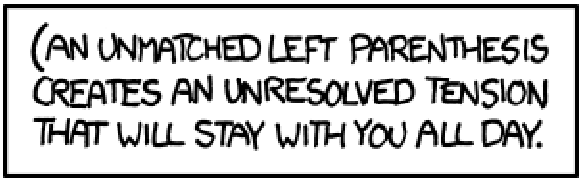
CS52 - Assignment 8

Due Friday 4/15 at 5:00pm


(AN UNMATCHED LEFT PARENTHESIS CREATES AN UNRESOLVED TENSION THAT WILL STAY WITH YOU ALL DAY.

`https://xkcd.com/859/`

This assignment is about scanning, parsing, and evaluating. It is a sneak peak into how programming languages are designed, compiled, and executed. Put your solutions in a file named `assign8.sml` and submit in the usual way.

High-level Overview

Our goal is to be able to write SML code that can process strings representing bitwise logical operations on bytes. It is very similar to the familiar expressions from ordinary arithmetic. There are five bitwise operations:

`!`	not
`&`	and
`|`	or
`^`	xor
`->`	implies

with the following single-bit truth tables:

	not
`a`	`!a`
0	1
1	0

		and	or	xor	implies
`a`	`b`	`a&b`	`a|b`	`a^b`	`a->b`
0	0	0	0	0	1
0	1	0	1	1	1
1	0	0	1	1	0
1	1	1	1	0	1

The bitwise operations operate *in parallel* on the 8 bits in a byte, so that

`!01010101 & 00001111` evaluates to `00001010`.

Language specification

We will use a standard EBNF specification for boolean expressions:

$$\begin{array}{ll}
P ::= E\,\texttt{;} & \text{program} \\
E ::= D\,[\,\texttt{->}\,E] & \text{expression} \\
D ::= C\,((\text{`}\texttt{|}\text{'}|\texttt{\^{}})\,C)^* & \text{disjunction} \\
C ::= L\,(\texttt{\&}\,L)^* & \text{conjunction} \\
L\ ::= \texttt{!}\,L\,|\,\text{`('}E\text{`)'}\,|\,B & \text{literal} \\
B ::= (0|1)^+ & \text{byte constant}
\end{array}$$

The specification has been carefully constructed to have an unambiguous order of operations. Just as in ordinary arithmetic, where multiplication is done before addition or subtraction, our grammar specifies that

- `!` is done first,
- followed by `&`,
- then `|` and `^` in any mixture, and finally
- `->` is last.

We can, of course, override these conventions with parentheses. When we have operations at the same level, `&`, `|`, and `^` associate to the left, while `->` associates to the right. Therefore,

`a	b & c`	is interpreted as	`a	(b & c)`,
`a	b ^ c`	is interpreted as	`(a	b) ^ c`,
`a -> b -> c`	is interpreted as	`a -> (b -> c)`, and
`a	b -> c`	is interpreted as	`(a	b) -> c`.

We will think of bytes using the unsigned integer representation of the bits, taking values from 0 to 255 inclusive. You must take care to ensure that your `int` values do not lie outside that range.

Your program

To accomplish this, you will write write three major functions:

- A lexical scanning function

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

  that converts a list of characters into a list of *tokens,*

- a parsing function

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

  that converts a list of tokens into a *syntax* tree, and

- an evaluation function

  ```
  eval : syntaxTree -> int
  ```

  that carries out the computation embodied in a syntax tree.

The whole process of evaluating the expression from an input string is then captured by the composition of four functions,

```
eval o parse o scan o explode,
```

but the functions are independent of one another, and you will work on each one separately.

Starter

I have included a starter file that has some helper functions as well as some datatype definitions at:

```
http://www.cs.pomona.edu/~dkauchak/classes/cs52/assignments/assign8/assign8-starter.sml
```

The contents of the starter are organized into three main sections. *Read through all of the sections and make sure you understand what all of the declarations are.* You will be using everything in this file at some point in working on this assignment.

- **helper functions**

  There are a number of helper functions available for performing bitwise operations on bytes. Note that many of the operations can raise exceptions.

- **type declarations**

  We have three main types that we will be playing with

  1. `operation`: the 5 different boolean operations we can perform
  2. `token`: the input strings are just a string of characters. A token has some semantic meaning and will represent either a byte, an operation, a parentheses or a semicolon. Notice that many of these tokens are just represented by a single character, though some are not, like bytes and the implies operator.
  3. `syntaxTree`: the syntax tree will represent the evaluation order of the operations and the arguments.

- **exceptions**: there are three general types of exceptions that can occur during evaluation that reflect the different stages of processing that happen to a string. When one of your functions encounters an issues, it should raise an exception with a descriptive message. In a top-level function, you will introduce code to handle the exceptions that are raised.

The `fun`

1. [**4 points**] From characters to tokens

   The first step is to from from a list of individual characters to a list of tokens. Write a function

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

   that converts a list of characters into a list of *tokens*. For example

   ```
   - scan (explode "!01010101 -> 00001111");
   val it = [Oper Not,Byte 85,Oper Implies,Byte 15] : token list
   ```

   The input alphabet consists of the following symbols:

   ```
   !  &  |  ^  -  >  )  (  ;  0  1
   ```

   together with the blank space. Blanks are insignificant and ignored, except in the middle of a byte constant or between the two characters of the `->` connective.

   There is mostly a one-to-one mapping between alphabet characters and tokens except: 1) sequences of bits need to be collected into a single `Byte` token and 2) the two characters "`->`" are combined into the `Oper Implies` token. Byte constants are specified by non-empty strings of `0`'s and `1`'s. The strings may be longer or shorter than 8 bits, but the unsigned values must be within the correct range for bytes.

   Requirements

   - The function `scan` must operate in a *single pass.* Do not create an intermediate list and then process it to produce the final result. Work directly with characters and do not use any built-in conversion functions.
   - The function should raise a `LexicalException` with a message "illegal character" if it finds a character outside the lexical alphabet described above
   - The function should also raise a `LexicalException` with a message like "constant too large" if a byte constant falls outside the designated range for bytes.

   Hints/Advice

   - Because you don't know how long the bit sequences will be, a good way to handle this is to write a helper function that processes the bytes. Because of the requirement to operate in a single pass, this function will likely need to be mutually recursive with the `scan` function.
   - The bytes are binary numbers written in the normal order with the most significant digit to the left. To turn them into an int in a single pass, you can still use a Horner's rule-like approach. Since it's in the normal order, you'll need to use a second argument that accumulates the result, though. It's very hard to do this other ways while keeping the requirement of a single pass!

2. [**8 points**] Parsing

   Write a function called `parse` that takes a list of tokens and produces the syntax tree those tokens represent:

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

Each of the operators will give rise to a node in the tree. Nodes corresponding to unary `Not` operation are `Uninode`s; all the other nodes are `Binode`s.

The key idea is that you should write a function to handle each of the component (e.g. expression, disjunction, etc.). These functions will generally be mutually exclusive since they will often call each other. Each of these functions will consume *some* tokens corresponding to the expression and will generate a syntax tree from the tokens that it consumes. Most of these functions will not consume *all* of the tokens in the input list and so the functions should also return the those tokens that did not get consumed. These functions will each have the type:

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

that is, they take in a list of tokens and produce a tuple containing the syntax tree (representing the tree for the part that did get consumed) and the remaining tokens.

An example

Consider the expression

```
   !1 | 011 & 101
```

This will get turned into the following token list:

```
[Oper Not,Byte 1,Oper Or,Byte 3,Oper And,Byte 5]
```

If we try and process this as an expression, we should get a valid syntax tree, since it is a valid expression, e.g. a call like:

```
- expression (scan (explode "!1 | 011 & 101"));
val it = (Binode (Uninode (Not,Leaf 1),Or,Binode (Leaf 3,And,Leaf 5)),[])
```

Remember this is a tuple with the syntax tree and the remaining set of tokens. The syntax tree looks something like:

```
         Or
       /    \
    Not      And
    /      /    \
   1      3      5
```

and the remaining tokens is [] since the entire set of tokens is in fact an expression. (Note, the actual output from SML is actually

```
val it = (Binode (Uninode (#,#),Or,Binode (#,#,#)),[])
```

since it doesn't print out all of the levels of the tree.)

We could also try and parse these tokens with some of the other EBNF components, e.g.:

```
- literal (scan (explode "!1 | 011 & 101"));
val it = (Uninode (Not,Leaf 1),[Oper Or,Byte 3,Oper And,Byte 5])
```

If we try and process this token list as a literal, we get back the tree containing "not 1" and then the rest of the tokens in the list that were not part of that literal.

Requirements

You should raise `GrammarException` with a descriptive message when an error is encountered. The following list of messages should be sufficient.

- "semicolon expected"
- "right parenthesis expected"
- "byte constant expected"
- "extra tokens" (after the semicolon)
- "byte constant too large"

The last error will not occur when the input to `parse` is the result of a (correctly written) `scan` function. Nevertheless, you should check the byte values because `parse` cannot be sure where its arguments come from.

Hints/advice

- 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 part of the EBNF specification. You can see that from the example above.
- Although there is a lot of interdependence between these functions (hence why they're mutually recursive), try hard to develop incrementally. For example, you can write the `byte` function first and test that it works. Then, you can write parts of the `literal` function, e.g. ignore handling parentheses for now. Then, write the `conjunction` function. If you do this you can test as you go!
- To help you get a feeling for this, I have provided an example `disjunction` function in the appendix of this assignment. Make sure that you understand how this function works. Note that this function won't work initially, since it relies on having `conjunction` defined.

3. [**4 points**] Almost there!

Write the function `eval` that takes as input a syntax tree and returns the correct byte value of the calculation expressed by the tree.

```
eval : syntaxTree -> int
```

Requirements

There are two possible errors: “byte constant too large” and “malformed tree.” The latter occurs when the operation in a `Uninode` is something other than `Not` or when the operation in a `Binode` is `Not`. Neither of these errors will occur when the parse tree is the result of the functions `scan` and `parse`, but we may test your individual functions for correctness.

Hints/Advice

- The syntax tree already encodes all of the order of operations, etc., so this function should be fairly straightforward.
- Don’t overthink this one. Trees are naturally recursive structures and your recursion should mirror that.
- Remember, only operations will be the nodes in the trees.
- We have provided helper functions to do all of the bitwise operations on bytes. Use them!

4. [**3 points**] And the answer is...

Write a function `calculate` that takes a string in our syntax and produces another *string* that contains the result of the calculation.

```
calculate : string -> string
```

Requirements The function `calculate` must handle all nine kinds of exceptions mentioned above. It will return either the correct answer or a message about what went wrong. It is *not* necessary to identify the location within the string of an error, nor do you have to find any errors other than the first.

Hint/Advice

- `o`
- The supplied function `toByteString` will be useful here.
- Do not overthink this function. Let the previous functions do the work and simply relay their result (and exceptions) appropriately.

Sample output

The transcript below shows some possible applications of `calculate`. Your program should give the same results, although the exact text of the error messages need not be identical. Keep in mind that these tests are not exhaustive.

```
calculate "!01011;";
val it = "11110100" : string

calculate "0101 & ! 0100 ^ 0011;";
val it = "00000010" : string

calculate "10101010";
val it = "Grammar error: semicolon expected" : string

calculate "0 & (;";
val it = "Grammar error: byte constant expected" : string

calculate "0001 | (1100 & 1010;";
val it = "Grammar error: right parenthesis expected" : string

calculate "110000001;";
val it = "Lexical error: byte constant too large" : string

calculate "0011 | 11 00;";
val it = "Grammar error: semicolon expected" : string

calculate "0011 -  > 1100;";
val it = "Lexical error: illegal character" : string

calculate "I did it!";
val it = "Lexical error: illegal character" : string

calculate "01011100 -> 10100011;101010;";
val it = "Grammar error: extra tokens in input stream" : string
```

When you're done

Double check the following things:

- Make sure your code runs without any errors (i.e. `use "assign8.sml"` does not execute an error). If you didn't finish a function and it gives an error, just comment it out and leave a note.
- Make sure that your functions match the specifications *exactly*, i.e. the names should be exactly as written (including casing) and make sure your function takes the appropriate number of parameters and is curried/uncurried appropriately.
- Make sure you have used proper style and formatting. See the course readings for more information on this. Be informative and consistent with your formatting!

- Make sure you've properly commented your code. You should include:
  - A comment header at the top of the file with your name, the date, the assignment number, etc.
  - Each problem should be delimited by comment stating the problem number.
  - Each function should have a comment above it explaining what the function does.
  - Complicating or unusual lines in functions should also be commented.

  Don't go overboard with commenting, but do be conscientious about it.

When you're ready to submit, upload your assignment via the online submission mechanism. You may submit as many times as you'd like up until the deadline. We will only grade the most recent submission.

Grading

`scan`	4
`parse`	8
`eval`	4
`calculate`	3
comments/style	3
Total	22

Appendix on the `disjunction` function

As an example of one of the parsing functions, we discuss `disjunction`. Recall that `disjunction` 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 `disjunction` handles `Or` and `Xor` with most of the real work passed off to `conjunction`.

$$D ::= C\,((\text{‘}\texttt{|}\text{’}|\hat{\ })\,C)^*$$

The first call to `conjunction` returns a tree-list pair. If the first element of the resulting list is a `Or` or `Xor` token, there is another call to `conjunction`, which again returns a tree-list pair. The two trees are combined into 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 a `Or` or `Xor` token.

Here is one variant of the `disjunction` function. The idea is that it acquires one factor and passes the resulting pair to `collectConj`. The first component of the argument to `collectConj` acts as an accumulator for the tree. As long as the first of the unused tokens is multiplication or division, `collectFact` acquires another factor and packages it into a tree with the existing tree. Note the tail recursive structure of `collectConj`.

```
fun disjunction toklst = collectConj (conjunction toklst)

and collectConj (tree, (Oper Or)::rest) =
        let
            val (rtree, rrest) = conjunction rest;
        in
            collectConj (Binode (tree, Or, rtree), rrest)
        end

  | collectConj (tree, (Oper Xor)::rest) =
         ... similar ...

  | collectConj (tree, rest) = (tree, rest);
```

Be careful not to over-generalize from this example. Most of the other parts of the EBNF specification have different structures and require different analyses.