Wellesley College ◊ CS301 Compiler Design ◊ October 4, 2000
Handout #07

PROBLEM SET 3

Due on Friday, October 13

READING

- Appel, Sections 3.1-3.2 (Recursive Descent Parsing)
- Fokker's article "Functional Parsers"
- Hutton and Meijer's article "Monadic Parsing in Haskell"

OVERVIEW

The purpose of this problem set is to help you gain familiarity with the practice of monadic parsing. You will using the monadic parsing combinators presented in class to construct a parser for Kitty.

This assignment has one problem worth a total of 150 points. The problem is organized into several parts. The result of all the parts will be a single file `Parser.sml` implementing the Kitty parser.

COLLABORATION DETAILS

You should break up into three teams of two members each, subject to the following constraints:

- You should not work with the same partner as on PS1 and PS2.
- Kate and Holly cannot work together on this assignment (since both have seen monadic parsers before, and we should "spread around" their expertise.

SUBMISSION DETAILS

Each team should turn in a single hardcopy submission packet for all problems by slipping it under Lyn's office door by 5pm on the due date. The packet should include (1) your final version of the file `Parser.sml`; (2) your test files for parsing; (3) transcripts showing how your parser works on your test cases; and (5) any auxiliary files you created. Your hardcopy submission packet should also include a header sheet for each team member (see the end of this assignment for the header sheet), and should indicate where the softcopy submission can be found.

Your softcopy submission should consist of your local versions of the `kitty/parser` and `kitty/test` program directories described below.

PROBLEM 1 [150]

Overview

In this problem you will implement and test a monadic parser for Kitty. Your parser should having the following behavior:

- For any syntactically legal Kitty program, it should return an abstract syntax tree for that program.
- For any illegal Kitty program, the parser should reject the program. Ideally, the parser should report why the program is illegal. However, sophisticated error reporting is beyond the scope of this parser; instead, we will use crude indications of the source of parsing errors.

The specification of Kitty syntax can be found in the Kitty Reference Manual (Handout #03). In particular, pay attention to the grammar specification in Figure 1 on page 3 of this manual. *Note:* the sample programs in previous versions of the manual contained several bugs, which have been corrected in the most recently posted version. In particular: (1) the **`var`** keyword for variable declarations was missing in previous versions and has been added; and (2) missing semi-colons have been added and extraneous semi-colons have been removed.

What You Need

Start the problem by making copying the following directories and files to your local file system. You should preserve the relative positions of directories when you make your local copies.

- Make a local copy of the directory `~cs301/download/kitty/parser.`
- Make a local copy of the directory `~cs301/download/kitty/utils.`
- Make a local copy of the directory `~cs301/download/kitty/test.` (The test files in this directory are corrected versions of previous test files that supersede the previous versions.)
- Copy the most recent versions of `Scanner.sml` and `Token.sml` from `~cs301/download/kitty/scanner` to your local copy of this directory.

It is also assumed that:

- Your local version of the `~cs301/download/kitty` directory contains `AST.sml` from Problem Set 1, as well as your working evaluator from that problem set.
- Your local version of the `~cs301/download/kitty/scanner` directory contains your working Kitty lexical analyzer (`Kitty.lex.sml`) from Problem Set 2.

Documentation

Your task is to flesh out the skeleton file `parser/Parser.sml` so that it implements a Kitty parser. The skeleton file contains a number of predefined functions that will simplify your task; these are documented in Appendix A.

You will be using the ML monadic parsing library presented in class to implement your parser. Documentation of this library appears in Appendix B. The library is implemented in the file `parser/MonadicParser.sml`.

Compiling and Testing

You can compile your Kitty parser by executing the following in the SML interpreter.

```
CM.make’(“Parser.cm”);
```

As usual, you may want to name this command via a small function name to avoid retyping the whole command. E.g.:

```
fun p() = CM.make’(“Parser.cm”);
```

Before testing your parser, you will probably want to increase ML’s default print depth and length as follows:

```
Compiler.Control.Print.printDepth := 1000;(* or some other large number *)
Compiler.Control.Print.printLength := 1000;
```

You can test your Kitty parser via one of the following two functions within the Parser structure:

`val parseString : string -> AST.exp`
Returns the abstract syntax tree of the expression corresponding to all the tokens in the given string. Raises an exception if the tokens cannot be parsed into an expression, giving an indication of why this is so.

`val parseFile : string -> AST.exp`
Like `parseString`, but parses the tokens in the contents of the file named by the given string.

Both of the above functions will use the lexical analyzer that is in `scanner/Kitty.lex.sml`, which presumably is your working scanner from Problem Set 2. Please contact me if your PS2 scanner is not working, as it is necessary to have a working scanner in order to test your parser.

Below are a few examples of the above testing functions in action:

```
- Parser.parseString "2 * 3 * 4";
val it = BinApp (Mul,BinApp (Mul,IntLit 2,IntLit 3),IntLit 4) : AST.exp

- Parser.parseString "2 + 3 * 4";
val it = BinApp (Add,IntLit 2,BinApp (Mul,IntLit 3,IntLit 4)) : AST.exp

- Parser.parseString
  "1 + 2 + 3 < 4 * 5 * 6 & 7 - 8 - 9 = 10 % 4 | 11 / 2 <> 12 & 13 >= 14";
val it =
  Let
    ([Decl
        ("or1",
         If
           (BinApp
              (Lt,BinApp (Add,BinApp (Add,IntLit 1,IntLit 2),IntLit 3),
               BinApp (Mul,BinApp (Mul,IntLit 4,IntLit 5),IntLit 6)),
            BinApp
              (Eq,BinApp (Sub,BinApp (Sub,IntLit 7,IntLit 8),IntLit 9),
               BinApp (Mod,IntLit 10,IntLit 4)),Const False))],
      If
        (VarRef "or1",VarRef "or1",
         If
           (BinApp (Neq,BinApp (Div,IntLit 11,IntLit 2),IntLit 12),
            BinApp (Geq,IntLit 13,IntLit 14),Const False))) : AST.exp
```

```
- Parser.parseString "let var c := 0 in (c := c + 1; c) end";
val it =
  Let
    ([Decl ("c",IntLit 0)],
     Seq [Assign ("c",BinApp (Add,VarRef "c",IntLit 1)),
          VarRef "c"]) : AST.exp


(* ../test/count.kty contains the following Kitty program:

   /* Count the number of characters in the input stream */
   let var count := 0;
       var c := readc()
   in while c >= 0 do
        (count := count + 1;
         c := readc());
      writei(count)
   end
 *)

- Parser.parseFile "../test/count.kty";
val it =
  Let
    ([Decl ("count",IntLit 0),Decl ("c",NullApp ReadC)],
     Seq
       [While
          (BinApp (Geq,VarRef "c",IntLit 0),
           Seq
             [Assign ("count",BinApp (Add,VarRef "count",IntLit 1)),
              Assign ("c",NullApp ReadC)]),UnApp (WriteI,VarRef "count")])
  : AST.exp

(* ../test/uppercase.kty contains the following Kitty program:

   /* Copy input chars to output chars, capitalizing all letters */
   let var c := readc()
    in while c >= 0 do
         (writec(if c >= 'a' & c <= 'z' /* is c a lower case letter? */
                 then c + 'a' - 'A'     /* Yes -- capitalize it */
                 else c);               /* No -- just copy it */
         c := readc())
   end
  *)

- Parser.parseFile "../test/uppercase.kty";
val it =
  Let
    ([Decl ("c",NullApp ReadC)],
     While
       (BinApp (Geq,VarRef "c",IntLit 0),
        Seq
          [UnApp
             (WriteC,
              If
                (If
                   (BinApp (Geq,VarRef "c",CharLit #"a"),
                    BinApp (Leq,VarRef "c",CharLit #"z"),Const False),
                 BinApp
                   (Sub,BinApp (Add,VarRef "c",CharLit #"a"),CharLit #"A"),
                 VarRef "c")),Assign ("c",NullApp ReadC)])) : AST.exp
```

Here are a few simple examples of the kinds of error messages that should be generated by `parseString` (similar ones should be generated by `parseFile`).

```
(* Missing operator *)
- Parser.parseString "2 * 3 4";

*************************** PARSE ERROR ***************************
Could not parse tokens as a expression:

[INT(2) <2, 3>] [MUL <4, 5>] [INT(3) <6, 7>] [INT(4) <8, 9>]
*******************************************************************

(* Missing var in declaration *)
- Parser.parseString "let c := 1 in c end";

*************************** PARSE ERROR ***************************
Could not parse a prefix of the following tokens as a declarations of a
LET:

[ID("c") <6, 7>] [GETS <8, 10>] [INT(1) <11, 12>] [IN <13, 15>]
[ID("c") <16, 17>] [END <18, 21>]
*******************************************************************

(* Missing := in declaration *)
- Parser.parseString "let var c 1 in c end";

*************************** PARSE ERROR ***************************
Could not parse a prefix of the following tokens as a GETS:

[INT(1) <12, 13>] [IN <14, 16>] [ID("c") <17, 18>] [END <19, 22>]
*******************************************************************

(* Missing rhs definition in let expression *)
- Parser.parseString "let var c := in end";

*************************** PARSE ERROR ***************************
Could not parse a prefix of the following tokens as a rhs of a variable
declaration:

[IN <15, 17>] [END <18, 21>]
*******************************************************************

(* Missing in in let expression *)
- Parser.parseString "let var c := 1 c end";

*************************** PARSE ERROR ***************************
Could not parse a prefix of the following tokens as a IN:

[ID("c") <17, 18>] [END <19, 22>]
*******************************************************************

(* Misspelled end in let expression *)
- Parser.parseString "let var c := 1 in c en ";

*************************** PARSE ERROR ***************************
Could not parse a prefix of the following tokens as a END:

[ID("en") <22, 24>]
*******************************************************************
```

Debugging

Monadic parsers can be tricky to debug. Typical bugs are:

· The parser goes into an infinite regress. See the ML Warts section (below) for an explanation of some common situations where this can happen.

· The parser returns a lazy list of attempts in an improper order. As discussed in Part (d) below, many parts of the parser assume an invariant where attempts that consume more tokens should precede those that consume fewer tokens.

· Uses of `choose` (rather than `plus`) that cut off backtracking. Not that combinators like `many`, `sepby`, and the `chain` combinators uses `choose` rather than `plus` in their definitions. This can lead to unexpected results if you are not careful with how attempts are ordered.

Here are some tips for debugging your code:

· Follow the steps (Parts (a) – (f)) described below. Completely debug one step before moving to the next one. A big problem with debugging monadic parsers is that it's very difficult to pinpoint where the source of an error is. If you implement the parser in small chunks, you drastically cut down the number of spots that can be giving rise to errors.

· Using `nonZero` to annotate you parsers (as described in Part (b)) not only helps find bugs in the Kitty code you are parsing but can also point out bugs in the parser itself. Any feedback is helpful!

· The `trace` combinator can be used to annotate any parser and print out the tokens encountered when that parser is applied. For instance, if `parseExp` is defined as

```
fun parseExp () = body
```

you can trace every call to the `(parseExp())` parser by redefining it as

```
fun parseExp () = trace “parseExp” (body)
```

Due to the backtracking nature of monadic parsers, the order of information printed out by `trace` can be counterintuitive. Nevertheless, the information is often very useful.

ML Warts

The fact that ML is a call-by-value language and only allows the definition of recursive *functions* and not general recursive *values* makes expressing monadic parsers in ML less elegant than in a language like Haskell (which supports laziness and general recursive values) or even Scheme (which is call-by-value but at least supports general recursive values). In particular, you will often want to make two or more parsers mutually recursive. Your first attempt might be:

```
val parseA = ... parseB ...
val parseB = ... parseA ...
```

This won't work in ML because value declarations are sequential, and the reference to `parseB` on the first line is not in the scope of the declaration of `parseB` on the second line.

To get the correct scoping, it is necessary to convert both parseA and parseB into *thunks* (i.e., nullary functions). Of course, references to the thunks within the function bodies must now be forced (i.e, applied to zero arguments).

```
   fun parseA () = ... parseB() ...
   and parseB () = ... parseA() ...
```

The use of the keyword **and** to introduce the second declaration is crucial, because it makes the two declarations mutually recursive. Replacing **and** by **def** would make the function declarations sequential, which would not achieve the desired scoping. Accidentally using **def** instead of **and** is a common error that against which you should be vigilant.

The call-by-value nature of ML also leads to problems. For example, call-by-value evaluation makes the following parenthesis combinator I introduced in class a*bad* idea:

```
     (* parens : 'a parser ->  parser *)
     (* Bad version that uses an unthunked parser *)
     fun parens parser =
           bind (parseTag(LPAREN)) (fn _ =>
            bind parser (fn ans =>
             bind (parseTag(RPAREN)) (fn _ =>
              return ans)))
```

The reason why this is bad is that it makes it easy to fall into infinite recursions. For example, consider using the above form of parens to implement a parser for expressions that includes explicitly parenthesized expressions:

```
     fun parseExp () = plus (parens(parseExp()) ...
```

Here, the occurrence of parseExp() on the right is not delayed in any, and so a call to parseExp() leads to an infinite regress.

This problem can be addressed by modifying parens to accept a parser-returning *thunk* (i.e., function of zero arguments) rather than the parser itself:

```
     (* parens : (unit 'a parser) ->  parser *)
     (* The parser argument is thunked to help avoid infinite recursions *)
     fun parens parserThunk =
           bind (parseTag(LPAREN)) (fn _ =>
            bind (parserThunk()) (fn ans =>
             bind (parseTag(RPAREN)) (fn _ =>
              return ans)))
```

Now we can safely define parseExp without the threat of an infinite regress as follows:

```
   fun parseExp () = plus (parens parseExp) ...
```

The reason that this is safe is that the forcing of the thunk (applying it to zero arguments) is “protected” or “delayed” by the outermost (fn _ => ...) in the definition of parens. Thus, (parens parseExp) can return a parser without first applying parserExp to zero arguments.

As another example of problems due to call-by-value evaluation, consider seq operator I introduced in class, which is also a bad idea. Recall that seq is defined as:

```
   fun seq p q = bind p (fn _ => q)
```

Suppose we try to use seq in our “fixed” version of parens:

```
(* parens : (unit 'a parser) ->  parser *)
(* The parser argument is thunked to help avoid infinite recursions *)
fun parens parserThunk =
      seq (parseTag(LPAREN))
          (parserThunk()) (fn ans =>
            seq (parseTag(RPAREN))
                (return ans)))
```

Unfortunately, this removes the "delaying" effect of the outermost `(fn _ => > ...)`, and the invocation `(parserThunk())` is again "unprotected". This will reintroduce an infinite regress in the `parseExp` example from above. For this reason, I strongly recommend that you **not** use the `seq` combinator unless you really know what you're doing!

These sorts of problems are an artifact of ML's call-by-value evaluation strategy and would not be seen in a lazy language like Haskell.

Problem Decomposition

You could flesh out the definition of `Parser.sml` in one go before testing it, but this would make debugging *very* difficult. Monadic parsers are somewhat fragile programs in which it's very easy to encounter unexpected results due to the ordering of arguments in a `plus` or a `choose`. Moreover, the call-by-value nature of ML makes infinite recursions a real threat if you're not careful.

It's a good idea to implement the Kitty parser in small steps, and make sure that each step is completely working before going on to the next step. Below is a suggested sequence of steps. Although you are not required to do the problem in the order suggested, it is strongly recommended that you debug and test the steps in this order.

The key function you are implementing in all cases is the `parseExp` function, but you should define lots of auxiliary functions to enhance readability and simplify testing and debugging.

Part a. Literals [10]

Implement the parsing of Kitty literals: integer literals, character literals, and constants. It is helpful (though not necessary) to use the function `parseTagFunction` provided in `Parser.sml`. (See Appendix A for documentation.)

Part b. Simple Applications [20]

Implement the parsing of nullary and unary applications, as well as unary negation and applications of `writes`. Recall that unary application desugars to binary subtraction from zero, and that `writes` desugars into a sequence of application of `writec`.

The `parseTag` function provided in `Parser.sml` is helpful for parsing a token with a particular tag. As in Part (a), you will probably also find it helpful to use `parseTagFunction` function as well.

Once you know that you are parsing a particular kind of expression (a unary application, say) you can use `nonZero` combinator (described Appendix B) to flag as an error anything that doesn't match the expected form. For instance, consider a monadic parser implementation of the `E → +(E, E)` rule for prefix-style integer expressions:

```
fun parseAdd () =
      bind (parseToken(OP(Add))) (fn _ =>
       bind (parseToken(LPAREN)) (fn _ =>
        bind (parseExp()) (fn left =>
         bind (parseToken(COMMA)) (fn _ =>
          bind (parseExp()) (fn right =>
           bind (parseToken(RPAREN)) (fn _ =>
            return Binop(Add, left, right)))))))
```

(Note: this definition uses different conventions than those in `Parser.sml`.) Here is how the above parser can be annotated with `nonZero` to signal an error when the form of the addition expression is incorrect:

```
   fun parseAdd () =
         bind (parseToken(OP(Add))) (fn _ =>
          bind (nonZero "left paren of add" (parseToken(LPAREN))) (fn _ =>
           bind (nonZero "left argument of add" (parseExp())) (fn left =>
            bind (nonZero "comma of add" (parseToken(COMMA))) (fn _ =>
             bind (nonZero "right argument of add" (parseExp())) (fn right =>
              bind (nonZero "right paren of add" (parseToken(RPAREN))) (fn _ =>
               return Binop(Add, left, right)))))))
```

The `nonZero` combinator asserts that when the annotated parser is invoked on a token list, it should produce a non-empty lazy list of attempts. If this assertion is violated, then the parser raises a `zeroParse` exception with the given string and token list. The default exception handler of the top-level `parseAllErr` function prints out an error with the string and string representations of the tokens. (Examples of such errors appear on the previous page.)

Note that `nonZero` should only be used when the parser is "sure" that it is parsing an expression of a particular type. For example, `nonZero` should not be used to annotate `(parseToken(OP(Add)))` in the above example. Presumably it would just be one of several parsers that were tried against a given token stream, and an error shouldn't necessarily be raised if the first token is not a +.

Part c. Expessions Beginning with Keywords [55]

Implement the keyword-introduced expressions `if`, `let`, `while`, and `for`. Recall that `for` desugars into a let expression whose body contains a `while` loop.

For the desugaring, it will be helpful to use the `freshId` function in `Parser.sml` to generate "fresh" identifiers. For instance, the first call to `freshId("foo")` might return (`"foo.17"`); a later call might return (`"foo.42"`). A call to `freshId` is guaranteed to return an identifier that is different from every other identifier in the program, including identifiers previously returned by `freshId`. This is accomplished by appending a dot and the value of a counter to the given base string. The dot guarantees that the identifiers returned by `freshId` are disjoint from other program identifiers, since identifiers in user programs cannot contain a dot.

As part of parsing `let` expressions you will need to (1) parse semi-colon-separated sequences of variable declarations and (2) parse semi-colon-separated sequences of expressions in the body. The `sepby1` combinator is handy in both cases.

Recall that an `if` expression has two forms: `if E1 then E2` and `if E1 then E2 else E3`. The former desugars into the latter where `E3` is `()`. As noted in the Kitty reference manual, an `else` clause binds most tightly to the innermost enclosing `if` . Thus, you must parse

```
   if E1 then if E2 then E3 else E4
```

as if it were explicitly parenthesized as

```
   if E1 then (if E2 then E3 else E4)
```

and not as

```
   if E1 then (if E2 then E3) else E4.
```

As in Part (b), it is helpful to use `nonZero` to flag syntax errors in these expressions. For parsing keywords that *must* be present in an expression, the following `parseTagRequired` function provided in `Parser.sml` is handy:

```
   fun parseTagRequired tag = nonZero (tagToString(tag)) (parseTag tag)
```

Without binary operators, variable references, and variable assignments, it is difficult to test this class of expressions on realistic examples. Nevertheless, you should extensively test each kind of keyword-introduced expression using simple subexpressions before proceeding.

Part d. Variable References and Assignments [10]

Implement variable references and assignments. Because both of these expressions begin with an identifier, it is impossible to distinguish them based on the first token. It turns out that the rest of the parser depends on an invariant that attempts should be ordered so that ones consuming the most tokens come first. Intuitively, this means that when the parser has a choice, it should choose a parsing that matches more tokens before one that matches fewer tokens.

In the context of variable references and assignments, this means that checking for an assignment should be done *before* checking for a variable reference.

Part e. Parenthesized Expressions [5]

Implement the parsing of parenthesized expressions, which includes:

· `()` Nothing

· `(E)` Explicit grouping

· `(E1;…;En)` Sequence expressions

If you plan to use a `parens` combinator, be sure to first study the discussion in the ML Warts section above.

Part f. Binary Application [50]

· The final, but perhaps trickiest, step is to implement applications of Kitty's binary operators. There are several issues here:

· The parser must implement the precedence of the binary operators. Recall that the precedence, arranged from high to low precedence by levels, is as follows:

```
      *, /
      +, -
      <, <=, =, <>, >=, >
      &
      |
```

· The parser must implement the associativity of the binary operators. All operators except for relational operators are left-associative. The relational operators are non-associative.

· The short-circuit conjunction (`&`) and disjunction (`|`) operators desugar into `if` expressions. Unless you are very careful, it is easy to get accidental infinite regresses.

Here are some hints on how to address these concerns:

- Use the infix integer expression language discussed in class as a model. Treat the constructs in Parts (a) – (e) as “factors” for the binary operators. Construct a hierarchy like the expression/term/factor hierarchy that can handle the five levels of precedence in Kitty.

- Use `chainl1` to implement left associativity of most of the binary operators.

- Use `freshId` in the desugaring of `|`.

- Do not try to use nonZero to detect errors in binary applications – it’s unlikely that you can use it to do anything reasonable.

Once you complete this part, you should be able to test your parser on any Kitty program.

Appendix A: Parser.sml

The skeleton file `Parser.sml` contains the definitions of several functions that you may find helpful:

```
val parseTag : token.tag -> unit parser
```

Returns a parser that consumes the first token if its tag matches the given tag and otherwise doesn't consume any token.

```
val parseTagRequired : token.tag -> unit parser
```

Like `parseTag`, but signals a `zeroParse` exception if the given token is not the first token of the list to be parsed.

```
val parseTagFunction : (token.tag ->'a option) -> 'a parser
```

Given a function f that maps tags to options, returns a parser that applies `f` to the tag of the first token encountered. If the result is `SOME(v)`, the token is consumed, and answer `v` is returned. If the result is `NONE`, the token is not consumed.

```
val freshId : string -> string
```

`freshId(base)` generates a unique identifier that includes the base string `base`. The resulting identifier is guaranteed to be different from any other identifier in the program and from any other identifier returned by `freshId`.

```
val parseScanner : Scanner.scanner -> AST.exp
```

Returns the first abstract syntax tree that can be derived by consuming *all* the tokens in the given scanner. If no expression can be derived, or if there are tokens remaining after an expression is derived, raises a noParse exception. Prints out error messages for any noParse and zeroParse exceptions.

```
val parseString : string -> AST.exp
```

Returns the abstract syntax tree of the expression corresponding to all the tokens in the given string. Raises an exception if the tokens cannot be parsed into an expression, giving an indication of why this is so.

```
val parseFile : string -> AST.exp
```

Like `parseString`, but parses the tokens in the contents of the file named by the given string.

Appendix B: The ML Monadic Parser library

The file parser/MonadicParser.sml contains an implementation of Graham Hutton and Erik Meijer's monadic parsing library in ML. For details about the Haskell version of this library, see:

> Graham Hutton and Erik Meijer. "Monadic Parser Combinators". *Journal of Functional Programming*, 8(4):437-444. Cambridge, University Press, July 1998.

Of related interest is the description of combinator parsing in:

> Jeroen Fokker. "Functional Parsing". In *Advanced Functional Programming*, Johan Jeuring and Erik Meijer (Eds.) Lecture Notes in Computer Science 925, pp. 1-23. Springer Verlag, 1995

This ML library generalizes Hutton and Meijer's work by allowing parsers to work over lists of any type of element, not just lists of characters. This is achieved by defining monadic parsers over a token structure that must supply a `toString` function.

Here's a comparison of the parser types in Hutton and Meijer, Fokker, and in this library:

- Hutton and Meijer's parser type would be expressed in ML as:

```
datatype 'result parser =
  Parser of char list -> ('result * (char list)) LazyList.lazyList
```

- Fokker's parser type would be expressed in ML as:

```
datatype ('token, 'result) parser =
  Parser of 'token list -> ('result * ('token list)) LazyList.lazyList
```

- In this library, the parser type used is:

```
datatype 'answer parser =
  Parser of token list -> ('answer * (token list)) LazyList.lazyList
```

In this library, `token` is the type "built into" and instance of the parser by applying the functor `MonadicParserFunctor` to a structure satisfying the `SHOWABLE` signature:

```
signature SHOWABLE = sig
  type element
  val toString : element -> string
end
```

Within the structure returned by invoking the functor, the `token` type is defined to be a synonym for the `element` type of the argument structure. The `toString` function is used to display tokens when an error is encountered by the parser.

Here is a example of how the MonadicParserFunctor functor can be applied to yield a monadic parser structure specialized to a particular token type:

```
structure TokenMonadicParser =
    MonadicParserFunctor(struct
                          type element = Token.token
                          val toString = Token.toString
                         end
                        )
```

In this case, a functor is used in ML to achieve the same effect as the `Showable` type class in Haskell. In general, functors are a more flexible way than type classes for parameterizing modules.

Below are annotated signatures of the functions exported by the ML monadic parsing library. We will use the term "attempt" to pairs of the form (answer, token list). Thus, a parser can abstractly be viewed as a function that maps a (non-lazy) list of tokens to a lazy list of attempts.

`exception noParse of token list`
Exception raised by the `parseAll` function when there are no parses that consume all tokens.

`exception zeroParse of string * token list`
Exception raised by parsers constructed with the `nonZero` combinator when the underlying parser generates an empty sequence of attempts.

`val zero : unit -> 'a parser`
Returns a parser that always returns zero attempts.

`val return : 'a -> 'a parser`
`return v` returns a parser that when invoked on a list of tokens `ts` generates a single attempt (v,ts).

`val item : token parser`
`item` is a parser that consumes and returns the first token of a token list. When invoked on an empty list, it behaves like `zero()`.

`val bind : 'a parser -> ('a -> 'b parser) -> 'b parser`
Suppose that `p` is a parser that when invoked on a list of tokens ts returns the attempts (a_1,ts_1), (a_2, ts_2), … Then `bind p f` concatenates together the attempt lists that result from applying the parser (f a_i) to the token list ts_i.

`val seq : 'a parser -> 'b parser -> 'b parser`
Suppose that `p` is a parser that when invoked on a list of tokens ts returns the attempts (a_1,ts_1), (a_2, ts_2), … Then `seq p q` concatenates together the attempt lists that result from applying the parser `q` to the token list ts_i.

`val sat : (token -> bool) -> token parser`
Given a predicate `pred`, returns a parser that consumes the first token of a list only if it satisfies `pred`.

`val plus : 'a parser -> 'a parser -> 'a parser`
The invocation `plus p q` returns a parser that concatenates the results of parsing a token list with `p` and with `q`.

`val plusList : 'a parser list -> 'a parser`
Returns the result of folding `plus` over the given list of parsers.

`val choose : 'a parser -> 'a parser -> 'a parser`
The invocation choose p q returns a parser that is like plus p q but which avoids backtracking by only keeping the *first* attempt (any other attempts are effectively thrown away).

`val chooseList : 'a parser list -> 'a parser`
Returns the result of folding `choose` over the given list of parsers.

`val times : 'a parser -> 'b parser -> ('a * 'b) parser`
The invocation `times p q` returns a parser that applies the parsers p and q in order on a given token list and returns attempts whose answers are pairs of the two answers from p and q.

`val map : ('a -> 'b) -> 'a parser -> 'b parser`

The invocation `map f p` returns the attempts that result by mapping `p` over the answer component of each attempt generated by `p`.

`val many : 'a parser -> 'a list parser`

The invocation `many p` returns a parser that parses the maximal number of tokens matched by a sequence of zero or more invocations of `p` and returns a list of the answers of the individual invocations of `p`.

`val many1 : 'a parser -> 'a list parser`

Like `many`, but for a sequence of one or more invocations `p`.

`val sepby : 'a parser -> 'b parser -> 'a list parser`

The invocation `sepby p q` returns a parser that parses the sequences of zero or more tokens matched by p that are separated by tokens match by q.

`val sepby1 : 'a parser -> 'b parser -> 'a list parser`

Like `sepby`, but for a sequence of one or more invocations `p` separated by tokens matching q.

`val chainr : 'a parser -> ('a -> 'a -> 'a) parser -> 'a -> 'a parser`

Like `sepby`, except that the separator results are functions that are folded over from right to left (via foldr) the separated elements.

`val chainr1 : 'a parser -> ('a -> 'a -> 'a) parser -> 'a parser`

The version of `chainr` that requires at least one element.

`val chainl : 'a parser -> ('a -> 'a -> 'a) parser -> 'a -> 'a parser`

Like `chainr`, except the results are folded from left to right (via foldl).

`val chainl1 : 'a parser -> ('a -> 'a -> 'a) parser -> 'a parser`

The version of `chainl` that requires at least one element.

`val until : 'b parser -> 'a parser -> 'a list parser`

Returns a parser that parses the maximal number of ‘a elements before a ‘b element is encountered.

`val parse : 'a parser -> token list -> ('a * (token list)) LazyList.lazyList`

Applies the given parser to the given list of tokens.

`val parseAll : 'a parser -> token list -> 'a`

Applies the given parser to the given list of tokens, and returns the first answer that consumes all tokens.

`val parseAllErr : string -> 'a parser -> token list -> 'a`

Like parseAll, but prints out error messages for noParse and zeroParse exceptions.

`val trace : string -> 'a parser -> 'a parser`

Invoking `trace s p` returns a parser that prints out s and ts whenever it is applied to a list of tokens ts.

`val nonZero : string -> 'a parser -> 'a parser`

Invoking `nonZero s p` returns a parser that prints out s and ts whenever it is applied to a list of tokens ts and no attempts are generated.

Problem Set Header Page
Please make this the first page of your hardcopy submission.

CS301 Problem Set 3

Due Friday, October 13, 2000 (Scary!)

Names of Team Members:

Date & Time Submitted:

Soft Copy Directory:

Collaborators (*any teams collaborated with in the process of doing the problem set*):

*In the **Time** column, please estimate the total time each team member spent on the parts of this problem set. (Note that spending less time than your partner does not necessarily imply that you contributed less.) Please try to be as accurate as possible; this information will help me to design future problem sets. I will fill out the **Score** column when grading your problem set.*

Part	**Time For** ______ **(Team Member #1)**	**Time For** ______ **(Team Member #2)**	**Score**
General Reading			
Problem 1a [10]			
Problem 1b [20]			
Problem 1c [50]			
Problem 1d [10]			
Problem 1e [10]			
Problem 1f [50]			
Total			