1 Parsing

We want to convert strings like "(1+2+3)*4" into a data structure

\[
\text{Times (Plus (Val 1,Plus (Val 2,Val 3)),Val 4)}
\]

A function that performs this task is called a parser. In ML, we can represent
a parser as a function with signature

\[
\text{Token list} \to (\text{Token list} \times \text{a}) \text{ list}
\]

So (informally) a parser takes a stream of tokens and produces a list of pairs,
with first element the remainder of the stream and the second the structure for
what has been parsed so far. It’s a list because there may be many ways of
parsing the start of the stream (for example, English can be ambiguous). In our
arithmetic example,

\[
\text{datatype Expr} = \text{Val of int} \mid \text{Plus of Expr} \times \text{Expr} \mid \text{Minus of Expr} \times \text{Expr} \mid \\
\text{Times of Expr} \times \text{Expr} \mid \text{Divide of Expr} \times \text{Expr};
\]

\[
\text{datatype Token} = \text{tPlus} \mid \text{tMinus} \mid \text{tMultiply} \mid \text{tDivision} \mid \text{tValue of int} \mid \text{tLBracket} \mid \text{tRBracket};
\]

and we can convert a string into a list of Tokens like this (you need to define
the \text{getWhile} function):

\[
\text{fun string2tokens s =}
\]

\[
\text{let s2t [] = []}
\]

\[
\mid s2t (c::cs) = \text{if Char.isDigit c then}
\]

\[
\text{let val (ns, cs') = getWhile Char.isDigit cs}
\]

\[
\text{val (SOME v) = Int.fromString(implode (c::ns)) in}
\]

\[
\text{(tVal v) :: (s2t cs')}
\]

\[
\text{end}
\]

\[
\text{else case c of}
\]

\[
\#	ext{"("} => \text{tLBracket :: (s2t cs) |}
\]

\[
\#	ext{"\)"} => \text{tRBracket :: (s2t cs) |}
\]

\[
\#	ext{"\*"} => \text{tMultiply :: (s2t cs) |}
\]

\[
\#	ext{"/"} => \text{tDivision :: (s2t cs) |}
\]

\[
\#	ext{"-"} => \text{tMinus :: (s2t cs) |}
\]

\[
\#	ext{"+"} => \text{tPlus :: (s2t cs) |}
\]

\[
\_ => (s2t cs) \text{ in}
\]

\[
\text{s2t (explode s)}
\]

end;

So that \text{string2tokens "(1+2+3)*4"} evaluates to

\[
[t\text{LBracket},t\text{Val 1},t\text{Plus},t\text{Val 2},t\text{Plus},t\text{Val 3},t\text{RBracket},t\text{Multiply},t\text{Val 4}]
\]

Let us define a parser that parses a single token from the start of a token stream:

\[
\text{fun parseTok t [] = []}
\]

\[
\mid \text{parseTok t (t':::tl) = if (t' = t) then [(t1,t)] else []};
\]

page 1 of 3
For example, \((\text{parseTok tPlus}) \ [(\text{tPlus,tRBra},\text{tMinus})]\) evaluates to
\[\{(\text{tRBra, tMinus}), \text{tPlus}\}\]
The stream has been successfully parsed and the rest of the token list can be
passed to another parser, and the results combined in some way. To this end,
we can define a new parser from a parser \(p\), and, for each structure \(a\) parsed
using \(p\), another parser \(f \ a\).

\[
\text{infixr } >>=; \\
\text{fun } p >>= f = \text{fn } tl => \text{List.concat (map (fn (tl',a) => (f a) tl') (p tl))};
\]

Let's informally describe this definition in words. \(p \ tl\) is the result of the first
parser. It is a list of all the different ways that the start of the stream can be
parsed. Each result generates a new parser \(f \ a\), which itself generates a list
of ways of parsing the stream beginning where \(p\) ended parsing. We are left
with a list of lists, which is concatenated into a single list, which is the result
of the new parser. For convenience, we define a parser \(\text{return } a\) for every data
structure \(a\).

\[
\text{fun } \text{return } a = \text{fn } tl => \{(tl,a)\};
\]
which takes an input stream, does nothing with it and returns the given data
structure. We can now succinctly define the combination of two parsers; one
applied after the other:

\[
\text{infixr } \text{pThen;} \\
\text{fun } p1 \text{ pThen } p2 = p1 >>= (\text{fn } a => (p2 >>= (\text{fn } b => \text{return } (a,b)))));
\]

For an example, let’s define a parser that parses a value token from the start of
a stream, returning the integer value:

\[
\text{fun } \text{parseVal } [] = [] \\
| \text{parseVal } (t'::tl) = \text{case } t' \text{ of } tVal n => \{(tl, Val n)\} | _ => [];
\]

Then

\[
\text{parseVal } \text{pThen } (\text{parseTok tPlus}) \text{ pThen } \text{parseVal};
\]
is a parser that parses strings like "1+2" so

\[
(\text{parseVal } \text{pThen } (\text{parseTok tPlus}) \text{ pThen } \text{parseVal}) \text{ (string2tokens } "1+2")\]
evaluates to \[\{(\[], \text{(Val 1, (tPlus, Val 2)))}\}\]. Another important way of com-
bining two parsers is the ‘or’ combinator, which is how I say that I can either
parse a string using the first parser or the second. Its definition is the simple
joining of the lists of possible parsings:

\[
\text{infixr } \text{pOr;} \\
\text{fun } p1 \text{ pOr } p2 = \text{fn } tl => (p1 \ tl) \ @ (p2 \ tl);
\]
Using these functions and a grammar of arithmetic we can define the arithmetic parser. Informally, a grammar is a set of rules for generating strings by recursively replacing substrings by strings (e.g. see Syntactic Structures by Chomsky). At the most general level, a grammar is basically a Turing machine. A special class of grammars, called context free, can be written down in BNF notation. The grammar of arithmetic, and the grammars of most programming languages are context free.

\[
\begin{align*}
{\text{expr}} & ::= \{\text{term}\} '+' \{\text{expr}\} \mid \{\text{term}\} '-' \{\text{expr}\} \mid \{\text{term}\} \\
{\text{term}} & ::= \{\text{factor}\} '*' \{\text{term}\} \mid \{\text{factor}\} '/' \{\text{term}\} \mid \{\text{factor}\} \\
{\text{factor}} & ::= '(' \{\text{expr}\} ')' \mid \text{number}
\end{align*}
\]

For our purposes, we can replace \mid with \text{pOr} and the spaces with \text{pThen}. We define by mutual recursion (fill in the blanks):

\[
\begin{align*}
\text{fun tok2op } t & = \text{case } t \text{ of } t\text{Times} => \text{Times} \mid t\text{Divide} => \text{Divide} \mid t\text{Minus} => \text{Minus} \mid t\text{Plus} => \text{Plus}; \\
\text{fun parseExpr } tl & = \left(\left((\text{parseTerm } \text{pThen } ((\text{parseTok } t\text{Plus}) \text{pOr } (\text{parseTok } t\text{Minus})) \text{pThen } \text{parseExpr}) \right.\right. \\
& \left. \left. >>= (\text{fn } (t1,(t2)) \Rightarrow \text{return } ((\text{tok2op } t1) \langle t1, t2 \rangle))\right) \right. \text{pOr } \text{parseTerm} tl \\
\text{and parseTerm } tl & = \ldots \\
\text{and parseFactor } tl & = \ldots
\end{align*}
\]

Then

\[
\begin{align*}
\text{fun fst } (x,\_ ) & = x; \\
\text{fun snd } (\_,y) & = y; \\
\text{fun isEmpty } [] & = \text{true} \mid \text{isEmpty } \_ = \text{false}; \\
\text{fun parse } s & = \text{map } \text{snd} \left(\text{List.filter } (\text{isEmpty } \circ \text{fst}) \left(\text{parseExpr } \langle\text{string2tokens } s\rangle\right)\right);
\end{align*}
\]

and you can try it out on the example at the beginning, or whatever you want. Incidentally, parsers are examples of monads, which are data structures for which >>\text{=} and \text{return} have been defined (they also need to satisfy certain rules).

2 Task

\textbf{task }∞) In the TV game show Countdown\footnote{\url{http://en.wikipedia.org/wiki/Countdown_%28game_show%29}} contestants are asked to find an arithmetic expression using a given set of numbers, each appearing only once in the expression, which evaluates to a given number. For example, if contestants are asked how to make 953 from \{100, 50, 3, 4, 6\} an answer is ((((100/4) − 6) * 50) + 3), or as an \texttt{Expr} (you can use parse):

\begin{align*}
\text{Plus } & \langle\text{Minus } (\text{Divide } \langle\text{Val } 100,\text{Val } 4\rangle,\text{Val } 6)\rangle,\text{Val } 50,\text{Val } 3\rangle
\end{align*}

Write a program that finds a solution to this problem.