Exam 1 Study Guide

CPSC 3520 - Brendan McGuire

Disclaimer: while I attempted to be as accurate as possible, I am a student just like you. If you see anything wrong or unclear here, please let me know and I will be sure to correct it!

Good luck today!

General Computing Topics

Moore's Law – The idea that computer transistor density doubles roughly every 18 months. This is a descriptive (rather than prescriptive) law; an observation about the rate of progress. In recent years, this has not held to be as constantly true

Babbage's Difference Engine – An early mechanical computer invented by Charles Babbage, designed in the 1820s.

ENIAC – Electronic Numerical Integrator and Computer, the first programmable general-purpose digital computer made in 1945 by IBM. It was used at large institutions, like finance, military, and government.

Types of Programming Languages

Programming languages come in my flavors, but many of them subscribe to one or more of the following ideologies for programming languages, different ways to write programs.

Imperative language	You tell the computer what to do directly: create this variable, assign it, add it to this other variable.	C, JavaScript, FORTRAN
Declarative language	You describe what the program should do (the problem domain), but do not describe how	Prolog
Functional language	These languages are oriented around applying and composing functions in a declarative way.	ML, CAML, Lisp
Rule-Based Language	Expresses facts and rules in formal logic to represent the problem domain	OPS5, clips, and soar
Event-Driven Language	The control flow of the program is determined by external events such as user interaction, network activity, or sensor readings.	nginx, Nodejs "Virtually all object-oriented languages are event-driven"
Parallel Language	Concerned with coordinating multiple threads and programs simultaneously, handling shared memory, messaging, and the inherent complexities of multi-threaded programming.	MPI/OpenMP

Dynamic Programming

A means of problem solving by breaking a large problem into smaller subproblems, where the large problem's solution is dependent on the solution of the smaller problem. Dynamic programming is mainly used as an optimization in cases where you might use recursion.

Grammars And Languages

At the heart of any sort of communication is language, a way of representing meaning between participants. We are particularly interested in this class about the mechanics of human-computer languages, which are designed to be understood by both humans and computers, as these have more strict and formalized rules.

Language – A system of symbols to mediate communication between entities, in particular humans and computers.

Grammar – The formal specification that defines the rules for a language

Alphabet

For a specific language, the alphabet is the list of symbols that compose all of the other tokens, all of the symbols that may appear in a program. For a lot of languages, this is the ASCII character set, but this is not necessarily the case.

alphabet – A set of symbols, usually denoted with V.

Notation & Terminology

A string over V is a sequence of symbols formed by concatenating zero or more symbols of V. The length of a string s is denoted |s|, and is the number of symbols in the string.

V^+ is the set of nonempty strings over the alphabet V. This means all of the strings of length 1 (V), length 2 (V^1), etc… unioned together. It is also known as the *positive closure* of V

$V^+ = V \cup V^2 \cup V^3 \cup \ldots$ *(Read: The set V+ is defined as all strings of length 1 unioned with all the strings of length 2, etc…)*

V^* is the same as the V^+ except it also includes the empty string ϵ. It is known as the closure of V.

Grammar

A grammar is composed of a number of elements used to distinguish it. The mathematical way of writing a grammar with all of its components is shown below

$$G = (V_T, V_N, P, S)$$

1. V_T – An alphabet of **terminals**. Terminals represent all of the elemental building blocks of the grammar and corresponding language. A C compiler might have an int terminal.

2. V_N – An alphabet of **non-terminals**. Non-terminals are intermediate symbols used in the generation (or analysis) of a program, but what wouldn't appear in the final representation. Similarly, a C compiler might have a *VariableDeclaration* non-terminal, which would include all of the symbols in the source code representing that variable declaration.

3. P – A list of production rules which define how we can rewrite symbols and define the structure of a grammar. **Important**: the production rules govern which strings can be formed in this grammar.

4. S – A start symbol that is representative of the entire program. The start symbol MUST be a non-terminal.

Production Rules

Because production rules define the structure of a grammar, it is worth understanding how they are written, and how they work. We use **Backus–Naur Form (BNF)** to notate production rules.

```
<decl> ::= e | <type> <variable-list>
<variable-list> ::= <variable> | <variable>, <variable-list>
<variable> ::= <identifier>
<command> ::= <variable> = <expr>
```

An example of production rules represented using Backus-Naur Form. The general structure lists an item on the left and an equivalence on the right.

If you are working with a restricted type of language (Context-Free Language – see below) then it is possible to put your production rules in **Chomsky Normal Form (CNF)**, which makes it possible to apply common parsing algorithms (such as Cocke-Younger-Kasami). All rules in CNF must be of the following two forms.

$$A \rightarrow BC$$

Rewrite two non-terminals in sequence as a single non-terminal

$$A \rightarrow a$$

Rewrite a terminal as a single non-terminal

Modes To Use Grammars

There are two main ways to use grammars, shown below with an explanation. The use of grammar is dependent on your application.

Generative – Using the rules of the language, the grammar to generate new programs (valid syntax in the language)

Analytic – Using the rules of the language to understand a given string, validate it against the grammar, and process it.

Chomsky Grammar Hierarchy

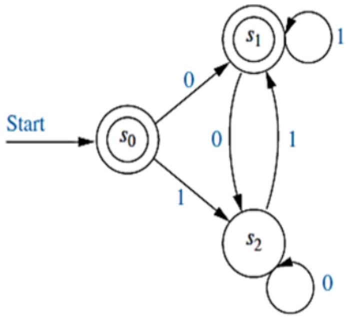
Not all grammars are made equal. We divide grammars into groups based on their inherent complexity. Not surprisingly, the more complex a grammar is, the harder it is to compute efficiently, so the most simple, restricted grammars are the easiest to work with.

T0	Turing Computable	The most complex grammars live here. T0 grammars are defined as those which can be parsed at all using a Turing Machine. This covers basically all known grammars you can comprehend.
T1	Context-Sensitive CSG	These grammars are allowed to have non-terminals on the left side of their production rules. For example, `aAb` → `aBb` (where a and b are non-terminals) This is a more powerful way to represent grammars, but parsing these grammars in an efficient way is still an open problem in computer science.
T2	Context-Free CFG	This is a more restricted set of grammars, ones whose production rules can be put into **Chomsky Normal Form** (CNF). Context-Free (and Regular) languages can be parsed using the CYK algorithm. A → BC A → a Context-Free Languages can be modeled using Push-Down Automata. Most programming and markup languages you know and love are found here.
T3	Regular	The most restricted (and thus easiest to parse) grammars are regular. This means they can be modelled using Finite State Machines (and regular expressions!) An important note about regular languages is that they cannot have any nested state. Examples of this kind of nested state is tags in HTML (you can have as many nested tags as you want), declarations in C/C++ (you can nest as many statement blocks as you want in C). Any language with this kind of nesting is **impossible to parse with a regular grammar**

Modeling Grammars

Finite State Machines - Regular Languages

Regular languages are so restricted, they can be represented using finite state machines, an abstract machine that can be in any one state at a time. It starts at one state, and each character in the string allows it to transition to a different state.



Regular Expressions

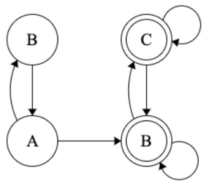
One way of compactly writing a Finite State Machine is using regular expressions, which is a compact way to represent regular grammars. These regular expressions can be used to parse regular grammars. A regular expression consists of the follow elements:

1) Atoms (essentially terminals) – matches a single symbol in the alphabet
2) Concatenation (two adjacent elements means that these elements must be present next to each other in the sequence)
3) Alternation (either this or that must appear) – represented using a vertical bar |
4) Kleene Star (zero or more copies of this must appear here) – represented using a star *
5) Parenthesis (used to group elements together to apply these in tandem)

```
(a|b)*(b|c)*
```

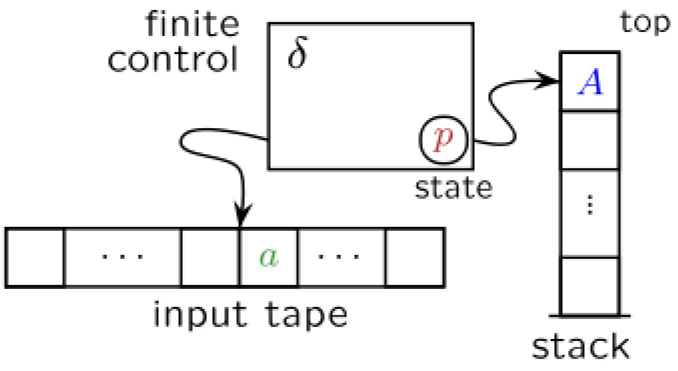
Group 1: matches a or b
Group 2: matches b or c

Groups may be repeated zero or more times



Push-Down Automata - Context-Free Languages

Context-Free Languages are not so simple as to be computable using a Finite State Machine, as there is additional complexity that needs modeling. A pushdown automaton is essentially a finite state machine with a stack, that allows you to have multiple states on the stack at once. This allows you to represent the more complex context-free languages



Parsing Grammars

What good are these grammar specifications if we can't do anything with them! Before parsing a grammar, however, it is important to think about what exact analysis/generation you are planning on doing. This may help you pick one of the following processing steps for grammars

Syntactic Analysis/Generation (flex)

This step is involved in understanding and parsing the physical form (syntax) of the language. Syntax analysis is not concerned with the meaning of the grammar, that analysis is reserved for semantic analysis later on. The end result of syntactic analysis is a **Parse Tree** (aka Derivation Tree). Alternatively, when generating, a parse tree is converted into syntax.

Lexing (Scanning) – The first step when parsing grammars. Lexing converts individual symbols (like i, n, t) into tokens that can be more easily acted upon (like int). This allows you to specify your grammar rules at a higher level of abstraction, and let the lexical analyzer take care of whitespace differences, commas, semicolons, and the like that don't really define the structure of your language. Tokens produced by the scanner are usually one of a few types: **identifiers** (like the name for a variable or function), **reserved word** (a special word in the language that has a specific meaning, like *int* or *auto* or class), or other simple symbols like brackets.

Syntax Parsing – Forming the series of tokens created by the scanner into a parse tree.

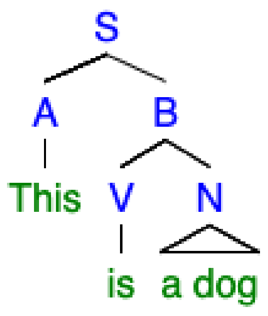
Semantic Analysis/Generation (bison)

This is the process of understanding the actual meaning of the program, using the parse tree from an earlier step. In a traditional programming language, this is where checks such as the type system would be enforced.

Parse Tree

Also known as a Derivation Tree, the Parse Tree represents a program as a tree of all of the elements in the program, for which it is easier to do semantic analysis. This is because similar syntax is grouped together. The rules that govern the parse tree are shown below:

1) The root of the parse tree is the start symbol S in V_N
2) The leaf (bottom) nodes are terminal symbols in V_T
3) Interior Nodes are non-terminals in V_N
4) The children of any non-leaf node represent the right-hand side (RHS) of some production rule in P, where the parent represents the left-hand side of that production rule (LHS)



Consider the parse tree for the English sentence “This is a dog.” At the root, we see the start symbol, which here represents the entire sentence. All of the nodes in blue represent non-terminals, while the leaf nodes in green are the terminals. Some production rules that might produce this parse tree from the sentence are shown below, in Chomsky Normal Form

S → AB
B → VN
A → this | a | the
V → is
N → dog | cat

<u>Top-Down & Bottom-Up</u>

There are two different ways to fill in the parse tree from the list of tokens produced by the scanner. The parse type you choose is dependent on the needs of your parser.

Top-Down Parse – Filling the parse tree from the top-down (starting at the root node and working down).

Bottom-Up Parse – Starting from the bottom leaf nodes, and gradually working our way up.

Cocke-Younger-Kasami Parsing Algorithm

The CYK Algorithm allows us to parse a Context Free Grammar, expressed in **Chomsky Normal Form** in O(n^3) proportional to the number of characters in the string. Remember that to be in Chomsky Normal Form, a CFG must have all of its production rules in one of the two following forms

A → BC	A → a
Rewrite two non-terminals in sequence as a single non-terminal	*Rewrite a terminal as a single non-terminal*

Extended Example

Consider the grammar defined by the following production rules. Is the string dldr in L(G), that is does it follow the rules of the grammar G? Follow along to find out!

S → RS | DA | r
A → LA | DS | l
L → l
R → r
D → d

The production rules, expressed in **Chomsky Normal Form**, will allow us to parse the string dldr.

D	L, A	D	R, S
d	l	d	r

In the first step, we fill in the bottom row of the CYK table. The bottom row represents all of the non-terminals that could be substituted for a single terminal at the bottom.

d (D)
l (L, A)
d (D)
r (R, S)

S	-	A	
D	L, A	D	R, S
d	l	d	r

Next, we fill in the second row of the table. These three columns represent all of the length 2 strings, and what they can be parsed to. All of the combinations possible for that row are shown, with their substitutions in parenthesis

DL, DA (S)
LD, AD
DR, DS (A)

-	A		
S	-	A	
D	L, A	D	R, S
d	l	d	r

Now, consider the third row of the table. These two boxes represent all of the length 3 substrings, and what they can be parsed as. As above, all combinations are shown, with their corresponding substitutions in parenthesis

SL, SA
LA (A), AA

S			
-	A		
S	-	A	
D	L, A	D	R, S
d	l	d	r

Finally, consider the top left box. This is all of the possible parsings for the entire string!

SA, DA (S)

Because the start symbol S is a possible parse in the top left box, the string matches the grammar! dldr is in the grammar defined by the production rules shown above!