CS 314 Principles of Programming Languages

Lecture 8

Zheng Zhang

Department of Computer Science
Rutgers University

Friday 30th September, 2016
Class Information

- Homework 3 due this coming Wednesday, 11:55pm EDT.
- First project will be announced next week or the week after next week. Stay tuned.
Basic Idea:

- The parse tree is constructed from the root, expanding **non-terminal** nodes on the tree’s frontier following a left-most derivation.
- The input program is read from left to right, and input tokens are read (consumed) as the program is parsed.
- The next **non-terminal** symbol is replaced by one of its rules. The particular choice has to be unique, and uses parts of the input (partially parsed program), for instance the first **token** of the remaining input.
Review: LL(1) Grammar

Define $FIRST^+(\delta)$ for rule $A ::= \delta$

- $FIRST(\delta) - \{\epsilon\} \cup \text{Follow}(A)$, if $\epsilon \in FIRST(\delta)$
- $FIRST(\delta)$ otherwise

A grammar is LL(1) iff for any pair of rules that correspond to the same non-terminal

$(A ::= \alpha$ and $A ::= \beta$) implies

$FIRST^+(\alpha) \cap FIRST^+(\beta) = \emptyset$
Basic idea:

For any two productions $A ::= \alpha | \beta$, we would like a distinct way of choosing the correct production to expand.

For some rhs $\alpha \in G$, define $\text{FIRST}(\alpha)$ as the set of tokens that appear as the first symbol in some string derived from $\alpha$. That is

$$x \in \text{FIRST}(\alpha) \iff \alpha \Rightarrow^* x\gamma \text{ for some } \gamma, \text{ and}$$

$$\epsilon \in \text{FIRST}(\alpha) \iff \alpha \Rightarrow^* \epsilon$$

For a non-terminal $A$, define $\text{FOLLOW}(A)$ as the set of terminals that can appear immediately to the right of $A$ in some sentential form.

Thus, a non-terminal’s FOLLOW set specifies the tokens that can legally appear after it. A terminal symbol has no FOLLOW set.

FIRST and FOLLOW sets can be constructed automatically
FIRST set construction

For a string of grammar symbols \( \alpha \), define FIRST(\( \alpha \)) as

- the set of terminal symbols that begin strings derived from \( \alpha \)
- if \( \alpha \Rightarrow^* \epsilon \), then \( \epsilon \in \text{FIRST}(\alpha) \)

FIRST(\( \alpha \)) contains the set of tokens valid in the first position of \( \alpha \)

**STEP 1:** Build FIRST(\( X \)) for all grammar symbols \( X \):

1. if \( X \) is a terminal, FIRST(\( X \)) is \{X\}
2. if \( X ::= \epsilon \), then \( \epsilon \in \text{FIRST}(X) \)
3. iterate until no more terminals or \( \epsilon \) can be added to any FIRST(\( X \)):
   - if \( X ::= Y_1 Y_2 \cdots Y_k \) then
     - \( a \in \text{FIRST}(X) \) if \( a \in \text{FIRST}(Y_1) \) OR \( a \in \text{FIRST}(Y_i) \) and \( \epsilon \in \text{FIRST}(Y_j) \) for all \( 1 \leq j < i \)
     - \( \epsilon \in \text{FIRST}(X) \) if \( \epsilon \in \text{FIRST}(Y_i) \) for all \( 1 \leq i \leq k \)
   - end iterate

(If \( \epsilon \notin \text{FIRST}(Y_1) \), then FIRST(\( Y_i \)) is irrelevant, for \( 1 < i \))

Zheng Zhang
**The FIRST set**

**STEP 2:** Build \( \text{FIRST}(\alpha) \) for \( \alpha = X_1X_2 \cdots X_n \):

- \( a \in \text{FIRST}(\alpha) \) if \( a \in \text{FIRST}(X_i) \) OR \( \text{FIRST}(X_i) \) AND \( \epsilon \in \text{FIRST}(X_j) \) for all \( 1 \leq j < i \)
- \( \epsilon \in \text{FIRST}(\alpha) \) if \( \epsilon \in \text{FIRST}(X_i) \) for all \( 1 \leq i \leq n \)
**FOLLOW set construction**

For a non-terminal \( A \), define \( \text{FOLLOW}(A) \) as the set of terminals that can appear immediately to the right of \( A \) in some sentential form.

Thus, a non-terminal’s \( \text{FOLLOW} \) set specifies the tokens that can legally appear after it. A terminal symbol has no \( \text{FOLLOW} \) set.

To build \( \text{FOLLOW}(X) \) for non-terminal \( X \):

1. place \text{eof} in \( \text{FOLLOW}(⟨\text{start}⟩) \)

   iterate until no more terminals or \( \epsilon \) can be added to any \( \text{FOLLOW}(X) \):

2. if \( A ::= αBβ \) then
   
   if \( \epsilon \in \text{FIRST}(β) \)
   
   put \{\text{FIRST}(β) − \epsilon\} in \( \text{FOLLOW}(B) \)
   
   put \( \text{FOLLOW}(A) \) in \( \text{FOLLOW}(B) \)
   
   else
   
   put \{\text{FIRST}(β)\} in \( \text{FOLLOW}(B) \)

3. if \( A ::= αB \) then
   
   put \( \text{FOLLOW}(A) \) in \( \text{FOLLOW}(B) \)

   end iterate
Recursive descent LL(1) parsing is one of the simplest parsing techniques used in practical compilers:

- Each non–terminal has an associated parsing procedure that can recognize any sequence of tokens generated by that non–terminal.
- There is a main routine to initialize all globals (e.g.: token) and call the start symbol. On return, check whether token == eof, and whether errors occurred.
- Within a parsing procedure, both non–terminals and terminals can be matched:
  - non–terminal $A$ — call parsing procedure for $A$
  - token $t$ — compare $t$ with current input token; if match, consume input, otherwise ERROR
- Parsing procedures may contain code that performs some useful “computation” (syntax directed translation).
Recursive Descent Parsing (pseudo code)

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>eof</th>
<th>other</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>aSb</td>
<td>ɛ</td>
<td>ɛ</td>
<td>error</td>
</tr>
</tbody>
</table>

main: {
    token := next_token( );
    if (S( ) and token == eof) print 'accept' else print 'error';
}

bool S( ):
    switch token {
    case a: token := next_token( );
        call S( );
        if token == b {
            token := next_token( )
            return true;
        } else
            return false;
        break;
    case eof: return true;
    case b: return true;
        break;
    default: return false;
    }

How to parse input a a a b b b ?
Example: Simple Calculator Language

\[
\begin{align*}
program & \rightarrow \ stmt\_list \ \$$ \\
stmt\_list & \rightarrow \ stmt\ stmt\_list \mid \epsilon \\
stmt & \rightarrow \ id \ := \ expr \mid \ read\ id \mid \ write\ expr \\
expr & \rightarrow \ term\ term\_tail \\
term\_tail & \rightarrow \ add\_op\ term\ term\_tail \mid \epsilon \\
term & \rightarrow \ factor\ factor\_tail \\
factor\_tail & \rightarrow \ mult\_op\ factor\ factor\_tail \mid \epsilon \\
factor & \rightarrow \ ( \ expr \ ) \mid id \mid number \\
add\_op & \rightarrow + \mid - \\
mult\_op & \rightarrow * \mid / 
\end{align*}
\]
Example: Simple Calculator Language

First sets without $\epsilon$ and follow sets

**FIRST**

- program \{id, read, write, $\$$\}
- stmt_list \{id, read, write\}
- stmt \{id, read, write\}
- expr \{(+, id, number\}
- term_tail \{+, -\}
- term \{(+, id, number\}
- factor_tail \{*, /\}
- factor \{(+, id, number\}
- add_op \{+, -\}
- mult_op \{*, /\}

**FOLLOW**

- program $\emptyset$
- stmt_list \{$\$$\}
- stmt \{id, read, write, $\$$\}
- expr \{(+, id, read, write, $\$$\}
- term_tail \{(+, id, read, write, $\$$\}
- term \{+, -, id, read, write, $\$$\}
- factor_tail \{+, -, id, read, write, $\$$\}
- factor \{+, -, *, /, id, read, write, $\$$\}
- add_op \{(+, id, number\}
- mult_op \{(+, id, number\}

**PREDICT**

1. program $\rightarrow$ stmt_list $\$$ \{id, read, write, $\$$\}
2. stmt_list $\rightarrow$ stmt stmt_list \{id, read, write\}
3. stmt_list $\rightarrow$ $\epsilon$ \{$\$$\}
4. stmt $\rightarrow$ id := expr \{id\}
5. stmt $\rightarrow$ read id \{read\}
6. stmt $\rightarrow$ write expr \{write\}
7. expr $\rightarrow$ term term_tail \{(+, id, number\}
8. term_tail $\rightarrow$ add_op term term_tail \{+, -\}
9. term_tail $\rightarrow$ $\epsilon$ \{(+, id, read, write, $\$$\}
10. term $\rightarrow$ factor factor_tail \{(+, id, number\}
11. factor_tail $\rightarrow$ mult_op factor factor_tail \{*, /\}
12. factor_tail $\rightarrow$ $\epsilon$ \{+, -, id, read, write, $\$$\}
13. factor $\rightarrow$ ( expr \{()\}
14. factor $\rightarrow$ id \{id\}
15. factor $\rightarrow$ number \{number\}
16. add_op $\rightarrow$ + \{+\}
17. add_op $\rightarrow$ - \{-\}
18. mult_op $\rightarrow$ * \{*\}
19. mult_op $\rightarrow$ / \{/\}

Zheng Zhang 
12
Example: Simple Calculator Language

Recursive descent parser implementation

```plaintext
procedure match(expected)
    if input_token = expected then consume_input_token()
    else parse_error

-- this is the start routine:
procedure program()
    case input_token of
        id, read, write, $$:
            stmtList()
            match($$)
        otherwise parse_error

procedure stmtList()
    case input_token of
        id, read, write : stmt(); stmtList()
        $$ : skip         -- epsilon production
        otherwise parse_error

procedure stmt()
    case input_token of
        id : match(id); match(=); expr()
        read : match(read); match(id)
        write : match(write); expr()
        otherwise parse_error

procedure expr()
    case input_token of
        id, number, ( : term(); term_tail)
        otherwise parse_error

procedure term_tail()
    case input_token of
        +, - : add_op(); term(); term_tail()
        ) , id, read, write, $$:
            skip         -- epsilon production
        otherwise parse_error

procedure term()
    case input_token of
        id, number, ( : factor(); factor_taid()
        otherwise parse_error

procedure add_op()
    case input_token of
        + : match(+)
        - : match(-)
        otherwise parse_error

procedure mult_op()
    case input_token of
        * : match(*)
        / : match(/
        otherwise parse_error

procedure factor_tail()
    case input_token of
        *, / : mult_op(); factor(); factor_tail()
        +, -, ) , id, read, write, $$:
            skip         -- epsilon production
        otherwise parse_error
```

Zheng Zhang
Things to do:
Start programming in C. Check out the web for tutorials.

Read Scott: Chap. 3.1 - 3.3; ALSU Chap. 7.1
Read Scott: Chap. 8.1 - 8.2; ALSU Chap. 7.1 - 7.3