LL(1)

14fall-cs314-Section1
Sep. 30th
LL(1)

Overview:
A LL(1) grammar allows the machine to make right(unique) decisions on choosing certain production among numbers of options, based on the knowledge of the next 1 terminal.

First set;
Follow set;
First+ set;
**LL(1)**

First set:
\[ \text{First}(A) = \{ \text{Every possible terminals that could appears at the beginning of } A\text{'s derivation} \} \]
Constructing First-set for a NT, X
A => X | Y | ...

X => aaa | bbb | ccc | ddd | ...
  Y | Z...
P Q |
  | eps

First(X) = {a, b, c, d...
  First(Y), First(Z)... 
  First(P),
  First(Q) if First(P) has eps
  eps
}

\textbf{LL(1)}

**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 FIRST(X)
3. iterate until no more terminals or \epsilon can be added to any FIRST(X):
   
   if X ::= Y_1Y_2 \cdots Y_k then
   
   a \in FIRST(X) if a \in FIRST(Y_i)
   
   and \epsilon \in FIRST(Y_j) for all 1 \leq j < i
   
   \epsilon \in FIRST(X) if \epsilon \in FIRST(Y_i) for all 1 \leq i \leq k
   
   end iterate

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

Follow set:
First(A)={every possible terminals that can appear immediately to the right of A in some sentential form}
LL(1)

Constructing Follow-set for a NT, X
check all the production rules that refer to X

A => XY
Y => a | b | c
Follow(X) = \{a, b, c\}

A => XYZ
Y = a | b | c | \epsilon
Z = d | e
Follow(X) = \{a, b, c, d, e\}

A => XY
Y = a | b | \epsilon
Follow(X) = \{a, b, \text{Follow}(A)\}

1. place eof in FOLLOW(\langle \text{goal} \rangle)
   iterate until no more terminals or \epsilon can be added to any FOLLOW(X):

2. if A ::= \alpha B \beta then
   put \{FIRST(\beta) - \epsilon\} in FOLLOW(B)

3. if A ::= \alpha B then
   put FOLLOW(A) in FOLLOW(B)

4. if A ::= \alpha B \beta and \epsilon \in FIRST(\beta) then
   put FOLLOW(A) in FOLLOW(B)
end iterate
Constructing First+ set for a NT, X

1. First(X) has eps
   => X could be replaced by eps
   => T’s in Follow(X) could be the next T

2. First(X) doesn’t have eps
   => Only T’s in First(X) could be the next T

grammar G is LL(1)
   => Given next input T, only one unique production could be chosen
   => Always being able to choose the right production, no need to backtrack.

**LL(1)**

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

$(A ::= \alpha \text{ and } A ::= \beta)$ implies

$FIRST^+(\alpha) \cap FIRST^+(\beta) = \emptyset$
LL(1)

Parse Table:
Like DFA, we create a table, that maintains the right choice based on the next T from the scanner.

BNF for grammar G:
E -> T E'
E'-> + T E' | \text{eps}
T -> F T'
T'-> * F T' | \text{eps}
F -> ( E ) | id
LL(1)

G:
E \rightarrow T E'
E' \rightarrow + T E' \mid \text{eps}
T \rightarrow F T'
T' \rightarrow * F T' \mid \text{eps}
F \rightarrow ( E ) \mid \text{id}

***********
First(+ T E')={+}
First+(+ T E')={+}

First(\text{eps})={\text{eps}}
Follow(E')={\text{Follow}(E)}
Follow(E)={\$, )}
First+(\text{eps})=First(\text{eps})-{\text{eps}} \cup \text{Follow}(E')
=\{\text{eps}\}-{\text{eps}} \cup \{\$, )\}
=\{\$, )\}

1. place \text{eof} in \text{FOLLOW}((\text{goal}))
   iterate until no more terminals or \text{eps}
can be added to any \text{FOLLOW}(X):

2. if \( A := aB \beta \) then
   put \{\text{FIRST} (\beta) - \text{eps}\} in \text{FOLLOW}(B)
3. if \( A := aB \) then
   put \text{FOLLOW}(A) in \text{FOLLOW}(B)
4. if \( A := aB \beta \) and \( \epsilon \in \text{FIRST}(\beta) \) then
   put \text{FOLLOW}(A) in \text{FOLLOW}(B)
   and iterate

Define \( \text{FIRST}^{+}(\delta) \) for rule \( A := \delta \)

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

A grammar is LL(1) iff
\( (A := \alpha \text{ and } A := \beta) \) implies
\( \text{FIRST}^{+}(\alpha) \cap \text{FIRST}^{+}(\beta) = \emptyset \)
LL(1)

First+(+ T E')={ + }
First+(eps)=First(eps)-{eps} U Follow(E)
    ={eps}-{eps} U {\$, )}
    ={\$, )}

<table>
<thead>
<tr>
<th></th>
<th>id</th>
<th>+</th>
<th>*</th>
<th>(</th>
<th>)</th>
<th>$</th>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
<td>E-&gt;TE'</td>
<td></td>
<td></td>
<td>E-&gt;TE'</td>
<td></td>
<td></td>
</tr>
<tr>
<td>E'</td>
<td></td>
<td>E'-&gt;+ T E'</td>
<td></td>
<td>E'-&gt;eps</td>
<td>E'-&gt;eps</td>
<td></td>
</tr>
<tr>
<td>T</td>
<td>T-&gt;F T'</td>
<td></td>
<td></td>
<td>T-&gt;FT'</td>
<td></td>
<td></td>
</tr>
<tr>
<td>T'</td>
<td></td>
<td>T'-&gt;eps</td>
<td>T'-&gt;* F T'</td>
<td>T'-&gt;eps</td>
<td>T'-&gt;eps</td>
<td></td>
</tr>
<tr>
<td>F</td>
<td>F-&gt;id</td>
<td></td>
<td></td>
<td>F-&gt;( E )</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
**LL(1)**

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

$(A ::= \alpha \text{ and } A ::= \beta)$ implies $FIRST^+(\alpha) \cap FIRST^+(\beta) = \emptyset$

---

A grammar is LL(1) iff

for every NT with 2 or more productions, each production’s first+ set is disjoint from others’.

==every block in parse table holds at most 1 production.
Recursive Descent Parsing

1) create a procedure for every NT
2) the procedure on the top level of the recursion, could be taken as the main()
3) during the procedure of a NT, there could be procedure calls to the procedure itself
4) the procedure could perform some other useful stuff
   • type checking(checker)
   • counting(performance predictor)
   • calculating(interpreter)
   • generating assembly code(code generator)
   or
   • printing+counting(HW3_Q4)
Recursive Descent Parsing

<program> ::= program <block>

```c
program ()
{
    switch token
    {
        case PROGRAM:
            token := next_token();
            return block();
            if PERIOD != token
            {error;}
            token := next_token();
            if EOF != token
            {error;}
            token := next_token();
            if EOF != token
            {error;}
            break;
        default:
            error;
    }
}
```
Recursive Descent Parsing

\[ <\text{program}> ::= \text{program} \ <\text{block}> \]

\begin{verbatim}
program ()
{
    switch token
    {
    case PROGRAM:
        token := next_token();
        goto block();
        if PERIOD != token
            {error;}
        token := next_token();
        if EOF != token
            {error;}
        //print “asg_num”; print”ref_num”…
        break;
    default: error;
    }
}
\end{verbatim}

Lec7-Pag10-Pag13

http://paul.rutgers.edu/~ll557/314rec_2014fall/