TOP-DOWN PARSING

CMPT 379 Lecture 15b

Lecture Overview

- FIRST, FOLLOWS, FIRST$^+$
- Parsing Tables
- LL(1) Parsing
- Recursive Descent

Analyzing a Grammar

- Suppose we have a grammar G with productions P. We wish to tell if there is an automatic way to parse G.
- We will use the following example grammar:

$$E \rightarrow T\ E'$$
$$E' \rightarrow +\ T\ E'\ |\ -T\ E'\ |\ \varepsilon$$
$$T \rightarrow F\ T'$$
$$T' \rightarrow *\ F\ T'\ |\ /\ F\ T'\ |\ \varepsilon$$
$$F \rightarrow i\ |\ c\ |\ (\ E\ )$$

FIRST

- For any grammar symbol B(terminal, nonterminal, ε, or <eof>) we define FIRST(B) to be the set of terminals that can appear as the first word in a string derived from B.
- If B is a terminal, ε, or <eof>, then FIRST(B) = B.

FIRST(E) = { i, c, (}
FIRST(E') = {+, –, ε}
FIRST(T) = {i, c, (}
FIRST(T') = {*, /, ε}
FIRST(F) = { i, c, (}

FIRST OVER STRINGS

For any grammar symbol string $s = B_1B_2B_3 \ldots B_k$ we define FIRST(s) to be the set of terminals that can appear as the first word in a string derived from s. It is the union of FIRST sets for $B_1B_2 \ldots B_n$, where B_n is the first symbol whose FIRST set does not contain ε. ε is in FIRST(s) iff it is in FIRST(B_i) for all i = 1 to k.

FIRST(EE') = { i, c, (}
FIRST(E'ET) = {+, –, i, c, (}
FIRST(T'E') = {*, /, +, –, ε}

FOLLOW

- For any nonterminal N we define FOLLOW(N) to be the set of terminals that can appear to the immediate right of a string derived from N.

FOLLOW(E) = {<eof>,) }
FOLLOW(E') = {<eof>,) }
FOLLOW(T) = {<eof>, +, –,) }
FOLLOW(T') = {<eof>, +, –,) }
FOLLOW(F) = {<eof>, +, –, *, /,) }

FIRST$^+$

- For any production $A \rightarrow \beta$, where β is a string of terminals and nonterminals, we define $FIRST^+(A\rightarrow\beta)$ to be

$$FIRST^+(A\rightarrow\beta) = \begin{cases} FIRST(\beta) \cup FOLLOW(A) & \text{if } \varepsilon \in FIRST(\beta) \\ FIRST(\beta) & \text{otherwise} \end{cases}$$

$FIRST^+(E' \rightarrow + \ T \ E') = \{+\}$
$FIRST^+(E' \rightarrow - \ T \ E') = \{-\}$
$FIRST^+(E' \rightarrow \varepsilon) = \{\text{<eof>}, \ )\ \}$

FIRST$^+$

In a predictive grammar, if there are two productions $A \rightarrow \beta_1$ and $A \rightarrow \beta_2$, then it must be that FIRST$^+(A\rightarrow\beta_1)$ and FIRST$^+(A\rightarrow\beta_2)$ are disjoint.

In this case, the FIRST$^+$ sets completely encode the decisions needed for parsing. We can express them in a table called a LL(1) parsing table.

First we must number our (BNF) productions.

LL(1) Parsing Table

1. E → T E'
2. E' → + T E'
3. E' → – T E'
4. E' → ε
5. T → F T'
6. T' → * F T'
7. T' → / F T'
8. T' → ε
9. F → i
10. F → c
11. F → (E)

	+	–	*	/	i	c	(	)	eof
E	–	–	–	–	1	1	1	–	–
E'	2	3	–	–	–	–	–	4	4
T	–	–	–	–	5	5	5	–	–
T'	8	8	6	7	–	–	–	8	8
F	–	–	–	–	9	10	11	–	–

Parsing Example

	+	–	*	/	i	c	(	)	eof
E	–	–	–	–	1	1	1	–	–
E'	2	3	–	–	–	–	–	4	4
T	–	–	–	–	5	5	5	–	–
T'	8	8	6	7	–	–	–	8	8
F	–	–	–	–	9	10	11	–	–

1. E → T E'
2. E' → + T E'
3. E' → – T E'
4. E' → ε
5. T → F T'
6. T' → * F T'
7. T' → / F T'
8. T' → ε
9. F → i
10. F → c
11. F → (E)

stack	input	prod.
eof E	c + c * i eof	1
eof E' T	c + c * i eof	5
eof E' T' F	c + c * i eof	10
eof E' T' c	c + c * i eof	match
eof E' T'	+ c * i eof	8
eof E'	+ c * i eof	2
eof E' T +	+ c * i eof	match
eof E' T	c * i eof	5
eof E' T' F	c * i eof	10
eof E' T' c	c * i eof	match
eof E' T'	* i eof	6
eof E' T' F *	* i eof	match
eof E' T' F	i eof	9
eof E' T' i	i eof	match
eof E' T'	eof	8
eof E'	eof	4
eof	eof	match

Leftmost Derivation

If we write the matched terminals followed by the stack contents (from top to bottom), we get an interesting sequence of strings

matched + stack

E
T E'
F T' E'
c T' E'
c T' E'
c E'
c + T E'
c + T E'
c + F T' E'
c + c T' E'
c + c T' E'
c + c T' E'
c + c * F T' E'
c + c * F T' E'
c + c * i T' E'
c + c * i T' E'
c + c * i E'
c + c * i

This is a leftmost derivation in the grammar. In each step, the leftmost nonterminal is replaced by one of its productions' right-hand sides.

LL(1) Parsing Algorithm

```
token = NextToken()
create stack T
T.push(eof)
T.push(S)                // start symbol

loop forever:
   focus = T.top()       // not T.pop()
   if focus = token = eof
      report success; exit loop;
   else if focus is a terminal or eof
      if focus matches token
         T.pop()
         token = nextToken()
      else
         report error looking for token
             at top of the stack
   else
      if Table(focus, token) is
               A → B1B2...Bk
         T.pop()
         for i = k to 1 step -1
            if Bi is not ε
               T.push(Bi)
      else
         report an error expanding
               focus
```

LL(1) Table Creation

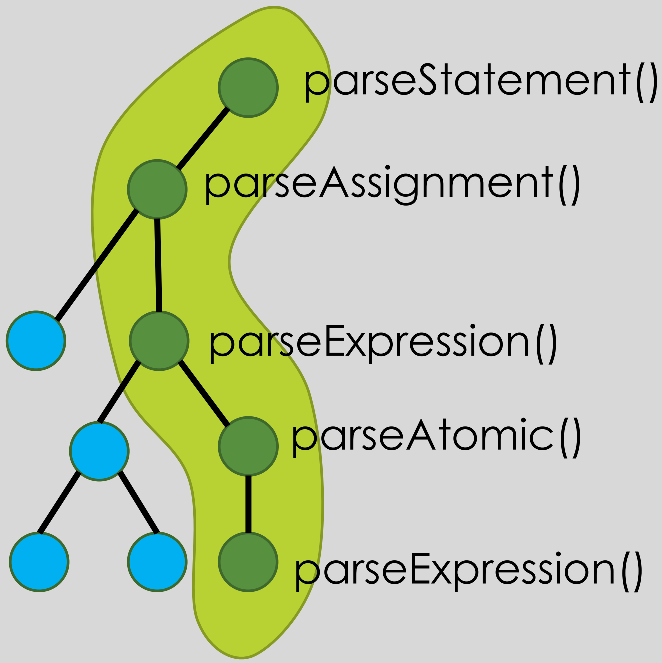
build **FIRST, FOLLOW, FIRST⁺** sets.

```
for each nonterminal A
    for each terminal w                                  // including eof
        Table[A, w] = error;

    for each production p of the form A → β
        for each terminal w in FIRST⁺(A → β)             // including eof
            Table[A, w] = p
```

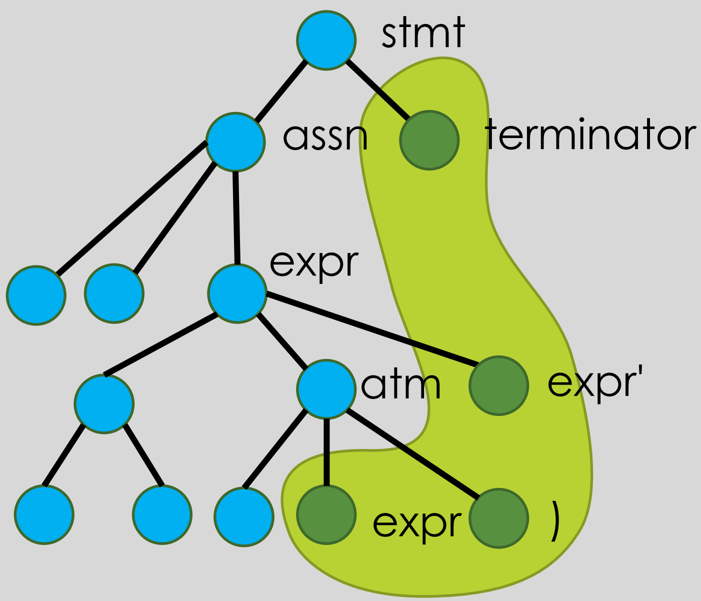
Top-Down Parsing

Recursive Descent



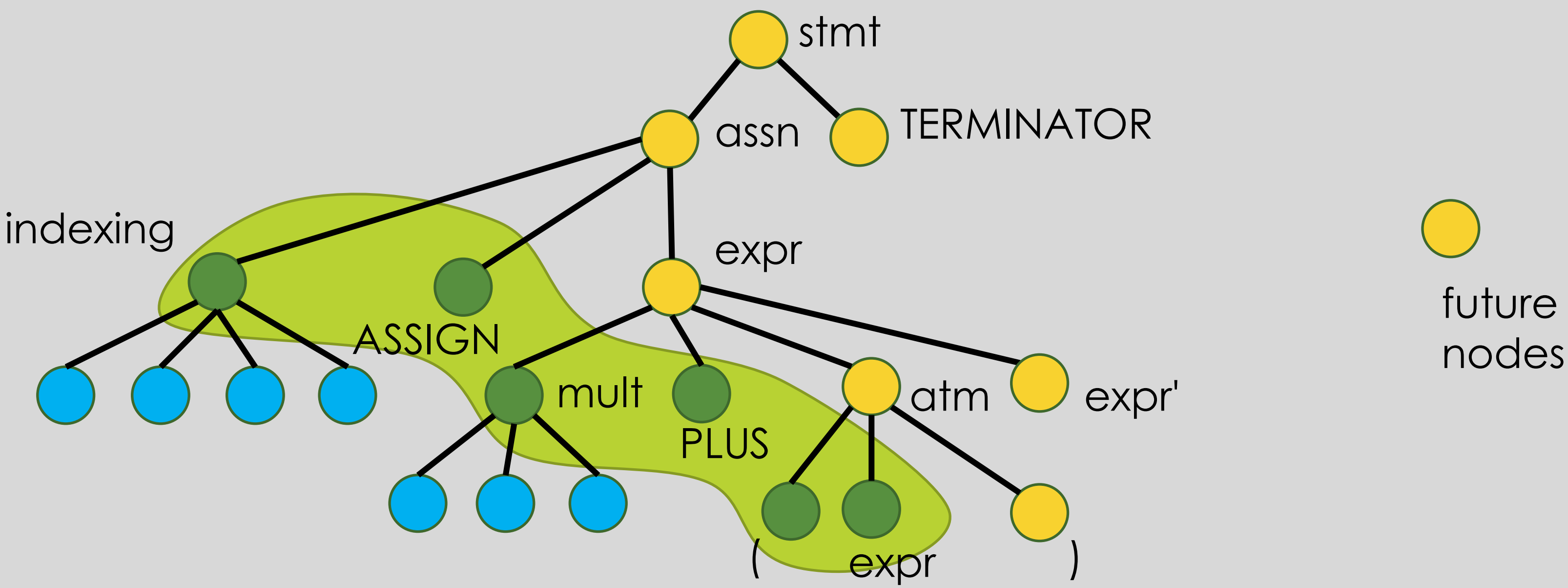
Recursion stack contains path from root to current node

LL(1) Table Parsing



Stack contains right children of path from root to current node

Bottom-up Parsing



Stack contains roots of discovered subtrees