CS 311: Computational Structures

James Hook

November 29, 2012

1 Overview

1.1 Course Goals (from course description)

Introduces the foundations of computing. Regular languages and finite automata. Context-free languages and pushdown automata. Turing machines and equivalent models of computation. Computability. Introduction to complexity.

The main goal of the course is that students obtain those skills in the theoretical foundations of computing that are used in the study and practice of computer science. A second goal is that students become familiar with Prolog as an experimental tool for testing properties of computational structures.

Upon the successful completion of this course students will be able to:

1. Find regular grammars and context-free grammars for simple languages whose strings are described by given properties.
2. Apply algorithms to: transform regular expressions to NFAs, NFAs to DFAs, and DFAs to minimum-state DFAs; construct regular expressions from NFAs or DFAs; and transform between regular grammars and NFAs.
3. Apply algorithms to transform: between PDAs that accept by final state and those that accept by empty stack; and between context-free grammars and PDAs that accept by empty stack.
4. Describe LL(k) grammars; perform factorization if possible to reduce the size of k; and write recursive descent procedures and parse tables for simple LL(1) grammars.
5. Transform grammars by removing all left recursion and by removing all possible productions that have the empty string on the right side.
6. Apply pumping lemmas to prove that some simple languages are not regular or not context-free.
7. State the Church-Turing Thesis and solve simple problems with each of the following models of computation: Turing machines (single-tape and multi-tape); while-loop programs; partial recursive functions; Markov algorithms; Post algorithms; and Post systems.

8. Describe the concepts of unsolvable and partially solvable; state the halting problem and prove that it is unsolvable and partially solvable; and use diagonalization to prove that the set of total computable functions cannot be enumerated.

9. Describe the hierarchy of languages and give examples of languages at each level that do not belong in a lower level.

10. Describe the complexity classes P, NP, and PSPACE.

Note: Not all topics will receive equal priority.

1.2 Outline

In this course offering I expect to present topics in approximately this order. I have do not yet have a detailed schedule. Professor Sheard and I are trying to focus the course on computational problems in an attempt to ground the subject in its motivation and application.

1. Computation, Algorithms, and Languages.
   (a) Binary addition as a calculation
   (b) Representing the calculation as a machine
   (c) Binary addition as a language
   (d) Languages as a lens to view computation
   (e) Making addition "faster"

2. Regular languages
   (a) Describing Finite Automata: From cartoons to structures
   (b) Non-deterministic finite automata
   (c) Equivalence of DFAs and NFAs
   (d) Language level properties (Closure properties)
   (e) The regular languages
   (f) Applications of regular languages
   (g) Fast addition revisited: can we speed up all DFAs?
   (h) Properties revisited: using language properties to show that some languages are not regular

3. Context-free languages
   (a) Recognizing palindromes
   (b) Push-down automata
   (c) Describing languages with context-free grammars
   (d) Ambiguity

   (e) PDAs can parse CFGs
   (f) The vocabulary of parsing (top-down, bottom-up)

4. Computability
   (a) A LISP-like language
   (b) A self-interpreter
       Perhaps use: `http://www.paulgraham.com/rootsoflisp.html`

5. Computability unplugged
   (a) When people were computers, reprise.
   (b) Turing's machine
   (c) Simple Turing machine programming
   (d) Variations on Turing machines

2 Lecture I: On Computation and Languages

2.1 Binary Addition

How do computers add numbers? The most common representation of numbers in computers is as a fixed width string of bits. For example, the simple sum $2 + 3 = 5$ is represented in binary as $010 + 011 = 101$.

There are two standard circuits for addition: the ripple-carry adder and the carry-lookahead adder. The ripple-carry adder is simpler, uses less hardware, but introduces greater delay. The carry-lookahead adder is more complicated, but introduces less delay. We first consider the ripple-carry adder.

The first problem to consider is how to add two bits. The first few cases are obvious:

$$0+0 = 0 \quad (1)$$
$$0+1 = 1 \quad (2)$$
$$1+0 = 1 \quad (3)$$
$$1+1 = ? \quad (4)$$

The last case is the interesting case because it generates a carry. So to characterize the sum of two bits we need two bits of result, the sum and the carry. We can represent that in the following truth table:

a	b	s	c
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Exercise 2.1 *What logical functions describe the calculation of s and c?*

$$\begin{aligned} s &= && (5)\\ c &= && (6)\\ h(a,b) &= (s,c) && (7) \end{aligned}$$

Now that we have described how to add 1-bit numbers, can we add 2-bit numbers? Suppose $a = a_1a_0$ and $b = b_1b_0$. We can use the adder function above to compute the sum and carry of a_0 and b_0, but then we must combine three numbers c, a_1, and b_1 to get the final answer. For this reason the simple adder above is generally called a *half-adder*. We need instead a *full-adder*:

a	b	c_{in}	s	c_{out}
0	0	0	0	0
0	1	0	1	0
1	0	0	1	0
1	1	0	0	1
0	0	1	1	0
0	1	1	0	1
1	0	1	0	1
1	1	1	1	1

(8)

Exercise 2.2 *Build a full-adder out of two half-adders. What additional logic is necessary? Can you write down logical formulas to compute s and c_{out} from a, b, and c_{in}?*

$$\begin{aligned} s &= && (9)\\ c_{out} &= && (10)\\ f(a,b,c_{in}) &= \mathit{let}\ \ (s',c') = h(a,b) && (11)\\ &\qquad\ \ \ (s'',c'') = h(s',c_{in}) &&\\ &\quad\ \ \mathit{in}\ \ (s'',\qquad\qquad) && \end{aligned}$$

From the full adder, we can build an adder for any fixed number of bits. To illustrate this, consider a 4 bit adder. Given $a = a_3a_2a_1a_0$ and $b = b_3b_2b_1b_0$, compute $s = s_3s_2s_1s_0$, where $s = a + b \bmod 2^4$. Since we want to "ripple" the carry all the way through the computation, we will fix $c_0 = 0$ as the "carry in" of the sum a_0 and b_0. We will label the "carry out" of a_i and b_i as c_{i+1}, which is the "carry in" of bits a_{i+1} and b_{i+1}. We can then build the 4 bit adder out of full adders as follows:

$$\begin{aligned} c_0 &= 0\\ (s_0,c_1) &= f(a_0,b_0,c_0)\\ (s_1,c_2) &= f(a_1,b_1,c_1)\\ (s_2,c_3) &= f(a_2,b_2,c_2)\\ (s_3,c_4) &= f(a_3,b_3,c_2) \end{aligned}$$

Exercise 2.3 *Draw this as a circuit (you can assume a full adder as a circuit element). Convince yourself that you can build an adder of any fixed width in this manner.*

If you want to guarantee that there is no carry out, how many bits are needed to represent the sum of two n-bit numbers?

2.2 Reflection

At this point we have described completely a very simple computation that is performed by every digital device. How can we talk about this calculation? How hard is it? How fast is it? How good is it? Is it biased towards sequential computation? Or is it essentially parallel?

Models of computation give a vocabulary to discuss some of these issues. This class is about models of computation. We will use this example to introduce a few key ideas about models.

The first observation is that we have described addition as a *computation.* We have given rules to calculate an output from inputs in a series of steps. Each step is well defined, and is easily checked.

The second, is that when we are thinking of circuits, the natural model of execution is parallel. In the circuit diagram you created in Exercise 2.3 you can imagine all the inputs $a_0, b_0, a_1, b_1, \ldots, a_3, b_3$ arriving at once, and the calculation flowing through the computational units as soon as all inputs are available.

Now look at the pattern of availability. Since the carry out of one bit is the carry in of the next most significant bit, no two full-adders ever operate at the same time. This circuit does not really achieve useful parallel execution. Only one full adder can be used at a time, since it must wait for the carry to be output from the previous stage. So if it takes one "adder delay" to compute each bit, then the total time to compute the 4-bit sum is 4 adder delays. It should be clear that this generalizes to any width. So an n-bit (ripple-carry) adder will take n adder delays to compute the answer.

This is an example of a *combinatorial circuit* model. A combinatorial circuit is an acyclic graph where every computational unit can be defined as a function by a truth-table mapping inputs to outputs. In particular, all computational element are stateless.

In a combinatorial circuit, the *inputs* are sources in the graph (nodes with no input edges). The *outputs* are sinks (nodes with no output edges).[1] For combinatorial circuits, the time to calculate the answer is the depth of the graph, that is, the length of the longest path from a source to a sink. If each full adder is considered 1 unit, then the depth of an n-bit adder is n.

Another measure of combinatorial circuits (and other parallel models) is how much *work* they do. The *work* of a parallel computation is the number of steps

[1]See Hein Section 1.4.1 for basic graph definitions. Note that an *acyclic graph* is a directed graph in which there are no cycles. A *source* is a node in a digraph with indegree 0. A *sink* is a node in a digraph with outdegree 0.

it takes to simulate it sequentially. The ripple-carry adder is very work efficient. It can add n-bit numbers doing approximately n full-adder work steps.

This raises the question: Is it possible to add n-bit numbers with a circuit of depth less than n? We will find that there are circuits that add numbers with less depth. The carry-lookahead adder, which we will discuss in the future, uses approximately $\log_2 n$ depth. However, it requires more than n work steps.

Studying models of computation will let us talk about concepts like:

- Is a problem inherently sequential?
- Can it be made parallel?
- When can we calculate a parallel computation from a sequential one?
- How can we characterize the trade-offs between solutions? (space vs. time, depth vs. work)

In this class we will introduce a few classic models of computation. These computational models will classify most computational problems, and articulate key features of their computational solutions.

2.3 Describing Addition by a Machine

In our description of circuits we described computation as a directed graph of circuit elements. In this section we begin to express this computation in a different form. We want to build a machine, called a Mealy machine, that will take as input the bit pairs to be added and produce as output the sum. So if we are doing $2 + 3$, which as 4 bit binary numbers is $0010 + 0011$, we wish to get the answer 5, which is 0101. Since we need to start with the low-order bits, we will actually compute our numbers "backwards", that is we will start with the rightmost bits, 0 and 1, which we write $(0, 1)$, and proceed to the leftmost bits. More formally, the input to our machine representing the problem above is:

$$(0,1)(1,1)(0,0)(0,0)$$

From this we wish to calculate the answer, lowest-order bit first:

$$1010$$

Mealy machines are a kind of *finite state* machines with output. Finite state machines are the simplest machines we study. Finite state machines can be described by giving a finite graph of the states, and labeling each transition with the input and output symbols that occur on that transition.

For the addition machine we will have two states, corresponding to the value of the carry, c. The states are 0 (no carry) and 1 (carry). We start in state 0 (no carry).

In state 0, when we read the first symbol, $(0, 1)$, we output a 1, since $0+1 = 1$. As there was no carry, we stay in the same state 0.

On the second symbol, $(1, 1)$, we output a 0, since $1 + 1 = 0$ with a carry. So we transition to state 1.

On the third symbol, $(0, 0)$, we output a 1, since $0 + 0 = 0$, but we had a carry in. Since we output the 1 we clear the carry, returning to state 0 (no carry).

We can represent this machine as a graph:

We can also represent this machine by giving the following specification, which we call M:

$$\begin{array}{lcll}
S & = & \{0,1\} & \text{States (finite set)} \\
A & = & \{0,1\} \times \{0,1\} & \text{Input Alphabet (finite set)} \\
B & = & \{0,1\} & \text{Output Alphabet (finite set)} \\
T(0,(0,0)) & = & 0 & \text{Transition function} \\
T(0,(0,1)) & = & 0 & \\
T(0,(1,0)) & = & 0 & \\
T(0,(1,1)) & = & 1 & \\
T(1,(0,0)) & = & 0 & \\
T(1,(0,1)) & = & 1 & \\
T(1,(1,0)) & = & 1 & \\
T(1,(1,1)) & = & 1 & \\
O(0,(0,0)) & = & 0 & \text{Output function} \\
O(0,(0,1)) & = & 1 & \\
O(0,(1,0)) & = & 1 & \\
O(0,(1,1)) & = & 0 & \\
O(1,(0,0)) & = & 1 & \\
O(1,(0,1)) & = & 0 & \\
O(1,(1,0)) & = & 0 & \\
O(1,(1,1)) & = & 1 & \\
0 & & & \text{Start state}
\end{array} \tag{12}$$

Note that this can be seen as a very simple kind of programming language for specifying Mealy machines. To give the machine, we list

1. a finite set of states, S,
2. a finite input alphabet, A,
3. a finite output alphabet, B,
4. a transition function, T, of type $S \times A \to S$,
5. an output function, O, of type $S \times A \to B$, and
6. a start state.

These elements become part of the formal definition of a Mealy machine introduced in the next section.

Given the definition of the machine M, we can now calculate its behavior on the input string $(0,1)(1,1)(0,0)(0,0)$ by using the transition function T applied to the start state to construct a state sequence:

$$\begin{array}{rcl} T(0,(0,1)) & = & 0 \\ T(0,(1,1)) & = & 1 \\ T(1,(0,0)) & = & 0 \\ T(0,(0,0)) & = & 0 \end{array}$$

From the state sequence and the input we construct the output with O:

$$\begin{array}{rcl} O(0,(0,1)) & = & 1 \\ O(0,(1,1)) & = & 0 \\ O(1,(0,0)) & = & 1 \\ O(0,(0,0)) & = & 0 \end{array}$$

Exercise 2.4 *Calculate the following sums:*

1. $0+0$

2. $7+7$

3. $5+2$

4. $7+1$

For each calculation give the binary representation of the numbers, the sequence of bit-pairs that are input to the machine (low order first), the sequence of states, the output sequence, and the decimal number represented by the output sequence.

2.4 Reasoning about machines

Having built the Mealy machine for addition, how do we gain confidence that it behaves as we expect? In this section we will explore how we can make the principles that informed our design more explicit, and how to use clear definitions to reason about the specific machine we have constructed. As we do this, we will lay the foundation for one of the recurring themes of the course: what we can learn about how hard a problem is by the kinds of machines that can solve it.

The first step in building confidence in the machine we constructed is to directly express the properties that informed its design. When we introduced the machine we explicitly stated that the machine state represents the carry state of the computation: 0 for no carry, 1 for carry. The transition and output functions are consistent with the specification of the full adder (Equation 8). This can be made even more explicit by giving equations that capture the properties. To express the equations we label the state c:

$$T(c,(a,b)) \quad = \quad c+a+b \text{ div } 2 \qquad (13)$$

$$O(c,(a,b)) \quad = \quad c+a+b \bmod 2 \qquad (14)$$

The operation a div b represents the greatest integer less than or equal to the quotient a/b. So 3 div 2 = 1, 2 div 2 = 1, 1 div 2 = 0 and 0 div 2 = 0. [2]

Exercise 2.5 *Verify these formulas by inspection of the definition of T and O in the definition of M in Equation 12.*

What property characterizes correctness of the machine? The original specification was of addition. Naively we might want the property:

Proposition 2.1 (Naive Add) *If $a = a_n a_{n-1} \dots a_0$ and $b = b_n b_{n-1} \dots b_0$ then M on input $(a_0, b_0)(a_1, b_1) \dots (a_n, b_n)$ generates output $s_0 s_1 \dots s_n$ and $s = s_n s_{n-1} \dots s_0$ represents the sum $a + b$.*

In the rest of this section we will review key concepts from set theory and logic that will enable us to state this precisely, and in a manner that address issues to the carry.

2.4.1 Alphabets, Strings, and Languages

All of the models of computation that are considered in this class characterize problems in terms of their symbolic representation. The basic symbols will be drawn from a finite set, called an alphabet.

Definition 2.2 (Alphabet) *An* alphabet *A is a finite, non-empty set.*

Any finite set will do. In the addition problem it was natural to take binary digits $\{0, 1\}$ as an alphabet. In some programming language contexts it may be desirable to regard keywords like "if" or "then" as single symbols in the alphabet. For some hardware-inspired problems we may want to regard bytes (units of 8 bits) as an alphabet.

What isn't an alphabet? We can't take the set of natural numbers as an alphabet because it is not a finite set.

Given an alphabet, we combine symbols to form strings. Informally, a string is just an ordered sequence of symbols. Taking binary digits as the alphabet the following are strings: $0, 1, 00, 01, 10, 11, \dots$. Strings can be concatenated to form other strings, for example, if we concatenate 00 with 11 we get 0011. Sometimes we will explicitly indicate concatenation with a binary operator, $\circ$, such as $00 \circ 11 = 0011$. However, like multiplication in standard algebra this occurs so frequently that when two symbols representing strings are written next to each other concatenation is implied. That is, if $x = 00$ and $y = 11$ then $xy = x \circ y = 0011$.

When we define strings we explicitly include the empty string, which Hein writes Λ (other texts write ϵ, and in lecture I may occasionally fall into that old habit). The empty string is a left and right identity of concatenation. That is, for any string x, $x = \Lambda \circ x = x \circ \Lambda$.

To build all the machinery we just described, we use the tools of set theory. First, following Hein Section 3.1.2, we define the strings over an alphabet by induction:

[2] See Hein, example 3.20 for an alternative presentation.

Definition 2.3 (Strings over A) *The set of strings over a finite alphabet A is denoted $A^\star$. It is defined inductively as follows:*

- *The empty string, Λ, is a string.*
- *If $s \in A^\star$ and $a \in A$ then $as \in A^\star$.*

In computer science we will frequently encounter situations where we want to name a set of strings. We call a set of strings a *language*. For example, the following are all languages over $\{0,1\}^\star$:

1. The string 01.
2. The set of strings that represent the number 2 in binary.
3. The set of strings that represent even numbers in binary.
4. The set of all strings of binary digits.
5. The set of strings that represent odd numbers.
6. The empty set.
7. The set of binary numbers that can be represented in 8 bits.

We will frequently make reference to the concept of language.

2.5 References

Hein discusses half-adders, full-adders, and Digital Circuits in Section 10.2.2. on pages 640 through 644.

3 Lecture 2: Finite Automata in More Detail

3.1 Lexical analysis of numbers

In the first lecture some of the examples were derived from the calculation of number from the digit sequence that represents it. In general, an n digit number can be written

$$d_{n-1}d_{n-2}\dots d_0$$

The value of the number in base b is given by

$$\sum_{i\in 0..n-1} d_i b^i$$

This form isn't very convenient for calculation. To write a program (or build a machine) to read digit sequences and compute values, it is helpful to factor this in the style of Horner's method (`http://en.wikipedia.org/wiki/Horner's_method`) This can be refactored as:

$$d_0 + b(d_1 + b(d_2 + b(\cdots + bd_{n-1})))$$

We can calculate this efficiently as we scan the digits $a_1a_2 \ldots a_n$ as follows:

$$\begin{aligned} v_0 &= 0 \\ v_{i+1} &= a_{i+1} + bv_i \end{aligned}$$

When this acts on the input it generates the number represented by the digits as v_n. For example, 120 in base 3 is interpreted:

$$\begin{aligned} v_0 &= 0 \\ v_1 &= 1 + 3v_0 = 1 \\ v_2 &= 2 + 3v_1 = 5 \\ v_3 &= 0 + 3v_2 = 15 \end{aligned}$$

This method of calculating the value of a number can be used to build finite automata that accumulate values with modular arithmetic. For example, in lecture we built a machine that accepts even numbers written in base 3. This machine has two states, 0 and 1, representing the two equivalence classes of numbers mod 2. It has an input alphabet with three symbols $0, 1, 2$. The transition function is $T(s, a) = a + 3 \times s \bmod 2$. The start state is state 0. Since we wanted even numbers, we designate state 0 as the final state. This machine was presented graphically in lecture on Tuesday.

Since the pattern of defining a machine is common, we define a notation for it. Making DFAs precise is the subject of the next section.

3.2 Definition of Deterministic Finite Automata

Definition 3.1 (DFA) *A Deterministic Finite Automaton (DFA) is defined by a 5-tuple, $\langle S, A, T, s, F\rangle$, where S is a finite, non-empty set (states), A is a finite non-empty set (alphabet), $T : S \times A \to S$ (the transition function), $s \in S$ (start state), and $F \subseteq S$ (final states).*

In this form, we define the machine for even numbers base three as:

$$M = \langle \{0, 1\}, \{0, 1, 2\}, T, 0, \{0\}\rangle \quad \text{where } T(s, a) = a + 3 \times s \bmod 2$$

DFAs define languages by the set of strings they accept. We make that concept precise with the following definition:[3]

Definition 3.2 (DFA Acceptance) *A DFA $M = \langle S, A, T, s, F\rangle$ accepts a string $w = a_1a_2 \ldots a_n$ if there is a sequence of states $r_0, r_1, \ldots, r_n$ such that:*

1. $r_0 = s$
2. *For each* $i \in \{0, \ldots, n-1\}$, $T(r_i, a_{i+1}) = r_{i+1}$
3. $r_n \in F$

[3]This definition is similar to Hein, page 722, but most closely follows Sipser's text.

To illustrate this, consider the action of M on the string 121. Since T is a function, the input and the start state completely determine the state sequence (this will not be the case for all models we study). We calculate it as follows:

$$\begin{aligned} r_0 &= 0 \\ r_1 &= T(0,1) = 1 \\ r_2 &= T(1,2) = 1 \\ r_3 &= T(1,1) = 0 \end{aligned}$$

Now we verify the three properties of acceptance: (1) follows immediately by choice of r_0. (2) is established by the calculation of r_1 to r_3. (3) is established since $0 \in \{0\}$. Hence M accepts 121.

Exercise 3.1 *Experiment with the definition on some more strings. Try:*

1. Λ
2. 0
3. 1
4. 12
5. 11
6. 111

Having defined the action of a DFA on a string, we can now talk about the language defined by a DFA.

Definition 3.3 *The language defined by DFA M, written $\mathcal{L}(M)$, is the set of strings accepted by M. That is:*

$$\mathcal{L}(M) = \{w | M \textit{ accepts } w\}$$

Exercise 3.2 *Construct DFAs for the following languages:*

1. *The empty language.*
2. *The language of all strings, $A^\star$.*
3. *Bit strings starting with* 1.
4. *Bit strings ending with* 0.
5. *Bit strings starting with* 1 *or ending with* 0.
6. *Bit strings starting with* 1 *and ending with* 0.
7. *Bit strings containing the substring* 011. *Test this one on* 01011.

3.3 Mealy definitions

Definition 3.4 (Mealy Machine) *A Mealy machine is a 6-tuple, $M = \langle S, A, B, T, O, s\rangle$ where S, A, and B are finite, non-empty sets, (states, input alphabet, and output alphabet respectively), $T : S \times A \to S$ is the transition function, $O : S \times A \to B$ is the output function, and $s \in S$ is the start state.*

Mealy machines are string transducers. That is, they define a mapping from strings over A to strings over B.

Definition 3.5 (Mealy output) *A Mealy machine $M = \langle S, A, B, T, O, s\rangle$ outputs $w = b_1 b_2 \ldots b_n$ on input $x = a_1 a_2 \ldots a_n$ if there is a sequence of states $r_0, r_1, \ldots r_n$ such that*

1. $r_0 = s$
2. $T(r_i, a_{i+1}) = r_{i+1}$ *for* $0 \leq i < n$
3. $O(r_i, a_{i+1}) = b_{i+1}$ *for* $0 \leq i < n$

Exercise 3.3 *Illustrate the definition of output of a Mealy machine on an example from Lecture 1.*

3.4 Correctness of Mealy Machine for addition

3.4.1 Some preliminaries

Since the addition machine processes strings of digits low-order bit first, we need a function that scans a string with this interpretation. This function is defined by the recursion principle on strings introduced by Hein in Section 3.2.2.

$$\begin{aligned}\text{num } \Lambda &= 0 \\ \text{num } ax &= a + 2 \times \text{num } x \quad \text{where } a \in A \text{ and } x \in A^\star\end{aligned}$$

To get strings of digits from strings of pairs of digits we define two more helper functions:

$$\begin{aligned}\text{fst } \Lambda &= \Lambda \\ \text{fst } (a,b)x &= a(\text{fst } x) \\ \text{snd } \Lambda &= \Lambda \\ \text{snd } (a,b)x &= b(\text{snd } x)\end{aligned}$$

To project a string of bit pairs as a number we combine these with the shorthand notation of indexed brackets ($[\cdot]_a$):

$$\begin{aligned}[x]_a &= \text{num } (\text{fst } x) \\ [x]_b &= \text{num } (\text{snd } x)\end{aligned}$$

Exercise 3.4 *Build an input string for the addition machine representing a few simple sums, perhaps $2+5$ and $3+5$. Verify that you can recover the intended arguments by calculation from the input string and the projections above.*

Calculate the output string generated by the Mealy machine for addition, M, developed in Exercise 2.5. Confirm that it computes the sum.

Exercise 3.5 *Verify the equation $v = (v \mathit{mod}\, 2) + 2(v\ \mathit{div}\, 2)$.*

The desired correctness property can be stated:

Proposition 3.6 *If M on input x outputs w then $[x]_a + [x]_b = \mathit{num}\, w\ \mathit{mod}\, 2^{|x|}$.*

We want to prove this property by induction on the input string, x. If we try this the base case will work, since

$$[\Lambda]_a + [\Lambda]_b = 0 + 0 = 0 = \text{num } \Lambda$$

However, we fail to be able to prove the step since we can't apply the induction hypothesis in the case where a carry is generated (that is, on the input symbol $(1,1)$.

To overcome this, we must generalize the proposition:

Proposition 3.7 *If M in state c on input x outputs w then $c + [x]_a + [x]_b = \mathit{num}\, w\ \mathit{mod}\, 2^{|x|}$.*

The proof is by induction on x.

Base case: $x = \Lambda$. In this case we must show $c+[\Lambda]_a+[\Lambda]_b = \text{num } \Lambda \text{ mod } 2^0$. Since $2^0 = 1$ and all numbers are congruent to 0 mod 1, this equation is trivially satisfied.

Step case: $x = (a,b)x'$. For the step we must prove $c + [x]_a + [x]_b = \text{num } w \text{ mod } 2^{|x|}$ given that for any c', $c' + [x']_a + [x']_b = \text{num } w' \text{ mod } 2^{|x'|}$, where $w = dw'$.

Since the machine begins in state c, we can compute the next state, c' using the transition function: $c' = T(c,(a,b)) = c + a + b \text{ div } 2$. Similarly we can compute the output $d = O(c,(a,b)) = c + a + b \text{ mod } 2$. This lets us reason as follows:

$$\begin{aligned}
c + [x]_a + [x]_b &= c + [(a,b)x']_a + [(a,b)x']_b \text{ mod } 2^{|x|} \\
&= c + (a + 2[x']_a) + (b + 2[x']_b) \text{ mod } 2^{|x|} \\
&= c + a + b + 2([x']_a + [x']_b) \text{ mod } 2^{|x|} \\
&= (c + a + b \text{ mod } 2) + 2(c + a + b \text{ div } 2) + 2([x']_a + [x']_b) \text{ mod } 2^{|x|} \\
&= d + 2c' + 2([x']_a + [x']_b) \text{ mod } 2^{|x|} \\
&= d + 2(c' + [x']_a + [x']_b) \text{ mod } 2^{|x|} \\
&= d + 2(\text{num } w') \text{ mod } 2^{|x|} \\
&= \text{num } (dw') \text{ mod } 2^{|x|} \\
&= \text{num } w \text{ mod } 2^{|x|}
\end{aligned}$$

Note that although the induction hypothesis used a different modulus, since it was used in a context multiplied by 2 the modular reasoning is sound. This calculation establishes the desired property, completing the proof.

Exercise 3.6 *Annotate every line of the equational calculation with the rule that justifies it.*

Exercise 3.7 *Having proved Proposition 3.7, show that the original property in Proposition 3.6 holds.*

3.5 Other Sources

Sipser, *Introduction to the theory of Computation, Second Edition*, gives a very concise and elegant presentation of this material.

Hopcroft, Motwani, and Ullman, *Introduction to Automata Theory, Languages, and Computation, Second Edition*, presents a detailed development.

Both texts are excellent.

4 Lecture 3: Simple Closure Properties

In Lecture 2 we saw that we can talk about the language recognized by a DFA. Such languages are called regular.

Definition 4.1 *A language is* regular *if it is accepted by a DFA.*

In this lecture we explore the structure of the regular languages as a class of languages. In particular, we are focusing on operations on languages (not just strings). We are asking what operators on languages, when applied to regular languages, will yield a regular language. Such properties are called *closure properties*. For example, we say that the regular languages are closed under complement because if L is a regular language then the complement of L is regular.

4.1 DFAs under Complement, Union and Intersection

Complement

Definition 4.2 *If L is a language over A, i.e. $L \subseteq A^\star$, then the* complement *of L, $\bar{L}$ is $\{x \in A^\star | x \notin L\}$.*

Proposition 4.3 *If L is a regular language then $\bar{L}$ is a regular language.*

Proof: Since L is regular, $L = \mathcal{L}(M)$ for some DFA $M = \langle S, A, T, s, F\rangle$. Construct M' to recognize $\bar{L}$ by taking the final states to be the relative complement of F with respect to S. That is, $M' = \langle S, A, T, s, (S - F)\rangle$.

Before proving that $\mathcal{L}(M') = \bar{L}$, note that for a DFA the state is uniquely determined by the start state and the input.

To show $\mathcal{L}(M') = \bar{L}$, we will show that $\mathcal{L}(M') \subseteq \bar{L}$ and $\mathcal{L}(M') \supseteq \bar{L}$. Consider $x \in \mathcal{L}(M')$. By definition there is a sequence of states satisfying acceptance ending in a state r_n in $S - F$. On input x machine M reaches the same state. Since $r_n \notin F$ machine M rejects x.

Similarly, consider $x \in \bar{L}$. Since x is in $\bar{L}$ we know that there is not a sequence of states for machine M witnessing acceptance. Furthermore, since the sequence of states is uniquely determined and T is total, there is a sequence $r_0, r_1, \ldots, r_n$ that satisfy conditions 1 and 2 of acceptance but not 3. Specifically $r_n \notin F$. Hence, $r_n \in S - F$. Hence M' accepts x, as required.

This completes the proof.

Intersection The next operator we consider is the intersection of two regular languages. We want to show that if L_1 and L_2 are regular languages then $L_1 \cap L_2$ is regular.

The basic idea is that we will take two DFAs, M_1 and M' for L_1 and L_2 respectively, and construct a new machine N to recognize the intersection. Machine N will essentially simulate both M_1 and M_2 step by step. It will accept if both machines accept.

To implement the simulation we will build the states of N out of ordered pairs of states of M_1 and M_2.

More formally: Since L_1 and L_2 are regular languages, there are DFAs $M_1 = \langle S_1, A, T_1, s_1, F_1 \rangle$ and $M_2 = \langle S_2, A, T_2, s_2, F_2 \rangle$ such that $L_1 = \mathcal{L}(M_1)$ and $L_2 = \mathcal{L}(M_2)$. Construct a DFA $N = \langle S_1 \times S_2, A, T, \langle s_1, s_2 \rangle, F_1 \times F_2 \rangle$, where $T(\langle r_1, r_2 \rangle, a) = \langle T_1(r_1, a), T_2(r_2, a) \rangle$.

Claim: $\mathcal{L}(N) = L_1 \cap L_2$. We show this equality by showing the two containments:

Subclaim 1: $\mathcal{L}(N) \subseteq L_1 \cap L_2$. Consider $x \in \mathcal{L}(N)$. By definition, there exists a sequence of states $\langle r_{10}, r_{20} \rangle, \langle r_{11}, r_{21} \rangle, \ldots, \langle r_{1n}, r_{2n} \rangle$. That satisfies the three conditions of acceptance: (1) $\langle r_{10}, r_{20} \rangle = \langle s_1, s_2 \rangle$, (2) all transitions obey function T, and (3) $\langle r_{1n}, r_{2n} \rangle \in F_1 \times F_2$. From this, we conclude that $x \in \mathcal{L}(M_1)$ and $x \in \mathcal{L}(M_2)$ since the state sequence $r_{10}, r_{11}, \ldots, r_{1n}$ satisfies the conditions for acceptance of M_1 and the state sequence $r_{20}, r_{21}, \ldots, r_{2n}$ satisfies the conditions for acceptance of M_2.

Subclaim 2: $L_1 \cap L_2 \subseteq \mathcal{L}(N)$. Consider $x \in L_1 \cap L_2$. In that case $x \in \mathcal{L}(M_1)$ and $x \in \mathcal{L}(M_2)$. The definition of acceptance gives state sequences witnessing acceptance, $r_{10}, r_{11}, \ldots, r_{1n}$ and $r_{20}, r_{21}, \ldots, r_{2n}$. These can be combined into the state sequence for N: $\langle r_{10}, r_{20} \rangle, \langle r_{11}, r_{21} \rangle, \ldots, \langle r_{1n}, r_{2n} \rangle$. Hence, $x \in \mathcal{L}(N)$, as required.

The proofs of the subclaims establish the claim. This completes the proof that the regular languages are closed under intersection.

Union The construction for intersection can easily be adapted to compute the union of two languages, L_1 and L_2. The simulation machinery is identical. The only change is the set of final states. In this case the goal is to accept if either machine accepts, not both. This changes the selection of the final states for N to states that have a final state of M_1 or a final state of M_2. In symbols, $F_1 \times S_2 \cup S_1 \times F_2$.

Exercise 4.1 *Complete the details of the construction and proof that the regular languages are closed under union.*

4.2 Nondeterminism and Λ transitions

The next closure operation we wish to consider is the concatenation of languages.

Definition 4.4 *The* concatenation *of languages L_1 and L_2, written $L_1 \circ L_2$, or when clear from context $L_1 L_2$, is $\{xy | x \in L_1, y \in L_2\}$.*

Exercise 4.2
1. *If L_1 is the language consisting exactly of the empty string what is $L_1 \circ L_2$?*
2. *If L_1 is the empty language what is $L_1 \circ L_2$.*

Although it is possible to build a concatenation operation directly on DFAs, it is inconvenient. The construction is significantly simpler if we relax the transition relation from a function to a relation. With this change the start state and input no longer uniquely determine the state. We construct NFAs in two steps:

Definition 4.5 (NFA version 1) *A Nondeterministic Finite Automaton (NFA) is defined by a 5-tuple, $\langle S, A, T, s, F\rangle$, where S is a finite, non-empty set (states), A is a finite non-empty set (alphabet), $T : S \times A \rightarrow$ power (S) (the transition relation), $s \in S$ (start state), and $F \subseteq S$ (final states).*[4]

The only change from the definition of a DFA is the type of the transition function. It now returns a (possibly empty) set of states. It is no longer a total function.

Definition 4.6 (NFA version 1 Acceptance) *An NFA $M = \langle S, A, T, s, F\rangle$ accepts a string $w = a_1 a_2 \dots a_n$ if there is a sequence of states $r_0, r_1, \dots, r_n$ such that:*

1. *$r_0 = s$*
2. *For each $i \in \{0, \dots, n-1\}$, $r_{i+1} \in T(r_i, a_{i+1})$*
3. *$r_n \in F$*

Again, the only change in the definition of acceptance is related to the transition function. However, notice that now the state is no longer uniquely determined by the machine and the input.

[4]Following Hein (page 17) I'm using power (X) for the power set of set X. Other common notations are $\mathcal{P}(X)$ and 2^X.

Exercise 4.3 1. *Build an NFA that accepts strings over* $\{0, 1\}$ *with a* 1 *two symbols from the right. For example, accept* 100*.*

2. *Illustrate the definition of acceptance on the strings* 111*,* 00100*, and* 111111*.*

An additional convenience mechanism can be added to non-deterministic automata, allowing transitions to be made without consuming any input. These transitions are called Λ-transitions (also known as ϵ-transitions in some texts). This change is also made by changing the type of the transition function, T. For NFAs with Λ transitions the T now accepts a symbol from A or a Λ. This is written $T : S \times (A \cup \{\Lambda\}) \rightarrow$ power (S).

This change requires we revise the definition of acceptance to allow transitions on Λ's as well as on elements of A.

The complete definitions are repeated here:

Definition 4.7 (NFA) *A Nondeterministic Finite Automaton (NFA) is defined by a 5-tuple,* $\langle S, A, T, s, F\rangle$*, where* S *is a finite, non-empty set (states),* A *is a finite non-empty set (alphabet),* $T : S \times (A \cup \{\Lambda\}) \rightarrow$ *power* (S) *(the transition relation),* $s \in S$ *(start state), and* $F \subseteq S$ *(final states).*

Definition 4.8 (NFA Acceptance) *An NFA* $M = \langle S, A, T, s, F\rangle$ *accepts a string* $w = a_1a_2 \ldots a_n$*, where* $a_i \in A \cup \{\Lambda\}$*, if there is a sequence of states* $r_0, r_1, \ldots, r_n$ *such that:*

1. $r_0 = s$

2. *For each* $i \in \{0, \ldots, n-1\}$*,* $r_{i+1} \in T(r_i, a_{i+1})$

3. $r_n \in F$

Exercise 4.4 1. *Draw the NFA* $M = \langle\{0, 1\}, \{a, b\}, T, 0, \{1\}\rangle$ *where* $T(0, \Lambda) = \{1\}$*.*

2. *Illustrate the definition of acceptance by showing* M *accepts* Λ*.*

3. *What does the machine do on input* a*?*

Exercise 4.5 *Construct an NFA that accepts stings over* $\{a, b\}$ *that end with an* a *followed by five other symbols. Illustrate on a few examples. Can you build a DFA for this language? Would it be more complicated?*

Exercise 4.6 *Construct an NFA to recognize strings with the substring abb. How does this compare to the DFA? Is it necessary to manage failure transitions explicitly?*

4.3 Concatenation and Iteration

We have not yet proven any relationship between the languages we can recognize with NFAs and the languages we can recognize with DFAs, although it should be obvious that any language recognized by a DFA can be recognized by an NFA since DFAs are just very boring NFAs.

Before we address this issue, we show that languages defined by NFAs are closed under two important operations, concatenation and iteration. In lecture I will do this with pictures and cartoons. In these notes I'm including a few details of the construction.

The idea of the concatenation construction is to build a machine for $L_1 \circ L_2$ from a machine for L_1 and a machine for L_2 by adding Λ transitions from the final states of the first to the start state of the second, and keep the final states of the second as the final states of the concatenation machine.

Proposition 4.9 *If L_1 and L_2 are languages accepted by an NFA then $L_1 \circ L_2$ is a language accepted by an NFA.*

Proof: Since L_1 and L_2 are accepted by NFAs, there are machines M_1 and M_2 (with disjoint sets of states) accepting them. Let $M_i = \langle S_i, A, T_i, s_i, F_i\rangle$. Construct a machine N to recognize the concatenation as follows: $N = \langle S_1 \cup S_2, A, T, s_1, F_2\rangle$ where T is defined:

$$\begin{array}{lcll r}
T(r,a) & = & T_1(r,a) & \text{if } r \in S_1 - F_1 & (1)\\
T(r,a) & = & T_1(r,a) & \text{if } r \in F_1 \text{ and } a \neq \Lambda & (1)\\
T(r,\Lambda) & = & T_1(r,\Lambda) \cup \{s_2\} & \text{if } r \in F_1 & (2)\\
T(r,a) & = & T_2(r,a) & \text{if } r \in S_2 & (3)
\end{array}$$

Claim: $\mathcal{L}(N) = \mathcal{L}(M_1) \circ \mathcal{L}(M_2)$

Subclaim 1: $\mathcal{L}(N) \subseteq \mathcal{L}(M_1) \circ \mathcal{L}(M_2)$

Consider $x \in \mathcal{L}(N)$. By definition, there is a sequence of states $r_0, r_1, \ldots, r_n$ witnessing acceptance of $x = a_1a_2\ldots a_n$, where $a_i \in A \cup \{\Lambda\}$. By property 1 of acceptance, we have $r_0 = s_1$. By property 2 we know that all transitions satisfy T. By property 3 we know that $r_n \in F_2$. By construction of T we observe that transitions of type (1) map from S_1 to S_1, transitions of type (3) map from S_2 to S_2. Transitions of type (2) are all Λ transitions. They can either be from S_1 to S_1 by transitions consistent with T_1 or from S_1 to S_2 by the transitions explicitly added by the construction. Hence, any state transition sequence will consist of some number of type (1) or type (2) transitions within S_1, a final type (2) transition to state s_2 (in S_2), followed by some number of type (3) transitions within S_2. Let i be the index of the last state from S_1. Then $r_i \in F_1$ and $r_{i+1} = s_2$. Let $x_1 = a_1a_2\ldots a_i$ and $x_2 = a_{i+2}a_{i+3}\ldots a_n$. Note that $x = x_1\Lambda x_2 = x_1x_2$. Now observe that $x_1 \in \mathcal{L}(M_1)$ because every transition is a T_1 transition, the start state is s_1 and the $r_i \in F_1$. Similarly, note that $x_2 \in \mathcal{L}(M_2)$ because every transition is a T_2 transition, the start state is s_2 and $r_n \in F_2$.

Subclaim 2: $\mathcal{L}(M_1) \circ \mathcal{L}(M_2) \subseteq \mathcal{L}(N)$

Proof: TBD.

Establishing the claim completes the proof of the proposition.

Iteration

Definition 4.10 *The* iteration *or* Kleene Closure *of a language L, written $L^\star$, is defined inductively:*

1. $\Lambda \in L^\star$
2. *If $x \in L$ and $y \in L^\star$ then $xy \in L^\star$*

Exercise 4.7 *Describe Kleene Closure of the following languages over $\{a, b\}$:*

1. $\{a\}$
2. $\{aa\}$
3. $\{a, b\}$
4. $\emptyset$
5. $\{\Lambda\}$

Proposition 4.11 *If L is accepted by an NFA then $L^\star$ is accepted by an NFA*

Proof: TBD.

4.4 NFAs and DFAs take 1

Sketch intuition for simulation of an NFA by a DFA. Key idea: Keep track of the set of states that the NFA may have reached on an input string.

5 Lecture 4: Proofs About Constructions

5.1 NFAs and DFAs define same class of languages

5.2 Regular Expressions: Starting from the closure properties

5.3 NFAs for all Regular Expressions

6 Lecture 5: Regular Expressions

6.1 Calculating Regular Expressions from NFAs

Basic idea: Define an algorithm that starts with an NFA and calculates a regular expression incrementally. To do this, we need a new model that is like an NFA with Λ transitions, only even more general. This model allows arbitrary regular expressions to label transitions.

Generalized NFA

1. Finite set of states, including at least two: the start and final state.
2. Finite alphabet
3. Transitions are a directed graph on states. The start state has indegree zero (it is a source). The final state has outdegree zero (it is a sink). All other possible edges exist and are labeled with a regular expression, possibly $\emptyset$.

Exercise 6.1 *Adapt the definition of acceptance to the generalized NFA model.*

6.2 Minimization of DFAs

See Hein 11.3.3.

Definition 6.1 *Two strings x and y are indistinguishable with respect to the language L, written $x \equiv_L y$ if $\forall z . xz \in L \leftrightarrow yz \in L$.*

Fact: $\equiv_L$ is an equivalence relation.

Fact: if L is recognized by DFA M and $T(s, x) = T(s, y)$ (where s is the start state of M), then $x \equiv_L y$.

What about the other way? Can we build a machine out of an equivalence relation?

Not always, but if there are only finitely many equivalence classes we can!

Quotient construction. Equivalence classes Hein 4.2.2.

7 Lecture 7: Regular and Non-regular Languages

This lecture explores the questions:

1. Are all languages regular?
2. How can we show a language is *not* regular?

Counting argument: How many languages? How many DFAs? Given that we need at least one DFA per language we know that non-regular languages exist.

Building on Minimization: Recall last time that we gave a general construction for a DFA for a language L with a finite number of equivalence classes with respect to $\equiv_L$. What if there are an infinite number? We did not prove it, but it will turn out that in such cases the language is not regular. (This actually follows from the argument sketched in lecture that the partition of $A^\star$ induced by the set of strings that reach each state must be a refinement of $A^\star / \equiv_L$.)

What about $\{a^n b^n | n \geq 0\}$? Is it regular? Can you find strings that distinguish a^i and a^j whenever $i \neq j$?

7.1 Pumping Lemma

The pumping lemma is a tool that captures a property of every regular language. It can often be used to prove that a language is not regular by showing that a language does not have the “pumping” property.

7.1.1 Warm up

If I have a language L that is accepted by a DFA M with k states, how many tests do I need to perform to determine if L is infinite?

Do I learn anything from tests on strings that are shorter than k symbols?

Do I learn anything from tests on strings that take k or more symbols?

If I start with strings of length k and test all strings, when can I stop testing and have confidence that all tests will fail?

7.1.2 Pumping idea

Any DFA with k states that accepts a string of length k or greater must repeat at least one state in the sequence of states visited. That is, if M accepts u and $|u| \geq k$, then u can be decomposed into x, y, and z such that

1. $u = xyz$
2. There is a state r such that $T(s, x) = r$, $T(r, y) = r$, and $T(r, z) = t$ for some $t \in F$.

In this case, M accepts xy^iz for all natural numbers i.

Exercise 7.1 *Find a string of length* 3 *or greater that is accepted by the mod 3 counter constructed earlier. Identify states and strings satisfying the property above.*

The pumping lemma generalizes this property.

The challenge of generalizing this property is that in general we don’t know what the number k is. We just know that there is some such number. This number is informally called the “pumping length”.

Theorem 7.1 *For any regular language L, there exists a natural number m, such that for any string u, $|u| \geq m$ implies that there exist strings x, y, and z such that*

1. $u = xyz$
2. $y \neq \Lambda$ *(or equivalently* $|y| > 0$*)*
3. $|xy| \leq m$
4. $xy^kz \in L$ *for all* $k \geq 0$

Notational aside: Hein uses s where I use u, but s is used for the start state of a generic machine M so I am adopting a different convention.

7.2 Proof of Pumping Lemma

Since L is a regular language, there is some machine $M = \langle S, A, T, s, F\rangle$ that recognizes it. Let the pumping length, m, be $|S|$. Consider a string $u = u_1 \ldots u_n$, $u \in L$ of length n, where $n \geq m$. Since $u \in L$ there exists a sequence of states $r_0, r_1, \ldots, r_n$ such that (1) $r_0 = s$, (2) $r_{i+1} = T(r_i, u_{i+1})$, and $r_n \in F$. Consider the first $m+1$ states in this sequence, $r_0, \ldots, r_m$. By the pigeon hole principle there must be a repeated state in this sequence. Let i and j be the indexes of distinct occurrences of a repeated state. Furthermore pick $i < j$. Take $x = u_1 \ldots u_i$, $y = u_{i+1} \ldots u_j$ and $z = u_{j+1} \ldots u_n$. We verify that the conditions of the pumping lemma are verified as follows:

1. $u = xyz$. Note that $xyz = u_1 \ldots u_i u_{i+1} \ldots u_j u_{j+1} \ldots u_n$ which is exactly the string u, as required.

2. $|y| > 0$. Since $i \neq j$, there is at least one symbol in the string $y = u_{i+1} \ldots u_j$, as required.

3. $|xy| \leq m$. From the choice of x and y we establish that $|xy| = j$. Since i and j are taken from the first $m+1$ states in the accepting sequence for u, we establish that $j \leq m$ as required.

4. $xy^k z \in L$ for any k. This follows from the following two facts: (1) for any k, $T(s, xy^k) = r_j$ and (2) $T(r_j, z) = r_n$. We will prove the first by induction. The second is a consequence of the choice of z.

   Proof of (1): **Basis:** $k = 0$. In this case $T(s, xy^0) = T(s, x) = r_i$, but by choice of i and j, $r_i = r_j$ as required. **Step:** show $T(s, xy^{k+1}) = r_j$ given $T(s, xy^k) = r_j$. So $T(s, xy^{k+1}) = T(s, xy^k y) = T(T(s, xy^k), y)$. Applying the induction hypothesis this becomes $T(r_j, y)$. Since $r_i = r_j$ we have $T(r_j, y) = T(r_i, y) = r_j$ as required. This completes the inductive proof of the condition and the proof of the lemma.

7.3 Applications of Pumping Lemma

Example 7.2 *Use the Pumping Lemma to show* $L = \{a^n b^n | n \geq 0\}$ *is not regular.*

Assume for the sake of contradiction that L is regular. Since L is regular there must be a pumping length m as given by the Pumping Lemma. Consider the string $u = a^m b^m$. Since $|u| \geq m$, there are substrings x, y, and z satisfying the conditions of the pumping lemma. By property 3 ($|xy| \leq m$) and choice of u we know that strings x and y are both in $a^\star$. By property 2 ($|y| > 0$) we know that y is non-trivial. Thus we can conclude there are integers j, k, and l such that:

$$\begin{array}{rcl} x & = & a^j \\ y & = & a^k \\ z & = & a^l b^m \end{array} \qquad \begin{array}{rrcl} \text{where} & m & = & j+k+l \\ & l & > & 0 \end{array}$$

Consider the string xy^2z. By property 4, $xy^2z \in L$. However $xy^2z = a^j a^{2k} a^l b^m = a^{m+k} b^m$. Since $k > 0$, $m \neq m + k$, hence $xy^2z \notin L$. This is a contradiction.

Having obtained a contradiction we reject the supposition that L is regular. This concludes the proof that L is not regular.

8 Lecture 8: Mealy Revisited

In this section we return to the Mealy machine for addition introduced in Section 3.4. When we introduced addition in Section 2.3 we described a circuit that computed the sum of two binary numbers one digit at a time, propagating the carry out of one position to the carry in of the next. We noted that this propagation was the critical path in the time necessary for the hardware to compute the sum. In Section 2.1 we argued that to compute a sum of n-bits it takes n "adder delays."

When we introduced this form of the adder, called the ripple-carry adder, we noted that there was an adder that introduced less delay, but did more redundant work. That adder algorithm is called the carry lookahead adder. It will use $\log_2 n$ delays to compute the sum of two n bit numbers. In this section we show how to derive the carry lookahead algorithm from the Mealy machine for addition.

A diagram of a carry lookahead adder can be found here: `http://www.aoki.ecei.tohoku.ac.jp/arith/mg/image/bcla.gif`.

Throughout this section the goal will be to build a hardware implementation of a fixed width adder for n bits. At times we will assume we are trying to calculate bit 17 of the sum of two 16 bit numbers (with the implicit assumption that there are leading zero bits on the 16 bit numbers). As before we assume we see the least significant bit first and the most significant bit last. (This is in reverse of standard conventions for doing arithmetic by hand.)

8.1 Speeding up Addition

Recall the definition of a Mealy machine. It includes two functions, the transition function T, that maps states and input symbols to new states, and the output function O that maps states and input symbols to output symbols.

If we want to know what the 17th output symbol is, all we need to calculate is the 16th state, which we could do using just T and the first 16 inputs, and then apply the output function.

Writing symbolically and generalizing we get:

$$\begin{aligned}
T &: S \times A \to S \\
O &: S \times A \to B \\
r_0 &= s \\
r_{i+1} &= T(r_i, a_{i+1}) \\
b_{i+1} &= O(r_i, a_{i+1})
\end{aligned}$$

We can extend T to act on strings with the following definition:

$$\begin{aligned}\hat{T}(r, \Lambda) &= r \\ \hat{T}(r, ax) &= \hat{T}(T(r, a), x)\end{aligned}$$

This lets us rewrite the computation of the output symbols as:

$$b_{i+1} = O(\hat{T}(s, a_1 \ldots a_i), a_{i+1})$$

Expressing the problem in this form suggests that a good strategy to be able to quickly compute, say, b_{17}, will come from finding a way to quickly compute $\hat{T}(s, a_1 \ldots a_{16})$.

8.1.1 Knowns and unknowns

Recall that for addition the input symbols are pairs of binary digits, that is $A = \{0, 1\} \times \{0, 1\}$. To enhance readability we will write the pair of bits (a, b) as $\left[\begin{array}{c} a \\ b \end{array}\right]$. When we have strings of bit pairs $(a_1, b_1) \ldots (a_n, b_n)$ we will write that as $\left[\begin{array}{c} a_1 \ldots a_n \\ b_1 \ldots b_n \end{array}\right]$.

When we think of the circuit computing addition, we assume that it "sees" the input in parallel. That is, all input symbols are available with zero delay. However, in the carry chain adder each full adder must wait for the carry-in to come from the lower order bits before it can generate the sum and carry-out.

To avoid the carry-propagation delay we change the representation of the transition function. We use an isomorphic type that separates the information available immediately (the input symbols) from the information available after a delay (the previous state). The type of the new transition function is given as:

$$T' : A \to (S \to S)$$

We expect T' to satisfy:

$$T'(a) = f \quad \text{where } f(s) = T(s, a) \tag{15}$$

For the adder, what does $S \to S$ look like? Recall $S = \{0, 1\}$, so $S \to S$ is all functions from $\{0, 1\}$ to itself. There are four such functions:

$$\begin{aligned}Ix &= x \\ X0 &= 1 \\ X1 &= 0 \\ \mathbf{0}x &= 0 \\ \mathbf{1}x &= 1\end{aligned}$$

Now we can define T':

$$T'(\left[\begin{array}{c} 0 \\ 0 \end{array}\right]) = \mathbf{0}$$

$$\begin{aligned}
T'(\begin{bmatrix} 0 \\ 1 \end{bmatrix}) &= I \\
T'(\begin{bmatrix} 1 \\ 0 \end{bmatrix}) &= I \\
T'(\begin{bmatrix} 1 \\ 1 \end{bmatrix}) &= \mathbf{1}
\end{aligned}$$

Exercise 8.1 *Verify that T' satisfies Equation 15.*

Exercise 8.2 *Compute the fourth digit of the sum $2 + 7$ using this method.*

8.1.2 Composition

Another key to speeding up the calculation is to find ways to combine partial results. Having changed representations to T', we can now ask how a group of bits will change the state. We see this by extending T' to strings of bits:

$$\begin{aligned}
\hat{T}'(\Lambda) &= I \\
\hat{T}'(ax) &= T'(a) \,;\, \hat{T}'(x)
\end{aligned}$$

The symbol ; represents function composition in "diagrammatic" order, that is $(f \;;\; g)(x) = g(f(x))$. Why is this called diagrammatic order? Because if we draw a directed graph where the nodes are sets and the edges are functions this is the order in which the functions appear in the diagram. The other notation for composition, generally written $f \circ g$, is called non-diagrammatic order. It is defined as $(f \circ g)(x) = f(g(x))$. So $f \;;\; g = g \circ f$.

Why use diagrammatic order here? Because it will keep the inputs in logical order. Consider $\hat{T}'(a_1 a_2 \ldots a_n)$. Using diagrammatic order we can write this as $T'(a_1) \;;\; T'(a_2) \;;\; \ldots \;;\; T'(a_n)$. In non-diagrammatic order it would be reversed. That is, $T'(a_n) \circ \ldots \circ T'(a_2) \circ T'(a_1)$.

The next step in developing the algorithm is to realize that we can compute the function that results from composing functions without knowing the input. That is, we can know $f \;;\; I = f$. Similarly we know $f \;;\; \mathbf{0} = \mathbf{0}$. The following table computes all compositions in $\{0, 1\} \to \{0, 1\}$:

;	I	X	$\mathbf{0}$	$\mathbf{1}$
I	I	X	$\mathbf{0}$	$\mathbf{1}$
X	X	I	$\mathbf{0}$	$\mathbf{1}$
$\mathbf{0}$	$\mathbf{0}$	$\mathbf{1}$	$\mathbf{0}$	$\mathbf{1}$
$\mathbf{1}$	$\mathbf{1}$	$\mathbf{0}$	$\mathbf{0}$	$\mathbf{1}$

Given that we can compute the composition of functions without knowing their inputs, this gives us a way to combine information about the inputs before we know what the carry in is. Furthermore, since function composition is associative, that is:

$$f \;;\; (g \;;\; h) = (f \;;\; g) \;;\; h$$

we can be strategic about how we combine functions. An immediate consequence of this is that we can decompose a string in any manner we wish:

Exercise 8.3 *Prove by induction that* $\hat{T}'(xy) = \hat{T}'(x) \mathbin{;} \hat{T}'(y)$.

Example 8.4 *Consider adding eight bit binary numbers* 01001101 *and* 01100001. *By our convention this is represented:*

$$\left[\begin{array}{c} 10110010 \\ 10000110 \end{array}\right]$$

Compute the 8th bit of the answer as follows:

$$\begin{aligned}
s_8 &= O(c_7, \left[\begin{array}{c} 0 \\ 0 \end{array}\right]) \\
c_7 &= \hat{T}(0, \left[\begin{array}{c} 1011001 \\ 1000011 \end{array}\right]) \\
&= (\hat{T}'(\left[\begin{array}{c} 1011001 \\ 1000011 \end{array}\right]))(0) \\
&= (\hat{T}'(\left[\begin{array}{c} 1011 \\ 1000 \end{array}\right]) \mathbin{;} \hat{T}'(\left[\begin{array}{c} 001 \\ 011 \end{array}\right]))(0) \\
&= ((\hat{T}'(\left[\begin{array}{c} 10 \\ 10 \end{array}\right]) \mathbin{;} \hat{T}'(\left[\begin{array}{c} 11 \\ 00 \end{array}\right])) \mathbin{;} (\hat{T}'(\left[\begin{array}{c} 00 \\ 01 \end{array}\right]) \mathbin{;} \hat{T}'(\left[\begin{array}{c} 1 \\ 1 \end{array}\right])))(0) \\
&= (((\hat{T}'(\left[\begin{array}{c} 1 \\ 1 \end{array}\right]) \mathbin{;} \hat{T}'(\left[\begin{array}{c} 0 \\ 0 \end{array}\right])) \mathbin{;} (\hat{T}'(\left[\begin{array}{c} 1 \\ 0 \end{array}\right]) \mathbin{;} \hat{T}'(\left[\begin{array}{c} 1 \\ 0 \end{array}\right]))) \mathbin{;} ((\hat{T}'(\left[\begin{array}{c} 0 \\ 0 \end{array}\right]) \mathbin{;} \hat{T}'(\left[\begin{array}{c} 0 \\ 1 \end{array}\right])) \mathbin{;} (\hat{T}'(\left[\begin{array}{c} 1 \\ 1 \end{array}\right]))))(0) \\
&= (((\mathbf{1} \mathbin{;} \mathbf{0}) \mathbin{;} (I \mathbin{;} I)) \mathbin{;} ((\mathbf{0} \mathbin{;} I) \mathbin{;} (\mathbf{1})))(0) \\
&= ((\mathbf{0} \mathbin{;} I) \mathbin{;} (\mathbf{0} \mathbin{;} \mathbf{1}))(0) \\
&= (\mathbf{0} \mathbin{;} \mathbf{1})(0) \\
&= \mathbf{1}(0) \\
&= 1 \\
s_8 &= O(1, \left[\begin{array}{c} 0 \\ 0 \end{array}\right]) \\
&= 1
\end{aligned}$$

In this calculation we see that with the right problem decomposition, the state change function for a group of k bits can be calculated in $\log_2 k$ compositions.

If we imagine hardware based on this principle, given two n bit number with all bits available at once we can calculate bit k of the output as follows:

- At the first stage convert to the function symbol corresponding to the input ($T'a_i$).
- At the next $\log_2 k$ stages combine the function symbols according to the function composition table above to obtain $\hat{T}'(a_1 \ldots a_{k-1})$.
- Obtain r_{k-1} by applying the resulting function to 0
- Compute $O(r_{k-1}, a_k)$

This notional circuit has depth $\log_2 k + 3$. All data dependencies are from one stage to a later stage—the circuit never has to wait for data from within a stage. Hence the total cumulative delay is $\log_2 k + 3$ computation delays.

8.1.3 Representing I, X, $\mathbf{0}$, and $\mathbf{1}$

How should the function symbols I, X, $\mathbf{0}$ and $\mathbf{1}$ be represented? For the addition algorithm first note that X is never used. In the traditional presentation of this algorithm the three functions that occur are represented by two bits, one called carry generate (g), and the other called carry propagate (p). The equivalence is given by:

f	g	p
I	0	1
$\mathbf{0}$	0	0
$\mathbf{1}$	1	0

Exercise 8.5 *Given f_1 represented by (g_1, p_1) and f_2 represented by (g_2, p_2) give logical formulae to compute the representation of $f_1 \,;\, f_2$, (g, p).*

Exercise 8.6 *For each stage in the notional hardware algorithm above, identify what Boolean logic computations are required to compute that stage.*

8.1.4 Reusing results

But how much work is done?

So far each bit in the sum is computed independently. In this version of the algorithm nothing learned about bit 5 is used when computing bit 6. This is very wasteful, and is based on unrealistic assumptions about being able to duplicate values an arbitrary number of times without cost.

The final step towards realistic carry-lookahead addition algorithms is to organize the reuse of partial results. There are several well known hardware algorithms that achieve significant reuse of partial results. These algorithms are collectively called parallel prefix computation algorithms.

The parallel prefix problem is a generalization of the addition problem. It is stated as follows: Given an associative operator $\odot$ and a list of inputs $a_1 \ldots a_n$ compute

$$\begin{aligned} b_1 &= a_1 \\ b_2 &= a_1 \odot a_2 \\ &\vdots \\ b_n &= a_1 \odot a_2 \odot \ldots \odot a_n \end{aligned}$$

This class of problems was identified by Ladner and Fischer in their 1980 paper in the Journal of the ACM (Vol 27, No. 4, pp. 831–838). `http://dx.doi.org/10.1145/322217.322232`

Some more recent work on parallel prefix is linked to from Mary Sheeran's home page: `http://www.cse.chalmers.se/~ms/`. This includes here recent article in the Journal of Functional Programming, *Functional and dynamic programming in the design of parallel prefix networks*, JFP 21(1). `http://journals.cambridge.org/repo_A82t9U30`.

8.2 Closing comments

The construction works for any Mealy machine. Since the state space is finite, $S \to S$ will have a finite representation.

This means, for example, the Instruction Length Decoder (ILD) developed in the first homework can be implemented with this technique. This allows hardware to pull in a full cache line and identify all instructions and operands with a circuit of depth proportional to the log of the size of the cache line.

9 Context Free Grammars

9.1 General Grammars

Note general definition of a grammar in Hein in Section 3.3.

Definition 9.1 *A* grammar G *is specified by a tuple* $\langle N, T, S, P\rangle$ *where*

1. N *is a finite set of* nonterminal *symbols (alphabet)*
2. T *is a finite set of* terminal *symbols*
3. $S \in T$ *is a designated start symbol*
4. P *is a finite set of productions of the form* $\alpha \to \beta$*, where* α *and* β *are strings over* $N \cup T$.

Define derivation for general grammar (Hein 3.3.3).

Definition 9.2 *If* x *and* y *are strings over* $N \cup T$ *and* $\alpha \to \beta \in P$ *then* $x\alpha y \Rightarrow x\beta y$*. The relation* $\Rightarrow$ *is called the* derivation relation.

Following standard conventions: $\Rightarrow$ represents derives in exactly one step, $\Rightarrow^{+}$ represents derives in one or more steps, and $\Rightarrow^{\star}$ represents derives in zero or more steps.

Leftmost derivation mentioned in context of general grammar (Hein p. 184). In the notes I will use the symbol $\Rightarrow_L$ for the leftmost derivation relation.

Language of a grammar (Hein p. 184).

Definition 9.3 *If* $G = \langle N, T, S, P\rangle$ *is a grammar, then the* language of the grammar *is the set:*

$$\mathcal{L}(G) = \{s \in T^{\star} | S \Rightarrow^{+} s\}$$

9.2 Context Free Grammars

Definition 9.4 *A* Context Free Grammar *is a restriction on general grammar in which productions are of the form $V \to \beta$ for $V \in N$.*

Definition 9.5 *A language M over $T^\star$ is* context free *if $M = \mathcal{L}(G)$ for some context free grammar G.*

Exercise 9.1 *State the corresponding restriction for a Regular Grammar.*

Exercise 9.2 *Give a more specific definition of* leftmost derivation *($\Rightarrow_L$)for Context Free Grammars.*

Exercise 9.3 *Prove that for any CFG grammar $S \Rightarrow^\star x$ if and only if $S \Rightarrow_L^\star x$.*

Problem 9.4 *Can you construct a general grammar that does not have property 9.3?*

Definition 9.6 *A CFG G derives a string x* ambiguously *if there are two distinct leftmost derivations. A grammar G is* ambiguous *if there is a string $x \in \mathcal{L}(G)$ that is derived ambiguously.*

Note: ambiguity is a property of the grammar, not the language. There are some context free languages for which all grammars are ambiguous. Such languages are called *inherently ambiguous*. They will not be discussed in this class.

9.3 Closure Properties

Note that the Context Free Languages are closed under union, concatenation, and Kleene star.

10 Push Down Automata

The basic idea of a push down automaton is to retain the finite control of the finite automata, but add a single stack to function as a store. Notationally there are several ways to do this.

It will turn out that the expressive power of deterministic PDAs and nondeterministic PDAs will be different. We will focus primarily on the nondeterministic PDAs; we will eventually show that they describe exactly the context free languages.

Hein takes an approach that names the stack operations and introduces the idea of a "PDA instruction" to describe how the stack and states change together. Although the notation looks quite different, Hopcroft, Motwani, and Ullman use an isomorphic formulation of PDAs. Sipser is a little more minimalist. The power point materials accompanying the lecture follow Hopcoft *et al.*. I will attempt to follow Hein.

Motivating example: build a machine to recognize $\{a^n b^n | n \geq 0\}$. The key idea will be to count the number of a's by pushing a marker on the stack for each a. We will then want to verify that the number of b's corresponds by popping the markers as the b's are read. To get exactly the number of a's we will want to have a distinct marker at the bottom of the stack.

To build this we need to:

1. Mark the bottom of the stack: let's mark the bottom with the stack symbol \$.
2. Count the number of a's by pushing a symbol, I choose †.
3. We will transition to a different state to count the b's.
4. After consuming a b for every †, transition to an accept state without consuming any input.

To do this, we need a way to talk about state transitions that are coupled with stack transformations.

The basic stack operations are pop, to remove a symbol, nop, to do nothing, and push b to add the symbol b to the top of the stack.

The basic move a PDA is described by what Hein calls an "instruction". An instruction is a tuple that includes: the source state, p, an input symbol (or Λ), a, a stack symbol b, a stack operation, O, and a target state q. This instruction is written (p, a, b, O, q).

Example 10.1 *Draw a cartoon for the machine.*

Definition 10.1 *A* Push Down Automaton (PDA) *is specified by a 7 tuple* $\langle S, A, B, I, E, s, F\rangle$ *where:*

1. *S is the set of states (finite, non-empty)*
2. *A is the input alphabet (finite, non-empty)*
3. *B is the stack alphabet (finite, non-empty)*
4. *I is the "instruction set", a subset of $S \times (A \cup \{\Lambda\}) \times B \times \mathit{StkCalls} \times S$, where $\mathit{StkCalls} = \{nop, pop\} \cup \{push\ b | b \in B\}$.*
5. *$E \in B$ is the initial stack symbol*
6. *$s \in S$ is the start state*
7. *$F \subseteq S$ is the set of final states*

Definition 10.2 *Instantaneous Descriptions*

Definition 10.3 *Acceptance by Final State*

11 Primitive and Partial Recursive Functions

The primitive and partial recursive functions approach computability from a different point of view than the Turing machine. Much like the regular expressions, we will develop an inductive definition of functions. In this case, it will be an inductive definition of functions from natural numbers to natural numbers.

We will first study the primitive recursive functions. All functions in this class will be obviously *computable*, that is, you could write a program in your favorite programming language that reads the description of a function and computes its value on any input. They will also be *total*, that is, they will be defined on all inputs.

The second class we will study are the partial recursive functions. Like the primitive recursive functions, they will all be computable. However they will not all be total. It will be possible to write functions that cannot be computed on some inputs.

When defining both classes of functions we will pay careful attention to how many arguments each function takes. We will call a function that takes n arguments as an n-place function or a function of *arity* n.

11.1 Terminating Recursion

In homework and exams we have seen the following definition of addition:

$$\begin{aligned} Z + y &= y \\ S(x) + y &= S(x + y) \end{aligned}$$

This is a good recursive definition. It defines $+$ as a total function. For any pair of natural numbers x and y, the computation $x + y$ will terminate. Specifically, it will require x applications of the second rule and 1 application of the first rule.

An example of a bad definition of a function is:

$$f(x) = f(x)$$

This equation does not define the function f. If we treat this as a rewrite rule, where the left hand side is rewritten to the right hand side, we can apply this an infinite number of times without making any progress toward an answer. While this equation is a bad definition, it is a reasonable property. All functions satisfy this property.

The rule for primitive recursive functions captures a particular style of good definitions on the natural numbers. This style says that any function f defined by two equations:

$$\begin{aligned} f(Z, x_1, \ldots, x_k) &= e_Z \\ f(S(n), x_1, \ldots, x_k) &= e_S \end{aligned}$$

is a good definition provided:

1. The expression e_Z only mentions variables $x_1, \ldots, x_k$ and makes no reference to f.
2. The expression e_S only mentions variables $n, x_1, \ldots, x_k$, and there is at most one recursive call to f which is exactly of the form $f(n, x_1, \ldots, x_k)$.

Note this is consistent with the style of definition Hein presents in Section 3.2.

The primitive recursion rule enforces this style of definition by requiring that the expressions e_Z and e_S be characterized by two functions, h and g. The user no longer has the freedom to write down the recursive calls. Instead they just provide these functions, and they are automatically plugged into the following definition schema:

$$\begin{array}{rcl} f(Z, x_1, \ldots, x_k) & = & h(x_1, \ldots, x_k) \\ f(S(n), x_1, \ldots, x_k) & = & g(n, f(n, x_1, \ldots, x_k), x_1, \ldots, x_k) \end{array}$$

Returning to the definition of $+$, the first equation needs to satisfy $h_+(x_1) = x_1$. So h_+ must be the identity function. The second equation needs to satisfy

$$g_+(n, n + x_1, x_1) = S(n + x_1)$$

This yields the function:

$$g_+(a, b, c) = S(b)$$

In this manner we will say that $+$ is defined by primitive recursion using these two functions h_+ and g_+. Note that $+$ is a 2-place function, h_+ is a 1-place function, and g_+ is a 3-place function. This pattern will generalize. To define a k-place function f we will need a $k-1$ place function h and a $k+1$-place function g.

11.2 Primitive Recursive Functions

There are five rules for defining the primitive recursive functions. The first three define a set of basic primitive functions. The last two rules explain how to build functions by combining other functions.

I. Zero For every arity k there is a constant function Z satisfying $Z(x_1, \ldots, x_k) = 0$.

II. Successor There is a 1-place function, S, that computes the successor. That is, S satisfies $S(x) = x + 1$.

III. Projection For every arity k there are k projection functions, $P_1, \ldots, P_k$ satisfying: $P_i(x_1, \ldots, x_i, \ldots x_k) = x_i$.

IV. Composition (or Substitution) An arity k function, f, can be combined with k functions of arity l, $g_1, \ldots, g_k$ to produce a new arity l function, $C\ f\ [g_1, \ldots, g_k]$, satisfying:

$$C\ f\ [g_1, \ldots, g_k](x_1, \ldots, x_l) = f(g_1(x_1, \ldots, x_l), \ldots, g_k(x_1, \ldots, x_l))$$

V. Primitive Recursion Given a k place function h and a $k{+}2$ place function g, the $k+1$ place function f defined by primitive recursion, written PR h g, satisfies:

$$\begin{array}{rcl} f(0, x_1, \ldots, x_k) & = & h(x_1, \ldots, x_k) \\ f(n+1, x_1, \ldots, x_k) & = & g(n, f(n, x_1, \ldots, x_k), x_1, \ldots, x_k) \end{array}$$

11.3 Discussion and Exercises

Note that the official definitions of the functions do not have variable names. They are almost like an assembly language for functions.

Exercise 11.1 *Compute the following:*

1. $P_2(1, 2, 3)$
2. $Z(21, 17, 42)$
3. $S(13)$
4. $(C\ S\ [Z])(1, 2, 3)$
5. $(\mathrm{PR}\ Z\ Z)(3)$
6. $(\mathrm{PR}\ Z\ (C\ S\ (P_1)))(3)$

Exercise 11.2 *Use rules I, II, III, and IV to do the following:*

1. *Use composition, successor, and zero to define the constant function 2.*
2. *What is the identity function?*
3. *Assuming f is a 2-place function build a new 1-place function g that applies f to two copies of its argument. That is $g(x) = f(x, x)$.*
4. *Assuming f is a 2-place function, build a new 5-place function g that applies f to its second and fourth arguments. That is $g(x_1, x_2, x_3, x_4, x_5) = f(x_2, x_4)$.*
5. *Assuming f is a 2-place function build a new 2-place function g that applies f with its arguments reversed. That is $g(x, y) = f(y, x)$.*
6. *Construct the 3-place function g_+ used in the definition of $+$ in the previous section.*

Exercise 11.3 *Complete the definition of + by applying rule V to the appropriate results from the previous exercise.*

Exercise 11.4 *Multiplication can be defined as follows:*

$$\begin{aligned} Z \times y &= Z \\ S(n) \times y &= y + (n \times y) \end{aligned}$$

Construct the Primitive Recursive function that implements this definition of multiplication. You may reuse your definition of addition.

11.4 An Implementation of Primitive Recursion

The primitive recursive functions can be defined in the programming language Haskell as a recursive datatype, `PR`. A simple evaluator is given by the Haskell function `eval`. What follows is a complete Haskell program:

```
data PR =  Z
         | S
         | P Int
         | C PR [PR]
         | PR PR PR

eval :: PR -> [Integer] -> Integer
eval Z _ = 0
eval S [x] = x+1
eval (P n) xs = nth n xs
eval (C f gs) xs = eval f (map (\g -> eval g xs) gs)
eval (PR g h) (0:xs) = eval g xs
eval (PR g h) (x:xs) = eval h ((x-1) : eval (PR g h) ((x-1):xs) : xs)

nth _ [] = error "nth nil"
nth 0 _ = error "nth index"
nth 1 (x:_) = x
nth (n) (_:xs) = nth (n-1) xs
```

This is a simplified version of a file that is posted on the web site as Naturalpr.hs.

The type of primitive recursive functions here is called `PR`. The constructors are `Z`, `S`, `P`, `C`, and `PR`. The constructor `C` takes two arguments, one of type `PR` and the other is a list (indicated by the `[ ]`) or `PR`.

The function `eval` is defined inductively on the type `PR`. It takes a value of type `PR` and a list of Integer's representing the tuple of arguments, and returns an Integer.

The addition function can be defined by the Haskell value:

```
plus = PR (P 1) (C S [P 2])
```

In this case, we can evaluate the sum of 2 and 3 as follows:

```
*Main> eval plus [2,3]
5
```

11.5 Expressive Power

What functions can you write by primitive recursion?

In practice you can write most of the Integer functions that people use in routine mathematics. However, you do not get all integer functions. You do not even get all total computable integer functions.

The class of functions are sufficiently expressive that you can encode Turing machines as numbers, using what is called a *Gödel numbering.* This is essentially a representation of a programming language data type for the formal definition of a Turing machine that allows you to recover the parts of the definition. In particular, you can recover the set of states, the start state, and the set of instructions that define the Turing machine. You can also encode instantaneous descriptions and computation histories as numbers, and you can define a function that can test if they are correct, halting computations. In particular, the function T such that $T(e, x, y)$ is true if y is a halting computation of machine e on input x is a primitive recursive function.

11.6 Partial Recursive functions

To increase the expressive power to include all computable functions it is necessary to overshoot the set of total functions, and enter the domain of partial functions. That is, we will allow functions that are undefined on some values.

In this paradigm the basic way to introduce partiality is to add a search capability that is defined by a potentially non-terminating recursion. Consider the following program:

```
try_from :: (Integer -> Bool) -> Integer -> Integer
try_from test n =
  if test n
     then n
     else try_from test (n + 1)

try :: (Integer -> Bool) -> Integer
try test = try_from test 0
```

This program takes a predicate called `test` and applies it to every natural number starting from 0 and counting up until it finds one that has the expected property.

For example, `try even` returns 0 and `try odd` returns 1.

The unbounded search rule, defined as a sixth rule for defining functions, is given in the next section. It essentially encodes the search strategy of the `try` function in the context of these integer functions.

See Hein 13.2.3.

VI. Unbounded Search Given a $k+1$-place function g, the k-place function μg satisfies exactly $(\mu g)(x_1, \ldots x_k) = n$ if and only if $g(n, x_1, \ldots, x_n) = 0$ and for all $y < n$, $g(y, x_1, \ldots, x_n)$ is defined and $g(y, x_1, \ldots, x_n) \neq 0$.

Problem 11.5 *Modify* `eval` *to accommodate this additional definition scheme. To conform with conventions used later in these notes, I recommend naming the type of partial recursive functions* `MuR` *and naming the new data constructor* `Mu`*. The name* `MuR` *can be pronounced as the "mu-recursive" (or μ-recursive) functions. It is unfortunate that primitive and partial have the same abbreviation.*

With unbounded search it is now possible to define all computable functions. In particular, if you use μ to search through possible computation histories in the T function discussed above you can search the space of all possible computations of a Turing machine e on input x. From the resulting computation history y, which will be returned by μ if such a history exists, you can decode the value computed by the function. In this way it is possible to express any function computed by a Turing machine as a partial recursive function.

For more information about these results you can read about the Kleene T predicate and the Kleene Normal Form theorem. These topics are typically covered in texts on recursion theory. A web search yielded several reasonable articles.

12 Exploring Computability

Informally, the basic building blocks of computability are

1. A syntactic notion of program that can be written in a numbered list.
2. The ability to write down the trace of a computation that can be verified by a series of simple (terminating) steps.
3. Having a large enough set of programs, in particular there needs to be a universal program (also known as an interpreter) that can read a program and its input and generate its output.

For Turing machines we had Turing machine descriptions, computation histories, and we asserted the existance of universal Turing machines.

For the partial recursive functions we will give a few more details. We have already shown an interpreter written in Haskell for the primitive recursive functions. The extension of this interpreter to partial recursive functions is given as an exercise (Problem 11.5).

12.1 Pairing Functions

Pairing functions take a pair of natural numbers and encode them as a single natural number. The pairs can be recovered from the original number.

Cantor developed a pairing function that is one-to-one and onto. There is a reasonable article on these pairing functions on Wikipedia (`http://en.wikipedia.org/wiki/Pairing_function`).

The code fragment below implements Cantor's pairing function:

```
pair :: Integer -> Integer -> Integer
pair k1 k2 = ((k1 + k2) * (k1 + k2 +1) `div` 2) + k2
```

The pairs can be deconstructed by this code fragment:

```
unpair :: Integer -> (Integer,Integer)
unpair z = let w = (squareRoot (8*z + 1) - 1) `div` 2
               t = (w * w + w) `div` 2
               y = z - t
               x = w - y
           in (x, y)
```

Some sample values:

	0	1	2	3	4
0	0	2	5	9	14
1	1	4	8	13	19
2	3	7	12	18	25
3	6	11	17	24	32
4	10	16	23	31	40

Using pairing functions, it is possible to encode other datatypes. For example, the following fragment of Haskell encodes and decodes lists of Integers:

```
eList :: [Integer] -> Integer
eList [] = pair 0 0
eList (x:xs) = pair 1 (pair x (eList xs))

dList :: Integer -> [Integer]
dList l = let (t,c) = unpair l
              (h, tl) = unpair c
          in case t of
             0 -> []
             1 -> h:(dList tl)
             _ -> []  -- make it total; nonsense is nil
```

In this encoding each list element is represented by a pair. If the first element is a `0` then it represents the empty list (nil). If the first element is a `1` then it represents a non-empty list (cons). In that case the second element can be decoded into the head of the list and the tail of the list (which is itself a pair).

For example:

$[]$	$(0,0)$	0
$[2]$	$(1,(2,(0,0)))$	13
$[2,3]$	$(1,(2,(1,(3,(0,0)))))$	$246,751$
$[2,3,4]$	$(1,(2,(1,(3,(1,(4,(0,0)))))))$	$94,523,914,127,548,123,793,040,376$

In a similar manner the encoding can be extended to any datatype. For example, the primitive recursive functions can be encoded and decoded with the following fragment:

```
ePR :: PR -> Integer
ePR Z = pair 0 0
ePR S = pair 1 0
ePR (P i) = pair 2 (toInteger i)
ePR (C f gs) = pair 3 (pair (ePR f) (eList (map ePR gs)))
ePR (PR g h) = pair 4 (pair (ePR g) (ePR h))

dPR x = let (t,b) = unpair x
            (b1,b2) = unpair b -- note:  Lazy
        in case t of
           0 -> Z
           1 -> S
           2 -> P (fromInteger b)
           3 -> C (dPR b1) (map dPR (dList b2))
           4 -> PR (dPR b1) (dPR b2)
           _ -> Z
```

Using this encoding, called a Gödel numbering, we can see that the Gödel number of the plus function (PR P_1 (C S [P_2])) is

```
*Goedel> ePR $ PR (P 1) (C S [P 2])
4511739842654672905730185440573223378237806974280320
*Goedel> dPR $ 4511739842654672905730185440573223378237806974280320
PR (P 1) (C S [P 2])
```

This example uses the $ notation from Haskell, which stands for an open parenthesis that extends as far to the right as possible, Thus `f $ x + 1` means the same thing as `f (x + 1)`. Contrast this with `f x + 1`, which means `(f (x)) + 1`. While this idiom at first seems unusual, in many cases it will remove quite a bit of syntactic clutter from programs.

12.2 Why we need Partial Functions

How hard is it to define a function that is not in the set of Primitive Recursive Functions?

In this section we will write a Haskell function that is total, computable, but not primitive recursive. We will build it by using a technique called diagonalization. The concept of diagonalization was introduced by Cantor to show that the reals and the natural numbers have different cardinalities, that is, there is not a one-to-one correspondence between them. Hein presents diagonalization in Section 2.4.3 on page 125.

In Cantor's construction, a countable list of decimal expansions of real numbers is assumed. From that list a procedure is defined to build a number that is different from all in the list. This number is built by taking the diagonal of the list—building a number that differs from the first in the first position, from the second in the second position, and so on.

How do we diagonalize over the primitive recursive functions? We start by building a table that lists all such functions and their values:

x	`dPR` x	0	1	2	3	4	5	6	7	8	9	10
0	Z	0	0	0	0	0	0	0	0	0	0	0
1	S	1	2	3	4	5	6	7	8	9	10	11
2	Z	0	0	0	0	0	0	0	0	0	0	0
3	P_1	0	1	2	3	4	5	6	7	8	9	10
4	S	1	2	3	4	5	6	7	8	9	10	11
5	Z	0	0	0	0	0	0	0	0	0	0	0
6	$C\ Z\ []$	0	0	0	0	0	0	0	0	0	0	0
7	P_1	0	1	2	3	4	5	6	7	8	9	10
8	S	1	2	3	4	5	6	7	8	9	10	11
9	Z	0	0	0	0	0	0	0	0	0	0	0
10	PR $Z\ Z$	0	0	0	0	0	0	0	0	0	0	0

We want to compute a function that differs from all functions in the list, so we construct a function that adds one to the result found on the diagonal above. That is, where the diagonal above has as its first 10 values:

$$0, 2, 0, 3, 5, 0, 0, 7, 9, 0, 0$$

We wish to define a function that instead returns:

$$1, 3, 1, 4, 6, 1, 1, 8, 10, 1, 1$$

Such a function can be expressed in Haskell as

```
diagonal x = (eval p (ncopies (arity p) x))+1
   where p = dPR x
```

This function takes one natural number argument, x. It interprets it as a primitive recursive function, p. It then evaluates that function (applied to its own encoding x, repeated as many times as necessary to match the arity of the function). It adds one to the result.

The Haskell function `diagonal` is a total computable function. It is defined on all natural numbers.

The function `diagonal` is different from any primitive recursive function. Why? Suppose it was primitive recursive. Then there would be some program, p in the language of primitive recursive functions, and that function would have a number, n, such that $n =$ `ePR` p. But $p(n) = z$ for some number z, and `diagonal` $n = z + 1$. Thus `diagonal` $\neq p$. Having obtained a contradiction we conclude that `diagonal` is not a primitive recursive function.

What facts about primitive recursive functions did we use in this argument? Not very many. We relied on the property that primitive recursive functions are total. We relied on the existence of the `eval` function. And we relied on the existence of a function from numbers to programs. This situation is called an *effective enumeration*.

Definition 12.1 *An* effective enumeration *of a set of functions is a mapping of the natural numbers onto the set of functions, $\phi_0, \phi_1, \ldots$, together with a computable function* `eval` *such that* `eval` $i\ x = \phi_i(x)$.

Theorem 12.2 *Any effective enumeration of a set of total functions is incomplete. That is, there are total computable functions that are not included in the enumeration.*

The proof is simply a generalization of the argument given above that there is a total computable function that is not primitive recursive.

Corollary 12.3 *There is no effective enumeration of the set of total computable functions.*

Thus any theory of the computable functions that includes all total computable functions must also include partial functions.

Problem 12.1 *How does the possibility that* `eval` *might not terminate change the argument that effective enumerations are incomplete? (This is probably too unstructured to assign as a problem, but is an important question to consider.)*

12.3 Pairing is Primitive Recursive

To implement pairing with the Primitive Recursive functions, the basic arithmetic and logical operations you implemented in homework need to be extended. Some of the operations are given here:

```
-- Elementary arithmetic
mkconst 0 = Z
mkconst n = C S [mkconst (n-1)]

one    = mkconst 1
plus   = PR (P 1) (C S [P 2])
times  = PR Z (C plus [P 2, P 3])
pred   = PR Z (P 1)
monus' = PR (P 1) (C pred [P 2])
monus  = C monus' [P 2, P 1]


-- Elementary logic
false = Z
true  = one
and   = PR false (P 3)
or    = PR (P 1) true
not   = PR true false
ifte  = PR (P 2) (P 3)
```

```
-- Predicates
iszero  = PR true false
nonzero = PR false true
equal   = C and [C iszero [monus],
                 C iszero [monus']]
lteq    = C iszero [monus]
lt      = C nonzero [monus']
```

In this style of programming, one tool that is helpful is to search for values that satisfy a predicate. In this development I made extensive use of a simple operation called bounded minimization, which returns the least value less than a bound that makes a predicate hold.

Mathematically, if $P(x_1, x_2, \ldots, x_k)$ is a predicate (returns 0 if false, 1 if true), then

$$\mu x < n.P(x, x_2, \ldots, x_k)$$

returns the least x such that $P(x, x_2, \ldots, x_k) \neq 0$, provided there is such an x less than n. If no such element exists it returns a value $x \geq n$. Note that this is very similar to the unbounded search capability in rule VI, in Section 11.6. Adding the bound significantly restricts its expressive power, and makes it possible to define search within the Primitive Recursive functions.

Bounded minimization gets implemented as a k-place primitive recursive function that takes the bound as its first argument. That function is generated by the Haskell function `bmin`, which takes the predicate P and its arity as arguments. For example, the following interaction reports the least integer less than 2 that satisfies the `true` predicate and the least integer less than 2 that satisfies the `false` predicate:

```
*PRCantor> eval (bmin true 1) [2]
0
*PRCantor> eval (bmin false 1) [2]
2
```

The Haskell function that generates instances of `bmin` is given below. First, the "model" code of a Haskell function that behaves like the primitive recursive function is given. Then the code that actually generates the primitive recursive predicate is given.

```
-- Bounded minimization (search)
bmin_model p (0:xs) = 1
bmin_model p (x:xs) = let r = (bmin_model p (x-1:xs))
                      in if r < x then r
                         else if p (x-1:xs) then x-1 else x


bmin p n = PR one
              (C ifte [C lt [P 2, P 1],
                       P 2,
                       C ifte [C p (P 1: take (n-1) (map (\j -> P j) [3..])),
                               P 1,
                               C S [P 1]]])
```

In this code, the expression `[3..]` indicates the infinite list of numbers starting at 3. These numbers are turned into instances of the projection function by the map function. Thus this expression:

```
map (\j -> P j) [3..])
```

stands for the infinite list of projection functions, `[P 3, P 4, ... ]`. The function `take` only requires the first `n-1` elements of this list. Haskell uses a technique called lazy evaluation that makes it possible to write functions that manipulate potentially infinite objects, as long as they ultimately only demand finitely many values.

For example, here is a hand formatted version of the output of a simple predicate generate by `bmin`:

```
*PRCantor> bmin false 1
PR (C S [Z])
   (C (PR (P 2) (P 3))
      [C (C (PR Z (C S [Z]))
            [PR (P 1) (C (PR Z (P 1)) [P 2])])
         [P 2,P 1],
       P 2,
       C (PR (P 2) (P 3))
         [C Z [P 1],P 1,C S [P 1]]])
```

Which can be simplified to:

```
*PRCantor> bmin false 1
PR one
   (C ifte
      [C lt
         [P 2,P 1],
       P 2,
       C ifte
         [C false [P 1],
          P 1,
          C S [P 1]]]])
```

Once bounded minimization is available, it is possible to write naive search-based algorithms to find values easily characterized by predicates. These algorithms tend to be inefficient, but they do demonstrate that the functions can be computed by functions in the class.

For example, we can implement division of natural number a by natural number b as a search for the least q such that $q * b \leq a$ and $a < (q+1) * b$. This condition is coded up as the three place predicate `isdiv` that expects arguments q, a and b:

```
isdiv = C and [ C lteq [C times [P 1, P 3], P 2],
                C lt   [P 2, C times [C S [P 1], P 3]]]
```

Using this predicate and bounded minimization we can implement natural number division as follows:

```
div = C (bmin isdiv 3) [C S [P 1], P 1, P 2]
```

In a similar manner, we can implement a very inefficient square root algorithm:

```
issqrt = C and [ C lteq [C times [P 1, P 1],P 2],
                 C lt   [P 2, C times [C S [P 1], C S [P 1]]]]
sqrt = C (bmin issqrt 2) [P 1, P 1]
```

With these functions defined, we can now give primitive recursive definitions that implement Cantor's pairing function. The function `pair` constructs pairs. The function `pi1` is the first (left) projection. The function `pi2` is the second (right) projection.

```
-- Cantor Pairing Function and Projections
pair = C plus [C div [C times [C plus [P 1, P 2],
                               C S [C plus [P 1, P 2]]],
                      mkconst 2],
               P 2]

w = C div [C pred [C sqrt [C S [C times [mkconst 8, P 1]]]],
           mkconst 2]
t = C div [C plus [C times [w,w],w], mkconst 2]
pi2 = C monus [P 1,t]
pi1 = C monus [w,pi2]
```

At this point we have demonstrated that primitive recursive functions are sufficiently powerful to express functions that allow data structures like lists, trees, and program abstract syntax, to be encoded in the natural numbers.

In particular, the primitive recursive pairing functions that can encode the descriptions of Turing machines, inputs to Turing machines, instantaneous descriptions, and computation histories.

We will assert without demonstration that, using similar techniques, we can define primitive recursive functions that test properties of lists, trees, and programs. However, henceforth we will demonstrate such ideas with programming languages, such as Haskell, Scheme, or C.

Problem 12.2 *Given a 3-place primitive recursive function f, write an equivalent 1-place function f' that takes as input a single number representing the triple as nested pairs and behaves as f. That is, $f'($*`pair`*$\ x_1$ $($*`pair`*$\ x_2\ x_3)) = f(x_1, x_2, x_3)$.*

Discussion The diagonalization presented here generalizes well to all effective enumerations of total functions, but it does not reflect much of the essence of the class of primitive recursive functions. Ackermann developed a function that diagonalizes over the primitive recursive functions in a different way. He constructs a function that grows faster than any primitive recursive functions. Hein discusses Ackermann's function in Section 13.2.3. He presents a proof sketch. If you wish to elaborate the proof sketch, consider calculating the number n based on the nesting of PR operations in the primitive recursive function.

Ackermann's diagonalization is in some ways more satisfying than the one presented above, but is specific to the primitive recursive functions.

12.4 Certifying Computation with Traces

One of the characteristics of a computational system is that it proceeds by a series of easily verifiable steps. For a given system, a set of these verifiable steps is called a trace. For the finite automata we only needed to check that the state transitions were appropriate for the input. For push down automata we had to additionally manage the stack contents. For Turing machines we introduced instantaneous descriptions or configurations and verified that one followed from another according to the definition of the machine.

How can we certify a computation for the primitive recursive or partial recursive functions? Each step is a very simple calculation. In this section we present a simple record of that calculation as a tree. The root node of the tree is at the bottom of the picture. The children of each node are those lines above that node that are indented exactly 3 spaces. For example, consider the adding 2 and 3 (e.g. $plus(2,3)$ = PR(P 1)(C S [P 2]) (2,3)). That calculation is described by the trace tree in the picture below.

```
10             S (4) = 5
9              P 2 (1,4,3) = 4
8           C S [P 2] (1,4,3) = 5
7                 S (3) = 4
6                 P 2 (0,3,3) = 3
5              C S [P 2] (0,3,3) = 4
4                 P 1 (3) = 3
3              PR (P 1) (C S [P 2]) (0,3) = 3
2           PR (P 1) (C S [P 2]) (1,3) = 4
1        PR (P 1) (C S [P 2]) (2,3) = 5
```

Each line represents an easily verified calculation step. Line 1 asserts that the root of the computation ($add(2,3)$) is equal to 5. To certify this we must verify the assertions on lines 2 and 8, which are the immediate sub-trees of the root (note the indentation). These two subtrees follow directly from the way a `PR` node is evaluated. Study the case for primitive recursion on non-zero inputs (the second line below) from Section 11.2

$$\begin{aligned} f(0,x_1,\ldots,x_k) &= h(x_1,\ldots,x_k) \\ f(n+1,x_1,\ldots,x_k) &= g(n,f(n,x_1,\ldots,x_k),x_1,\ldots,x_k) \end{aligned}$$

The first subtree `PR (P 1) (C S [P 2]) (1,3) = 4` (line 2) follows from the computation $f(n, x_1, \ldots, x_k)$, and the second subtree `C S [P 2] (1,4,3) = 5` (line 8) follows from the computation $g(n, f(n, x_1, \ldots, x_k), x_1, \ldots, x_k)$ where the sub term $f(n, x_1, \ldots, x_k)$ has been replaced by its value (4) to get $g(n, 4, x_1, \ldots, x_k)$.

The subtree on line 2 is also a `PR` term so it has subtrees on lines 3 and 5. This pattern repeats until a node without subterms (`Z`, `S`, or `(P n)`) is reached, in which case we have a simple assertion (lines 4, 6, 7, 9, and 10).

What rules are we using to verify the calculations? For the atomic primitive recursive functions Z, S and P_i, we directly verify that the result is obtained from the arguments correctly.

For the composition form $(C\ f\ [g_1,\ldots,g_k])(x_1,\ldots,x_n) = z$, we need to verify that the tree has the shape:

$$\begin{array}{ll} & f(y_1,\ldots,y_k) \ = \ z \\ & g_k(x_1,\ldots,x_n) \ = \ y_1 \\ & \vdots \\ & g_1(x_1,\ldots,x_n) \ = \ y_k \\ C\ f & [g_1,\ldots,g_k])(x_1,\ldots,x_n) \ = \ z \end{array}$$

Where each `C` node has $k+1$ subtrees Note that the subtrees beginning on lines 5 and 8 conform to this pattern.

For the primitive recursion form we have two cases: one for when the first argument is zero and one for when the first argument is nonzero. We consider them in order.

In the zero case we have the form $(\text{PR}\ f\ g)(0, x_2 \ldots, x_k) = z$. In this case we verify we have the tree has shape

$$\begin{array}{ll} & f(x_2,\ldots,x_n) \ = \ z \\ (\text{PR} & f\ g)\ (0, x_2 \ldots, x_k) \ = \ z \end{array}$$

Where the `PR` node has one subtree. This occurs in line 3 of the example.

For the nonzero case we have the form $(\text{PR}\ f\ g)(x+1, x_2 \ldots, x_k) = z$. In this case we need to verify that the tree has the shape

$$\begin{array}{ll} & g(x, r, x_2, \ldots, x_n) \ = \ z \\ & (\text{PR}\ f\ g)(x, x_2 \ldots, x_k) \ = \ r \\ (\text{PR} & f\ g)(x+1, x_2 \ldots, x_k) \ = \ z \end{array}$$

Where the `PR` node has exactly two subtrees. This pattern occurs in line 1 and 2 of the example.

Exercise 12.3 *Test your understanding of the example by showing the explicit correspondence between these rules and the example. Label each part of the example with the corresponding variables in the three patterns presented above.*

Exercise 12.4 *Develop trace trees for the following computations:*

1. `(PR (C S [Z]) Z) [0]`
2. `(PR (C S [Z]) Z) [1]`
3. `(PR Z (C S [Z])) [0]`
4. `(PR Z (C S [Z])) [1]`
5. `(PR (P 2) (P 3)) [1,2,3]`

Problem 12.5 *How can we verify a step of unbounded search? Develop verification rules for rule VI.*

12.4.1 Haskell code to generate and verify traces (optional)

To demonstrate that this is all directly computable we implement a version of trace trees in Haskell. We represent a step as a triple consisting of the function, the arguments, and the result. We represent this by the Haskell type `(PR,[Integer],Integer)`. A trace tree will store such a triple at every node in the tree. A trace tree node is *good* if every stored triple is consistent with the sub-trees of that node. A trace tree root justifies the final calculation in the trace. So, the example justifies that `plus` of 2 and 3 is 5.

First we give the Haskell code to define a trace tree.

```
data TraceTree x = Ztree x
          | Stree x
          | Ptree Int x
          | Ctree (TraceTree x) [TraceTree x] x
          | PRtree0 (TraceTree x) x
          | PRtreeN (TraceTree x) (TraceTree x) x
          | Stuck x
          deriving (Show, Eq)

tag (Ztree x) = x
tag (Stree x) = x
tag (Ptree n x) = x
tag (Ctree x xs y) = y
tag (PRtree0 y z) = z
tag (PRtreeN x y z) = z
tag (Stuck z) = z

res x = z where (_,_,z) = tag x
```

Note that each node type is tagged with a polymorphic component of type `x`. We can select this tag using the `tag` function. In our code the tag is always a triple `(PR,[Integer],Integer)`, so the function `res` selects the final result from a node.

The checking function is similar in its structure to the `eval` function, in that it analyzes the structure of the primitive recursive function.

```
trace2:: PR -> [Integer] -> TraceTree (PR,[Integer],Integer)
trace2 Z xs = Ztree (Z,xs,0)
trace2 S (x:xs) = Stree (S,(x:xs),x+1)
trace2 S [] = Stuck (S,[],0)
trace2 (P n) xs | n <= length xs = Ptree n (P n,xs,nth n xs)
trace2 (P n) xs = Stuck (P n,xs,0)
trace2 (C f gs) xs = let gtraces = map (\g -> trace2 g xs) gs
                         ys = map res gtraces
                         ftrace = trace2 f ys
                     in Ctree ftrace gtraces (C f gs,xs,res ftrace)
trace2 (PR g h) (0:xs) =
   let trace = trace2 g xs
   in PRtree0 trace (PR g h,0:xs,res trace)
trace2 (PR g h) (x:xs) =
   let rectrace = trace2 (PR g h) ((x-1):xs)
       r = res rectrace
       htrace = trace2 h ((x-1):r:xs)
       r' = res htrace
   in PRtreeN rectrace htrace (PR g h,x:xs,r')
trace2 (PR g h) [] = Stuck (PR g h,[],0)
```

The trace in the example was generated by this function, although it was pretty printed by a function (not shown) which indicates the tree structure by indenting.

In addition to generating trace trees it is possible to check them to see if they are good traces. The following code verifies that a trace is good. The `do` construct propagates the failure behavior of Maybe using a mechanism called a Monad.

```
check:: TraceTree (PR,[Integer],Integer) -> PR -> Maybe Integer
check (Ztree (Z,xs,0)) Z = Just 0
check (Stree (S,x:xs,y)) S =
   if y == (x+1) then Just y else Nothing
check (Ptree n (P m,xs,i)) (P k) =
   if (n==m) && (k==m) && (m <= length xs) && (nth m xs == i)
      then Just i
      else Nothing
check (Ctree t ts (C f gs,xs,i)) (C h ks) =
  do { ys <- zipM check ts gs
     ; j <- check t f
     ; let (_,zs,k) = tag t
     ; if (zs == ys) && (j == i) && (k == i) && (h==f) && (gs==ks)
          then (Just i)
          else Nothing }
```

```
check (PRtree0 t (PR f g,0:xs,i)) (PR h j) =
  do { z <- check t f
     ; if (z==i) && (f==h) && (g==j)
          then Just i
          else Nothing }
check (PRtreeN s t (PR f g,n:xs,i)) (PR h j) =
  do { r <- check s (PR f g)
     ; k <- check t g
     ; let (_,ws,_) = tag s
           (_,zs,_) = tag t
     ; if (f==h) && (g==j) && (k==i) && (ws==(n-1):xs) && (zs==(n-1):r:xs)
          then (Just i)
          else Nothing }
check x y = Nothing
```

Problem 12.6 *Extend to the partial recursive functions. That is, build a verifier and tracing evaluator to verify and generate verifiable traces for the* `MuR` *functions.*

As you study this program look at the structure of the recursion. The check function is defined by induction on `TraceTree`. It is a total function.

Exercise 12.7 *Define a function* `verify` *that takes a* `TraceTree` *for a computation and returns a Boolean.*

12.4.2 Trace Summary (NOT optional).

Note: This section does not follow the notation from Hein. In some cases it follows notation from Hopcroft, Motwani, and Ullman.

Consider the following descriptions of traces for variaous systems.

- **DFA.** A DFA $A = (Q, \Sigma, S, F, T)$ accepts a string $x_1 x_2 \dots x_n$ if there exists a sequence of states $q_1 q_2 \dots q_n q_{n+1}$ if
  1. $q_1 = S$
  2. $q_{i+1} = T(q_i, x_i)$
  3. $q_{n+1} \in F$

  Given such a DFA, a string, and a sequence of states, it is easy to verify whether or not the string is accepted. (A Hein consistent definition of acceptance appears in this document as Definition 3.2.)

- **PDA.** A PDA[5] can be defined as a 7-tuple: $A = (Q, \Sigma, \Gamma, \delta, S, Z_0, F)$. A configuration (or Instantaneous Description (ID)) is a triple (q, w, α) such that

[5]See Definition 10.1 for a Hein consistent definition of PDA. Refer to class notes for the corresponding definition of acceptance. The definition Hook presented in lecture corresponds more closely to that found in Sipser than the one presented here.

1. q is the current state
2. w is the remaining part of the input
3. α is the current content of the stack, with top of the stack on the left

Two IDs are related: $(q, aw, X\beta) \vdash (p, w, a\beta)$ when $(p, \alpha) \in \delta(q, a, X)$

A PDA accepts a string $w = x_1 x_2 \ldots x_n$ by final state if there exists a sequence of related IDs $id_1 \vdash id_2 \vdash \ldots \vdash id_n$ such that.

1. $id_1 = (S, w, Z_0)$
2. $id_n = (p, \Lambda, \alpha)$ for some state $p \in F$, and any stack α.

Given a chain of related IDs, it is easy to verify that a string is accepted.

- **Turing Machine.** A Turing machine can be defined as a 7-tuple: $A = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$. [6] An ID for a TM is a string of the form $\alpha q \beta$ where

  1. $\alpha \in \Sigma^*$
  2. $\beta \in \Sigma^*$
  3. $q \in Q$
  4. The string α represents the non-blank tape contents to the left of the head.
  5. The string β represents the non-blank tape contents to the right of the head, including the currently scanned cell.

  Two IDs are related

  1. if $\delta(q, x_i) = (p, Y, \textbf{Right})$. then $(x_1 \ldots x_{i-1} q x_i \ldots x_k) \vdash (x_1 \ldots x_{i-1}\ Y\ p\ x_{i+1} \ldots x_k)$
  2. if $\delta(q, x_i) = (p, Y, \textbf{Left})$. then $(x_1 \ldots x_{i-1} q x_i \ldots x_k) \vdash (x_1 \ldots\ p\ x_{i-1}\ Y\ x_{i+1} \ldots x_k)$

  A turing machine accepts a string w if exist a sequence of related IDs $id_1 \vdash id_2 \vdash \ldots \vdash id_n$ such that.

  1. $id_1 = q_0\ w$
  2. $id_n = \alpha\ p\ \beta$ where $p \in F$

  Given a chain of related IDs, it is easy to verify that a string is accepted.

- **Primitive Recursive Functions.** A primitive recusive function p applied to arguments $(x_1, \ldots, x_n)$ computes a value n, if there exists a trace tree t, such that the tag of t is $(p, (x_1, \ldots, x_n), n)$ and *check* $t\ p$ = Just n

[6] In the Hook section an isomorphic definition was used, but with Hein's notation. The primary differences are that instead of a transition function, Hein presents a set of transitions instructions. Also, in addition to moving the tape head left and right, Hein specifies an operation that does not move the tape head.

Conclusion: All the computational systems we have studied can be verified by simple terminating programs, given the appropriate trace object. It may be very expensive to find a trace, but given a trace it is easy to check that if the trace really describes the computation. In addition the strategy we showed for the primitive recursive functions easily extends to traces for the partial recursive functions.

Problem 12.8 *Consider the context free grammars. Outline what a trace for a context free grammar might look like. Define a simple CFG, and provide a trace in the form you described above for a short string.*

12.5 Partial Recursive Functions are Turing Complete

We will briefly return to a point made at the end of Section 11.6. The partial recursive functions are Turing complete.

We review the evidence supporting this claim.

1. We can represent Turing machine descriptions as numbers using primitive recursive pairing functions.

2. We can represent Turing machine instantaneous descriptions and computation histories as numbers using pairing functions.

3. We can write a primitive recursive predicate, known as the Kleene T predicate, that takes the description of a Turing machine, e, the input to the Turing machine, x, and a computation history of the Turing machine, y, and returns 1 if y is a halting computation history of machine e on input x, and 0 otherwise.

4. We can write a primitive recursive function, U, that takes a halting computation history y and returns the final output of the computation.

5. Given a Turing machine e and input x, we can use unbounded search to define a partial recursive function of e and x that returns the least y such that $T(e, x, y)$ holds. That function can be composed with U to yield a partial function θ such that $\theta(e, x) = z$ if and only if the Turing machine e on input x computes value z.

The last three steps of this argument are asserted without proof.

12.6 Limits of Computability

The first step in exploring the limits of computability is to understand how non-termination can be handled. We start by taking two concepts we have developed above, evaluation and verification, and observing that even when evaluation is partial we can make the corresponding verification total, provided we have a certificate justifying the calculation.

In Section 12.4 we showed explicitly how to construct a verifier for the primitive recursive functions. We left as an exercise the verification of the partial recursive functions (Problems 12.5, 12.6). We have described in Section 12.5 how to verify computation histories for Turing machines.

In this section we assume that the partial recursive functions have been implemented and are represented by the datatype `MuR` (for mu-recursive, or μ-recursive). The `MuR` type has the same constructors as `PR`, plus the additional constructor `Mu`. As before we have a function `eval` that takes a `MuR` and a list of `Integer` arguments and returns an `Integer`. However, now `eval` is a partial function. For example, the `MuR` program `Mu S` is undefined on all values. The `eval` function goes into an infinite loop when given this program. (These datatypes and functions were developed in Problem 11.5).

Furthermore, we assume that we have developed a tracing evaluator, `trace`, for the `MuR` functions, and a trace verifier, `verify`, that determines if a trace is valid. These functions were described in Problem 12.6. They are very similar to the analogous functions on `PR` given above, except that, like `eval`, the trace generator can run forever. Of course the verifier can only verify finite traces. Note that the verifier is still a total function.

It is convenient to repackage `verify` as the following function `valid`, which captures the goal that it is verifying:

```
valid result trace = verify trace && (last trace == result)
```

The first observation we make is that if the `MuR` function f applied to numbers $n_1, \dots, n_k$ is defined and equal to z, then `eval` of $f(n_1, \dots, n_k)$ returns z and `trace` produces a certificate `c` such that `valid` $(f(n_1, \dots, n_k) = z)$ `c` returns `True`.

The construction of the functions `trace` and `valid` can be seen as a proof of this statement.

The next observation is that if there is a certificate that validates $f(n_1, \dots, n_k) = z$ then $f(n_1, \dots, n_k)$ is indeed defined and equal to z. These observations are combined in the theorem:

Theorem 12.4 `eval` $f\ (n_1, \dots, n_k) = z$ *if and only if there exists a certificate* c *such that* `valid` $(f(n_1, \dots, n_k) = z)\ c$.

Note that this theorem relates the partial function `eval` to the total function `valid`.

12.6.1 Halting

In Section 12.2, we used the technique of diagonalization to construct a total function that was not primitive recursive. In this section we will use a similar technique to prove that there is not a total `MuR` function `halt` with the specification that `halt` $f\ (n_1, \dots, n_k)$ if and only if $f(n_1, \dots, n_k)$ is defined.

Suppose there were such a function. Then we could construct the function:

```
diagonal x = if (halt (dMuR x) (repeat x)) then loop else 0

loop = loop    -- a value that loops
```

If we apply `diagonal` to the code for a value that loops when given its own description as input, such as `Mu S`, we should get a function that terminates and returns 0. On the other hand, if we apply `diagonal` to a program that returns a value when applied to its own description, then the diagonal function will loop.

Since by supposition `halt` is a partial recursive function, the function `diagonal` is also partial recursive. Since it is a partial recursive function, there is formal description of it in the type `MuR`. Call that description p. Furthermore, p can be encoded by a natural number, n using the `eMuR` function. That is there is an n such that $n =$ `eMuR` p.

Consider `diagonal` n.

Elaborating the definition gives:

```
if (halt (dMuR n) (repeat n)) then loop else 0
```

Since we know n to represent p, and know p is a one place function, this can be rewritten as:

```
if (halt p [n]) then loop else 0
```

We proceed by cases on `halt p [n]`:

- `halt p [n] = true`

  In this case `diagonal n` simplifies to `loop`. But that contradicts that $p =$ `diagonal` since p halts on n while `diagonal` loops.

- `halt p [n] = false`

  In this case `diagonal n` returns 0. This also contradicts $p =$ `diagonal` since p does not halt on n while `diagonal` returns the value 0.

Having obtained contradictions in both cases, we conclude that there is no partial recursive function equivalent to `halt`, as we had assumed.

This proves that the halting problem is not computable:

Theorem 12.5 *There is no total computable function that computes* `halt`*.*

The halting problem can be formulated for all reasonable Turing complete models of computation.

13 Consequences of the Halting Problem

So what do we know exactly about the Halting problem?

We know that if a partial recursive function p is defined on input x then the `eval` function will terminate and find the answer. That means we can algorithmically say "yes" to the question "does p halt on x?" in all cases where the answer is "yes".

However, we know that there is no total function that decides the halting problem.

Are these statements in conflict? Not exactly.

We will classify how hard a problem is by how well a computable function can answer a language membership question. There are three cases we look at today: (1) we can definitively say yes or no to any question is $x \in A$ with a total computable function, (2) we can definitively say yes if $x \in A$, but in some cases where $x \notin A$ the function is undefined, and (3) none of the above: there are both $x \in A$ and $y \notin A$ on which the function in question is undefined.

In the first case we will say that A is *decidable*. There is a total function p that says yes or no to all questions about membership in A. ($p\ x \leftrightarrow x \in A$)

In the second case we will say that A is *recognizable* (Hein calls this *partially decidable*). That is, there is a partial recursive function that accepts all elements of A. ($x \in A \rightarrow p\ x$)

The third case we will refer to as *not recognizable*.

In the context of this vocabulary, we will say that the set of pairs (p, x) such that p halts on input x is recognizable but not decidable.

In practical terms, decidability is a strongly desirable property. If a problem is not decidable then any total algorithmic solution will be approximate. That is, if it always generates an answer it will get some answers wrong!

Do undecidable problems arrise in practice? The answer to this is yes, particularly in the context of properties of programs. There is a general theorem, known as Rice's theorem, that says that any non-trivial property of program behaviors is undecidable. That is, if ϕ is a nontrivial property of programs, and we want $\phi(p)$ to imply $\phi(q)$ whenever p and q behave the same on all inputs (i.e. $\forall x. p(x) = q(x)$), then ϕ is undecidable.

14 Reflection on Course Objectives

Recall the course objectives. I add comments on how well they have been addressed in this offering.

1. Find regular grammars and context-free grammars for simple languages whose strings are described by given properties.

   I believe students should have significant mastery of the ability to define regular and context free languages from natural language descriptions.

2. Apply algorithms to: transform regular expressions to NFAs, NFAs to DFAs, and DFAs to minimum-state DFAs; construct regular expressions from NFAs or DFAs; and transform between regular grammars and NFAs.

   Students should have significant mastery of some of these algorithms. The notion that formal descriptions can be manipulated algorithmically to build related formal descriptions should now be familiar.

3. Apply algorithms to transform: between PDAs that accept by final state and those that accept by empty stack; and between context-free grammars and PDAs that accept by empty stack.

   As above.

4. Describe LL(k) grammars; perform factorization if possible to reduce the size of k; and write recursive descent procedures and parse tables for simple LL(1) grammars.

   Not discussed in detail.

5. Transform grammars by removing all left recursion and by removing all possible productions that have the empty string on the right side.

   Some familiarity in homework.

6. Apply pumping lemmas to prove that some simple languages are not regular or not context-free.

   Experience in homework and exams.

7. State the Church-Turing Thesis and solve simple problems with each of the following models of computation: Turing machines (single-tape and multi-tape); while-loop programs; partial recursive functions; Markov algorithms; Post algorithms; and Post systems.

   Basic ideas covered in lecture and study questions. Not all formal systems considered.

8. Describe the concepts of unsolvable and partially solvable; state the halting problem and prove that it is unsolvable and partially solvable; and use diagonalization to prove that the set of total computable functions cannot be enumerated.

   Last two lectures. Exposure. Not mastery.

9. Describe the hierarchy of languages and give examples of languages at each level that do not belong in a lower level.

   Integrates the whole class.

10. Describe the complexity classes P, NP, and PSPACE.

    Not discussed