embedded-input = input embedded;
val embedded-position = embedded-input >> Input.source-content;

val path-input = group (fn () => file name/path specification) embedded-input;
val path = path-input >> Input.string-of;
val path-binding = group (fn () => path binding (strict file name)) (position embedded);

val session-name = group (fn () => session name) name-position;
val theory-name = group (fn () => theory name) name-position;

val liberal-name = keyword-with Token.ident-or-symbolic || name;

val parname = Scan.optional ($$$ ( |-- name --| $$$ )) ;
val parbinding = Scan.optional ($$$ ( |-- binding --| $$$ )) Binding.empty;


(* type classes *)

val class = group (fn () => type class) (inner-syntax embedded);

val sort = group (fn () => sort) (inner-syntax embedded);

val type-const = group (fn () => type constructor) (inner-syntax embedded);

val arity = type-const -- ($$$ :: |-- !!!
  (Scan.optional ($$$ ( |-- !!! (list1 sort --| $$$ ))) [] -- sort)) >> Scan.triple2;

val multi-arity = and-list1 type-const -- ($$$ :: |-- !!!
  (Scan.optional ($$$ ( |-- !!! (list1 sort --| $$$ ))) [] -- sort)) >> Scan.triple2;


(* types *)

val typ = group (fn () => type) (inner-syntax embedded);

fun type-arguments arg =
  arg >> single ||
  $$$ ( |-- !!! (list1 arg --| $$$ )) ||
  Scan.succeed [];
```

```
val type-args = type-arguments type-ident;
val type-args-constrained = type-arguments (type-ident -- Scan.option ($$$ :: |-- !!! sort));


(* mixfix annotations *)

local

val mfix = input (string || cartouche);

val mixfix- =
  mfix -- !!! (Scan.optional ($$$ [ |-- !!! (list nat --| $$$ ])) [] -- Scan.optional nat 1000)
    >> (fn (sy, (ps, p)) => fn range => Mixfix (sy, ps, p, range));

val structure- = $$$ structure >> K Structure;

val binder- =
  $$$ binder |-- !!! (mfix -- ($$$ [ |-- nat --| $$$ ] -- nat || nat >> (fn n => (n, n))))
    >> (fn (sy, (p, q)) => fn range => Binder (sy, p, q, range));

val infixl- = $$$ infixl
            |-- !!! (mfix -- nat >> (fn (sy, p) => fn range => Infixl (sy, p, range)));
val infixr- = $$$ infixr
            |-- !!! (mfix -- nat >> (fn (sy, p) => fn range => Infixr (sy, p, range)));
val infix- = $$$ infix
            |-- !!! (mfix -- nat >> (fn (sy, p) => fn range => Infix (sy, p, range)));

val mixfix-body = mixfix- || structure- || binder- || infixl- || infixr- || infix-;

fun annotation guard body =
  Scan.trace ($$$ ( |-- guard (body --| $$$ )))
    >> (fn (mx, toks) => mx (Token.range-of toks));

fun opt-annotation guard body = Scan.optional (annotation guard body) NoSyn;

in

val mixfix = annotation !!! mixfix-body;
val mixfix' = annotation I mixfix-body;
val opt-mixfix = opt-annotation !!! mixfix-body;
val opt-mixfix' = opt-annotation I mixfix-body;

end;


(* syntax mode *)

val syntax-mode-spec =
```

```
  ($$$ output >> K (, false)) || name -- Scan.optional ($$$ output >> K false) true;

val syntax-mode =
  Scan.optional ($$$ ( |-- !!! (syntax-mode-spec --| $$$ ))) Syntax.mode-default;


(* fixes *)

val where- = $$$ where;

val const-decl = name -- ($$$ :: |-- !!! typ) -- opt-mixfix >> Scan.triple1;
val const-binding = binding -- ($$$ :: |-- !!! typ) -- opt-mixfix >> Scan.triple1;

val param-mixfix = binding -- Scan.option ($$$ :: |-- typ) -- mixfix' >> (single o
Scan.triple1);

val params =
  (binding -- Scan.repeat binding) -- Scan.option ($$$ :: |-- !!! (Scan.ahead typ -- embed-
ded))
    >> (fn ((x, ys), T) =>
        (x, Option.map #1 T, NoSyn) :: map (fn y => (y, Option.map #2 T, NoSyn)) ys);

val vars = and-list1 (param-mixfix || params) >> flat;

val for-fixes = Scan.optional ($$$ for |-- !!! vars) [];


(* embedded source text *)

val ML-source = input (group (fn () => ML source) embedded);
val document-source = input (group (fn () => document source) embedded);

val document-marker =
  group (fn () => document marker)
    (RESET-VALUE (Scan.one Token.is-document-marker >> Token.input-of));


(* terms *)

val const = group (fn () => constant) (inner-syntax embedded);
val term = group (fn () => term) (inner-syntax embedded);
val prop = group (fn () => proposition) (inner-syntax embedded);

val literal-fact = inner-syntax (group (fn () => literal fact) (alt-string || cartouche));


(* patterns *)

val is-terms = Scan.repeat1 ($$$ is |-- term);
```

```
val is-props = Scan.repeat1 ($$$ is |-- prop);

val propp = prop -- Scan.optional ($$$ ( |-- !!! (is-props --| $$$ ))) [];
val termp = term -- Scan.optional ($$$ ( |-- !!! (is-terms --| $$$ ))) [];


(* target information *)

val private = position ($$$ private) >> #2;
val qualified = position ($$$ qualified) >> #2;

val target = ($$$ ( -- $$$ in) |-- !!! (name-position --| $$$ ));
val opt-target = Scan.option target;


(* arguments within outer syntax *)

local

val argument-kinds =
  [Token.Ident, Token.Long-Ident, Token.Sym-Ident, Token.Var, Token.Type-Ident, To-
ken.Type-Var,
  Token.Nat, Token.Float, Token.String, Token.Alt-String, Token.Cartouche];

fun arguments is-symid =
  let
    fun argument blk =
      group (fn () => argument)
        (Scan.one (fn tok =>
          let val kind = Token.kind-of tok in
            member (op =) argument-kinds kind orelse
            Token.keyword-with is-symid tok orelse
            (blk andalso Token.keyword-with (fn s => s = ,) tok)
          end));

    fun args blk x = Scan.optional (args1 blk) [] x
    and args1 blk x =
      (Scan.repeats1 (Scan.repeat1 (argument blk) || argsp ( ) || argsp [ ])) x
    and argsp l r x = (token ($$$ l) ::: !!! (args true @@@ (token ($$$ r) >> single))) x;
  in (args, args1) end;

in

val args = #1 (arguments Token.ident-or-symbolic) false;
fun args1 is-symid = #2 (arguments is-symid) false;

end;
```

```
(* attributes *)

val attrib = token liberal-name ::: !!! args;
val attribs = $$$ [ |-- list attrib --| $$$ ];
val opt-attribs = Scan.optional attribs [];


(* theorem references *)

val thm-sel = $$$ ( |-- list1
 (nat --| minus -- nat >> Facts.FromTo ||
  nat --| minus >> Facts.From ||
  nat >> Facts.Single) --| $$$ );

val thm =
  $$$ [ |-- attribs --| $$$ ] >> pair (Facts.named ) ||
  (literal-fact >> Facts.Fact ||
    name-position -- Scan.option thm-sel >> Facts.Named) -- opt-attribs;

val thms1 = Scan.repeat1 thm;


(* options *)

val option-name = group (fn () => option name) name-position;
val option-value = group (fn () => option value) ((token real || token name) >> Token.content-of);

val option =
  option-name :-- (fn (-, pos) =>
    Scan.optional ($$$ = |-- !!! (position option-value)) (true, pos));

val options = $$$ [ |-- list1 option --| $$$ ];


(* embedded ML *)

val embedded-ml =
  input underscore >> ML-Lex.read-source ||
  embedded-input >> ML-Lex.read-source ||
  control >> (ML-Lex.read-symbols o Antiquote.control-symbols);


(* read embedded source, e.g. for antiquotations *)



(** C basic parsers **)
```

```
(* embedded source text *)

val C-source = input (group (fn () => C source) embedded);

(* AutoCorres (MODIFIES) *)

val star = sym-ident :-- (fn * => Scan.succeed () | - => Scan.fail) >> #1;

end;

structure C-Parse-Native: C-PARSE =
struct
open Token
open Parse

(** C basic parsers **)

(* embedded source text *)

val C-source = input (group (fn () => C source) embedded);

(* AutoCorres (MODIFIES) *)

val star = sym-ident :-- (fn * => Scan.succeed () | - => Scan.fail) >> #1;
end;

structure C-Parse-Read =
struct
(* read embedded source, e.g. for antiquotations *)

fun read-with-commands'0 keywords syms =
  Source.of-list syms
  |> C-Token.make-source keywords {strict = false}
  |> Source.filter (not o C-Token.is-proper)
  |> Source.exhaust

fun read-with-commands' keywords scan syms =
  Source.of-list syms
  |> C-Token.make-source keywords {strict = false}
  |> Source.filter C-Token.is-proper
  |> Source.source
       C-Token.stopper
       (Scan.recover
         (Scan.bulk scan)
         (fn msg =>
           Scan.one (not o C-Token.is-eof)
           >> (fn tok => [C-Scan.Right
                       let
                          val msg = case C-Token.is-error' tok of SOME msg0 => msg0 ^ ( ^
```

```
msg ^ )
                                                    | NONE => msg
                        in ( msg
                           , [((C-Token.pos-of tok, Markup.bad ()), msg)]
                           , tok)
                        end])))
  |> Source.exhaust;

fun read-antiq' keywords scan = read-with-commands' keywords (scan >> C-Scan.Left);
end
›
```

ML — `~~/src/Pure/Thy/thy_header.ML`

```
‹
structure C-Thy-Header =
struct
val bootstrap-keywords =
  C-Keyword.empty-keywords' (Keyword.minor-keywords (Thy-Header.get-keywords @{theory}))

(* theory data *)

structure Data = Theory-Data
(
  type T = C-Keyword.keywords;
  val empty = bootstrap-keywords;
  val merge = C-Keyword.merge-keywords;
);

val add-keywords = Data.map o C-Keyword.add-keywords;
val add-keywords-minor = Data.map o C-Keyword.add-keywords-minor;

val get-keywords = Data.get;
val get-keywords' = get-keywords o Proof-Context.theory-of;

end
›
```

end

3.7 Annotation Language: Command Parser Registration

theory *C-Parser-Annotation*
  imports *C-Lexer-Annotation C-Environment*
begin

ML — `~~/src/Pure/Isar/outer_syntax.ML`

```
‹
```

```
structure C-Annotation =
struct

(** outer syntax **)

(* errors *)

fun err-command msg name ps =
  error (msg ^ quote (Markup.markup Markup.keyword1 name) ^ Position.here-list ps);

fun err-dup-command name ps =
  err-command Duplicate annotation syntax command  name ps;


(* command parsers *)

datatype command-parser =
  Parser of (Symbol-Pos.T list * (bool * Symbol-Pos.T list)) * Position.range ->
            C-Env.eval-time c-parser;

datatype command = Command of
 {comment: string,
  command-parser: command-parser,
  pos: Position.T,
  id: serial};

fun eq-command (Command {id = id1, ...}, Command {id = id2, ...}) = id1 = id2;

fun new-command comment command-parser pos =
  Command {comment = comment, command-parser = command-parser, pos = pos, id = serial
()};

fun command-pos (Command {pos, ...}) = pos;

fun command-markup def (name, Command {pos, id, ...}) =
  Position.make-entity-markup def id Markup.commandN (name, pos);



(* theory data *)

structure Data = Theory-Data
(
  type T = command Symtab.table;
  val empty = Symtab.empty;
  fun merge data : T =
    data |> Symtab.join (fn name => fn (cmd1, cmd2) =>
      if eq-command (cmd1, cmd2) then raise Symtab.SAME
      else err-dup-command name [command-pos cmd1, command-pos cmd2]);
```

```
);

val get-commands = Data.get;
val dest-commands = get-commands #> Symtab.dest #> sort-by #1;
val lookup-commands = Symtab.lookup o get-commands;


(* maintain commands *)

fun add-command name cmd thy =
    let
      val - =
        C-Keyword.is-command (C-Thy-Header.get-keywords thy) name orelse
          err-command Undeclared outer syntax command  name [command-pos cmd];
      val - =
        (case lookup-commands thy name of
          NONE => ()
        | SOME cmd' => err-dup-command name [command-pos cmd, command-pos cmd'];
      val - =
        Context-Position.report-generic (Context.the-generic-context ())
          (command-pos cmd) (command-markup {def = true} (name, cmd));
    in Data.map (Symtab.update (name, cmd)) thy end;

fun delete-command (name, pos) thy =
    let
      val - =
        C-Keyword.is-command (C-Thy-Header.get-keywords thy) name orelse
          err-command Undeclared outer syntax command  name [pos];
    in Data.map (Symtab.delete name) thy end;


(* implicit theory setup *)

type command-keyword = string * Position.T;

fun raw-command0 kind (name, pos) comment command-parser =
  C-Thy-Header.add-keywords [((name, pos), Keyword.command-spec (kind, [name]))]
  #> add-command name (new-command comment command-parser pos);

fun raw-command (name, pos) comment command-parser =
  let val setup = add-command name (new-command comment command-parser pos)
  in Context.>> (Context.mapping setup (Local-Theory.background-theory setup)) end;

fun command (name, pos) comment parse =
  raw-command (name, pos) comment (Parser parse);

fun command'' kind (name, pos) comment parse =
  raw-command0 kind (name, pos) comment (Parser parse);
```

```
val command' = command'' Keyword.thy-decl;


(** toplevel parsing **)

(* parse spans *)


(* parse commands *)

local
fun scan-stack' f b = Scan.one f >> (pair b o C-Token.content-of')
in
val before-command =
  C-Token.scan-stack C-Token.is-stack1
  -- Scan.optional (    scan-stack' C-Token.is-stack2 false
                    || scan-stack' C-Token.is-stack3 true)
                   (pair false [])
end

fun parse-command thy =
  Scan.ahead (before-command |-- C-Parse.position C-Parse.command) :|-- (fn (name, pos)
=>
    let val command-tags = before-command -- C-Parse.range C-Parse.command
                          >> (fn (cmd, (-, range)) => (cmd, range));
    in
      case lookup-commands thy name of
        SOME (cmd as Command {command-parser = Parser parse, ...}) =>
          C-Parse.!!! (command-tags :|-- parse)
          >> pair [((pos, command-markup {def = false} (name, cmd)), )]
      | NONE =>
          Scan.fail-with (fn - => fn - =>
            let
              val msg = undefined command ;
            in msg ^ quote (Markup.markup Markup.keyword1 name) end)
    end)


(* check commands *)

fun command-reports thy tok =
  if C-Token.is-command tok then
    let val name = C-Token.content-of tok in
      (case lookup-commands thy name of
        NONE => []
      | SOME cmd => [((C-Token.pos-of tok, command-markup {def = false} (name, cmd)), )])
    end
  else [];
```

```
fun check-command ctxt (name, pos) =
  let
    val thy = Proof-Context.theory-of ctxt;
    val keywords = C-Thy-Header.get-keywords thy;
  in
    if C-Keyword.is-command keywords name then
      let
        val markup =
          C-Token.explode0 keywords name
          |> maps (command-reports thy)
          |> map (#2 o #1);
        val - = Context-Position.reports ctxt (map (pair pos) markup);
      in name end
    else
      let
        val completion-report =
          Completion.make-report (name, pos)
            (fn completed =>
              C-Keyword.dest-commands keywords
              |> filter completed
              |> sort-strings
              |> map (fn a => (a, (Markup.commandN, a))));
      in error (Bad command ^ quote name ^ Position.here pos ^ completion-report) end
  end;
end
›
```

ML — `~~/src/Pure/PIDE/resources.ML`

```
‹
structure C-Resources:
sig
  val parse-file: C-Parse.T list -> (theory -> Token.file) * C-Parse.T list
end =
struct

val parse-file =
  (Scan.ahead C-Parse.not-eof >> C-Token.get-files) -- C-Parse.path-input
    >> (the-single oo Resources.parsed-files single)

end;
›
```

end

3.8 Evaluation Scheduling

theory *C-Eval*

imports *C-Parser-Language*
C-Parser-Annotation
begin

3.8.1 Evaluation Engine for the Core Language

ML — *Isabelle-C.C-Environment* ‹
structure C-Stack =
struct
type 'a stack-elem = (LALR-Table.state, 'a, Position.T) C-Env.stack-elem0
type stack-data = (LALR-Table.state, C-Grammar.Tokens.svalue0, Position.T) C-Env.stack0
type stack-data-elem = (LALR-Table.state, C-Grammar.Tokens.svalue0, Position.T) C-Env.stack-elem0

fun map-svalue0 f (st, (v, pos1, pos2)) = (st, (f v, pos1, pos2))

structure Data-Lang =
struct
val empty' = ([], C-Env.empty-env-lang)
structure Data-Lang = Generic-Data
(
*type T = (stack-data * C-Env.env-lang) option*
val empty = NONE
val merge = K empty
)
open Data-Lang
fun get' context = case get context of NONE => empty' | SOME data => data
fun setmp data f context = put (get context) (f (put data context))
end

structure Data-Tree-Args : GENERIC-DATA-ARGS =
struct
*type T = C-Position.reports-text * C-Env.error-lines*
val empty = ([], [])
fun merge ((l11, l12), (l21, l22)) = (l11 @ l21, l12 @ l22)
end

structure Data-Tree = Generic-Data (Data-Tree-Args)

fun setmp-tree f context =
let val x = Data-Tree.get context
val context = f (Data-Tree.put Data-Tree-Args.empty context)
in (Data-Tree.get context, Data-Tree.put x context) end

fun stack-exec0 f {context, reports-text, error-lines} =
let val ((reports-text', error-lines'), context) = setmp-tree f context
in { context = context
, reports-text = append reports-text' reports-text
, error-lines = append error-lines' error-lines } end

```
fun stack-exec env-dir data-put =
   stack-exec0 o Data-Lang.setmp (SOME (apsnd (C-Env.map-env-directives (K env-dir))
data-put))
end
›
```

ML — `~~/src/Pure/ML/ml_context.ML`

```
‹
structure C-Context0 =
struct
(* theory data *)

type env-direct = bool (* internal result for conditional directives: branch skipping *)
             * (C-Env.env-directives * C-Env.env-tree)

structure Directives = Generic-Data
(
  type T = (Position.T list
           * serial
           * ( (* evaluated during lexing phase *)
               (C-Lex.token-kind-directive
                -> env-direct
                -> C-Env.antiq-language list (* nested annotations from the input *)
                  * env-direct (*NOTE: remove the possibility of returning a too modified env?*))
             * (* evaluated during parsing phase *)
               (C-Lex.token-kind-directive -> C-Env.env-propagation-directive)))
           Symtab.table
  val empty = Symtab.empty
  val merge = Symtab.join (K #2)
);
end
›
```

ML — *Isabelle-C.C-Environment* ‹

```
structure C-Hook =
struct
fun add-stream0 (syms-shift, syms, ml-exec) stream-hook =
  case
     fold (fn - => fn (eval1, eval2) =>
         (case eval2 of e2 :: eval2 => (e2, eval2)
                      | [] => ([], []))
         |>> (fn e1 => e1 :: eval1))
       syms-shift
       ([], stream-hook)
  of (eval1, eval2) => fold cons
                           eval1
                           (case eval2 of e :: es => ((syms-shift, syms, ml-exec) :: e) :: es
```

```
                                    | [] => [[(syms-shift, syms, ml-exec)]])

fun advance00 stack ml-exec =
  case ml-exec of
    (-, C-Env.Bottom-up (C-Env.Exec-annotation exec), env-dir, -) =>
      (fn arg => C-Env.map-env-tree (C-Stack.stack-exec env-dir (stack, #env-lang arg) (exec
NONE))
                                    arg)
  | (-, C-Env.Bottom-up (C-Env.Exec-directive exec), env-dir, -) =>
     C-Env.map-env-lang-tree (curry (exec NONE env-dir))
  | ((pos, -), C-Env.Top-down exec, env-dir, -) =>
    tap (fn - => warning (Missing navigation, evaluating as bottom−up style instead of top−down
                          ^ Position.here pos))
     #>
      (fn arg => C-Env.map-env-tree (C-Stack.stack-exec env-dir (stack, #env-lang arg) (exec
NONE))
                                    arg)

fun advance0 stack (-, syms-reduce, ml-exec) =
  let
    val len = length syms-reduce
  in
    if len = 0 then
      I #>> advance00 stack ml-exec
    else
      let
        val len = len − 1
      in
        fn (arg, stack-ml) =>
          if length stack-ml − 2 <= len then
             ( C-Env.map-stream-hook-excess
                 (add-stream0 (map-range I (len − length stack-ml + 2), syms-reduce, ml-exec))
                 arg
             , stack-ml)
             |> tap (fn - => warning (Navigation out of bounds,
                                      ^ (if length stack-ml <= len then maximum depth
                                                                   else internal value)
                                      ^ reached (
                                      ^ Int.toString (len − length stack-ml + 3)
                                      ^ in excess)
                                      ^ Position.here (Symbol-Pos.range syms-reduce
                                                       |> Position.range-position)))
          else
            (arg, nth-map len (cons ml-exec) stack-ml)
      end
  end

fun advance stack = (fn f => fn (arg, stack-ml) => f (#stream-hook arg) (arg, stack-ml))
 (fn [] => I
```

```
  | l :: ls => fold-rev (advance0 stack) l #>> C-Env.map-stream-hook (K ls))

fun add-stream exec =
  C-Env.map-stream-hook (add-stream0 exec)
end
›

ML — Isabelle-C.C-Lexer-Language ‹
structure C-Grammar-Lexer : ARG-LEXER1 =
struct
structure LALR-Lex-Instance =
struct
  type ('a,'b) token = ('a, 'b) C-Grammar.Tokens.token
  type pos = Position.T
  type arg = C-Grammar.Tokens.arg
  type svalue0 = C-Grammar.Tokens.svalue0
  type svalue = arg -> svalue0 * arg
  type state = C-Grammar.ParserData.LALR-Table.state
end

type stack =
              (LALR-Lex-Instance.state,  LALR-Lex-Instance.svalue0,  LALR-Lex-Instance.pos)
C-Env.stack'

fun makeLexer ((stack, stack-ml, stack-pos, stack-tree), arg) =
  let val (token, arg) = C-Env-Ext.map-stream-lang' (fn (st, []) => (NONE, (st, []))
                                                      | (st, x :: xs) => (SOME x, (st, xs)))
                                                    arg
      fun return0' f =
        (arg, stack-ml)
        |> C-Hook.advance stack
        |> f
        |> (fn (arg, stack-ml) => rpair ((stack, stack-ml, stack-pos, stack-tree), arg))
      fun return0 x = — Warning: C-Hook.advance must not be early evaluated here, as it might
generate undesirable markup reporting (in annotation commands).
                    — Todo: Arrange C-Hook.advance as a pure function, so that the overall could
be eta-simplified.
        return0' I x
      val encoding = fn C-Lex.Encoding-L => true | - => false
      open C-Ast
      fun token-err pos1 pos2 src =
        C-Grammar-Tokens.token-of-string
          (C-Grammar.Tokens.error (pos1, pos2))
          (ClangCVersion0 (From-string src))
          (CChar (From-char-hd 0) false)
          (CFloat (From-string src))
          (CInteger 0 DecRepr (Flags 0))
          (CString0 (From-string src, false))
          (Ident (From-string src, 0, OnlyPos NoPosition (NoPosition, 0)))
```

```
        src
        pos1
        pos2
        src
    open C-Scan
in
  case token
  of NONE =>
      return0 (C-Grammar.Tokens.x25-eof (Position.none, Position.none))
   | SOME (Left (antiq-raw, l-antiq)) =>
      makeLexer
       ( (stack, stack-ml, stack-pos, stack-tree)
       , (arg, false)
         |> fold (fn C-Env.Antiq-stack (-, C-Env.Parsing ((syms-shift, syms), ml-exec)) =>
                      I #>> C-Hook.add-stream (syms-shift, syms, ml-exec)
                   | C-Env.Antiq-stack (-, C-Env.Never) => I ##> K true
                   | - => I)
                 l-antiq
         |> (fn (arg, false) => arg
              | (arg, true) => C-Env-Ext.map-stream-ignored (cons (Left antiq-raw)) arg))
   | SOME (Right (tok as C-Lex.Token (-, (C-Lex.Directive dir, -)))) =>
      makeLexer
       ( (stack, stack-ml, stack-pos, stack-tree)
       , arg
         |> let val context = C-Env-Ext.get-context arg
            in
             fold (fn dir-tok =>
                   C-Hook.add-stream
                    ( []
                    , []
                    , ( Position.no-range
                      , C-Env.Bottom-up (C-Env.Exec-directive
                                           (dir |> (case Symtab.lookup
                                                           (C-Context0.Directives.get context)
                                                           (C-Lex.content-of dir-tok)
                                                    of NONE => K (K (K I))
                                                     | SOME (-, -, (-, exec)) => exec)))
                      , Symtab.empty
                      , true)))
                  (C-Lex.directive-cmds dir)
            end
        |> C-Env-Ext.map-stream-ignored (cons (Right tok)))
   | SOME (Right (C-Lex.Token ((pos1, pos2), (tok, src)))) =>
     case tok of
       C-Lex.String (C-Lex.Encoding-file (SOME err), -) =>
       return0' (apfst
                  (C-Env.map-env-tree (C-Env.map-error-lines (cons (err ^ Position.here pos1)))))
                  (token-err pos1 pos2 src)
     | - =>
```

```
        return0
          (case tok of
             C-Lex.Char (b, [c]) =>
              C-Grammar.Tokens.cchar
                 (CChar (From-char-hd (case c of Left (c, -) => c | - => chr 0)) (encoding b),
pos1, pos2)
           | C-Lex.String (b, s) =>
             C-Grammar.Tokens.cstr
              (CString0 ( From-string ( implode (map (fn Left (s, -) => s | Right - => chr 0)
s))
                                      , encoding b)
                        , pos1
                        , pos2)
           | C-Lex.Integer (i, repr, flag) =>
             C-Grammar.Tokens.cint
              ( CInteger i (case repr of C-Lex.Repr-decimal => DecRepr0
                                       | C-Lex.Repr-hexadecimal => HexRepr0
                                       | C-Lex.Repr-octal => OctalRepr0)
                 (C-Lex.read-bin
                   (fold (fn flag =>
                           map (fn (bit, flag0) =>
                                 ( if flag0 = (case flag of
                                                 C-Lex.Flag-unsigned => FlagUnsigned0
                                               | C-Lex.Flag-long => FlagLong0
                                               | C-Lex.Flag-long-long => FlagLongLong0
                                               | C-Lex.Flag-imag => FlagImag0)
                                   then 1
                                   else bit
                                 , flag0)))
                         flag
                         ([FlagUnsigned, FlagLong, FlagLongLong, FlagImag] |> rev
                                                                           |> map (pair 0))
                    |> map #1)
                  |> Flags)
               , pos1
               , pos2)
           | C-Lex.Float s =>
             C-Grammar.Tokens.cfloat (CFloat (From-string (implode (map #1 s))), pos1, pos2)
           | C-Lex.Ident - =>
             let val (name, arg) = C-Grammar-Rule-Lib.getNewName arg
                 val ident0 = C-Grammar-Rule-Lib.mkIdent
                              (C-Grammar-Rule-Lib.posOf' false (pos1, pos2))
                              src
                              name
             in if C-Grammar-Rule-Lib.isTypeIdent src arg then
                  C-Grammar.Tokens.tyident (ident0, pos1, pos2)
                else
                  C-Grammar.Tokens.ident (ident0, pos1, pos2)
             end
```

```
        | - => token-err pos1 pos2 src)
  end
end
›
```

This is where the instancing of the parser functor (from *Isabelle-C.C-Parser-Language*) with the lexer (from *Isabelle-C.C-Lexer-Language*) actually happens ...

ML — *Isabelle-C.C-Parser-Language* ‹

```
structure C-Grammar-Parser =
  LALR-Parser-Join (structure LrParser = LALR-Parser-Eval
                    structure ParserData = C-Grammar.ParserData
                    structure Lex = C-Grammar-Lexer)
›
```

ML — `~~/src/Pure/ML/ml_compiler.ML` ‹

```
structure C-Language = struct

open C-Env

fun exec-tree write msg (Tree ({rule-pos, rule-type}, l-tree)) =
  case rule-type of
    Void => write msg rule-pos VOID NONE
  | Shift => write msg rule-pos SHIFT NONE
  | Reduce (rule-static, (rule0, vacuous, rule-antiq)) =>
      write
        msg
        rule-pos
        (REDUCE  ^ Int.toString rule0 ^   ^ (if vacuous then X else O))
        (SOME (C-Grammar-Rule.string-reduce rule0 ^   ^ C-Grammar-Rule.type-reduce rule0))
      #> (case rule-static of SOME rule-static => rule-static #>> SOME | NONE => pair
NONE)
      #-> (fn env-lang =>
            fold (fn (stack0, env-lang0, (-, C-Env.Top-down exec, env-dir, -)) =>
                    C-Stack.stack-exec env-dir
                                       (stack0, Option.getOpt (env-lang, env-lang0))
                                       (exec (SOME rule0))
                  | - => I)
                 rule-antiq)
      #> fold (exec-tree write (msg ^ )) l-tree

fun exec-tree' l env-tree = env-tree
  |> fold (exec-tree let val ctxt = Context.proof-of (#context env-tree)
                         val write =
                           if Config.get ctxt C-Options.parser-trace
                              andalso Context-Position.is-visible ctxt
                           then fn f => tap (tracing o f) else K I
                     in fn msg => fn (p1, p2) => fn s1 => fn s2 =>
                       write (fn - => msg ^ s1 ^   ^ Position.here p1 ^   ^ Position.here p2
```

```
                                        ^(case s2 of SOME s2 =>    ^s2 | NONE => ))
                    end
                    )
          l

fun uncurry-context f pos = uncurry (fn (stack, stack-ml, stack-pos, stack-tree) =>
  (* executing stack-tree *)
   (fn arg => map-env-tree' (f pos stack stack-tree (#env-lang arg)) arg)
  #> apfst (pair (stack, stack-ml, stack-pos, stack-tree)))

fun eval env-lang start err accept stream-lang =
 make env-lang stream-lang
 #> C-Grammar-Parser.makeLexer
 #> C-Grammar-Parser.parse
     ( 0
     , uncurry-context (fn (next-pos1, next-pos2) => fn stack => fn stack-tree => fn env-lang
=>
         C-Env.map-reports-text
          (cons ( ( Position.range-position (case hd stack of (-, (-, pos1, pos2)) =>
                                                              (pos1, pos2))
                  , Markup.bad ())
                , )
          #> (case rev (tl stack) of
                - :: - :: stack =>
              append
                (map-filter
                  (fn (pos1, pos2) =>
                   if Position.offset-of pos1 = Position.offset-of pos2
                   then NONE
                   else SOME ((Position.range-position (pos1, pos2), Markup.intensify), ))
                  ((next-pos1, next-pos2)
                   :: map (fn (-, (-, pos1, pos2)) => (pos1, pos2)) stack))
             | - => I))
         #> exec-tree' (rev stack-tree)
         #> err
             env-lang
             stack
             (Position.range-position
               (case hd stack-tree of Tree ({rule-pos = (rule-pos1, -), ...}, -) =>
                                      (rule-pos1, next-pos2))))
     , Position.none
     , start
     , uncurry-context (fn - => fn stack => fn stack-tree => fn env-lang =>
         exec-tree' stack-tree
         #> accept env-lang (stack |> hd |> C-Stack.map-svalue0 C-Grammar-Rule.reduce0))
     , fn (stack, arg) => arg |> map-rule-input (K stack)
                              |> map-rule-output (K empty-rule-output)
     , fn (rule0, stack0, pre-ml) => fn arg =>
         let val rule-output = #rule-output arg
```

```
            val env-lang = #env-lang arg
            val (delayed, actual) =
              if #output-vacuous rule-output
              then let fun f (-, -, -, to-delay) = to-delay
                       in (map (filter f) pre-ml, map (filter-out f) pre-ml) end
              else ([], pre-ml)
            val actual = flat (map rev actual)
          in
            ( (delayed, map (fn x => (stack0, env-lang, x)) actual, rule-output)
            , fold (fn (-, C-Env.Bottom-up (C-Env.Exec-annotation exec), env-dir, -) =>
                       C-Env.map-env-tree
                         (C-Stack.stack-exec env-dir (stack0, env-lang) (exec (SOME rule0)))
                     | (-, C-Env.Bottom-up (C-Env.Exec-directive exec), env-dir, -) =>
                       C-Env.map-env-lang-tree (curry (exec (SOME rule0) env-dir))
                     | - => I)
                   actual
                   arg)
          end)
 #>
 (fn (stream, (((stack, stack-ml, stack-pos, stack-tree), user), arg)) =>
  let
    fun shift-max acc stream =
      let val (tok, stream) = C-Grammar-Parser.Stream.get stream
      in
       if LALR-Parser-Eval.Token.sameToken (tok, C-Grammar.Tokens.x25-eof (Position.none,
Position.none)) then
          (acc, stream)
        else
          shift-max (tok :: acc) stream
      end

    (* executing stack-ml *)
    val arg = fold (fold-rev (C-Hook.advance00 stack)) stack-ml arg

    (* executing stream-lang *)
    val (-, (-, ((stack, stack-ml, -, -), arg))) =
      shift-max [] (stream, ((stack, [[], []], stack-pos, stack-tree), arg))
  in
    arg
    (* executing stream-hook *)
    |> (fn arg =>
        fold (uncurry
              (fn pos =>
                fold-rev (fn (syms-shift, syms-reduce, ml-exec) =>
                             tap (fn - => warning (Navigation out of bounds,\
                                                   \ maximum breadth reached (
                                                  ^ Int.toString (pos + 1)
                                                  ^  in excess)
                                                  ^ Position.here (Symbol-Pos.range syms-shift
```

```
                                                |> Position.range-position)))
                        #> C-Hook.advance0 stack (syms-shift, syms-reduce, ml-exec))))
          (map-index I (#stream-hook arg))
          (arg, stack-ml)
      |> fst)

  (* executing stream-hook-excess *)
  |> (fn arg => fold (fold-rev (fn (-, -, ml-exec) => C-Hook.advance00 stack ml-exec))
                (#stream-hook-excess arg)
                arg)

  (**)
  |> pair user o #env-tree
 end)
end
›
```

3.8.2 Full Evaluation Engine (Core Language with Annotations)

ML — `~~/src/Pure/ML/ml_context.ML`

```
‹
structure C-Context =
struct
fun fun-decl a v s ctxt =
 let
  val (b, ctxt') = ML-Context.variant a ctxt;
  val env = fun ^ b ^ ^ v ^ = ^ s ^ ^ v ^ ;\n;
  val body = ML-Context.struct-name ctxt ^ . ^ b;
  fun decl (-: Proof.context) = (env, body);
 in (decl, ctxt') end;

(* parsing *)

local

fun scan-antiq context syms =
 let val keywords = C-Thy-Header.get-keywords' (Context.proof-of context)
 in ( C-Parse-Read.read-antiq'
      keywords
      (C-Parse.!!! (Scan.trace (C-Annotation.parse-command (Context.theory-of context))
                >> (I #>> C-Env.Antiq-stack)))
      syms
   , C-Parse-Read.read-with-commands'0 keywords syms)
 end

fun print0 s =
 maps
  (fn C-Lex.Token (-, (t as C-Lex.Directive d, -)) =>
```

```
      (s ^ @{make-string} t) :: print0 (s ^ "  ") (C-Lex.token-list-of d)
    | C-Lex.Token (-, t) =>
      [case t of (C-Lex.Char -, -) => Text Char
               | (C-Lex.String -, -) => Text String
               | - => let val t' = @{make-string} (#2 t)
                      in
                        if String.size t' <= 2 then @{make-string} (#1 t)
                        else
                          s ^ @{make-string} (#1 t) ^
                            ^ (String.substring (t', 1, String.size t' - 2)
                               |> Markup.markup Markup.intensify)
                      end]

val print = tracing o cat-lines o print0

open C-Scan

fun markup-directive ty = C-Grammar-Rule-Lib.markup-make (K NONE) (K ()) (K ty)

in

fun markup-directive-command data =
  markup-directive
    directive command
    (fn cons' => fn def =>
     fn C-Ast.Left - =>
          cons' (Markup.keyword-properties (if def then Markup.free else Markup.keyword1))
      | C-Ast.Right (-, msg, f) => tap (fn - => Output.information msg)
                                   #> f
                                   #> cons' (Markup.keyword-properties Markup.free))
    data

fun directive-update (name, pos) f tab =
  let val pos = [pos]
      val data = (pos, serial (), f)
      val - = Position.reports-text
              (markup-directive-command (C-Ast.Left (data, C-Env-Ext.list-lookup tab name))
                                        pos
                                        name
                                        [])
  in Symtab.update (name, data) tab end

fun markup-directive-define in-direct =
  C-Env.map-reports-text ooo
  markup-directive
    directive define
    (fn cons' => fn def => fn err =>
        (if def orelse in-direct then I else cons' Markup.language-antiquotation)
      #> (case err of C-Ast.Left - => I
```

```
                | C-Ast.Right (-, msg, f) => tap (fn - => Output.information msg) #> f)
    #> (if def then cons' Markup.free else if in-direct then I else cons' Markup.antiquote))


(* evaluation *)

fun eval env start err accept (ants, ants-err) {context, reports-text, error-lines} =
  let val error-lines = ants-err error-lines
      fun scan-comment tag pos (antiq as {explicit, body, ...}) cts =
          let val (res, l-comm) = scan-antiq context body
          in
            Left
              ( tag
              , antiq
              , l-comm
              , if forall (fn Right - => true | - => false) res then
                  let val (l-msg, res) =
                        split-list (map-filter (fn Right (msg, l-report, l-tok) =>
                                                   SOME (msg, (l-report, l-tok))
                                                 | - => NONE)
                                               res)
                      val (l-report, l-tok) = split-list res
                  in [( C-Env.Antiq-none
                        (C-Lex.Token
                         (pos, ( (C-Lex.Comment o C-Lex.Comment-suspicious o SOME)
                                 ( explicit
                                 , cat-lines l-msg
                                 , if explicit then flat l-report else [])
                               , cts)))
                     , l-tok)]
                  end
                else
                  map (fn Left x => x
                        | Right (msg, l-report, tok) =>
                            (C-Env.Antiq-none
                              (C-Lex.Token
                                ( C-Token.range-of [tok]
                                , ( (C-Lex.Comment o C-Lex.Comment-suspicious o SOME)
                                      (explicit, msg, l-report)
                                  , C-Token.content-of tok)))
                            , [tok]))
                      res)
          end

      val ants = map (fn C-Lex.Token (pos, (C-Lex.Comment (C-Lex.Comment-formal antiq),
cts)) =>
                         scan-comment C-Env.Comment-language pos antiq cts
                       | tok => Right tok)
                  ants
```

```
fun map-ants f1 f2 = maps (fn Left x => f1 x | Right tok => f2 tok)

val ants-none =
    map-ants (fn (-, -, -, l) => maps (fn (C-Env.Antiq-none x, -) => [x] | - => []) l)
        (K [])
        ants

val - = Position.reports (maps (fn Left (-, -, -, [(C-Env.Antiq-none -, -)]) => []
                            | Left (-, {start, stop, range = (pos, -), ...}, -, -) =>
                              (case stop of SOME stop => cons (stop, Markup.antiquote)
                                          | NONE => I)
                                [(start, Markup.antiquote),
                                 (pos, Markup.language-antiquotation)]
                            | - => [])
                          ants);
val - =
  Position.reports-text
    (maps C-Lex.token-report ants-none
     @ maps (fn Left (-, -, -, [(C-Env.Antiq-none -, -)]) => []
              | Left (-, -, l, ls) =>
                  maps (fn (C-Env.Antiq-stack (pos, -), -) => pos | - => []) ls
                  @ maps (maps (C-Token.reports (C-Thy-Header.get-keywords
                                                   (Context.theory-of context))))
                         (l :: map #2 ls)
              | - => [])
          ants);
val error-lines = C-Lex.check ants-none error-lines;

val ((ants, {context, reports-text, error-lines}), env) =
  C-Env-Ext.map-env-directives'
    (fn env-dir =>
      let val (ants, (env-dir, env-tree)) =
        fold-map
          let
            fun subst-directive tok (range1 as (pos1, -)) name (env-dir, env-tree) =
              case Symtab.lookup env-dir name of
                NONE => (Right (Left tok), (env-dir, env-tree))
              | SOME (data as (-, -, (exec-toks, exec-antiq))) =>
                  env-tree
                  |> markup-directive-define
                      false
                      (C-Ast.Right ([pos1], SOME data))
                      [pos1]
                      name
                  |> (case exec-toks of
                        Left exec-toks =>
                          C-Env.map-context' (exec-toks (name, range1))
                          #> apfst
```

```
                                (fn toks =>
                                  (toks, Symtab.update (name, ( #1 data
                                                              , #2 data
                                                              , (Right toks, exec-antiq)))
                                                       env-dir))
                            | Right toks => pair (toks, env-dir))
                        ||> C-Env.map-context (exec-antiq (name, range1))
                        |-> (fn (toks, env-dir) =>
                              pair (Right (Right (pos1, map (C-Lex.set-range range1) toks)))
                              o pair env-dir)
                in
                 fn Left (tag, antiq, toks, l-antiq) =>
                     fold-map
                      (fn antiq as (C-Env.Antiq-stack (-, C-Env.Lexing (-, exec)), -) =>
                            apsnd (C-Stack.stack-exec0 (exec C-Env.Comment-language)) #> pair
antiq
                        | (C-Env.Antiq-stack
                            (rep, C-Env.Parsing (syms, (range, exec, -, skip))), toks) =>
                            (fn env as (env-dir, -) =>
                              ( ( C-Env.Antiq-stack
                                    (rep, C-Env.Parsing (syms, (range, exec, env-dir, skip)))
                                , toks)
                              , env))
                        | antiq => pair antiq)
                      l-antiq
                     #> apfst (fn l-antiq => Left (tag, antiq, toks, l-antiq))
                  | Right tok =>
                  case tok of
                    C-Lex.Token (-, (C-Lex.Directive dir, -)) =>
                      pair false
                      #> fold
                          (fn dir-tok =>
                            let val name = C-Lex.content-of dir-tok
                                val pos1 = [C-Lex.pos-of dir-tok]
                            in
                              fn env-tree as (-, (-, {context = context, ...})) =>
                              let val data = Symtab.lookup (C-Context0.Directives.get context) name
                              in
                              env-tree
                              |> apsnd (apsnd (C-Env.map-reports-text (markup-directive-command
                                                                        (C-Ast.Right (pos1, data))
                                                                        pos1
                                                                        name)))
                             |> (case data of NONE => I | SOME (-, -, (exec, -)) => exec dir #>
#2)
                              end
                            end)
                          (C-Lex.directive-cmds dir)
                      #> snd
```

```
                #> tap
                    (fn - =>
                      app (fn C-Lex.Token ( (pos, -)
                                          , (C-Lex.Comment (C-Lex.Comment-formal -), -)) =>
                             (Position.reports-text [((pos, Markup.ML-comment), )];
                              (* not yet implemented *)
                              warning (Ignored annotation in directive
                                       ^ Position.here pos))
                            | - => ())
                          (C-Lex.token-list-of dir))
                #> pair (Right (Left tok))
              | C-Lex.Token (pos, (C-Lex.Keyword, cts)) => subst-directive tok pos cts
              | C-Lex.Token (pos, (C-Lex.Ident -, cts)) => subst-directive tok pos cts
              | - => pair (Right (Left tok))
            end
            ants
            ( env-dir
            , {context = context, reports-text = reports-text, error-lines = error-lines})
        in ((ants, env-tree), env-dir) end)
      env

  val ants-stack =
    map-ants (single o Left o (fn (-, a, -, l) => (a, maps (single o #1) l)))
             (map Right o (fn Left tok => [tok] | Right (-, toks) => toks))
             ants
  val - =
      Position.reports-text (maps (fn Right (Left tok) => C-Lex.token-report tok
                                    | Right (Right (pos, [])) => [((pos, Markup.intensify), )]
                                    | - => [])
                                  ants);
  val ctxt = Context.proof-of context
  val () = if Config.get ctxt C-Options.lexer-trace andalso Context-Position.is-visible ctxt
           then print (map-filter (fn Right x => SOME x | - => NONE) ants-stack)
           else ()
in
 C-Language.eval env
               start
               err
               accept
               ants-stack
               {context = context, reports-text = reports-text, error-lines = error-lines}
end


(* derived versions *)

fun eval' env start err accept ants =
  Context.>>> (fn context =>
           C-Env-Ext.context-map'
```

```
              (eval (env context) (start context) err accept ants
               #> apsnd (tap (Position.reports-text o #reports-text)
                        #> tap (#error-lines #> (fn [] => () | l => error (cat-lines (rev l))))
                        #> (C-Env.empty-env-tree o #context)))
              context)
end;

fun eval-source env start err accept source =
  eval' env (start source) err accept (C-Lex.read-source source);

fun eval-source' env start err accept source =
  eval env (start source) err accept (C-Lex.read-source source);

fun eval-in o-context env start err accept toks =
  Context.setmp-generic-context o-context
    (fn () => eval' env start err accept toks) ();

fun expression struct-open range name constraint body ants context = context |>
  ML-Context.exec
    let val verbose = Config.get (Context.proof-of context) C-Options.ML-verbose
    in fn () =>
      ML-Context.eval (ML-Compiler.verbose verbose ML-Compiler.flags) (#1 range)
       (ML-Lex.read (Context.put-generic-context (SOME (let open " ^ struct-open ^ " val ) @
                                                        ML-Lex.read-range range name @
        ML-Lex.read (: " ^ constraint ^ " =) @ ants @
        ML-Lex.read (in " ^ body ^ " end (Context.the-generic-context ()))));))
    end;
end
›

end
```

3.9 Interface: Inner and Outer Commands

```
theory C-Command
  imports C-Eval
  keywords C :: thy-decl % ML
       and C-file :: thy-load % ML
       and C-export-boot :: thy-decl % ML
       and C-export-file :: thy-decl
       and C-prf :: prf-decl % proof
       and C-val :: diag % ML
begin
```

3.9.1 Parsing Entry-Point: Error and Acceptance Cases

ML — `~~/src/Pure/Tools/ghc.ML`

```
‹
```

```
structure C-Serialize =
struct

(** string literals **)

fun print-codepoint c =
  (case c of
    10 => \\n
  | 9 => \\t
  | 11 => \\v
  | 8 => \\b
  | 13 => \\r
  | 12 => \\f
  | 7 => \\a
  | 27 => \\e
  | 92 => \\\\
  | 63 => \\?
  | 39 => \\'
  | 34 => \\\
  | c =>
    if c >= 32 andalso c < 127 then chr c
    else error Not yet implemented);

fun print-symbol sym =
  (case Symbol.decode sym of
    Symbol.Char s => print-codepoint (ord s)
  | Symbol.UTF8 s => UTF8.decode-permissive s |> map print-codepoint |> implode
  | Symbol.Sym s => \\092< ^ s ^ >
  | Symbol.Control s => \\092<^ ^ s ^ >
  | - => translate-string (print-codepoint o ord) sym);

val print-string = quote o implode o map print-symbol o Symbol.explode;
end
›
```

ML — `~~/src/Pure/Tools/generated_files.ML`

```
‹
structure C-Generated-Files =
struct

val c-dir = C;
val c-ext = c;
val c-make-comment = enclose /* */;

(** context data **)

(* file types *)
```

```
fun get-file-type ext =
  if ext =  then error Bad file extension
  else if c-ext = ext then ()
  else error (Unknown file type for extension  ^ quote ext);



(** Isar commands **)

(* generate-file *)

fun generate-file (binding, src-content) lthy =
  let
    val (path, pos) = Path.dest-binding binding;
    val () =
      get-file-type (#2 (Path.split-ext path))
        handle ERROR msg => error (msg ^ Position.here pos);
    val header = c-make-comment  generated by Isabelle ;
    val content = header ^ \n ^ src-content;
  in lthy |> (Local-Theory.background-theory o Generated-Files.add-files) (binding, Bytes.string
content) end;



(** concrete file types **)

val - =
  Theory.setup
    (Generated-Files.file-type binding ‹C›
      {ext = c-ext,
       make-comment = c-make-comment,
       make-string = C-Serialize.print-string});
end
›

ML — Isabelle-C.C-Eval ‹

signature C-MODULE =
 sig
    structure Data-Accept    : GENERIC-DATA
    structure Data-In-Env    : GENERIC-DATA
    structure Data-In-Source : GENERIC-DATA
    structure Data-Term      : GENERIC-DATA

    structure C-Term:
      sig
        val key0-default: string
        val key0-expression: string
        val key0-external-declaration: string
```

```
    val key0-statement: string
    val key0-translation-unit: string
    val key-default: Input.source
    val key-expression: Input.source
    val key-external-declaration: Input.source
    val key-statement: Input.source
    val key-translation-unit: Input.source
    val map-default: (C-Grammar-Rule.ast-generic -> C-Env.env-lang -> local-theory ->
term) -> theory -> theory
    val map-expression: (C-Grammar-Rule-Lib.CExpr -> C-Env.env-lang -> local-theory
-> term) -> theory -> theory
    val map-external-declaration:
      (C-Grammar-Rule-Lib.CExtDecl -> C-Env.env-lang -> local-theory -> term) ->
theory -> theory
    val map-statement: (C-Grammar-Rule-Lib.CStat -> C-Env.env-lang -> local-theory ->
term) -> theory -> theory
    val map-translation-unit:
      (C-Grammar-Rule-Lib.CTranslUnit -> C-Env.env-lang -> local-theory -> term) ->
theory -> theory
    val tok0-expression: string * ('a * 'a -> (C-Grammar.Tokens.svalue, 'a)
LALR-Parser-Eval.Token.token)
    val tok0-external-declaration: string * ('a * 'a -> (C-Grammar.Tokens.svalue, 'a)
LALR-Parser-Eval.Token.token)
    val tok0-statement: string * ('a * 'a -> (C-Grammar.Tokens.svalue, 'a)
LALR-Parser-Eval.Token.token)
    val tok0-translation-unit: string * ('a * 'a -> (C-Grammar.Tokens.svalue, 'a)
LALR-Parser-Eval.Token.token)
    val tok-expression: Input.source * ('a * 'a -> (C-Grammar.Tokens.svalue, 'a)
LALR-Parser-Eval.Token.token)
    val tok-external-declaration: Input.source * ('a * 'a -> (C-Grammar.Tokens.svalue, 'a)
LALR-Parser-Eval.Token.token)
    val tok-statement: Input.source * ('a * 'a -> (C-Grammar.Tokens.svalue, 'a)
LALR-Parser-Eval.Token.token)
    val tok-translation-unit: Input.source * ('a * 'a -> (C-Grammar.Tokens.svalue, 'a)
LALR-Parser-Eval.Token.token)
    val tokens: (string * ('a * 'a -> (C-Grammar.Tokens.svalue, 'a)
LALR-Parser-Eval.Token.token)) list
  end
 structure C-Term':
  sig
    val accept:
      local-theory ->
        (Input.source * (Position.range -> (C-Grammar.Tokens.svalue, Position.T)
LALR-Parser-Eval.Token.token)) option ->
        (Input.source -> Context.generic -> (C-Grammar.Tokens.svalue, Position.T)
LALR-Parser-Eval.Token.token) *
          (Data-In-Env.T ->
            'a * (C-Grammar-Rule.ast-generic * 'b * 'c) ->
              {context: Context.generic, error-lines: 'd, reports-text: 'e} ->
```

```
                term * {context: Context.generic, error-lines: 'd, reports-text: 'e})
      val err:
        C-Env.env-lang ->
          (LALR-Table.state * (C-Grammar-Parser.svalue0 * Position.T * Position.T)) list
->
           Position.T ->
            {context: Context.generic, error-lines: string list, reports-text: Position.report-text
list} ->
                 term * {context: Context.generic, error-lines: string list, reports-text: Posi-
tion.report-text list}
      val eval-in:
        Input.source ->
          Context.generic ->
            (Context.generic ->
               C-Env.env-lang) ->
                 (Input.source * (Position.range -> (C-Grammar.Tokens.svalue, Position.T)
LALR-Parser-Eval.Token.token)) option
                  -> C-Lex.token list * (C-Env.error-lines -> string list) -> term
      val parse-translation:
           ('a * (Input.source * (Position.range -> (C-Grammar.Tokens.svalue, Position.T)
LALR-Parser-Eval.Token.token)) option)
          list
            -> ('a * (Proof.context -> term list -> term)) list
    end

  val accept:
    Data-In-Env.T ->
      'a * (C-Grammar-Rule.ast-generic * 'b * 'c) ->
        {context: Context.generic, error-lines: 'd, reports-text: 'e} ->
          unit * {context: Context.generic, error-lines: 'd, reports-text: 'e}
  val accept0:
    (Context.generic -> C-Grammar-Rule.ast-generic -> Data-In-Env.T -> Context.generic
-> 'a) ->
       Data-In-Env.T -> C-Grammar-Rule.ast-generic -> Context.generic -> 'a
  val c-enclose: string -> string -> Input.source -> C-Lex.token list * (string list -> string
list)
  val env: Context.generic -> Data-In-Env.T
  val env0: Proof.context -> Data-In-Env.T
  val err:
    C-Env.env-lang ->
      (LALR-Table.state * (C-Grammar-Parser.svalue0 * Position.T * Position.T)) list ->
        Position.T ->
            {context: Context.generic, error-lines: string list, reports-text: Position.report-text
list} ->
                 unit * {context: Context.generic, error-lines: string list, reports-text: Posi-
tion.report-text list}
  val err0:
    'a ->
      'b ->
```

```
        Position.T ->
          {context: 'c, error-lines: string list, reports-text: 'd} ->
            {context: 'c, error-lines: string list, reports-text: 'd}
  val eval-in: Input.source -> Context.generic option -> C-Lex.token list * (C-Env.error-lines
-> string list) -> unit
  val eval-source: Input.source -> unit
  val exec-eval: Input.source -> Context.generic -> Context.generic
   val start: Input.source -> Context.generic -> (C-Grammar.Tokens.svalue, Position.T)
LALR-Parser-Eval.Token.token


  (* toplevel command semantics of Isabelle-C *)
  val C: Input.source -> Context.generic -> Context.generic
  val C': C-Env.env-lang option -> Input.source -> Context.generic -> Context.generic
  val C-export-boot: Input.source -> Context.generic -> generic-theory
  val C-export-file: Position.T * 'a -> Proof.context -> Proof.context
  val C-prf: Input.source -> Proof.state -> Proof.state

 end

structure C-Module : C-MODULE =
struct

structure Data-In-Source = Generic-Data
(
 type T = Input.source list
 val empty = []
 val merge = K empty
)

structure Data-In-Env = Generic-Data
(
 type T = C-Env.env-lang
 val empty = C-Env.empty-env-lang
 val merge = K empty
)

structure Data-Accept = Generic-Data
(
  type T = C-Grammar-Rule.ast-generic -> C-Env.env-lang -> Context.generic -> Con-
text.generic
 fun empty - - = I
 val merge = #2
)

structure Data-Term = Generic-Data
(
  type T = (C-Grammar-Rule.ast-generic -> C-Env.env-lang -> local-theory -> term)
Symtab.table
```

```
  val empty = Symtab.empty
  val merge = #2
)

(* keys for major syntactic categories *)
structure C-Term =
struct
val key-translation-unit = ‹translation-unit›
val key-external-declaration = ‹external-declaration›
val key-statement = ‹statement›
val key-expression = ‹expression›
val key-default = ‹default›

local
val source-content = Input.source-content #> #1
in
val key0-translation-unit = source-content key-translation-unit
val key0-external-declaration = source-content key-external-declaration
val key0-statement = source-content key-statement
val key0-expression = source-content key-expression
val key0-default = source-content key-default
end

val tok0-translation-unit = (key0-translation-unit, C-Grammar.Tokens.start-translation-unit)
val tok0-external-declaration = ( key0-external-declaration
                                , C-Grammar.Tokens.start-external-declaration)
val tok0-statement = (key0-statement, C-Grammar.Tokens.start-statement)
val tok0-expression = (key0-expression, C-Grammar.Tokens.start-expression)

val tok-translation-unit = (key-translation-unit, C-Grammar.Tokens.start-translation-unit)
val tok-external-declaration = ( key-external-declaration
                               , C-Grammar.Tokens.start-external-declaration)
val tok-statement = (key-statement, C-Grammar.Tokens.start-statement)
val tok-expression = (key-expression, C-Grammar.Tokens.start-expression)

val tokens = [ tok0-translation-unit
             , tok0-external-declaration
             , tok0-statement
             , tok0-expression ]

local
fun map-upd0 key v = Context.theory-map (Data-Term.map (Symtab.update (key, v)))
fun map-upd key start f = map-upd0 key (f o the o start)
in
val map-translation-unit = map-upd key0-translation-unit C-Grammar-Rule.get-CTranslUnit
val map-external-declaration = map-upd key0-external-declaration
C-Grammar-Rule.get-CExtDecl
val map-statement = map-upd key0-statement C-Grammar-Rule.get-CStat
val map-expression = map-upd key0-expression C-Grammar-Rule.get-CExpr
```

```
val map-default = map-upd0 key0-default
end

end

fun env0 ctxt =
  case Config.get ctxt C-Options.starting-env of
    last => Data-In-Env.get (Context.Proof ctxt)
  | empty => C-Env.empty-env-lang
  | s => error (Unknown option:  ^ s ^ Position.here (Config.pos-of C-Options.starting-env))

val env = env0 o Context.proof-of

fun start source context =
 Input.range-of source
 |>
  let val s = Config.get (Context.proof-of context) C-Options.starting-rule
  in case AList.lookup (op =) C-Term.tokens s of
       SOME tok => tok
     | NONE => error (Unknown option:  ^ s
                                       ^ Position.here (Config.pos-of C-Options.starting-rule))
  end

fun err0 - - pos =
  C-Env.map-error-lines (cons (Parser: No matching grammar rule ^ Position.here pos))

val err = pair () oooo err0

fun accept0 f env-lang ast =
  Data-In-Env.put env-lang
  #> (fn context => f context ast env-lang (Data-Accept.get context ast env-lang context))

fun accept env-lang (-, (ast, -, -)) =
  pair () o C-Env.map-context (accept0 (K (K (K I))) env-lang ast)

val eval-source = C-Context.eval-source env start err accept

fun c-enclose bg en source =
  C-Lex.@@ ( C-Lex.@@ (C-Lex.read bg, C-Lex.read-source source)
           , C-Lex.read en);

structure C-Term' =
struct
val err = pair Term.dummy oooo err0

fun accept ctxt start-rule =
  let
    val (key, start) =
      case start-rule of NONE => (C-Term.key-default, start)
```

```
                  | SOME (key, start-rule) =>
                      (key, fn source => fn - => start-rule (Input.range-of source))
    val (key, pos) = Input.source-content key
  in
    ( start
    , fn env-lang => fn (-, (ast, -, -)) =>
        C-Env.map-context'
          (accept0
            (fn context =>
              pair oo (case Symtab.lookup (Data-Term.get context) key of
                         NONE => tap (fn - => warning (Representation function associated to\
                                                       \ \ ^ key ^ \ ^ Position.here pos
                                                       ^ not found (returning a dummy term)))
                                     (fn - => fn - => @{term ()})
                       | SOME f => fn ast => fn env-lang => f ast env-lang ctxt))
            env-lang
            ast))
  end

fun eval-in text context env start-rule =
  let
    val (start, accept) = accept (Context.proof-of context) start-rule
  in
    C-Context.eval-in (SOME context) env (start text) err accept
  end

fun parse-translation l = l |>
  map
  (apsnd
    (fn start-rule => fn ctxt => fn args =>
      let val msg = (case start-rule of NONE => C-Term.key-default
                                      | SOME (key, -) => key)
                  |> Input.source-content |> #1
          fun err () = raise TERM (msg, args)
      in
       case args of
         [(c as Const (syntax-const ‹-constrain›, -)) $ Free (s, -) $ p] =>
          (case Term-Position.decode-position p of
            SOME (pos, -) =>
            c
            $ let val src =
                  uncurry
                    (Input.source false)
                    let val s0 = Symbol-Pos.explode (s, pos)
                        val s = Symbol-Pos.cartouche-content s0
                    in
                      ( Symbol-Pos.implode s
                      , case s of [] => Position.no-range
                                | (-, pos0) :: - => Position.range (pos0, s0 |> List.last |> snd))
```

```
            end
          in
          eval-in
            src
            (case Context.get-generic-context () of
               NONE => Context.Proof ctxt
             | SOME context => Context.mapping I (K ctxt) context)
            (C-Stack.Data-Lang.get #> (fn NONE => env0 ctxt
                                        | SOME (-, env-lang) => env-lang))
            start-rule
            (c-enclose  src)
          end
        $ p
      | NONE => err ())
    | - => err ()
    end))
end

fun eval-in text ctxt = C-Context.eval-in ctxt env (start text) err accept

fun exec-eval source =
  Data-In-Source.map (cons source)
  #> ML-Context.exec (fn () => eval-source source)

fun C-prf source =
  Proof.map-context (Context.proof-map (exec-eval source))
  #> Proof.propagate-ml-env

fun C-export-boot source context =
  context
  |> Config.put-generic ML-Env.ML-environment ML-Env.Isabelle
  |> Config.put-generic ML-Env.ML-write-global true
  |> exec-eval source
  |> Config.restore-generic ML-Env.ML-write-global context
  |> Config.restore-generic ML-Env.ML-environment context
  |> Local-Theory.propagate-ml-env

fun C source =
  exec-eval source
  #> Local-Theory.propagate-ml-env

val C' =
  let
    fun C env-lang src context =
      context
      |> C-Env.empty-env-tree
      |> C-Context.eval-source'
           env-lang
           (fn src => start src context)
```

```
          err
          accept
          src
      |> (fn (-, {context, reports-text, error-lines}) =>
         tap (fn - => case error-lines of [] => () | l => warning (cat-lines (rev l)))
           (C-Stack.Data-Tree.map (curry C-Stack.Data-Tree-Args.merge (reports-text, []))
                                  context))
  in
    fn NONE => (fn src => C (env (Context.the-generic-context ())) src)
     | SOME env-lang => C env-lang
  end

fun C-export-file (pos, -) lthy =
  let
    val c-sources = Data-In-Source.get (Context.Proof lthy)
    val binding =
      Path.binding
        ( Path.appends [ Path.basic C-Generated-Files.c-dir
                       , Path.basic (string-of-int (length c-sources))
                      , lthy |> Proof-Context.theory-of |> Context.theory-base-name |> Path.explode
                             |> Path.ext C-Generated-Files.c-ext ]
        , pos)
  in
    lthy
    |> C-Generated-Files.generate-file (binding, rev c-sources |> map (Input.source-content #>
#1)
                                                              |> cat-lines)
    |> tap (Proof-Context.theory-of
          #> (fn thy => let val file = Generated-Files.get-file thy binding
                        in Generated-Files.export-file thy file;
                           writeln (Export.message thy Path.current);
                           writeln (prefix    (Generated-Files.print-file file))
                        end))
  end
end
›
```

3.9.2 Definitions of C11 Directives as C-commands

Initialization

```
ML — Pure ‹
structure C-Directive :
 sig
    val setup-define:
       Position.T ->
         (C-Lex.token list -> string * Position.range -> Context.generic
          -> C-Lex.token list * Context.generic) ->
         (string * Position.range -> Context.generic -> Context.generic) -> theory -> theory
  end =
```

```
struct
local
fun directive-update keyword data = C-Context.directive-update keyword (data, K (K (K I)))
fun return f (env-cond, env) = ([], (env-cond, f env))
fun directive-update-define pos f-toks f-antiq =
  directive-update (define, pos)
    (return
     o
      (fn C-Lex.Define (-, C-Lex.Group1 ([], [tok3]), NONE, C-Lex.Group1 ([], toks)) =>
          let val map-ctxt =
              case (tok3, toks) of
                (C-Lex.Token ((pos, -), (C-Lex.Ident -, ident)),
                 [C-Lex.Token (-, (C-Lex.Integer (-, C-Lex.Repr-decimal, []), integer))]) =>
                  C-Env.map-context
                    (Context.map-theory
                      (Named-Target.theory-map
                        (Specification.definition-cmd
                          (SOME (Binding.make (ident, pos), NONE, NoSyn))
                          []
                          []
                          (Binding.empty-atts, ident ^ ≡ ^ integer)
                          true
                         #> tap (fn ((-, (-, t)), ctxt) =>
                                   Output.information
                                     (Generating
                                      ^ Pretty.string-of (Syntax.pretty-term ctxt (Thm.prop-of t))
                                      ^ Position.here
                                          (Position.range-position
                                            ( C-Lex.pos-of tok3
                                            , C-Lex.end-pos-of (List.last toks)))))
                         #> #2)))
            | - => I
          in
            fn (env-dir, env-tree) =>
              let val name = C-Lex.content-of tok3
                  val pos = [C-Lex.pos-of tok3]
                  val data = (pos, serial (), (C-Scan.Left (f-toks toks), f-antiq))
              in
                ( Symtab.update (name, data) env-dir
                , env-tree |> C-Context.markup-directive-define
                               false
                               (C-Ast.Left (data, C-Env-Ext.list-lookup env-dir name))
                               pos
                               name
                           |> map-ctxt)
              end
          end
        | C-Lex.Define (-, C-Lex.Group1 ([], [tok3]), SOME (C-Lex.Group1 (- :: toks-bl, -)), -)
          =>
```

```
          tap (fn - => (* not yet implemented *)
                warning (Ignored functional macro directive
                      ^ Position.here
                        (Position.range-position
                          (C-Lex.pos-of tok3, C-Lex.end-pos-of (List.last toks-bl)))))
      | - => I))
in
val setup-define = Context.theory-map o C-Context0.Directives.map ooo directive-update-define

val - =
  Theory.setup
  (Context.theory-map
    (C-Context0.Directives.map
      (directive-update-define here (K o pair) (K I)
       #>
       directive-update (undef, here)
        (return
         o
          (fn C-Lex.Undef (C-Lex.Group2 (-, -, [tok])) =>
                (fn (env-dir, env-tree) =>
                  let val name = C-Lex.content-of tok
                      val pos1 = [C-Lex.pos-of tok]
                      val data = Symtab.lookup env-dir name
                  in ( (case data of NONE => env-dir | SOME - => Symtab.delete name env-dir)
                     , C-Context.markup-directive-define true
                                                 (C-Ast.Right (pos1, data))
                                                 pos1
                                                 name
                                                 env-tree)
                  end)
            | - => I)))))
end
end
›
```

3.9.3 Definitions of C Annotation Commands

Library

ML — `~~/src/Pure/Isar/toplevel.ML` ‹

```
structure C-Inner-Toplevel =
struct
val theory = Context.map-theory
fun local-theory' target f gthy =
  let
    val (finish, lthy) = Target-Context.switch-named-cmd target gthy;
    val lthy' = lthy
      |> Local-Theory.new-group
      |> f false
      |> Local-Theory.reset-group;
```

```
  in finish lthy' end
val generic-theory = I
fun keep'' f = tap (f o Context.proof-of)
end
›

ML — ~~/src/Pure/Isar/isar_cmd.ML ‹
structure C-Inner-Isar-Cmd =
struct


(** theory declarations **)

(* generic setup *)

fun setup0 f-typ f-val src =
 fn NONE =>
    let val setup = setup
    in C-Context.expression
        C-Ast
        (Input.range-of src)
        setup
        (f-typ C-Stack.stack-data
           C-Stack.stack-data-elem -> C-Env.env-lang -> Context.generic -> Context.generic)
        (fn context => \
          \let val (stack, env-lang) = C-Stack.Data-Lang.get' context \
          \in  ^f-val setup stack ^ (stack |> hd) env-lang end context)
        (ML-Lex.read-source src) end
  | SOME rule =>
    let val hook = hook
    in C-Context.expression
        C-Ast
        (Input.range-of src)
        hook
        (f-typ C-Stack.stack-data
              (C-Grammar-Rule.type-reduce rule
             ^ C-Stack.stack-elem -> C-Env.env-lang -> Context.generic -> Context.generic))
        (fn context => \
          \let val (stack, env-lang) = C-Stack.Data-Lang.get' context \
          \in  ^f-val hook
                       stack ^
                      ^(stack \
                        \|> hd \
                        \|> C-Stack.map-svalue0 C-Grammar-Rule.reduce ^ Int.toString rule ^)\
                       \env-lang \
          \end \
          \  context)
        (ML-Lex.read-source src)
    end
```

```
val setup = setup0 (fn a => fn b => a ^ -> ^ b) (fn a => fn b => a ^ ^ b)
val setup' = setup0 (K I) K


(* print theorems, terms, types etc. *)

local

fun string-of-term ctxt s =
  let
    val t = Syntax.read-term ctxt s;
    val T = Term.type-of t;
    val ctxt' = Proof-Context.augment t ctxt;
  in
    Pretty.string-of
      (Pretty.block [Pretty.quote (Syntax.pretty-term ctxt' t), Pretty.fbrk,
        Pretty.str ::, Pretty.brk 1, Pretty.quote (Syntax.pretty-typ ctxt' T)])
  end;

fun print-item string-of (modes, arg) ctxt =
  Print-Mode.with-modes modes (fn () => writeln (string-of ctxt arg)) ();

in

val print-term = print-item string-of-term;

end;

end
›

ML — ~~/src/Pure/Isar/outer_syntax.ML ‹
structure C-Inner-Syntax =
struct
val drop1 = fn C-Scan.Left f => C-Scan.Left (K o f)
          | C-Scan.Right (f, dir) => C-Scan.Right (K o f, dir)

val drop2 = fn C-Scan.Left f => C-Scan.Left (K o K o f)
          | C-Scan.Right (f, dir) => C-Scan.Right (K o K o f, dir)

val bottom-up = C-Env.Bottom-up o C-Env.Exec-annotation

(**)

fun pref-lex name = # ^ name
val pref-bot = I
fun pref-top name = name ^⇓

(**)
```

```
fun command2' cmd f (pos-bot, pos-top) =
  let fun cmd' dir = cmd (C-Scan.Right (f, dir)) Keyword.thy-decl
  in   cmd' bottom-up (pref-bot, pos-bot)
    #> cmd' C-Env.Top-down (pref-top, pos-top)
  end

fun command3' cmd f (pos-lex, pos-bot, pos-top) =
     cmd (C-Scan.Left f) (pref-lex, pos-lex)
  #> command2' (K o cmd) f (pos-bot, pos-top)

fun command2 cmd f (name, pos-bot, pos-top) =
  command2' (fn f => fn kind => fn (name-pref, pos) => cmd f kind (name-pref name, pos))
          f
          (pos-bot, pos-top)

fun command3 cmd f (name, pos-lex, pos-bot, pos-top) =
  command3' (fn f => fn (name-pref, pos) => cmd f (name-pref name, pos))
          f
          (pos-lex, pos-bot, pos-top)

(**)

fun command00 f kind scan name =
  C-Annotation.command'' kind name
    (case f of
       C-Scan.Left f =>
        (fn - =>
          C-Parse.range scan >>
            (fn (src, range) =>
              C-Env.Lexing (range, f src range)))
     | C-Scan.Right (f, dir) =>
         fn ((stack1, (to-delay, stack2)), -) =>
          C-Parse.range scan >>
            (fn (src, range) =>
             C-Env.Parsing ((stack1, stack2), (range, dir (f src range), Symtab.empty, to-delay))))

fun command00-no-range f kind name =
  C-Annotation.command'' kind name
    (case f of
       C-Scan.Left f =>
        (fn (-, range) =>
          Scan.succeed () >>
            K (C-Env.Lexing (range, f range)))
     | C-Scan.Right (f, dir) =>
         fn ((stack1, (to-delay, stack2)), range) =>
          Scan.succeed () >>
           K (C-Env.Parsing ((stack1, stack2), (range, dir (f range), Symtab.empty, to-delay))))
```

```
(**)

fun command' f = command00 (drop1 f) Keyword.thy-decl

fun command f scan = command2 (fn f => fn kind => command00 f kind scan) (K o f)

fun command-range f = command00-no-range f Keyword.thy-decl

val command-range' = command3 (command-range o drop1)

fun command-no-range' f = command00-no-range (drop1 f) Keyword.thy-decl

fun command-no-range f = command2 command00-no-range (K f)

fun command0 f scan = command3 (fn f => command' (drop1 f) scan) f

fun local-command' (name, pos-lex, pos-bot, pos-top) scan f =
  command3' (fn f => fn (name-pref, pos) =>
            command' (drop1 f)
                  (C-Token.syntax' (Parse.opt-target -- scan name-pref))
                  (name-pref name, pos))
          (fn (target, arg) => C-Inner-Toplevel.local-theory' target (f arg))
          (pos-lex, pos-bot, pos-top)

fun local-command'' spec = local-command' spec o K

val command0-no-range = command-no-range' o drop1

fun command0' f kind scan =
  command3 (fn f => command00 (drop2 f) kind scan) f
end
›

ML — ~~/src/Pure/ML/ml_file.ML ‹
structure C-Inner-File =
struct

fun command-c ({lines, pos, ...}: Token.file) =
  C-Module.C (Input.source true (cat-lines lines) (pos, pos));

fun C get-file gthy =
  command-c (get-file (Context.theory-of gthy)) gthy;

fun command-ml environment debug get-file gthy =
  let
    val file = get-file (Context.theory-of gthy);
    val source = Token.file-source file;

      val - = Document-Output.check-comments (Context.proof-of gthy) (Input.source-explode
```

source);

    val flags: *ML-Compiler.flags* =
      {*environment* = *environment*, *redirect* = *true*, *verbose* = *true*,
        debug = *debug*, *writeln* = *writeln*, *warning* = *warning*};
  in
    gthy
    |> *ML-Context.exec* (*fn* () => *ML-Context.eval-source flags source*)
    |> *Local-Theory.propagate-ml-env*
  end;

val ML = *command-ml* ;
val SML = *command-ml ML-Env.SML*;
end;
›

Initialization

setup — *Pure* ‹
C-Thy-Header.add-keywords-minor
  (*maps* (*fn* ((*name*, *pos-lex*, *pos-bot*, *pos-top*), *ty*) =>
        [((*C-Inner-Syntax.pref-lex name*, *pos-lex*), *ty*)
        , ((*C-Inner-Syntax.pref-bot name*, *pos-bot*), *ty*)
        , ((*C-Inner-Syntax.pref-top name*, *pos-top*), *ty*)])
      [((*apply*, ***here***, ***here***, ***here***), *Keyword.command-spec* (*Keyword.prf-script*, [*proof*]))
      , ((*by*, ***here***, ***here***, ***here***), *Keyword.command-spec* (*Keyword.qed*, [*proof*]))
      , ((*done*, ***here***, ***here***, ***here***), *Keyword.command-spec* (*Keyword.qed-script*, [*proof*]))])
›

ML — *Pure* ‹
local
val semi = *Scan.option* (*C-Parse.*$$$;);

structure C-Isar-Cmd =
struct
fun ML source = *ML-Context.exec* (*fn* () =>
                  ML-Context.eval-source (*ML-Compiler.verbose true ML-Compiler.flags*) *source*)
#>
                Local-Theory.propagate-ml-env

fun theorem schematic ((*long*, *binding*, *includes*, *elems*, *concl*), (*l-meth*, *o-meth*)) *int lthy* =
    (*if schematic then Specification.schematic-theorem-cmd else Specification.theorem-cmd*)
      long Thm.theoremK NONE (*K I*) *binding includes elems concl int lthy*
  |> *fold* (*fn m* => *tap* (*fn* - => *Method.report m*) #> *Proof.apply m* #> *Seq.the-result*)
l-meth
  |> (*case o-meth of*
      NONE => *Proof.global-done-proof*
    | *SOME* (*m1*, *m2*) =>
        tap (*fn* - => (*Method.report m1*; *Option.map Method.report m2*))

```
        #> Proof.global-terminal-proof (m1, m2))

fun definition (((decl, spec), prems), params) =
  #2 oo Specification.definition-cmd decl params prems spec

fun declare (facts, fixes) =
  #2 oo Specification.theorems-cmd  [(Binding.empty-atts, flat facts)] fixes
end

local
val long-keyword =
  Parse-Spec.includes >> K  ||
  Parse-Spec.long-statement-keyword;

val long-statement =
  Scan.optional (Parse-Spec.opt-thm-name : --| Scan.ahead long-keyword) Binding.empty-atts
--
  Scan.optional Parse-Spec.includes [] -- Parse-Spec.long-statement
    >> (fn ((binding, includes), (elems, concl)) => (true, binding, includes, elems, concl));

val short-statement =
  Parse-Spec.statement -- Parse-Spec.if-statement -- Parse.for-fixes
    >> (fn ((shows, assumes), fixes) =>
      (false, Binding.empty-atts, [], [Element.Fixes fixes, Element.Assumes assumes],
        Element.Shows shows));
in
fun theorem spec schematic =
  C-Inner-Syntax.local-command'
    spec
    (fn name-pref =>
      (long-statement || short-statement)
      -- let val apply = Parse.$$$ (name-pref apply) |-- Method.parse
        in Scan.repeat1 apply -- (Parse.$$$ (name-pref done) >> K NONE)
        || Scan.repeat apply -- (Parse.$$$ (name-pref by)
                    |-- Method.parse -- Scan.option Method.parse >> SOME)
        end)
    (C-Isar-Cmd.theorem schematic)
end

val opt-modes =
  Scan.optional (keyword ‹(› |-- Parse.!!! (Scan.repeat1 Parse.name --| keyword ‹)›)) [];

val - = Theory.setup
(   C-Inner-Syntax.command (C-Inner-Toplevel.generic-theory oo C-Inner-Isar-Cmd.setup)
                           C-Parse.ML-source
                           (≈setup, here, here)
  #> C-Inner-Syntax.command0 (C-Inner-Toplevel.theory o Isar-Cmd.setup)
                            C-Parse.ML-source
                            (setup, here, here, here)
```

```
#> C-Inner-Syntax.command0 (C-Inner-Toplevel.generic-theory o C-Isar-Cmd.ML)
                          C-Parse.ML-source
                          (ML, here, here, here)
#> C-Inner-Syntax.command0 (C-Inner-Toplevel.generic-theory o C-Module.C)
                          C-Parse.C-source
                          (C, here, here, here)
#> C-Inner-Syntax.command0' (C-Inner-Toplevel.generic-theory o C-Inner-File.ML NONE)
                           Keyword.thy-load
                           (C-Resources.parse-file --| semi)
                           (ML-file, here, here, here)
#> C-Inner-Syntax.command0' (C-Inner-Toplevel.generic-theory o C-Inner-File.C)
                           Keyword.thy-load
                           (C-Resources.parse-file --| semi)
                           (C-file, here, here, here)
#> C-Inner-Syntax.command0 (C-Inner-Toplevel.generic-theory o C-Module.C-export-boot)
                          C-Parse.C-source
                          (C-export-boot, here, here, here)
#> C-Inner-Syntax.command-range'
                     (Context.map-theory o Named-Target.theory-map o C-Module.C-export-file)
                          (C-export-file, here, here, here)
#> C-Inner-Syntax.command-no-range
    (C-Inner-Toplevel.generic-theory oo C-Inner-Isar-Cmd.setup
      ‹fn ((-, (-, pos1, pos2)) :: -) =>
          (fn - => fn - =>
            tap (fn - =>
                 Position.reports-text [((Position.range (pos1, pos2)
                                          |> Position.range-position, Markup.intensify), )]))
        | - => fn - => fn - => I›)
    (highlight, here, here)
#> theorem (theorem, here, here, here) false
#> theorem (lemma, here, here, here) false
#> theorem (corollary, here, here, here) false
#> theorem (proposition, here, here, here) false
#> theorem (schematic-goal, here, here, here) true
#> C-Inner-Syntax.local-command''
    (definition, here, here, here)
    (Scan.option Parse-Spec.constdecl -- (Parse-Spec.opt-thm-name : -- Parse.prop) --
      Parse-Spec.if-assumes -- Parse.for-fixes)
    C-Isar-Cmd.definition
#> C-Inner-Syntax.local-command''
    (declare, here, here, here)
    (Parse.and-list1 Parse.thms1 -- Parse.for-fixes)
    C-Isar-Cmd.declare
#> C-Inner-Syntax.command0
    (C-Inner-Toplevel.keep'' o C-Inner-Isar-Cmd.print-term)
    (C-Token.syntax' (opt-modes -- Parse.term))
    (term, here, here, here))
in end
›
```

3.9.4 Definitions of Outer Classical Commands

Library

ML — `~~/src/Pure/Isar/parse.ML` ‹
```
structure C-Outer-Parse =
struct
  val C-source = Parse.input (Parse.group (fn () => C source) Parse.embedded)
end
```
›

ML — `~~/src/Pure/Isar/outer_syntax.ML` ‹
```
structure C-Outer-Syntax =
struct
val - =
  Outer-Syntax.command command-keyword ‹C›
    (C-Outer-Parse.C-source >> (Toplevel.generic-theory o C-Module.C));
end
```
›

ML — `~~/src/Pure/Isar/isar_cmd.ML` ‹
```
structure C-Outer-Isar-Cmd =
struct
(* diagnostic ML evaluation *)

structure Diag-State = Proof-Data
(
  type T = Toplevel.state option;
  fun init - = NONE;
);

fun C-diag source state =
  let
    val opt-ctxt =
      try Toplevel.generic-theory-of state
      |> Option.map (Context.proof-of #> Diag-State.put (SOME state));
  in Context.setmp-generic-context (Option.map Context.Proof opt-ctxt)
    (fn () => C-Module.eval-source source) () end;

fun diag-state ctxt =
  (case Diag-State.get ctxt of
    SOME st => st
  | NONE => Toplevel.make-state NONE);

val diag-goal = Proof.goal o Toplevel.proof-of o diag-state;

val - = Theory.setup
  (ML-Antiquotation.value (Binding.qualify true Isar binding ‹C-state›)
    (Scan.succeed C-Outer-Isar-Cmd.diag-state ML-context) #>
   ML-Antiquotation.value (Binding.qualify true Isar binding ‹C-goal›)
```

```
    (Scan.succeed C-Outer-Isar-Cmd.diag-goal ML-context));

end
›

ML — ~~/src/Pure/ML/ml_file.ML ‹
structure C-Outer-File =
struct

fun command-c ({src-path, lines, digest, pos}: Token.file) =
  let
    val provide = Resources.provide (src-path, digest);
  in I
    #> C-Module.C (Input.source true (cat-lines lines) (pos, pos))
    #> Context.mapping provide (Local-Theory.background-theory provide)
  end;

fun C get-file gthy =
  command-c (get-file (Context.theory-of gthy)) gthy;

end;
›
```

Setup for `C` and `C_file` Command Syntax

```
ML — Pure ‹
local

val semi = Scan.option keyword‹;›;

val - =
  Outer-Syntax.command command-keyword‹C-file› read and evaluate Isabelle/C file
    (Resources.parse-file --| semi >> (Toplevel.generic-theory o C-Outer-File.C));

val - =
  Outer-Syntax.command command-keyword‹C-export-boot›
    C text within theory or local theory, and export to bootstrap environment
    (C-Outer-Parse.C-source >> (Toplevel.generic-theory o C-Module.C-export-boot));

val - =
  Outer-Syntax.command command-keyword‹C-prf› C text within proof
    (C-Outer-Parse.C-source >> (Toplevel.proof o C-Module.C-prf));

val - =
  Outer-Syntax.command command-keyword‹C-val› diagnostic C text
    (C-Outer-Parse.C-source >> (Toplevel.keep o C-Outer-Isar-Cmd.C-diag));

val - =
  Outer-Syntax.local-theory command-keyword‹C-export-file› diagnostic C text
```

(*Scan.succeed* () >> *K* (*C-Module.C-export-file Position.no-range*));
in end›

3.9.5 Term-Cartouches for C Syntax

syntax -*C-translation-unit* :: ‹*cartouche-position* ⇒ *string*› (**C**$_{unit}$ -)
syntax -*C-external-declaration* :: ‹*cartouche-position* ⇒ *string*› (**C**$_{decl}$ -)
syntax -*C-expression* :: ‹*cartouche-position* ⇒ *string*› (**C**$_{expr}$ -)
syntax -*C-statement* :: ‹*cartouche-position* ⇒ *string*› (**C**$_{stmt}$ -)
syntax -*C* :: ‹*cartouche-position* ⇒ *string*› (**C** -)

parse-translation ‹
C-Module.C-Term'.parse-translation
[(***syntax-const*** ‹-*C-translation-unit*›, *SOME C-Module.C-Term.tok-translation-unit*)
, (***syntax-const*** ‹-*C-external-declaration*›, *SOME C-Module.C-Term.tok-external-declaration*)
, (***syntax-const*** ‹-*C-expression*›, *SOME C-Module.C-Term.tok-expression*)
, (***syntax-const*** ‹-*C-statement*›, *SOME C-Module.C-Term.tok-statement*)
, (***syntax-const*** ‹-*C*›, *NONE*)]
›

ML‹*C-Module.env* (*Context.the-generic-context*())›

ML‹*open Args*›

3.9.6 C-env related ML-Antiquotations as Programming Support

ML‹

(*
was in Isabelle2020:
(*Args.context* −− *Scan.lift Args.embedded-position* >> (*fn* (*ctxt*, (*name*, *pos*)) =>

with:

val embedded-token = *ident* || *string* || *cartouche*;
val embedded-inner-syntax = *embedded-token* >> *Token.inner-syntax-of*;
val embedded-input = *embedded-token* >> *Token.input-of*;
val embedded = *embedded-token* >> *Token.content-of*;
val embedded-position = *embedded-input* >> *Input.source-content*;

defined in args.

Setting it to :

(*Args.context* −− *Scan.lift Args.name-position* >> (*fn* (*ctxt*, (*name*, *pos*)) =>

makes this syntactically more restrictive.
*)

```
val - = Theory.setup(
  ML-Antiquotation.value-embedded binding ‹C_env›
    (Args.context -- Scan.lift Args.name-position >> (fn (ctxt, (name, pos)) =>
      (warningarg variant not implemented;C-Module.env (Context.the-generic-context())))
    || Scan.succeed C-Module.env (Context.the-generic-context())))

›
```

Note that this anti-quotation is controlled by the `C_starting_env` - flag.

```
declare[[C_env0 = last]]
ML‹@{C_env}›

declare[[C_env0 = empty]]
ML‹@{C_env}›
```

3.9.7 The Standard Store C11-AST's generated from C-commands

Each call of the C command will register the parsed root AST in this theory-name indexed table.

```
ML‹
structure Root-Ast-Store = Generic-Data
  (type T = C-Grammar-Rule.ast-generic list Symtab.table
   val empty = Symtab.empty
   val merge = K empty);


Root-Ast-Store.map: (   C-Grammar-Rule.ast-generic list Symtab.table
                     -> C-Grammar-Rule.ast-generic list Symtab.table)
                   -> Context.generic -> Context.generic;


fun update-Root-Ast filter ast - ctxt =
    let val theory-id = Context.theory-long-name(Context.theory-of ctxt)
        val insert-K-ast  = Symtab.map-default (theory-id,[]) (cons ast)
    in  case filter ast of
         NONE => (warning No appropriate c11 ast found - store unchanged.; ctxt)
        |SOME - => (Root-Ast-Store.map insert-K-ast) ctxt
    end;


fun get-Root-Ast filter thy =
  let val ctxt = Context.Theory thy
      val thid = Context.theory-long-name(Context.theory-of ctxt)
      val ast = case Symtab.lookup (Root-Ast-Store.get ctxt) (thid) of
                SOME (a::-) => (case filter a of
                                  NONE => error Last C command is not of appropriate AST-class.
                                | SOME x => x)
              | - => errorNo C command in the current theory.
  in ast
```

end

val get-CExpr = get-Root-Ast C-Grammar-Rule.get-CExpr;
val get-CStat = get-Root-Ast C-Grammar-Rule.get-CStat;
val get-CExtDecl = get-Root-Ast C-Grammar-Rule.get-CExtDecl;
val get-CTranslUnit = get-Root-Ast C-Grammar-Rule.get-CTranslUnit;
›

setup ‹*Context.theory-map* (*C-Module.Data-Accept.put* (*update-Root-Ast SOME*))›

ML‹
(∗ *Was* : *Args.embedded-position changed to* : *Args.name-position.*
See comment above. ∗)
val - = Theory.setup(
ML-Antiquotation.value-embedded ***binding*** ‹*C11-CTranslUnit*›
(*Args.context* −− *Scan.lift Args.name-position* >> (*fn* (*ctxt*, (*name*, *pos*)) =>
(*warningarg variant not implemented*;*get-CTranslUnit* (*Context.the-global-context*())))
|| *Scan.succeed get-CTranslUnit* (*Context.the-global-context*()))
#>
ML-Antiquotation.value-embedded ***binding*** ‹*C11-CExtDecl*›
(*Args.context* −− *Scan.lift Args.name-position* >> (*fn* (*ctxt*, (*name*, *pos*)) =>
(*warningarg variant not implemented*;*get-CExtDecl* (*Context.the-global-context*())))
|| *Scan.succeed get-CExtDecl* (*Context.the-global-context*()))
#>
ML-Antiquotation.value-embedded ***binding*** ‹*C11-CStat*›
(*Args.context* −− *Scan.lift Args.name-position* >> (*fn* (*ctxt*, (*name*, *pos*)) =>
(*warningarg variant not implemented*;*get-CStat* (*Context.the-global-context*())))
|| *Scan.succeed get-CStat* (*Context.the-global-context*()))
#>
ML-Antiquotation.value-embedded ***binding*** ‹*C11-CExpr*›
(*Args.context* −− *Scan.lift Args.name-position* >> (*fn* (*ctxt*, (*name*, *pos*)) =>
(*warningarg variant not implemented*;*get-CExpr* (*Context.the-global-context*())))
|| *Scan.succeed get-CExpr* (*Context.the-global-context*()))
)
›

end

3.10 Support for Document Preparation: Text-Antioquotations.

theory *C-Document*
imports *C-Command*
begin

ML — `~~/src/Pure/Thy/document_output.ML`

```
‹
structure C-Document-Output =
struct

(* output document source *)

fun output-comment ctxt (kind, syms) =
  (case kind of
    Comment.Comment =>
      Input.cartouche-content syms
      |> output-document (ctxt |> Config.put Document-Antiquotation.thy-output-display false)
          {markdown = false}
      |> XML.enclose %\n\\isamarkupcmt{ %\n}
  | Comment.Cancel =>
      Symbol-Pos.cartouche-content syms
      |> Latex.symbols-output
      |> XML.enclose %\n\\isamarkupcancel{ }
  | Comment.Latex => Latex.symbols (Symbol-Pos.cartouche-content syms)
  | Comment.Marker => [])
and output-comment-document ctxt (comment, syms) =
  (case comment of
    SOME kind => output-comment ctxt (kind, syms)
  | NONE => Latex.symbols syms)
and output-document-text ctxt syms =
  Comment.read-body syms |> maps (output-comment-document ctxt)
and output-document ctxt {markdown} source =
  let
    val pos = Input.pos-of source;
    val syms = Input.source-explode source;

    val output-antiquotes =
      maps (Document-Antiquotation.evaluate (output-document-text ctxt) ctxt);

    fun output-line line =
      (if Markdown.line-is-item line then Latex.string \\item  else []) @
        output-antiquotes (Markdown.line-content line);

    fun output-block (Markdown.Par lines) =
          separate (XML.Text \n) (map (Latex.block o output-line) lines)
      | output-block (Markdown.List {kind, body, ...}) =
          Latex.environment (Markdown.print-kind kind) (output-blocks body)
    and output-blocks blocks =
      separate (XML.Text \n\n) (map (Latex.block o output-block) blocks);
  in
    if Toplevel.is-skipped-proof (Toplevel.presentation-state ctxt) then []
    else if markdown andalso exists (Markdown.is-control o Symbol-Pos.symbol) syms
    then
      let
        val ants = Antiquote.parse-comments pos syms;
```

```
      val reports = Antiquote.antiq-reports ants;
      val blocks = Markdown.read-antiquotes ants;
      val - = Context-Position.reports ctxt (reports @ Markdown.reports blocks);
    in output-blocks blocks end
  else
    let
      val ants = Antiquote.parse-comments pos (trim (Symbol.is-blank o Symbol-Pos.symbol)
syms);
      val reports = Antiquote.antiq-reports ants;
      val - = Context-Position.reports ctxt (reports @ Markdown.text-reports ants);
    in output-antiquotes ants end
end;


(* output tokens with formal comments *)

local

val output-symbols-antiq =
  (fn Antiquote.Text syms => Latex.symbols-output syms
    | Antiquote.Control {name = (name, -), body, ...} =>
        Latex.string (Latex.output-symbols [Symbol.encode (Symbol.Control name)]) @
          Latex.symbols-output body
    | Antiquote.Antiq {body, ...} =>
        XML.enclose %\n\\isaantiq\n {}%\n\\endisaantiq\n (Latex.symbols-output body));

fun output-comment-symbols ctxt {antiq} (comment, syms) =
  (case (comment, antiq) of
    (NONE, false) => Latex.symbols-output syms
  | (NONE, true) =>
      Antiquote.parse-comments (#1 (Symbol-Pos.range syms)) syms
      |> maps output-symbols-antiq
  | (SOME comment, -) => output-comment ctxt (comment, syms));

fun output-body ctxt antiq bg en syms =
  Comment.read-body syms
  |> maps (output-comment-symbols ctxt {antiq = antiq})
  |> XML.enclose bg en;

in

fun output-token ctxt tok =
  let
    fun output antiq bg en =
      output-body ctxt antiq bg en (Input.source-explode (C-Token.input-of tok));
  in
    (case C-Token.kind-of tok of
      Token.Comment NONE => []
    | Token.Comment (SOME Comment.Marker) => []
```

```
  | Token.Command => output false \\isacommand{ }
  | Token.Keyword =>
      if Symbol.is-ascii-identifier (C-Token.content-of tok)
      then output false \\isakeyword{ }
      else output false
  | Token.String => output false {\\isachardoublequoteopen} {\\isachardoublequoteclose}
  | Token.Alt-String => output false {\\isacharbackquoteopen} {\\isacharbackquoteclose}
  | Token.Cartouche => output false {\\isacartoucheopen} {\\isacartoucheclose}
  | Token.Control control => output-body ctxt false  (Antiquote.control-symbols control)
  | - => output false  )
 end handle ERROR msg => error (msg ^ Position.here (C-Token.pos-of tok));


end;
end;
›
```

ML — `~~/src/Pure/Thy/document_antiquotations.ML`

```
‹
structure C-Document-Antiquotations =
struct

(* quasi−formal text (unchecked) *)

local

fun report-text ctxt text =
  let val pos = Input.pos-of text in
    Context-Position.reports ctxt
      [(pos, Markup.language-text (Input.is-delimited text)),
       (pos, Markup.raw-text)]
  end;

fun prepare-text ctxt =
  Input.source-content #> #1 #> Document-Antiquotation.prepare-lines ctxt;

val theory-text-antiquotation =
        Document-Output.antiquotation-raw-embedded     binding ‹C-theory-text›     (Scan.lift
Parse.embedded-input)
    (fn ctxt => fn text =>
      let
        val keywords = C-Thy-Header.get-keywords' ctxt;

        val - = report-text ctxt text;
        val - =
          Input.source-explode text
          |> C-Token.tokenize keywords {strict = true}
          |> maps (C-Token.reports keywords)
```

```
      |> Context-Position.reports-text ctxt;
    in
      prepare-text ctxt text
      |> C-Token.explode0 keywords
      |> maps (C-Document-Output.output-token ctxt)
      |> Document-Output.isabelle ctxt
    end);

in

val - =
  Theory.setup theory-text-antiquotation;

end;

(* C text *)

local

fun c-text name c =
  Document-Output.antiquotation-verbatim-embedded name (Scan.lift Parse.embedded-input)
    (fn ctxt => fn text =>
      let val - = C-Module.eval-in text (SOME (Context.Proof ctxt)) (c text)
      in #1 (Input.source-content text) end);

in

val - = Theory.setup
 (c-text binding‹C› (C-Module.c-enclose  ) #>
  c-text binding‹C-text› (K C-Lex.read-init));

end;

end;
›

end
```

4 Appendix II: Examples for C11 Lexis and Syntax; and Isabelle/C User-Interaction

theory *C0*
imports *../main/C-Main*
begin

declare[[*C-lexer-trace*]]

4.1 Regular C Code

4.1.1 Comments, Keywords and Pragmas

C — Nesting of comments following the example suite of https://gcc.gnu.org/onlinedocs/cpp/Initial-processing.html ‹
/∗ inside /∗ inside ∗/ int a = outside;
// inside // inside until end of line
int a = outside;
/∗ inside
// inside
inside
∗/ int a = outside;
// inside /∗ inside until end of line
int a = outside;
›

C — Backslash newline ‹
i
n
t a = /∗ // /
∗
fff ∗/
;
›

C — Backslash newline, Directive https://gcc.gnu.org/onlinedocs/cpp/Initial-processing.html ‹
/
∗
∗/ # /∗
∗/ defi
ne FO
O 10
20›

C — Directive: conditional ‹
```
#ifdef a
#elif
#else
#if
#endif
#endif
```
›

C — Directive: macro ‹
```
#define a zz
#define a(x1,x2) z erz(( zz
#define a (x1,x2) z erz(( zz
#undef z
#if
#define a zz
#define a(x1,x2) z erz(( zz
#define a (x1,x2) z erz(( zz
#endif
```
›

4.1.2 Scala/jEdit Latency on Multiple Bindings

C — Example of obfuscated code https://en.wikipedia.org/wiki/International_Obfuscated_C_Code_Contest ‹
```
#define - -F<00||--F-OO--;
int F=00,OO=00;main(){F-OO();printf(%1.3f\n,4.*-F/OO/OO);}F-OO()
{
          -.-.-.-
      -.-.-.-.-.-.-.-.-.-
    -.-.-.-.-.-.-.-.-.-.-.-.-.-
  -.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-
 -.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-
 -.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-
-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-
-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-
-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-
-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-
 -.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-
 -.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-
  -.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-
    -.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-
       -.-.-.-.-.-.-.-.-.-
          -.-.-.-.
}
```
›

Select inside the ball, experience the latency. A special keyboard combination “Ctrl-

like key[1] + Shift + Enter" lets Isabelle/Scala/jEdit enter in a mode where the selected bound occurrences can be all simultaneously replaced by new input characters typed on the keyboard. (The "select-entity" action exists since Isabelle2016-1, see the respective section "Prover IDE – Isabelle/Scala/jEdit" in the NEWS.)

4.1.3 Lexing and Parsing Obfuscated Sources

Another lexer/parser - stress test: parsing an obfuscated C source.

C — Example of obfuscated code https://www.ioccc.org/2018/endoh1/prog.c ‹

```
#define/*--Int3rn^ti[]n/l-()I3fusc^t3|]-C-C<>I7E-C[]nt3st--*/L/*--MMXVIII--*/for
 #include/*!'()*+,-./12357:;<=>?CEFGHIJKLMNSTUVWXYZ[]^-`cfhijklmnrstuvwxyz{|}*/<stdio.h>
 char*r,F[1<<21]=~T/}3(|+G{>/zUhy;Jx+5wG<v>>u55t.?sIZrC]n.;m+:l+Hk]WjNJi/Sh+2f1>c2H`)(-2(^L\
 -]=([1/Z<2Y7/X12W:.VFFU1,T77S+;N?;M/>L..K1+JCCI<<H:(G*5F--E11C=5?.(>+(=3)Z-;*(:*.Y/5(-=)2*-
 /+-?5'(,+++***''EE>T,215IEUF:N`2`:?GK;+^`+?>)5?>U>-)5GxG).2K.2};}-235(]:5,S7E1(vTSS,-SSTvU(<-HG\
 -2E2/2L2/EE->E:?EE,2XMMMM1Hy`)5rHK;+.T+?[n2/-2{LKN2/-|cK2+.2`;}:?{KL57?|cK:2{NrHKtMMMK2nrH;r
 CkM-E21-E,-1->E(-:mSE/LhLE/mm:2Ul;2M>,2KW-+.-u).5Lm?fM`2`2nZXjj?[n<YcK?2}yC}H[^7N7LX^7N7UN<
 ZWXI<^I2K?>T+?KH~-?f<;G-x2;;2XT7LXIuuVF2X(G(GVV-:-:KjJ]HKLyN7UjJ3.WXjNI2KN<l|cKt2~[IsHfI2w{|
 GIfZG>x#&#&&$#$;ZXIc###$&$$#>7[LMv{&&&&#&##L,l2TY.&$#$#&&$,(iiii,#&&&#$#$?TY2.$#$1(x##
 SSEz.SW-k,T&&jC?E-.$##  &#&57+$$#  &&& W1-&$$7W  -J$#$kEN&#&
 $##C^+$##W,h###n/+L2YE
 2nJk/H;YNs#$[,:TU(#$  ,:  &&~H>&#  Y;  &&G-x&#2;  ,mT&$YE-#&  5G
 $#VVF$#&zNs$$&Ej]HELy\
 CN/U^Jk71<(#&:G7E+^&#  l|?1  $$Y.2$$  7lzs  WzZw>&$E  -<V-wE(2$$  G>x;
 2zsW/$$#HKt&$$v>+t1(>
 7>S7S,;TT,&$;S7S>7&#>E-::U  $$',op  ,*G=  F,*I=957+F  ;int*t,k,O,  i,
 j,T[+060<<+020];int M(
 int m,int nop){;;;return+  m%(0+nop  );;}  int*tOo,w,  h,z,W;void(C)  (int
 n){n=putchar(n);}int
 f,c,H=11,Y=64<<2,Z,pq,X  ;void(E/*d  */)(  int/*RP*/n  ){L(Z=k+00;  Z;
 Z/=+2+000)G[000]=*G*!!f
 |M(n,2)<<f,pq=2,f=+06  <f?++pq,++pq  ,G++  ,z:f+001,n  /=2;;}void  (V)(
 int/*opqrstabd*/n){C(n
 %Y);;C(n/Y+00);;}void  J(){L(pq--,pq  =j  =O=-1+0;++  j<240;I[6+  (h
 +6+j/12/2*2+M(j/2,2))*
 W+M(j/2/2,+06)*2+w*014  +00+M(00+  000+j,002  +00)]=000
 +00+k)k=M(G[j/2/2+(*r-+
 32)**<nopqabdeg],/*4649&96#*/3);/*&oaogoqo*/;}/*xD%P$Q#Rq*/int/*dbqpdbqpxyzzyboo3570OQ*/main()
 {L(X=Y-1;i<21*3;i++,I++)L(r=G,G+=2;*G++;)*G>=13*3?*G-*r?*I+++=*G:(*I++=r[1],*I++=r[2]):1;L(j=1
 =I;(*I=i=getchar())>-1;I++)i-7-3?I--=i<32||127<=i,j+=12:(H+=17+3,W=W<j?j:W,j=12);L(;*r>-1;r++)*r
 7-3?J(),w++:(w=z,h+=17+3);C(71);C(73);V('*'*'1'*7);C(57);C(32*3+1);V(W);V(H);C(122*2);L(V(i=z);i
 <32*3;)C(i++/3*X/31);C(33);C(X);C(11);L(G=SJYXHFUJ735;*G;)C(*G++-5);C(3);V(1);L(V(j=z);j<21*3;
 j++){k=257;V(63777);V(k<<2);V(M(j,32)?11:511);V(z);C(22*2);V(i=f=z);V(z);V(W);V(H);V(1<<11);r=
  G=I+W*H;L(t=T;i<1<<21;i++)T[i]=i<Y?i:-1;E(Y);L(i=-1;++i<W*H;t=T+Z*Y+Y)c=I[i]?I[i]*31-31:(31<
   j?j-31:31-j),Z=c[t[c]<z?E(Z),k<(1<<12)-2?t[c]=++k,T:T:t];E(Z);E(257);L(G++;k=G-r>X?X:G-r
    ,C(k),k;)L(;k--;C(*r++/*---#$%&04689@ABDOPQRabdegopq---*/));}C(53+6);return(z);}
```

›

[1] on Apple: Cmd

4.2 Experiments with `../../src_ext/parser_menhir`

declare[[*C-lexer-trace* = *false*]]

4.2.1 Expecting to succeed

C-file ‹../../*src-ext/parser-menhir/tests/argument-scope.c*›
C-file ‹../../*src-ext/parser-menhir/tests/atomic.c*›
C-file ‹../../*src-ext/parser-menhir/tests/atomic-parenthesis.c*›
C-file ‹../../*src-ext/parser-menhir/tests/bitfield-declaration-ambiguity.c*›
C-file ‹../../*src-ext/parser-menhir/tests/bitfield-declaration-ambiguity.ok.c*›
C-file ‹../../*src-ext/parser-menhir/tests/block-scope.c*›
C-file ‹../../*src-ext/parser-menhir/tests/c11_noreturn.c*›
C-file ‹../../*src-ext/parser-menhir/tests/c11_unions.c*›
C-file ‹../../*src-ext/parser-menhir/tests/char−literal−printing.c*›
C-file ‹../../*src-ext/parser-menhir/tests/c−namespace.c*›
C-file ‹../../*src-ext/parser-menhir/tests/control−scope.c*›
C-file ‹../../*src-ext/parser-menhir/tests/dangling-else.c*›
C-file ‹../../*src-ext/parser-menhir/tests/dangling-else-lookahead.c*›
C-file ‹../../*src-ext/parser-menhir/tests/dangling-else-lookahead.if.c*›
C-file ‹../../*src-ext/parser-menhir/tests/declaration-ambiguity.c*›
C-file ‹../../*src-ext/parser-menhir/tests/declarators.c*›
C-file ‹../../*src-ext/parser-menhir/tests/declarator-visibility.c*›
C-file ‹../../*src-ext/parser-menhir/tests/designator.c*›
C-file ‹../../*src-ext/parser-menhir/tests/enum.c*›
C-file ‹../../*src-ext/parser-menhir/tests/enum-constant-visibility.c*›
C-file ‹../../*src-ext/parser-menhir/tests/enum-shadows-typedef.c*›
C-file ‹../../*src-ext/parser-menhir/tests/enum−trick.c*›
C-file ‹../../*src-ext/parser-menhir/tests/expressions.c*›
C-file ‹../../*src-ext/parser-menhir/tests/function−decls.c*›
C-file ‹../../*src-ext/parser-menhir/tests/function-parameter-scope.c*›
C-file ‹../../*src-ext/parser-menhir/tests/function-parameter-scope-extends.c*›
C-file ‹../../*src-ext/parser-menhir/tests/if-scopes.c*›
C-file ‹../../*src-ext/parser-menhir/tests/local-scope.c*›
C-file ‹../../*src-ext/parser-menhir/tests/local-typedef.c*›
C-file ‹../../*src-ext/parser-menhir/tests/long−long−struct.c*›
C-file ‹../../*src-ext/parser-menhir/tests/loop-scopes.c*›
C-file ‹../../*src-ext/parser-menhir/tests/namespaces.c*›
C-file ‹../../*src-ext/parser-menhir/tests/no-local-scope.c*›
C-file ‹../../*src-ext/parser-menhir/tests/parameter-declaration-ambiguity.c*›
C-file ‹../../*src-ext/parser-menhir/tests/parameter-declaration-ambiguity.test.c*›
C-file ‹../../*src-ext/parser-menhir/tests/statements.c*›
C-file ‹../../*src-ext/parser-menhir/tests/struct−recursion.c*›
C-file ‹../../*src-ext/parser-menhir/tests/typedef-star.c*›
C-file ‹../../*src-ext/parser-menhir/tests/types.c*›
C-file ‹../../*src-ext/parser-menhir/tests/variable-star.c*›

4.2.2 Expecting to fail

C-file ‹../../*src-ext/parser-menhir/tests/bitfield-declaration-ambiguity.fail.c*›

C-file /../../src-ext/parser-menhir/tests/dangling-else-misleading.fail.c/

end

5 Appendix III: Examples for the SML Interfaces to Generic and Specific C11 ASTs

theory *C1*
imports *../main/C-Main*
begin

5.1 Access to Main C11 AST Categories via the Standard Interface

For the parsing root key's, c.f. `C_Command.thy`

declare [[C_{rule0} = *expression*]]
C‹$a + b * c - a / b$›
ML‹*val ast-expr* = @{*C11-CExpr*}›

declare [[C_{rule0} = *statement*]]
C‹$a = a + b;$›
ML‹*val ast-stmt* = @{*C11-CStat*}›

declare [[C_{rule0} = *external-declaration*]]
C‹*int m* ();›
ML‹*val ast-ext-decl* = @{*C11-CExtDecl*}›

declare [[C_{env0} = *last*]]
declare [[C_{rule0} = *translation-unit*]]
C‹*int b*; *int a* = *a* + *b*;›
ML‹*val ast-unit* = @{*C11-CTranslUnit*}
val env-unit = @{C_{env}}
›

... and completely low-level in ML:

declare [[C_{rule0} = *expression*]]
ML‹
val src = ‹*a* + *d*›;
val ctxt = (*Context.Theory* @{*theory*});
val ctxt' = *C-Module.C'* (*SOME* @{C_{env}}) *src ctxt*;
val tt = *Context.the-theory ctxt'*;
›

5.1.1 Queries on C11-Asts via the iterator

ML‹

```
fun selectIdent0 (a:C11-Ast-Lib.node-content) b c=  if #tag a = Ident0 then a::c else c;

(* and here comes the hic >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
*)

val S =  (C11-Ast-Lib.fold-cTranslationUnit selectIdent0 ast-unit []);

(* ... end of hic *)

fun print ({args = (C11-Ast-Lib.data-string S)::-::C11-Ast-Lib.data-string S'::[],
           sub-tag = STAG, tag = TAG}
          :C11-Ast-Lib.node-content)
         = let fun dark-matter (x:bstring) = XML.content-of (YXML.parse-body x)
           in writeln (:>^dark-matter(S)^<:>^(S')^<:>^STAG^<:>^TAG^<:) end;

app print S; (* these strings are representations for C-Ast.abr-string,
                where the main constructor is C-Ast.SS-base. *)
map (YXML.parse-body o (fn {args = (C11-Ast-Lib.data-string S)::-::C11-Ast-Lib.data-string
S'::[],
           sub-tag = -, tag = -} =>S)) S ;
```
›

5.1.2 A small compiler to Isabelle term's.

ML‹

```
fun drop-dark-matter x = (XML.content-of o YXML.parse-body) x


fun node-content-2-free (x : C11-Ast-Lib.node-content) =
    let  val C11-Ast-Lib.data-string a-markup = hd(#args(x));
         val id = hd(tl(String.tokens (fn x => x = #\)(drop-dark-matter a-markup)))
    in Free(id,dummyT) end  (* no type inference *);



fun selectIdent0Binary (a as { tag, sub-tag, args }:C11-Ast-Lib.node-content)
                       (b:  C-Ast.nodeInfo )
                       (c : term list)=
    case tag of
      Ident0 => (node-content-2-free a)::c
     |CBinary0 => (case (drop-dark-matter sub-tag, c) of
                    (CAddOp0,b::a::R) => (Const(Groups.plus-class.plus,dummyT) $ a $ b :: R)
                   | (CMulOp0,b::a::R) => (Const(Groups.times-class.times,dummyT) $ a $ b ::
R)
                 | (CDivOp0,b::a::R) => (Const(Rings.divide-class.divide,dummyT) $ a $ b :: R)
```

```
                | (CSubOp0,b::a::R) => (Const(Groups.minus-class.minus,dummyT) $ a $ b ::
R)
                | - => (writeln (sub-tag all  ^sub-tag^ :>> ^ @{make-string} c);c ))
    | - => c;


›
```

And here comes the ultra-hic: direct compilation of C11 expressions into (untyped) λ-terms in Isabelle. The term-list of the `C11_Ast_Lib.fold_cExpression` - iterator serves as term-stack in which sub-expressions were stored in reversed polish notation. The example shows that the resulting term is structurally equivalent.

```
ML‹
val S =  (C11-Ast-Lib.fold-cExpression selectIdent0Binary ast-expr []);
val S' = @{term a + b * c - a / b};
›
```

5.2 Late-binding a Simplistic Post-Processor for ASTs and ENVs

5.2.1 Definition of Core Data Structures

The following setup just stores the result of the parsed values in the environment.

```
ML‹
structure Data-Out = Generic-Data
(
  type T = (C-Grammar-Rule.ast-generic * C-Antiquote.antiq C-Env.stream) list
  val empty = []
  val merge = K empty
)

fun get-CTranslUnit thy =
  let val context = Context.Theory thy
  in (Data-Out.get context
       |> map (apfst (C-Grammar-Rule.get-CTranslUnit #> the)), C-Module.Data-In-Env.get
context)
  end

fun get-CExpr thy =
  let val context = Context.Theory thy
  in (Data-Out.get context
     |> map (apfst (C-Grammar-Rule.get-CExpr #> the)), C-Module.Data-In-Env.get context)
  end

›
```

... this gives :

```
ML‹ Data-Out.map: (   (C-Grammar-Rule.ast-generic * C-Antiquote.antiq C-Env.stream) list
```

```
    -> (C-Grammar-Rule.ast-generic * C-Antiquote.antiq C-Env.stream) list)
   -> Context.generic -> Context.generic ›
```

5.2.2 Registering A Store-Function in `C_Module.Data_Accept.put`

... as C-method call-back.

```
setup ‹Context.theory-map (C-Module.Data-Accept.put
                    (fn ast => fn env-lang =>
                      Data-Out.map (cons (ast, #stream-ignored env-lang |> rev))))›
```

5.2.3 Registering an ML-Antiquotation with an Access-Function

```
ML‹
val - = Theory.setup(
  ML-Antiquotation.value-embedded binding ‹C11-AST-CTranslUnit›
    (Args.context -- Scan.lift Args.name-position >> (fn (ctxt, (name, pos)) =>
      (warningarg variant not implemented;get-CTranslUnit (Context.the-global-context())))
    || Scan.succeed get-CTranslUnit (Context.the-global-context())))

›
```

5.2.4 Accessing the underlying C11-AST's via the ML Interface.

```
declare [[C_rule0 = translation-unit]]
C‹
void swap(int *x,int *y)
{
    int temp;

    temp = *x;
    *x = *y;
    *y = temp;
}
›

ML‹
local open C-Ast in
val - = CTranslUnit0
val (A::R, -) = @{C11-AST-CTranslUnit};
val (CTranslUnit0 (t,u), v) = A
fun rule-trans (CTranslUnit0 (t,u), v) = case C-Grammar-Rule-Lib.decode u of
                Left (p1,p2) => writeln (Position.here p1 ^  ^ Position.here p2)
              | Right S => warning (Not expecting that value:^S)
val bb = rule-trans A
end

val (R, env-final) = @{C11-AST-CTranslUnit};
val rules = map rule-trans R;
@{C_env}
```

›

5.3 Example: A Possible Semantics for *#include*

Implementation

The CPP directive `#include _` is used to import signatures of modules in C. This has the effect that imported identifiers are included in the C environment and, as a consequence, appear as constant symbols and not as free variables in the output.

The following structure is an extra mechanism to define the effect of `#include _` wrt. to its definition in its environment.

ML ‹
```
structure Directive-include = Generic-Data
(
  type T = (Input.source * C-Env.markup-ident) list Symtab.table
  val empty = Symtab.empty
  val merge = K empty
)
```
›

ML — *Pure* ‹
```
local
fun return f (env-cond, env) = ([], (env-cond, f env))

val - =
  Theory.setup
  (Context.theory-map
    (C-Context0.Directives.map
      (C-Context.directive-update (include, here)
        ( (return o K I)
        , fn C-Lex.Include (C-Lex.Group2 (toks-bl, -, tok :: -)) =>
              let
                fun exec file =
                  if exists (fn C-Scan.Left - => false | C-Scan.Right - => true) file then
                    K (error (Unsupported character
                              ^ Position.here
                                (Position.range-position
                                  (C-Lex.pos-of tok, C-Lex.end-pos-of (List.last toks-bl)))))
                  else
                    fn (env-lang, env-tree) =>
                      fold
                        (fn (src, data) => fn (env-lang, env-tree) =>
                          let val (name, pos) = Input.source-content src
                          in C-Grammar-Rule-Lib.shadowTypedef0''''
                               name
                               [pos]
                               data
                               env-lang
```

```
                    env-tree
                end)
              (these (Symtab.lookup (Directive-include.get (#context env-tree))
                                    (String.concat
                                      (maps (fn C-Scan.Left (s, -) => [s] | - => []) file))))
              (env-lang, env-tree)
        in
          case tok of
            C-Lex.Token (-, (C-Lex.String (-, file), -)) => exec file
          | C-Lex.Token (-, (C-Lex.File (-, file), -)) => exec file
          | - => tap (fn - => (* not yet implemented *)
                              warning (Ignored directive
                                       ^ Position.here
                                         (Position.range-position
                                           ( C-Lex.pos-of tok
                                           , C-Lex.end-pos-of (List.last toks-bl)))))
          end |> K |> K
      | - => K (K I)))))
in end
›

ML ‹
structure Include =
struct
fun init name vars =
  Context.theory-map
    (Directive-include.map
      (Symtab.update
        (name, map (rpair {global = true, params = [], ret = C-Env.Previous-in-stack}) vars)))

fun append name vars =
  Context.theory-map
    (Directive-include.map
      (Symtab.map-default
        (name, [])
         (rev o fold (cons o rpair {global = true, params = [], ret = C-Env.Previous-in-stack})
vars
            o rev)))

val show =
  Context.theory-map
    (Directive-include.map
      (tap
        (Symtab.dest
         #>
          app (fn (fic, vars) =>
            writeln (Content of \ ^ fic ^ \:
                  ^ String.concat (map (fn (i, -) => let val (name, pos) = Input.source-content i
                                                     in name ^ Position.here pos ^   end)
```

vars))))))
end
›

setup ‹*Include.append stdio.h* [‹*printf*›, ‹*scanf*›]›

Tests

C ‹
//@ setup ‹Include.append tmp [‹b›]›
#include tmp
int a = b;

›

C ‹
int b = 0;
//@ setup ‹Include.init tmp [‹b›]›
#include tmp
int a = b;
›

C ‹
int c = 0;
//@ setup ‹Include.append tmp [‹c›]›
//@ setup ‹Include.append tmp [‹c›]›
#include tmp
int a = b + c;
//@ setup ‹Include.show›
›

C‹

#include <stdio.h>
*#include /*sdfsdf */ <stdlib.h>*
#define a B
#define b(C)
#pragma / just exists syntaxically */*
›

In the following, we retrieve the C11 AST parsed above.

ML‹ *val ((C-Ast.CTranslUnit0 (t,u), v)::R, env) = @{C11-AST-CTranslUnit};*
val u = C-Grammar-Rule-Lib.decode u;
C-Ast.CTypeSpec0; ›

5.4 Defining a C-Annotation Commands Language

ML — *Isabelle-C.C-Command* ‹
— setup for a dummy ensures : the "Hello World" of Annotation Commands

```
local
datatype antiq-hol = Term of string (* term *)

val scan-opt-colon = Scan.option (C-Parse.$$$ :)

fun msg cmd-name call-pos cmd-pos =
  tap (fn - =>
       tracing (‹Hello World› reported by \ ^ cmd-name ^ \ here ^ call-pos cmd-pos))

fun command (cmd as (cmd-name, -)) scan0 scan f =
  C-Annotation.command'
    cmd

    (fn (-, (cmd-pos, -)) =>
       (scan0 -- (scan >> f) >> (fn - => C-Env.Never |> msg cmd-name Position.here
cmd-pos)))
in
val - = Theory.setup (    C-Inner-Syntax.command-no-range
                          (C-Inner-Toplevel.generic-theory oo C-Inner-Isar-Cmd.setup ‹K (K (K
I))›)
                        (loop, here, here)
                    #> command (ensures, here) scan-opt-colon C-Parse.term Term
                    #> command (invariant, here) scan-opt-colon C-Parse.term Term
                    #> command (assigns, here) scan-opt-colon C-Parse.term Term
                    #> command (requires, here) scan-opt-colon C-Parse.term Term
                    #> command (variant, here) scan-opt-colon C-Parse.term Term)
end
›

C‹
/*@ ensures result >= x && result >= y
 */

int max(int x, int y) {
  if (x > y) return x; else return y;
}
›
```

What happens on C11 AST level:

```
ML‹
val ((C-Ast.CTranslUnit0 (t,u), v)::R, env) = get-CTranslUnit @{theory};
val u = C-Grammar-Rule-Lib.decode u
›
```

5.4.1 C Code: Various Annotated Examples

This example suite is drawn from Frama-C and used in our GLA - TPs.

```
C‹
int sqrt(int a) {
```

```
  int i = 0;
  int tm = 1;
  int sum = 1;

  /*@ loop invariant 1 <= sum <= a+tm
      loop invariant (i+1)*(i+1) == sum
      loop invariant tm+(i*i) == sum
      loop invariant 1<=tm<=sum
      loop assigns i, tm, sum
      loop variant a-sum
   */
  while (sum <= a) {
    i++;
    tm = tm + 2;
    sum = sum + tm;
  }

  return i;
}
›

C‹
/*@ requires n >= 0
    requires valid(t+(0..n-1))
    ensures exists integer i; (0<=i<n && t[i] != 0) <==> result == 0
    ensures (forall integer i; 0<=i<n ==> t[i] == 0) <==> result == 1
    assigns nothing
 */

int allzeros(int t[], int n) {
  int k = 0;

  /*@ loop invariant 0 <= k <= n
      loop invariant forall integer i; 0<=i<k ==> t[i] == 0
      loop assigns k
      loop variant n-k
   */
  while(k < n) {
    if (t[k]) return 0;
    k = k + 1;
  }
  return 1;
}

›

C‹

/*@ requires n >= 0
```

```
    requires valid(t+(0..n-1))
    ensures (forall integer i; 0<=i<n ==> t[i] != v) <==> result == -1
    ensures (exists integer i; 0<=i<n && t[i] == v) <==> result == v
    assigns nothing
 */

int binarysearch(int t[], int n, int v) {
  int l = 0;
  int u = n-1;

  /*@ loop invariant false
   */
  while (l <= u) {
    int m = (l + u) / 2;
    if (t[m] < v) {
      l = m + 1;
    } else if (t[m] > v) {
      u = m - 1;
    }
    else return m;
  }
  return -1;
}
```

C

```
/*@ requires n >= 0
    requires valid(t+(0..n-1))
    requires (forall integer i,j; 0<=i<=j<n ==> t[i] <= t[j])
    ensures exists integer i; (0<=i<n && t[i] == x) <==> result == 1
    ensures (forall integer i; 0<=i<n ==> t[i] != x) <==> result == 0
    assigns nothing
 */

int linearsearch(int x, int t[], int n) {
  int i = 0;

  /*@ loop invariant 0<=i<=n
      loop invariant forall integer j; 0<=j<i ==> (t[j] != x)
      loop assigns i
      loop variant n-i
   */
  while (i < n) {
    if (t[i] < x) {
      i++;
    } else {
      return (t[i] == x);
    }
```

```
  }

  return 0;
}
```

5.4.2 Example: An Annotated Sorting Algorithm

C

```
#include <stdio.h>

int main()
{
  int array[100], n, c, d, position, swap;

  printf(Enter number of elements\n);
  scanf(%d, &n);

  printf(Enter %d integers\n, n);

  for (c = 0; c < n; c++) scanf(%d, &array[c]);

  for (c = 0; c < (n - 1); c++)
  {
    position = c;

    for (d = c + 1; d < n; d++)
    {
      if (array[position] > array[d])
        position = d;
    }
    if (position != c)
    {
      swap = array[c];
      array[c] = array[position];
      array[position] = swap;
    }
  }

printf(Sorted list in ascending order:\n);

  for (c = 0; c < n; c++)
    printf(%d\n, array[c]);

  return 0;
}
```

A better example implementation:

C

```
#include <stdio.h>
#include <stdlib.h>

#define SIZE 10

void swap(int *x,int *y);
void selection-sort(int* a, const int n);
void display(int a[],int size);

void main()
{

    int a[SIZE] = {8,5,2,3,1,6,9,4,0,7};

    int i;
    printf(The array before sorting:\n);
    display(a,SIZE);

    selection-sort(a,SIZE);

    printf(The array after sorting:\n);
    display(a,SIZE);
}

/*
    swap two integers
*/
void swap(int *x,int *y)
{
    int temp;

    temp = *x;
    *x = *y;
    *y = temp;
}
/*
    perform selection sort
*/
void selection-sort(int* a,const int size)
{
    int i, j, min;

    for (i = 0; i < size - 1; i++)
    {
        min = i;
        for (j = i + 1; j < size; j++)
        {
            if (a[j] < a[min])
            {
```

```
                min = j;
            }
        }
        swap(&a[i], &a[min]);
    }
}
/*
    display array content
*/
void display(int a[],const int size)
{
    int i;
    for(i=0; i<size; i++)
        printf(%d ,a[i]);
    printf(\n);
}
›
```

5.5 C Code: Floats Exist

C‹

```
int a;
float b;
int m() {return 0;}
```

›

end

6 Appendix IV : Examples for Annotation Navigation and Context Serialization

theory *C2*
imports *../main/C-Main*
HOL−ex.Cartouche-Examples
begin

Operationally, the **C** command can be thought of as behaving as the **ML** command, where it is for example possible to recursively nest C code in C. Generally, the present chapter assumes a familiarity with all advance concepts of ML as described in `~~/src/HOL/Examples/ML.thy`, as well as the concept of ML antiquotations (`~~/src/Doc/Implementation/ML.thy`). However, even if **C** might resemble to **ML**, we will now see in detail that there are actually subtle differences between the two commands.

6.1 Setup of ML Antiquotations Displaying the Environment (For Debugging)

ML‹
```
fun print-top make-string f - (-, (value, -, -)) - = tap (fn - => writeln (make-string value)) o f

fun print-top' - f - - env = tap (fn - => writeln (ENV ^ C-Env.string-of env)) o f

fun print-stack s make-string stack - - thy =
  let
    val () = Output.information (SHIFT ^ (case s of NONE => | SOME s => \ ^ s ^ \ )
                          ^ Int.toString (length stack − 1) ^ +1 )
    val () =  stack
          |> split-list
          |> #2
          |> map-index I
          |> app (fn (i, (value, pos1, pos2)) =>
                writeln (  ^ Int.toString (length stack − i) ^  ^ make-string value
                      ^  ^ Position.here pos1 ^  ^ Position.here pos2))
  in thy end

fun print-stack' s - stack - env thy =
  let
    val () = Output.information (SHIFT ^ (case s of NONE => | SOME s => \ ^ s ^ \ )
                          ^ Int.toString (length stack − 1) ^ +1 )
    val () = writeln (ENV ^ C-Env.string-of env)
  in thy end
```

›

setup ‹*ML-Antiquotation.inline* @{*binding print-top*}
(*Args.context*
>> *K* (*print-top* ^ *ML-Pretty.make-string-fn* ^ *I*))›
setup ‹*ML-Antiquotation.inline* @{*binding print-top'*}
(*Args.context*
>> *K* (*print-top'* ^ *ML-Pretty.make-string-fn* ^ *I*))›
setup ‹*ML-Antiquotation.inline* @{*binding print-stack*}
(*Scan.peek* (*fn* - => *Scan.option Parse.embedded*)
>> (*fn name* => (*print-stack*
^ (*case name of NONE* => *NONE*
| *SOME s* => (*SOME* \ ^ *s* ^ \))
^ ^ *ML-Pretty.make-string-fn*)))›
setup ‹*ML-Antiquotation.inline* @{*binding print-stack'*}
(*Scan.peek* (*fn* - => *Scan.option Parse.embedded*)
>> (*fn name* => (*print-stack'*
^ (*case name of NONE* => *NONE*
| *SOME s* => (*SOME* \ ^ *s* ^ \))
^ ^ *ML-Pretty.make-string-fn*)))›

declare[[*C-lexer-trace*]]

6.2 Introduction to C Annotations: Navigating in the Parsing Stack

6.2.1 Basics

Since the present theory `C1.thy` is depending on *Isabelle-C.C-Lexer-Language* and *Isabelle-C.C-Parser-Language*, the syntax one is writing in the **C** command is C11. Additionally, `C1.thy` also depends on *Isabelle-C.C-Parser-Annotation*, making it possible to write commands in C comments, called annotation commands, such as ≈**setup**.

C — Nesting ML code in C comments ‹
int a = (((*0*))); /*@ *highlight* */
/*@ ≈*setup* ‹@{*print-stack*}› */
/*@ ≈*setup* ‹@{*print-top*}› */
›

In terms of execution order, nested annotation commands are not pre-filtered out of the C code, but executed when the C code is still being parsed. Since the parser implemented is a LALR parser [1], C tokens are uniquely read and treated from left to right. Thus, each nested command is (supposed by default to be) executed when the parser has already read all C tokens before the comment associated to the nested command, so when the parser is in a particular intermediate parsing step (not necessarily final) [2].

[1] https://en.wikipedia.org/wiki/LALR
[2] https://en.wikipedia.org/wiki/Shift-reduce_parser

The command ≈**setup** is similar to the command **setup** except that the former takes a function with additional arguments. These arguments are precisely depending on the current parsing state. To better examine these arguments, it is convenient to use ML antiquotations (be it for printing, or for doing any regular ML actions like PIDE reporting).

Note that, in contrast with **setup**, the return type of the ≈**setup** function is not `theory -> theory` but `Context.generic -> Context.generic`.

C — Positional navigation: referring to any previous parsed sub-tree in the stack ‹
```
int a = (((0
    + 5)))  /*@@ ≈setup ‹print-top @{make-string} I›
              @ highlight
             */
    * 4;
float b = 7 / 3;
›
```

The special @ symbol makes the command be executed whenever the first element E in the stack is about to be irremediably replaced by a more structured parent element (having E as one of its direct children). It is the parent element which is provided to the ML code.

Instead of always referring to the first element of the stack, N consecutive occurrences of @ will make the ML code getting as argument the direct parent of the N-th element.

C — Positional navigation: referring to any previous parsed sub-tree in the stack ‹
```
int a = (((0 + 5)))  /*@@ highlight */
    * 4;

int a = (((0 + 5)))  /*@& highlight */
    * 4;

int a = (((0 + 5)))  /*@@@@@ highlight */
    * 4;

int a = (((0 + 5)))  /*@&&&&& highlight */
    * 4;
›
```

& behaves as @, but instead of always giving the designated direct parent to the ML code, it finds the first parent ancestor making non-trivial changes in the respective grammar rule (a non-trivial change can be for example the registration of the position of the current AST node being built).

C — Positional navigation: moving the comment after a number of C token ‹
```
int b = 7 / (3) * 50;
/*@+++@@ highlight */
long long f (int a) {
  while (0) { return 0; }
}
int b = 7 / (3) * 50;
```

```
›
```

N consecutive occurrences of + will delay the interpretation of the comment, which is ignored at the place it is written. The comment is only really considered after the C parser has treated N more tokens.

```
C — Closing C comments */ must close anything, even when editing ML code ‹
int a = (((0 //@ (* inline *) ≈setup ‹fn - => fn - => fn - => fn context => let in (* */ *)
context end›
            /*@ ≈setup ‹(K o K o K) I› (*   * /   *) */
        )));
›

C — Inline comments with antiquotations ‹
 /*@ ≈setup‹(K o K o K) (fn x => K x @{con\
text (**)})› */ // break of line activated everywhere (also in antiquotations)
int a = 0; //\
@ ≈setup‹(K o K o K) (fn x => K x @{term ‹a \
        + b› (* (**) *\
\
)})›
›
```

6.2.2 Erroneous Annotations Treated as Regular C Comments

```
C — Permissive Types of Antiquotations ‹
int a = 0;
  /*@ ≈setup (* Errors: Explicit warning + Explicit markup reporting *)
   */
  /** ≈setup (* Errors: Turned into tracing report information *)
   */

  /** ≈setup ‹fn - => fn - => fn - => I› (* An example of correct syntax accepted as usual *)
   */
›

C — Permissive Types of Antiquotations ‹
int a = 0;
  /*@ ≈setup ‹fn - => fn - => fn - => I›
      ≈setup (* Parsing error of a single command does not propagate to other commands *)
      ≈setup ‹fn - => fn - => fn - => I›
      context
   */
  /** ≈setup ‹fn - => fn - => fn - => I›
      ≈setup (* Parsing error of a single command does not propagate to other commands *)
      ≈setup ‹fn - => fn - => fn - => I›
      context
   */

  /*@ ≈setup (* Errors in all commands are all rendered *)
```

```
    ≈setup (* Errors in all commands are all rendered *)
    ≈setup (* Errors in all commands are all rendered *)
  */
  /** ≈setup (* Errors in all commands makes the whole comment considered as an usual
comment *)
    ≈setup (* Errors in all commands makes the whole comment considered as an usual comment
*)
    ≈setup (* Errors in all commands makes the whole comment considered as an usual comment
*)
  */
›
```

6.2.3 Bottom-Up vs. Top-Down Evaluation

```
ML‹
structure Example-Data = Generic-Data
(
  type T = string list
  val empty = []
  val merge = K empty
)

fun add-ex s1 s2 =
  Example-Data.map (cons s2)
  #> (fn context => let val () = Output.information (s1 ^ s2)
                       val () = app (fn s => writeln (  Data content:  ^ s))
                                    (Example-Data.get context)
                   in context end)
›

setup ‹Context.theory-map (Example-Data.put [])›

declare[[ML-source-trace]]
declare[[C-parser-trace]]

C — Arbitrary interleaving of effects: ≈setup vs ≈setup⇓ ‹
int b,c,d/*@@ ≈setup ‹fn s => fn x => fn env => @{print-top} s x env
                                    #> add-ex evaluation of  3-print-top›
        */,e = 0; /*@@
            ≈setup ‹fn s => fn x => fn env => @{print-top} s x env
                                    #> add-ex evaluation of  4-print-top› */

int b,c,d/*@@ ≈setup⇓ ‹fn s => fn x => fn env => @{print-top} s x env
                                    #> add-ex evaluation of  6-print-top›
        */,e = 0; /*@@
            ≈setup⇓ ‹fn s => fn x => fn env => @{print-top} s x env
                                    #> add-ex evaluation of  5-print-top› */
›
```

6.2.4 Out of Bound Evaluation for Annotations

C — Bottom-up and top-down + internal initial value ‹
int a = 0 ;
int /@ @ ML ‹writeln 2›*
@@@ ML ‹writeln 4›
+@ ML ‹writeln 3›
(+++@ ML ‹writeln 6›*)*
*ML⇓‹writeln 1› */*
// a d /@ @ ML ‹writeln 5› */;*
int a;
›

C — Ordering of consecutive commands ‹
int a = 0 /@ ML‹writeln 1› */;*
int /@ @@@@@ML‹writeln 5 › @@@ML‹writeln 4 › @@ML‹writeln 2 › */*
/@ @@@@@ML‹writeln 5′› @@@ML‹writeln 4′› @@ML‹writeln 2′› */*
a = 0;
int d = 0; /@ ML‹writeln 3› */*
›

C — Maximum depth reached ‹
int a = 0 /@ ++@@@@ML‹writeln 2›*
*++@@@ ML‹writeln 1› */;*
›

6.3 Reporting of Positions and Contextual Update of Environment

To show the content of the parsing environment, the ML antiquotations *print-top′* and *print-stack′* will respectively be used instead of *print-top* and *print-stack*. This example suite allows to explore the bindings represented in the C environment and made accessible in PIDE for hovering.

6.3.1 Reporting: *typedef*, *enum*

declare [[*ML-source-trace = false*]]
declare [[*C-lexer-trace = false*]]

C — Reporting of Positions ‹
typedef int i, j;
/@@ ≈setup ‹@{print-top′}› @highlight */ //@ +++++@ ≈setup ‹@{print-top′}› +++++@highlight*
int j = 0;
typedef int i, j;
j jj1 = 0;
j jj = jj1;
j j = jj1 + jj;

```
typedef i j;
typedef i j;
typedef i j;
i jj = jj;
j j = jj;
›
```

C — Nesting type definitions ‹
```
typedef int j;
j a = 0;
typedef int k;
int main (int c) {
  j b = 0;
  typedef int k;
  typedef k l;
  k a = c;
  l a = 0;
}
k a = a;
›
```

C — Reporting *enum* ‹
```
enum a b; // bound case: undeclared
enum a {aaa}; // define case
enum a {aaa}; // define case: redefined
enum a -; // bound case

--thread (f ( enum a,  enum a vv));

enum a /* <-- C_Grammar_Rule_Wrap_Overloading.function_definition4*/ f (enum a a)
{
}

--thread enum a /* <-- C_Grammar_Rule_Wrap_Overloading.declaration_specifier2*/ f
(enum a a) {
  enum c {ccc}; // define case
  --thread enum c f (enum c a) {
    return 0;
  }
   enum c /* <-- C_Grammar_Rule_Wrap_Overloading.nested_function_definition2*/ f
(enum c a) {
    return 0;
  }
  return 0;
}

enum z {zz}; // define case
int main (enum z *x) /* <-- C_Grammar_Rule_Wrap_Overloading.parameter_type_list2*/
{
```

```
  return zz; }
int main (enum a *x, ...) /* ←— C_Grammar_Rule_Wrap_Overloading.parameter_type_list3*/
{
  return zz; }
›
```

6.3.2 Continuation Calculus with the C Environment: Presentation in ML

declare [[*C-parser-trace* = *false*]]

```
ML‹
val C = C-Module.C' NONE
val C' = C-Module.C' o SOME
›
```

C — Nesting C code without propagating the C environment ‹

```
int a = 0;
int b = 7 / (3) * 50
  /*@@@@@@ ≈setup ‹fn - => fn - => fn - =>
              C      ‹int b = a + a + a + a + a + a + a
                     ;› › */;
›
```

C — Nesting C code and propagating the C environment ‹

```
int a = 0;
int b = 7 / (3) * 50
  /*@@@@@@ ≈setup ‹fn - => fn - => fn env =>
              C' env ‹int b = a + a + a + a + a + a + a
                     ;› › */;
›
```

6.3.3 Continuation Calculus with the C Environment: Presentation with Outer Commands

```
ML‹
val - = Theory.setup
        (C-Inner-Syntax.command0
          (fn src => fn context => C' (C-Stack.Data-Lang.get' context |> #2) src context)
          C-Parse.C-source
          (C', here, here, here))
›
```

C — Nesting C code without propagating the C environment ‹

```
int f (int a) {
  int b = 7 / (3) * 50 /*@ C ‹int b = a + a + a + a + a + a + a;› */;
  int c = b + a + a + a + a + a + a;
} ›
```

C — Nesting C code and propagating the C environment ‹

```
int f (int a) {
  int b = 7 / (3) * 50 /*@ C' ‹int b = a + a + a + a + a + a + a;› */;
  int c = b + b + b + b + a + a + a + a + a + a;
} ›
```

C — Miscellaneous ‹

```
int f (int a) {
  int b = 7 / (3) * 50 /*@ C ‹int b = a + a + a + a + a; //@ C' ‹int c = b + b + b + b +
a;› › */;
  int b = 7 / (3) * 50 /*@ C' ‹int b = a + a + a + a + a; //@ C' ‹int c = b + b + b + b +
a;› › */;
  int c = b + b + b + b + a + a + a + a + a + a;
} ›
```

6.3.4 Continuation Calculus with the C Environment: Deep-First Nesting vs Breadth-First Folding: Propagation of `C_Env.env_lang`

C — Propagation of report environment while manually composing at ML level (with #>)
— In *c1* #> *c2*, *c1* and *c2* should not interfere each other. ‹

```
//@ ML ‹fun C-env src - - env = C' env src›
int a;
int f (int b) {
int c = 0; /*@ ≈setup ‹fn - => fn - => fn env =>
    C' env ‹int d = a + b + c + d; //@ ≈setup ‹C-env ‹int e = a + b + c + d;›››
  #> C     ‹int d = a + b + c + d; //@ ≈setup ‹C-env ‹int e = a + b + c + d;›››
  #> C' env ‹int d = a + b + c + d; //@ ≈setup ‹C-env ‹int e = a + b + c + d;›››
  #> C     ‹int d = a + b + c + d; //@ ≈setup ‹C-env ‹int e = a + b + c + d;›››
› */
int e = a + b + c + d;
}›
```

C — Propagation of directive environment (evaluated before parsing) to any other annotations (evaluated at parsing time) ‹

```
#undef int
#define int(a,b) int
#define int int
int a;
int f (int b) {
int c = 0; /*@ ≈setup ‹fn - => fn - => fn env =>
    C' env ‹int d = a + b + c + d; //@ ≈setup ‹C-env ‹int e = a + b + c + d;›››
  #> C     ‹int d = a + b + c + d; //@ ≈setup ‹C-env ‹int e = a + b + c + d;›››
  #> C' env ‹int d = a + b + c + d; //@ ≈setup ‹C-env ‹int e = a + b + c + d;›››
  #> C     ‹int d = a + b + c + d; //@ ≈setup ‹C-env ‹int e = a + b + c + d;›››
› */
#undef int
int e = a + b + c + d;
}
›
```

6.3.5 Continuation Calculus with the C Environment: Deep-First Nesting vs Breadth-First Folding: Propagation of `C_Env.env_tree`

ML‹
structure Data-Out = Generic-Data
(
  type T = int
  val empty = 0
  val merge = K empty
)

fun show-env0 make-string f msg context =
  Output.information ((^ *msg* ^ *)* ^ *make-string (f (Data-Out.get context)))*

val show-env = tap o show-env0 @{make-string} I
›

setup ‹*Context.theory-map (C-Module.Data-Accept.put (fn - => fn - => Data-Out.map (fn x => x + 1)))*›

C — Propagation of Updates ‹
typedef int i, j;
int j = 0;
typedef int i, j;
j jj1 = 0;
j jj = jj1; /@@ ≈setup ‹fn - => fn - => fn - => show-env POSITION 0› @≈setup ‹@{print-top'}› */*
typedef int k; /@@ ≈setup ‹fn - => fn - => fn env =>*
C' env ‹k jj = jj; //@@ ≈setup ‹@{print-top'}›
k jj = jj + jj1;
typedef k l; //@@ ≈setup ‹@{print-top'}››
*#> show-env POSITION 1› */*
j j = jj1 + jj; //@@ ≈setup ‹@{print-top'}›
typedef i j; /@@ ≈setup ‹fn - => fn - => fn - => show-env POSITION 2› */*
typedef i j;
typedef i j;
i jj = jj;
j j = jj;
›

ML‹*show-env POSITION 3 (Context.Theory @{theory})*›

setup ‹*Context.theory-map (C-Module.Data-Accept.put (fn - => fn - => I))*›

6.3.6 Reporting: Scope of Recursive Functions

declare [[C_{env0} = *last*]]

C — Propagation of Updates ‹
int a = 0;

*int b = a * a + 0;*
int jjj = b;
*int main (void main(int *x,int *y),int *jjj) {*
* return a + jjj + main(); }*
int main2 () {
* int main3 () { main2() + main(); }*
* int main () { main2() + main(); }*
* return a + jjj + main3() + main(); }*
›

C ‹
int main3 () { main2 (); }
›

declare $[[C_{env0} = empty]]$

6.3.7 Reporting: Extensions to Function Types, Array Types

C ‹*int f (int z);*›
C ‹*int * f (int z);*›
C ‹*int (* f) (int z /* ←— C_Grammar_Rule_Wrap_Overloading.declarator1*/);*›
C ‹*typedef int (* f) (int z);*›
C ‹*int f (int z) {}*›
C ‹*int * f (int z) {return z;}*›
C ‹*int ((* f) (int z1, int z2)) {return z1 + z2;}*›
C ‹*int (* (* f) (int z1, int z2)) {return z1 + z2;}*›
C ‹*typedef int (* f) (int z); f uuu (int b) {return b;};*›
C ‹*typedef int (* (* f) (int z, int z)) (int a); f uuu (int b) {return b;};*›
C ‹*struct z { int (* f) (int z); int (* (* ff) (int z)) (int a); };*›
C ‹*double (* (* f (int a /* ←— C_Grammar_Rule_Wrap_Overloading.declarator1*/)) (int a, double d)) (char a);*›
C ‹*double (* (((* f) []) (int a)) (int b, double c)) (char d) {int a = b + c + d;}*›
C ‹*double ((*((f) (int a))) (int a /* ←— C_Grammar_Rule_Lib.doFuncParamDeclIdent*/, double)) (char c) {int a = 0;}*›

C — Nesting functions ‹
double ((* f (int a)) (int a, double)) (char c) {*
double ((* f (int a)) (double a, int a)) (char) {*
* return a;*
}
}
›

C — Old function syntax ‹
f (x) int x; {return x;}
›

6.4 General Isar Commands

locale *zz* **begin definition** $z' = ()$
end

C — Mixing arbitrary commands ‹
int a = 0;
*int b = a * a + 0*;
int jjj = b;
/*@
@@@ *ML* ‹@{*lemma* ‹$A \wedge B \longrightarrow B \wedge A$› *by* (*ml-tactic* ‹*blast-tac ctxt 1*›)}›
definition $a' = ()$
declare [[*ML-source-trace*]]
lemma (*in zz*) ‹$A \wedge B \longrightarrow B \wedge A$› *by* (*ml-tactic* ‹*blast-tac ctxt 1*›)
definition (*in zz*) *z* = ()
corollary $zz.z' = ()$
apply (*unfold zz.z'-def*)
by blast
theorem True &&& True by (*auto*, *presburger?*)
*/
›

declare [[*ML-source-trace* = *false*]]

C — Backslash newlines must be supported by `C_Token.syntax'` (in particular in keywords) ‹
//@ *lem*\
ma (*i*\
n z\
z) \
‹\
AA ∧ *B*\
⟶\
B ∧ *A*\
\
A› *b*\
y (*ml-t*\
actic ‹\
bla\
st-tac c\
txt\
0\
001›)
›

6.5 Starting Parsing Rule

6.5.1 Basics

C — Parameterizing starting rule ‹

/*@
declare [[C_{rule0} = *statement*]]
C ‹*while* (*a*) {}›
C ‹*a* = *2*;›
declare [[C_{rule0} = *expression*]]
C ‹*2* + *3*›
C ‹*a* = *2*›
C ‹*a*[*1*]›
C ‹&*a*›
C ‹*a*›
*/
›

6.5.2 Embedding in Inner Terms

term ‹**C** — default behavior of parsing depending on the activated option ‹*0*››
term ‹$\mathbf{C}_{unit}$ — force the explicit parsing ‹*f* () {*while* (*a*) {}; *return 0*;} *int a* = *0*;››
term ‹$\mathbf{C}_{decl}$ — force the explicit parsing ‹*int a* = *0*; ››
term ‹$\mathbf{C}_{expr}$ — force the explicit parsing ‹*a*››
term ‹$\mathbf{C}_{stmt}$ — force the explicit parsing ‹*while* (*a*) {}››

declare [[C_{rule0} = *translation-unit*]]

term ‹**C** — default behavior of parsing depending on the current option ‹*int a* = *0*;››

6.5.3 User Defined Setup of Syntax

setup ‹*C-Module.C-Term.map-expression* (*fn* - => *fn* - => *fn* - => @{*term 10* :: *nat*})›
setup ‹*C-Module.C-Term.map-statement* (*fn* - => *fn* - => *fn* - => @{*term 20* :: *nat*})›
value ‹$\mathbf{C}_{expr}$‹*1*› + $\mathbf{C}_{stmt}$‹*for* (;;);››

setup — redefinition ‹*C-Module.C-Term.map-expression*
(*fn* - => *fn* - => *fn* - => @{*term 1000* :: *nat*})›
value ‹$\mathbf{C}_{expr}$‹*1*› + $\mathbf{C}_{stmt}$‹*for* (;;);››

setup ‹*C-Module.C-Term.map-default* (*fn* - => *fn* - => *fn* - => @{*term True*})›

6.5.4 Validity of Context for Annotations

ML ‹*fun fac x* = *if x* = *0 then 1 else x* ∗ *fac* (*x* − *1*)›

ML — Execution of annotations in term possible in (the outermost) **ML** ‹
***term*‹ **C** ‹*int c* = *0*; /*@ *ML* ‹*fac 100*› */› ›
›

definition — Execution of annotations in term possible in `local_theory` commands (such as **definition**) ‹
term = **C** ‹*int c* = *0*; /*@ *ML* ‹*fac 100*› */›
›

6.6 Scopes of Inner and Outer Terms

```
ML ‹
local
fun bind scan ((stack1, (to-delay, stack2)), -) =
  C-Parse.range scan
  >> (fn (src, range) =>
      C-Env.Parsing
       ( (stack1, stack2)
       , ( range
         , C-Inner-Syntax.bottom-up
            (fn - => fn context =>
              ML-Context.exec
                (tap (fn - => Syntax.read-term (Context.proof-of context)
                                               (Token.inner-syntax-of src)))
                context)
         , Symtab.empty
         , to-delay)))
in
val - =
  Theory.setup
   (   C-Annotation.command'
        (term_inner, here)

        (bind (C-Token.syntax' (Parse.token Parse.cartouche)))
    #> C-Inner-Syntax.command0
        (C-Inner-Toplevel.keep'' o C-Inner-Isar-Cmd.print-term)
        (C-Token.syntax' (Scan.succeed [] -- Parse.term))
        (term_outer, here, here, here))
end
›

C ‹
int z = z;
 /*@ C  ‹//@ term_outer ‹C_expr‹z›››
     C' ‹//@ term_outer ‹C_expr‹z›››
          term_outer ‹C_expr‹z››
     C  ‹//@ term_inner ‹C_expr‹z›››
     C' ‹//@ term_inner ‹C_expr‹z›››
          term_inner ‹C_expr‹z›› */›
term ‹C_expr‹z››

C ‹
int z = z;
 /*@ C  ‹//@ term_outer ‹C_expr‹z›››
     C' ‹//@ term_outer ‹C_expr‹z›››
          term_outer ‹C_expr‹z››
     C  ‹//@ term_inner ‹C_expr‹z›››
     C' ‹//@ term_inner ‹C_expr‹z›››
```

$term_{inner}$ ‹$\mathbf{C}_{expr}$‹z›› $*/$›
term ‹$\mathbf{C}_{expr}$‹z››

declare [[C_{env0} = *last*]]

C ‹
int z = z;
/@ C ‹//@ term*$_{outer}$ ‹$\mathbf{C}_{expr}$‹z›››
C′ ‹//@ term$_{outer}$ ‹$\mathbf{C}_{expr}$‹z›››
$term_{outer}$ ‹$\mathbf{C}_{expr}$‹z››
C ‹//@ term$_{inner}$ ‹$\mathbf{C}_{expr}$‹z›››
C′ ‹//@ term$_{inner}$ ‹$\mathbf{C}_{expr}$‹z›››
$term_{inner}$ ‹$\mathbf{C}_{expr}$‹z›› $*/$›
term ‹$\mathbf{C}_{expr}$‹z››

declare [[C_{env0} = *empty*]]

C — Propagation of report environment while manually composing at ML level ‹
int a;
int f (int b) {
int c = 0;
/@ ≈setup ‹fn - => fn - => fn env =>*
C′ env ‹int d = a + b + c + d; //@ $term_{inner}$ ‹$\mathbf{C}_{expr}$‹c› + $\mathbf{C}_{expr}$‹d›› $term_{outer}$ ‹$\mathbf{C}_{expr}$‹c›
+ $\mathbf{C}_{expr}$‹d›››
#> C ‹int d = a + b + c + d; //@ $term_{inner}$ ‹$\mathbf{C}_{expr}$‹c› + $\mathbf{C}_{expr}$‹d›› $term_{outer}$ ‹$\mathbf{C}_{expr}$‹c›
+ $\mathbf{C}_{expr}$‹d›››
#> C′ env ‹int d = a + b + c + d; //@ $term_{inner}$ ‹$\mathbf{C}_{expr}$‹c› + $\mathbf{C}_{expr}$‹d›› $term_{outer}$
‹$\mathbf{C}_{expr}$‹c› + $\mathbf{C}_{expr}$‹d›››
#> C ‹int d = a + b + c + d; //@ $term_{inner}$ ‹$\mathbf{C}_{expr}$‹c› + $\mathbf{C}_{expr}$‹d›› $term_{outer}$ ‹$\mathbf{C}_{expr}$‹c›
+ $\mathbf{C}_{expr}$‹d›››
›
$term_{inner}$ ‹$\mathbf{C}_{expr}$‹c› + $\mathbf{C}_{expr}$‹d››
$term_{outer}$ ‹$\mathbf{C}_{expr}$‹c› + $\mathbf{C}_{expr}$‹d›› $*/$
int e = a + b + c + d;
}›

6.7 Calculation in Directives

6.7.1 Annotation Command Classification

C — Lexing category vs. parsing category ‹
int a = 0;

// — Category 2: only parsing

//@ ≈setup ‹K (K (K I))› (evaluation at parsing *)*
//@@ ≈setup⇓ ‹K (K (K I))› (evaluation at parsing *)*

//@ highlight (evaluation at parsing *)*

```
//@@  highlight⇓          (* evaluation at parsing *)

// — Category 3: with lexing

//@  #setup  I            (* evaluation at lexing (and directives resolving) *)
//@   setup  I            (* evaluation at parsing *)
//@@  setup⇓ I            (* evaluation at parsing *)

//@  #ML     I            (* evaluation at lexing (and directives resolving) *)
//@   ML     I            (* evaluation at parsing *)
//@@  ML⇓    I            (* evaluation at parsing *)

//@  #C      ‹›           (* evaluation at lexing (and directives resolving) *)
//@   C      ‹›           (* evaluation at parsing *)
//@@  C⇓     ‹›           (* evaluation at parsing *)
›

C — Scheduling example ‹
//@+++++   ML  ‹writeln 2›
int a = 0;
//@@       ML⇓ ‹writeln 3›
//@       #ML  ‹writeln 1›
›

C — Scheduling example ‹
//*  lemma True  by simp
//* #lemma True #by simp
//* #lemma True  by simp
//*  lemma True #by simp
›

C — Scheduling example ‹ /*@
lemma ‹1 = one›
      ‹2 = two›
      ‹two + one = three›
by auto

#definition [simp]: ‹three = 3›
#definition [simp]: ‹two   = 2›
#definition [simp]: ‹one   = 1›
*/ ›
```

6.7.2 Generalizing ML Antiquotations with C Directives

```
ML ‹
structure Directive-setup-define = Generic-Data
(
  type T = int
  val empty = 0
```

```
  val merge = K empty
)

fun setup-define1 pos f =
  C-Directive.setup-define
    pos
    (fn toks => fn (name, (pos1, -)) =>
      tap (fn - => writeln (Executing " ^ name ^ Position.here pos1 ^ " (only once)))
      #> pair (f toks))
    (K I)

fun setup-define2 pos = C-Directive.setup-define pos (K o pair)
›

C — General scheme of C antiquotations ‹
/*@
    #setup — Overloading #define ‹
      setup-define2
        here
        (fn (name, (pos1, -)) =>
          op ‘ Directive-setup-define.get
          #>> (case name of f3 => curry op * 152263 | - => curry op + 1)
          #>  tap (fn (nb, -) =>
                  tracing (Executing antiquotation " ^ name ^ Position.here pos1
                    ^ " (number = " ^ Int.toString nb ^ )))
          #>  uncurry Directive-setup-define.put)
    ›
*/
#define f1
#define f2 int a = 0;
#define f3
      f1
      f2
      f1
      f3

//@ #setup — Resetting #define ‹setup-define2 here (K I)›
      f3
#define f3
      f3
›

C — Dynamic token computing in #define ‹

//@ #setup ‹setup-define1 here (K [])›
#define f int a = 0;
      f f f f

//@ #setup ‹setup-define1 here (fn toks => toks @ toks)›
```

```
#define f int b = a;
        f f

//@ #setup ‹setup-define1 here I›
#define f int a = 0;
        f f
›
```

6.8 Miscellaneous

C — Antiquotations acting on a parsed-subtree ‹

```
# /**/ include  <a\b\\c> // backslash rendered unescaped
f(){0 +  0;} /**/  // val - : theory => 'a => theory
# /* context */ if if elif
#include <stdio.h>
if then else ;
# /* zzz */  elif /**/
#else\

#define FOO  00 0  ((
FOO(FOO(a,b,c))
#endif›
```

C — Header-names in directives ‹

```
#define F <stdio.h>
#define G stdio\h // expecting an error whenever expanded
#define H stdio-h // can be used anywhere without errors
int f = /*F*/ ;
int g = /*G*/ ;
int h =   H   ;

#include F
›
```

C — Parsing tokens as directives only when detecting space symbols before # ‹/*

```
 */ \
    \

 //
        #  /*
*/   define /**/ \
 a
a a /*#include <>*/ // must not be considered as a directive
›
```

C — Universal character names in identifiers and Isabelle symbols ‹

```
#include <stdio.h>
int main () {
  char * - = \x00001;
```

```
  char * ă-ă = ă;
  char * ó⊕ò = ó⊕ò;
  printf (%s %s, ó⊕ò, -ă);
}
›

— The core lexer ...
ML‹ C-Parse.ML-source ›

declare[[C_env0 = last]]
ML‹@{C_env}›


ML‹C-Stack.Data-Lang.get' :
   Context.generic ->
        (LALR-Table.state, C-Grammar-Rule.svalue0, Position.T) C-Env.stack-elem0 list *
C-Env.env-lang;
   C-Parse.C-source: Input.source C-Parse.parser ;
   C-Inner-Syntax.command0 ;
   C';
   C;

›

declare [[C_rule0 = expression]]

ML‹
val src = ‹a + b›;
val ctxt = (Context.Proof @{context});
val ctxt' = C' @{C_env} src ctxt;
C-Module.Data-In-Env.get ctxt'
›

ML‹val - = @{term ‹3::nat›}›
ML‹ ML-Antiquotation.inline-embedded;
›


declare [[C_rule0 = translation-unit]]

end
```

7 Examples from the F-IDE Paper

theory *C-paper*
imports *../main/C-Main*
begin

This theory contains the examples presented in F-IDE 2019 paper [23].

7.1 Setup

ML‹
— Annotation Commands Mimicking the **setup** command
val - = Theory.setup
*(C-Inner-Syntax.command C-Inner-Isar-Cmd.setup' C-Parse.ML-source (≃setup, **here**, **here**))*

val C = C-Module.C'

fun C-def dir name - - =
Context.map-theory
(C-Inner-Syntax.command'
(C-Inner-Syntax.drop1
(C-Scan.Right ((fn src => fn context =>
C-Module.C' (SOME (C-Stack.Data-Lang.get' context |> #2)) src context)
, dir)))
C-Parse.C-source
name)

— Defining the ML Antiquotation *C-def* to define on the fly new C annotation commands
local
in
val - = Theory.setup
(ML-Antiquotation.declaration
@{binding C-def}
(Scan.lift (Parse.sym-ident -- Parse.position Parse.name))
(fn - => fn (top-down, (name, pos)) =>
tap (fn ctxt => Context-Position.reports ctxt [(pos, Markup.keyword1)]) #>
C-Context.fun-decl
cmd x (C-def
^(case top-down of ⇑ => C-Inner-Syntax.bottom-up
| ⇓ => C-Env.Top-down
| - => error Illegal symbol)
^ (\ ^name ^\, ^ML-Syntax.print-position pos ^))))
end

›

The next command is predefined here, so that the example below can later refer to the constant.

definition [*simplified*]: *UINT-MAX* ≡ (*2* :: *nat*) ^ *32* − *1*

7.2 Defining Annotation Commands

ML — *Isabelle-C.C-Command* ‹
local
datatype antiq-hol = *Invariant of string* (∗ *term* ∗)
val scan-colon = *C-Parse.*$$$: >> *SOME*
fun command cmd scan0 scan f =
  C-Annotation.command' cmd (*K* (*scan0* −− (*scan* >> *f*)
                                >> *K C-Env.Never*))
in
val - = *Theory.setup* ((∗ *1* '@' ∗)
                  command (*INVARIANT*, ***here***) *scan-colon C-Parse.term Invariant*
                #> *command* (*INV*, ***here***) *scan-colon C-Parse.term Invariant*)
end
›

Demonstrating the Effect of Annotation Command Context Navigation

C ‹
int sum1(*int a*)
{
  while (*a* < *10*)
    /∗@ @ *INV*: ‹. . . ›
        @ *highlight* ∗/
    { *a* = *a* + *1*; }
  return a;
}›

C ‹
int sum2(*int a*)
/∗@ ++@ *INV*: ‹. . . ›
    ++@ *highlight* ∗/
{
  while (*a* < *10*)
    { *a* = *a* + *1*; }
  return a;
}›

C — starting environment = empty ‹

```
int a (int b) { return &a + b + c; }
/*@ ≃setup ‹fn stack-top => fn env =>
          C (SOME env) ‹int c = &a + b + c;››
   ≃setup ‹fn stack-top => fn env =>
          C  NONE      ‹int c = &a + b + c;››
   declare [[C_env0 = last]]
   C       (*SOME*)    ‹int c = &a + b + c;›
*/›
```

7.3 Proofs inside C-Annotations

```
C ‹
#define SQRT-UINT-MAX 65536
/*@ lemma uint-max-factor [simp]:
     UINT-MAX = SQRT-UINT-MAX * SQRT-UINT-MAX - 1
   by (clarsimp simp: UINT-MAX-def SQRT-UINT-MAX-def)
*/›

term SQRT-UINT-MAX
```

7.4 Scheduling the Effects on the Logical Context

```
C ‹int -;
/*@ @ C ‹//@ C1 ‹int -; //@ @ ≃setup⇓ ‹@{C-def ⇑ C2}› \
                          @ C1  ‹//* C2 ‹int -;›› \
                          @ C1⇓ ‹//* C2 ‹int -;›› ››
   @ C ‹//* C2 ‹int -;›                          ›
     ≃setup ‹@{C-def ⇑ (* bottom−up *) C1 }›
     ≃setup ‹@{C-def ⇓ (* top−down  *) C1⇓}›
*/›
```

7.5 As Summary: A Spaghetti Language — Bon Appétit!

... with the Bonus of a local C-inside-ML-inside-C-inside-Isar ...

```
ML‹
fun highlight (-, (-, pos1, pos2)) =
  tap (fn - => Position.reports-text [((Position.range (pos1, pos2)
                                |> Position.range-position, Markup.intensify), )])
›
```

C — the command starts with a default empty environment

```
‹int f (int a)
  //@ ++& ≃setup ‹fn stack-top => fn env => highlight stack-top›
  { /*@ @ ≃setup ‹fn stack-top => fn env =>
                C (SOME env) (* the command starts with some provided environment *)
                 ‹int b = a + b; //@ C1' ‹int c; //@ @ ≃setup⇓ ‹@{C-def ⇑ C2'}› \
                                          @ C1'  ‹//* C2' ‹int d;›› \
```

```
                                                @ C1'⇓ ‹//* C2' ‹int d;››          ›
                    int b = a + b + c + d;››
      @ ≃setup ‹fn stack-top => fn env => C NONE ‹#define int int
                                                int b = a + b; //* C2' ‹int c = b;›››
        ≃setup ‹@{C-def ⇑ (* bottom−up *)  C1'  }›
        ≃setup ‹@{C-def ⇓ (* top−down  *) C1'⇓}›
    */
    return a + b + c + d; /* explicit highlighting */ }›
```

Note that in the current design-implementation of Isabelle/C, C directives have a propagation side-effect to any occurring subsequent C annotations, even if C directives are supposed to be all evaluated before any C code. (Making such effect inexistent would be equally easier to implement though, this is what was the default behavior of directives in previous versions of Isabelle/C.)

end

Bibliography

[1] R. Aïssat, F. Voisin, and B. Wolff. Infeasible paths elimination by symbolic execution techniques - proof of correctness and preservation of paths. In *Interactive Theorem Proving - 7th International Conference, ITP 2016, Nancy, France, August 22-25, 2016, Proceedings*, pages 36–51, 2016. doi: 10.1007/978-3-319-43144-4_3. URL https://doi.org/10.1007/978-3-319-43144-4_3.

[2] B. Barras, L. D. C. González-Huesca, H. Herbelin, Y. Régis-Gianas, E. Tassi, M. Wenzel, and B. Wolff. Pervasive parallelism in highly-trustable interactive theorem proving systems. In J. Carette, D. Aspinall, C. Lange, P. Sojka, and W. Windsteiger, editors, *Intelligent Computer Mathematics - MKM, Calculemus, DML, and Systems and Projects 2013, Held as Part of CICM 2013, Bath, UK, July 8-12, 2013. Proceedings*, volume 7961 of *Lecture Notes in Computer Science*, pages 359–363. Springer, 2013. ISBN 978-3-642-39319-8. doi: 10.1007/978-3-642-39320-4_29. URL https://doi.org/10.1007/978-3-642-39320-4_29.

[3] J. A. Bockenek, P. Lammich, Y. Nemouchi, and B. Wolff. Using Isabelle/UTP for the Verification of Sorting Algorithms A Case Study, July 2018. URL https://easychair.org/publications/preprint/CxRV. Isabelle Workshop 2018, Colocated with Interactive Theorem Proving. As part of FLOC 2018, Oxford, GB.

[4] A. D. Brucker, F. Tuong, and B. Wolff. Featherweight OCL: A proposal for a machine-checked formal semantics for OCL 2.5. *Archive of Formal Proofs*, 2014, 2014. URL https://www.isa-afp.org/entries/Featherweight_OCL.shtml.

[5] A. D. Brucker, I. Aït-Sadoune, P. Crisafulli, and B. Wolff. Using the isabelle ontology framework - linking the formal with the informal. In *Intelligent Computer Mathematics - 11th International Conference, CICM 2018, Hagenberg, Austria, August 13-17, 2018, Proceedings*, pages 23–38, 2018. doi: 10.1007/978-3-319-96812-4_3. URL https://doi.org/10.1007/978-3-319-96812-4_3.

[6] E. Carrascosa, J. Coronel, M. Masmano, P. Balbastre, and A. Crespo. Xtratum hypervisor redesign for LEON4 multicore processor. *SIGBED Review*, 11(2): 27–31, 2014. doi: 10.1145/2668138.2668142. URL https://doi.org/10.1145/2668138.2668142.

[7] CEA-List. The frama-c home page, Jan. 2019. URL https://frama-c.com. Accessed March 24, 2019.

[8] E. Cohen, M. Dahlweid, M. A. Hillebrand, D. Leinenbach, M. Moskal, T. Santen, W. Schulte, and S. Tobies. VCC: A practical system for verifying concurrent C. In

Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17-20, 2009. Proceedings, pages 23–42, 2009. doi: 10.1007/978-3-642-03359-9_2. URL https://doi.org/10.1007/978-3-642-03359-9_2.

[9] J. Earley. An efficient context-free parsing algorithm. *Commun. ACM*, 13(2):94–102, 1970. doi: 10.1145/362007.362035. URL https://doi.org/10.1145/362007.362035.

[10] D. Greenaway, J. Lim, J. Andronick, and G. Klein. Don't sweat the small stuff: formal verification of C code without the pain. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, pages 429–439, 2014. doi: 10.1145/2594291.2594296. URL http://doi.acm.org/10.1145/2594291.2594296.

[11] Z. Hou, D. Sanán, A. Tiu, and Y. Liu. Proof tactics for assertions in separation logic. In M. Ayala-Rincón and C. A. Muñoz, editors, *Interactive Theorem Proving - 8th International Conference, ITP 2017, Brasília, Brazil, September 26-29, 2017, Proceedings*, volume 10499 of *Lecture Notes in Computer Science*, pages 285–303. Springer, 2017. ISBN 978-3-319-66106-3. doi: 10.1007/978-3-319-66107-0_19. URL https://doi.org/10.1007/978-3-319-66107-0_19.

[12] G. Hutton. Higher-order functions for parsing. *J. Funct. Program.*, 2(3): 323–343, 1992. doi: 10.1017/S0956796800000411. URL https://doi.org/10.1017/S0956796800000411.

[13] J. Jourdan and F. Pottier. A simple, possibly correct LR parser for C11. *ACM Trans. Program. Lang. Syst.*, 39(4):14:1–14:36, 2017. doi: 10.1145/3064848. URL https://doi.org/10.1145/3064848.

[14] C. Keller. Tactic program-based testing and bounded verification in isabelle/hol. In *Tests and Proofs - 12th International Conference, TAP 2018, Held as Part of STAF 2018, Toulouse, France, June 27-29, 2018, Proceedings*, pages 103–119, 2018. doi: 10.1007/978-3-319-92994-1_6. URL https://doi.org/10.1007/978-3-319-92994-1_6.

[15] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. sel4: formal verification of an OS kernel. In J. N. Matthews and T. E. Anderson, editors, *Proceedings of the 22nd ACM Symposium on Operating Systems Principles 2009, SOSP 2009, Big Sky, Montana, USA, October 11-14, 2009*, pages 207–220. ACM, 2009. ISBN 978-1-60558-752-3. doi: 10.1145/1629575.1629596. URL https://doi.org/10.1145/1629575.1629596.

[16] G. Klein, J. Andronick, K. Elphinstone, T. C. Murray, T. Sewell, R. Kolanski, and G. Heiser. Comprehensive formal verification of an OS microkernel. *ACM Trans. Comput. Syst.*, 32(1):2:1–2:70, 2014. doi: 10.1145/2560537. URL http://doi.acm.org/10.1145/2560537.

[17] P. Lammich and S. Wimmer. IMP2 - simple program verification in isabelle/hol. *Archive of Formal Proofs*, 2019, 2019. URL https://www.isa-afp.org/entries/IMP2.html.

[18] D. Leinenbach and T. Santen. Verifying the microsoft hyper-v hypervisor with VCC. In *FM 2009: Formal Methods, Second World Congress, Eindhoven, The Netherlands, November 2-6, 2009. Proceedings*, pages 806–809, 2009. doi: 10.1007/978-3-642-05089-3_51. URL https://doi.org/10.1007/978-3-642-05089-3_51.

[19] X. Leroy. Formal verification of a realistic compiler. *Commun. ACM*, 52(7):107–115, 2009. doi: 10.1145/1538788.1538814. URL http://doi.acm.org/10.1145/1538788.1538814.

[20] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer, 2002. ISBN 3-540-43376-7. doi: 10.1007/3-540-45949-9. URL https://doi.org/10.1007/3-540-45949-9.

[21] D. Sanán, Y. Zhao, Z. Hou, F. Zhang, A. Tiu, and Y. Liu. Csimpl: A rely-guarantee-based framework for verifying concurrent programs. In A. Legay and T. Margaria, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 23rd International Conference, TACAS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings, Part I*, volume 10205 of *Lecture Notes in Computer Science*, pages 481–498, 2017. ISBN 978-3-662-54576-8. doi: 10.1007/978-3-662-54577-5_28. URL https://doi.org/10.1007/978-3-662-54577-5_28.

[22] F. Tuong and B. Wolff. A meta-model for the isabelle API. *Archive of Formal Proofs*, 2015, 2015. URL https://www.isa-afp.org/entries/Isabelle_Meta_Model.shtml.

[23] F. Tuong and B. Wolff. Deeply Integrating C11 Code Support into Isabelle/PIDE. In R. Monahan, V. Prevosto, and J. Proença, editors, *Proceedings 5th Workshop on Formal Integrated Development Environment, F-IDE@FM 2019, Porto, Portugal, 7 October 2019.*, volume 310 of *EPTCS*, 2019. doi: 10.4204/EPTCS.310.3. URL http://dx.doi.org/10.4204/EPTCS.310.3.

[24] F. Tuong and B. Wolff. Clean - an abstract imperative programming language and its theory. *Archive of Formal Proofs*, 2019, 2019. URL https://www.isa-afp.org/entries/Clean.html.

[25] M. Wenzel. Asynchronous user interaction and tool integration in isabelle/pide. In G. Klein and R. Gamboa, editors, *Interactive Theorem Proving - 5th International Conference, ITP 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 14-17, 2014. Proceedings*, volume 8558 of *Lecture Notes in Computer Science*, pages 515–530. Springer, 2014. ISBN 978-3-319-08969-0. doi: 10.1007/978-3-319-08970-6_33. URL https://doi.org/10.1007/978-3-319-08970-6_33.

[26] M. Wenzel. System description: Isabelle/jedit in 2014. In C. Benzmüller and B. Woltzenlogel Paleo, editors, *Proceedings Eleventh Workshop on User Interfaces for Theorem Provers, UITP 2014, Vienna, Austria, 17th July 2014.*, volume 167 of *EPTCS*, pages 84–94, 2014. doi: 10.4204/EPTCS.167.10. URL https://doi.org/10.4204/EPTCS.167.10.

[27] S. Winwood, G. Klein, T. Sewell, J. Andronick, D. Cock, and M. Norrish. Mind the gap. In S. Berghofer, T. Nipkow, C. Urban, and M. Wenzel, editors, *Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17-20, 2009. Proceedings*, volume 5674 of *Lecture Notes in Computer Science*, pages 500–515. Springer, 2009. ISBN 978-3-642-03358-2. doi: 10.1007/978-3-642-03359-9_34. URL https://doi.org/10.1007/978-3-642-03359-9_34.