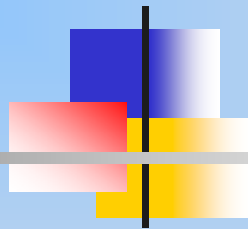
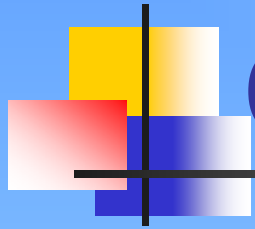


CSE302: Compiler Design



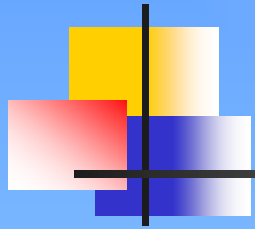
Instructor: Dr. Liang Cheng
Department of Computer Science and Engineering
P.C. Rossin College of Engineering & Applied Science
Lehigh University

February 22, 2007



Outline

- Recap
 - Syntax analysis basics (Sections 4.1 & 4.2)
- Writing a grammar (Section 4.3)
- Top-down parsing (Section 4.4)
- Summary and homework



Input And Output Of Parsers

- A stream of tokens coming from lexer
- Generate some representation of the parse tree
 - Collecting information about tokens into the symbol table
 - Type checking and static semantic analysis
 - Error handling



Notations for Context-free Grammar

- $stmt \rightarrow \mathbf{if} (expr) stmt \mathbf{else} stmt$
- Terminals
 - Lowercase letters early in the alphabet (a, b, c)
 - Operator symbols
 - Punctuation symbols
 - The digits $0, 1, \dots, 9$
 - Boldface strings
- Nonterminals
 - Uppercase letters early in the alphabet (A, B, C, D, E, F) & T
 - E : expressions; T : terms; F : factors
 - Letter S or the head of the 1st production: start symbol
 - Lowercase, italic names



More Notations for Context-free Grammar

- Uppercase letters late in the alphabet (X, Y, Z) represent grammar symbols
 - Either nonterminals or terminals
- Lowercase Greek letters ($\alpha, \beta, \gamma, \dots$) represent strings of grammar symbols
 - $A \rightarrow \alpha$
- Lowercase letter late in the alphabet (u, v, w, x, y, z) represent strings of terminals
- A set of productions $A \rightarrow \alpha_1, A \rightarrow \alpha_2, \dots, A \rightarrow \alpha_k$, with a common head A , may be written as
 - $A \rightarrow \alpha_1 \mid \alpha_2 \mid \dots \mid \alpha_k$



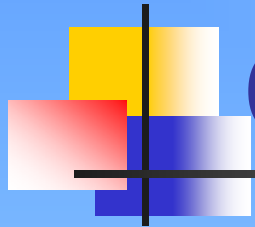
Some Terminologies

- If $S \overset{*}{\Rightarrow}$ means “derives in zero or more steps”
 - $program \overset{*}{\Rightarrow} a = b + \mathbf{const}$
- The symbol $\overset{+}{\Rightarrow}$ means “derives in one or more steps”



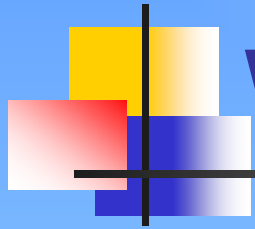
BNF vs. Regular Expressions

- Every construct that can be described by a regular expression can be described by a BNF grammar
- A regular expression may not be able to define a language that can be defined by a BNF



Outline

- Recap
 - Syntax analysis basics (Sections 4.1 & 4.2)
- **Writing a grammar (Section 4.3)**
- Top-down parsing (Section 4.4)
- Summary and homework



Writing A Grammar

- Eliminating ambiguity
- Elimination of left recursion
 - For top-down parsing
- Left factoring
 - For top-down parsing



Eliminating Ambiguity

- Ambiguity associated with operator precedence
- Ambiguity associated with operator associativity
- Dangling-else ambiguity
 - `stmt` → **if** `expr` **then** `stmt`
 - | **if** `expr` **then** `stmt` **else** `stmt`
 - | `other`



Eliminating Ambiguity

- Ambiguity associated with operator precedence
- Ambiguity associated with operator associativity
- Dangling-else ambiguity
 - Add a disambiguity rule
 - Match each **else** with the closest unmatched **then**



Remove Left Recursion (01/25)

$$A \rightarrow A\alpha \mid A\beta \mid \gamma$$

$$A \rightarrow \gamma R$$

$$R \rightarrow \alpha R \mid \beta R \mid \varepsilon$$



Eliminating Left Recursion

$$A \rightarrow A\alpha_1 \mid A\alpha_2 \mid \dots \mid A\alpha_m \mid \beta_1 \mid \beta_2 \mid \dots \mid \beta_n$$

$$A \rightarrow \beta_1 A' \mid \beta_2 A' \mid \dots \mid \beta_n A'$$

$$A' \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \dots \mid \alpha_m A' \mid \varepsilon$$



Eliminating Left Recursion

- Immediately left recursive

$$A \rightarrow A\alpha_1 \mid A\alpha_2 \mid \dots \mid A\alpha_m \mid \beta_1 \mid \beta_2 \mid \dots \mid \beta_n$$
$$\Leftrightarrow$$

$$A \rightarrow \beta_1 A' \mid \beta_2 A' \mid \dots \mid \beta_n A'$$

$$A' \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \dots \mid \alpha_m A' \mid \varepsilon$$

- How about non-immediately left recursive productions?

- $A \xrightarrow{+} A\alpha$



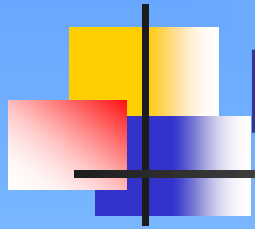
Eliminating Left Recursion

- Grammar G with no cycles or ε -productions
 - Arrange the nonterminals in order A_1, A_2, \dots, A_n
for (each i from 1 to n) {
for (each j from 1 to $i-1$) {
replace $A_i \rightarrow A_j \alpha$ by $A_i \rightarrow \beta_1 \alpha \mid \beta_2 \alpha \mid \dots \mid \beta_k \alpha$ using
existing A_j -productions of $A_j \rightarrow \beta_1 \mid \beta_2 \mid \dots \mid \beta_k$
}
eliminate the immediate left recursions among
the A_i -productions
}
}
- $S \rightarrow Aa \mid b$
 $A \rightarrow Ac \mid Sd \mid \varepsilon$



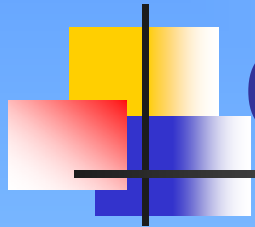
Left Factoring

- When the choices between two alternative A-productions is not clear
 - Rewrite the productions to defer the decision until enough of the input has been seen
 - `stmt` → **if** `expr` **then** `stmt`
| **if** `expr` **then** `stmt` **else** `stmt`
| `other`



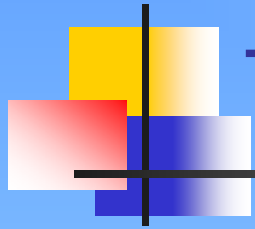
Left Factoring

- For each nonterminal A , find the longest prefix α common to two or more of its alternatives
 - $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \dots \mid \alpha\beta_n \mid \gamma$
- Replace the above A -productions as
 - $A \rightarrow \alpha A' \mid \gamma$
 - $A' \rightarrow \beta_1 \mid \beta_2 \mid \dots \mid \beta_n$



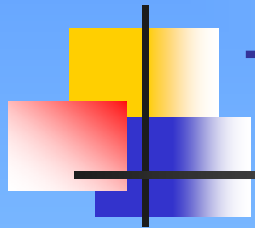
Outline

- Recap
 - Syntax analysis basics (Sections 4.1 & 4.2)
- Writing a grammar (Section 4.3)
- **Top-down parsing (Section 4.4)**
- Summary and homework



Top-Down Parsing

- Creating the parse-tree nodes in preorder (depth-first)
 - Finding a leftmost derivation for an input string
- $E \rightarrow E + T \mid T$
 $T \rightarrow T * F \mid F$
 $F \rightarrow (E) \mid \mathbf{id}$
- Draw the parse tree for the input **id+id*id**



Top-Down Parsing

- At each step the key problem is determining the production to be applied for a nonterminal, say A
 - Recursive-descent parsing
 - May require backtracking to find the correct A -production
 - Predictive parsing
 - No backtracking is required
 - Look ahead at the input a fixed number (k) of symbols
 - $LL(k)$ class grammars



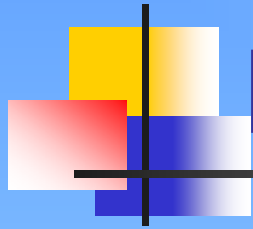
Recursive-Descent Parsing

```
■ void A() {  
    Choose an A-production,  $A \rightarrow X_1X_2 \dots X_n$   
    for ( $i=1$  to  $n$ ) {  
        if ( $X_i$  is a nonterminal) call  $X_i()$ ;  
        else if ( $X_i$  equals the current input  $a$ )  
            advance the input to the next symbol;  
        else /* an error occurred */  
    }  
}
```



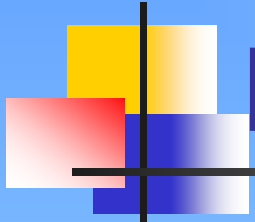
An Backtrack Example

- 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
 - 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



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 $\text{FOLLOW}(A)$
 - If $\alpha \overset{*}{\Rightarrow} \varepsilon$, then β does not derive any string beginning with a terminal in $\text{FOLLOW}(A)$



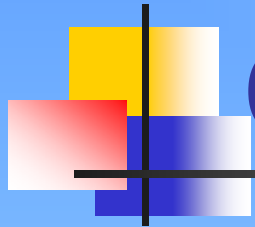
FIRST Function and Set

- During top-down parsing, FIRST and FOLLOW allow us to choose which production to apply
 - $\text{FIRST}(\alpha)$ is the set of **terminals** that begin strings derived from α
 - If $\alpha \overset{*}{\Rightarrow} \varepsilon$, then ε is also in $\text{FIRST}(\alpha)$
- $A \rightarrow \alpha$ and $A \rightarrow \beta$
 - $\text{FIRST}(\alpha)$ and $\text{FIRST}(\beta)$ are disjoint sets
 - If a is in $\text{FIRST}(\alpha)$ then choose $A \rightarrow \alpha$



Compute FIRST Set

- If X is a terminal, then $\text{FIRST}(X) = \{X\}$
- If X is a nonterminal and $X \rightarrow Y_1 Y_2 \dots Y_k$
 - If $X \rightarrow \varepsilon$ is a production, then add ε to $\text{FIRST}(X)$
 - Place a in $\text{FIRST}(X)$ if for some i , a is in $\text{FIRST}(Y_i)$ and ε is in all of $\text{FIRST}(Y_1), \dots, \text{FIRST}(Y_{i-1})$
 - If ε is in all of $\text{FIRST}(Y_1), \dots, \text{FIRST}(Y_k)$, then add ε to $\text{FIRST}(X)$
- Everything in $\text{FIRST}(Y_1)$ is in $\text{FIRST}(X)$
- If Y_1 does not derive ε , then stop
- If Y_1 does derive ε , then add $\text{FIRST}(Y_2)$ to $\text{FIRST}(X)$
- If Y_2 does not derive ε , then stop
- If Y_2 does derive ε , then add $\text{FIRST}(Y_3)$ to $\text{FIRST}(X)$
- ...
- **Examples**



Outline

- Recap
 - Syntax analysis basics (Sections 4.1 & 4.2)
- Writing a grammar (Section 4.3)
- Top-down parsing (Section 4.4)
- **Summary and homework**