Lecture 3

Compiler Design

Awanish Pandey

Department of Computer Science and Engineering
Indian Institute of Technology
Roorkee

January 24, 2025

Code Generation

- Usually, a two-step process

Code Generation

- Usually, a two-step process
  - Generate **intermediate code** from the semantic representation of the program.

Code Generation

- Usually, a two-step process
  - Generate **intermediate code** from the semantic representation of the program.
  - Generate machine code from the intermediate code

Code Generation

- Usually, a two-step process
  - Generate **intermediate code** from the semantic representation of the program.
  - Generate machine code from the intermediate code
- Each phase will be simple

Code Generation

- Usually, a two-step process
  - Generate **intermediate code** from the semantic representation of the program.
  - Generate machine code from the intermediate code
- Each phase will be simple
- m * n vs m + n problem

if m source language specifications and n target specifications, then m * n compilers have to be made.
But if there is some common (universal) intermediate representation, then only m + n compilers have to be built.

Design of Intermediate Language

Design of Intermediate Language

Level of abstraction

Source Level	Target Level
Identifier	Memory
Expression	Register
Operator	Stack
Statement	Opcode
Condition	Addressing Mode
Functions	Libraries
Iteration	OS interaction

Design of Intermediate Language

Level of abstraction

Source Level	Target Level
Identifier	Memory
Expression	Register
Operator	Stack
Statement	Opcode
Condition	Addressing Mode
Functions	Libraries
Iteration	OS interaction

- IR semantics should ideally be independent of both the source and target language

Design of Intermediate Language

Level of abstraction

Source Level	Target Level
Identifier	Memory
Expression	Register
Operator	Stack
Statement	Opcode
Condition	Addressing Mode
Functions	Libraries
Iteration	OS interaction

- IR semantics should ideally be independent of both the source and target language
- It is next to impossible to design a single intermediate language to accommodate all programming languages.

Design of Intermediate Language

Level of abstraction

Source Level	Target Level
Identifier	Memory
Expression	Register
Operator	Stack
Statement	Opcode
Condition	Addressing Mode
Functions	Libraries
Iteration	OS interaction

- IR semantics should ideally be independent of both the source and target language
- It is next to impossible to design a single intermediate language to accommodate all programming languages.
- However, common IRs for similar languages and similar machines have been designed and are used for compiler development

Code Generation

- Code generation is a mapping from the source-level abstractions to target machine abstractions.

Code Generation

- Code generation is a mapping from the source-level abstractions to target machine abstractions.
- Map identifiers to memory.

Code Generation

- Code generation is a mapping from the source-level abstractions to target machine abstractions.
- Map identifiers to memory.
- Map operators to opcode(s).

Code Generation

- Code generation is a mapping from the source-level abstractions to target machine abstractions.
- Map identifiers to memory.
- Map operators to opcode(s).
- Convert conditionals and iterations to a test/jump or compare instructions

Code Generation

- Code generation is a mapping from the source-level abstractions to target machine abstractions.
- Map identifiers to memory.
- Map operators to opcode(s).
- Convert conditionals and iterations to a test/jump or compare instructions
- Layout parameter passing protocols: locations for parameters, return values, etc.

Code Generation

- Code generation is a mapping from the source-level abstractions to target machine abstractions.
- Map identifiers to memory.
- Map operators to opcode(s).
- Convert conditionals and iterations to a test/jump or compare instructions
- Layout parameter passing protocols: locations for parameters, return values, etc.
- Interface calls to library, runtime systems, and operating systems

Symbol Table

Symbol Table

- Information required about the program variables during compilation

Symbol table contains information about keywords and identifiers only. No other information about syntactic category (token_type) apart from keywords and identifiers will be stored in symbol table.

Symbol Table

- Information required about the program variables during compilation
  - Class of variable: keyword, identifier, etc.

Symbol Table

- Information required about the program variables during compilation
  - Class of variable: keyword, identifier, etc.
  - Type of variable: integer, float, array, function, etc.

Symbol Table

- Information required about the program variables during compilation
  - Class of variable: keyword, identifier, etc.
  - Type of variable: integer, float, array, function, etc.
  - Amount of storage required

Symbol Table

- Information required about the program variables during compilation
  - Class of variable: keyword, identifier, etc.
  - Type of variable: integer, float, array, function, etc.
  - Amount of storage required
  - Address in the memory

Symbol Table

- Information required about the program variables during compilation
  - Class of variable: keyword, identifier, etc.
  - Type of variable: integer, float, array, function, etc.
  - Amount of storage required
  - Address in the memory
  - Scope information

Symbol Table

- Information required about the program variables during compilation
  - Class of variable: keyword, identifier, etc.
  - Type of variable: integer, float, array, function, etc.
  - Amount of storage required
  - Address in the memory
  - Scope information
- Two ways to store this data

Symbol Table

- Information required about the program variables during compilation
  - Class of variable: keyword, identifier, etc.
  - Type of variable: integer, float, array, function, etc.
  - Amount of storage required
  - Address in the memory
  - Scope information
- Two ways to store this data
  - Attach the attribute with the symbols

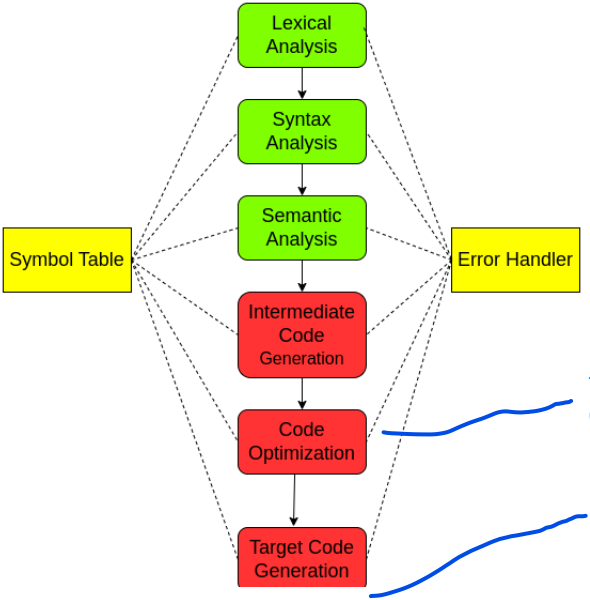
Symbol Table

- Information required about the program variables during compilation
  - Class of variable: keyword, identifier, etc.
  - Type of variable: integer, float, array, function, etc.
  - Amount of storage required
  - Address in the memory
  - Scope information
- Two ways to store this data
  - Attach the attribute with the symbols
  - Store it at a separate location.

maintain pointer to the location where the information or the attributes of identifier or keyword are stored in symbol table.

Final Compiler Structure

Final Compiler Structure



this is machine independent code optimization.

At last, we can have machine dependent code optimization.

Advantage of this model(Analysis-Synthesis model)

Advantage of this model(Analysis-Synthesis model)

- Front-end phases are known as analysis phases

Advantage of this model(Analysis-Synthesis model)

- Front-end phases are known as analysis phases
- Back-end phases are known as synthesis phases

Advantage of this model(Analysis-Synthesis model)

- Front-end phases are known as analysis phases
- Back-end phases are known as synthesis phases
- Each phase has a well-defined work

Advantage of this model(Analysis-Synthesis model)

- Front-end phases are known as analysis phases
- Back-end phases are known as synthesis phases
- Each phase has a well-defined work
- Each phase handles a logical activity in the process of compilation

Advantage of this model(Analysis-Synthesis model)

- Front-end phases are known as analysis phases
- Back-end phases are known as synthesis phases
- Each phase has a well-defined work
- Each phase handles a logical activity in the process of compilation
- Source and machine-independent code optimization is possible.

Advantage of this model(Analysis-Synthesis model)

- Front-end phases are known as analysis phases
- Back-end phases are known as synthesis phases
- Each phase has a well-defined work
- Each phase handles a logical activity in the process of compilation
- Source and machine-independent code optimization is possible.
- The optimization phase can be inserted after the front and back end phases have been developed and deployed

How can we trust compiler-generated code?

How can we trust compiler-generated code?

- Prove that the compiler is correct.

How can we trust compiler-generated code?

- Prove that the compiler is correct.
- No such technique proves such a large piece of code.

How can we trust compiler-generated code?

- Prove that the compiler is correct.
- No such technique proves such a large piece of code.
- Do systematic testing to increase the confidence level

How can we trust compiler-generated code?

- Prove that the compiler is correct.
- No such technique proves such a large piece of code.
- Do systematic testing to increase the confidence level
- Regression testing

How can we trust compiler-generated code?

- Prove that the compiler is correct.
- No such technique proves such a large piece of code.
- Do systematic testing to increase the confidence level
- Regression testing
  - Maintain a suite of test programs

How can we trust compiler-generated code?

- Prove that the compiler is correct.
- No such technique proves such a large piece of code.
- Do systematic testing to increase the confidence level
- Regression testing
  - Maintain a suite of test programs
  - Expected behavior of each program is documented

How can we trust compiler-generated code?

- Prove that the compiler is correct.
- No such technique proves such a large piece of code.
- Do systematic testing to increase the confidence level
- Regression testing
  - Maintain a suite of test programs
  - Expected behavior of each program is documented
  - All the test programs are compiled using the compiler, and deviations are reported to the compiler writer

How can we trust compiler-generated code?

- Prove that the compiler is correct.
- No such technique proves such a large piece of code.
- Do systematic testing to increase the confidence level
- Regression testing
  - Maintain a suite of test programs
  - Expected behavior of each program is documented
  - All the test programs are compiled using the compiler, and deviations are reported to the compiler writer
- Design of test suite

How can we trust compiler-generated code?

- Prove that the compiler is correct.
- No such technique proves such a large piece of code.
- Do systematic testing to increase the confidence level
- Regression testing
  - Maintain a suite of test programs
  - Expected behavior of each program is documented
  - All the test programs are compiled using the compiler, and deviations are reported to the compiler writer
- Design of test suite
  - Test programs should exercise every statement of the compiler at least once

How can we trust compiler-generated code?

- Prove that the compiler is correct.
- No such technique proves such a large piece of code.
- Do systematic testing to increase the confidence level
- Regression testing
  - Maintain a suite of test programs
  - Expected behavior of each program is documented
  - All the test programs are compiled using the compiler, and deviations are reported to the compiler writer
- Design of test suite
  - Test programs should exercise every statement of the compiler at least once
  - Usually requires great ingenuity to design such a test suite

How to reduce development and testing effort?

- **DO NOT WRITE COMPILERS**

How to reduce development and testing effort?

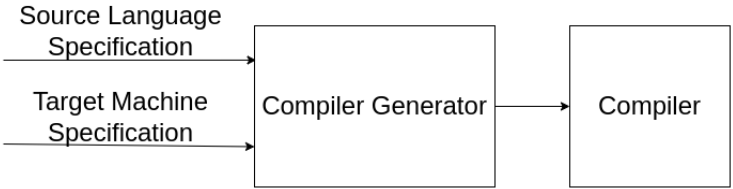
- **DO NOT WRITE COMPILERS**
- **GENERATE** compilers

How to reduce development and testing effort?

- **DO NOT WRITE COMPILERS**
- **GENERATE** compilers
- A compiler generator should be able to "generate" a compiler from the source language and target machine specifications

How to reduce development and testing effort?

- **DO NOT WRITE COMPILERS**
- **GENERATE** compilers
- A compiler generator should be able to "generate" a compiler from the source language and target machine specifications



Specification and Compiler Generator

Specification and Compiler Generator

- Language is broken into subcomponents like lexemes, structure, semantics, etc.

Specification and Compiler Generator

- Language is broken into subcomponents like lexemes, structure, semantics, etc.
- Each component can be specified separately.

Specification and Compiler Generator

- Language is broken into subcomponents like lexemes, structure, semantics, etc.
- Each component can be specified separately.
- For example, an identifier may be specified as:

Specification and Compiler Generator

- Language is broken into subcomponents like lexemes, structure, semantics, etc.
- Each component can be specified separately.
- For example, an identifier may be specified as:
  - A string of characters that has at least one alphabet

Specification and Compiler Generator

- Language is broken into subcomponents like lexemes, structure, semantics, etc.
- Each component can be specified separately.
- For example, an identifier may be specified as:
  - A string of characters that has at least one alphabet
  - Starts with an alphabet followed by an alphanumeric

Specification and Compiler Generator

- Language is broken into subcomponents like lexemes, structure, semantics, etc.
- Each component can be specified separately.
- For example, an identifier may be specified as:
  - A string of characters that has at least one alphabet
  - Starts with an alphabet followed by an alphanumeric
  - $letter(letter|digit)*$

Specification and Compiler Generator

- Language is broken into subcomponents like lexemes, structure, semantics, etc.
- Each component can be specified separately.
- For example, an identifier may be specified as:
  - A string of characters that has at least one alphabet
  - Starts with an alphabet followed by an alphanumeric
  - $letter(letter|digit)*$
- Grammar can be defined as:

Specification and Compiler Generator

- Language is broken into subcomponents like lexemes, structure, semantics, etc.
- Each component can be specified separately.
- For example, an identifier may be specified as:
  - A string of characters that has at least one alphabet
  - Starts with an alphabet followed by an alphanumeric
  - $letter(letter|digit)*$
- Grammar can be defined as:

$$
\begin{aligned}
selection\_statement : \; & IF(expression) \quad statement \\
& |IF(expression) \quad statement \quad ELSE \quad statement \\
& |SWITCH(expression) \quad statement;
\end{aligned}
$$

Specification and Compiler Generator

- Language is broken into subcomponents like lexemes, structure, semantics, etc.
- Each component can be specified separately.
- For example, an identifier may be specified as:
  - A string of characters that has at least one alphabet
  - Starts with an alphabet followed by an alphanumeric
  - $letter(letter|digit)*$
- Grammar can be defined as:

$$
\begin{aligned}
selection\_statement : \; & IF(expression) \quad statement \\
& |IF(expression) \quad statement \quad ELSE \quad statement \\
& |SWITCH(expression) \quad statement;
\end{aligned}
$$

- Changing specifications of a phase can lead to a new compiler

Specification and Compiler Generator

- Language is broken into subcomponents like lexemes, structure, semantics, etc.
- Each component can be specified separately.
- For example, an identifier may be specified as:
  - A string of characters that has at least one alphabet
  - Starts with an alphabet followed by an alphanumeric
  - $letter(letter|digit)*$
- Grammar can be defined as:

$$
\begin{aligned}
selection\_statement : \; & IF(expression) \quad statement \\
& |IF(expression) \quad statement \quad ELSE \quad statement \\
& |SWITCH(expression) \quad statement;
\end{aligned}
$$

- Changing specifications of a phase can lead to a new compiler
- Cuts down development/maintenance time by almost 30-40%

research proven fact

you don't need to write compilers each time, just change the specifications.

Specification and Compiler Generator

- Language is broken into subcomponents like lexemes, structure, semantics, etc.
- Each component can be specified separately.
- For example, an identifier may be specified as:
  - A string of characters that has at least one alphabet
  - Starts with an alphabet followed by an alphanumeric
  - $letter(letter|digit)*$
- Grammar can be defined as:

$$
\begin{aligned}
selection\_statement : \; & IF(expression) \quad statement \\
& |IF(expression) \quad statement \quad ELSE \quad statement \\
& |SWITCH(expression) \quad statement;
\end{aligned}
$$

- Changing specifications of a phase can lead to a new compiler
- Cuts down development/maintenance time by almost 30-40%
- Tool development and testing is a one-time effort

Specification and Compiler Generator

- Language is broken into subcomponents like lexemes, structure, semantics, etc.
- Each component can be specified separately.
- For example, an identifier may be specified as:
  - A string of characters that has at least one alphabet
  - Starts with an alphabet followed by an alphanumeric
  - $letter(letter|digit)*$
- Grammar can be defined as:

$$
\begin{aligned}
selection\_statement : \; & IF(expression) \quad statement \\
& |IF(expression) \quad statement \quad ELSE \quad statement \\
& |SWITCH(expression) \quad statement;
\end{aligned}
$$

- Changing specifications of a phase can lead to a new compiler
- Cuts down development/maintenance time by almost 30-40%
- Tool development and testing is a one-time effort
- Compiler performance can be improved by improving a tool