JFP **13** (1): 17–38, January 2003. © 2003 Cambridge University Press
DOI: 10.1017/S0956796803000510 Printed in the United Kingdom

Chapter 3

Expressions

In this chapter, we describe the syntax and informal semantics of Haskell *expressions*, including their translations into the Haskell kernel, where appropriate. Except in the case of `let` expressions, these translations preserve both the static and dynamic semantics. Free variables and constructors used in these translations always refer to entities defined by the `Prelude`. For example, "`concatMap`" used in the translation of list comprehensions (Section 3.11) means the `concatMap` defined by the `Prelude`, regardless of whether or not the identifier "`concatMap`" is in scope where the list comprehension is used, and (if it is in scope) what it is bound to.

In the syntax that follows, there are some families of nonterminals indexed by precedence levels (written as a superscript). Similarly, the nonterminals op, $varop$, and $conop$ may have a double index: a letter l, r, or n for left-, right- or non-associativity and a precedence level. A precedence-level variable i ranges from 0 to 9; an associativity variable a varies over $\{l,\ r,\ n\}$. For example

$$aexp \quad \rightarrow \quad \texttt{(}\ exp^{i+1}\ qop^{(a,i)}\ \texttt{)}$$

actually stands for 30 productions, with 10 substitutions for i and 3 for a.

exp	$\rightarrow$	exp^{0} `::` $[context$ `=>`$]$ $type$	(expression type signature)
	$\mid$	exp^{0}	
exp^{i}	$\rightarrow$	exp^{i+1} $[qop^{(\mathrm{n},i)}\ exp^{i+1}]$	
	$\mid$	$lexp^{i}$	
	$\mid$	$rexp^{i}$	
$lexp^{i}$	$\rightarrow$	$(lexp^{i} \mid exp^{i+1})\ qop^{(\mathrm{l},i)}\ exp^{i+1}$	
$lexp^{6}$	$\rightarrow$	`-` exp^{7}	

$rexp^i$	$\rightarrow$	$exp^{i+1}\ qop^{(\mathrm{r},i)}\ (rexp^i \mid exp^{i+1})$	
exp^{10}	$\rightarrow$	`\` $apat_1\ \ldots\ apat_n$ `->` exp	(lambda abstraction, $n \geq 1$)
	$\mid$	`let` $decls$ `in` exp	(let expression)
	$\mid$	`if` exp `then` exp `else` exp	(conditional)
	$\mid$	`case` exp `of {` $alts$ `}`	(case expression)
	$\mid$	`do {` $stmts$ `}`	(do expression)
	$\mid$	$fexp$	
$fexp$	$\rightarrow$	$[fexp]\ aexp$	(function application)
$aexp$	$\rightarrow$	$qvar$	(variable)
	$\mid$	$gcon$	(general constructor)
	$\mid$	$literal$	
	$\mid$	`(` exp `)`	(parenthesized expression)
	$\mid$	`(` exp_1 `,` $\ldots$ `,` exp_k `)`	(tuple, $k \geq 2$)
	$\mid$	`[` exp_1 `,` $\ldots$ `,` exp_k `]`	(list, $k \geq 1$)
	$\mid$	`[` exp_1 [`,` exp_2] `..` [exp_3] `]`	(arithmetic sequence)
	$\mid$	`[` $exp \mid qual_1$ `,` $\ldots$ `,` $qual_n$ `]`	(list comprehension, $n \geq 1$)
	$\mid$	`(` $exp^{i+1}\ qop^{(a,i)}$ `)`	(left section)
	$\mid$	`(` $lexp^i\ qop^{(l,i)}$ `)`	(left section)
	$\mid$	`(` $qop^{(a,i)}_{\langle - \rangle}\ exp^{i+1}$ `)`	(right section)
	$\mid$	`(` $qop^{(r,i)}_{\langle - \rangle}\ rexp^i$ `)`	(right section)
	$\mid$	$qcon$ `{` $fbind_1$ `,` $\ldots$ `,` $fbind_n$ `}`	(labeled construction, $n \geq 0$)
	$\mid$	$aexp_{\langle qcon \rangle}$ `{` $fbind_1$ `,` $\ldots$ `,` $fbind_n$ `}`	(labeled update, $n \geq 1$)

Expressions involving infix operators are disambiguated by the operator's fixity (see Section 4.4.2). Consecutive unparenthesized operators with the same precedence must both be either left or right associative to avoid a syntax error. Given an unparenthesized expression "$x\ qop^{(a,i)}\ y\ qop^{(b,j)}\ z$", parentheses must be added around either "$x\ qop^{(a,i)}\ y$" or "$y\ qop^{(b,j)} z$" when $i = j$ unless $a = b = \mathrm{l}$ or $a = b = \mathrm{r}$.

Negation is the only prefix operator in Haskell; it has the same precedence as the infix `-` operator defined in the Prelude (see Section 4.4.2, Figure 4.1).

The grammar is ambiguous regarding the extent of lambda abstractions, let expressions, and conditionals. The ambiguity is resolved by the meta-rule that each of these constructs extends as far to the right as possible.

Sample parses are shown below.

This	Parses as
`f x + g y`	`(f x) + (g y)`
`- f x + y`	`(- (f x)) + y`
`let { ... } in x + y`	`let { ... } in (x + y)`
`z + let { ... } in x + y`	`z + (let { ... } in (x + y))`
`f x y :: Int`	`(f x y) :: Int`
`\ x -> a+b :: Int`	`\ x -> ((a+b) :: Int)`

A note about parsing. Expressions that involve the interaction of fixities with the let/lambda meta-rule may be hard to parse. For example, the expression

```
let x = True in x == x == True
```

cannot possibly mean

```
let x = True in (x == x == True)
```

because `(==)` is a non-associative operator; so the expression must parse thus:

```
(let x = True in (x == x)) == True
```

However, implementations may well use a post-parsing pass to deal with fixities, so they may well incorrectly deliver the former parse. Programmers are advised to avoid constructs whose parsing involves an interaction of (lack of) associativity with the let/lambda meta-rule.

For the sake of clarity, the rest of this section shows the syntax of expressions without their precedences.

3.1 Errors

Errors during expression evaluation, denoted by $\perp$, are indistinguishable by a Haskell program from non-termination. Since Haskell is a non-strict language, all Haskell types include $\perp$. That is, a value of any type may be bound to a computation that, when demanded, results in an error. When evaluated, errors cause immediate program termination and cannot be caught by the user. The Prelude provides two functions to directly cause such errors:

```
error     :: String -> a
undefined :: a
```

A call to `error` terminates execution of the program and returns an appropriate error indication to the operating system. It should also display the string in some system-dependent manner. When `undefined` is used, the error message is created by the compiler.

Translations of Haskell expressions use `error` and `undefined` to explicitly indicate where execution time errors may occur. The actual program behavior when an error occurs is up to the implementation. The messages passed to the `error` function in these translations are only suggestions; implementations may choose to display more or less information when an error occurs.

3.2 Variables, Constructors, Operators, and Literals

aexp	$\rightarrow$	*qvar*	(variable)
	$\mid$	*gcon*	(general constructor)
	$\mid$	*literal*	

gcon	$\rightarrow$	`()`	
	$\mid$	`[]`	
	$\mid$	`(,{,})`	
	$\mid$	*qcon*	
var	$\rightarrow$	*varid* $\mid$ `(` *varsym* `)`	(variable)
qvar	$\rightarrow$	*qvarid* $\mid$ `(` *qvarsym* `)`	(qualified variable)
con	$\rightarrow$	*conid* $\mid$ `(` *consym* `)`	(constructor)
qcon	$\rightarrow$	*qconid* $\mid$ `(` *gconsym* `)`	(qualified constructor)
varop	$\rightarrow$	*varsym* $\mid$ `` ` `` *varid* `` ` ``	(variable operator)
qvarop	$\rightarrow$	*qvarsym* $\mid$ `` ` `` *qvarid* `` ` ``	(qualified variable operator)
conop	$\rightarrow$	*consym* $\mid$ `` ` `` *conid* `` ` ``	(constructor operator)
qconop	$\rightarrow$	*gconsym* $\mid$ `` ` `` *qconid* `` ` ``	(qualified constructor operator)
op	$\rightarrow$	*varop* $\mid$ *conop*	(operator)
qop	$\rightarrow$	*qvarop* $\mid$ *qconop*	(qualified operator)
gconsym	$\rightarrow$	`:` $\mid$ *qconsym*	

Haskell provides special syntax to support infix notation. An *operator* is a function that can be applied using infix syntax (Section 3.4), or partially applied using a *section* (Section 3.5).

An *operator* is either an *operator symbol*, such as `+` or `$$`, or is an ordinary identifier enclosed in grave accents (backquotes), such as `` `op` ``. For example, instead of writing the prefix application `op x y`, one can write the infix application ``x `op` y``. If no fixity declaration is given for `` `op` `` then it defaults to highest precedence and left associativity (see Section 4.4.2).

Dually, an operator symbol can be converted to an ordinary identifier by enclosing it in parentheses. For example, `(+) x y` is equivalent to `x + y`, and `foldr (*) 1 xs` is equivalent to `foldr (\x y -> x*y) 1 xs`.

Special syntax is used to name some constructors for some of the built-in types, as found in the production for *gcon* and *literal*. These are described in Section 6.1.

An integer literal represents the application of the function `fromInteger` to the appropriate value of type `Integer`. Similarly, a floating point literal stands for an application of `fromRational` to a value of type `Rational` (that is, `Ratio Integer`).

Translation: The integer literal i is equivalent to `fromInteger` i, where `fromInteger` is a method in class `Num` (see Section 6.4.1).
The floating point literal f is equivalent to `fromRational` (n `Ratio.%` d), where `fromRational` is a method in class `Fractional` and `Ratio.%` constructs a rational from two integers, as defined in the `Ratio` library. The integers n and d are chosen so that $n/d = f$.

3.3 Curried Applications and Lambda Abstractions

fexp	$\rightarrow$	[*fexp*] *aexp*	(function application)
exp	$\rightarrow$	`\` $apat_1$ ... $apat_n$ `->` *exp*	(lambda abstraction, $n \geq 1$)

Function application is written $e_1\ e_2$. Application associates to the left, so the parentheses may be omitted in `(f x) y`. Because e_1 could be a data constructor, partial applications of data constructors are allowed.

Lambda abstractions are written `\` p_1 ... p_n `->` e, where the p_i are *patterns*. An expression such as `\x:xs->x` is syntactically incorrect; it may legally be written as `\(x:xs)->x`.

The set of patterns must be *linear* – no variable may appear more than once in the set.

Translation: The following identity holds:

`\` p_1 ... p_n `->` e = `\` x_1 ... x_n `-> case (`x_1`,` ...`,` x_n`) of (`p_1`,` ...`,` p_n`) ->` e

where the x_i are new identifiers.

Given this translation combined with the semantics of case expressions and pattern matching described in Section 3.17.3, if the pattern fails to match, then the result is $\perp$.

3.4 Operator Applications

exp	$\rightarrow$	exp_1 *qop* exp_2	
	|	`-` *exp*	(prefix negation)
qop	$\rightarrow$	*qvarop* | *qconop*	(qualified operator)

The form e_1 qop e_2 is the infix application of binary operator qop to expressions e_1 and e_2.

The special form `-`e denotes prefix negation, the only prefix operator in Haskell, and is syntax for `negate` (e). The binary `-` operator does not necessarily refer to the definition of `-` in the Prelude; it may be rebound by the module system. However, unary `-` will always refer to the `negate` function defined in the Prelude. There is no link between the local meaning of the `-` operator and unary negation.

Prefix negation has the same precedence as the infix operator `-` defined in the Prelude (see Table 4.1, p. 57). Because `e1-e2` parses as an infix application of the binary operator `-`, one must write `e1(-e2)` for the alternative parsing. Similarly, `(-)` is syntax for `(\ x y -> x-y)`, as with any infix operator, and does not denote `(\ x -> -x)` – one must use `negate` for that.

Translation: The following identities hold:

$$\begin{array}{lcl} e_1\ op\ e_2 & = & (op)\ e_1\ e_2 \\ \texttt{-}e & = & \texttt{negate}\ (e) \end{array}$$

3.5 Sections

$$\begin{array}{llll} aexp & \rightarrow & \texttt{(}\ exp^{i+1}\ qop^{(a,i)}\ \texttt{)} & \text{(left section)} \\ & | & \texttt{(}\ lexp^{i}\ qop^{(l,i)}\ \texttt{)} & \text{(left section)} \\ & | & \texttt{(}\ qop^{(a,i)}_{\langle \texttt{-} \rangle}\ exp^{i+1}\ \texttt{)} & \text{(right section)} \\ & | & \texttt{(}\ qop^{(r,i)}_{\langle \texttt{-} \rangle}\ rexp^{i}\ \texttt{)} & \text{(right section)} \end{array}$$

Sections are written as `(` op e `)` or `(` e op `)`, where op is a binary operator and e is an expression. Sections are a convenient syntax for partial application of binary operators.

Syntactic precedence rules apply to sections as follows. `(`op e`)` is legal if and only if `(x` op e`)` parses in the same way as `(x` op `(`e`))`; and similarly for `(`e op`)`. For example, `(*a+b)` is syntactically invalid, but `(+a*b)` and `(*(a+b))` are valid. Because `(+)` is left associative, `(a+b+)` is syntactically correct, but `(+a+b)` is not; the latter may legally be written as `(+(a+b))`. As another example, the expression

```
    (let n = 10 in n +)
```

is invalid because, by the let/lambda meta-rule (Section 3), the expression

```
    (let n = 10 in n + x)
```

parses as

```
    (let n = 10 in (n + x))
```

rather than

```
((let n = 10 in n) + x)
```

Because `-` is treated specially in the grammar, `(-` *exp* `)` is not a section, but an application of prefix negation, as described in the preceding section. However, there is a `subtract` function defined in the Prelude such that `(subtract` *exp* `)` is equivalent to the disallowed section. The expression `(+ (-` *exp* `))` can serve the same purpose.

Translation: The following identities hold:

$$\begin{array}{lcl} (\ op\ e\ ) & = & \backslash\ x\ \texttt{->}\ x\ op\ e \\ (\ e\ op\ ) & = & \backslash\ x\ \texttt{->}\ e\ op\ x \end{array}$$

where op is a binary operator, e is an expression, and x is a variable that does not occur free in e.

3.6 Conditionals

$$exp \quad \rightarrow \quad \texttt{if}\ exp_1\ \texttt{then}\ exp_2\ \texttt{else}\ exp_3$$

A *conditional expression* has the form `if` e_1 `then` e_2 `else` e_3 and returns the value of e_2 if the value of e_1 is `True`, e_3 if e_1 is `False`, and $\perp$ otherwise.

Translation: The following identity holds:

$$\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \quad = \quad \texttt{case}\ e_1\ \texttt{of \{ True ->}\ e_2\ \texttt{; False ->}\ e_3\ \texttt{\}}$$

where `True` and `False` are the two nullary constructors from the type `Bool`, as defined in the Prelude. The type of e_1 must be `Bool`; e_2 and e_3 must have the same type, which is also the type of the entire conditional expression.

3.7 Lists

$$\begin{array}{lcll} exp & \rightarrow & exp_1\ qop\ exp_2 & \\ aexp & \rightarrow & \texttt{[}\ exp_1\ \texttt{,}\ \ldots\ \texttt{,}\ exp_k\ \texttt{]} & (k \geq 1) \\ & | & gcon & \\ gcon & \rightarrow & \texttt{[]} & \\ & | & qcon & \end{array}$$

$qcon$	$\rightarrow$	`(` $gconsym$ `)`
qop	$\rightarrow$	$qconop$
$qconop$	$\rightarrow$	$gconsym$
$gconsym$	$\rightarrow$	`:`

Lists are written `[`e_1`,` ... `,` e_k`]`, where $k \geq 1$. The list constructor is `:`, and the empty list is denoted `[]`. Standard operations on lists are given in the Prelude (see Section 6.1.3, and Chapter 8 notably Section 8.2).

Translation: The following identity holds:

$$[e_1, \ldots, e_k] \quad = \quad e_1 : (e_2 : (\ldots (e_k : [])))$$

where `:` and `[]` are constructors for lists, as defined in the Prelude (see Section 6.1.3). The types of e_1 through e_k must all be the same (call it t), and the type of the overall expression is `[`t`]` (see Section 4.1.2).

The constructor "`:`" is reserved solely for list construction; like `[]`, it is considered part of the language syntax, and cannot be hidden or redefined. It is a right-associative operator, with precedence level 5 (Section 4.4.2).

3.8 Tuples

$aexp$	$\rightarrow$	`(` exp_1 `,` ... `,` exp_k `)`	$(k \geq 2)$
	$\vert$	$qcon$	
$qcon$	$\rightarrow$	`(,{,})`	

Tuples are written `(`e_1`,` ... `,` e_k`)`, and may be of arbitrary length $k \geq 2$. The constructor for an n-tuple is denoted by `(,...,)`, where there are $n - 1$ commas. Thus `(a,b,c)` and `(,,) a b c` denote the same value. Standard operations on tuples are given in the Prelude (see Section 6.1.4 and Chapter 8).

Translation: `(`e_1`,` ... `,` e_k`)` for $k \geq 2$ is an instance of a k-tuple as defined in the Prelude, and requires no translation. If t_1 through t_k are the types of e_1 through e_k, respectively, then the type of the resulting tuple is `(`t_1`,` ... `,` t_k`)` (see Section 4.1.2).

3.9 Unit Expressions and Parenthesized Expressions

aexp	$\rightarrow$	*gcon*
	$\vert$	`(` *exp* `)`
gcon	$\rightarrow$	`()`

The form `(` e `)` is simply a *parenthesized expression*, and is equivalent to e. The *unit expression* `()` has type `()` (see Section 4.1.2). It is the only member of that type apart from $\perp$, and can be thought of as the "nullary tuple" (see Section 6.1.5).

Translation: `(` e `)` is equivalent to e.

3.10 Arithmetic Sequences

aexp $\rightarrow$ `[` exp_1 [`,` exp_2] `..` [exp_3] `]`

The *arithmetic sequence* `[` e_1 `,` e_2 `..` e_3 `]` denotes a list of values of type t, where each of the e_i has type t, and t is an instance of class `Enum`.

Translation: Arithmetic sequences satisfy these identities:

`[` e_1`.. ]`	$=$	`enumFrom` e_1
`[` e_1`,`e_2`.. ]`	$=$	`enumFromThen` e_1 e_2
`[` e_1`..`e_3 `]`	$=$	`enumFromTo` e_1 e_3
`[` e_1`,`e_2`..`e_3 `]`	$=$	`enumFromThenTo` e_1 e_2 e_3

where `enumFrom`, `enumFromThen`, `enumFromTo`, and `enumFromThenTo` are class methods in the class `Enum` as defined in the Prelude (see Figure 6.1, p. 85).

The semantics of arithmetic sequences therefore depends entirely on the instance declaration for the type t. See Section 6.3.4 for more details of which `Prelude` types are in `Enum` and their semantics.

3.11 List Comprehensions

aexp	$\rightarrow$	`[` *exp* $\vert$ $qual_1$ `,` ... `,` $qual_n$ `]`	(list comprehension, $n \geq 1$)
qual	$\rightarrow$	*pat* `<-` *exp*	(generator)
	$\vert$	`let` *decls*	(local declaration)
	$\vert$	*exp*	(guard)

A *list comprehension* has the form `[` e `|` q_1 `,` ... `,` q_n `]`, $n \geq 1$, where the q_i qualifiers are either

- *generators* of the form p `<-` e, where p is a pattern (see Section 3.17) of type t and e is an expression of type `[`t`]`
- *guards*, which are arbitrary expressions of type `Bool`
- *local bindings* that provide new definitions for use in the generated expression e or subsequent guards and generators.

Such a list comprehension returns the list of elements produced by evaluating e in the successive environments created by the nested, depth-first evaluation of the generators in the qualifier list. Binding of variables occurs according to the normal pattern matching rules (see Section 3.17), and if a match fails then that element of the list is simply skipped over. Thus:

```
[ x |  xs   <- [ [(1,2),(3,4)], [(5,4),(3,2)] ],
      (3,x) <- xs ]
```

yields the list `[4,2]`. If a qualifier is a guard, it must evaluate to `True` for the previous pattern match to succeed. As usual, bindings in list comprehensions can shadow those in outer scopes; for example:

```
[ x | x <- x, x <- x ] = [ z | y <- x, z <- y]
```

Translation: List comprehensions satisfy these identities, which may be used as a translation into the kernel:

```
[ e | True ]            = [e]
[ e | q ]               = [ e | q, True ]
[ e | b, Q ]            = if b then [ e | Q ] else []
[ e | p <- l, Q ]       = let ok p = [ e | Q ]
                              ok _ = []
                          in concatMap ok l
[ e | let decls, Q ]    = let decls in [ e | Q ]
```

where e ranges over expressions, p over patterns, l over list-valued expressions, b over boolean expressions, *decls* over declaration lists, q over qualifiers, and Q over sequences of qualifiers. `ok` is a fresh variable. The function `concatMap`, and boolean value `True`, are defined in the Prelude.

As indicated by the translation of list comprehensions, variables bound by `let` have fully polymorphic types while those defined by `<-` are lambda bound and are thus monomorphic (see Section 4.5.4).

3.12 Let Expressions

exp	$\rightarrow$	`let` $decls$ `in` exp

Let expressions have the general form `let {` d_1 `;` ... `;` d_n `} in` e, and introduce a nested, lexically-scoped, mutually-recursive list of declarations (`let` is often called `letrec` in other languages). The scope of the declarations is the expression e and the right hand side of the declarations. Declarations are described in Chapter 4. Pattern bindings are matched lazily; an implicit `~` makes these patterns irrefutable. For example,

```
let (x,y) = undefined in e
```

does not cause an execution-time error until `x` or `y` is evaluated.

Translation: The dynamic semantics of the expression `let {` d_1 `;` ... `;` d_n `} in` e_0 are captured by this translation: After removing all type signatures, each declaration d_i is translated into an equation of the form $p_i = e_i$, where p_i and e_i are patterns and expressions respectively, using the translation in Section 4.4.3. Once done, these identities hold, which may be used as a translation into the kernel:

`let {`p_1`=`e_1`;` ... `;` p_n`=`e_n`} in` e_0	=	`let (~`p_1`,` ... `,~`p_n`) = (`e_1`,` ... `,`e_n`) in` e_0
`let` p `=` e_1 `in` e_0	=	`case` e_1 `of ~`p `->` e_0
		where no variable in p appears free in e_1
`let` p `=` e_1 `in` e_0	=	`let` p `= fix ( \ ~`p `->` e_1`) in` e_0

where `fix` is the least fixpoint operator. Note the use of the irrefutable patterns `~`p. This translation does not preserve the static semantics because the use of `case` precludes a fully polymorphic typing of the bound variables. The static semantics of the bindings in a `let` expression are described in Section 4.4.3.

3.13 Case Expressions

exp	$\rightarrow$	`case` exp `of {` $alts$ `}`	
$alts$	$\rightarrow$	alt_1 `;` ... `;` alt_n	$(n \geq 1)$
alt	$\rightarrow$	pat `->` exp [`where` $decls$]	
	|	pat $gdpat$ [`where` $decls$]	
	|		$(empty\ alternative)$
$gdpat$	$\rightarrow$	gd `->` exp [$gdpat$]	
gd	$\rightarrow$	| exp^0	

A *case expression* has the general form

`case` e `of {` p_1 $match_1$ `;` ... `;` p_n $match_n$ `}`

where each $match_i$ is of the general form

$$\begin{array}{ll} |\ g_{i1} & \texttt{->}\ e_{i1} \\ \ldots & \\ |\ g_{im_i} & \texttt{->}\ e_{im_i} \\ \multicolumn{2}{l}{\texttt{where}\ decls_i} \end{array}$$

(Notice that in the syntax rule for gd, the "|" is a terminal symbol, not the syntactic metasymbol for alternation.) Each alternative $p_i\ match_i$ consists of a pattern p_i and its matches, $match_i$. Each match in turn consists of a sequence of pairs of guards g_{ij} and bodies e_{ij} (expressions), followed by optional bindings ($decls_i$) that scope over all of the guards and expressions of the alternative. An alternative of the form

$$pat\ \texttt{->}\ exp\ \texttt{where}\ decls$$

is treated as shorthand for:

$$\begin{array}{l} pat\ |\ \texttt{True}\ \ \texttt{->}\ exp \\ \texttt{where}\ decls \end{array}$$

A case expression must have at least one alternative and each alternative must have at least one body. Each body must have the same type, and the type of the whole expression is that type.

A case expression is evaluated by pattern matching the expression e against the individual alternatives. The alternatives are tried sequentially, from top to bottom. If e matches the pattern in the alternative, the guards for that alternative are tried sequentially from top to bottom, in the environment of the case expression extended first by the bindings created during the matching of the pattern, and then by the $decls_i$ in the `where` clause associated with that alternative. If one of the guards evaluates to `True`, the corresponding right-hand side is evaluated in the same environment as the guard. If all the guards evaluate to `False`, matching continues with the next alternative. If no match succeeds, the result is $\perp$. Pattern matching is described in Section 3.17, with the formal semantics of case expressions in Section 3.17.3.

A note about parsing. The expression

```
case x of { (a,_) | let b = not a in b :: Bool -> a }
```

is tricky to parse correctly. It has a single unambiguous parse, namely

```
case x of { (a,_) | (let b = not a in b :: Bool) -> a }
```

However, the phrase `Bool -> a` is syntactically valid as a type, and parsers with limited lookahead may incorrectly commit to this choice, and hence reject the program. Programmers are advised, therefore, to avoid guards that end with a type signature – indeed that is why a gd contains an exp^0 not an exp.

3.14 Do Expressions

exp	$\rightarrow$	`do {` *stmts* `}`	(do expression)
stmts	$\rightarrow$	$stmt_1 \ldots stmt_n$ *exp* [`;`]	$(n \geq 0)$
stmt	$\rightarrow$	*exp* `;`	
	|	*pat* `<-` *exp* `;`	
	|	`let` *decls* `;`	
	|	`;`	(*empty statement*)

A *do expression* provides a more conventional syntax for monadic programming. It allows an expression such as

```
putStr "x: "    >>
getLine         >>= \l ->
return (words l)
```

to be written in a more traditional way as:

```
do putStr "x: "
   l <- getLine
   return (words l)
```

Translation: Do expressions satisfy these identities, which may be used as a translation into the kernel, after eliminating empty *stmts*:

```
do {e}                 =  e
do {e;stmts}           =  e >> do {stmts}
do {p <- e; stmts}     =  let ok p = do {stmts}
                              ok _ = fail "..."
                            in e >>= ok
do {let decls; stmts}  =  let decls in do {stmts}
```

The ellipsis `"..."` stands for a compiler-generated error message, passed to `fail`, preferably giving some indication of the location of the pattern-match failure; the functions `>>`, `>>=`, and `fail` are operations in the class `Monad`, as defined in the Prelude; and `ok` is a fresh identifier.

As indicated by the translation of `do`, variables bound by `let` have fully polymorphic types while those defined by `<-` are lambda bound and are thus monomorphic.

3.15 Datatypes with Field Labels

A datatype declaration may optionally define field labels (see Section 4.2.1). These field labels can be used to construct, select from, and update fields in a manner that is independent of the overall structure of the datatype.

Different datatypes cannot share common field labels in the same scope. A field label can be used at most once in a constructor. Within a datatype, however, a field label can be used in more than one constructor provided the field has the same typing in all constructors. To illustrate the last point, consider:

```
data S = S1 { x :: Int } | S2 { x :: Int }    -- OK
data T = T1 { y :: Int } | T2 { y :: Bool }   -- BAD
```

Here `S` is legal but `T` is not, because `y` is given inconsistent typings in the latter.

3.15.1 Field Selection

$aexp \quad \rightarrow \quad qvar$

Field labels are used as selector functions. When used as a variable, a field label serves as a function that extracts the field from an object. Selectors are top level bindings and so they may be shadowed by local variables but cannot conflict with other top level bindings of the same name. This shadowing only affects selector functions; in record construction (Section 3.15.2) and update (Section 3.15.3), field labels cannot be confused with ordinary variables.

Translation: A field label f introduces a selector function defined as:

f `x = case x of {` $C_1\ p_{11}\ \ldots\ p_{1k}$ `->` e_1 `;` $\ldots$ `;` $C_n\ p_{n1}\ \ldots\ p_{nk}$ `->` e_n `}`

where $C_1 \ldots C_n$ are all the constructors of the datatype containing a field labeled with f, p_{ij} is `y` when f labels the jth component of C_i or `_` otherwise, and e_i is `y` when some field in C_i has a label of f or `undefined` otherwise.

3.15.2 Construction Using Field Labels

$aexp$	$\rightarrow$	$qcon$ `{` $fbind_1$ `,` $\ldots$ `,` $fbind_n$ `}`	(labeled construction, $n \geq 0$)
$fbind$	$\rightarrow$	$qvar$ `=` exp	

A constructor with labeled fields may be used to construct a value in which the components are specified by name rather than by position. Unlike the braces used in declaration lists, these are not subject to layout; the `{` and `}` characters must be explicit. (This is also true of field updates and field patterns.) Construction using field labels is subject to the following constraints:

- Only field labels declared with the specified constructor may be mentioned.
- A field label may not be mentioned more than once.
- Fields not mentioned are initialized to $\perp$.

- A compile-time error occurs when any strict fields (fields whose declared types are prefixed by `!`) are omitted during construction. Strict fields are discussed in Section 4.2.1.

The expression `F {}`, where `F` is a data constructor, is legal *whether or not* `F` *was declared with record syntax* (provided `F` has no strict fields – see the third bullet above); it denotes `F` $\perp_1 \ \ldots \ \perp_n$, where n is the arity of `F`.

Translation: In the binding f = v, the field f labels v.

$$C \ \texttt{\{}\ bs\ \texttt{\}} \quad = \quad C\ (pick^C_1\ bs\ \texttt{undefined})\ \ldots\ (pick^C_k\ bs\ \texttt{undefined})$$

where k is the arity of C.
The auxiliary function $pick^C_i\ bs\ d$ is defined as follows:

> If the ith component of a constructor C has the field label f, and if $f = v$ appears in the binding list bs, then $pick^C_i\ bs\ d$ is v. Otherwise, $pick^C_i\ bs\ d$ is the default value d.

3.15.3 Updates Using Field Labels

$aexp \quad \rightarrow \quad aexp_{\langle qcon \rangle}\ \texttt{\{}\ fbind_1\ \texttt{,}\ \ldots\ \texttt{,}\ fbind_n\ \texttt{\}} \qquad$ (labeled update, $n \geq 1$)

Values belonging to a datatype with field labels may be non-destructively updated. This creates a new value in which the specified field values replace those in the existing value. Updates are restricted in the following ways:

- All labels must be taken from the same datatype.
- At least one constructor must define all of the labels mentioned in the update.
- No label may be mentioned more than once.
- An execution error occurs when the value being updated does not contain all of the specified labels.

Translation: Using the prior definition of $pick$,

$$
\begin{array}{lcl}
e\ \texttt{\{}\ bs\ \texttt{\}} & = & \texttt{case}\ e\ \texttt{of} \\
 & & \quad C_1\ v_1\ \ldots\ v_{k_1}\ \texttt{->}\ C_1\ (pick^{C_1}_1\ bs\ v_1)\ \ldots\ (pick^{C_1}_{k_1}\ bs\ v_{k_1}) \\
 & & \qquad \ldots \\
 & & \quad C_j\ v_1\ \ldots\ v_{k_j}\ \texttt{->}\ C_j\ (pick^{C_j}_1\ bs\ v_1)\ \ldots\ (pick^{C_j}_{k_j}\ bs\ v_{k_j}) \\
 & & \quad \texttt{\_ -> error "Update error"}
\end{array}
$$

where $\{C_1, \ldots, C_j\}$ is the set of constructors containing all labels in bs, and k_i is the arity of C_i.

Here are some examples using labeled fields:

```
data T    = C1 {f1,f2 :: Int}
          | C2 {f1 :: Int,
                f3,f4 :: Char}
```

Expression	Translation
`C1 {f1 = 3}`	`C1 3 undefined`
`C2 {f1 = 1, f4 = 'A', f3 = 'B'}`	`C2 1 'B' 'A'`

`x {f1 = 1}`	`case x of C1 _ f2    -> C1 1 f2`
	`           C2 _ f3 f4 -> C2 1 f3 f4`

The field `f1` is common to both constructors in T. This example translates expressions using constructors in field-label notation into equivalent expressions using the same constructors without field labels. A compile-time error will result if no single constructor defines the set of field labels used in an update, such as `x {f2 = 1, f3 = 'x'}`.

3.16 Expression Type-Signatures

$exp \quad \rightarrow \quad exp$ `::` $[context$ `=>`$]$ $type$

Expression type-signatures have the form e `::` t, where e is an expression and t is a type (Section 4.1.2); they are used to type an expression explicitly and may be used to resolve ambiguous typings due to overloading (see Section 4.3.4). The value of the expression is just that of exp. As with normal type signatures (see Section 4.4.1), the declared type may be more specific than the principal type derivable from exp, but it is an error to give a type that is more general than, or not comparable to, the principal type.

Translation:

e `::` t = `let {` v `::` t`;` $v = e$ `} in` v

3.17 Pattern Matching

Patterns appear in lambda abstractions, function definitions, pattern bindings, list comprehensions, do expressions, and case expressions. However, the first five of these ultimately translate into case expressions, so defining the semantics of pattern matching for case expressions is sufficient.

3.17.1 Patterns

Patterns have this syntax:

pat	$\rightarrow$	var + $integer$	(successor pattern)
	$\mid$	pat^0	
pat^i	$\rightarrow$	pat^{i+1} [$qconop^{(\mathrm{n},i)}$ pat^{i+1}]	
	$\mid$	$lpat^i$	
	$\mid$	$rpat^i$	
$lpat^i$	$\rightarrow$	$(lpat^i \mid pat^{i+1})$ $qconop^{(\mathrm{l},i)}$ pat^{i+1}	
$lpat^6$	$\rightarrow$	- $(integer \mid float)$	(negative literal)
$rpat^i$	$\rightarrow$	pat^{i+1} $qconop^{(\mathrm{r},i)}$ $(rpat^i \mid pat^{i+1})$	
pat^{10}	$\rightarrow$	$apat$	
	$\mid$	$gcon\ apat_1\ \ldots\ apat_k$	(arity $gcon = k$, $k \geq 1$)
$apat$	$\rightarrow$	var [@ $apat$]	(as pattern)
	$\mid$	$gcon$	(arity $gcon = 0$)
	$\mid$	$qcon$ { $fpat_1$, ... , $fpat_k$ }	(labeled pattern, $k \geq 0$)
	$\mid$	$literal$	
	$\mid$	_	(wildcard)
	$\mid$	(pat)	(parenthesized pattern)
	$\mid$	(pat_1 , ... , pat_k)	(tuple pattern, $k \geq 2$)
	$\mid$	[pat_1 , ... , pat_k]	(list pattern, $k \geq 1$)
	$\mid$	~ $apat$	(irrefutable pattern)
$fpat$	$\rightarrow$	$qvar$ = pat	

The arity of a constructor must match the number of sub-patterns associated with it; one cannot match against a partially-applied constructor.

All patterns must be *linear* – no variable may appear more than once. For example, this definition is illegal:

```
    f (x,x) = x   -- ILLEGAL; x used twice in pattern
```

Patterns of the form var@pat are called *as-patterns*, and allow one to use var as a name for the value being matched by pat. For example,

```
  case e of { xs@(x:rest) -> if x==0 then rest else xs }
```

is equivalent to:

```
  let { xs = e } in
    case xs of { (x:rest) -> if x==0 then rest else xs }
```

Patterns of the form _ are *wildcards* and are useful when some part of a pattern is not referenced on the right-hand side. It is as if an identifier not used elsewhere were put in its place. For example,

```
  case e of { [x,_,_]  ->  if x==0 then True else False }
```

is equivalent to:

```
case e of { [x,y,z]  ->  if x==0 then True else False }
```

3.17.2 Informal Semantics of Pattern Matching

Patterns are matched against values. Attempting to match a pattern can have one of three results: it may *fail*; it may *succeed*, returning a binding for each variable in the pattern; or it may *diverge* (i.e. return $\perp$). Pattern matching proceeds from left to right, and outside to inside, according to the following rules:

1. Matching the pattern var against a value v always succeeds and binds var to v.

2. Matching the pattern ~$apat$ against a value v always succeeds. The free variables in $apat$ are bound to the appropriate values if matching $apat$ against v would otherwise succeed, and to $\perp$ if matching $apat$ against v fails or diverges. (Binding does *not* imply evaluation.)

   Operationally, this means that no matching is done on a ~$apat$ pattern until one of the variables in $apat$ is used. At that point the entire pattern is matched against the value, and if the match fails or diverges, so does the overall computation.

3. Matching the wildcard pattern _ against any value always succeeds, and no binding is done.

4. Matching the pattern $con\ pat$ against a value, where con is a constructor defined by `newtype`, depends on the value:

   - If the value is of the form $con\ v$, then pat is matched against v.
   - If the value is $\perp$, then pat is matched against $\perp$.

   That is, constructors associated with `newtype` serve only to change the type of a value.

5. Matching the pattern $con\ pat_1\ \ldots\ pat_n$ against a value, where con is a constructor defined by `data`, depends on the value:

   - If the value is of the form $con\ v_1\ \ldots\ v_n$, sub-patterns are matched left-to-right against the components of the data value; if all matches succeed, the overall match succeeds; the first to fail or diverge causes the overall match to fail or diverge, respectively.
   - If the value is of the form $con'\ v_1\ \ldots\ v_m$, where con is a different constructor to con', the match fails.
   - If the value is $\perp$, the match diverges.

6. Matching against a constructor using labeled fields is the same as matching ordinary constructor patterns except that the fields are matched in the order they are named in the field list. All fields listed must be declared by the constructor; fields may not be named more than once. Fields not named by the pattern are ignored (matched against _).

7. Matching a numeric, character, or string literal pattern k against a value v succeeds if v `==` k, where `==` is overloaded based on the type of the pattern. The match diverges if this test diverges.

   The interpretation of numeric literals is exactly as described in Section 3.2; that is, the overloaded function `fromInteger` or `fromRational` is applied to an `Integer` or `Rational` literal (resp) to convert it to the appropriate type.

8. Matching an n+k pattern (where n is a variable and k is a positive integer literal) against a value v succeeds if x `>=` k, resulting in the binding of n to x `-` k, and fails otherwise. Again, the functions `>=` and `-` are overloaded, depending on the type of the pattern. The match diverges if the comparison diverges.

   The interpretation of the literal k is the same as in numeric literal patterns, except that only integer literals are allowed.

9. Matching an as-pattern *var*`@`*apat* against a value v is the result of matching *apat* against v, augmented with the binding of *var* to v. If the match of *apat* against v fails or diverges, then so does the overall match.

Aside from the obvious static type constraints (for example, it is a static error to match a character against a boolean), the following static class constraints hold:

- An integer literal pattern can only be matched against a value in the class `Num`.
- A floating literal pattern can only be matched against a value in the class `Fractional`.
- An n+k pattern can only be matched against a value in the class `Integral`.

Many people feel that n+k patterns should not be used. These patterns may be removed or changed in future versions of Haskell.

It is sometimes helpful to distinguish two kinds of patterns. Matching an *irrefutable pattern* is non-strict: the pattern matches even if the value to be matched is $\perp$. Matching a *refutable* pattern is strict: if the value to be matched is $\perp$ the match diverges. The irrefutable patterns are as follows: a variable, a wildcard, N *apat* where N is a constructor defined by `newtype` and *apat* is irrefutable (see Section 4.2.3), *var*`@`*apat* where *apat* is irrefutable, or of the form `~`*apat* (whether or not *apat* is irrefutable). All other patterns are *refutable*.

Here are some examples:

1. If the pattern `['a','b']` is matched against `['x',`$\perp$`]`, then `'a'` *fails* to match against `'x'`, and the result is a failed match. But if `['a','b']` is matched against `[`$\perp$`,'x']`, then attempting to match `'a'` against $\perp$ causes the match to *diverge*.

2. These examples demonstrate refutable vs. irrefutable matching:

```
(\ ~(x,y) -> 0) ⊥    ⇒    0
(\  (x,y) -> 0) ⊥    ⇒    ⊥
(\ ~[x] -> 0) []    ⇒    0
(\ ~[x] -> x) []    ⇒    ⊥

(\ ~[x,~(a,b)] -> x) [(0,1),⊥]    ⇒    (0,1)
(\ ~[x, (a,b)] -> x) [(0,1),⊥]    ⇒    ⊥

(\  (x:xs) -> x:x:xs) ⊥   ⇒   ⊥
(\ ~(x:xs) -> x:x:xs) ⊥   ⇒   ⊥:⊥:⊥
```

3. Consider the following declarations:

   ```
   newtype N = N Bool
   data    D = D !Bool
   ```

   These examples illustrate the difference in pattern matching between types defined by `data` and `newtype`:

   ```
   (\  (N True) -> True) ⊥     ⇒    ⊥
   (\  (D True) -> True) ⊥     ⇒    ⊥
   (\ ~(D True) -> True) ⊥     ⇒    True
   ```

   Additional examples may be found in Section 4.2.3.

Top level patterns in case expressions and the set of top level patterns in function or pattern bindings may have zero or more associated *guards*. A guard is a boolean expression that is evaluated only after all of the arguments have been successfully matched, and it must be true for the overall pattern match to succeed. The environment of the guard is the same as the right-hand-side of the case-expression alternative, function definition, or pattern binding to which it is attached.

The guard semantics have an obvious influence on the strictness characteristics of a function or case expression. In particular, an otherwise irrefutable pattern may be evaluated because of a guard. For example, in

```
f :: (Int,Int,Int) -> [Int] -> Int
f ~(x,y,z) [a] | (a == y) = 1
```

both `a` and `y` will be evaluated by `==` in the guard.

3.17.3 Formal Semantics of Pattern Matching

The semantics of all pattern matching constructs other than `case` expressions are defined by giving identities that relate those constructs to `case` expressions. The semantics of `case` expressions themselves are in turn given as a series of identities, in Figures 3.1 and 3.2. Any implementation should behave so that these identities hold; it is not expected that it will use them directly, since that would generate rather inefficient code.

```
(a)  case $e$ of { $alts$ } $=$ (\$v$ -> case $v$ of { $alts$ }) $e$
     where $v$ is a new variable
(b)  case $v$ of { $p_1$ $match_1$;  $\ldots$ ; $p_n$ $match_n$ }
     $=$  case $v$ of { $p_1$ $match_1$ ;
                        _  -> $\ldots$ case $v$ of {
                                   $p_n$ $match_n$ ;
                                   _  -> error "No match" }$\ldots$}
      where each $match_i$ has the form:
       | $g_{i,1}$  -> $e_{i,1}$ ; $\ldots$ ; | $g_{i,m_i}$ -> $e_{i,m_i}$ where { $decls_i$ }
(c)  case $v$ of { $p$ | $g_1$ -> $e_1$ ; $\ldots$
                      | $g_n$ -> $e_n$ where { $decls$ }
                    _     -> $e'$ }
     $=$ case $e'$ of
       {$y$ ->  (where $y$ is a new variable)
        case $v$ of {
              $p$ -> let { $decls$ } in
                     if $g_1$ then $e_1$ $\ldots$ else if $g_n$ then $e_n$ else $y$ ;
              _ -> $y$ }}
(d)  case $v$ of { ~$p$ -> $e$; _ -> $e'$ }
     $=$ (\$x_1 \ldots x_n$ -> $e$ ) (case $v$ of { $p$-> $x_1$ })$\ldots$(case $v$ of { $p$ -> $x_n$})
     where $x_1, \ldots, x_n$ are all the variables in $p$
(e)  case $v$ of { $x$@$p$ -> $e$; _ -> $e'$ }
     $=$ case $v$ of { $p$ -> ( \ $x$ -> $e$ ) $v$ ; _ -> $e'$ }
(f)  case $v$ of { _ -> $e$; _ -> $e'$ } $=$ $e$
```

Figure 3.1: Semantics of Case Expressions, Part 1

In Figures 3.1 and 3.2: e, e' and e_i are expressions; g and g_i are boolean-valued expressions; p and p_i are patterns; v, x, and x_i are variables; K and K' are algebraic datatype (`data`) constructors (including tuple constructors); and N is a `newtype` constructor.

Rule (b) matches a general source-language `case` expression, regardless of whether it actually includes guards – if no guards are written, then `True` is substituted for the guards $g_{i,j}$ in the $match_i$ forms. Subsequent identities manipulate the resulting `case` expression into simpler and simpler forms.

Rule (h) in Figure 3.2 involves the overloaded operator `==`; it is this rule that defines the meaning of pattern matching against overloaded constants.

These identities all preserve the static semantics. Rules (d), (e), (j), (q), and (s) use a lambda rather than a `let`; this indicates that variables bound by `case` are monomorphically typed (Section 4.1.4).

```
(g) case $v$ of { $K\,p_1 \ldots p_n$ -> $e$; _ -> $e'$ }
    = case $v$ of {
          $K\,x_1 \ldots x_n$ -> case $x_1$ of {
                                 $p_1$ -> ... case $x_n$ of { $p_n$ -> $e$ ; _ -> $e'$ } ...
                                 _  -> $e'$ }
          _ -> $e'$ }
    at least one of $p_1, \ldots, p_n$ is not a variable; $x_1, \ldots, x_n$ are new variables

(h) case $v$ of { $k$ -> $e$; _ -> $e'$ } = if ($v$==$k$) then $e$ else $e'$
    where $k$ is a numeric, character, or string literal.

(i) case $v$ of { $x$ -> $e$; _ -> $e'$ } = case $v$ of { $x$ -> $e$ }

(j) case $v$ of { $x$ -> $e$ } = ( \ $x$ -> $e$ ) $v$

(k) case $N\,v$ of { $N$ $p$ -> $e$; _ -> $e'$ }
    = case $v$ of { $p$ -> $e$; _ -> $e'$ }
    where $N$ is a newtype constructor

(l) case $\bot$ of { $N$ $p$ -> $e$; _ -> $e'$ } = case $\bot$ of { $p$ -> $e$ }
    where $N$ is a newtype constructor

(m) case $v$ of { $K$ { $f_1$ = $p_1$ , $f_2$ = $p_2$ , ...} -> $e$ ; _ -> $e'$ }
    = case $e'$ of {
       $y$ ->
        case $v$ of {
          $K$ { $f_1$ = $p_1$ } ->
                case $v$ of { $K$ { $f_2$ = $p_2$ , ... } -> $e$ ; _ -> $y$ };
                _ -> $y$ }}
    where $f_1, f_2, \ldots$ are fields of constructor $K$; $y$ is a new variable

(n) case $v$ of { $K$ { $f$ = $p$ } -> $e$ ; _ -> $e'$ }
    = case $v$ of {
         $K\,p_1 \ldots p_n$ -> $e$ ; _ -> $e'$ }
    where $p_i$ is $p$ if $f$ labels the $i$th component of $K$, _ otherwise

(o) case $v$ of { $K$ {} -> $e$ ; _ -> $e'$ }
    = case $v$ of {
         $K$ _..._ -> $e$ ; _ -> $e'$ }

(p) case ($K'$ $e_1$ ... $e_m$) of { $K$ $x_1$ ... $x_n$ -> $e$; _ -> $e'$ } = $e'$
    where $K$ and $K'$ are distinct data constructors of arity $n$ and $m$, respectively

(q) case ($K$ $e_1$ ... $e_n$) of { $K$ $x_1$ ... $x_n$ -> $e$; _ -> $e'$ }
    = (\$x_1 \ldots x_n$ -> $e$) $e_1$ ... $e_n$
    where $K$ is a data constructor of arity $n$

(r) case $\bot$ of { $K$ $x_1$ ... $x_n$ -> $e$; _ -> $e'$ } = $\bot$
    where $K$ is a data constructor of arity $n$

(s) case $v$ of { $x$+$k$ -> $e$; _ -> $e'$ }
    = if $v$ >= $k$ then (\$x$ -> $e$)($v$-$k$) else $e'$
    where $k$ is a numeric literal
```

Figure 3.2: Semantics of Case Expressions, Part 2