CS 571
Homework 1 Solution

Due: Sep 28 **100 points**

Important Reminder: As per the course Academic Honesty Statement, cheating of any kind will minimally result in receiving an F letter grade for the entire course.

To be submitted on paper in class.

1. The Ruby Programming Language book describes a ruby integer literal as follows:

   > You write integers using an optional leading sign, an optional base indicator (0 for octal, 0x for hex, or 0b for binary), followed by a string of digits in the appropriate base. Underscore characters are ignored in the digit string.

   Note that the above description is somewhat informal; the letters mentioned in the above description may be either upper or lower case and the underscore characters can only occur **strictly within** the digit string.

   Provide a regular expression which describes ruby integers. *5-points*

   Whitespace and #-comments added in regex for readability:

```
[-+]?(                                      #optional sign
0[0-7_]*[0-7] |                             #octal
0[xX][0-9a-fA-F][0-9a-fA-F_]*[0-9a-fA-F] |  #hex
0[bB][01][01_]*[01] |                       #binary
[1-9][0-9_]*[0-9]                           #decimal
)
```

2. The Java Language Specification describes floating point numbers as follows:

   > A floating-point literal has the following parts: a whole-number part, a decimal or hexadecimal point (represented by an ASCII period character), a fractional part, an exponent, and a type suffix. A floating point number may be written either as a decimal value or as a hexadecimal value. For decimal literals, the exponent, if present, is indicated by the ASCII letter e or E followed by an optionally signed integer. For hexadecimal literals, the exponent is always required and is indicated by the ASCII letter p or P followed by an optionally signed integer.
   >
   > For decimal floating-point literals, at least one digit, in either the whole number or the fraction part, and either a decimal point, an exponent, or a float type suffix are required. All other parts are optional. For hexadecimal floating-point literals, at least one digit is required in either the whole number or fraction part, the exponent is mandatory, and the float type suffix is optional.
   >
   > A floating-point literal is of type float if it is suffixed with an ASCII letter F or f; otherwise its type is double and it can optionally be suffixed with an ASCII letter D or d.

   Note also that a hexadecimal floating point number must start with either `0x` or `0X` and its significand (i.e. the whole number and fraction parts) may contain hexadecimal digits while its exponent is restricted to contain only decimal digits.

   Using standard regex syntax, give a regex for Java floating-point literals. Instead of writing a single regex, you may use a *regex definition* using a CFG-style syntax with a sequence of named regex definitions with later regex definitions referring to earlier regex definitions by name (enclosed within angle-brackets). So notation like:

   ```
   <digit>
     : [0-9]
     ;
   ```

```
  <integer>
   : <digit>+
   ;
```

is acceptable.

Your regex need not make any attempt to restrict the range of the represented numbers.

15-points

```
<dec>
  : [0-9]
  ;
<dec_exp>
  : [eE][-+]?<dec>+
  ;
<suffix>
  : [fFdD]
  ;
<decimal_float>
  : <dec>+(\.<dec>*)<dec_exp>?<suffix>?   #require dec before point + point
  | <dec>*(\.<dec>+)<dec_exp>?<suffix>?   #require dec after point + point
  | <dec>+(\.<dec>*)?<dec_exp><suffix>?   #require dec before point + exp
  | <dec>+(\.<dec>*)?<dec_exp>?<suffix>   #require dec before point + suffix
  ;
hex
  : [0-9a-fA-F]
  ;
<hex_exp>
  : [pP][-+]?<dec>+
<hex_float>
  : 0[xX]<hex>+(\.<hex>*)?<hex_exp><suffix>? #require hex before point + exp
  | 0[xX]<hex>*(\.<hex>+)<hex_exp><suffix>?  #require hex after point + exp
  ;
<float>
 : <decimal_float>
 | <hex_float>
 ;
```

Note the overlapping nature of the alternatives.

3. As mentioned in the **Rationale for the Requirements** for Project 1, the requirements do not allow any kind of whitespace characters within constructed regex's.

   (a) How would you modify the requirements to permit whitespace to be specified?

   (b) How would you change the implementation to permit whitespace within the constructed regex's. *10-points*

   (a) One obvious change to the requirements would use some kind of quotation mechanism. Specifically, the quoting mechanism from C-derived languages could be used: any character (including a whitespace character or a \ character following a \ character would be regarded literally. Hence a ugly regex which matches a space or a `a` could be written as `chars(\ , a)`. Note also that this extension would allow something like `chars(,, a)` to be written as `chars(\,, a)` which is arguably more readable.

   (b) The change required in the implementation would merely require a change in the scanner. Specifically, the code in the scanner which returns a `CHAR` token-kind would be changed: if the character seen is a \, then that character would be skipped and the next character is returned as the value of the `CHAR` lexeme.

4. As mentioned in the **Rationale for the Requirements** for Project 1, the translated regex's may have redundant parentheses. How would you modify your parser to ensure that the translation contains only non-redundant parentheses.

   For example, the project requires the input `((chars(a)) . chars(b))+chars(c)` be translated to `(((([a])[b]))|[c])` where all the parentheses in the output are redundant; it could simply be translated to `[a][b]|[c]`. OTOH, `chars(a) . chars(b) + chars(c)` translates to `([a]([b]|[c]))`, where not all the parentheses are redundant; without redundant parentheses, it would be `[a]([b]|[c])`.

   Hint: consider providing an additional parameter to each parsing function which gives it some idea of the context within which it is being called. *20-points*

Each parsing function knows what operator (if any) it encounters. It should parenthesize it's **sub**-expressions iff the top-level operator of that sub-expression has lower precedence than the operator seen in the parsing function.

We associate a level with each type of regex: 0 for character classes, 1 for closures and 2 for concatenations and 3 for alternations. Parsing functions have an additional parameter giving the level of the parent operator. Additionally, each parsing function return the level of its top-level operator (in addition to the translation).

- Character classes never need to be parenthesized. 0 should be the level returned for character classes.
- A closure regex A`*` needs to be parenthesized as `(`A`)*` iff the incoming level of A is ¿ 1. 1 should be the level returned for a closure regex's.
- The A in a concatenation regex AB should be parenthesized iff the incoming level for A is > 2. The B in a concatenation regex AB should be parenthesized iff the incoming level for B is $>= 2$. The difference in treatment for A and B is because concatenation is left-associative and we would like to have the translation retain the associativitiy specified in the input. 2 should be the level returned for an alternation regex's.
- The A in a alternation regex $A|B$ need never be parenthesized. The B in a alternation regex $A|B$ should be parenthesized iff the incoming level for B is 3. The difference in treatment for A and B is because alternation is left-associative and we would like to have the translation retain the associativity specified in the input. 3 should be the level returned for an alternation regex's.
- A regex which is parenthesized in the input will not be parenthesized in the output unless needed to by its context. Hence its return level is simply its incoming level.

Though no code is required for this question, a Ruby parser for ugly regex's which produces translations without redundant parentheses is given in min_paren_parser.rb. The relevant portions are shown below:

```
#Return value of all parsing functions is a 2-element list
```

```
#containing the translated string and the level.
def uglyRegexp(level)
  term_ret = term(level)
  return regexpRest(*term_ret)
end

def regexpRest(t, level)
  if check(:CHAR, ".")
    match(:CHAR, ".")
    t1 = term(1)
    return regexpRest("#{paren(2, t, level)}#{paren(1, *t1)}", 2)
  else
    return [t, level]
   end
end

def term(level)
  f = factor(level)
  return termRest(*f)
end

def termRest(f, level)
  if check(:CHAR, "+")
    match(:CHAR, "+")
    f1 = factor(2)
    return termRest("#{paren(3, f, level)}|#{paren(2, *f1)}", 3)
  else
    return [f, level]
  end
end

def factor(level)
  if check(:CHAR, "(")
    match(:CHAR, "(")
    e = uglyRegexp(level)
    match(:CHAR, ")")
    return e
  elsif check(:CHAR, "*")
```

```
      match(:CHAR, "*")
      f = factor(1)
      return [f[0], 1]
    else
      match(:CHARS)
      match(:CHAR, "(")
      s = @lookahead.lexeme
      match(:CHAR)
      chars = chars(quote(s))
      match(:CHAR, ")")
      return [ "[#{chars}]", 0]
    end
  end

  def chars(s)
    if check(:CHAR, ",")
      match(:CHAR, ",")
      s1 = @lookahead.lexeme
      match(:CHAR)
      return chars("#{s}#{quote(s1)}")
    else
      return s
    end
  end

  #return parenthesized exp iff exp_level > level.
  def paren(level, exp, exp_level)
    (exp_level > level) ? "(#{exp})" : exp
  end
```

Note that token kinds are denoted using Ruby symbols which are indicated using a leading `:`.

Since all the parsing functions (except `chars()` needs to return 2 values (the translation and the level), the return value for the parsing functions is a 2-element list containing the translation and the level within square brackets. To avoid having to index the list within parsing functions, the 2 values are passed to the parsing functions as separate arguments.

This is achieved using Ruby's prefix `*` *splat* operator; i.e. for list `v`, `fn(v[0], v[1])` is written as `fn(*v)`.

5. A language $\mathcal{L}$ consists of strings containing n `a`'s followed by $n + 4$ `b`'s for $n >= 0$.

   (a) Give a CFG for the language $\mathcal{L}$.

   (b) Give an inductive proof that your CFG describes precisely the language $\mathcal{L}$. *15-points*

   (a) The grammar using terminals `'a'`, `'b'` and non-terminal `S` is shown below:

   ```
   S
   : 'b' 'b' 'b' 'b'     //(1)
   | 'a' S 'b'           //(2)
   ;
   ```

   (b) It is clear from the description of the language that strings in the language will have length $2 \times k + 4$ for $k >= 0$. The proof will be by induction over k.

   Base case: For $k = 0$, the language consists of `b b b b`. This is described by rule (1).

   Assume as induction hypothesis that S generates sentences with $2 \times k + 4$; i.e. S is of the form $\mathtt{a}^k\mathtt{b}^{k+4}$. Hence rule (2) generates $\mathtt{a}\mathtt{a}^k\mathtt{b}^{k+4}\mathtt{b}$ which is $\mathtt{a}^{k+1}\mathtt{b}^{(k+1)+4}$ which is the language for $k + 1$. Hence the grammar generates the correct language.

6. Why is a recursive-descent parser for a grammar amenable to recursive-descent parsing guaranteed to terminate? *5-points*

   The only way a recursive-descent parser will fail to terminate is if there is an infinite loop involving direct or indirect recursion. If a grammar is amenable to recursive-descent parsing then before the parser makes any recursive calls it must have `match`'d one-or-more tokens, consuming those tokens. Since the size of the token-stream is bounded, the parser must terminate.

7. The following C function in program t.c:

```
static void
f(int a, double b, int c)
{
  int t1 = a + b;
  int t3 = t1 + c;
  double t2 = b + c;
  int *p1 = &a;
  printf("address of a is %p\n", &a);
  printf("address of b is %p\n", &b);
  printf("address of c is %p\n", &c);
  printf("address of p1 is %p\n", &p1);
  printf("address of t1 is %p\n", &t1);
  printf("address of t2 is %p\n", &t2);
  printf("address of t3 is %p\n", &t3);
}
```

prints out the addresses of its arguments and local variables. On one run, the output was:

```
address of a is 0x7ffcd3fe829c
address of b is 0x7ffcd3fe8290
address of c is 0x7ffcd3fe8298
address of p1 is 0x7ffcd3fe82b0
address of t1 is 0x7ffcd3fe82a0
address of t2 is 0x7ffcd3fe82a8
address of t3 is 0x7ffcd3fe82a4
```

(a) Show the layout for the stack frame for the above function invocation. Specifically, show the stack frame with the specific addresses occupied by the different variables.

(b) Explain the above layout in terms of what has been mentioned in class about stack-frame layout.

(c) Guess at possible addresses where the return address for the above function invocation may be stored. *15-points*

(a) The layout is shown below:

```
               +----------------+
0x7ffcd3fe8290 |       b        |
0x7ffcd3fe8294 |                |
               +----------------+
0x7ffcd3fe8298 |       c        |
               +----------------+
0x7ffcd3fe829c |       a        |
               +----------------+
0x7ffcd3fe82a0 |       t1       |
               +----------------+
0x7ffcd3fe82a4 |       t3       |
               +----------------+
0x7ffcd3fe82a8 |       t2       |
               +----------------+
0x7ffcd3fe82b0 |       p1       |
0x7ffcd3fe82b4 |                |
               +----------------+
```

Note that there is nothing in the problem indicating the size of the variables; however, from the addresses it seems reasonable to assume that the code was run on a 64-bit machine with integers having size 4 and doubles and pointers having size 8.

(b) This problem did not work out as intended. `:-(`

Assuming that the stack grows towards low memory, based on what was mentioned in class the arguments should have been located at higher addresses and the local variables at lower addresses with some seemingly unused space between the argument area and the local variables. That is not the case in the above trace, as the compiler seems to have optimized the stack layout.

(c) If the problem had worked out as intended, the return address would have been stored in the space between the area uses for the arguments and the area used for the local variables. However, with the given addresses there does not appear to be any place within the stack frame for the return address; presumably it was stored within a register.

8. Discuss the validity of the following statements:

(a) Garbage collection was a new technique first used by the Java programming language.

(b) In a typical programming language, if the name referring to an entity goes "out of scope" and is no longer accessible, then the entity becomes inaccessible and can be destroyed.

(c) Global variables cannot be used in multi-threaded programs.

(d) A scanner must always discard whitespace and comments.

(e) Stack allocation can be used for entities having arbitrary lifetimes.
15-points

(a) Garbage collection has been around since one of the earliest languages: Lisp. Hence the statement is **false**.

(b) This is not necessarily true. For example, when a function uses a reference parameter returns, the reference parameter goes out of scope but the entity referred to by that parameter may still be accessible in the caller and cannot be destroyed. Hence the statement is **false**.

(c) Global variables can be used in multi-threaded programs. They can definitely be used if they are read-only variables. Even if they are read-write variables they can be used with locks to ensure no inconsistencies when used by multiple threads. Hence the statement is **false**.

(d) This statement would be true for scanner's used for compilers. However this would be unacceptable behavior for a scanner used for a syntax-directed editor. Hence the statement is **false**.

(e) Stack allocation can only be used for entities with nested lifetimes typically corresponding to function activations. Hence the statement is **false**.