```
%{
        (*i $Id: parse.mly,v 3.23 2001/03/19 22:41:38 sufrin Exp $  i*)

        open Syntax

        (* Implements the syntactic sugar:

                $\lambda x_1 x_2 ... . e ==> \lambda x_1 . \lambda x_2 . .... . e$

           Needs bv as composite or constant patterns to be implemented

        *)

        let rec mkFun : string list * expr -> expr = function
            |   ([i],   e) -> Fun(i, e)
            |   (i::is, e) -> Fun(i, mkFun(is, e))


        (*
            Implements the syntactic sugar:
                            $a ~binop~ b ==> (binop)(a, b)$
        *)
        let mkApply f e1 e2 = Apply(f, Pair(e1, e2))
        let mkCons = mkApply (ConId "::")

        (* Implements outfix application notation *)
        let mkOutfix (id, left, right) expr (id', left', right') =
            if right=right' then Apply(Id id, expr) else failwith ("non-matching closing bracket: "^left^estring
expr^right')

        let quoteOutfix (id, left, right) (id', left', right') =
            if right=right' then id else failwith ("non-matching closing bracket: "^left^right')

(*
        Normally each equation in LET declns IN expr is deemed recursive; but DEF anddecl IN expr
        makes the equations in anddecl nonrecursive. This, for the moment, is the workhorse that
        strips RecDec where necessary. It is also used to consolidate the individual RecDecs of
        equations in REC ... AND .... No doubt things could have been simpler and could still be
        made simpler. The entire grammar needs some work.
*)

       let rec stripRec = function
           |  AndDec (l, r) -> AndDec(stripRec l, stripRec r)
           |  RecDec d      -> d
           |  other         -> failwith ("Error in stripRec: "^dstring other)

        let unRec = stripRec

        let mkRecDec dec = RecDec(stripRec dec)

(* \subsubsection*{Infix operators and Sections}

Unlike Haskell, infixes are uncurried. Thus for any infix, say ↓, defined by v↓w = E

\begin{verbatim}
                (↓)     means { (v,w) ->  E }
                e1 ↓ e2 means E[v, w := e1, e2]
                (↓expr) means \ v . v↓expr
                (expr↓) means \ v . expr↓v
\end{verbatim}

           Except that

\begin{verbatim}
                (-expr) means (0-expr)
\end{verbatim}
*)
        let bvCount = ref 0
        let newbv() = (bvCount := 1+(!bvCount); Format.sprintf "$bv%d" (!bvCount))

        let mkRSection : expr -> string -> expr = fun e -> function
            |   "-"  -> mkApply (Id "-") (Num Number.zero) e
            |   op   -> let id = newbv() in Fun(id, mkApply (Id op) (Id id) e)

        let mkLSection : expr -> string -> expr =
        fun e op -> let id = newbv() in Fun(id, mkApply (Id op) e (Id id))


        (*
            Syntactic sugar for function declarations is elaborated here

                $f x_1 x_2 ... = e ==> f = \lambda  x_1 -> \lambda x_2 -> ... -> e$

            Appropriate treatment is meted out to bound variables specified by
            constant or composite patterns; to wit they are turned into singly-branched
            case functions. For example:

                $f (x,y) [z] = x+y+z ==> {(x,y) -> { z::Nil -> (x + (y + z))}$

        *)
        let mkValDec pat expr =
        let rec abstractFrom = fun loc expr pat -> match stripAt(pat) with
            (* Direct pattern on lhs of declaration *)
            |   Id     _
            |   Hole   _
```

```
            |   Pair   _
            |   ConId  _
            |   Num    _
            |   Has    _
            |   Unit
            |   Apply(ConId _, _) -> ValDec(pat, expr)
            (* Otherwise pattern as bound variable *)
            |   Apply(rator, Id i) -> abstractFrom loc (Fun(i, expr)) rator
            |   Apply(rator, rand) ->
            ( match rand with
            | Num   _
            | ConId _
            | Unit
            | Apply(ConId _, _)
            | Has _
            | Pair  _  -> abstractFrom loc (At(CaseFun([(rand, expr)]), loc)) rator
            | other    -> failwith ("invalid argument pattern in LHS of declaration:" ^ estring pat ^ " = " ^
estring expr)
            )
            | other  -> failwith ("invalid LHS of declaration:" ^ estring pat ^ " = " ^ estring expr)
        in  match pat with
            | At (pat, loc) -> if holeIn expr then failwith (" hole on rhs of declaration starting at "^lstring loc)
else abstractFrom loc expr pat
            | _             -> abstractFrom nowhere  expr pat

        let mkLet(decls, body) = Let(decls, body)

        (* Type expressions *)

        open Type
        open Env

        let mkSignature decls  =
            let has(HasType (i, t, _)) env = (i, t)::env
        in  StructType (List.fold_right has decls emptyenv)

        (* The built in type definition environment must be consistent with this *)
        let mkType =  function
            |   "Num"    -> NumType
            |   "Bool"   -> BoolType
            |   "Char"   -> CharType
            |   "String" -> ExpandedType(ConType("String", []), ListType CharType)
            |    cid     -> ConType(cid, [])

        let rec flatten = function
        | PairType(l, r) -> l::flatten r
        | other          -> [other]

        (* We want to distinguish between an arity=1 type constructor being applied
           to a tuple type, and an arity>1 type constructor applied to a "tuple"
           of argument types. At this point (the parser) we are prepared to be
           sloppy about exact argument numbers.
        *)

        let multiarg conid = false

        let mkConType cid arg =
            match arg with
            | PairType _ -> ConType(cid, if multiarg cid then flatten arg  else [arg])
            | _          -> ConType(cid, [arg])

        exception Syntax

        let location: Lexing.position  -> location =
            fun pos -> (pos.pos_lnum, pos.pos_cnum - pos.pos_bol, pos.pos_fname)

        let locate:   Lexing.position -> expr -> expr =
            fun pos e ->
                match e with
                | At _  -> e
                | _     -> At(e, location pos)

        let dlocate pos e = e

        let locatePhrase: Lexing.position -> phrase -> phrase =
            fun pos phrase ->
                (match phrase with
                | Located _ -> phrase
                | _         -> Located(phrase, location pos)
                )
%}

%token <string> NUM ID CONID EQ
                BINR0 BINL0 CONR0 CONL0
                BINR1 BINL1 CONR1 CONL1
                BINR2 BINL2 CONR2 CONL2
                BINR3 BINL3 CONR3 CONL3
                BINR4 BINL4 CONR4 CONL4
                BINR5 BINL5 CONR5 CONL5
                BINR6 BINL6 CONR6 CONL6
                BINR7 BINL7 CONR7 CONL7
                BINR8 BINL8 CONR8 CONL8
                BINR9 BINL9 CONR9 CONL9
                CONS
```

```
%token <string*string*string> LEFT RIGHT

%token <string> STRING

%token <Utf8string.unicodechar> CHAR (* a string encoded in utf8 *)

%token FUN NUF LAM CURLYBRA CURLYKET BRA KET COMMA DOT  AT TO LET REC AND WHERE IN
       END ALL COLON SEMI EOF IF THEN ELSE SQBRA SQKET
       DATA ALT IMPORT HOLE
       NOTATION TYPE INSIDE DEF WITH DO

%right TO

(* Increasing priorities: operators *)
%right       WHERE
%right       WITH
%right       AT
%left        DOT
%right BINR0, CONR0
%left  BINL0, CONL0
%right BINR1, CONR1
%left  BINL1, CONL1
%right BINR2, CONR2
%left  BINL2, CONL2
%right BINR3, CONR3
%left  BINL3, CONL3
%right BINR4, CONR4, CONS
%left  BINL4, CONL4
%right BINR5, CONR5
%left  BINL5, CONL5
%right BINR6, CONR6
%left  BINL6, CONL6
%right BINR7, CONR7
%left  BINL7, CONL7
%right BINR8, CONR8
%left  BINL8, CONL8
%right BINR9, CONR9
%left  BINL9, CONL9


(* Phrases are evaluated at the top-level *)

%start          phrase
%start          interact
%type           <Syntax.phrase> phrase
%type           <Syntax.phrase> interact

%%
        (* Interactive definitions are in two forms: some people like the consistency of terminating interactive
           phrases with semicolons and will use DEF primdecl ; others are happy to use LET declns ;; It's not
           easy to dispense with the ;; after an interactive LET, because there's then an ambiguity arising
           from LET declns IN expr.

           [If I were to make declarations distinct from equality expressions the various ambiguities might
            disappear. But I hate "==" for equal.]

           TODO: In this grammar all simple declarations are automatically made recursive; but collections of
           mutually recursive declarations have to be introduced by REC ... AND ... AND ... . This is not
           as convenient as (eg) the Haskell practice of automatically collecting mutually-dependent
           equations into clusters, each of which is treated as if it had been made by REC ... AND ... AND ....
           The implementation is more or less completely straightforward, and could be done in a phase
           before the type checker runs.
        *)


interact:
                expr          SEMI              {if holeIn $1 then Error (" hole in expression at
"^lstring(location($startpos($1)))) else Expr $1}
        |       expr          END               {if holeIn $1 then Error (" hole in expression at
"^lstring(location($startpos($1)))) else Expr $1}
        |       DO expr       End               {if holeIn $2 then Error (" hole in expression at
"^lstring(location($startpos($2)))) else Expr $2}
        |       LET  declns   END               {Decl $2}
        |       REC  anddecl  End               {Decl (mkRecDec $2)}
        |       DATA datadefs End               {Data $2}
        |       TYPE typedefs End               {Type $2}
        |       IMPORT STRING End               {Utils.usefile $2; Expr Unit}
        |       IMPORT ID     End               {Utils.usefile $2; Expr Unit}
        |       IMPORT CONID  End               {Utils.usefile $2; Expr Unit}
        |       error                           {Error (lstring(location($startpos))) }
        |       SEMI                            {Expr Unit}
        |       END                             {Expr Unit}
        |       NOTATION notations End          {Utils.declareNotations (lstring(location($startpos))) $2; Expr
Unit}
        |       EOF                             {Eof}


phrase  :       LET declns END                  {Decl $2}
        |       declns     END                  {Decl $1}
        |       declns     EOF                  {Decl $1}
        (*      Force expression evaluation within a file (files normally contain only definitions) *)
        |       DO expr    End                  {if holeIn $2 then Error (" hole in expression at
"^lstring(location($startpos($2)))) else Expr $2}
        (*      ************************************************************************* *)
        |       DATA datadefs End               {Data $2}
```

```
        |       TYPE typedefs End              {Type $2}
        |       IMPORT STRING End              {Utils.usefile $2; Expr Unit}
        |       IMPORT ID     End              {Utils.usefile $2; Expr Unit}
        |       IMPORT CONID  End              {Utils.usefile $2; Expr Unit}
        |       NOTATION notations End         {Utils.declareNotations (lstring(location($startpos))) $2; Expr
Unit}
        |       EOF                            {Eof}

End     :       SEMI {()}
        |       END  {()}
        |       EOF  {()}


notations:      notation                       { [$1] }
        |       notation AND notations         { $1 :: $3 }

notation:       ID number symbols              { ($1, $2, $3) }

number  :       NUM                            { $1 }
        |                                      { "0" }

symbols:                                       { [] }
        |       INFIX symbols                  { $1::$2 }
        |       ID symbols                     { $1::$2 }
        |       CONID symbols                  { $1::$2 }

declns  :       seqdecl                        {$1}
        |       seqdecl WHERE declns           {WhereDec($1, $3)}
        |       error                          {failwith "in declaration"}


expr    :       term                           { locate $startpos $1 }       (* located *)
        |       term  EQ expr                  { locate $startpos @@ mkApply (Id "=") $1 $3}
        |       LET declns IN expr             { locate $startpos @@ mkLet($2, $4) }
        |       DEF anddecl IN expr            { locate $startpos @@ mkLet(unRec $2, $4) }
        (*      \ x y z -> e ==> \x -> \y -> \z -> e *)
        |       LAM ids TO expr                { locate $startpos @@ mkFun($2, $4)}
        |       IF expr THEN expr ELSE expr    { locate $startpos @@ If($2, $4, $6)}
        |       term  INSIDE expr              { dlocate $startpos @@ Inside($1, $3) }

ids     :       ID                             {[$1]}
        |       ID ids                         {$1::$2}

term    :       app                            { dlocate $startpos @@ $1}
        |       term  WITH term                { dlocate $startpos @@ With ($1, $3)}
        |       term  BINR term                { dlocate $startpos @@ mkApply (Id $2) $1 $3}
        |       term  BINL term                { dlocate $startpos @@ mkApply (Id $2) $1 $3}
        |       term  CONR term                { dlocate $startpos @@ Apply (ConId $2, Pair($1, $3))}
        |       term  CONL term                { dlocate $startpos @@ Apply (ConId $2, Pair($1, $3))}
        |       term  AT   term                { dlocate $startpos @@ Apply($1, $3)}
        |       term  CONS term                { dlocate $startpos @@ mkCons $1 $3}

app     :       prim                           { dlocate $startpos @@ $1}
        |       app prim                       { locate $startpos @@ Apply($1, $2)}
        |       app DOT  id                    { dlocate $startpos @@ Select($1, $3)}

prim    :       id                             {Id $1}
        |       HOLE                           {mkHole()}
        |       CONID                          {ConId $1}
        |       NUM                            {Num (Number.num_of_string $1)}
        |       CHAR                           {Char $1}
        |       STRING                         {String $1}
        |       BRA expr COMMA exprs KET       {Pair($2, $4)}
        |       SQBRA elist SQKET              {$2}
        |       BRA expr KET                   {$2}
        (* HACK

           The automatic lifting to polytypes of types used to type-hint expressions
           is counterintuitive; but it makes the implementation very easy.

           The hack: the type of a hinted expression (e: t) is computed by unifying the
           type of e with an INSTANTIATION of t.

           Here's why the hack is horrible, and annotations should be
           used sparingly, if at all right now. Consider this version
           of compose:

            let (g: b->c) `compose` (f: a->b) = 〈 (x: a) -> g(f x) 〉;;

           It types, as expected, to

              (`compose`) : @a,b,c.(c->b,a->c)->a->b

           But then so do these:

            let (g: @b,c. b->c) `compose` (f: @b,c.b->c) = 〈 (x: a) -> g(f x) 〉;;
            let (g: b->c) `compose` (f: b->c) = 〈 (x: a) -> g(f x) 〉;;
            let (g: b->c) `compose` (f: b->c) = 〈 (x) -> g(f x) 〉;;

           It is evident that the annotations don't capture the
           expected relations between the types of the parameters
           here, but that the outcome is a correct typing.

           In fact the regular type inference algorithm imposes
           additional constraints on the variables generated by the
```

```
           INSTANTIATIONS of the types in the annotations. The
           annotations are interpreted as hints, with scope strictly
           local to the expression they annotate.

           TODO: fix this mess. Probably by finding a way of using
           signature information to prime the environment used to
           infer the environment resulting from a declaration. Right
           now we simply /check/ the signature for consistency with
           (perhaps a specialization of) the inferred type.

        *)
        |       BRA typedexpr COMMA typedexprs KET  {Pair($2, $4) }
        |       BRA typedexpr  KET                     { $2 }
        |       LEFT expr RIGHT                 {mkOutfix $1 $2 $3}
        |       BRA KET                         {Unit}
        |       BRA INFIX term KET              {mkRSection $3  $2 }
        |       BRA term INFIX KET              {mkLSection $2  $3 }
        |       FUN cases NUF                   {CaseFun (List.rev $2) }
        |       LAM BRA cases KET               {CaseFun (List.rev $3) }
        |       BRA ALT bcases KET              {CaseFun ($3) }
        |       CURLYBRA equations CURLYKET     {Struct $2 }

equations:      equation                        { $1 }
        |       equations SEMI equation         { AndDec($1, $3) }

cases   :       case                            {[$1]}
        |       cases ALT case                  {$3 :: $1}

bcases  :                                       {[]}
        |       case                            {[$1]}
        |       case ALT bcases                 {$1 :: $3}

case    :       expr TO expr                    { ($1,$3) }

exprs   :       expr                            {$1}
        |       expr COMMA exprs                {Pair($1, $3)}

typedexprs :    typedexpr                       {$1}
           |    typedexpr COMMA typedexprs      {Pair($1, $3)}

typedexpr  :    expr COLON polytype             {Has($1, $3)}


elist   :                                       {ConId "Nil"}
        |       expr COMMA elist                {mkCons $1 $3}
        |       expr                            {mkCons $1 (ConId"Nil")}

seqdecl :       recdecl                         {$1}
        |       recdecl  SEMI seqdecl           {SeqDec($1, $3)}

recdecl :       anddecl                         {$1}
        |       REC anddecl                     {mkRecDec $2}


anddecl :       primdecl                        {$1}
        |       primdecl AND  anddecl           {AndDec($1, $3)}

        (* SUGAR:
                Simple equational definitions are deemed recursive.
                The decorating RecDec added here may be stripped later depending on context.
                See mkRecDec, and unRec for details (and an apology for the inelegance).
        *)
primdecl:       equation                        {RecDec $1}
        |       polytypespec                    {$1}

equation:       term EQ expr                    {mkValDec (locate $startpos $1) (locate ($startpos($3)) $3)}  (*
located *)

(*
   A polytypespec sugars the type in a top level type specification
   to be polymorphic if it has type variables in it.

   A fieldspec uses simply uses the type expression as written (it
   may be written as polymorphic).
*)
polytypespec:   id  COLON polytype              {HasType($1, $3, location($startpos($3)))}
fieldspec:      id  COLON typexpr               {HasType($1, $3, location($startpos($3)))}

polytype:       ALL idlist DOT typexpr          {PolyType($2, $4)}
        |       typexpr                         {mkPolyType $1}

typexpr :       primtype                        {$1}
        |       typexpr TO typexpr              {FunType($1, $3)}
        |       typexpr WITH typexpr            {WithType($1, $3)}
        |       CONID                           {mkType $1}
        |       CONID primtype                  {mkConType $1 $2}
        |       error                           {failwith "in type expression"}

primtype:       ID                              {VarType $1}
        |       BRA KET                         {UnitType}
        |       SQBRA typexpr SQKET             {ListType $2}
        |       BRA typexpr COMMA typexprs KET  {PairType($2, $4)}
        |       BRA typexpr KET                 {$2}
        |       CURLYBRA signature CURLYKET     {mkSignature $2}
```

```
signature:                                      {[]}
        |       fieldspec                       {[$1]}
        |       fieldspec SEMI signature        { $1 :: $3 }

typexprs:       typexpr                         {$1}
        |       typexpr COMMA typexprs          {PairType($1, $3)}

idlist  :       ID                              {[$1]}
        |       ID COMMA idlist                 {$1 :: $3}

id      :       ID                              {$1}
        |       BRA INFIX KET                   {$2}
        |       LEFT RIGHT                      {quoteOutfix $1 $2}

datadef :       CONID typarams EQ alts          {($1, $2, $4)}

datadefs:       datadef                         {[$1]}
        |       datadef AND datadefs            {$1::$3}

typedef :       CONID typarams EQ typexpr       {($1, $2, $4)}

typedefs:       typedef                         {[$1]}
        |       typedef AND typedefs            {$1::$3}

typarams:                                       {[]}
        |       BRA idlist KET                  {$2}
        |       ID                              {[$1]}

alts    :       alt                             {[$1]}
        |       alt ALT alts                    {$1::$3}

alt     :       CONID                           {($1, None)}
        |       CONID typexpr                   {($1, Some $2)}
        (* Allow infix data constructors *)
        |       primtype CONL primtype          {($2, Some (PairType($1, $3)))}
        |       primtype CONR primtype          {($2, Some (PairType($1, $3)))}

INFIX   :       BINL                            {$1}
        |       BINR                            {$1}
        |       CONL                            {$1}
        |       CONR                            {$1}
        |       CONS                            {$1}
        |       EQ                              {$1}

%inline
BINR    :  BINR0 {$1} | BINR1 {$1} | BINR2 {$1} | BINR3 {$1} | BINR4 {$1}
        |  BINR5 {$1} | BINR6 {$1} | BINR7 {$1} | BINR8 {$1} | BINR9 {$1}
%inline
BINL    :  BINL0 {$1} | BINL1 {$1} | BINL2 {$1} | BINL3 {$1} | BINL4 {$1}
        |  BINL5 {$1} | BINL6 {$1} | BINL7 {$1} | BINL8 {$1} | BINL9 {$1}
%inline
CONR    :  CONR0 {$1} | CONR1 {$1} | CONR2 {$1} | CONR3 {$1} | CONR4 {$1}
        |  CONR5 {$1} | CONR6 {$1} | CONR7 {$1} | CONR8 {$1} | CONR9 {$1}
%inline
CONL    :  CONL0 {$1} | CONL1 {$1} | CONL2 {$1} | CONL3 {$1} | CONL4 {$1}
        |  CONL5 {$1} | CONL6 {$1} | CONL7 {$1} | CONL8 {$1} | CONL9 {$1}
```