CSE 413 Winter 2002 Final Exam

Question 1. (6 points, 3 points each) Regular expressions.

(a) Describe the set of strings generated by the regular expression

```
(a|b)(a|b)c(x|y)+
```

(b) Describe the set of strings generated by the regular expression

```
p*qp*qp*
```

Question 2. (5 points) In C, integer constants may be written as decimal numbers (normal case), octal (base-8) numbers, or hexadecimal (base-16) numbers. The exact rules are as follows (from Kernighan & Ritchie, *The C Programming Language*):

"An integer constant consisting of a sequence of digits is taken to be octal if it begins with `0` (digit `0`), decimal otherwise. Octal constants do not contain the digits 8 or 9. A sequence of digits preceded by `0x` or `0X` (digit `0`) is taken to be a hexadecimal integer. The hexadecimal digits include `a` or `A` through `f` or `F` with values 10 through 15."

Examples: 1579 0177 0x1a9FE 1000 01000 0X1000, but not 0789, 12af.

Write a regular expression or collection of regular expressions that generate integer constants as described above.

CSE 413 Winter 2002 Final Exam

Question 3. (15 points) Here is a context free grammar for a fragment of English.

```
sentence ::= subject verb object .
subject ::= noun-phrase
object ::= modifier noun-phrase | noun-phrase
noun-phrase ::= article noun | article adjective noun
              | adjective noun | noun
modifier ::= like | as
article ::= a | an | the
adjective ::= fruit | pretty | yellow
noun ::= flies | arrow | banana | time | fruit
verb ::= like | runs | flies
```

(a) (5 points) List the terminal, non-terminal, and start symbols for this grammar.

Terminals:

Non-terminals:

Start symbol:

(b) (5 points) Draw a parse tree for the sentence "fruit flies like a banana" ("fruit flies" is the subject; "like" is the verb).

(continued next page)

Question 3 (cont.)

[grammar repeated to save page flipping]

> *sentence* ::= *subject verb object* **.**
> *subject* ::= *noun-phrase*
> *object* ::= *modifier noun-phrase* | *noun-phrase*
> *noun-phrase* ::= *article noun* | *article adjective noun*
> | *adjective noun* | *noun*
> *modifier* ::= like | as
> *article* ::= a | an | the
> *adjective* ::= fruit | pretty | yellow
> *noun* ::= flies | arrow | banana | time | fruit
> *verb* ::= like | runs | flies

(c) (5 points) Show that this grammar is ambiguous.

Question 4. (12 points) Not really Scheme hacking.

Scheme has a very simple expression syntax. This question gives an informal description of a subset of the legal expressions in Scheme; your job is to write a context free grammar that generates these expressions.

For this problem, an expression is either an atomic expression, a function application, or a special form. Atomic expressions are either integer constants (`0`, `17`, `42`), or symbols (`car`, `cdr`, `cons`, `+`, `*`, `100bottles-of-beer-on-the-wall`, etc.), or the empty list `()`. A function application is a non-empty list of expressions: `(e0)`, `(e0 e1 … en)`, etc. The two special forms you should include are `(if e0 e1 e2)` and `lambda` (for this question a lambda expression consists of the word `lambda` followed by a parenthesized argument list – possibly empty – followed by a single expression).

Give a context free grammar that generates these expressions. For full credit, your grammar must generate only these expressions (i.e., the regular expression `[a-zA-Z0-9+*()]*` does generate these Scheme expressions, but it includes many things that are not legal expressions). However, you do not need to worry about whether a syntactically legal expression is semantically legal: the expression `(1 2)` is syntactically legal, even though it is not a proper function application.

Your grammar should not be ambiguous (this isn't hard – almost any reasonable solution to the problem will be unambiguous). The grammar should contain non-terminals for major constructs like expressions and applications, which would make it relatively easy to construct a recursive descent parser if we wanted (not requested in this question).

You should assume that the non-terminal *symbol* generates all legal symbols and the non-terminal *int* generates all legal integer constants, just as was done in the D grammar for identifiers and integer constants. You do not need to define these two non-terminals further; just use them as needed in your grammar.

Question 5. (18 points) x86 hacking.

Consider the following D main program and function definition:

```
// return smallest power of 2 >= n.
// return 0 if n < 0
int pwr(int n) {
   int ans;
   if (0 > n) {
      return 0;
   }
   ans = 1;
   while (n > ans)
      ans = 2 * ans;
   return ans;
}
```

```
// test pwr function
int main() {
   int in; int out;
   in = get();
   out = put(pwr(in));
   return out;
}
```

(a) (6 points) Draw a picture showing the layouts of the stack frames for methods `pwr` and `main`. This should show the stack layout immediately after the function prologue code has been executed, but just before the first statement of the function body is executed. Be sure to show where registers `ebp` and `esp` point, and the location and offsets from `ebp` of each parameter and local variable.

Question 5. (cont)

(b) (12 points) Translate both functions `pwr` and `main` into x86 assembly language. You do not need to slavishly imitate the code generated by your compiler – straightforward x86 code is fine as long as it uses the registers properly and obeys the x86 conventions used in the compiler project for stack frame layout, function calls, etc. It will help us grade your answer if you include the source code as comments near the corresponding x86 instructions. The D code is repeated here to save some page flipping.

```
// return smallest power of 2 >= n.
// return 0 if n < 0
int pwr (int n) {
   int ans;
   if (0 > n) {
      return 0;
   }
   ans = 1;
   while (n > ans)
      ans = 2 * ans;
   return ans;
}
```

```
// test pwr function
int main() {
   int in; int out;
   in = get();
   out = put(pwr(in));
   return out;
}
```

CSE 413 Winter 2002 Final Exam

Question 6. (14 points) Compiler hacking.

The original FORTRAN language had an interesting conditional statement that tested the value of an integer expression and executed one of three possible statements depending on whether the expression was positive, zero, or negative. For this question, you should write a parser/code generator method for a D compiler to add a similar statement to D.

The new rule in the grammar is:

statement ::= `ifsign ( `*exp*` )` *statement statement statement*

Execution of this statement is carried out by first evaluating the integer expression *exp*. If the value is negative, the first of the three statements is executed and the remaining two are skipped. If the value is zero, the middle statement (only) is executed; if the value is positive, the third statement is executed.

Example: The following code fragment would store –1, 0, or +1 in variable `sign` depending on whether an input value is negative, zero, or positive respectively:

```
n = get();
ifsign (n)
   sign = 0-1;
   { sign = 17; sign = sign-sign; }
   sign = 1;
```

Complete the definition of method `ifsignStmt` on the next page to parse and generate code for this new statement.

Hint: You may find it helps to organize your thoughts if you first sketch out the method needed to parse this new construct, and separately think about the code that needs to be generated, before you write your solution.

Reference information: You should assume that classes `Token`, `CompilerIO`, and `Scanner` are available as described in the compiler project handouts.

```
// description of a single lexical token
class Token {
   public int kind;         // kind of token (see constants below)
   public int val;          // if kind = INT, this is the int
   public String ident;     // if kind = ID, this is the identifier
                            // lexical classes:
   public static int ID     = 0; //   identifier
   public static int INT    = 1; //   integer constant
   public static int LPAREN = 2; //   left parentheses
   public static int RPAREN = 3; //   right parentheses
   public static int LBRACE = 4; //   left curly brace
   public static int RBRACE = 5; //   right curly brace
   ...                           //   etc.
}
```

Question 6. (cont)

```
// parser
class parser {
   CompilerIO cio;            // I/O interface (initialized elsewhere)
   Scanner    scan;           // Scanner (initialized elsewhere)

   Token tok;                 // next unprocessed source program token

   // update tok by advancing to next token
   void nextTok() { tok = scan.nextToken(); }

   // write generated code string s to output file
   void gen(String s) { cio.println(s); }

   // return new unique assembly language label (without trailing ':')
   String newLabel() { ... }

   // compile ifsign ( exp ) statement  statement  statement
   void ifsignStmt() {
      // write your answer below

   }
}
```

Question 7. (7 points) Memory management – the question that was left off the midterm(!)

(a) (4 points) *Reference counting* is a simple algorithm for automatically managing dynamically allocated memory. Each allocated block of storage contains a count of the number of references (pointers) to that block, which is updated as references change. If the count is ever decreased to 0, that block of storage is unused and can be automatically reclaimed.

Although it is simple to understand, reference counting is not used in practice to manage storage for functional and object-oriented languages like Scheme and Java. Give **two** distinct technical reasons why it is not the best choice for managing storage in these languages.

(b) (3 points) In class we looked at two simple garbage collection algorithms: the classic mark-sweep collector, and a copying collector. Although it is harder to implement, a copying collector has some technical advantages over mark-sweep. Describe one of them.

Question 8. (3 points, 1 point each) Java and .NET implementation strategies.

(a) True or false: A compiled Java `.class` file specifies the actual memory layout of objects (instances) of that class.

(b) The .NET compilers generate output files containing Microsoft Intermediate Language (MSIL). How is the MSIL code actually executed? (circle)

1. The MSIL code is interpreted during execution, as was true in the original Java virtual machines.
2. The MSIL code is interpreted at first, but when the interpreter detects that a segment of interpreted code is being executed repeatedly (a loop body, for example), a Just-In-Time (JIT) compiler translates it to native code, which is then executed.
3. The MSIL is translated to native code before anything is executed, and that translated code is what actually runs.

(c) In the .NET framework, *managed code* refers to safe code that, among other things, can be garbage collected automatically. The C++ compiler for .NET includes the full C++ language, including both managed and unmanaged language constructs. Is the C++ `&` operator (compute the address of a variable, as in `&x`) in the managed or unmanaged part of C++.NET?