Chapter 4
Syntax Analysis

Syntax Analysis - Parsing

- **An overview of parsing :**
  - **Functions & Responsibilities**
- **Context Free Grammars**
  - **Concepts & Terminology**
- **Writing and Designing Grammars**
- **Resolving Grammar Problems / Difficulties**
- **Top-Down Parsing**
  - **Recursive Descent & Predictive LL**
- **Bottom-Up Parsing**
  - **LR & LALR**

An Overview of Parsing

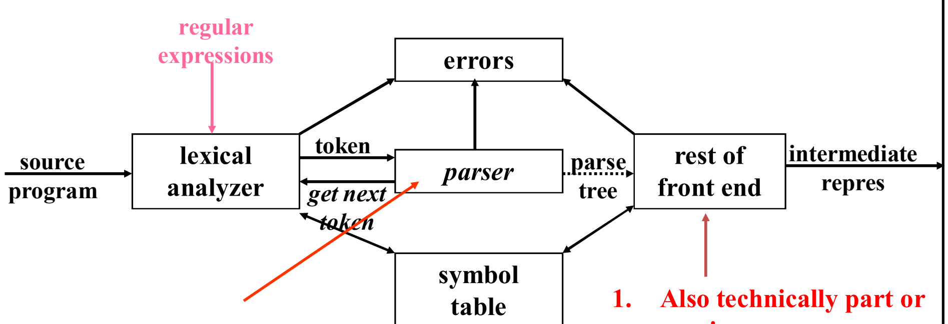
Why are Grammars to formally describe Languages Important ?

1. **Precise, easy-to-understand representations**
2. **Compiler-writing tools can take grammar and generate a compiler**
3. **Allow language to be evolved (new statements, changes to statements, etc.) Languages are not static, but are constantly upgraded to add new features or fix "old" ones**

ADA → ADA9x, C→ C++ , Templates, exceptions,

How do grammars relate to parsing process ?

Parsing During Compilation



- **uses a grammar to check structure of tokens**
- **produces a parse tree**
- **syntactic errors and recovery**
- **recognize correct syntax**
- **report errors**

1. **Also technically part or parsing**
2. **Includes augmenting info on tokens in source, type checking, semantic analysis**

Parsing Responsibilities

Syntax Error Identification / Handling

Recall typical error types:

- Lexical : Misspellings
- Syntactic : Omission, wrong order of tokens
- Semantic : Incompatible types
- Logical : Infinite loop / recursive call

Majority of error processing occurs during syntax analysis

NOTE: Not all errors are identifiable !!

Key Issues – Error Processing

1. Detecting errors
2. Finding position at which they occur
3. Clear / accurate presentation
4. Recover (pass over) to continue and find later errors
5. Don’t impact compilation of “correct” programs

Error Recovery Strategies

Panic Mode – Discard tokens until a "synchronous" token is found (end, ";", "}", etc.)
-- Decision of designer
-- Problems:
skip input ⇒miss declaration – causing more errors
⇒miss errors in skipped material
-- Advantages:
simple ⇒suited to 1 error per statement

Phrase Level – Local correction on input
-- "," ⇒";" – Delete "," – insert ";"
-- Also decision of designer
-- Not suited to all situations
-- Used in conjunction with panic mode to allow less input to be skipped

What are some Typical Errors ?

```
#include<stdio.h>
int f1(int v)
{     int i,j=0;
      for (i=1;i<5;i++)
      {  j=v+f2(i) }
      return j; }
int f2(int u)
{     int j;
      j=u+f1(u*u);
      return j; }
int main()
{     int i,j=0;
            for (i=1;i<10;i++)
            {     j=j+i*i  printf("%d\n",i);        }
      printf("%d\n",f1(j));
      return 0;
}
```

As reported by C Compiler

'f2' undefined;
syntax error : missing ';' before '}'
syntax error : missing ';' before identifier 'printf'

Which are "easy" to recover from?
Which are "hard" ?

Motivating Grammars

- Regular Expressions
  - → Basis of lexical analysis
  - → Represent regular languages
- Context Free Grammars
  - → Basis of parsing
  - → Represent language constructs

Context Free Grammars : Concepts & Terminology

Definition: A Context Free Grammar, CFG, is described by T, NT, S, PR, where:

T: Terminals / tokens of the language

NT: Non-terminals to denote sets of strings generated by the grammar & in the language

S: Start symbol, $S \in NT$, which defines all strings of the language

PR: Production rules to indicate how T and NT are combined to generate valid strings of the language.

$$PR: NT \rightarrow (T \mid NT)^*$$

Like a Regular Expression / DFA / NFA, a Context Free Grammar is a mathematical model

How does this relate to Languages?

$E \rightarrow E\ A\ E \mid (\ E\ ) \mid -E \mid id$

$A \rightarrow + \mid - \mid * \mid / \mid \uparrow$

Let G be a CFG with start symbol S. Then $S \overset{+}{\Rightarrow} W$ (where W has no non-terminals) represents the language generated by G, denoted L(G). So $W \in L(G) \Leftrightarrow S \overset{+}{\Rightarrow} W$.

W : is a sentence of G

When $S \Rightarrow \alpha$ (and α may have NTs) it is called a sentential form of G.

EXAMPLE: id * id is a sentence

Here's the derivation:

$E \Rightarrow E\ A\ E \Rightarrow E * E \Rightarrow id * E \Rightarrow id * id$

Sentential forms

$E \overset{*}{\Rightarrow} id * id$

Other Derivation Concepts

Leftmost: Replace the leftmost non-terminal symbol

$E \underset{lm}{\Rightarrow} E\ A\ E \underset{lm}{\Rightarrow} id\ A\ E \underset{lm}{\Rightarrow} id * E \underset{lm}{\Rightarrow} id * id$

Rightmost: Replace the right most non-terminal symbol

$E \underset{rm}{\Rightarrow} E\ A\ E \underset{rm}{\Rightarrow} E\ A\ id \underset{rm}{\Rightarrow} E * id \underset{rm}{\Rightarrow} id * id$

Important Notes: $A \rightarrow \delta$

If $\beta A \gamma \underset{lm}{\Rightarrow} \beta\, \delta\, \gamma$, what's true about β ?

If $\beta A \gamma \underset{rm}{\Rightarrow} \beta\, \delta\, \gamma$, what's true about γ ?

Derivations: Actions to parse input can be represented pictorially in a parse tree.

Examples of LM / RM Derivations

$E \rightarrow E + E \mid E*E \mid (E) \mid -E \mid id$

A leftmost derivation of : id + id * id

A rightmost derivation of : id + id * id

Parse Tree & Derivation

$E \Rightarrow E + E$

$\Rightarrow id + E$

$\Rightarrow id + E * E$

$\Rightarrow id + id * E$

$\Rightarrow id + id * id$