Introduction

Compiler Design

CSE 504

1. Overview
2. Syntax-Directed Translation
3. Phases of Translation
What is a Compiler?

- Programming problems are easier to solve in *high-level languages*
What is a Compiler?

- Programming problems are easier to solve in *high-level languages*
- High-level languages are closer to the problem domain
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, …
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
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
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:
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.
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:
LFB2:
    pushq %rbp
LCFI0:
    movq %rsp, %rbp
LCFI1:
    movl %edi, %ecx
    movl %esi, %edi
    testl %ecx, %ecx
    jne L11
    jmp L3
    .align 4, 0x90
L13:
    movl %edx, %ecx
L11:
    movl %edi, %edx
cmpl %edi, %ecx
    jg L6
    movl %edi, %eax
sarl $31, %edx
    idivl %ecx
L6:
    movl %ecx, %edi
testl %edx, %edx
    jne L13
L3:
    movl %edi, %eax
    leave
    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 $\equiv$ Syntax
  - Meaning $\equiv$ 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

- Source Program
- Syntax Analysis
- Abstract Syntax Tree
- Semantic Processing
- Target Program
Syntax-Directed Translation

Phases of Translation

Translation Steps

- **Syntax Analysis Phase**: Recognizes “sentences” in the program using the *syntax* of the language
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
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.
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.
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).
Syntax-Directed Translation: the *structure* (syntax) of a sentence in a language is used to give it a *meaning* (semantics).

- Bawat tao’y isinilang na may laya at magkakapantay ang taglay na dangal at karapatan.
Syntax-Directed Translation: the *structure* (syntax) of a sentence in a language is used to give it a *meaning* (semantics).

- Bawat tao’y isinilang na may laya at magkakapantay ang taglay na dangal at karapatan.
- Green wire connect after first not cut white also red wire.
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).

- Bawat tao’y isinilang na may laya at magkakapantay ang taglay na dangal at karapatan.
- Green wire connect after first not cut white also red wire.
- He sailed the coffee out of the leaf.
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).

- Bawat tao’y isinilang na may laya at magkakapantay ang taglay na dangal at karapatan.
- Green wire connect after first not 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 $\Sigma$).
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 $\Sigma$).
- Each string in the language is called a *sentence*. 

---

**Overview**

---

**Syntax-Directed Translation**

---

**Phases of Translation**

---

**Compiler Design**

---

**Introduction**

---

**CSE 504 11 / 33**
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 $\Sigma$).
- 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 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 $L(G)$. 

Example:

```
Stmt → while Expr do Stmt
```
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.
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
  \[ X \rightarrow \beta \]
  where
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

\[ X \rightarrow \beta \]

where

- \( X \) is a single non-terminal symbol representing a set of phrases in the language, and
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

\[ X \rightarrow \beta \]

where

- \( X \) is a single non-terminal symbol representing a set of phrases in the language, and
- \( \beta \) is a sequence of terminal and non-terminal symbols
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

  \[ X \rightarrow \beta \]

  where
  - \( X \) is a single non-terminal symbol representing a set of phrases in the language, and
  - \( \beta \) is a sequence of terminal and non-terminal symbols

- Example:

  \[ Stmt \rightarrow \text{while} \ Expr \ do \ Stmt \]
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

\[ X \rightarrow \beta \]

where

- \( X \) is a single non-terminal symbol representing a set of phrases in the language, and
- \( \beta \) is a sequence of terminal and non-terminal symbols

- Example:

\[ Stmt \rightarrow \text{while} \ Expr \ \text{do} \ Stmt \]

- A unique non-terminal, called the start symbol, represents the set of all sentences in the language.
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

  \[ X \rightarrow \beta \]

  where

  - \( X \) is a single non-terminal symbol representing a set of phrases in the language, and
  - \( \beta \) is a sequence of terminal and non-terminal symbols

- Example:

  \[ Stmt \rightarrow \text{while Expr do Stmt} \]

- 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):

\[
\begin{align*}
L & \rightarrow L + L \\
L & \rightarrow L - L \\
L & \rightarrow D \\
D & \rightarrow 0|1|\ldots|9 \\
\end{align*}
\]

Derivation of $9-5+2$ from $L$:

\[
L \quad \rightarrow \quad L - L
\]
Example Grammar

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

\[ L \rightarrow L + L \]
\[ L \rightarrow L - L \]
\[ L \rightarrow D \]
\[ D \rightarrow 0 | 1 | \ldots | 9 \]

Derivation of 9−5+2 from L:

\[ L \rightarrow L - L \]
\[ \quad \rightarrow D - L \]
Example Grammar

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

\[
\begin{align*}
L & \rightarrow L + L \\
L & \rightarrow L - L \\
L & \rightarrow D \\
D & \rightarrow 0 | 1 | \ldots | 9
\end{align*}
\]

Derivation of 9−5+2 from \( L \):

\[
\begin{align*}
L & \Rightarrow L - L \\
& \Rightarrow D - L \\
& \Rightarrow 9 - L
\end{align*}
\]
Example Grammar

“List of digits separated by + and – signs” (Example 2.1 in book):

\[
\begin{align*}
L & \rightarrow L + L \\
L & \rightarrow L - L \\
L & \rightarrow D \\
D & \rightarrow 0 | 1 | \ldots | 9
\end{align*}
\]

Derivation of \(9-5+2\) from \(L\):

\[
\begin{align*}
L & \Rightarrow L - L \\
\Rightarrow D - L \\
\Rightarrow 9 - L \\
\Rightarrow 9 - L + L
\end{align*}
\]
Example Grammar

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

\[
L \rightarrow L + L \\
L \rightarrow L - L \\
L \rightarrow D \\
D \rightarrow 0 | 1 | \ldots | 9
\]

Derivation of 9−5+2 from L:

\[
L \Rightarrow L - L \\
\Rightarrow D - L \\
\Rightarrow 9 - L \\
\Rightarrow 9 - L + L \\
\Rightarrow 9 - D + L \\
\Rightarrow 9 - 5 + L \\
\Rightarrow 9 - 5 + D \\
\Rightarrow 9 - 5 + 2
\]
Parse Trees

Pictorial representation of derivations

\[
\begin{align*}
L & \Rightarrow L - L \\
& \Rightarrow D - L \\
& \Rightarrow 9 - L \\
& \Rightarrow 9 - L + L \\
& \Rightarrow 9 - D + L \\
& \Rightarrow 9 - 5 + L \\
& \Rightarrow 9 - 5 + D \\
& \Rightarrow 9 - 5 + 2
\end{align*}
\]
Parse Trees

Pictorial representation of derivations

\[
\begin{align*}
L & \Rightarrow L - L \\
& \Rightarrow D - L \\
& \Rightarrow 9 - L \\
& \Rightarrow 9 - L + L \\
& \Rightarrow 9 - D + L \\
& \Rightarrow 9 - 5 + L \\
& \Rightarrow 9 - 5 + D \\
& \Rightarrow 9 - 5 + 2 \\

L & \Rightarrow L - L \\
& \Rightarrow L - L + L \\
& \Rightarrow L - L + D \\
& \Rightarrow L - L + 2 \\
& \Rightarrow L - D + 2 \\
& \Rightarrow L - 5 + 2 \\
& \Rightarrow D - 5 + 2 \\
& \Rightarrow 9 - 5 + 2
\end{align*}
\]

Note: one parse tree may correspond to multiple derivations!
A grammar is **ambiguous** if some sentence in the language has more than one parse tree.
Associativity and Precedence

- $9-5+2 \equiv (9-5)+2$
- $9-5-2 \equiv (9-5)-2$
- $9+5+2 \equiv (9+5)+2$
- “+” 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:

\[
\begin{align*}
L & \rightarrow L + D \\
L & \rightarrow L - D \\
L & \rightarrow D \\
D & \rightarrow 0|1|\ldots|9
\end{align*}
\]
Syntax-Directed Translation Schemes

- A notation that attaches “program fragments” (also called actions) to productions in a grammar.
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.
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:

  
  
  $$
  L \rightarrow L + D \quad \{\text{add}\}
  
  L \rightarrow L - D \quad \{\text{sub}\}
  
  L \rightarrow D
  
  D \rightarrow 0 \quad \{\text{push } 0\}
  
  D \rightarrow 1 \quad \{\text{push } 1\}
  
  :$
  

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:

```
L → L + D {add}
L → L - D {sub}
L → D
D → 0 {push 0}
D → 1 {push 1}
...
```

L L D
D + L -
5 2
D {push 9} {push 5} {push 2} {sub} {add}
9
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:

\[ L \rightarrow L + D \quad \{\text{add}\} \]
\[ L \rightarrow L - D \quad \{\text{sub}\} \]
\[ L \rightarrow D \]
\[ D \rightarrow 0 \quad \{\text{push 0}\} \]
\[ D \rightarrow 1 \quad \{\text{push 1}\} \]
\[ \vdots \]
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:

\[
\begin{align*}
L & \rightarrow L + D \quad \{\text{add}\} \\
L & \rightarrow L - D \quad \{\text{sub}\} \\
L & \rightarrow D \\
D & \rightarrow 0 \quad \{\text{push 0}\} \\
D & \rightarrow 1 \quad \{\text{push 1}\}
\end{align*}
\]
The language (i.e. set of allowed strings) of most programming languages can be specified using CFGs.
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.
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:

\[
\begin{align*}
I & \rightarrow P \mid + P \mid - P \\
P & \rightarrow D \ P \\
P & \rightarrow D \\
D & \rightarrow 0|1|\ldots|9
\end{align*}
\]
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:

  \[
  I \rightarrow P \mid + P \mid - P \\
  P \rightarrow D P \\
  P \rightarrow D \\
  D \rightarrow 0 \mid 1 \mid \ldots \mid 9
  \]

- For simpler fragments, the notation of regular expressions may be used.
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:

\[
I \rightarrow P \mid + P \mid - P \\
P \rightarrow D P \\
P \rightarrow D \\
D \rightarrow 0 \mid 1 \mid \ldots \mid 9
\]

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**.
Syntax Analysis in Practice

- Usually divided into **Lexical Analysis** followed by **Parsing**.
- **Lexical Analysis:**
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*.
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**).
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.
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:
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.
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.
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).
Translation Process
Syntax Analysis

Source Program

Lexical Analysis

Token Stream

Parsing

Abstract Syntax Tree
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 “++”.
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 irrelevant 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:

```
If–then–else

==

m

0

AST for S1

AST for S2
```
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

```
If-then-else
==
m 0
```

is transformed into

```
If-then-else
== : boolean
m : integer 0 : integer
```

```
AST for S1
AST for S2
```
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

If−then−else

\[
\begin{align*}
\text{AST for } S1 & \quad \text{AST for } S2 \\
\text{loadint } m & \\
\text{loadimmed } 0 & \\
\text{intequal} & \\
\text{jmpnz } .L1 & \\
\text{jmp } .L2 & \\
\text{.L1:} & \\
\text{.... code for } S1 & \\
\text{jmp } .L3 & \\
\text{.L2:} & \\
\text{.... code for } S2 & \\
\text{jmp } .L3 & \\
\text{.L3:} & \\
\end{align*}
\]
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

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

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

loadint m
jmpnz .L2
.L1:
    .... code for S1
    jmp .L3
.L2:
    .... code for S2
.L3:

⇒

movl 8(%ebp), %esi
testl %esi, %esi
jne .L2
.L1:
    .... code for S1
    jmp .L3
.L2:
    .... code for S2
.L3: