Principles of Programming Languages

Graham Hutton

Department of Computer Science
University of Nottingham

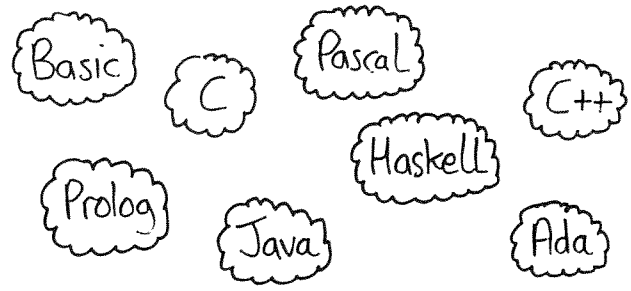
Principles of Programming Languages

Graham Hutton

Lecture 1 - Introduction

BACKGROUND

There are many different programming languages.



Most courses focus on practical aspects of programming in particular languages, such as:

- Language features;
- Algorithms and data structures;
- Space and time efficiency;
- Design methods.

This course takes a more abstract approach, focussing on some of the mathematical concepts underlying programming languages in general:

- Syntax;
- Operational semantics;
- Denotational semantics;
- Lambda calculus;
- Type theory;
- Domain theory.

The central topic of the course is semantics.

WHAT IS SEMANTICS?

Here are a few definitions:

- Oxford Minidictionary:

  "Study of meaning";

- Collins Gem English Dictionary:

  "Study of linguistic meaning";

- Webster's New Dictionary:

  "Science of the meaning of words".

WHERE IS SEMANTICS STUDIED?

Here are the three main areas:

- Linguistics and Philosophy:

  The meaning of sentances in a natural language (e.g. English);

- Mathematics:

  The meaning of terms in a formal logic (e.g. predicate logic.);

✸ Computing Science

  The meaning of programs in a programming language (e.g. C.).

WHY SEMANTICS MATTERS

Program semantics are useful in a number of areas of computing, including:

- Documentation

  A formal semantics provides a complete and precise specification of a language for programmers and compiler writers.

- Formal reasoning

  A formal semantics is a prerequisite for proofs about a programming language and programs written in it.

- Language design

  Semantic issues have a growing influence on the design of programming languages.

- Education

  A knowledge of semantics gives a deeper understanding of programming languages, which will be useful when learning new languages, and when comparing different languages.

- Research

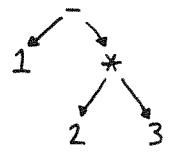
  The elegance of semantics may inspire you to think about applying for a Ph.D. position in programming language research.

COURSE TOPICS

① Syntax

The syntactic structure of programs as trees is specified by a grammar, e.g.:

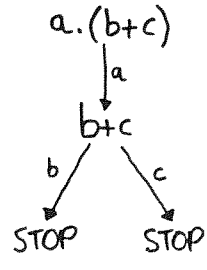
$1 - 2 * 3$ means



② Operational semantics

The meaning of programs as transition systems is specified by transition rules, e.g.:

$a.(b+c)$ means



③ Denotational semantics

The meaning of programs as mathematical functions is specified by recursion equations, e.g.:

$$x := 1 \quad \text{means} \quad f(s) = \text{update}(s, x, 1).$$

④ Lambda calculus

A theory of functions used to study many foundational issues in semantics, e.g.:

$$\begin{aligned} \text{true}\ x\ y &= x \\ \text{false}\ x\ y &= y \\ \text{not}\ x &= x\ \text{false}\ \text{true}. \end{aligned}$$

⑤ Domain theory

Solves the foundational problems with the semantics of recursive types and values, e.g.:

```
data List     = Nil | Cons Int List

len           :: List → Int
len Nil       = 0
len (Cons n ns) = 1 + len ns .
```


COURSE MATERIAL

- Lecture notes: copies of my slides.
- Textbook: none required, but any book on the semantics, principles, or theory of programming languages will be useful for background reading.

ASSESSMENT

- Non-assessed exercises for each lecture;
- One written examination (100%).

HASKELL

Throughout the course, the language Haskell will be used for implementing semantic concept

- Haskell is well-suited to the kind of symbolic manipulation used in semantics;
- Implementing the concepts helps to make them easier to understand, and also allows them to be executed;
- Automatic type checking in Haskell helps to avoid mistakes in our definitions;
- Lecture 2 revises the basics of Haskell.

A TASTE OF SEMANTICS

In most languages, the Booleans (false and true) are built-in to the language.

Question: is this theoretically necessary?

Answer: no, they can be encoded in a simple way using functions!

The type of Booleans:

```
type Bool = a → a → a
```


The basic values:

```
true, false :: Bool
true  x y   = x
false x y   = y
```

The "logical negation" function:

```
not   :: Bool → Bool
not x = x false true
```

Examples:

```
not false = false false true
          = true

not true  = true false true
          = false
```


EXERCISES

① In a similar manner to the not example, define a "logical and" function:

```
and :: Bool → Bool → Bool
```

② Show that your definition has the expected behaviour for all possible combinations of Boolean arguments.

③ Repeat both exercises above for the "logical or" function:

```
or :: Bool → Bool → Bool
```

Principles of Programming Languages

Graham Hutton

Lecture 2 - Review of Haskell

INTRODUCTION

This course assumes some experience with the functional programming language Haskell.

For example, see:

> Lecture notes in functional programming,
> Graham Hutton, Univ. of Nottingham, 1997.

We'll be using the Hugs 1.3 version of Haskell, available for most systems from:

> http://www.cs.nott.ac.uk/
> Department/Staff/mpj/hugs.html.

This lecture reviews the basics of Haskell.

STARTING HUGS FROM UNIX

hugs	Start Hugs;
hugs ⟨filenames⟩	Start and load files.

USEFUL COMMANDS WITHIN HUGS

:l ⟨filenames⟩	Load files;
:r	Repeat last load;
:a ⟨filenames⟩	Load extra files;
:e ⟨filename⟩	Edit file;
:e	Edit last file;
:t ⟨expression⟩	Type an expression;
:?	List commands;
:q	Quit

LAYOUT

In a sequence of definitions, each definition must begin in precisely the same column.

```
a = b+c
      where
        b=10
        c=15-5
d = a*2
```

means

```
{a = b+c
      where
        {b=10;
         c=15-5
 d = a*2}
```

COMMENTS

```
-- A single-line comment.

{- A multiple-line comment,
   which can be nested.  -}
```

CONDITIONALS

```
abs x = if x >=0 then x else -x
```

GUARDS

```
signum x | x<0  = -1
         | x==0 = 0
         | x>0  = 1
```

PATTERN MATCHING

```
not False = True
not True  = False
```

CASE ANALYSIS

```
not x = case x of
          False → True
          True → False
```


LIST COMPREHENSIONS

```
pairs xs     = [(x,y) | x ← xs, y ← xs]
positives xs = [n | n ← ns, n>0]
concat xss   = [x | xs ← xss, x ← xs]
```

ABBREVIATIONS

`f x y z` means `((f x) y) z`

`a → b → c → d` means `a → (b → (c → d))`

`[1,2,3,4]` means `1:(2:(3:(4:[])))`

`"Hello"` means `['H','e','l','l','o']`

BASIC TYPES

Bool	Booleans: False True
Int	Integers: ... -1 0 1 ...
Char	Characters: ... 'a' 'b' ...
a b c ...	Variables

COMPOUND TYPES

[a]	Lists
(a,b)	Pairs
a → b	Functions

TYPE DEFINITIONS

```
type String = [Char]
type Table k v = [(k,v)]
```

DATA DEFINITIONS

```
data Bool   = False | True
data Result = Error | OK Int
data List a = Nil | Cons a (List a)
```

OVERLOADING

```
(+) :: Num a ⇒ a → a → a
(==) :: Eq a ⇒ a → a → Bool
```

RECURSION

```
fac       :: Int → Int
fac 0     = 1
fac (n+1) = (n+1) * fac n

length       :: [a] → Int
length []    = 0
length (x:xs) = 1 + length xs

flatten          :: Tree a → [a]
flatten (Leaf x) = [x]
flatten (Node l r) = flatten l ++ flatten r
```


INPUT/OUTPUT

The library file "IO.hs" defines:

IO a - the type of interactive programs that return results of type a.

together with:

```
getChar   :: IO Char
getLine   :: IO String
putChar   :: Char → IO ()
putStr    :: String → IO ()
putStrLn  :: String → IO ()
sequence  :: [IO a] → IO ()
return    :: a → IO a
```


Examples

```
getChar :: IO Char
getChar = do x ← getCh
             putChar x
             return x

putStr       :: String → IO ()
putStr []    = return ()
putStr (x:xs) = do putChar x
                   putStr xs

getLine :: IO String
getLine = do x ← getChar
             if x == '\n' then
                return ""
             else
                do xs ← getLine
                   return (x:xs)
```


LIBRARY FUNCTIONS

For reference, here are some of the most commonly used library functions:

Booleans

```
(&&) :: Bool → Bool → Bool
(||) :: Bool → Bool → Bool
not  :: Bool → Bool
```

Lists

```
(:)  :: a → [a] → [a]
(++) :: [a] → [a] → [a]
head :: [a] → a
tail :: [a] → [a]
last :: [a] → a
init :: [a] → [a]
```

```
(!!)     :: [a] → Int → a
null     :: [a] → Bool
length   :: [a] → Int
elem     :: Eq a ⇒ a → [a] → Bool
reverse  :: [a] → [a]
zip      :: [a] → [b] → [(a,b)]
```

Pairs

```
fst      :: (a,b) → a
snd      :: (a,b) → b
```

Higher-order functions

```
(.)      :: (b→c) → (a→b) → (a→c)
map      :: (a→b) → [a] → [b]
filter   :: (a→Bool) → [a] → [a]
```


EXAMPLE

Let us write a program for displaying a table showing the distribution of a list of integer exam marks in the range 0-99.

For example:

```
? showtable [57,68,41,61,...
 0:
10: *
20:
30: ****
40: ***********
50: **********************
60: ****************
70: ******
80: ***
90:
```


Displaying a mark:

```
showmark     :: Int → IO()
showmark 0 = putStr " 0"
showmark n = putStr (show n)
```

Displaying a line of a table:

```
showline          :: Int → [Int] → IO()
showline m ms =
  do showmark m
     putStr ": "
     putStrLn stars
  where
     stars = ['*' | _ ← [1..num]]
     num   = length (filter valid ms)
     valid x = (m <= x) && (x <= m+9)
```

Displaying a table:

```
showtable    :: [Int] → IO()
showtable ms = sequence [showline (m*10) ms | m ← [0..9]]
```


EXERCISES

1. Modify the showtable program to also display the min, max, and average mark.

2. Define as many of the library functions from slides 2.11 and 2.12 as you can, without looking up their definitions.

   Hint: when testing your definitions in Hugs, you'll need to rename the functions to avoid clashing with the built-in definitions.

Principles of Programming Languages

Graham Hutton

Lecture 3 - Syntax I

INTRODUCTION

At the lowest level, a program can be viewed simply as a string of characters.

In order to study the meaning of programs at a higher level, we first need to study how programs are formed as strings:

meaning = semantics;

form = syntax.

The next two lectures introduce the basic mathematics underlying program syntax.

LOGIC

We begin by reviewing the operators of propositional (or Boolean) logic:

Notation	Meaning
$A \wedge B$	"and";
$A \vee B$	"or";
$\neg A$	"not";
$A \Rightarrow B$	"implies";
$A \Leftrightarrow B$	"equivalent to".

Note: the $\Leftrightarrow$ operator is sometimes read as "if and only if", abbreviated by "iff".

Predicate logic extends propositional logic by introducing two quantifiers:

Notation	Meaning
$\forall x.\ P(x)$	"for all";
$\exists x.\ P(x)$	"there exists".

Examples

$\forall x.\ (Man(x) \Rightarrow Person(x))$ means

"Every man is a person";

$\exists x.\ (Person(x) \wedge \neg Man(x))$ means

"Not every person is a man".

SETS

A set is a collection of objects, such that:

1. The order of objects is not important;
2. Duplicate objects are not important.

The objects are usually called "elements" or "members".

Notation	Meaning
$\emptyset$	"empty set";
$\{a_1, a_2, \ldots, a_n\}$	"explicitly defined set";
$\{x \mid P(x)\}$	"implicitly defined set";
$x \in S$	"is a member of";
$x \notin S$	"is not a member of".

Useful operations on sets:

Union : $S \cup T = \{x \mid x \in S \vee x \in T\}$

Intersection : $S \cap T = \{x \mid x \in S \wedge x \in T\}$

Subset : $S \subseteq T \Leftrightarrow \forall x.\ x \in S \Rightarrow x \in T$

Equality : $S = T \Leftrightarrow S \subseteq T \wedge T \subseteq S$

Product : $S \times T = \{(x,y) \mid x \in S \wedge y \in T\}$

Powerset : $\mathbb{P}(S) = \{T \mid T \subseteq S\}$

We can also extend $\cup$ and $\cap$ to sets of sets:

$$\bigcup X = \{x \mid \exists S.\ S \in X \wedge x \in S\}$$

$$\bigcap X = \{x \mid \forall S.\ S \in X \Rightarrow x \in S\}$$

STRINGS

A string over a finite set S is a finite sequence of elements of S, with duplicates permitted.

Notation	Meaning
ε	"empty string";
$a_1 a_2 \ldots a_n$	"explicitly defined string";
$s \frown t$	"concatenation".

The underlying set of a string is usually called the "alphabet" of the string.

Examples

1011 and 00101 are both strings over the alphabet $\{0,1\}$ of binary digits.

Useful sets of strings

A^n : All strings of length n over the alphabet A;

A^+ : All non-empty strings over A;

A^* : All strings over A.

Examples

$\{0,1\}^0 = \{\varepsilon\}$

$\{0,1\}^1 = \{0,1\}$

$\{0,1\}^2 = \{00, 01, 10, 11\}$

$\{0,1\}^+ = \{0, 1, 00, 01, 10, 11, \ldots\}$

$\{0,1\}^* = \{\varepsilon, 0, 1, 00, 01, 10, 11, \ldots\}$