Compiler Design

Lecture-6

Predictive Parsing

Topics Covered

- Introduction to Predictive Parsing
- How we make Parsing Table.

Predictive parsing

- Recall the main idea of top-down parsing:
  - Start at the root, grow towards leaves
  - Pick a production and try to match input
  - May need to backtrack
- Can we avoid the backtracking?
  - Given $A \rightarrow \alpha \mid \beta$ the parser should be able to choose between α and β
- How?
  - What if we do some "preprocessing" to answer the question: Given a non-terminal A and lookahead t, which (if any) production of A is guaranteed to start with a t?

Predictive parsing

- If we have two productions: $A \to \alpha \mid \beta$, we want a distinct way of choosing the correct one.
- Define:
  - for $\alpha \in G$, $\mathbf{x} \in FIRST(\alpha)$ iff $\alpha \Rightarrow^* \mathbf{x}\gamma$
- If FIRST(α) and FIRST(β) contain no common symbols, we will know whether we should choose $A \to \alpha$ or $A \to \beta$ by looking at the lookahead symbol.

Predictive parsing

- Compute FIRST(X) as follows:
  - if X is a terminal, then FIRST(X)={X}
  - if $X \to \varepsilon$ is a production, then add ε to FIRST(X)
  - if X is a non-terminal and $X \to Y_1 Y_2 \ldots Y_n$ is a production, add FIRST(Y_i) to FIRST(X) if the preceding Y_js contain ε in their FIRSTs

Predictive parsing

- What if we have a "candidate" production $A \to \alpha$ where $\alpha = \varepsilon$ or $\alpha \Rightarrow^* \varepsilon$?
- We could expand if we knew that there is some sentential form where the current input symbol appears after A.
- Define:
  - for $A \in N$, $\mathbf{x} \in \text{FOLLOW}(A)$ iff $\exists\ S \Rightarrow^* \alpha A \mathbf{x} \beta$

Predictive parsing

- Compute FOLLOW as follows:
  - FOLLOW(S) contains EOF
  - For productions $A \to \alpha B \beta$, everything in FIRST(β) except ε goes into FOLLOW(B)
  - For productions $A \to \alpha B$ or $A \to \alpha B \beta$ where FIRST(β) contains ε, FOLLOW(B) contains everything that is in FOLLOW(A)

Predictive parsing

- Armed with
  - FIRST
  - FOLLOW
- we can build a parser where no backtracking is required!

Predictive parsing (w/table)

- For each production $A \rightarrow \alpha$ do:
  - For each terminal **a** $\in$ FIRST(α) add $A \rightarrow \alpha$ to entry M[A,**a**]
  - If $\varepsilon \in$FIRST(α), add $A \rightarrow \alpha$ to entry M[A,**b**] for each terminal **b** $\in$ FOLLOW(A).
  - If $\varepsilon \in$FIRST(α) and EOF$\in$FOLLOW(A), add $A \rightarrow \alpha$ to M[A,EOF]
- Use table and stack to simulate recursion.

Recursive Descent Parsing

- Basic idea:
  - Write a routine to recognize each lhs
  - This produces a parser with mutually recursive routines.
  - Good for hand-coded parsers.
- Example:
  - $A \rightarrow aB \mid b$ will correspond to

```
A() {
    if (lookahead == 'a')
        match('a');
        B();
    else if (lookahead == 'b')
        match ('b');
    else error();
}
```

Building a parser

- Original grammar:

$$E \rightarrow E+E$$
$$E \rightarrow E*E$$
$$E \rightarrow (E)$$
$$E \rightarrow id$$

- This grammar is left-recursive, ambiguous and requires left-factoring. It needs to be modified before we build a predictive parser for it:

Remove ambiguity:

$$E \rightarrow E+T$$
$$T \rightarrow T*F$$
$$F \rightarrow (E)$$
$$F \rightarrow id$$

Remove left recursion:

$$E \rightarrow TE'$$
$$E' \rightarrow +TE' \mid \varepsilon$$
$$T \rightarrow FT'$$
$$T' \rightarrow *FT' \mid \varepsilon$$
$$F \rightarrow (E)$$
$$F \rightarrow id$$

Building a parser

- The grammar:

$E \to TE'$
$E' \to \mathbf{+}TE' \mid \varepsilon$
$T \to FT'$
$T' \to \mathbf{*}FT' \mid \varepsilon$
$F \to \mathbf{(}E\mathbf{)}$
$F \to \mathbf{id}$

FIRST(E) = FIRST(T) = FIRST(F) = {**(**, **id**}
FIRST(E') = {**+**, ε}
FIRST(T') = {*****, ε}
FOLLOW(E) = FOLLOW(E') = {**$**, **)**}
FOLLOW(T) = FOLLOW(T') = {**+**, **$**, **)**}
FOLLOW(F) = {*****, **+**, **$**, **)**}

Now, we can either build a table or design a recursive descend parser.

Parsing table

	+	*****	**(**	**)**	**id**	**$**
E			E→TE'		E→TE'	
E'	E'→**+**TE'			E'→ε		E'→ε
T			T→FT'		T→FT'	
T'	T'→ε	T'→*****FT'		T'→ε		T'→ε
F			F→**(**E**)**		F→**id**	
+	match					
*		match				
(			match			
)				match		
id					match	
$						accept

Parsing table

Parse the input id*id using the parse table and a stack

Step	Stack	Input	Next Action
1	**$**E	**id*id$**	E→TE'
2	**$**E'T	**id*id$**	T→FT'
3	**$**E'T'F	**id*id$**	F→**id**
4	**$**E'T'**id**	**id*id$**	match **id**
5	**$**E'T'	***id$**	T'→*****FT'
6	**$**T'F*****	***id$**	match *****
7	**$**T'F	**id$**	F→**id**
8	**$**T'**id**	**id$**	match **id**
9	**$**T'	**$**	T'→ε
10	**$**	**$**	accept

Recursive descend parser

```
parse() {
  token = get_next_token();
  if (E() and token == '$')
  then return true
  else return false
}
```

```
E() {
    if (T())
    then return Eprime()
    else return false
}
```

```
Eprime() {
    if (token == '+')
    then token=get_next_token()
         if (T())
         then return Eprime()
         else return false
    else if (token==')' or token=='$')
    then return true
    else return false
}
```

The remaining procedures are similar.

LL(1) parsing

- Our parser scans the input Left-to-right, generates a Leftmost derivation and uses 1 symbol of lookahead.
- It is called an LL(1) parser.
- If you can build a parsing table with no multiply-defined entries, then the grammar is LL(1).
- Ambiguous grammars are never LL(1)
- Non-ambiguous grammars are not necessarily LL(1)

LL(1) parsing

- For example, the following grammar will have two possible ways to expand S' when the lookahead is **else**.

$S \rightarrow$ **if** E **then** S S' | other
$S' \rightarrow$ **else** S | ε
$E \rightarrow$ **id**

– It may expand $S' \rightarrow$ **else** S or $S' \rightarrow \varepsilon$

– We can resolve the ambiguity by instructing the parser to always pick $S' \rightarrow$ **else** S. This will match each **else** to the closest previous **then**.

LL(1) parsing

- Here's an example of a grammar that is NOT LL(k) for any k:

$$S \rightarrow C\mathbf{a} \mid C\mathbf{b}$$
$$C \rightarrow \mathbf{c}C \mid \mathbf{c}$$

  – Why? Suppose the grammar was LL(k) for some k. Consider the input string $\mathbf{c}^{k+1}\mathbf{a}$. With only k lookaheads, the parser would not be able to decide whether to expand using $S \rightarrow C\mathbf{a}$ or $S \rightarrow C\mathbf{b}$

  – Note that the grammar is actually regular: it generates strings of the form $c^+(a|b)$

Error detection in LL(1) parsing

- An error is detected whenever an empty table slot is encountered.
- We would like our parser to be able to recover from an error and continue parsing.
- Phase-level recovery
  – We associate each empty slot with an error handling procedure.
- Panic mode recovery
  – Modify the stack and/or the input string to try and reach state from which we can continue.

Error recovery in LL(1) parsing

- Panic mode recovery
  - Idea:
    - Decide on a set of synchronizing tokens.
    - When an error is found and there's a nonterminal at the top of the stack, discard input tokens until a synchronizing token is found.
    - Synchronizing tokens are chosen so that the parser can recover quickly after one is found
      - e.g. a semicolon when parsing statements.
    - If there is a terminal at the top of the stack, we could try popping it to see whether we can continue.
      - Assume that the input string is actually missing that terminal.

Error recovery in LL(1) parsing

- Panic mode recovery
  - Possible synchronizing tokens for a nonterminal A
    - the tokens in FOLLOW(A)
      - When one is found, pop A of the stack and try to continue
    - the tokens in FIRST(A)
      - When one is found, match it and try to continue
    - tokens such as semicolons that terminate statements