Lecture 8

Syntax Analysis IV

Predictive Parsing

Predictive Parsing

- Like recursive-descent but parser can "predict" which production to use
  - By looking at the next few tokens (*lookahead*)
  - No backtracking
- Predictive parsers restrict CFGs and accept LL(k) grammars
  - 1st L – The input is scanned from left to right
  - 2nd L – Create left-most derivation
  - k – Number of symbols of look-ahead
- Most parsers work with one symbol of look-ahead [LL(1)]
- Informally, LL(1) has no left-recursion and has been left-factored.

Left Factoring

- At each step, only one choice of production

- Given the grammar:

  $E \rightarrow T + E \mid T$

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

- Hard to predict which production to use because of common prefixes.

- We need to left-factor the grammar

- The Left-factored grammar:

  $E \rightarrow T\, X$

  $X \rightarrow + E \mid \varepsilon$

  $T \rightarrow (E) \mid int\, Y$

  $Y \rightarrow * T \mid \varepsilon$

LL(1) Parsing

- Left factored Grammar:

$$E \rightarrow T\,X \qquad\qquad T \rightarrow (\,E\,) \mid int\; Y$$
$$X \rightarrow +\,E \mid \varepsilon \qquad\qquad Y \rightarrow *\,T \mid \varepsilon$$

- LL(1) parsing table:

Next input token

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

Leftmost non-terminal

RHS of production to use

Predictive Parsing Algorithm

- For the leftmost non-terminal X
- We look at the next input token *a*
- Makes use of an explicit parsing table of the form M[X, a]

- An explicit external stack records frontier of the parse tree
  - Non-terminals that have yet to be expanded
  - Terminals that have yet to matched against the input
  - Top of stack = leftmost pending terminal or non-terminal

- Reject on reaching error state
- Accept on *end of input* & *empty stack*

Predictive Parsing Algorithm

- The parse table entry M[X, a] indicates which production to use if the top of the stack is a non-terminal ‘X’ and the current input token is ‘a’.

- In that case ‘POP X’ from the stack and ‘PUSH’ all the RHS symbols of the production M[X, a] in reverse order.

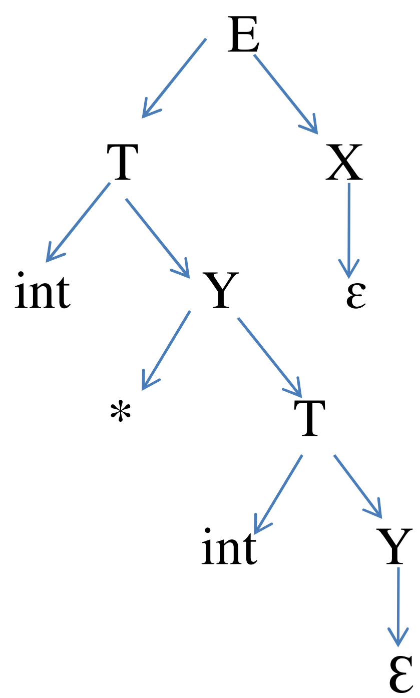
- Assume that '$' is a special token that is at the bottom of the stack and terminates the input string

```
if X = a = $ then halt
if X = a  $ then pop(x) and ip++
if X is a non terminal
    then if M[X, a] = {X → UVW}
            then begin pop(X); push(W,V,U)
         end
else error
```

LL(1) Parsing

STACK	INPUT	ACTION
E$	int * int $	TX
TX$	int * int $	intY
intYX$	int * int $	Terminal (match)
YX$	*int$	*T
*TX$	*int$	Terminal (match)
TX$	int$	intY
intYX$	int$	Terminal (match)
YX$	$	ε
X$	$	ε
$	$	Accept

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



Construction of Parsing Table

- Given the production $A \rightarrow \alpha$, and a token t
- M[A, t] = α occurs in two cases:

1. If $\alpha \rightarrow^* t\beta$
   - α can derive a ‘t’ in the first position in 0 or more moves
   - We say that $t \in First(\alpha)$
     - ‘t’ is one of the terminals that can produce in the first position
2. If $A \rightarrow \alpha$ and $\alpha \rightarrow^* \varepsilon$ and $X \rightarrow^* \beta\, At\delta$
   - Stack has A, input is t, and A cannot derive t
   - In this case only option is to get rid of A (by deriving)
   - Can work only if ‘t’ follows A in at least one derivation
   - We say t $\in$ Follow(A)

First Sets

- Definition
  - First(X) = { t | t is a terminal and $X \rightarrow^* t\alpha$} or {ε|$X \rightarrow^* \varepsilon$}

- To build FIRST(X):
  1. If X ϵ terminal, then FIRST(X) is {X}
  2. If (($X \rightarrow \varepsilon$) or (($X \rightarrow Y_1 \ldots Y_k$) and ($\varepsilon \in [\square i: 1 < i \leq k \text{ First } (Y_i)]$))) then add ε to FIRST(X)
  3. If (($X \rightarrow Y_1 Y_2 \ldots Y_k \alpha$) and ($\varepsilon \in [\square i: 1 < i \leq k \text{ First } (Y_i)]$))) then add FIRST($\alpha$) to FIRST(X)

- Find the first sets in the grammar:

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

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

First Sets

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

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

- First (+)={+}
- First (*)={*}
- First (()={(}
- First ())={)}
- First (int)={int}

- First (E)=First(T)
- First (T) = {(, int}
- First(X) = {+, ε}
- First (Y) = {*, ε}

Follow Set

- Definition:
  - Follow(X) = { t | S $\rightarrow^*$ $\alpha X t \beta$}
  - 't' is said to follow of 'X' if we can obtain a sentential form where the terminal 't' comes immediately after 'X'

- *Follow set for a given symbol never concerns what the symbol can generate, but depends on where that symbol can appear in the derivations.*

- If (X$\rightarrow$AB) then
  - First(B) is in Follow(A) and
  - Follow(X) is in Follow(B)
  - If (B $\rightarrow^*\varepsilon$) then
    - Follow(X) is in Follow(A)

Follow Set

- To build FOLLOW(X)
  1. Add $ to FOLLOW(S) [If S is the start symbol]
  2. If ($A \rightarrow \alpha B \beta$), then

     Add everything in FIRST(β) except ε to FOLLOW(B)
  3. If (($A \rightarrow \alpha B \beta$ and $\beta \rightarrow^* \varepsilon$) or ($A \rightarrow \alpha B$))

     Add everything in FOLLOW(A) to FOLLOW(B)

- *ε never appears in Follow sets, so Follow sets are just sets of terminals*

- Find the follow sets in the grammar:

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

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

Parsing Table Construction

- for each production $A \rightarrow \alpha$ {
  - for each terminal 't' in FIRST(α)
    - $M[A, t] = \alpha$;
  - if ε is in FIRST(α) , then
  - for each terminal 'b' (including '$') *in* Follow (A)
    - $M[A, b] = \alpha$;

}

- Find the follow sets in the grammar:

$E \rightarrow T\,X \qquad X \rightarrow + E \mid \varepsilon$

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

Recognizing LL(1) Grammars

- Consider the grammar X → Xa | b
- First (X) = {b} Follow(X) = {$, a}
- Parsing Table:

	a	b	$
X		b Xa	

Multiply defined entry: This is how we know that the grammar is not LL(1)

Recognizing LL(1) Grammars

- Formally, a grammar is LL(1) iff:
  - It is not left-recursive
  - Not ambiguous
  - A grammar G is LL(1) *iff* whenever A → u | v are two distinct productions of G, the following conditions hold:
    - For no terminal 'a' do both 'u' and 'v' derive strings beginning with 'a' (i.e., FIRST sets are disjoint)
    - At most one of 'u' and 'v' can derive the empty string
    - if v →* ε then 'u' does not derive any string beginning with a terminal in Follow(A) (i.e., if ε ϵ FIRST(v), then FIRST(u) and FOLLOW(A) are disjoint sets)

Most programming languages are not LL(1)

Error Handling

- Stop at the first error and print a message
  - Compiler writer friendly
  - But not user friendly

- Every reasonable compiler must recover from error and identify as many errors as possible

- However, multiple error messages (cascaded spurious error messages) due to a single fault must be avoided

- Error recovery methods
  - Panic mode
  - Phrase level recovery
  - Error productions
  - Global correction

Panic Mode

- Simplest and the most popular method

- Most tools provide for specifying panic mode recovery in the grammar

- When an error is detected
  - Discard tokens until one with a clear role is found
  - Continue from there

- Looking for synchronizing tokens
  - Typically the statement or expression terminators

Panic Mode

- Consider following code

```
{
  a = b + c;
  d = m n ;
  e = d - f;
}
```

Syntax Error

- The second expression has syntax error
  – Skip ahead to next ';' and try to parse the next expression
- It discards one expression and tries to continue parsing

Phrase Level Recovery

- Make local correction to the input
  - Eg: Fill in the blank entries in the predictive parsing table with pointers to error routines

- Works only in limited situations
  - A common programming error which is easily detected
  - For example insert a ';' after closing '}' of a class definition

- Does not work very well when the actual error has occurred before the point of detection

Error Productions

- Add erroneous constructs as productions in the grammar
- Works only for most common mistakes which can be easily identified
- Essentially makes common errors as part of the grammar
- Complicates the grammar and does not work very well
- Eg:
  - Write 5 x instead of 5 * x
  - Add the production E $\rightarrow$ … | E E

Global Corrections

- Idea: find a correct "nearby" program
  - Try token insertions and deletions
  - Nearness may be measured using certain metric (Edit Distance)
  - Exhaustive search

- PL/C compiler implemented this scheme: anything could be compiled.

- Disadvantages:
  - Complicated - hard to implement
  - Slows down parsing of correct programs
  - "Nearby" is not necessarily "the intended" program

Error Recovery in LL(1) Parser

- Error occurs when a parse table entry M[A, a] is empty or terminal on stack-top do not match with input.

- Skip symbols in the input until a token in a selected set (synch) appears

- Place symbols in follow(A) in synch set. Skip tokens until an element in follow(A) is seen. Pop(A) and continue parsing

- Add symbol in first(A) in synch set. Then it may be possible to resume parsing according to A if a symbol in first(A) appears in input.

Thanks