CS2135: What is a Programming Language?

Kathi Fisler, WPI

September 19, 2003

1 The Design and Programming Perspectives

To start to understand what comprises a programming language, let's consider sample programs in four different languages (see Figure 1). What differences do you observe?

1. Syntax: each uses rather different notations for writing down programs. Quite true, but syntax is only so interesting; let's look for deeper differences.
2. What kinds of data they inherently support. Fortran supported various kinds of numbers, C includes arrays, while Scheme included lists (define-struct came in later), Java has classes.
3. Program organization techniques: The original Fortran didn't even have functions or basic blocks of code. Modern languages all provide functions, many now provide classes and objects, as well as larger organizational constructs like packages and modules.
4. What kinds of control constructs the programmer can use to express computations. Fortran provided only branches (conditionals) and goto-statements. Later languages provided loops, function calls, exceptions, and other ways for programmers to control how a program executes.
5. What the language will do for the programmer. Languages like C require programmers to declare variables and free memory. Languages like Scheme and Java use garbage collection (programs that reclaim memory when a program is no longer using it). The different degrees of typechecking in modern languages (and there are lots of interesting differences) also fall into this category.

Even though these samples are from large, mainstream languages, the same differences define the boundaries of small, domain specific languages (such as those you implement to create a particular software system). Thus, our quick contrast of these languages suggests the questions you need to ask when you're about to define a new language:[1]

1. What kind of data is this language designed to process?
2. What operations can someone perform on the data?
3. What control operators do I need to sequence operations?
4. What work is the language trying to save the programmer from doing?
5. What decisions should be postponed until run-time?

[1]Side note: Languages are fascinating artifacts, because they are carefully engineered to tradeoff a range of concerns like the ones in this list (while retaining efficiency and a host of other important features). If you're interested in this material, you should take a more substantial and upper-level course than 2135. For now, your best bet at WPI is the graduate course CS536 [Programming Language Design], which undergraduates can count as a 4000-level class.

FORTRAN (mid 1950's;1964)[a]

```
10 IF (X.GT.0.000001) GO TO 20
11 X = -X
   IF (X.LT.0.000001) GO TO 50
20 IF (X*Y.LT.0.000001) GO TO 30
   X = X*Y-Y
30 X = X+Y
   ...
50 CONTINUE
   X = A
   Y = B-A
   GO TO 11
   ...
```

[a]sample from slides for John Mitchell's PL course at Stanford

Scheme (1975)

(**define-struct** *boa* (*name length eats*))

(**define** (*all-boa-foods ani-lst*)
  (*map boa-eats* (*filter boa? ani-lst*)))

C (1978)[a]

```
#include <stdio.h>
#include <malloc.h>

/* Linked list structure */
struct c_linked_list
{
   char data[20]; /* String */
   struct c_linked_list *next;
};

/* Type def for linked list */
typedef struct c_linked_list link;

void deallocate_list(link *element) {
   link *current;

   /* Cycle through list */
   while (element != NULL)
   {
      /* Copy pointer */
      current = element;
      /* Get next pointer */
      element = (link *) element->next;
      /* Free old pointer */
      free (current);
   }
}
```

[a]Code adapted from http://www.stat.cmu.edu/ hseltman/c/Reilly.html

Java (mid 1990s)[a]

```
class Body {
  public long idNum;
  public String name=''<unnamed>'';
  public Body orbits = null ;
  private static long nextID = 0;

  Body() {
    idnum = nextID++;
  }
}

Body sun = new Body();
sun.idNum = Body.nextID++;
sun.name = ''Sol'';
sun.orbits = null ;
```

[a]Code from *The Java Programming Language*. Ken Arnold and James Gosling. Addison Wesley 1996

Figure 1: Sample code fragments from four well-known programming languages.

2 The Implementation Perspective

The previous section gave a high-level view of what a language is: a collection of data, operations, control operators, and other features that let us write programs. That doesn't tell us how to implement a language though. What does it mean to implement a programming language?

An implementation of a language allows someone to write and run programs in that language. If you want to implement a language, you must provide:

1. A syntax for writing programs in the language, and

2. A program or process that executes programs written in that syntax.

The second requirement means that we need to be able to write programs (called *interpreters*) that take programs as input and produce the results of running those programs as output. In order to "take a program as input", we will need data definitions for programs (if you don't see why, try writing the contract for an interpreter). These data definitions give us one (albeit clumsy) syntax for writing down programs. A real language implementation puts a cleaner syntax before the data definition to make programs easier to write. In other words, a language implementation based on interpreters has three stages:

$$\begin{tikzcd}
\boxed{\text{Source Code}} \arrow[r, "\text{Parser}"] & \boxed{\begin{array}{c}\text{Data Structure}\\\text{for Programs}\end{array}} \arrow[r, "\text{Interpreter}"] & \begin{array}{l}\text{Result of}\\\text{Running Program}\end{array}
\end{tikzcd}$$

We will touch on both stages. We'll start with the later stage (data definitions for languages and writing interpreters), then we'll discuss one technique (macros) to quickly create a cleaner syntax on top of those data definitions.

2.1 But What Is a Programming Language, Really?

More abstractly, languages are what we call *abstractions*: ways of seeing or organizing the world according to certain patterns, so that a task becomes easier to carry out. More concretely, think about a while loop in C/C++. When you write a while loop, you expect the machine to carry out certain tasks for you automatically: testing the termination condition, running the loop code if the test passes, exiting the loop if it doesn't, etc. When you write down a while loop, you don't write down all of these steps. The while loop is a common and useful pattern in programming that the designers of C gave you in the form of a language construct. The while loop is an abstraction: a reusable pattern where the language executes part of the pattern automatically, and you supply the parts that are different. You _could_ write down all of those steps manually, but then your program would be longer, harder to read, and more painful to write and debug.

Similarly, map and filter from Scheme are abstractions: patterns of programs that share a common structure: you write your code in terms of map and filter, and those constructs execute the common pattern. The main difference between map/filter and while loops in this context is that you can define map/filter and similar functions in your programs, while C builds while into the language, and doesn't give you good support for defining your own loops.

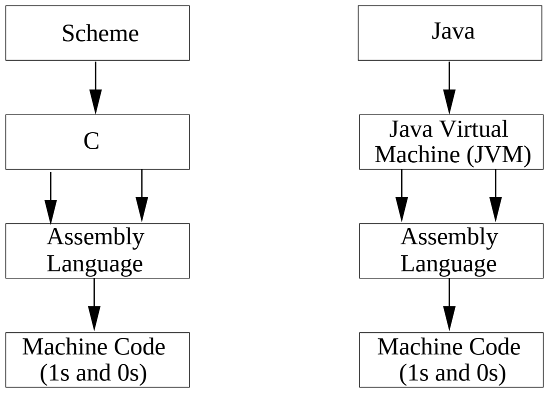
> *Languages are collections of abstractions: collections of common patterns that programmers can combine into working programs.*

When you implement a language, you implement those patterns (abstractions) that make up the language. You provide programmers with the tools they need to express the computations they want to perform. One task of a language designer is therefore to understand what programmers want to express (according to questions like those presented in Section 1), and then to implement the abstractions (using interpreters or compilers) corresponding to those patterns so someone can program in the new set of constructs.

2.2 Interpreters Versus Compilers

The interpreter-based strategy outlined previously is one way to implement a language. Another strategy is to translate programs in your new language into programs in an existing (target) language, then use the tools for executing programs in the target language to run your program. This approach is called *compilation*. If we wanted to compile

Scheme, for example, we would need to write a program (a compiler) that translates Scheme into some known language (such as C). Another compiler would translate C into assembly, and so forth. The code for implementing a C-style while-loop is in the compiler that takes C-code down to Assembly language. At the lowest level in this hierarchy, an interpreter runs the final program to produce an answer. The following hierarchies show common strategies for implementing Scheme and Java systems:



Each arrow represents a different program that translates from one language into the next. If there are multiple arrows from one box to the next, it means that there are multiple (architecture-specific) versions of the lower language, so multiple translators get used.[2]

3 A Concrete Example: Implementing a Slideshow Package

Let's implement a slide-presentation system, similar in spirit to Powerpoint. The web site has a sample Powerpoint presentation for those of you who aren't familiar with it. What would a naïve implementation of the slideshow look like (using a simple, text-based interface)? Look at Figure 2: what's wrong with this implementation (aside from the unappealing text-based interface)?

- The programmer manually inserts the begin/end of slide lines each time.
- The programmer has to explicitly add the *await-click* commands, even though those are standard between slides.
- The programmer has to call *print-string* all the time, even though all the slide data are strings.
- There's little here that the programmer could reuse when writing another slideshow presentation.

This example is crying out for a language for writing slideshows, and for an interpreter to run those slideshows. To start defining the language, we need to answer the five questions from Section 1:

1. Slides are the data. Slides have titles and content. The content may be a bunch of text, or lists that are either bulleted or numbered. We may also have overlays, which are slides that sit on top of other slides, so that we can present content in stages.
2. The main operation on slides seems to be displaying them on the screen.
3. We sequence slides by putting them in some (linear) order.
4. The language should save the programmer from waiting for clicks to move between slides, from printing titles manually, and from manually specifying/tracking item numbers in itemized lists.
5. It's not clear at the moment what decisions should be postponed until run-time, so we'll come back to that one later.

[2]One of the technical advances that Java contributes is allowing richer sets of language constructs (a richer abstraction) before the code specializes per architecture. This picture enables the "write once, run anywhere" goal behind Java's design.

```
;; Stage -1 : Powerpoint without a language

;; print-string : string → void
;; prints string and a newline to the screen
(define (print-string str)
  (printf "~a~n" str))

;; await-click : → void
;; mimics a mouse click by waiting for the user to type a character
(define (await-click) (read))

(begin
  (print-string "——————————————")
  (print-string "Hand Evals in DrScheme")
  (print-string "Hand evaluation helps you learn how Scheme reduces programs to values")
  (print-string "——————————————")
  (await-click)
  (print-string "——————————————")
  (print-string "Example 1")
  (print-string "(+ (* 2 3) 6)")
  (await-click)
  (print-string "(+ 6 6)")
  (await-click)
  (print-string "12")
  (print-string "——————————————")
  (await-click)
  (print-string "——————————————")
  (print-string "Summary: How to Hand Eval")
  (print-string "Find the innermost expression")
  (await-click)
  (print-string "Evaluate one step")
  (await-click)
  (print-string "Repeat until have a value")
  (print-string "——————————————")
  )
```

Figure 2: A naïve implementation of a slideshow.

Let's start by implementing a simple prototype of our slideshow system. A *prototype* is a working version of a scaled-down version of the system. The prototype lets us figure out how to implement the key concepts, and we'll build on those concepts later as we refine the language. For our first prototype, we'll implement basic sequences of slides (no overlays), and use a simple text display rather than a fancier graphics display.

3.1 The First Prototype

Note: *Refer to the code posted with the notes on the class website for a more complete picture of the finished code.*

Recall that implementing a language requires us to write programs that take programs in our language as input and execute those programs. In other words, we're writing a program

```
;; slideshow-implementation : slideshow-program → ???
;; executes a program in our (new) slideshow language
```

What is a *slideshow-program*? From this contract, it looks like something we need a **define-struct** for (since it's certainly not built-in to Scheme. That's exactly right – we need a data definition for writing down programs, *then* (and only then!) we'll need a program for implementing the slideshow programs.

Let's start by considering how we model slideshow programs as data:

3.2 Modeling the Data

We've already observed that our data consists of slides, so we need a data definition for slides:

```
;; a slide is a (make-slide string slide-body)
(define-struct slide (title body))

;; a slide-body is either
;; - a string (paragraph), or
;; - a (make-pointlist list[string] boolean)
(define-struct pointlist (points numbered?))

;; Examples
   (make-slide
      "Hand Evals in DrScheme"
      "Hand evaluation helps you learn how Scheme reduces programs to values")

   (make-slide
      "Example 1"
      (make-pointlist (list "(+ (* 2 3) 6)" "(+ 6 6)" "12") false))
```

3.3 Modeling the Operations

At this stage, we are not *implementing* the operations, we are instead trying to *represent programs as data*, so we can later write an implementation. We must develop a data definition for the set of operations in our language. Right now, we only have a display operation (though we might have more later). Let's write a data definition for operations (here called *commands*):

```
;; A cmd is
;; - (make-display slide)
(define-struct display (slide))

;; Example
   (make-display
      (make-slide
         "Hand Evals in DrScheme"
         "Hand evaluation helps you learn how Scheme reduces programs to values"))
```

Remember, this isn't trying to *implement* the display operation: it's merely giving us a way to write down that someone wants to perform a display operation on a slide when we run the program. There are two different times here: writing the program, and running the program (you're quite familiar with this distinction in practice, but it does often seem odd to handle both in code when you start implementing languages).

3.4 Modeling Programs

In the slideshow example, a "program" is just a talk, which is itself a sequence of operations on slides. We need to model talks as data. The word "sequence" suggests lists. So, we'll view talks/programs as containing lists of operations:

```
;; A talk is a (make-talk list[cmd])
(define-struct talk (cmds))

;; A program is a talk
```

Now that we've modeled programs, operations, control, and data through data definitions, we are ready to write our first program (even though we can't run it yet):

```
(define talk1
  (let ([intro-slide
         (make-slide
          "Hand Evals in DrScheme"
          "Hand evaluation helps you learn how Scheme reduces programs to values")]
        [arith-eg-slide
         (make-slide
          "Example 1"
          (make-pointlist (list "(+ (* 2 3) 6)" "(+ 6 6)" "12") false))]
        [func-eg-slide
         (make-slide
          "Example 2"
          (make-pointlist (list "(define (foo x) (+ x 3))" "(* (foo 5) 4)" "(* (+ 5 3) 4)" "(* 8 4)"
                                "32") false))]
        [summary-slide
         (make-slide
          "Summary: How to Hand Eval"
          (make-pointlist (list "Find the innermost expression"
                                "Evaluate one step"
                                "Repeat until have a value")
                          true))])
    (make-talk
     (list (make-display intro-slide)
           (make-display arith-eg-slide)
           (make-display func-eg-slide)
           (make-display summary-slide)))))
```

This code uses **let** instead of *local*. **let** is a bit more compact because you don't need to write **define** on each case. The **let** code is as if you wrote (**define** *intro-slide* (*make-slide* ...)). We use local definitions here to keep the slides internal to the talk (otherwise, we'd prevent ourselves from defining another *summary-slide* for a different talk.

Next class, we'll implement the prototype.