Haskell: Syntactical Details

- Now time for a proper introduction to the language of Haskell.
  - Online: http://haskell.org/onlinereport
- In this lecture we refer to sections of that report thus:
  - [H98 3.4]
  - Haskell 98 Report, Section 3.4
- Recently, the Haskell 2010 Report has been issued (we will ignore this).

Program Structure [H98 1.1]

A Haskell script can be viewed as having four levels:
1. A Haskell program is a set of modules, that control namespaces and software re-use in large programs.
2. A module consists of a collection of declarations, defining ordinary values, datatypes, type classes, and fixity information.
3. Next are expressions, that denote values and have static types.
4. At the bottom level is the lexical structure, capturing the concrete representation of programs in text files.
(We focus on the bottom three today).

Namespaces [H98 1.4]

- Six kinds of names in Haskell:
  1. Variables, denoting values;
  2. (Data-)Constructors, denoting values;
  3. Type-variables, denoting types;
  4. Type-constructors, denoting ‘type-builders’;
  5. Type-classes, denoting groups of ‘similar’ types;
  6. Module-names, denoting program modules.
- Two constraints on naming:
  - Variables (??) and type variables (??) begin with lowercase letters or underscore, other names begin with uppercase letters.
  - An identifier cannot denote both a type-constructor (??) and type-class (??) in the same scope.
- So the name Thing (e.g.) can denote a module, class and constructor in a single scope.
Notational Conventions \([H98 \ 2.1]\)

- We use the following notation for syntax:
  - \([pattern]\)  \(\) optional
  - \(\{pattern\}\)  zero or more repetitions
  - \((pattern)\)  grouping
  - \(pat_1|pat_2\)  choice
  - \(pat_{\langle pat'\rangle}\)  difference—elements generated by \(pat\), except those generated by \(pat'\)

- fibonacci terminal syntax in typewriter font

- We use BNF-like syntax, with productions of the form:

  \[
  \text{nonterm} \rightarrow \text{alt}_1|\text{alt}_2|\ldots|\text{alt}_n
  \]

- The trick is distinguishing \(\mid\) (alternative separator) from \(\mid\), the vertical bar character (and similarly for characters \{\} [] ()).

Character Types (I) \([H98 \ 2.2]\)

- The characters can be grouped as follows:
  - \(special : ( ) , ; [ ] ' \{ \} \)
  - \(whitechar \rightarrow \text{newline}|\text{vertab}|\text{space}|\text{tab}\)
  - \(small \rightarrow a|b|\ldots|z|\)
  - \(large \rightarrow A|B|\ldots|Z\)
  - \(digit \rightarrow 0|1|\ldots|9\)
  - \(symbol : ! \# \$ \% \& * + . / = ? @ ^ _ \)

- the following characters are not explicitly grouped: : " ' _

(There is also stuff regarding Unicode characters (beyond ASCII) that we shall ignore—so the above is not exactly as shown in \([H98 \ 2.2]\)).

Character Types (II) \([H98 \ 2.2]\)

- From the previous groups we assemble as follows:
  - \(graphic \rightarrow \text{small}|\text{large}|\text{symbol}|\text{digit}|\text{special}|:\'|\)
  - \(\text{any} \rightarrow \text{graphic}|\text{space}|\text{tab}\)
  - \(\text{ANY} \rightarrow \text{graphic}|\text{whitechar}\)

- Any character not in \(\text{ANY}\) is illegal in a Haskell program.

Lexemes (I) \([H98 \ 2.2.2.4]\)

- We have eight types of lexeme:

  \[
  \text{lexeme} \rightarrow \text{qvarid}|\text{qconid}|\text{qvarsym}|\text{qconsym} \\
  \quad |\text{literal}|\text{special}|\text{reservedop}|\text{reservedid}
  \]

- \(\text{literal} \rightarrow \text{integer}|\text{float}|\text{char}|\text{string}\)

- \(\text{qXXX} \rightarrow \{\text{modid} .\}XXX\) (qualified name)

  where \(XXX\) is \(\text{varid}, \text{conid}, \text{varsym}, \text{or consym}.

- \(\text{reservedop} \rightarrow \) any of: \(\ldots : \ldots = \ldots \mid \leftrightarrow \rightarrow \leftarrow\)

- \(\text{reservedid} \rightarrow \) any of:

  - case class data default deriving do else if import in infix infixl infixr instance let module newtype of then type where _
Lexemes (II) [H98 2.4]

Identifiers \((\text{varid, conid})\) are usually prefix, whilst operators \((\text{varsym, consym})\) are usually infix.

- \(\text{varid} \rightarrow (\text{small} \{\text{small} | \text{large} | \text{digit} | ^*\})_{\text{reservedid}}\)
  - Examples: \(x \ x' \ a123 \ \text{myGUI} \ \_HASH \ \text{very\_long\_Ident\_indeed'}\)
- \(\text{conid} \rightarrow \text{large} \{\text{small} | \text{large} | \text{digit} | ^*\}\)
  - Examples: \(T \ \text{Tree} \ \text{Tree'} \ \text{My\_New\_Datatype} \ \text{Variant123}\)
- \(\text{varsym} \rightarrow (\text{symbol} \{\text{symbol} | :\})_{\text{reservedop}}\)
  - Examples: \(<+> | :| ++ + - === == && #!#\)
- \(\text{consym} \rightarrow (:\{\text{symbol} | :\})_{\text{reservedop}}\)
  - Examples: \(:+: \ :~ :=== :$$&\)

6 Name Types [H98 2.4]

- The rest of the report then uses identifiers and variables as follows:
  - \(\text{varid}\) \hspace{1cm} \text{variables}
  - \(\text{conid}\) \hspace{1cm} \text{constructors}
  - \(\text{tyvar} \rightarrow \text{varid}\) \hspace{1cm} \text{type variables}
  - \(\text{tycon} \rightarrow \text{conid}\) \hspace{1cm} \text{type constructors}
  - \(\text{tycls} \rightarrow \text{conid}\) \hspace{1cm} \text{type classes}
  - \(\text{modid} \rightarrow \text{conid}\) \hspace{1cm} \text{type constructors}
- So, a \(\text{varid}\) can be a (value-)variable or a type-variable;
- Whereas a \(\text{conid}\) can denote a (data-)constructor, type-constructor, type-class or module-identifier.

4 Literals (H98 2.5, 2.6)

We give a simplified introduction to literals

- \(\text{integer} \rightarrow \text{digit}[\text{digit}]\)
  - Examples: \(0 \ 123\)
- \(\text{float} \rightarrow \text{decimal} . \text{decimal}[\text{exponent}]\)
- \(\text{exponent} \rightarrow (e|E)[+][-] \text{decimal}\)
  - Examples: \(0.0 \ 1.2e3 \ 1.4e-45 \ \backslash\)
- \(\text{char} \rightarrow (\text{graphic} | \text{space} | \text{escape})\)
  - Examples: \('a' \ '$' \  '\'' \  '\64' \ '\n'\)
- \(\text{string} \rightarrow "(\text{graphic} | \text{space} | \text{escape})"\)
  - Examples: \"Hello World" \"I 'like' you\"
  - Examples: \"\" is a dbl-quote" \"line1\line2"\"

Layout [H98 2.7]

Offside rule example, for expression

\(f(x) \ \text{where} \ x = y + 3 \ \land \ z = 10 \ \land \ f(a) = a + 2z\)

- Full syntax:
  \(f \ x \ \text{where} \ {x = y + 3 ; z = 10 ; f a = a + 2 * z }\)
- Using Layout:
  \(f \ x \ \text{where} \ x = y + 3\)
  \(z = 10\)
  \(f \ a = a + 2 \ast z\)
- Using Layout (alternative):
  \(f \ x \ \text{where} \ x = y + 3\)
  \(z = 10\)
  \(f \ a \ = \ a + 2 \ast z\)
Operators [H98 3]

- Expressions can built up as expected in many programming languages:
  \[ 3 \cdot x + x + y \quad (x \leq y) \quad a + c*d - (c * (a / b)) \]

- Some operators are left-associative like \(+ - * /:\)
  \[ a + b + c \] parses as \((a + b) + c\)

- Some operators are right-associative like \(: \& \& \| \|:\)
  \[ a : b : c : [] \] parses as \((a : (b : (c : [])))\)

- Other operators are non-associative like \(== /= <\ll\geq >:\)
  \[ a <= b <= c \] is illegal,
  but \((a <= b) \& \& (b <= c)\) is ok.

- The minus sign is tricky: \(e - f\) parses as “e subtract f”,
  \((- f)\) parses as “minus f”,
  but \(e (- f)\) parses as “function e applied to argument minus f”.

Function Application/Types

- Function application is denoted by juxtaposition, and is left associative.
  \(f x y z\) parses as \(((f x) y) z\)

- If we want \(f\) applied to \(x\) and the application of \(g\) to \(y\),
  we must write \(f x (g y)\)

- In types, the function arrow is right associative:
  \(Int \rightarrow Char \rightarrow Bool\) parses as \(Int \rightarrow (Char \rightarrow Bool)\)
  The type of a function whose first argument is itself a function,
  has to be written as \((a \rightarrow b) \rightarrow c\)

- Note the following types are identical:
  \((a \rightarrow b) \rightarrow (c \rightarrow d)\)
  \((a \rightarrow b) \rightarrow c \rightarrow d\)

Sections [H98 3.2,3.5]

- A “section” is an operator, with possibly one argument surrounded by parentheses,
  which can be treated as a prefix function name.

- \((+\) is a prefix function adding its arguments (e.g.
  \((+23 = 5)\)

- \((/\) is a prefix function dividing its arguments (e.g.
  \((/2.0 4.0 = 0.5)\)

- \((/4.0)\) is a prefix function dividing its single argument by 4
  (e.g. \((/4.0)10.0 = 2.5)\)

- \((10.0/)\) is a prefix function dividing 10 by its single argument
  (e.g. \((10/)4.0 = 2.5)\)

- \((-\ e)\) is not a section, use \(\text{subtract} \ e\) instead. (e.g.
  \((\text{subtract}1)4 = 3)\)

Conditionals [H98 3.6]

- For expressions, we can write a conditional using if
  \(\ldots\) then \ldots else

  \[ \text{exp} \rightarrow \text{if exp then exp else exp} \]

- The else-part is compulsory, and cannot be left out (why not?)

- The (boolean-valued) expression after if is evaluated:
  If true, the value is of the expression after then
  If false, the value is of the expression after else
Lists [H98 3.7,3.10]

- Fundamentally lists are built from "nil" ([]) and "cons" (:)
- We use square brackets to provide syntactical sugar in a variety of ways
  - Enumeration: [a,b,c,d] for a:b:c:d:[]
  - Ranges: [4..9] for [4,5,6,7,8,9]
    also [4,7..20] for [4,7,10,13,16,19]
  - Comprehension:
    [ x*x | x <- [1..10], even x] for [4,16,36,64,100]
    Comprehensions are more complex than this (see later, or [H98 3.11])
  - Strings are a special notation of lists of characters
    "Hello" for ['H','e','l','l','o']

Tuples [H98 3.8,3.9]

- Tuples are expressions surrounded by parentheses, and separated by commas
  e.g. (1,2) (3,True,'c')
- A n-tuple can also be built by applying a "tuple-constructor" to the components:
  (1,a,3,"Hi") can also be written as (,,,) 1 a 3 "Hi"
  (why might one want to do this?)
- (e) and e are exactly the same (no 1-tuples)
- () is a special value, belonging to the special unit type, (also written as ()).
  (what's this for?)

Local Declarations [H98 3.12]

- A let-expression has the form:

  let {d_1; ...; d_n} in e

  d_i are declarations, e is an expression
- Scope of each d_i is e and right side of all the d_i's (mutual recursion)
- Example

  solve a b c
  = let twoa = 2 * a
     discr = b*b - 2 * twoa * c
     droot = sqrt discr
     in ( (droot-b)/twoa , negate ((droot+b)/twoa)

  (What does this compute?)

The Haskell Prelude [H98 8]

- The "Prelude" is a standard library of functions loaded automatically into any Haskell program.
- Contains most commonly used datatypes and functions
- [H98 8] is a specification of the Prelude
  the actual code is compiler dependent
Infix declarations

- **infixr 9 .**
- **infixr 8 ``, ``, `**
- **infixl 7 `*`, `/`, `quot`, `rem`, `div`, `mod`**
- **infixl 6 `+`, `-**
- **infixl 5 :**
- **infix 4 `==`, `/=`, `<`, `<=`, `>`, `>=`**
- **infixr 3 `&&`**
- **infixr 2 `||`**
- **infixl 1 `>>`, `>>=`**
- **infixr 1 `=<<`**
- **infixr 0 `$`, `!`, `seq`**

Higher precedence numbers bind tighter.

### Prelude extracts (II)

**Numeric Functions**

- `subtract` :: (Num a) => a -> a -> a
- `even`, `odd` :: (Integral a) => a -> Bool
- `gcd` :: (Integral a) => a -> a -> a
- `lcm` :: (Integral a) => a -> a -> a
- `(^^)` :: (Fractional a, Integral b) => a -> b -> a
- `(^^)` :: (Fractional a, Integral b) => a -> b -> a

### Prelude extracts (III)

**Function Functions**

- `id` :: a -> a
- `const` :: a -> b -> a
- `(.)` :: (b -> c) -> (a -> b) -> a -> c
- `flip` :: (a -> b -> c) -> b -> a -> c
- `seq` :: a -> b -> b
- `($)`, `(!!)` :: (a -> b) -> a -> b

### Prelude extracts (IV)

**Boolean Type & Functions**

- `data Bool = False | True`
- `(&&), (||)` :: Bool -> Bool -> Bool
- `not` :: Bool -> Bool
- `otherwise` :: Bool
List Functions

- `map :: (a -> b) -> [a] -> [b]`
- `(++) :: [a] -> [a] -> [a]`
- `filter :: (a -> Bool) -> [a] -> [a]`
- `concat :: [[a]] -> [a]`
- `head :: [a] -> a`
- `tail :: [a] -> [a]`
- `null :: [a] -> Bool`
- `length :: [a] -> Int`
- `(!!) :: [a] -> Int -> a`
- `repeat :: a -> [a]`
- `take :: Int -> [a] -> [a]`
- `drop :: Int -> [a] -> [a]`