CSE302: Compiler Design

Instructor: Dr. Liang Cheng
Department of Computer Science and Engineering
P.C. Rossin College of Engineering & Applied Science
Lehigh University

March 13, 2007

Outline

- Recap
  - Top-down parsing (Section 4.4)
- Summary and homework

Top-Down Parsing

- Finding a leftmost derivation for an input string
  - At each step the key problem is determining the production to be applied for a nonterminal, say *A*
  - Recursive-descent parsing
    - Predictive parsing for LL(1) grammars

Recursive-Descent Parsing

```
void A() {
    record input pointer;
    Choose an A-production, A → X1X2 ... Xn
    for (i=1 to n) {
        if (Xi is a nonterminal) call Xi();
        else if (Xi equals the current input a)
            advance the input to the next symbol;
        else  /* an error occurred, backtrack */
    }
}
```

- Grammar
  - $S \rightarrow cAd$
  - $A \rightarrow ab \mid a$
- Input string
  - *cad*

Predictive Parsers

- Recursive-descent parsers with one input symbol lookahead that requires no backtracking
  - No backtracking means deterministic in choosing a production
  - Can be constructed for a class of grammars called LL(1)
    - 1st L: scanning the input from left to right
    - 2nd L: producing a leftmost derivation

FIRST and FOLLOW Sets

- FIRST(α) is the set of terminals that begin strings derived from α
- FOLLOW sets for ALL nonterminals A
  - Place \$ in FOLLOW(S), where S is the start symbol, and \$ is the input right endmarker
    - \$ is not a symbol of any grammar
  - If there is a production $A \rightarrow \alpha B \beta$, then everything in FIRST(β) except ε is in FOLLOW(B)
  - If there is a production $A \rightarrow \alpha B$, or a production $A \rightarrow \alpha B \beta$, where FIRST($\beta$) contains ε (i.e. $\beta \Rightarrow^* \varepsilon$), then everything in FOLLOW(A) is is in FOLLOW(B)
    - Whatever followed A must follow B

LL(1) Grammars

- Whenever $A \rightarrow \alpha$ and $A \rightarrow \beta$ are two distinct A-productions of G, the following conditions hold
  - For no terminal a do both α and β derive strings beginning with a
  - At most one of α and β can derive the empty string
  - If $\beta \overset{*}{\Rightarrow} \varepsilon$, then α does not derive any string beginning with a terminal in FOLLOW(A)
  - If $\alpha \overset{*}{\Rightarrow} \varepsilon$, then β does not derive any string beginning with a terminal in FOLLOW(A)

Why Such Conditions?

- In top-down parsing
  - At each step the key problem is determining the production to be applied for a nonterminal, say A
- $S \underset{lm}{\overset{*}{\Rightarrow}} \gamma A \lambda$
- $A \rightarrow \alpha$ and $A \rightarrow \beta$
  - FIRST(α) and FIRST(β) should be disjoint sets
  - If a is in FIRST(β) then choose $A \rightarrow \beta$
  - $S \overset{*}{\Rightarrow} \gamma A \lambda \Rightarrow \gamma \beta \lambda$
  - Except the following possibility in applying $A \rightarrow \alpha$
  - $S \overset{*}{\Rightarrow} \gamma A \lambda \Rightarrow \gamma \alpha \lambda \overset{*}{\Rightarrow} \gamma \lambda$
    Thus we should eliminate such possiblity for deterministic decision of choosing a A-production
    - If ε is in First(α), then FOLLOW(A) and FIRST(β) should be disjoint sets

Predictive Parsing For LL(1) Grammar

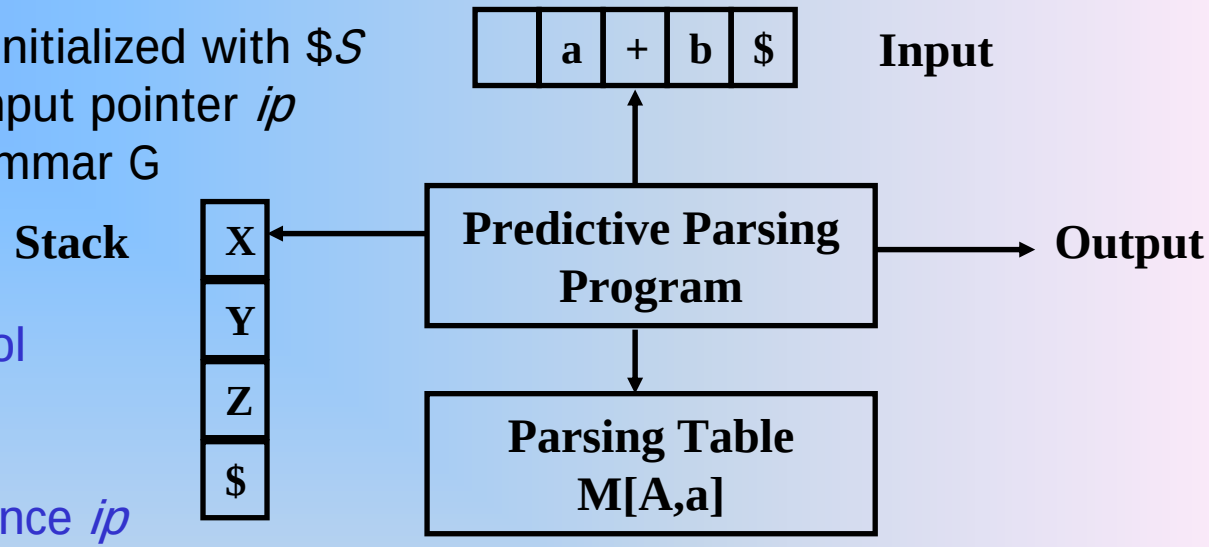
- The production $A \rightarrow \alpha$ is chosen if
  - The next input symbol *a* is in FIRST(α)
  - The next input symbol *a* (or $) is in FOLLOW(A) and ε is in FIRST(α)
    - The next symbol could be $
- Thus we should construct a parsing table M where M[*A*,*a*] = $A \rightarrow \alpha$
  - In function *A* if the input is *a*, then call functions and/or match terminals of α

Constructing A Predictive Parsing Table M For ANY Grammar G

- For each production $A \rightarrow \alpha$
  - For each terminal a in FIRST(A), add $A \rightarrow \alpha$ to M[A,a]
  - If ε is in FIRST(α), then for each terminal b in FOLLOW(A), add $A \rightarrow \alpha$ to M[A,b]
  - If ε is in FIRST(α) and if \$ is in FOLLOW(A), add $A \rightarrow \alpha$ to M[A,\$]
- If, after performing the above, there is no production at all in M[A,a], then set M[A,a] to **error**

Non-recursive Predictive Parsing

- A stack storing symbols, initialized with $\$S$
- An input buffer with an input pointer *ip*
- A parsing table M for grammar G



Point *ip* to the 1st input symbol
Set *A* to the top stack symbol
while($A \neq \$$) {
    if (*A* is *a*) pop stack; advance *ip*
    else if (*A* is a terminal) error();
    else if (M[*A*,*a*] is an error entry) error();
    else if (M[*A*,*a*] = $A \rightarrow X_1X_2...X_k$) {
        output the production or other actions;
        pop the stack;
        push X_k, ... ,X_2 , X_1 onto the stack with X_1 on top;
    }
    Set *A* to the top stack symbol;
}

Error Recovery in Predictive Parsing

- A non-terminal on stack top and M(A,a) empty
  - Idea: skip symbols on the input until a token in a selected set of synchronizing tokens is found.
  - The choice for a synchronizing set is important
    - Note that symbols in FIRST(A) are by default in the synchronizing set because they have entries in the parsing table. In this case we can skip input and once we find a token in FIRST(A) we resume parsing with A.
    - Define the synchronizing set of A to be FOLLOW(A), then skip input until a token in FOLLOW(A) appears and pop A from the stack. Resume parsing...
    - Productions that lead to $\in$ if available might be used.
- A terminal on stack top and not match the input
  - Pop it and continue parsing (issuing an error message)
  - Or skip input until a matching terminal is reached

Implementing Error Recovery in Predictive Parsing

- General approach: modify the empty cells in the parsing table
  - if M[*A*,*a*] = {empty} and *a* belongs to Follow(*A*) then we set M[*A*,*a*] = "synch"
  - Error-recovery strategy for X=top-of-the-stack and *a*=current-input,
    - If *X* is a nonterminal and M[*X*,*a*] = {empty} then skip over *a* in the input
    - If *X* is a nonterminal and M[*X*,*a*] = {synch} then pop *X* off of stack
    - If *X* is a terminal and *X*!=*a* then pop the terminal (essentially inserting it)

Revised Parsing Table

Non-terminal	INPUT SYMBOL					
	id	+	*	(	)	$
E	E→TE'	____	____	E→TE'	____	____
E'	____	E'→+TE'	____	____	E'→∈	E'→∈
T	T→FT'	____	____	T→FT'	____	____
T'	____	T'→∈	T'→*FT'	____	T'→∈	T'→∈
F	F→id	____	____	F→(E)	____	____

Skip input symbol.

From Follow sets. Pop nonterminal on top of stack.

It's a "synch" action.

- $E \rightarrow T\,E'$
- $E' \rightarrow +\ T\,E' \mid \varepsilon$
- $T \rightarrow F\,T'$
- $T' \rightarrow *\ F\,T' \mid \varepsilon$
- $F \rightarrow (\ E\ ) \mid$ **id**

Example Error Messages

- Every non-terminal symbolizes an abstract language construct.
  - E means an expression
    - top-of-stack is E, input is +
      "Error at location i, expressions cannot start with a '+'" or
      "error at location i, invalid expression"
  - T means a summation term
    - Top-of-stack is T, input is *
      "error at location i, invalid operand."
  - When the top-of-the stack is a terminal that does not match...
    - Top-of-stack is id and the input is +
      "error at location i: identifier expected"
    - Top-of-stack is) and the input is terminal other than)
      "error at location i: left parenthesis at location m has no closing right parenthesis"

Implement Top-down Parsers

- Stack: write ADT to manipulate its contents
- Input stream: responsibility of lexical analyzer
- Key issue: how is parsing table implemented? Assign unique IDs

Non-terminal	INPUT SYMBOL					
	id	+	*	(	)	$
E	E→TE'			E→TE'	synch	synch
E'		E'→+TE'			E'→∈	E'→∈
T	T→FT'	synch		T→FT'	synch	synch
T'		T'→∈	T'→*FT'		T'→∈	T'→∈
F	F→id	synch	synch	F→(E)	synch	synch

All rules have unique IDs

Ditto for synch actions

Also for blanks which handle errors

Implement Top-down Parsers

Non-terminal	INPUT SYMBOL					
	id	+	*	(	)	$
E	1	18	19	1	9	10
E’	20	2	21	22	3	3
T	4	11	23	4	12	13
T’	24	6	5	25	6	6
F	8	14	15	7	16	17

1 $E \rightarrow TE'$
2 $E' \rightarrow +TE'$
3 $E' \rightarrow \in$
4 $T \rightarrow FT'$
5 $T' \rightarrow *FT'$
6 $T' \rightarrow \in$
7 $F \rightarrow (E)$
8 $F \rightarrow id$

9 – 17 : Sync Actions

18 – 25 : Error Handlers

Implement Top-down Parsers

- Each # (or set of #s) corresponds to a procedure that:
  - Uses stack ADT
  - Gets tokens
  - Prints error messages
  - Prints diagnostic messages
  - Handles errors

Implement Top-down Parsers

```
state = M[ top(s), current_token ]
switch (state)
  {
    case 1:   proc_E_TE'( ) ;
              break ;
    …
    case 8:  proc_F_id( ) ;
              break ;
    case 9:   proc_sync_9( ) ;
              break ;
    …
    case 17:  proc_sync_17( ) ;
              break ;
    case 18:
    …
    case 25:
  }
```

Procs to handle errors

Combine → put in another switch

Some sync actions may be same

Some error handlers may be similar

Outline

- Recap
- Top-down parsing (Section 4.4)
- Summary and homework