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 then creates a parse tree
- Usually the parser calls the scanner: **getNextToken()**

Syntax analyzer (aka parser)

- Based on a context-free grammar which specifies precisely the syntactic structure of well-formed programs
  - Token names are terminal symbols of this grammar
- Error checking, reporting, and recovery are important concerns; we will not discuss them

Overview

Context-free grammars

Ambiguous grammars

Top-down parsing

- Essential first step: elimination of left recursion
- Predictive parsing for LL(1) grammars

Bottom-up parsing

- Example: shift-reduce parsing for LR(1) grammars

Context-Free Grammars

Productions: **x** → **y**

- **x** is a single non-terminal: the left side
- **y** is has zero or more terminals and non-terminals: the right side of the production
- E.g. *expr* → *expr* **+** *const*

Alternative notation: Backus-Naur Form (BNF)

- E.g. <expr> ::= <expr> **+** <const>

Example: simple arithmetic expressions

E → *E* **+** *T* | *E* **-** *T* | *T*

T → *T* ***** *F* | *T* **/** *F* | *F*

F → **(** *E* **)** | **id**

Derivations and Parse Trees

Start with the starting non-terminal, apply productions until a string of terminals is derived

- Leftmost derivation: the leftmost non-terminal at each step is chosen for expansion
- Rightmost derivation: the rightmost non-terminal

Each derivation can be represented by a parse tree

- Leaves are terminals or non-terminals
- After a full derivation: leaves are terminals (or **ε**)

Parser: builds the parse tree for a given string of terminals

Example: using the grammar from the previous slide, show the parse tree for **a + b * (c + d) * e**

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 to 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**

- Choice 1: complex non-ambiguous version in Fig 4.10 in the Dragon Book (**else** is matched with the closest unmatched **then**); do not need to study it
- Choice 2: a "hint" to the parser (used in our projects)

Elimination of Ambiguity

expr → *expr* **+** *expr* | *expr* ***** *expr* | **(** *expr* **)** | **id**

Why is this grammar ambiguous?

Earlier grammar: equivalent non-ambiguous grammar with the "normal" precedence and associativity

- ***** has higher precedence than **+**
- both are left-associative

Recall the 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

- To use this: need to eliminate any left recursion from the grammar
- In the general case, parsing may require backtracking

Predictive recursive-descent parsing

- LL(*k*) grammars: only need to look at the next *k* symbols to decide which production to apply (no backtracking)
    - Important case in practice: LL(1) grammars

Prerequisite: Elimination of Left Recursion

Left-recursive grammar: possible $A \Rightarrow \ldots \Rightarrow A\alpha$

Simple case (here α and β are arbitrary sequences of terminals and non-terminals)

- 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: $A \rightarrow A\alpha_1 \mid \ldots \mid A\alpha_m \mid \beta_1 \mid \ldots \mid \beta_n$
- New: $A \rightarrow \beta_1 A' \mid \ldots \mid \beta_n A'$ and $A' \rightarrow \alpha_1 A' \mid \ldots \mid \alpha_m A' \mid \varepsilon$

Still not enough

- E.g. S is left-recursive in $S \rightarrow A\mathbf{a} \mid \mathbf{b}$ and $A \rightarrow A\mathbf{c} \mid S\mathbf{d} \mid \varepsilon$
- More details in Section 4.3.3 of Dragon book; we will not discuss in this course

Example with Left Recursion

Original grammar

$E \rightarrow E \mathbf{+} T \mid E \mathbf{-} T \mid T$

$T \rightarrow T \mathbf{*} F \mid T \mathbf{/} F \mid F$

$F \rightarrow \mathbf{(}\, E \,\mathbf{)} \mid \mathbf{id}$

Modified grammar

$E \rightarrow T\, E'$

$E' \rightarrow \mathbf{+}\, T\, E' \mid \mathbf{-}\, T\, E' \mid \varepsilon$

$T \rightarrow F\, T'$

$T' \rightarrow \mathbf{*}\, F\, T' \mid \mathbf{/}\, F\, T' \mid \varepsilon$

$F \rightarrow \mathbf{(}\, E \,\mathbf{)} \mid \mathbf{id}$

Recursive-Descent Parsing

One procedure for each non-terminal

Parsing starts with a call to the procedure for the starting non-terminal

- Success: if at the end of this call, the entire input string has been processed (no leftover symbols)

void A() /* procedure for a non-terminal *A* */
    choose some production $A \rightarrow X_1\ X_2\ \ldots\ X_k$
    for (i = 1 to k)
        if (X_i is non-terminal) 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 $A \rightarrow X_1\ X_2\ \ldots\ X_k$ 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)

LL(1) Grammars

Suitable for predictive 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 $A \rightarrow \alpha \mid \beta$

- FIRST(α) and FIRST(β) must be disjoint sets
  - FIRST(α) = terminals that could be the first symbol of something derived from $\boldsymbol{\alpha}$
- If current input symbol is in FIRST(α): use $A \rightarrow \alpha$
- If current input symbol is in FIRST(β): use $A \rightarrow \beta$
- Otherwise report parsing error
- Only look at current input symbol to make a decision

Some Examples of Sets FIRST

Grammar with eliminated left recursion

$E \rightarrow T\,E'$

$E' \rightarrow$ **+** $T\,E'$ | **-** $T\,E'$ | ε

$T \rightarrow F\,T'$

$T' \rightarrow$ ***** $F\,T'$ | **/** $F\,T'$ | ε

$F \rightarrow$ **(** E **)** | **id**

FIRST(F) = FIRST(T) = FIRST(E) = { **(** , **id** }

FIRST(E') = { **+** , **-** , ε } and FIRST(T') = { ***** , **/** , ε }

FIRST(**(** E **)**) = { **(** } and FIRST(**id**) = { **id** }

Use for LL(1) parsing:

e.g. for $F \rightarrow$ **(** E **)** | **id**

Parser code for F

```
if (currToken==LPAREN) …
else if (currToken==ID) …
else error()
```

Sets FIRST

For any string α of terminals and non-terminals:

FIRST(α) contains all terminals that could be the first symbol of some string derived from $\boldsymbol{\alpha}$

- $\alpha \overset{*}{\Rightarrow} a\beta$ where *a* is a terminal, means $a \in$ FIRST(α)
- $\alpha \overset{*}{\Rightarrow} \boldsymbol{\varepsilon}$ means $\boldsymbol{\varepsilon} \in$ FIRST(α) – some complications …

The simple cases:

- If α is just a single terminal *a*, FIRST(α) = { *a* }
- If α is a terminal *a* followed by anything, FIRST(α) = { *a* }
- If α is the empty string $\boldsymbol{\varepsilon}$, FIRST(α) = { $\boldsymbol{\varepsilon}$ }

The more complex cases: next slide

- If α is just a single non-terminal
- If α is a non-terminal followed by something

Sets FIRST (cont)

FIRST(X) for a non-terminal X : consider each production $X \rightarrow Y_1\ Y_2 \ldots Y_n$

- Any terminal in FIRST(Y_1) is also in FIRST(X)
- If $\varepsilon \in$ FIRST(Y_1), any terminal in FIRST(Y_2) is in FIRST(X)
  - And if $\varepsilon \in$ FIRST(Y_2), any terminal in FIRST(Y_3) is in FIRST(X), etc.
  - If $\varepsilon \in$ FIRST(Y_i) for all i, FIRST(X) also contains ε
- If $X \rightarrow \varepsilon$ is a production, FIRST(X) contains ε

FIRST($X_1X_2 \ldots X_n$)

- Any terminal in FIRST(X_1)
- If FIRST(X_1) contains ε, any terminal in FIRST(X_2), etc.
- If all FIRST(X_i) contain ε, FIRST($X_1X_2 \ldots X_n$) contains ε

Special Case: $\varepsilon \in$ FIRST(…)

Example: consider $E' \rightarrow$ **+** $T\ E'$ | **-** $T\ E'$ | ε

- FIRST(**+**TE') = { **+** }, FIRST(**-**TE') = { **-** }, FIRST(ε) = { ε }
- When do we choose production $E' \rightarrow \varepsilon$?
- What is the actual code for the parser?

We will not discuss in this course, but there is a systematic approach to handle this; leads to

```
if (currToken==PLUS) {nextToken(); T(); Eprime();}
else if (currToken==MINUS) { … }
else if (currToken==RPAREN ||
         currToken==END_INPUT) { } // do nothing
else error()
```

LL(1) Parser

- Define a predictive parsing table
  - A row for a non-terminal *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	$E \rightarrow T E'$					$E \rightarrow T E'$		
E'		$E' \rightarrow + T E'$	$E' \rightarrow - T E'$				$E' \rightarrow \varepsilon$	$E' \rightarrow \varepsilon$
T	$T \rightarrow F T'$					$T \rightarrow F T'$		
T'		$T' \rightarrow \varepsilon$	$T' \rightarrow \varepsilon$	$T' \rightarrow * F T'$	$T' \rightarrow / F T'$		$T' \rightarrow \varepsilon$	$T' \rightarrow \varepsilon$
F	$F \rightarrow$ **id**					$F \rightarrow (E)$		

Example: **a + b * (c + d) * e**

E	**a + b * (c + d) * e $**	$E \to T E'$
T	**a + b * (c + d) * e $**	$T \to F T'$
F	**a + b * (c + d) * e $**	$F \to$ **id**
T'	**+ b * (c + d) * e $**	$T' \to \varepsilon$
E'	**+ b * (c + d) * e $**	$E' \to$ **+** $T E'$
T	**b * (c + d) * e $**	$T \to F T'$
F	**b * (c + d) * e $**	$F \to$ **id**
T'	*** (c + d) * e $**	$T' \to$ ***** $F T'$
F	**(c + d) * e $**	$F \to$ **(** E **)**
E	**c + d) * e $**	$E \to T E'$
T	**c + d) * e $**	$T \to F T'$
F	**c + d) * e $**	$F \to$ **id**

…

Instead of Procedure Calls: Explicit Stack

Top of stack: terminal or nonterminal X ; current input symbol: terminal a

1. Push S on top of stack
2. 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 $X \rightarrow Y_1Y_2\ldots Y_n$
     Pop stack
     Push Y_n, Push Y_{n-1}, …, Push Y_1

Exercise at home: apply to the example from the previous slide

Different Approach: 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 (now it is OK to be left-recursive)

$E \rightarrow E$ **+** T | E **-** T | T

$T \rightarrow T$ ***** F | T **/** F | F

$F \rightarrow$ **(** E **)** | **id**

At a reduction step, a substring matching the right side a production is replaced with the left size

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

Parsing is a sequence of reduction steps

(1) **id * id** (2) F * **id** (3) T * **id**

(4) T * F (5) T (6) E

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

Overview of Shift-Reduce Parsing

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 $S = \gamma_0 \Rightarrow \gamma_1 \Rightarrow \ldots \Rightarrow \gamma_n = w$
- 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 β 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}

Overview of Shift-Reduce Parsing (cont)

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 non-terminal, 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	$\mathbf{id}_1$ * $\mathbf{id}_2$ **$**	Shift
$\mathbf{id}_1$	* $\mathbf{id}_2$ **$**	Reduce by $F \rightarrow$ **id**
F	* $\mathbf{id}_2$ **$**	Reduce by $T \rightarrow F$
T	* $\mathbf{id}_2$ **$**	Shift
T *	$\mathbf{id}_2$ **$**	Shift
T * $\mathbf{id}_2$	**$**	Reduce by $F \rightarrow$ **id**
T * F	**$**	Reduce by $T \rightarrow T * F$
T	**$**	Reduce by $E \rightarrow T$
E	**$**	Accept

LR Parsers and Grammars

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: we can define an LR(*k*) parser for it

Non-LR grammar: conflicts during parsing

- Shift/reduce conflict: shift or reduce?
- Reduce/reduce conflict: several possible reductions
- Typical example: any ambiguous grammar

SLR parsers ("simple-LR", Section 4.6), LALR parsers ("lookahead-LR", Section 4.7), canonical-LR (most general; Section 4.7); details will not be discussed

CUP Parser Generator

- Input: grammar specification
  - Has embedded Java code to be executed during parsing
- Output: a parser written in Java
- Often uses a scanner produced by JFLex
- Key components of the specification:
  - Terminals and non-terminals
  - Precedence and associativity
  - Productions: terminals, non-terminals, actions
- Project 2: change the parser from Project 1
  - And related changes to scanner and AST
  - **while** loops; **for** loops; operators <, <=, >, >=, ==, !=