CSE 4111 Computability

Jeff Edmonds

Assignment 1

Due: One week after shown in slides

First Person:	Second Person:
Family Name:	Family Name:
Given Name:	Given Name:
Student #:	Student #:
Email:	Email:

Guidelines:

- You are strongly encouraged to work in groups of two. Do not get solutions from other pairs. Though you are to teach & learn from your partner, you are responsible to do and learn the work yourself. Write it up together. Proofread it.
- Please make your answers clear and succinct.
- Relevant Readings: 01-Model Slides from the lectures and Cooks (pg 54) notes
- This page should be the cover of your assignment.

Problem Name	Max Mark	
1 Primitive Recursion	10	
2 Number of Registers	10	
3 Running Time	10	
4 Circuit Depth	10	
5 TM to Circuit	10	
6 Poly Size Circuit	10	
7 Easy Context Free Grammar	10	
8 Harder Context Free Grammar	10	
9 Parsing	10	
10 Harder Parsing	10	
11 Context Sensitive Grammar	10	
Total	110	

1. Primitive Recursion

   (a) The slides computes whether or not y is divisible by x. Explain how you would compute whether x is a power of 2, i.e. 2^r for some integer r.

   (b) In the slides, I give an algorithm for computing $\lfloor \frac{y}{x} \rfloor$. Give a similar primitive recursive program for computing $Log(y) = \lfloor \log_2 y \rfloor$. The algorithm counts the number of powers of two less than or equal to y.

   (c) Somebody gave me another fun solution. Recursively define $MinLog(x, y) = \min(\lfloor \log_2 x \rfloor, y)$. Then define $Log(x) = \lfloor \log_2 x \rfloor$ from $MinLog(x, y)$.

2. Number of Registers: Recall that each *Register Machine* is defined to have some fixed finite number of registers that can depend on the computational problem being solved, but not on the size of the input. Now we need to prove that the number of registers used is in fact bounded. Moreover, I am curious whether limiting the number of registers to be say at most 10, limits what register machines can compute.

   (a) Give code for a register machine using as few registers as possible to compute the following:

```
if($R_i = 0$) goto line $k_0$
if($R_i = 1$) goto line $k_1$
if($R_i = 2$) goto line $k_2$
if($R_i = 3$) goto line $k_3$
...
if($R_i = r$) goto line $k_r$
end algorithm
```

   (b) Given a recursive algorithm for a problem, the slides describes how to convert this into a register machine algorithm. How many registers does this algorithm use as a function of the complexity of the given recursive algorithm?

   As a warm up, suppose that f is defined using the primitive recursive call $f(x, y) = h(x, y - 1, f(x, y - 1))$. Suppose that your implementation of h requires q resisters. How many does your implementation of f require? Be sure to reuse registers that you are done with.

   Recall that the recursive JAVA program $factorial(y) = y \times factorial(y - 1)$, unwinds giving y different stackframes. This is way too much memory. This is why the slides turns this recursion into a loop.

   (c) Forget recursive algorithms. Prove that register machines can compute anything *computable* by proving that any TM can be simulated by a register machine. Hint: The proof is identical to that in the slides for recursive algorithms simulating TMs. To make your life easier, if the input to the TM is a binary string s, let the input to the register machine be the integer x obtained by taking of the reverse of s in binary. Output is represented in the same format. Give the code for the register machine. Again use pseudo code completes with loop invariants. Feel free to refer any program in the slides as a subroutine. Use meaningful register/variable names.

   (d) How many registers does your register machine use as a function of the complexity of the TM that it is simulating? Does limiting the number of registers to be say at most 10, limits what register machines can compute. What can you compute with only one register? two?

3. Running Time: A strengthened version of Church's Thesis states that all "reasonable" models of computation are able to compute in time within a polynomial of each other. More formally, for any

two "reasonable" models of computation M_1 and M_2, there exists a constant c (depending on M_1 and M_2 but not on P) such that for any computational problem P, if M_1 requires time $T(n)$, then M_2 requires at most time $(T(n) \times n)^c$. For each of the following models, give and explain it's running time for computing $x + y$ in terms of the inputs x and y. Also give it's running time in terms of the number of bits $n = \log_2(x) + \log_2(y)$ to represent the input.

(a) Single tape TM.

(b) Three tape TM.

(c) Primitive Recursive.

(d) Register Machine.

In terms of the strengthened version of Church's Thesis, are these all "reasonable" models? If not, what is the key operation that the faster models has that needs to be given to the slower ones so that they are "reasonable" as well?

4. Circuit Depth: The goal of this questions is to multiply two n-bit binary integers with a polynomial sized *and/or/not* circuit with minimum depth. The first step multiplies the first number x with each of the bits y_i of the second number. This of course is simply x if $y_i = 1$ and zero if $y_i = 0$. The i^{th} of these is shifted by i bits as required. The answer z is simply the sum of these n integers. What remains is to solve the following different problem. The input consists of n different $2n$-bit binary integers $x_1, \ldots, x_n$. The output is their sum $z = \sum_i x_i$.

   (a) We saw in the slides that adding two n-bit binary integers requires depth $\mathcal{O}(\log n)$ depth. Using this as a black box, design a circuit for summing the n integers. What is the minimum depth using this method?

   (b) Here is a new problem. The input consists of three different n-bit binary integers x_1, x_2, and x_3. The output consists of two $n+1$-bit binary integers y_1 and y_2 such that $x_1 + x_2 + x_3 = y_1 + y_2$. Note that there is lots of freedom in how you choose the output values. The trick is that you must be able to compute it with a constant depth circuit. Hint: As a black box, use a circuit whose input is three binary values a, b, and c and whose output is two binary bits $sum_{low}(a, b, c)$ and $sum_{high}(a, b, c)$ where sum_{low} is the low order bit of the sum $a + b + c$ and sum_{high} is the high order bit of the sum. For example if $a = b = c$ than the sum is $3 = 11_2$ and hence $sum_{low} = sum_{high} = 1$.

   (c) Using the circuit for the last question as a black box, design a circuit for summing the n integers. What is the minimum depth using this method?

   (d) Prove that no binary circuit can multiply two integers with any smaller depth (within a constant factor).

5. TM to Circuit: Given a TM that runs in time $T(n)$, we described in class how to build a circuit of size $T^2(n)$. We quickly stated that if the movement of the TM's head was predictable then the circuit could be build much smaller.

   (a) Use this much space to explain how this is done. What is the size of the circuit?

   (b) Let the location of the TM's head one input I at time t be denoted $HeadPosition(I, t)$. For your construction to work what can this location depend and not depend on? Can it depend on I, on t, on $n = |I|$?

(c) Express in first order logic your answer to the last question.

6. Poly-Size Circuits: Can polynomial sized circuits compute more than poly-time TMs? Consider the following odd problem. The n-bit input I is broken into three blocks. The first $4 \log n$ bits is interpreted as the description of a JAVA program J. The second $3 \log n$ bits is interpreted as an input I'. The remaining $n - 7 \log n$ bits of I are simply ignored. The required output is whether J on input I' halts, denoted $J(I') = halts$.

   (a) Describe a non-uniform circuit using binary *and*, *or*, and *not* gates that computes this problem. Try to use as few gates as possible. Recall non-uniform means that there is a completely different circuit for each input size n that we might not know how to construct.

   (b) How many gates are used as a function the input size n? Is this a polynomial sized circuit?

   (c) What depth is needed?

   (d) What time is needed to compute this problem by a TM? Can it be done in polynomial time?

7. Consider s = "(((1) * 2 + 3) * 5 * 6 + 7)".

   (a) Give a derivation of the expression s.

   (b) Draw the tree structure of the expression s.

   (c) Trace out the execution of your program on $GetExp(s, 1)$. In other words, draw a tree with a box for each time a routine is called. For each box, include only whether it is an expression, term, or factor and the string $s[i], \ldots, s[j-1]$ that is parsed.

8. Context Free Grammar: Give a context free grammar for generating the language $\alpha\#\beta$ where $\alpha, \beta \in \{0,1\}^*$, with $\alpha \neq \beta$. Hints:

   (a) They might have different lengths, i.e $\{0,1\}^i \ \# \ \{0,1\}^i \ \{0,1\} \ \{0,1\}^*$.

   (b) or they might differ in the $i+1^{st}$ bit, i.e. $\{0,1\}^i \ 0 \ \{0,1\}^* \ \# \ \{0,1\}^i \ 1 \ \{0,1\}^*$.

   (c) Which blocks need to be linked and hence must get spewed together?

   (d) Be sure to give meaning and explanation to your nonterminals. The easiest way is to give the regular expression generated by each nonterminal. For example say "A generates any string $\{0,1\}^*$".

9. Parsing

   Look Ahead One: A grammar is said to be *look ahead one* if, given any two rules for the same non-terminal, the first place that the rules differ is a difference in a terminal. (Equivalently the rules can be views as paths down a tree.) This feature allows our parsing algorithm to look only at the next token in order to decide what to do next. Thus the algorithm runs in linear time. An example of a good set of rules would be:

   A ⇒ B 'u' C 'w' E
   A ⇒ B 'u' C 'x' F
   A ⇒ B 'u' C
   A ⇒ B 'v' G H

(Actually, even this grammar could also be problematic if when $s = bbbucccweee$, B could either be parsed as bbb or as $bbbu$. Having B *eat* the $'u'$ would be a problem.)
An example of a bad set of rules would be:

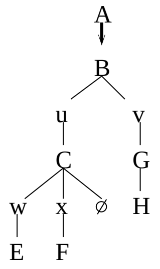
A ⇒ B C
A ⇒ D E

With such a grammar, you would not know whether to start parsing the string as a B or a D. If you made the wrong choice, you would have to back up and repeat the process.

Consider a grammar G which includes the four look ahead rules for A given above. Give the code for $GetA\,(s,i)$ that is similar to that for $GetExp\,(s,i)$. We can assume that it can be parsed, so do not bother with error detection. HINT: The code should contain NO loops.

10. Harder Parsing: If you are feeling bold, try to write a recursive program for a generic parsing algorithm. The input is $\langle G,T,s,i\rangle$, where G is a look ahead one grammar, T is a non-terminal of G, s is a string of terminals, and i is an index. The output consists of a parsing of the longest substring $s[i], s[i+1], \ldots, s[j-1]$ of s that starts at index i and is a valid "T" according to the grammar G. In other words, the parsing starts with non-terminal T and ends with the string $s[i], s[i+1], \ldots, s[j-1]$. The output also includes the index j of the token that comes immediately after the parsed expression. For example, $GetExp(s,i)$ is the same as calling this algorithm on $\langle G, exp, s, i\rangle$ where G is the grammar given above.

    It is helpful to think of the grammar as a tree. The grammar from the last question would be:

    The loop invariant is that you have parsed a prefix $s[i], s[i+1], \ldots, s[j'-1]$ of s producing a partial parsing p and the rest of the string $s[j'], s[j'+1], \ldots, s[j-1]$ will be parsed using the one of the partial rules in the set R. For example, suppose the grammar G includes the four look ahead rules for A given above, we are starting with the non-terminal $T = A$, and we are parsing the string $s = bbbucccweee$. Initially, we have parsed nothing and R contains all of each of the four rules, namely $R = \{$BuCwE, BuCxF, BuC, BvGH$\}$. After two iterations, we have parsed $bbbu$ using a parsing p_B for bbb followed by the character u. We must parse the rest of the string $cccweee$ using one of the rules in $R = \{$CwE, CxF, C$\}$. Note that the used up the prefix Bu from the consistent rules and the inconsistent rules were deleted. Because the grammar is *look ahead one* we know that either the first token in each rule of R is the same non-terminal B, or each rule of R begins with a terminal or is the empty rule. These are the two cases your iteration needs to deal with.

11. Context Sensitive Grammar: There are no context free grammars for generating the language $\alpha\#\alpha$ where $\alpha \in \{0,1\}^*$. (The proof is an ugly pumping lemma thing, which you don't have to do.) Give a context sensitive grammar for generating this language.

    A loop invariant stating what string that grammar now has is REQUIRED. Give meaning and explanation to your nonterminals.

The following are hints about two different ways to do it. Think about both, but only hand in one.

(a) Start with $S \to S'{<}$
  $S' \to 0S'0 \mid 1S'1 \mid \#{>}Q$
  This produces the string $\alpha\#{>}Q\alpha^R{<}$
  Then move the head Q back and forth to reverse the order of α^R giving α.

(b) Start with $S \to S'{<}$
  $S' \to 0S'C_0 \mid 1S'C_1 \mid \#$
  This produces the string $\alpha\#\beta{<}$
  where $\beta = \alpha^R$, except 0 is replaced by C_0 and 1 by C_1.
  Then have the C_0 and the C_1 move on their own and convert when in place to 0 and 1.