Compilers

Are

Way

Cool !!!

CS 3510 – Compiler Theory and Practice

Dr. Dave Gallagher

Purpose

A reminder of our purpose in CS 3510

We are here to glorify and honor God in the way we conduct ourselves in this course!

Objectives

The Course Title is: CS 3510, Compiler Theory and Practice

- The title outlines our objectives – We are here to prepare students to serve God as software engineers by:
  - Learning the theory which is foundational to compiler construction
  - Practicing object-oriented techniques for implementing a working compiler
- All compiler courses have these two objectives
  - They vary, however, in how they weight the relative importance
  - We will try to weigh them fairly equally

Objectives

- Unlike other courses (like 2210), we don't have a multitude of hidden objectives (like learn Netbeans, learn Swing, learn UML)
  - About the only add-on tools we will be looking at are:
    - Lex
    - Yacc
  - These are open-source tools for doing scanning and parsing
    - If you are going to build a real full-scale compiler, you would want to use these tools (or some like them)
  - For this course, we will want you to write java code directly rather than use these tools
    - One project with lex

Introduction

- How many of you plan to go out into industry and write compilers?

- Then why do we make you take a compiler course?

  1. The theory portion of the course contains many principles foundational to Computer Science
     - Finite automata, language theory, etc
  2. Implementation techniques very instructive
     - Data structures (parse tree, polymorphism)
     - Algorithms (optimizations, register allocation)
  3. Understanding how a compiler works will make you a better programmer
  4. Great hands-on project that performs a useful function
     - Rather than a part-task project like “build a stack”

Introduction

- The best programmers are those who have a firm understanding of how the compiler, operating system, and hardware work

  - How well does your code interface with the target architecture?

Example: which of the following code pieces will execute fastest?

```
int childIndex = findKey (node, key);
TFNode childNode = node.getChild[childIndex];
Item minItem = childNode.getItem[0];
return minItem.element();
```

```
return  ((node.getChild[findKey(node,key)]).getItem[0]).element();
```

Introduction

- Understanding how the compiler works will allow you to program smarter

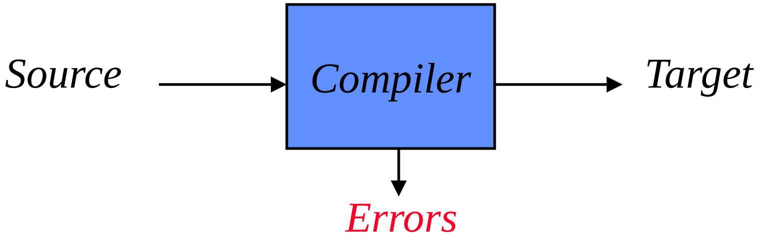
Example: which of the following code pieces will execute fastest?

```
int i = 4;
int j = 5;

for (int k = 0; k < 100; k++) {
   A[k] = i*j + k;
}
```

```
int i = 4;
int j = 5;
int h = i * j;

for (int k = 0; k < 100; k++) {
   A[k] = h + k;
}
```

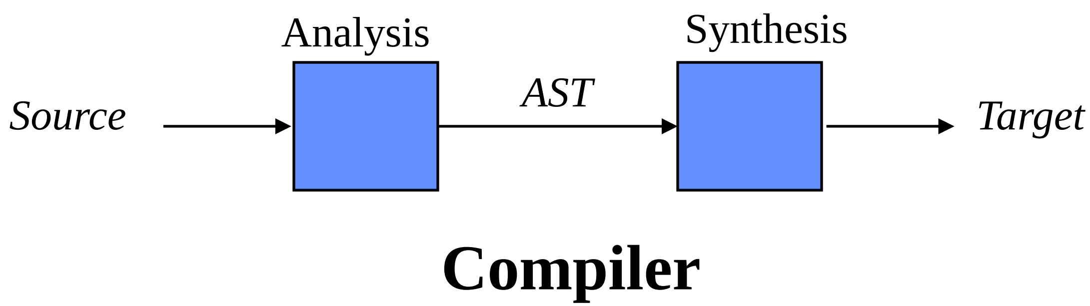
Introduction

- I guess if we are going to study compilers we had better figure out what they are
  - What is a “compiler”?
- A computer program which translates a program written in one language into another language
  - It reads in from the source language
  - It produces code in the target language
  - If things go like they normally do for me, it also produces error messages based on what it saw



Introduction

- Compilers have many phases or stages of execution
  - Also known as compiler passes
- But, in essence, there are just 2 main tasks of a compiler
  - Analysis (or Front End)
    - Figure out what the source says
  - Synthesis (or Back End)
    - Convert it into an alternate form



Introduction

- Source language
  - Often HLL source code

- Target language
  - Could be object code (like a .o file in C++)
  - Frequently it is assembly code (like a .s file or .class file)
    - Compiler may use an existing assembler to convert assembly code into object code
    - In the case of java, an interpreter executes .class file
  - Could be some intermediate code
    - Academic compilers, java, .net
  - Could be source code in another language
    - Cross-compilers

Introduction

- Typically, we think of taking a program and converting it to an executable
- But there are other forms of compilers
  - Silicon compiler
    - Takes a circuit description in a hardware description language (HDL) and converts it into a form that can be used to program a gate array or to make a VLSI circuit
  - Text formatter
    - Takes something like latex code and converts it into PDF or Postscript
  - Database query interpreter
    - Takes natural language input and makes query in a form the database can understand
- Each form still needs to do analysis and synthesis

Introduction

- Other tools similar to compilers
  - Smart editors
    - Have to analyze input, and do things like change color or automatically generate matching braces
  - Program analyzers
    - Read in a program and analyze problems it might have
      - Code that can't be executed
      - Variables used before defined
      - Memory leaks
  - Interpreters
    - How does an interpreter differ from a compiler?

It does analysis phase, but rather than doing synthesis, it "executes" the operation found in the analysis phase

Phases of Compilation

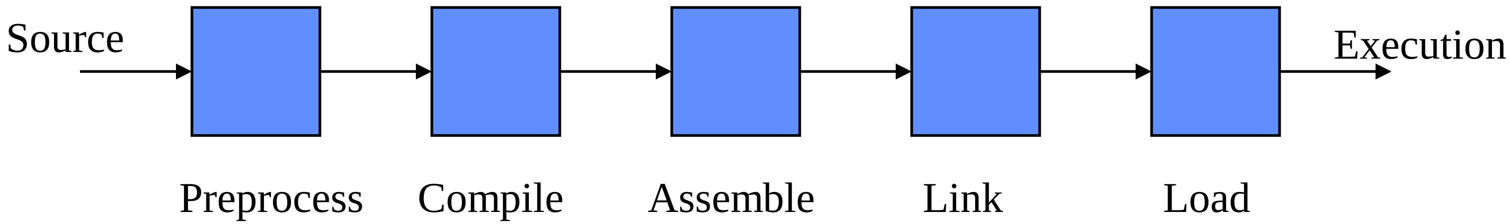
- What happens when you tell the compiler “go”?

- If you are like me:
  1. You wait 5 minutes for Netbeans to decide to do something
  2. You spend hours getting your paths correct
  3. You spend the rest of the day fixing a myriad of bugs
  4. About then, Windows crashes and you start over
  5. Eventually, if you ever fix all your bugs, you get it to compile

- If you ever get the code to compile, what steps typically happen?

Phases of Compilation

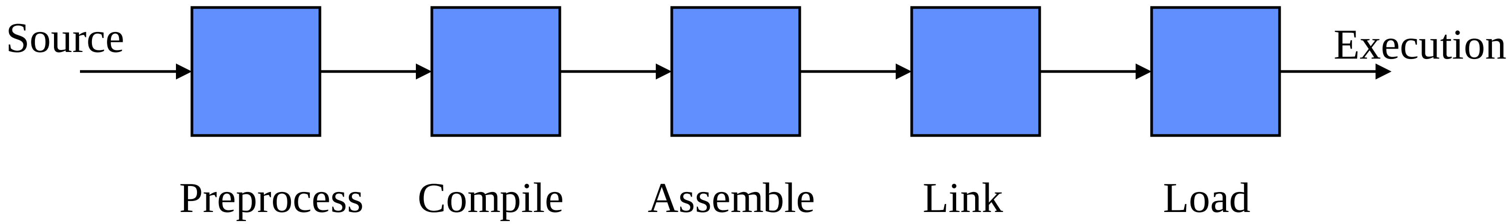
- Typical phases of preparing a C-type program for execution
  - Preprocessing
    - Eliminates compiler directives - #include, #define
  - Compilation - creates assembly language
  - Assembly – makes object code
  - Linker – takes code separately compiled into .o files (plus library files) and combines into a single executable file
    - Computes all the relative addressing for variables and functions
  - Loader – assigns memory space for program, starts up the process to run your program



Phases of Compilation

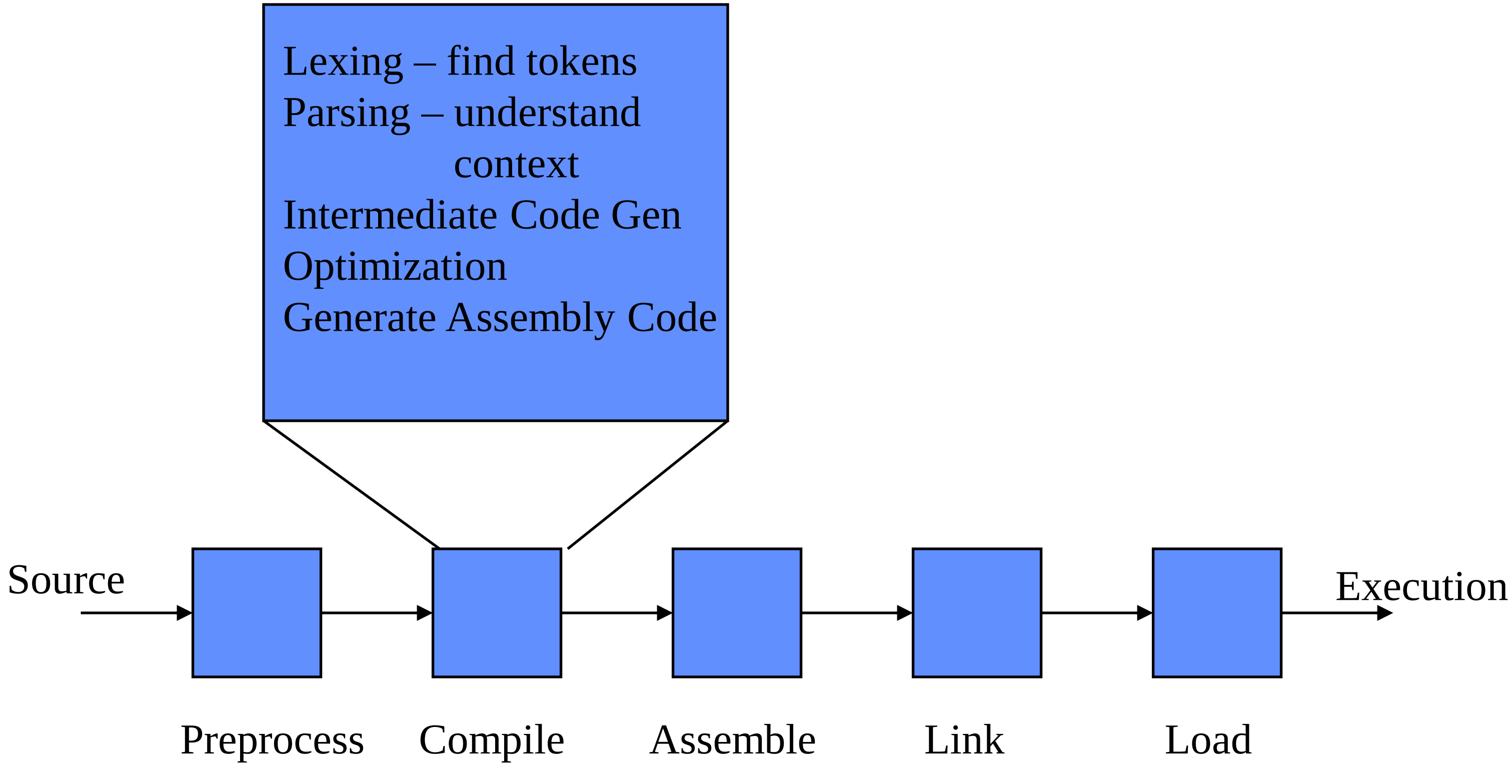
- So, what's the deal with java?
  - This stuff may make sense for C++ code, but does java do all this?
  - Which of the 5 steps are done to java code?

- Really only the compile phase is executed (by javac)
  - Not preprocessing is done (adding it has been discussed)
  - The "java" interpreter program is assembled, linked and loaded, but it directly "interprets" your .class file, so your code doesn't go through these phases



Compilation

- Let's focus just on the compilation phase
  - What goes on there?



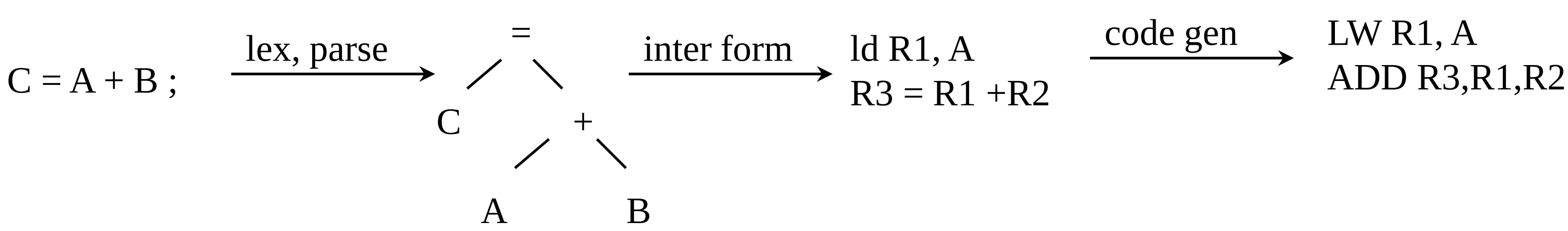
Compilation

- Compilation is typically accomplished in several passes
  - A pass is a scan through all lines of the code
- Look at code for Tiny compiler on pg 502-504
  - Note the stages of execution
  - How many passes does this compiler do?

parse ()	line 69
buildSymtab( )	line 77
typeCheck()	line 79
codeGen()	line 94

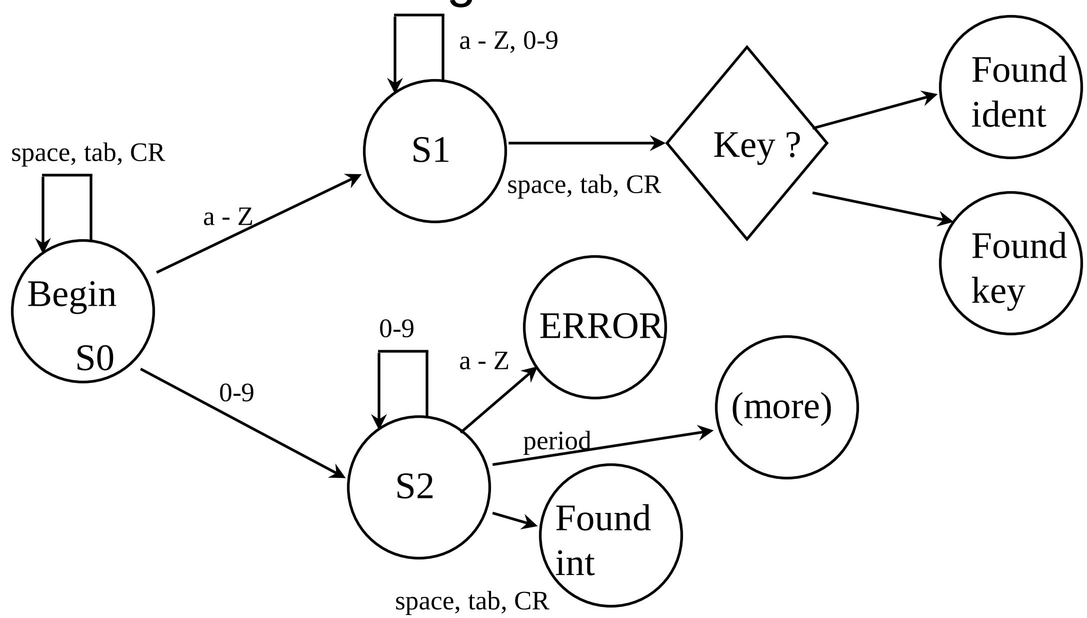
Compilation

- What has to be accomplished during compilation?
  - Note: input is ASCII text, output is ASCII text
  - Need to read in ASCII characters and get it in a form the compiler can operate on
    - Lexing, parsing
  - Translate this source-level representation into an intermediate form which looks more like individual instructions (RTL)
  - Optimize intermediate code
  - Translate intermediate code into assembly language

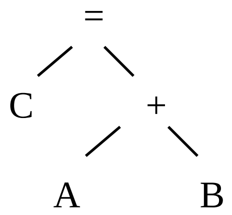


Lexing

- First step is *lexing* or *scanning*
  - Unix uses a tool called *lex*
  - Basically just tokenizes the input ASCII text
    - NewDist=Dist+10 becomes: ident assign ident plus int
      - Where *ident*, *assign*, *int* and *plus* are types of tokens
    - Done using finite automata or finite state machines



Parsing



- Next step is *parsing*
  - Take tokens and create a parse tree representation
  - Perform syntax checking
- Parsing is accomplished by comparing tokens to a predefined *grammar*
  - Grammar consists of a series of *productions*
    - One element of grammar (the LHS) converts into several other elements (the RHS)
  - Examples
    - assign_stmt -> ident assignOp expr ;
    - expr -> expr operator expr
    - expr -> ident

Parsing

The code being scanned is already in expanded form

- To determine if program syntax is correct, need to work backwards to see if it reflects legal productions

- Using the grammar, the parser scans through the tokens and tries to make legal *reductions*
  - Requires look ahead to make correct semantic decisions (and check for correct syntax)
    - D = A + B * C;
  - As we make reductions, we hook up data structures into parse tree

```
assign stmt -> ident assignOp expr ;
expr -> expr operator expr
expr -> ident
```

High Level Transformations

- Once we have a parse-tree (source-level) representation, compiler does a couple of steps to ensure the program is semantically correct
  - Biggest is type checking
- Next step is high-level code transformation
  - Profiling
  - Inlining

- Profiling
  - Instrument program to take data on
    - How often each block is executed
      - Can guide how much optimization to do
    - Frequency of branching
      - Effects polarity of branches used in assembly code

```
Incr_cnt(1);
for (i=0;i<n;i++){
   Incr_cnt(2);
   if (i<10) {
     [stmts]
     Incr_cnt(3);
    }
   else {
     [stmts]
     Incr_cnt(4);
   }
}
```

Inlining

- Inlining
  - Inserts callee function into caller
  - Purposes:
    - More code to optimize
    - Eliminates call overhead
    - Can optimize for fixed parameters

stmt1
call sub1 ()
stmt2
	sub1 ()
	{
	stmt3
	stmt4
	}
		stmt1
		stmt3
		stmt4
		stmt2

Before inlining

After inlining

Translation Into Low-Level Code

- So far we've read in the ASCII text, understood it syntactically/semantically, and done some code transformations on the high-level (parse tree) code
- Next we will translate the code into a low-level data format which looks more like assembly language
  - Must evaluate expressions in a semantically correct order
    - Done by walking parse tree in post-order and converting each expression into low-level language form
  - Example: A = (B+C) * D + F

Compilation

- Now we have low-level code, usually in some 3-operand register-transfer language (RTL)
  - Next step is to optimize the code
  - Classic code optimization examples
    - Common sub-expression elimination
      - D=A+B   C =A+B   -->   C = D
    - Copy propagation
      - D=C   A=D+1   -->   A = C + 1
    - Dead-code elimination
      - #define DEBUG 0   if (DEBUG) {}
    - Constant folding
      - A = 3   B = A + 4   -->   B = 7

Compilation

- Classic optimizations (cont)
  - Loop based
    - Loop-invariant code removal
      - { A[i] + B[3] }   -->   R1 = ld B[3] before loop
    - Load-store removal
      - { A = A + B[i] }   -->   R1=ld A   {R1=R1+B[i]} st A, R1
    - Strength reduction
      - ld (A + i*4) i++   -->   ld A+i i=i+4

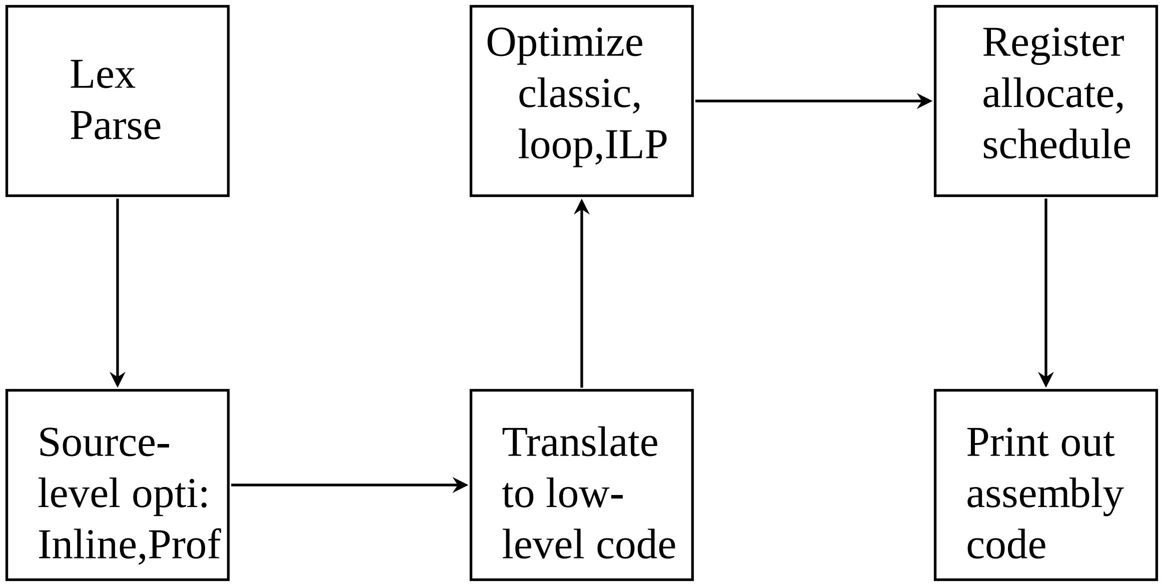
- Advanced optimizations - for speeding up superscalar and VLIW architectures
  - Loop unrolling
  - Superblock formation
  - Register renaming

Compilation

- Final stages of compilation
  - Register allocation
    - Map variables to actual hardware registers
  - Scheduling
    - Reordering code so it will flow through the target architecture pipeline smoothly (highest throughput)
  - Assembly generation - simple translation process

Compilation

- Summary of phases



Summary

- Where are we going – look at Table of Contents
  - Lexing – Chapter 2
  - Parsing – Chapter 3-5
  - Semantic/Type checking – Chapter 6
  - Code Generation – Chapter 7/8
  - Optimization