

Exercises for Language Processors (WS2001-2002) *

Prof. Dr. Peter Thiemann
Simon Helsen

3 Model Solution

3.1 LL(0) grammars

What languages are described by LL(0) grammars? Explain.

Languages produced by LL(0) grammars only contain *one* element! Indeed, by definition, a grammar is LL(0) iff for productions $A \rightarrow \alpha$ and $A \rightarrow \beta$ where $\alpha \neq \beta$ it follows that

$$(first_0(\alpha) \oplus_0 follow_0(A)) \cap (first_0(\beta) \oplus_0 follow_0(A)) = \emptyset$$

Since the latter is not possible, there are no productions $A \rightarrow \alpha$ and $A \rightarrow \beta$ where $\alpha \neq \beta$! So, there is only one path to reach a final state, which specifies the one word belonging to the language!

3.2 The first and follow functions

This exercise requires the module `grammar.ml` (with `grammar.mli`) found at:

<http://www.informatik.uni-freiburg.de/proglang/teaching/ws2001-2002/langproc>

1. Implement the $first_k$ function. The following is given:

```
let first grammar k =  
  let table = first_map grammar k in  
  lookup_first grammar k table
```

so, we have to implement the following functions:

*©2002: it is not permitted to lend, distribute or copy any of these exercises and their model solutions either by paper or in electronic form other than for personal use. The authors cannot be held responsible for mistakes or inaccuracies in either formulation or solution of the exercises.

```

lookup_first : ('n, 't, 'attrib) grammar -> int -> assocTable
              -> ('n, 't) symbol list -> 't list list
initial_first_map : ('n, 't, 'attrib) grammar -> assocTable
lhs_next_first : 'n -> ('n, 't, 'attrib) grammar -> int
                -> assocTable -> 't list list
next_first_map : ('n, 't, 'attrib) grammar -> int
                -> assocTable -> assocTable
first_map : ('n, 't, 'attrib) grammar -> int -> assocTable

```

Here follows the implementation:

```

let lookup_first g k map l =
  let rec loop rhs_rest =
    match rhs_rest with
    [] -> [[]]
  | symbol::rest ->
    let rest_first = loop rest in
    match symbol with
    T(t) -> uniq (List.map (append_truncate k [t]) rest_first)
  | NT(n) ->
    union_list
      (List.map
        (function first_rest ->
          List.map
            (function first_first ->
              (append_truncate k first_first first_rest))
            (List.assoc n map))
          rest_first)
    | Error -> []
  in loop l

let initial_first_map g =
  List.map (function n -> (n, [])) g.nonterminals

let lhs_next_first lhs g k old_map =
  let rec loop rules first =
    match rules with
    [] -> first
  | rule::rest ->
    let rhs_first = lookup_first g k old_map (rule_rhs rule) in
    loop rest (union first rhs_first)
  in loop (rules_with_lhs g lhs) []

let next_first_map g k old_map =
  List.map
    (function (nonterminal, _) ->
      (nonterminal, lhs_next_first nonterminal g k old_map))
    old_map

let first_equal first_1 first_2 =

```

```

(List.length first_1 = List.length first_2) &&
let rec loop first_1 =
  match first_1 with
  [] -> true
  | x::xs -> (List.mem x first_2) && (loop xs)
in loop first_1

let map_equal map_1 map_2 =
  let rec loop map_1 =
    match map_1 with
    [] -> true
    | (nonterminal_1, first_1)::rest ->
      let first_2 = List.assoc nonterminal_1 map_2 in
      (first_equal first_1 first_2) && (loop rest)
  in loop map_1

let first_map g k =
  let rec loop map =
    let new_map = next_first_map g k map in
    if map_equal map new_map
    then new_map
    else loop new_map
  in loop (initial_first_map g)

```

2. **Optional:** Implement the $follow_k$ function. We proceed again by implementing a set of auxiliaries:

```

initial_follow_map : ('n, 't, 'attrib) grammar -> assocTable
next_follow_map : ('n, 't, 'attrib) grammar -> int
                 -> firstFunctionType
                 -> assocTable -> assocTable
update_follow_map : assocTable -> 'n -> 't list list -> assocTable
follow_map : ('n, 't, 'attrib) grammar -> int
            -> assocTable -> assocTable

```

The actual implementation is not very difficult now:

```

let initial_follow_map g =
  List.map
    (function n ->
      if n = g.start then (n, [[]]) else (n, []))
    g.nonterminals

let update_follow_map map nonterminal follow_symbol =
  let rec loop map =
    match map with
    ((nonterminal', follow) as entry)::rest ->
      if nonterminal = nonterminal'
      then (nonterminal', union follow_symbol follow)::rest
      else entry::(loop rest)

```

```

in loop map

let next_follow_map g k first_g_k old_map =
  let rec loop rules old_map =
    match rules with
    [] -> old_map
  | rule::rest ->
    let rec rhs_loop rhs_rest old_map =
      match rhs_rest with
      [] -> loop rest old_map
    | symbol::rhs_rest ->
      match symbol with
      T(t) -> rhs_loop rhs_rest old_map
    | NT(n) ->
      let first_rest = first_g_k rhs_rest in
      let follow_lhs =
        List.assoc (rule_lhs rule) old_map in
      let follow_symbol =
        uniq (pair_map (append_truncate k) first_rest follow_lhs)
      in
      rhs_loop
        rhs_rest
        (update_follow_map old_map n follow_symbol)
    | Error -> rhs_loop rhs_rest old_map
  in rhs_loop (rule_rhs rule) old_map
in loop g.rules old_map

let follow_map g k =
  let rec loop map =
    let new_map = next_follow_map g k (first g k) map in
    if map_equal map new_map
    then new_map
    else loop new_map
  in loop (initial_follow_map g)

let follow g k =
  let map = follow_map g k in
  function n -> List.assoc n map

```

3.3 LL(1) grammars

We first eliminate left-recursion:

$$\begin{aligned}
\langle \text{bexpr} \rangle & ::= \langle \text{bterm} \rangle \langle \text{bexpr}' \rangle \\
\langle \text{bexpr}' \rangle & ::= \epsilon \mid \text{or } \langle \text{bexpr} \rangle \\
\langle \text{bterm} \rangle & ::= \langle \text{bfactor} \rangle \langle \text{bterm}' \rangle \\
\langle \text{bterm}' \rangle & ::= \epsilon \mid \text{and } \langle \text{bterm} \rangle \\
\langle \text{bfactor} \rangle & ::= \text{not } \langle \text{bfactor} \rangle \mid (\langle \text{bexpr} \rangle) \mid \text{true} \mid \text{false}
\end{aligned}$$

For each (new) production, we calculate the $first_1$ and $follow_1$ sets, and using

these, the LLA_1 lookahead function of each production.

| $A ::= \alpha$ | $follow_1(A)$ | $first_1(\alpha)$ | $(first_1(\alpha) \oplus_1 follow_1(A))$ |
|---|--|----------------------------------|--|
| $\langle \text{bexpr} \rangle ::= \langle \text{bterm} \rangle \langle \text{bexpr}' \rangle$ | $\{\epsilon,)\}$ | $\{\text{true, false, not, (}\}$ | $\{\text{true, false, not, (}\}$ |
| $\langle \text{bexpr}' \rangle ::= \epsilon$ | $\{\epsilon,)\}$ | $\{\epsilon\}$ | $\{\epsilon,)\}$ |
| $\langle \text{bexpr}' \rangle ::= \text{or } \langle \text{bexpr} \rangle$ | $\{\epsilon,)\}$ | $\{\text{or}\}$ | $\{\text{or}\}$ |
| $\langle \text{bterm} \rangle ::= \langle \text{bfactor} \rangle \langle \text{bterm}' \rangle$ | $\{\epsilon, \text{or},)\}$ | $\{\text{true, false, not, (}\}$ | $\{\text{true, false, not, (}\}$ |
| $\langle \text{bterm}' \rangle ::= \epsilon$ | $\{\epsilon, \text{or},)\}$ | $\{\epsilon\}$ | $\{\epsilon, \text{or},)\}$ |
| $\langle \text{bterm}' \rangle ::= \text{and } \langle \text{bterm} \rangle$ | $\{\epsilon, \text{or},)\}$ | $\{\text{and}\}$ | $\{\text{and}\}$ |
| $\langle \text{bfactor} \rangle ::= \text{not } \langle \text{bfactor} \rangle$ | $\{\epsilon, \text{or}, \text{and},)\}$ | $\{\text{not}\}$ | $\{\text{not}\}$ |
| $\langle \text{bfactor} \rangle ::= (\langle \text{bexpr} \rangle)$ | $\{\epsilon, \text{or}, \text{and},)\}$ | $\{(}\}$ | $\{(}\}$ |
| $\langle \text{bfactor} \rangle ::= \text{true}$ | $\{\epsilon, \text{or}, \text{and},)\}$ | $\{\text{true}\}$ | $\{\text{true}\}$ |
| $\langle \text{bfactor} \rangle ::= \text{false}$ | $\{\epsilon, \text{or}, \text{and},)\}$ | $\{\text{false}\}$ | $\{\text{false}\}$ |

The first-sets are read immediately from the grammar. For the follow-sets, we simply obtain a set of equations (also by inspecting the grammar) and solve them (ie. search their fixpoint).

$$\begin{aligned}
follow_1(\langle \text{bexpr} \rangle) &= follow_1(\langle \text{bexpr}' \rangle) \cup \{\epsilon\} \cup \{)\} \\
follow_1(\langle \text{bexpr}' \rangle) &= follow_1(\langle \text{bexpr} \rangle) \\
follow_1(\langle \text{bterm} \rangle) &= follow_1(\langle \text{bterm}' \rangle) \cup follow_1(\langle \text{bexpr} \rangle) \cup first_1(\langle \text{bexpr}' \rangle) \\
follow_1(\langle \text{bterm}' \rangle) &= follow_1(\langle \text{bterm} \rangle) \\
follow_1(\langle \text{bfactor} \rangle) &= first_1(\langle \text{bterm}' \rangle) \cup follow_1(\langle \text{bterm}' \rangle)
\end{aligned}$$

It is now quite easy to see from the above table that every non-terminal left-hand-side has a unique LLA_1 -set to choose the appropriate right-hand-side. Hence, the grammar is $LL(1)$.