

Syntax Analysis

Chapter 1, Section 1.2.2

Chapter 4, Section 4.1, 4.2, 4.3, 4.4, 4.5

CUP Manual

Inside the Compiler: Front End

- Lexical analyzer (aka scanner)
 - Provides a stream of token to the syntax analyzer (aka parser), which creates a **parse tree**
 - Usually the parser calls the scanner: **getNextToken()**
- Syntax analyzer (aka parser)
 - Based on a **grammar** which specifies precisely the syntactic structure of well-formed programs
 - Token names are terminal symbols of this grammar
 - A parse tree does **not** need to be constructed explicitly
 - The parser could be integrated with the **semantic analyzer** and the **generator of intermediate code**
 - Error checking & recovery is an important concern

Languages and Grammars (1/2)

- **Alphabet**: finite set Σ of symbols (e.g. token names)
- **String** over an alphabet: finite sequence of symbols
 - Empty string ϵ ; Σ^* - set of all strings over Σ (incl. ϵ);
 Σ^+ - set of all non-empty strings over Σ
- **Language**: countable set of strings $L \subseteq \Sigma^*$
- **Grammar**: $G = (N, T, S, P)$
 - Finite set of **nonterminal** symbols N , finite set of **terminal** symbols T , starting nonterminal $S \in N$, finite set of **productions** P
 - For us: terminal = token name (we'll say "token")
 - Defines a language over the alphabet T

Languages and Grammars (2/2)

- Production: $x \rightarrow y$ where $x \in (N \cup T)^+$, $y \in (N \cup T)^*$
 - All $S \rightarrow y$ (S is the starting nonterminal) are shown first
- Applying a production: $u x v \Rightarrow u y v$
- String derivation
 - $w_1 \Rightarrow w_2 \Rightarrow \dots \Rightarrow w_n$; denoted $w_1 \xRightarrow{*} w_n$
- Language generated by a grammar
 - $L(G) = \{ w \in T^* \mid S \xRightarrow{*} w \}$
- Classification of languages and grammars: regular \subset context-free \subset context-sensitive \subset unrestricted
 - **Regular**: equivalent to regular expressions/NFA/DFA
 - **Context-free**: used in programming languages

Context-Free Grammars

- Productions: $x \rightarrow y$ where $x \in N$, $y \in (N \cup T)^*$
 - x is a single nonterminal: the **left side** (or **head**)
 - y has zero or more terminals and nonterminals: the **right side** (or **body**) of the production
 - E.g. $expr \rightarrow expr + const$
- Alternative notation: Backus-Naur Form (BNF)
 - E.g. $\langle expr \rangle ::= \langle expr \rangle + \langle const \rangle$
- Notation we will use in this course – see Sect. 4.2.2
- Example: simple arithmetic expressions

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

$$T \rightarrow T * F \mid T / F \mid F$$

$$F \rightarrow (E) \mid id$$

Derivations and Parse Trees

- **Sentential form**: anything derivable from the starting nonterminal
 - If it contains only terminals: **sentence**
- **Leftmost** derivation: the leftmost nonterminal of each sentential form is always chosen
 - Rightmost derivation: the rightmost nonterminal
- Each derivation can be represented by a **parse tree**
 - Leaves are terminals or nonterminals
 - Left-to-right, they constitute a sentential form
- Many-to-one mapping from derivation to tree
 - Thus, we often use the leftmost derivation or the rightmost derivation (1-to-1)

Ambiguity

- **Ambiguous grammar:** more than one parse tree for some sentence
 - Choice 1: make the grammar unambiguous
 - Choice 2: leave the grammar ambiguous, but define some disambiguation rules for use during parsing
- Example: the dangling-else problem

```
stmt →  if expr then stmt
      |  if expr then stmt else stmt
      |  other
```
- Two parse trees for **if a then if b then x=1 else x=2**
- See a non-ambiguous version in Fig 4.10
 - **else** is matched with the closest unmatched **then**

Elimination of Ambiguity

$expr \rightarrow expr + expr \mid expr * expr \mid (expr) \mid id$

1. Prove that this grammar is ambiguous
2. Create an equivalent non-ambiguous grammar with the appropriate precedence and associativity
 - $*$ has higher precedence than $+$
 - both are left-associative

Example: parse tree for $a + b * (c + d) * e$

Top-Down Parsing

- Goal: find the leftmost derivation for a given string
- General solution: **recursive-descent parsing**
 - Need to eliminate any **left recursion** from the grammar
 - In the general case, may require **backtracking**: multiple scans over the input
- **Predictive parsing**: no need for backtracking
 - **LL(k) grammars**: only need to look at the next k symbols to decide which production to apply
 - Important case in practice: LL(1) grammars
 - May need to perform **left factoring** of the grammar

Elimination of Left Recursion

- **Left-recursive** grammar: possible $A \Rightarrow \dots \Rightarrow A\alpha$
- Simple case
 - Original grammar: $A \rightarrow A\alpha \mid \beta$
 - New grammar: $A \rightarrow \beta A'$ and $A' \rightarrow \alpha A' \mid \varepsilon$
- More complex case
 - Original: $A \rightarrow A\alpha_1 \mid \dots \mid A\alpha_m \mid \beta_1 \mid \dots \mid \beta_n$
 - New: $A \rightarrow \beta_1 A' \mid \dots \mid \beta_n A'$ and $A' \rightarrow \alpha_1 A' \mid \dots \mid \alpha_m A' \mid \varepsilon$
- Still not enough
 - E.g. S is left-recursive in $S \rightarrow A\mathbf{a} \mid \mathbf{b}$ and $A \rightarrow A\mathbf{c} \mid S\mathbf{d} \mid \varepsilon$
- Section 4.3.3: algorithm for grammars w/o cycles ($A \Rightarrow \dots \Rightarrow A$) and w/o ε -productions ($A \rightarrow \varepsilon$)

Example with Left Recursion

- Original grammar

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

$$T \rightarrow T * F \mid T / F \mid F$$

$$F \rightarrow (E) \mid \text{id}$$

- Modified grammar

$$E \rightarrow T E'$$

$$E' \rightarrow + T E' \mid - T E' \mid \varepsilon$$

$$T \rightarrow F T'$$

$$T' \rightarrow * F T' \mid / F T' \mid \varepsilon$$

$$F \rightarrow (E) \mid \text{id}$$

Recursive-Descent Parsing

- One procedure for each nonterminal
- Parsing starts with a call to the procedure for the starting nonterminal
 - Success: if at the end of this call, the entire input string has been processed (no leftover symbols)

```
void A() /* procedure for a nonterminal A */  
  choose some production  $A \rightarrow X_1 X_2 \dots X_k$   
  for (i = 1 to k)  
    if ( $X_i$  is nonterminal) call  $X_i()$   
    else if ( $X_i$  is equal to the current input symbol)  
      move to the next input symbol  
    otherwise report parse error
```

A Few Issues

- Choosing which production $A \rightarrow X_1 X_2 \dots X_k$ to use
 - There could be many possible productions for A
 - If one of the choices does not work, backtrack the algorithm and try another choice
 - Expensive and undesirable in practice
- Top-down parsing for programming languages: **predictive** recursive-descent (no backtracking)
- A left-recursive grammar may lead to infinite recursion (even if we have backtracking)
 - When we try to expand A , we eventually reach A again without having consumed any symbols in between

Sets FIRST

- For any string α of grammar symbols: $\text{FIRST}(\alpha)$ contains any terminal that could be the first symbol of some string derived from α
 - $\alpha \xRightarrow{*} a\beta$ means $a \in \text{FIRST}(\alpha)$ where a is a terminal
 - $\alpha \xRightarrow{*} \epsilon$ means $\epsilon \in \text{FIRST}(\alpha)$
- For $A \rightarrow \alpha \mid \beta$, if $\text{FIRST}(\alpha)$ and $\text{FIRST}(\beta)$ are disjoint, we can predict which production should be used simply by looking at the current input symbol
 - Basis for predictive parsing through LL(1) grammars

Computing FIRST

- For any grammar symbol X
 - If X is a terminal: $\text{FIRST}(X) = \{ X \}$
 - If X is a nonterminal: for any production $X \rightarrow Y_1 Y_2 \dots Y_n$
 - Any terminal in $\text{FIRST}(Y_1)$ is in $\text{FIRST}(X)$
 - If $\text{FIRST}(Y_1)$ contains ϵ , any terminal in $\text{FIRST}(Y_2)$ is in $\text{FIRST}(X)$
 - If $\text{FIRST}(Y_2)$ contains ϵ , etc.
 - If all $\text{FIRST}(Y_i)$ contain ϵ , $\text{FIRST}(X)$ also contains ϵ
 - If $X \rightarrow \epsilon$ is a production, $\text{FIRST}(X)$ contains ϵ
- For any string of grammar symbols $X_1 X_2 \dots X_n$
 - Any terminal in $\text{FIRST}(X_1)$
 - If $\text{FIRST}(X_1)$ contains ϵ , any terminal in $\text{FIRST}(X_2)$, etc.
 - If all $\text{FIRST}(X_i)$ contain ϵ , add ϵ

Sets FOLLOW

- For any nonterminal A : $\text{FOLLOW}(A)$ contains any terminal that could appear immediately to the right of A in some sentential form
 - $S \xRightarrow{*} \alpha A a \beta$ means $a \in \text{FOLLOW}(A)$ where a is a terminal
 - $S \xRightarrow{*} \alpha A$ means $\$ \in \text{FOLLOW}(A)$; $\$$ is a special “endmarker” that is not in the grammar (i.e. end-of-file)
- $\$ \in \text{FOLLOW}(S)$ where S is the starting nonterminal
- $A \rightarrow \alpha B \beta$: everything in $\text{FIRST}(\beta)$ except for ϵ is in $\text{FOLLOW}(B)$
- $A \rightarrow \alpha B$ or $A \rightarrow \alpha B \beta \wedge \epsilon \in \text{FIRST}(\beta)$: everything in $\text{FOLLOW}(A)$ is in $\text{FOLLOW}(B)$

Example of FIRST and FOLLOW Sets

- Grammar with eliminated left recursion

$$E \rightarrow T E'$$

$$E' \rightarrow + T E' \mid - T E' \mid \varepsilon$$

$$T \rightarrow F T'$$

$$T' \rightarrow * F T' \mid / F T' \mid \varepsilon$$

$$F \rightarrow (E) \mid \text{id}$$

- $\text{FIRST}(F) = \text{FIRST}(T) = \text{FIRST}(E) = \{ (, \text{id} \}$
- $\text{FIRST}(E') = \{ +, -, \varepsilon \}$ and $\text{FIRST}(T') = \{ *, /, \varepsilon \}$
- $\text{FOLLOW}(E) = \text{FOLLOW}(E') = \{ \$,) \}$
- $\text{FOLLOW}(T) = \text{FOLLOW}(T') = \{ +, -, \$,) \}$
- $\text{FOLLOW}(F) = \{ *, /, +, -, \$,) \}$

LL(1) Grammars

- Suitable for predictive (no backtracking) recursive-descent parsing
 - LL = “scan the input left-to-right; produce a leftmost derivation”; 1 = “use 1 symbol to decide”
 - A left-recursive grammar cannot be LL(1)
 - An ambiguous grammar cannot be LL(1)
- For any $A \rightarrow \alpha \mid \beta$
 - $\text{FIRST}(\alpha)$ and $\text{FIRST}(\beta)$ are disjoint sets (including for ϵ)
 - If $\epsilon \in \text{FIRST}(\alpha)$: $\text{FIRST}(\beta)$ and $\text{FOLLOW}(A)$ are disjoint
 - If $\epsilon \in \text{FIRST}(\beta)$: $\text{FIRST}(\alpha)$ and $\text{FOLLOW}(A)$ are disjoint
- The production to apply can be chosen based on the current input symbol

LL(1) Parser

- Define a predictive parsing table
 - A row for a nonterminal A , a column for a terminal a
 - Cell $[A, a]$ is the production that should be applied when we are inside A 's parsing procedure and we see a
 - If the grammar is LL(1) – only one choice per cell

	id	+	-	*	/	()	\$
E	$E \rightarrow TE'$					$E \rightarrow TE'$		
E'		$E' \rightarrow +TE'$	$E' \rightarrow -TE'$				$E' \rightarrow \epsilon$	$E' \rightarrow \epsilon$
T	$T \rightarrow FT'$					$T \rightarrow FT'$		
T'		$T' \rightarrow \epsilon$	$T' \rightarrow \epsilon$	$T' \rightarrow *FT'$	$T' \rightarrow /FT'$		$T' \rightarrow \epsilon$	$T' \rightarrow \epsilon$
F	$F \rightarrow \text{id}$					$F \rightarrow (E)$		

Left Factoring of a Grammar

- The decision is impossible due to a common prefix
- Original grammar: $A \rightarrow \gamma \mid \alpha\beta_1 \mid \dots \mid \alpha\beta_n$
- New grammar: $A \rightarrow \gamma \mid \alpha A'$ and $A' \rightarrow \beta_1 \mid \dots \mid \beta_n$
- Example (ignore the ambiguity for now)

$stmt \rightarrow$ **if** *expr* **then** *stmt*
| **if** *expr* **then** *stmt* **else** *stmt*
| **other**

- Left-factored version

$stmt \rightarrow$ **if** *expr* **then** *stmt* *rest* | **other**
 $rest \rightarrow$ **else** *stmt* | ϵ

Example: Dangling Else

- Full grammar

$stmt \rightarrow \text{if } expr \text{ then } stmt \text{ rest} \mid \text{other}$

$rest \rightarrow \text{else } stmt \mid \varepsilon$

$expr \rightarrow \text{bool}$

- $FIRST(stmt) = \{ \text{if}, \text{other} \}$ $FIRST(rest) = \{ \text{else}, \varepsilon \}$

- $FOLLOW(stmt) = FOLLOW(rest) = \{ \$, \text{else} \}$

	other	bool	else	if	then	\$
<i>stmt</i>	$stmt \rightarrow \text{other}$			$stmt \rightarrow \text{if } expr \text{ then } stmt \text{ rest}$		
<i>rest</i>			$rest \rightarrow \text{else } stmt$ $rest \rightarrow \varepsilon$			$rest \rightarrow \varepsilon$
<i>expr</i>		$expr \rightarrow \text{bool}$				



Another Algorithm: Explicit Stack

- Instead of using recursion (which implicitly maintains a call stack): use an explicit stack
 - X is top of stack, a is current input symbol
- Push S on top of stack
- While stack is not empty
 - If ($X == a$)
 - Pop stack and move to the next input symbol
 - Else if ($X ==$ some other terminal) Error
 - Else if (table cell $[X, a]$ is empty) Error
 - Else: table cell $[X, a]$ contains $X \rightarrow Y_1 Y_2 \dots Y_n$
 - Pop stack
 - Push Y_n , Push Y_{n-1} , ..., Push Y_1

Bottom-Up Parsing

- In general, more powerful than top-down parsing
 - E.g., $LL(k)$ grammars are not as general as $LR(k)$
- Basic idea: start at the leaves and work up
 - The parse tree “grows” upwards
- **Shift-reduce parsing**: general style of bottom-up parsing
 - Used for parsing $LR(k)$ grammars
 - Used by **automatic parser generators**: given a grammar, it generates a shift-reduce parser for it (e.g., yacc, CUP)
 - yacc = “Yet Another Compiler Compiler”
 - CUP = “Constructor of Useful Parsers”

Reductions

- Expressions again (OK to be left-recursive)

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

$$T \rightarrow T * F \mid T / F \mid F$$

$$F \rightarrow (E) \mid \text{id}$$

- At a **reduction step**, a substring matching the body of a production is replaced with the head

– E.g., $E + T$ is reduced to E because of $E \rightarrow E + T$

- Parsing is a sequence of reduction steps

$$(1) \text{id} * \text{id} \quad (2) F * \text{id} \quad (3) T * \text{id}$$

$$(4) T * F \quad (5) T \quad (6) E$$

- This is a **derivation in reverse**: $E \Rightarrow T \Rightarrow T * F \Rightarrow$

$$T * \text{id} \Rightarrow F * \text{id} \Rightarrow \text{id} * \text{id}$$

Overview of Shift-Reduce Parsing (1/2)

- Left-to-right scan of the input
- Perform a sequence of reduction steps which correspond (in reverse) to a **rightmost** derivation
 - If the grammar is not ambiguous: there exists a unique rightmost derivation $S = \gamma_0 \Rightarrow \gamma_1 \Rightarrow \dots \Rightarrow \gamma_n = w$
 - Each step also updates the tree (adds a parent node)
- At each reduction step, find a “handle”
 - If $\gamma_k \Rightarrow \gamma_{k+1}$ is $\alpha A v \Rightarrow \alpha \beta v$, then **production $A \rightarrow \beta$ in the position following α** is a handle of γ_{k+1}
 - Note that v is a string of terminals
 - Non-ambiguous grammar: only one handle of γ_{k+1}
 - For convenience we will call β the handle, not $A \rightarrow \beta$

Overview of Shift-Reduce Parsing (2/2)

- A **stack** holds grammar symbols; an **input buffer** holds the rest of the string to be parsed
 - Initially: the stack is empty, the buffer contains the entire input string
 - Successful completion: the stack contains the starting nonterminal, the buffer is empty
- Repeat until success or error
 - **Shift** zero or more input symbols from the buffer to the stack, until the top of the stack forms a handle
 - **Reduce** the handle

Example of Shift-Reduce Parsing

Stack	Input	Action
empty	$\text{id}_1 * \text{id}_2 \$$	Shift
id_1	$* \text{id}_2 \$$	Reduce by $F \rightarrow \text{id}$
F	$* \text{id}_2 \$$	Reduce by $T \rightarrow F$
T	$* \text{id}_2 \$$	Shift
$T *$	$\text{id}_2 \$$	Shift
$T * \text{id}_2$	$\$$	Reduce by $F \rightarrow \text{id}$
$T * F$	$\$$	Reduce by $T \rightarrow T * F$
T	$\$$	Reduce by $E \rightarrow T$
E	$\$$	Accept

Conflicts During Shift-Reduce Parsing

- **LR(k) parser**: knowing the content of the stack and the next k input symbols is enough to decide
 - LR=“scan left-to-right; produce a rightmost derivation”
 - LR(k) grammar: exists an LR(k) parser for it
 - For each LR(k) grammar there is an equivalent LR(1) grammar; thus, we only consider LR(1) parsers
- **Non-LR grammar: conflicts during parsing**
 - Shift/reduce conflict: shift or reduce?
 - Reduce/reduce conflict: several possible reductions
 - Typical example: **any ambiguous grammar**
- See examples in Section 4.5.4

LR Parsers

- A category of shift-reduce parsers
 - Table-driven; no backtracking; efficient
 - Enough for real-world programming languages
 - Detect parse errors early (error messages/recovery)
 - Cover all LL grammars, and beyond
- **SLR parsers** (“simple-LR”, Section 4.6), **LALR parsers** (“lookahead-LR”, Section 4.7), **canonical-LR** (most general; Section 4.7)
 - LARL is the approach most often used in practice – e.g., yacc, bison, CUP
- Many technical details; we will not cover them

CUP Parser Generator

- www.cs.princeton.edu/~appel/modern/java/CUP/
 - These are the “old” versions: 0.10k and older
 - Version 11 available, but we will not use it
- Input: grammar specification
 - Has embedded Java code to be executed during parsing
- Output: a parser written in Java
- Often uses a scanner produced by JLex or JFLex
- Key components of the specification:
 - Terminals and nonterminals
 - Precedence and associativity
 - Productions: terminals, nonterminals, actions

Simple CUP Example

[Assignment: get it from the web page under “Resources”, run it, and understand it – today!]

- **calc** example: already considered for JFlex
 - Sample input: $5*(6-3)+1$;
 - Sample output: $5 * (6 - 3) + 1 = 16$

```
import java_cup.runtime.*;
parser code { : some Java code : };
terminal SEMI, PLUS, MINUS, TIMES, DIVIDE, LPAREN, RPAREN;
terminal Integer NUMBER;
non terminal Object expr_list, expr_part;
non terminal Integer expr, factor, term;
expr_list ::= expr_list expr_part | expr_part;
expr_part ::= expr:e { : System.out.println(" = " + e); : } SEMI;
expr ::= expr:e PLUS factor:f { : RESULT = new Integer(e.intValue() + f.intValue()); : }
      | expr:e MINUS factor:f { : RESULT = new Integer(e.intValue() - f.intValue()); : }
      | factor:f { : RESULT = new Integer(f.intValue()); : };
factor ::= ...
term ::= LPAREN expr:e RPAREN { : RESULT = e; : } | NUMBER:n { : RESULT = n; : } ;
```



Copied in the produced parser.java



Value for token is java.lang.Integer



Starting nonterminal first

Project 2

- Extend Project 1 with a parser
- Use Main from the web page (instead of MyLexer)
 - Similar to the Main class in **calc**
- Each non terminal has an associated String value
 - **non terminal String X;** in simpleC.cup
 - The String value: pretty printing of the sub-tree
 - The String value for the root should be a compilable C program that has exactly the same behavior as the input C program
 - No printing to System.out in the scanner or the parser