# *L  GRAMMAR*

```
[pattern]   optional
{pattern}   zero or more repetitions
(pattern)   grouping
pat1 | pat2 choice
pat<pat'>   difference---elements generated by pat
            except those generated by pat'
```

```
Exp   → infixexp

infixexp → lexp

lexp  →      \ apat1 … apatn -> exp      (lambda abstraction, n ≥ 1)
      |      let decls in exp     (let expression)
      |      if exp [;] then exp [;] else exp        (conditional)
      |      case exp of { alts }      (case expression)
      |      do { stmts }       (do expression)
      |      fexp


fexp  →      fexp aexp (function application)
      |      aexp

aexp  →      qvar      (variable)
      |      gcon      (general constructor)
      |      literal
      |      ( exp )          (parenthesized expression)
      |      ( exp1 , … , expk )         (tuple, k ≥ 2)
      |      [ exp1 , … , expk ]         (list, k ≥ 1)


qvar → qvarid |(qvarsym) (qualified variable)

gcon → ()
     | []
     | (,{,})
     | qcon



stmts →      stmt1 … stmtn exp [;]      (n ≥ 0)

stmt  →      exp ;
      |      pat <- exp ;
      |      let decls ;
      |            ;
```

- ## *Declarations' syntax*

```
decls →    { decl1 ; … ; decln }      (n ≥ 0)

decl   →   (funlhs | pat) rhs


funlhs →   var apat { apat } ( function binding)
       |   pat varop pat  (pattern binding)
       |   ( funlhs ) apat { apat }

rhs    →   = exp

var    →   varid | ( varsym )     (variable)
```

### *(For function definition)*

A function binding binds a variable to a function value. The general form of a function binding for variable x is:

```
x     p11 … p1k   match1
…
x     pn1 … pnk   matchn
```

where each pij is a pattern (an argument of the function), and where each matchi is of the general form:

```
= ei
```

- - or with guards

```
| gsi1 = ei1
…
| gsimi = eimi
where { declsi }
```

### *(Pattern Binding)*

A pattern binding binds variables to values. A simple pattern binding has form p = e. The pattern p is matched "lazily" as an irrefutable pattern, as if there were an implicit ~ in front of it.

The general form of a pattern binding is p match, where a match is the same structure as for function bindings above; in other words, a pattern binding is:

```
p     | gs1 = e1
      | gs2 = e2
      …
      | gsm = em
      where { decls }
```

- ## *Patterns' syntax:*

```
pat   →    lpat : pat     (infix constructor)
      |    lpat

lpat  →    apat
```

```
        |       - (integer | float)         (negative literal)
        |       gcon apat1 … apatk          (arity gcon  =  k, k ≥ 1)

apat  →    gcon        (arity gcon  =  0)
        |       qcon { fpat1 , … , fpatk }        (labeled pattern, k ≥ 0)
        |       literal
        |       _           (wildcard)
        |       ( pat )         (parenthesized pattern)
        |       ( pat1 , … , patk )          (tuple pattern, k ≥ 2)
        |       [ pat1 , … , patk ]          (list pattern, k ≥ 1)
        |       ~ apat         (irrefutable pattern)

fpat  →    qvar = pat



qcon  ->    qconid | ( gconsym )    (qualified constructor)

qconop      ->     gconsym | `qconid `     (qualified constructor operator)

gconsym     ->    : | qconsym

qconid      ->    [ modid . ] conid

conid       ->    large {small | large | digit | ' }

qconsym     ->    [ modid . ] consym

consym      ->    (: {symbol | :})<reservedop>

reservedop  ->    .. | : | :: | = | \ | | | <- | -> | @ | ~ | =>

symbol      ->    ascSymbol | uniSymbol<special | _ | : | " | '>

ascSymbol   ->    ! | # | $ | % | & | * | + | . | / | < | = | > | ? | @ |
                  \ | ^ | | | - | ~
uniSymbol   ->    any Unicode symbol or punctuation
```