Language Design and Optimizations

By – Louis Jenkins
Presentation Summary

• Create our own programming language
  • Demystify and Explore how they are created
    • Grammars, Lexers, Syntax Trees, etc.

• Define semantics for our language

• Interpret and run programs written in our language

• High-Level overview of program execution
  • Control Flow Graphs
Defining Our Language

• Domain Specific Language
  • Language created to solve a particular problem domain
    • Our domain is academic research
  • Opposite of a general-purpose language
    • C, C++, Java, Go, etc.

• Grammars
  • Defines our actual language
  • Lexers tokenizes input defined in the grammar
    • Lexer is also known as scanner or tokenizer
    • Determines syntactic correctness
  • Parsers infer meaning from sequences of tokens
    • Also determine syntactic correctness
  • Lexer and Parser provide different granularities in providing syntactic correctness.
Building the Parser

• YACC
  • Yet Another Compiler Compiler
  • A LALR Parser generator
    • Look Ahead Left To Right
    • Can look ahead K symbols to determine the right action to take.
  • Reads in BNF grammar
    • Backus-Naur Form is a context-free grammar capable of defining any language
      • Also known as meta syntax that can even define itself.

• Shift-Reduce Parser
  • \textit{Shift} pushes the symbol on the stack
  • \textit{Reduce} combines the symbols on the top of the stack into a single symbol if it satisfies a grammar rule.
  • Is a push-down automaton
The Grammar Pt.1

• **Lexer**
  - Also known as a *scanner* or *tokenizer*
  - Deals with parsing characters into streams of *tokens*
    • Tokens are the primitives that make up a language
      • Example: ‘var’, ‘this’, ‘if’, ‘else’, ‘while’, ‘for’
    • Tokens are also referred to as *lexemes*
  - Determines syntactic correctness
    • If input cannot be tokenized, it is syntactically invalid.
The Grammar Pt. 2

- Semantics
  - Parser Expressions
    - Snippets of code called upon reduction
      - Define semantics
    - Create our Abstract Syntax Tree here
      - Each node has defined with it semantics
        - Has an Action
          - AdditionBinaryASTNode performs type checking and handles addition of both expression operators.
Abstract Syntax Tree

- Tree representation of the syntactic structure of a program
  - Each node represents some constant in the source code
- Why is it Abstract
  - Does not contain all details, only what is important
    - Difference between ‘+’ and ‘AdditionBinaryASTNode’
- Applications and Uses
  - “Walking” the syntax tree
    - Also known as the ‘visitor’ pattern
    - Allows us to make interesting observations and compile-time checks
      - Type-Checking, Definitions, Etc.
      - Could even be *interpreted*
Interpreter – Executing our Language

• Symbol Table
  • Mapping from a name to it’s symbol
    • In the interpreter, the symbol keeps track of it’s value
  • We do not support lexical scoping
    • Only one instance of a variable name can exist in the program at one time.

• ‘Visitor’ execution
  • Walking the tree allows us to obtain the structure of the program
    • This structure can be used to interpret the intent behind the original instructions and executed
  • Similar style can be used to construct the Control Flow Graph
    • Each root of a statement subtree can be used to reconstruct the intent of the original statement
      • Each statement is a node with an edge to the next statement node.
    • But what about loops and conditional statements?
Control Flow Graph

• A directed graph that shows flow of control from one statement to another
  - Normal statements, Conditional Statements, Loops

• Basic Blocks
  - Sequence of statements that are *dominated* by a predecessor
    - A statement $s_1$ is said to *dominate* $s_2$ if all path of execution must flow through $s_1$ to reach $s_2$
      - Denoted $s_1 \text{ dom } s_2$
    - A basic block is a sequence ($s_1, s_2, \ldots, s_n$) such that $\forall i \in [1, N - 1] \ s_i \text{ dom } s_{i+1}$
  - A Control Flow Graph composed solely of basic blocks is said to be a *reduced* control flow graph.
var x = 0;
var hasReset = 0;
while (x < 5) {
    if (x == 4) {
        if (hasReset == 0) {
            x = 0;
            hasReset = 1;
        }
    }
    print "X: " + x;
    x = x + 1;
}
Language Test – Duff’s Device

- Duff’s device is a loop unrolling optimization that reduces the number of conditional evaluations in a loop
  - By unrolling the loop, we do not need to explicitly check on each pass
    - I.E: 1000 checks vs 1000 / N checks
      - N is the amount of the loop duplicated/unrolled on each iteration
        - In most cases 8
  - Trade-Off
    - Larger program size

- The Test
  - Implement loop unrolling in our language as the ultimate test
    - Tests everything needed to prove we are Turing Complete
Turing Completeness

- A language is Turing Complete if it can *simulate* a single-tape Turing-Machine
  - *read* and *write* values to a tape
    - Variables do precisely this.
      - Given a variable $X$ and a tape position $Y$, then the concatenation of the variable and position $XY$ can store information at that tape position.
        - 1\textsuperscript{st} Tape Position: $X1$
        - $N^{th}$ Tape Position: $XN$
    - To maintain state, a variable $q$ can be defined
  - Act conditionally based on current state and tape contents
    - Conditional statements do precisely this.
      - Wrap in a while loop
        - Now we can simulate the *halting problem*
    - Chain *if-else* statements to check current state
      - Read and write variables and transition states as needed

- Our Domain Specific Language is Turing Complete
Conclusion

• Recap
  • Created our own language by defining it’s grammar
  • Created the Abstract Syntax Tree from the grammar
    • Interpreted and ran ACTUAL code in our language by walking the syntax tree
  • Created the Control Flow Graph
    • Reduced it into basic blocks, handles loops and conditionals
    • Established Turing Completeness of our language

• Was it fun?
  • Yep
    • Taught myself everything in a compiler design course