Computer Science 52

Assignment 7

Due Friday, November 3, 2017, at 5:00 PM

Important note on scheduling Because of the intervening Fall Recess, we have a bit of overlap on Assignments 6 and 7. The former is a relatively short assignment that would normally be due on October 20, but we are giving you a little extra flexibility and making it due a week later. However, Assignment 7 is will be due one week later, on November 3, and it is a a more substantial assignment. We strongly encourage you to finish Assignment 6 early and move on to Assignment 7!

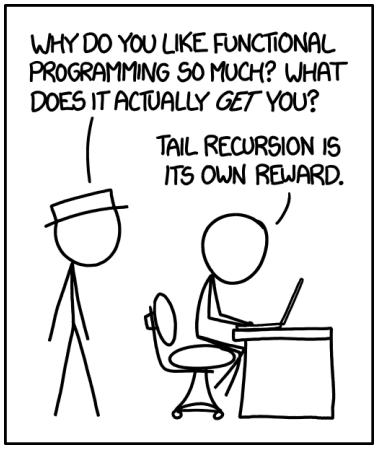
This assignment is about scanning, parsing, and evaluating. It is intended to suggest how programming languages are designed, compiled, and executed. You will write a *recursive descent* parser that takes an arithmetic expression and produces a corresponding CS52 Machine program. Along the way, you will encounter

- the composition operator `o`,
- handling exceptions in SML,
- mutual recursion with the keyword `and`,
- EBNF specifications for grammars and languages, and
- generating low-level code to be executed on a computer.

Reading The SML topics can be found in your favorite reference source. EBNF and the execution model are covered in the first section of the course document *Languages and Models of Computation.*

Preparation We will use the EBNF grammar in Figure 1 to specify standard arithmetic expressions. One purpose of the grammar is to specify which sequences of characters are legal expressions. This grammar is a little odd in that we follow the SML convention and use ~ for unary minus. A consequence is that `~~2;` is not a legal expression, but `3*~2;` is.

Perhaps more important, the grammar specifies how to dissect an expression like $(30 + 88 \,\%\, 2) * 7 - 2$. Notice in the grammar that S depends on E, E on T, T on F, F on P, and finally P on E, completing the circle. This is the "recursive" part of "recursive descent." The EBNF specification will help us to unwind an arithmetic expression and generate code for the CS52 Machine to evaluate it.



https://xkcd.com/1270

S	::= E ;	calculation
E	::= T {(+ | -) T}	expression
T	::= F {(* | / | %) F}	term
F	::= [~] P	factor, possibly with negation
P	::= '(' E ')' | N	positive factor, without negation
N	::= D {D}	number
D	::= 0 | 1 | ... | 9	digit

Figure 1: The EBNF specification for our grammar.

You will write three major functions:

- `scan : char list -> token list`, which converts a list of characters into a list of *tokens;*
- `parse : token list -> syntaxTree`, which converts a list of tokens into a syntax tree; and
- `encode : syntaxTree -> string list`, which generates a sequence of CS52 Machine operations to carry out the computation embodied in a syntax tree.

The whole process of evaluating the expression in a string is captured by the composition of four functions, `encode o parse o scan o explode`, but the functions are independent of one another, and you will work on each one separately. At the end of the assignment, you will have the satisfaction of seeing the CS52 Machine execute the code generated by your functions.

We have provided some "starter" code in the template file `asgt07-template.sml`. It provides the basic data structures that you are to use, some exceptions, and an error-reporting function. Take some time to study the type declarations and the exceptions before you begin to write code. You need not worry about the function `errorln`, except to use it in Problem 4.

We have also provided you with some examples of the values your functions will produce. The file `asgt07-examples.sml` contains five strings and the results of processing each one after the various stages of the assignment. These examples will show you the kinds of results to expect.

Submission As usual, copy the template and check files to your working directory. Rename the template file to`asgt07.sml`. Put your solutions in it and, after verifying your work with the check file, submit it in the usual way.

1. [Lexical scanning, 4 points] The alphabet consists of the ten digits,

the blank space, and the following symbols:

```
~ + - * / % ) ( ;
```

Write a function `scan` that transforms a list of characters into a list of `tokens`. The conversion is one-for-one, except that consecutive sequences of digits are collected into single `Number` tokens.

```
scan : char list -> token list
```

Requirements

- The function `scan` must operate in a *single pass* over the given list. Do not create an intermediate list and then process it to produce the final result. The reason for this restriction is to make our little program look like a real compiler. It is often too inefficient to make several passes over a source file.
- Work directly with characters, one at a time from left to right. Do not use any built-in conversion functions like `Int.fromString`.
- Blank spaces are (almost) never significant; your function will simply discard them and move on. The only exception is when a blank occurs between two digits. A *number* is a sequence of consecutive digits, without intervening blanks. If a blank appears between two digits, `scan` will recognize two distinct `Number` tokens.
- The function `scan` should raise a `LexicalException` with a message "illegal character" if it finds a character outside the lexical alphabet described above.
- If the source string contains a number that is too large to be stored in an ordinary `int`, SML will raise the `Overflow` exception. If that occurs, handle the `Overflow` exception and replace it with a `LexicalException` having a message like "integer too large."

Hints

- Recall the built-in function `Char.isDigit`.
- Review your `charToDigit` implementation from Assignment 3. The situation here is a little simpler, because we are working solely in base 10.
- One challenge here is to process multi-digit numbers while adhering to the single-pass restriction. A good option is to use an auxiliary function that has an accumulator argument and is mutually recursive with `scan`.
  Numbers are written in the normal order with the most significant digit on the left. Horner's rule, which we used on Assignment 2 (look back if you are rusty), can be used to accumulate

the values of the digits. The accumulator argument is necessary because we are reading from left to right.

Examples

- `(scan o explode) "~2 + 49;"` yields

```
[Operation Negate, Number 2, Operation Plus,
 Number 49, Semicolon]
```

- `(scan o explode) "+ %; 23 456"` yields

```
[Operation Plus, Operation Remainder, Semicolon,
 Number 23, Number 456]
```

Even though the string is nonsensical to us, it is valid from `scan`'s point of view. It does not obey the requirements of the grammar, and those problems will be uncovered and diagnosed by the next function, `parse`.

2. [Parsing, 8 points] Write a function called `parse` that takes a list of tokens and produces the corresponding syntax tree.

```
parse : token list -> syntaxTree
```

Each of the operators will give rise to a node in the tree. Nodes corresponding to unary `Negation` operation are `Uninodes`; all the other nodes are `Binodes`.

You will actually end up writing many functions, one for each of the clauses in the EBNF grammar plus a few auxiliary functions. There will be a function for expressions, another function for terms, and so on. These functions will rely on one another and hence will be mutually recursive.

Each clause in the grammar has a specific purpose, and the corresponding function will reflect that clause. A function will consume *some* of the tokens from the front of the list and generate a syntax tree. Most functions will not consume *all* of the tokens, so the functions must return a *pair*, consisting of the syntax tree and the list of unconsumed tokens. The functions that correspond to clauses in the grammar will have type

```
token list -> syntaxTree * token list
```

The caller of such a function, when it receives such a pair, will most likely consume some more tokens and return to *its* caller a pair consisting of a new, larger `syntaxTree` and an even shorter `token list`.

Example Consider the string `"~2 + 15 * 12;"`. It will be converted by `(scan o explode)` into a `token list`.

```
[Operation Negate, Number 2, Operation Plus, Number 15,
 Operation Times, Number 12, Semicolon]
```

The expression is a valid one and will produce a tree that we as humans would draw like this.

```
      Plus
     /    \
Negate    Times
 /        /    \
2        15    12
```

SML will represent the same tree like this.

```
Binode(Uninode(Negate, Leaf 2),
       Plus,
       Binode(Leaf 15, Times, Leaf 12))
```

If `expn` is the function corresponding to the grammar clause for expressions, then

```
(expn o scan o explode) "~2 + 15 * 12;";
```

will return

```
( Binode(Uninode(Negate, Leaf 2),
         Plus,
         Binode(Leaf 15, Times, Leaf 12)),
  [Semicolon] )
```

The call will have consumed all the tokens except the `Semicolon` and produced the tree depicted above.

Finally, the call

```
(parse o scan o explode) "~2 + 15 * 12;";
```

will receive the result from `expn`, see that there is only a `Semicolon` left, and return just the tree. If the list from `expn` were anything other than a list with a single `Semicolon`, the function `parse` will signal an error.

Requirements

- Follow the EBNF syntax precisely. It will guide the implementation of your functions.
- The binary operations must all be left associative. For example, the result of parsing `"1-2-3;"` will be a tree that evaluates to `~4`, not `2`. The grammar is written to ensure left associativity.
- The `parse` function generates only a syntax tree. It does not carry out any arithmetic or simplification.
- Raise `GrammarException` with a descriptive message when an error is encountered. The following list of messages should be sufficient.

- "semicolon expected"
- "right parenthesis expected"
- "number expected"
- "extra tokens" (after the semicolon)

Hints and advice

- Use pattern-matching to avoid complicated clauses.
- Do not think about all the ways that an expression can be wrong; there are too many of them. Instead, concentrate on what is correct and issue messages when the specifications are violated.
- Likewise, resist the temptation to do "too much" in any one function. For each part of the EBNF specification, there will be just one or two functions. Each of those functions should handle *only* the tokens mentioned in the corresponding clause of the EBNF specification. You can see that from the example above.
- Make sure you understand the example `term` function listed in the appendix. It can serve as a good guide for the other functions. (But do not take it too literally. The function for expressions will look very much like `term`, but the other functions will reflect the quite different structure of their corresponding clauses. The important observations are that `term` reflects the corresponding EBNF clause, that it takes a list of tokens, and returns a pair—`syntaxTree` and `token list`.)
- The mutually recursive functions have many interdependencies, but each one has a rather narrow specific function. Each function trusts that the others will do their part of the job. Consequently, it is possible to develop your functions incrementally. Here is an ordering of steps that works for some programmers.
  1. Write the number function.
  2. Write the positive factor function without handling parenthesis. For now, pretend the EBNF clause is just $P ::= N$.
  3. Write the factor function.
  4. Copy the `term` function from the appendix.
  5. Write the expression function.
  6. Go back and fix the positive factor function to include parentheses.
  7. Write parse.

  Other programmers find a more "top down" approach more intuitive.

  1. Write the non-recursive functions, the ones for parse and number.
  2. Write shells for the expression, term, factor, and positive factor functions. At this point, each one will simply pass its argument to the next lower function and return whatever

comes back. If all goes well, you should be able to parse expressions like `"99;"`.
3. Write the expression function. Now you will be able to parse `"99+3-45;"`.
4. Continue, filling in the term, factor, and positive factor functions—one at a time.

Either way you choose, write your functions so that you can test as you go along. *It is very hard to develop the function* `parse` *if you try to write it all at once!*

- SML is stingy about writing out complex results. You are likely to see results like this.

```
Binode (Uninode (#,#), Plus, Binode (#,#,#))
```

To see your results in all their glory, give the following commands to SML:

```
Control.Print.printLength := 200;  (* default is 12 *)
Control.Print.printDepth := 30;    (* default is 5  *)
```

3. [Code generation, 4 points] Write the function `encode` that takes a value of type `syntaxTree` and returns a list of strings, each of which is a line in a valid CS52 Machine assembly program.

```
encode : syntaxTree -> string list
```

For this part of the assignment, you may use `Int.toString` as an aid in constructing the individual strings.

Put the string `"BEGIN"` at the start of your list and the string `"END"` at the end. The reason for the `BEGIN`/`END` pair is that it is difficult in SML to suppress all the usual responses to functions, making it difficult to find the program among the clutter of other messages. Our compromise is to ask the CS52 Machine to ignore everything that is not strictly between the `BEGIN`/`END` markers.

It is possible to avoid the `BEGIN`/`END`, but not worth the trouble.

The stack-based model We will use a stack-based execution model to carry out the computation. Intermediate values will be stored on the stack. For example, consider the very simple expression `"2+4;"`. The resulting syntax tree is

```
Binode (Leaf 2, Plus, Leaf 4)
```

and is visualized as

```
   Plus
  /   \
 2     4
```

The central idea is to traverse the tree recursively. When called upon to carry out an addition—that is, when the function `encode` encounters `Binode(ltree,Plus,rtree)`—it will generate code to evaluate

`ltree`, follow it with code to evaluate `rtree`, and finish it with a sequence "pop, pop, add, push."

In our example, we would push 2 (the left side of the tree; no evaluation is necessary), push 4 (the right side of the tree) and then perform the actual addition by popping the two values off the stack, adding them, and pushing the result back onto the stack.

```
lcw r3 2
psh r3        ; put 2 onto the stack
lcw r3 4
psh r3        ; put 4 onto the stack
pop r3        ; get the first argument off of the stack
pop r2        ; get the second argument off of the stack
add r3 r2 r3  ; add the two numbers
psh r3        ; put sum onto the stack
```

The comments are here for clarity; they will not appear in your result.

We can continue with more complicated strings, like `"2 + 4 - 7;"` which produces a syntax tree

```
(Binode (Binode (Leaf 2, Plus, Leaf 4), Minus, Leaf 7))
```

that is visualized as

```
    Minus
    /   \
  Plus   7
 /   \
2     4
```

The assembly instructions will start with the same code as above, corresponding to the tree for `Plus`. It will be followed by instructions to push the 7, then carry out the final subtraction, and leave the result -1 on the stack.

```
lcw r3 2
psh r3        ; put 2 onto the stack
lcw r3 4
psh r3        ; put 4 onto the stack
pop r2        ; get the second argument off of the stack
pop r3        ; get the first argument off of the stack
add r3 r3 r2  ; add the two numbers
psh r3        ; put the result onto the stack
lcw r3 7
psh r3        ; put 7 onto the stack
pop r2        ; get the second argument off of the stack
pop r3        ; get the first argument off of the stack
sub r3 r3 r2  ; subtract the two numbers
```

```
psh r3        ; put the result onto the stack
```

The examples here are fragments of code to illustrate the execution model. Your function `encode` will produce a list of strings starting with `"BEGIN"` and ending with `"END"`. All the other strings in the list will be CS52 Machine instructions. You will need a prolog to set up the stack, and a postscript to write out the final result of the computation.

For more fun, see the final appendix for an optional optimization opportunity.

Requirements

- You must use the stack-based model.
- Your function must produce a list of instructions that is a completely valid CS52 program, including commands to setup the stack and the `hlt` instruction.
- There must an instruction to print the final answer of the computation.
- Your function should raise a `CodeException` if it encounters an invalid syntax tree, for example, a `Uninode` with an operation that is not `Negate` or a `Binode` with `Negate`. These situations should never arise if your `scan` and `parse` functions are correct, but it is important that each function be complete. We may test your functions individually for correctness.

Hints and reminders

- Remember to establish in `r1` the initial address of the top of the stack and reserve space for the stack at the end of the code.
- To carry out multiplication and division, you must include the `mullib` and `divilib` libraries. These libraries can be found in the Resources section of the course web page. The `divilib` also supports remainder. Take a look at the comments at the top of the library to see how it is done.
- Remember that CS52 Machine opcodes must be preceded by at least one blank space.
- Execution errors, like “division by zero” or “overflow in computation” will be detected upon execution by the CS52 Machine. You need not worry about them as you generate code.

4. [Putting it all together, 3 points] Write a function `compile` that takes a string in our syntax and produces a list of strings that, when interpreted in the CS52 Machine’s assembly language, will carry out the arithmetic and print the result.

```
compile : string -> string list
```

Here, for example, is a typical error-free application of `compile`.

Do not make this too hard. The function `compile` will just be a composition of functions you have already written, with a little additional code to handle exceptions.

```
- compile "2/4-(3+~3)*100;";
val it = ["BEGIN",
          " lcw r1 stack",
          " lcw r3 2",
          " psh r3",
          " lcw r3 4",
          ...,
          "END"] : string list
```

The function `compile` must handle all seven kinds of exceptions mentioned above. It will return either the correct answer—a list of CS52 Machine instructions—or print a descriptive error message. Use `errorln` to print the messages. It is *not* necessary to identify the location within the source string of an error, nor do you have to find any errors other than the first. The examples below show some possible applications of `compile` with appropriate error messages.

```
- compile "6";
Grammar error: semicolon expected

- compile "3+(;";
Grammar error: number expected

- compile "3+(4-6;";
Grammar error: right parenthesis expected

- compile "99999999999999999999;";
Lexical error: integer too large

- compile "3/~~(2-2);";
Grammar error: number expected

- compile "Al Capone";
Lexical error: illegal character

- compile "3+4*6;88;";
Grammar error: extra tokens in input stream
```

Observe that numbers like `1,234,567` are valid in SML but too large for the CS52 Machine's 16-bit words. Our compiler will not catch those values—because the grammar says they are legal—but the CS52 Machine's assembler will complain when it tries to generate native code.

5. [Running the completed program, 1 point] When you are satisfied that your function `compile` is working correctly, obtain the file `asgt07-driver.sml` from the course page. It is set up to read an arithmetic expression from the command line, use your code to compile it, and print the resulting program to be passed to the CS52 Machine.

Be sure that your file is named correctly, `asgt07.sml`, or the driver program will not find it.

Although there is nothing explicit to submit for this part of the assign-

ment, be sure that your code functions properly with the driver file. We will use the driver file to test your code.

Now you can see the full result of your work by typing a command and an expression on two lines in the terminal window.

```
sml asgt07-driver.sml | java -jar cs52-machine.jar -p -f
2/4-(3+~3)*100;
CS52 says > 0
```

The example assumes that you have copies of `cs52-machine.jar`, `mullib.a52`, and `divilib.a52` in your working directory.

The vertical bar is a “pipe” which passes the output of one program to another. In this case, the output of your compiler is passed to the CS52 Machine. Details about the command line and its use of input and output are in an appendix to this document. See the CS52 Machine’s documentation for the meaning of the flags to the Java application.

Appendix: A Sample `term` function

As an example of one of the parsing functions, we discuss `term`. Recall that `term` takes a list of tokens. It consumes some of the tokens, turns them into a tree, and returns a pair consisting of the tree and a list of the remaining (unconsumed) tokens.

As indicated in the syntax specification, the function `term` handles multiplication and division, with most of the real work passed off to `factor`.

$$T ::= F\,\{(* \mid / \mid \%)\,F\}$$

The first call to `factor` returns a tree-list pair. If the first element of the list is a multiplication, division, or remainder token, there is another call to `factor`, which again returns a tree-list pair. The two trees are combined in a `Binode` to obtain new tree. We now have a new tree-list pair, and the process continues until the list no longer begins with one of the three tokens that appear in the term clause of the grammar.

Here is one variant of the `term` function. The idea is that `term` acquires one factor and passes the resulting pair to `termAux`, a function that collects the various factors subject to multiplication, division, and remainder. The first argument to `termAux` acts as an accumulator for the tree. As long as the first of the unused tokens is multiplication, division, or remainder, `termAux` acquires another factor and packages it into a tree with the existing tree. Notice the tail recursive structure of `termAux`.

```
fun term tknList = termAux (factor tknList)

and termAux (tr, (Operation Times)::rest)  =
        let
            val (rtree, rrest) = factor rest;
        in
            termAux (Binode(tr, Times, rtree), rrest)
        end
  | termAux (tr, (Operation Divide)::rest) =
        ... other cases similarly ...
  | termAux (tr, rest)                     = (tr, rest);
```

Appendix: Working on the Command Line

Programs begin with three communication streams for input and output:

- the *standard input* which normally passes the user's keystrokes to the program,
- the *standard output* which normally prints any information in the terminal window, and
- the *standard error* which normally prints error messages to the terminal window.

We say "normally" because it is possible to *redirect* input and output to files or other programs.

If we want the input to come from a file instead of the keyboard, we may redirect the standard input like this:

```
programName < inputFileName
```

Similarly, if we want to write the results of a program to a file, we may redirect the standard output like this:

```
programName > outputFileName
```

It is, of course, possible to combine the two methods and redirect both the standard input and standard output.

Often it is convenient to pass the standard output of one program to the standard input of another. We may do that with a *pipe,* denoted by the vertical bar character:

```
firstProgramName | secondProgramName
```

We saw a pipe in Problem 5 in which we passed the output of a SML session to the CS52 Machine.

```
sml asgt07-driver.sml | java -jar cs52-machine.jar -p -f
```

As you are working on your program, it may be helpful to examine the code it generates or to observe the code as it executes in the visual CS52 Machine simulator. One way to do that is to is to pass the output of your compiler to a that filters out everything but the CS52 Machine code and then place the assembly code in a file.

```
sml asgt07-driver.sml | awk '/^END$/flag=0flag;/^BEGIN$/flag=1' > asmfile.a52
```

(The program `awk` is an old, but powerful and sophisticated, utility that processes text. In this case, it passes only the lines that lie between a `BEGIN/END` pair. As you see, the syntax of `awk` is a little cryptic; we will not try to explain it here.)

Working on the command line may frustrating and confusing at first, but it quickly becomes easy and natural. There are many features that can be used, and they can make your work much more efficient. Spend some time experimenting with the command line.

Appendix: An Optimization Opportunity

If you have followed the proposed model carefully, you will see your program generating lots of useless sequences like `psh r3`, `pop r3`.

If you like, you may modify the stack-based model so that your program generates more efficient CS52 Machine code. This part of the assignment is entirely optional. If you decide to try it, we suggest that you first complete the assignment using the execution model presented above and in class, and then make the changes.

The modification is a result of viewing the evaluation stack a little differently. You can avoid repetitive `psh`/`pop` sequences by using register `r3` as the top of the evaluation stack and the CS52 Machine's stack as the rest of the stack.

Be conservative. Our suggestion involves only a handful of changes to the instruction list produced by the function `encode`.