1.2. Введение в язык *Haskell*

История языка *Haskell*

- начало 1930-х: Church, формализация функций в λ-исчислении
- 1960: John McCarthy, LISP – первый функциональный язык программирования
- 1978: John Backus, FP – система комбинаторного программирования
- конец 1970-х: Edinburgh univ., ML – meta-language
- 1985-1986: David Turner, Miranda – функциональный язык с «ленивыми» вычислениями
- 1990: Ericsson, Erlang – «коммерческий» функциональный язык
- 1988: Paul Hudak, Haskell – первая версия языка Haskell
- 1999: Haskell group, Haskell'98 – «стандартная» версия языка Haskell
  http://haskell.org/onlinereport/index.html - пересмотренное сообщение о языке Haskell'98
  http://haskell.org/tutorial - «Нежное» введение в Haskell

Haskell – чисто функциональный язык программирования, названный в честь Хаскелла Карри (Haskell B. Curry – 1900-1982), известного, главным образом, благодаря работам в области математической логики и комбинаторной логики в конце 1950-х – начале 1960-х годов

Типы данных и базовые конструкции языка *Haskell*

Элементарные типы данных

- `Integer, Int` – целые значения (25, -17, 111222333444555666777888)
- `Float, Double` – вещественные значения (3.14, -2.718281828459045)
- `Char` – символьные значения (`'A', '*', '3'`)
- `Bool` – логические значения (`True, False`)

Идентификаторы: `fact, fAcToRiAl, fact_1, fact''`

Знаки операций: `+, -, *, <, ==`

Идентификаторы применяются для обозначения констант – значений разных типов (простых, составных, функций) и типов. Любому идентификатору можно сопоставить тип и значение:

```
school :: Integer
school = 366

piHalf :: Double
piHalf = 3.1415926536 / 2
```

Кортежи и функции

Значения могут объединяться в более сложные с помощью *кортежирования*.

Например:

```
pair :: (Double, Double)
pair = (2.7, 3.14)

attributes :: (Char, (Int, Int, Int), Bool)
attributes = ('M', (17, 4, 1955), True)
```

Тип функции определяется типами аргументов и результата, например:

```
sin :: Double -> Double          -- аргумент и результат типа Double
plusInt :: Int -> Int -> Int     -- два аргумента типа Int, результат
Int
divMod :: (Int, Int) -> (Int, Int)  -- аргумент и результат - кортежи
```

Выражения составляются из констант применением операций и функций, например:

```
result = sin (3.1416 / 4) - 2.5
c10 = 3 + plusInt 3 4
pair = divMod (1458, plusInt 176 192)
```

Операции и функции отличаются только формой записи. Следующие выражения эквивалентны:

```
3 + 8                    и        (+) 3 8
27 `div` 4               и        div 27 4
7 `plusInt` 11           и        plusInt 7 11
```

Определение функций с помощью уравнений

Уравнения задают правила, по которым происходит вычисление функции, то есть каким образом результат получается из аргументов функции, например:

```
plusInt :: Int -> Int -> Int
plusInt a b = a + b

divMod :: (Int, Int) -> (Int, Int)
divMod (a, b) = (a `div` b, a `mod` b)
```

Уравнения могут содержать условные выражения и рекурсивные обращения, например:

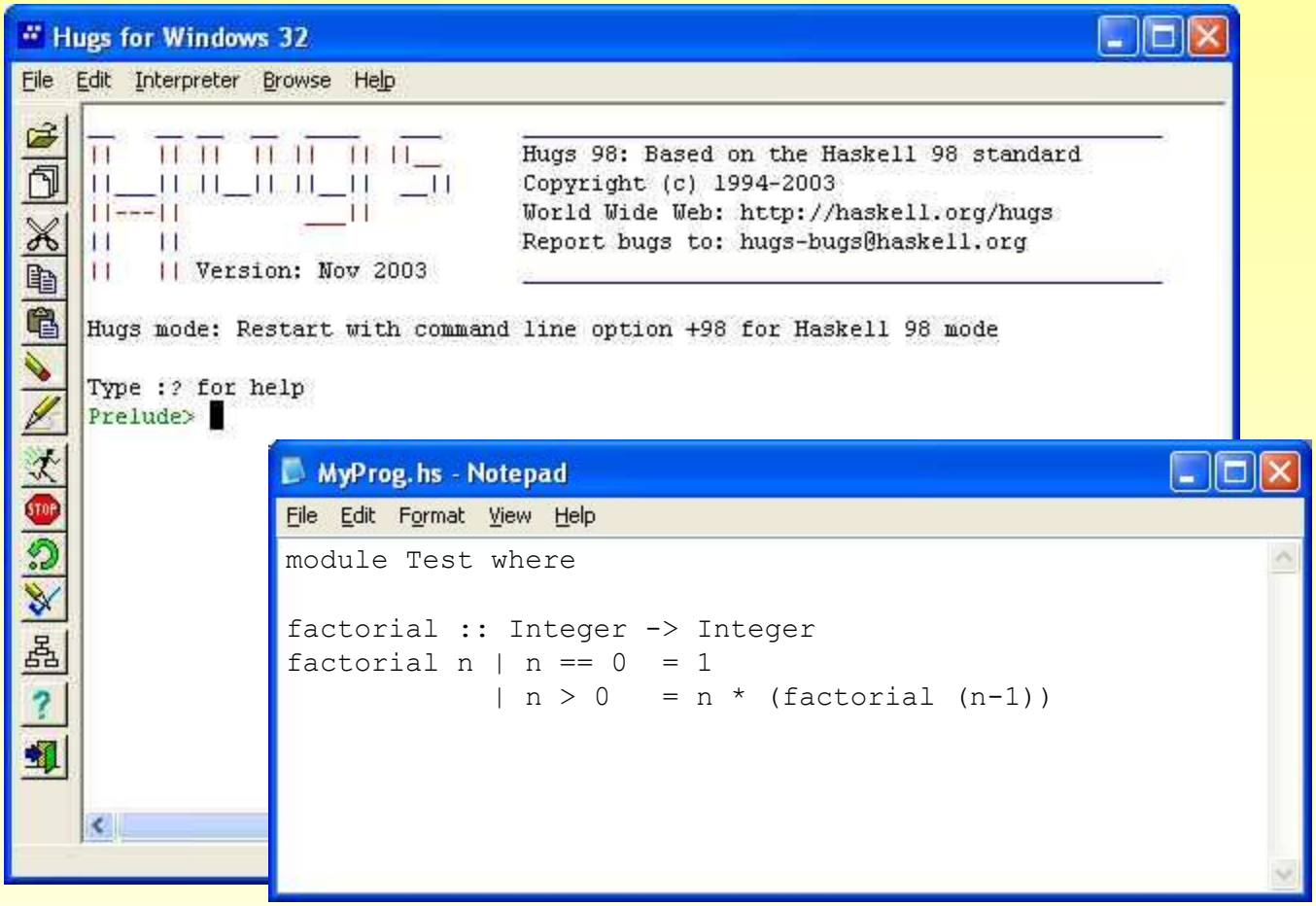
```
factorial :: Integer -> Integer
factorial n = if n == 0 then 1
                        else n * (factorial (n-1))

sum :: Integer -> Integer
sum n = n + if n == 0 then 0 else sum (n-1)
```

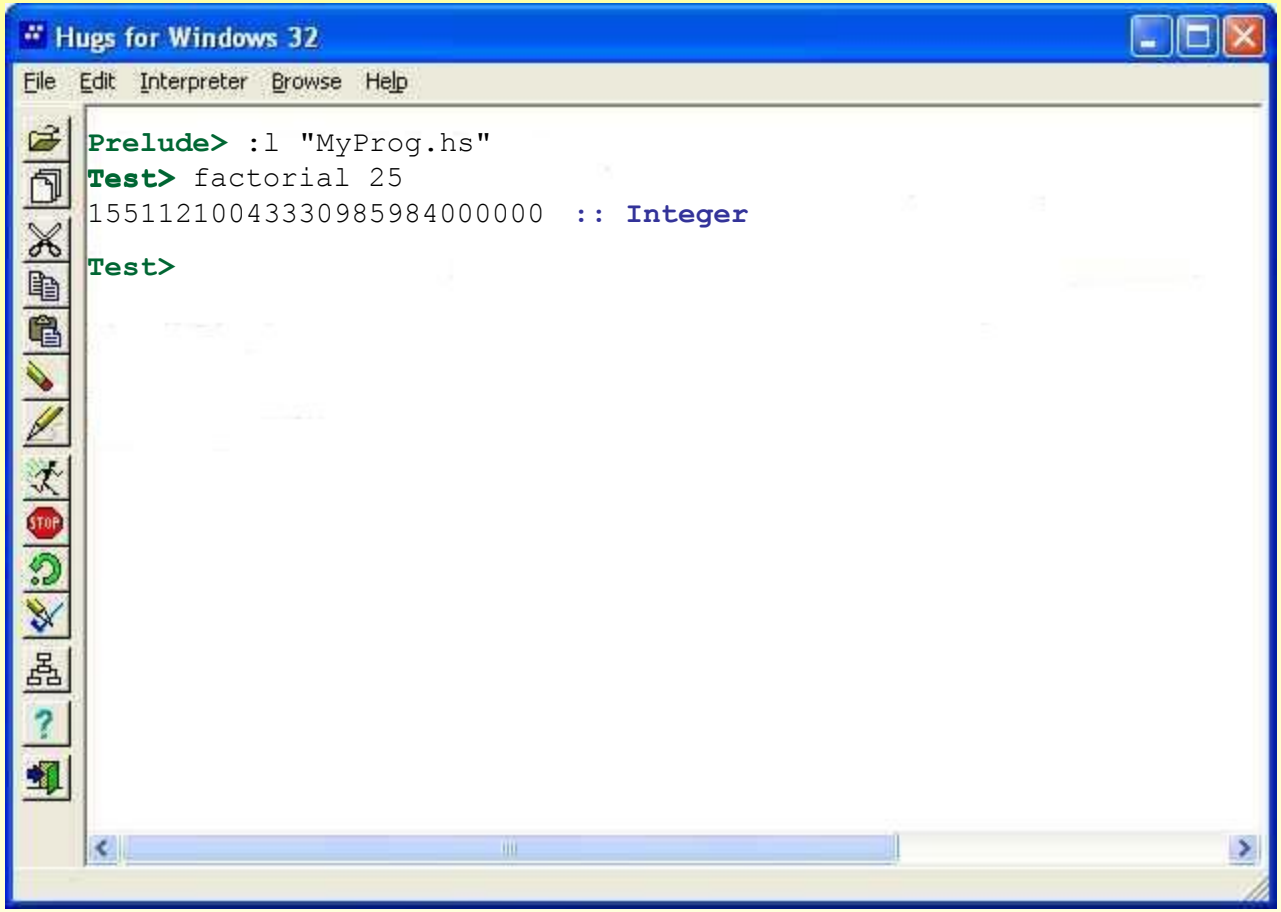
Уравнений для одной функции может быть несколько, тогда аргументы последовательно *сопоставляются с образцами*:

```
factorial2 :: Integer -> Integer
factorial2 0 | n == 0   = 1
factorial2 n | n > 0 = n * (factorial2 (n-1))
```

Подготовка и запуск программ



Пример запуска программы на исполнение



Исполнение программ с помощью текстовой подстановки

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


factorial 3
3 * (factorial (3-1))
3 * (factorial 2)
3 * (2 * (factorial (2-1)))
3 * (2 * (factorial 1))
3 * (2 * (1 * (factorial (1-1))))
3 * (2 * (1 * (factorial 0)))
3 * (2 * (1 * 1))
6
```

Несколько определений простых арифметических функций

```
-- Вычисление наибольшего общего делителя двух натуральных чисел
gcd              :: Integer -> Integer -> Integer
gcd m n | m < n  =  gcd n m
        | n < 0  =  error "gcd: Wrong argument"
gcd m 0          =  m
gcd m n          =  gcd n (m `mod` n)


-- Проверка заданного натурального числа на простоту
prime             :: Integer -> Bool
prime'            :: Integer -> Integer -> Bool
prime p    | p <= 0          =  error "prime: Non-positive argument"
           | otherwise       =  prime' 2 p
prime' d p | d * d > p       =  True
           | p `mod` d == 0  =  False
           | otherwise       =  prime' (d+1) p
```

Эффективность рекурсивных функций.

$f_1 = f_2 = 1$

$f_n = f_{n-1} + f_{n-2}$ при n > 2

```
-- Вычисление числа Фибоначчи, заданного порядковым номером
fib        :: Integer -> Integer
fib 1      =  1
fib 2      =  1
fib n      =  fib (n-1) + fib (n-2)
```

```
fib 6
fib 5 + fib 4
(fib 4 + fib 3) + fib 4
((fib 3 + fib 2) + fib 3) + fib 4
(((fib 2 + fib 1) + fib 2) + fib 3) + fib 4
(((1 + 1) + 1) + (fib 2 + fib 1)) + fib 4
(3 + 2) + (fib 3 + fib 2)
(3 + 2) + ((fib 2 + fib 1) + 1)
(3 + 2) + ((1 + 1) + 1)
8
```

Эффективность рекурсивных функций. Концевая рекурсия.

```
fib   :: Integer -> Integer
fib'  :: Integer -> Integer -> Integer -> Integer -> Integer
fib'  n k fk fk1 | k == n  =  fk
                 | k < n   =  fib' n (k+1) (fk+fk1) fk
fib 1 = 1
fib n = fib' n 2 1 1

fib 6
fib' 6 2 1 1
fib' 6 3 2 1
fib' 6 4 3 2
fib' 6 5 5 3
fib' 6 6 8 5
8

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

factorial  :: Integer -> Integer
factorial' :: Integer -> Integer -> Integer
factorial n = factorial' n 1              -- (factorial' n f) == (f * n!)
factorial' n f | n == 0  =  f
               | n > 0   =  factorial' (n-1) (n*f)
```