Chapter 1

Basic Haskell Syntax

1.1 Introduction to Haskell Syntax

Haskell's syntax reflects its foundation in mathematical logic and functional programming principles. The language employs a minimalist syntax that emphasizes expressiveness through composition of functions rather than imperative statements. Distinctive features include significant whitespace for block structure, a strong static type system with type inference, and pervasive use of pattern matching. This section systematically explores these elements through concrete examples and detailed explanations.

1.2 Module Declaration

```haskell
module Main where
```

Every Haskell program begins with a module declaration. The `Main` module serves as the entry point for executable programs, analogous to the `main` function in C-family languages. The `where` keyword initiates the module body, containing all subsequent declarations. When no explicit export list is provided (as in this example), all top-level bindings become publicly accessible. For library components, explicit exports clarify the public interface:

```haskell
module Geometry.Sphere
  ( volume
  , area
  ) where
```

Here, only the `volume` and `area` functions are exposed to other modules. The module name `Geometry.Sphere` corresponds to the file path `Geometry/Sphere.hs`, following Haskell's hierarchical module convention. This encapsulation mechanism enables information hiding and modular program architecture.

1.3 Basic Data Types and Variables

Haskell's type system provides both primitive types and rich abstraction mechanisms. Consider these fundamental type declarations:

```haskell
intVal :: Int
intVal = 42

floatVal :: Float
floatVal = 3.14159
```

```haskell

inferredVal = "Hello Haskell!"
```

The `Int` type represents machine-precision integers (typically 64-bit), while `Float` denotes single-precision floating-point numbers. Type signatures use the `::` symbol to associate values with types. The third example demonstrates type inference: the compiler automatically deduces `inferredVal` has type `String` (equivalent to `[Char]`), a list of characters. All bindings are immutable; once defined, their values cannot be altered.

1.4 Functions

1.4.1 Function Definition and Application

```haskell
add :: Int -> Int -> Int
add x y = x + y
```

This function definition contains three essential components:

1. The **type signature** `Int -> Int -> Int` specifies that `add` accepts two `Int` arguments and returns an `Int`. The arrow associates to the right, meaning the type is equivalent to `Int -> (Int -> Int)`, reflecting Haskell's curried function application.

2. The **parameter list** `x y` declares two integer parameters. Unlike imperative languages, parameters are separated by whitespace rather than commas.

3. The **function body** `x + y` defines the computation. The equals sign denotes definitional equality rather than assignment.

1.4.2 Recursive Definitions with Pattern Matching

```haskell
factorial :: Integer -> Integer
factorial 0 = 1
factorial n = n * factorial (n - 1)
```

The factorial function demonstrates two fundamental Haskell features: recursion and pattern matching. The type `Integer` represents arbitrary-precision integers, contrasting with fixed-size `Int`. The definition contains two clauses:

- The base case `factorial 0 = 1` matches when the input is exactly zero, terminating the recursion.
- The recursive case `factorial n = n * factorial (n - 1)` matches any positive integer, decomposing the problem into smaller subproblems. Parentheses around `n - 1` are required due to function application having higher precedence than arithmetic operators.

1.4.3 Conditional Execution with Guards

```haskell
absolute :: Int -> Int
absolute x
  | x < 0     = -x
  | otherwise = x
```

Guards provide a clean syntax for conditional branching. The vertical bar `|` introduces a Boolean expression, with the corresponding right-hand side executing when the guard evaluates to `True`. Guards are evaluated top-to-bottom, with `otherwise` (defined as `True`) serving as a default case. This example returns the absolute value of integer `x`, demonstrating how guards can replace nested `if-then-else` expressions.

1.5 Lists and Tuples

1.5.1 List Construction and Manipulation

```
numbers :: [Int]
numbers = [1, 2, 3, 4, 5]

evens = [2, 4 .. 20]

myList = 1 : 2 : 3 : []
```

Lists are homogeneous sequences with square bracket syntax. The type `[Int]` denotes a list of integers. Three construction methods are shown:

- Explicit enumeration: `[1,2,3,4,5]` creates a list through direct element specification.
- Arithmetic sequences: The `..` operator generates elements using step values. `[2,4..20]` produces even numbers through step inference from the first two elements.
- Cons operator: The colon `:` prepends elements to an existing list. The expression `1:2:3:[]` builds a list equivalent to `[1,2,3]`.

1.5.2 List Comprehensions

```
squares = [x^2 | x <- [1..10]]

pythagoreanTriples = [(a,b,c) | a <- [1..10],
                                b <- [a..10],
                                c <- [b..10],
                                a^2 + b^2 == c^2]
```

List comprehensions provide declarative set-builder notation. The first example generates squares of numbers 1 through 10. The vertical bar separates the output expression x$\hat{2}$ from the generator `x <- [1..10]`.

The Pythagorean triples example demonstrates multiple generators and filters:

- `a <- [1..10]` iterates values for side `a`
- `b <- [a..10]` ensures `b` is at least `a`
- `c <- [b..10]` ensures `c` is at least `b`
- The condition a$\hat{2}$ + b$\hat{2}$ == c$\hat{2}$ filters valid right triangles

1.5.3 Tuple Types

```
coordinates :: (Double, Double)
coordinates = (3.5, 4.2)

person :: (String, Int, Bool)
person = ("Alice", 30, True)
```

Tuples store fixed-size, heterogeneous collections. The type `(Double, Double)` represents a pair of double-precision numbers, while `(String, Int, Bool)` contains three elements of different types. Unlike lists, tuples preserve type information for each position, enabling structured data storage without custom types.

1.6 Control Structures

1.6.1 Conditional Expressions

```
signum' :: Int -> Int
signum' x = if x < 0
            then -1
            else if x > 0
                 then 1
                 else 0
```

Haskell's `if-then-else` construct differs from imperative languages by being an expression rather than a statement. This example returns:

- -1 for negative inputs
- 1 for positive inputs
- 0 for zero

Each `if` must have both `then` and `else` branches, as all expressions must evaluate to a value. The indentation aligns alternatives for readability.

1.6.2 Pattern Matching with Case

```
describeList :: [a] -> String
describeList lst = case lst of
  []     -> "Empty"
  [x]    -> "Singleton"
  (_:xs) -> "Multiple elements"
```

The `case` expression performs structural pattern matching on lists:

- `[]` matches the empty list
- `[x]` matches singleton lists, binding the element to `x`
- `(_:xs)` matches the cons operator, ignoring the head (_) and binding the tail to `xs`

This approach elegantly handles different list forms without explicit length checking.

1.7 Type Declarations

1.7.1 Algebraic Data Types (ADTs)

```
data Shape = Circle Float
           | Rectangle Float Float
```

ADTs allow creating new types through combinations of sums (alternatives) and products (fields). This definition:

- Declares a `Shape` type with two constructors
- `Circle` takes a single `Float` (radius)
- `Rectangle` takes two `Float` values (width and height)

Constructors can be used in pattern matching:

```haskell
area :: Shape -> Float
area (Circle r) = pi * r^2
area (Rectangle w h) = w * h
```

1.7.2 Record Syntax

```haskell
data Person = Person
  { name :: String
  , age  :: Int
  } deriving (Show)
```

Records add named fields to data constructors:

- `name` and `age` are field labels
- Automatically generates accessor functions: `name :: Person -> String`
- `deriving (Show)` enables string representation

Construction and access:

```haskell
alice = Person {name = "Alice", age = 30}
aliceName = name alice  -- "Alice"
```

1.8 Where vs Let

1.8.1 Where Clauses

```haskell
volume r = (4.0 / 3.0) * pi * r^3
  where pi = 3.141592653589793
```

The `where` clause attaches local definitions to the entire function body:

- `pi` is visible throughout the equation
- Typically used for auxiliary definitions
- Appears after the main expression

1.8.2 Let Expressions

```haskell
surfaceArea r = let pi = 3.141592653589793
                in 4 * pi * r^2
```

The `let` form binds variables locally within an expression:

- `pi` is only visible in the `in` block
- Can appear in any expression context
- Allows multiple bindings separated by semicolons

1.9 Indentation Rules

Haskell uses layout-sensitive syntax for code blocks:

```
main = do
  putStrLn "Enter your name:"
  name <- getLine
  putStrLn ("Hello, " ++ name ++ "!")
```

Key rules:

- All lines in a block must align vertically
- The `do` keyword initiates a sequence of IO actions
- Indentation level determines block membership
- Tabs are discouraged; use spaces for consistent formatting

1.10 Complete Example Programs

1.10.1 Hello World

```
module Main where

main :: IO ()
main = putStrLn "Hello, World!"
```

This canonical example demonstrates:

- Mandatory `Main` module for executables
- `main` function with `IO ()` type
- `putStrLn` function for string output
- The unit type `()` indicating no meaningful return value

1.10.2 Factorial Calculator

```
module Main where

factorial :: Integer -> Integer
factorial 0 = 1
factorial n = n * factorial (n - 1)

main :: IO ()
main = do
  putStrLn "Enter a number:"
  input <- getLine
  let n = read input
  putStrLn $ "Factorial: " ++ show (factorial n)
```

This interactive program showcases:

1. Recursive factorial definition with pattern matching
2. `do` notation for sequencing IO actions
3. `<-` operator binding input result

4. `read` converting String to Integer
5. `show` rendering numerical result as String
6. The `$` operator reducing parentheses

Conclusion

This chapter provided a comprehensive examination of Haskell's syntax through detailed examples and explanations. Key concepts included module structure, type declarations, function definitions with recursion and pattern matching, list processing, and control structures. These foundations enable the construction of type-safe, expressive programs characteristic of the Haskell paradigm. Subsequent chapters will build upon this base to explore type classes, monadic computation, and advanced functional patterns.