Source Listings

```
fun makedfa(rules) =
let val StateTab = ref nulldict
and fintab = ref nullidict
and transtab = ref nullidict
and tctab = ref nullidict
and (fp, leaf, tcpairs) = leafdata(rules);

fun visit (state,statenum) =
        let val transitions = gettrans(state) in
        fintab := ienter(!fintab)(statenum,getfin(state));
        tctab := ienter(!tctab)(statenum,gettc(state));
        transtab := ienter(!transtab)(statenum,transitions)
        end

and visitstarts (states) =
        let val i = ref 0;
        val rec vs = fn
                  nil => ()
                | (hd::tl) => (visit(hd,!i); i := !i + 1; vs(tl))
        in vs(states)
        end

and hashstate(s: int list) =
        let val rec hs = fn
                  (nil,z) => z
                | (x::y,z) => hs(y,z ^ " " ^ itoa(x))
        in hs(s,"")
        end

and find(s) = lookup(!StateTab)(hashstate(s))

and add(s,n) = StateTab := enter(!StateTab)(hashstate(s),n)

and getstate (state) =
        find(state)
        handle lookup with _ => let val n = ++StateNum in
                add(state,n); visit(state,n); n
                end

and getfin (state) =
        let val fins = ref nil;
        val rec ckfin = fn
                  nil => ()
                | hd::tl => (if isend(leaf sub hd) then fins := hd::(!fins)
                        else (); ckfin(tl))
        in ckfin(state); !fins
        end

and gettc (state) =
        let val tcs = ref nil;
        val rec cktc = fn
                  nil => ()
                | hd::tl => (if istrail(leaf sub hd) then tcs := hd::(!tcs)
                        else (); cktc(tl))
        in cktc(state); !tcs
        end
```

```
and gettrans (state) =
        let val c = ref 0
        and nextleaves = ref nil
        and tlist = ref nil;
        val rec cktrans = fn
                  nil => ()
                | hd::tl => (if isclass(leaf sub hd)
                        andalso class(leaf sub hd) sub !c
                        then nextleaves := union(!nextleaves,fp sub hd)
                        else (); cktrans(tl))
        in while !c <= 127 do (nextleaves := nil; cktrans(state);
                if !nextleaves<>nil then
                tlist := (!c,getstate(!nextleaves))::(!tlist)
                else (); c := !c + 1);
        !tlist
        end

and startstates() =
        let val startarray = array(!StateNum + 1, nil);
        val rec makess = fn
                  nil => ()
                | (startlist,e)::tl => (fix(startlist,firstpos(e));makess(tl))
        and fix = fn
                  (nil,_) => ()
                | (s::tl,firsts) => (update(startarray,s,
                        union(firsts,startarray sub s));
                        fix(tl,firsts))
        in makess(rules);listofarray(startarray, !StateNum + 1)
        end

in visitstarts(startstates());
(listofidict(!fintab),listofidict(!transtab),listofidict(!tctab),tcpairs)
end
```

```
abstype 'a dictionary =
        dict of (string * 'a)list

with
        val nulldict = dict nil

        exception lookup : unit

        fun lookup (dict entrylist:'a dictionary) (key:string) : 'a =
                let fun search nil = raise lookup
                        | search((k,item)::entries) =
                        if key=k then item
                        else if key<k then raise lookup
                        else search entries
                in search entrylist
        end

        fun enter (dict entrylist)
                (newentry as (key, item:'a)) : 'a dictionary =
                let fun update nil = [ newentry ]
                        | update ((entry as (k,_))::entries) =
                        if key=k then newentry::entries
                        else if key<k then newentry::(entry::entries)
                        else entry::update entries
                in dict(update entrylist)
        end

        fun listofdict (dict entrylist) : (string * 'a) list =
                let fun dump nil = nil
                        | dump (hd::tl) = hd::dump(tl)
                in dump entrylist
        end
end
```

```
fun main(s) =
let val flag = ref true in
        yyin := open_ibuf(s);
        while !flag do yylex()
        handle eof with _ => flag := false;
        close_ibuf(!yyin)
end;
```

```
val SymTab = ref nulldict;

fun GetExp () : exp =

        let val rec optional = fn e => ALT(EPS,e)

        and newline = fn () => let val c = array(128,false) in
                update(c,10,true); c
                end

        and endline = fn e => trail(e,CLASS(newline(),0))

        and trail = fn (e1,e2) => CAT(CAT(e1,TRAIL(0)),e2)

        and closure1 = fn e => CAT(e,CLOSURE(e))

        and repeat = fn (min,max,e) => let val rec rep = fn
                  (0,0) => EPS
                | (0,1) => ALT(e,EPS)
                | (0,i) => CAT(rep(0,1),rep(0,i-1))
                | (i,j) => CAT(e,rep(i-1,j-1))
        in rep(min,max)
        end

        and exp0 = fn () => case GetTok() of
                  CHARS(c) => exp1(CLASS(c,0))
                | LP => let val e = exp0() in if !NextTok = RP then
                  (AdvanceTok(); e) else raise syntax_error end
                | ID(name) => exp1(lookup(!SymTab)(name))
                | _ => raise syntax_error

        and exp1 = fn (e) => case !NextTok of
                  SEMI => e
                | ARROW => e
                | EOF => e
                | LP => exp2(e,exp0())
                | RP => e
                | t => (AdvanceTok(); case t of
                          QMARK => exp1(optional(e))
                        | STAR => exp1(CLOSURE(e))
                        | PLUS => exp1(closure1(e))
                        | CHARS(c) => exp2(e,CLASS(c,0))
                        | BAR => ALT(e,exp0())
                        | DOLLAR => endline(e)
                        | SLASH => trail(e,exp0())
                        | REPS(i,j) => exp1(repeat(i,j,e))
                        | ID(name) => exp2(e,lookup(!SymTab)(name))
                        | _ => raise syntax_error)

        and exp2 = fn (e1,e2) => case !NextTok of
                  SEMI => CAT(e1,e2)
                | ARROW => CAT(e1,e2)
                | EOF => CAT(e1,e2)
                | LP => exp2(CAT(e1,e2),exp0())
                | RP => CAT(e1,e2)
                | t => (AdvanceTok(); case t of
```

```
                                  QMARK => exp1(CAT(e1,optional(e2)))
                                | STAR => exp1(CAT(e1,CLOSURE(e2)))
                                | PLUS => exp1(CAT(e1,closure1(e2)))
                                | CHARS(c) => exp2(CAT(e1,e2),CLASS(c,0))
                                | BAR => ALT(CAT(e1,e2),exp0())
                                | DOLLAR => endline(CAT(e1,e2))
                                | SLASH => trail(CAT(e1,e2),exp0())
                                | REPS(i,j) => exp1(CAT(e1,repeat(i,j,e2)))
                                | ID(name) => exp2(CAT(e1,e2),lookup(!SymTab)(name))
                                | _ => raise syntax_error)
    in exp0()
    end;
```

```
fun leafdata(e:(int list * exp) list) =
        let val fp = array(!LeafNum + 1,nil)
        and leaf = array(!LeafNum + 1,EPS)
        and tcpairs = ref nil
        and trailmark = ref ~1;
        val rec add = fn
                  (nil,x) => ()
                | (hd::tl,x) => (update(fp,hd,union(fp sub hd,x));
                        add(tl,x))
        and moredata = fn
                  CLOSURE(e1) =>
                        (moredata(e1); add(lastpos(e1),firstpos(e1)))
                | ALT(e1,e2) => (moredata(e1); moredata(e2))
                | CAT(e1,e2) => (moredata(e1); moredata(e2);
                        add(lastpos(e1),firstpos(e2)))
                | CLASS(x,i) => update(leaf,i,CLASS(x,i))
                | TRAIL(i) => (update(leaf,i,TRAIL(i)); if !trailmark = ~1
                        then trailmark := i else ())
                | END(i) => (update(leaf,i,END(i)); if !trailmark <> ~1
                        then (tcpairs := (!trailmark,i)::(!tcpairs);
                        trailmark := ~1) else ())
                | _ => ()
        and makedata = fn
                  nil => ()
                | (_,x)::tl => (moredata(x);makedata(tl))
        in trailmark := ~1; makedata(e); (fp,leaf,!tcpairs)
        end;
```

```
abstype 'a idictionary =
        idict of (int * 'a)list

with
        val nullidict = idict nil

        exception ilookup : unit

        fun ilookup (idict entrylist:'a idictionary) (key:int) : 'a =
                let fun search nil = raise lookup
                        | search((k,item)::entries) =
                        if key=k then item
                        else if key<k then raise lookup
                        else search entries
                in search entrylist
        end

        fun ienter (idict entrylist)
                (newentry as (key, item:'a)) : 'a idictionary =
                let fun update nil = [ newentry ]
                        | update ((entry as (k,_))::entries) =
                        if key=k then newentry::entries
                        else if key<k then newentry::(entry::entries)
                        else entry::update entries
                in idict(update entrylist)
        end

        fun listofidict (idict entrylist) : (int * 'a) list =
                let fun dump nil = nil
                        | dump (hd::tl) = hd::dump(tl)
                in dump entrylist
        end
end
```

```
abstype ibuf =
        buf of instream * (string list ref)

with

        fun make_ibuf(s) = buf(s, ref nil)

        fun open_ibuf(f) = buf(open_in(f), ref nil)

        fun close_ibuf(buf(s,_)) = close_in(s)

        exception eof : unit

        fun getch(buf(s, bufptr)) =
                let fun get nil =
                        if end_of_stream(s) then raise eof
                        else input(s,1)
                | get l = let val c = hd(l)
                in bufptr := tl(!bufptr); c
                end
        in get (!bufptr)
        end

        fun ungetch(c:string, buf(s, bufptr)) =
                let fun put nil = ()
                | put x = let val t = tl(x)
                in bufptr := hd(x) :: !bufptr; put(t)
                end
        in put(rev(explode(c)))
        end

end;
```

```
val LeafNum = ref ~1;

fun renum(e : exp) : exp =
        let val rec label = fn
          EPS => EPS
        | CLASS(x,_) => CLASS(x,++LeafNum)
        | CLOSURE(e) => CLOSURE(label(e))
        | ALT(e1,e2) => ALT(label(e1),label(e2))
        | CAT(e1,e2) => CAT(label(e1),label(e2))
        | TRAIL(i) => TRAIL(++LeafNum)
        | END(i) => END(++LeafNum)
in label(e)
end;
```

```
use ["types.ml","misc.ml","dict.ml","idict.ml","input.ml","tok.ml","exp.ml",
     "states.ml","label.ml","parse.ml","make.ml","follow.ml","dfa.ml",
     "main.ml"];
```

```
fun LexGen(infile,outfile) =
(LexBuf := open_ibuf(infile);
LexOut := open_out(outfile);
StateNum := 1;
StateTab := enter(nulldict)("INITIAL",0);
LeafNum := ~1;
fcopy("skel.hd",!LexOut);
let val (rules,ends) = parse();
val (fins,trans,tctab,tcpairs) = makedfa(rules) in
        makefin(fins);
        maketrans(trans);
        maketc(tctab);
        maketcpairs(tcpairs);
        makebegin();
        fcopy("skel.mid",!LexOut); makeaccept(ends);
        fcopy("skel.tl",!LexOut);
        close_ibuf(!LexBuf); close_out(!LexOut)
end);
```

```
fun makebegin () : unit =
        let val rec fprint = fn x => output(!LexOut,x)
        and make = fn
                  nil => ()
                | (x,n)::y => (fprint("val "^x^" = "^itoa(n)^";\n");
                               make(y))
in fprint("\n(* start state definitions *)\n");
make(listofdict(!StateTab))
end;

fun makefin (fins:(int * (int list)) list) : unit =
        let val firste = ref true
        and firsti = ref true
        and col = ref 0;
        val rec fprint = fn x => output(!LexOut,x)
        and ckfirst = fn f => if !f then f := false else fprint(",")
        and ckcol = fn () => if ++col > 15 then
                (fprint("\n"); col := 0) else ()
        and makeEntry = fn
                  nil => ()
                | (_,x)::y => (ckfirst(firste); ckcol(); fprint("[");
                               firsti := true; makeItems(x);
                               fprint("]"); makeEntry(y))
        and makeItems = fn
                  nil => ()
                | hd::tl => (ckfirst(firsti); ckcol(); fprint(itoa(hd));
                             makeItems(tl))
in fprint("\n(* accepting states *)\nval yyfin = arrayoflist([\n");
makeEntry(fins);
fprint("\n]);")
end;

fun maketrans (trans:(int * ((int * int) list)) list) : unit =
        let val firste = ref true
        and firsti = ref true
        and col = ref 0;
        val rec fprint = fn x => output(!LexOut,x)
        and ckfirst = fn f => if !f then f := false else fprint(",")
        and ckcol = fn () => if ++col > 8 then
                (fprint("\n"); col := 0) else ()
        and makeEntry = fn
                  nil => ()
                | (_,x)::y => (ckfirst(firste); col := 0; fprint("\n[");
                               firsti := true; makeItems(x);
                               fprint("]"); makeEntry(y))
        and makeItems = fn
                  nil => ()
                | (x,y)::tl => (ckfirst(firsti); ckcol();
                               fprint("("^itoa(x)^","^itoa(y)^")");
                               makeItems(tl))
in fprint("\n(* state transitions *)\nval yytab = arrayoflist([");
makeEntry(trans);
fprint("\n]);")
end;

fun maketc (tcs:(int * (int list)) list) : unit =
```

```
        let val first = ref true
        and col = ref 0;
        val rec fprint = fn x => output(!LexOut,x)
        and ckfirst = fn () => if !first then first := false else fprint(",")
        and ckcol = fn () => if ++col > 8 then
                (fprint("\n"); col := 0) else ()
        and makeEntry = fn
                  nil => ()
                | (s,tlist)::tl => let val rec make = fn
                          nil => ()
                        | x::y => (ckfirst(); ckcol();
                                fprint("("^itoa(s)^","^itoa(x)^")"); make(y))
                        in (make(tlist); makeEntry(tl))
                        end
in fprint("\n(* trailing context mark states *)\nval yytcstates = [\n");
makeEntry(tcs);
fprint("\n];")
end;

fun maketcpairs (tcpairs:(int * int) list) : unit =
        let val first = ref true
        and col = ref 0;
        val rec fprint = fn x => output(!LexOut,x)
        and ckfirst = fn () => if !first then first := false else fprint(",")
        and ckcol = fn () => if ++col > 8 then
                (fprint("\n"); col := 0) else ()
        and makeEntry = fn
                  nil => ()
                | (t,e)::tl => (ckfirst(); ckcol();
                        fprint("("^itoa(t)^","^itoa(e)^")"); makeEntry(tl))
in fprint("\n(* trailing context/endmark pairs *)\nval yytcpairs = [\n");
makeEntry(tcpairs);
fprint("\n];")
end;

fun makeaccept (ends) : unit =
        let val first = ref true;
        val rec fprint = fn x => output(!LexOut,x)
        and startline = fn () => if !first then (first := false;
                fprint("  ")) else fprint("| ")
        and make = fn
                  nil => (startline();fprint("_ => raise LexError"))
                | (x,a)::y => (startline();
                        fprint(x^" => ("^a^")\n");
                        make(y))
in make(listofdict(ends));()
end;
```

```
fun isletter(x:string) = x>="a" andalso x<="z" orelse x>="A" andalso x<="Z";
fun isdigit(x:string) = x>="0" andalso x<="9";

fun atoi(s:string) : int =
        let val rec num = fn
        (x::y,n) => if isdigit(x) then num(y,10*n+ord(x)-48) else n
        | (_,n) => n
in num(explode(s),0)
end;

fun itoa(i:int) : string =
        let val rec cvt = fn
          (0,s) => if s=nil then "0" else implode(s)
        | (n,s) => cvt(n div 10,chr(n mod 10 + 48)::s)
in cvt(i,nil)
end;

val isaction = fn ACTION(_) => true | _ => false;
val isid = fn ID(_) => true | _ => false;
val isstate = fn STATE(_) => true | _ => false;

val action = fn ACTION(x) => x;
val id = fn ID(x) => x;
val state = fn STATE(x) => x;

val isend = fn END(_) => true | _ => false;
val istrail = fn TRAIL(_) => true | _ => false;
val isclass = fn CLASS(_) => true | _ => false;

val class = fn CLASS(a,_) => a;

fun union(a,b) = let val rec merge = fn
          (nil,nil,z) => z
        | (nil,el::more,z) => merge(nil,more,el::z)
        | (el::more,nil,z) => merge(more,nil,el::z)
        | (x::morex,y::morey,z) => if (x:int)=(y:int)
                then merge(morex,morey,x::z)
                else if x>y then merge(morex,y::morey,x::z)
                else merge(x::morex,morey,y::z)
        in merge(rev(a),rev(b),nil)
end;

val rec nullable = fn
          EPS => true
        | CLASS(_) => false
        | CLOSURE(_) => true
        | ALT(n1,n2) => nullable(n1) orelse nullable(n2)
        | CAT(n1,n2) => nullable(n1) andalso nullable(n2)
        | TRAIL(_) => true
        | END(_) => false

and firstpos = fn
          EPS => nil
        | CLASS(_,i) => [i]
        | CLOSURE(n) => firstpos(n)
        | ALT(n1,n2) => union(firstpos(n1),firstpos(n2))
```

```
        | CAT(n1,n2) => if nullable(n1) then union(firstpos(n1),firstpos(n2))
                        else firstpos(n1)
        | TRAIL(i) => [i]
        | END(i) => [i]

and lastpos = fn
          EPS => nil
        | CLASS(_,i) => [i]
        | CLOSURE(n) => lastpos(n)
        | ALT(n1,n2) => union(lastpos(n1),lastpos(n2))
        | CAT(n1,n2) => if nullable(n2) then union(lastpos(n1),lastpos(n2))
                        else lastpos(n2)
        | TRAIL(i) => [i]
        | END(i) => [i]
        ;

fun ++(x) : int = (x := !x + 1; !x);

fun fcopy(file, s2) =
let val s1 = open_in(file) in
        while not(end_of_stream(s1)) do
        output(s2,input(s1,512));
        close_in(s1)
end;

fun listofarray(a,n) =
let val i = ref(n-1) and l = ref nil
in while !i >= 0 do (l := (a sub !i)::(!l); i := !i - 1); !l
end;
```

```
exception parse_error : unit;

val LexOut = ref(std_out);

fun parse() : ((int list * exp) list * string dictionary) =
        let val Accept = ref nulldict;
        val rec ParseRtns = fn () => case getch(!LexBuf) of
                  "%" => let val c = getch(!LexBuf) in
                        if c="%" then ()
                        else (output(!LexOut,"%" ^ c); ParseRtns())
                        end
                | c => (output(!LexOut,c); ParseRtns())
        and ParseDefs = fn () =>
                (LexState:=0; AdvanceTok(); case !NextTok of
                  LEXMARK => ()
                | LEXSTATES => (AdvanceTok(); while isid(!NextTok) do
                        (StateTab := enter(!StateTab)(id(!NextTok),++StateNum);
                        ++StateNum; AdvanceTok());
                        if !NextTok = SEMI then ParseDefs()
                        else raise syntax_error)
                | ID(x) => (LexState:=1; AdvanceTok(); if GetTok() = ASSIGN
                          then (SymTab := enter(!SymTab)(x,GetExp());
                                if !NextTok = SEMI then ParseDefs()
                                else raise syntax_error)
                        else raise syntax_error)
                | _ => raise syntax_error)
        and ParseRules =
                fn rules => (LexState:=1; AdvanceTok(); case !NextTok of
                  LEXMARK => rules
                | EOF => rules
                | _ => let val s = GetStates()
                        and e = renum(CAT(GetExp(),END(0)))
                        in if !NextTok = ARROW then
                                (LexState:=2; AdvanceTok();
                                let val act = GetTok() in if isaction(act)
                                        andalso !NextTok = SEMI then
                                        (Accept := enter(!Accept)
                                                (itoa(!LeafNum),action(act));
                                        ParseRules((s,e)::rules))
                                        else raise syntax_error
                                end)
                                else raise syntax_error
                        end)
in ParseRtns(); ParseDefs(); (ParseRules(nil),!Accept)
handle syntax_error with _ =>
        (print "\nSyntax error in line ";
        print(!LineNum);
        print "\n"; raise syntax_error)
end;
```

```
val StateTab = ref(enter(nulldict)("INITIAL",0));

val StateNum = ref 1;

fun GetStates () : int list =

        let val states = ref nil;

        val rec add = fn
                  nil => ()
                | x::y => (states := union([lookup(!StateTab)(x)],!states);
                           add(y))

        and addall = fn () => let val i = ref 0
        in while !i <= !StateNum do (states := union([!i],!states);
        i := !i + 2)
        end

        and incall = fn () => let val rec inc = fn
                  nil => nil
                | x::y => (x+1)::inc(y)
        in states := inc(!states)
        end

        and addincs = fn () => let val rec dual = fn
                  nil => nil
                | x::y => x::(x+1)::dual(y)
        in states := dual(!states)
        end

        and start = fn () => if isstate(!NextTok) then
                (add(state(!NextTok)); LexState := 1; AdvanceTok())
                else addall()

        and cknl = fn () => if !NextTok = CARAT then (incall(); LexState := 1;
                AdvanceTok()) else addincs()

in start(); cknl(); !states
end;
```

```
exception syntax_error : unit ; (* error in user's input file *)

exception lex_error : unit ; (* unexpected error in lexer *)

val LexBuf = ref(make_ibuf(std_in));
val LexState = ref 0;
val LineNum = ref 1;
val NextTok = ref BOF;
val inquote = ref false;

fun AdvanceTok () : unit =

        let val rec skipws = fn () => case nextch() of
                  " " => skipws()
                | "\t" => skipws()
                | "\n" => skipws()
                | x => x

        and nextch = fn () => let val c = getch(!LexBuf) in
                if c = "\n" then LineNum := !LineNum + 1 else (); c
                end

        and escaped = fn () => case nextch() of
                  "b" => "\008"
                | "n" => "\n"
                | "t" => "\t"
                | x => x

        and onechar = fn x => let val c = array(128,false) in
                update(c,ord(x),true); CHARS(c)
                end

        in case !LexState of 0 => let val makeTok = fn () =>
                case skipws() of
                        (* Lex % operators *)
                  "%" => (case nextch() of
                          "%" => LEXMARK
                        | "s" => LEXSTATES
                        | "S" => LEXSTATES
                        | _ => raise syntax_error)
                        (* semicolon (for end of LEXSTATES) *)
                | ";" => SEMI
                        (* anything else *)
                | ch => if isletter(ch) then let val rec getID =
                        fn (matched) => let val x = nextch() in
                                if isletter(x) orelse isdigit(x) then
                                getID(matched^x)
                                else (ungetch(x,!LexBuf); matched)
                        end
                in ID(getID(ch))
                end
                else raise syntax_error
        in NextTok := makeTok()
        end
        | 1 => let val rec makeTok = fn () =>
                if !inquote then case nextch() of
```

```
                (* inside quoted string *)
          "\\" => onechar(escaped())
        | "\"" => (inquote := false; makeTok())
        | x => onechar(x)
      else case skipws() of
                (* single character operators *)
          "?" => QMARK
        | "*" => STAR
        | "+" => PLUS
        | "|" => BAR
        | "(" => LP
        | ")" => RP
        | "^" => CARAT
        | "$" => DOLLAR
        | "/" => SLASH
        | ";" => SEMI
        | "." => let val c = array(128,true) in
                         update(c,10,false); CHARS(c)
                 end
                (* assign and arrow *)
        | "=" => let val c = nextch() in
                 if c=">" then ARROW else (ungetch(c,!LexBuf); ASSIGN)
      end
                (* character set *)
        | "[" => let val rec classch = fn () => let val x = skipws()
                         in if x="\\" then escaped() else x
                         end;
                 val first = classch();
                 val flag = (first<>"^");
                 val c = array(128,not flag);
                 val rec add = fn x => if x="" then ()
                         else update(c,ord(x),flag)
                 and range = fn (x,y) =>
                         if x>y then raise syntax_error
                         else let val i = ref(ord(x)) and j = ord(y)
                         in while !i<=j do (add(chr(!i)); i := !i + 1)
                         end
                 and getClass = fn (last) => case classch() of
                           "]" => (add(last); c)
                         | "-" => if last<>"" then
                         let val x = classch() in
                                 if x="]" then (add(last);add("-"); c)
                                 else (range(last,x);getClass(""))
                         end
                         else getClass("-")
                         | x => (add(last); getClass(x))
      in CHARS(getClass(if first="^" then "" else first))
      end
                (* Start States specification *)
        | "<" => let val rec get_state = fn (prev,matched) =>
                 case nextch() of
                   ">" => matched::prev
                 | "," => get_state(matched::prev,"")
                 | x => if isletter(x) then get_state(prev,matched^x)
                         else raise syntax_error
      in STATE(get_state(nil,""))
```

```
                end
                        (* {id} or repititions *)
                | "{" => let val ch = nextch() in if isletter(ch) then
                        let val rec getID = fn (matched) =>
                        case nextch() of
                          "}" => matched
                        | x => if isletter(x) orelse isdigit(x) then
                                getID(matched^x)
                                else raise syntax_error
                        in ID(getID(ch))
                        end
                        else if isdigit(ch) then let val rec get_r = fn
                                (matched,r1) => case nextch() of
                                  "}" => let val n = atoi(matched) in
                                        if r1 = ~1 then (n,n) else (r1,n)
                                end
                                | "," => if r1 = ~1 then get_r("",atoi(matched))
                                        else raise syntax_error
                                | x => if isdigit(x) then get_r(matched^x,r1)
                                else raise syntax_error
                        in REPS(get_r(ch,0))
                        end
                        else raise syntax_error
                end
                        (* Lex % operators *)
                | "%" => if nextch() = "%" then LEXMARK else raise syntax_error
                        (* backslash escape *)
                | "\\" => onechar(escaped())
                        (* start quoted string *)
                | "\"" => (inquote := true; makeTok())
                        (* anything else *)
                | ch => onechar(ch)
        in NextTok := makeTok()
        end
        | 2 => let val MakeTok = fn _ => case skipws() of
                  "(" => let val lpct = ref 0 in
                        let val rec GetAct = fn x => case getch(!LexBuf) of
                                  "(" => (lpct:= !lpct+1; GetAct(x^"("))
                                | ")" => if !lpct = 0 then x
                                        else (lpct:= !lpct-1; GetAct(x^")"))
                                | y => GetAct(x^y)
                        in ACTION(GetAct(""))
                        end
                end
                | ";" => SEMI
                | _ => raise syntax_error
        in NextTok := MakeTok()
        end
        | _ => raise lex_error
end
handle eof with _ => NextTok := EOF ;

fun GetTok (_:unit) : token =
        let val t = !NextTok in AdvanceTok(); t
        end;
```

```
datatype token = CHARS of bool array | QMARK | STAR | PLUS | BAR
       | LP | RP | CARAT | DOLLAR | SLASH | STATE of string list
       | REPS of int * int | ID of string | ACTION of string
       | BOF | EOF | ASSIGN | SEMI | ARROW | LEXMARK | LEXSTATES
       ;

datatype exp = EPS | CLASS of bool array * int | CLOSURE of exp
       | ALT of exp * exp | CAT of exp * exp | TRAIL of int
       | END of int
       ;
```

```
(* Lexical Analyzer produced by LexGen *)

exception LexError : unit;

abstype ibuf =
        buf of instream * (string list ref)

with

        fun make_ibuf(s) = buf(s, ref nil)

        fun open_ibuf(f) = buf(open_in(f), ref nil)

        fun close_ibuf(buf(s,_)) = close_in(s)

        exception eof : unit

        fun getch(buf(s, bufptr)) =
                let fun get nil =
                        if end_of_stream(s) then raise eof
                        else input(s,1)
                | get l = let val c = hd(l)
                in bufptr := tl(!bufptr); c
                end
        in get (!bufptr)
        end

        fun ungetch(c:string, buf(s, bufptr)) =
                let fun put nil = ()
                | put x = let val t = tl(x)
                in bufptr := hd(x) :: !bufptr; put(t)
                end
        in put(rev(explode(c)))
        end

end;

val yyin = ref(make_ibuf(std_in));      (* default input buffer *)
val yyout = ref(std_out);               (* default output buffer *)

(* global variables/functions used by application *)

val yytext = ref "";                    (* text matched by expression *)
val yyprev = ref "\n";                  (* used for matching ^exp at beg-of-line *)
val yynext = ref "\n";                  (* used with yyprev *)
val yymorfg = ref false;                (* if true, yylex() will not flush yytext *)
val yylineno = ref 1;                   (* line number of input *)

fun yyinput () = let val c = getch(!yyin)
        in yyprev := !yynext; yynext := c; if c="\n" then
        yylineno := !yylineno + 1 else (); c
end
fun yyunput(c) =
        (if c="\n" then yylineno := !yylineno - 1 else (); ungetch(c,!yyin));
fun yyoutput(c) = output(!yyout,c);
fun yyless(n) = while length(!yytext) > n do
```

```
        let val l = rev(explode(!yytext)) in
                yyunput(hd(l));
                yytext := implode(rev(tl(l)))
        end
fun yywrap () = false;
val yybegin = ref 0;
val yyends = ref nil;
val yytcs = ref nil;

(* application's routines follow *)
```

```
fun yyaccept(states: (int * int) list) =
let val rec ECHO = fn () => (yyoutput(!yytext); yylex())
and yyacc = fn
          nil => (yyless(0); yytext := yyinput(); ECHO())
        | (yyhds,yyhdl)::yytl => let val REJECT = fn () => yyacc(yytl)
and BEGIN = fn x => yybegin := x
in yymorfg := false; yyless(yyhdl); case yyhds of

(* application's actions follow *)
```

```
(* end of application's actions *)
        end
in yyacc(states)
end

and yytran(s: int) =
let val yychar = yyinput();
val yycharnum = ord(yychar);
exception notcmatch : unit;
val rec yyckend = fn
          nil => ()
        | x::y => (yyends := (x,yyaccleng(x))::(!yyends)
                handle notcmatch with _ => (); yyckend(y))
and yyaccleng = fn x =>
        let val rec yyal = fn
                  nil => raise notcmatch
                | (e,l)::tl => if e = x then l else yyal(tl)
        in if yytcmatch(x) then yyal(!yytcs) else length(!yytext)
        end
and yytcmatch = fn x =>
        let val rec yym = fn
                  nil => false
                | (t,e)::tl => if e = x then true else yym(tl)
        in yym(yytcpairs)
        end
and yycktc = fn
          nil => ()
        | (hds,hdt)::tl => if hds > s then () else (if hds = s then
                yytcs := (yytcend(hdt),length(!yytext))::(!yytcs) else ();
                yycktc(tl))
and yytcend = fn trail =>
        let val rec yytce = fn
                  nil => raise LexError
                | (t,e)::tl => if t = trail then e else yytce(tl)
        in yytce(yytcpairs)
        end
and yynext = fn
          nil => (yyunput(yychar); yyaccept(!yyends))
        | (c,s2)::tl => if c=yycharnum then (yytext := !yytext^yychar;
                yytran(s2)) else if c<yycharnum then (yyunput(yychar);
                yyaccept(!yyends)) else yynext(tl)
in if length(!yytext) > 0 then (yycktc(yytcstates); yyckend(yyfin sub s))
        else (); yynext(yytab sub s)
handle eof with _ => if length(!yytext) = 0 then raise eof
        else yyaccept(!yyends)
end

and yymore () = (yymorfg := true; yylex())

and yylex () =
(yytcs := nil; yyends := nil; if not(!yymorfg) then yytext := "" else ();
yytran(if !yyprev = "\n" then !yybegin + 1 else !yybegin)
handle eof with _ => if yywrap() then yylex() else raise eof);
```