Programming Languages (CSCI 4430/6430)

History, Syntax, Semantics, Essentials, Paradigms

Carlos Varela
Rennselaer Polytechnic Institute

August 30, 2016

The first programmer ever

Ada Augusta, the Countess of Lovelace, the daughter of the poet Lord Byron

Circa 1843

Using Babbage’s Analytical Engine

The first “high-level” (compiled) programming language

FORTRAN

1954

Backus at IBM

It was called “an automatic coding system”, not a “programming language”

Used for numerical computing

The first functional programming language

Lisp

1958

McCarthy at Stanford

For LISts Processing---lists represent both code and data

Used for symbolic manipulation

The first object oriented programming language

Simula

1962

Dahl and Nygaard at University of Oslo, Norway

Used for computer simulations

The first logic programming language

Prolog

1972

Roussel and Colmerauer at Marseilles University, France

For "PROgrammation en LOGique".

Used for natural language processing and automated theorem proving

The first concurrent programming language

Concurrent Pascal

1974

Hansen at Caltech

Used for operating systems development

The first concurrent actor programming language

PLASMA

1975

Hewitt at MIT

Used for artificial intelligence (planning)

The first scripting language

REXX

1982

Cowlishaw at IBM

Only one data type: character strings

Used for “macro” programming and prototyping

The first multi-paradigm programming language

Oz

1995

Smolka at Saarland University, Germany

A logic, functional, imperative, object-oriented, constraint, concurrent, and distributed programming language

Used for teaching programming and programming language research

Other programming languages

Imperative

Algol (Naur 1958)
Cobol (Hopper 1959)
BASIC (Kennedy and Kurtz 1964)
Pascal (Wirth 1970)
C (Kernighan and Ritchie 1971)
Ada (Whitaker 1979)

Functional

ML (Milner 1973)
Scheme (Sussman and Steele 1975)
Haskell (Hughes et al 1987)

Object-Oriented

Smalltalk (Kay 1980)
C++ (Stroustrop 1980)
Eiffel (Meyer 1985)
Java (Gosling 1994)
C# (Hejlsberg 2000)

Actor-Oriented

Act (Lieberman 1981)
ABCL (Yonezawa 1988)
Actalk (Briot 1989)
Erlang (Armstrong 1990)
E (Miller et al 1998)
SALSA (Varela and Agha 1999)

Scripting

Python (van Rossum 1985)
Perl (Wall 1987)
Tcl (Ousterhout 1988)
Lua (Ierusalimschy et al 1994)
JavaScript (Eich 1995)
PHP (Lerdorf 1995)
Ruby (Matsumoto 1995)

Declarative Computation Model

Defining practical programming languages (CTM 2.1)

Carlos Varela
RPI
August 30, 2016

Adapted with permission from:
Seif Haridi
KTH
Peter Van Roy
UCL

Programming Concepts

- A computation model: describes a language and how the sentences (expressions, statements) of the language are executed by an abstract machine
- A set of programming techniques: to express solutions to the problems you want to solve
- A set of reasoning techniques: to reason about programs to increase the confidence that they behave correctly and to calculate their efficiency

Declarative Programming Model

- Guarantees that the computations are evaluating functions on (partial) data structures
- The core of functional programming (LISP, Scheme, ML, Haskell)
- The core of logic programming (Prolog, Mercury)
- Stateless programming vs. stateful (imperative) programming
- We will see how declarative programming underlies concurrent and object-oriented programming (Erlang, C++, Java, SALSA)

Defining a programming language

- Syntax (grammar)
- Semantics (meaning)

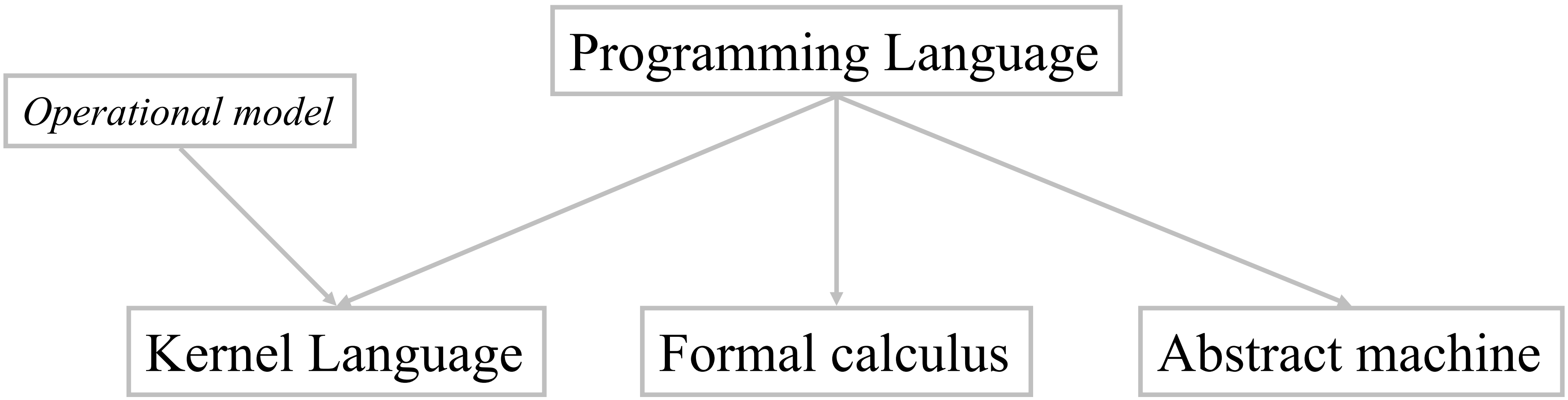
Language syntax

- Defines what are the legal programs, i.e. programs that can be executed by a machine (interpreter)
- Syntax is defined by grammar rules
- A grammar defines how to make ‘sentences’ out of ‘words’
- For programming languages: sentences are called statements (commands, expressions)
- For programming languages: words are called tokens
- Grammar rules are used to describe both tokens and statements

Language Semantics

- Semantics defines what a program does when it executes
- Semantics should be simple and yet allows reasoning about programs (correctness, execution time, and memory use)

Approaches to semantics



Aid the programmer in reasoning and understanding

Mathematical study of programming (languages) λ-calculus, predicate calculus, π-calculus

Aid to the implementer Efficient execution on a real machine

Programming Paradigms

- We will cover theoretical and practical aspects of three different programming paradigms:

Paradigm	Theory	Languages
Functional Programming	Lambda Calculus	Oz Haskell
Concurrent Programming	Actor Model	SALSA Erlang
Logic Programming	First-Order Logic Horn Clauses	Prolog Oz

- Each paradigm will be evaluated with a Programming Assignment (PA) and an Exam.
- Two highest PA grades count for 40% of total grade. Lowest PA grade counts for 10% of the total grade. Two highest Exam grades count for 40% of total grade. Lowest Exam grade counts for 10% of the total grade.

Lambda Calculus (PDCS 2)

alpha-renaming, beta reduction, applicative and normal evaluation orders, Church-Rosser theorem, combinators

Carlos Varela

Rennselaer Polytechnic Institute

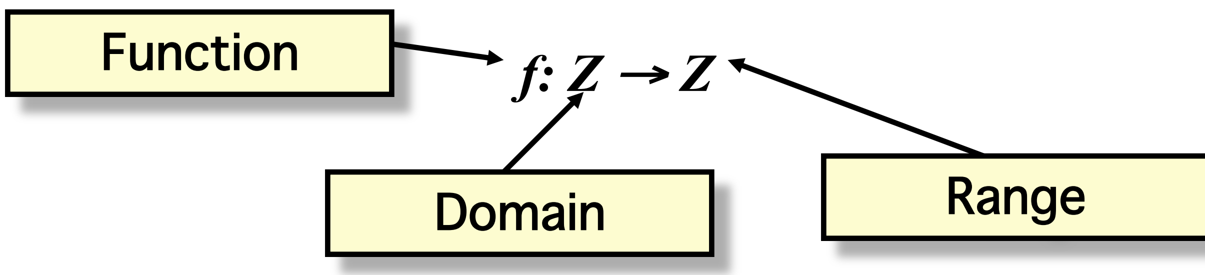
August 30, 2016

Mathematical Functions

Take the mathematical function:

$$f(x) = x^2$$

f is a function that maps integers to integers:



We apply the function f to numbers in its domain to obtain a number in its range, e.g.:

$$f(-2) = 4$$

Function Composition

Given the mathematical functions:

$$f(x) = x^2 \;,\;\; g(x) = x+1$$

$f \bullet g$ is the composition of f and g:

$$f \bullet g\ (x) = f(g(x))$$

$$f \bullet g\ (x) = f(g(x)) = f(x+1) = (x+1)^2 = x^2 + 2x + 1$$
$$g \bullet f\ (x) = g(f(x)) = g(x^2) = x^2 + 1$$

Function composition is therefore not commutative. Function composition can be regarded as a (*higher-order*) function with the following type:

$$\bullet : (Z \to Z) \times (Z \to Z) \to (Z \to Z)$$

Lambda Calculus (Church and Kleene 1930's)

A unified language to manipulate and reason about functions.

Given

$f(x) = x^2$

$\lambda x.\ x^2$

represents the same ***f*** function, except it is *anonymous*.

To represent the function evaluation $f(2) = 4$,
we use the following λ-calculus syntax:

$(\lambda x.\ x^2\ 2) \Rightarrow 2^2 \Rightarrow 4$

Lambda Calculus Syntax and Semantics

The syntax of a λ-calculus expression is as follows:

e	**::=**	**v**	variable
	|	***λ*v.e**	functional abstraction
	|	***(*e e*)***	function application

The semantics of a λ-calculus expression is called beta-reduction:

$$(\lambda x.E\ M) \Rightarrow E\{M/x\}$$

where we alpha-rename the lambda abstraction **E** if necessary to avoid capturing free variables in **M**.

Currying

The lambda calculus can only represent functions of *one* variable. It turns out that one-variable functions are sufficient to represent multiple-variable functions, using a strategy called *currying*.

E.g., given the mathematical function: $h(x,y) = x+y$

of type $h: Z \times Z \rightarrow Z$

We can represent ***h*** as ***h'*** of type: $h': Z \rightarrow Z \rightarrow Z$

Such that

$$h(x,y) = h'(x)(y) = x+y$$

For example,

$$h'(2) = g, \text{ where } g(y) = 2+y$$

We say that ***h'*** is the *curried* version of ***h***.

Function Composition in Lambda Calculus

S:	$\lambda x.(s\ x)$	(Square)
I:	$\lambda x.(i\ x)$	(Increment)
C:	$\lambda f.\lambda g.\lambda x.(f\ (g\ x))$	(Function Composition)

Recall semantics rule:

$$(\lambda x.E\ M) \Rightarrow E\{M/x\}$$

((C S) I)

$$((\lambda f.\lambda g.\lambda x.(f\ (g\ x))\ \lambda x.(s\ x))\ \lambda x.(i\ x))$$
$$\Rightarrow (\lambda g.\lambda x.(\lambda x.(s\ x)\ (g\ x))\ \lambda x.(i\ x))$$
$$\Rightarrow \lambda x.(\lambda x.(s\ x)\ (\lambda x.(i\ x)\ x))$$
$$\Rightarrow \lambda x.(\lambda x.(s\ x)\ (i\ x))$$
$$\Rightarrow \lambda x.(s\ (i\ x))$$

Free and Bound Variables

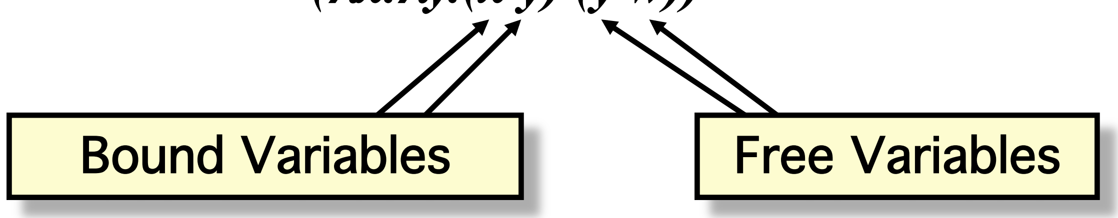
The lambda functional abstraction is the only syntactic construct that *binds* variables. That is, in an expression of the form:

$$\lambda \mathbf{v}.\mathbf{e}$$

we say that free occurrences of variable **v** in expression **e** are *bound*. All other variable occurrences are said to be *free*.

E.g.,

$$(\lambda x.\lambda y.(x\ y)\ (y\ w))$$



α-renaming

Alpha renaming is used to prevent capturing free occurrences of variables when reducing a lambda calculus expression, e.g.,

$$\underline{(\lambda x.\lambda y.(x\ y)\ (y\ w))}$$
$$\Rightarrow \lambda y.((y\ w)\ y)$$

This reduction **erroneously** captures the free occurrence of ***y***.

A correct reduction first renames ***y*** to ***z***, (or any other *fresh* variable) e.g.,

$$(\lambda x.\lambda y.(x\ y)\ (y\ w))$$
$$\Rightarrow \underline{(\lambda x.\lambda z.(x\ z)\ (y\ w))}$$
$$\Rightarrow \lambda z.((y\ w)\ z)$$

where ***y*** remains *free*.

Order of Evaluation in the Lambda Calculus

Does the order of evaluation change the final result?
Consider:

$$\lambda x.(\lambda x.(s\ x)\ (\lambda x.(i\ x)\ x))$$

Recall semantics rule:

$$(\lambda x.E\ M) \Rightarrow E\{M/x\}$$

There are two possible evaluation orders:

$$\lambda x.(\lambda x.(s\ x)\ \underline{(\lambda x.(i\ x)\ x)})$$
$$\Rightarrow \lambda x.\underline{(\lambda x.(s\ x)\ (i\ x))}$$
$$\Rightarrow \lambda x.(s\ (i\ x))$$

Applicative Order

and:

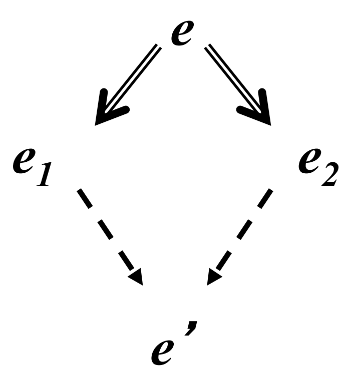
$$\lambda x.\underline{(\lambda x.(s\ x)\ (\lambda x.(i\ x)\ x))}$$
$$\Rightarrow \lambda x.(s\ \underline{(\lambda x.(i\ x)\ x)})$$
$$\Rightarrow \lambda x.(s\ (i\ x))$$

Normal Order

Is the final result always the same?

Church-Rosser Theorem

If a lambda calculus expression can be evaluated in two different ways and both ways terminate, both ways will yield the same result.



Also called the *diamond* or *confluence* property.

Furthermore, if there is a way for an expression evaluation to terminate, using normal order will cause termination.

Order of Evaluation and Termination

Consider:

$$(\lambda x.y\ (\lambda x.(x\ x)\ \lambda x.(x\ x)))$$

Recall semantics rule:

$$(\lambda x.E\ M) \Rightarrow E\{M/x\}$$

There are two possible evaluation orders:

$$(\lambda x.y\ \underline{(\lambda x.(x\ x)\ \lambda x.(x\ x))})$$
$$\Rightarrow (\lambda x.y\ (\lambda x.(x\ x)\ \lambda x.(x\ x)))$$

Applicative Order

and:

$$\underline{(\lambda x.y\ (\lambda x.(x\ x)\ \lambda x.(x\ x)))}$$
$$\Rightarrow y$$

Normal Order

In this example, normal order terminates whereas applicative order does not.

Combinators

A lambda calculus expression with *no free variables* is called a *combinator*. For example:

I:	$\lambda x.x$	(Identity)
App:	$\lambda f.\lambda x.(f\ x)$	(Application)
C:	$\lambda f.\lambda g.\lambda x.(f\ (g\ x))$	(Composition)
L:	$(\lambda x.(x\ x)\ \lambda x.(x\ x))$	(Loop)
Cur:	$\lambda f.\lambda x.\lambda y.((f\ x)\ y)$	(Currying)
Seq:	$\lambda x.\lambda y.(\lambda z.y\ x)$	(Sequencing--normal order)
ASeq:	$\lambda x.\lambda y.(y\ x)$	(Sequencing--applicative order)

where ***y*** denotes a *thunk, i.e.*, a lambda abstraction wrapping the second expression to evaluate.

The meaning of a combinator is always the same independently of its context.

Combinators in Functional Programming Languages

Most functional programming languages have a syntactic form for lambda abstractions. For example the identity combinator:

$$\lambda x.x$$

can be written in Oz as follows:

```
fun {$ X} X end
```

in Haskell as follows:

```
\x -> x
```

and in Scheme as follows:

```
(lambda(x) x)
```

Currying Combinator in Oz

The currying combinator can be written in Oz as follows:

```
fun {$ F}
        fun {$ X}
                fun {$ Y}
                        {F X Y}
                end
        end
end
```

It takes a function of two arguments, F, and returns its curried version, e.g.,

{{{Curry Plus} 2} 3} ⇒ 5

Exercises

1. PDCS Exercise 2.11.1 (page 31).
2. PDCS Exercise 2.11.2 (page 31).
3. PDCS Exercise 2.11.5 (page 31).
4. PDCS Exercise 2.11.6 (page 31).
5. Define Compose in Haskell. Demonstrate the use of curried Compose using an example.