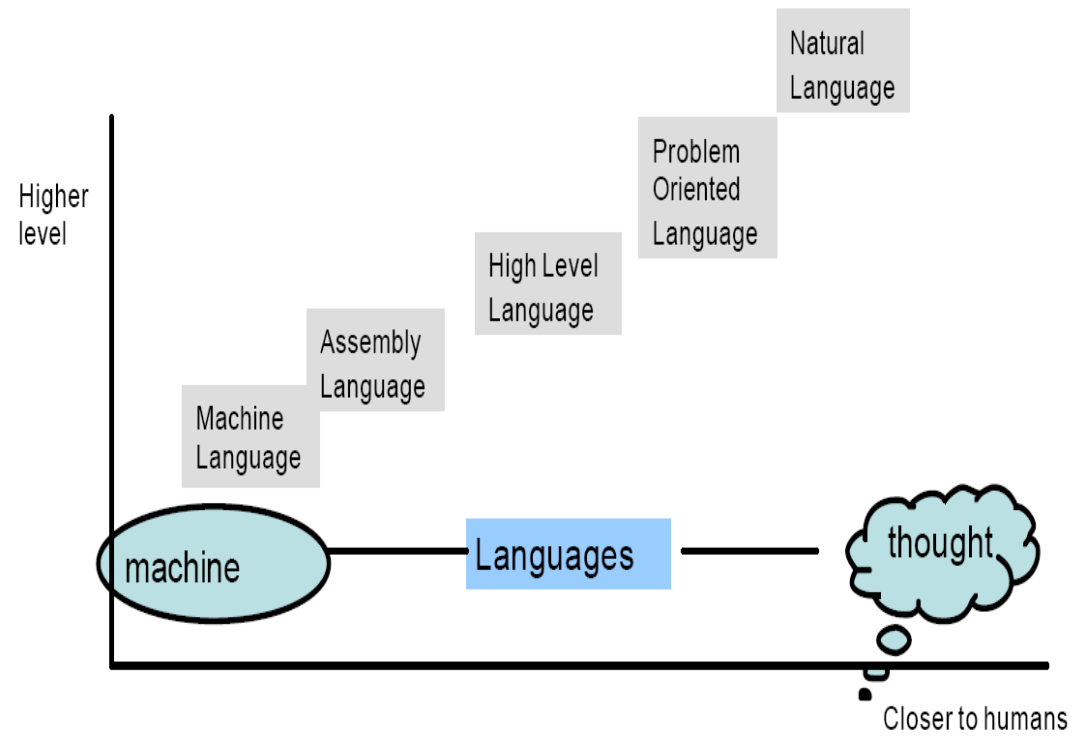
Compiler Design – Introduction

What is Language?

- **Language:** “any system of formalized symbols, signs, etc., used or conceived as a means of communication.”
  - Communicate: to transmit or exchange thought or knowledge.
  - Programming language: communicate between a person and a machine
  - Programming language is an intermediary

Hierarchy of (programming) languages

- Machine language
- Assembly language
- High level language
- Problem oriented
- Natural language

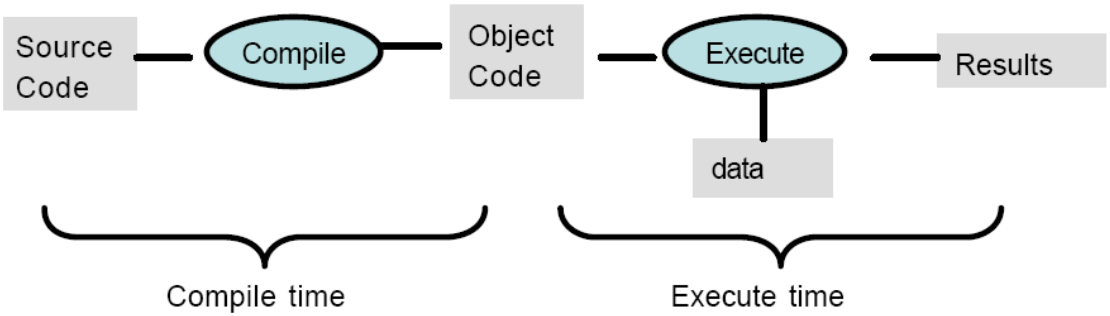


Language Translators

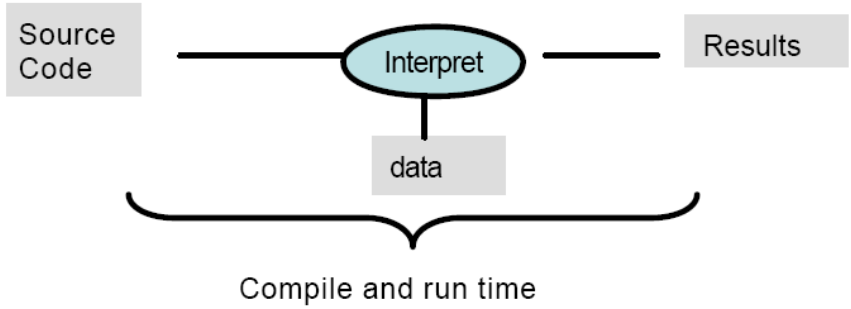
- **Translator:** Translate one language into another language (e.g., from C++ to Java)
  - A generic term.
- For high level programming languages (such as java, C):
  - **Compiler**: translate high level programming language code into host machine code
  - **Interpreter**: process the source program and data at the same time. No equivalent machine code is generated.
- **Assembler**: translate an assembly language to machine code.
- **Decompiler** and **Disassembler** refer to translators which attempt to take object code at a low level and regenerate source code at a higher level.
- **Cross-translators** generate code for machines other than the host machine.

Compiler Vs Interpreter

- Compiler



- Interpreter



Goals of translation

1. A better compiler is one which generates smaller code
2. Correct Code
3. Output runs fast
4. The compiler itself must run fast
5. compilation time must be proportional to program size.
6. Good Diagnostics for errors
7. Compiler must be portable

Some early machines and implementations

- **IBM developed 704 in 1954. All programming was done in assembly language. Cost of software** development far exceeded cost of hardware. Low productivity.
- Speedcoding interpreter: programs ran about 10 times slower than hand written assembly code
- John Backus (in 1954): Proposed a program that translated high level expressions into native machine code. Skeptism all around. Most people thought it was impossible
- Fortran I project . (1954-1957): The first compiler was released
  - The whole new field of compiler design was started
  - Modern compilers preserve the basic structure of the Fortran I compiler !!!

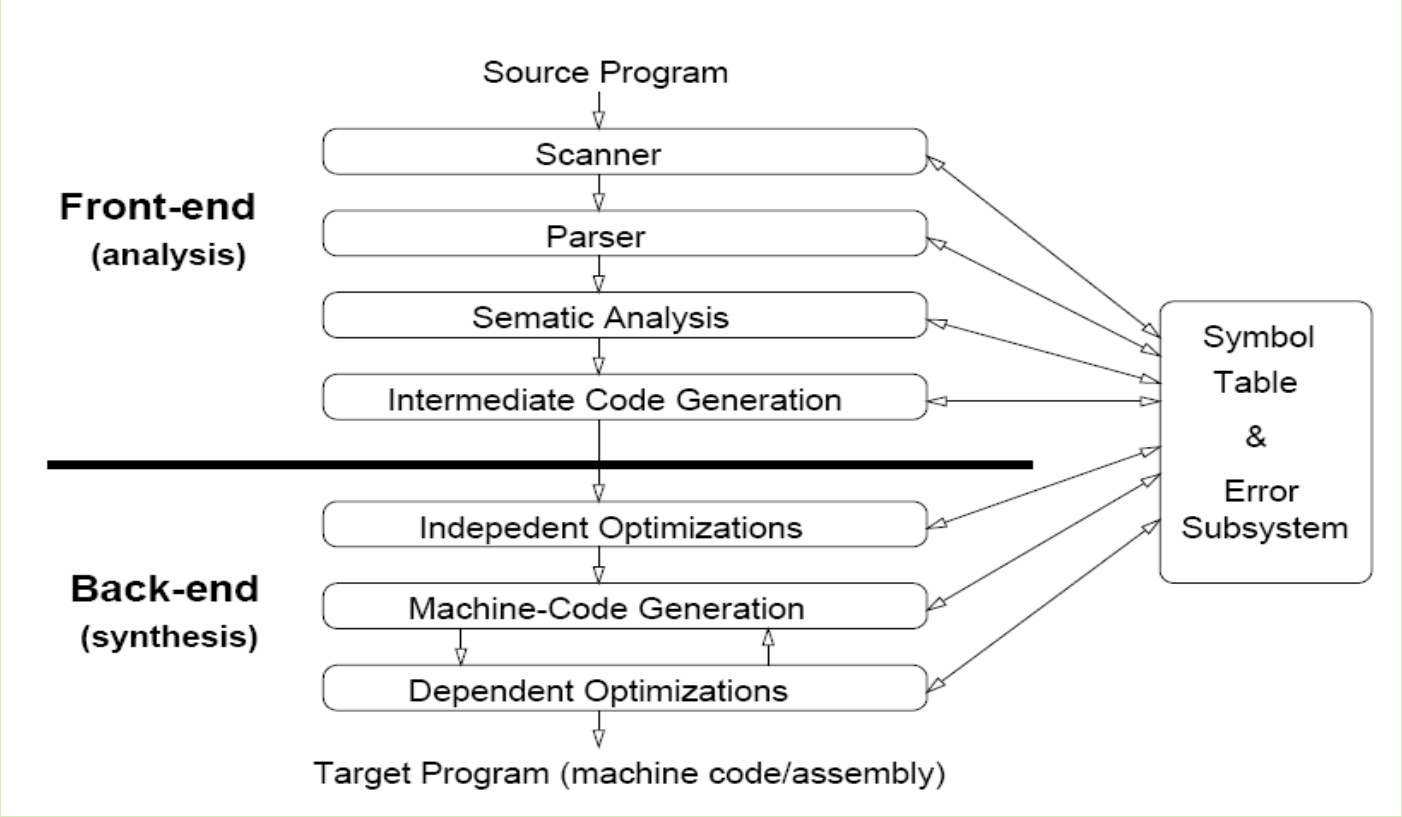
Why study compilers?

- Compilers use the whole spectrum of language processing technology
- Used in Variety of other fileds
  - Editors/ Word processors
  - Interpreters
  - Silicon Complier design – Circuit
  - Query Compilers etc…

The process of understanding a sentence

1. Recognize characters (alphabet, mathematical symbols, punctuations).
2. Group characters into logical entities (words).
3. Check the words form a structurally correct sentence
4. Check that the combination of words make sense
5. Plan what you have to do to accomplish the task
6. Execute it.

The structure (phases) of a compiler



Front end (analysis): depend on source language, independent on machine
- This is what we will focus (mainly the blue parts).

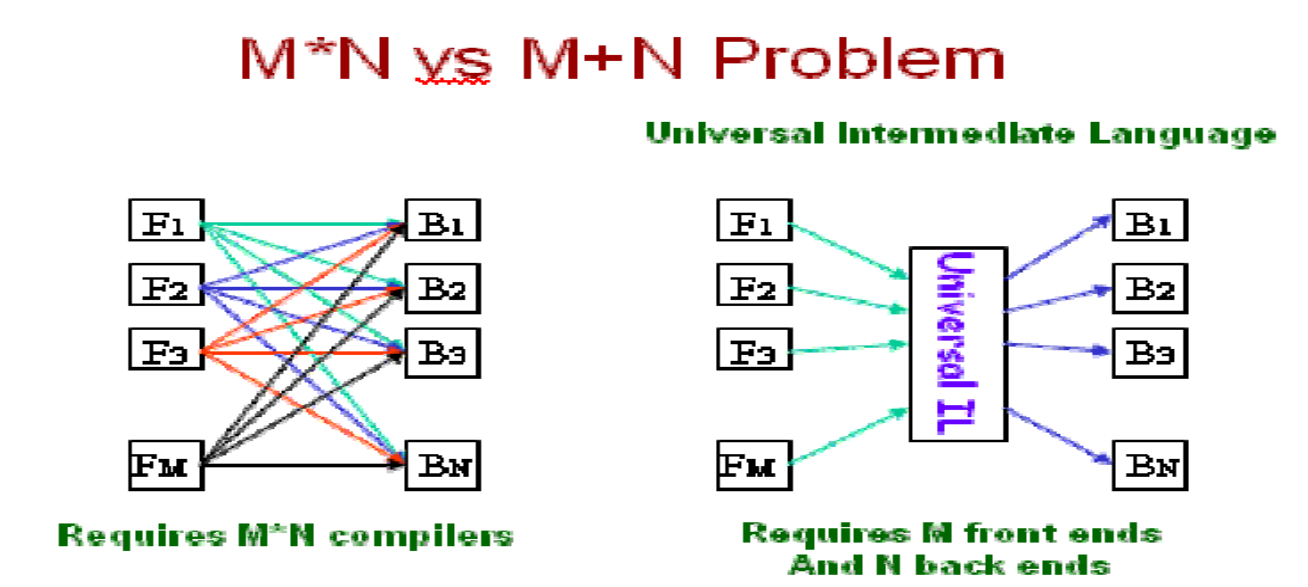
Backend (synthesis): dependent on machine and intermediate code, independent of source code.

Why front end and backend?

Source language	Target machine
C	Solaris
C#	AIX
Pascal	Pentium
Fortran	

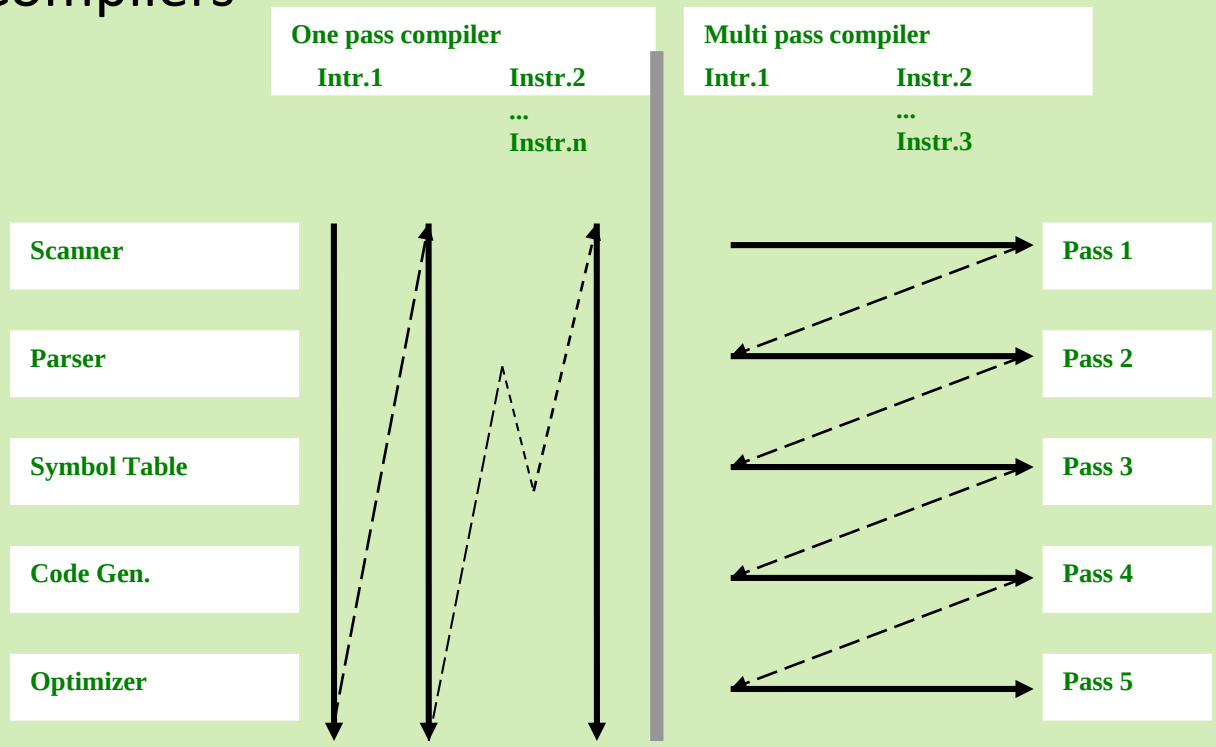
4 languages on 3 machines=12 compilers?

4 front end + 3 back end = 7 !

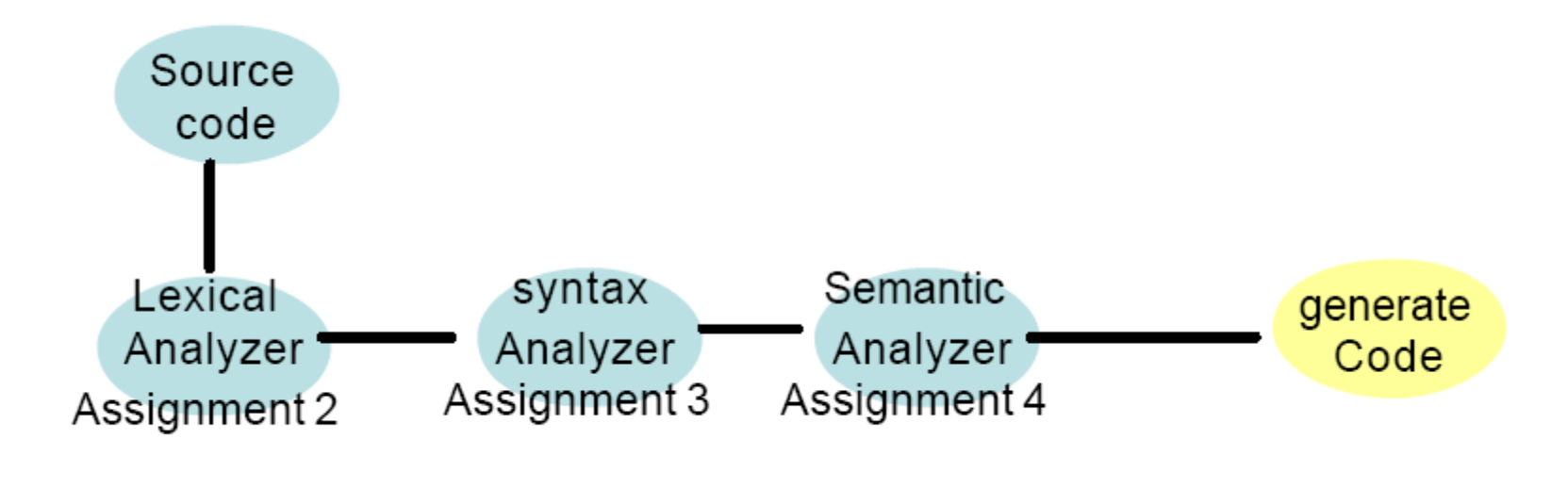


Efficiency Issues

- Pass- Single reading of the source file
  - Single Pass compliers
  - Multi-Pass Compilers

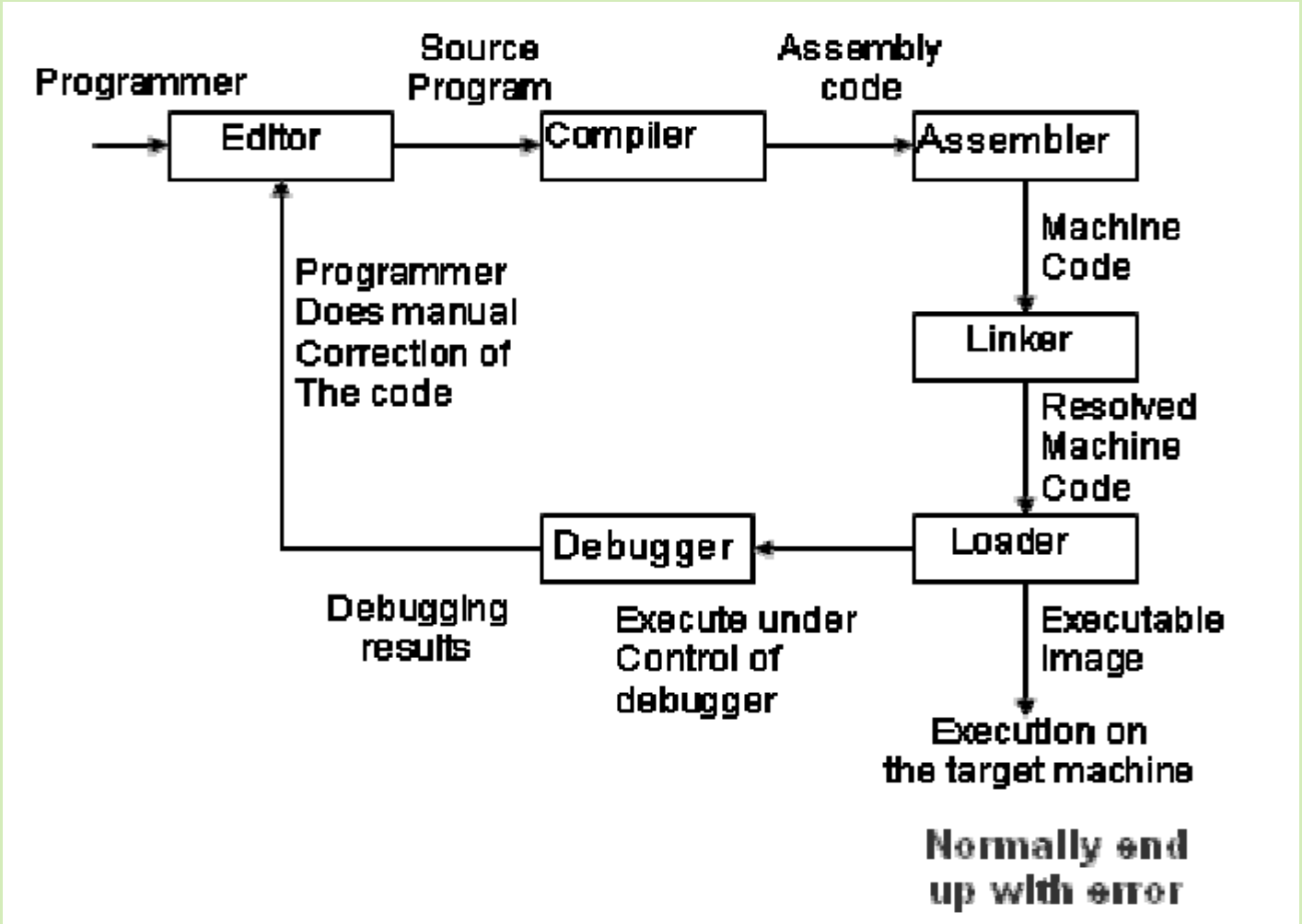


Assignments overview



The Big picture

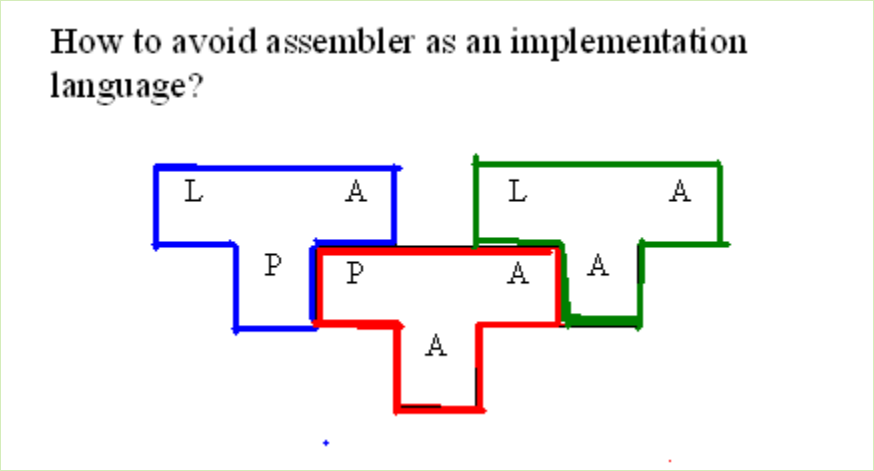
- **Compiler is part of program development environment**
  - The other typical components of this environment are editor, assembler, linker, loader, debugger, profiler etc.
  - The compiler (and all other tools) must support each other for easy program development



Compiler engineering techniques

- Straightforward approach
- Bootstrapping
- Using cross-compilers
- Using virtual machines
- Just-in-time compiling

Bootstrapping



1. Suppose that there exists a compiler K_A: $P \rightarrow A$, where P is a language with level higher than that of assembly.
2. We implement a compiler K_P: $L \rightarrow A$, and apply K_A to process the source code of the compiler K_P, resulting in a compiler $K_A = K_A(K_P)$: $L \rightarrow A$.
3. The described scheme (also shown on the slide) is called *bootstrapping*; the compiler K_P is said to be "bootstrapped" by the compiler K_A.

Cross Compilers

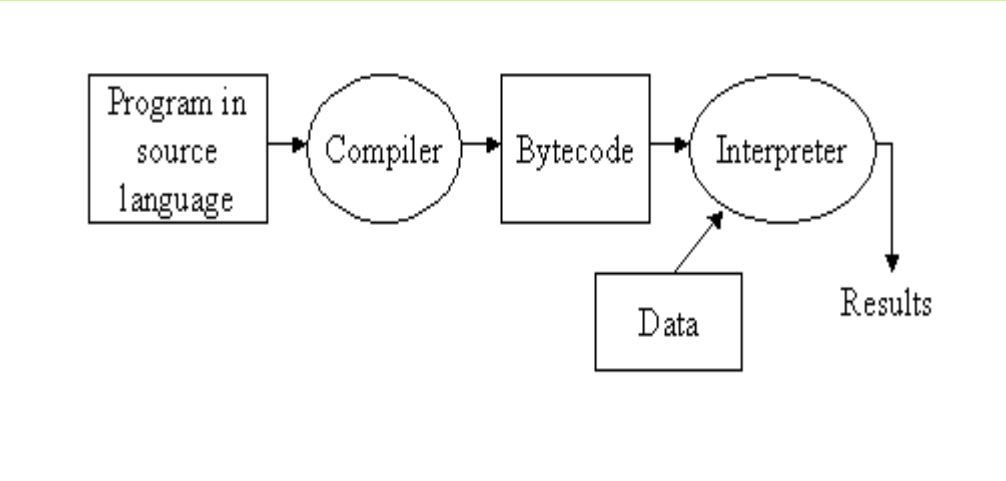
Suppose that we have two computers:

1. One is the computer *M* with the assembler language *A*.
2. The other one is the computer M_1 with the assembler language A_1, and a compiler $K_{A1}: P \rightarrow A_1$.
3. The computer *M* is unavailable, or there is no compiler $K_A: P \rightarrow A$.
4. Our objective is a compiler $K_A: L \rightarrow A$.
5. In this situation, M_1 can be used as the so-called *host computer* to implement a compiler $K_P: L \rightarrow A$.
6. The compiler $K_P: L \rightarrow A$ is called a *cross-compiler*.
7. Once *M* becomes available, K_P can be ported to *M* and bootstrapped with the aid of K_A.

A compiler is said to be **retargetable** if it can generate code for a range of target processors.

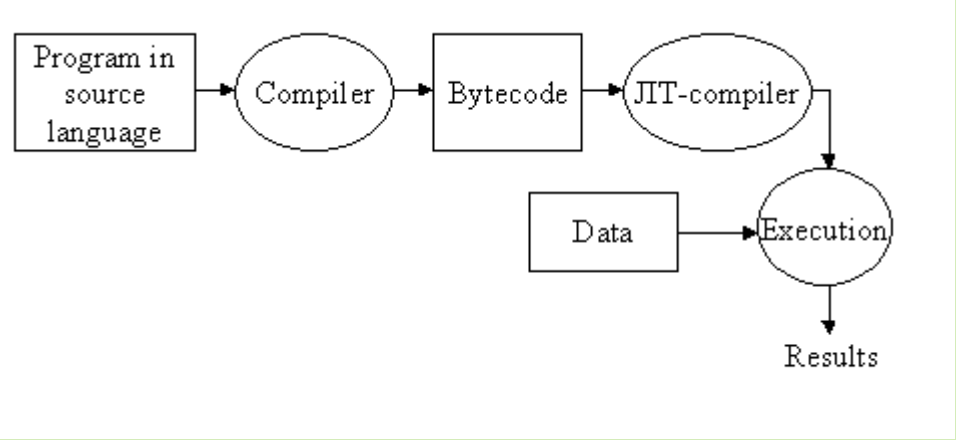
Usually retargetable compiler contains almost **no processor-specific code**, and characteristics of the target machines are captured in explicit target descriptions.

Virtual Machines



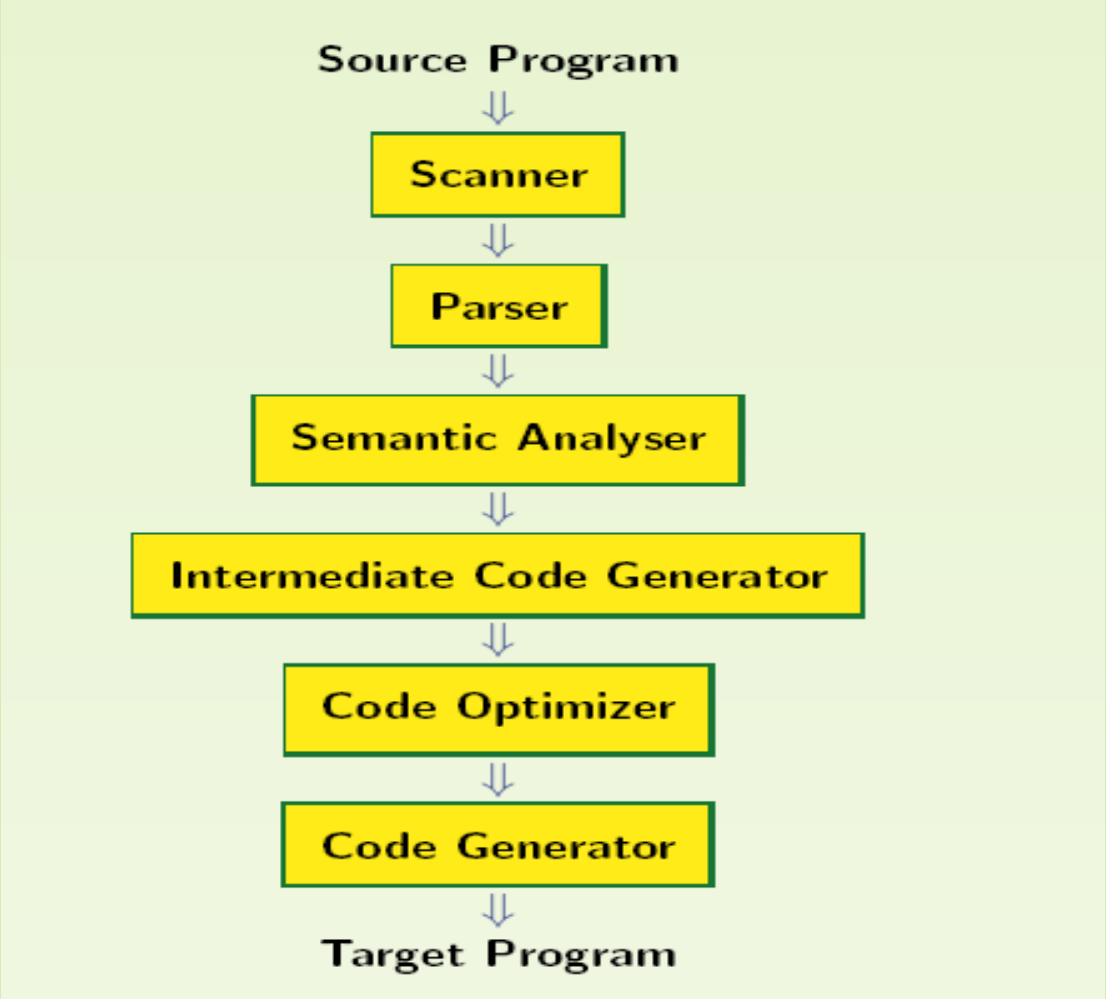
1. Another method uses *virtual machines* to obtain *portable* compilers
2. This approach implies translation of the source language into machine codes of a specially designed machine, which is not supposed to be implemented as a hardware device.
3. Then a virtual machine emulator is written for every target platform.

Just-in-time Compiling



1. The main disadvantage of virtual machines is the low performance of a program being interpreted compared to a program compiled to native machine code.
2. To improve application performance, the *just-in-time compiling* technology (sometimes also referred to as dynamic compilation) is used.
3. The idea is that JIT-compiler generates machine code directly in memory without storing it, which creates a great improvement of the performance.

Components of a Compiler



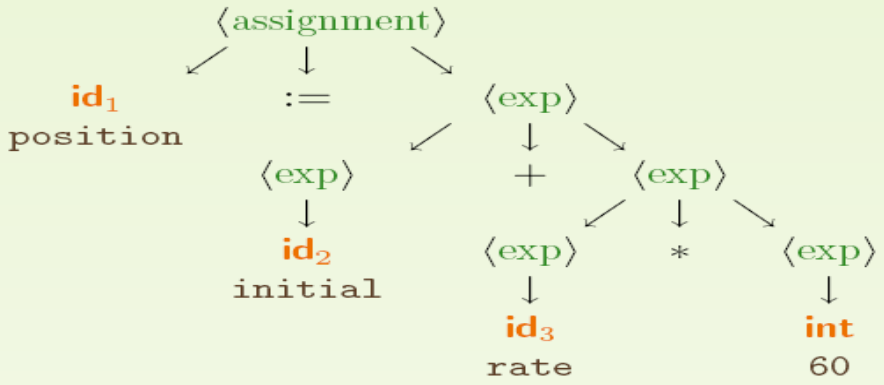
Analysis phase

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

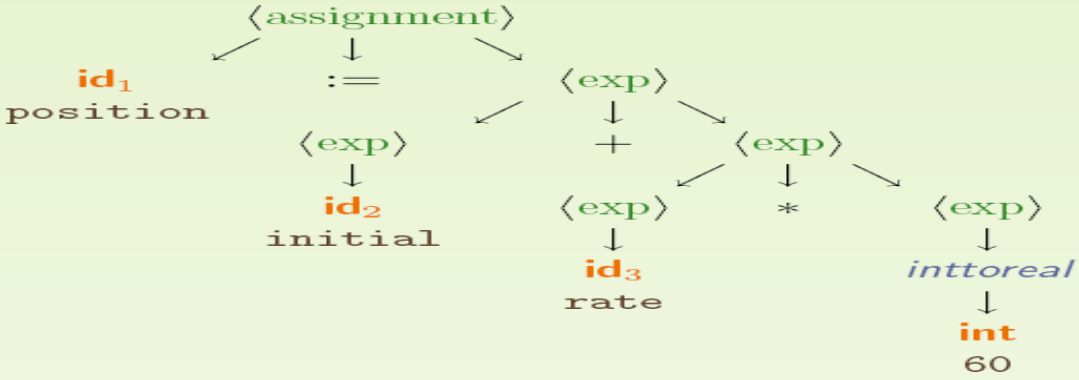
Lexical Analysis

$$\mathbf{id_1} := \mathbf{id_2} + \mathbf{id_3} * \mathbf{60}$$

Syntactic Analysis



Semantic Analysis



Lexical Analysis

- *Scanner*
  - groups characters into *tokens* – the basic unit of syntax

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

    becomes

    1. The identifier `position`
    2. The assignment operator `:=`
    3. The identifier `initial`
    4. The plus sign `+`
    5. The identifier `rate`
    6. The multiplication sign `*`
    7. The integer constant `60`.

  - character string forming a token is a *lexeme*
  - eliminates white space (blanks, tabs and returns)

- a key issue is *speed*

Syntactic Analysis

- *Parser*
  - groups tokens into grammatical phrases
  - represents the grammatical phases as a parse tree
  - produces meaningful error messages
  - attempts error detection and recovery
- You will come to know that the syntax of a language is specified by a *context-free grammar*.
- The typical arithmetic expressions are defined:

$$
\begin{array}{lcl}
\langle \text{expr} \rangle & \rightarrow & \langle \text{expr} \rangle + \langle \text{term} \rangle \mid \langle \text{expr} \rangle - \langle \text{term} \rangle \mid \langle \text{term} \rangle \\
\langle \text{term} \rangle & \rightarrow & \langle \text{term} \rangle * \langle \text{factor} \rangle \mid \langle \text{term} \rangle \,/\, \langle \text{factor} \rangle \mid \langle \text{factor} \rangle \\
\langle \text{factor} \rangle & \rightarrow & (\ \langle \text{expr} \rangle\ ) \mid \textbf{id} \mid \textbf{num}
\end{array}
$$

- *Parser generators mechanise much of the work* (yacc)

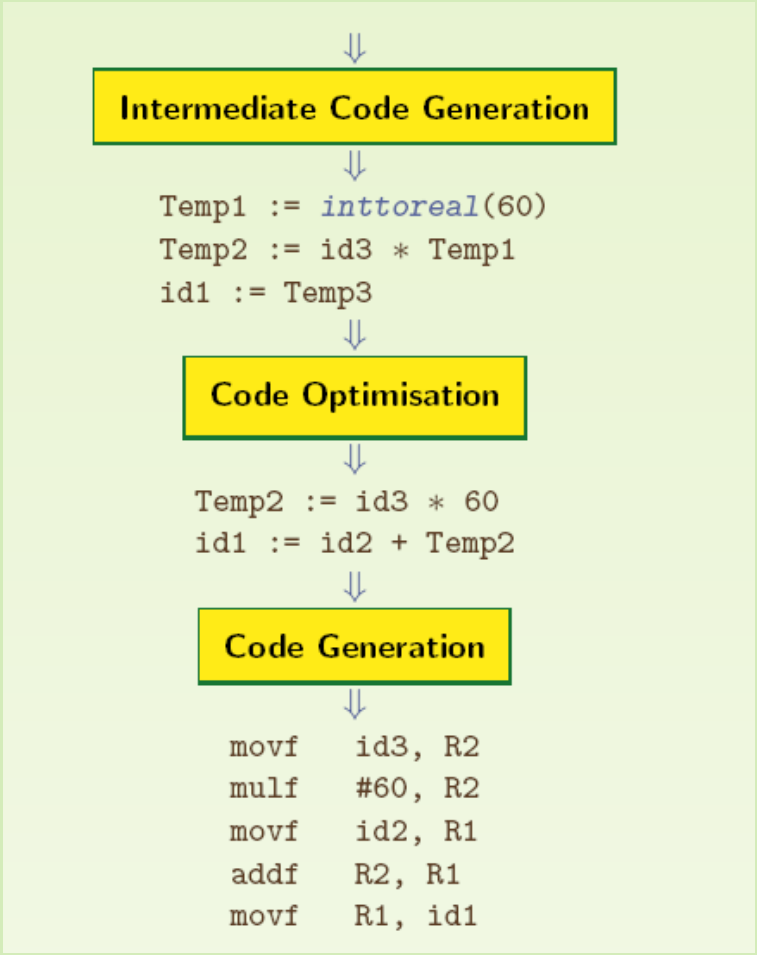
Semantic Analysis

- *Semantic Analyser*
  - Checks the program for semantic errors
    - variables defined before used
    - operands called with compatible types
    - procedures called with the right number and types of arguments
  - An important task: *type checking*
    - reals cannot be used to index an array
    - type conversions when some opeand coercions are permitted
  - The *symbol table* will be consulted

Name	Type	
initial	real	···
position	real	···
rate	real	···

- *It is assumed all three variables are declared reals*

Synthesis Phase



ICG

- *Intermediate Code Generator*
  - generates an explicit IR
- *Important IR properties:*
  - ease of generation
  - ease of translation into machine instructions
- Subtle decisions in the IR design have major effects on the speed and effectiveness of the compiler.
- *Popular IRs:*
  - Syntax trees
  - Directed acyclic graphs (DAGs)
  - Postfix notation
  - Three address code (3AC or quadruples)

Code Optimization

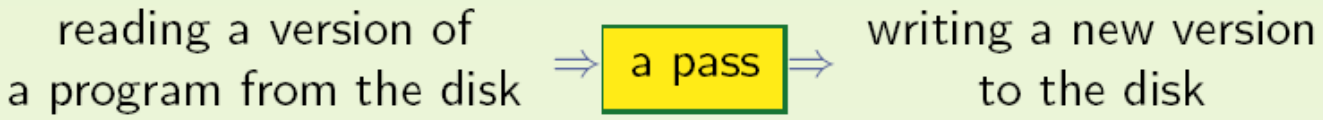
- *Code Optimiser*
  - analyses and improves IR
  - goal is to reduce runtime
  - must preserve values
- *Typical Optimisations*
  - discover & propagate some constant value
  - move a computation to a less frequently executed place
  - discover a redundant computation & remove it
  - remove code that is useless or unreachable
  - encode an idiom in some power instructions
- *There are simple* peephole *optimsations that significantly improve the running time.*
- *Will be introduced in just one lecture*

Code Generation

- *Code Generator*
  - generates target code: either relocatable machine code or assembly code
  - chooses instructions for each IR operation
  - decide what to keep in registers at each point
- *A crucial aspect is the assignment of variables to registers.*

Structure of a Compiler

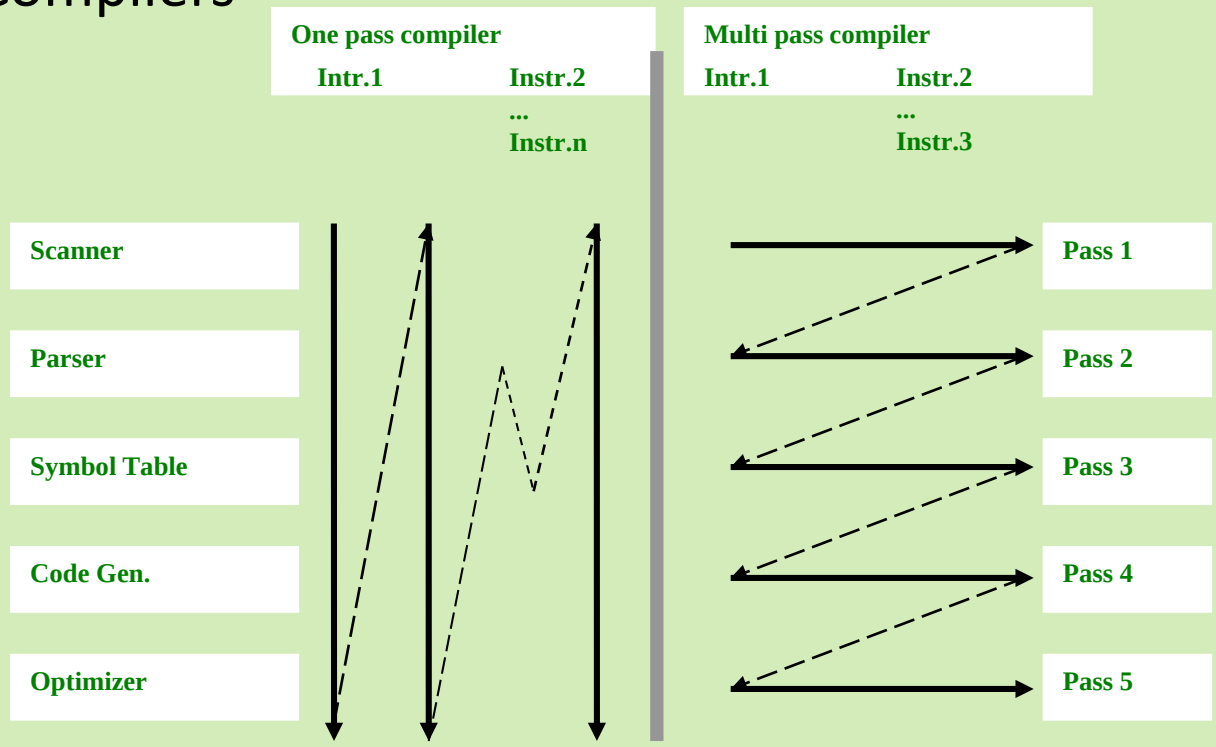
- What is a pass?



- The structures of a compiler:
  - One-Pass Compiler
  - Two-Pass Compiler

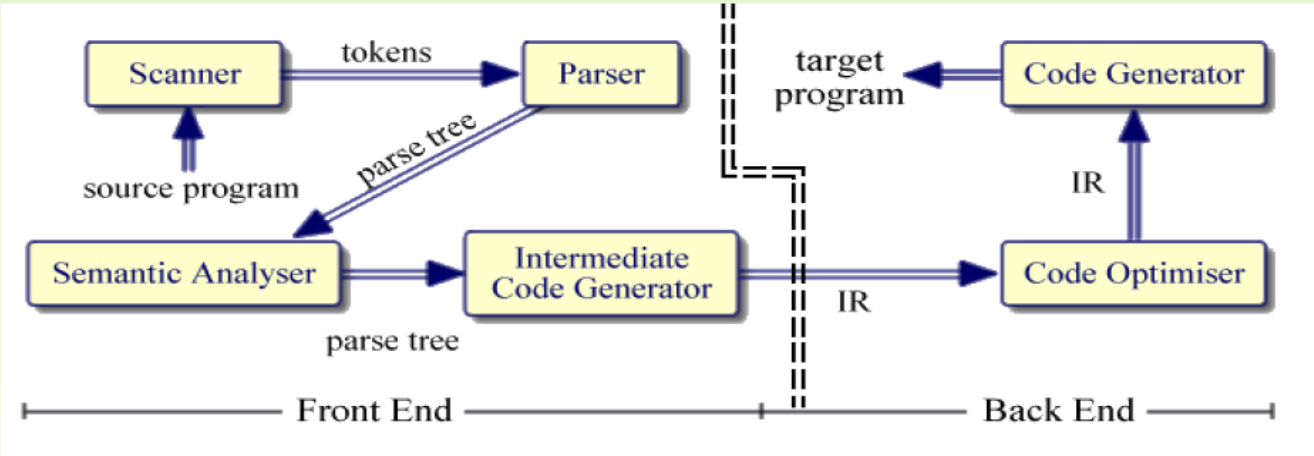
Efficiency Issues

- Pass- Single reading of the source file
  - Single Pass compliers
  - Multi-Pass Compilers



Two Pass Compiler

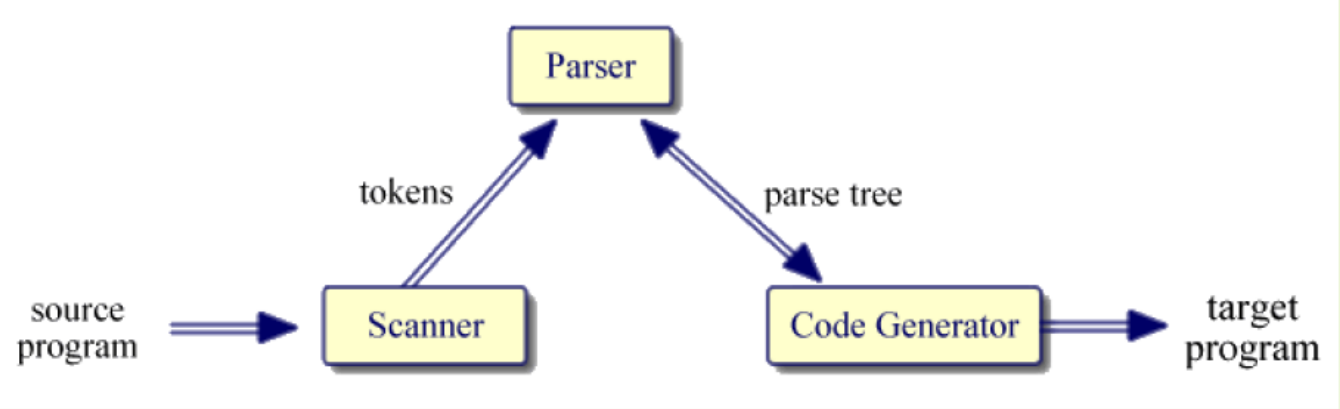
Two-Pass Compiler



- Pass 1 – The front end.
  - The parser is "in charge", i.e., acts as the main routine.
  - The parser calls the scanner as a subroutine to get the next token from the input.
  - After a statement analysed, the parser calls the intermediate code generator as subroutines to do the translation.
- Pass 2 – The back end.
  - An example: Ritchie's and Johnson's C compilers.

One Pass Complier

One-Pass Compiler



- *The three routines work in exactly the same way as in Pass 1 of a two-pass compiler.*
- Easy to implement, code inefficient, big memory required.
- Useful when the external storage is slow or unavailable.
- Non-applicable to some languages, e.g., PL/1, ALGOL68.
- An example: Wirth's and Ammann's first Pascal compilers.

Design Issues

Design Issues: One- or Multi-Pass?

- One-pass compiler wins when *compile speed* is the main concern.
  However, languages like PL/1 and ALGOL68 cannot be compiled in one pass, because they allow
  - *goto*'s that jump forward, and/or
  - variables to be used before they are declared. (Why?)
- Multi-pass compilers favored.
  - *modularity*
  - *speed of execution* more important since sophisticated code optimisations can be used, and/or
  - when *system resources* (e.g., memory) severely limited since requirements of each pass can be kept small.
  - Examples:
    - FORTRAN H Compiler has four passes.
    - Powell's Modula-2 Optimising Compiler has five passes.

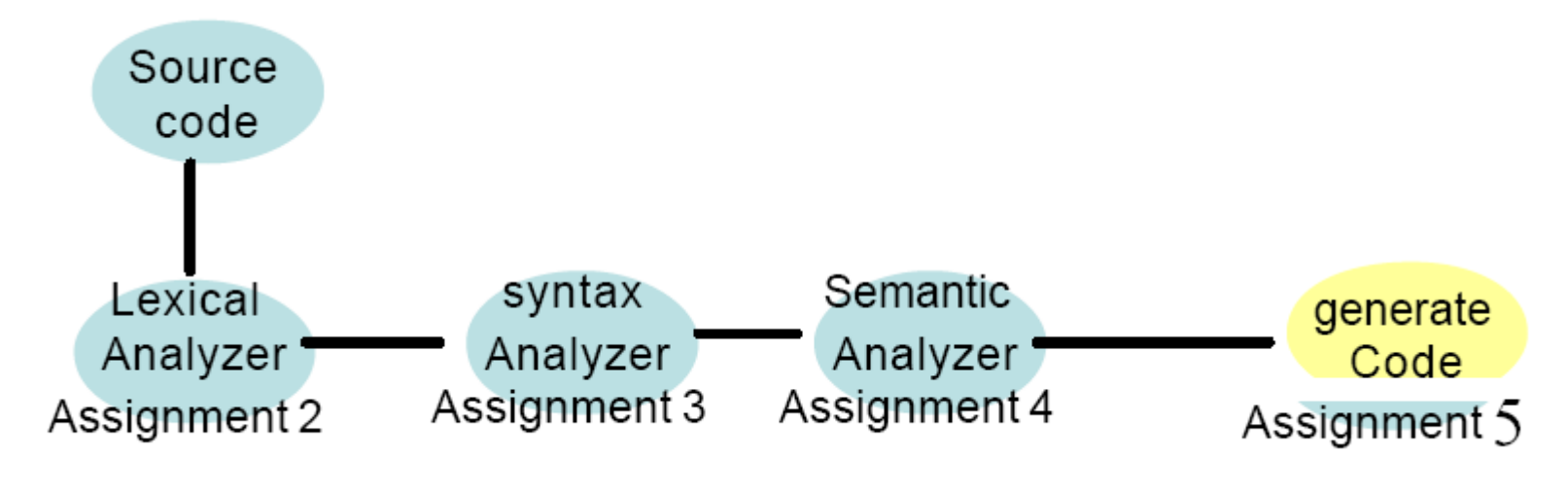
Symbol Table

The Symbol Table

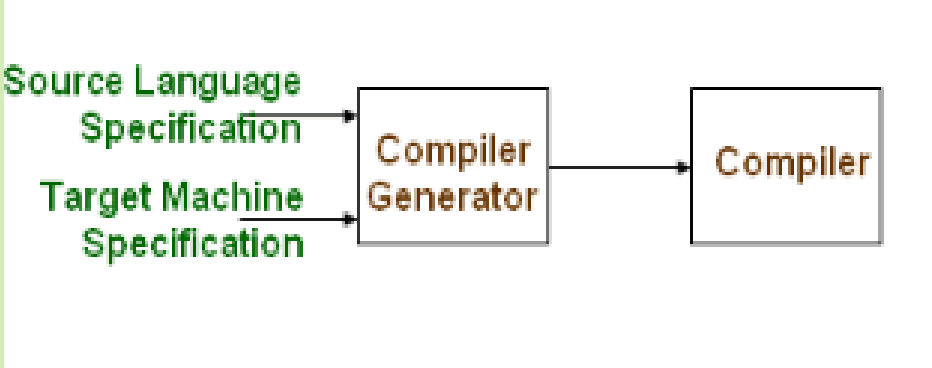
- A data structure containing a record for each identifier.
- The fields of an identifier's record contain the attributes of the identifier.
- The attributes of an identifier:
  - The storage allocated
  - Its type
  - Its scope (where in the program it is valid)
  - The number and types of its arguments and type returned in the case of procedure names.
  - etc.
- An example symbol table (conceptually):

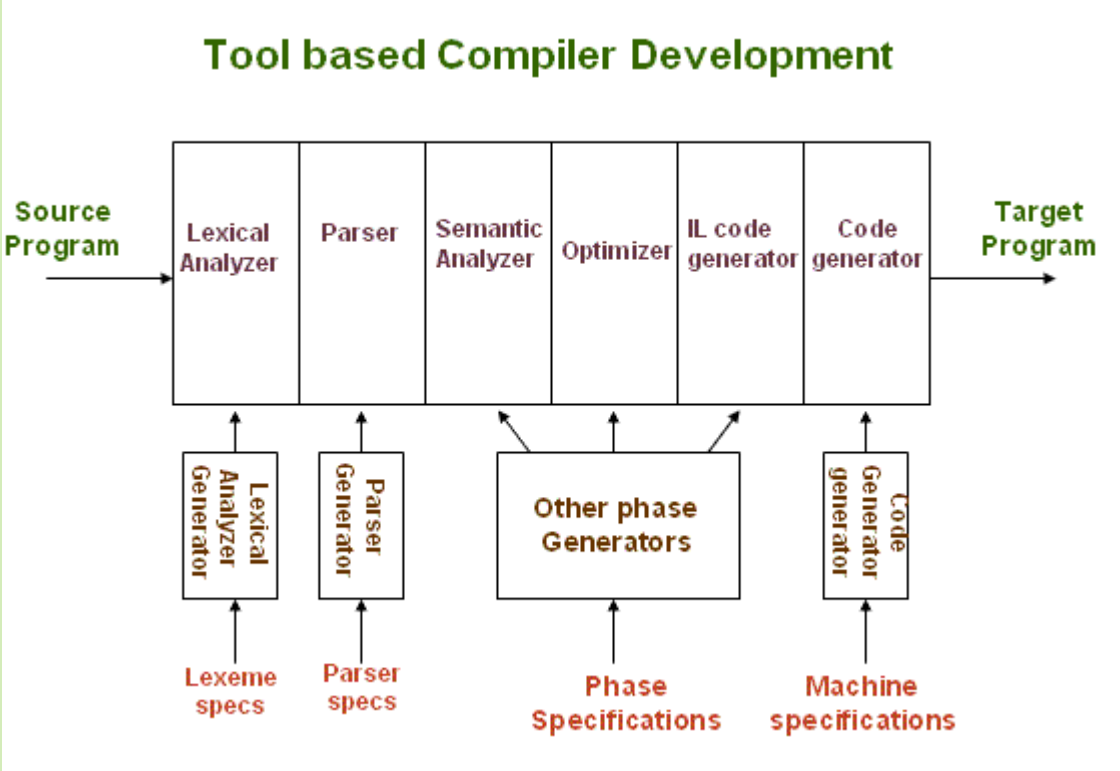
Name	Type	
initial	real	...
position	real	...
rate	real	...

Assignments overview



Compiler Generator





Lexical Analysis – Next class