A Quick Introduction to Compilers

CS212 Fall '02
Tom Roeder
Overview of Compilation

- Generally separated into 6 phases:
  - Lexing: *character stream* -> *tokens*
  - Parsing: *tokens* -> *parse tree*
  - Symbol Table Generation: *parse tree* -> *parse tree*
  - Type Checking: *parse tree* -> *parse tree*
  - Code Generation: *parse tree* -> *code*
  - Optimization: *code* -> *code*

- If you think that you could do this in one pass, you're right, sometimes. (gcc was about 1.5 passes, last I heard :-))
Stages of Compilation: Lexing

- Taking a stream of characters and separating it into tokens, which can be passed to the next phase.
  - Uses Regular Expressions to extract the tokens from the input stream. In our project, Java is saving your neck, since CS211In allows you to simply check against the obvious possibilities.
  - Tom's favorite tools: lex, flex, jflex, and of course, ANTLR (http://www.antlr.org)
Stages of Compilation: Parsing

• Takes as input a stream of *tokens* and produces a *parse tree* to pass to the next stage of input
  - The tree might look something like the following: (see next 3 slides)
  - There are many ways to parse, and I don't have time in such a short lecture to describe any of them. They all, however, involve CFG's. Theorem in parsing which says that you can parse any CFG in O(n^3)
  - If you recognize any of these, pat yourself on the back: LL(k), LALR(1), Viable prefix recognizer
  - Tom's tools: bison, (b)yacc, ANTLR, jcup
A sample Bali program

main()
var i;
{
    i = 0;
}
return i;
program

mainfn

mainheader

amein ( )

namedecl

block

returnexpr

name

var

ame

statement

{ }

name

expression

integer

0

return

expression

name


A simpler way

```
program
mainfn
mainheader
"main()"
namedecl
"var"
name
"i"
;
block
{";
statement
name
"i"
"="
expression
integer
"0"
";
"
}";
returnexpr
"return"
expression
name
"i"
;
```

Stages of Compilation: Type Checking

- Takes as input the *parse tree* and outputs the same *parse tree* or a/some errors.
  - Check for silly statements like `int a = true;` for example.
  - We aren't really asking you to type check much, but it is the beginning of the difficult work in compilation.
  - Type-safe vs. non-type-safe languages (eg. C). Did you know you can make C ignore its type system in at least four simple ways? Exercise Left For Students
Stages of Compilation: Symbol Table Generation

- Takes a *parse tree* as input and annotates it with *symbol tables*. The output is the annotated parse tree.
  - Since in Bali there's only one level of scope, you don't really care about this phase
  - Usually implemented as souped-up hash tables.
  - If you've ever seen scope before, then you understand this
Stages of Compilation: Code generation

• Takes as input the parse tree and outputs code in whatever the output language of your compiler is. Sometimes returns a parse tree of our new object language code.
  – Usually we try not to be very smart here, 'cause we'll be cleaning up our stupidities in the next phase. You can do optimizations here, but it's not the common place to do them
  – Usually a simple translation suffices
  – Historical note: Common intermediate language compilation
Stages of Compilation: Optimization

• Takes as input the code in our object language, and outputs code in the same object language, which hopefully runs faster
  - Current research focuses on ways to do better and better versions of this: dataflow analysis, CVSE analysis, etc.
  - Get all sorts of cool math problems: register coloring for example
General structure of the CS212 compiler

• Top down or recursive-descent parsing
• Each non-terminal in the grammar has a function associated with it, and we just make calls to ourself recursively
• Note that our object orientation gains us little
• Do a left-to-right walk through the input stream and try to match it to the grammar. This only works well for certain types of grammars. Can anyone tell me why? What is a solution?