Examples Syntax Analysis

Chapter 4

Errors in Programs

**Lexical**

if $x<1$ then $\mathbf{y}=5$:

“Typos”

**Syntactic**

if ((x<1) & (y>5))

{ ... { ... _ ... } }

**Semantic**

if (x+5) then ...

Type Errors

Undefined IDs, etc.

**Logical Errors**

if (i<9) then ...

Should be $\leq$ not $<$

Bugs

Compiler cannot detect Logical Errors
Requirements

Detect All Errors (Except Logical!)
Messages should be helpful.
   Difficult to produce clear messages!
   \textbf{Example}:
   Syntax Error
   \textbf{Example}:
   Line 23: Unmatched Paren
   \texttt{if ((x == 1) then}\n   \texttt{^}

Compiler Should Recover
Keep going to find more errors
\textbf{Example}:
\texttt{x := (a + 5)) \ast (b + 7));}

We’re in the middle of a statement
Skip tokens until we see a “;”
Resume Parsing
Misses a second error... Oh, well...
Checks most of the source
Difficult to generate clear and accurate error messages.

**Example**

```
function foo () {
    ...
    if (...) {
        ...
    } else {
        ...
    
    }

    <eof>
```

- Missing `}` here
- Not detected until here

**Example**

```
var myVarr: Integer;
...

x := myVar;
...
```

- Misspelled ID here
- Detected here as "Undeclared ID"
Error-Correcting Compilers

- Issue an error message
- Fix the problem
- Produce an executable

**Example**

Error on line 23: "myVarr" undefined.
"myVar" was used.

Is this a good idea???

Compiler *guesses* the programmer’s intent
A shifting notion of what constitutes a correct / legal / valid program
May encourage programmers to get sloppy
Declarations provide redundancy
⇒ Increased reliability
Error Recovery Approaches:  Panic Mode

Discard tokens until we see a “synchronizing” token.

Example

Skip to next occurrence of
} end ;
Resume by parsing the next statement

• Simple to implement
• Commonly used
• The key...
  Good set of synchronizing tokens
  Knowing what to do then
• May skip over large sections of source
Error Recovery Approaches: Phrase-Level Recovery

Compiler corrects the program by deleting or inserting tokens...so it can proceed to parse from where it was.

**Example**

```plaintext
while (x = 4) y := a+b; ...
```

Insert **do** to “fix” the statement.

• The key...
  Don’t get into an infinite loop
  ...constantly inserting tokens
  ...and never scanning the actual source
Error Recovery Approaches: Error Productions

Augment the CFG with “Error Productions”
Now the CFG accepts anything!
If “error productions” are used...
    Their actions:
    {print ("Error...")}

Used with...
    • LR (Bottom-up) parsing
    • Parser Generators

Error Recovery Approaches: Global Correction

Theoretical Approach
Find the minimum change to the source to yield a valid program
    (Insert tokens, delete tokens, swap adjacent tokens)
Impractical algorithms - too time consuming
CFG: Context Free Grammars

Example Rule:
Stmt → if Expr then Stmt else Stmt

Terminals
Keywords
else “else”
Token Classes
ID INTEGER REAL
Punctuation
; “;” :

Non-terminals
Any symbol appearing on the lefthand side of any rule

Start Symbol
Usually the non-terminal on the lefthand side of the first rule

Rules (or “Productions”)
BNF: Backus-Naur Form / Backus-Normal Form
Stmt ::= if Expr then Stmt else Stmt
Rule Alternatives

\[ E \rightarrow E + E \]
\[ E \rightarrow (E) \]
\[ E \rightarrow -E \]
\[ E \rightarrow ID \]

\[ E \rightarrow E + E \]
\[ \rightarrow (E) \]
\[ \rightarrow -E \]
\[ \rightarrow ID \]

\[ E \rightarrow E + E \]
\[ \mid (E) \]
\[ \mid -E \]
\[ \mid ID \]

\[ E \rightarrow E + E \mid (E) \mid -E \mid ID \]

All Notations are Equivalent
Notational Conventions

**Terminals**
- a
- b
- c

**Nonterminals**
- A
- B
- C
- S
- Expr

**Grammar Symbols (Terminals or Nonterminals)**
- X
- Y
- Z
- U
- V
- W

**Strings of Symbols**
- α
- β
- γ

**Strings of Terminals**
- x
- y
- z
- u
- v
- w

**Examples**

A → α B
- A rule whose right-hand side ends with a nonterminal

A → x α
- A rule whose right-hand side begins with a string of terminals (call it “x”)
Derivations

1. \[ E \rightarrow E + E \]
2. \[ E \rightarrow E \times E \]
3. \[ E \rightarrow (E) \]
4. \[ E \rightarrow -E \]
5. \[ E \rightarrow \text{ID} \]

A “Derivation” of “(id*id)”

\[
E \Rightarrow (E) \Rightarrow (E*E) \Rightarrow (id*E) \Rightarrow (id*id)
\]

“Sentential Forms”

A sequence of terminals and nonterminals in a derivation

(id*E)
Derives in one step $\Rightarrow$

If $A \rightarrow \beta$ is a rule, then we can write

$$\alpha A \gamma \Rightarrow \alpha \beta \gamma$$

Any sentential form containing a nonterminal (call it $A$) … such that $A$ matches the nonterminal in some rule.

---

Derives in zero-or-more steps $\Rightarrow^*$

$$E \Rightarrow^* (id*id)$$

If $\alpha \Rightarrow^* \beta$ and $\beta \Rightarrow \gamma$, then $\alpha \Rightarrow^* \gamma$

---

Derives in one-or-more steps $\Rightarrow^+$
Leftmost Derivations

In a derivation... always expand the \textit{leftmost} nonterminal.

\begin{align*}
E & \Rightarrow E + E \\
& \Rightarrow (E) + E \\
& \Rightarrow (E*E) + E \\
& \Rightarrow (id*E) + E \\
& \Rightarrow (id*id) + E \\
& \Rightarrow (id*id) + id
\end{align*}

Let $\Rightarrow_{LM}$ denote a step in a leftmost derivation ($\Rightarrow_{LM}^*$ means zero-or-more steps)

At each step in a leftmost derivation, we have

$wA\gamma \Rightarrow_{LM}^{*} w\beta\gamma$ where $A \rightarrow \beta$ is a rule

(Recall that $w$ is a string of terminals.)

Each sentential form in a leftmost derivation is called a “left-sentential form.”

If $S \Rightarrow_{LM}^* \alpha$ then we say $\alpha$ is a “left-sentential form.”
Rightmost Derivations

In a derivation... always expand the rightmost nonterminal.

\[
\begin{align*}
E & \quad \Rightarrow \quad E+E \\
& \quad \Rightarrow \quad E+id \\
& \quad \Rightarrow \quad (E)+id \\
& \quad \Rightarrow \quad (E*E)+id \\
& \quad \Rightarrow \quad (E*id)+id \\
& \quad \Rightarrow \quad (id*id)+id
\end{align*}
\]

Let \( \Rightarrow_{RM} \) denote a step in a rightmost derivation (\( \Rightarrow_{RM}^* \) means zero-or-more steps)

At each step in a rightmost derivation, we have

\[
\alpha A w \Rightarrow_{RM} \alpha \beta w
\]

where \( A \rightarrow \beta \) is a rule

(Recall that \( w \) is a string of terminals.)

Each sentential form in a rightmost derivation is called a “right-sentential form.”

If \( S \Rightarrow_{RM}^* \alpha \) then we say \( \alpha \) is a “right-sentential form.”
Parse Trees

Two choices at each step in a derivation...
- Which non-terminal to expand
- Which rule to use in replacing it

Leftmost Derivation:
1. \( E \rightarrow E + E \)
2. \( E \rightarrow E \times E \)
3. \( E \rightarrow (E) \)
4. \( E \rightarrow -E \)
5. \( E \rightarrow ID \)

The parse tree remembers only this
Parse Trees

Two choices at each step in a derivation...
- Which non-terminal to expand
- Which rule to use in replacing it

The parse tree remembers only this

Rightmost Derivation:

```
E
⇒ E+E
⇒ E+id
⇒ (E)+id
⇒ (E*E)+id
⇒ (E*id)+id
⇒ (id*id)+id
```

1. \( E \rightarrow E + E \)
2. \( \rightarrow E * E \)
3. \( \rightarrow (E) \)
4. \( \rightarrow -E \)
5. \( \rightarrow ID \)
Given a leftmost derivation, we can build a parse tree.
Given a rightmost derivation, we can build a parse tree.

Leftmost Derivation of \((id*id) + id\)
Rightmost Derivation of \((id*id) + id\)

Same Parse Tree

Every parse tree corresponds to...
- A single, unique leftmost derivation
- A single, unique rightmost derivation

**Ambiguity:**
However, one input string may have several parse trees!!!
Therefore:
- Several leftmost derivations
- Several rightmost derivations
Ambiguous Grammars

**Leftmost Derivation #1**

1. $E$  
2. $E \rightarrow E + E$
3. $E \rightarrow E * E$
4. $E \rightarrow (E)$
5. $E \rightarrow \text{ID}$

Input: $\text{id+id*id}$

- $E$  
- $E \rightarrow E + E$
- $E \rightarrow \text{id+E}$
- $E \rightarrow \text{id+E*E}$
- $E \rightarrow \text{id+id*E}$
- $E \rightarrow \text{id+id*id}$

**Leftmost Derivation #2**

- $E$  
- $E \rightarrow E * E$
- $E \rightarrow E + E * E$
- $E \rightarrow \text{id+E*E}$
- $E \rightarrow \text{id+id*E}$
- $E \rightarrow \text{id+id*id}$