Introduction to Compiler

Objectives

- Know how to build a compiler for a (simplified) (programming) language
- Know how to use compiler construction tools, such as generators for scanners and parsers
- Be able to write LL(1), LR(1), and LALR(1) grammars (for new languages)
- Be familiar with compiler analysis and optimization techniques
- … learn how to work on a larger software project!

Compilers and Interpreters

- "*Compilation*"
  - Translation of a program written in a source language into a semantically equivalent program written in a target language

Compilers and Interpreters (cont'd)

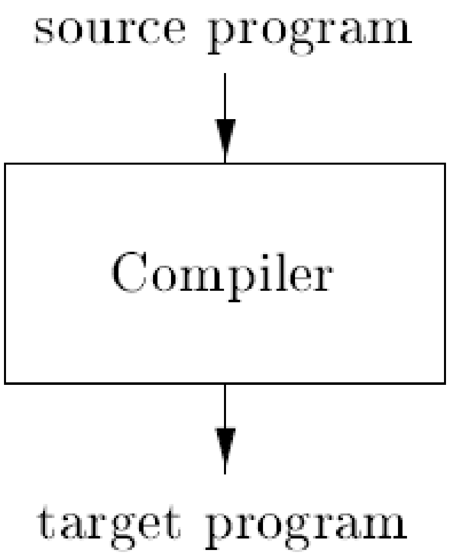


Figure 1.1: A compiler

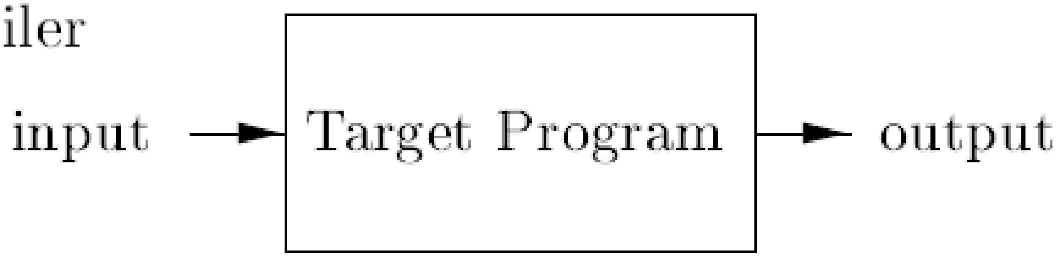


Figure 1.2: Running the target program

Compilers and Interpreters (cont'd)

- "*Interpretation*"
  - Performing the operations implied by the source program

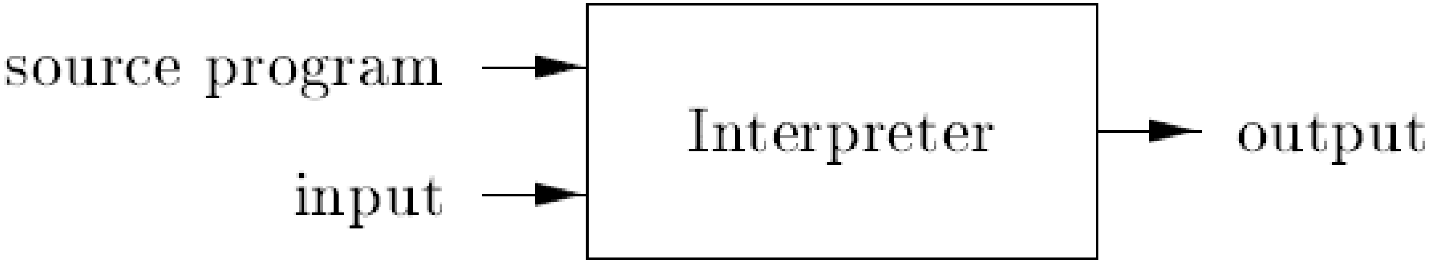


Figure 1.3: An interpreter

The Analysis-Synthesis Model of Compilation

- There are two parts to compilation:
  - *Analysis* determines the operations implied by the source program which are recorded in a tree structure
  - *Synthesis* takes the tree structure and translates the operations therein into the target program

Other Tools that Use the Analysis-Synthesis Model

- *Editors* (syntax highlighting)
- *Pretty printers* (e.g. doxygen)
- *Static checkers* (e.g. lint and splint)
- *Interpreters*
- *Text formatters* (e.g. TeX and LaTeX)
- *Silicon compilers* (e.g. VHDL)
- *Query interpreters/compilers* (Databases)

Preprocessors, Compilers, Assemblers, and Linkers

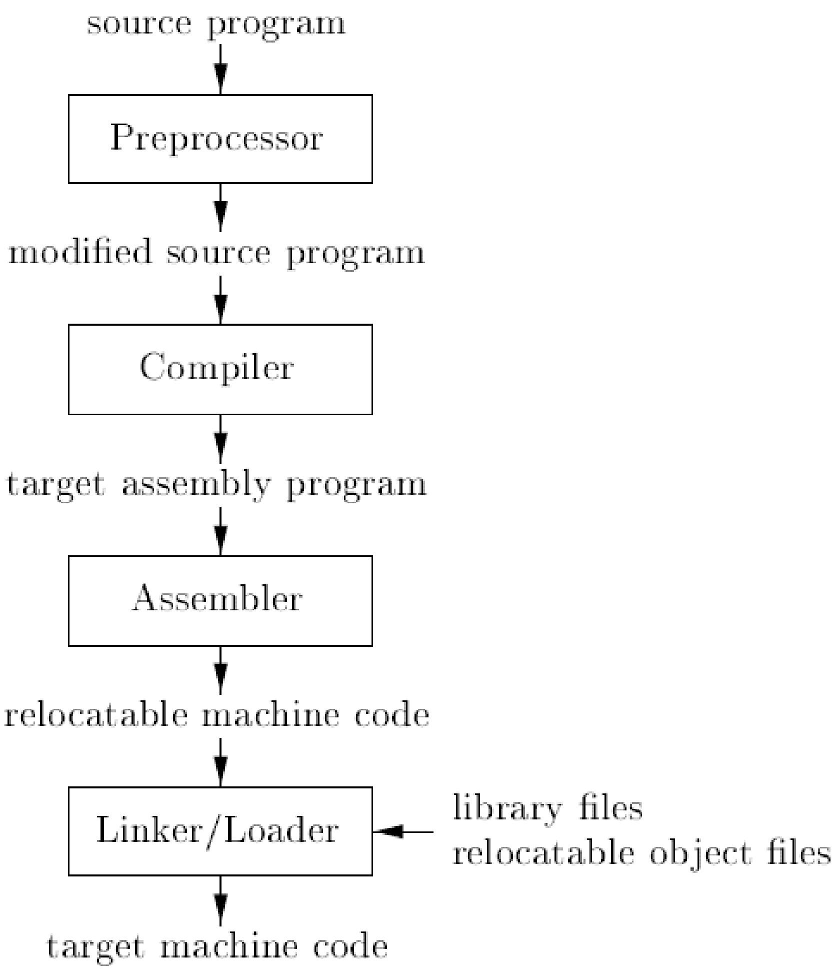


Figure 1.5: A language-processing system

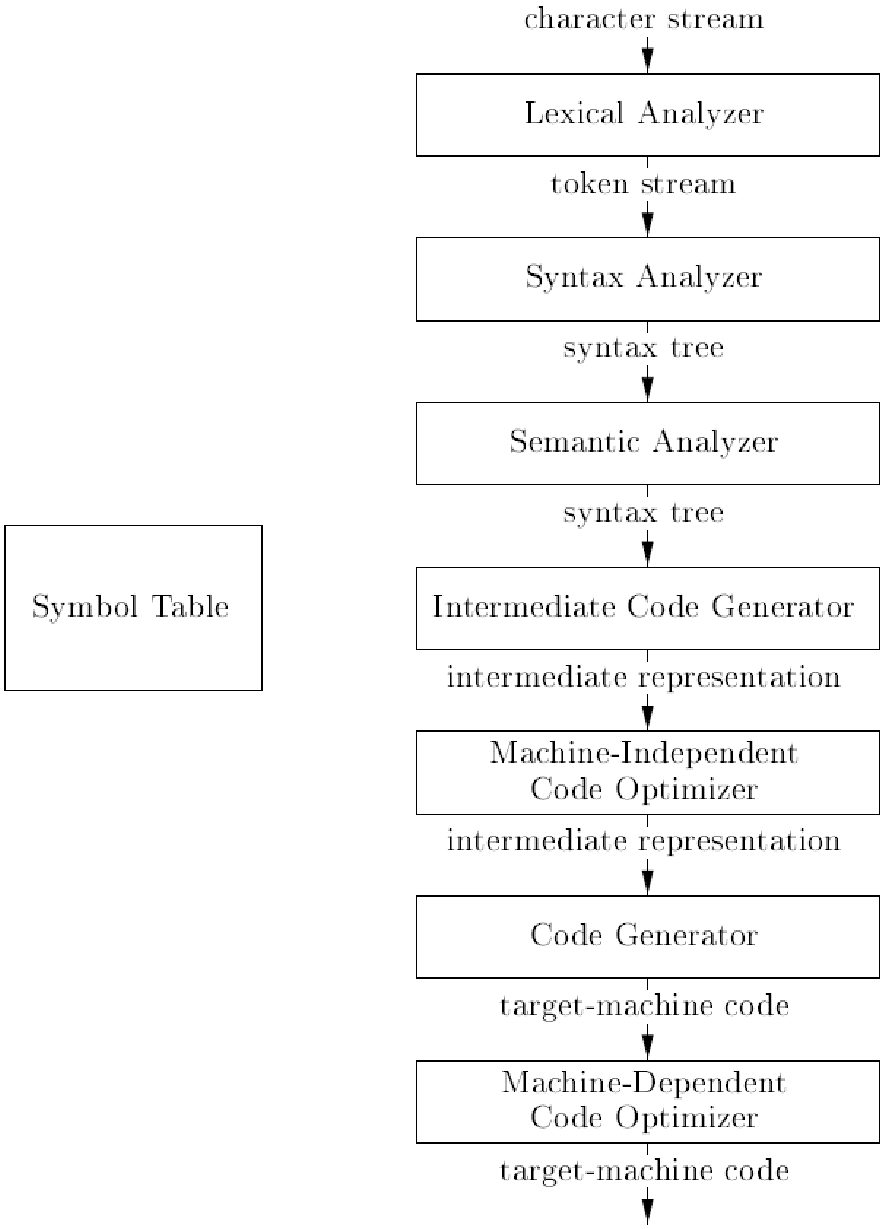
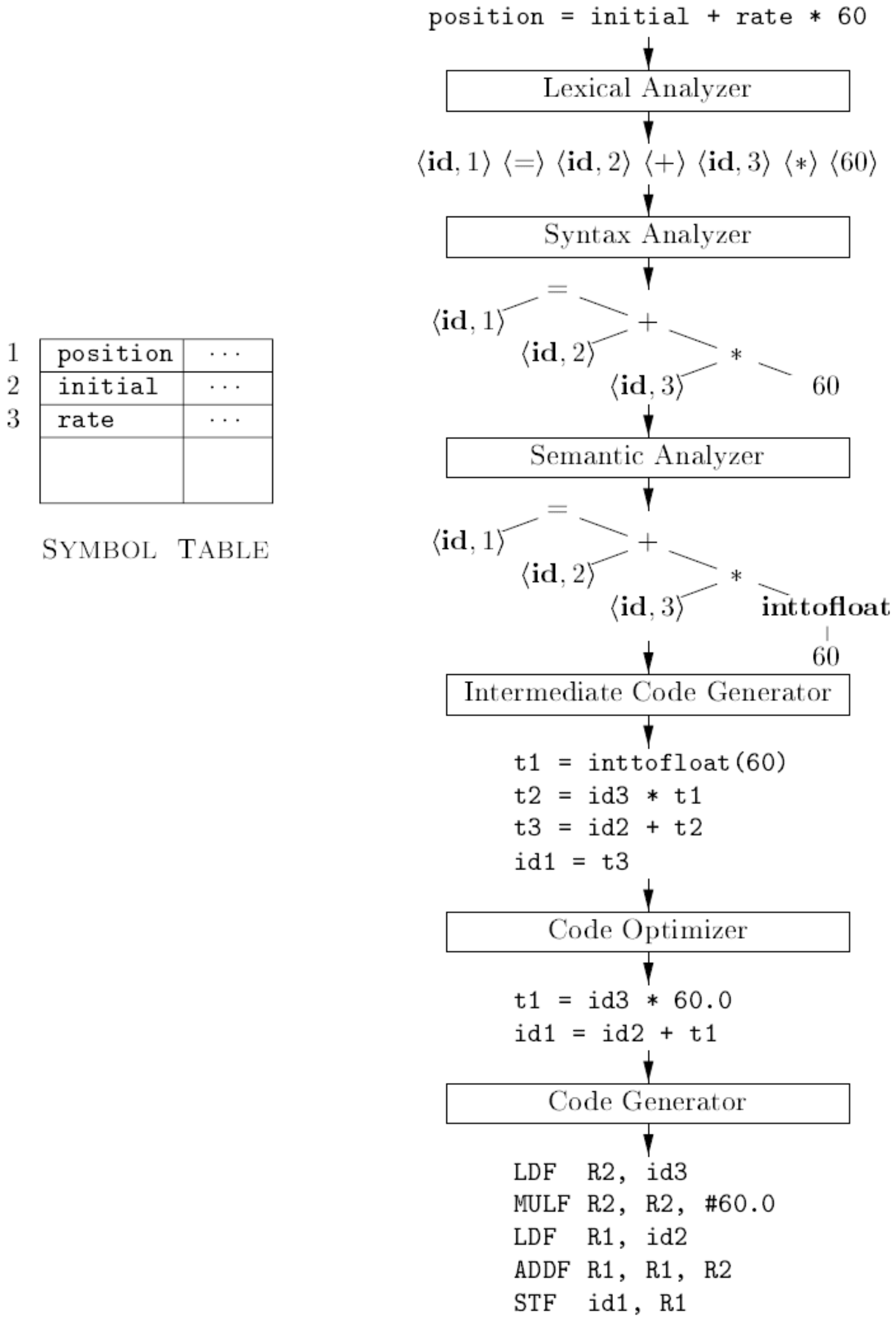


Figure 1.6: Phases of a compiler



1	position	···
2	initial	···
3	rate	···

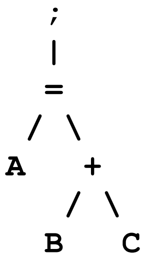
```
t1 = inttofloat(60)
t2 = id3 * t1
t3 = id2 + t2
id1 = t3
```

```
t1 = id3 * 60.0
id1 = id2 + t1
```

```
LDF  R2, id3
MULF R2, R2, #60.0
LDF  R1, id2
ADDF R1, R1, R2
STF  id1, R1
```

Figure 1.7: Translation of an assignment statement

The Phases of a Compiler

Phase	Output	Sample
Programmer	Source string	**`A=B+C;`**
Scanner (performs *lexical analysis*)	Token string	**‘A’, ‘=’, ‘B’, ‘+’, ‘C’, ‘;’** And *symbol table* for identifiers
Parser (performs *syntax analysis* based on the grammar of the programming language)	Parse tree or abstract syntax tree	
Semantic analyzer (type checking, etc)	Parse tree or abstract syntax tree	

Intermediate code generator	Three-address code, quads, or RTL	**`int2fp B t1`**
		`+ t1 C t2`
		`:= t2 A`

Optimizer	Three-address code, quads, or RTL	**`int2fp B t1`**
		`+ t1 #2.3 A`

Code generator	Assembly code	**`MOVF #2.3,r1`**
		`ADDF2 r1,r2`
		`MOVF r2,A`

Peephole optimizer	Assembly code	**`ADDF2 #2.3,r2`**
		`MOVF r2,A`

The Grouping of Phases

- Compiler front and back ends:
  - Analysis (*machine independent* front end)
  - Synthesis (*machine dependent* back end)
- Passes
  - A collection of phases may be repeated only once (*single pass*) or multiple times (*multi pass*)
  - Single pass: usually requires everything to be defined before being used in source program
  - Multi pass: compiler may have to keep entire program representation in memory

Compiler-Construction Tools

- Software development tools are available to implement one or more compiler phases
  - *Scanner generators*
  - *Parser generators*
  - *Syntax-directed translation engines*
  - *Automatic code generators*
  - *Data-flow engines*

Outline

- Ch. 1: Introduction
- Ch. 2: A Simple Syntax-Directed Translator
- Ch. 3: Lexical Analysis
- Ch. 4: Syntax Analysis
- Ch. 5: Syntax-Directed Translation
- Ch. 6: Intermediate Code Generation
- Ch. 7: Run-Time Environments
- Ch. 8: Code Generation