Final Examination

9:00 – 12:00, Tuesday, August 2, 2016 Instructor: Alex Brodsky

Name: ______________________

Student Number: ______________________

Student Signature: ______________________

Duration: 180 minutes
Aids allowed: None

1. Place your student card on the table beside you, it will be checked during the exam.
2. This examination has 18 pages. Ensure that you have a complete paper.
3. The use of calculators, computers, books, papers, memoranda, cell phones, or any other electronic device is strictly prohibited.
4. Place your book-bags, coats, and books at the front of the room.
5. You may not reenter the examination once you leave.
6. You may not leave the examination in the last 15 minutes of the exam.
7. You must hand in the exam. You may not remove the exam from the room.
8. You may not ask questions of invigilators, except in cases of supposed errors or ambiguities in examination questions.
9. Write your answers in the space provided, if you need more space use the back of the **current** page and **indicate** this in the provided space.
10. **Your answers should be at most 3 sentences long!**
11. No smoking is permitted.
12. Write legibly and neatly.
13. Complete as much of the exam as you can.
14. Good Luck!

Question	Value
1	/30
2	/10
3	/20
4	/20
5	/15
6	/20
7	/20
8	/15
Total	/150

1. [30 Marks] Answer True or False for the following questions and **provide a brief (one or two sentence) justification:**

   (a) [3] __________ A scanner generates a parse tree from stream of tokens.

   False. A parser takes in a stream of tokens **[1]** and produces a parse tree**[1]**. A scanner takes in a program**[1]** and outputs a stream of tokens. **[1]**

   (b) [3] __________ If L is a nonregular language then so is $\overline{L}$.

   True. Since regular languages are closed under complement**[1]**, if $\overline{L}$ was regular, then $\overline{\overline{L}} = L$ must also be regular. Hence, since L is not regular, $\overline{L}$ is not regular**[1]**.

   (c) [3] __________ Even if a grammar is LL(1) the language it specifies may still be inherently ambiguous.

   False. A language is inherently ambiguous if all grammars for it are ambiguous.**[1]** Since an LL(1) grammar is not ambiguous,**[1]**, the language cannot be inherently ambiguous.**[1]**

   (d) [3] __________ All local variables are bound at run-time.

   True. Local variables are stored on the stack (in a stack frame)**[1]**. Since the location of the stack frame is not known until runtime, local variables cannot be bound until runtime.**[1]**

   (e) [3] __________ Tail recursion is not possible in any function that calls itself multiple times.

   True. Tail recursion requires that the recursive call be the last action in the function.**[1]** If a function performs multiple recursive calls, than at least one of the recursive calls is not the last action in the function.

(f) [3] ____________ If DFAs M_1 and M_2 are different, then the languages $L(M_1)$ and $L(M_2)$ recognized by these DFAs must also be different.

False. For every infinite regular language there are infinitely many DFAs that recognize it.[1] Thus, M_1 and M_2 may be two DFAs that recognize the same language.[1] A counter example is also sufficient[2].

(g) [3] ____________ If a grammar G is such that for any two productions with the same left-hand size, $A \rightarrow \alpha$ and $A \rightarrow \beta$ such that FIRST(α) is disjoint from FIRST(β), then the grammar is LL(1).

False. If either FIRST(α) or FIRST(β) includes ϵ **[1]**, then PREDICT($A \rightarrow \alpha$) and PREDICT($A \rightarrow \beta$) may not be disjoint**[1]**, and hence G may not be LL(1).**[1]**

(h) [3] ____________ A synthesized attribute grammar can be used to enforce a rule requiring types to be declared before use.

False. Synthesized grammars propagate information up the parse tree.**[1]** But, to enforce the above rules requires an inherited grammar, which propagates information across the tree (left to right)**[1]**.

(i) [3] ____________ A closure called with the same arguments, will always return the same value.

False. If a closure performs an operation with side-effects**[1]** these could modify its reference environment and hence affect the values it returns.**[1]**

(j) [3] ____________ A generator can be implemented using a continuation.

True. Instead of returning a value, generators are functions that yield a value and then continue from the previous yield when they are called again. **[1]** A continuation can be used to return to the yield statement each time the generator is called.**[1]**

2. [5 Marks] These questions deal with general programming languages concepts.

   (a) [5] For each of the following errors, state which phase of program translation would be responsible for catching the error. Assume all snippets of code are in Java.

   ```
   foo( 1 2 )
   ```

   Syntax (Parser)

   ```
   3 = a
   ```

   :

   Either syntax (Parser) or Semantic Analysis

   ```
   void foo() { return 3; }
   ```

   Semantic Analysis

   ```
   int i = 3K;
   ```

   Lexical (Scanner or Tokenizer)

   ```
   if a == b return c;
   ```

   Syntax (Parser)

   (b) [2] A compiler is typically separated into two parts: a front-end and a back-end. Why is this distinction important?

   The front-end is platform independent, translating a program into a platform independent representation,**[1]** from which a platform dependent representation can be generated by the front-ends to back-ends**[1]** to create a variety of compilers.**[1]**

   (c) [3] Would it be easier to write an interpreter or a compiler for Scheme? Why?

   An interpreter**[1]**. Scheme supports the dynamic creation of functions**[1]** and continuations**[1]**, both of which can not easily be done in compiled code**[1]** without including an interpreter as part of the runtime.**[1]**

3. [20 Marks] These questions deal with scanners and regular languages.

   (a) Give regular expressions for the following languages. You may use a dot (.) to denote any symbol in the alphabet and the notation $[a - b]$ to denote a range of symbols from an alphabet. E.g. $[a - d]$ is equivalent to $(a|b|c|d)$.

      i. [2] The language of all English words that do not have two consecutive vowels. Note: you may use $\mathcal{V}$ to denote the range of all vowels and $\mathcal{C}$ to denote the range of consonants.

      $(\mathcal{V}|\epsilon)(\mathcal{C}\mathcal{C} * \mathcal{V}) * \mathcal{C}*$

      ii. [2] The language of integers that are not divisible by 100. Note: integers may have leading 0's. I.e., 007 is a valid integer.

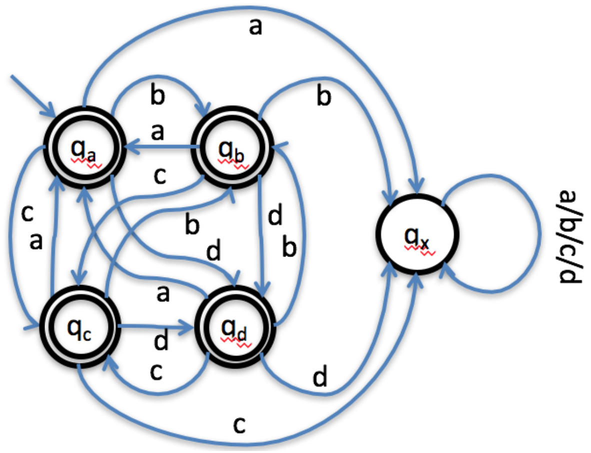
      Let $D = [0 - 9]$ and $P = [1 - 9]$. The regex is: $(D * (PD \mid P)$

      iii. [2] The language, over the alphabet $\Sigma = \{X, Y, Z\}$, consisting of strings that have exactly three Xs or exactly three Ys, or exactly three Zs. E.g., $XYZXYZX$ and $ZZYZ$ are in the language, but $XYXYXYXY$ is not.

      Let $\overline{X} = Y|Z$, $\overline{Y} = X|Z$, and $\overline{Z} = X|Y$. Then the Regex is: $(\overline{X} * X\overline{X} * X\overline{X} * X\overline{X}*)| \ (\overline{Y} * Y\overline{Y} * Y\overline{Y} * Y\overline{Y}*)| \ (\overline{Z} * Z\overline{Z} * Z\overline{Z} * Z\overline{Z}*)$

   (b) [5] Give a DFA that recognizes the language over the alphabet $\Sigma = \{a, b, c, d\}$ that consists of words where no two adjacent characters are the same. E.g., "abba" is not in the langauge, but "abcba" is.

   The following DFA accepts the specified language.

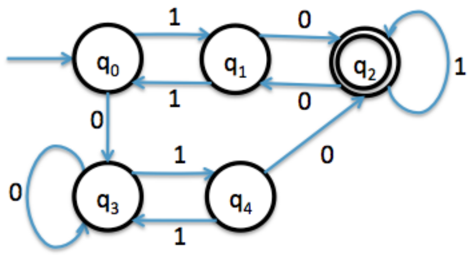


(c) [3] Describe a deterministic approach to determine if two regular expressions specify the same language. I.e., you could write a program to do this.

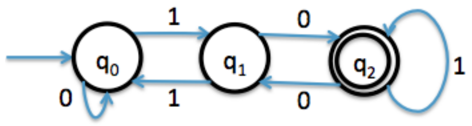
Given two REs,

i. Construct NFAs for respective REs. **[1]**
ii. Convert the NFA into to a DFA. **[1]**
iii. Minimize the DFAs. **[1]**
iv. If the DFAs are the same, the RE's specify the same language.**[1]**

(d) [6] Minimize the following DFA.



The following DFA is minimal.



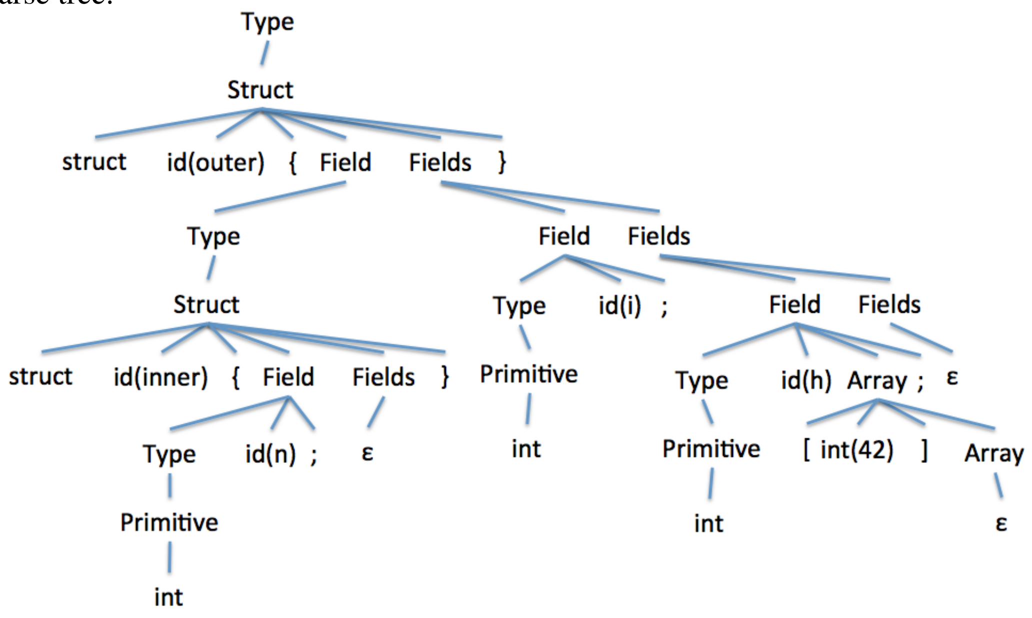
4. [20 marks] These questions deal with parsing and context free grammars.

   (a) Consider the grammar in Figure 1 (last page of this exam) and the following code.

```
struct outer {
  struct inner {
    int n;
  } s;
  int i;
  float h[42];
}
```

   i. [5] Using this grammar, derive a parse tree for the expression

Parse tree:



   ii. [4] Suppose an LL(1) parser was parsing this snippet according to the grammar in Figure 1. Show how the parser's stack looks before and after it performs the first three derivations.

S0 [Type]
S1 [Struct]
S2 [struct id { Field Fields }]
S3 [Type id Array ; Fields }]
or [Type id ; Fields }]

(b) [3] Describe the process for determining whether a grammar is LL(1).

(i) Construct the FIRST and FOLLOW tables**[1]**. (ii) Using the tables construct the PREDICT table**[1]**. (iii) If the PREDICT sets of the productions with the same LHS are disjoint, then the grammar is LL(1).**[1]**

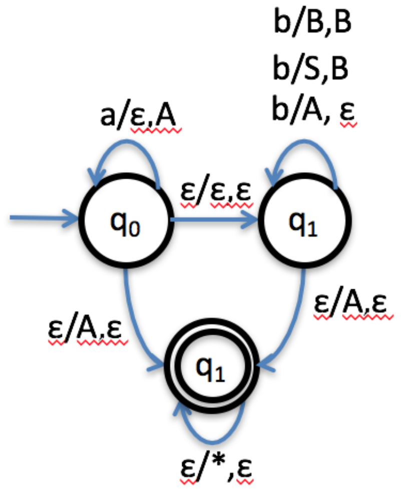
(c) Consider the language $L = \{a^m b^n \mid m \neq n\}$.

i. [4] Give a context free grammar for this language. For part marks include a brief explanation of the grammar.

$$\begin{aligned} S &\rightarrow aSb \\ S &\rightarrow aA \\ S &\rightarrow bB \\ A &\rightarrow aA \\ B &\rightarrow bB \\ A &\rightarrow \epsilon \\ B &\rightarrow \epsilon \end{aligned}$$

ii. [4] Give a DPDA for this language. Be sure to state method of acceptance. For part marks include a brief explanation of the DPDA.

Assume S is on the stack:



5. [15 marks] These questions deal with semantic analysis and attribute grammars.

   (a) [2] What is the difference between synthesized and inherited attributes?

   Synthesized attributes flow information up the tree, where the attributes of the LHS depend on the RHS of the production.**[1]** Whereas inherited attributes flow information down and across the tree, where the attributes of a nonterminal depend on all nonterminals to the left of it in the production.

   (b) [3] What two things does an attribute grammar consist of and what is their purpose?

   Attributes**[0.5]**, which store information in the non-terminal nodes of the parse tree**[1]**, and semantic rules**[0.5]**, which manipulate the attributes.**[1]**

   (c) Consider the grammar in Figure 1 on the last page of this final. In C, all fields in a struct must have different names. That is, it is an error to have two fields with the same name in one struct. Suppose you had access to a *Set* data type with the following operations:

   `Set()` returns a new empty set.
   `Insert(S,i)` inserts item i into set S.
   `Contains(S,i)` return *true* if and only if item i is in set S.

   For the grammar in Figure 1 you will be asked to create an attribute grammar that uses the above Set abstract data type to detect duplicate fields.

   i. [3] Although either S-grammar or L-grammar can be used to to detect if duplicate fields, why would an L-grammar be more appropriate in this case?

   L-grammars have inherited attributes**[1]**, which would allow us to propagate information across the tree**[1]**, and issue an error as soon as we encounter the first duplicate name.**[1]**

ii. [7] For the grammar in Figure 1 give an attribute grammar that uses the above Set abstract data type to detect duplicate fields. (Hint: to save time and space use the number of each production instead of rewriting the production.)

[Approximately one mark per production.]

$$
\begin{array}{rcl}
\mathit{Struct} & \rightarrow & `\texttt{struct}'\ \texttt{id}\ `\{'\ \mathit{Field}\ \mathit{Fields}\ `\}' \\
 & & \texttt{if}\ \mathit{Contains}(\mathit{Fields}.S, \mathit{Field}.id)\ \textbf{Error} \\
 & & \mathit{Insert}(\mathit{Fields}.S, \mathit{Field}.id) \\
\mathit{Fields} & \rightarrow & \mathit{Field}\ \mathit{Fields} \\
 & & \texttt{if}\ \mathit{Contains}(\mathit{Fields}.S, \mathit{Field}.id)\ \textbf{Error} \\
 & & \mathit{Insert}(\mathit{Fields}.S, \mathit{Field}.id) \\
\mathit{Fields} & \rightarrow & \epsilon \\
 & & \mathit{Fields}.S = \mathit{Set}() \\
\mathit{Field} & \rightarrow & \mathit{Type}\ \texttt{id}\ `;' \\
 & & \mathit{Field}.id = \texttt{id} \\
\mathit{Field} & \rightarrow & \mathit{Type}\ \texttt{id}\ \mathit{Array}\ `;' \\
 & & \mathit{Field}.id = \texttt{id}
\end{array}
$$

6. [20 marks] These questions deal with naming and binding.

(a) [3] Define the following terms:

Binding :

An association between a name and code or data**[1]**

Scope of a binding :

The region(s) of a program or time interval(s) in the programs execution where the binding is active.**[1]**

Reference environment :

A complete set of active bindings at a point in a program.**[1]**

(b) [2] Why are objects that are allocated from the heap bound at runtime rather than compile time or load time?

Objects allocated from the heap are allocated dynamically, at runtime.**[[]**1] Since the object's location in memory will not be known until it is allocated, it cannot be bound until then, at runtime.**[1]**

(c) [2] What is the difference between lexical and dynamic scoping?

Lexical scoping means that the current binding of a name is the one located in smallest enclosing scope.**[1]** Whereas in dynamic scoping the current binding of a variable is the one most recently encountered in the execution of the program.**[1]**

(d) [2] What is the difference between deep vs shallow binding?

Shallow vs deep binding defines when free variables in a passed function are bound:**[1]** at call time or pass time.**[1]**

(e) [2] What is a closure?

A closure comprises a function**[1]** and the referencing environment at the time of the closure's definition.**[1]**

(f) [5] With the help of a diagram, describe how the reference environment for a closure can be implement so that the closure can be invoked from anywhere in the program, even outside of the scope where it was created.

A reference environment consists of one or more frames**[1]**, where each frame contains the bindings defined in its scope.**[1]** Each frame also refers to its parent frame, corresponding to the parent scope.**[1]** As long as there is one or more references to a frame, the frame is not destroyed.**[1]** A closure has a reference to its frame, and hence to all bindings that were active when the closure was defined.**[1] [and 1 mark for the diagram]**

(g) [4] The C language uses lexical scoping, does not allow nested functions, and allows static local variables. It can be argued that C supports closures. Justify this.

In C pointer to functions are first class objects (can be passed to and returned from functions)**[1]**. Furthermore, since functions cannot be nested,**[1]** functions will always have access to the global and static variables.**[1]** Lastly, functions can save their local bindings by declaring them static as well**[1]**. Thus, C has a limited closure mechanism.

7. [20 marks] These questions deal with control flow and computation abstractions.

(a) [2] What is the difference between an expression and a statement?

An expression yields a value (result) **[1]** whereas a statement is executed to perform a side-effect.**[1]**

(b) [4] Why are short-circuit and non-short-circuit evaluations of boolean expressions considered equivalent in a purely functional language, but are not considered equivalent in an imperative language?

In short-circuit evaluations not all terms are evaluated if their result has no effect on the result of the expression.**[1]** In a purely functional language, with no side-effects, the evaluation of an expression has no side-effects.**[1]** Hence, it does not matter if all terms are evaluated in a boolean expression or not, because only the final result matters.**[1]** In an imperative language, the evaluation of a term can result in a side-effect.**[1]**, thus whether a term is evaluated or not matters.**[1]**

(c) [5] The Fibonacci sequence is defined as $1, 1, 2, 3, 5, 8, \ldots$ where the next number in the sequence is the sum of the previous two. Using pseudocode (or a language of your choice) write a generator that yields the next Fibonacci number in the sequence, starting with 1.

```
def fib():                  [1]
  f1 = 1                    [0,5]
  f2 = 1                    [0,5]
  while True:               [1]
    yield f1                [1]
    temp = f1               [0.5]
    f1 = f2                 [0.5]
    f2 = temp + f1          [0.5]
```

(d) Consider the following function, which computes the maximum of a list of numbers:

```
def max( L ):
  if len( L ) < 2:
    return L[0]
  else:
    m = max( L[1:] )  # call max() with rest of list
    if m < L[0]:
      m = L[0]
    return m
```

i. [1] Why is this **not** a tail-recursive function?

Some operations such as the last if statement being performed after the recursive call.

ii. [4] Rewrite this function (in pseudocode or language of your choice in a tail-recursive fashion. (You may use helper functions if needed.)

```
def mean( L ):
  return sum( 0, L ) / len( L )

def sum( s, L ):
  if len( L ) == 0:
    return s
  else:
    return sum( s + L[0], L[1:] )
```

(e) [2] What happens when an exception is raised? Be specific.

When an exception is raised, the system checks the list of exceptions in the current protected block and if a handler is found invokes the handler.**[1]** If the current block does not handle the exception, the system unrolls the stack to the previous guarded blocks and checks if it can handle the exception.**[1]** This is repeated until a handler is found, or the program terminates due to an unhandled exception.**[1]**

(f) [4] Suppose you were using a language that had continuations but not exceptions. That is, the language had a `call-cc( f )` function that created a continuation c, called function f and passed c as a parameter to f. How could you implement exceptions using continuations? I.e., how would you implement the `guard` and `raise` mechanisms, like you did in your last assignment? (This is a hard question.)

An implementation of `guard` would take the provided block of code that you wish to protect and the handlers, and wrap them in a function, f, that first pushes the handlers on a handler stack, along with the continuation and then executes the protected block of code.**[2]**.
The implementation of `raise` would pop a handler off the handler stack and check if it can handle the exception. If so, it would invoke the handler, and pass the result to the corresponding continuation. Otherwise, it would pop the next handler off the stack and check again, until it finds one that can handle the exception.**[2]**

8. [15 marks] These questions deal with types, data types and memory management.

(a) [2] What is a type system?

A type system is a mechanism for defining types **[1]** and Associating them with operations on objects of this type.**[1]**

(b) [2] Why is type analysis typically done at the semantic analysis stage and not earlier?

Types involve information that cannot be encoded in a CFG.**[1]** It typically requires the compiler to keep track of various identifiers and symbols.**[1]** Types typically work at the level of expressions rather than tokens.**[1]**

(c) [2] Most languages support some form of type coercion and casting. What is the difference between these two operations?

Coercion is performed automatically by the compiler**[1]** to promote a lower precision type to a higher precision type.**[1]**. Whereas casting is an explicit operation performed by the programmer**[1]** to treat a value of one type as a value of another type.**[1]**

(d) [2] Which of the two memory layout schemes for arrays, contiguous layout or row pointer layout, is more memory efficient? Why?

Contiguous layout **[1]** because no memory is needed to store row pointers for multi dimensional arrays. **[1]**

(e) [1] Why are dope vectors not needed if the language uses row-pointer layout for multidimensional arrays?

In row pointer layout each array is 1 dimensional**[1]** and the shape of multidimensional arrays are encoded in the type and size of each 1D array.**[0.5]**

(f) [2] What is the fundamental problem with languages that require the programmer to manage their own memory?

To safely deallocate memory, the programmer must keep track of all references to allocated memory**[1]** and when the memory is no longer in use.**[1]** For large programs that perform many allocations, this can be quite challenging.**[1]**

(g) One problem with the reference counting approach for garbage collection is that circular references cannot be garbage collected, e.g., a linked list where the tail points to the head of the list.

i. [2] Why are circular references a problem for reference counting?

Since reference counting only frees memory when the reference count is 0,**[1]** memory with circular references will never be freed because its reference count will never drop to 0.**[1]**

.

ii. [2] Why does mark and sweep garbage collection not suffer from the same problem?

Mark and sweep simply marks all reachable memory and frees all unreachable memory.**[1]** It does not care how many references there are to a piece of memory, only whether it is reachable.**[1]**

$$\begin{aligned}
\mathit{Type} &\rightarrow \mathit{Primitive} && (1)\\
\mathit{Type} &\rightarrow \mathit{Struct} && (2)\\
\mathit{Struct} &\rightarrow \text{`}\texttt{struct}' \ \texttt{id} \ \text{`}\{' \mathit{Field\ Fields} \ \text{`}\}' && (3)\\
\mathit{Fields} &\rightarrow \mathit{Field\ Fields} && (4)\\
\mathit{Fields} &\rightarrow \epsilon && (5)\\
\mathit{Field} &\rightarrow \mathit{Type} \ \texttt{id} \ \text{`};' && (6)\\
\mathit{Field} &\rightarrow \mathit{Type} \ \texttt{id} \ \mathit{Array} \ \text{`};' && (7)\\
\mathit{Array} &\rightarrow \epsilon && (8)\\
\mathit{Array} &\rightarrow \text{`}[' \ \texttt{int} \ \text{`}]' \ \mathit{Array} && (9)\\
\mathit{Primitive} &\rightarrow \text{`}\texttt{int}' && (10)\\
\mathit{Primitive} &\rightarrow \text{`}\texttt{float}' && (11)\\
\mathit{Primitive} &\rightarrow \text{`}\texttt{char}' && (12)
\end{aligned}$$

Figure 1: A simple grammar for C types, with start symbol Type.