Theory and Compiling

COMP360
“It has been said that man is a rational animal. All my life I have been searching for evidence which could support this.”

Bertrand Russell
Reading

Read sections 2.1 – 3.2 in the textbook
Language Recognition

• We say a theoretical machine can recognize a language if it can correctly identify a string of characters as being a syntactically correct program.

• If a theoretical machine can recognize a language, it can identify input that is not in the proper form, i.e. missing a semicolon or improper brackets.
Chomsky Hierarchy

• There are four classes of languages
• Each class is a subset of another class
• Regular languages are the simplest while recursively enumerable (RE) languages are the most complex
## Languages and Machines

<table>
<thead>
<tr>
<th>Language</th>
<th>Machine</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Regular Expressions</td>
<td>Deterministic Finite State Automata (DFA)</td>
<td>Regular Expressions</td>
</tr>
<tr>
<td>Context Free</td>
<td>Push Down Automata (PDA)</td>
<td>$a^n b^n$</td>
</tr>
<tr>
<td>Context Sensitive</td>
<td>Bounded Turing Machine</td>
<td>$a^n b^n c^n$</td>
</tr>
<tr>
<td>Recursively Enumerable</td>
<td>Turing Machine</td>
<td>Anything that can be recognized</td>
</tr>
</tbody>
</table>
Finite State Automata

• An FSA has circles to represent states and arrows to represent transitions between states

• Arrows are labeled with a character. That transition is taken if the FSA is in that state and the next input character matches the arrow label

• There is one or more final states that indicate the input matches the language

• If there is no matching transition for an input, the FSA rejects the input as a member of the language
Nondeterministic FSA

• In Computer Science theory, a Nondeterministic FSA (NFSA) can have multiple out arrows with the same label.

• An NFSA magically guesses which transition to take.

• Since an NFSA is no more powerful than a deterministic FSA *(and we don’t have magical programs)* we will only use deterministic FSA.

• Our FSA cannot have duplicate out arrows with the same label.
FSA for 3 Consecutive 1 inputs

• State A : no 1s detected
• State B : one 1 detected
• State C : two 1s detected
• State D : three 1s detected

Note that each state has 2 output arrows
Regular Expressions

• Regular expressions are often used to specify a regular language
• A regular expression defines a series of symbols in the order they can appear in the language
• Consider a very simple language with only the letters “a” and “b”
• \textit{aabb} is a regular expression for a string with two letters a followed by two b’s
Regular Expression Alternatives

• A regular expression can specify alternatives using the vertical bar character, | 
  • \texttt{aabb} | \texttt{ba} defines a regular language that accepts two a’s then two b’s or a string with just a b and an a
  • You can use parenthesis to group strings
  • (\texttt{aabb} | \texttt{ba})a defines a regular language that accepts \texttt{aabba} or \texttt{baa}
Regular Expression Repetitions

• A symbol or (group) might repeat multiple times in a valid string of the language

• A “*” indicates that a symbol or (group) repeats zero, one or many times

• A “+” indicates that a symbol or (group) repeats once or many times, but at least once

• \(a^{0..1}\) mean the symbol may appear zero or 1 time

• \((aabb \mid ba)^+a\) defines the strings \(aabbaabba, aabbbbaa, baa\) and more
Which strings are NOT \((aa \mid bb)^*a^+\)

A. a
B. bbaabba
C. aaa
D. aabb
E. bbbbbbbaaaaa
Write a Regular Expression

• Write a regular expression to define a string that begins and ends with an a and can have an even number of b’s between them
Possible Solution

• Write a regular expression to define a string that begins and ends with an `a` and can have an even number of `b`’s between them

\[ a \ (bb)\star \ a \]
Equivalent

• A FSA and a regular expression can define a regular language
• Regular expressions can be recognized by a regular language
Converting a Regular Expression to a FSA

- Consider the regular expression
  \[ a \ (bb) \* \ a \]

- It can be recognized by the FSA
Try it

• Write a regular expression that defines a double number, such as 12.3 or -12.3
• Use \texttt{n} to represent a numerical digit, 0..9
Possible Solution

• Write a regular expression that defines a double number, such as 12.3 or -12.3

• Use \textbf{n} to represent a numerical digit, 0..9

\[-0..1 \, n^+ \, . \, n^*\]
Degenerate Case

• 123. and .123 are both valid numbers
• A single period is not a valid number
• There must be a digit before or after the decimal point
• A possible solution might be

\[-0\.1 \ (n^+ \ . \ n^*) \ | \ (n^* \ . \ n^+)\]
FSA Limits

• The only “memory” that a FSA has is the current state
• As a FSA inputs symbols, it moves to different states
• A FSA cannot count or compare an arbitrary number of symbols
Push Down Automata

• A PDA has a FSA and a stack memory
• The top of the stack, input symbol and FSA state determine what a PDA will do
• A PDA can:
  • Push a new symbol on the stack
  • Pop the top symbol from the stack
  • Change the FSA state
Push Down Automata Languages

• A Push Down Automata (PDA) can recognize context free grammars
• Almost all modern programming languages are context free grammars
• A PDA can “count” things
• Nobody has developed a programming language more complicated than a context free grammar
Defining a Language

• For a program to be able to recognize a member of a language, you have to be able to accurately and unambiguously define the language.

• A regular language can be defined by a regular expression.

• A context free language can be defined by a BNF.
Backus-Naur Form

• Backus-Naur Form or **BNF** can be used to define Context Free Grammars

• Created by John Backus and Peter Naur. Independently created by Noam Chomsky

• BNF is a **metalanguage**, a language used to describe a language
BNF Fundamentals

• Terminal symbols are tokens or symbols in the language. They come from the lexical scanner.
• Nonterminal symbols are “variables” that represent patterns of terminals and nonterminal symbols.
• A BNF consists of a set of productions or rules.
• A language description is called a grammar.
BNF Structure

• Nonterminal symbols are sometimes enclosed in brackets to differentiate them from terminal symbols.

• I will use **bold font** for terminal symbols and no brackets.

• Productions are of the form:
  
  nonterminal → nonterminals or terminals

• Multiple definitions are separated by | meaning OR
  
  whatever → this | that
BNF to Define a Language

• There must be one nonterminal start symbol which must appear at least once on the left hand side of a rule
• The start symbol defines the language and all other rules follow from it
BNF Example

wloop  →  \textbf{while} ( logical ) stmt
logical → exp relation exp
relation → > | < | == | !=
stmt → \textit{something not defined here}
exp → \textit{something not defined here}
Compiler’s Purpose

• A compiler converts program source code into a form that can be executed by the hardware
• A compiler works with the language libraries and the system linker
Stages of a Compiler

• Source preprocessing
• Lexical Analysis (scanning)
• Syntactic Analysis (parsing)
• Semantic Analysis
• Optimization
• Code Generation
• Link to libraries
Source Preprocessing

• In C and C++, preprocessor statements begin with a #
• The preprocessor edits the source code based on the preprocessor statements
• `#include` is the same as copying the included file at that point with the editor
• The output of the preprocessor is expanded source code with no # statements
• Old C compilers had a separate preprocessor program
Lexical Analysis

• Lexical Analysis or scanning reads the source code (or expanded source code)
• It removes all comments and white space
• The output of the scanner is a stream of tokens
• Tokens can be words, symbols or character strings
• A scanner can be a finite state automata
Syntactic Analysis

• Syntactic Analysis or parsing reads the stream of tokens created by the scanner
• It checks that the language syntax is correct
• The output of the Syntactic Analyzer is a parse tree
• The parser can be implemented by a context free grammar stack machine
Semantic Analysis

• The Semantic Analysis inputs the parse tree from the parser
• Semantic Analysis checks that the operations are valid for the given operands (*i.e.* cannot divide a String)
• This stage determines what the program is to do
• The output of the Semantic Analysis is an intermediate code. This is similar to assembler language, but may include higher level operations
Optimization

• Most compilers will attempt to optimize the intermediate code
• Some compilers will also optimize after code generation
• There are many optimizations possible such as moving computations out of loops, avoiding redundant loads and stores, efficient use of registers, etc.
Code Generation

• The Code Generator inputs the intermediate language and outputs machine language for the target machine.
• The code generator is specific to the machine architecture.
Linking and Loading

• While not truly part of the compiler, the libraries provide the functionality that is more than just a few machine language statements

• The linker reads the object files and outputs an executable file
Simple Program

/* This is an example program */
A = Boy + Cat + Dog;
After Lexical Scan

A = Boy + Cat + Dog ;
Parsing

statement → variable = expression ;
variable → string of characters
expression → variable + expression |
variable
Semantic Analysis

• Semantic analysis will check
• Intermediate code defines a series of simple steps that will execute the program
• Example shown as text instead of an internal format

\[
\begin{align*}
\text{Temp1} &= B + C \\
\text{Temp2} &= \text{Temp1} + D \\
A &= \text{Temp2}
\end{align*}
\]
Simple Machine Language

• Load register with B
• Add C to register
• Store register in Temp1
• Load register with Temp1
• Add D to register
• Store register in Temp2
• Load register with Temp2
• Store register in A
Optimized Machine Language

• Load register with B
• Add C to register
• Store register in Temp1
• Load register with Temp1
• Add D to register
• Store register in Temp2
• Load register with Temp2
• Store register in A
Symbol Table

• Many stages of a compiler create and reference a symbol table
• The symbol table keeps a list of all of the names used in the program along with information about the name
• To assist debugging, the symbol table can be written into the output object file. This tells debuggers where variables are located
Reading

Read sections 2.1 – 3.2 in the textbook