Compiler: Code Generation

Lecture 38
Abstract Syntax Tree (AST)

- Concrete parse tree: Faithful representation of each grammar rule application
  - Often contains syntactic clutter

- Abstract syntax tree: Faithful representation of structure of program
  - Only semantically important information is included
Parse Tree

\[ \text{MEAN} := \text{SUM} \div 100; \]
AST

MEAN := SUM DIV 100;

Diagram:

```
  :=
  /  
 id  DIV
   / 
 id  int
   / 
 id  SUM
   / 
 id  100
```
Code Generation

- Output produced from the AST
- Semantic routines: one routine per internal node in AST
- Two approaches:
  - Create entire tree, then transform and walk the tree, generating output
  - Generate output as the grammar rules are recognized, bottom up
Example

- Code snippet
  \[ \text{MID} := (\text{MAX} + \text{MIN}) \div 2 \]

- Grammar rule
  \[ \text{<term>} ::= \text{<term>} \div \text{<factor>} \]

- Semantic routine:
  - Needs results from children, e.g. registers which contain values being divided
  - Generates output: machine code for dividing
  - Returns location where result is placed, for its parent to use
An optimizing compiler tries to generate the most efficient object code:
- Time (fast execution times)
- Space (small object files)

Requires sophisticated analysis.

Often uses an intermediate representation of code:
- IR is not executed directly
- IR is analyzed for deciding register allocation, instruction ordering, branch shadows, etc...
Example: LLVM IR

```assembly
@.str = internal constant [14 x i8] c"hello, world\0A\00"

declare i32 @printf(i8*, ...)

define i32 @main(i32 %argc, i8** %argv) nounwind {
  entry:
    %tmp1 = getelementptr [14 x i8],
      [14 x i8]* @.str, i32 0, i32 0
    %tmp2 = call i32 (i8*, ...) @printf( i8* %tmp1 ) nounwind
    ret i32 0
}
Compiler Compilers

- Write:
  - Token definitions (REs)
  - Grammar definition (CFG)
  - Semantic routines (code to execute when visiting/generating the nodes of the tree)

- Use a tool to translate this information into a compiler (in C or Java or...)
  - Translation tool ➔ a compiler compiler!

- Classic unix tools:
  - Old school: lex and yacc ("lexical analyzer", "yet another compiler compiler")
  - Better: Gnu's flex and bison
  - Output a lexer and a compiler that calls the generated lexer
Modern Tool: ANTLR

- **ANother Tool for Language Recognition**
  - See: antlr.org, github.com/antlr/antlr4
  - Examples: github.com/antlr/grammars-v4
    (simple one: arithmetic.g4)

- Can generate code in many languages
  (Java, C#, Python, JavaScript, C++...)

- Two parts:
  - The tool (processes grammar to generate the lexer/parser)
  - The runtime (libraries for running the generated lexer/parser)
ANTLR: Depth-First Walking

https://tomassetti.me/antlr-mega-tutorial/
ANTLR: Listener Pattern
ANTLR: Visitor Pattern

https://tomassetti.me/antlr-mega-tutorial/
ANTLR Demo: Generate Parser

- Download antlr jar (in ../lib)
- Download github/antlr arithmetic.g4
- Generate parser
  
  ```
  $ java -jar ../lib/antlr-4.7.1-complete.jar \
  arithmetic.g4
  $ javac -cp ../lib/antlr-4.7.1-complete.jar \
  *.java
  ```
- Test parser
  
  ```
  $ java -cp :/../lib/antlr-4.7.1-complete.jar \
  org.antlr.v4.gui.TestRig arithmetic equation -gui
  # enter an equation followed by ^D
  34 < 4 * (2 + 2) - 3 ^ 4 ^ 5
  ```
ANTLR Demo: Code Generation

- See EvalVisitor.java
  - Inspect different methods
- Run main
  
  $ java -cp ...:/lib/antlr-4.7.1-complete.jar \n  EvalVisitor
Summary

- Code generation
  - Semantic routines for grammar rules
- IR and optimizations
- Compiler compilers: lex/yacc, flex/bison, antlr
  - Generate lexer and parser from grammar
  - You write semantic routines
  - Walking the tree: Listener vs visitor