UNIVERSITY OF ENGINEERING & MANAGEMENT, KOLKATA

Course Name : Compiler Design

Prof. Sankhadeep Chatterjee

Prerequisite

- Formal language & Automata Theory
- Data Structure & Algorithms

References

- “Compilers: Principles Techniques and Tool” by Aho, Lam, Sethi, Ullman
- “Compilers : Principles and practice” by Dave and Dave
- “Programming Language Pragmatics” by M. L. Scott
- “Modern Compiler Implementation in C” by Andrew W. Appel
- “Compiler Design” by Manoj B Chandak
- “Compiler Design” by Santanu Chattopadhyay

Syllabus

Introduction to Compiling

Compilers, Analysis-synthesis model, The phases of the compiler, Cousins of the compiler.

Lexical Analysis

The role of the lexical analyzer, Tokens, Patterns, Lexemes, Input buffering, Specifications of a token, Recognition of tokens, Finite automata, From a regular expression to an NFA, From a regular expression to NFA, From a regular expression to DFA, Design of a lexical analyzer generator (Lex).

Syntax Analysis

The role of a parser, Context free grammars, Writing a grammar, Top down Parsing, Nonrecursive Predictive parsing (LL), Bottom up parsing, Handles, Viable prefixes, Operator precedence parsing, LR parsers (SLR, LALR), Parser generators (YACC). Error Recovery strategies for different parsing techniques.

Syllabus

Type checking

Type systems, Specification of a simple type checker, Equivalence of type expressions, Type conversions

Run time environments

Source language issues (Activation trees, Control stack, scope of declaration, Binding of names), Storage organization (Subdivision of run-time memory, Activation records), Storage allocation strategies, Parameter passing (call by value, call by reference, copy restore, call by name), Symbol tables, dynamic storage allocation techniques.

Intermediate code generation

Intermediate languages, Graphical representation, Three-address code, Implementation of three address statements (Quadruples, Triples, Indirect triples).

Code optimization

Introduction, Basic blocks & flow graphs, Transformation of basic blocks, Dag representation of basic blocks, The principle sources of optimization, Loops in flow graph, Peephole optimization.

Code generations

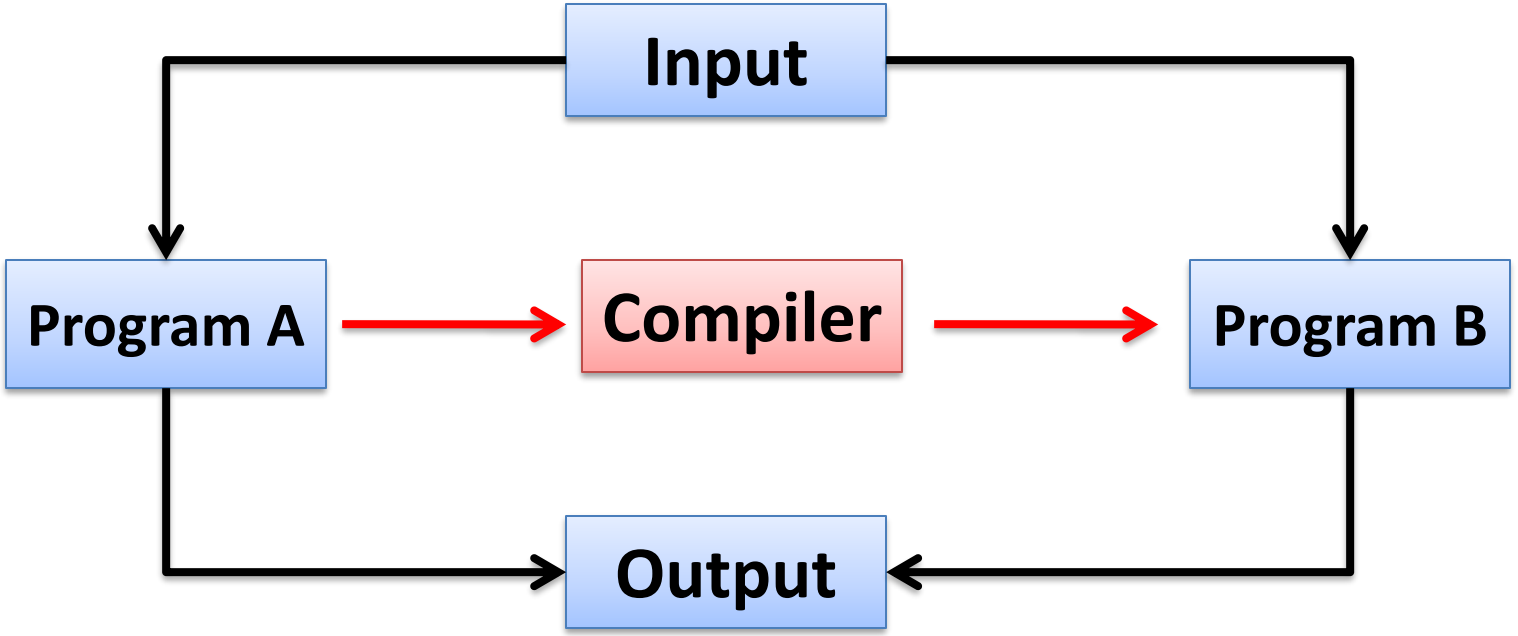
Issues in the design of code generator, a simple code generator, Register allocation & assignment.

What is a Compiler?

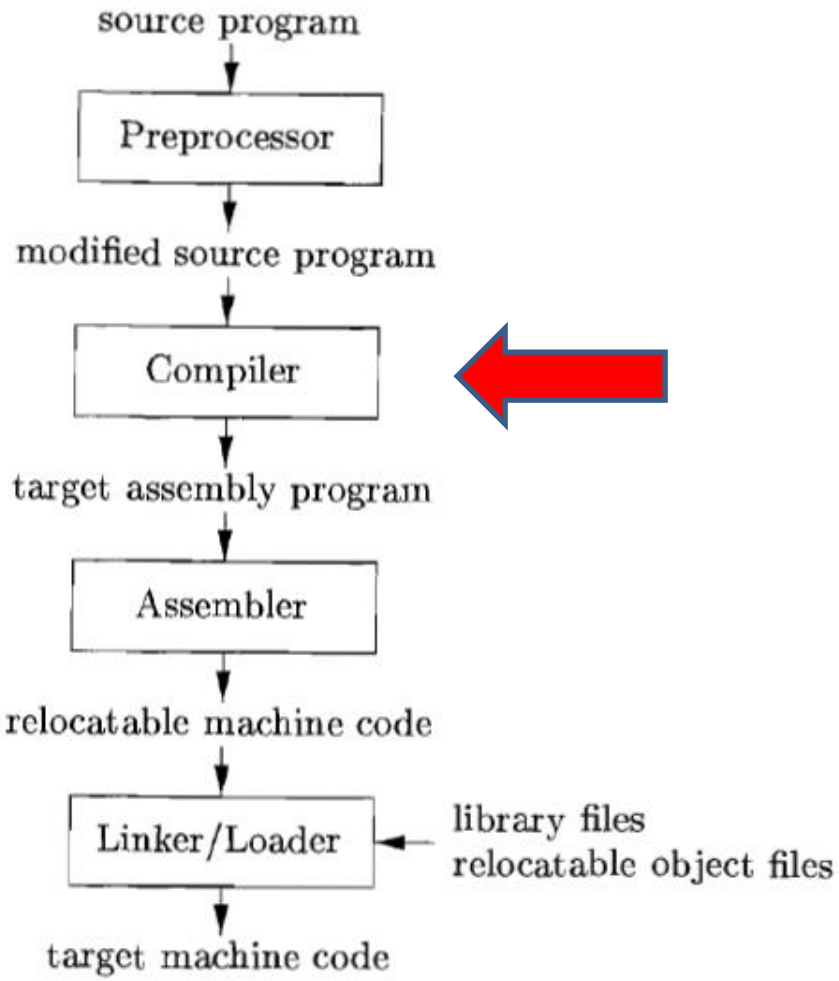
- A compiler is a program that can read a program in one language (the *source language*) *and* ***translate*** it into an equivalent program in another language (the *target language*)

What is a Compiler?

- By **equivalent program** we mean that for **same set of inputs** the programs should produce **same output**.



Language Processing System



One-pass compilers

- The compiler completes all its processing while scanning the source code only once.

- It has advantage of simpler and faster compiler, but it cannot do some of the sophisticated optimization.

Multi-pass compilers

- The compiler scans the source code several times to complete the translation. This allows for much better optimization.

- Example:
  - `DO 11 I = 1, 10 (Do -loop in FORTRAN)`
  - `DO 11 I = 1. 10 (Identifier 'DO11I')`

- Until we read after '`I`' compiler can't determine these two different cases

Compilation Phases

- The compilation (translating code) is not done at once.
- Instead, several small phases are involved.
- Each of these phases gradually converts the code to target language.

Initial Steps

- The first few steps can be understood by analogies to how humans comprehend a natural language
- The first step is recognizing/knowing alphabets of a language. For example
  - English text consists of lower and uppercase alphabets, digits, punctuations and white spaces
  - Written programs consist of characters from the ASCII characters set (normally 9-13, 32-126)

Initial Steps

- The next step to understand the sentence is recognizing words
  - How to recognize English words?
  - Words found in standard dictionaries

Initial Steps

- How to recognize words in a programming language?
  - a dictionary (of keywords etc.)
  - rules for constructing words (identifiers,numbers etc.)
- This is called lexical analysis

Initial Steps

- Recognizing words is not completely trivial. For example:

ify ouc an re adth is, yo uar eagen ius.

Initial Steps

- Recognizing words is not completely trivial. For example:

ify ouc an re adth is, yo uar eagen ius.

ify ouc an re adth is, yo uar eagen ius.

Lexical Analysis: Challenges

- We must know what the word separators are
- The language must define rules for breaking a sentence into a sequence of words.
- Normally white spaces and punctuations are word separators in languages.

Lexical Analyzer

- *The lexical* analyzer reads the stream of characters making up the source program and groups the characters into meaningful sequences called ***lexemes***. *For each* lexeme, the lexical analyzer produces as output a *token of the form;*

<token_name, attribute_value>

Lexical Analysis contd.

Consider the following ‘C’ language statement;

```
a = b + c ;
```

```
< a > < = > < b > < + > < c > < ; >
```

```
a -> identifier
= -> Assignment operator
b -> identifier
+ -> operator
c -> identifier
; -> Terminating symbol
```

Token Stream Representation

- Input character stream : **`a = b + c ;`**
- Lexemes : `< a > < = > < b > < + > < c > < ; >`
- Each lexeme is mapped to a token.
- For example;
  - Lexeme **<a>** is mapped to **<id,1>**
  - Lexeme <=> is mapped to <=> and so on.

Token Stream contd.

- Input character stream : **a = b + c ;**
- Lexemes : **< a > < = > < b > < + > < c > < ; >**
- Token Stream :

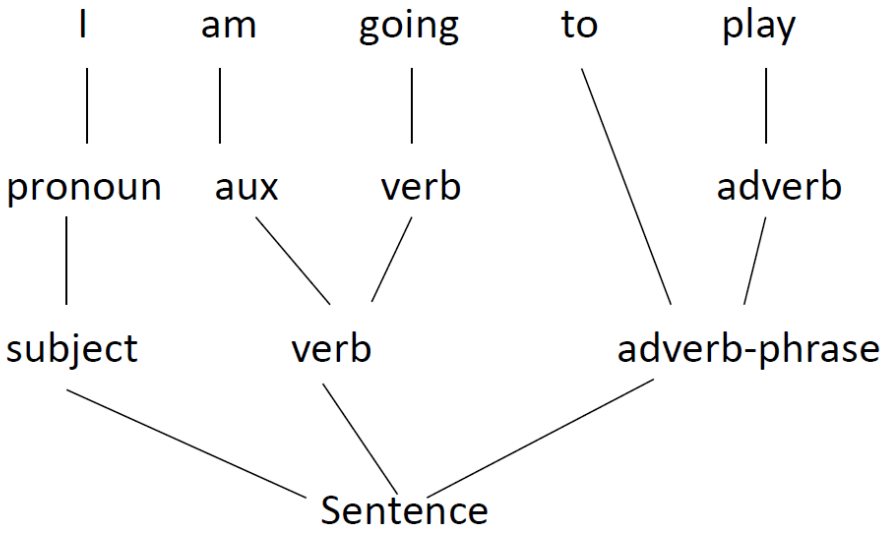
<id, 1> <=> <id, 2> <+> <id, 3> <;>

1	a	…
2	b	…
3	c	…
	…	…

Symbol Table

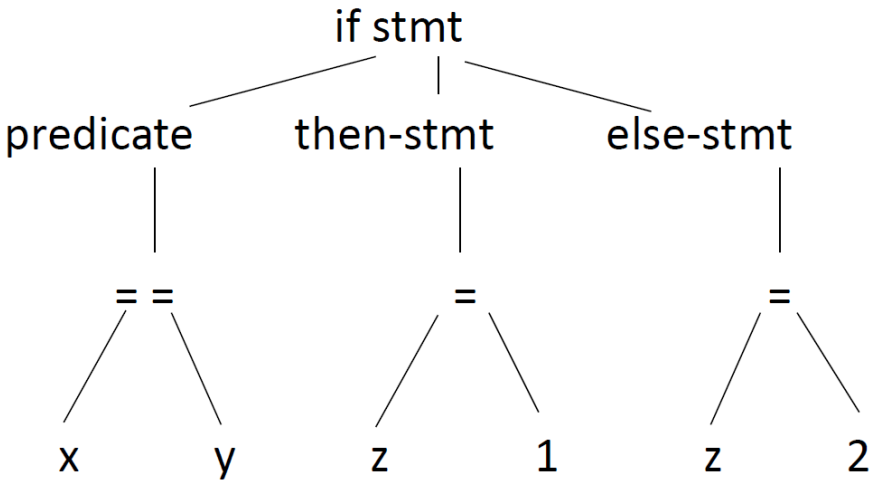
Next step

- Once the words are understood, the next step is to understand the structure of the sentence
- The process is known as *syntax checking or parsing*



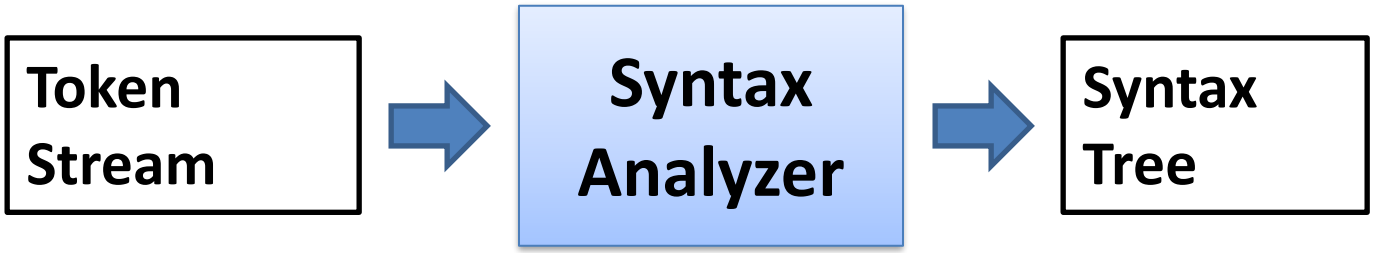
Syntax Analysis

- Parsing a program is exactly the same process as shown in previous slide.
- Consider an expression
  - if x == y then z = 1 else z = 2



Syntax Analyzer

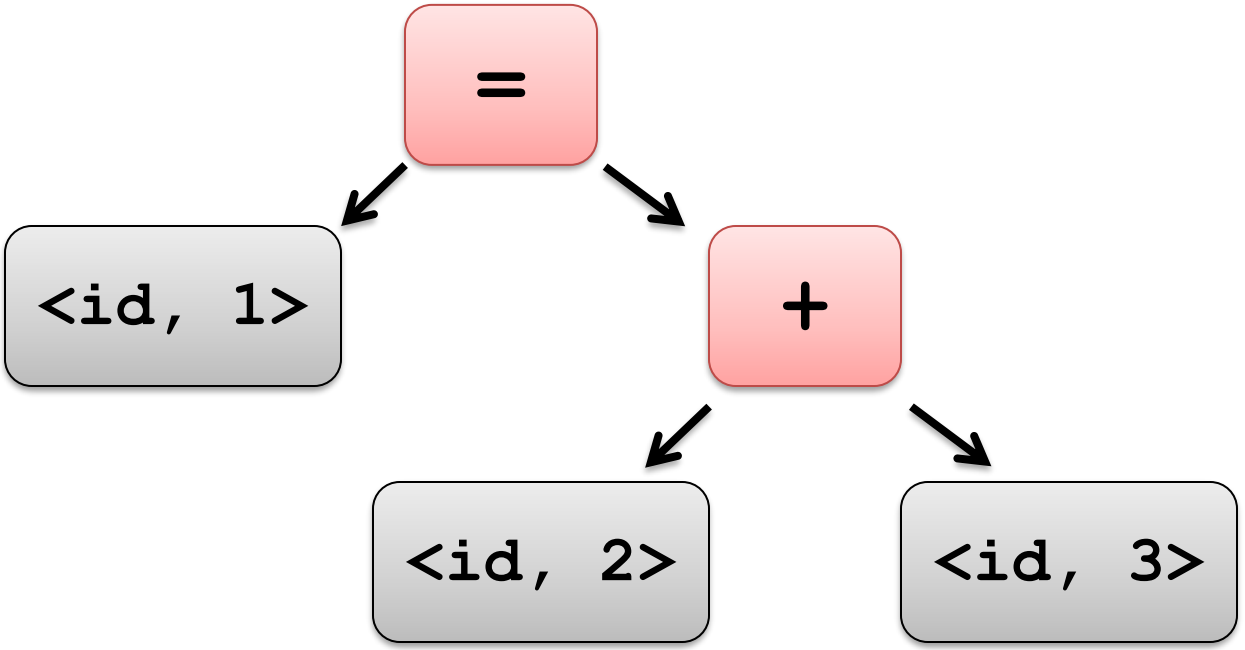
- The second phase of the compiler is *syntax analysis* or *parsing.*
- The parser uses the tokens produced by the lexical analyzer to create a tree-like intermediate representation that depicts the grammatical structure of the token stream.



Syntax Tree

- Input character stream : **a = b + c ;**
- Token Stream :

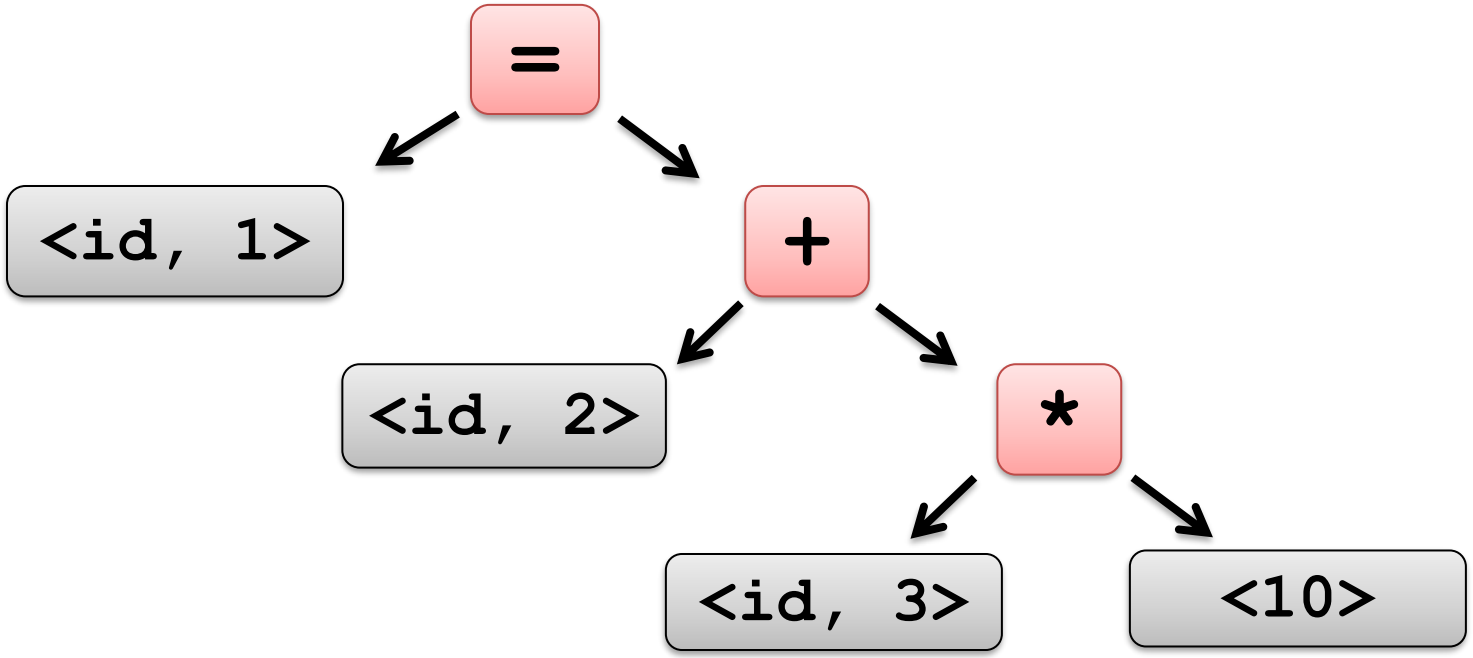
<id, 1> <=> <id, 2> <+> <id, 3> <;>



Syntax Tree contd.

- Input character stream : `a = b + c * 10`
- Token Stream :

`<id, 1> <=> <id, 2> <+> <id, 3> <*> <10>`



Semantic Analysis

- Once the sentence structure is understood we try to understand the meaning of the sentence (semantic analysis)

- A challenging task
  - Ravi said Ajay that he got his job offer from Google
  - Whom does ‘his’ referring to?
  - Who is the Lucky person!

Semantic Analysis

- Worse case:
  - Rajat said Rajat left his phone at home
- Too hard for compilers. They do not have capabilities similar to human understanding
- However, compilers do perform analysis to understand the meaning and catch inconsistencies

Semantic Analysis

- Programming languages define strict rules to avoid such ambiguities

```
{
    int x = 3;
    {
        int x = 4;
        cout << x;
    }
}
```

Front End Compilation

- Front End Compilation
- Machine Independent