Compiler Design

IIITS

Evaluation

- Will be announced.

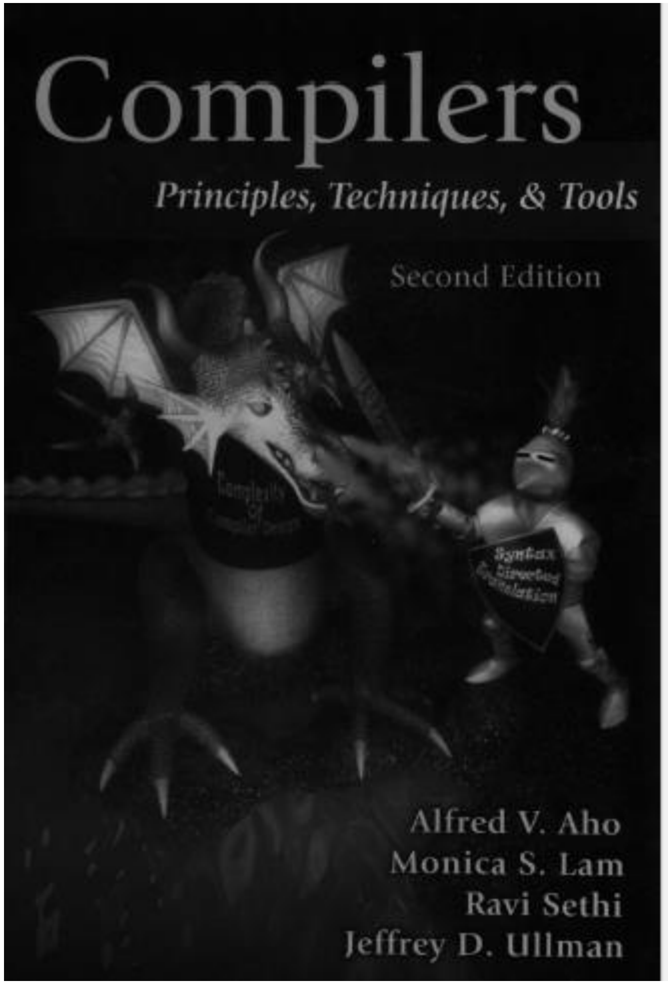
- Assignments and mini-project may play important roles.

Miniproject – Evaluated at various stages.

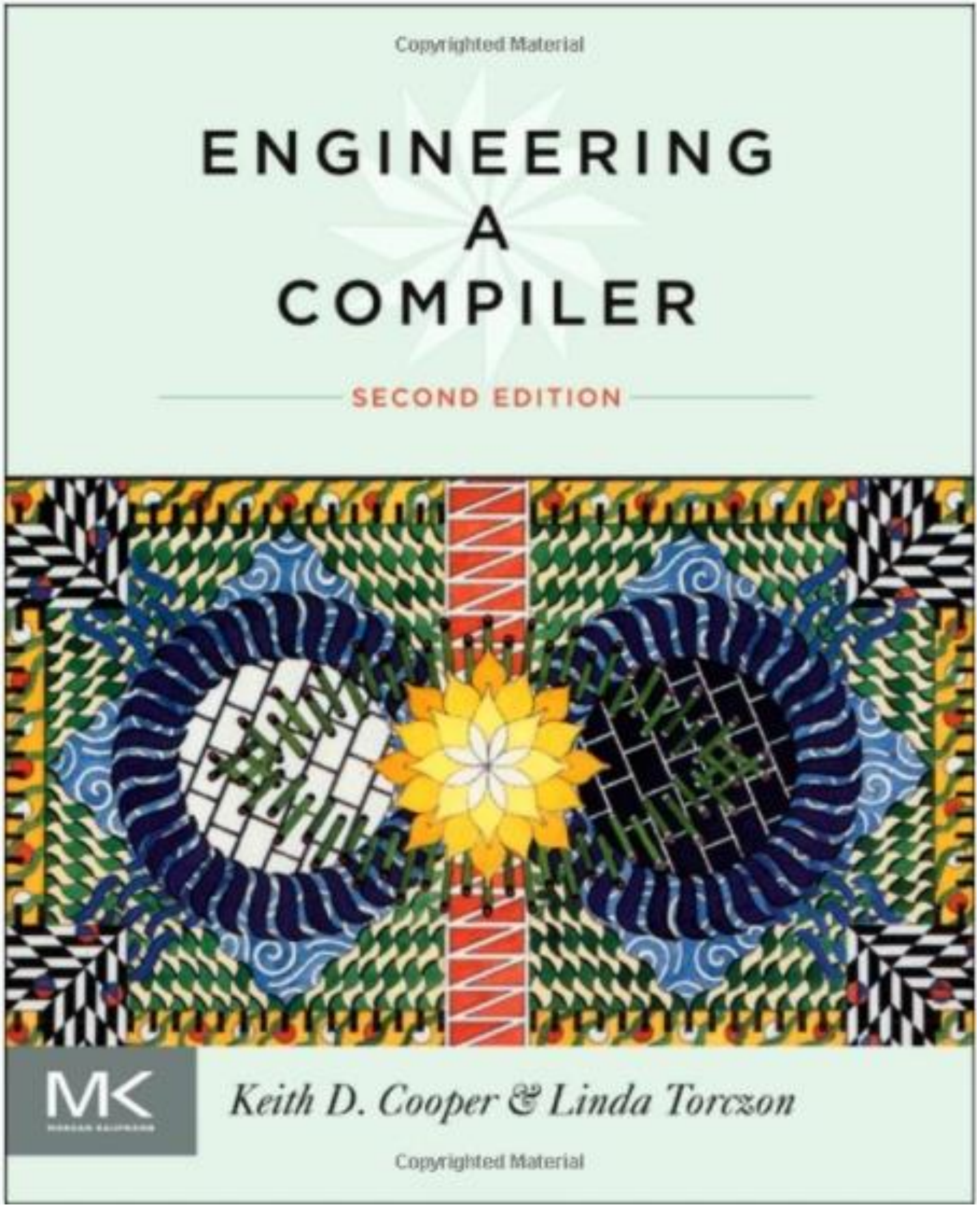
You will create a language and corresponding compiler or interpreter.

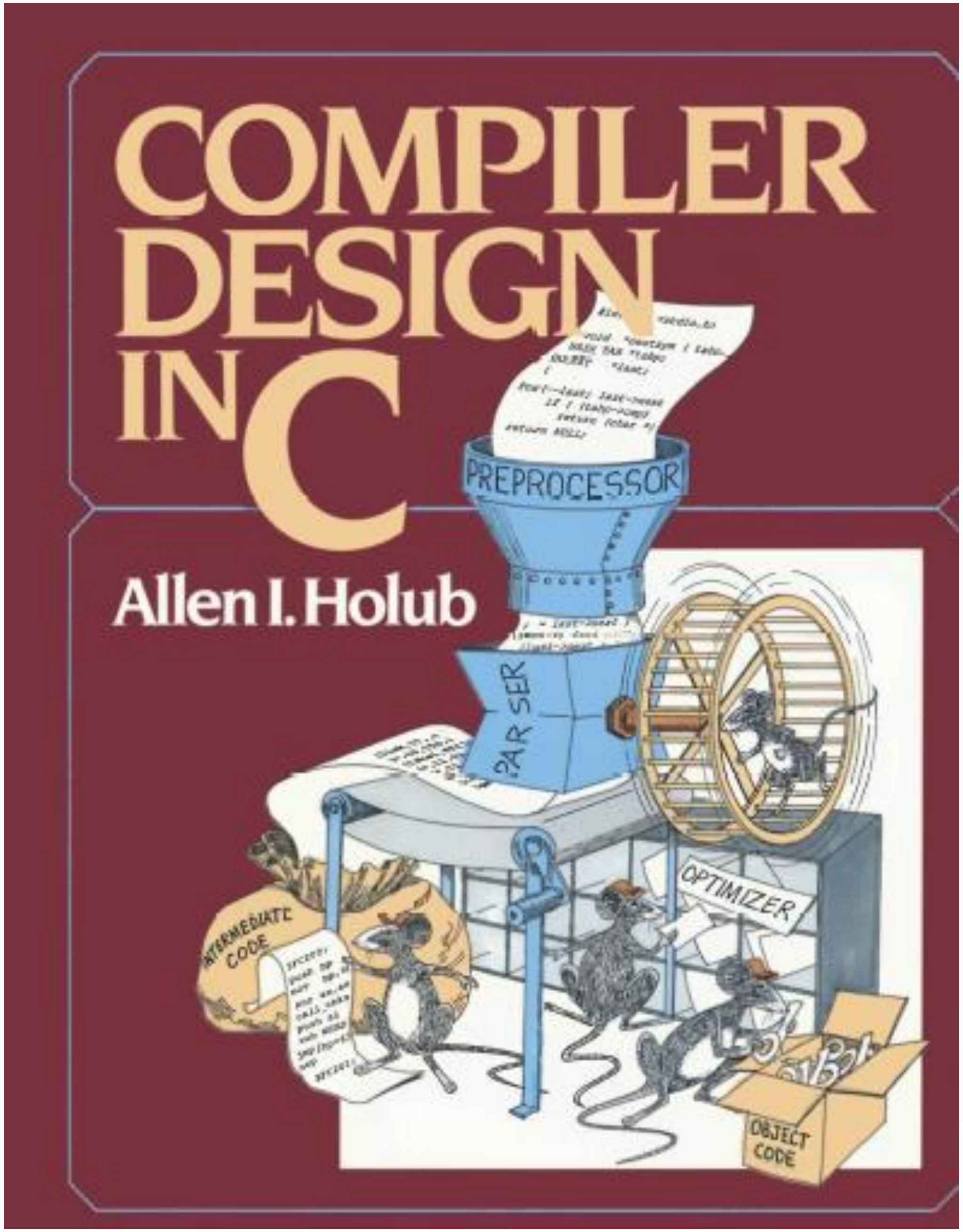
- From one HLL to another HLL
- Programs comparison – Finding out whether logically two programs are doing the same or not.
- Computer Graphics
  - Animation creation language
- Music Synthesis
- Data Structure /Program /Algorithm Visualization
- Games
- ….

The dragon book



- New versions of the book are available.
- Our Library also has several copies.
- PDF is available for this book (2006 version) in Internet.
- Later versions, if you can find please share.


Copyrighted Material
ENGINEERING
A
COMPILER
SECOND EDITION
MK
MORGAN KAUFMANN
Keith D. Cooper & Linda Torczon
Copyrighted Material



COMPILER DESIGN IN C

Allen I. Holub

History of compiler development

1953 IBM develops the 701 EDPM (Electronic Data Processing Machine), the first general purpose computer, built as a “defense calculator” in the Korean War

History of compilers (cont’d)

No high-level languages were available, so all programming was done in machine and assembly language.

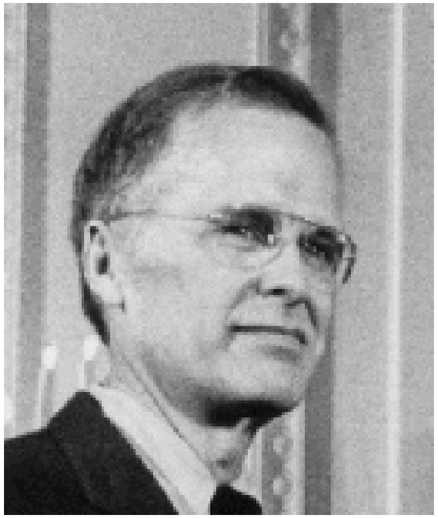
History of compilers (cont’d)

As expensive as these early computers were, most of the money companies spent was for software development, due to the complexities of assembly.

History of compilers (cont'd)

In 1953, John Backus came up with the idea of "speed coding", and developed the first interpreter. Unfortunately, this was 10-20 times slower than programs written in assembly.

He was sure he could do better.



John Backus

History of compilers (cont'd)

In 1954, Backus and his team released a research paper titled "Preliminary Report, Specifications for the IBM Mathematical FORmula TRANslating System, FORTRAN."

The initial release of FORTRAN was in 1956, totaling 25,000 lines of assembly code. Compiled programs run almost as fast as handwritten assembly!

History of compilers (cont’d)

Projects that had taken two weeks to write now took only 2 hours. By 1958 more than half of all software was written in FORTRAN.

Why study compilers?

- You may never write a commercial compiler, but that's not why we study compilers. We study compiler construction for the following reasons:

Why study compilers? **(cont’d)**

1) Writing a compiler gives a student experience with **large-scale** applications development. Your compiler program may be the largest program you write as a student. Experience working with really **big data structures** and complex interactions between **algorithms**s will help you out on your next big programming project

Why study compilers? **(cont’d)**

2) Compiler writing is one of the shining **triumphs** of CS theory. It is very helpfull in the solutions of different problems.

Why study compilers? **(cont’d)**

3) Compiler writing is a basic element of **programming language research**. Many language researchers write compilers for the languages they design.

Why study compilers? **(cont’d)**

4) **Many applications have similar properties to one or more phases of a compiler**, and compiler expertise and tools can help an application programmer working on other projects besides compilers

Cousins Of The Compiler

1) Preprocessor.
2) Assembler
3) Linker and loader.

Why new languages

- Applications demand
  - Business -- COBOL, SQL, ..
  - Scientific -- C++/Python/Matlab
  - Logic – PROLOG, LISP
  - Internet – Java
  - Systems Programming – C/C++
- Old languages have inertia
  - Difficult to enhance old one
  - Training programmers is difficult.

Why new languages

- New language that look similar to older ones are often created.
  - Java is very much like C++
  - Learning is easier

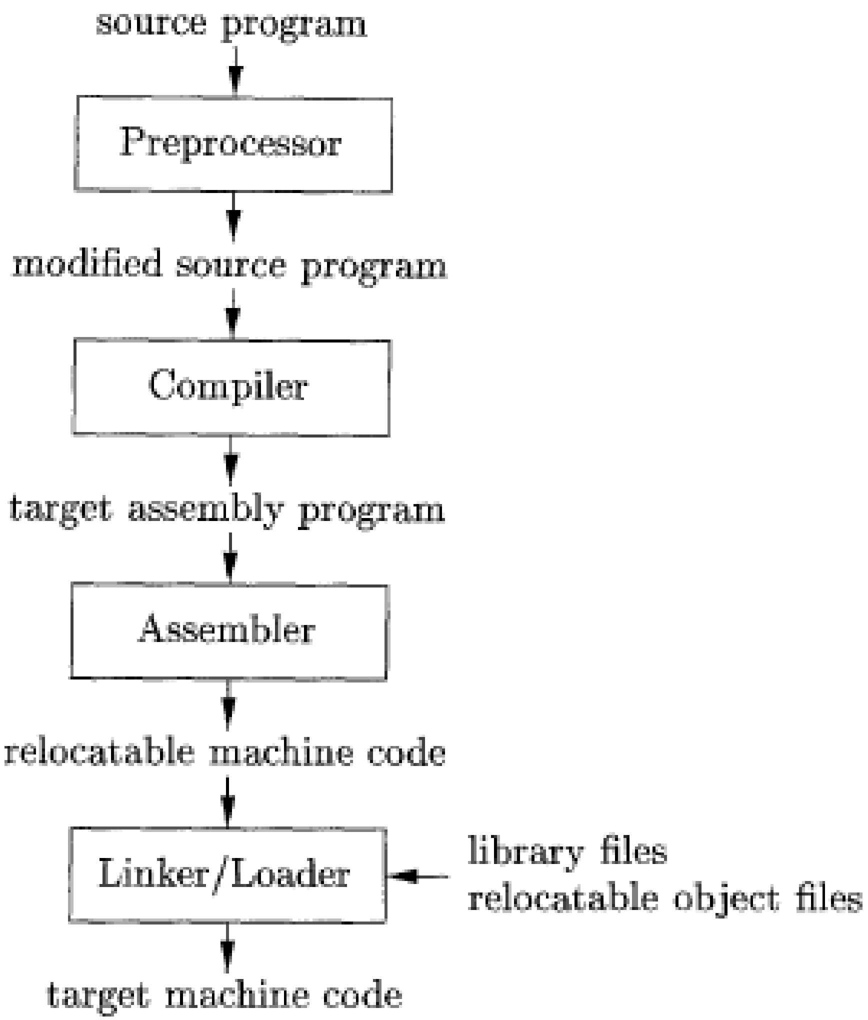


Figure 1.5: A language-processing system

Preprocessor: Expands shorthands/macros, inserts program code written in another file. Can create several versions of source program, perhaps, one very useful for debugging, the other may run faster without any i/o.

Modern Compilers

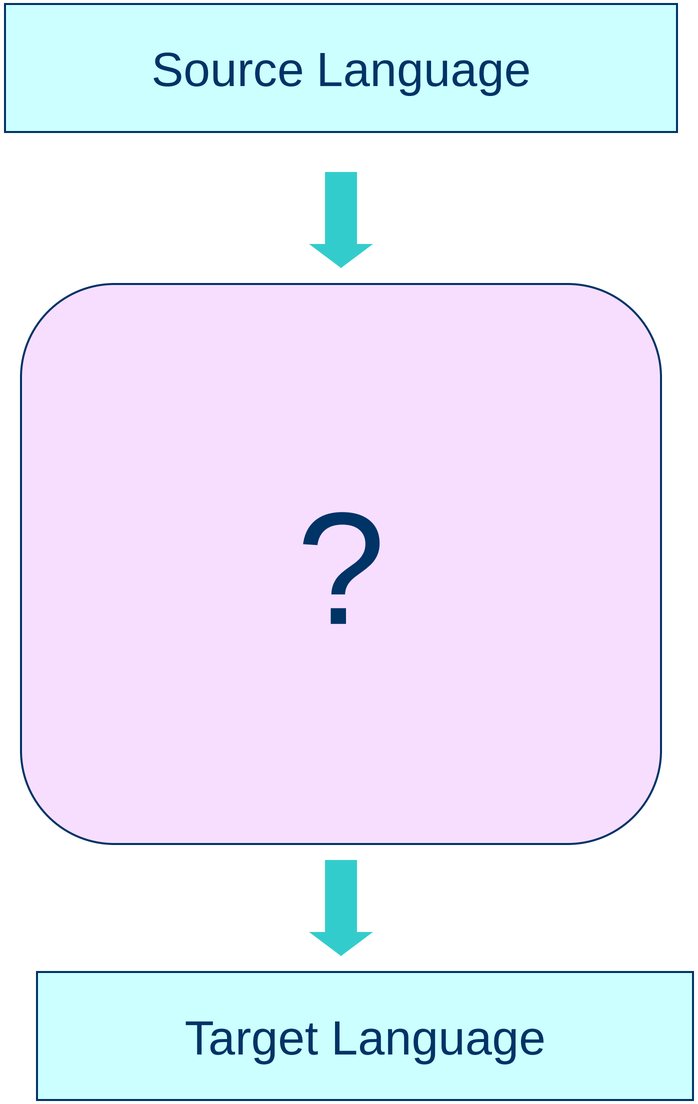
Compilers have not changed a great deal since the days of Backus. They still consist of two main components:

The **FRONT-END** reads in the program in the source languages, makes sense of it, and stores it in an internal representation…

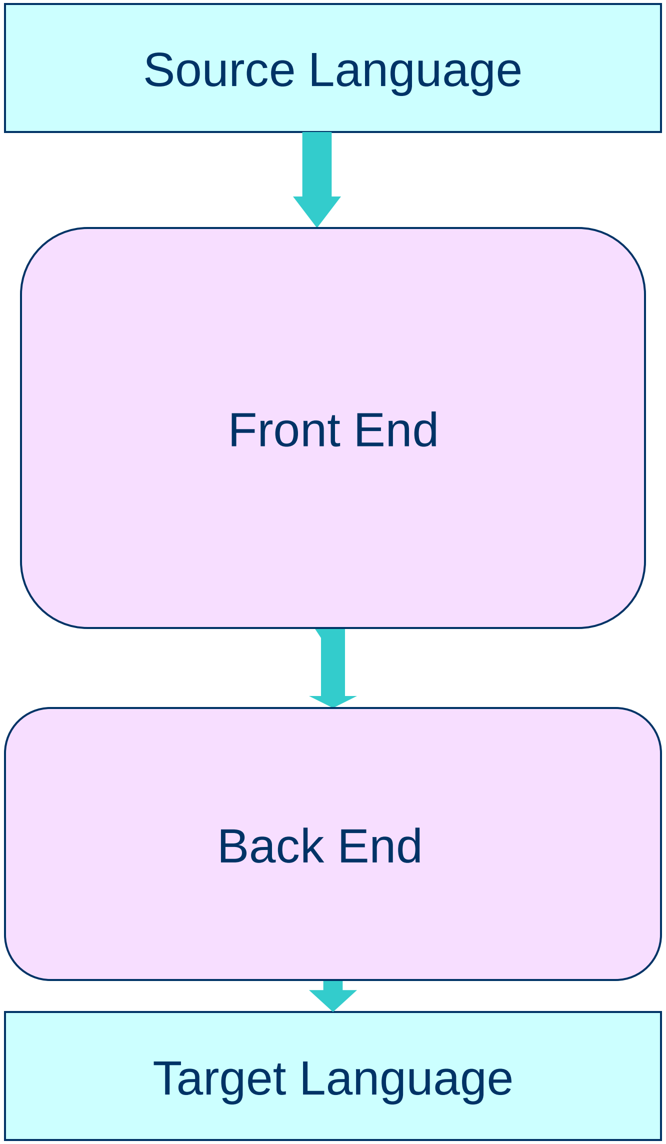
Modern Compilers(cont'd)

The **BACK-END**, which converts the internal representation into the target language, perhaps with optimizations. The target language used is typically an assembly language.

Structure of a Compiler



Structure of a Compiler

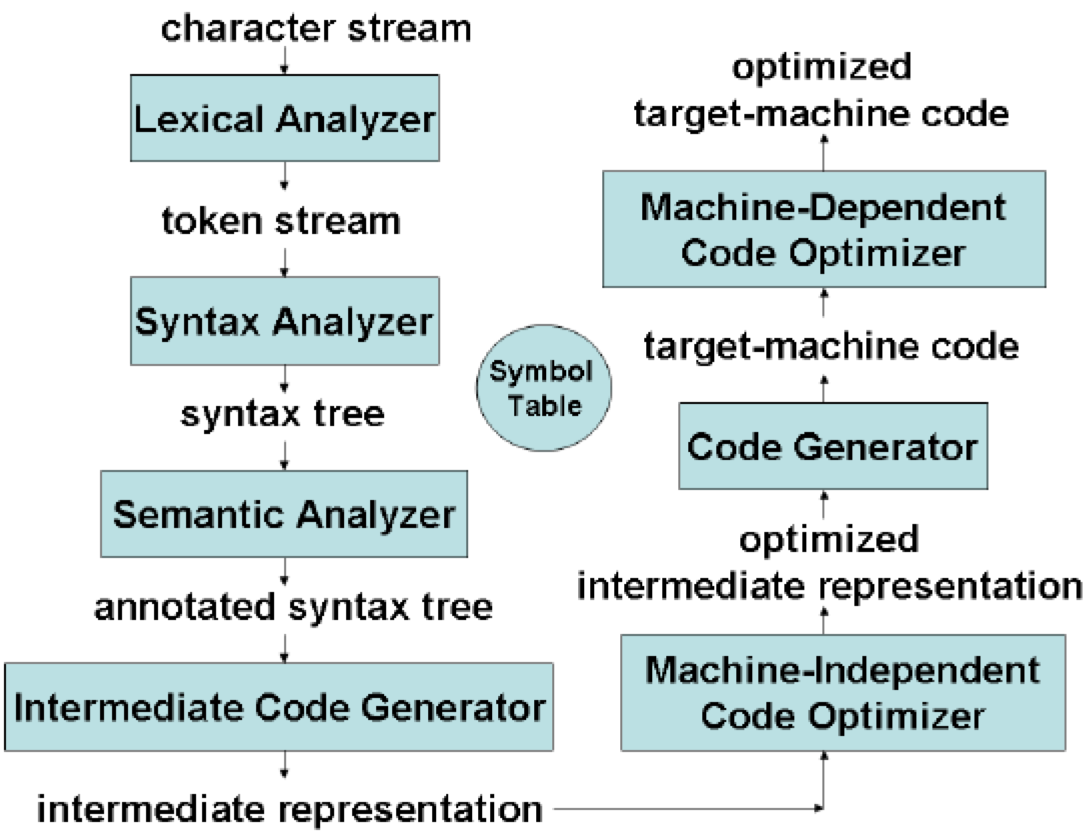


Let us open the black-box called compiler

- Analysis part – Front end – upto inter. code
  - Find syntactically *ill-formed* statements; semantically *unsound* statements; informs the user about errors.
  - Creates symbol table.
- Synthesis part– Back end – From inter. Code
  - Uses symbol table.
  - Goes via optimization (in two stages, m/c independent, m/c dependent) to code generation.

Compiler Overview

**Analysis/
Front end**



character stream

Lexical Analyzer

token stream

Syntax Analyzer

syntax tree

Semantic Analyzer

annotated syntax tree

Intermediate Code Generator

intermediate representation

Symbol Table

Machine-Independent Code Optimizer

optimized intermediate representation

Code Generator

target-machine code

Machine-Dependent Code Optimizer

optimized target-machine code

**Synthesis/
Back end**

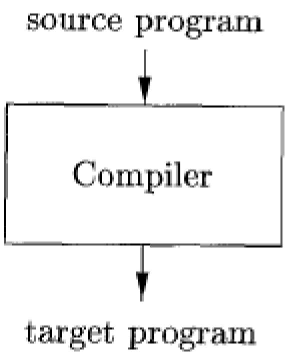


Figure 1.1: A compiler

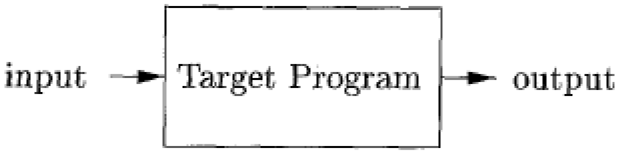


Figure 1.2: Running the target program

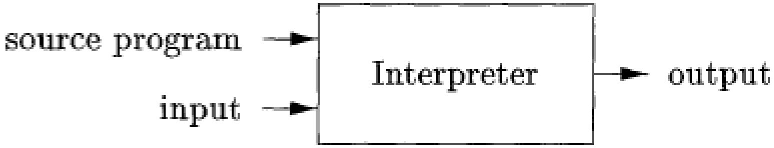
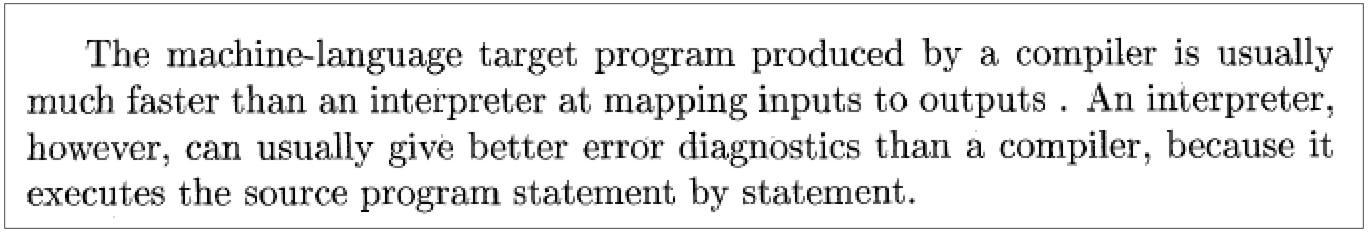


Figure 1.3: An interpreter



The machine-language target program produced by a compiler is usually much faster than an interpreter at mapping inputs to outputs . An interpreter, however, can usually give better error diagnostics than a compiler, because it executes the source program statement by statement.

Hybrid

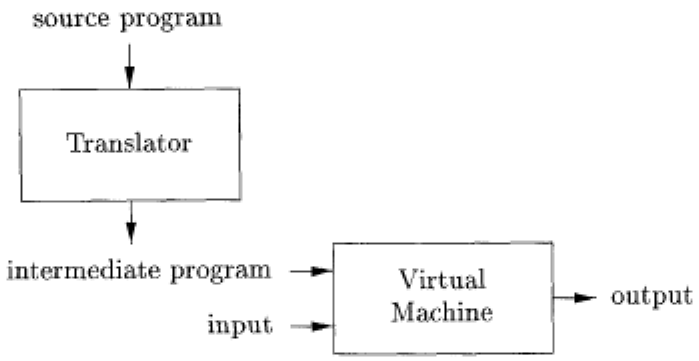
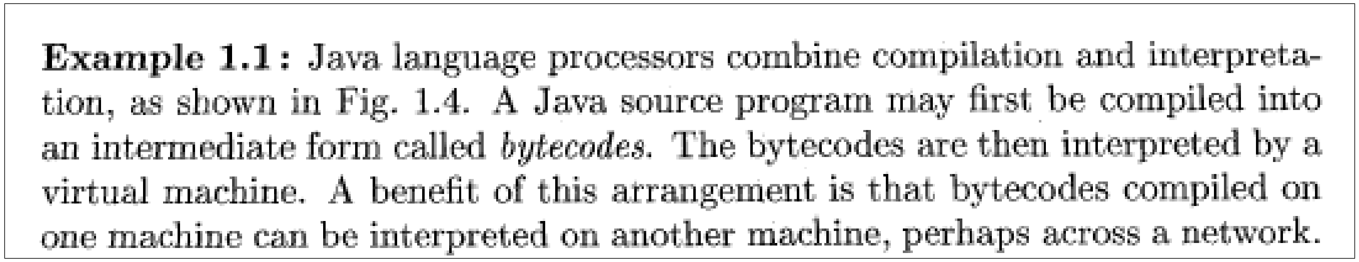


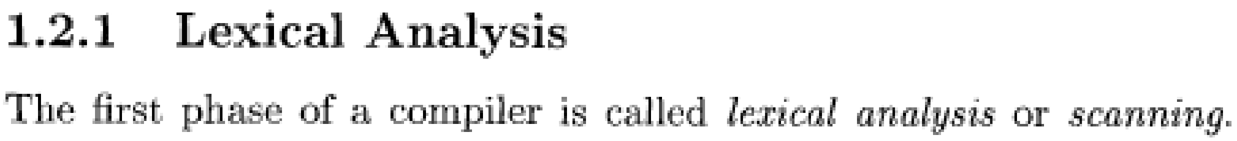
Figure 1.4: A hybrid compiler



Example 1.1 : Java language processors combine compilation and interpretation, as shown in Fig. 1.4. A Java source program may first be compiled into an intermediate form called *bytecodes*. The bytecodes are then interpreted by a virtual machine. A benefit of this arrangement is that bytecodes compiled on one machine can be interpreted on another machine, perhaps across a network.

Compilers and Interpreters

- Compilers generate machine code, whereas interpreters interpret intermediate code
- Interpreters are easier to write and can provide better error messages (symbol table is still available)
- Interpreters are at least 5 times slower than machine code generated by compilers
- Interpreters also require much more memory than machine code generated by compilers
- Examples: Perl, Python, Unix Shell, Java, BASIC, LISP



1.2.1 Lexical Analysis

The first phase of a compiler is called *lexical analysis* or *scanning*.

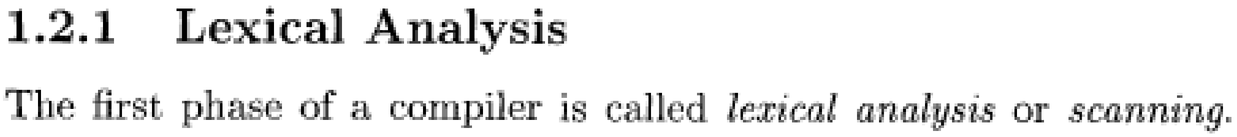
- A program is a sentence (sequence of characters) in a language

```
if (i == j)
    Z = 0;
else
    Z = 1;
```

```
\tif (i == j)\n\t\tz = 0;\n\telse\n\t\tz = 1;
```

A higher level abstration

- Sequence of characters ➔Sequence of tokens
- Token?
  - Class/category of a group of characters (lexeme)
- In English
  - Noun, verb, adjective, …
- In programming language
  - Identifier, keyword, open brace, number, operator, …



1.2.1 Lexical Analysis

The first phase of a compiler is called *lexical analysis* or *scanning.*

- Stream of characters ➔stream of tokens
  - – strings with an identified "meaning"
  - – For each lexeme a token:
    <*token-name, attribute-value*> is produced.
  - – token-name is abstract symbol representing the class of the lexeme. Eg: *id* stands for identifier
  - – attribute-value: points to symbol table entry (eg: an integer that gives the row number of the table)
- FSM (finite state machine) is used.

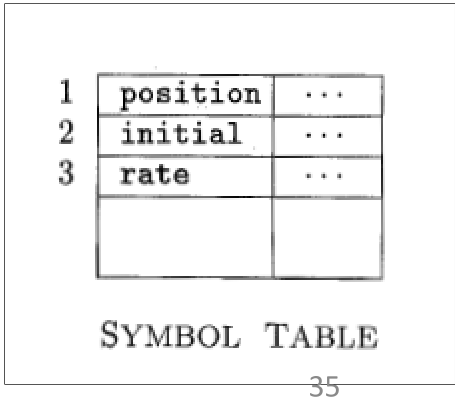
```
position = initial + rate * 60
```

1. position is a lexeme that would be 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. The assignment symbol = is a lexeme that is mapped into the token ⟨=⟩. Since this token needs no attribute-value, we have omitted the second component. We could have used any abstract symbol such as **assign** for the token-name, but for notational convenience we have chosen to use the lexeme itself as the name of the abstract symbol.

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

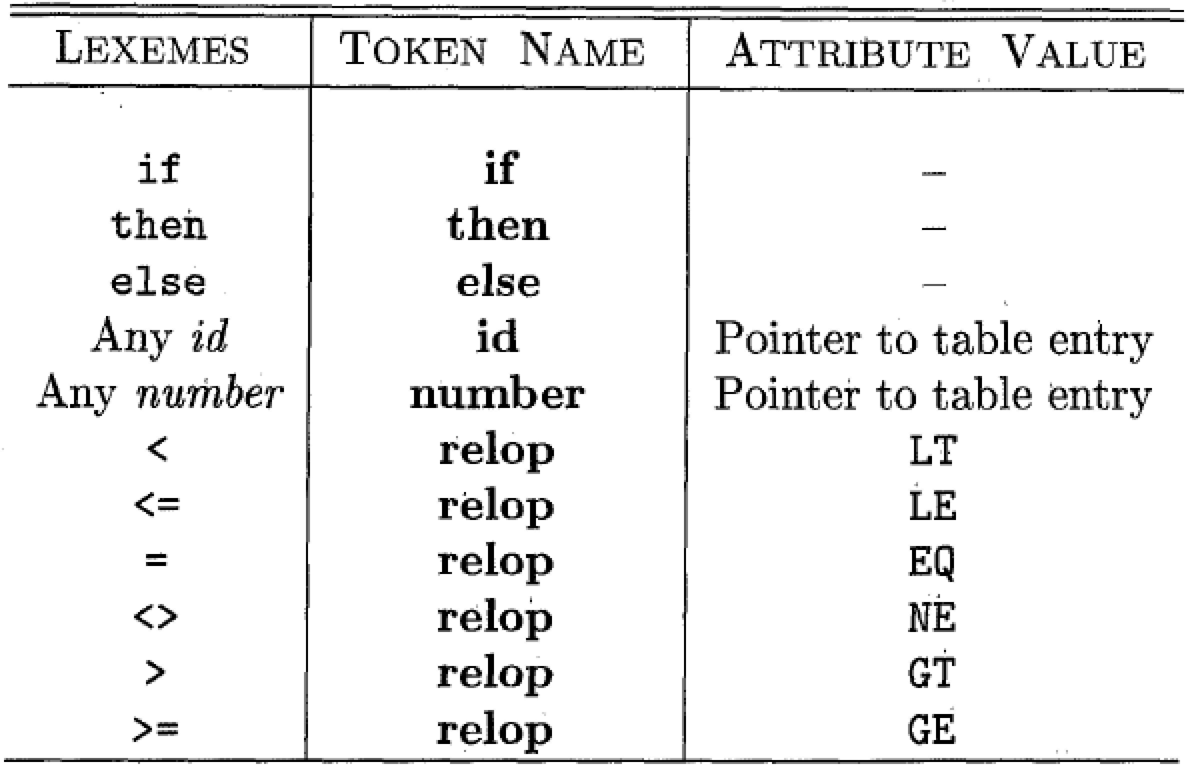
Technically <60> should be <number, 4>



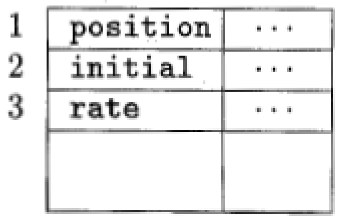
1	position	...
2	initial	...
3	rate	...

SYMBOL TABLE

Regular expression is used to specify a lexeme

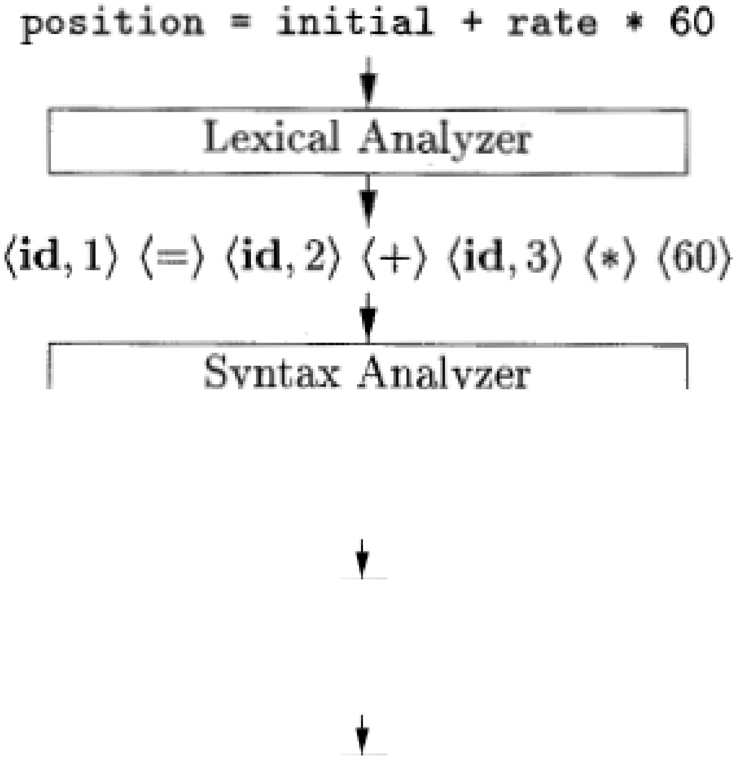


LEXEMES	TOKEN NAME	ATTRIBUTE VALUE
`if`	**if**	–
`then`	**then**	–
`else`	**else**	–
Any *id*	**id**	Pointer to table entry
Any *number*	**number**	Pointer to table entry
<	**relop**	`LT`
<=	**relop**	`LE`
=	**relop**	`EQ`
<>	**relop**	`NE`
>	**relop**	`GT`
>=	**relop**	`GE`



1	position	⋯
2	initial	⋯
3	rate	⋯

SYMBOL TABLE



position = initial + rate * 60

Lexical Analyzer

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

Syntax Analyzer

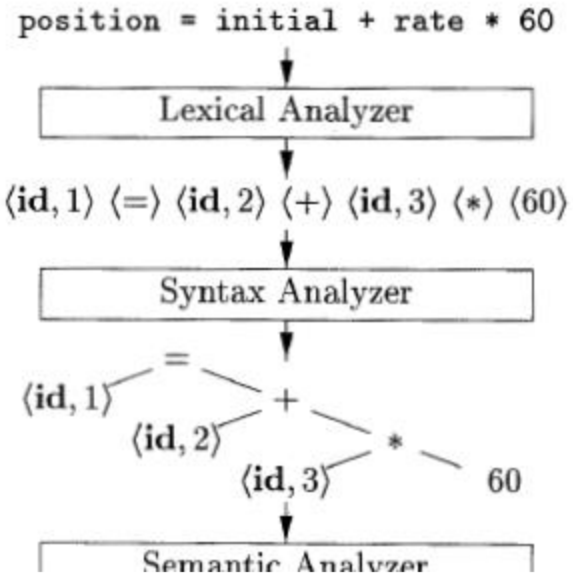
Lexical Analysis

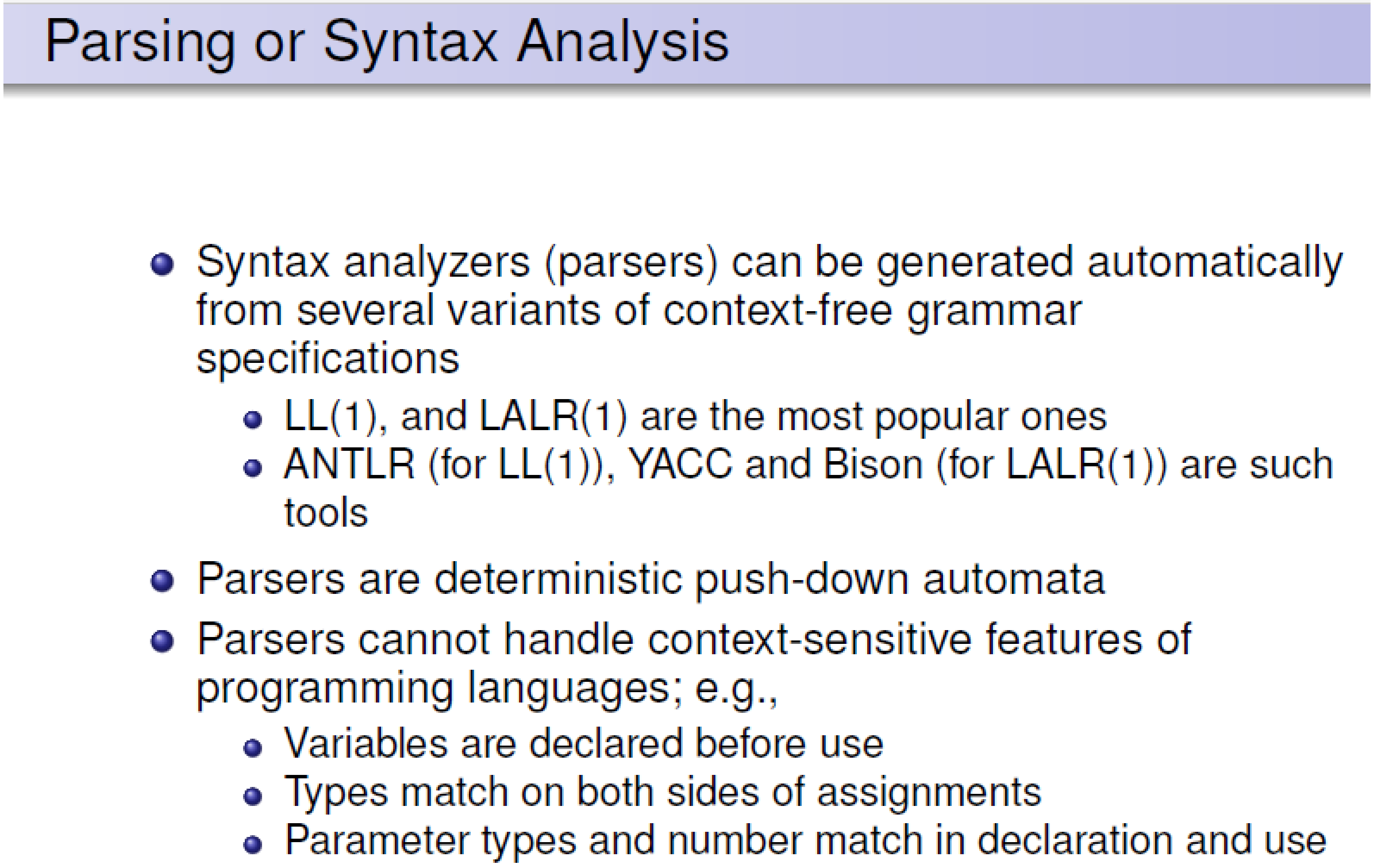
- LA can be generated automatically from regular expression specifications
    - LEX and Flex are two such tools
- LA is a deterministic finite state automaton
- Why is LA separate from parsing?
    - Simplification of design - software engineering reason
    - I/O issues are limited LA alone
    - LA based on finite automata are more efficient to implement than pushdown automata used for parsing (due to stack)

1.2.2 Syntax Analysis

The second phase of the compiler is *syntax analysis* or *parsing*.

- Syntax tree (Parse tree) is created.
- Grammar is used for this purpose.
- The structure inherent in the statement is exposed.
- Multiplication is done first, assignment is done in the last.





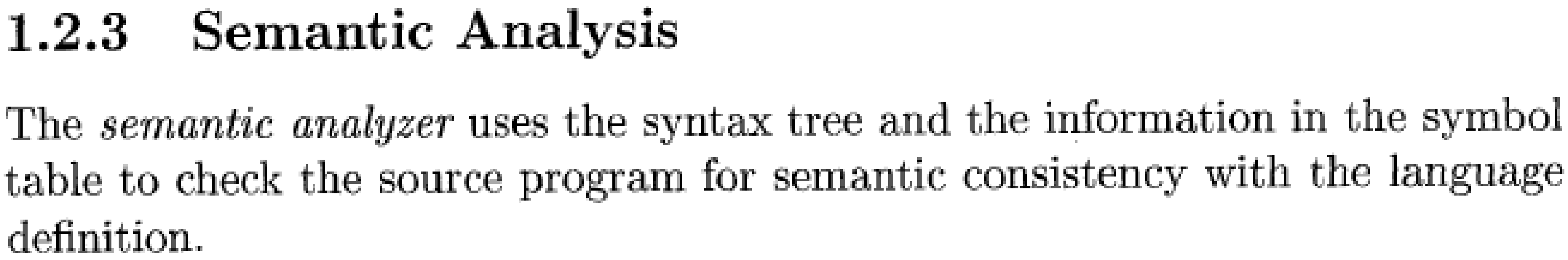
Parsing or Syntax Analysis

- Syntax analyzers (parsers) can be generated automatically from several variants of context-free grammar specifications
  - LL(1), and LALR(1) are the most popular ones
  - ANTLR (for LL(1)), YACC and Bison (for LALR(1)) are such tools
- Parsers are deterministic push-down automata
- Parsers cannot handle context-sensitive features of programming languages; e.g.,
  - Variables are declared before use
  - Types match on both sides of assignments
  - Parameter types and number match in declaration and use

ANTLR **=** ANother Tool for Language Recognition

YACC = Yet Another Compiler Compiler

- Both ANTLR and YACC are tools that take a description of a language (usually a programming language) and produce a new program that will recognize the language that is described.
- ANTLR generates a top down parser, whereas YACC generates a bottom up one.
- YACC generates C programs, and ANTLR generates Java, Python, and a few other languages.



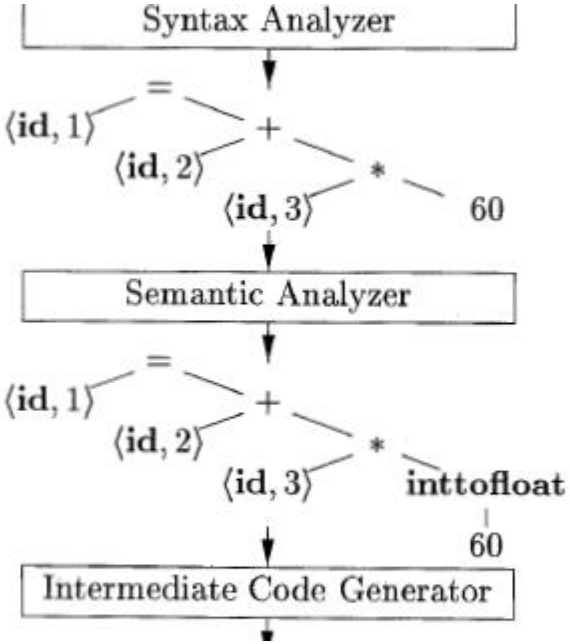
1.2.3 Semantic Analysis

The *semantic analyzer* uses the syntax tree and the information in the symbol table to check the source program for semantic consistency with the language definition.

Semantic Analysis – Errors Caught

- Following are some of tasks (example errors) done at semantic analysis:
  - Scope resolution
    - Multiple declaration of same variable in a scope.
    - Accessing an out of scope variable.
    - Undeclared variable.
  - Type checking
    - int a = “value”; /*No parsing error; type mismatch*/
    - Doing Automatic Type Conversion (Coercion)

Coercion (type conversion) is done in this example .



Semantic Analysis – Errors Caught

- Array-bound checking
- Reserved identifier misuse.
- Actual and formal parameter mismatch.

Attribute Grammar

- E → E + T is a production without any meaning.

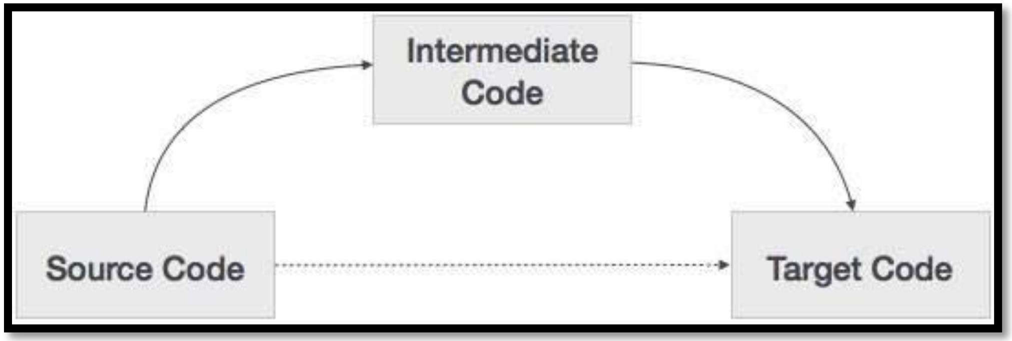
Attribute Grammar augments meaning

- E → E + T { E.value = E.value + T.value }

semantic rules specify how the grammar should be interpreted are augmented to form attribute grammars.

Intermediate Code

- A source code can directly be translated into its target machine code,
  - then why at all we need to translate the source code into an intermediate code which is then translated to its target code?
- Let us see the reasons why we need an intermediate code.



1.2.4 Intermediate Code Generation

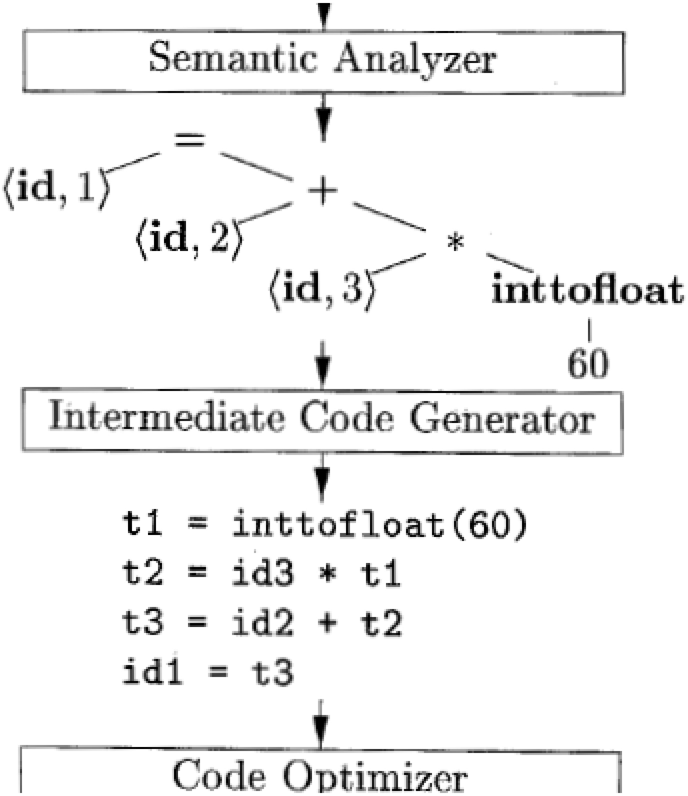
- While generating machine code directly from source code is possible, it entails two problems
  - With *m* languages and *n* target machines, we need to write $m \times n$ compilers
  - The code optimizer which is one of the largest and very-difficult-to-write components of any compiler cannot be reused
- By converting source code to an intermediate code, a machine-independent code optimizer may be written
- Intermediate code must be easy to produce and easy to translate to machine code
  - A sort of universal assembly language
  - Should not contain any machine-specific parameters (registers, addresses, etc.)

Two reasons for having intermediate code.

- $m + n$ compilers only; not $m \times n$ compilers.
- m/c independent code optimization constitutes 25 to 40% of the compiler (huge component).
- m/c indep. code opt. can be done independent of HLL and independent of m/c.
  - A researcher can work only on this and give a better optimizer. This can be replaced with the existing one across all compilers.

Third reason

- Writing interpreter for a standard and simple enough intermediate code is an easy task.



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

an intermediate form called *three-address code*

Different Types of Intermediate Code

- The type of intermediate code deployed is based on the application
- Quadruples, triples, indirect triples, abstract syntax trees are the classical forms used for machine-independent optimizations and machine code generation
- Static Single Assignment form (SSA) is a recent form and enables more effective optimizations
    - Conditional constant propagation and global value numbering are more effective on SSA
- Program Dependence Graph (PDG) is useful in automatic parallelization, instruction scheduling, and software pipelining

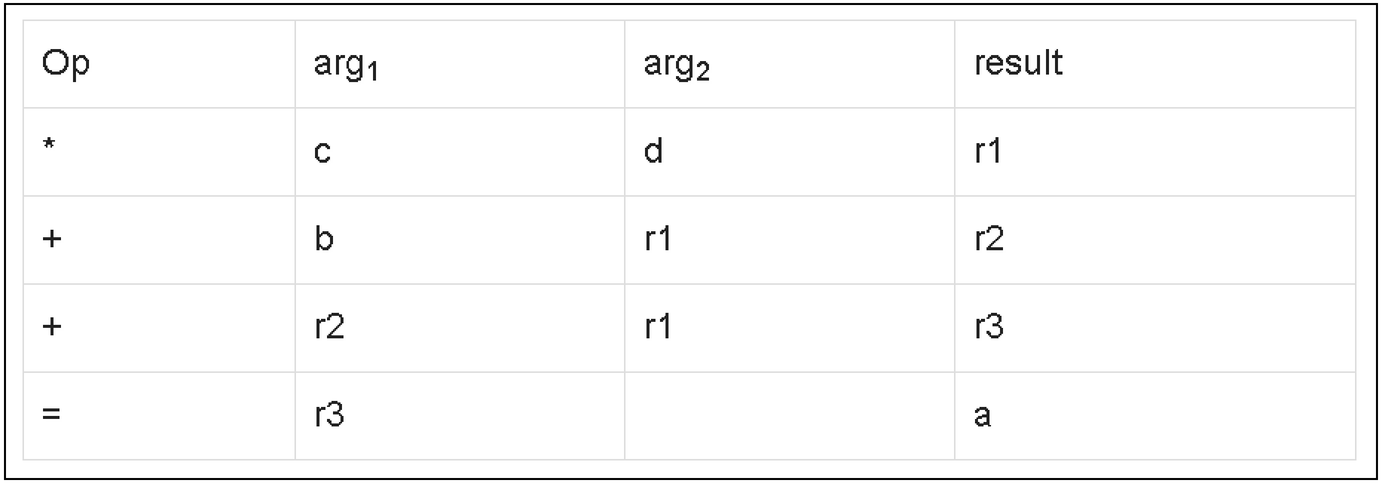
Three address code - Eg

- a = b + c * d;

```
r1 = c * d;
r2 = b + r1;
a = r2
```

- A three-address code can be represented in two forms : quadruples and triples.
  - Quadruples

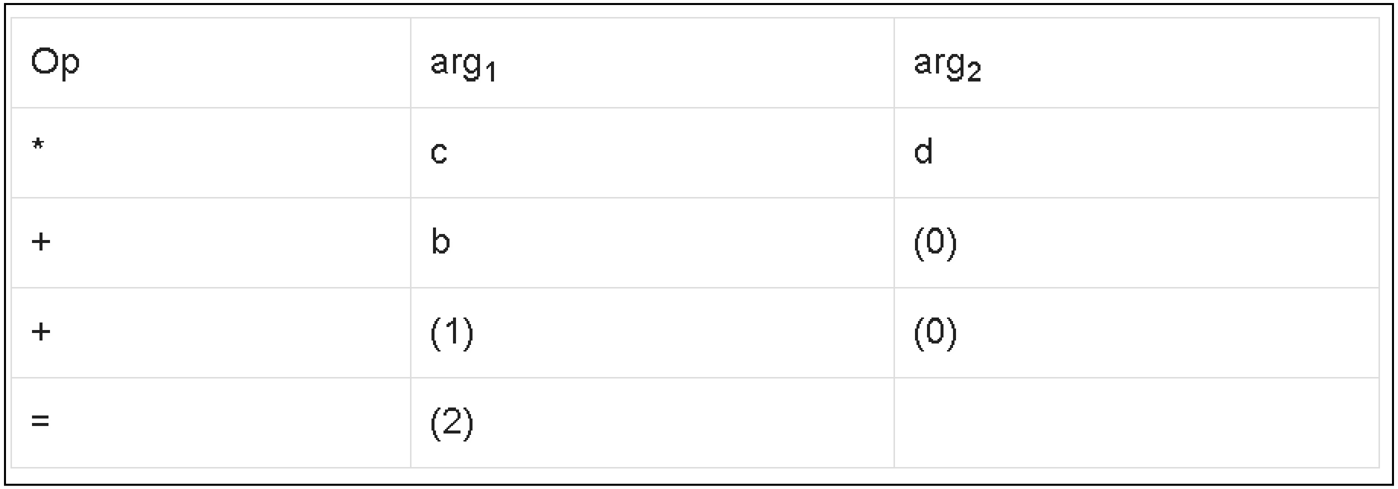
```
r1 = c * d;
r2 = b + r1;
a = r2
```



Op	arg_1	arg_2	result
*	c	d	r1
+	b	r1	r2
+	r2	r1	r3
=	r3		a

- A three-address code can be represented in two forms : quadruples and triples.
  - Triples

```
r1 = c * d;
r2 = b + r1;
a = r2
```



Op	arg_1	arg_2
*	c	d
+	b	(0)
+	(1)	(0)
=	(2)	

- Intermediate code can be either language specific
  - e.g., Byte Code for Java,   or
- language independent
  - three-address code.

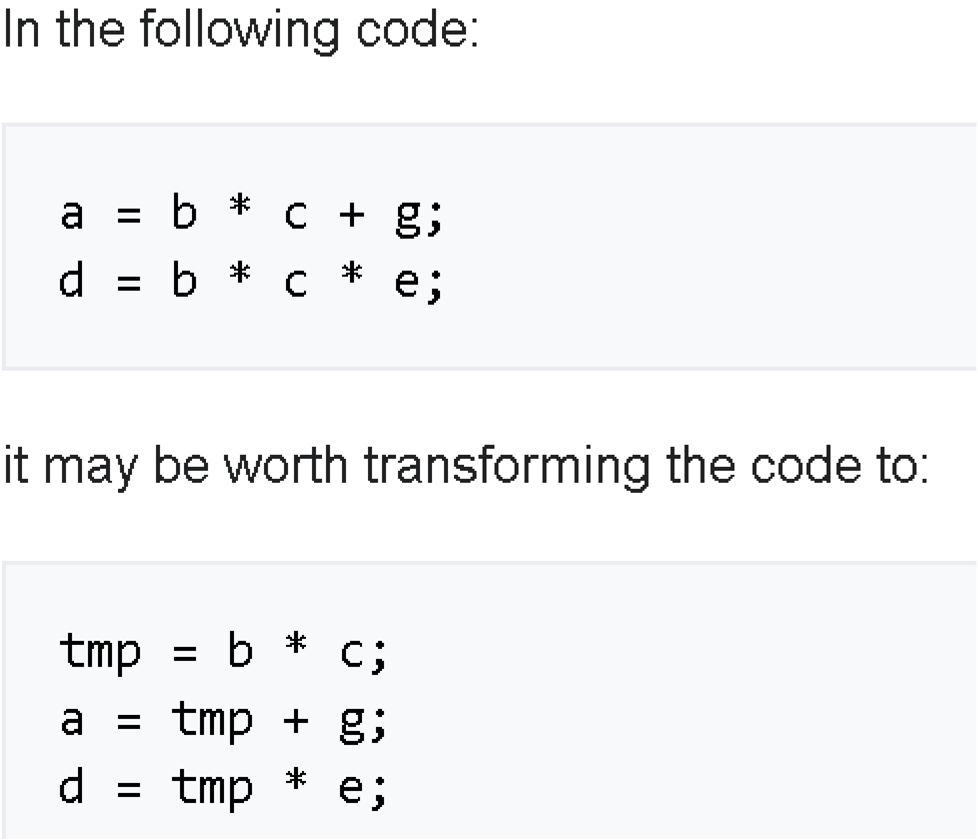
Machine-independent Code Optimization

- Intermediate code generation process introduces many inefficiencies
  - Extra copies of variables, using variables instead of constants, repeated evaluation of expressions, etc.
- Code optimization removes such inefficiencies and improves code
- Improvement may be time, space, or power consumption
- It changes the structure of programs, sometimes of beyond recognition
  - Inlines functions, unrolls loops, eliminates some programmer-defined variables, etc.
- Code optimization consists of a bunch of heuristics and percentage of improvement depends on programs (may be zero also)

Examples of Machine-Independant Optimizations

- Common sub-expression elimination
- Copy propagation
- Loop invariant code motion
- Partial redundancy elimination
- Induction variable elimination and strength reduction
- Code opimization needs information about the program
  - which expressions are being recomputed in a function?
  - which definitions reach a point?
- All such information is gathered through data-flow analysis

Common Subexpression Elimination



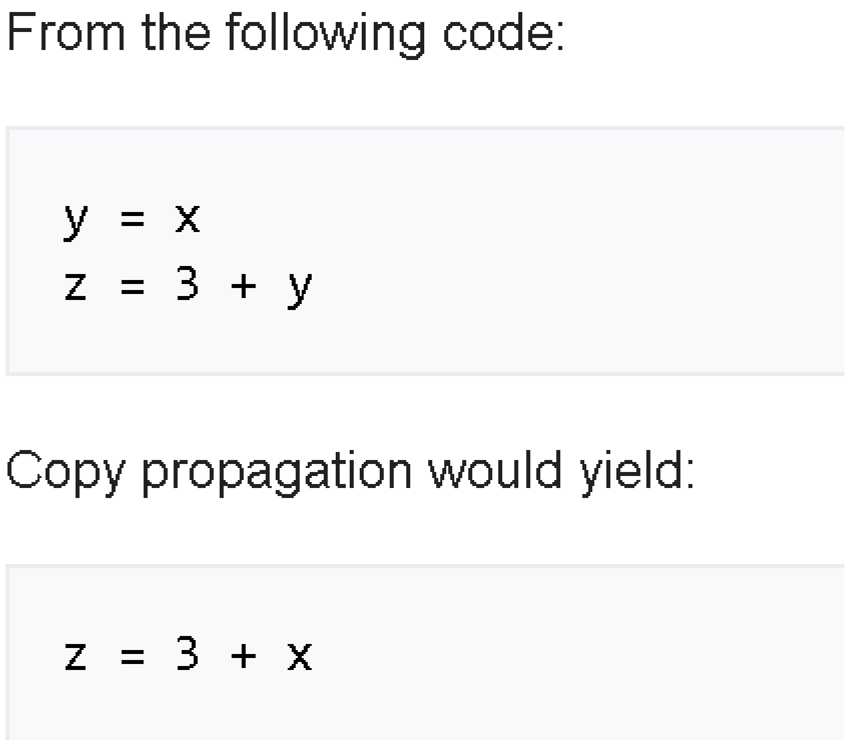
In the following code:

```
a = b * c + g;
d = b * c * e;
```

it may be worth transforming the code to:

```
tmp = b * c;
a = tmp + g;
d = tmp * e;
```

Copy Propagation



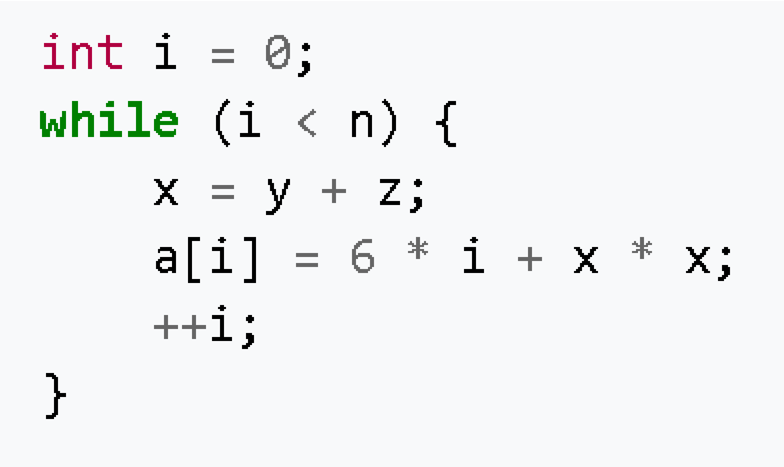
From the following code:

```
y = x
z = 3 + y
```

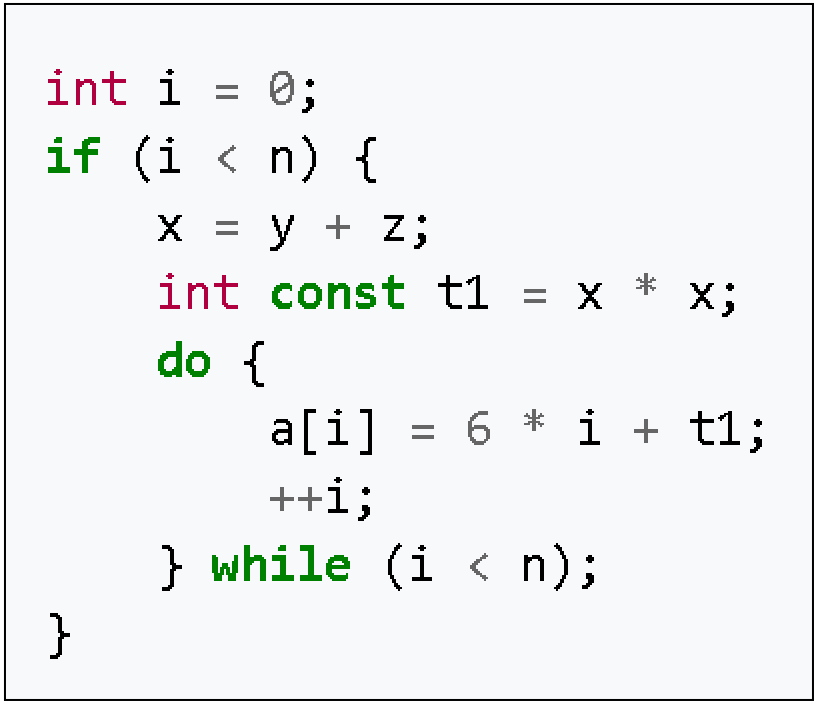
Copy propagation would yield:

```
z = 3 + x
```

Loop Invariant Code Motion



```
int i = 0;
while (i < n) {
    x = y + z;
    a[i] = 6 * i + x * x;
    ++i;
}
```



```
int i = 0;
if (i < n) {
    x = y + z;
    int const t1 = x * x;
    do {
        a[i] = 6 * i + t1;
        ++i;
    } while (i < n);
}
```

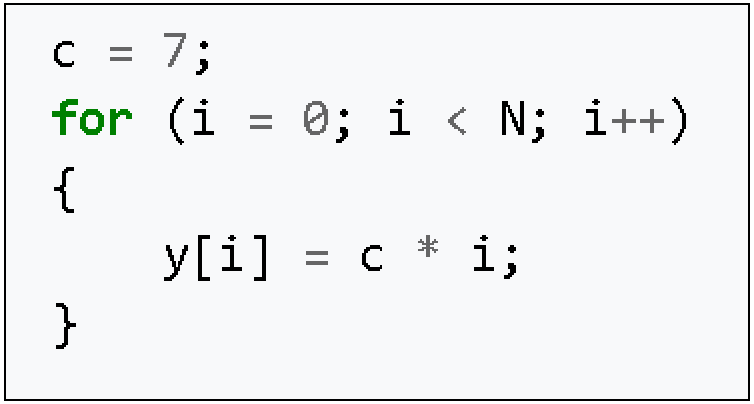
We could have simply moved $x = y + z$:

But, this may lead to errors; similarly $x * x$

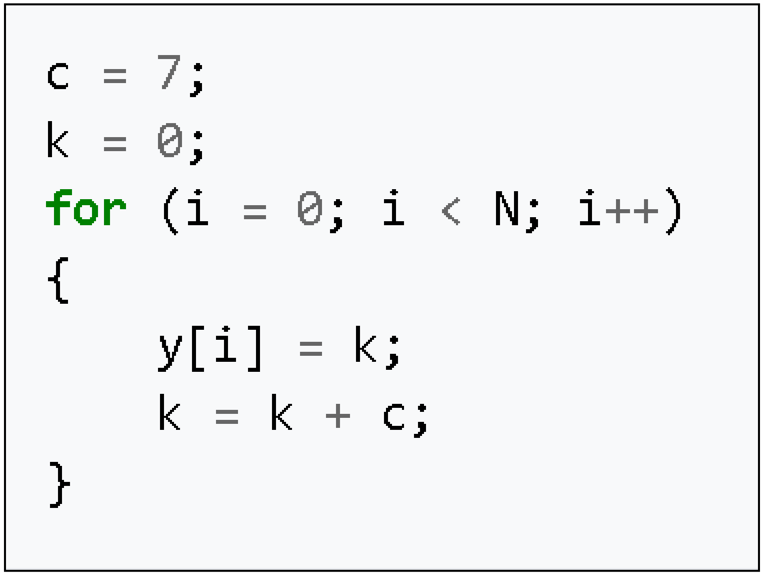
Strength Reduction

- Strength reduction looks for expressions involving a loop invariant and an induction variable.
- Some of those expressions can be simplified.
- For example, the multiplication of loop invariant c and induction variable i

- expensive operations are replaced with equivalent but less expensive operations
  - replacing a multiplication within a loop with an addition
  - replacing an exponentiation within a loop with a multiplication



```
c = 7;
for (i = 0; i < N; i++)
{
    y[i] = c * i;
}
```

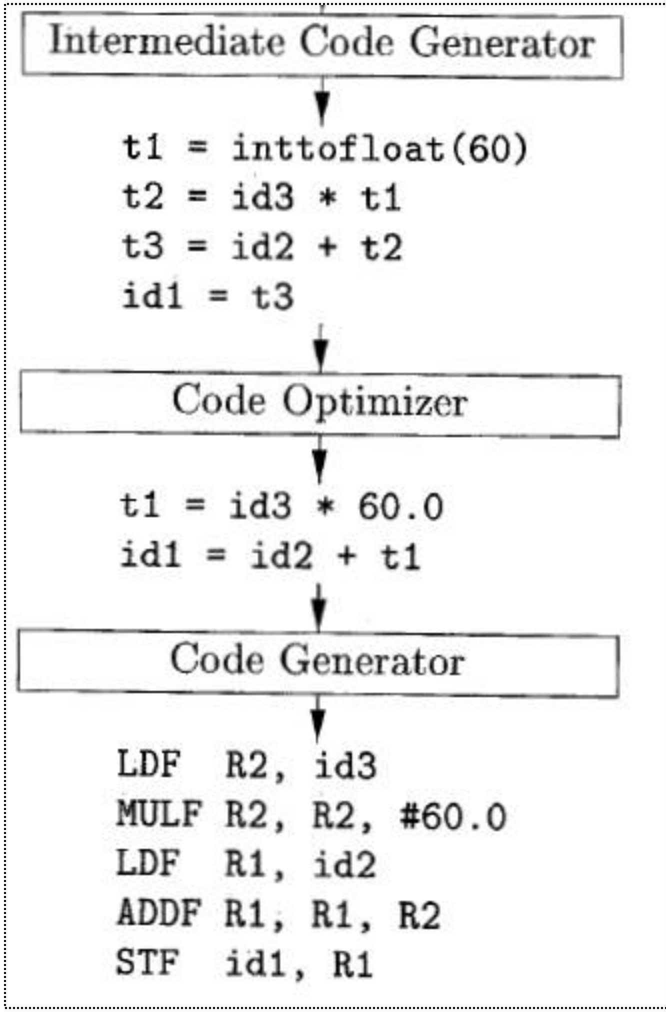


```
c = 7;
k = 0;
for (i = 0; i < N; i++)
{
    y[i] = k;
    k = k + c;
}
```

1.2.5 Code Optimization

The machine-independent code-optimization phase attempts to improve the intermediate code so that better target code will result.

1.2.6 Code Generation



Intermediate Code Generator

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

Code Optimizer

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

Code Generator

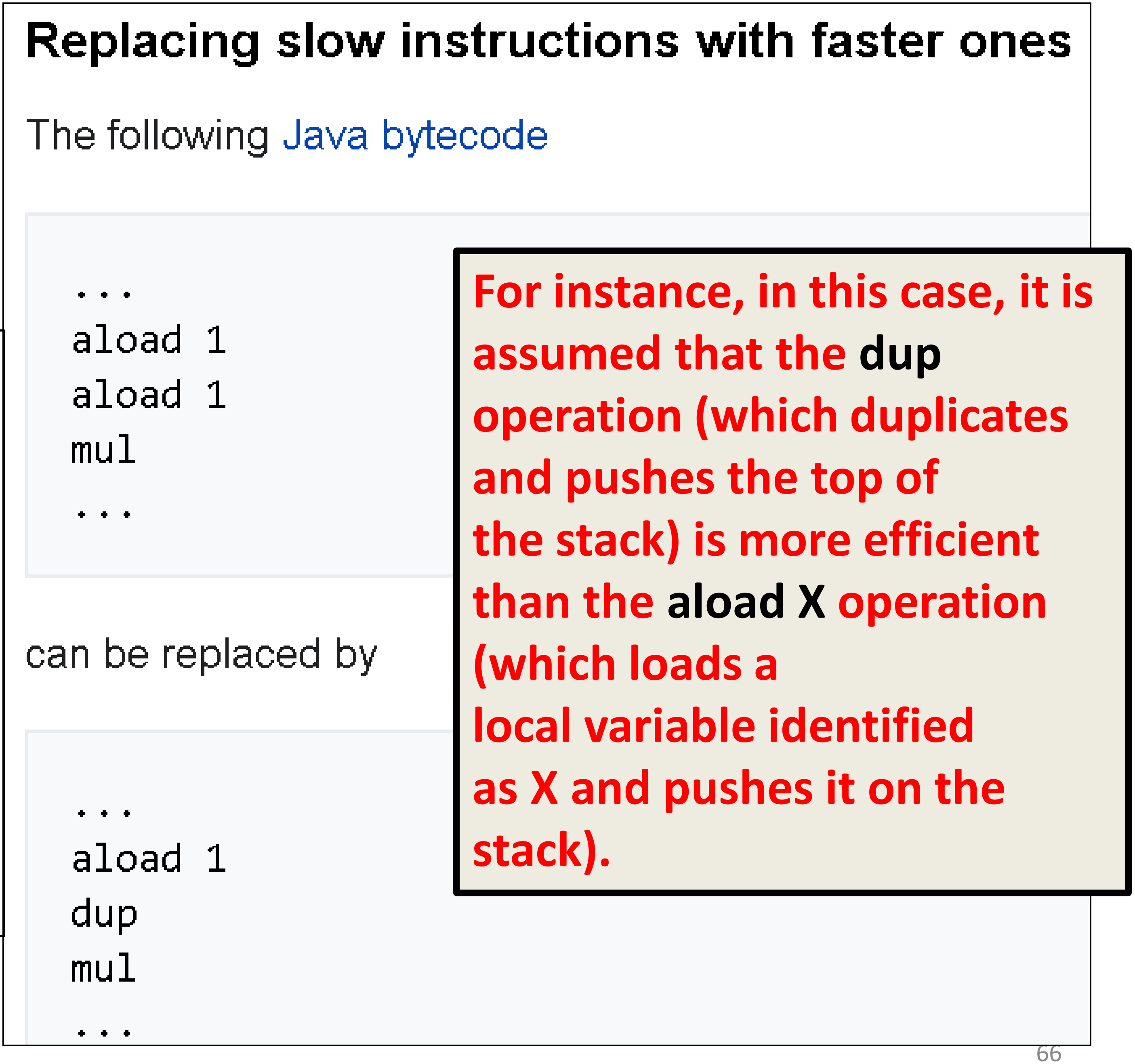
```
LDF  R2, id3
MULF R2, R2, #60.0
LDF  R1, id2
ADDF R1, R1, R2
STF  id1, R1
```

Machine-Dependent Optimizations

- Peephole optimizations
  - Analyze sequence of instructions in a small window (*peephole*) and using preset patterns, replace them with a more efficient sequence
  - Redundant instruction elimination
    e.g., replace the sequence [LD A,R1][ST R1,A] by [LD A,R1]
  - Eliminate "jump to jump" instructions
  - Use machine idioms (use INC instead of LD and ADD)
- Instruction scheduling (reordering) to eliminate pipeline interlocks and to increase parallelism
- Trace scheduling to increase the size of basic blocks and increase parallelism
- Software pipelining to increase parallelism in loops

Peephole optimization

Peephole optimization involves changing the small set of instructions to an equivalent set that has better performance.



Replacing slow instructions with faster ones

The following Java bytecode

```
...
aload 1
aload 1
mul
...
```

can be replaced by

```
...
aload 1
dup
mul
...
```

For instance, in this case, it is assumed that the dup operation (which duplicates and pushes the top of the stack) is more efficient than the aload X operation (which loads a local variable identified as X and pushes it on the stack).

Ref: 1st chapter of the Dragon book.

- Next
  - Lexical Analysis…