Compiler Development (CMPSC 401)

Janyl Jumadinova

January 17, 2018
What is a compiler?
What is a compiler?

```c
int a, b;
  a = 3;
if (a < 4) {
    b = 2;
  } else {
    b = 3;
}
```

```assembly
var a
var b
mov 3 a
mov 4 r1
cmpi a r1
jge l_e
mov 2 b
jmp l_d
l_e: mov 3 b
l_d: ;done
```
Compilers are translators

- Fortran
- C
- C++
- Java
- Text processing language
- HTML/XML
- Command & Scripting Languages
- Natural language
- Domain specific languages

- Machine code
- Virtual machine code
- Transformed source code
- Augmented source code
- Low-level commands
- Semantic components
- Another language
Compilers are optimizers

```plaintext
int a, b, c;
b = a + 3;
c = a + 3;
```

```plaintext
var a
var b
var c
mov a r1
addi 3 r1
mov r1 b
mov a r2
addi 3 r2
mov r2 c
```
Why Study Compilers?
Why Study Compilers?

- Compilers provide portability
- Compilers enable high performance and productivity
- Techniques used for compiler design are also useful for other things
- Compilers and interpreters for domain-specific languages are popular
Why Study Compilers?

- Compilers provide portability
- Compilers enable high performance and productivity
- Techniques used for compiler design are also useful for other things
- Compilers and interpreters for domain-specific languages are popular
- **Bring together**: Data structures and Algorithms, Formal Languages, Computer Architecture
Why Study Compilers?

- Compilers provide portability
- Compilers enable high performance and productivity
- Techniques used for compiler design are also useful for other things
- Compilers and interpreters for domain-specific languages are popular
- **Bring together**: Data structures and Algorithms, Formal Languages, Computer Architecture
- **Influence**: Language Design, Architecture (influence is bi-directional), Techniques used influence other areas (program analysis, testing, ...)
Common compiler types

- High level language $\rightarrow$ assembly language (e.g., gcc)
Common compiler types

- High level language $\rightarrow$ assembly language (e.g., gcc)
- High level language $\rightarrow$ machine independent bytecode (e.g., javac)
Common compiler types

- High level language → assembly language (e.g., gcc)
- High level language → machine independent bytecode (e.g., javac)
- Bytecode → native machine code (e.g., java’s JIT compiler)
Common compiler types

- High level language $\rightarrow$ assembly language (e.g., gcc)
- High level language $\rightarrow$ machine independent bytecode (e.g., javac)
- Bytecode $\rightarrow$ native machine code (e.g., java’s JIT compiler)
- High level language $\rightarrow$ high level language (e.g., domain specific languages, many research languages)
View from 50,000 feet
There are two parts to compilation:

1. Analysis
2. Synthesis
There are two parts to compilation:

1. **Analysis**: determines the operations implied by the source program which are recorded in a tree structure

2. **Synthesis**
There are two parts to compilation:

1. **Analysis**: determines the operations implied by the source program which are recorded in a tree structure.
2. **Synthesis**: takes the tree structure and translates the operations therein into the target program.
Common Compiler Phases

- Lexical Analysis (scanning)
- Syntax Analysis (parsing)
- Semantic Analysis (type checking)
- Intermediate Code Generation
- Machine Code Generation
- Optimization
- Memory Management
Grouping of phases

- Compiler front and back ends:
  - **Front-end**: analysis (machine independent)
  - **Back-end**: synthesis (machine dependent)
Grouping of phases

- Compiler front and back ends:
  - **Front-end**: analysis (machine independent)
  - **Back-end**: synthesis (machine dependent)

- Compiler passes:
  - A collection of phases is done only once (single pass) or multiple times (multi pass)
Grouping of phases

- Compiler front and back ends:
  - **Front-end**: analysis (machine independent)
  - **Back-end**: synthesis (machine dependent)

- Compiler passes:
  - A collection of phases is done only once (single pass) or multiple times (multi pass)

- **Single pass**: usually requires everything to be defined before being used in source program
Grouping of phases

- Compiler front and back ends:
  - **Front-end**: analysis (machine independent)
  - **Back-end**: synthesis (machine dependent)

- Compiler passes:
  - A collection of phases is done 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
Software development tools are available to implement one or more compiler phases
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
Traditional Two-Pass Compiler

Source code → Front End → IR → Back End → Machine code

Errors
The Front-End

\[ \text{Source code} \rightarrow \text{Scanner} \rightarrow \text{Parser} \rightarrow \text{IR} \]

\[ \text{tokens} \]

\[ \text{Errors} \]
Compiler starts by seeing only program text

```java
if (a < 4) {
    b = 5
}
```
Scanner

- Compiler starts by seeing only program text
- Scanner converts program text into string of tokens

```
'if' 'a' '<' '4'
'{' 
'b' '=' '5'
'}'
```
Scanner

- Compiler starts by seeing only program text
- Scanner converts program text into a string of tokens
- But we still don’t know what the syntactic structure of the program is
Parser

- Converts string of tokens into a parse tree or an abstract syntax tree
Parser

- Converts string of tokens into a *parse tree* or an *abstract syntax tree*
- Captures syntactic structure of the code
Parser

- Converts string of tokens into a parse tree or an abstract syntax tree
- Captures syntactic structure of the code

\[
\text{if} \quad ( \quad \text{ID(a)} \quad \text{OP(<)} \quad \text{LIT(4)} \quad ) \\
\{ \quad \text{ID(b)} \quad = \quad \text{LIT(5)} \quad \} \\
\]
Parser

- Converts string of tokens into a parse tree or an abstract syntax tree
- Captures syntactic structure of code

![Diagram of if-stmt, stmt_list, assign_stmt, and other nodes with labels a, 4, b, and 5 connected by edges labeled cond-expr, lhs, rhs, and then-clause.]
Semantic actions

- Interpret the **semantics** of syntactic constructs
Semantic actions

- Interpret the **semantics** of syntactic constructs
- Up to now we have been only concerned with what the **syntax** of the code is
Interpret the **semantics** of syntactic constructs

Up to now we have been only concerned with what the **syntax** of the code is

What’s the difference?
Syntax vs. Semantics

- **Syntax**: “grammatical” structure of a language

  
  But something that is syntactically correct may not mean anything!

  “colorless green ideas sleep furiously”
Syntax vs. Semantics

- **Syntax:** “grammatical” structure of a language
- What symbols, in what order, is a legal part of the language?
Syntax vs. Semantics

- **Syntax**: “grammatical” structure of a language
- What symbols, in what order, is a legal part of the language?
- But something that is syntactically correct may not mean anything!

“colorless green ideas sleep furiously”
Syntax vs. Semantics

- **Syntax**: “grammatical” structure of a language
- What symbols, in what order, is a legal part of the language?
- But something that is syntactically correct may not mean anything!
- “colorless green ideas sleep furiously”
Syntax vs. Semantics

- **Semantics**: meaning of the language
Syntax vs. Semantics

- **Semantics:** meaning of the language
- What does a particular set of symbols, in a particular order, mean?
Syntax vs. Semantics

- **Semantics**: meaning of the language
- What does a particular set of symbols, in a particular order, mean?
- What does it mean to be an if statement?
Syntax vs. Semantics

- **Semantics**: meaning of the language
- What does a particular set of symbols, in a particular order, mean?
- What does it mean to be an if statement?
- evaluate the conditional, if the conditional is true, execute the then clause, otherwise execute the else clause
Semantic Actions

- Actions taken by compiler based on the semantics of program statements:
  - Building a symbol table
  - Generating intermediate representations
Symbol tables

- A list of every declaration in the program, along with other information
- Variable declarations: types, scope
- Function declarations: return types, number and type of arguments

Program Example

```
Integer ii;
...
ii = 3.5;
...
print ii;
```

Symbol Table

```
Name  Type  Scope
ii    int   global
...
```
Intermediate representation

- Also called IR
Intermediate representation

- Also called IR
- A (relatively) low-level representation of the program
- But not machine-specific!
- One example: three address code

\[
\text{bge a, 4, done} \\
\text{mov 5, b} \\
\text{done: // done!}
\]

- Each instruction can take at most three operands (variables, literals, or labels)
- Note: no registers!
Optimizer

- Transforms code to make it more efficient
- Different kinds, operating at different levels
Optimizer

- Transforms code to make it more efficient
- Different kinds, operating at different levels
- **High-level optimizations**
  - Loop interchange, parallelization
  - Operates at level of AST, or even source code
Optimizer

- Transforms code to make it more efficient
- Different kinds, operating at different levels
- **High-level optimizations**
  - Loop interchange, parallelization
  - Operates at level of AST, or even source code
- **Scalar optimizations**
  - Dead code eliminations, common sub-expressions eliminations
  - Operates on IR
Optimizer

- Transforms code to make it more efficient
- Different kinds, operating at different levels

**High-level optimizations**
- Loop interchange, parallelization
- Operates at level of AST, or even source code

**Scalar optimizations**
- Dead code eliminations, common sub-expressions eliminations
- Operates on IR

**Local optimizations**
- Strength reduction, constant folding
- Operates on small sequences of instructions
Code generation

- Generate assembly from intermediate representation
- Select which instructions to use
- Schedule instructions
- Decide which registers to use

bge a, 4, done
mov 5, b
done: //done!

mov 4, r1
ld a, r2
cmp r1, r2
blt done
mov 5, r1
st r1, b
done:
Overall structure of a compiler

- **Scanner**: Use regular expressions to define tokens. Can then use scanner generators such as lex or flex.
- **Parser**: Define language using context free grammars. Can then use parser generators such as yacc or bison.
- **Semantic Routines**: Semantic routines done by hand. But can be formalized.
- **Optimizer**: Mostly written manually. Automation is an active research area (e.g., dataflow analysis frameworks)
- **Code Generation**: Written manually
Overall structure of a compiler

- **Scanner:** Use regular expressions to define tokens. Can then use scanner generators such as lex or flex.
- **Parser:** Define language using context free grammars. Can then use parser generators such as yacc or bison.
- **Semantic Routines:** Semantic routines done by hand. But can be formalized.
- **Optimizer:** Mostly written manually. Automation is an active research area (e.g., dataflow analysis frameworks)
- **Code Generation:** Written manually

Many of these can be combined!
Design Considerations

- Compiler and language designs influence each other
Design Considerations

- Compiler and language designs influence each other
- High-level languages are harder to compile
  - More work to bridge the gap between language and assembly
Design Considerations

- Compiler and language designs influence each other

- High-level languages are harder to compile
  - More work to bridge the gap between language and assembly

- Flexible languages are often harder to compile
  - Dynamic typing (Ruby, Python) makes a language very flexible, but it is hard for a compiler to catch errors

- Compiler design is influenced by architectures