

ECE 573: Compilers & Translation Systems Engineering

Instructor: Milind Kulkarni (milind@purdue.edu)

<http://www.engineering.purdue.edu/~milind/ece573/2011spring>

Announcements

- “Problem Set” 0 online
 - Short questionnaire
- Project step 0 due this Friday
 - Use subject line “ECE 573: Project Step 0”
 - Let me know if you plan to work with a partner
 - Send me an email even if you don’t!


Tuesday, January 11, 2011

Tuesday, January 11, 2011

What is a compiler?

- Traditionally: Program that analyzes and **translates** from a high level language (e.g., C++) to low-level assembly language that can be executed by hardware

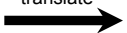
```
int a, b;
a = 3;
if (a < 4) {
    b = 2;
} else {
    b = 3;
}
```



```
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
```

Tuesday, January 11, 2011

Compilers are **translators**


- | | | |
|---|--|---|
| <ul style="list-style-type: none">• Fortran• C• C++• Java• Text processing language• HTML/XML• Command & Scripting Languages• Natural language• Domain specific languages | <p>translate</p>  | <ul style="list-style-type: none">◆ Machine code◆ Virtual machine code◆ Transformed source code◆ Augmented source code◆ Low-level commands◆ Semantic components◆ Another language |
|---|--|---|

Tuesday, January 11, 2011


Compilers are **optimizers**

- Can perform optimizations to make a program more efficient

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



```
var a
var b
var c
mov a r1
addi 3 r1
mov r1 b
mov a r2
addi 3 r2
mov r2 c
```



```
var a
var b
var c
mov a r1
addi 3 r1
mov r1 b
mov r1 c
```

Tuesday, January 11, 2011

Why do we need compilers?

- Compilers provide **portability**
- Old days: whenever a new machine was built, programs had to be rewritten to support new instruction sets
- IBM System/360 (1964): Common Instruction Set Architecture (ISA) — programs could be run on any machine which supported ISA
 - Common ISA is a huge deal (note continued existence of x86)
- But still a problem: when new ISA is introduced (EPIC) or new extensions added (x86-64), programs would have to be rewritten
- Compilers bridge this gap: write new compiler for an ISA, and then simply recompile programs!

Tuesday, January 11, 2011

Why do we need compilers? (II)

- Compilers enable **high performance and productivity**
- Old: programmers wrote in assembly language, architectures were simple (no pipelines, caches, etc.)
 - Close match between programs and machines — easier to achieve performance
- New: programmers write in high level languages (Ruby, Python), architectures are complex (superscalar, out-of-order execution, multicore)
- Compilers are needed to bridge this **semantic gap**
 - Compilers let programmers write in high level languages and still get good performance on complex architectures

Tuesday, January 11, 2011

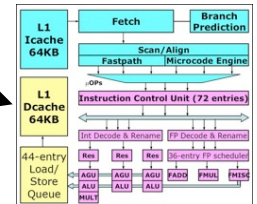
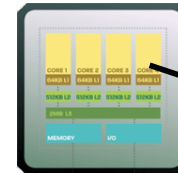
Semantic Gap

```
def index
  @posts = Post.find(:all)

  respond_to do |format|
    format.html # index.html.erb
    format.xml { render :xml =>
  @posts }
  end
end
```

Snippet of Ruby-on-rails code

AMD Barcelona architecture



Tuesday, January 11, 2011

Semantic Gap

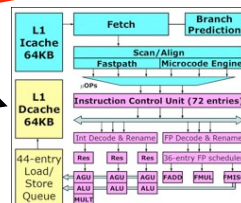
```
def index
  @posts = Post.find(:all)

  respond_to do |format|
    format.html # index.html.erb
    format.xml { render :xml =>
  @posts }
  end
end
```

Snippet of Ruby-on-rails

This would be impossible without compilers!

AMD Barcelona architecture



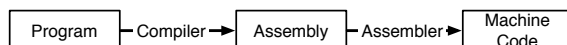
Tuesday, January 11, 2011

Some 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—including mine!)

Tuesday, January 11, 2011

HLL to Assembly

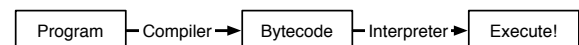


- Compiler converts program into assembly
- Assembler is machine-specific translator which converts assembly into machine code


```
add $7 $8 $9 ($7 = $8 + $9)  $\Rightarrow$  000000 00111 01000 01001 00000 100000
```
- Conversion is usually one-to-one with some exceptions
 - Program locations
 - Variable names

Tuesday, January 11, 2011

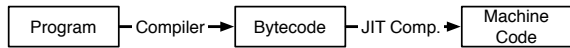
HLL to Bytecode



- Compiler converts program into machine independent bytecode
 - e.g., javac generates Java bytecode, C# compiler generates CIL
- Interpreter then executes bytecode “on-the-fly”
 - Bytecode instructions are “executed” by invoking methods of the interpreter, rather than directly executing on the machine
- Aside: what are the pros and cons of this approach?

Tuesday, January 11, 2011

Bytecode to Assembly



- Compiler converts program into machine independent bytecode
 - e.g., javac generates Java bytecode, C# compiler generates CIL
- Just-in-time compiler compiles code *while program executes* to produce machine code
- Is this better or worse than a compiler which generates machine code directly from the program?

Structure of a Compiler

Tuesday, January 11, 2011

Tuesday, January 11, 2011

Scanner

- Compiler starts by seeing only program text

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

Tuesday, January 11, 2011

Scanner

- Compiler starts by seeing only program text

```
'i' 'f' '_' '(' 'a' '<' '4' ')' '  
'_ ' '{' '\n' '\t' 'b' '=' '5' '  
'\n' '}'
```

Tuesday, January 11, 2011

Scanner

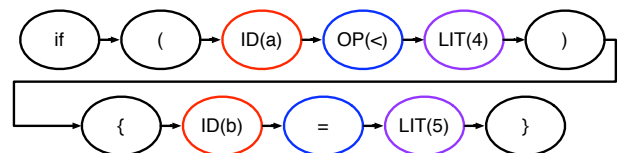
- Compiler starts by seeing only program text
- Scanner converts program text into string of *tokens*

```
'i' 'f' '_' '(' 'a' '<' '4' ')' '  
'_ ' '{' '\n' '\t' 'b' '=' '5' '  
'\n' '}'
```

Tuesday, January 11, 2011

Scanner

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

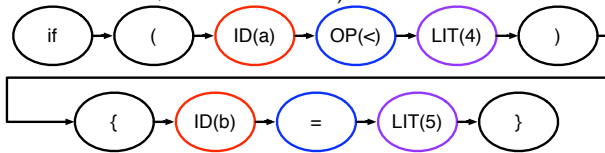


- But we still don't know what the *syntactic structure* of the program is

Tuesday, January 11, 2011

Parser

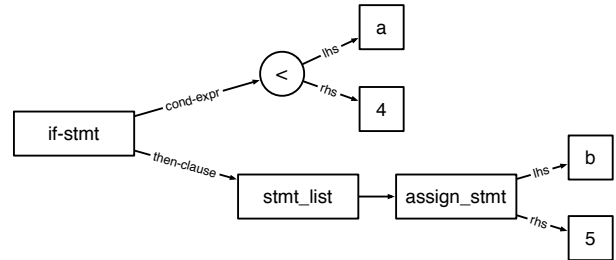
- Converts string of tokens into a *parse tree* or an *abstract syntax tree*.
- Captures syntactic structure of code (i.e., “this is an if statement, with a then-block”)



Tuesday, January 11, 2011

Parser

- Converts string of tokens into a *parse tree* or an *abstract syntax tree*.
- Captures syntactic structure of code (i.e., “this is an if statement, with a then-block”)



Tuesday, January 11, 2011

Semantic actions

- Interpret the *semantics* of syntactic constructs
 - Note that up until now we have only been concerned with what the *syntax* of the code is
 - What's the difference?

Tuesday, January 11, 2011

Syntax vs. Semantics

- Syntax: “grammatical” structure of language
 - What symbols, in what order, is a legal part of the language?
 - But something that is syntactically correct may mean nothing!
 - “colorless green ideas sleep furiously”
- Semantics: meaning of 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”

Tuesday, January 11, 2011

A note on semantics

- How do you define semantics?
 - Static semantics: properties of programs
 - All variables must have a type
 - Expressions must use consistent types
 - Can define using *attribute grammars*
 - Execution semantics: how does a program execute?
 - Can define an *operational* or *denotational* semantics for a language
 - Not covered in this class (take CS 565)
- For many languages, “the compiler is the specification”

Tuesday, January 11, 2011

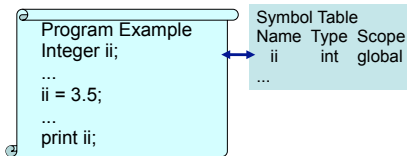
Semantic actions

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

Tuesday, January 11, 2011

Symbol tables

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



Tuesday, January 11, 2011

Intermediate representation

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

```
bge a, 4, done
mov 5, b
done: //done!
```
- Each instruction can take at most three operands (variables, literals, or labels)
- Note: no registers!

Tuesday, January 11, 2011

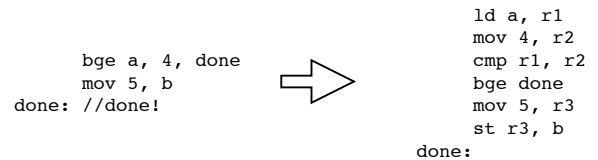
Optimizer

- 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 elimination, common sub-expression elimination
 - Operates on IR
- Local optimizations
 - Strength reduction, constant folding
 - Operates on small sequences of instructions

Tuesday, January 11, 2011

Code generation

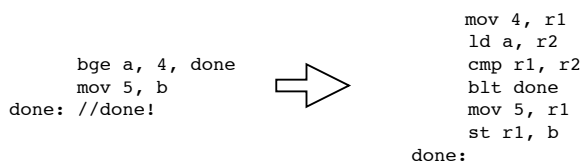
- Generate assembly from intermediate representation
 - Select which instructions to use
 - Schedule instructions
 - Decide which registers to use



Tuesday, January 11, 2011

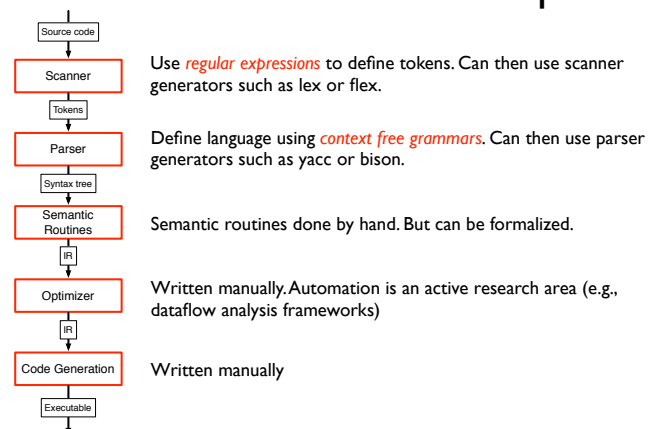
Code generation

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



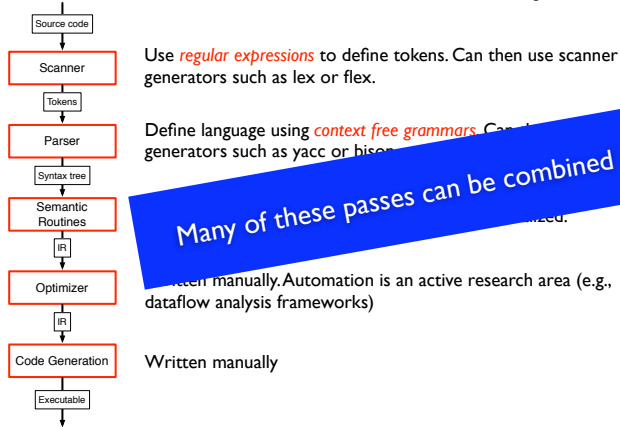
Tuesday, January 11, 2011

Overall structure of a compiler



Tuesday, January 11, 2011

Overall structure of a compiler



Tuesday, January 11, 2011

Design considerations (I)

- Compiler and language designs influence each other
- Higher 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 (in fact, many simply won't)

Tuesday, January 11, 2011

Design considerations (II)

- Compiler design is influenced by architectures
 - CISC vs. RISC
 - CISC designed for days when programmers wrote in assembly
 - For a compiler to take advantage of string manipulation instructions, it must be able to recognize them
 - RISC has a much simpler instruction model
 - Easier to compile for

Tuesday, January 11, 2011