Contents

Structure

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 *declerations*, 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 for now

Comments

A Haskell script has two kinds of comments

1. End of line comments, starting with `--`
2. Nested comments, started with `{-` and ending with `-}`

```
myfun x -- end of line comment
  = let
      y = 2 {- next but -- is ignored here -} ; z = 3
      {-
      y = 2
      z = 3
      -}
    in y + z * x
```

Namespaces

- 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 (only) on naming:
  - **Variables** and **Type-Variables** begin with lowercase letters or underscore
  - Other names begin with uppsercase letters
  - An identifier cannot denote both a **Type constructor** and **Type class** in the same scope
- So the name `Thing` cannot denote a module, data constructor, and either a class or type constructor in a single scope

Notational Conventions

- The report uses the following notation for syntax:
  - `[syn]` optional occurence of `syn`
  - `{syn}` zero or more repititons of `syn`

- `(syn)` grouping
- $\texttt{syn}_1\texttt{|syn}_2$ choice between alternatives
- $\texttt{syn}_{\texttt{<syn'>}}$ difference - elements generated by `syn` except those generated by `syn'`

Character Types

- special: `( ) , ; [ ] ' { }`
- whitechar -> `newline|vertab|space|tab`
- small -> `a|b|c|...|z|_`
- big -> `A|B|C|...|Z`
- digit -> `0|1|2|...|9`
- symbol: `! # % & * + . / < - > ? ...`
- the following characters are not explicitly grouped: `: " '`

Lexemes

The term *lexeme* refers to a single basic *word* in the language

- **Variable Identifiers** (varid) start with lowercase and continue with letters, numbers, underscore, and single-quote
- **Constructor Identifiers** (conid) start with uppercase letters and continue with letters, numbers, underscore and single-quote
- **Variable Operators** (varsym) start with any symbol, and continue with symbols and the colon
- **Constructor Operators** (consym) start with a colon and continue with symbols and the colon

Identifiers are usually prefix, whilst operators are usually infix

- **Reserved Identifiers** (reservedid) `case class data default deriving do else foreign if import in infix infixl infixr instance let module newtype of then type where _`
- *Reserved Operators* (reservedop) `.. : :: = \ | <- -> @ ~ =`

Literals

We give a simplified introduction to literals (actual basic values)

- *Integers* (integer) are sequence of digits
- *Floating Point* (float) has the same syntax as found in mainstream programming languages
- *Characters* (char) are inclosed in single quotes and can be escaped using backslash in standard ways
- *Strings* (string) are inclosed in double quotes and can also be escaped using backslash in standard ways

Layout Rule

- Some Haskell syntax specifies lists of declarations or actions as follows: `{item`$_1$`;item`$_2$`;item`$_3$`;...;item`$_N$`}`
- In some cases (after keywords `where`, `let`, `do`, `of`), we can drop `{, }` and `;`
- The layout (or off-side) rule takes effect whenever the open brace is omitted
  - When this happens, the indentation of the next lexeme (whether or not on a new line) is remembered and the omitted open brace is inserted (the whitespace preceding the lexeme may include comments)
  - For each subsequent line, if it contains only whitespace or is indented more, then the previous item is continued (nothing is inserted)
  - If it is indented the same amount, the a new item begins (a semi colon is inserted)
  - And if it is indented less, then the layout list ends (a close brace is inserted)

Offside rule example: $f(x) where x = y+3 \wedge z = 10 \wedge f(a) = a+2z$ - Full syntax:

```
let { x = y + 3; x = 10; f a = a + 2 * z } in f x
```

- Using layout:

```
let x = y + 3
    y = 10
    f a = f + 2 * z
in f x
```

or

```
let x = y + 3
    y = 10
    f a
      = f + 2 * z
in f x
```

Operators

- Expressions can build up as expected in many programming languages
- Some operators are left-associative like `+ - * /`
  - `a+b+c = (a+b)+c`
- Some operators are right-associative like `: . ^ && ||`
  - `a:b:c:[] = a:(b:(c:[]))`
- Other operators are non-associative like `== /= < <= >= >`
  - `a <= b <= c` is illegal
  - `(a <= b) && (b <= c)` is ok
- The minus sign is tricky:
  - `e-f` parses as `e subtract f`
  - `(-f)` parsed as `minus f`
  - `e (- f)` parsed as `function e applied to argument minus f`

Function Application/Types

- Function appication is denoted by juxtaposition, and if 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 -> Char -> Bool` parses as `Int -> (Char -> Bool)`
- The type of a function whose first argument is itself a function, has to be written at `(a -> b ) -> c`
- Note the following types are identical
  - `(a -> b) -> (c -> d)`
  - `(a -> b) -> c -> d)`

Identifiers as Operators

- We can take a variable identifier that denotes a function taking two arguments and turn it into an infix operator by surrounding it by backquotes
- `mod` is a prefix function that computes the vale of its first argument modulo its second
- Adding backquotes allows it to be used in an infix setting

```
> mod 37 5
2
> 37 `mod` 5
2
```

Don't confuse the backtick here with the single quote for characters

Sections

- A "section" is an operator, with possible one argument surrounded by parentheses, which can be treated as a prefix function name
- `(+)` is a prefix function adding its arguments, e.g. `(+) 2 3 = 5`
- `(/)` is a prefix function dividing its arguments
- `(/4.0)` is a prefix function dividing its single arguments by `4`, e.g. `(/4.0) 10.0 = 2.5`
- `(- e)` is not a section, use `subtract e` instead