



PSC 2021/22 (375AA, 9CFU)

Principles for Software Composition

Roberto Bruni

<http://www.di.unipi.it/~bruni/>

<http://didawiki.di.unipi.it/doku.php/magistraleinformatica/psc/start>

II - Haskell

Lambda notation, again

Bound variables

```
int f(int x) { return x^2 + 2*x + 5 }
```

```
int f(int y) { return y^2 + 2*y + 5 }
```

$$f \triangleq \lambda x. x^2 + 2x + 5$$
$$f \triangleq \lambda y. y^2 + 2y + 5$$

```
let f x = x^2 + 2*x + 5
```

```
let f y = y^2 + 2*y + 5
```

they are all the same!

names of local variables are not important:

alpha-conversion

Free variables

$$x^2 + 2x + 5$$

they are not the same!

$$y^2 + 2y + 5$$

names of global variables matter

$$\lambda x. x^2 + 2x + 5$$

the same enclosing context

$$\lambda x. y^2 + 2y + 5$$

can make a difference

$$\lambda x. t$$

we say it binds the occurrences of x in t

$$\lambda x. x^2 + 2z + 5$$

$$\lambda y. y^2 + 2z + 5$$

$$\lambda z. z^2 + 2z + 5$$

are they all equivalent
(by alpha-conversion)?

Free variables: formally

$$t ::= x \mid \lambda x. t \mid t t \mid \dots$$

$$\text{fv} : LTerms \rightarrow \wp(Var)$$

$$\begin{aligned} \text{fv}(x) &\triangleq \{x\} \\ \text{fv}(\lambda x. t) &\triangleq \text{fv}(t) \setminus \{x\} \\ \text{fv}(t_1 t_2) &\triangleq \text{fv}(t_1) \cup \text{fv}(t_2) \end{aligned}$$

$$\text{fv}(\lambda x. x^2 + 2z + 5) = \{z\}$$

$$\text{fv}(\lambda y. y^2 + 2z + 5) = \{z\}$$

$$\text{fv}(\lambda z. z^2 + 2z + 5) = \emptyset$$

Alpha-conversion, again

$$\lambda x. t \equiv \lambda y. (t^{[y/x]}) \quad \text{if } y \notin \text{fv}(\lambda x. t)$$

$$\lambda x. x^2 + 2z + 5 \equiv \lambda y. ((x^2 + 2z + 5)^{[y/x]}) = \lambda y. y^2 + 2z + 5$$

$$\lambda x. x^2 + 2z + 5 \not\equiv \lambda z. ((x^2 + 2z + 5)^{[z/x]}) \text{ because } z \in \text{fv}(\lambda x. x^2 + 2z + 5)$$

Beta rule, again

$$(\lambda x. t) e \equiv t[e/x]$$

how is (capture-avoiding) substitution defined?
and why is it called “capture-avoiding”?

Capture-avoiding substitution

Substitution, 1st try

$$y[e/x] \triangleq \begin{cases} e & \text{if } y = x \\ y & \text{otherwise} \end{cases}$$

$$(\lambda y. t)[e/x] \triangleq \begin{cases} \lambda y. t & \text{if } y = x \\ \lambda y. (t[e/x]) & \text{otherwise} \end{cases}$$

$$(t_1 t_2)[e/x] \triangleq t_1[e/x] (t_2[e/x])$$

$$t_1 \triangleq \lambda x. \lambda y. x^2 + 2y + 5$$

$$t_2 \triangleq y$$

$$t_1 t_2 \equiv (\lambda x. \lambda y. x^2 + 2y + 5) y$$

$$\equiv (\lambda y. x^2 + 2y + 5)[y/x]$$

$$\equiv \lambda y. ((x^2 + 2y + 5)[y/x])$$

$$\equiv \lambda y. y^2 + 2y + 5$$

captured variable!

free

Capture-avoiding

free variables occurring in e
should remain free after the application of $[^e/x]$

solution: alpha-convert before substituting!

$$\begin{aligned}(\lambda y. x^2 + 2y + 5)[^y/x] &\equiv (\lambda z. (x^2 + 2y + 5)[^z/y])[^y/x] \\ &\equiv (\lambda z. x^2 + 2z + 5)[^y/x] \\ &\equiv \lambda z. ((x^2 + 2z + 5)[^y/x]) \\ &\equiv \lambda z. y^2 + 2z + 5\end{aligned}$$

↙ free

Substitution, 2nd try

$$y[e/x] \triangleq \begin{cases} e & \text{if } y = x \\ y & \text{otherwise} \end{cases}$$

$$(\lambda y. t)[e/x] \triangleq \begin{cases} \lambda y. t \\ \lambda z. (t[z/y][e/x]) \end{cases}$$

superfluous: no free
if $y = x$ occurrences to replace
otherwise, with
 $z \notin \text{fv}(e) \cup \text{fv}(\lambda y. t) \cup \{x\}$

$$(t_1 t_2)[e/x] \triangleq t_1[e/x] (t_2[e/x])$$

Substitution, final

$$y[e/x] \triangleq \begin{cases} e & \text{if } y = x \\ y & \text{otherwise} \end{cases}$$

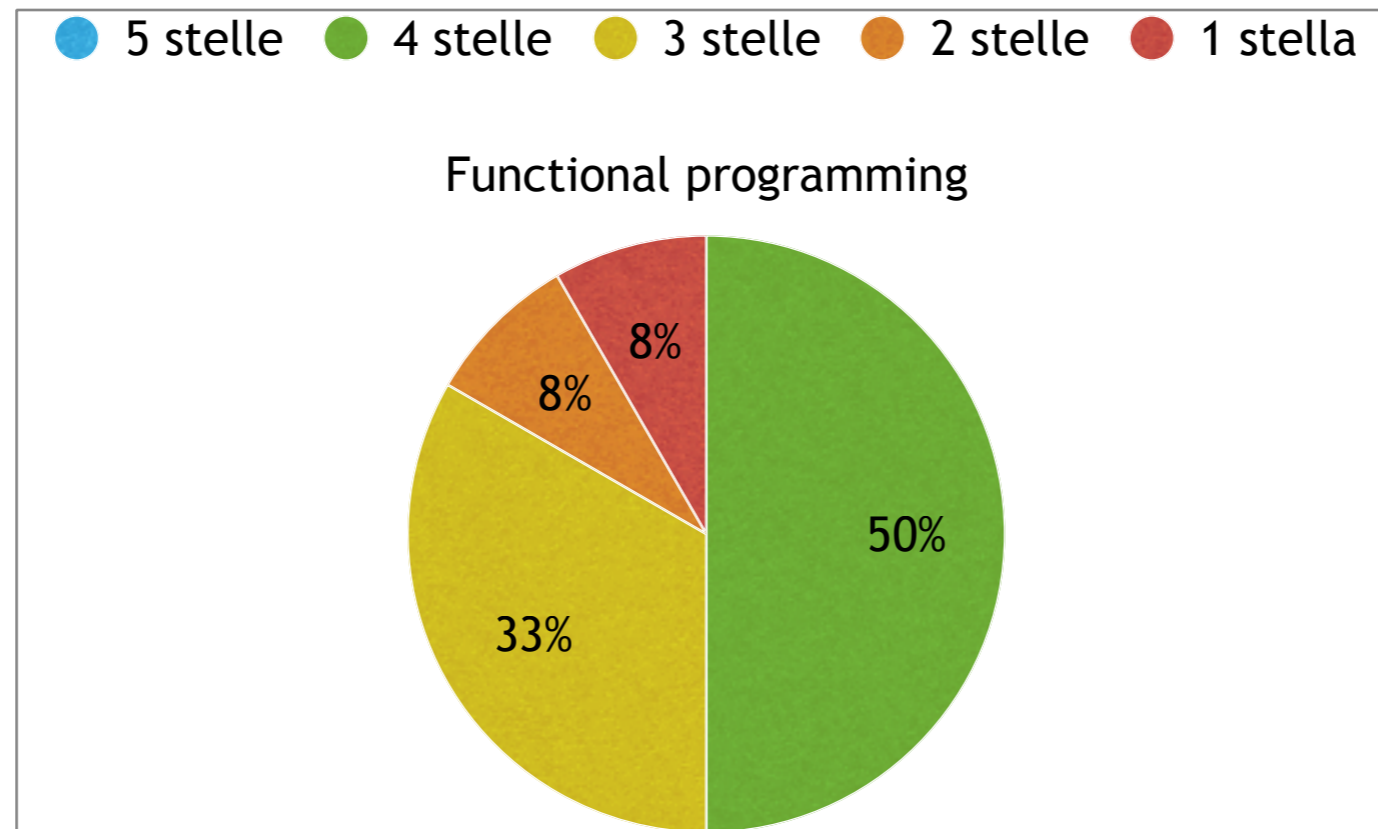
$$(\lambda y. t)[e/x] \triangleq \lambda z. (t[z/y][e/x]) \quad \text{with } z \notin \text{fv}(e) \cup \text{fv}(\lambda y. t) \cup \{x\}$$

$$(t_1 t_2)[e/x] \triangleq t_1[e/x] (t_2[e/x])$$

Higher Order Functional Languages

Haskell

From your forms



(over 12 answers)

Imperative vs Functional

Imperative style

tell the machine how to compute;
a sequence of tasks to execute;
manipulation of mutable states

Purely functional
style

tell the machine what to compute;
declarative style;
define what functions are,
not how to compute them;
functions have no side effects;
can't set and change variable's content;
manipulation of values

Declarative style

Any experience of functional programming?

Have you ever used a spreadsheet?

The value of a cell is defined in terms of those of other cells:
what is to be computed, not how it must be computed

we do not specify the order in which cells are calculated:
cells are computed according to their dependencies

we do not decide how to allocate memory:
only those cells which are in use are allocated

we specify the value of a cell by an expression:
its parts can be evaluated in any order

Functional style: HO

Higher-Order:

functions as values,
functions as parameters,
functions are returned,
functions are composed

how many elements of a
list will pass the test?

length (filter test xs)

a list in T^*

a predicate in $T \rightarrow \text{Bool}$

a function in $(T \rightarrow \text{Bool}) \rightarrow T^* \rightarrow T^*$

a function in $T^* \rightarrow \text{Int}$

Purity: no side effects

the result of a function is determined only by its input

a variable is just a name bound to some (HO) value:
shorthands for expressions

variables do not vary

programs are typically shorter, maybe less efficient;
closer to semantics, ease verification of correctness;
more robust, easier to maintain

Haskell: a purely functional programming language

<http://www.haskell.org/>

[Downloads](#)

[Community](#)

[Documentation](#)



An advanced, purely functional programming language

Declarative, statically typed code.

```
primes = filterPrime [2..]
  where filterPrime (p:xs) =
        p : filterPrime [x | x <- xs, x `mod` p /= 0]
```

Try it!

Type Haskell expressions in here.

λ

Got 5 minutes?

Type `help` to start the tutorial.

Or try typing these out and see what happens (click to insert):

```
23 * 36 or reverse "hello" or foldr (:) [] [1,2,3] or do line <- getLine;
putStrLn line or readFile "/welcome"
```

These IO actions are supported in this sandbox.

Haskell: origins

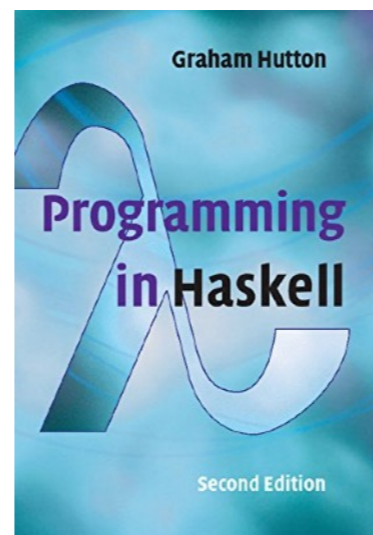
named after mathematical logician Haskell B. Curry

1987: Haskell project begun

1998: first version appear

2003: the Haskell Report was published
(first stable version)

Graham Hutton, “Programming in Haskell”, ch.1-8,14,15



Features

- Referential transparency** if a function is called twice with the same argument, it returns the same result; compiler can reason on program's behaviour; one can deduce a function is correct and build more complex functions by composition
- Statically typed** type inference: you don't have to label all data, their types can be figured out; many possible errors are caught at compile time
- Polymorphism** one definition of function works for many types
- Overloading** different definitions of the same function-name for different types
- Laziness** calculation starts only if some result is needed; infinite data structures can be manipulated

I'M NOT LAZY



**I'M JUST SAVING MY
ENERGY FOR WHEN I
REALLY NEED IT.**

More features (less bugs)

Purity: no side effects

Typeful: types are pervasive, no dubious use of types

Concise: shorter programs, less typing (on the keyboard)

High level: closer to the algorithm description

Memory managed: programmers can focus on the algorithm

Compositionality: solve problems by composing functions that solve smaller problems

Data encapsulation and polymorphism not exclusive to OOP: modules and type classes

A taste of Haskell

math. notation

$$f(x) = 2x + 3$$

$$g(x, y) = x^2 + xy + y^2$$

$$abs(x) = \begin{cases} x & \text{if } x \geq 0 \\ -x & \text{otherwise} \end{cases}$$

$$abs(f(g(2, 3)))$$

set comprehension

$$\{x \mid x \in X \wedge f(x) > 5\}$$

Haskell notation

$$f\ x = 2*x + 3$$

$$g\ (x, y) = x^2 + x*y + y^2$$

$$abs\ x \begin{cases} | & x \geq 0 & = & x \\ | & otherwise & = & -x \end{cases}$$

$$abs\ (f\ (g\ (3, 2)))$$

list comprehension

$$[x \mid x \leftarrow X, f\ x > 5]$$

The power of recursion

No assignments: no loops

(loops over lists exist: *list comprehension*)

Recursion is used in place of loops

```
power2 n
| n==0 = 1
| n>0  = 2 * power2(n-1)
```

Haskell: some principles

evaluate *expressions* (syntactic terms)

to yield *values* (abstract entities regarded as answers)

every value has an associated *type*

the association is called *typing*

you can think of types as sets of values

as expressions denote values

types are denoted by type expressions

values are first-class (passed around, returned as results)

types are not first-class

Haskell: GHCi

Interactive shell or interpreter, executing read-eval-print loop

programmers enter expressions/declarations one at a time

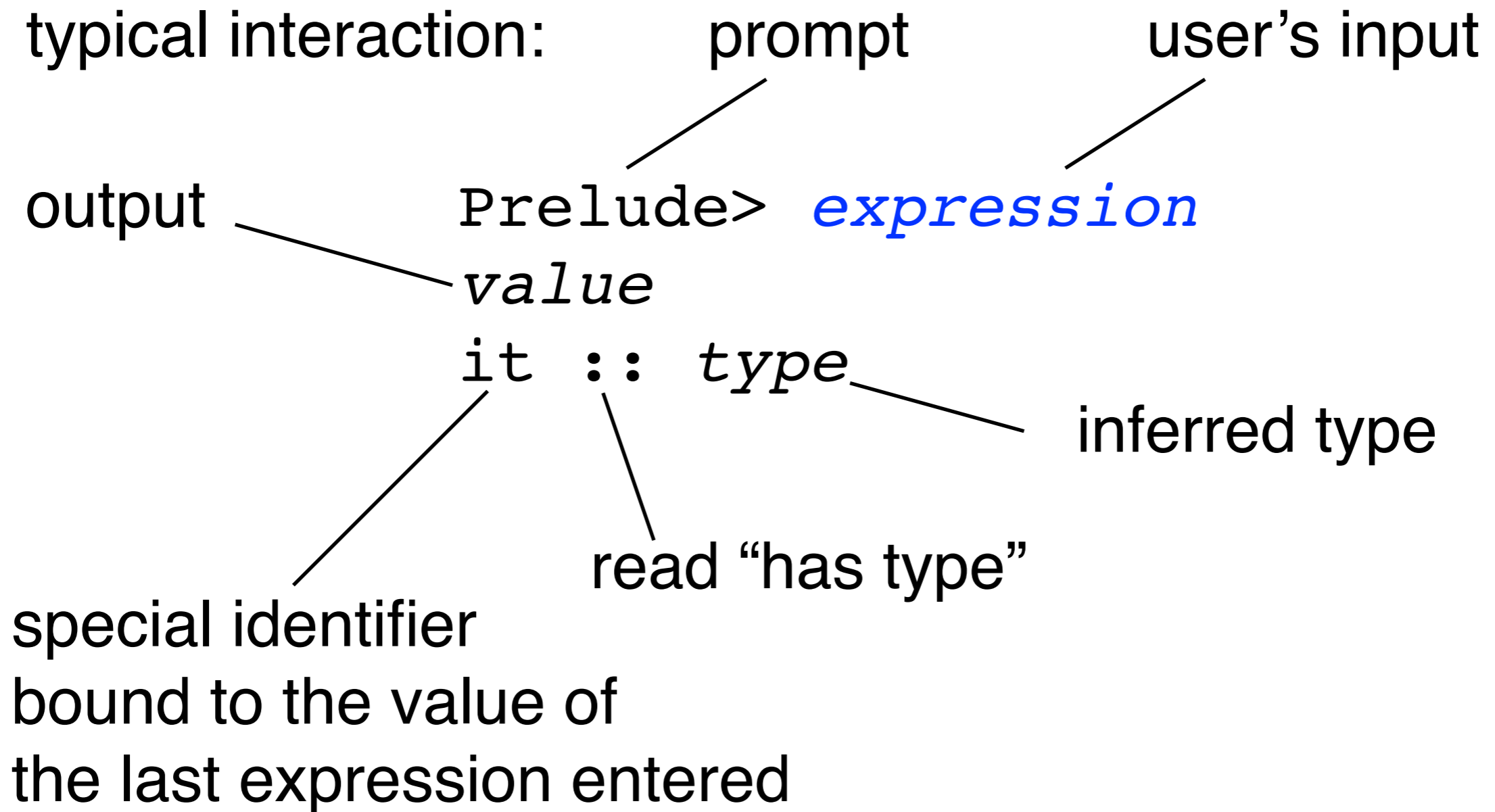
they are type checked, compiled and executed

if an expression does not parse correctly

or does not pass the type-checking phase of the compiler,
no code is generated and no code is executed

once an identifier is defined it is available at subsequent lines

GHCi expressions



GHCi declarations

typical interaction:

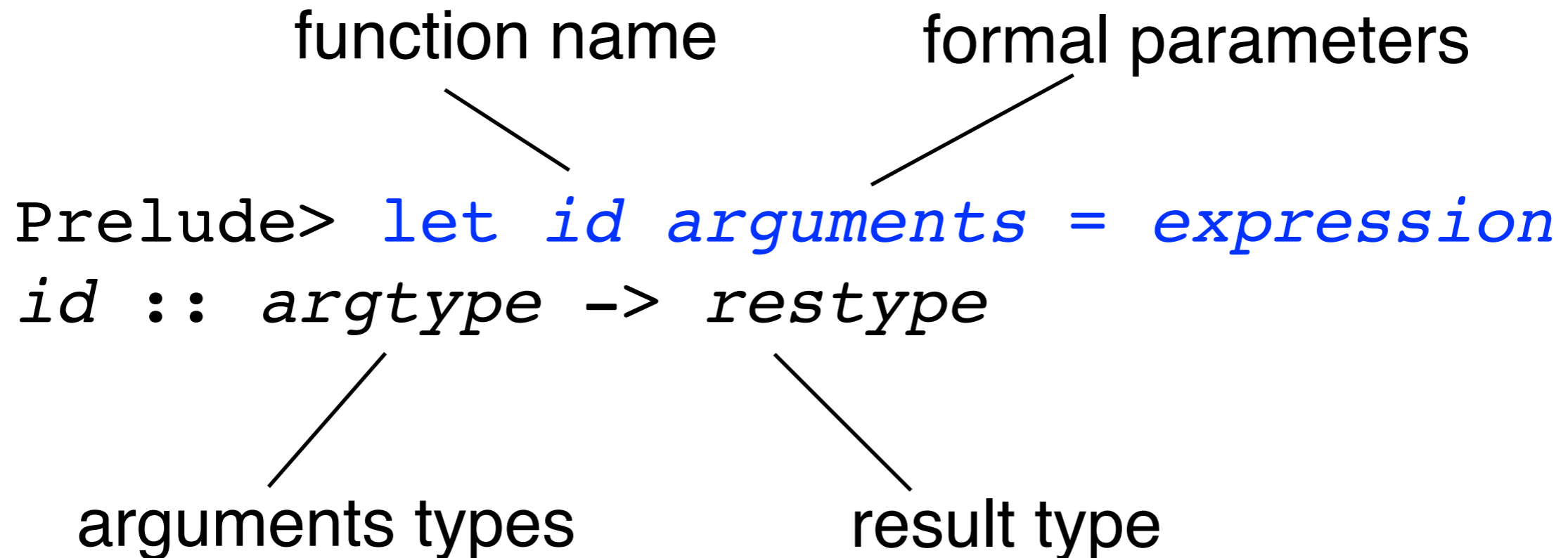
```
Prelude> let id = expression  
id :: type
```

keyword

defining symbol

GHCi declarations

more generally:



GHCi session

A terminal window titled 'bruni' with a path to the GHC framework. The window shows the output of running 'ghci', including the login time, the command executed, the GHCi version, and the prelude prompt.

```
bruni — ghc -B/Library/Frameworks/GHC.framework/Versions/8.6.3-x86_64/usr/li...
Last login: Wed Mar 18 11:13:21 on ttys000
[Cat:~ bruni$ ghci
GHCi, version 8.6.3: http://www.haskell.org/ghc/  :? for help
Prelude> ]
```