LL(1) Parser Generator

Sarah Tilscher and Simon Wimmer

March 17, 2025

Abstract

In this formalization, we implement an LL(1) parser generator that first pre-computes the NULLABLE set, FIRST map and FOLLOW map, to then build a lookahead table. We prove correctness, soundness and error-free termination for LL(1) grammars. We provide the JSON grammar and show how to parse a tokenized JSON string using a parser created with the verified parser generator. The proof structure is significantly based on Vermillion [2], an LL(1) parser generator verified in Coq.

Contents

1 Introduction

An LL Parser is a top-down parser, i.e., it constructs the parse tree starting at the root by selecting rules for the expansion of non-terminals. The selection between alternative rules is in general not deterministic as there may be multiple rules applicable for some non-terminal. For a more substantiated decision, an LL(k) parser can inspect the first k symbols of the remaining input. Grammars for which a lookahead of length k is sufficient to deterministically choose the correct rule for expansion are called LL(k) grammars. In this work, we focus on LL(1) parsers, i.e., parsers that only spy on the single next symbol of the remaining input [3].

In the first few theories we provide fixpoint algorithms to successively compute the NULLABLE set, FIRST and FOLLOW map for a grammar, and prove their termination and the soundness and completeness of the result. With these pre-computed attributes, a parse table recording which alternative rule needs to be chosen for which lookahead can be generated for LL(k) grammars. In case the input grammar is not LL(k), an ambiguous lookahead will be detected, and the parse table generation will return an error. For the case that a parse table is generated successfully, we prove its soundness and completeness. As a last step, a function for parsing a tokenized input with the help of the generated parse table is provided. It

either returns a parse tree as a result, rejects the input if it is not within the language described by the grammar, or returns an error. The soundness and completeness of the parser follow from the correctness theorems about the parse table. Additionally, the parser is shown to terminate without error for any successfully generated parse table.

For demonstration, we generate two parsers — one for a mini programming language and one for JSON strings — and use them to parse small tokenized strings.

2 Types and Definitions

theory *Grammar*
imports *Main*
begin

2.1 Grammar

We first define the datatypes for a grammar. A symbol is either a non-terminal of type *'n* or a terminal of type *'t*. A production is then a tuple of a non-terminal, and a list of symbols. An empty list of symbols corresponds to the empty word. A grammar is defined through a non-terminal as start symbol and a list of productions. Note that there may be more than one production for some non-terminal.

datatype *('n, 't) symbol = NT 'n | T 't*

type_synonym *('n, 't) rhs = "(('n, 't) symbol) list"*

type_synonym *('n, 't) prod = "'n × ('n, 't) rhs"*
type_synonym *('n, 't) prods = "('n, 't) prod list"*

datatype *('n, 't) grammar = G (start: "'n") (prods: "('n, 't) prods")*

An LL(1) parser considers a lookahead of size one to determine the appropriate rule for the next expansion. A lookahead may either be a terminal symbol or *EOF*, the special lookahead to mark the end of input.

datatype *'t lookahead = LA 't | EOF*

2.2 Definition of Nullable, First, Follow and Lookahead

The set of nullable symbols contains all nonterminals from which the empty word can be derived. This is the case, either when there is a production for the non-terminal with an empty right-hand side or when the right-hand side consists only of nullable symbols.

inductive *nullable_sym :: "('n, 't) grammar ⇒ ('n, 't) symbol ⇒ bool"*
    and *nullable_gamma :: "('n, 't) grammar ⇒ ('n, 't) rhs ⇒ bool"*

```
for g where
  NullableSym:
  "(x, gamma) ∈ set (prods g) ⟹ nullable_gamma g gamma
  ⟹ nullable_sym g (NT x)"
| NullableNil:
  "nullable_gamma g []"
| NullableCons:
  "nullable_sym g s ⟹ nullable_gamma g ss
  ⟹ nullable_gamma g (s # ss)"
```

First symbols are all symbols that are prefixes of possible derivations. For some lookahead, this is the terminal corresponding to the lookahead, and all non-terminals for which there exists a production where a first symbol occurs after a nullable prefix.

```
inductive first_sym
  :: "('n, 't) grammar ⇒ 't lookahead ⇒ ('n, 't) symbol ⇒ bool"
for g where
  FirstT: "first_sym g (LA y) (T y)"
| FirstNT:
  "(x, gpre @ s # gsuf) ∈ set (prods g) ⟹ nullable_gamma g gpre
  ⟹ first_sym g la s
  ⟹ first_sym g la (NT x)"

inductive first_gamma
  :: "('n, 't) grammar ⇒ 't lookahead ⇒ ('n, 't) symbol list ⇒ bool"
for g where
  FirstGamma:
  "nullable_gamma g gpre ⟹ first_sym g la s
  ⟹ first_gamma g la (gpre @ s # gsuf)"
```

The set of follow symbols contains for some non-terminal all symbols that may directly follow after a derivation for it. For the start symbol a follow symbol is EOF. In general, follow symbols of some non-terminal are all first symbols of the list of symbols following after an occurrence of this non-terminal in the productions right-hand sides. In case the list of symbols following a non-terminal in a production's right-hand side is nullable, the non-terminal on the left-hand side of the production is a follow symbol of it as well.

```
inductive follow_sym :: "('n, 't) grammar ⇒ 't lookahead ⇒ ('n, 't)
symbol ⇒ bool"
for g where
  FollowStart: "follow_sym g EOF (NT (start g))"
| FollowRight:
  "(x1, gpre @ (NT x2) # gsuf) ∈ set (prods g)
  ⟹ first_gamma g la gsuf
  ⟹ follow_sym g la (NT x2)"
| FollowLeft : "(x1, gpre @ (NT x2) # gsuf) ∈ set (prods g)
  ⟹ nullable_gamma g gsuf
```

```
  ⟹ follow_sym g la (NT x1)
  ⟹ follow_sym g la (NT x2)"
```

A symbol is a lookahead for some production if it is either a first symbol of the production's right-hand side or, when the right-hand side is nullable, a follow symbol of the non-terminal on the production's left-hand side.

```
definition lookahead_for
  :: "'t lookahead ⇒ 'n ⇒ ('n, 't) rhs ⇒ ('n, 't) grammar ⇒ bool"
where
  "lookahead_for la x gamma g = (
    first_gamma g la gamma
    ∨ (nullable_gamma g gamma ∧ follow_sym g la (NT x)))"
```

2.3 Left-Recursive Grammars

A left-recursive grammar is a grammar where some non-terminal symbol can be reached from the same non-terminal symbol via some nullable path. LL(1) grammars may not be left-recursive. We give a definition for left-recursive grammars to later use it as an error condition for parsing.

```
inductive nullable_path ::
  "('n, 't) grammar ⇒ 't lookahead ⇒ ('n, 't) symbol ⇒ ('n, 't) symbol
  ⇒ bool"
where
  DirectPath: "(x, gamma) ∈ set (prods g) ⟹ gamma = gpre @ NT z # gsuf
  ⟹ nullable_gamma g gpre ⟹ lookahead_for la x gamma g
  ⟹ nullable_path g la (NT x) (NT z)"
| IndirectPath: "(x, gamma) ∈ set (prods g)
  ⟹ gamma = gpre @ NT y # gsuf
  ⟹ nullable_gamma g gpre ⟹ lookahead_for la x gamma g
  ⟹ nullable_path g la (NT y) (NT z)
  ⟹ nullable_path g la (NT x) (NT z)"

abbreviation left_recursive ::
  "('n, 't) grammar ⇒ ('n, 't) symbol ⇒ 't lookahead ⇒ bool"
where
  "left_recursive g s la ≡ nullable_path g la s s"

end
```

3 Nullable Set

```
theory Nullable_Set
imports Grammar
begin


definition lhSet :: "('n, 't) prods ⇒ 'n set" where
```

```
  "lhSet ps = set (map fst ps)"

fun nullableGamma :: "('n, 't) rhs ⇒ 'n set ⇒ bool" where
  "nullableGamma [] _ = True"
| "nullableGamma ((T _)#_) _ = False"
| "nullableGamma ((NT x)#gamma') nu = (if x ∈ nu then nullableGamma gamma'
nu else False)"

definition updateNu :: "('n, 't) prod ⇒ 'n set ⇒ 'n set" where
  "updateNu ≡ λ(x, gamma) nu. (if nullableGamma gamma nu then insert
x nu else nu)"

definition nullablePass :: "('n, 't) prods ⇒ 'n set ⇒ 'n set" where
  "nullablePass ps nu = foldr updateNu ps nu"

function mkNullableSet' :: "('n, 't) prods ⇒ 'n set ⇒ 'n set" where
  "mkNullableSet' ps nu = (let nu' = nullablePass ps nu in
  (if nu=nu' then nu else mkNullableSet' ps nu'))"
  ⟨proof⟩

definition mkNullableSet :: "('n, 't) grammar ⇒ 'n set" where
  "mkNullableSet g = mkNullableSet' (prods g) {}"
```

3.1 Termination

```
definition countNullCands :: "('n, 't) prods ⇒ 'n set ⇒ nat" where
  "countNullCands ps nu = (let candidates = lhSet ps in card (candidates
- nu))"

lemma nullablePass_subset: "(nu::'n set) ⊆ (nullablePass ps nu)"
  ⟨proof⟩

lemma nullablePass_Nil[simp]: "nullablePass [] nu = nu"
  ⟨proof⟩

lemma nullablePass_cons[simp]: "nullablePass ((y,gamma)#ps') nu =
  (if nullableGamma gamma (nullablePass ps' nu) then insert y else id)
(nullablePass ps' nu)"
  ⟨proof⟩

lemma nullable_pass_mono:
  "nullablePass ps nu ⊆ nullablePass (qs @ ps) nu"
  ⟨proof⟩

lemma nullablePass_subset_lhSet:
  "nullablePass ps nu ⊆ lhSet ps ∪ nu"
  ⟨proof⟩

lemma nullablePass_neq_candidates_lt:
```

assumes *"nu ≠ nullablePass ps nu"*
shows *"countNullCands ps (nullablePass ps nu) < countNullCands ps nu"*
⟨*proof*⟩

termination *mkNullableSet'*
⟨*proof*⟩

3.2 Correctness Definitions

definition *nullable_set_sound :: "'n set ⇒ ('n, 't) grammar ⇒ bool"* **where**
"nullable_set_sound nu g = (∀x ∈ nu. nullable_sym g (NT x))"

definition *nullable_set_complete :: "'n set ⇒ ('n, 't) grammar ⇒ bool"* **where**
"nullable_set_complete nu g = (∀x. nullable_sym g (NT x) ⟶ x ∈ nu)"

abbreviation *nullable_set_for :: "'n set ⇒ ('n, 't) grammar ⇒ bool"* **where**
"nullable_set_for nu g ≡ nullable_set_sound nu g ∧ nullable_set_complete nu g"

3.3 Soundness

lemma *nu_sound_nullableGamma_sound:*
"nullable_set_sound nu g ⟹ nullableGamma gamma nu ⟹ nullable_gamma g gamma"
⟨*proof*⟩

lemma *nullablePass_preserves_soundness':*
"nullable_set_sound nu g ⟹ set ps ⊆ set (prods g)
⟹ nullable_set_sound (nullablePass ps nu) g"
⟨*proof*⟩

lemma *nullablePass_preserves_soundness:*
"nullable_set_sound nu g ⟹ nullable_set_sound (nullablePass (prods g) nu) g"
⟨*proof*⟩

lemmas *[simp del] = mkNullableSet'.simps*

lemma *mkNullableSet'_preserves_soundness:*
"nullable_set_sound nu g ⟹ nullable_set_sound (mkNullableSet' (prods g) nu) g"
⟨*proof*⟩

lemma *empty_nu_sound: "nullable_set_sound {} g"*
⟨*proof*⟩

theorem *mkNullableSet_sound: "nullable_set_sound (mkNullableSet g) g"*
⟨*proof*⟩

3.4 Completeness

lemma *nullablePass_add_equal: "x ∈ nu ⟹ nullablePass ps nu = insert x (nullablePass ps nu)"*
⟨*proof*⟩

lemma *nullable_gamma_nullableGamma_true:*
"nullable_gamma g ys ⟹ ∀x. (NT x) ∈ set ys ⟶ x ∈ nu ⟹ nullableGamma ys nu"
⟨*proof*⟩

lemma *nullableGamma_saturated_if_nullablePass_fixpoint:*
assumes *"nu = nullablePass ps nu"*
shows *"∀(x, gamma) ∈ set ps. nullableGamma gamma nu ⟶ x ∈ nu"*
⟨*proof*⟩

lemma *nullablePass_equal_complete':*
assumes *"nu = nullablePass (prods g) nu"*
shows *"nullable_sym g s ⟹ ∀y. s = NT y ⟶ y ∈ nu"*
"nullable_gamma g ys ⟹ ∀y. NT y ∈ set ys ⟶ y ∈ nu"
⟨*proof*⟩

lemma *nullablePass_equal_complete: "nu = (nullablePass (prods g) nu) ⟹ nullable_set_complete nu g"*
⟨*proof*⟩

lemma *mkNullableSet'_complete: "nullable_set_complete (mkNullableSet' (prods g) nu) g"*
⟨*proof*⟩

theorem *mkNullableSet_complete: "nullable_set_complete (mkNullableSet g) g"*
⟨*proof*⟩

theorem *mkNullableSet_correct: "nullable_set_for (mkNullableSet g) g"*
⟨*proof*⟩

end

4 First map

theory *First_Map*
imports *Nullable_Set "HOL-Library.Finite_Map"*
begin

type_synonym *('n, 't) first_map = "('n, 't lookahead list) fmap"*

fun *nullableSym :: "('n, 't) symbol ⇒ 'n set ⇒ bool"* **where**
"nullableSym (T _) _ = False"

```
| "nullableSym (NT x) nu = (x ∈ nu)"
```

definition *findOrEmpty :: "'n ⇒ ('n, 't) first_map ⇒ 't lookahead list"*
where
"findOrEmpty x m = (case fmlookup m x of None ⇒ [] | Some y ⇒ y)"

fun *firstSym :: "('n, 't) symbol ⇒ ('n, 't) first_map ⇒ 't lookahead list"* **where**
"firstSym (T x) _ = [LA x]"
| "firstSym (NT x) fi = findOrEmpty x fi"

definition *list_union :: "'a list ⇒ 'a list ⇒ 'a list"* (**infixr** *<@@> 65*)
where
"list_union ls1 ls2 = ls1 @ (filter (λx. x ∉ set ls1) ls2)"

lemma *in_atleast1_list: "a ∈ set (ls1 @@ ls2) ⟹ a ∈ set ls1 ∨ a ∈ set ls2"*
⟨*proof*⟩

lemma *set_list_union[simp]: "set (ls1 @@ ls2) = set ls1 ∪ set ls2"*
⟨*proof*⟩

lemma *mem_list_union: "ls1 = ls1 @@ ls2 ⟹ e ∈ set ls2 ⟹ e ∈ set ls1"*
⟨*proof*⟩

lemma *list_union_I2: "e ∈ set ls2 ⟹ e ∈ set (ls1 @@ ls2)"*
⟨*proof*⟩

fun *firstGamma :: "('n, 't) rhs ⇒ 'n set ⇒ ('n, 't) first_map ⇒ 't lookahead list"* **where**
"firstGamma [] _ _ = []"
| "firstGamma (s#gamma') nu fi =
(if nullableSym s nu then firstSym s fi @@ firstGamma gamma' nu fi else firstSym s fi)"

definition *updateFi :: "'n set ⇒ ('n, 't) prod ⇒ ('n, 't) first_map ⇒ ('n, 't) first_map"* **where**
"updateFi ≡ λnu (x, gamma) fi. (let
fg = firstGamma gamma nu fi;
xFirst = findOrEmpty x fi;
xFirst' = xFirst @@ fg in (if xFirst' = xFirst ∨ fg = [] then fi else fmupd x xFirst' fi))"

definition *firstPass :: "('n, 't) prods ⇒ 'n set ⇒ ('n, 't) first_map ⇒ ('n, 't) first_map"* **where**
"firstPass ps nu fi = foldr (updateFi nu) ps fi"

partial_function *(option) mkFirstMap' :: "('n, 't) prods ⇒ 'n set ⇒ ('n, 't) first_map*
⇒ ('n, 't) first_map option" **where**
"mkFirstMap' ps nu fi = (let fi' = firstPass ps nu fi in
(if fi = fi' then Some fi else mkFirstMap' ps nu fi'))"

definition *mkFirstMap :: "('n, 't) grammar ⇒ 'n set ⇒ ('n, 't) first_map"*
where
"mkFirstMap g nu = the (mkFirstMap' (prods g) nu fmempty)"

4.1 Termination

fun *leftmostLookahead :: "('n, 't) rhs ⇒ 't lookahead option"* **where**
"leftmostLookahead [] = None"
| "leftmostLookahead ((T y)#gamma') = Some (LA y)"
| "leftmostLookahead ((NT _)#gamma') = leftmostLookahead gamma'"

definition *leftmostLookaheads :: "('n, 't) prods ⇒ 't lookahead set"* **where**
"leftmostLookaheads ps = the ‘ leftmostLookahead ‘ snd ‘ set ps"

lemma *in_leftmostLookaheads_cons: "x ∈ leftmostLookaheads ps ⟹ x ∈ leftmostLookaheads (p # ps)"*
⟨*proof*⟩

definition *pairsOf :: "('n, 't) first_map ⇒ ('n × 't lookahead) set"*
where
"pairsOf fi = {(a, b). a |∈| fmdom fi ∧ b ∈ set (findOrEmpty a fi)}"

definition *all_nt :: "('n, 't) rhs ⇒ bool"* **where**
"all_nt gamma = (∀s ∈ set gamma. (case s of (NT _) ⇒ True | (T _) ⇒ False))"

definition *all_pairs_are_first_candidates:: "('n, 't) first_map ⇒ ('n, 't) prods ⇒ bool"* **where**
"all_pairs_are_first_candidates fi ps =
(∀x la. (x, la) ∈ pairsOf fi ⟶ (x ∈ lhSet ps ∧ la ∈ leftmostLookaheads ps))"

definition *countFirstCands :: "('n, 't) prods ⇒ ('n, 't) first_map ⇒ nat"* **where**
"countFirstCands ps fi = (let allCandidates = (lhSet ps) × (leftmostLookaheads ps) in
card (allCandidates - (pairsOf fi)))"

lemma *gpre_nullable_leftmost_lk_some: "all_nt gpre*
⟹ leftmostLookahead (gpre @ (T y) # gsuf) = Some (LA y)"
⟨*proof*⟩

lemma *gpre_nullable_in_leftmost_lks:*

```
  "(x, (gpre @ (T y) # gsuf)) ∈ set ps ⟹ all_nt gpre ⟹ (LA y) ∈ leftmostLookaheads
ps"
```
⟨*proof*⟩

lemma *in_firstGamma_in_leftmost_lks':*
  assumes *"(x, gpre @ gsuf) ∈ set ps" "all_pairs_are_first_candidates
fi ps" "all_nt gpre"*
  shows *"la ∈ set (firstGamma gsuf nu fi) ⟹ la ∈ leftmostLookaheads
ps"*
  ⟨*proof*⟩

lemma *in_firstGamma_in_leftmost_lks: "(x, gamma) ∈ set ps ⟹ all_pairs_are_first_candidat
fi ps*
  ⟹ la ∈ set (firstGamma gamma nu fi) ⟹ la ∈ leftmostLookaheads ps"
  ⟨*proof*⟩

lemma *updateFi_cases:*
  fixes *nu* **and** *x :: 'n* **and** *gamma :: "('n, 't) rhs"* **and** *fi*
  defines *"fg ≡ firstGamma gamma nu fi"*
  defines *"xFirst ≡ findOrEmpty x fi"*
  defines *"xFirst' ≡ xFirst @@ fg"*
  obtains *(unchanged) "xFirst' = xFirst" "updateFi nu (x, gamma) fi =
fi"*
  | (empty) "fg = []" "updateFi nu (x, gamma) fi = fi"
  | (new) la **where** *"xFirst' ≠ xFirst ⟹ la ∈ set fg ⟹ la ∉ set xFirst*
      ⟹ updateFi nu (x, gamma) fi = fmupd x xFirst' fi"
  ⟨*proof*⟩

lemma *firstPass_induct:*
  fixes *ps :: "('n, 't) prods"*
    and *nu :: "'n set"*
    and *fi :: "('n, 't) first_map"*
    and *P :: "('n, 't) prods ⇒ 'n set ⇒ ('n, 't) first_map ⇒ bool"*
  assumes *Nil: "P [] nu fi"*
    and *Cons_changed: "⋀p ps'. (P ps' nu fi ⟹ fi ≠ firstPass ps' nu
fi ⟹ P (p # ps') nu fi)"*
    and *Cons_same: "⋀p ps'. (P ps' nu fi ⟹ fi = firstPass ps' nu fi
⟹ P (p # ps') nu fi)"*
  shows *"P (ps :: ('n, 't) prods) nu fi"*
  ⟨*proof*⟩

lemma *in_findOrEmpty_iff_in_pairsOf: "la ∈ set (findOrEmpty x fi) ⟷
(x, la) ∈ pairsOf fi"*
  ⟨*proof*⟩

lemma *in_pairsOf_exists: "(x, la) ∈ pairsOf fi ⟷ (∃s. fmlookup fi
x = Some s ∧ la ∈ set s)"*
  ⟨*proof*⟩

lemma *in_findOrEmpty_exists_set:*
"la ∈ set (findOrEmpty x m) ⟷ (∃ s. fmlookup m x = Some s ∧ la ∈ set s)"
⟨*proof*⟩

lemma *in_add_value: "(x, la) ∈ pairsOf (fmupd x s fi) ⟷ la ∈ set s"*
⟨*proof*⟩

lemma *firstPass_Nil[simp]: "firstPass [] x y = y"*
⟨*proof*⟩

Lemma for the simplification of *firstPass*. In general, one function call should be unfolded instead of replacing it with its definition with *foldr*.

lemma *firstPass_cons[simp]: "firstPass (a # ps) nu fi = updateFi nu a (firstPass ps nu fi)"*
⟨*proof*⟩

lemma *unfold_updateFi: "updateFi nu (x, gamma) fi =*
(if findOrEmpty x fi @@ firstGamma gamma nu fi = findOrEmpty x fi
∨ findOrEmpty x fi @@ firstGamma gamma nu fi = []
then fi else fmupd x (findOrEmpty x fi @@ firstGamma gamma nu fi) fi)"
⟨*proof*⟩

lemma *in_add_keys: "la ∈ set s ⟷ (x, la) ∈ pairsOf (fmupd x s fi)"*
⟨*proof*⟩

lemma *in_add_keys_neq: "x ≠ y ⟹ (y, la) ∈ pairsOf fi ⟷ (y, la) ∈ pairsOf (fmupd x s fi)"*
⟨*proof*⟩

lemma *updateFi_subset: "pairsOf fi ⊆ pairsOf (updateFi nu p fi)"*
⟨*proof*⟩

lemma *firstPass_cons_subset: "pairsOf (firstPass ps nu fi) ⊆ pairsOf (firstPass (p # ps) nu fi)"*
⟨*proof*⟩

lemma *firstPass_mono: "pairsOf (firstPass ps nu fi) ⊆ pairsOf (firstPass (qs @ ps) nu fi)"*
⟨*proof*⟩

lemma *firstPass_subset: "pairsOf fi ⊆ pairsOf (firstPass ps nu fi)"*
⟨*proof*⟩

lemma *firstPass_empty_set:*
"fmlookup (firstPass ps nu fi) x = Some [] ⟹ fmlookup fi x = Some []"

⟨*proof*⟩

lemma *firstPass_None: "fmlookup (firstPass ps nu fi) x = None ⟹ fmlookup fi x = None"*
⟨*proof*⟩

lemma *firstPass_neq_findOrEmpty:*
assumes *"fmlookup fi x ≠ fmlookup (firstPass ps nu fi) x"*
shows *"findOrEmpty x fi ≠ findOrEmpty x (firstPass ps nu fi)"*
⟨*proof*⟩

Injectivity of *pairsOf*

lemma *firstPass_only_appends: "∃suf. findOrEmpty x (firstPass ps nu fi) = findOrEmpty x fi @ suf"*
⟨*proof*⟩

lemma *firstPass_suf_distinct: "findOrEmpty x (firstPass ps nu fi) = findOrEmpty x fi @ suf*
⟹ suf ≠ [] ⟹ la ∈ set suf ⟹ la ∉ set (findOrEmpty x fi)"
⟨*proof*⟩

lemma *pairsOf_inj: "fi ≠ firstPass ps nu fi ⟹ pairsOf fi ≠ pairsOf (firstPass ps nu fi)"*
⟨*proof*⟩

lemma *firstPass_not_equiv_subset:*
"fi ≠ firstPass ps nu fi ⟹ pairsOf fi ⊂ pairsOf (firstPass ps nu fi)"
⟨*proof*⟩

lemma *firstPass_subset_lhs_lks: "all_pairs_are_first_candidates (firstPass ps nu fi) ps*
⟹ pairsOf (firstPass ps nu fi) ⊆ lhSet ps × leftmostLookaheads ps ∪ pairsOf fi"
⟨*proof*⟩

lemma *finite_leftmostLookaheads: "finite (leftmostLookaheads ps)"*
⟨*proof*⟩

lemma *firstPass_not_equiv_candidates_lt: "all_pairs_are_first_candidates (firstPass ps nu fi) ps*
⟹ fi ≠ (firstPass ps nu fi)
⟹ countFirstCands ps (firstPass ps nu fi) < countFirstCands ps fi"
⟨*proof*⟩

lemma *firstPass_preserves_apac': "all_pairs_are_first_candidates fi (ps1 @ ps2)*
⟹ all_pairs_are_first_candidates (firstPass ps2 nu fi) (ps1 @ ps2)"
⟨*proof*⟩

lemma *firstPass_preserves_apac:*
"all_pairs_are_first_candidates fi ps ⟹ all_pairs_are_first_candidates (firstPass ps nu fi) ps"
⟨*proof*⟩

Termination proof for *mkFirstMap'* given that *all_pairs_are_first_candidates* holds for the first call and therefore also for every following iteration.

lemma *mkFirstMap'_dom_if_apac:*
"mkFirstMap' ps nu fi ≠ None" **if** *"all_pairs_are_first_candidates fi ps"*
⟨*proof*⟩

lemma *empty_fi_apac: "all_pairs_are_first_candidates fmempty ps"*
⟨*proof*⟩

lemma *mkFirstMap_simp: "mkFirstMap g nu ≡ (let fi' = firstPass (prods g) nu fmempty in*
(if fmempty = fi' then fmempty else the (mkFirstMap' (prods g) nu fi')))"
⟨*proof*⟩

4.2 Correctness Definitions

definition *first_map_sound :: "('n, 't) first_map ⇒ ('n, 't) grammar ⇒ bool"* **where**
"first_map_sound fi g =
(∀x la xFirst. fmlookup fi x = Some xFirst ∧ la ∈ set xFirst ⟶ first_sym g la (NT x))"

definition *first_map_complete :: "('n, 't) first_map ⇒ ('n, 't) grammar ⇒ bool"* **where**
"first_map_complete fi g = (∀la s x. first_sym g la s
∧ s = (NT x) ⟶ (∃xFirst. fmlookup fi x = Some xFirst ∧ la ∈ set xFirst))"

abbreviation *first_map_for :: "('n, 't) first_map ⇒ ('n, 't) grammar ⇒ bool"* **where**
"first_map_for fi g ≡ first_map_sound fi g ∧ first_map_complete fi g"

4.3 Soundness

lemma *firstSym_first_sym:* **assumes** *"first_map_sound fi g"* **and** *"la ∈ set (firstSym s fi)"*
shows *"first_sym g la s"*
⟨*proof*⟩

lemma *nullable_app: "nullable_gamma g xs ⟹ nullable_gamma g ys ⟹ nullable_gamma g (xs @ ys)"*

⟨*proof*⟩

lemma *nullableSym_nullable_sym:* **assumes** *"nullable_set_for nu g"*
shows *"nullableSym s nu ⟷ nullable_sym g s"*
⟨*proof*⟩

lemma *firstGamma_first_sym': "nullable_set_for nu g ⟹ first_map_sound fi g*
⟹ (x, gpre @ gsuf) ∈ set (prods g) ⟹ nullable_gamma g gpre
⟹ la ∈ set (firstGamma gsuf nu fi) ⟹ first_sym g la (NT x)"
⟨*proof*⟩

lemma *firstGamma_first_sym: "nullable_set_for nu g ⟹ first_map_sound fi g*
⟹ (x, gamma) ∈ set (prods g) ⟹ la ∈ set (firstGamma gamma nu fi) ⟹ first_sym g la (NT x)"
⟨*proof*⟩

lemma *firstPass_preserves_soundness': "nullable_set_for nu g ⟹ first_map_sound fi g*
⟹ set ps ⊆ set (prods g) ⟹ first_map_sound (firstPass ps nu fi) g"
⟨*proof*⟩

lemma *firstPass_preserves_soundness: "nullable_set_for nu g ⟹ first_map_sound fi g*
⟹ first_map_sound (firstPass (prods g) nu fi) g"
⟨*proof*⟩

lemma *mkFirstMap'_preserves_soundness: "nullable_set_for nu g ⟹ first_map_sound fi g*
⟹ all_pairs_are_first_candidates fi (prods g)
⟹ first_map_sound (the (mkFirstMap' (prods g) nu fi)) g"
⟨*proof*⟩

lemma *empty_fi_sound: "first_map_sound fmempty g"*
⟨*proof*⟩

theorem *mkFirstMap_sound: "nullable_set_for nu g ⟹ first_map_sound (mkFirstMap g nu) g"*
⟨*proof*⟩

4.4 Completeness

lemma *la_in_firstGamma_t: "nullable_set_for nu g ⟹ nullable_gamma g gpre*
⟹ LA y ∈ set (firstGamma (gpre @ T y # gsuf) nu fi)"
⟨*proof*⟩

lemma *la_in_firstGamma_nt: "nullable_set_for nu g ⟹ nullable_gamma g gpre*
⟹ fmlookup fi x = Some xFirst ⟹ la ∈ set xFirst
⟹ la ∈ set (firstGamma (gpre @ NT x # gsuf) nu fi)"
⟨*proof*⟩

lemma *firstPass_preserves_key_value_subset: "fmlookup fi x = Some xFirst*
⟹ ∃xFirst'. fmlookup (firstPass ps nu fi) x = Some xFirst' ∧ set xFirst ⊆ set xFirst'"
⟨*proof*⟩

lemma *firstPass_equiv_cons_tl:* **assumes** *"fi = firstPass (p # ps) nu fi"*
shows *"fi = firstPass ps nu fi"*
⟨*proof*⟩

lemma *firstPass_equiv_right_t': "(lx, gpre @ (T y) # gsuf) ∈ set psuf*
⟹ nullable_set_for nu g
⟹ nullable_gamma g gpre ⟹ fi = firstPass psuf nu fi ⟹ prods g = ppre @ psuf
⟹ (∃lxFirst. fmlookup fi lx = Some lxFirst ∧ (LA y) ∈ set lxFirst)"
⟨*proof*⟩

lemma *firstPass_equiv_right_t: "(lx, gpre @ (T y) # gsuf) ∈ set (prods g) ⟹ nullable_set_for nu g*
⟹ nullable_gamma g gpre ⟹ fi = firstPass (prods g) nu fi
⟹ ∃lxFirst. fmlookup fi lx = Some lxFirst ∧ LA y ∈ set lxFirst"
⟨*proof*⟩

lemma *firstPass_equiv_right_nt': "nullable_set_for nu g ⟹ fi = firstPass psuf nu fi*
⟹ (lx, gpre @ (NT rx) # gsuf) ∈ set psuf ⟹ nullable_gamma g gpre
⟹ fmlookup fi rx = Some rxFirst ⟹ la ∈ set rxFirst ⟹ ppre @ psuf = (prods g)
⟹ ∃lxFirst. fmlookup fi lx = Some lxFirst ∧ la ∈ set lxFirst"
⟨*proof*⟩

lemma *firstPass_equiv_right_nt: "nullable_set_for nu g ⟹ fi = firstPass (prods g) nu fi*
⟹ (lx, gpre @ (NT rx) # gsuf) ∈ set (prods g) ⟹ nullable_gamma g gpre
⟹ fmlookup fi rx = Some rxFirst ⟹ la ∈ set rxFirst
⟹ ∃lxFirst. fmlookup fi lx = Some lxFirst ∧ la ∈ set lxFirst"
⟨*proof*⟩

lemma *firstPass_equiv_complete:* **assumes** *"nullable_set_for nu g" "fi = firstPass (prods g) nu fi"*
shows *"first_map_complete fi g"*
⟨*proof*⟩

lemma *mkFirstMap'_complete: "nullable_set_for nu g ⟹ all_pairs_are_first_candidates fi (prods g)*
⟹ first_map_complete (the (mkFirstMap' (prods g) nu fi)) g"
⟨*proof*⟩

theorem *mkFirstMap_complete: "nullable_set_for nu g ⟹ first_map_complete (mkFirstMap g nu) g"*
⟨*proof*⟩

theorem *mkFirstMap_correct: "nullable_set_for nu g ⟹ first_map_for (mkFirstMap g nu) g"*
⟨*proof*⟩

declare *mkFirstMap'.simps[code]*

end

5 Follow map

theory *Follow_Map*
imports *First_Map*
begin

type_synonym *('n, 't) follow_map = "('n, 't lookahead list) fmap"*

fun *updateFo :: "'n set ⇒ ('n, 't) first_map ⇒ 'n ⇒ ('n, 't) rhs ⇒ ('n, 't) follow_map*
⇒ ('n, 't) follow_map" **where**
"updateFo nu fi lx [] fo = fo"
| "updateFo nu fi lx ((T _) # gamma') fo = updateFo nu fi lx gamma' fo"
| "updateFo nu fi lx ((NT rx) # gamma') fo = (let fo' = updateFo nu fi lx gamma' fo;
lSet = findOrEmpty lx fo';
rSet = firstGamma gamma' nu fi;
additions = (if nullableGamma gamma' nu then lSet @@ rSet else rSet)
in (case fmlookup fo' rx of
None ⇒ (if additions = [] then fo' else fmupd rx additions fo')
| Some rxFollow ⇒ (if set additions ⊆ set rxFollow then fo'
else fmupd rx (rxFollow @@ additions) fo')))"

definition *followPass :: "('n, 't) prods ⇒ 'n set ⇒ ('n, 't) first_map ⇒ ('n, 't) follow_map*
⇒ ('n, 't) follow_map" **where**
"followPass ps nu fi fo = foldr (λ(x, gamma) fo. updateFo nu fi x gamma fo) ps fo"

partial_function *(option) mkFollowMap' :: "('n, 't) grammar ⇒ 'n set ⇒ ('n, 't) first_map*
⇒ ('n, 't) follow_map ⇒ ('n, 't) follow_map option" **where**

```
  "mkFollowMap' g nu fi fo = (let fo' = followPass (prods g) nu fi fo
in
    (if fo = fo' then Some fo else mkFollowMap' g nu fi fo'))"
```

abbreviation *initial_fo :: "('n, 't) grammar ⇒ ('n, 't) follow_map"* **where**
"initial_fo g ≡ fmupd (start g) [EOF] fmempty"

definition *mkFollowMap :: "('n, 't) grammar ⇒ 'n set ⇒ ('n, 't) first_map ⇒ ('n, 't) follow_map"* **where**
"mkFollowMap g nu fi = the (mkFollowMap' g nu fi (initial_fo g))"

5.1 Termination

fun *ntsOfGamma :: "('n, 't) rhs ⇒ 'n set"* **where**
```
  "ntsOfGamma [] = {}"
| "ntsOfGamma ((T _)#gamma') = ntsOfGamma gamma'"
| "ntsOfGamma ((NT x)#gamma') = insert x (ntsOfGamma gamma')"
```

definition *ntsOf :: "('n, 't) grammar ⇒ 'n set"* **where**
"ntsOf g = {start g} ∪ fst ‘ set (prods g) ∪ ⋃(ntsOfGamma ‘ snd ‘ set (prods g))"

fun *lookaheadsOfGamma :: "('n, 't) rhs ⇒ 't lookahead set"* **where**
```
  "lookaheadsOfGamma [] = {}"
| "lookaheadsOfGamma ((T x)#gamma') = insert (LA x) (lookaheadsOfGamma
gamma') "
| "lookaheadsOfGamma ((NT _)#gamma') = lookaheadsOfGamma gamma'"
```

definition *lookaheadsOf :: "('n, 't) grammar ⇒ 't lookahead set"* **where**
"lookaheadsOf g = {EOF} ∪ ⋃(lookaheadsOfGamma ‘ snd ‘ set (prods g))"

definition *all_pairs_are_follow_candidates ::*
"('n, 't) follow_map ⇒ ('n, 't) grammar ⇒ bool" **where**
```
  "all_pairs_are_follow_candidates fo g =
  (∀(x, la) ∈ pairsOf fo. x ∈ ntsOf g ∧ la ∈ lookaheadsOf g)"
```

definition *countFollowCands :: "('n, 't) grammar ⇒ ('n, 't) follow_map ⇒ nat"* **where**
```
  "countFollowCands g fo =
  (let allCandidates = (ntsOf g) × (lookaheadsOf g) in card (allCandidates
- (pairsOf fo)))"
```

lemma *followPass_cons[simp]:*
```
  "followPass ((x, gamma) # ps) nu fi fo = updateFo nu fi x gamma (followPass
ps nu fi fo)"
```
⟨*proof*⟩

lemma *medial_t_in_lookaheadsOf:*

```
  "(x, gpre @ (T y) # gsuf) ∈ set (prods g) ⟹ (LA y) ∈ lookaheadsOf
g"
```
⟨*proof*⟩

lemma *first_sym_in_lookaheadsOf: "first_sym g la s ⟹ s = NT x ⟹*
la ∈ lookaheadsOf g"
⟨*proof*⟩

lemma *first_map_la_in_lookaheadsOf:*
"first_map_for fi g ⟹ fmlookup fi x = Some s ⟹ la ∈ set s ⟹ la
∈ lookaheadsOf g"
⟨*proof*⟩

lemma *in_firstGamma_in_lookaheadsOf:*
"first_map_for fi g ⟹ (x, gpre @ gsuf) ∈ set (prods g) ⟹ la ∈ set
(firstGamma gsuf nu fi)
⟹ la ∈ lookaheadsOf g"
⟨*proof*⟩

lemma *la_in_fo_in_lookaheadsOf: "fmlookup fo x = Some xFollow ⟹ la*
∈ set xFollow
⟹ all_pairs_are_follow_candidates fo g ⟹ la ∈ lookaheadsOf g"
⟨*proof*⟩

lemma *medial_nt_in_ntsOfGamma: "x ∈ ntsOfGamma (gpre @ (NT x) # gsuf)"*
⟨*proof*⟩

lemma *medial_nt_in_ntsOf: "(lx, gpre @ (NT rx) # gsuf) ∈ set (prods*
g) ⟹ rx ∈ (ntsOf g)"
⟨*proof*⟩

lemma *updateFo_induct_refined:*
fixes *nu :: "'n set"*
and *lx :: "'n"*
and *gamma' :: "('n, 't) symbol list"*
and *fi :: "('n, 't) first_map"*
and *fo :: "('n, 't) follow_map"*
and *P :: "'n set ⇒ ('n, 't) first_map ⇒ 'n ⇒ ('n, 't) symbol list*
⇒ ('n, 't) follow_map
⇒ bool"
defines *"additions ≡ (λnu fi lx gamma' fo. (if nullableGamma gamma'*
nu
then findOrEmpty lx (updateFo nu fi lx gamma' fo) @@ (firstGamma
gamma' nu fi)
else firstGamma gamma' nu fi))"
assumes *Nil: "(⋀nu fi lx fo. P nu fi lx [] fo)"*
and *T: "(⋀nu fi lx y gamma' fo. P nu fi lx gamma' fo ⟹ P nu fi*
lx (T y # gamma') fo)"
and *NT_None_same: "(⋀nu fi lx rx gamma' fo. P nu fi lx gamma' fo*

```
      ⟹ fmlookup (updateFo nu fi lx gamma' fo) rx = None ⟹ additions
nu fi lx gamma' fo = []
      ⟹ P nu fi lx (NT rx # gamma') fo)"
    and NT_None_new: "(⋀nu fi lx rx gamma' fo. P nu fi lx gamma' fo
      ⟹ fmlookup (updateFo nu fi lx gamma' fo) rx = None ⟹ additions
nu fi lx gamma' fo ≠ []
      ⟹ P nu fi lx (NT rx # gamma') fo)"
    and NT_Some_same: "(⋀nu fi lx rx gamma' fo rxFollow. P nu fi lx gamma'
fo
      ⟹ fmlookup (updateFo nu fi lx gamma' fo) rx = Some rxFollow
      ⟹ set (additions nu fi lx gamma' fo) ⊆ set rxFollow ⟹ P nu
fi lx (NT rx # gamma') fo)"
    and NT_Some_new: "(⋀nu fi lx rx gamma' fo rxFollow. P nu fi lx gamma'
fo
      ⟹ fmlookup (updateFo nu fi lx gamma' fo) rx = Some rxFollow
      ⟹ ¬ set (additions nu fi lx gamma' fo) ⊆ set rxFollow ⟹ P nu
fi lx (NT rx # gamma') fo)"
  shows "P (nu::'n set) fi lx (gamma :: ('n, 't) symbol list) fo"
  ⟨proof⟩

lemma updateFo_preserves_apac_fmupd_additions: assumes "first_map_for
fi g"
  and "all_pairs_are_follow_candidates (updateFo nu fi lx gamma' fo) g"
  and "(lx, gpre @ NT rx # gamma') ∈ set (prods g)"
  and "la ∈ set (if nullableGamma gamma' nu then (findOrEmpty lx (updateFo
nu fi lx gamma' fo))
    @@ (firstGamma gamma' nu fi) else firstGamma gamma' nu fi)"
  shows "rx ∈ ntsOf g ∧ la ∈ lookaheadsOf g"
⟨proof⟩

lemma updateFo_preserves_apac:
  "first_map_for fi g ⟹ (lx, gpre @ gsuf) ∈ set (prods g)
  ⟹ all_pairs_are_follow_candidates fo g
  ⟹ all_pairs_are_follow_candidates (updateFo nu fi lx gsuf fo) g"
⟨proof⟩

lemma followPass_preserves_apac': "first_map_for fi g ⟹ pre @ suf
= (prods g)
  ⟹ all_pairs_are_follow_candidates fo g
  ⟹ all_pairs_are_follow_candidates (followPass suf nu fi fo) g"
⟨proof⟩

lemma followPass_preserves_apac: "first_map_for fi g ⟹ all_pairs_are_follow_candidates
fo g
   ⟹ all_pairs_are_follow_candidates (followPass (prods g) nu fi fo)
g"
  ⟨proof⟩

lemma updateFo_subset: "pairsOf fo ⊆ pairsOf (updateFo nu fi x' gamma
```

```
fo)"
```
⟨*proof*⟩

lemma *followPass_subset: "pairsOf fo ⊆ pairsOf (followPass ps nu fi fo)"*
⟨*proof*⟩

lemma *updateFo_not_equiv_exists': "first_map_for fi g ⟹(lx, gpre @ gsuf) ∈ set (prods g)*
⟹ all_pairs_are_follow_candidates fo g
⟹ fo ≠ (updateFo nu fi lx gsuf fo)
⟹ ∃x' la. x' ∈ ntsOf g ∧ la ∈ lookaheadsOf g ∧ (x', la) ∉ pairsOf fo
∧ (x', la) ∈ pairsOf (updateFo nu fi lx gsuf fo)"
⟨*proof*⟩

lemma *updateFo_not_equiv_exists: "first_map_for fi g ⟹(lx, gamma) ∈ set (prods g)*
⟹ all_pairs_are_follow_candidates fo g
⟹ fo ≠ (updateFo nu fi lx gamma fo)
⟹ ∃x' la. x' ∈ ntsOf g ∧ la ∈ lookaheadsOf g ∧ (x', la) ∉ pairsOf fo
∧ (x', la) ∈ pairsOf (updateFo nu fi lx gamma fo)"
⟨*proof*⟩

lemma *followPass_equiv_or_exists': "first_map_for fi g ⟹ all_pairs_are_follow_candidates fo g*
⟹ pre @ suf = (prods g) ⟹ fo ≠ (followPass suf nu fi fo)
⟹ (∃x la. x ∈ (ntsOf g) ∧ la ∈ (lookaheadsOf g) ∧ (x, la) ∉ (pairsOf fo)
∧ (x, la) ∈ (pairsOf (followPass suf nu fi fo)))"
⟨*proof*⟩

lemma *followPass_not_equiv_exists: "first_map_for fi g ⟹ all_pairs_are_follow_candidates fo g*
⟹ fo ≠ followPass (prods g) nu fi fo ⟹ ∃x la. x ∈ ntsOf g ∧ la ∈ lookaheadsOf g
∧ (x, la) ∉ pairsOf fo ∧ (x, la) ∈ pairsOf (followPass (prods g) nu fi fo)"
⟨*proof*⟩

lemma *finite_ntsOfGamma: "finite (ntsOfGamma gamma)"*
⟨*proof*⟩

lemma *finite_ntsOf: "finite (ntsOf g)"*
⟨*proof*⟩

lemma *finite_lookaheadsOfGamma: "finite (lookaheadsOfGamma gamma)"*
⟨*proof*⟩

lemma *finite_lookaheadsOf: "finite (lookaheadsOf g)"*
⟨*proof*⟩

lemma *finite_allCandidates_follow: "finite (ntsOf g × lookaheadsOf g)"*
⟨*proof*⟩

lemma *followPass_not_equiv_candidates_lt:*
"first_map_for fi g ⟹ all_pairs_are_follow_candidates fo g
⟹ fo ≠ (followPass (prods g) nu fi fo)
⟹ countFollowCands g (followPass (prods g) nu fi fo) < countFollowCands g fo"
⟨*proof*⟩

Termination proof for mkFollowMap' with the assumption that fi is a correct first map, and all_pairs_are_follow_candidates holds in the beginning and thus for every other iteration

lemma *mkFollowMap'_dom_if_apac: "mkFollowMap' g nu fi fo ≠ None"*
if *"first_map_for fi g"* **and** *"all_pairs_are_follow_candidates fo g"*
⟨*proof*⟩

lemma *initial_fo_apac: "all_pairs_are_follow_candidates (initial_fo g) g"*
⟨*proof*⟩

5.2 Correctness Definitions

definition *follow_map_sound :: "('n, 't) follow_map ⇒ ('n, 't) grammar ⇒ bool"* **where**
"follow_map_sound fo g =
(∀x la xFollow. fmlookup fo x = Some xFollow ∧ la ∈ set xFollow ⟶ follow_sym g la (NT x))"

definition *follow_map_complete :: "('n, 't) follow_map ⇒ ('n, 't) grammar ⇒ bool"* **where**
"follow_map_complete fo g = (∀la s x. follow_sym g la s ∧ s = NT x
⟶ (∃xFollow. fmlookup fo x = Some xFollow ∧ la ∈ set xFollow))"

abbreviation *follow_map_for :: "('n, 't) follow_map ⇒ ('n, 't) grammar ⇒ bool"* **where**
"follow_map_for fo g ≡ follow_map_sound fo g ∧ follow_map_complete fo g"

5.3 Soundness

lemma *first_gamma_tail_cons: "nullable_sym g s ⟹ nullable_gamma g gpre ⟹ first_gamma g la gsuf*
⟹ first_gamma g la (gpre @ s # gsuf)"
⟨*proof*⟩

lemma *firstGamma_first_gamma: "nullable_set_for nu g ⟹ first_map_for fi g*
⟹ la ∈ set (firstGamma gamma nu fi) ⟹ first_gamma g la gamma"
⟨*proof*⟩

lemma *first_gamma_firstGamma: "nullable_set_for nu g ⟹ first_map_for fi g*
⟹ first_gamma g la gamma ⟹ la ∈ set (firstGamma gamma nu fi)"
⟨*proof*⟩

lemma *updateFo_preserves_soundness':*
"nullable_set_for nu g ⟹ first_map_for fi g ⟹ (lx, gpre @ gsuf) ∈ set (prods g)
⟹ follow_map_sound fo g ⟹ follow_map_sound (updateFo nu fi lx gsuf fo) g"
⟨*proof*⟩

lemma *updateFo_preserves_soundness: "nullable_set_for nu g ⟹ first_map_for fi g*
⟹ (lx, gamma) ∈ set (prods g) ⟹ follow_map_sound fo g
⟹ follow_map_sound (updateFo nu fi lx gamma fo) g"
⟨*proof*⟩

lemma *followPass_preserves_soundness': "nullable_set_for nu g ⟹ first_map_for fi g*
⟹ follow_map_sound fo g ⟹ pre @ suf = prods g
⟹ follow_map_sound (followPass suf nu fi fo) g"
⟨*proof*⟩

lemma *followPass_preserves_soundness: "nullable_set_for nu g ⟹ first_map_for fi g*
⟹ follow_map_sound fo g ⟹ follow_map_sound (followPass (prods g) nu fi fo) g"
⟨*proof*⟩

lemma *mkFollowMap'_preserves_soundness: "nullable_set_for nu g ⟹ first_map_for fi g*
⟹ follow_map_sound fo g ⟹ all_pairs_are_follow_candidates fo g
⟹ follow_map_sound (the (mkFollowMap' g nu fi fo)) g"
⟨*proof*⟩

lemma *initial_fo_sound: "follow_map_sound (initial_fo g) g"*
⟨*proof*⟩

theorem *mkFollowMap_sound:*
"nullable_set_for nu g ⟹ first_map_for fi g ⟹ follow_map_sound (mkFollowMap g nu fi) g"
⟨*proof*⟩

5.4 Completeness

lemma *updateFo_preserves_map_keys: "x |∈| fmdom fo ⟹ x |∈| fmdom (updateFo nu fi lx gamma fo)"*
⟨*proof*⟩

lemma *followPass_preserves_map_keys: "x |∈| fmdom fo ⟹ x |∈| fmdom (followPass ps nu fi fo)"*
⟨*proof*⟩

lemma *find_updateFo_cons_neq: "x ≠ x' ⟹ fmlookup (updateFo nu fi lx gsuf fo) x = Some xFollow*
⟷ fmlookup (updateFo nu fi lx (NT x' # gsuf) fo) x = Some xFollow"
⟨*proof*⟩

lemma *updateFo_value_subset:*
"fmlookup fo x = Some s1 ⟹ fmlookup (updateFo nu fi lx gamma fo) x = Some s2
⟹ set s1 ⊆ set s2"
⟨*proof*⟩

lemma *updateFo_only_appends:*
"fmlookup fo x = Some s1 ⟹ fmlookup (updateFo nu fi lx gamma fo) x = Some s2
⟹ ∃suf. s2 = s1 @ suf"
⟨*proof*⟩

lemma *followPass_value_subset:*
"fmlookup fo x = Some s1 ⟹ fmlookup (followPass ps nu fi fo) x = Some s2 ⟹ set s1 ⊆ set s2"
⟨*proof*⟩

lemma *followPass_only_appends: "fmlookup fo x = Some s1*
⟹ fmlookup (followPass ps nu fi fo) x = Some s2 ⟹ ∃suf. s2 = s1 @ suf"
⟨*proof*⟩

lemma *followPass_equiv_cons_tl: "fo = followPass ((x, gamma) # ps) nu fi fo*
⟹ fo = followPass ps nu fi fo"
⟨*proof*⟩

lemma *exists_follow_set_Cons:*
assumes *"nullable_set_for nu g" "first_map_for fi g"*
and *"∃rxFollow. fmlookup (updateFo nu fi lx gamma fo) rx = Some rxFollow*
∧ la ∈ set rxFollow"
shows *"∃rxFollow. fmlookup (updateFo nu fi lx (s # gamma) fo) rx = Some rxFollow*
∧ la ∈ set rxFollow"

⟨*proof*⟩

lemma *exists_follow_set_containing_first_gamma:*
"nullable_set_for nu g ⟹ first_map_for fi g ⟹ first_gamma g la gsuf
⟹ (∃rxFollow. fmlookup (updateFo nu fi lx (gpre @ NT rx # gsuf) fo) rx = Some rxFollow
∧ la ∈ set rxFollow)"
⟨*proof*⟩

lemma *followPass_equiv_right: "nullable_set_for nu g ⟹ first_map_for fi g*
⟹ fo = followPass psuf nu fi fo ⟹ (lx, gpre @ NT rx # gsuf) ∈ set psuf
⟹ first_gamma g la gsuf ⟹ ppre @ psuf = prods g
⟹ (∃rxFollow. fmlookup fo rx = Some rxFollow ∧ la ∈ set rxFollow)"
⟨*proof*⟩

lemma *nullable_gamma_nullableGamma:*
"nullable_set_for nu g ⟹ nullable_gamma g gamma ⟹ nullableGamma gamma nu"
⟨*proof*⟩

lemma *updateFo_preserves_membership_in_value:*
"fmlookup fo x = Some s ⟹ la ∈ set s ⟹ la ∈ set (findOrEmpty x (updateFo nu fi x' gamma fo))"
⟨*proof*⟩

lemma *exists_follow_set_containing_follow_left: "nullable_set_for nu g ⟹ first_map_for fi g*
⟹ nullable_gamma g gsuf ⟹ fmlookup fo lx = Some lxFollow ⟹ la ∈ set lxFollow
⟹ (∃rxFollow. fmlookup (updateFo nu fi lx (gpre @ NT rx # gsuf) fo) rx = Some rxFollow
∧ la ∈ set rxFollow)"
⟨*proof*⟩

lemma *followPass_equiv_left: "nullable_set_for nu g ⟹ first_map_for fi g*
⟹ fo = followPass psuf nu fi fo ⟹ (lx, gpre @ NT rx # gsuf) ∈ set psuf
⟹ ppre @ psuf = prods g ⟹ nullable_gamma g gsuf ⟹ fmlookup fo lx = Some lxFollow
⟹ la ∈ set lxFollow ⟹ (∃rxFollow. fmlookup fo rx = Some rxFollow ∧ la ∈ set rxFollow)"
⟨*proof*⟩

lemma *followPass_equiv_complete: "nullable_set_for nu g ⟹ first_map_for fi g*

```
  ⟹ (start g, EOF) ∈ pairsOf fo ⟹ fo = followPass (prods g) nu fi
fo
  ⟹ follow_map_complete fo g"
```
⟨*proof*⟩

lemma *mkFollowMap'_complete: "(start g, EOF) ∈ pairsOf fo ⟹ nullable_set_for nu g*
* ⟹ first_map_for fi g ⟹ all_pairs_are_follow_candidates fo g*
* ⟹ follow_map_complete (the (mkFollowMap' g nu fi fo)) g"*
⟨*proof*⟩

lemma *start_eof_in_initial_fo: "(start g, EOF) ∈ pairsOf (initial_fo g)"*
⟨*proof*⟩

theorem *mkFollowMap_complete:*
* "nullable_set_for nu g ⟹ first_map_for fi g ⟹ follow_map_complete (mkFollowMap g nu fi) g"*
⟨*proof*⟩

theorem *mkFollowMap_correct:*
* "nullable_set_for nu g ⟹ first_map_for fi g ⟹ follow_map_for (mkFollowMap g nu fi) g"*
⟨*proof*⟩

declare *mkFollowMap'.simps[code]*

end

6 Parse Table

theory *Parse_Table*
imports *Follow_Map*
begin

From the (correct) NULLABLE, FIRST, and FOLLOW sets we build a list of parse table entries.

type_synonym *('n, 't) table_key = "('n × 't lookahead)"*

type_synonym *('n, 't) parse_table = "(('n × 't lookahead), ('n, 't) prod) fmap"*

definition *firstKeysForProd ::*
* "('n, 't) prod ⇒ 'n set ⇒ ('n, 't) first_map ⇒ ('n, 't) table_key list"* **where**
* "firstKeysForProd ≡ (λ(x, gamma) nu fi. map (λla. (x, la)) (firstGamma gamma nu fi))"*

```
definition followKeysForProd :: "('n, 't) prod ⇒ 'n set ⇒ ('n, 't) first_map
    ⇒ ('n, 't) follow_map ⇒ ('n, 't) table_key list" where
  "followKeysForProd ≡ (λ(x, gamma) nu fi fo.
    map (λla. (x, la)) (if nullableGamma gamma nu then findOrEmpty x fo
else []))"

abbreviation keysForProd :: "'n set ⇒ ('n, 't) first_map ⇒ ('n, 't)
follow_map ⇒ ('n, 't) prod
    ⇒ ('n, 't) table_key list" where
  "keysForProd nu fi fo xp ≡ (firstKeysForProd xp nu fi) @ (followKeysForProd
xp nu fi fo)"

datatype ('n, 't) ll1_parse_table = PT "('n, 't) parse_table"
  | ERROR_GRAMMAR_NOT_LL1_AMB_LA "'t lookahead × ('n, 't) prod × ('n,
't) prod"

fun addEntries :: "('n × 't lookahead) list ⇒ ('n, 't) prod ⇒ ('n,
't) ll1_parse_table
  ⇒ ('n, 't) ll1_parse_table" where
  "addEntries (k # keys) xp (PT pt) = (case fmlookup pt k of
      None ⇒ addEntries keys xp (PT (fmupd k xp pt))
    | Some xp' ⇒ (if xp = xp' then addEntries keys xp (PT pt)
        else ERROR_GRAMMAR_NOT_LL1_AMB_LA (snd k, xp, xp')))"
| "addEntries keys xp pt = pt"

fun mkParseTable' :: "('n, 't) prods ⇒ 'n set ⇒ ('n, 't) first_map ⇒
('n, 't) follow_map
  ⇒ ('n, 't) ll1_parse_table ⇒ ('n, 't) ll1_parse_table" where
  "mkParseTable' [] nu fi fo pt = pt"
| "mkParseTable' (p # ps) nu fi fo pt = (let las = keysForProd nu fi fo
p in
    mkParseTable' ps nu fi fo (addEntries las p pt))"

definition mkParseTable :: "('n, 't) grammar ⇒ ('n, 't) ll1_parse_table"
where
  "mkParseTable g = (let
    nu = mkNullableSet g;
    fi = mkFirstMap g nu;
    fo = mkFollowMap g nu fi
  in mkParseTable' (prods g) nu fi fo (PT fmempty))"
```

6.1 Correctness Definitions

```
definition pt_sound :: "('n, 't) parse_table ⇒ ('n, 't) grammar ⇒ bool"
where
  "pt_sound pt g ≡ (∀ x x' la gamma. fmlookup pt (x', la) = Some (x, gamma)
  ⟶ x' = x ∧ (x, gamma) ∈ set (prods g) ∧ lookahead_for la x gamma
g)"
```

definition *pt_complete :: "('n, 't) parse_table ⇒ ('n, 't) grammar ⇒ bool"* **where**
"pt_complete pt g ≡ (∀ x la gamma. (x, gamma) ∈ set (prods g) ∧ lookahead_for la x gamma g ⟶ fmlookup pt (x, la) = Some (x, gamma))"

abbreviation *parse_table_correct :: "('n, 't) parse_table ⇒ ('n, 't) grammar ⇒ bool"* **where**
"parse_table_correct pt g ≡ pt_sound pt g ∧ pt_complete pt g"

6.2 Soundness

lemma *firstKeysForProd_lookaheads:*
assumes *"nullable_set_for nu g" "first_map_for fi g" "follow_map_for fo g"*
"(x, la) ∈ set (firstKeysForProd (y, gamma) nu fi)"
shows *"x = y ∧ lookahead_for la x gamma g"*
⟨*proof*⟩

lemma *followKeysForProd_lookaheads:*
assumes *"nullable_set_for nu g" "first_map_for fi g" "follow_map_for fo g"*
"(x, la) ∈ set (followKeysForProd (y, gamma) nu fi fo)"
shows *"x = y ∧ lookahead_for la x gamma g"*
⟨*proof*⟩

lemma *keys_are_lookaheads:*
assumes *"nullable_set_for nu g" "first_map_for fi g" "follow_map_for fo g"*
"(x, la) ∈ set (keysForProd nu fi fo (y, gamma))"
shows *"x = y ∧ lookahead_for la y gamma g"*
⟨*proof*⟩

lemma *findOrEmpty_sset_laOf_fi: "first_map_for fi g ⟹ set (findOrEmpty x fi) ⊆ lookaheadsOf g"*
⟨*proof*⟩

lemma *follow_sym_in_lookaheadsOf:*
"follow_sym g la (NT x) ⟹ la ∈ lookaheadsOf g"
⟨*proof*⟩

lemma *follow_map_la_in_lookaheadsOf:*
"follow_map_for fo g ⟹ fmlookup fo x = Some s ⟹ la ∈ set s ⟹ la ∈ lookaheadsOf g"
⟨*proof*⟩

lemma *findOrEmpty_sset_laOf_fo: "follow_map_for fo g ⟹ set (findOrEmpty x fo) ⊆ lookaheadsOf g"*
⟨*proof*⟩

lemma *addEntries_preserves_soundness:*
assumes *"nullable_set_for nu g" "first_map_for fi g" "follow_map_for fo g" "p ∈ set (prods g)"*
shows *"pt_sound pt g ⟹ set las ⊆ set (keysForProd nu fi fo p) ⟹ addEntries las p (PT pt) = PT pt' ⟹ pt_sound pt' g"*
⟨*proof*⟩

lemma *mkParseTable'_nested: "mkParseTable' suf nu fi fo (mkParseTable' pre nu fi fo pt) = mkParseTable' (pre @ suf) nu fi fo pt"*
⟨*proof*⟩

lemma *mkParseTable'_failure_preserved:*
"mkParseTable' pre nu fi fo pt = ERROR_GRAMMAR_NOT_LL1_AMB_LA e ⟹ mkParseTable' (pre @ suf) nu fi fo pt = ERROR_GRAMMAR_NOT_LL1_AMB_LA e"
⟨*proof*⟩

lemma *all_pre_pt_non_failure:*
"mkParseTable' (pre @ suf) nu fi fo (PT pt) = PT pt' ⟹ ∃pre_pt'. mkParseTable' pre nu fi fo (PT pt) = PT pre_pt'"
⟨*proof*⟩

lemma *mkParseTable'_preserves_soundness:*
assumes *"nullable_set_for nu g" "first_map_for fi g" "follow_map_for fo g"*
shows *"set ps ⊆ set (prods g) ⟹ pt_sound pt g ⟹ mkParseTable' ps nu fi fo (PT pt) = PT pt' ⟹ pt_sound pt' g"*
⟨*proof*⟩

lemma *initial_pt_sound: "pt_sound fmempty g"*
⟨*proof*⟩

theorem *mkParseTable_sound: "mkParseTable g = PT pt ⟹ pt_sound pt g"*
⟨*proof*⟩

6.3 Completeness

lemma *la_in_keysForProd:*
assumes *"nullable_set_for nu g" "first_map_for fi g" "follow_map_for fo g" "lookahead_for la x gamma g"*
shows*"(x, la) ∈ set (keysForProd nu fi fo (x, gamma))"*
⟨*proof*⟩

lemma *addEntries_lookup_same_or_none: "addEntries las xp (PT pt) = PT pt' ⟹ fmlookup pt k = Some x*

```
  ⟹ fmlookup pt' k = Some x"
```
⟨*proof*⟩

lemma *mkParseTable'_lookup_same_or_none: "mkParseTable' ps nu fi fo (PT pt) = PT pt'*
```
  ⟹ fmlookup pt k = Some x ⟹ fmlookup pt' k = Some x"
```
⟨*proof*⟩

lemma *addEntries_in_pt:*
```
  "k ∈ set las ⟹ addEntries las xp (PT pt) = PT pt' ⟹ fmlookup pt'
k = Some xp"
```
⟨*proof*⟩

lemma *mkParseTable'_complete': "nullable_set_for nu g ⟹ first_map_for fi g*
```
  ⟹ follow_map_for fo g ⟹ prods g = ppre @ psuf ⟹ (x, gamma) ∈
set psuf
  ⟹ lookahead_for la x gamma g ⟹ mkParseTable' psuf nu fi fo (PT pt)
= PT pt'
  ⟹ fmlookup pt' (x, la) = Some (x, gamma)"
```
⟨*proof*⟩

lemma *mkParseTable'_complete: "nullable_set_for nu g ⟹ first_map_for fi g ⟹ follow_map_for fo g*
```
  ⟹ (x, gamma) ∈ set (prods g) ⟹ lookahead_for la x gamma g
  ⟹ mkParseTable' (prods g) nu fi fo (PT fmempty) = PT pt
  ⟹ fmlookup pt (x, la) = Some (x, gamma)"
```
  ⟨*proof*⟩

theorem *mkParseTable_complete: "mkParseTable g = PT pt ⟹ pt_complete pt g"*
  ⟨*proof*⟩

theorem *mkParseTable_correct: "mkParseTable g = PT pt ⟹ parse_table_correct pt g"*
  ⟨*proof*⟩

end

7 Parser

theory *LL1_Parser*
  imports *Parse_Table*
begin

datatype *('n, 't) parse_tree = Node "'n" "('n, 't) parse_tree list" | Leaf "'t"*

```
datatype ('n, 't, 's) return_type = RESULT "('n, 't × 's) parse_tree"
"('t × 's) list"
  | ERROR "string" "'n" "('t × 's) list"
  | GRAMMAR_NOT_LL1 "string" "'t lookahead"
  | REJECT "string" "('t × 's) list"

fun peek :: "('t × 's) list ⇒ 't lookahead" where
  "peek [] = EOF"
| "peek (t#ts) = LA (fst t)"


locale parse =
  fixes showT :: "'t ⇒ string" and showS :: "'s ⇒ string"
begin

definition mismatchMessage :: "'t ⇒ 't × 's ⇒ string" where
  "mismatchMessage a ≡ λ(a', s).
  ''Token mismatch. Expected '' @ showT a @ '', saw '' @ showT a' @ ''
('' @ showS s @ '')''"

function (domintros) parseSymbol ::
  "('n, 't) parse_table ⇒ ('n, 't) symbol ⇒ ('t × 's) list ⇒ 'n fset
⇒ ('n, 't, 's) return_type"
and
  parseGamma ::
  "('n, 't) parse_table ⇒ 'n ⇒ ('n, 't) symbol list ⇒ ('t × 's) list
⇒ 'n fset
  ⇒ ('n, 't, 's) return_type"
  where
    "parseSymbol _ (T a) [] _ = REJECT ''input exhausted'' []"
  | "parseSymbol pt (T a) (t#ts) vis = (if fst t = a then RESULT (Leaf
t) ts
    else REJECT (mismatchMessage a t) (t#ts))"
  | "parseSymbol pt (NT x) ts vis = (if x |∈| vis then ERROR ''left recursion
detected'' x ts
    else (case fmlookup pt (x, peek ts) of
      None ⇒ REJECT ''lookup failure'' ts
    | Some (x', gamma) ⇒ (if x ≠ x' then ERROR ''malformed parse table''
x ts
        else parseGamma pt x gamma ts (finsert x vis))
    ))"
  | "parseGamma pt n [] ts vis = RESULT (Node n []) ts"
  | "parseGamma pt n (s#gamma') ts vis = (let parse_s = parseSymbol pt
s ts vis in
    (case parse_s of
      RESULT t r ⇒
        (let parse_g = parseGamma pt n gamma' r (if length r < length
ts then {|||} else vis) in
```

```
          (case parse_g of
            RESULT (Node n tls) r' ⇒ RESULT (Node n (t # tls)) r'
          | e ⇒ e))
    | e ⇒ e))"
```
⟨*proof*⟩

definition *nt_from_pt :: "('n, 't) parse_table ⇒ 'n fset"* **where**
"nt_from_pt pt = fst |`| fmdom pt"

definition *parse_ind_meas_sym ::*
"('n, 't) parse_table ⇒ ('n, 't) symbol ⇒ ('t × 's) list ⇒ 'n fset ⇒ nat × nat × nat" **where**
"parse_ind_meas_sym pt s ts vis = (length ts, fcard (nt_from_pt pt |-| vis), 0)"

definition *parse_ind_meas_sym_list ::*
"('n, 't) parse_table ⇒ ('n, 't) symbol list ⇒ ('t × 's) list ⇒ 'n fset ⇒ nat × nat × nat"
where
"parse_ind_meas_sym_list pt ss ts vis = (length ts, fcard (nt_from_pt pt |-| vis), length ss + 1)"

definition *parse_ind_meas :: "('n, 't) parse_table ⇒ ('n, 't) symbol + ('n, 't) symbol list ⇒*
('t × 's) list ⇒ 'n fset ⇒ nat × nat × nat" **where**
"parse_ind_meas pt ss ts vis = (length ts, fcard (nt_from_pt pt |-| vis),
(case ss of Inl ss' ⇒ 0 | Inr ss' ⇒ length ss' + 1))"

definition *lex_triple ::*
"('a × 'a) set ⇒ ('b × 'b) set ⇒ ('c × 'c) set ⇒ (('a × 'b × 'c) × ('a × 'b × 'c)) set"
where *"lex_triple ra rb rc = ra <*lex*> (rb <*lex*> rc)"*

lemma *in_lex_triple[simp]: "((a, b, c), (a', b', c')) ∈ lex_triple r s t*
⟷ (a, a') ∈ r ∨ a = a' ∧ (b, b') ∈ s ∨ a = a' ∧ b = b' ∧ (c, c') ∈ t"
⟨*proof*⟩

lemma *wf_lex_triple[intro!]:*
assumes *"wf ra" "wf rb" "wf rc"*
shows *"wf (lex_triple ra rb rc)"*
⟨*proof*⟩

definition *mlex_triple :: "('a ⇒ nat × nat × nat) ⇒ ('a × 'a) set"*
where
"mlex_triple f = inv_image (lex_triple less_than less_than less_than) f"

lemma *parseSymbol_length_bound_partial:*
"parseSymbol_parseGamma_dom (Inl (pt, s, ts, vis))
⟹ (⋀tr r. parseSymbol pt s ts vis = RESULT tr r ⟹ length r ≤ length ts)" **and**
parseGamma_length_bound_partial:
"parseSymbol_parseGamma_dom (Inr (pt, n, gamma, ts, vis))
⟹ (⋀tr r. parseGamma pt n gamma ts vis = RESULT tr r ⟹ length r ≤ length ts)"
⟨*proof*⟩

lemma *fcard_diff_insert_less:*
assumes *"x |∉| vis"* *"fmlookup pt (x,peek ts) = Some (x,ss)"*
shows *"fcard (nt_from_pt pt - (finsert x vis)) < fcard (nt_from_pt pt - vis)"*
⟨*proof*⟩

termination
⟨*proof*⟩

fun *parse ::*
"('n, 't) ll1_parse_table ⇒ ('n, 't) symbol ⇒ ('t × 's) list ⇒ ('n, 't, 's) return_type"
where
"parse (PT pt) s ts = parseSymbol pt s ts {|||}"
| "parse (ERROR_GRAMMAR_NOT_LL1_AMB_LA l) s ts =
(case l of (a, p1, p) ⇒ GRAMMAR_NOT_LL1 ''Grammar not LL1, ambigous lookahead '' a)"

fun *concatWithSep :: "string ⇒ string ⇒ string"* **where**
"concatWithSep [] [] = []"
| "concatWithSep [] acc = acc"
| "concatWithSep s [] = s"
| "concatWithSep s (c # acc) = (if c = CHR '' '' then s @ c # acc else s @ '' '' @ c # acc)"

fun *parseTreeToString :: "('n, 't × 's) parse_tree ⇒ string"* **where**
"parseTreeToString (Leaf (a, s)) = ''('' @ showT a @ '', '' @ showS s @ '')''"
| "parseTreeToString (Node n ls) = foldr concatWithSep (map parseTreeToString ls) ''''"

fun *parseToString ::*
"('n, 't) ll1_parse_table ⇒ ('n, 't) symbol ⇒ ('t × 's) list ⇒ string"
where
"parseToString (PT pt) s ts = (case parseSymbol pt s ts {|||} of
RESULT t [] ⇒ parseTreeToString t)"
| "parseToString (ERROR_GRAMMAR_NOT_LL1_AMB_LA l) s ts = ''Grammar not LL1, ambigous lookahead '' @

```
    (case l of (a, p1, p) ⇒ (case a of LA t ⇒ showT t | EOF ⇒ ''EOF''))"
```

7.1 Soundness

inductive *sym_derives_prefix ::*
```
  "('n, 't) grammar ⇒ ('n, 't) symbol ⇒ ('t × 's) list
  ⇒ ('n, 't × 's) parse_tree ⇒ ('t × 's) list ⇒ bool"
  and gamma_derives_prefix :: "('n, 't) grammar ⇒ 'n ⇒ ('n, 't) symbol
list ⇒ ('t × 's) list
    ⇒ ('n, 't × 's) parse_tree ⇒ ('t × 's) list ⇒ bool" for g
where
  T_sdp: "sym_derives_prefix g (T a) [(a, s)] (Leaf (a, s)) r"
| NT_sdp: "(x, gamma) ∈ set (prods g) ⟹ lookahead_for (peek (w @ r))
x gamma g
    ⟹ gamma_derives_prefix g x gamma w t r ⟹ sym_derives_prefix g
(NT x) w t r"
| Nil_gdp: "gamma_derives_prefix g x [] [] (Node x []) r"
| Cons_gdp: "sym_derives_prefix g s wpre v (wsuf @ r)
    ⟹ gamma_derives_prefix g n ss wsuf (Node n vs) r
    ⟹ gamma_derives_prefix g n (s#ss) (wpre @ wsuf) (Node n (v # vs))
r"
```

lemma *parseSymbol_ts_contains_remainder:*
```
  "⋀t r. parseSymbol pt s ts vis = RESULT t r ⟹ ∃ts'. ts' @ r = ts"
```
and
```
  parseGamma_ts_contains_remainder:
  "⋀t r. parseGamma pt n gamma ts vis = RESULT t r ⟹ ∃ts'. ts' @ r
= ts"
```
⟨*proof*⟩

lemma *parse_meas_induct:*
```
  assumes "⋀y. (⋀x. ((x,y) ∈ mlex_triple (λx. case x of
      Inl (pt, s, ts, vis) ⇒ parse_ind_meas_sym pt s ts vis
    | Inr (pt, x, ss, ts, vis) ⇒ parse_ind_meas_sym_list pt ss ts vis)
⟹ P x)) ⟹ P y"
  shows "P z"
```
⟨*proof*⟩

lemma *parseGamma_node: "parseSymbol pt s ts vis = RESULT v r ⟹ True"*
```
  "parseGamma pt x gamma ts vis = RESULT v r ⟹ ∃ls. v = Node x ls"
```
⟨*proof*⟩

lemma *parseSymbol_parseGamma_sound: "case x of Inl (pt, s, ts, vis) ⇒
parse_table_correct pt g
  ⟶ parseSymbol pt s ts vis = RESULT v r ⟶ (∃w. w @ r = ts ∧ sym_derives_prefix
g s w v r)
  | Inr (pt, n, gamma, ts, vis) ⇒ parse_table_correct pt g
  ⟶ (parseGamma pt n gamma ts vis = RESULT v r
  ⟶ (∃w. w @ r = ts ∧ gamma_derives_prefix g n gamma w v r))"*

⟨*proof*⟩

theorem *parse_sound: "parse_table_correct pt g ⟹ parse (PT pt) s (w @ r) = RESULT v r*
  ⟹ sym_derives_prefix g s w v r"
  ⟨*proof*⟩

7.2 Completeness

lemma *parseSymbol_parseGamma_complete_or_error:*
  assumes *"parse_table_correct pt g"*
  shows *"sym_derives_prefix g s w v r*
    ⟹ (∀vis. (∃m x ts'. parseSymbol pt s (w @ r) vis = ERROR m x ts')
    ∨ (parseSymbol pt s (w @ r) vis = RESULT v r))"
    and *"gamma_derives_prefix g y ss w v r*
     ⟹ (∀vis. (∃m x ts'. parseGamma pt y ss (w @ r) vis = ERROR m x ts')
    ∨ (parseGamma pt y ss (w @ r) vis = RESULT v r))"
⟨*proof*⟩

lemma *parse_complete_or_error: "parse_table_correct pt g ⟹ sym_derives_prefix g s w v r*
  ⟹ ∃m x ts'. parse (PT pt) s (w @ r) = ERROR m x ts'
  ∨ (parse (PT pt) s (w @ r) = RESULT v r)"
  ⟨*proof*⟩

7.3 Error-free Termination

inductive *sized_first_sym :: "('n, 't) grammar ⇒ 't lookahead ⇒ ('n, 't) symbol ⇒ nat ⇒ bool"*
  for *g* **where**
    SzFirstT: "sized_first_sym g (LA y) (T y) 0"
  | SzFirstNT: "(x, gpre @ s # gsuf) ∈ set (prods g) ⟹ nullable_gamma g gpre
    ⟹ sized_first_sym g la s n ⟹ sized_first_sym g la (NT x) (Suc n)"

lemma *first_sym_exists_size: "first_sym g la s ⟹ ∃n. sized_first_sym g la s n"*
  ⟨*proof*⟩

lemma *sized_fs_fs: "sized_first_sym g la s n ⟹ first_sym g la s"*
  ⟨*proof*⟩

lemma *medial : "pre @ s # suf = pre' @ s' # suf'*
    ⟹ s ∈ set pre'∨ s' ∈ set pre ∨ pre = pre' ∧ s = s' ∧ suf = suf'"
⟨*proof*⟩

lemma *nullable_sym_in: "nullable_gamma g gamma ⟹ s ∈ set gamma ⟹ nullable_sym g s"*

⟨*proof*⟩

lemma *nullable_split: "nullable_gamma g (xs @ ys) ⟹ nullable_gamma g ys"*
⟨*proof*⟩

lemma *first_gamma_split:*
"first_gamma g la ys ⟹ nullable_gamma g xs ⟹ first_gamma g la (xs @ ys)"
⟨*proof*⟩

lemma *follow_pre: "(x, pre @ suf) ∈ set (prods g) ⟹ s ∈ set pre ⟹ nullable_gamma g pre*
⟹ first_gamma g la suf ⟹ follow_sym g la s"
⟨*proof*⟩

lemma *no_first_follow_conflicts:*
assumes *"parse_table_correct tbl g"*
shows *"first_sym g la s ⟹ nullable_sym g s ⟹ ¬ follow_sym g la s"*
⟨*proof*⟩

lemma *first_sym_rhs_eqs: "parse_table_correct t g ⟹ (x, pre @ s # suf) ∈ set (prods g)*
⟹ (x, pre' @ s' # suf') ∈ set (prods g) ⟹ nullable_gamma g pre ⟹ nullable_gamma g pre'
⟹ first_sym g la s ⟹ first_sym g la s' ⟹ pre = pre' ∧ s = s' ∧ suf = suf'"
⟨*proof*⟩

lemma *sized_first_sym_det:*
assumes *"parse_table_correct t g"*
shows *"sized_first_sym g la s n ⟹ (∀n'. s = s' ⟶ sized_first_sym g la s' n' ⟶ n = n')"*
⟨*proof*⟩

lemma *sized_first_sym_np: "nullable_path g la x y ⟹ first_sym g la y*
⟹ ∃nx ny. sized_first_sym g la x nx ∧ sized_first_sym g la y ny ∧ ny < nx"
⟨*proof*⟩

inductive *sized_nullable_sym :: "('n, 't) grammar ⇒ ('n, 't) symbol ⇒ nat ⇒ bool"*
and *sized_nullable_gamma :: "('n, 't) grammar ⇒ ('n, 't) symbol list ⇒ nat ⇒ bool"* **for** *g* **where**
SzNullableSym: "(x, gamma) ∈ set (prods g)
⟹ sized_nullable_gamma g gamma n ⟹ sized_nullable_sym g (NT x) (Suc n)"

```
| SzNullableNil: "sized_nullable_gamma g [] 0"
| SzNullableCons: "sized_nullable_sym g s n ⟹ sized_nullable_gamma g
ss n'
    ⟹ sized_nullable_gamma g (s # ss) (n + n')"
```

lemma *sized_ng_ng: "sized_nullable_sym g s n ⟹ nullable_sym g s"*
"sized_nullable_gamma g gamma n ⟹ nullable_gamma g gamma"
⟨*proof*⟩

lemma *ng_sized_ng: "nullable_sym g s ⟹ ∃n. sized_nullable_sym g s
n"*
"nullable_gamma g gamma ⟹ ∃n. sized_nullable_gamma g gamma n"
⟨*proof*⟩

lemma *sized_nullable_sym_det':*
assumes *"parse_table_correct pt g"*
shows *"sized_nullable_sym g s n*
*⟹ (⋀n'. follow_sym g la s ⟹ sized_nullable_sym g s n' ⟹ n =
n')"*
and *"sized_nullable_gamma g gsuf n ⟹ (⋀x gpre n'. (x, gpre @ gsuf)
∈ set (prods g)*
*⟹ follow_sym g la (NT x) ⟹ sized_nullable_gamma g gsuf n' ⟹
n = n')"*
⟨*proof*⟩

lemma *sized_nullable_sym_det:*
assumes *"parse_table_correct t g"*
shows *"sized_nullable_sym g s n ⟹ follow_sym g la s ⟹ sized_nullable_sym
g s n' ⟹ n = n'"*
⟨*proof*⟩

lemma *sym_in_gamma_size_le: "nullable_gamma g gamma ⟹ s ∈ set gamma*
*⟹ ∃n n'. sized_nullable_sym g s n ∧ sized_nullable_gamma g gamma
n' ∧ n ≤ n'"*
⟨*proof*⟩

lemma *sized_ns_np:*
assumes *"(x, pre @ NT y # suf) ∈ set (prods g)" "nullable_gamma g (pre
@ NT y # suf)"*
"nullable_sym g (NT y)"
shows *"∃nx ny. sized_nullable_sym g (NT x) nx ∧ sized_nullable_sym
g (NT y) ny ∧ ny < nx"*
⟨*proof*⟩

lemma *exist_decreasing_nullable_sym_sizes_in_null_path:*
shows *"nullable_path g la x y ⟹ parse_table_correct t g ⟹ nullable_sym
g x*
⟹ follow_sym g la x
⟹ ∃nx ny. sized_nullable_sym g x nx ∧ sized_nullable_sym g y ny

```
∧ ny < nx"
```
⟨*proof*⟩

lemma *nullable_path_exists_production: "nullable_path g la (NT x) y*
⟹ ∃gamma. (x, gamma) ∈ set (prods g) ∧ lookahead_for la x gamma g"
⟨*proof*⟩

lemma *ll1_parse_table_impl_no_left_recursion:*
assumes *"parse_table_correct tbl (g :: ('n, 't) grammar)"*
shows *"¬ left_recursive g (NT x) la"*
⟨*proof*⟩

lemma *input_length_lt_or_nullable_sym: "case x of Inl (pt, s, ts, vis)*
⇒ parse_table_correct pt g
⟶ parseSymbol pt s ts vis = RESULT v r ⟶ length r < length ts
∨ nullable_sym g s
| Inr (pt, x, ss, ts, vis) ⇒ parse_table_correct pt g ⟶ parseGamma
pt x ss ts vis = RESULT v r
⟶ length r < length ts ∨ nullable_gamma g ss"
⟨*proof*⟩

lemma *input_length_eq_nullable_sym:*
"parse_table_correct tbl g ⟹ parseSymbol tbl s ts vis = RESULT v ts
⟹ nullable_sym g s"
⟨*proof*⟩

lemma *error_conditions: "case y of*
Inl (pt, s, ts, vis) ⇒ parse_table_correct pt g ⟶ parseSymbol pt
s ts vis = ERROR m z ts'
⟶ ((z |∈| vis ∧ (s = NT z ∨ nullable_path g (peek ts) s (NT z)))
∨ (∃la. left_recursive g (NT z) la))
| Inr (pt, x, ss, ts, vis) ⇒ parse_table_correct pt g ⟶
parseGamma pt x ss ts vis = ERROR m z ts' ⟶ (∃pre s suf. ss =
pre @ s # suf
∧ nullable_gamma g pre ∧ z |∈| vis ∧ (s = NT z ∨ nullable_path
g (peek ts) s (NT z)))
∨ (∃la. (left_recursive g (NT z) la))"
⟨*proof*⟩

theorem *parse_terminates_without_error:*
"parse_table_correct pt g ⟹ parse (PT pt) s (w @ r) ≠ ERROR m x ts'"
⟨*proof*⟩

theorem *parse_complete: "parse_table_correct pt g ⟹ sym_derives_prefix*
g s w v r
⟹ parse (PT pt) s (w @ r) = RESULT v r"
⟨*proof*⟩

end

declare *parse.parseSymbol.simps [code]*
declare *parse.parseGamma.simps [code]*
declare *parse.parse.simps [code]*
declare *parse.parseToString.simps [code]*
declare *parse.parseTreeToString.simps [code]*
declare *parse.mismatchMessage_def [code]*

end

7.4 Interpretation

theory *LL1_Parser_show*
  imports *LL1_Parser "Show.Show"*
begin

global_interpretation *parse_show: parse "show" "show"*
  defines *parse = parse_show.parse*
    and *parseToString = parse_show.parseToString*
    and *parseSymbol = parse_show.parseSymbol*
    and *parseGamma = parse_show.parseGamma*
    and *mismatchMessage = parse_show.mismatchMessage*
  ⟨*proof*⟩

end

8 Examples

theory *Parser_Example*
  imports *LL1_Parser_show "Show.Show_Instances"*
begin

In this section we present two examples for LL1-grammars to show how the parser generator can be used to create a parse tree from a sequence of symbols.

8.1 Mini-language

The first example is based on Grammar 3.11 from Appel's "Modern Compiler Implementation in ML" [1]:

S → if E then S else S | begin S L | print E

L → end | ; S L

E → num = num

datatype *terminal = If | Then | Else | Begin | Print | End | Semi | Num | Eq*

datatype *nterminal = S | L | E*

```
derive "show" "terminal"
derive "show" "nterminal"
derive "show" "(terminal, nterminal) symbol"

definition gr :: "(nterminal, terminal) grammar" where
  "gr = G S [
    (S, [T If, NT E, T Then, NT S, T Else, NT S]),
    (S, [T Begin, NT S, NT L]),
    (S, [T Print, NT E]),

    (L, [T End]),
    (L, [T Semi, NT S, NT L]),

    (E, [T Num, T Eq, T Num])
  ]"

definition pt :: "(nterminal, terminal) ll1_parse_table" where
  "pt = mkParseTable gr"
```

— We parse lists of pairs of terminal symbols and lexemes. We ignore the latter here.

```
definition L where
  "L x = (x, ())"

lemma "parse pt (NT (start gr))
  (map L [If, Num, Eq, Num, Then, Print, Num, Eq, Num, Else, Print, Num,
Eq, Num]) =
  RESULT (map_parse_tree id L
     (Node S
       [Leaf If, Node E [Leaf Num, Leaf Eq, Leaf Num], Leaf Then,
        Node S [Leaf Print, Node E [Leaf Num, Leaf Eq, Leaf Num]], Leaf
Else,
        Node S [Leaf Print, Node E [Leaf Num, Leaf Eq, Leaf Num]]]))
     []"
  ⟨proof⟩
```

Example input:

```
if 2 = 5 then
  print 2 = 5
else
  print 42 = 42
```

```
lemma "parseToString pt (NT (start gr))
  (map L [If, Num, Eq, Num, Then, Print, Num, Eq, Num, Else, Print, Num,
Eq, Num]) =
  ''(If, ()) (Num, ()) (Eq, ()) (Num, ()) (Then, ()) (Print, ()) (Num,
()) (Eq, ()) (Num, ()) ''
```

```
  @ ''(Else, ()) (Print, ()) (Num, ()) (Eq, ()) (Num, ())''"
```
⟨*proof*⟩

Example input:

```
if 2 5 then
  print 2 = 5
else
  print 42 = 42
```

lemma *"parse pt (NT (start gr))*
(map L [If, Num, Num, Then, Print, Num, Eq, Num, Else, Print, Num, Eq, Num]) =
REJECT ''Token mismatch. Expected Eq, saw Num (())''
(map L [Num, Then, Print, Num, Eq, Num, Else, Print, Num, Eq, Num])"
⟨*proof*⟩

end

8.2 Generating a JSON Parser

theory *Json_Parser*
imports *LL1_Parser_show "Show.Show_Instances"*
begin

datatype *terminal =*
TInt
| *Float*
| *Str*
| *Tru*
| *Fls*
| *Null*
| *LeftBrace*
| *RightBrace*
| *LeftBrack*
| *RightBrack*
| *Colon*
| *Comma*

datatype *nterminal =*
Value
| *Pairs*
| *PairsTl*
| *Pair*
| *Elts*
| *EltsTl*

derive *"show" "terminal"*
derive *"show" "nterminal"*

```
derive "show" "(terminal, nterminal) symbol"

definition jsonGrammar :: "(nterminal, terminal) grammar" where
  "jsonGrammar = G Value [
    (Value, [T LeftBrace, NT Pairs, T RightBrace]),
    (Value, [T LeftBrack, NT Elts, T RightBrack]),
    (Value, [T Str]),
    (Value, [T TInt]),
    (Value, [T Float]),
    (Value, [T Tru]),
    (Value, [T Fls]),
    (Value, [T Null]),

    (Pairs, []),
    (Pairs, [NT Pair, NT PairsTl]),

    (PairsTl, []),
    (PairsTl, [T Comma, NT Pair, NT PairsTl]),

    (Pair, [T Str, T Colon, NT Value]),

    (Elts, []),
    (Elts, [NT Value, NT EltsTl]),

    (EltsTl, []),
    (EltsTl, [T Comma, NT Value, NT EltsTl])
  ]"

definition pt :: "(nterminal, terminal) ll1_parse_table" where
  "pt = mkParseTable jsonGrammar"

datatype lex =
    LInt (lex_int: int)
  | LFloat
  | LStr (lex_str: string)
  | LNone

derive "show" "lex"

definition
  "mkS x = (x, LNone)"

definition
  "StrS s = (Str, LStr s)"

abbreviation
  "LeftBraceS ≡ mkS LeftBrace"

abbreviation
```

```
  "RightBraceS ≡ mkS RightBrace"

abbreviation
  "LeftBrackS ≡ mkS LeftBrack"

abbreviation
  "RightBrackS ≡ mkS RightBrack"

abbreviation
  "ColonS ≡ mkS Colon"

abbreviation
  "CommaS ≡ mkS Comma"

abbreviation
  "FlsS ≡ mkS Fls"

abbreviation
  "TruS ≡ mkS Tru"

definition
  "IntS s = (TInt, LInt s)"
```

Example input: `{`

```
  "items": []
}
```

```
lemma "parse pt (NT (start jsonGrammar))
  [LeftBraceS, StrS ''items'', ColonS, LeftBrackS, RightBrackS, RightBraceS]
=
  RESULT
   (Node Value
       [Leaf (mkS LeftBrace),
        Node Pairs [
          Node Pair [Leaf (StrS ''items''), Leaf (mkS Colon),
            Node Value [Leaf (mkS LeftBrack), Node Elts [], Leaf (mkS
RightBrack)]],
          Node PairsTl []],
        Leaf (mkS RightBrace)])
   []"
```

⟨*proof*⟩

Example input: `{`

```
  "items": [
    {
    "id":           65,
    "description":  "Title",
    "visible":      false},
```

```
    {
    "id":           42,
    "visible":      true}
  ]
}
```

lemma *"parse pt (NT (start jsonGrammar))*

```
  [LeftBraceS, StrS ''items'', ColonS, LeftBrackS,
    LeftBraceS, StrS ''id'', ColonS, IntS 65, CommaS, StrS ''description'',
ColonS, StrS ''Title'',
    CommaS, StrS ''visible'', ColonS, FlsS, RightBraceS, CommaS,
    LeftBraceS, StrS ''id'', ColonS, IntS 42, CommaS, StrS ''visible'',
ColonS, TruS,
    RightBraceS, RightBrackS, RightBraceS] =
  RESULT
   (Node Value
       [Leaf LeftBraceS,
        Node Pairs [
          Node Pair [Leaf (StrS ''items''), Leaf ColonS,
            Node Value
             [Leaf LeftBrackS,
              Node Elts
               [Node Value
                 [Leaf LeftBraceS,
                  Node Pairs
                   [Node Pair [Leaf (StrS ''id''), Leaf ColonS, Node Value
[Leaf (IntS 65)]],
                    Node PairsTl [Leaf CommaS,
                      Node Pair [
                        Leaf (StrS ''description''), Leaf ColonS, Node
Value [Leaf (StrS ''Title'')]
                      ],
                      Node PairsTl [Leaf CommaS,
                      Node Pair [
                        Leaf (StrS ''visible''), Leaf ColonS,
                        Node Value [Leaf FlsS]], Node PairsTl []
                      ]]],
                  Leaf RightBraceS],
                Node EltsTl
                 [Leaf CommaS,
                  Node Value
                   [Leaf LeftBraceS,
                    Node Pairs
                     [Node Pair [Leaf (StrS ''id''), Leaf ColonS, Node
Value [Leaf (IntS 42)]],
                      Node PairsTl [Leaf CommaS,
                      Node Pair [Leaf (StrS ''visible''), Leaf ColonS,
Node Value [Leaf TruS]],
                      Node PairsTl []]],
```

```
                Leaf RightBraceS],
              Node EltsTl []]],
          Leaf RightBrackS]],
      Node PairsTl []],
    Leaf RightBraceS])
  []"
```
⟨*proof*⟩

8.3 Reading the Parse Tree

datatype *JSON =*
```
  Object "(string × JSON) list"
| Array "JSON list"
| String string
```
— An Isabelle string rather than a JSON string
```
| Int int
```
— The number type is split into natural Isabelle/HOL types
```
| Nat nat
| Rat nat int
| Boolean bool
```
— True and False are contracted to one constructor
```
| Nil
```

primrec *fold_parse_tree :: "('t ⇒ 'a) ⇒ ('a list ⇒ 'n ⇒ 'a) ⇒ ('n, 't) parse_tree ⇒ 'a"*
where
```
  "fold_parse_tree t n (Leaf x) = t x"
| "fold_parse_tree t n (Node x ts) = n (map (fold_parse_tree t n) ts) x"
```

definition
```
  "json_leaf ≡ λ(t, s). (case t of
    Str => JSON.String (lex_str s) |
    TInt => JSON.Int (lex_int s) |
    Float ⇒ JSON.Rat 1 0 |
    Tru ⇒ JSON.Boolean True |
    Fls => JSON.Boolean False |
    Null => JSON.Nil |
    _ ⇒ JSON.Nil
)"
```

definition
```
  "combine_objects x y = (case x of JSON.Object xs ⇒ case y of JSON.Object ys ⇒
    JSON.Object (xs @ ys)
  )"
```

definition
```
  "cons_array x y = (case y of JSON.Array ys ⇒
    JSON.Array (x # ys)
  )"
```

definition
```
  "the_str = (λJSON.String s ⇒ s)"
```

definition
```
  "json_node vs = (
  λ Value ⇒ (
    case vs of
      [x] ⇒ x
    | [x,y,z] ⇒ y
  )
  | Pair ⇒ (
    case vs of
      [s, _, v] ⇒ JSON.Object [(the_str s, v)]
  )
  | Pairs ⇒ (
    case vs of
      [] ⇒ JSON.Object []
    | [n,ntl] ⇒ combine_objects n ntl
  )
  | PairsTl ⇒ (
    case vs of
      [] ⇒ JSON.Object []
    | [_,n,ntl] ⇒ combine_objects n ntl
  )
  | Elts ⇒ (
    case vs of
      [] ⇒ JSON.Array []
    | [n,ntl] ⇒ cons_array n ntl
  )
  | EltsTl ⇒ (
    case vs of
      [] ⇒ JSON.Array []
    | [_,n,ntl] ⇒ cons_array n ntl
  )
)"
```

definition
```
  "parse_tree_to_json = fold_parse_tree json_leaf json_node"
```

definition
```
  "the_RESULT = (λRESULT r _ ⇒ r)"
```

lemma `"parse_tree_to_json (the_RESULT (parse pt (NT (start jsonGrammar))
  [LeftBraceS, StrS ''items'', ColonS, LeftBrackS, RightBrackS, RightBraceS]))
= Object [(''items'', Array [])]"`
  ⟨*proof*⟩

lemma `"parse_tree_to_json (the_RESULT (parse pt (NT (start jsonGrammar))
  [LeftBraceS, StrS ''items'', ColonS, LeftBrackS,`

```
    LeftBraceS, StrS ''id'', ColonS, IntS 65, CommaS, StrS ''description'',
ColonS, StrS ''Title'',
    CommaS, StrS ''visible'', ColonS, FlsS, RightBraceS, CommaS,
    LeftBraceS, StrS ''id'', ColonS, IntS 42, CommaS, StrS ''visible'',
ColonS, TruS,
    RightBraceS, RightBrackS, RightBraceS]))
= Object
    [(''items'',
      Array
       [Object
         [(''id'', JSON.Int 65), (''description'', String ''Title''),
          (''visible'', Boolean False)],
        Object [(''id'', JSON.Int 42), (''visible'', Boolean True)]])]"
```
⟨*proof*⟩

Note that in Vermillion one can attach the functions to read parse trees directly to the production rules. While this would be possible in Isabelle/HOL, it would give little advantage over defining the reader function directly, since we are missing dependent types.

end

References

[1] A. W. Appel. *Modern Compiler Implementation in ML*. Cambridge University Press, 1998.

[2] S. Lasser, C. Casinghino, K. Fisher, and C. Roux. A Verified LL(1) Parser Generator. In J. Harrison, J. O'Leary, and A. Tolmach, editors, *10th International Conference on Interactive Theorem Proving (ITP 2019)*, volume 141 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 24:1–24:18, Dagstuhl, Germany, 2019. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[3] R. Wilhelm, H. Seidl, and S. Hack. *Compiler Design - Syntactic and Semantic Analysis*. Springer, 2013.