

HASKELL-patterns

pat → var
| -
| integer
| float
| char
| string
| (constr pat₁ ... pat_n)
| var@pat
| (var + integer)
| [pat₁, ..., pat_n], $n \geq 0$
| (pat₁, ..., pat_n), $n \geq 0$

HASKELL-types

type → (tyconstr type₁ ... type_n), $n \geq 0$
| [type]
| (type₁ → type₂)
| (type₁, ..., type_n), $n \geq 0$
| var

tyconstr → string starting with upper case symbol

Top declarations and type introduction

topdecl \rightarrow decl

| **type** tyconstr var₁ \dots var_n = type, $n \geq 0$

| **data** tyconstr var₁ \dots var_n =

constr₁ type_{1,1} \dots type_{1,n₁} |
⋮

constr_k type_{k,1} \dots type_{k,n_k},

$n \geq 0, k \geq 1, n_i \geq 0$

Definition of simple data structures

```
data Color = Red | Yellow | Green
```

```
data MyBool = MyTrue | MyFalse
```

```
traffic_light :: Color -> Color
```

```
traffic_light Red = Green
```

```
traffic_light Green = Yellow
```

```
traffic_light Yellow = Red
```

```
und :: MyBool -> MyBool -> MyBool
```

```
und MyTrue y = y
```

```
und _ _ = MyFalse
```

Definition of natural numbers

```
data Nats = Zero | Succ Nats
```

```
plus :: Nats -> Nats -> Nats
```

```
plus Zero      y = y
```

```
plus (Succ x) y = Succ (plus x y)
```

```
half :: Nats -> Nats
```

```
half Zero      = Zero
```

```
half (Succ Zero) = Zero
```

```
half (Succ (Succ x)) = Succ (half x)
```

Definition of lists

```
data List a = Nil | Cons a (List a)
```

```
len :: List a -> Nats
```

```
len Nil = Zero
```

```
len (Cons x xs) = Succ (len xs)
```

```
append :: List a -> List a -> List a
```

```
append Nil ys = ys
```

```
append (Cons x xs) ys = Cons x (append xs ys)
```