Compiler Design - Introduction

Sitara K.

sitara@nitt.edu

Syllabus

UNIT I Introduction to Compilation

Compilers - Analysis of the source program - Phases of a compiler - Cousins of the Compiler - Grouping of Phases - Compiler construction tools - Lexical Analysis - Role of Lexical Analyzer - Input Buffering - Specification of Tokens.

Lab Component: Tutorial on LEX / FLEX tool, Tokenization exercises using LEX.

UNIT II Syntax Analysis

Role of the parser - Writing Grammars - Context-Free Grammars - Top Down parsing - Recursive Descent Parsing - Predictive Parsing - Bottom-up parsing - Shift Reduce Parsing - Operator Precedent Parsing - LR Parsers - SLR Parser - Canonical LR Parser - LALR Parser.

Lab Component: Tutorial on YACC tool, Parsing exercises using YACC tool.

UNIT III Intermediate Code Generation

Intermediate languages - Declarations - Assignment Statements - Boolean Expressions - Case Statements - Back patching - Procedure calls.

Lab Component: A sample language like C-lite is to be chosen. Intermediate code generation exercises for assignment statements, loops, conditional statements using LEX/YACC.

UNIT IV Code Optimization and Run Time Environments

Introduction - Principal Sources of Optimization - Optimization of basic Blocks - DAG representation of Basic Blocks - Introduction to Global Data Flow Analysis - Runtime Environments - Source Language issues - Storage Organization - Storage Allocation strategies - Access to non-local names - Parameter Passing - Error detection and recovery.

Lab Component: Local optimization to be implemented using LEX/YACC for the sample language.

UNIT V Code Generation

Issues in the design of code generator - The target machine - Runtime Storage management - Basic Blocks and Flow Graphs - Next-use Information - A simple Code generator - DAG based code generation - Peephole Optimization.

Lab Component: DAG construction, Simple Code Generator implementation, DAG based code generation using LEX/YACC for the sample language.

Objectives

- To introduce the major concept areas in compiler design and know the various phases of the compiler
- To understand the various parsing algorithms and comparison of the same
- To provide practical programming skills necessary for designing a compiler
- To gain knowledge about the various code generation principles
- To understand the necessity for code optimization

Course Outcomes

- Apply the knowledge of LEX & YACC tool to develop a scanner and parser
- Design and develop software system for backend of the compiler
- Suggest the necessity for appropriate code optimization techniques
- Conclude the appropriate code generator algorithm for a given source language
- Design a compiler for any programming language

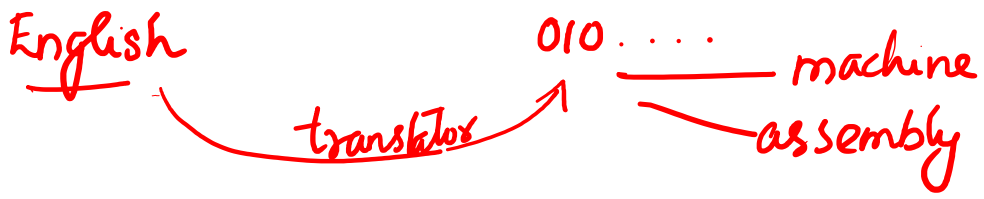
Text Books

- Alfred V. Aho, Jeffrey D Ullman, “Compilers: Principles, Techniques and Tools”, Pearson Education Asia, 2012.
- Jean Paul Tremblay, Paul G Serenson, “The Theory and Practice of Compiler Writing”, BS Publications, 2005.
- Dhamdhere, D. M., “Compiler Construction Principles and Practice”, Second Edition, Macmillan India Ltd., New Delhi, 2008.

History

8085 – A, B, C, D

- Software – essential component of the current scenario.
- Early software was written in assembly languages
- Drawbacks
  - Very difficult to remember instructions
- The benefits of reusing software on different CPUs became greater than the cost of designing compiler
- Very cumbersome to write
- Need for a software that will understand human language

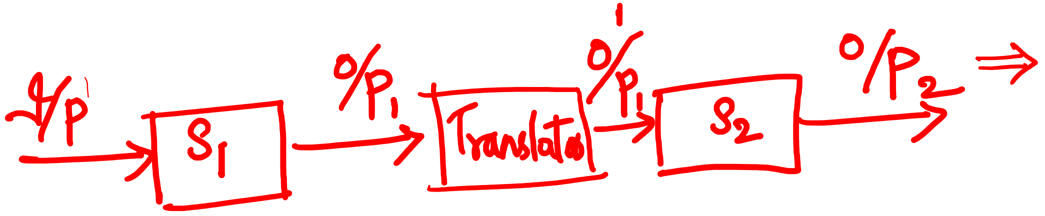


Language Processors.

- A **translator** inputs and then converts a **source program** into an **object or target** program.

- **Source program** is written in one language
- **Object program** belongs to an object language

- A translators could be: **Assembler, Compiler, Interpreter**



Options

- Design an interpreter / translator to convert human language to machine language
  - Difficult – Parsing, interpreting, ambiguous

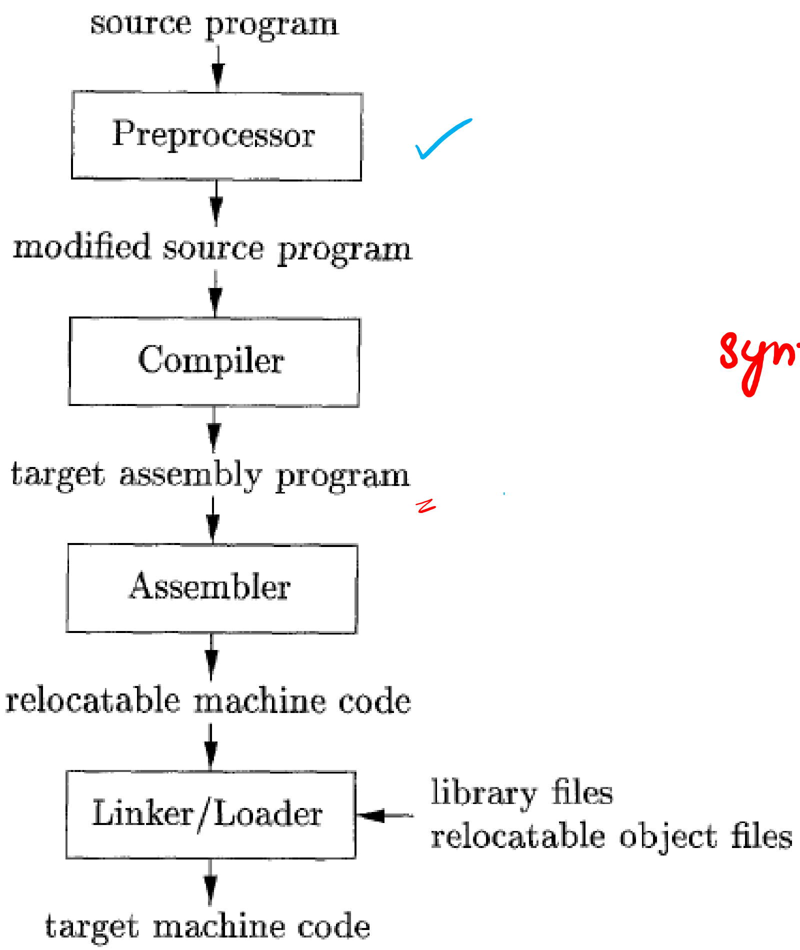
Options

- Design a compiler that will understand high level (not necessary English) language to assembly language
  - Relatively simpler, but need a mapping of the high level language to assembly language

Options

- Design an assembler that converts assembly language to machine language
    - Target language need to be specified. Output of the various compilers to be known prior time

Language Processing System



#define MAX 100

for (

float

int a[10]; i=0;

a[i] = 5;

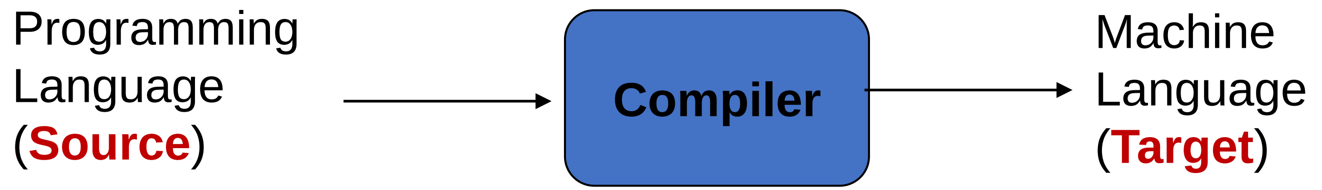
syntax / semantic

Compiler

- The first real compiler
  - FORTRAN compilers of the late 1950s
  - 18 person-years to build

What are Compilers?

- A compiler acts as a translator,
  transforming human-oriented programming languages
  into computer-oriented machine languages.
- No concern about machine-dependent details for programmer



Compiler

- Processes source program
- Prompts errors in source program
- Recovers / Corrects the errors
- Produce assembly language program
- Compiler + assembler – Converts this to relocatable machine code

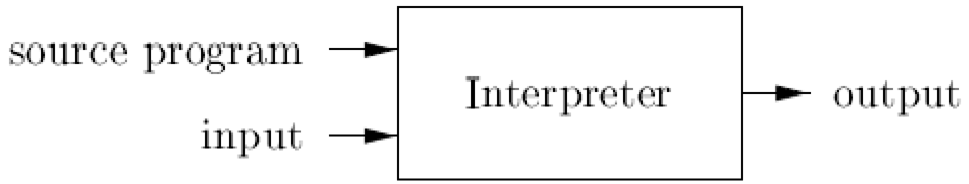
```
f1( )
{
      p;
p= f2( )
}

f2( )
{
      a;
  fa
}
```

Compiler - Overview

- Translates a source program written in a High-Level Language (HLL) such as Pascal, C++ into computer's machine language (Low-Level Language (LLL))
- The time of conversion from source program into object program is called **compile time**
- The object program is executed at **run time**

Interpreter

- Language processor that executes the operation as specified in the source program
- Inputs are supplied by the user
- Processes an internal form of the source program and data at the same time (at run time); no object program is generated.

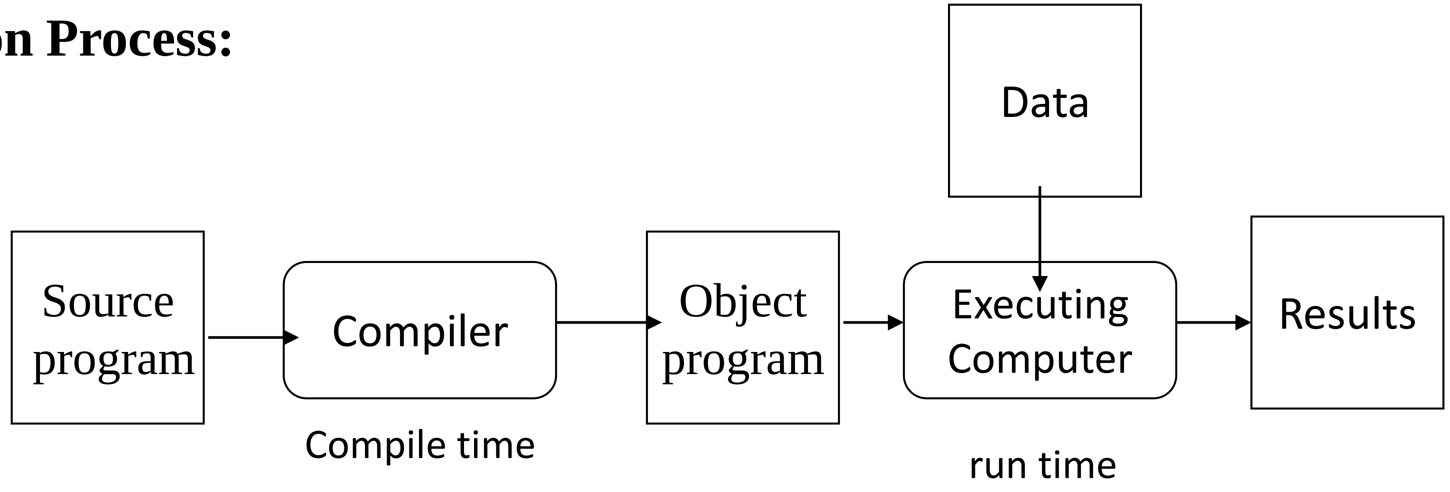


Compiler vs Interpreter

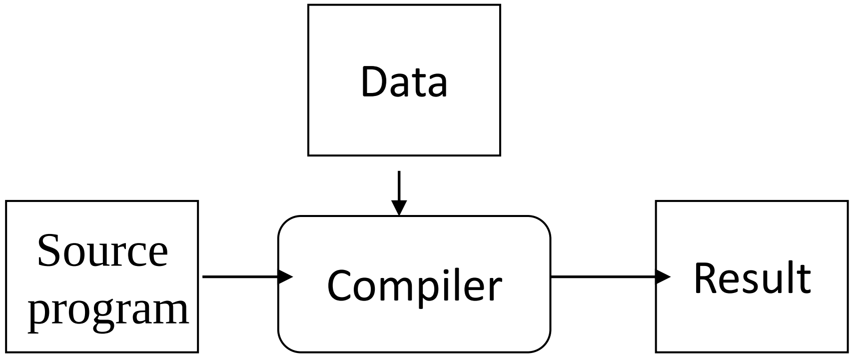
- The machine-language target program produced by a compiler is much faster than an interpreter at mapping inputs to outputs
- An interpreter, is better with error diagnostics as it executes the source program statement by statement

Overview of Compilers

Compilation Process:



Interpretive Process:



Compiler

- analysis of source program: The analysis part breaks up the source program into constituent pieces and imposes a grammatical structure on them. It then uses this structure to create an intermediate representation of the source program.

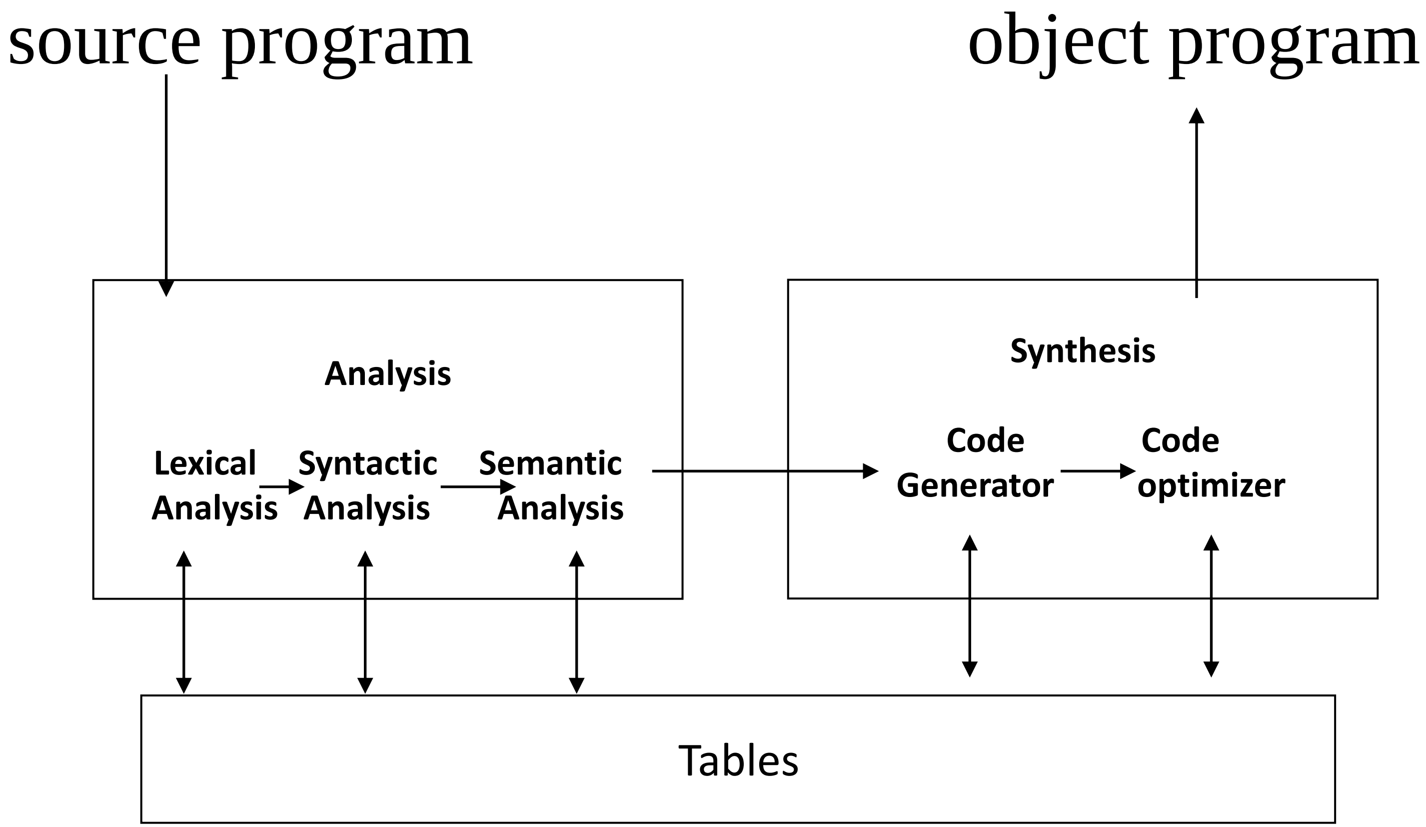
Compiler

- synthesis of its corresponding program: constructs the desired target program from the intermediate representation and the information in the symbol table.
- The ***analysis part*** is often called the ***front end*** of the compiler; the ***synthesis part*** is the ***back end***.

Compiler

- Front End – Language Dependent – Depends on the source language and Target Independent
- Back End – Target Dependent – Depends on the target language but Source independent

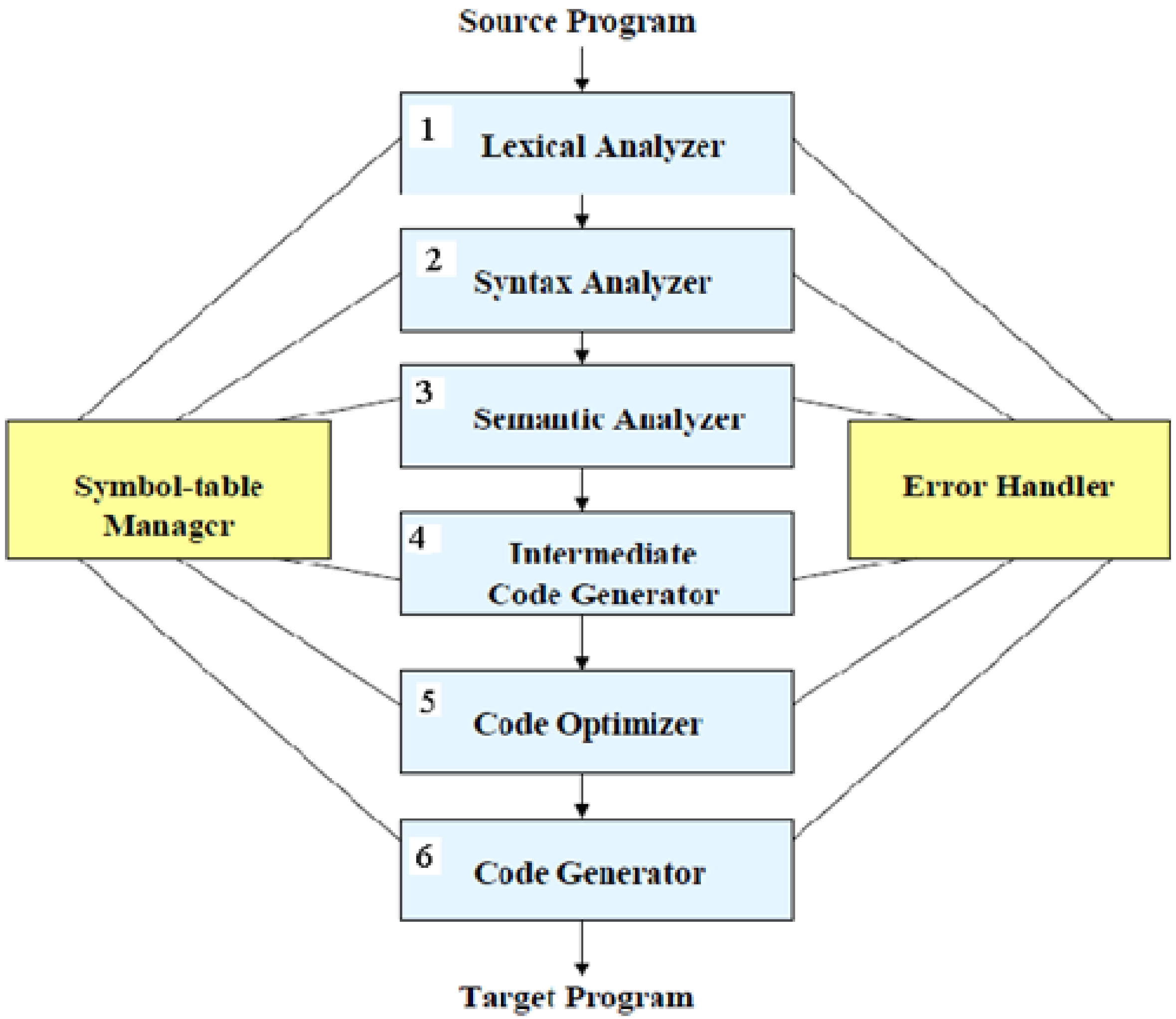
Flow of Compiler



Compiler Passes

- How many passes should the compiler go through?
- One for analysis and one for synthesis?
- One for each division of the analysis and synthesis?
- The work done by a compiler is grouped into phases

Phases of the compiler



Lexical Analysis (scanner): The first phase of a compiler

C

identifiers.

✓ area

✓ area_1

✗ 1area

- Lexical analyzer reads the stream of characters from the source program and combines the characters into meaningful sequences called ***lexeme***
- For every lexeme, the lexer produces a token of the form which is passed to the next phase of the compiler

(token-name, attribute-value) ✓

Lexical Analysis (scanner): The first phase of a compiler

- Token-name: an abstract symbol is used during syntax analysis, an attribute-value: points to an entry in the symbol table for this token.
- Blanks will be discarded by the lexical analyser

```
position = initial + rate * 60
```

↓

Lexical Analyzer

↓

$\langle \mathbf{id}, 1\rangle \ \langle = \rangle \ \langle \mathbf{id}, 2\rangle \ \langle + \rangle \ \langle \mathbf{id}, 3\rangle \ \langle * \rangle \ \langle 60 \rangle$

<number, 4>

<number>

↓

Syntax Analyzer

SYMBOL TABLE

1	position	...
2	initial	...
3	rate	...
4	number	60

<, <=, >, >=

relop.

<relop>

<relop, G>

<relop, GE>

a=b+c;

a = b + c;

Example: position =initial + rate * 60

1. ”position” is a lexeme mapped into a token (id, 1), where id is an abstract symbol standing for identifier and 1 points to the symbol table entry for position. The symbol-table entry for an identifier holds information about the identifier, such as its name and type.
2. = is a lexeme that is mapped into the token (=). Since this token needs no attribute-value, we have omitted the second component. For notational convenience, the lexeme itself is used as the name of the abstract symbol.
3. “initial” is a lexeme that is mapped into the token (id, 2), where 2 points to the symbol-table entry for initial

4. + is a lexeme that is mapped into the token (+).
5. “rate” is a lexeme mapped into the token (id, 3), where **3** points to the symbol-table entry for rate.
6. * is a lexeme that is mapped into the token (*) .
7. 60 is a lexeme that is mapped into the token (60)

Lexical Analysis

- Interface of the compiler to the outside world
- Scans input program, identifies valid words of the language in it
- Removes extra white spaces, comments etc
- Expand user defined macros
- Reports presence of foreign words
- May perform case conversions
- Generates tokens
- Generally implemented as finite automata

Area

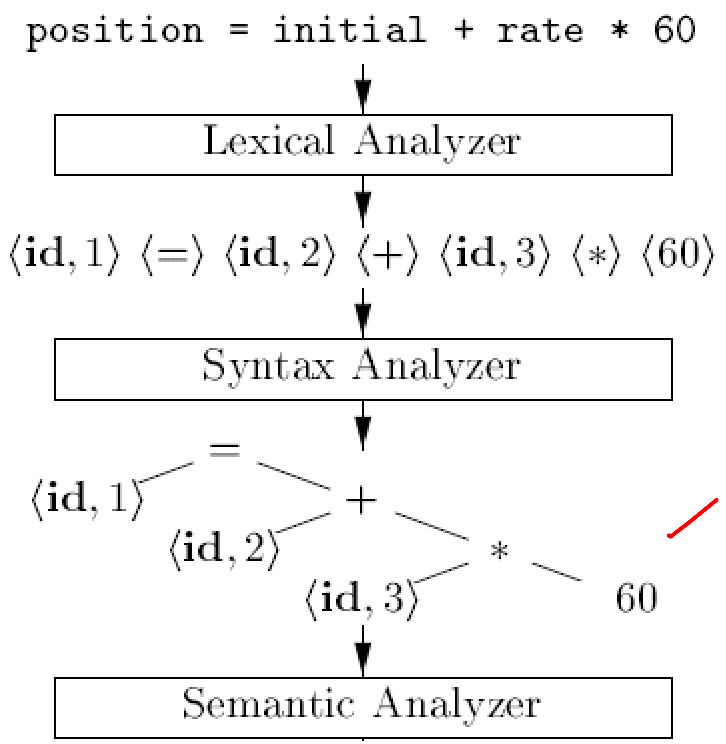
area

Syntax Analysis (parser) : The second phase of the compiler

- The parser uses the tokens produced by the lexer to create a tree-like intermediate representation that verifies the grammatical structure of the sequence of tokens
- Works hand-in-hand with lexical analyzer
- Identifies sequence of grammar rules to derive the input program from the start symbol
- A parse tree is constructed
- Error messages are flashed for syntactically incorrect programs

Syntax Analysis (parser) : The second phase of the compiler

- A typical representation is a <u>syntax tree</u> in which each interior node represents an operation and the children of the node represent the arguments of the operation



Syntax phase

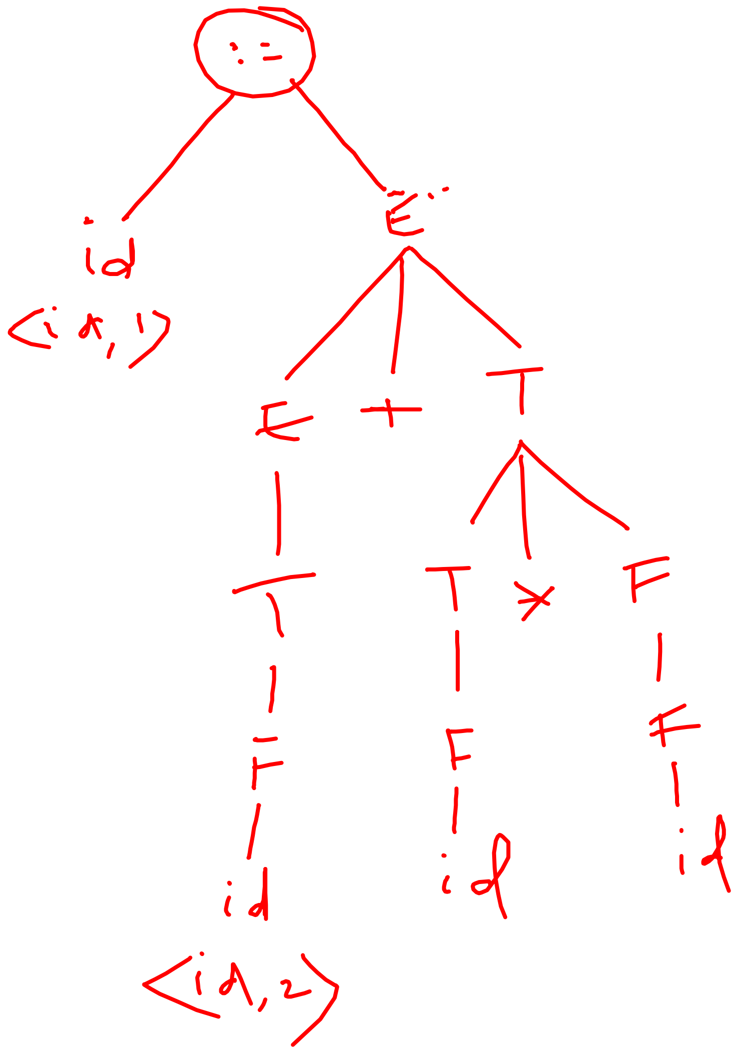
- Grammar for assignment expression

  S → id := E

  E → E + T | T

  T → T * F | F

  F → (E) | id



Semantic Analysis: Third phase of the compiler

- The semantic analyzer uses the output of the parser – syntax tree and the information in the symbol table to check for semantic consistency in the source program
- Gathers type information and saves it in either the syntax tree or the symbol table, for subsequent use during intermediate-code generation.

a % b

Semantic Analysis: Third phase of the compiler

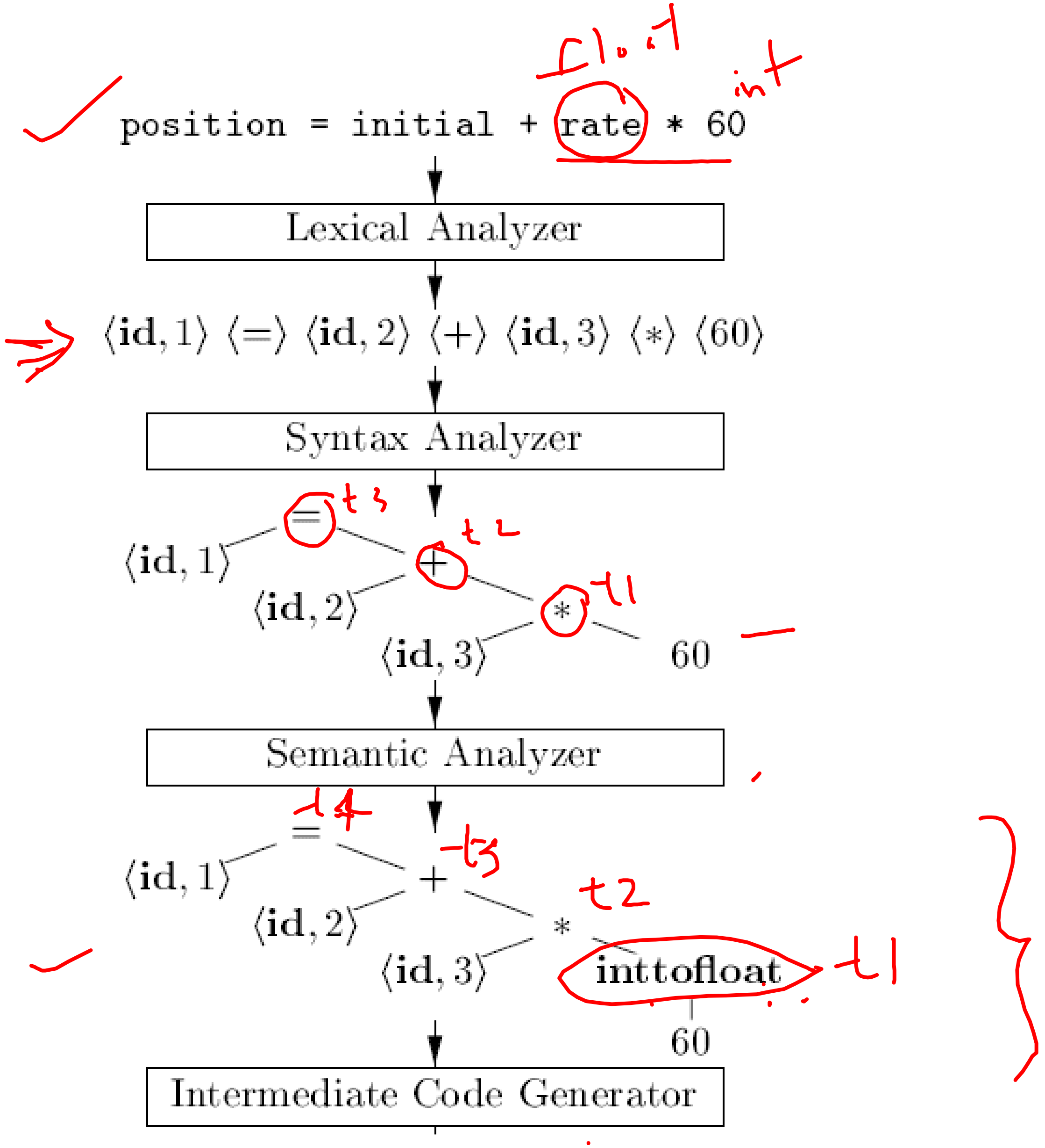
- Type checking, where the compiler checks that each operator has matching operands.
    - Array index need to be an integer; the compiler must identify an error if a floating-point number is used to as an array index
- Scope rules of the language are applied to determine types – static scope or dynamic scope

Semantic Analysis: Third phase of the compiler

- Coercions – a way of type conversion
- For example, a binary arithmetic operator may be applied to either a pair of integers or to a pair of floating-point numbers. If the operator is applied to a floating-point number and an integer, the compiler may convert or coerce the integer into a floating-point number.

float int float

a = b + c;



float
int
position = initial + rate * 60
Lexical Analyzer
⟨id, 1⟩ ⟨=⟩ ⟨id, 2⟩ ⟨+⟩ ⟨id, 3⟩ ⟨*⟩ ⟨60⟩
Syntax Analyzer
t3
t2
t1
⟨id, 1⟩
⟨id, 2⟩
⟨id, 3⟩
60
Semantic Analyzer
t4
t3
t2
t1
⟨id, 1⟩
⟨id, 2⟩
⟨id, 3⟩
inttofloat
60
Intermediate Code Generator

Intermediate Code Generation: Fourth phase of the compiler

- Optional towards target code generation
- Compilers generate an explicit low-level or machine-like intermediate representation (a program for an abstract machine).This intermediate representation:
  - should be easy to produce
  - should be easy to translate into the target machine
  - Powerful enough to express the programming language constructs
- Helps to retarget the code from one processor to another

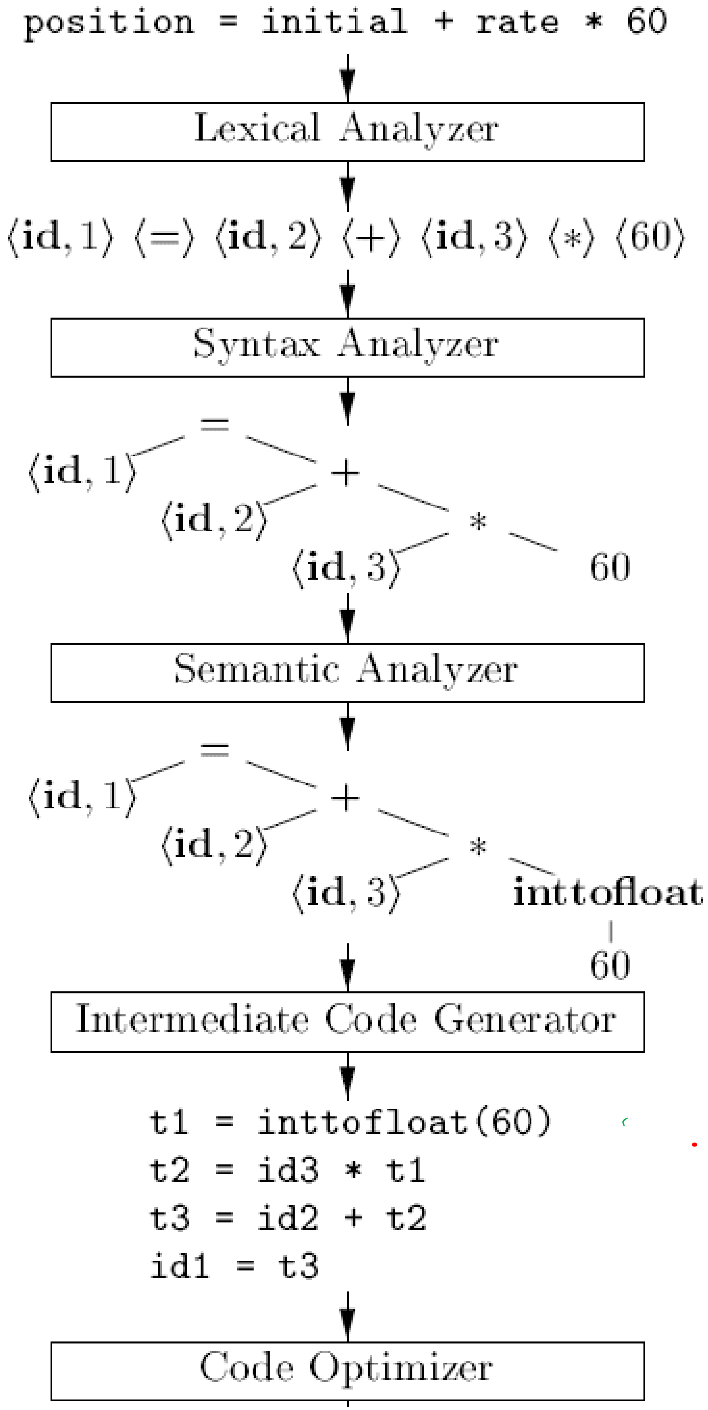
Intermediate code Generation : Three address code

- A convention for Intermediate code generation is three address code
- Three operands at the most and 2 operators
- Example:
  - x = y op z
  - x = op y

$x = -y$

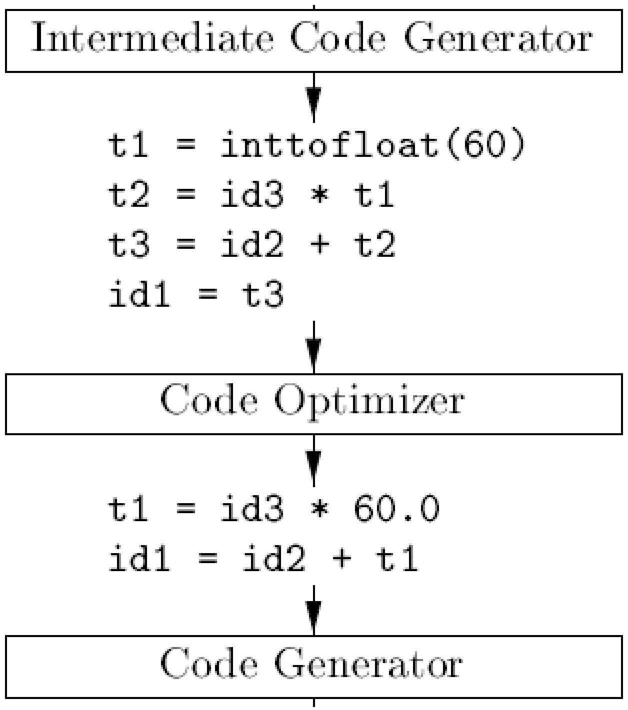
$x = ++y$

$y = y+1$


position = initial + rate * 60
Lexical Analyzer
⟨id, 1⟩ ⟨=⟩ ⟨id, 2⟩ ⟨+⟩ ⟨id, 3⟩ ⟨*⟩ ⟨60⟩
Syntax Analyzer
=
⟨id, 1⟩
+
⟨id, 2⟩
*
⟨id, 3⟩
60
Semantic Analyzer
=
⟨id, 1⟩
+
⟨id, 2⟩
*
⟨id, 3⟩
inttofloat
60
Intermediate Code Generator
t1 = inttofloat(60)
t2 = id3 * t1
t3 = id2 + t2
id1 = t3
Code Optimizer

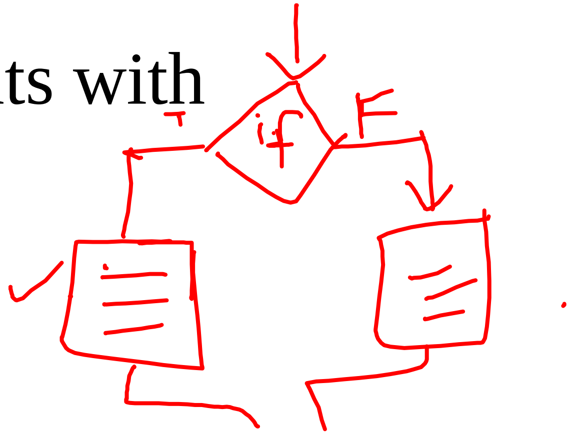
Code Optimization: - Fifth phase of the compiler

- attempts to improve the intermediate code for better target code
  - faster, shorter code, or target code that consumes less power.
- simple optimizations that significantly improve the running time of the target program without slowing down compilation



Code Optimization: - Fifth phase of the compiler

- Automated steps of compiler generate lots of redundant code that can possibly eliminated
- Code is divided into *basic blocks* – a sequence of statements with single entry and exit
- *Local optimizations* restrict within a single basic block
- *Global optimizations* spans across basic blocks
- Optimize loops, algebraic simplifications, elimination of load-and-store are common optimizations



for(10 times)

1 min ⇒ 1 sec

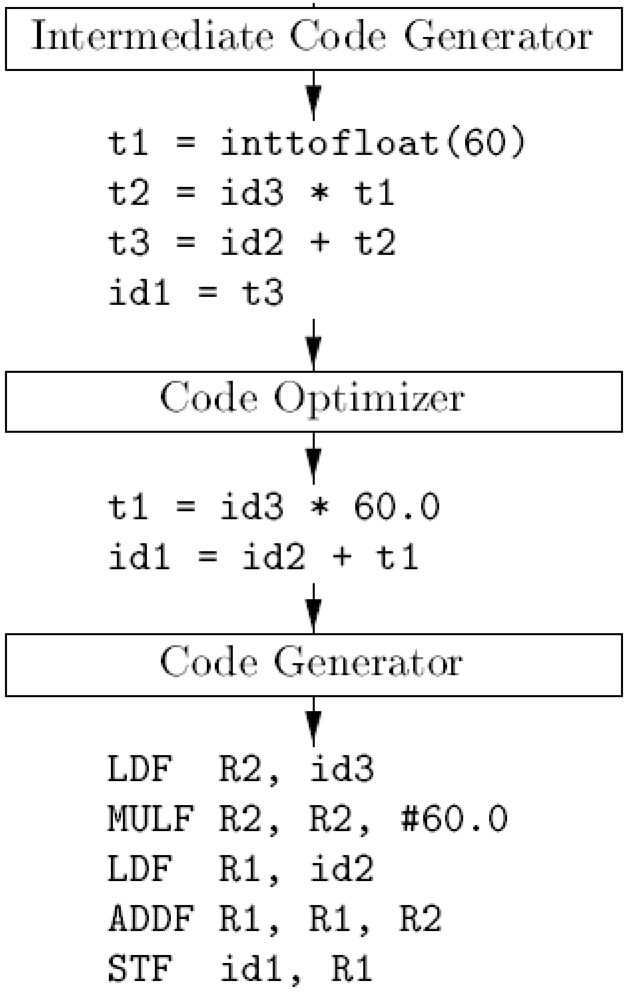
$a = b + d/10$

$c = a + d/10$

$t1 = d/10;$

Code Generation: Sixth phase of the compiler

- If the target language is machine code, then registers or memory locations are selected for each of the variables used by the program.
- Then, the intermediate instructions are translated into sequences of machine instructions to complete an operation



Code Generation: Sixth phase of the compiler

- Important consideration of code generation is the assignment of registers to hold variables.
- Choice of instructions involving registers, memory or a mix of the two

Symbol-Table Management: - Interaction with all the compiler's phases

- The symbol table is a data structure containing a record for each variable name (all symbols defined in the source program), with fields for the attributes of the name.
- The data structure is designed to help the compiler to identify and fetch the record for each name quickly
- To store or retrieve data from that record quickly
- Not part of the final code, but used as reference by all phases
- Generally created by lexical and syntax analyzer

Symbol-Table Management: - Interaction with all the compiler's phases

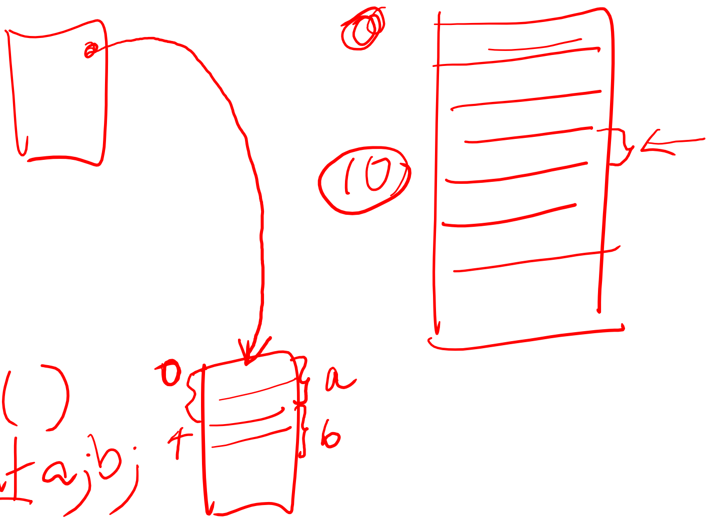
- attributes may provide information about the storage allocated for a name, its type, its scope, size, relative offset of variables
- Function or Procedure names, such things as the number and types of its arguments, the method of passing each argument and the return type

int x;

```
{ int x = 10;
  int a = 1;
  { int x = 5;
    printf(a, x);
  }
  printf(a, x);
}
```

```
fn1()
{
}
```

```
fn2()
{ int a,b;
}
```




```
fn1()
{ int a;
  int b;
  fun2();
}
```

```
fun2()
{
  a,b;
}
```

Error Handling and Recovery

- An important criteria for selecting the quality of the compiler
- For semantic errors, compiler can proceed
- For syntax errors, parser enters into erroneous state
- Needs to undo some processing already carried out by parser
- A few tokens may need to be discarded to reach a descent state
- Recovery is essential to provide a bunch of errors to the users

Compiler Phases vs Passes

- Several phases can be implemented as a single pass consist of reading an input file and writing an output file.

Compiler Phases vs Passes

- A typical multi-pass compiler looks like:
  - First pass: preprocessing, macro expansion
  - Second pass: syntax-directed translation, IR code generation
  - Third pass: optimization
  - Last pass: target machine code generation

Cousins of Compilers

- Preprocessors
- Assemblers
  - Compiler may produce assembly code instead of generating relocatable machine code directly.

Cousins of the Compiler

- Loaders and Linkers
  - Loader copies code and data into memory, allocates storage, setting protection bits, mapping virtual addresses, .. Etc
  - Linker handles relocation and resolves symbol references.
- Debugger