Operators [*H2010* 3]

- Expressions can built up as expected in many programming languages

  `3`   `x`   `x+y`   `(x<=y)`   `a + c * d - (e * (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 `== /= < <= >= >` :
  `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`"

The Haskell Prelude [*H2010* 9]

- The "`Standard Prelude`" is a library of functions loaded automatically (by default) into any Haskell program.
- Contains most commonly used datatypes and functions
- [*H2010* 9] is a *specification* of the Prelude
  the actual code is compiler dependent

Prelude extracts (I)

- Infix declarations

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

Higher precedence numbers bind tighter.
Function application binds tightest of all

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
(^)       :: (Num a, Integral b) => a -> b -> a
(^^)      :: (Fractional a, Integral b) => a -> b -> a
```

The `Num`, `Integral` and `Fractional` annotations have to do with *type-classes* — see later.

Prelude extracts (III)

- Boolean Type & Functions

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

Prelude extracts (IV)

- 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]
elem   :: Eq a => a -> [a] -> Bool
```

Prelude extracts (V)

- 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
```

We will re-visit these later — note that type-polymorphism here means that the possible implementations of some of these are extremely constrained!

Lists [*H2010* 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 [*H2010* 3.11])
- Strings are a special notation of lists of characters
  `"Hello"` for `['H','e','l','l','o']`

Function: `head`

`head xs` returns the first element of `xs`, if non-empty

Type Signature

```
head :: [a] -> a
```

Non-Empty List

```
head (x:_) = x
```

Empty List

```
head [] = error "Prelude.head: empty list"
```

We have to fail in the last case because there is no way to generate a value `v` of type `a`, where `a` can be any possible type, if there is no such value input to the function.
Empty list `[]`, of type `a`, contains no value of type `a` !

Undefinedness in Haskell

- Sometimes a Haskell function is *partial*:
  it doesn't return a value for some input, because it can't without violating type restrictions.
- Haskell provides two ways to explicitly define such a undefined "value":

  ```
  undefined :: a
  error :: String -> a
  ```

  Evaluating either of these results in a run-time error
- There are two ways in which "undefined" can occur implicitly:
  - If we use pattern-matching that is incomplete, so that some input values fail to match.
  - if a recursive function fails to terminate
- When talking about the meaning of Haskell, it is traditional to use the symbol $\perp$, a.k.a. "bottom", to denote undefinedness.

Why Not define a default value for `head []`?

- Why don't we define a default value for each type (`default ::a`) so that we can define (possible using Haskell classes):

  ```
  head [] = default -- for any given type a
  ```

  rather than having head $[] = \perp$?
- Why not have `default` for `Int` equal to 0?
- A key design principle behind Haskell libraries and programs is to have programs (functions!) that obey nice obvious laws:

  ```
  xs = head xs : tail xs
  sum (xs ++ ys) = sum xs + sum ys
  product (xs ++ ys) = product xs * product ys
  ```

- Consider the product law if `default = 0` and `xs = []`, and assume that both `sum` and `product` use `default` for the empty list case. Lefthand side is then `product ys` while thre righthand side is zero.

Function: `tail`

`tail xs`, for non-empty `xs` returns it with first element removed

Type Signature

```
tail :: [a] -> [a]
```

Non-Empty List

```
tail (_:xs) = xs
```

Empty List

```
tail [] = error "Prelude.tail: empty list"
```

Here again, we have tail $[] = \perp$.

`tail [] /= []` — Why Not?

- Why don't we define `tail [] = []`? The typing allows it.
- Re-consider the following law, given `xs = tail []`

```
xs = head xs : tail xs
```

- ```
        tail [] = head ( tail[]) : (tail : tail [])
  =   [] = head [] : tail []
  =   [] = ⊥ : []
  ```
  We have managed to show that the empty list is the same as a singleton list containing an undefined element.
- "Obvious" fixes can have unexpected consequences.

Function: `last`

`last xs` returns the last element of `xs`, if non-empty

Type Signature

```
last  ::  [a] -> a
```

Singleton List

```
last [x]  =  x
-- must occur before (_:xs) clause
```

Non-Empty List

```
last (_:xs)  =  last xs
```

Empty List

```
last []  =  error "Prelude.last: empty list"
```

Function: `init`

`init xs`, for non-empty `xs` returns it with last element removed

Type Signature

```
init  ::  [a] -> [a]
```

Singleton List

```
init [x]  =  []
```

Non-Empty List

```
init (x:xs)  =  x : init xs
```

Empty List

```
init []  =  error "Prelude.init: empty list"
```
```