Course Objectives

To learn the process of translating a modern high-level language to executable code.

- Learn the fundamental techniques from lectures, text book and exercises from the book.
- Apply these techniques in practice to construct *a fully working compiler* for a non-trivial subset of Java called "`Decaf`".

In the end, you should be able to compile small Java-like programs with your compiler, and see it actually work!

What is a Compiler?

- Programming problems are easier to solve in *high-level languages*
  - High-level languages are closer to the problem domain
  - E.g. Java, Python, SQL, Tcl/Tk, ...
- Solutions have to be executed by a machine
  - Instructions to a machine are specified in a language that reflects to the cycle-by-cycle working of a processor
- **Compilers** are the bridges:
  - Software that *translates* programs written in high-level languages to efficient executable code.

An Example

```
int gcd(int m, int n)
{
  if (m == 0)
      return n;
  else if (m > n)
    return gcd(n, m);
  else
    return gcd(n%m, m);
}
```

```
gcd:
        pushl %ebp
        movl %esp,%ebp
        cmpl $0,8(%ebp)
        jne .L2
        movl 12(%ebp),%eax
        jmp .L1
        .align 16
        jmp .L3
        .align 16
.L2:
        movl 8(%ebp),%eax
        cmpl %eax,12(%ebp)
        jge .L4
        movl 8(%ebp),%eax
        pushl %eax
...
```


Example (contd.)

```
gcd:
        pushl %ebp
        movl %esp,%ebp
        pushl %esi
        pushl %ebx
        movl 8(%ebp),%esi
        movl 12(%ebp),%ebx
.L11:
        testl %esi,%esi
        jne .L8
        movl %ebx,%eax
        jmp .L13
        .align 16
.L8:
        cmpl %ebx,%esi
        jg .L14
        movl %ebx,%eax
        cltd
        idivl %esi
        movl %edx,%ebx
.L14:
        movl %esi,%ecx
        movl %ebx,%esi
        movl %ecx,%ebx
        jmp .L11
        .align 16
.L13:
        leal -8(%ebp),%esp
        popl %ebx
        popl %esi
        movl %ebp,%esp
        popl %ebp
        ret
```

Requirements

- In order to translate statements in a language, one needs to understand both
  - the *structure* of the language: the way "sentences" are constructed in the language, and
  - the *meaning* of the language: what each "sentence" stands for.
- Terminology:
  - Structure ≡ **Syntax**
  - Meaning ≡ **Semantics**

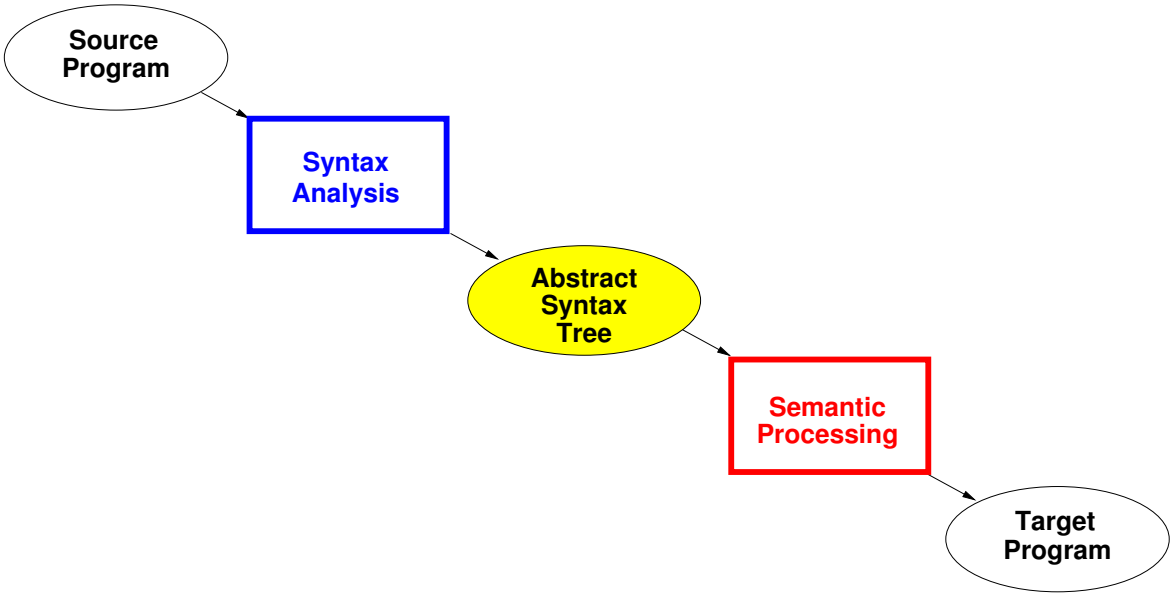
Translation Strategy

Classic Software Engineering Problem

- **Objective:** Translate a program in a high level language into *efficient* executable code.
- **Strategy:** Divide translation process into a series of phases
  Each phase manages some particular aspect of translation.

Interfaces between phases governed by specific intermediate forms.

Translation Process



Translation Steps

- **Syntax Analysis Phase:** Recognizes "sentences" in the program using the *syntax* of the language
- **Semantic Analysis Phase:** Infers information about the program using the *semantics* of the language
- **Intermediate Code Generation Phase:** Generates "abstract" code based on the syntactic structure of the program and the semantic information from Phase 2.
- **Optimization Phase:** Refines the generated code using a series of *optimizing* transformations.
- **Final Code Generation Phase:** Translates the abstract intermediate code into specific machine instructions.

Structure of a Compiler: an Analogy

Syntax-Directed Translation: the *structure* (syntax) of a sentence in a language is used to give it a *meaning* (semantics).

- Die CD-ROM Informatikunterricht liefert alle wichtigen Programme und Bücher, die für den Unterricht oder zur Bearbeitung von Hausaufgaben notwendig sind.
- Green wire connect after first cut white also red wire.
- He sailed the coffee out of the leaf.
- This sentence has four words.

Syntax

Defining and Recognizing Sentences in a Language

- Layered approach
- Alphabet: defines allowed symbols
- Lexical Structure: defines allowed words
- Syntactic Structure: defines allowed sentences

We will later associate *meaning* with sentences (semantics) based on their syntactic structure.

Formal Language Specification

Solid theoretical results applied to a practical problem.

- Declarative vs. Operational Notations
- Declarative notation is used to define a language
  - Defines precisely the set of allowed objects (words/sentences)
  - Examples: Regular expressions, Grammars.
- Operational notation is used to recognize statements in a language
  - Defines an algorithm for determining whether or not a given word/sentence is in the language
  - Example: Automata
- Results from theory on converting between the two notations.

Formal Languages

A *language* is a set of strings over a set of symbols.

- The set of symbols of a language is called its alphabet (usually denoted by Σ.
- Each string in the language is called a sentence.
- Parts of sentences are called phrases.

Context-Free Grammars

A well-studied notation for defining formal languages.

- A Context Free Grammar (CFG, or "grammar" unless otherwise qualified) is defined over an alphabet, called terminal symbols.
- A CFG is defined by a set of productions.
- Each production is of the form

$$X \longrightarrow \beta$$

  where
  - X is a *single non-terminal symbol* representing a set of phrases in the language, and
  - β is a *sequence of terminal and non-terminal symbols*
- Example:

$$\mathit{Stmt} \longrightarrow \texttt{while}\ \mathit{Expr}\ \texttt{do}\ \mathit{Stmt}$$

- A unique non-terminal, called the start symbol, represents the set of all sentences in the language.
- The language defined by a grammar G is denoted by $\mathcal{L}(G)$.

Example Grammar

"List of digits separated by + and − signs" (Example 2.1 in book):

$$\begin{aligned} L &\longrightarrow L + L \\ L &\longrightarrow L - L \\ L &\longrightarrow D \\ D &\longrightarrow 0 \mid 1 \mid \ldots \mid 9 \end{aligned}$$

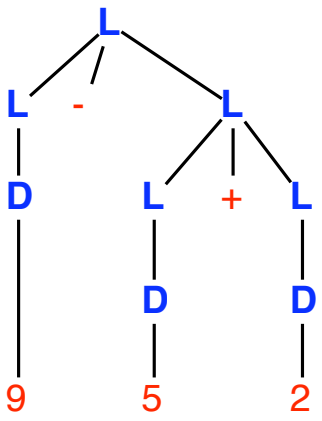
Derivation of 9−5+2 from L:

$$\begin{aligned} L &\Longrightarrow L - L \\ &\Longrightarrow D - L \\ &\Longrightarrow 9 - L \\ &\Longrightarrow 9 - L + L \\ &\Longrightarrow 9 - D + L \\ &\Longrightarrow 9 - 5 + L \\ &\Longrightarrow 9 - 5 + D \\ &\Longrightarrow 9 - 5 + 2 \end{aligned}$$

Parse Trees

Pictorial representation of derivations

$$
\begin{aligned}
L &\Longrightarrow L - L \\
&\Longrightarrow D - L \\
&\Longrightarrow 9 - L \\
&\Longrightarrow 9 - L + L \\
&\Longrightarrow 9 - D + L \\
&\Longrightarrow 9 - 5 + L \\
&\Longrightarrow 9 - 5 + D \\
&\Longrightarrow 9 - 5 + 2
\end{aligned}
$$

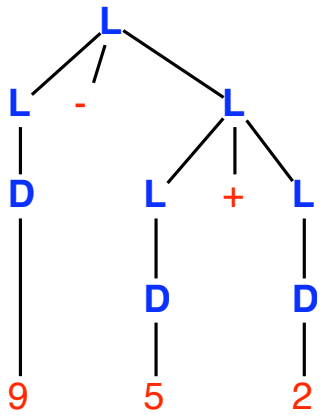


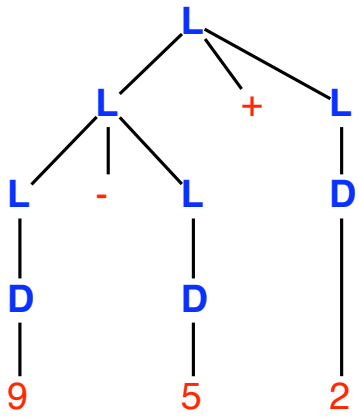
$$
\begin{aligned}
L &\Longrightarrow L - L \\
&\Longrightarrow L - L + L \\
&\Longrightarrow L - L + D \\
&\Longrightarrow L - L + 2 \\
&\Longrightarrow L - D + 2 \\
&\Longrightarrow L - 5 + 2 \\
&\Longrightarrow D - 5 + 2 \\
&\Longrightarrow 9 - 5 + 2
\end{aligned}
$$

Note: one parse tree may correspond to multiple derivations!

Ambiguity

A grammar is ambiguous if some sentence in the language has more than one parse tree.





Associativity and Precedence

- 9−5+2 ≡ (9−5)+2
- 9−5−2 ≡ (9−5)−2
- 9+5+2 ≡ (9+5)+2
- "+" and "−" usually have the same precedence and are left-associative.
  i.e. the second parse tree in the previous slide is the "correct" one
- The grammar can be changed to reflect the associativity and precedence:

$$
\begin{aligned}
L &\longrightarrow L + D \\
L &\longrightarrow L - D \\
L &\longrightarrow D \\
D &\longrightarrow 0 \mid 1 \mid \ldots \mid 9
\end{aligned}
$$

Syntax-Directed Translation Schemes

- A notation that attaches "program fragments" (also called actions) to productions in a grammar.
- The intuition is, whenever a production is used in recognizing a sentence, the corresponding action will be taken.
- Example:

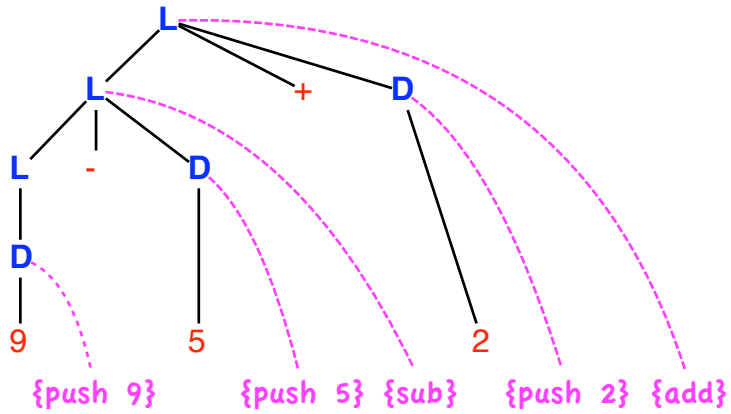
$$
\begin{aligned}
L &\longrightarrow L + D \quad \{\texttt{add}\} \\
L &\longrightarrow L - D \quad \{\texttt{sub}\} \\
L &\longrightarrow D \\
D &\longrightarrow 0 \quad \{\texttt{push 0}\} \\
D &\longrightarrow 1 \quad \{\texttt{push 1}\} \\
&\vdots
\end{aligned}
$$

Syntax-Directed Translation

- Actions can be seen as "additional leaves" introduced into a parse tree.
- Reading the actions left-to-right in the tree gives the "translation".

Example:

$$
\begin{array}{lcll}
L & \longrightarrow & L + D & \{\texttt{add}\} \\
L & \longrightarrow & L - D & \{\texttt{sub}\} \\
L & \longrightarrow & D & \\
D & \longrightarrow & 0 & \{\texttt{push 0}\} \\
D & \longrightarrow & 1 & \{\texttt{push 1}\} \\
\vdots & & &
\end{array}
$$



Grammars for Language Specification

- The language (i.e. set of allowed strings) of most programming languages can be specified using CFGs.
- The grammar notation may be tedious for some aspects of a language.
- For instance, an integer is defined by a grammar of the following form:

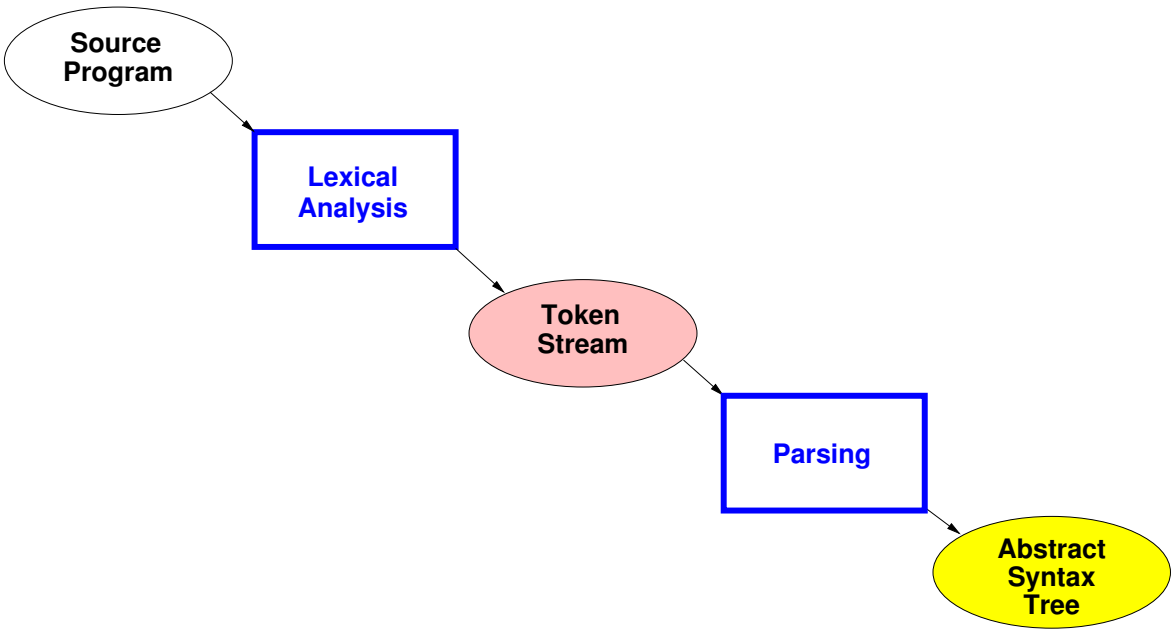
$$
\begin{array}{lcl}
I & \longrightarrow & P \mid + P \mid - P \\
P & \longrightarrow & D\ P \\
P & \longrightarrow & D \\
D & \longrightarrow & 0 \mid 1 \mid \ldots \mid 9
\end{array}
$$

- For simpler fragments, the notation of regular expressions may be used.
- $I = (+|-)?[0-9]+$

Syntax Analysis in Practice

- Usually divided into Lexical Analysis followed by Parsing.
- Lexical Analysis:
  - A lexical analyzer converts a stream of characters into a stream of tokens.
  - Each token has a *name* (associated with terminal symbols) and a *value* (also called its attribute).
  - A lexical analyzer is specified by a set of regular expression patterns and actions that are executed when the patterns are matched.
- Parsing:
  - A parser converts a stream of tokens into a tree.
  - Parsing uncovers the *structure* of a sentence in the language.
  - Parsers are specified by grammars (actually, by translation schemes which are sets of productions associated with actions).

Syntax Analysis



Lexical Analysis

First step of syntax analysis

- **Objective:** Convert the *stream of characters representing input program* into a sequence of *tokens*.
- Tokens are the "words" of the programming language.
- Examples:
  - The sequence of characters "`static int`" is recognized as two tokens, representing the two words "`static`" and "`int`".
  - The sequence of characters "`*x++`" is recognized as three tokens, representing "`*`", "`x`" and "`++`".

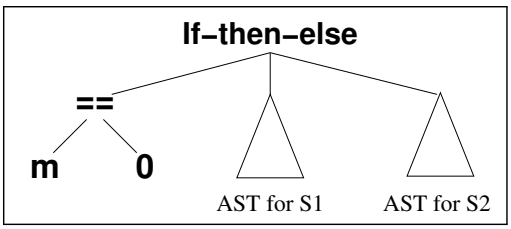
Parsing

Second step of syntax analysis

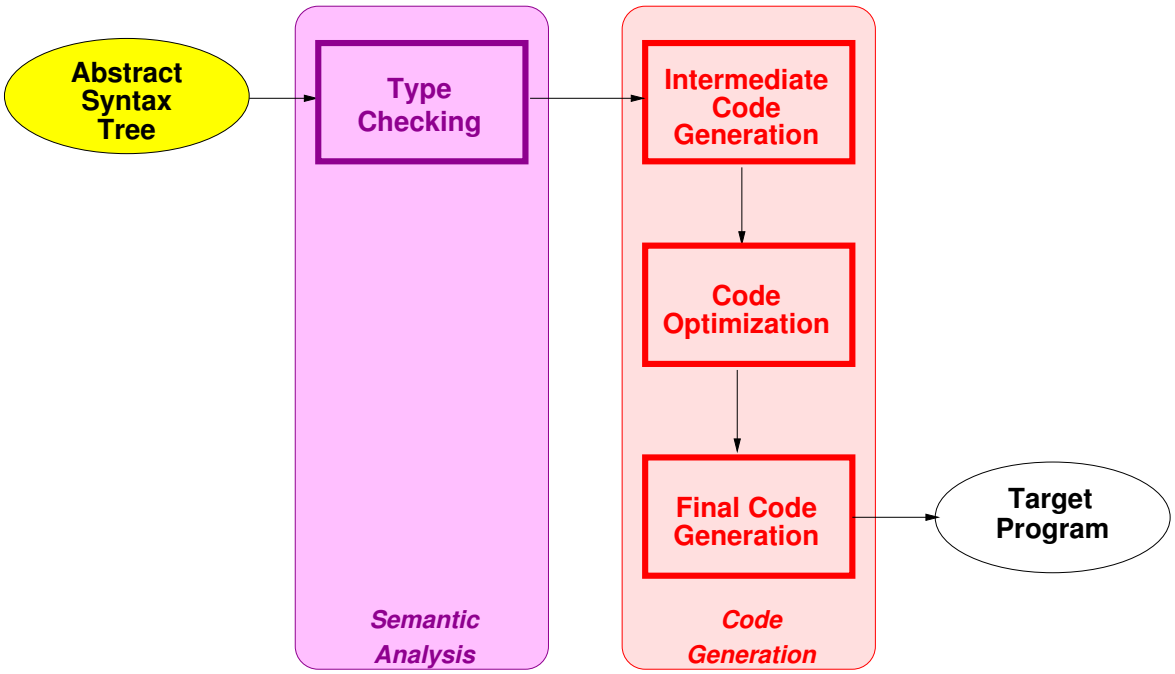
- **Objective:** Uncover the *structure* of a sentence in the program from a stream of *tokens*.
- For instance, the phrase "`x = +y`", which is recognized as four tokens, representing "x", "=" and "+" and "y", has the structure **=(x, +(y))**, i.e., an assignment expression, that operates on "x" and the expression "+(y)".
- **Output:** A *tree* called *abstract syntax tree* that reflects the structure of the input sentence.

Abstract Syntax Tree (AST)

- Represents the syntactic structure of the program, hiding a few details that are irrelevent to later phases of compilation.
- For instance, consider a statement of the form: "`if (m == 0) S1 else S2`" where S1 and S2 stand for some block of statements. A possible AST for this statement is:



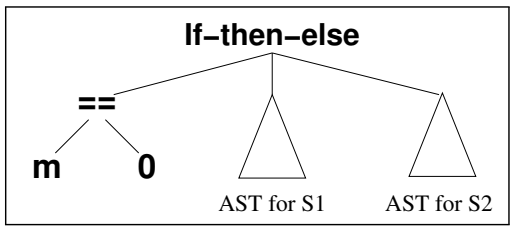
Semantic Processing



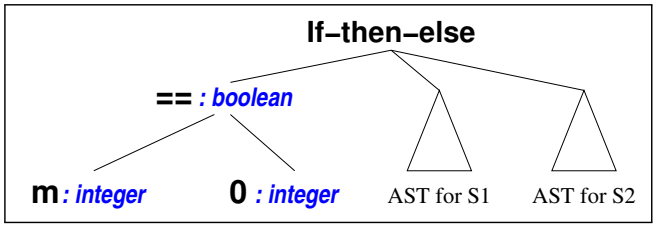
Type Checking

A instance of "Semantic Analysis"

- **Objective:** Decorate the AST with semantic information that is necessary in later phases of translation.
- For instance, the AST



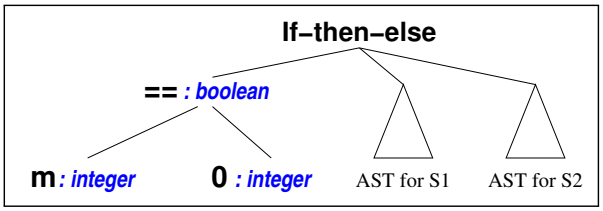
is transformed into



Intermediate Code Generation

- **Objective:** Translate each sub-tree of the decorated AST into *intermediate code*.
- Intermediate code hides many machine-level details, but has instruction-level mapping to many assembly languages.
- Main motivation for using an intermediate code is *portability*.

Intermediate Code Generation, an Example

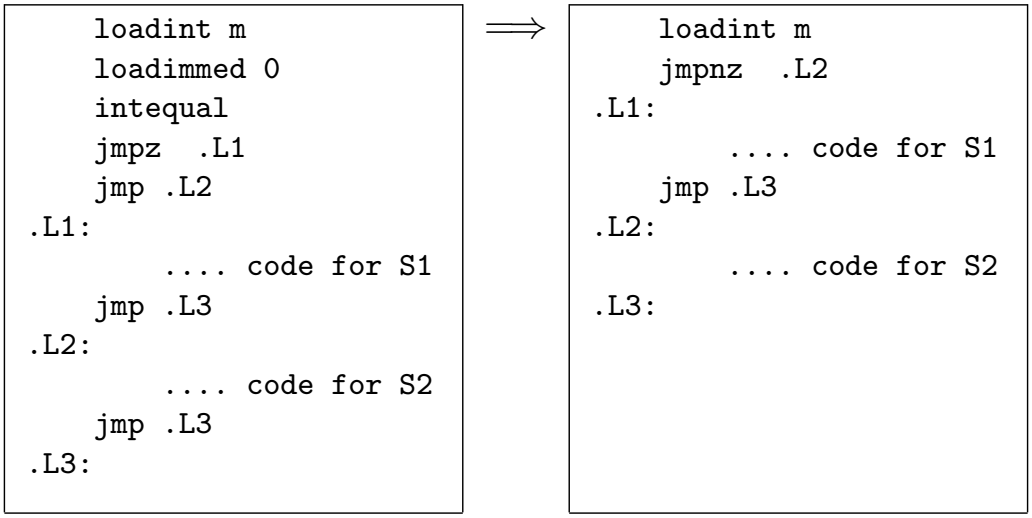



```
    loadint m
    loadimmed 0
    intequal
    jmpnz  .L1
    jmp .L2
.L1:
        .... code for S1
    jmp .L3
.L2:
        .... code for S2
    jmp .L3
.L3:
```


Code Optimization

- **Objective:** Improve the time and space efficiency of the generated code.
- Usual strategy is to perform a series of transformations to the intermediate code, with each step representing some efficiency improvement.
- *Peephole optimizations*: generate new instructions by combining/expanding on a small number of consecutive instructions.
- *Global optimizations*: reorder, remove or add instructions to change the structure of generated code.

Code Optimization, an Example



Final Code Generation

- **Objective:** Map instructions in the intermediate code to specific machine instructions.
- Supports standard object file formats.
- Generates sufficient information to enable symbolic debugging.

Final Code Generation, an Example