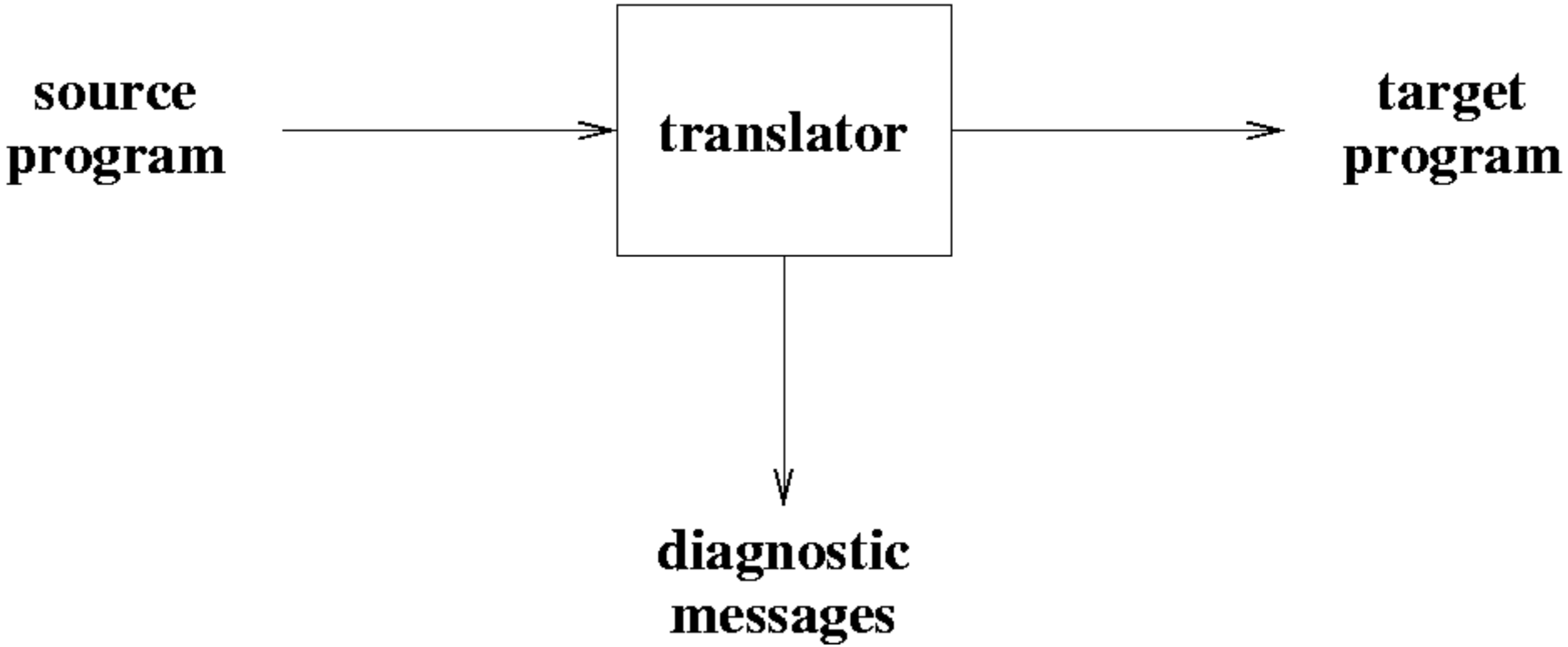
EECS 665 – Introduction

- Background and Future
- Concepts Introduced in Chapter 1
  - Phases
  - Compiler Construction Tools
  - Front Ends and Back Ends
  - Analysis-Synthesis Model
  - Assemblers
  - Linkers and Loaders

Compiler / Translator

- A translator is a program that reads a program written in a source language and translates it to an equivalent program written in a target language.



History and Milestones

- Until 1952
  - Most programming in assembly language
- 1952
  - Grace Hopper writes first compiler for the A-0 programming language
- 1957-58
  - John Backus and team writes first Fortran compiler
  - Optimization was an integral component of the compiler

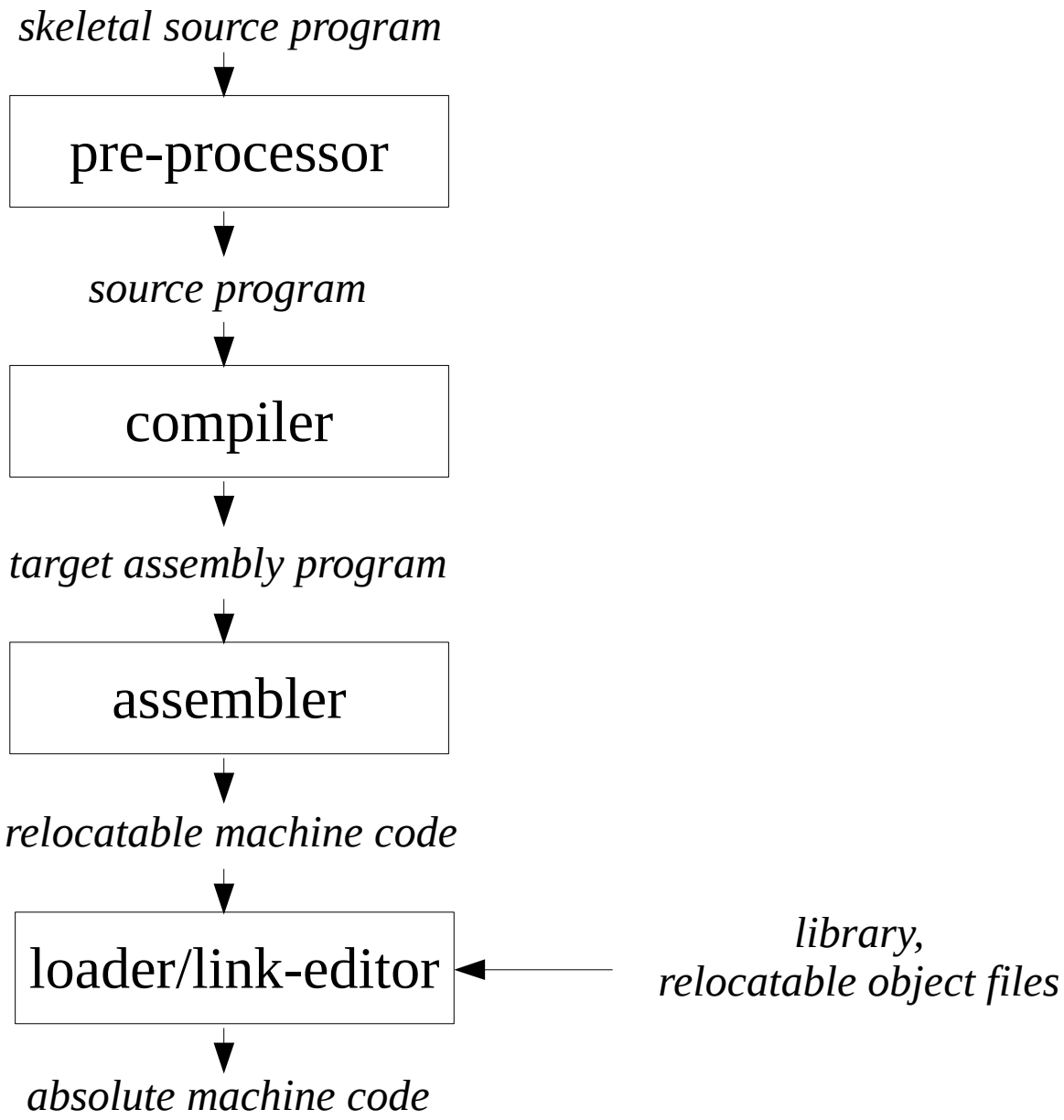
What the Future Holds

- Compiler construction is considered one of the success stories of computer science
  - Teaches us a lot about how to handle complex software projects
- Challenges for the future
  - Performance of generated code still important
  - Applications of compilers in security, safety, trust-worthiness
  - Multicore

Knowledge Required for Implementing a Successful Compiler

- Programming Languages
- Computer Architecture
- Formal Languages
- Algorithms
- Graph Theory
- Software Engineering

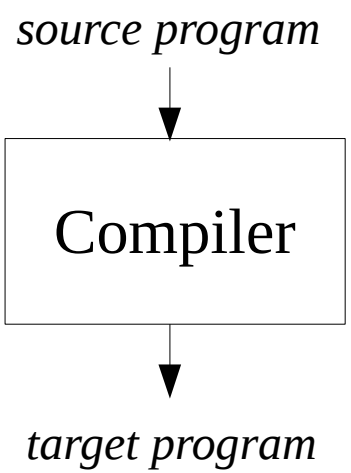
Language Processing System



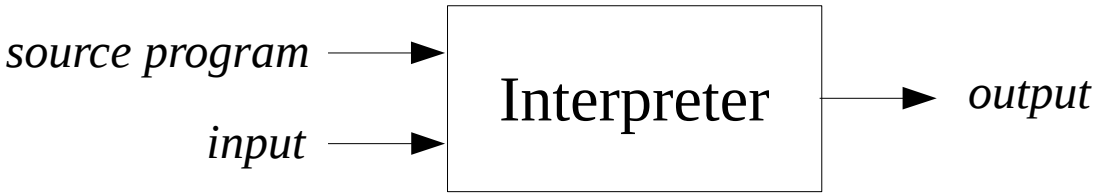
Applications Related to Compilers

- Compiler Relatives
  - Interpreters
  - Structure Editors
  - Pretty Printers
  - Static Checkers
  - Debuggers
- Other Applications
  - Text Formatters
  - Silicon Compilers
  - Query Interpreters

Compiler Vs. Interpreter

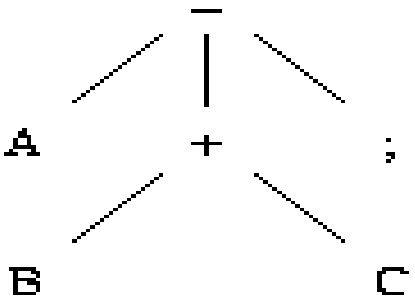


1. Execution of a compiled program



2. Execution of an interpreted program

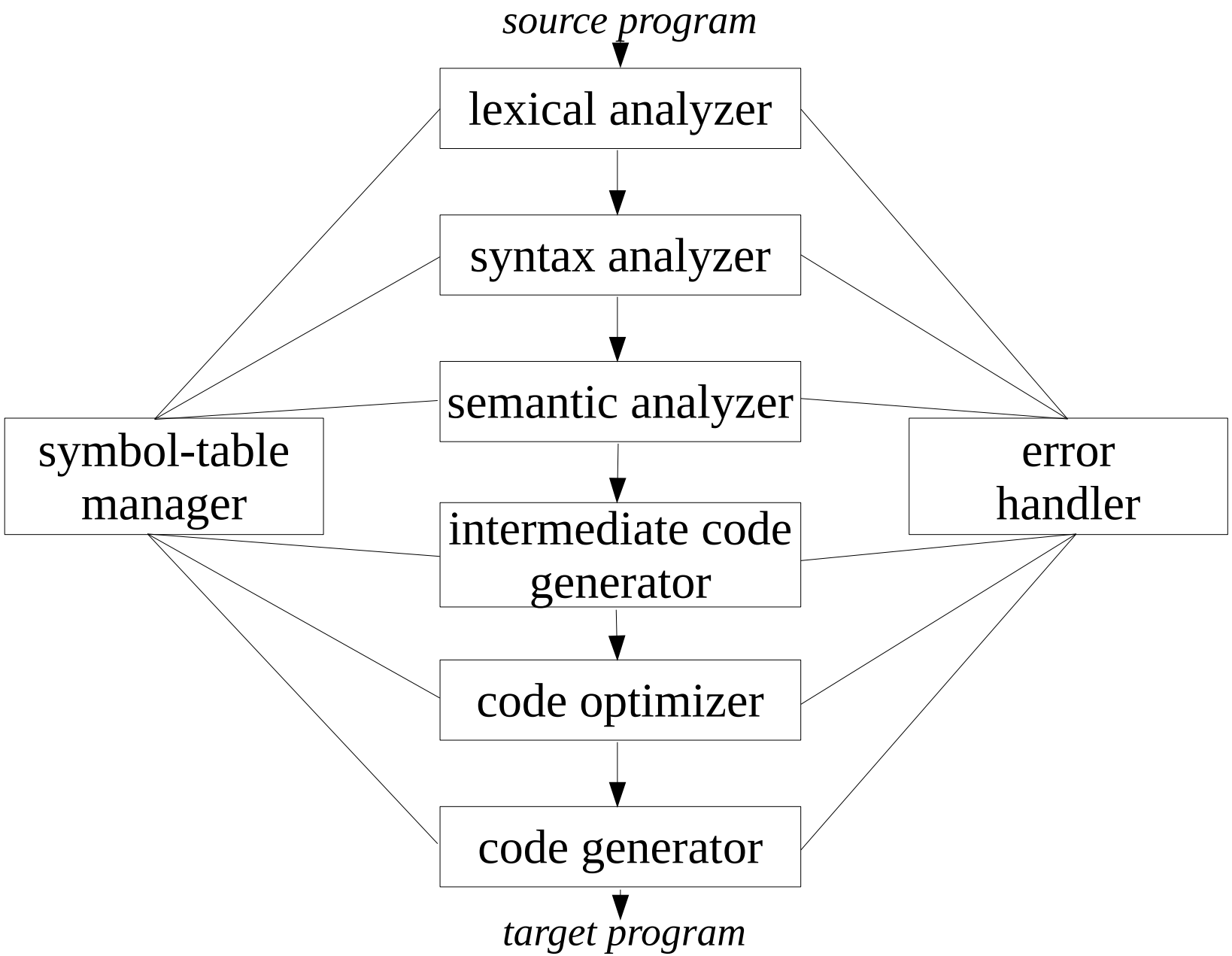
Compiler Phases

Phase	Output	Sample
programmer	source string	A=B+C;
scanner	token string	A, =, B, +, C, ;
parser	tree	
intermediate code generator	quads	t12 = float C A = B float add t12
optimizer	quads	A = B float add t9

code generator	assembly code	movf C,r1
		addf2 r1,r2
		movf r2,A

peephole optimizer	assembly code	addf2 C,r2 movf r2,A

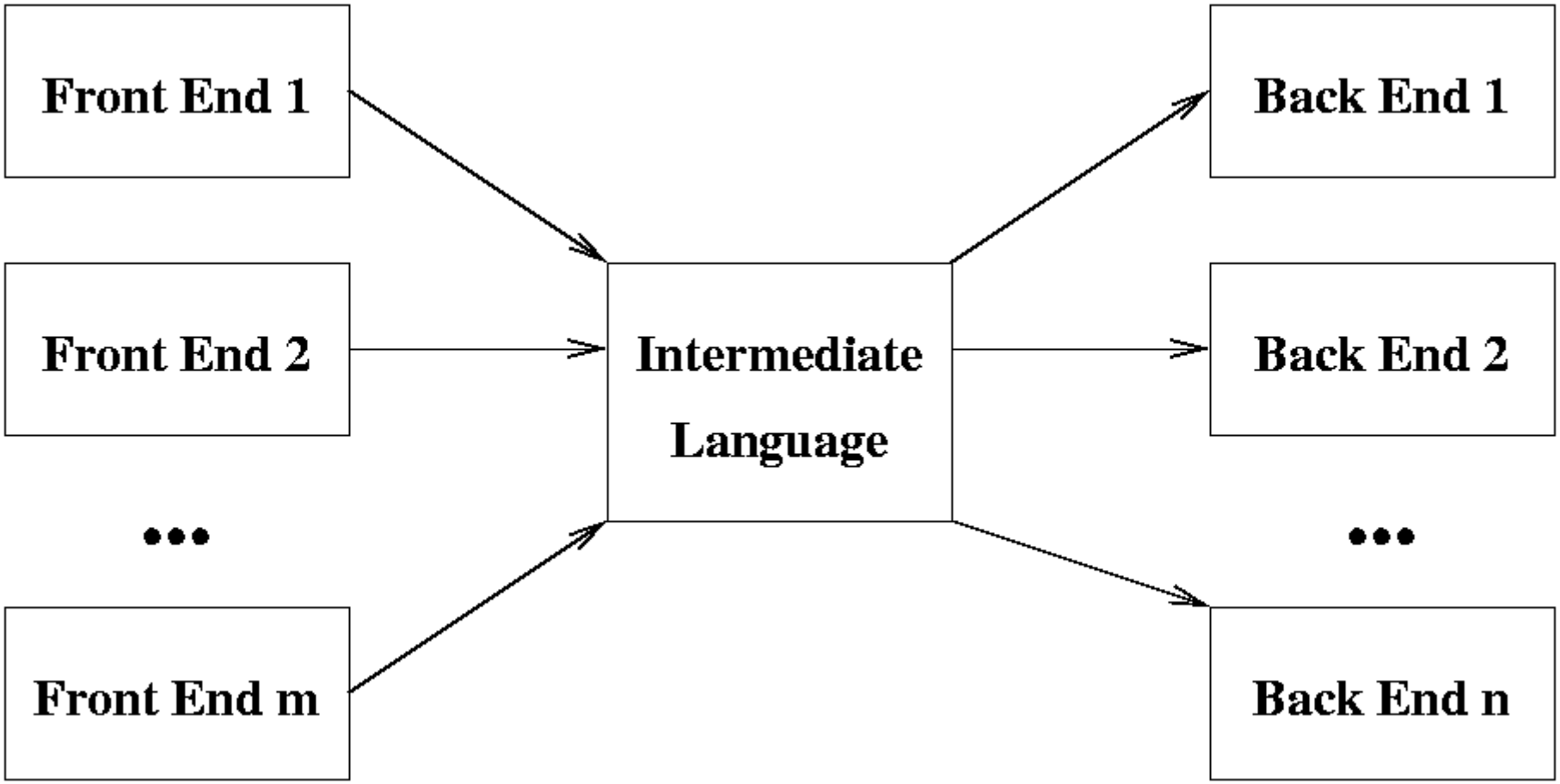
Phases of a Compiler



Compiler Construction Tools

- Front End (Analysis)
  - Scanner Generators: Lex
  - Parser Generators: Yacc
  - Syntax-Directed Translation Engines
- Back End (Synthesis)
  - Automatic Code Generators
  - Peephole Optimizer Construction Tools

Front Ends and Back Ends



Analysis-Synthesis Model of Compilation

- Analysis Part
  - Breaks up the source program into pieces and creates an intermediate representation.
- Synthesis Part
  - Constructs a target program from the intermediate representation.

3 Phases of Analysis in a Compiler

- Linear Analysis
  - Read a stream of characters and group into tokens.
- Hierarchical Analysis
  - Group tokens into hierarchical structures.
- Semantic Analysis
  - Perform certain checks to ensure that the program components fit together correctly.

Linear Analysis

- In a compiler this is also called lexical analysis or scanning.

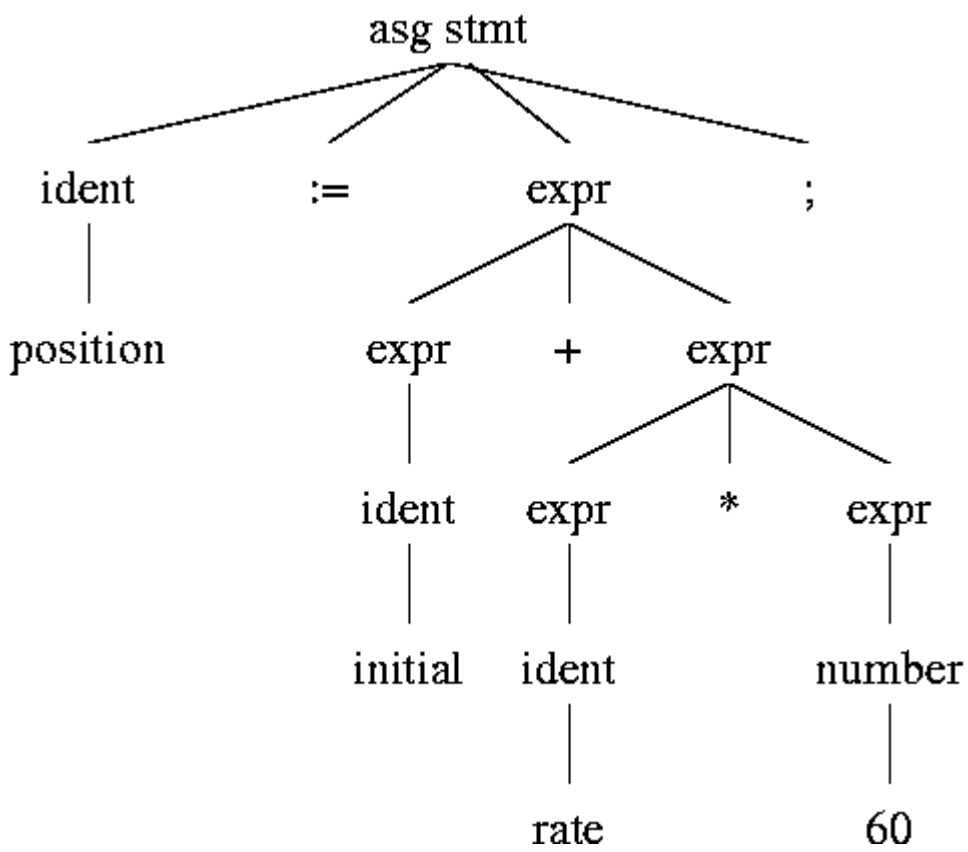
```
position := initial+rate*60;
```

=>

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

Hierarchical Analysis

- In a compiler this is called parsing or syntax analysis.
- It is usually expressed in a set of recursive rules called a grammar.
- Can be represented in a parse tree.



Semantic Analysis

- Checks for errors that can't be checked though syntax analysis alone.
  - Consistent use of types.
  - Variables declared before referenced.
- Determines where conversions need to be performed.

Intermediate Code Generation

- After analysis, most compilers generate an intermediate representation of a program.
- Properties
  - machine-independent
  - easy to translate to the target machine language
- Can have a common intermediate language that is the target of several front ends and is input to several back ends.

Code Optimization

- Often performed on intermediate code.
- Goals
  - Make program run faster.
  - Make program take up less space.
  - Make program use less power.
- Should never change the semantic behavior of the program.

Code Generation

- Produces assembly or object code from the intermediate representation.
- Each intermediate operation is translated to an equivalent sequence of machine instructions.
- Special features of the architecture are exploited.

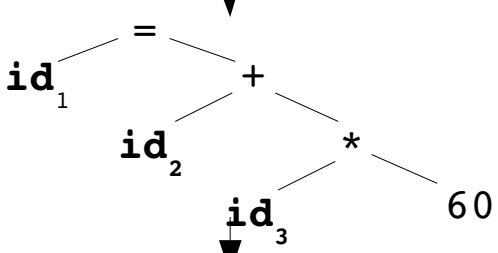
Translation of a Statement

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

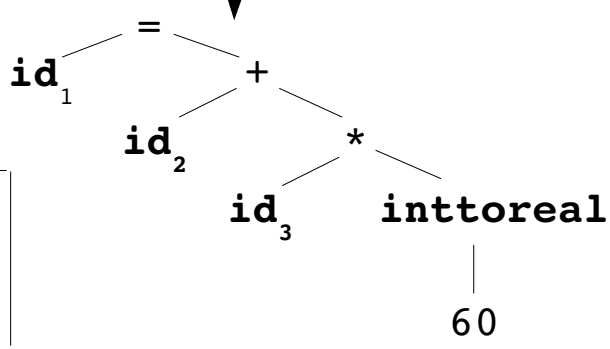
lexical analyzer

$id_1 = id_2 + id_3 * 60$

syntax analyzer



semantic analyzer



symbol table

position	...
initial	...
rate	...

intermediate code generator

```
temp1 = inttoreal(60)
temp2 = id3 * temp1
temp3 = id2 + temp2
id1   = temp3
```

code optimizer

```
temp1 = id3 * 60.0
id1   = id2 + temp1
```

code generator

```
MOVF  id3, R2
MULF  #60.0, R2
MOVF  id2, R1
ADDF  R2, R1
MOVF  R1, id1
```

Preprocessors

- Perform some preliminary processing on a source module.
  - definitions and macros
    - #define
  - file inclusion
    - #include
  - conditional compilation
    - #ifdef
  - line numbering
    - #line

Assemblers

- Typically accomplished in 2 passes.
  - Pass 1: Stores all of the identifiers representing tokens in a table.
  - Pass 2: Translates the instructions and data into bits for the machine code.
- Produces relocatable code.

Linkers and Loaders

- Linker
  - Produces an executable file.
  - Resolves external references.
  - Includes appropriate libraries.
- Loader
  - Creates a process from the executable.
  - Loads the process (or a portion of it) into main memory.
  - Produces absolute machine code.