

Names and scripts

A pure functional program is *just an expression*.

Large expressions composed only of literals and operators are textually cumbersome.

```
Prelude> (1+2)*(3-4*5+6*7) `mod` 8-(9+8*7-6)*5+(4*3+2-1)
-279
```

We can keep expressions manageable by giving subexpressions *names*, and using the names in place of the subexpressions. An expression name begins with a *lowercase* letter. Names enable expressions to be large *logically* without being textually unwieldy.

Collections of names and their definitions are placed in text files called *scripts*.

A script whose file name is HE0.1hs:

```
> module HE0 where
> a = 1+2
> b = 3-4*5+6*7
> c = 9+8*7-6
> d = 4*3+2-1 -- This is a comment.
This is another comment.
```

Each script contains a module; the module's name, which appears between the reserved words **module** and **where**, begins with an *uppercase* letter.

.1hs \Rightarrow *literate* Haskell script: Any nonblank line that does not begin with ' $\>$ ' is a comment. Comment lines are separated from code by blank lines.

A script's file name = its module's name ++ (".1hs" or ".hs").

30 January 2005

Compiling scripts

We ask Hugs to *load* the script, which Hugs then compiles (the `load` command can be abbreviated to "`l`").

```
Prelude> :load HE0
Reading file "HE0.1hs":
Hugs session for:
/usr/share/hugs/lib/Prelude.hs
HE0.1hs
```

(Commands other than expressions begin with '`:`'.)

Then we ask Hugs to evaluate our expression:

```
HE0> a * b `mod` 8 - c * 5 + d
-279
```

A command-line expression can refer to any of the script's names (nice for stepwise development):

```
HE0> a
3
HE0> c
59
HE0> b*d
325
```

If no file name is given, the `load` command unloads all scripts except for `Prelude.hs`:

```
HE0> :l
Hugs session for:
/usr/share/hugs/lib/Prelude.hs
Prelude>
```

`Prelude.hs`, a script of standard definitions, is loaded automatically when HUGS is launched.

30 January 2005

Scope of names

In HE0.1hs, the names `a`, `b`, `c`, and `d` are *global*, i.e., visible everywhere in the script and in command-line expressions.

Names can be declared *local* to a definition in a **where** clause:

```
> module HE1 where
> a = 1+2
> b = 3-x+a
> where
> x = 4*5
> a = 6*7 -- hides the global `a`
> c = 9+8*7-6
> d = 4*3+2-1
```

Demonstrating HE1:

```
Hugs session for:
:lib:Prelude.hs
he1.1hs
HE1> a
3
HE1> b
25
HE1> x
ERROR: Undefined variable "x"
```

Variable `x` is invisible at the command line because the *scope* of names defined in a **where** clause is limited to

- other definitions in the same **where** clause
- the righthand side of the definition to which the **where** clause belongs

30 January 2005

Names can also be declared local to *expressions*, using **let**.

Example: an alternative definition of `b`:

```
> b = let x = 4*5
>       a = 6*7
>       in 3-x+a
```

Program Layout

In most programming languages, a program text is simply a string, and formatting is a matter of taste.

In Haskell, however (as in Python), layout is *syntactically meaningful*:

Every line of a definition (after the first) begins to the *right* of its first line.

Hence a line that begins in the same column as the current definition is not part of the current definition (it may start a new definition, or it may continue an enclosing definition).

This is known as the *offside rule*.

The offside rule is optional: `b`'s definition could be written on one line:

```
> b = let x = 4*5; a = 6*7 in 3-x+a
```

but the offside style, which minimizes punctuation, is generally preferred.

When the offside rule is in effect, **tabs matter!**

To avoid problems, do one of the following:

- set your editor to 8-column tabs.
- don't use tab characters at all— use spaces.

30 January 2005

Selection expressions

Haskell provides two forms of compound expression which select one subexpression from a set of alternatives.

The simpler form is the familiar *conditional* expression, which selects one of *two* alternatives:

if b **then** e_1 **else** e_2

in which

- b is a Boolean expression
- e_1 and e_2 are expressions of arbitrary (but identical) type.

For example,

if $b \wedge 2 \succcurlyeq 4 * a * c$ **then** "real" **else** "complex"

Its rewrite rules:

if True **then** x **else** y $\rightsquigarrow x$

if False **then** x **else** y $\rightsquigarrow y$

Notes:

- Because **if** p **then** x **else** y is an expression, the **else** branch is *not optional*.
- Evaluating **if** p **then** x **else** y evaluates p and *either* x *or* y (but *not both*).

The other form of selection expression is the **case** expression:

case exp **of**

$a_0 \rightarrow e_0$

$a_1 \rightarrow e_1$

...

$a_n \rightarrow e_n$

A case-expression's value is e_i , where

$i = (\downarrow k \mid 0 \leq k \leq n : a_k == exp)$.

If $(\forall k : 0 \leq k \leq n : a_k \neq exp)$, the case-expression's value is \perp .

(The symbol \perp stands for the *undefined* "value", and is pronounced "bottom".)

Example:

case month **of**

"feb" \rightarrow 28

"sep" \rightarrow 30

"apr" \rightarrow 30

"jun" \rightarrow 30

"nov" \rightarrow 30

_ \rightarrow 31

The case label '_' is the *wildcard* pattern; it matches *any* value.

(We'll see later that case-expressions allow more powerful forms of pattern-matching than mere equality tests.)

Functional expressions

The standard method for generalizing an expression is to replace one or more of its constants by *variables*. The result is an expression that denotes a *function*.

The classic notation for generalized expressions is the λ -*expression*.

Example: The expression

$3 * 2$

can be generalized in three different ways:

$\lambda x \rightarrow x * 2$

$\lambda y \rightarrow 3 * y$

$\lambda x y \rightarrow x * y$

The ' λ ' (an ASCII substitute for λ) introduces the λ -expression's *bound variable*, i.e., its *parameter*.

In most programming languages, defining a function also names it. In Haskell, defining functions and naming them are *separable*.

A λ -expression is *applied* to an argument by simply writing it in front of the argument (prefix notation), as in

$(\lambda x \rightarrow x * 2) (6+7)$

The parentheses around $6+7$ are needed (*only*) because

function application has **highest precedence** (higher than any operator)

Parentheses are placed around arguments only in order to override precedence and associativity.

An application of a λ -expression to an argument is *evaluated* by substituting the argument for each occurrence of the parameter:

$(\lambda x \rightarrow x * 2) (6+7)$

$\rightsquigarrow (6+7) * 2$

$\rightsquigarrow 13 * 2$

$\rightsquigarrow 26$

This is an example of a *calculation*. It is a faithful analog of the actual HUGS98 evaluation process.

Note that *application* and *generalization* are each other's inverses:

- generalization replaces constants with variables
- application replaces variables with constants

A λ -expression having several parameters takes arguments in the same left-to-right order as the parameters:

$(\lambda x y \rightarrow 3 * x + 2 * y) 8 5$

$\rightsquigarrow (\lambda y \rightarrow 3 * 8 + 2 * y) 5$

$\rightsquigarrow 3 * 8 + 2 * 5$

$\rightsquigarrow \dots$

• No parentheses around arguments, and no commas between them.

A multiparameter λ -expression is "syntactic sugar" for *nested* single-parameter λ -expressions:

$(\lambda x y \rightarrow 3 * x + 2 * y) = (\lambda x \rightarrow (\lambda y \rightarrow 3 * x + 2 * y))$

so

$(\lambda x \rightarrow (\lambda y \rightarrow 3 * x + 2 * y)) 8 5$

$\rightsquigarrow (\lambda y \rightarrow 3 * 8 + 2 * y) 5$

$\rightsquigarrow 3 * 8 + 2 * 5$

$\rightsquigarrow \dots$

• The function takes its arguments **one at a time**.