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'))"
  by auto

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)"
  by (induction ps) (auto simp add: nullablePass_def updateNu_def)

lemma nullablePass_Nil[simp]: "nullablePass [] nu = nu"
  by (simp add: nullablePass_def)

lemma nullablePass_cons[simp]: "nullablePass ((y,gamma)#ps') nu =
  (if nullableGamma gamma (nullablePass ps' nu) then insert y else id)
(nullablePass ps' nu)"
  by (simp add: nullablePass_def updateNu_def)

lemma nullable_pass_mono:
  "nullablePass ps nu ⊆ nullablePass (qs @ ps) nu"
  by (induction qs) (auto simp add: nullablePass_def updateNu_def)

lemma nullablePass_subset_lhSet:
  "nullablePass ps nu ⊆ lhSet ps ∪ nu"
  by (induction ps arbitrary: nu; fastforce split: if_split_asm simp:
lhSet_def)
```

```
lemma nullablePass_neq_candidates_lt:
  assumes "nu ≠ nullablePass ps nu"
  shows "countNullCands ps (nullablePass ps nu) < countNullCands ps nu"
proof -
  have "finite (lhSet ps)" by (simp add: lhSet_def)
  then have A: "finite (lhSet ps - nu)" using finite_subset[where B =
"lhSet ps"] lhSet_def by auto
  moreover have B: "lhSet ps - nullablePass ps nu ⊂ lhSet ps - nu"
    using nullablePass_subset nullablePass_subset_lhSet assms by fastforce
  ultimately show ?thesis unfolding countNullCands_def by (simp add:
Let_def psubset_card_mono)
qed

termination mkNullableSet'
  using nullablePass_neq_candidates_lt
  by (relation "measure (λ(ps, nu). countNullCands ps nu)") force+
```

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"
  by (induction rule: nullableGamma.induct)
     (auto
       intro: nullable_sym_nullable_gamma.intros split: if_split_asm simp:
nullable_set_sound_def)

lemma nullablePass_preserves_soundness':
  "nullable_set_sound nu g ⟹ set ps ⊆ set (prods g)
  ⟹ nullable_set_sound (nullablePass ps nu) g"
proof (induction ps)
  case Nil
  then show ?case by simp
next
  case (Cons p ps)
  obtain x gamma where p_def: "p = (x, gamma)" by fastforce
```

```
  show ?case
  proof (cases "nullableGamma gamma (nullablePass ps nu)")
    case True
    have "nullable_set_sound (nullablePass ps nu) g"
      using Cons.IH Cons.prems(1,2) by force
    moreover have "nullablePass (p # ps) nu ⊆ nullablePass ps nu ∪ {x}"
      by (cases "nullableGamma gamma (nullablePass ps nu)")
        (auto simp add: ‹p = (x, gamma)›)
    ultimately show ?thesis
      using Cons.prems(2) True nu_sound_nullableGamma_sound nullable_sym.simps
p_def
      by (fastforce simp add: nullable_set_sound_def)
  next
    case False
    then show ?thesis using Cons p_def by force
  qed
qed

lemma nullablePass_preserves_soundness:
  "nullable_set_sound nu g ⟹ nullable_set_sound (nullablePass (prods
g) nu) g"
  using nullablePass_preserves_soundness' by auto

lemmas [simp del] = mkNullableSet'.simps

lemma mkNullableSet'_preserves_soundness:
  "nullable_set_sound nu g ⟹ nullable_set_sound (mkNullableSet' (prods
g) nu) g"
  by (induction "prods g" nu rule: mkNullableSet'.induct)
     (subst mkNullableSet'.simps, auto intro: nullablePass_preserves_soundness
simp: Let_def)

lemma empty_nu_sound: "nullable_set_sound {} g"
  by (simp add: nullable_set_sound_def)

theorem mkNullableSet_sound: "nullable_set_sound (mkNullableSet g) g"
  unfolding mkNullableSet_def using empty_nu_sound by (rule mkNullableSet'_preserves_soundn
```

3.4 Completeness

```
lemma nullablePass_add_equal: "x ∈ nu ⟹ nullablePass ps nu = insert
x (nullablePass ps nu)"
  using nullablePass_subset by fastforce

lemma nullable_gamma_nullableGamma_true:
  "nullable_gamma g ys ⟹ ∀x. (NT x) ∈ set ys ⟶ x ∈ nu ⟹ nullableGamma
ys nu"
  by (induction rule: nullableGamma.induct; force elim: nullable_gamma.cases
nullable_sym.cases)
```

```
lemma nullableGamma_saturated_if_nullablePass_fixpoint:
  assumes "nu = nullablePass ps nu"
  shows "∀(x, gamma) ∈ set ps. nullableGamma gamma nu ⟶ x ∈ nu"
  using assms nullablePass_add_equal by (induction ps) (fastforce split:
if_split_asm)+

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"
  using assms
proof (induction rule: nullable_sym_nullable_gamma.inducts)
  case (NullableSym x gamma)
  then show ?case
    using nullableGamma_saturated_if_nullablePass_fixpoint nullable_gamma_nullableGamma_tru
    by force
qed auto

lemma nullablePass_equal_complete: "nu = (nullablePass (prods g) nu)
⟹ nullable_set_complete nu g"
  by (simp add: nullablePass_equal_complete' nullable_set_complete_def)

lemma mkNullableSet'_complete: "nullable_set_complete (mkNullableSet'
(prods g) nu) g"
proof (induction "prods g" nu rule: mkNullableSet'.induct)
  case (1 nu)
  then show ?case
    by (subst mkNullableSet'.simps)
      (simp add: Let_def nullable_set_complete_def nullablePass_equal_complete')
qed

theorem mkNullableSet_complete: "nullable_set_complete (mkNullableSet
g) g"
  unfolding mkNullableSet_def
  using mkNullableSet'_complete by blast

theorem mkNullableSet_correct: "nullable_set_for (mkNullableSet g) g"
  using mkNullableSet_sound mkNullableSet_complete by auto

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"
  unfolding list_union_def
  by auto

lemma set_list_union[simp]: "set (ls1 @@ ls2) = set ls1 ∪ set ls2"
  unfolding list_union_def
  by auto

lemma mem_list_union: "ls1 = ls1 @@ ls2 ⟹ e ∈ set ls2 ⟹ e ∈ set
ls1"
  by (metis Un_iff set_list_union)

lemma list_union_I2: "e ∈ set ls2 ⟹ e ∈ set (ls1 @@ ls2)"
  by simp

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)"*
unfolding *leftmostLookaheads_def* **by** *auto*

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)"
  by (induction gpre) (auto simp: all_nt_def split: symbol.splits)

lemma gpre_nullable_in_leftmost_lks:
  "(x, (gpre @ (T y) # gsuf)) ∈ set ps ⟹ all_nt gpre ⟹ (LA y) ∈ leftmostLookaheads
ps"
proof (induction ps)
  case Nil
  then show ?case by auto
next
  case (Cons p ps')
  then show ?case
  proof (cases "p = (x, gpre @ (T y) # gsuf)")
    case True
    then show ?thesis unfolding leftmostLookaheads_def
      using Cons.prems(2) gpre_nullable_leftmost_lk_some by fastforce
  next
    case False
    then show ?thesis using Cons by (auto simp add: in_leftmostLookaheads_cons)
  qed
qed

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"
  using assms
proof (induction gsuf arbitrary: gpre)
  case Nil
  then show ?case by auto
next
  case (Cons y gsuf)
  consider (T) a where "y = T a" | (NT_nullable) a where "y = NT a"
and "nullableSym y nu"
    | (NT_not_nullable)  a where "y = NT a" and "¬ nullableSym y nu"
by (cases y; auto)
  then show ?case
  proof cases
    case T
    then show ?thesis using Cons.prems by (auto intro: gpre_nullable_in_leftmost_lks)
  next
    case (NT_nullable a)
    then consider (in_findOrEmpty) "la ∈ set (findOrEmpty a fi)"
      | (in_firstGamma) "la ∈ set (firstGamma gsuf nu fi)"
      using Cons.prems(1) by fastforce
    then show ?thesis
    proof cases
```

```
      case in_findOrEmpty
      then have "(a, la) ∈ pairsOf fi" unfolding findOrEmpty_def
        using fmdom_notD in_findOrEmpty pairsOf_def by fastforce
      then show ?thesis using Cons.prems(3) unfolding all_pairs_are_first_candidates_def
by auto
    next
      case in_firstGamma
      then show ?thesis using Cons.prems(2,3,4) by
        (auto intro: Cons.IH[of "gpre @ [y]"] simp add: all_nt_def NT_nullable)
    qed
  next
    case NT_not_nullable
    then have "(a, la) ∈ pairsOf fi" using Cons.prems(1) unfolding pairsOf_def
by
      (cases "fmlookup fi a"; auto intro: fmdomI simp add: NT_not_nullable
findOrEmpty_def)
    then show ?thesis using Cons.prems(3) unfolding all_pairs_are_first_candidates_def
by auto
  qed
qed

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"
  by (auto intro: in_firstGamma_in_leftmost_lks'[of x "[]" gamma] simp
add: all_nt_def)

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"
  unfolding updateFi_def fg_def[symmetric] xFirst_def[symmetric] xFirst'_def[symmetric]
  by atomize_elim (auto split: if_split_asm simp: Let_def xFirst'_def
fg_def xFirst_def)

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"
  using Nil Cons_changed Cons_same
  by -(rule list.induct[where ?P = "λls. P ls nu fi"]; blast)

lemma in_findOrEmpty_iff_in_pairsOf: "la ∈ set (findOrEmpty x fi) ⟷
(x, la) ∈ pairsOf fi"
  unfolding findOrEmpty_def pairsOf_def using fmdom_notD by fastforce

lemma in_pairsOf_exists: "(x, la) ∈ pairsOf fi ⟷ (∃s. fmlookup fi
x = Some s ∧ la ∈ set s)"
  unfolding pairsOf_def findOrEmpty_def using fmlookup_dom_iff by fastforce

lemma in_findOrEmpty_exists_set:
    "la ∈ set (findOrEmpty x m) ⟷ (∃s. fmlookup m x = Some s ∧ la
∈ set s)"
  using in_findOrEmpty_iff_in_pairsOf in_pairsOf_exists by fast

lemma in_add_value: "(x, la) ∈ pairsOf (fmupd x s fi) ⟷ la ∈ set
s"
  by (simp add: in_pairsOf_exists)

lemma firstPass_Nil[simp]: "firstPass [] x y = y"
  unfolding firstPass_def by simp
```

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)"
  by (simp add: firstPass_def)

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)"
  by (auto simp add: updateFi_def Let_def list_union_def)

lemma in_add_keys: "la ∈ set s ⟷ (x, la) ∈ pairsOf (fmupd x s fi)"
  by (simp add: in_findOrEmpty_iff_in_pairsOf[symmetric] findOrEmpty_def)

lemma in_add_keys_neq: "x ≠ y ⟹ (y, la) ∈ pairsOf fi ⟷ (y, la)
∈ pairsOf (fmupd x s fi)"
  by (simp add: findOrEmpty_def pairsOf_def)


lemma updateFi_subset: "pairsOf fi ⊆ pairsOf (updateFi nu p fi)"
proof (rule subrelI)
  fix y la
```

```
  assume A: "(y, la) ∈ pairsOf fi"
  obtain x gamma where p_def: "p = (x, gamma)" by (cases p)
  then consider "updateFi nu (x, gamma) fi = fi"
    | "x = y" "updateFi nu (x, gamma) fi = fmupd x (findOrEmpty x fi @@
firstGamma gamma nu fi) fi"
    | "x ≠ y" "updateFi nu (x, gamma) fi = fmupd x (findOrEmpty x fi
@@ firstGamma gamma nu fi) fi"
    using unfold_updateFi by metis
  then show "(y, la) ∈ pairsOf (updateFi nu p fi)"
  proof (cases)
    case 1
    then show ?thesis using p_def A by simp
  next
    case 2
    then show ?thesis
      by (simp add: A in_add_value in_findOrEmpty_iff_in_pairsOf p_def
list_union_def)
  next
    case 3
    then show ?thesis
      by (metis A in_add_keys_neq p_def)
  qed
qed

lemma firstPass_cons_subset: "pairsOf (firstPass ps nu fi) ⊆ pairsOf
(firstPass (p # ps) nu fi)"
  using updateFi_subset by (cases p, simp, blast)

lemma firstPass_mono: "pairsOf (firstPass ps nu fi) ⊆ pairsOf (firstPass
(qs @ ps) nu fi)"
  by (induction qs arbitrary: ps) (simp, metis append_Cons firstPass_cons_subset
subsetD subsetI)

lemma firstPass_subset: "pairsOf fi ⊆ pairsOf (firstPass ps nu fi)"
  using firstPass_cons_subset by (induction ps; simp add: firstPass_def;
blast)

lemma firstPass_empty_set:
  "fmlookup (firstPass ps nu fi) x = Some [] ⟹ fmlookup fi x = Some
[]"
proof (induction ps)
  case Nil
  then show ?case by simp
next
  case (Cons p ps)
  then show ?case using Cons by (cases p) (auto simp add: unfold_updateFi
split: if_split_asm)
qed
```

```
lemma firstPass_None: "fmlookup (firstPass ps nu fi) x = None ⟹ fmlookup
fi x = None"
proof (induction ps)
  case Nil
  then show ?case by simp
next
  case (Cons p ps)
  then show ?case using Cons by (cases p) (auto simp add: unfold_updateFi
split: if_split_asm)
qed

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 (cases "fmlookup (firstPass ps nu fi) x")
  case None
  then have "fmlookup fi x = None" by (auto intro: firstPass_None)
  then show ?thesis using None assms by auto
next
  case (Some xSet)
  then have "xSet ≠ []" using firstPass_empty_set assms by fastforce
  then show ?thesis
    using assms
    by (auto simp add: Some findOrEmpty_def split: option.splits)
qed
```

Injectivity of `pairsOf`

```
lemma firstPass_only_appends: "∃suf. findOrEmpty x (firstPass ps nu
fi) = findOrEmpty x fi @ suf"
  unfolding firstPass_def
proof (induction ps)
  case Nil
  then show ?case by auto
next
  case (Cons p ps)
  obtain y gamma where p_def: "p = (y, gamma)" by (cases p)
  let ?fg = "firstGamma gamma nu (foldr (updateFi nu) ps fi)"
  let ?xFirst = "findOrEmpty y (foldr (updateFi nu) ps fi)"
  let ?xFirst' = "?xFirst @@ ?fg"
  show ?case
  proof (cases "?xFirst' = ?xFirst ∨ ?fg = []")
    case True
    then show ?thesis using p_def Cons by (auto simp add: updateFi_def)
  next
    case False
    note outerFalse = this
    have "findOrEmpty x (foldr (updateFi nu) (p # ps) fi)
        = findOrEmpty x (fmupd y ?xFirst' (foldr (updateFi nu) ps fi))"
      using p_def False by (auto simp add: updateFi_def)
```

```
    then show ?thesis using Cons by (cases "x = y") (auto simp add:
list_union_def findOrEmpty_def)
  qed
qed

lemma firstPass_suf_distinct: "findOrEmpty x (firstPass ps nu fi) = findOrEmpty
x fi @ suf
  ⟹ suf ≠ [] ⟹ la ∈ set suf ⟹ la ∉ set (findOrEmpty x fi)"
proof (induction ps arbitrary: x fi suf)
  case Nil
  then show ?case by auto
next
  case (Cons p ps)
  obtain y gamma where p_def: "p = (y, gamma)" by (cases p)
  let ?fg = "firstGamma gamma nu (foldr (updateFi nu) ps fi)"
  let ?xFirst = "findOrEmpty y (foldr (updateFi nu) ps fi)"
  let ?xFirst' = "?xFirst @@ ?fg"
  show ?case
  proof (cases "?xFirst' = ?xFirst ∨ ?fg = []")
    case True
    then show ?thesis
      using Cons.IH Cons.prems(1-3) p_def
      by (simp add: firstPass_def updateFi_def)
  next
    case False
    obtain suf' where suf'_def: "findOrEmpty x (firstPass ps nu fi) =
findOrEmpty x fi @ suf'"
      by (meson firstPass_only_appends)
    then show ?thesis
      proof (cases "x = y")
        case True
        then have "findOrEmpty x (firstPass (p # ps) nu fi) = ?xFirst'"
          using False p_def
          by (simp add: findOrEmpty_def firstPass_def updateFi_def)
        then have "findOrEmpty x fi @ suf
            = (findOrEmpty x fi @ suf') @@ firstGamma gamma nu (firstPass
ps nu fi)"
          using Cons.prems(1) suf'_def True
          by (auto simp add: firstPass_def)
        then show ?thesis unfolding list_union_def using Cons suf'_def
by force
      next
        case False
        then have "findOrEmpty x (firstPass (p # ps) nu fi) = findOrEmpty
x (firstPass ps nu fi)"
          using p_def by (auto simp add: findOrEmpty_def updateFi_def
Let_def)
        then show ?thesis using Cons by auto
      qed
```

```
  qed
qed

lemma pairsOf_inj: "fi ≠ firstPass ps nu fi ⟹ pairsOf fi ≠ pairsOf
(firstPass ps nu fi)"
proof -
  assume "fi ≠ firstPass ps nu fi"
  then obtain y where y_diff: "findOrEmpty y fi ≠ findOrEmpty y (firstPass
ps nu fi)"
    using firstPass_neq_findOrEmpty
    by (metis fmap_ext)
  obtain suf where suf_def: "findOrEmpty y (firstPass ps nu fi) = findOrEmpty
y fi @ suf"
      and "suf ≠ []"
    using firstPass_only_appends y_diff by force
  then obtain la where "la ∈ set suf" and "la ∉ set (findOrEmpty y
fi)"
    by (meson firstPass_suf_distinct last_in_set suf_def)
  then show ?thesis
    using in_findOrEmpty_iff_in_pairsOf suf_def
    by fastforce
qed

lemma firstPass_not_equiv_subset:
  "fi ≠ firstPass ps nu fi ⟹ pairsOf fi ⊂ pairsOf (firstPass ps nu
fi)"
  using pairsOf_inj firstPass_subset by blast

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 (induction ps)
  case Nil
  then show ?case by simp
next
  case (Cons p ps)
  obtain x gamma where "p = (x, gamma)" by fastforce
  from Cons.prems have "all_pairs_are_first_candidates (firstPass ps
nu fi) (p # ps)"
    unfolding all_pairs_are_first_candidates_def using firstPass_cons_subset
by blast
  then have "set (firstGamma gamma nu (firstPass ps nu fi)) ⊆ leftmostLookaheads
(p # ps)"
    by (auto intro: in_firstGamma_in_leftmost_lks simp add: ‹p = (x, gamma)›)
  then show ?case using Cons.prems all_pairs_are_first_candidates_def
by fastforce
qed
```

```
lemma finite_leftmostLookaheads: "finite (leftmostLookaheads ps)"
  unfolding leftmostLookaheads_def by auto

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 -
  assume A1: "all_pairs_are_first_candidates (firstPass ps nu fi) ps"
    and A2: "fi ≠ (firstPass ps nu fi)"
  have "finite (lhSet ps × leftmostLookaheads ps)"
    by (simp add: finite_leftmostLookaheads lhSet_def)
  moreover have "lhSet ps × leftmostLookaheads ps - pairsOf (firstPass
ps nu fi)
      ⊂ lhSet ps × leftmostLookaheads ps - pairsOf fi"
    using firstPass_not_equiv_subset firstPass_subset_lhs_lks A1 A2 by
fastforce
  ultimately have "card (lhSet ps × leftmostLookaheads ps - pairsOf (firstPass
ps nu fi))
    < card (lhSet ps × leftmostLookaheads ps - pairsOf fi)"
    by (auto intro: psubset_card_mono)
  then show "countFirstCands ps (firstPass ps nu fi) < countFirstCands
ps fi"
    unfolding countFirstCands_def by simp
qed

lemma firstPass_preserves_apac': "all_pairs_are_first_candidates fi (ps1
@ ps2)
  ⟹ all_pairs_are_first_candidates (firstPass ps2 nu fi) (ps1 @ ps2)"
proof (induction ps2 arbitrary: ps1)
  case Nil
  then show ?case unfolding all_pairs_are_first_candidates_def by (simp
add: firstPass_def)
next
  case (Cons p ps2')
  obtain x gamma where p_def: "p = (x, gamma)" by fastforce
  then show ?case using Cons.IH[of "ps1 @ [p]"] Cons.prems
  proof (cases rule:
      updateFi_cases[where x = x and nu = nu and gamma = gamma and
fi = "firstPass ps2' nu fi"])
    case (new la)
    then have "x' ∈ lhSet (ps1 @ p # ps2') ∧ la' ∈ leftmostLookaheads
(ps1 @ p # ps2')"
      if "(x', la') ∈ pairsOf (firstPass (p # ps2') nu fi)" for x' la'
    proof -
      from that consider "x = x'" and "la' ∈ set (findOrEmpty x (firstPass
ps2' nu fi))"
        | "x = x'" and "la' ∈ set (firstGamma gamma nu (firstPass ps2'
nu fi))"
```

```
        | "(x', la') ∈ pairsOf (firstPass ps2' nu fi)"
        unfolding p_def
        using firstPass_cons[of "(x, gamma)" ps2' nu fi] unfold_updateFi[of
nu x gamma]
          in_add_keys in_add_keys_neq in_atleast1_list
        by metis
      then show ?thesis using Cons.prems Cons.IH[of "ps1 @ [p]"]
        by (cases, auto intro: in_firstGamma_in_leftmost_lks
          simp: lhSet_def p_def all_pairs_are_first_candidates_def in_findOrEmpty_iff_in_pa
    qed
    then show ?thesis by (simp add: all_pairs_are_first_candidates_def)
  qed (auto simp add: p_def)
qed

lemma firstPass_preserves_apac:
  "all_pairs_are_first_candidates fi ps ⟹ all_pairs_are_first_candidates
(firstPass ps nu fi) ps"
  using firstPass_preserves_apac'[of fi "[]" ps] by auto
```

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"
  using that
proof (induction "(ps, nu, fi)" arbitrary: fi
    rule: measure_induct_rule[where f = "λ(ps, nu, fi). countFirstCands
ps fi"])
  case (less fi)
  have "fi ≠ firstPass ps nu fi ⟹ countFirstCands ps (firstPass ps
nu fi) < countFirstCands ps fi"
    using less.prems by (simp add: firstPass_not_equiv_candidates_lt firstPass_preserves_ap
  moreover have "fi ≠ firstPass ps nu fi ⟹ all_pairs_are_first_candidates
(firstPass ps nu fi) ps"
    using less.prems by (rule firstPass_preserves_apac)
  ultimately have F: "fi ≠ firstPass ps nu fi ⟹ mkFirstMap' ps nu (firstPass
ps nu fi) ≠ None" by
    (simp add: less.hyps)
  then show ?case
  proof (cases "fi ≠ firstPass ps nu fi")
    case True
    then show ?thesis using F by (simp add: mkFirstMap'.simps[of ps
nu fi])
  next
    case False
    then show ?thesis by (simp add: mkFirstMap'.simps[of ps nu fi])
  qed
qed
```

lemma *empty_fi_apac: "all_pairs_are_first_candidates fmempty ps"*
unfolding *all_pairs_are_first_candidates_def pairsOf_def* **by** *auto*

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')))"*
unfolding *mkFirstMap_def*
by *(smt (verit, del_insts) mkFirstMap'.simps option.sel)*

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 *(cases s)*
case *(NT x)*
then show *?thesis* **using** *assms first_map_sound_def in_findOrEmpty_exists_set* **by** *fastforce*
next
case *(T x)*
then show *?thesis* **using** *assms(2) FirstT* **by** *fastforce*
qed

lemma *nullable_app: "nullable_gamma g xs ⟹ nullable_gamma g ys ⟹ nullable_gamma g (xs @ ys)"*
by *(induction xs; force elim: nullable_gamma.cases intro: NullableCons)*

lemma *nullableSym_nullable_sym:* **assumes** *"nullable_set_for nu g"*
shows *"nullableSym s nu ⟷ nullable_sym g s"*
proof *(cases s)*

```
  case (NT x1)
  then show ?thesis using assms nullable_set_sound_def nullable_set_complete_def
by fastforce
next
  case (T x2)
  then show ?thesis by (simp add: nullable_sym.simps)
qed

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 (induction gsuf arbitrary: gpre)
  case Nil
  then show ?case by auto
next
  case (Cons s syms)
  then show ?case
  proof (cases "nullableSym s nu")
    case True
    consider (la_in_firstSym) "la ∈ set (firstSym s fi)"
      | (la_in_firstGamma) "la ∈ set (firstGamma syms nu fi)"
      using Cons.prems(5) True by auto
    then show ?thesis
    proof (cases)
      case la_in_firstSym
      then show ?thesis using Cons.prems(2,3,4) FirstNT firstSym_first_sym
by fast
    next
      case la_in_firstGamma
      have "(x, (gpre @ [s]) @ syms) ∈ set (prods g)" using Cons.prems(3)
by auto
      moreover have "nullable_gamma g (gpre @ [s])"
      proof (rule nullable_app)
        show "nullable_gamma g gpre" by (simp add: Cons.prems(4))
      next
        have "nullable_sym g s" using Cons.prems(1) True nullableSym_nullable_sym
by auto
        then show "nullable_gamma g [s]" by - (rule NullableCons, auto
simp add: NullableNil)
      qed
      moreover have "la ∈ set (firstGamma syms nu fi)" by (simp add:
la_in_firstGamma)
      ultimately show ?thesis using Cons.prems(1,2) by - (rule Cons.IH[where
gpre = "gpre @ [s]"])
    qed
  next
    case False
    then show ?thesis using firstSym_first_sym Cons.prems(2,3,4,5) FirstNT
```

```
by fastforce
  qed
qed

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)"
  using NullableNil append.left_neutral firstGamma_first_sym' by force

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 (induction ps)
  case Nil
  then show ?case by (simp add: firstPass_def)
next
  case (Cons a suf)
  obtain x gamma where "a = (x, gamma)" by fastforce
  let ?fi' = "firstPass suf nu fi"
  let ?fg = "firstGamma gamma nu ?fi'"
  let ?xFirst = "findOrEmpty x ?fi'"
  let ?xFirst' = "?xFirst @@ ?fg"
  have fi'_sound: "first_map_sound ?fi' g" using Cons by auto
  show ?case
  proof (cases "?xFirst = ?xFirst'")
    case True
    then show ?thesis using ‹a = (x, gamma)› True fi'_sound by (auto
simp add: unfold_updateFi)
  next
    case False
    have "fmlookup (fmupd x (?xFirst @@ ?xFirst') ?fi') y = Some yFirst
⟹ la ∈ set yFirst
        ⟹ first_sym g la (NT y)" for y yFirst la
    proof (cases "x = y")
      case True
      assume "fmlookup (fmupd x (?xFirst @@ ?xFirst') ?fi') y = Some
yFirst" "la ∈ set yFirst"
      then consider (la_in_xFirst) "la ∈ set ?xFirst" | (la_in_fg) "la
∈ set ?fg"
        using ‹la ∈ set yFirst› True
        by auto
      then show ?thesis
      proof (cases)
        case la_in_xFirst
        then have "la ∈ set (firstSym (NT y) ?fi')" using True by auto
        then show ?thesis by - (rule firstSym_first_sym, auto simp add:
fi'_sound)
```

```
      next
        case la_in_fg
        have "(y, gamma) ∈ set (prods g)" using Cons.prems(3) True ‹a
= (x, gamma)› by auto
      then show ?thesis using fi'_sound Cons.prems(1) la_in_fg by
        - (rule firstGamma_first_sym[of nu g ?fi' y gamma], auto)
      qed
    next
      case False
      assume "fmlookup (fmupd x (?xFirst @@ ?xFirst') ?fi') y = Some
yFirst" "la ∈ set yFirst"
      then have "fmlookup ?fi' y = Some yFirst" by (simp add: False)
      then show ?thesis using ‹la ∈ set yFirst› fi'_sound first_map_sound_def
by fastforce
    qed
    then have "first_map_sound (fmupd x ?xFirst' ?fi') g" unfolding first_map_sound_def
by auto
    then show ?thesis using ‹a = (x, gamma)› False fi'_sound by (auto
simp add: unfold_updateFi)
  qed
qed

lemma firstPass_preserves_soundness: "nullable_set_for nu g ⟹ first_map_sound
fi g
    ⟹ first_map_sound (firstPass (prods g) nu fi) g"
  by (simp add: firstPass_preserves_soundness')

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 (induction "countFirstCands (prods g) fi" arbitrary: fi rule: less_induct)
  case less
  let ?fi' = "firstPass (prods g) nu fi"
  obtain fi'' where fi''_def: "mkFirstMap' (prods g) nu ?fi' = Some fi''"
    using firstPass_preserves_apac[of fi "prods g" nu] less.prems(3)
      mkFirstMap'_dom_if_apac[of "firstPass (prods g) nu fi"] by auto
  moreover have "first_map_sound (if fi = ?fi' then fi else fi'') g"
  proof (cases "fi = ?fi'")
    case True
    then show ?thesis using less.prems(2) by auto
  next
    case False
    have "countFirstCands (prods g) ?fi' < countFirstCands (prods g) fi"
using less.prems(3) False
      by (simp add: firstPass_not_equiv_candidates_lt firstPass_preserves_apac)
    moreover have "first_map_sound ?fi' g" using less.prems by
      - (rule firstPass_preserves_soundness, auto)
    ultimately show ?thesis using firstPass_preserves_apac less fi''_def
```

```
by fastforce
  qed
  ultimately show ?case by (auto simp add: mkFirstMap'.simps Let_def)
qed

lemma empty_fi_sound: "first_map_sound fmempty g"
  unfolding first_map_sound_def by auto

theorem mkFirstMap_sound: "nullable_set_for nu g ⟹ first_map_sound
(mkFirstMap g nu) g"
  unfolding mkFirstMap_def
  by (simp add: empty_fi_sound mkFirstMap'_preserves_soundness empty_fi_apac)
```

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 (induction gpre)
  case Nil
  then show ?case by simp
next
  case (Cons s gpre)
  from Cons.prems(2) have "nullable_sym g s" by (cases) auto
  then have "nullableSym s nu" using Cons.prems(1) nullableSym_nullable_sym
by blast
  from Cons.prems(2) have "nullable_gamma g gpre" by (cases) simp
  then have "LA y ∈ set (firstGamma (gpre @ T y # gsuf) nu fi)" using
Cons.IH Cons.prems(1) by auto
  then show ?case using ‹nullableSym s nu› by auto
qed

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 (induction gpre)
  case Nil
  then show ?case by (simp add: findOrEmpty_def)
next
  case (Cons s gpre)
  from Cons.prems(2) have "nullable_sym g s" by (cases) auto
  then have "nullableSym s nu" using Cons.prems(1) nullableSym_nullable_sym
by blast
  from Cons.prems(2) have "nullable_gamma g gpre" by (cases) simp
  then have "la ∈ set (firstGamma (gpre @ NT x # gsuf) nu fi)"
    using Cons.IH Cons.prems(1,3,4) by blast
  then show ?case using ‹nullableSym s nu› by auto
qed
```

```
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 (induction ps arbitrary: x)
  case Nil
  then show ?case unfolding firstPass_def by auto
next
  case (Cons p ps)
  then obtain y gamma where "p = (y, gamma)" by fastforce
  let ?fi' = "firstPass ps nu fi"
  let ?fg = "firstGamma gamma nu ?fi'"
  let ?yFirst = "findOrEmpty y ?fi'"
  let ?yFirst' = "?yFirst @@ ?fg"
  obtain xFirst' where "fmlookup ?fi' x = Some xFirst'" and "set xFirst
⊆ set xFirst'"
    using Cons by auto
  show ?case
  proof (cases "?yFirst = ?yFirst'")
    case True
    then have "fmlookup (firstPass (p # ps) nu fi) x = fmlookup ?fi'
x" by
      (simp add: ‹p = (y, gamma)› unfold_updateFi)
    then show ?thesis using ‹fmlookup ?fi' x = Some xFirst'› ‹set xFirst
⊆ set xFirst'› by simp
  next
    case False
    show ?thesis
    proof (cases "x = y")
      case True
      then have "fmlookup (firstPass (p # ps) nu fi) x = Some ?yFirst'"
using False by
        (auto simp add: ‹p = (y, gamma)› unfold_updateFi list_union_def)
      moreover have "set xFirst' ⊆ set ?yFirst'"
        using True ‹fmlookup ?fi' x = Some xFirst'› findOrEmpty_def in_findOrEmpty_exists_s
        by fastforce
      ultimately show ?thesis using ‹set xFirst ⊆ set xFirst'› by blast
    next
      case False
      then have "fmlookup (firstPass (p # ps) nu fi) x = fmlookup ?fi'
x"
        by (simp add: ‹p = (y, gamma)› unfold_updateFi)
      then show ?thesis using ‹fmlookup ?fi' x = Some xFirst'› ‹set xFirst
⊆ set xFirst'› by simp
    qed
  qed
qed

lemma firstPass_equiv_cons_tl: assumes "fi = firstPass (p # ps) nu fi"
```

```
  shows "fi = firstPass ps nu fi"
proof-
  obtain x gamma where "p = (x, gamma)" by fastforce
  let ?fi' = "firstPass ps nu fi"
  let ?fg = "firstGamma gamma nu ?fi'"
  let ?xFirst = "findOrEmpty x ?fi'"
  let ?xFirst' = "?xFirst @@ ?fg"
  show "fi = firstPass ps nu fi"
  proof (cases "?xFirst = ?xFirst'")
    case True
    then show ?thesis using True assms by (auto simp add: <p = (x, gamma)>
unfold_updateFi)
  next
    case False
    show ?thesis
    proof -
      have "fi = fmupd x ?xFirst' ?fi'" using False assms by
        (simp add: <p = (x, gamma)> unfold_updateFi list_union_def split:
if_splits)
      then have "fmlookup fi x = Some ?xFirst'" by (metis fmupd_lookup)
      have "firstPass ps nu fi = fi"
        by (metis assms firstPass_cons_subset firstPass_subset pairsOf_inj
subset_antisym)
      then have "firstGamma gamma nu (firstPass ps nu fi) = []"
        using <fmlookup fi x = Some ?xFirst'> False
        unfolding findOrEmpty_def by (auto split: option.splits)
      moreover have "firstGamma gamma nu (firstPass ps nu fi) ≠ []"
        by (metis False append_self_conv empty_iff filter_True list.set(1)
list_union_def)
      ultimately show "fi = firstPass ps nu fi" by auto
    qed
  qed
qed

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 (induction psuf arbitrary: ppre)
  case Nil
  then show ?case by auto
next
  case (Cons p psuf)
  obtain lx' gamma where "p = (lx', gamma)" by fastforce
  from Cons.prems(1) consider (prod_is_p) "(lx, gpre @ T y # gsuf) = p"
    | (prod_in_psuf) "(lx, gpre @ T y # gsuf) ∈ set psuf" by auto
  then show ?case
  proof (cases)
```

```
    case prod_is_p
    let ?fi' = "firstPass psuf nu fi"
    let ?fg = "firstGamma (gpre @ T y # gsuf) nu ?fi'"
    let ?lxFirst = "findOrEmpty lx ?fi'"
    let ?lxFirst' = "?lxFirst @@ ?fg"
    from Cons.prems(4) have " fi = firstPass ((lx, gpre @ T y # gsuf)
# psuf) nu fi"
      using prod_is_p by blast
    then consider (same) "?lxFirst = ?lxFirst'" "fi = firstPass psuf nu
fi"
      | (new) "?lxFirst ≠ ?lxFirst'" "fi = fmupd lx ?lxFirst' ?fi'"
      by (metis Nil_is_append_conv firstPass_cons list_union_def unfold_updateFi)
    then show ?thesis
    proof (cases)
      case same
      have "LA y ∈ set ?fg" using Cons.prems(2,3) by - (rule la_in_firstGamma_t,
auto)
      then have "LA y ∈ set ?lxFirst" using same(1) by (auto intro:
mem_list_union)
      then have "LA y ∈ set (findOrEmpty lx fi)" using same(2) by auto
      then show ?thesis by (simp add: in_findOrEmpty_exists_set)
    next
      case new
      from new(2) obtain lxFirst where "fmlookup fi lx = Some lxFirst"
and "?lxFirst' = lxFirst" by
        (metis fmupd_lookup)
      then have "LA y ∈ set lxFirst" using la_in_firstGamma_t Cons.prems(2,3)
by fastforce
      then show ?thesis using ‹fmlookup fi lx = Some lxFirst› by simp
    qed
  next
    case prod_in_psuf
    then have "(lx, gpre @ T y # gsuf) ∈ set psuf" by auto
    moreover have "fi = firstPass psuf nu fi" using Cons.prems(4) firstPass_equiv_cons_tl
by blast
    moreover have "prods g = (ppre @ [p]) @ psuf" by (simp add: Cons.prems(5))
    ultimately show ?thesis using Cons.prems(2,3) by
      - (rule Cons.IH[where ppre = "ppre @ [p]"], auto)
  qed
qed

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"
  by (simp add: firstPass_equiv_right_t')

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 (induction psuf arbitrary: ppre)
  case Nil
  then show ?case by auto
next
  case (Cons p psuf)
  obtain lx' gamma where "p = (lx', gamma)" by fastforce
  from Cons.prems(1) consider (prod_is_p) "(lx, gpre @ NT rx # gsuf) =
p"
    | (prod_in_psuf) "(lx, gpre @ NT rx # gsuf) ∈ set psuf" using Cons.prems(3)
by auto
  then show ?case
  proof (cases)
    case prod_is_p
    let ?fi' = "firstPass psuf nu fi"
    let ?fg = "firstGamma (gpre @ NT rx # gsuf) nu ?fi'"
    let ?lxFirst = "findOrEmpty lx ?fi'"
    let ?lxFirst' = "?lxFirst @@ ?fg"
    from Cons.prems(2) have "fi = firstPass ((lx, gpre @ NT rx # gsuf)
# psuf) nu fi"
      using prod_is_p by blast
    then consider (same) "?lxFirst = ?lxFirst'" "fi = firstPass psuf nu
fi"
      | (new) "?lxFirst ≠ ?lxFirst'" "fi = fmupd lx ?lxFirst' ?fi'"
      using firstPass_cons la_in_firstGamma_nt[OF Cons.prems(1,4-6)]
        unfold_updateFi[where gamma = "gpre @ NT rx # gsuf"]
      unfolding list_union_def
      by (auto split: if_splits)
    then show ?thesis
    proof (cases)
      case same
      then have "la ∈ set (findOrEmpty lx fi)" using Cons.prems(1,4,5,6)
la_in_firstGamma_nt by
          (metis mem_list_union)
      then show ?thesis by (simp add: in_findOrEmpty_exists_set)
    next
      case new
      have "la ∈ set ?lxFirst'"
      proof (rule list_union_I2)
        from Cons.prems(2) have "fi = firstPass psuf nu fi" by (rule
firstPass_equiv_cons_tl)
        then have "fmlookup (firstPass psuf nu fi) rx = Some rxFirst"
using Cons.prems(5) by auto
        then show "la ∈ set ?fg"
          using Cons.prems(1,4,6) ‹fi = firstPass ((lx, gpre @ NT rx #
gsuf) # psuf) nu fi›
```

```
        by - (rule la_in_firstGamma_nt[where xFirst = "rxFirst"])
      qed
      then show ?thesis by (metis fmupd_lookup new(2))
    qed
  next
    case prod_in_psuf
    then have "(lx, gpre @ NT rx # gsuf) ∈ set psuf" by auto
    moreover have "fi = firstPass psuf nu fi" using Cons.prems(2) firstPass_equiv_cons_tl
by blast
    moreover have "prods g = (ppre @ [p]) @ psuf" by (simp add: Cons.prems(7))
    ultimately show ?thesis using Cons.prems(1,4,5,6,7) prod_in_psuf
by
      - (rule Cons.IH[where ppre = "ppre @ [p]"], auto)
  qed
qed

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"
  by (simp add: firstPass_equiv_right_nt')

lemma firstPass_equiv_complete: assumes "nullable_set_for nu g" "fi
= firstPass (prods g) nu fi"
  shows "first_map_complete fi g"
proof -
  have "first_sym g la s
    ⟹ (∀x. s = NT x ⟶ (∃ xFirst. fmlookup fi x = Some xFirst ∧ la
∈ set xFirst))" for la s
  proof (induction rule: first_sym.induct)
    case (FirstT)
    then show ?case by blast
  next
    case (FirstNT lx gpre s gsuf la)
    then show ?case
    proof (cases s)
      case (NT rx)
      then obtain rxFirst where "fmlookup fi rx = Some rxFirst ∧ la
∈ set rxFirst"
        using FirstNT.IH by auto
      then show ?thesis using FirstNT.hyps(1,2) NT assms firstPass_equiv_right_nt
by fastforce
    next
      case (T y)
      from FirstNT.hyps(3) have "la = LA y" by (cases) (auto simp add:
T)
      then show ?thesis using firstPass_equiv_right_t using FirstNT.hyps(1,2)
```

```
T assms by fastforce
    qed
  qed
  then show ?thesis by (simp add: first_map_complete_def)
qed

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 (induction "countFirstCands (prods g) fi" arbitrary: fi rule: less_induct)
  case less
  let ?fi' = "firstPass (prods g) nu fi"
  obtain fi' where fi'_def: "mkFirstMap' (prods g) nu ?fi' = Some fi'"
    by (meson firstPass_preserves_apac less.prems(2) mkFirstMap'_dom_if_apac
not_Some_eq)
  moreover have "first_map_complete (if fi = ?fi' then fi else fi') g"
  proof (cases "fi = ?fi'")
    case True
    then show ?thesis using firstPass_equiv_complete less.prems(1) by
auto
  next
    case False
    have "countFirstCands (prods g) ?fi' < countFirstCands (prods g) fi"
by
      (simp add: False firstPass_not_equiv_candidates_lt firstPass_preserves_apac
less.prems(2))
    moreover have "nullable_set_for nu g" by (simp add: less.prems(1))
    moreover have "all_pairs_are_first_candidates ?fi' (prods g)" by
      (simp add: firstPass_preserves_apac less.prems(2))
    ultimately show ?thesis
      using fi'_def less.hyps by force
  qed
  ultimately show ?case by (auto simp add: mkFirstMap'.simps Let_def)
qed

theorem mkFirstMap_complete: "nullable_set_for nu g ⟹ first_map_complete
(mkFirstMap g nu) g"
  unfolding mkFirstMap_def
  using empty_fi_apac mkFirstMap'_complete by fastforce

theorem mkFirstMap_correct: "nullable_set_for nu g ⟹ first_map_for
(mkFirstMap g nu) g"
  using mkFirstMap_sound mkFirstMap_complete
  by fastforce

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)"
  unfolding followPass_def by auto

lemma medial_t_in_lookaheadsOf:
  "(x, gpre @ (T y) # gsuf) ∈ set (prods g) ⟹ (LA y) ∈ lookaheadsOf
g"
proof -
  assume A: "(x, gpre @ T y # gsuf) ∈ set (prods g)"
  have "LA y ∈ lookaheadsOfGamma (gpre @ T y # gsuf)"
  proof (induction gpre)
    case Nil
    then show ?case by auto
  next
    case (Cons a gpre)
    then show ?case by (cases a) (auto simp add: Cons.IH)
  qed
  then have "LA y ∈ ⋃ (lookaheadsOfGamma ` snd ` set (prods g))" us-
ing A image_def by fastforce
  then show "(LA y) ∈ lookaheadsOf g" unfolding lookaheadsOf_def by
auto
qed
```

```
lemma first_sym_in_lookaheadsOf: "first_sym g la s ⟹ s = NT x ⟹
la ∈ lookaheadsOf g"
proof (induction arbitrary: x rule: first_sym.induct)
  case (FirstT y)
  then show ?case by auto
next
  case (FirstNT x' gpre s gsuf la)
  show ?case
  proof (cases s)
    case (NT _)
    then show ?thesis using FirstNT.IH by auto
  next
    case (T y)
    from FirstNT.hyps(3) have "la = LA y" using T by (auto elim: first_sym.cases)
    have "(x', gpre @ (T y) # gsuf) ∈ set (prods g)" using FirstNT.hyps(1)
T by auto
    then have "(LA y) ∈ lookaheadsOf g" by - (rule medial_t_in_lookaheadsOf,
auto)
    then show ?thesis using ‹la = LA y› by auto
  qed
qed

lemma first_map_la_in_lookaheadsOf:
  "first_map_for fi g ⟹ fmlookup fi x = Some s ⟹ la ∈ set s ⟹ la
∈ lookaheadsOf g"
  unfolding first_map_sound_def by (rule first_sym_in_lookaheadsOf[where
s = "NT x"], auto)

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 (induction gsuf arbitrary: gpre)
  case Nil
  then show ?case by auto
next
  case (Cons s gsuf)
  have "la ∈ set (if nullableSym s nu then firstSym s fi @@ firstGamma
gsuf nu fi
      else firstSym s fi)"
    using Cons.prems(3) by auto
  then consider (la_in_firstSym) "la ∈ set (firstSym s fi)"
    | (la_in_firstGamma) "la ∈ set (firstGamma gsuf nu fi)"
    by (auto split: if_splits)
  then show ?case
  proof cases
    case la_in_firstSym
    then show ?thesis
    proof (cases s)
```

```
      case (NT x)
      then obtain s where "fmlookup fi x = Some s" and "la ∈ set s"
        using in_findOrEmpty_exists_set la_in_firstSym by fastforce
      with Cons.prems(1) show ?thesis by (rule first_map_la_in_lookaheadsOf[where
?s = "s"])
    next
      case (T x)
      then show ?thesis
        using Cons.prems(2) la_in_firstSym medial_t_in_lookaheadsOf by
fastforce
    qed
  next
    case la_in_firstGamma
    then show ?thesis using Cons.prems by - (rule Cons.IH[where ?gpre
= "gpre @ [s]"], auto)
  qed
qed

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 -
  assume "fmlookup fo x = Some xFollow" "la ∈ set xFollow" "all_pairs_are_follow_candidate
fo g"
  then have "la ∈ set (findOrEmpty x fo)" by (simp add: findOrEmpty_def)
  then have "(x, la) ∈ pairsOf fo" by (simp add: in_findOrEmpty_iff_in_pairsOf)
  with ‹all_pairs_are_follow_candidates fo g› show ?thesis
    unfolding all_pairs_are_follow_candidates_def by auto
qed

lemma medial_nt_in_ntsOfGamma: "x ∈ ntsOfGamma (gpre @ (NT x) # gsuf)"
proof (induction gpre)
  case Nil
  then show ?case by auto
next
  case (Cons a gpre)
  then show ?case
  proof (cases a)
    case (NT y)
    then show ?thesis unfolding ntsOf_def by (simp add: Cons.IH)
  next
    case (T _)
    then show ?thesis by (simp add: Cons.IH)
  qed
qed

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

```
  case Nil
  then show ?case by auto
next
  case (Cons a x)
  then show ?case unfolding ntsOf_def using medial_nt_in_ntsOfGamma by
fastforce
qed

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"
  unfolding additions_def
proof (rule updateFo.induct[where ?P = "P"])
  fix nu fi lx fo rx gamma'
  assume "P nu fi lx gamma' fo"
  let ?fo' = "updateFo nu fi lx gamma' fo"
```

```
  consider "fmlookup ?fo' rx = None" "additions nu fi lx gamma' fo = []"
    | "fmlookup ?fo' rx = None" "additions nu fi lx gamma' fo ≠ []"
    | rxFollow where "fmlookup ?fo' rx = Some rxFollow"
        and "set (additions nu fi lx gamma' fo) ⊆ set rxFollow"
      | rxFollow where "fmlookup ?fo' rx = Some rxFollow"
        and "¬ set (additions nu fi lx gamma' fo) ⊆ set rxFollow"
    by blast
  then show "P nu fi lx (NT rx # gamma') fo" by
    (cases) (auto simp add: ‹P nu fi lx gamma' fo› assms)
qed (auto simp add: Nil T)

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
  show "rx ∈ ntsOf g" by (meson assms(3) medial_nt_in_ntsOf)
next
  from assms(4) consider
      (la_in_findOrEmpty) "la ∈ set (findOrEmpty lx (updateFo nu fi lx
gamma' fo))"
      | (la_in_firstGamma) "la ∈ set (firstGamma gamma' nu fi)"
    by (auto split: if_splits)
  then show "la ∈ lookaheadsOf g"
  proof (cases)
    case la_in_findOrEmpty
    then obtain lxFollow where "fmlookup (updateFo nu fi lx gamma' fo)
lx = Some lxFollow"
      "la ∈ set lxFollow" using in_findOrEmpty_exists_set assms(3) by
fast
    then show ?thesis by (auto intro: la_in_fo_in_lookaheadsOf simp add:
assms(2))
  next
    case la_in_firstGamma
    with assms(1,3) show ?thesis by
      (auto intro: in_firstGamma_in_lookaheadsOf[where gpre = "gpre @
[NT rx]"])
  qed
qed

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 (induction nu fi lx gsuf fo arbitrary: gpre rule: updateFo_induct_refined)
```

```
  case (1 nu fi lx fo)
  then show ?case by simp
next
  case (2 nu fi lx y gamma' fo)
  from 2(2-) show ?case by (auto intro: 2(1)[where gpre = "gpre @ [T
y]"])
next
  case (3 nu fi lx rx gamma' fo)
  from 3(2-) have "all_pairs_are_follow_candidates (updateFo nu fi lx
gamma' fo) g" by
    (auto intro: 3(1)[where gpre = "gpre @ [NT rx]"])
  then show ?case by (simp add: "3.hyps")
next
  case (4 nu fi lx rx gamma' fo)
  let ?fo' = "updateFo nu fi lx gamma' fo"
  let ?lSet = "findOrEmpty lx ?fo'"
  let ?rSet = "firstGamma gamma' nu fi"
  let ?additions = "if nullableGamma gamma' nu then ?lSet @@ ?rSet else
?rSet"
  have IH: "all_pairs_are_follow_candidates ?fo' g" by
    (auto intro: "4.IH"[where ?gpre = "gpre @ [NT rx]"] simp add: "4.prems")
  have "x ∈ ntsOf g ∧ la ∈ lookaheadsOf g"
    if "(x ,la) ∈ pairsOf (fmupd rx ?additions ?fo')" for x la
  proof (cases "x = rx")
    case True
    then have "(lx, gpre @ NT x # gamma') ∈ set (prods g)" by (auto simp
add: "4.prems"(2))
    moreover have "la ∈ set ?additions" using that by (simp only: True
in_add_keys[symmetric])
    ultimately show ?thesis using "4.prems"(1,2) IH by
      - (rule updateFo_preserves_apac_fmupd_additions)
  next
    case False
    then have "(x, la) ∈ pairsOf ?fo'" by (metis in_add_keys_neq that)
    then show ?thesis using IH all_pairs_are_follow_candidates_def by
fastforce
  qed
  then show ?case unfolding all_pairs_are_follow_candidates_def by (auto
simp add: 4 Let_def)
next
  case (5 nu fi lx rx gamma' fo rxFollow)
  from 5(2-) have "all_pairs_are_follow_candidates (updateFo nu fi lx
gamma' fo) g" by
    (auto intro: 5(1)[where gpre = "gpre @ [NT rx]"])
  then show ?case by (simp add: "5.hyps")
next
  case (6 nu fi lx rx gamma' fo rxFollow)
  let ?fo' = "updateFo nu fi lx gamma' fo"
  let ?lSet = "findOrEmpty lx ?fo'"
```

```
  let ?rSet = "firstGamma gamma' nu fi"
  let ?additions = "if nullableGamma gamma' nu then ?lSet @@ ?rSet else
?rSet"
  have IH: "all_pairs_are_follow_candidates ?fo' g"
    by (auto intro: "6.IH"[where ?gpre = "gpre @ [NT rx]"] simp add:
"6.prems")
  have "x ∈ ntsOf g ∧ la ∈ lookaheadsOf g"
    if "(x ,la) ∈ pairsOf (fmupd rx (rxFollow @@ ?additions) ?fo')" for
x la
  proof (cases "x = rx")
    case True
    from that have "la ∈ set (rxFollow @@ ?additions)" by (simp only:
True in_add_keys[symmetric])
    then consider (la_in_rxFollow) "la ∈ set rxFollow" | (la_in_additions)
"la ∈ set ?additions" by
        auto
    then show ?thesis
    proof (cases)
      case la_in_rxFollow
      then show ?thesis
        using "6.hyps"(1) "6.prems"(2) IH True la_in_fo_in_lookaheadsOf
medial_nt_in_ntsOf
        by fastforce
    next
      case la_in_additions
      then show ?thesis using "6.prems"(1,2) IH True updateFo_preserves_apac_fmupd_additio
by
        fastforce
    qed
  next
    case False
    then have "(x, la) ∈ pairsOf ?fo'" by (metis in_add_keys_neq that)
    then show ?thesis using IH all_pairs_are_follow_candidates_def by
fastforce
  qed
  then show ?case unfolding all_pairs_are_follow_candidates_def by (auto
simp add: 6 Let_def)
qed

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 (induction suf arbitrary: pre)
  case Nil
  show ?case unfolding followPass_def by (simp add: Nil.prems(3))
next
  case (Cons a suf)
  obtain x gamma where "a = (x, gamma)" by fastforce
```

```
  have IH: "all_pairs_are_follow_candidates (followPass suf nu fi fo)
g"
    using Cons.IH[where pre = "pre @ [a]"] Cons.prems by auto
  have "(x, gamma) ∈ set (prods g)" by
    (metis ‹a = (x, gamma)›[symmetric] in_set_conv_decomp Cons.prems(2))
  have "all_pairs_are_follow_candidates (updateFo nu fi x gamma (followPass
suf nu fi fo)) g"
    using Cons.prems(1) ‹(x, gamma) ∈ set (prods g)› IH
    by - (rule updateFo_preserves_apac[where gpre = "[]"], auto)
  then show ?case by (simp add: ‹a = (x, gamma)›)
qed

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"
  by (rule followPass_preserves_apac'[where pre = "[]"]) auto

lemma updateFo_subset: "pairsOf fo ⊆ pairsOf (updateFo nu fi x' gamma
fo)"
proof (induction nu fi x' gamma fo rule: updateFo_induct_refined)
  case (4 nu fi lx rx gamma' fo)
  let ?fo' = "updateFo nu fi lx gamma' fo"
  let ?lSet = "findOrEmpty lx ?fo'"
  let ?rSet = "firstGamma gamma' nu fi"
  let ?additions = "(if nullableGamma gamma' nu then ?lSet @@ ?rSet else
?rSet)"
  have "(x, la) ∈ pairsOf (fmupd rx ?additions ?fo')" if "(x, la) ∈ pairsOf
fo" for x la
  proof (cases "x = rx")
    case True
    then show ?thesis using "4.IH" "4.hyps"(1) True that by
      (fastforce simp add: in_pairsOf_exists)
  next
    case False
    have "(x, la) ∈ pairsOf ?fo'" using "4.IH" that by auto
    then show ?thesis using in_add_keys_neq[where ?fi = "?fo'"] False
by auto
  qed
  then show ?case by (auto simp add: Let_def 4)
next
  case (6 nu fi lx rx gamma' fo rxFollow)
  let ?fo' = "updateFo nu fi lx gamma' fo"
  let ?lSet = "findOrEmpty lx ?fo'"
  let ?rSet = "firstGamma gamma' nu fi"
  let ?additions = "(if nullableGamma gamma' nu then ?lSet @@ ?rSet else
?rSet)"
  have "(x, la) ∈ pairsOf (fmupd rx (rxFollow @@ ?additions) ?fo')"
    if "(x, la) ∈ pairsOf fo" for x la
```

```
  proof (cases "x = rx")
    case True
    have "la ∈ set (rxFollow @@ ?additions)" using "6.IH" "6.hyps"(1)
True that by
      (fastforce simp add: in_pairsOf_exists)
    then show ?thesis using in_add_keys[where ?fi = "?fo'"] True by
auto
  next
    case False
    have "(x, la) ∈ pairsOf ?fo'" using "6.IH" that by auto
    then show ?thesis using in_add_keys_neq[where ?fi = "?fo'"] False
by auto
  qed
  then show ?case by (auto simp add: Let_def 6)
qed auto

lemma followPass_subset: "pairsOf fo ⊆ pairsOf (followPass ps nu fi
fo)"
proof (induction ps)
  case Nil
  then show ?case by (simp add: followPass_def)
next
  case (Cons p ps)
  obtain x gamma where "p = (x, gamma)" by fastforce
  have "pairsOf (followPass ps nu fi fo)
    ⊆ pairsOf (updateFo nu fi x gamma (followPass ps nu fi fo))" using
updateFo_subset by fast
  then have "pairsOf fo ⊆ pairsOf (updateFo nu fi x gamma (followPass
ps nu fi fo))"
    using Cons.IH by blast
  then show ?case unfolding followPass_def by (simp add: ‹p = (x, gamma)›)
qed

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 (induction nu fi lx gsuf fo arbitrary: gpre rule: updateFo_induct_refined)
  case (1 nu fi lx fo)
  then show ?case by simp
next
  case (2 nu fi lx y gsuf fo)
  have "∃x' la. x' ∈ ntsOf g ∧ la ∈ lookaheadsOf g ∧ (x', la) ∉ pairsOf
fo
    ∧ (x', la) ∈ pairsOf (updateFo nu fi lx gsuf fo)" using "2.prems"
    by - (rule "2.IH"[where ?gpre = "gpre @ [T y]"], auto)
```

```
  then show ?case by (simp only: updateFo.simps)
next
  case (3 nu fi lx rx gsuf fo)
  from "3.prems"(4) have "fo ≠ updateFo nu fi lx gsuf fo" by (simp add:
"3.hyps"(1) "3.hyps"(2))
  then have "∃x' la. x' ∈ ntsOf g ∧ la ∈ lookaheadsOf g ∧ (x', la)
∉ pairsOf fo
    ∧ (x', la) ∈ pairsOf (updateFo nu fi lx gsuf fo)" using "3.prems"(1,2,3)
    by - (rule "3.IH"[where ?gpre = "gpre @ [NT rx]"], auto)
  then show ?case by (simp add: "3.hyps"(1) "3.hyps"(2))
next
  case (4 nu fi lx rx gsuf fo)
  let ?fo' = "updateFo nu fi lx gsuf fo"
  let ?lSet = "findOrEmpty lx ?fo'"
  let ?rSet = "firstGamma gsuf nu fi"
  let ?additions = "(if nullableGamma gsuf nu then ?lSet @@ ?rSet else
?rSet)"
  obtain la where "la ∈ set ?additions" using "4.hyps"(2) list.set_sel(1)
by auto
  have "rx ∈ ntsOf g ∧ la ∈ lookaheadsOf g" "(rx, la) ∉ pairsOf fo"
    "(rx, la) ∈ pairsOf (updateFo nu fi lx (NT rx # gsuf) fo)"
  proof -
    have "all_pairs_are_follow_candidates ?fo' g" using "4.prems"(1,2,3)
by
      (auto intro: updateFo_preserves_apac[where ?gpre = "gpre @ [NT
rx]"])
    then show "rx ∈ ntsOf g ∧ la ∈ lookaheadsOf g" using "4.prems"(1,2)
‹la ∈ set ?additions› by
      - (rule updateFo_preserves_apac_fmupd_additions)
  next
    have "(rx, la) ∉ pairsOf ?fo'" using "4.hyps"(1) by (fastforce simp
add: in_pairsOf_exists)
    then show "(rx, la) ∉ pairsOf fo" using updateFo_subset by fastforce
  next
    have "(rx, la) ∈ pairsOf (fmupd rx ?additions ?fo')" using ‹la ∈
set ?additions› in_add_keys by
      fast
    then show "(rx, la) ∈ pairsOf (updateFo nu fi lx (NT rx # gsuf) fo)"
by (simp add: 4 Let_def)
  qed
  then show ?case by auto
next
  case (5 nu fi lx rx gsuf fo rxFollow)
  from "5.prems"(4) have "fo ≠ updateFo nu fi lx gsuf fo" by (simp add:
"5.hyps"(1) "5.hyps"(2))
  then have "∃x' la. x' ∈ ntsOf g ∧ la ∈ lookaheadsOf g ∧ (x', la)
∉ pairsOf fo
    ∧ (x', la) ∈ pairsOf (updateFo nu fi lx gsuf fo)"
    using "5.prems"(1,2,3) by - (rule "5.IH"[where ?gpre = "gpre @ [NT
```

```
rx]"], auto)
  then show ?case by (simp add: "5.hyps"(1,2))
next
  case (6 nu fi lx rx gsuf fo rxFollow)
  let ?fo' = "updateFo nu fi lx gsuf fo"
  let ?lSet = "findOrEmpty lx ?fo'"
  let ?rSet = "firstGamma gsuf nu fi"
  let ?additions = "(if nullableGamma gsuf nu then ?lSet @@ ?rSet else
?rSet)"
  obtain la where "la ∈ set ?additions" "la ∉ set rxFollow" using "6.hyps"(2)
by auto
  have "rx ∈ ntsOf g ∧ la ∈ lookaheadsOf g" "(rx, la) ∉ pairsOf fo"
    "(rx, la) ∈ pairsOf (updateFo nu fi lx (NT rx # gsuf) fo)"
  proof -
    have "all_pairs_are_follow_candidates ?fo' g" using "6.prems"(1,2,3)
by
      (auto intro: updateFo_preserves_apac[where ?gpre = "gpre @ [NT
rx]"])
    then show "rx ∈ ntsOf g ∧ la ∈ lookaheadsOf g" using "6.prems"(1,2)
‹la ∈ set ?additions› by
      - (rule updateFo_preserves_apac_fmupd_additions)
  next
    have "(rx, la) ∉ pairsOf ?fo'" using ‹la ∉ set rxFollow› "6.hyps"(1)
by
      (fastforce simp add: in_pairsOf_exists)
    then show "(rx, la) ∉ pairsOf fo" using updateFo_subset by fastforce
  next
    have "la ∈ set (rxFollow @@ ?additions)" using ‹la ∈ set ?additions›
by simp
    then have "(rx, la) ∈ pairsOf (fmupd rx (rxFollow @@ ?additions)
?fo')"
      using ‹la ∈ set ?additions› in_add_keys by fast
    then show "(rx, la) ∈ pairsOf (updateFo nu fi lx (NT rx # gsuf) fo)"
by (simp add: 6 Let_def)
  qed
  then show ?case by auto
qed

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)"
  by (rule updateFo_not_equiv_exists'[where gpre = "[]"]) auto

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 (induction suf arbitrary: pre fo)
  case Nil
  have "fo = followPass [] nu fi fo" by (simp add: followPass_def)
  then show ?case using Nil.prems(4) by blast
next
  case (Cons a suf)
  obtain x gamma where "a = (x, gamma)" by fastforce
  show ?case
  proof (cases "fo ≠ followPass suf nu fi fo")
    case True
    then obtain x' la where "x' ∈ ntsOf g" "la ∈ lookaheadsOf g" "(x',
la) ∉ pairsOf fo"
      "(x', la) ∈ pairsOf (followPass suf nu fi fo)"
      using Cons.IH[of fo "pre @ [a]"] Cons.prems(1,2,3) by auto
    moreover have "(x', la) ∈ pairsOf (followPass (a # suf) nu fi fo)"
      using updateFo_subset ‹(x', la) ∈ pairsOf (followPass suf nu fi
fo)›
      by (simp add: ‹a = (x, gamma)›) fast
    ultimately show ?thesis by blast
  next
    case False
    have A2: "(x, gamma) ∈ set (prods g)" by
      (metis ‹a = (x, gamma)› in_set_conv_decomp Cons.prems(3))
    have A3: "all_pairs_are_follow_candidates (followPass suf nu fi fo)
g" using Cons.prems by
      - (rule followPass_preserves_apac'[where pre = "pre @ [a]"], auto)
    then have "followPass suf nu fi fo ≠ followPass (a # suf) nu fi fo"
using False by
      (auto simp add: Cons.prems(4))
    then have A4: "followPass suf nu fi fo ≠ updateFo nu fi x gamma (followPass
suf nu fi fo)"
      using False by (simp add: ‹a = (x, gamma)›)
    have "(∃x' la. x' ∈ ntsOf g ∧ la ∈ lookaheadsOf g
        ∧ (x', la) ∉ pairsOf (followPass suf nu fi fo)
        ∧ (x', la) ∈ pairsOf (updateFo nu fi x gamma (followPass suf nu
fi fo)))"
      using Cons.prems(1) A2 A3 A4 by - (rule updateFo_not_equiv_exists,
auto)
    then show ?thesis using False by (auto simp add: ‹a = (x, gamma)›)
  qed
qed

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)"
  using followPass_equiv_or_exists' by fastforce

lemma finite_ntsOfGamma: "finite (ntsOfGamma gamma)"
proof (induction gamma)
  case Nil
  then show ?case by auto
next
  case (Cons a gamma)
  then show ?case by (cases a) auto
qed

lemma finite_ntsOf: "finite (ntsOf g)"
  unfolding ntsOf_def by (simp add: finite_ntsOfGamma)

lemma finite_lookaheadsOfGamma: "finite (lookaheadsOfGamma gamma)"
proof (induction gamma)
  case Nil
  then show ?case by auto
next
  case (Cons a gamma)
  then show ?case by (cases a) auto
qed

lemma finite_lookaheadsOf: "finite (lookaheadsOf g)"
  unfolding lookaheadsOf_def by (simp add: finite_lookaheadsOfGamma)

lemma finite_allCandidates_follow: "finite (ntsOf g × lookaheadsOf g)"
  using finite_lookaheadsOf finite_ntsOf by auto

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"
  unfolding countFollowCands_def Let_def
proof (rule psubset_card_mono)
  show "finite (ntsOf g × lookaheadsOf g - pairsOf fo)" using finite_allCandidates_follow
by auto
next
  assume "first_map_for fi g" "all_pairs_are_follow_candidates fo g"
    "fo ≠ (followPass (prods g) nu fi fo)"
  then obtain x la where "x ∈ ntsOf g ∧ la ∈ lookaheadsOf g ∧ (x, la)
∉ pairsOf fo
    ∧ (x, la) ∈ pairsOf (followPass (prods g) nu fi fo)" using followPass_not_equiv_exists
by blast
  then show "ntsOf g × lookaheadsOf g - pairsOf (followPass (prods g)
```

```
nu fi fo)
    ⊂ ntsOf g × lookaheadsOf g - pairsOf fo" using followPass_subset
by fastforce
qed
```

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"
  using that
proof (induction "(g, nu, fi, fo)" arbitrary: fi fo
    rule: measure_induct_rule[where f = "λ(g, nu, fi, fo). countFollowCands
g fo"])
  case (less fi fo)
  have "fo ≠ followPass (prods g) nu fi fo
    ⟹ countFollowCands g (followPass (prods g) nu fi fo)  < countFollowCands
g fo"
    using less.prems by (simp add: followPass_not_equiv_candidates_lt)
  moreover have "fo ≠ followPass (prods g) nu fi fo
    ⟹ all_pairs_are_follow_candidates (followPass (prods g) nu fi fo)
g" using less.prems
    by - (rule followPass_preserves_apac)
  ultimately have "fo ≠ followPass (prods g) nu fi fo
    ⟹ mkFollowMap' g nu fi (followPass (prods g) nu fi fo) ≠ None" by
(simp add: less)
  then show ?case
    by (cases "fo ≠ followPass (prods g) nu fi fo") (auto simp add: mkFollowMap'.simps)
qed

lemma initial_fo_apac: "all_pairs_are_follow_candidates (initial_fo g)
g"
  unfolding all_pairs_are_follow_candidates_def
proof
  fix a
  assume A: "a ∈ pairsOf (initial_fo g)"
  show "case a of (x, la) ⇒ x ∈ ntsOf g ∧ la ∈ lookaheadsOf g"
  proof
    fix x la
    assume "a = (x, la)"
    show "x ∈ ntsOf g ∧ la ∈ lookaheadsOf g"
    proof (cases "x = start g")
      case True
      have "la = EOF" using A True ‹a = (x, la)› by (fastforce simp add:
in_add_value)
      then show ?thesis unfolding ntsOf_def lookaheadsOf_def by (simp
add: ‹x = start g›)
    next
      case False
```

```
    then show ?thesis using A ‹a = (x, la)› by (fastforce simp add:
in_pairsOf_exists)
  qed
 qed
qed
```

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 -
  assume "nullable_sym g s" "nullable_gamma g gpre" "first_gamma g la
gsuf"
  obtain s' gpre' gsuf' where "nullable_gamma g gpre'" "first_sym g la
s'"
    "gsuf = gpre' @ s' # gsuf'" using ‹first_gamma g la gsuf› by cases
blast
  have "nullable_gamma g [s]" using ‹nullable_sym g s› by (simp add:
NullableCons NullableNil)
  then have "nullable_gamma g ((gpre @ [s]) @ gpre')"
    using ‹nullable_gamma g gpre'› ‹nullable_gamma g gpre› nullable_app
by blast
  then have "first_gamma g la ((gpre @ s # gpre') @ s' # gsuf')" us-
ing ‹first_sym g la s'› by
    - (rule FirstGamma, auto)
  then show "first_gamma g la (gpre @ s # gsuf)" by (simp add: ‹gsuf
= gpre' @ s' # gsuf'›)
qed

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 (induction gamma)
  case Nil
  then show ?case by (auto elim: first_gamma.cases)
next
  case (Cons s gamma)
  consider (la_in_firstSym) "la ∈ set (firstSym s fi)"
    | (la_in_firstGamma) "nullableSym s nu" "la ∈ set (firstGamma gamma
nu fi)"
    using Cons.prems(3)
    by (metis firstGamma.simps(2) in_atleast1_list)
  then show ?case
  proof (cases)
    case la_in_firstSym
    have "first_sym g la s" using Cons.prems(2) la_in_firstSym by
      - (rule firstSym_first_sym[where fi = "fi"], auto)
    then show ?thesis using FirstGamma NullableNil by fastforce
  next
    case la_in_firstGamma
    then have "first_gamma g la gamma" using Cons.prems by - (rule Cons.IH)
    moreover have "nullable_sym g s"
      using nullableSym_nullable_sym Cons.prems(1) la_in_firstGamma(1)
by blast
    ultimately have "first_gamma g la ([] @ s # gamma)" using NullableNil
by
      - (rule first_gamma_tail_cons)
    then show ?thesis by auto
  qed
qed

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 (induction gamma)
  case Nil
  then show ?case by (auto elim: first_gamma.cases)
next
  case (Cons s gamma)
  from Cons.prems(3) obtain s' gpre gsuf where "nullable_gamma g gpre"
"first_sym g la s'"
    "gpre @ s' # gsuf = s # gamma" by (auto elim: first_gamma.cases)
  show ?case
  proof (cases gpre)
    case Nil
    then have "s = s'" "gsuf = gamma" using ‹gpre @ s' # gsuf = s # gamma›
by auto
    then have "first_sym g la s" using ‹first_sym g la s'› by auto
    show ?thesis
    proof (cases s)
```

```
      case (NT x)
      from Cons.prems(2) obtain xFirst where "fmlookup fi x = Some xFirst"
"la ∈ set xFirst"
        unfolding first_map_complete_def using ‹first_sym g la s› NT by
fast
      then have "la ∈ set (firstGamma (gpre @ NT x # gsuf) nu fi)"
        using Cons.prems ‹nullable_gamma g gpre› by - (rule la_in_firstGamma_nt)
      then show ?thesis by (simp add: NT ‹gsuf = gamma› Nil)
    next
      case (T y)
      then show ?thesis using ‹first_sym g la s› by (auto elim: first_sym.cases)
    qed
  next
    case Cons_gpre: (Cons s'' gpre')
    have "s'' = s" "gpre' @ s' # gsuf = gamma"
      using ‹gpre @ s' # gsuf = s # gamma› Cons_gpre by auto
    from ‹nullable_gamma g gpre› have "nullable_gamma g gpre'" "nullable_sym
g s''" using Cons_gpre
      by (auto elim: nullable_gamma.cases)
    show ?thesis
    proof (cases "nullableSym s nu")
      case True
      from ‹nullable_gamma g gpre'› have "first_gamma g la gamma" us-
ing ‹first_sym g la s'›
        by (auto intro: FirstGamma simp add: ‹gpre' @ s' # gsuf = gamma›[symmetric])
      then have "la ∈ set (firstGamma gamma nu fi)" using Cons.prems(1,2)
by (auto intro: Cons.IH)
      then show ?thesis by (simp add: True)
    next
      case False
      from ‹nullable_sym g s''› have "nullableSym s nu" using Cons.prems(1)
        by (auto simp add: nullableSym_nullable_sym ‹s'' = s›)
      then show ?thesis using False by auto
    qed
  qed
qed

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 (induction nu fi lx gsuf fo arbitrary: gpre rule: updateFo_induct_refined)
  case (1 nu fi lx fo)
  then show ?case by auto
next
  case (2 nu fi lx y gsuf fo)
  then show ?case by (auto intro: "2.IH"[where gpre = "gpre @ [T y]"])
next
```

```
  case (3 nu fi lx rx gsuf fo)
  then show ?case by (auto intro: "3.IH"[where gpre = "gpre @ [NT rx]"])
next
  case (4 nu fi lx rx gsuf fo)
  let ?fo' = "updateFo nu fi lx gsuf fo"
  let ?lSet = "findOrEmpty lx ?fo'"
  let ?rSet = "firstGamma gsuf nu fi"
  let ?additions = "(if nullableGamma gsuf nu then ?lSet @@ ?rSet else
?rSet)"
  have IH: "follow_map_sound (updateFo nu fi lx gsuf fo) g"
    by (auto intro: "4.IH"[where gpre = "gpre @ [NT rx]"] simp add: "4.prems"(1,2,3,4))
  have simp_updFo: "updateFo nu fi lx (NT rx # gsuf) fo = fmupd rx ?additions
?fo'"
    by (simp add: 4 Let_def)
  have "fmlookup (updateFo nu fi lx (NT rx # gsuf) fo) x = Some xFollow
∧ la ∈ set xFollow
    ⟹ follow_sym g la (NT x)" for x xFollow la
  proof (cases "rx = x")
    case True
    assume "fmlookup (updateFo nu fi lx (NT rx # gsuf) fo) x = Some xFollow
∧ la ∈ set xFollow"
      and "rx = x"
    then have "la ∈ set ?additions" using simp_updFo by auto
    then consider (la_in_lSet) "nullableGamma gsuf nu" "la ∈ set ?lSet"
      | (la_in_rSet) "la ∈ set ?rSet" by (auto split: if_splits)
    then show "follow_sym g la (NT x)"
    proof (cases)
      case la_in_lSet
      then obtain lxFollow where "fmlookup ?fo' lx = Some lxFollow" "la
∈ set lxFollow"
        using in_findOrEmpty_exists_set by fast
      then have "follow_sym g la (NT lx)" using follow_map_sound_def
IH by fastforce
      moreover have "(lx, gpre @ NT x # gsuf) ∈ set (prods g)" using
"4.prems"(3)  ‹rx = x› by blast
      moreover have "nullable_gamma g gsuf"
        using la_in_lSet "4.prems"(1) nu_sound_nullableGamma_sound by
blast
      ultimately show ?thesis by - (rule FollowLeft)
    next
      case la_in_rSet
      then have "first_gamma g la gsuf" using firstGamma_first_gamma
"4.prems"(1,2) by fastforce
      moreover have "(lx, gpre @ NT x # gsuf) ∈ set (prods g)" using
"4.prems"(3) True by blast
      ultimately show ?thesis by - (rule FollowRight)
    qed
  next
    case False
```

```
    assume A: "fmlookup (updateFo nu fi lx (NT rx # gsuf) fo) x = Some
xFollow ∧ la ∈ set xFollow"
    then show "follow_sym g la (NT x)" using False IH follow_map_sound_def
simp_updFo by fastforce
  qed
  then show ?case using follow_map_sound_def by fast
next
  case (5 nu fi lx rx gsuf fo rxFollow)
  then show ?case by (auto intro: "5.IH"[where gpre = "gpre @ [NT rx]"])
next
  case (6 nu fi lx rx gsuf fo rxFollow)
  let ?fo' = "updateFo nu fi lx gsuf fo"
  let ?lSet = "findOrEmpty lx ?fo'"
  let ?rSet = "firstGamma gsuf nu fi"
  let ?additions = "(if nullableGamma gsuf nu then ?lSet @@ ?rSet else
?rSet)"
  have IH: "follow_map_sound (updateFo nu fi lx gsuf fo) g"
    by (auto intro: "6.IH"[where gpre = "gpre @ [NT rx]"] simp add: "6.prems"(1,2,3,4))
  have simp_updFo: "updateFo nu fi lx (NT rx # gsuf) fo = fmupd rx (rxFollow
@@ ?additions) ?fo'"
    by (simp add: 6 Let_def)
  have "fmlookup (updateFo nu fi lx (NT rx # gsuf) fo) x = Some xFollow
∧ la ∈ set xFollow
        ⟹ follow_sym g la (NT x)" for x xFollow la
  proof (cases "rx = x")
    case True
    assume "fmlookup (updateFo nu fi lx (NT rx # gsuf) fo) x = Some xFollow
∧ la ∈ set xFollow"
      and "rx = x"
    then have "la ∈ set rxFollow ∨ la ∈ set ?additions" using simp_updFo
by auto
    then consider (la_in_rxFollow) "la ∈ set rxFollow"
      | (la_in_lSet) "nullableGamma gsuf nu" "la ∈ set ?lSet" | (la_in_rSet)
"la ∈ set ?rSet"
      by (auto split: if_splits)
    then show "follow_sym g la (NT x)"
    proof (cases)
      case la_in_rxFollow
      then show ?thesis using IH True follow_map_sound_def using "6.hyps"(1)
by fastforce
    next
      case la_in_lSet
      then obtain lxFollow where "fmlookup ?fo' lx = Some lxFollow" "la
∈ set lxFollow"
        using in_findOrEmpty_exists_set by fast
      then have "follow_sym g la (NT lx)" using IH follow_map_sound_def
by fastforce
      moreover have "(lx, gpre @ NT x # gsuf) ∈ set (prods g)"
        using ‹rx = x› "6.prems"(3) by auto
```

```
      moreover have "nullable_gamma g gsuf"
        using la_in_lSet "6.prems"(1) nu_sound_nullableGamma_sound by
auto
      ultimately show ?thesis by - (rule FollowLeft)
    next
      case la_in_rSet
      then have "first_gamma g la gsuf" using firstGamma_first_gamma
"6.prems"(1,2) by fastforce
      moreover have "(lx, gpre @ NT x # gsuf) ∈ set (prods g)" using
"6.prems"(3) True by auto
      ultimately show ?thesis by - (rule FollowRight)
    qed
  next
    case False
    assume A: "fmlookup (updateFo nu fi lx (NT rx # gsuf) fo) x = Some
xFollow ∧ la ∈ set xFollow"
    have "updateFo nu fi lx (NT rx # gsuf) fo = fmupd rx (rxFollow @@
?additions) ?fo'" by
      (simp add: 6 Let_def)
    then have "fmlookup (updateFo nu fi lx gsuf fo) x = Some xFollow"
using A by
      (auto simp add: False)
    then show "follow_sym g la (NT x)" using IH A(1) unfolding follow_map_sound_def
by blast
  qed
  then show ?case unfolding follow_map_sound_def by blast
qed

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"
  by (metis self_append_conv2 updateFo_preserves_soundness')

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 (induction suf arbitrary: pre)
  case Nil
  then show ?case by (simp add: followPass_def)
next
  case (Cons p suf)
  obtain lx gamma where "p = (lx, gamma)" by fastforce
  have "follow_map_sound (updateFo nu fi lx gamma (followPass suf nu fi
fo)) g"
  proof (rule updateFo_preserves_soundness)
    show "(lx, gamma) ∈ set (prods g)" using Cons.prems(4)
      by (metis ‹p = (lx, gamma)› in_set_conv_decomp)
```

```
  next
    have "(pre @ [p]) @ suf = prods g" using Cons.prems(4) by auto
    then show "follow_map_sound (followPass suf nu fi fo) g"
      using Cons.prems(1,2,3) by - (rule Cons.IH[where pre = "pre @ [p]"])
  qed (auto simp add: Cons.prems(1,2))
  then show ?case by (simp add: ‹p = (lx, gamma)›)
qed

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"
  by (simp add: followPass_preserves_soundness')

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 (induction "countFollowCands g fo" arbitrary: fo rule: less_induct)
  case less
  let ?fo' = "followPass (prods g) nu fi fo"
  have "mkFollowMap' g nu fi fo ≠ None" by (simp add: less.prems(2,4)
mkFollowMap'_dom_if_apac)
  moreover have "follow_map_sound (if fo = ?fo' then fo else the (mkFollowMap'
g nu fi ?fo')) g"
  proof (cases "fo = ?fo'")
    case True
    then show ?thesis using less.prems(3) by auto
  next
    case False
    have "countFollowCands g ?fo' < countFollowCands g fo"
      by (simp add: False followPass_not_equiv_candidates_lt less.prems(2,4))
    moreover have "follow_map_sound ?fo' g"
      using less.prems by - (rule followPass_preserves_soundness, auto)
    ultimately show ?thesis using followPass_preserves_apac less by fastforce
  qed
  ultimately show ?case using mkFollowMap'.simps[of g nu fi fo] by (auto
simp add: Let_def)
qed

lemma initial_fo_sound: "follow_map_sound (initial_fo g) g"
  unfolding follow_map_sound_def using FollowStart by auto

theorem mkFollowMap_sound:
  "nullable_set_for nu g ⟹ first_map_for fi g ⟹ follow_map_sound
(mkFollowMap g nu fi) g"
  unfolding mkFollowMap_def
  by (simp add: initial_fo_apac initial_fo_sound mkFollowMap'_preserves_soundness)
```

5.4 Completeness

```
lemma updateFo_preserves_map_keys: "x |∈| fmdom fo ⟹ x |∈| fmdom (updateFo
nu fi lx gamma fo)"
  by (induction nu fi lx gamma fo rule: updateFo_induct_refined) (auto
simp add: Let_def)

lemma followPass_preserves_map_keys: "x |∈| fmdom fo ⟹ x |∈| fmdom
(followPass ps nu fi fo)"
proof (induction ps)
  case Nil
  then show ?case by (simp add: followPass_def)
next
  case (Cons p ps)
  obtain x gamma where "p = (x, gamma)" by fastforce
  then show ?case by (simp add: updateFo_preserves_map_keys Cons)
qed

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 -
  assume "x ≠ x'"
  let ?fo' = "updateFo nu fi lx gsuf fo"
  let ?lSet = "findOrEmpty lx ?fo'"
  let ?rSet = "firstGamma gsuf nu fi"
  let ?additions = "(if nullableGamma gsuf nu then ?lSet @@ ?rSet else
?rSet)"
  show "(fmlookup ?fo' x = Some xFollow) =
    (fmlookup (updateFo nu fi lx (NT x' # gsuf) fo) x = Some xFollow)"
  proof (cases "fmlookup (updateFo nu fi lx gsuf fo) x'")
    case None
    show ?thesis
    proof (cases "?additions = []")
      case True
      show ?thesis by (auto simp add: True None)
    next
      case False
      have "fmlookup (updateFo nu fi lx (NT x' # gsuf) fo) x =
        fmlookup (updateFo nu fi lx gsuf fo) x" using ‹x ≠ x'›
        by (auto simp add: Let_def False None)
      then show ?thesis by auto
    qed
  next
    case (Some rxFollow)
    then show ?thesis
    proof (cases "set ?additions ⊆ set rxFollow")
      case True
      show ?thesis by (auto simp add: True Some)
    next
```

```
      case False
      have "fmlookup (updateFo nu fi lx (NT x' # gsuf) fo) x =
        fmlookup (updateFo nu fi lx gsuf fo) x" using ‹x ≠ x'›
        by (auto simp add: Let_def False Some)
      then show ?thesis by auto
    qed
  qed
qed

lemma updateFo_value_subset:
  "fmlookup fo x = Some s1 ⟹ fmlookup (updateFo nu fi lx gamma fo) x
= Some s2
  ⟹ set s1 ⊆ set s2"
proof (induction nu fi lx gamma fo arbitrary: s2 rule: updateFo_induct_refined)
  case (4 nu fi lx rx gamma' fo)
  show ?case
  proof (cases "x = rx")
    case True
    from "4.prems"(1) have "x |∈| fmdom fo" by (simp add: fmdomI)
    then have "x |∈| fmdom (updateFo nu fi lx gamma' fo)" by
      (auto intro: updateFo_preserves_map_keys)
    moreover have "x |∉| fmdom (updateFo nu fi lx gamma' fo)" using
"4.hyps"(1) by
      (simp add: True fmdom_notI)
    ultimately have "False" by auto
    then show ?thesis by auto
  next
    case False
    with "4.prems"(2) have "fmlookup (updateFo nu fi lx gamma' fo) x
= Some s2" by
      (auto simp add: Let_def "4.hyps")
    then show ?thesis using "4.IH" "4.prems"(1) by auto
  qed
next
  case (6 nu fi lx rx gamma' fo rxFollow)
  then show ?case
  proof (cases "x = rx")
    case True
    let ?fo' = "updateFo nu fi lx gamma' fo"
    let ?lSet = "findOrEmpty lx ?fo'"
    let ?rSet = "firstGamma gamma' nu fi"
    let ?additions = "(if nullableGamma gamma' nu then ?lSet @@ ?rSet else
?rSet)"
    from "6.prems"(2) have "set s2 = set (?additions @@ rxFollow)"
      by (auto simp add: Let_def "6.hyps" True)
    moreover have "set s1 ⊆ set rxFollow" using "6.IH" "6.hyps"(1) "6.prems"(1)
True by blast
    ultimately show ?thesis by auto
  next
```

```
    case False
    with "6.prems"(2) have "fmlookup (updateFo nu fi lx gamma' fo) x
= Some s2" by
      (auto simp add: Let_def "6.hyps")
    then show ?thesis using "6.IH" "6.prems"(1) by auto
  qed
qed auto

lemma updateFo_only_appends:
  "fmlookup fo x = Some s1 ⟹ fmlookup (updateFo nu fi lx gamma fo) x
= Some s2
  ⟹ ∃suf. s2 = s1 @ suf"
proof (induction nu fi lx gamma fo arbitrary: s2 rule: updateFo_induct_refined)
  case (4 nu fi lx rx gamma' fo)
  then show ?case
  proof (cases "x = rx")
    case True
    have "x |∈| fmdom fo" by (simp add: "4.prems"(1) fmdomI)
    then have "x |∈| fmdom (updateFo nu fi lx gamma' fo)" by (simp add:
updateFo_preserves_map_keys)
    then have "False" using "4.hyps"(1) by (simp add: True fmdom_notI)
    then show ?thesis by auto
  next
    case False
    have "fmlookup (updateFo nu fi lx gamma' fo) x = Some s2"
      using "4.prems"(2) False find_updateFo_cons_neq by fast
    then show ?thesis using "4.IH" "4.prems"(1) by auto
  qed
next
  case (6 nu fi lx rx gamma' fo rxFollow)
  let ?fo' = "updateFo nu fi lx gamma' fo"
  let ?lSet = "findOrEmpty lx ?fo'"
  let ?rSet = "firstGamma gamma' nu fi"
  let ?additions = "(if nullableGamma gamma' nu then ?lSet @@ ?rSet else
?rSet)"
  have "updateFo nu fi lx (NT rx # gamma') fo = (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'))"
    by (metis updateFo.simps(3))
  then have A: "updateFo nu fi lx (NT rx # gamma') fo = fmupd rx (rxFollow
@@ ?additions) ?fo'"
    by (simp add: "6.hyps"(1) "6.hyps"(2))
  show ?case
  proof (cases "x = rx")
    case True
    then show ?thesis
      using "6.IH"[OF "6.prems"(1)] "6.hyps"(1) "6.prems"(2) A unfold-
```

```
ing list_union_def by auto
  next
    case False
    then show ?thesis by (meson "6.IH" "6.prems"(1) "6.prems"(2) find_updateFo_cons_neq)
  qed
qed auto


lemma followPass_value_subset:
  "fmlookup fo x = Some s1 ⟹ fmlookup (followPass ps nu fi fo) x = Some
s2 ⟹ set s1 ⊆ set s2"
proof (induction ps arbitrary: s1 s2)
  case Nil
  then show ?case by (simp add: followPass_def)
next
  case (Cons p ps)
  obtain y gamma where "p = (y, gamma)" by fastforce
  have "x |∈| fmdom (followPass ps nu fi fo)" by
    (simp add: Cons.prems(1) fmdomI followPass_preserves_map_keys)
  then obtain s where s_def: "fmlookup (followPass ps nu fi fo) x = Some
s" by
    (auto simp add: fmdomI)
  then have s1_subset_s: "set s1 ⊆ set s" using Cons.prems(1) Cons.IH
by auto
  have "fmlookup (updateFo nu fi y gamma (followPass ps nu fi fo)) x =
Some s2" using Cons.prems(2)
    by (simp add: ‹p = (y, gamma)›)
  then have s_subset_s2: "set s ⊆ set s2" using s_def by
    - (rule updateFo_value_subset[where ?fo = "followPass ps nu fi fo"])
  show ?case using s1_subset_s s_subset_s2 by auto
qed

lemma followPass_only_appends: "fmlookup fo x = Some s1
  ⟹ fmlookup (followPass ps nu fi fo) x = Some s2 ⟹ ∃suf. s2 = s1
@ suf"
proof (induction ps arbitrary: s1 s2)
  case Nil
  then show ?case by (simp add: followPass_def)
next
  case (Cons p ps)
  obtain y gamma where "p = (y, gamma)" by fastforce
  have "x |∈| fmdom (followPass ps nu fi fo)" by
    (simp add: Cons.prems(1) fmdomI followPass_preserves_map_keys)
  then obtain s where s_def: "fmlookup (followPass ps nu fi fo) x = Some
s" by
    (auto simp add: fmdomI)
  moreover obtain suf where "s = s1 @ suf" using Cons.prems(1) Cons.IH
s_def by auto
  moreover have "fmlookup (updateFo nu fi y gamma (followPass ps nu fi
```

```
fo)) x = Some s2"
    using Cons.prems(2) by (simp add: ‹p = (y, gamma)›)
  ultimately show ?case using updateFo_only_appends[OF s_def] by fastforce
qed

lemma followPass_equiv_cons_tl: "fo = followPass ((x, gamma) # ps) nu
fi fo
  ⟹ fo = followPass ps nu fi fo"
proof (rule fmap_ext)
  fix y
  assume assm: "fo = followPass ((x, gamma) # ps) nu fi fo"
  then show "fmlookup fo y = fmlookup (followPass ps nu fi fo) y"
  proof (cases "fmlookup fo y")
    case None
    then have "y |∉| fmdom (followPass ((x, gamma) # ps) nu fi fo)" us-
ing assm by auto
    then have "y |∉| fmdom (followPass ps nu fi fo)" by (auto intro:
updateFo_preserves_map_keys)
    then show ?thesis using None by auto
  next
    case (Some yFollow)
    then have "y |∈| fmdom (followPass ps nu fi fo)" by
        (simp add: fmdomI followPass_preserves_map_keys)
    then obtain yFollow' where yFollow'_def: "fmlookup (followPass ps
nu fi fo) y = Some yFollow'"
      by auto
    from assm have "fmlookup (updateFo nu fi x gamma (followPass ps nu
fi fo)) y = Some yFollow" by
        (simp add: Some)
    then have "∃suf. yFollow = yFollow' @ suf" using updateFo_only_appends[OF
yFollow'_def] by fast
    moreover have "∃suf. yFollow' = yFollow @ suf"
      using followPass_only_appends[OF Some yFollow'_def] by auto
    ultimately show ?thesis using Some yFollow'_def by fastforce
  qed
qed

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 (cases s)
  case (NT rx')
  then show ?thesis
  proof (cases "rx = rx'")
    case True
```

```
    let ?fo' = "updateFo nu fi lx gamma fo"
    let ?lSet = "findOrEmpty lx ?fo'"
    let ?rSet = "firstGamma gamma nu fi"
    let ?additions = "(if nullableGamma gamma nu then ?lSet @@ ?rSet else
?rSet)"
    obtain rxFollow where rxFollow_def: "fmlookup ?fo' rx = Some rxFollow"
"la ∈ set rxFollow"
      using assms(3) by auto
    then have "fmlookup ?fo' rx' = Some rxFollow" using True by auto
    then show ?thesis
    proof (cases "set ?additions ⊆ set rxFollow")
      case True
      then show ?thesis by (simp add: NT ‹fmlookup ?fo' rx' = Some rxFollow›
True rxFollow_def)
    next
      case False
      have "updateFo nu fi lx (NT rx' # gamma) fo =
        fmupd rx' (rxFollow @@ ?additions) ?fo'"
        by (simp add: NT ‹fmlookup ?fo' rx' = Some rxFollow› False Let_def)
      then have "fmlookup (updateFo nu fi lx (s # gamma) fo) rx =
        Some (rxFollow @@ ?additions)" by (simp add: NT True)
      moreover have "la ∈ set (rxFollow @@ ?additions)" using rxFollow_def(2)
by auto
      ultimately show ?thesis by auto
    qed
  next
    case False
    then show ?thesis using find_updateFo_cons_neq NT assms(3) by fastforce
  qed
next
  case (T y)
  then show ?thesis by (simp add: assms)
qed

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 (induction gpre)
  case Nil
  let ?fo' = "updateFo nu fi lx gsuf fo"
  let ?lSet = "findOrEmpty lx ?fo'"
  let ?rSet = "firstGamma gsuf nu fi"
  let ?additions = "(if nullableGamma gsuf nu then ?lSet @@ ?rSet else
?rSet)"
  show ?case
  proof (cases "fmlookup (updateFo nu fi lx gsuf fo) rx")
```

```
    case None
    then show ?thesis
    proof (cases "?additions = []")
      case True
      then show ?thesis
        by (metis Nil.prems UnI2 empty_iff first_gamma_firstGamma list.set(1)
set_list_union)
    next
      case False
      have "la ∈ set (firstGamma gsuf nu fi)" using Nil first_gamma_firstGamma
by blast
      then have "la ∈ set ?additions" by auto
      moreover have "fmlookup (updateFo nu fi lx ([] @ NT rx # gsuf)
fo) rx = Some ?additions" by
        (simp add: None False Let_def)
      ultimately show ?thesis by auto
    qed
  next
    case (Some rxFollow)
    then show ?thesis
    proof (cases "set ?additions ⊆ set rxFollow")
      case True
      then show ?thesis
        using Nil Some first_gamma_firstGamma by fastforce
    next
      case False
      have "la ∈ set (firstGamma gsuf nu fi)" using Nil first_gamma_firstGamma
by blast
      then have "la ∈ set (rxFollow @@ ?additions)" by auto
      moreover have "fmlookup (updateFo nu fi lx ([] @ NT rx # gsuf)
fo) rx =
        Some (rxFollow @@ ?additions)" by (simp add: Some False Let_def)
      ultimately show ?thesis by auto
    qed
  qed
next
  case (Cons s gpre)
  then show ?case using exists_follow_set_Cons by fastforce
qed

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 (induction psuf arbitrary: ppre)
  case Nil
  then show ?case  by auto
```

```
next
  case (Cons p ps)
  obtain x gamma where "p = (x, gamma)" by fastforce
  from Cons.prems(4) have "(lx, gpre @ NT rx # gsuf) ∈ set (p # ps)"
by auto
  then consider (is_p) "(lx, gpre @ NT rx # gsuf) = (x, gamma)"
    | (in_ps) "(lx, gpre @ NT rx # gsuf) ∈ set ps" using ‹p = (x, gamma)›
by auto
  then show ?case
  proof cases
    case is_p
    have "∃rxFollow. fmlookup (updateFo nu fi lx (gpre @ NT rx # gsuf)
(followPass ps nu fi fo)) rx
      = Some rxFollow ∧ la ∈ set rxFollow" using Cons.prems(1,2,5)
      by (rule exists_follow_set_containing_first_gamma)
    then show ?thesis
      using Cons.prems(3) ‹p = (x, gamma)› is_p by auto
  next
    case in_ps
    have "fo = followPass ps nu fi fo" using Cons.prems(3) ‹p = (x, gamma)›
by
      - (rule followPass_equiv_cons_tl, auto)
    moreover have "(ppre @ [p]) @ ps = prods g" by (simp add: Cons.prems(6))
    ultimately show ?thesis using Cons.IH Cons.prems(1,2,5) in_ps by
fastforce
  qed
qed

lemma nullable_gamma_nullableGamma:
  "nullable_set_for nu g ⟹ nullable_gamma g gamma ⟹ nullableGamma
gamma nu"
proof (induction gamma)
  case Nil
  then show ?case by auto
next
  case (Cons s gamma)
  from Cons.prems(2) have "nullable_gamma g gamma" "nullable_sym g s"
by
    (auto elim: nullable_gamma.cases)
  from ‹nullable_sym g s› obtain x where "s = NT x" by (auto intro:
nullable_sym.cases)
  then have "x ∈ nu" using ‹nullable_sym g s› Cons.prems(1) unfolding
nullable_set_complete_def by
    auto
  moreover have "nullableGamma gamma nu" using ‹nullable_gamma g gamma›
by
    (auto intro: Cons.IH simp add: Cons.prems(1))
  ultimately show ?case by (simp add: ‹s = NT x›)
qed
```

```
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 -
  assume A: "fmlookup fo x = Some s" "la ∈ set s"
  then have "x |∈| fmdom fo" by (simp add: fmdomI)
  then have "x |∈| fmdom (updateFo nu fi x' gamma fo)" by (simp add:
updateFo_preserves_map_keys)
  then obtain s' where s'_def: "fmlookup (updateFo nu fi x' gamma fo)
x = Some s'" by auto
  then have "set s ⊆ set s'" using A by - (rule updateFo_value_subset)
  then show ?thesis unfolding findOrEmpty_def using A(2) s'_def by auto
qed

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 (induction gpre)
  case Nil
  let ?fo' = "updateFo nu fi lx gsuf fo"
  let ?lSet = "findOrEmpty lx ?fo'"
  let ?rSet = "firstGamma gsuf nu fi"
  let ?additions = "(if nullableGamma gsuf nu then ?lSet @@ ?rSet else
?rSet)"
  have "nullableGamma gsuf nu" using Nil.prems(1,3) nullable_gamma_nullableGamma
by auto
  then have "?additions = ?lSet @@ ?rSet" by auto
  show ?case
  proof (cases "fmlookup (updateFo nu fi lx gsuf fo) rx")
    case None
    show ?thesis
    proof (cases "?additions = []")
      case True
      then show ?thesis
        using Nil.prems(1,3-5) nullable_gamma_nullableGamma updateFo_preserves_membership_i
        by (metis Nil_is_append_conv emptyE empty_set list_union_def)
    next
      case False
      have "la ∈ set (?lSet @@ ?rSet)" by
        (simp add: Nil.prems(4,5) updateFo_preserves_membership_in_value)
      moreover have "updateFo nu fi lx (NT rx # gsuf) fo =
        fmupd rx ?additions (updateFo nu fi lx gsuf fo)" by (simp add:
None False Let_def)
      ultimately show ?thesis using ‹?additions = ?lSet @@ ?rSet› by
```

```
simp
    qed
  next
    case (Some rxFollow)
    show ?thesis
    proof (cases "set ?additions ⊆ set rxFollow")
      case True
      then show ?thesis
        using Nil.prems(4,5) Some ‹nullableGamma gsuf nu› updateFo_preserves_membership_in_
        by fastforce
    next
      case False
      have "la ∈ set (?lSet @@ ?rSet)" by
          (simp add: Nil.prems(4,5) updateFo_preserves_membership_in_value)
      moreover have "updateFo nu fi lx (NT rx # gsuf) fo =
        fmupd rx (rxFollow @@ ?additions) (updateFo nu fi lx gsuf fo)"
        by (simp add: Some False Let_def)
      ultimately show ?thesis using ‹?additions = ?lSet @@ ?rSet› by
simp
    qed
  qed
next
  case (Cons s gpre)
  then show ?case using exists_follow_set_Cons by fastforce
qed

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 (induction psuf arbitrary: ppre)
  case Nil
  then show ?case  by auto
next
  case (Cons p ps)
  let ?fo' = "followPass ps nu fi fo"
  obtain x gamma where "p = (x, gamma)" by fastforce
  from Cons.prems(4) consider (is_p) "(lx, gpre @ NT rx # gsuf) = (x,
gamma)"
    | (in_ps) "(lx, gpre @ NT rx # gsuf) ∈ set ps" using ‹p = (x, gamma)›
by auto
  then show ?case
  proof cases
    case is_p
    have "fmlookup ?fo' lx = Some lxFollow"
```

```
      using Cons.prems(3,7) ‹p = (x, gamma)› followPass_equiv_cons_tl
by fastforce
    then have "∃rxFollow. fmlookup (updateFo nu fi lx (gpre @ NT rx #
gsuf) ?fo') rx =
      Some rxFollow ∧ la ∈ set rxFollow" using Cons.prems(1,2,6,8)
      by - (rule exists_follow_set_containing_follow_left)
    then show ?thesis using Cons.prems(3) ‹p = (x, gamma)› is_p by auto
  next
    case in_ps
    have "fo = ?fo'" using Cons.prems(3) ‹p = (x, gamma)› by - (rule
followPass_equiv_cons_tl, auto)
    moreover have "(ppre @ [p]) @ ps = prods g" by (simp add: Cons.prems(5))
    ultimately show ?thesis using Cons.IH Cons.prems(1,2,6,7,8) in_ps
by fastforce
  qed
qed

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 -
  assume A: "nullable_set_for nu g" "first_map_for fi g"
    "(start g, EOF) ∈ pairsOf fo" "fo = followPass (prods g) nu fi fo"
  have "follow_sym g la s
    ⟹ (∀x. s = NT x ⟶ (∃xFollow. fmlookup fo x = Some xFollow ∧
la ∈ set xFollow))" for la s
  proof -
    assume "follow_sym g la s"
    then show "(∀x. s = NT x ⟶ (∃xFollow. fmlookup fo x = Some xFollow
∧ la ∈ set xFollow))"
    proof (induction rule: follow_sym.induct)
      case FollowStart
      show ?case by (simp add: A(3) in_pairsOf_exists[symmetric])
    next
      case (FollowRight x1 gpre x2 gsuf la)
      then show ?case using A followPass_equiv_right by fast
    next
      case (FollowLeft x1 gpre x2 gsuf la)
      then show ?case using A followPass_equiv_left by fast
    qed
  qed
  then show ?thesis unfolding follow_map_complete_def by auto
qed

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 (induction "countFollowCands g fo" arbitrary: fo rule: less_induct)
  case less
  let ?fo' = "followPass (prods g) nu fi fo"
  have "mkFollowMap' g nu fi fo ≠ None" by (simp add: less.prems(3,4)
mkFollowMap'_dom_if_apac)
  moreover have "follow_map_complete (if fo = ?fo' then fo else the (mkFollowMap'
g nu fi ?fo')) g"
  proof (cases "fo = ?fo'")
    case True
    then show ?thesis using followPass_equiv_complete less.prems(1,2,3)
by auto
  next
    case False
    have "countFollowCands g ?fo' < countFollowCands g fo" by
      (simp add: False followPass_not_equiv_candidates_lt less.prems(3,4))
    moreover have "(start g, EOF) ∈ pairsOf ?fo'" using followPass_subset
less.prems(1) by blast
    moreover have "all_pairs_are_follow_candidates ?fo' g" by
      (simp add: followPass_preserves_apac less.prems(3,4))
    ultimately show ?thesis by (auto intro: less.hyps simp add: False
less.prems(2,3))
  qed
  ultimately show ?case using mkFollowMap'.simps[of g nu fi fo] by (auto
simp add: Let_def)
qed

lemma start_eof_in_initial_fo: "(start g, EOF) ∈ pairsOf (initial_fo
g)"
  by (simp add: in_add_value)

theorem mkFollowMap_complete:
  "nullable_set_for nu g ⟹ first_map_for fi g ⟹ follow_map_complete
(mkFollowMap g nu fi) g"
  by (simp add: initial_fo_apac mkFollowMap'_complete mkFollowMap_def
start_eof_in_initial_fo)

theorem mkFollowMap_correct:
  "nullable_set_for nu g ⟹ first_map_for fi g ⟹ follow_map_for (mkFollowMap
g nu fi) g"
  by (simp add: mkFollowMap_complete mkFollowMap_sound)

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"*
using *assms firstGamma_first_gamma* **unfolding** *firstKeysForProd_def lookahead_for_def*
by *fastforce*

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 *(cases "nullableGamma gamma nu")*
case *True*
with *assms(4)* **have** *"la ∈ set (findOrEmpty x fo)"* **by** *(auto simp add:*
followKeysForProd_def)
then have *"follow_sym g la (NT x)"* **using** *assms(3) follow_map_sound_def*
in_findOrEmpty_exists_set
by *fast*
then have *"nullable_gamma g gamma ∧ follow_sym g la (NT x)"*

```
    using True assms(1) nu_sound_nullableGamma_sound by blast
  moreover have "x = y" using assms(4) by (auto simp add: followKeysForProd_def)
  ultimately show ?thesis by (simp add: lookahead_for_def)
next
  case False
  then show ?thesis using assms unfolding followKeysForProd_def lookahead_for_def
by simp
qed

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"
  using assms firstKeysForProd_lookaheads followKeysForProd_lookaheads
  by fastforce

lemma findOrEmpty_sset_laOf_fi: "first_map_for fi g ⟹ set (findOrEmpty
x fi) ⊆ lookaheadsOf g"
proof
  fix y
  assume A: "first_map_for fi g" "y ∈ set (findOrEmpty x fi)"
  then obtain s where "fmlookup fi x = Some s" "y ∈ set s" using in_findOrEmpty_exists_se
by
      fastforce
  then show "y ∈ lookaheadsOf g" using A by (auto intro: first_map_la_in_lookaheadsOf)
qed

lemma follow_sym_in_lookaheadsOf:
  "follow_sym g la (NT x) ⟹ la ∈ lookaheadsOf g"
proof (induction rule: follow_sym.induct)
  case (FollowRight x1 gpre x2 gsuf la)
  have "la ∈ lookaheadsOf g"
    using FollowRight.hyps mkFirstMap_correct mkNullableSet_correct
    by - (rule in_firstGamma_in_lookaheadsOf[where ?gpre="gpre @ [NT
x2]"],
      auto intro: first_gamma_firstGamma)
  then show ?case unfolding lookaheadsOf_def by auto
qed (simp add: lookaheadsOf_def)

lemma follow_map_la_in_lookaheadsOf:
  "follow_map_for fo g ⟹ fmlookup fo x = Some s ⟹ la ∈ set s ⟹
la ∈ lookaheadsOf g"
  unfolding follow_map_sound_def using follow_sym_in_lookaheadsOf by
fastforce

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

```
  fix y
  assume A: "follow_map_for fo g" "y ∈ set (findOrEmpty x fo)"
  then obtain s where "fmlookup fo x = Some s" "y ∈ set s" using in_findOrEmpty_exists_se
by
      fastforce
  then show "y ∈ lookaheadsOf g" using A by (auto intro: follow_map_la_in_lookaheadsOf)
qed

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 (induction las arbitrary: pt pt')
  case Nil
  then show ?case by auto
next
  case (Cons k las)
  consider (lookup_pt_None) "fmlookup pt k = None"
    | (lookup_pt_Same) "fmlookup pt k = Some p" using Cons.prems(3) by
fastforce
  then show ?case
  proof cases
    case lookup_pt_None
    have "pt_sound (fmupd k p pt) g" unfolding pt_sound_def
    proof clarify
      fix x x' la gamma
      assume assm: "fmlookup (fmupd k p pt) (x', la) = Some (x, gamma)"
      show "x' = x ∧ (x, gamma) ∈ set (prods g) ∧ lookahead_for la x
gamma g"
      proof (cases "k = (x', la)")
        case True
        then have "p = (x, gamma)" using assm by auto
        then show ?thesis using assms(1-4) Cons.prems(2) keys_are_lookaheads
True by fastforce
      next
        case False
        then show ?thesis using Cons.prems(1) assm unfolding pt_sound_def
by auto
      qed
    qed
    then show ?thesis using Cons.IH Cons.prems(2,3) lookup_pt_None by
auto
  next
    case lookup_pt_Same
    then show ?thesis using Cons by auto
  qed
qed
```

```
lemma mkParseTable'_nested: "mkParseTable' suf nu fi fo (mkParseTable'
pre nu fi fo pt)
  = mkParseTable' (pre @ suf) nu fi fo pt"
  by (induction pre arbitrary: pt) auto

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 (induction suf arbitrary: pre)
  case Nil
  then show ?case by auto
next
  case (Cons p suf)
  have "mkParseTable' (pre @ [p]) nu fi fo pt
    = mkParseTable' [p] nu fi fo (mkParseTable' pre nu fi fo pt)"
    by (simp add: mkParseTable'_nested)
  then show ?case using Cons.IH[where pre = "pre @ [p]"] Cons.prems
by
    (auto simp add: mkParseTable'_nested)
qed

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'"
  by (cases "mkParseTable' pre nu fi fo (PT pt)")
    (auto simp add: mkParseTable'_failure_preserved)

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 (induction ps arbitrary: pt)
  case Nil
  show ?case using Nil.prems(2,3) by auto
next
  case (Cons p ps)
  obtain pre_pt' where pre_pt'_def: "mkParseTable' [p] nu fi fo (PT pt)
= PT pre_pt'"
    using all_pre_pt_non_failure[where pre = "[p]"] Cons.prems(3) by
fastforce
  then have "pt_sound pre_pt' g" using assms Cons.prems pre_pt'_def by
    (auto intro: addEntries_preserves_soundness)
  then show ?case using Cons.IH Cons.prems(1,3) pre_pt'_def by auto
qed

lemma initial_pt_sound: "pt_sound fmempty g"
```

```
  unfolding pt_sound_def by simp

theorem mkParseTable_sound: "mkParseTable g = PT pt ⟹ pt_sound pt g"
proof -
  assume assm: "mkParseTable g = PT pt"
  let ?nu = "mkNullableSet g"
  let ?fi = "mkFirstMap g ?nu"
  let ?fo = "mkFollowMap g ?nu ?fi"
  have "nullable_set_for ?nu g" by (rule mkNullableSet_correct)
  moreover have "first_map_for ?fi g" using calculation(1) by (rule
mkFirstMap_correct)
  moreover have "follow_map_for ?fo g" using calculation(1,2) by - (rule
mkFollowMap_correct)
  ultimately show "pt_sound pt g" by
    (metis assm initial_pt_sound mkParseTable'_preserves_soundness mkParseTable_def
order_refl)
qed
```

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 (cases "first_gamma g la gamma")
  case True
  then show ?thesis using assms(1,2) by
    (auto intro: first_gamma_firstGamma simp add: firstKeysForProd_def)
next
  case False
  have "nullableGamma gamma nu" using assms(1,4) False by
    (auto intro: nullable_gamma_nullableGamma simp add: lookahead_for_def)
  moreover have "la ∈ set (findOrEmpty x fo)" using assms(3,4) False
by
    (auto simp add: lookahead_for_def follow_map_complete_def in_findOrEmpty_exists_set)
  ultimately show ?thesis by (simp add: followKeysForProd_def)
qed

lemma addEntries_lookup_same_or_none: "addEntries las xp (PT pt) = PT
pt' ⟹ fmlookup pt k = Some x
  ⟹ fmlookup pt' k = Some x"
proof (induction las arbitrary: pt pt')
  case Nil
  then show ?case by auto
next
  case (Cons k' las)
  then show ?case by (cases "k' = k") (auto split: option.splits if_splits)
qed
```

```
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 (induction ps arbitrary: pt pt')
  case Nil
  then show ?case by auto
next
  case (Cons p ps)
  obtain pre_pt' where pre_pt'_def: "mkParseTable' [p] nu fi fo (PT pt)
= PT pre_pt'"
    using Cons.prems(1) all_pre_pt_non_failure[where pre = "[p]"] by
fastforce
  then have "fmlookup pre_pt' k = Some x" by
    (simp add: Cons.prems(2) addEntries_lookup_same_or_none)
  then show ?case using Cons.IH Cons.prems(1) pre_pt'_def by auto
qed

lemma addEntries_in_pt:
  "k ∈ set las ⟹ addEntries las xp (PT pt) = PT pt' ⟹ fmlookup pt'
k = Some xp"
proof (induction las arbitrary: pt pt')
  case Nil
  then show ?case by auto
next
  case (Cons k' las)
  consider (is_k) "k = k'" | (in_las) "k ∈ set las" using Cons.prems(1)
by auto
  then show ?case
  proof cases
    case is_k
    obtain pre_pt' where pre_pt'_def: "addEntries [k'] xp (PT pt) = PT
pre_pt'" using Cons.prems(2)
      by (auto split: option.splits if_splits)
    then have "fmlookup pre_pt' k' = Some xp" by (auto split: option.splits
if_splits)
    moreover have "addEntries las xp (PT pre_pt') = PT pt'" using Cons.prems(2)
by
      (auto simp add: pre_pt'_def[symmetric] split: option.split)
    ultimately show ?thesis
      using addEntries_lookup_same_or_none[where pt = pre_pt'] pre_pt'_def
is_k by auto
  next
    case in_las
    then show ?thesis using Cons.IH Cons.prems(2) by (auto split: option.splits
if_splits)
  qed
qed
```

```
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 (induction psuf arbitrary: ppre pt)
  case Nil
  then show ?case by auto
next
  case (Cons p psuf)
  obtain x' gamma' where "p = (x', gamma')" by fastforce
  obtain pre_pt' where pre_pt'_def: "mkParseTable' [p] nu fi fo (PT pt)
= PT pre_pt'"
    using Cons.prems(7) all_pre_pt_non_failure[where pre = "[p]"] by
fastforce
  from Cons.prems(5) consider (in_psuf) "(x, gamma) ∈ set psuf" | (is_p)
"p = (x, gamma)" by auto
  then show ?case
  proof cases
    case in_psuf
    then show ?thesis using Cons pre_pt'_def by auto
  next
    case is_p
    then have "(x, la) ∈ set (keysForProd nu fi fo p)" using Cons.prems(1-3,6)
la_in_keysForProd by
      fastforce
    then have "fmlookup pre_pt' (x, la) = Some (x, gamma)"
      using addEntries_in_pt is_p pre_pt'_def by fastforce
    then show ?thesis using Cons.prems(7) mkParseTable'_lookup_same_or_none
pre_pt'_def by fastforce
  qed
qed

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)"
  by (auto intro: mkParseTable'_complete'[where ?ppre = "[]"])

theorem mkParseTable_complete: "mkParseTable g = PT pt ⟹ pt_complete
pt g"
  unfolding pt_complete_def mkParseTable_def
  by (meson mkFirstMap_correct mkFollowMap_correct mkNullableSet_correct
mkParseTable'_complete)

theorem mkParseTable_correct: "mkParseTable g = PT pt ⟹ parse_table_correct
```

```
pt g"
  by (auto simp add: mkParseTable_complete mkParseTable_sound)
```

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 (goal_cases)
  case (1 P x)
  show ?case
  proof (cases x)
    case (Inl a)
    then show ?thesis
    proof (cases a)
      case (fields t u v w)
      then show ?thesis using "1" Inl by (cases u; cases v) auto
    qed
  next
    case (Inr b)
    then show ?thesis
    proof (cases b)
      case (fields t u v w)
      then show ?thesis using "1" Inr by (cases v) auto
    qed
  qed
qed auto

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"
  by (auto simp:lex_triple_def)

lemma wf_lex_triple[intro!]:
  assumes "wf ra" "wf rb" "wf rc"
  shows "wf (lex_triple ra rb rc)"
  by (simp add: assms parse.lex_triple_def wf_lex_prod)

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 (induction rule: parseSymbol_parseGamma.pinduct)
  case (1 pt a vis)
  then show ?case
    by (simp add: parseSymbol.psimps(1))
next
  case (2 pt a t ts vis)
  then show ?case
```

```
    by (simp add: parseSymbol.psimps(2) split: if_splits)
next
  case (3 pt x ts vis)
  then show ?case by (auto simp add: parseSymbol.psimps(3) split: if_splits
option.splits)
next
  case (4 pt n ts vis)
  then show ?case by (auto simp add: parseGamma.psimps(1))
next
  case (5 pt n s gamma' ts vis)
  then show ?case
    by (fastforce simp add: parseGamma.psimps(2) split: return_type.splits
parse_tree.splits)
qed

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 -
  from assms(2) have "x |∈| nt_from_pt pt" by (metis fimage_eqI fmdomI
fst_conv nt_from_pt_def)
  then have "x |∈| nt_from_pt pt - vis" using assms(1) by simp
  then show ?thesis by (metis fcard_fminus1_less fminus_finsert)
qed

termination
proof (relation "mlex_triple (λx. case x of
      Inl (pt, s, ts, vis) ⇒ parse_ind_meas_sym pt s ts vis
    | Inr (pt, n, ss, ts, vis) ⇒ parse_ind_meas_sym_list pt ss ts vis)",
goal_cases)
  case 1
  then show ?case by (simp add: mlex_triple_def wf_lex_triple)
next
  case (2 pt x ts vis s x' gamma)
  have "fcard (nt_from_pt pt - (finsert x vis)) < fcard (nt_from_pt pt
- vis)"
    using "2" fcard_diff_insert_less by fastforce
  then show ?case unfolding mlex_triple_def parse_ind_meas_sym_list_def
parse_ind_meas_sym_def by
      auto
next
  case (3 pt s gamma' ts vis)
  then show ?case unfolding mlex_triple_def parse_ind_meas_sym_list_def
parse_ind_meas_sym_def by
      auto
next
  case (4 pt s gamma' ts vis y z str r)
  then show ?case unfolding mlex_triple_def parse_ind_meas_sym_list_def
```

```
parse_ind_meas_sym_def
    by (fastforce dest: parseSymbol_length_bound_partial)
qed

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 (induction pt s ts vis and pt n gamma ts vis rule: parseSymbol_parseGamma.induct)
  case (5 pt n s gamma' ts vis)
  then show ?case by (fastforce split: return_type.splits parse_tree.splits
option.splits if_splits)
qed (auto split: return_type.splits parse_tree.splits option.splits if_splits)

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"
  using assms
  by (auto intro: wf_induct_rule[where r = "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)"]
       simp add: mlex_triple_def wf_lex_triple)

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 (induction pt s ts vis and pt x gamma ts vis arbitrary: and v r
    rule: parseSymbol_parseGamma.induct)
  case (5 pt n s' gamma' ts vis)
  obtain a b where A: "parseSymbol pt s' ts vis = RESULT a b"
    using 5(2,3) by (auto split: return_type.splits parse_tree.splits
if_splits)
  show ?case
  proof (cases gamma')
    case Nil
    then have "parseGamma pt n gamma' b (if length b < length ts then
{||} else vis)
        = RESULT (Node n []) b"
      by auto
    then show ?thesis using 5(3) A
      by (auto split: return_type.splits parse_tree.splits)
  next
    case (Cons x'' gamma'')
```

```
    obtain ls where "v = Node n ls" using 5(2,3) A by
        (auto split: return_type.splits parse_tree.splits)
    then show ?thesis by auto
  qed
qed auto

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 (induction arbitrary: v r rule: parse_meas_induct)
  case (1 y)
  note IH = "1"
  {
    fix pt s ts vis
    assume A: "y = Inl (pt, s, ts, vis)" "parse_table_correct pt g"
        "parseSymbol pt s ts vis = RESULT v r"
    consider (T) a lex ts' where "s = T a" "ts = (a,lex)#ts'" "v = Leaf
(a, lex)" "r = ts'"
      | (NT) x ss where "s = NT x" "x |∉| vis" "fmlookup pt (x, peek
ts) = Some (x, ss)"
        "parseSymbol pt s ts vis = parseGamma pt x ss ts (finsert x vis)"
      using A(3) by (cases s; cases ts) (auto split: if_splits option.splits)
    then have "∃w. w @ r = ts ∧ sym_derives_prefix g s w v r"
    proof cases
      case T
      then show ?thesis by (auto simp add: T_sdp)
    next
      case NT
      then have "(Inr (pt, x, ss, ts, (finsert x vis)), 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)"
        unfolding mlex_triple_def parse_ind_meas_sym_def parse_ind_meas_sym_list_def
        using A(1) fcard_diff_insert_less by simp
      then obtain w where " w @ r = ts" "gamma_derives_prefix g x ss
w v r"
        using IH[of "Inr (pt, x, ss, ts, (finsert x vis))"] A NT by auto
      then show ?thesis using A(2) NT(1,3) NT_sdp pt_sound_def by fastforce
    qed
  }
  note 1 = this
  {
    fix pt x ss ts vis
    assume A: "y = Inr (pt, x, ss, ts, vis)" "parse_table_correct pt
```

```
g"
        "parseGamma pt x ss ts vis = RESULT v r"
    consider (Nil) "ss = []"
      | (ConsLe) s ss' t tls r1 where "ss = s#ss'"
        "parseSymbol pt s ts vis = RESULT t r1" "length r1 < length ts"
        "parseGamma pt x ss' r1 {||} = RESULT (Node x tls) r" "v = Node
x (t # tls)"
      | (ConsEq) s ss' t tls where "ss = s#ss'" "parseSymbol pt s ts
vis = RESULT t ts"
        "parseGamma pt x ss' ts vis = RESULT (Node x tls) r" "v = Node
x (t # tls)"
    proof (cases ss)
      case Nil
      then show ?thesis by (simp add: that(1))
    next
      case (Cons s ss')
      then show ?thesis
        using parseSymbol_ts_contains_remainder parseGamma_node(2) A(3)
        by (fastforce simp: ConsLe ConsEq split: return_type.splits if_splits)
    qed
    then have "∃w. w @ r = ts ∧ gamma_derives_prefix g x ss w v r"
    proof (cases)
      case Nil
      then show ?thesis using A(3) Nil_gdp by auto
    next
      case ConsLe
      obtain wpre wsuf where "ts = wpre @ wsuf @ r" "wsuf @ r = r1"
        using parseGamma_ts_contains_remainder A(3) parseSymbol_ts_contains_remainder
ConsLe(2,4)
        by metis
      then have "gamma_derives_prefix g x ss' wsuf (Node x tls) r"
        using IH[of "Inr (pt, x, ss', r1, {||})"] A(1,2) ConsLe(3,4)
        unfolding mlex_triple_def parse_ind_meas_sym_list_def
        by (auto simp add: A(1))
      moreover have "sym_derives_prefix g s wpre t (wsuf @ r)"
        using IH[of "Inl (pt, s, ts, vis)"] ‹ts = wpre @ wsuf @ r› ‹wsuf
@ r = r1› A ConsLe(2)
        unfolding mlex_triple_def parse_ind_meas_sym_def parse_ind_meas_sym_list_def
        by (auto simp add: A(1))
      ultimately show ?thesis by (simp add: ConsLe(1,5) Cons_gdp ‹ts
= wpre @ wsuf @ r›)
    next
      case ConsEq
      obtain w where "ts = w @ r" using ConsEq(3) parseGamma_ts_contains_remainder
by
          fastforce
      have "length ss' < length ss" by (cases w) (auto simp add: ‹ts
= w @ r› ConsEq(1))
      then have "gamma_derives_prefix g x ss' w (Node x tls) r"
```

```
      using IH[of "Inr (pt, x, ss', ts, vis)"] A(1,2) ConsEq(3)
      unfolding mlex_triple_def parse_ind_meas_sym_list_def
      by (auto simp add: A(1) ‹ts = w @ r›)
    moreover have "sym_derives_prefix g s [] t ts"
      using IH[of "Inl (pt, s, ts, vis)"] ‹ts = w @ r› A ConsEq(2)
      unfolding mlex_triple_def parse_ind_meas_sym_def parse_ind_meas_sym_list_def
      by (auto simp add: A(1))
    ultimately show ?thesis using ConsEq(1,4) Cons_gdp ‹ts = w @ r›
by fastforce
  qed
 }
 note 2 = this
 then show ?case using 1 2 by (cases y) (auto split: prod.splits)
qed

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"
 using parseSymbol_parseGamma_sound[where x = "Inl (pt, s, (w @ r),
{||})"]
 by auto
```

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 (induction rule: sym_derives_prefix_gamma_derives_prefix.inducts)
 case (T_sdp a r)
 then show ?case by auto
next
 case (NT_sdp x gamma w r v)
 have "fmlookup pt (x,peek (w @ r)) = Some (x, gamma)"
   using NT_sdp.IH(1) NT_sdp.IH(2) assms pt_complete_def by fastforce
 then have "(∃m y ts'. parseSymbol pt (NT x) (w @ r) vis = ERROR m y
ts')
   ∨ parseSymbol pt (NT x) (w @ r) vis = RESULT v r" for vis
 proof (cases "x |∈| vis")
   case True
   then show ?thesis by simp
 next
   case False
   then have "parseSymbol pt (NT x) (w @ r) vis = parseGamma pt x gamma
```

```
(w @ r) (finsert x vis)"
      using ‹fmlookup pt (x, peek (w @ r)) = Some (x, gamma)› by auto
    then show ?thesis using NT_sdp.IH(4) by auto
  qed
  then show ?case by auto
next
  case (Nil_gdp r)
  then show ?case by auto
next
  case (Cons_gdp s wpre v wsuf r n ss vs)
  then have "(∃m x ts'. parseGamma pt n (s # ss) ((wpre @ wsuf) @ r)
vis = ERROR m x ts')
    ∨ parseGamma pt n (s # ss) ((wpre @ wsuf) @ r) vis = RESULT (Node
n (v # vs)) r" for vis
  proof -
    have "∃m x ts'. parseSymbol pt s (wpre @ wsuf @ r) vis = ERROR m
x ts'
      ∨ parseSymbol pt s (wpre @ wsuf @ r) vis = RESULT v (wsuf @ r)"
using Cons_gdp.IH(2) by auto
    moreover have "∃m x ts'. parseGamma pt n ss (wsuf @ r) vis = ERROR
m x ts'
      ∨ parseGamma pt n ss (wsuf @ r) vis = RESULT (Node n vs) r" us-
ing Cons_gdp(4) by auto
    moreover have "∃m x ts'. parseGamma pt n ss (wsuf @ r) {||} = ERROR
m x ts'
      ∨ parseGamma pt n ss (wsuf @ r) {||} = RESULT (Node n vs) r" us-
ing Cons_gdp(4) by auto
    ultimately show ?thesis by (cases "parseSymbol pt s (wpre @ wsuf
@ r) vis") auto
  qed
  then show ?case by auto
qed

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)"
  using parseSymbol_parseGamma_complete_or_error by fastforce
```

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"
  using SzFirstT SzFirstNT
  by -(induction rule: first_sym.induct; fastforce)

lemma sized_fs_fs: "sized_first_sym g la s n ⟹ first_sym g la s"
  by (induction rule: sized_first_sym.induct) (auto simp add: FirstT FirstNT)

lemma medial : "pre @ s # suf = pre' @ s' # suf'
    ⟹ s ∈ set pre'∨ s' ∈ set pre ∨ pre = pre' ∧ s = s' ∧ suf = suf'"
proof (induction pre arbitrary: pre')
  case Nil
  then show ?case by (cases pre') auto
next
  case (Cons a pre)
  then show ?case by (cases pre') auto
qed

lemma nullable_sym_in: "nullable_gamma g gamma ⟹ s ∈ set gamma ⟹
nullable_sym g s"
proof (induction gamma)
  case (Cons s' gamma)
  have "nullable_gamma g gamma" using Cons.prems(1) by (cases rule: nullable_gamma.cases)
  moreover have "nullable_sym g s'" using Cons.prems(1) by (cases rule:
nullable_gamma.cases)
  ultimately show ?case using Cons.IH Cons.prems(2) by (cases "s = s'")
auto
qed simp

lemma nullable_split: "nullable_gamma g (xs @ ys) ⟹ nullable_gamma
g ys"
proof (induction xs arbitrary: ys)
  case Nil
  then show ?case by auto
next
  case (Cons s xs)
  from Cons.prems have "nullable_gamma g (xs @ ys)" by (cases rule: nullable_gamma.cases)
auto
  then show ?case using Cons.IH by auto
qed

lemma first_gamma_split:
  "first_gamma g la ys ⟹ nullable_gamma g xs ⟹ first_gamma g la (xs
@ ys)"
proof (induction xs)
  case Nil
  then show ?case by auto
next
```

```
  case (Cons s xs)
  from Cons.prems(2) have "nullable_gamma g xs" by (cases) auto
  then have "first_gamma g la (xs @ ys)" using Cons.IH Cons.prems(1)
by auto
  then obtain xs' s' ys' where "first_sym g la s'" "nullable_gamma g
xs'" "xs @ ys = xs' @ s' # ys'"
    by (cases rule: first_gamma.cases) auto
  have "nullable_sym g s" using Cons.prems(2) by (cases) auto
  then have "nullable_gamma g (s # xs')" by (simp add: NullableCons ‹nullable_gamma
g xs'›)
  then show ?case using FirstGamma[where gpre = "s # xs'" and s = s'
and gsuf = "ys'"] by
      (simp add: ‹first_sym g la s'› ‹xs @ ys = xs' @ s' # ys'›)
qed

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 goal_cases
  case 1
  then obtain l1 l2 where l1_l2_def: "pre = l1 @ s # l2" by (metis split_list)
  then show ?case
  proof (cases s)
    case (NT x)
    have "nullable_gamma g (l1 @ NT x # l2)" using "1"(3) by (auto simp
add: l1_l2_def NT)
    then have "nullable_gamma g l2" using nullable_split[where xs = "l1
@ [NT x]"] by auto
    then have "first_gamma g la (l2 @ suf)" by (simp add: "1"(4) first_gamma_split)
    then show ?thesis using FollowRight[where gsuf = "l2 @ suf"] NT
"1"(1) l1_l2_def by auto
  next
    case (T t)
    then show ?thesis using "1"(2,3) nullable_sym.cases nullable_sym_in
by blast
  qed
qed

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"
  unfolding not_def
proof (intro impI, induction rule: first_sym.induct)
  case (FirstT y)
  then show ?case by (cases rule: nullable_sym.cases)
next
  case (FirstNT x gpre s gsuf la)
  from FirstNT.prems(1) obtain ys where ys_props: "(x, ys) ∈ set (prods
```

```
g)" "nullable_gamma g ys" by
      cases auto
  have ys_eq: "ys = gpre @ s # gsuf"
  proof -
    have "lookahead_for la x (gpre @ s # gsuf) g"
      by (simp add: FirstGamma FirstNT.hyps(2,3) lookahead_for_def)
    moreover have "lookahead_for la x ys g"
      by (simp add: FirstNT.prems(2) ys_props(2) lookahead_for_def)
    ultimately show ?thesis using assms by
        (metis FirstNT.hyps(1) Pair_inject ys_props(1) option.inject pt_complete_def)
  qed
  have "nullable_sym g s" using ys_props(2) ys_eq nullable_sym_in by
fastforce
  then show ?case
  proof (cases s)
    case (NT x)
    have "nullable_gamma g gsuf" using nullable_split[where xs = "gpre
@ [s]"] ys_props(2) ys_eq by
        auto
    then have "follow_sym g la s" using FirstNT.hyps(1) FirstNT.prems(2)
FollowLeft NT by fastforce
    then show ?thesis using FirstNT.IH ‹nullable_sym g s› by auto
  next
    case (T t)
    show ?thesis using ‹nullable_sym g s› T by (cases rule: nullable_sym.cases)
auto
  qed
qed

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 (goal_cases)
  case 1
  have "pre @ s # suf = pre' @ s' # suf'"
  proof -
    have "lookahead_for la x (pre @ s # suf) g" by (simp add: "1"(4,6)
FirstGamma lookahead_for_def)
    moreover have "lookahead_for la x (pre' @ s' # suf') g" by
        (simp add: "1"(5,7) FirstGamma lookahead_for_def)
    ultimately show ?thesis
      using "1"(1-3) unfolding pt_complete_def by (metis Pair_inject option.inject)
  qed
  then consider (s_in_pre') "s ∈ set pre'" | (s'_in_pre) "s' ∈ set pre"
    | (eq) "pre = pre' ∧ s = s' ∧ suf = suf'" using medial by fastforce
  then show ?case
```

```
  proof cases
    case s_in_pre'
    then have "nullable_sym g s" using "1"(5) using nullable_sym_in by
auto
    moreover have "follow_sym g la s"
    proof -
      have "first_gamma g la (s' # suf')" using "1"(7) FirstGamma[of _
"[]" _ s'] NullableNil by
          auto
      then show ?thesis using "1"(3,5) s_in_pre' follow_pre[where suf
= "s' # suf'"] by auto
    qed
    ultimately show ?thesis using "1"(1,6) no_first_follow_conflicts
by auto
  next
    case s'_in_pre
    then have "nullable_sym g s'" using "1"(4) using nullable_sym_in
by auto
    moreover have "follow_sym g la s'"
    proof -
      have "first_gamma g la (s # suf)" using "1"(6) FirstGamma[of _ "[]"
_ s] NullableNil by auto
      then show ?thesis using "1"(2,4) s'_in_pre follow_pre[where suf
= "s # suf"] by auto
    qed
    ultimately show ?thesis using "1"(1,7) no_first_follow_conflicts
by auto
  next
    case eq
    then show ?thesis by blast
  qed
qed

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 (induction arbitrary: s' rule: sized_first_sym.inducts)
  case (SzFirstT g y)
  then show ?case by (auto, cases rule: sized_first_sym.cases) auto
next
  case (SzFirstNT x pre s suf la n)
  have "s' = NT x ⟶ sized_first_sym g la s' n' ⟶ Suc n = n'" for n'
  proof (intro impI, goal_cases)
    case 1
    obtain gpre gsuf s0 n0 where s0_props: "(x, gpre @ s0 # gsuf) ∈ set
(prods g)"
      "nullable_gamma g gpre" "sized_first_sym g la s0 n0" "n' = Suc n0"
      using 1(1) by (cases rule: sized_first_sym.cases[OF 1(2)]) auto
```

```
    then have "first_sym g la s" "first_sym g la s0" using SzFirstNT.hyps(3)
sized_fs_fs by auto
    then have "s = s0" using first_sym_rhs_eqs SzFirstNT.hyps s0_props
assms by fastforce
    then have "n = n0" using SzFirstNT.IH ‹sized_first_sym g la s0 n0›
by auto
    then show ?case using ‹n' = Suc n0› by auto
  qed
  then show ?case by auto
qed

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 (induction rule: nullable_path.induct)
  case (DirectPath x gamma g gpre z gsuf la)
  then obtain n where "sized_first_sym g la (NT z) n" using first_sym_exists_size
by blast
  then have "sized_first_sym g la (NT x) (Suc n)" using DirectPath.hyps
by (simp add: SzFirstNT)
  then show ?case using ‹sized_first_sym g la (NT z) n› by blast
next
  case (IndirectPath x gamma g gpre y gsuf la z)
  then obtain nx ny where nx_ny_ex:
    "sized_first_sym g la (NT y) nx ∧ sized_first_sym g la (NT z) ny ∧
ny < nx" by auto
  then have "sized_first_sym g la (NT x) (Suc nx)" using IndirectPath.hyps
by (simp add: SzFirstNT)
  then show ?case using nx_ny_ex less_SucI by blast
qed

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"
  by (induction rule: sized_nullable_sym_sized_nullable_gamma.inducts)
    (auto simp add: NullableSym NullableNil NullableCons)
```

```
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"
  using SzNullableSym SzNullableNil SzNullableCons
  by (induction rule: nullable_sym_nullable_gamma.inducts) fastforce+

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 (induction rule: sized_nullable_sym_sized_nullable_gamma.inducts)
  case (SzNullableSym x gamma n)
  from SzNullableSym.prems(2) obtain n0 gamma0 where "(x, gamma0) ∈
set (prods g)"
    "sized_nullable_gamma g gamma0 n0" "n' = Suc n0" by (cases) auto
  have "gamma0 = gamma"
  proof -
    have "lookahead_for la x gamma0 g" unfolding lookahead_for_def
      using sized_ng_ng SzNullableSym.prems(1) ‹sized_nullable_gamma g
gamma0 n0› by blast
    then have lookup_1: "fmlookup pt (x, la) = Some (x, gamma0)"
      using assms ‹(x, gamma0) ∈ set (prods g)› by (auto simp add: pt_complete_def)
    have "lookahead_for la x gamma g" unfolding lookahead_for_def
      using sized_ng_ng SzNullableSym.prems(1) SzNullableSym.IH(2) by
blast
    then have lookup_2: "fmlookup pt (x, la) = Some (x, gamma)" using
assms SzNullableSym.IH(1)
      by (auto simp add: pt_complete_def)
    show ?thesis using lookup_1 lookup_2 by auto
  qed
  then have "n0 = n"
    using ‹sized_nullable_gamma g gamma0 n0› SzNullableSym.IH(3)[of _
"[]"] SzNullableSym.IH(1)
      SzNullableSym.prems(1) by auto
  then show ?case by (simp add: ‹n' = Suc n0›)
next
  case SzNullableNil
  then show ?case by (cases rule: parse.sized_nullable_gamma.cases) auto
next
  case (SzNullableCons s n1 ss n2 x)
  from SzNullableCons.prems(3) obtain n1' n2' where
    "sized_nullable_sym g s n1'" "sized_nullable_gamma g ss n2'" "n' =
n1' + n2'" by (cases) auto
  have "follow_sym g la s"
```

```
  proof (cases s)
    case (NT x1)
    have "nullable_gamma g ss" using sized_ng_ng SzNullableCons.IH(3)
by fastforce
    then show ?thesis using FollowLeft NT SzNullableCons.prems(1,2) by
fastforce
  next
    case (T x2)
    from ‹sized_nullable_sym g s n1› show ?thesis using T by (cases)
auto
  qed
  then have "n1 = n1'" using SzNullableCons.IH(2)[of n1'] ‹sized_nullable_sym
g s n1'› by auto
  moreover have "n2 = n2'"
    using SzNullableCons.IH(4)[of x "gpre @ [s]" n2'] ‹sized_nullable_gamma
g ss n2'›
    by (simp add: SzNullableCons.prems(1,2))
  ultimately show ?case using ‹n' = n1' + n2'› by auto
qed

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'"
  using sized_nullable_sym_det' assms by blast

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 (induction gamma)
  case Nil
  then show ?case by auto
next
  case (Cons a gamma)
  from Cons.prems(1) have nullable: "nullable_sym g a" "nullable_gamma
g gamma" by
    (cases rule: nullable_gamma.cases, auto)+
  then show ?case
  proof (cases "a = s")
    case True
    then show ?thesis using SzNullableCons nullable ng_sized_ng by fastforce
  next
    case False
    then show ?thesis using Cons.IH Cons.prems(2) SzNullableCons nullable
ng_sized_ng(1) by
      fastforce
  qed
qed
```

```
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 -
  obtain n n' where
    "sized_nullable_sym g (NT y) n" "sized_nullable_gamma g (pre @ NT
y # suf) n'" "n ≤ n'"
    using sym_in_gamma_size_le[of g "pre @ NT y # suf" "NT y"] assms(2)
by auto
  then have "sized_nullable_sym g (NT x) (Suc n')"
    using SzNullableSym[where gamma = "pre @ NT y # suf"] assms(1) by
auto
  then show ?thesis using ‹n ≤ n'› ‹sized_nullable_sym g (NT y) n› le_imp_less_Suc
by blast
qed

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 (induction rule: nullable_path.induct)
  case (DirectPath x gamma g pre y suf la)
  from DirectPath.prems(2) obtain ys where "(x, ys) ∈ set (prods g)"
"nullable_gamma g ys" by
      (cases rule: nullable_sym.cases) auto
  from DirectPath.hyps(4) consider (fst_gamma) "first_gamma g la gamma"
    | (null_gamma) "nullable_gamma g gamma" "follow_sym g la (NT x)"
    unfolding lookahead_for_def by auto
  then show ?case
  proof cases
    case fst_gamma
    then have "first_sym g la (NT x)" using DirectPath.hyps(1) FirstNT
by cases auto
    then have "¬ follow_sym g la (NT x)" using DirectPath.prems(1,2)
by
      (auto simp add: no_first_follow_conflicts)
    then show ?thesis using DirectPath.prems(3) by auto
  next
    case null_gamma
    have "nullable_sym g (NT y)" using DirectPath.hyps(2) null_gamma(1)
nullable_sym_in by fastforce
    then show ?thesis using DirectPath.hyps(1,2) null_gamma(1) parse.sized_ns_np
by fastforce
  qed
```

```
next
  case (IndirectPath x gamma g gpre y gsuf la z)
  from IndirectPath.hyps(4) consider (fst_gamma) "first_gamma g la gamma"
    | (null_gamma) "nullable_gamma g gamma" "follow_sym g la (NT x)"
    unfolding lookahead_for_def by auto
  then show ?case
  proof cases
    case fst_gamma
    then have "first_sym g la (NT x)" using IndirectPath.hyps(1) FirstNT
by cases auto
    then have "¬ follow_sym g la (NT x)" using IndirectPath.prems(1,2)
by
        (auto simp add: no_first_follow_conflicts)
    then show ?thesis using IndirectPath.prems(3) by auto
  next
    case null_gamma
    have "nullable_sym g (NT y)" using IndirectPath.hyps(2) null_gamma(1)
nullable_sym_in by
        fastforce
    have "nullable_gamma g gsuf"
      using null_gamma(1) IndirectPath.hyps(2) nullable_split[where xs
= "gpre @ [NT y]"] by auto
    then have "follow_sym g la (NT y)" using null_gamma IndirectPath.hyps(1,2)
by
        (auto simp add: FollowLeft)
    then obtain ny nz where
      "sized_nullable_sym g (NT y) ny" "sized_nullable_sym g (NT z) nz"
"nz < ny"
      using IndirectPath.IH IndirectPath.prems(1) ‹nullable_sym g (NT
y)› by auto
    obtain nx ny' where
      "sized_nullable_sym g (NT x) nx" "sized_nullable_sym g (NT y) ny'"
"ny' < nx"
      using IndirectPath.hyps(1,2) null_gamma(1) sized_ns_np ‹nullable_sym
g (NT y)› by fastforce
    then have "ny = ny'" using IndirectPath.prems(1) ‹follow_sym g la
(NT y)›
        ‹sized_nullable_sym g (NT y) ny› sized_nullable_sym_det by blast
    then have "nz < nx" using ‹ny' < nx› ‹nz < ny› by auto
    then show ?thesis  using ‹sized_nullable_sym g (NT x) nx› ‹sized_nullable_sym
g (NT z) nz› by
        auto
  qed
qed

lemma nullable_path_exists_production: "nullable_path g la (NT x) y
  ⟹ ∃gamma. (x, gamma) ∈ set (prods g) ∧ lookahead_for la x gamma g"
  by (cases rule: nullable_path.cases) auto
```

```
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
  assume x_left_rec: "left_recursive g (NT x) la"
  then obtain gamma where "(x, gamma) ∈ set (prods g)" "lookahead_for
la x gamma g"
    using nullable_path_exists_production by fastforce
  then consider "first_gamma g la gamma" | "nullable_gamma g gamma" "follow_sym
g la (NT x)"
    by (auto simp: lookahead_for_def)
  then show "False"
  proof (cases)
    case 1
    then have "first_sym g la (NT x)" using FirstNT ‹(x, gamma) ∈ set
(prods g)› by (cases) auto
    then obtain nx ny where "sized_first_sym g la (NT x) nx" "sized_first_sym
g la (NT x) ny"
      "ny < nx" using sized_first_sym_np x_left_rec by blast
    then have "ny = nx" using assms sized_first_sym_det by fastforce
    then show ?thesis using ‹ny < nx› by auto
  next
    case 2
    then have "nullable_sym g (NT x)" using ‹(x, gamma) ∈ set (prods
g)› by
        (auto simp add: NullableSym)
    then obtain n n' where "sized_nullable_sym g (NT x) n" "sized_nullable_sym
g (NT x) n'" "n' < n"
      using "2"(2) x_left_rec assms exist_decreasing_nullable_sym_sizes_in_null_path
by blast
    then show ?thesis using "2"(2) assms sized_nullable_sym_det by blast
  qed
qed

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 (induction arbitrary: v r rule: parse_meas_induct)
  case (1 y)
  note IH = 1
  then show ?case
  proof (cases y)
    case (Inl a)
    then obtain pt s ts vis where "a = (pt, s, ts, vis)" by (cases y)
auto
```

```
    have "length r < length ts ∨ nullable_sym g s"
      if "parseSymbol pt s ts vis = RESULT v r" and "parse_table_correct
pt g"
    proof (cases s)
      case (NT x)
      from that obtain ss where parse_ss_simp: "x |∉| vis" "fmlookup
pt (x, peek ts) = Some (x, ss)"
        "parseGamma pt x ss ts (finsert x vis) = RESULT v r"
        by (auto simp: NT split: if_splits option.splits)
      then have "(x, ss) ∈ set (prods g)" using that(2) by (auto simp
add: pt_sound_def)
      have "fcard (nt_from_pt pt |-| finsert x vis) < fcard (nt_from_pt
pt |-| vis)"
        using fcard_diff_insert_less parse_ss_simp(1,2) by fastforce
      then have "length r < length ts ∨ nullable_gamma g ss"
        using IH[of "Inr (pt, x, ss, ts, finsert x vis)"] parse_ss_simp(3)
        unfolding mlex_triple_def parse_ind_meas_sym_def parse_ind_meas_sym_list_def
        by (auto simp: Inl ‹a = (pt, s, ts, vis)› that(2))
      then show ?thesis using NT NullableSym ‹(x, ss) ∈ set (prods g)›
by force
    next
      case (T t)
      from that obtain s where "ts = (t, s) # r"
        by (cases ts, auto simp: T split: if_splits option.splits)
      then show ?thesis by simp
    qed
    then show ?thesis by (simp add: Inl ‹a = (pt, s, ts, vis)›)
  next
    case (Inr a)
    then obtain pt x ss ts vis where "a = (pt, x, ss, ts, vis)" by (cases
y) auto
    have "length r < length ts ∨ nullable_gamma g ss" if
      "parseGamma pt x ss ts vis = RESULT v r" "parse_table_correct pt
g"
    proof (cases ss)
      case Nil
      then show ?thesis by (simp add: NullableNil)
    next
      case (Cons s ss')
      from that(1) consider v0 v1 where "parseSymbol pt s ts vis = RESULT
v0 ts"
        "parseGamma pt x ss' ts vis = RESULT (Node x v1) r" "v = Node
x (v0 # v1)"
      | v0 v1 r0 where "parseSymbol pt s ts vis = RESULT v0 r0" "length
r0 < length ts"
        "parseGamma pt x ss' r0 {||} = RESULT (Node x v1) r" "v = Node
x (v0 # v1)"
        using parseSymbol_ts_contains_remainder[of pt s ts vis] Cons parseGamma_node(2)[of
pt x ss']
```

```
      by (fastforce split: return_type.splits parse_tree.splits if_splits)
    then show ?thesis
    proof cases
      case 1
      then have "length ts < length ts ∨ nullable_sym g s"
        using IH[of "Inl (pt, s, ts, vis)"] that(2)
        unfolding mlex_triple_def parse_ind_meas_sym_def parse_ind_meas_sym_list_def
        by (auto simp add: Inr ‹a = (pt, x, ss, ts, vis)›)
      moreover have "length r < length ts ∨ nullable_gamma g ss'"
        using IH[of "Inr (pt, x, ss', ts, vis)"] that(2) 1
        unfolding mlex_triple_def parse_ind_meas_sym_def parse_ind_meas_sym_list_def
        by (auto simp add: Inr ‹a = (pt, x, ss, ts, vis)› Cons)
      ultimately show ?thesis using NullableCons Cons by blast
    next
      case 2
      have "length r ≤ length r0" using parseGamma_ts_contains_remainder
"2"(3) by fastforce
      then show ?thesis using "2"(2) by auto
    qed
  qed
  then show ?thesis by (simp add: Inr ‹a = (pt, x, ss, ts, vis)›)
 qed
qed

lemma input_length_eq_nullable_sym:
 "parse_table_correct tbl g ⟹ parseSymbol tbl s ts vis = RESULT v ts
⟹ nullable_sym g s"
 using input_length_lt_or_nullable_sym[where x = "Inl (tbl, s, ts, vis)"]
by auto

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 (induction arbitrary: m z ts' rule: parse_meas_induct)
 case (1 y)
 note IH = "1"
 then show ?case
 proof (cases y)
   case (Inl a)
   then obtain pt s ts vis where "a = (pt, s, ts, vis)" using prod_cases4
by blast
```

```
    then show ?thesis
    proof (cases s)
      case (NT x)
      consider (x_in_vis) "x |∈| vis" | (lookup_none) "fmlookup pt (x,
peek ts) = None"
        | (lookup_some_neq) x' ss' where "x |∉| vis" "x ≠ x'"
          "fmlookup pt (x, peek ts) = Some (x',ss')"
        | (lookup_some_eq )ss' where "x |∉| vis" "fmlookup pt (x, peek
ts) = Some (x, ss')"
        by fastforce
      then show ?thesis
      proof (cases)
        case lookup_some_eq
        then have "fcard (nt_from_pt pt |-| finsert x vis) < fcard (nt_from_pt
pt |-| vis)"
          using fcard_diff_insert_less[of x] by auto
        then have IH': "parse_table_correct pt g
            ⟹ parseGamma pt x ss' ts (finsert x vis) = ERROR m z ts'
            ⟹ ((∃pre s suf. ss' = pre @ s # suf ∧ nullable_gamma g
pre ∧ z |∈| (finsert x vis)
            ∧ (s = NT z ∨ nullable_path g (peek ts) s (NT z)))
            ∨ (∃la. left_recursive g (NT z) la))"
          using IH[of "Inr (pt, x, ss', ts, (finsert x vis))" m z ts']
          unfolding mlex_triple_def parse_ind_meas_sym_def parse_ind_meas_sym_list_def
          by (auto simp add: Inl_<a = (pt, s, ts, vis)>)
        have "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)"
        proof -
          assume assms: "parse_table_correct pt g" "parseSymbol pt s
ts vis = ERROR m z ts'"
          have "parseGamma pt x ss' ts (finsert x vis) = ERROR m z ts'"
            using assms(2) lookup_some_eq NT by auto
          then consider pre s' suf where "ss' = pre @ s' # suf" "nullable_gamma
g pre"
            "z |∈| (finsert x vis)" "s' = NT z ∨ nullable_path g (peek
ts) s' (NT z)"
          | la where "left_recursive g (NT z) la" using IH' assms(1,2)
by auto
          then show "z |∈| vis ∧ (s = NT z ∨ nullable_path g (peek ts)
s (NT z))
            ∨ (∃la. left_recursive g (NT z) la)"
          proof (cases)
            case 1
            have "fmlookup pt (x,(peek ts)) = Some (x, pre @ s' # suf)"
              using "1"(1) lookup_some_eq(2) by blast
            then have x_la: "(x, pre @ s' # suf) ∈ set (prods g)"
```

```
            "lookahead_for (peek ts) x (pre @ s' # suf) g" using ‹parse_table_correct
pt g›
            by (simp_all add: pt_sound_def)
          consider (s'_is_ntz) "s' = NT z" | (null_path) "nullable_path
g (peek ts) s' (NT z)"
            using "1"(4) by auto
          then show ?thesis
          proof (cases)
            case s'_is_ntz
            then show ?thesis using "1"(2-4) DirectPath NT  x_la by
fastforce
          next
            case null_path
            then obtain n n' where "s' = NT n" "NT x = NT n'" by cases
auto
            then show ?thesis using "1"(2,3) IndirectPath null_path
x_la NT by fastforce
          qed
        next
          case 2
          then show ?thesis by blast
        qed
      qed
      then show ?thesis by (simp add: Inl ‹a = (pt, s, ts, vis)›)
    qed (auto simp add: Inl NT ‹a = (pt, s, ts, vis)› pt_sound_def)
  next
    case (T _)
    then show ?thesis using Inl T ‹a = (pt, s, ts, vis)› by (cases
ts) auto
  qed
next
  case (Inr a)
  then obtain pt x ss ts vis where "a = (pt, x, ss, ts, vis)" using
prod_cases5 by blast
  have "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 -
    assume assms: "parse_table_correct pt g" "parseGamma pt x ss ts
vis = ERROR m z ts'"
    then obtain s ss' where "ss = s # ss'" by (auto elim: parseGamma.elims)
    then consider (parse_s_error) "parseSymbol pt s ts vis = ERROR m
z ts'"
      | (parse_ss'_error_le) str r where "parseSymbol pt s ts vis =
RESULT str r"
        "length r < length ts" "parseGamma pt x ss' r {||} = ERROR m
```

```
z ts'"
        | (parse_ss'_error_eq) str where "parseSymbol pt s ts vis = RESULT
str ts"
          "parseGamma pt x ss' ts vis = ERROR m z ts'"
        using assms(2) parseSymbol_ts_contains_remainder
        by (fastforce split: return_type.splits if_splits parse_tree.splits)
      then show "(∃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 cases
        case parse_s_error
        have "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)"
          using IH[of "Inl (pt, s, ts, vis)" m z ts'] assms
          unfolding mlex_triple_def parse_ind_meas_sym_def parse_ind_meas_sym_list_def
          by (auto simp add: Inr ‹a = (pt, x, ss, ts, vis)›)
        then show ?thesis using NullableNil ‹ss = s # ss'› assms(1) parse_s_error
by blast
      next
        case parse_ss'_error_le
        then have "(parse_ind_meas_sym_list pt ss' r {||},
          parse_ind_meas_sym_list pt ss ts vis) ∈ lex_triple less_than
less_than less_than"
          unfolding parse_ind_meas_sym_def parse_ind_meas_sym_list_def
          using ‹ss = s # ss'› parse_ss'_error_le(2) by auto
        then show ?thesis using IH[of "Inr (pt, x, ss', r, {||})" m z]
assms parse_ss'_error_le(3)
          by (auto simp add: mlex_triple_def Inr ‹a = (pt, x, ss, ts,
vis)›)
      next
        case parse_ss'_error_eq
        have "(parse_ind_meas_sym_list pt ss' ts vis,
          parse_ind_meas_sym_list pt ss ts vis) ∈ lex_triple less_than
less_than less_than"
          unfolding parse_ind_meas_sym_def parse_ind_meas_sym_list_def
          using ‹ss = s # ss'› by auto
        then consider pre s0 suf where "ss' = pre @ s0 # suf" "nullable_gamma
g pre" "z |∈| vis"
          "(s0 = NT z ∨ nullable_path g (peek ts) s0 (NT z))"
        | la where "left_recursive g (NT z) la"
          using IH[of "Inr (pt, x, ss', ts, vis)" m z ts'] assms parseSymbol_ts_contains_re
          by (auto simp add: mlex_triple_def Inr ‹a = (pt, x, ss, ts,
vis)› parse_ss'_error_eq(2))
        then show ?thesis
        proof cases
```

```
          case 1
          have "nullable_sym g s"
            using input_length_eq_nullable_sym assms(1) parse_ss'_error_eq(1)
by fastforce
          then have "nullable_gamma g (s # pre)" by (simp add: "1"(2)
NullableCons)
          then show ?thesis by (metis "1"(1,3,4) Cons_eq_appendI ‹ss
= s # ss'›)
        next
          case 2
          then show ?thesis by auto
        qed
      qed
    qed
    then show ?thesis by (simp add: Inr ‹a = (pt, x, ss, ts, vis)›)
  qed
qed

theorem parse_terminates_without_error:
  "parse_table_correct pt g ⟹ parse (PT pt) s (w @ r) ≠ ERROR m x ts'"
proof
  assume A: "parse_table_correct pt g" "parse (PT pt) s (w @ r) = ERROR
m x ts'"
  then have "¬ (∃la. left_recursive g (NT x) la)" using ll1_parse_table_impl_no_left_recur
    using ‹parse_table_correct pt g› by fastforce
  then have "(x |∈| {||} ∧ (s = NT x ∨ nullable_path g (peek (w @ r))
s (NT x)))"
    using error_conditions[of g m x ts' "Inl (pt, s, (w @ r), {||})"]
A by auto
  then show "False" by auto
qed

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"
  using parse_terminates_without_error parse_complete_or_error by fastforce

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
  done

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]]]))
     []"
  by eval
```

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, ())''"
  by eval
```

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])"
  by eval

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)])
[]"
by *eval*

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])
    []"
  by eval
```

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 [])]"
  by eval

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)]])]"
```
by *eval*

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.