Unit 7
Predictive Parsing

Nguyen Thi Thu Huong
Hanoi University of Technology

Top Down Parsing Methods

- Simplest method is a full-backup *recursive descent* parse
- Write recursive recognizers (subroutines) for each grammar rule
  - If rules succeeds perform some action (I.e., build a tree node, emit code, etc.)
  - If rule fails, return failure. Caller may try another choice or fail
  - On failure it “backs up” which might have problem if it needs to return a lexical symbol to the input stream

Problems

- Also remember left recursion problem
- Need to backtrack , suppose that you could always tell what production applied by looking at one (or more) tokens of lookahead – called predictive parsing
- Factoring

Summary of Recursive Descent

- Simple and general parsing strategy usually coupled with simple handcrafted lexer
  - Left-recursion must be eliminated first
  - … but that can be done automatically
- Unpopular because of backtracking
  - Thought to be too inefficient

- In practice, backtracking is eliminated by restricting the grammar

Elimination of Immediate Left Recursion

$A \text{ -> } A\,\alpha_1 \;\mid A\,\alpha_2 \mid \ldots\; A\,\alpha_m \;\mid \beta_1 \mid \beta_2 \mid \beta_n$

$A \text{ –> } \beta_1\, A' \mid \beta_2\, A' \;\ldots\; \beta_n\, A'$

$A' \text{ -> } \alpha_1\, A' \mid \alpha_2\, A' \mid \;\ldots\; \alpha_m A' \mid \varepsilon$

Predictive Parsers

- Like recursive-descent but parser can “predict” which production to use
  - By looking at the next few tokens
  - No backtracking
- Predictive parsers accept LL(k) grammars
  - L means “left-to-right” scan of input
  - L means “leftmost derivation”
  - k means “predict based on k tokens of lookahead”
- In practice, LL(1) is used

LL(1) Languages

- In recursive-descent, for each non-terminal and input token there may be a choice of production
- LL(1) means that for each non-terminal and token there is only one production
- Can be specified via 2D tables
  - One dimension for current non-terminal to expand
  - One dimension for next token
  - A table entry contains one production

Predictive Parsing and Left Factoring

- Recall the grammar

  $E \to T + E \mid T$

  $T \to int \mid int * T \mid (E)$

- Hard to predict because
  - For T two productions start with int
  - For E it is not clear how to predict

- A grammar must be <u>left-factored</u> before use for predictive parsing

Left-Factoring Example

- Consider the grammar

$$E \rightarrow T + E \mid T$$

$$T \rightarrow \text{int} \mid \text{int} * T \mid (E)$$

- Factor out common prefixes of productions

$$E \rightarrow T X$$

$$X \rightarrow + E \mid \varepsilon$$

$$T \rightarrow (E) \mid \text{int } Y$$

$$Y \rightarrow * T \mid \varepsilon$$

LL(1) Parsing Table Example

- Left-factored grammar

  $E \to T\ X$ $\qquad$ $X \to +\ E \mid \varepsilon$

  $T \to (\ E\ ) \mid \text{int}\ Y$ $\qquad$ $Y \to *\ T \mid \varepsilon$

- The LL(1) parsing table:

	int	*	+	(	)	$
E	T X			T X		
X			+ E		ε	ε
T	int Y			(E)		
Y		* T	ε		ε	ε

LL(1) Parsing Table Example (Cont.)

- Consider the [E, int] entry
  - "When current non-terminal is E and next input is int, use production $E \rightarrow T X$
  - This production can generate an int in the first place
- Consider the [Y,+] entry
  - "When current non-terminal is Y and current token is +, get rid of Y"
  - Y can be followed by + only in a derivation in which $Y \rightarrow \varepsilon$

LL(1) Parsing Tables. Errors

- Blank entries indicate error situations
  - Consider the [E,*] entry
  - “There is no way to derive a string starting with * from non-terminal E”

Using Parsing Tables

- Method similar to recursive descent, except
  - For each non-terminal S
  - We look at the next token a
  - And chose the production shown at [S,a]
- We use a stack to keep track of pending non-terminals
- We reject when we encounter an error state
- We accept when we encounter end-of-input

LL(1) Parsing Algorithm

```
initialize stack = <S $> and next
repeat
   case stack of
      <X, rest>   : if T[X,*next] = Y1...Yn
                        then stack ← <Y1... Yn rest>;
                        else  error ();
      <t, rest>   : if t == *next ++
                        then  stack ← <rest>;
                        else error ();
until stack == < >
```

LL(1) Parsing Example

Stack	Input	Action
E $	int * int $	T X
T X $	int * int $	int Y
int Y X $	int * int $	terminal
Y X $	* int $	* T
* T X $	* int $	terminal
T X $	int $	int Y
int Y X $	int $	terminal
Y X $	$	ε
X $	$	ε
$	$	ACCEPT

Constructing Parsing Tables

- LL(1) languages are those defined by a parsing table for the LL(1) algorithm
- No table entry can be multiply defined

- We want to generate parsing tables from CFG

Constructing Parsing Tables (Cont.)

- If $A \rightarrow \alpha$, where in the line of A we place α ?
- In the column of t where t can start a string derived from α
  - $\alpha \rightarrow^* t\ \beta$
  - We say that $t \in \text{First}(\alpha)$
- In the column of t if α is ε and t can follow an A
  - $S \rightarrow^* \beta\ A\ t\ \delta$
  - We say $t \in \text{Follow}(A)$

Computing First Sets

Definition: $\text{First}(X) = \{ t \mid X \to^* t\alpha\} \cup \{\varepsilon \mid X \to^* \varepsilon\}$

Algorithm sketch (see book for details):

1. for all terminals t do $\text{First}(X) \leftarrow \{ t \}$
2. for each production $X \to \varepsilon$ do $\text{First}(X) \leftarrow \{ \varepsilon \}$
3. if $X \to A_1 \ldots A_n\ \alpha$ and $\varepsilon \in \text{First}(A_i)$, $1 \le i \le n$ do
   - add $\text{First}(\alpha)$ to $\text{First}(X)$
4. for each $X \to A_1 \ldots A_n$ s.t. $\varepsilon \in \text{First}(A_i)$, $1 \le i \le n$ do
   - add ε to $\text{First}(X)$
5. repeat steps 4 & 5 until no First set can be grown

First Sets. Example

- Recall the grammar

  E → T X　　　X → + E | ε

  T → (E) | int Y　　　Y → * T | ε

- First sets

  First(( ) = { (}　　　First(T) = {int, (}

  First( )) = {) }　　　First(E) = {int, (}

  First(int) = { int }　　　First(X) = {+, ε }

  First(+) = { + }　　　First(Y) = {*, ε }

  First(*) = { * }

Computing Follow Sets

- Definition:

$$\text{Follow}(X) = \{ t \mid S \rightarrow^* \beta\, X\, t\, \delta \}$$

- Intuition
  - If S is the start symbol then $\$ \in \text{Follow}(S)$
  - If $X \rightarrow A\ B$ then $\text{First}(B) \subseteq \text{Follow}(A)$ and $\text{Follow}(X) \subseteq \text{Follow}(B)$
  - Also if $B \rightarrow^* \varepsilon$ then $\text{Follow}(X) \subseteq \text{Follow}(A)$

Computing Follow Sets (Cont.)

Algorithm sketch:

1. Follow(S) $\leftarrow$ { \$ }
2. For each production $A \to \alpha\ X\ \beta$
   - add First(β) - {ε} to Follow(X)
3. For each $A \to \alpha\ X\ \beta$ where $\varepsilon \in$ First(β)
   - add Follow(A) to Follow(X)

- repeat step(s) ___ until no Follow set grows

Follow Sets. Example

- Recall the grammar

$$E \to T\,X \qquad\qquad X \to +\,E \mid \varepsilon$$
$$T \to (\,E\,) \mid \text{int}\;Y \qquad\qquad Y \to *\;T \mid \varepsilon$$

- Follow sets

Follow(+) = { int, (}   Follow(*) = { int, (}

Follow(( ) = { int, (}   Follow(E) = {), $}

Follow(X) = {$,) }   Follow(T) = {+,) , $}

Follow( )) = {+,) , $}   Follow(Y) = {+,) , $}

Follow(int) = {*, +,) , $}

Constructing LL(1) Parsing Tables

- Construct a parsing table T for CFG G

- For each production $A \rightarrow \alpha$ in G do:
  - For each terminal $t \in$ First(α) do
    - T[A, t] = α
  - If $\varepsilon \in$ First(α), for each $t \in$ Follow(A) do
    - T[A, t] = α
  - If $\varepsilon \in$ First(α) and $\$ \in$ Follow(A) do
    - T[A, $] = α

Notes on LL(1) Parsing Tables

- If any entry is multiply defined then G is not LL(1)
  - If G is ambiguous
  - If G is left recursive
  - If G is not left-factored
- Most programming language grammars are not LL(1)
- There are tools that build LL(1) tables