Implement it yourself
A good way to learn a programming language.

Herman Venter
University of Port Elizabeth
http://www.cs.upe.ac.za/staff/csabhv

1 Introduction

Whether we like it or not, Computer Scientists have to keep learning new programming languages (if only so that we can safely ignore them). A thorough study of programming languages therefore constitutes a vital part of both the undergraduate and graduate education of computer scientists.

If the above statement seems less than self-evident to you, put yourself in the shoes of a graduate of some years ago, now in a leadership position in industry. Most likely, this graduate was fed with a steady diet of Pascal while at university, only to find that his new employer actually wanted COBOL or C or even RPG skills. Meanwhile the trade magazines have started to sing the praises of C++, SmallTalk, Ada 9x and so on, promising great productivity gains to those who adopt them. More insidiously, the graduate is being assailed by numerous partially disguised programming languages such as PL/SQL, SQL Windows, Visual Age and Delphi, all of which are proclaimed to be the greatest thing since sliced bread.

The only way that we can equip our graduates to cope with this continual onslaught, is to provide them with an education that provides them with a thorough understanding of the essential concepts of programming languages and the fundamental design issues and trade-offs. And because most of our graduates leave us after only three years, we have to fit a course into the second or third year.

My experience with teaching the undergraduate course is that there is not enough time to provide students with as thorough an understanding of programming languages as is desirable. It thus seems obvious to me that a second course in programming languages is an essential part of the education of a Computer Scientist. In practice, this means an honours course.

2 Approach

How one should approach an honours course, is less obvious. One can broaden the already broad survey done in the undergraduate course and/or deepen the study of particular topics, but this inevitably involves considerable duplication of the undergraduate course.

Another problem is to find a suitable textbook: The typical programming language textbook (for example[1, 2, 6, 9, 12, 13]) amounts to a broad and shallow survey of languages and paradigms and does not lend itself to being split up into an undergraduate and a graduate course. One can choose a thin book such as the one by Bal and Grune[1] for the undergraduate course and a thick(er) one such as the one by Sethi[9] for the graduate course, but this makes it very difficult to avoid duplication.

An alternative to the concepts+paradigms+survey type of course is to model the honours course after the typical American graduate PL course, which deals with the formal semantics of programming languages. If you choose to do this, you can use the brilliant textbook by Bertrand Meyer[7] or one of the many others (for example [3, 4, 10]). However, this approach may swamp the students with formalism to such an extent that the goal of providing them with a thorough understanding of real programming languages is not achieved. I am equally sceptical that a comprehensive study of the Functional and/or Logical Programming paradigms will achieve the above goal.

I am therefore about to embark on a course which combines the traditional, but perhaps less vital, compiler course with the programming language course, resulting in a course where the students study a number of programming languages in depth by actually implementing them. This not only solves the duplication problem, but helps to find a space for the course in the very packed honours curriculum.

To make this approach feasible, the traditional study of scanners, parsers and code-generation techniques must be sacrificed in favour of an emphasis on the semantics of the programming languages. In other words, the students must be given a system that generates language implementations from the formal descriptions of their semantics.

Students can then study a programming language, formalize its semantics and generate an implementation which they have to test. In the process they should gain a thorough understanding of the power and limitations of the language and the impact of its design trade-offs.

3 Semantics without tears

The only formal notation with which most of our students are reasonably comfortable with, even as graduates, is a programming language. It thus makes sense to try to use a familiar programming language as the formal notation in which to describe the semantics of the programming language being studied.

Although somewhat unpopular with theorists, the use of programming languages to describe the semantics of other programming languages is hardly a novel idea: you do it every time you write an interpreter for language X in language Y.

Of course, writing an interpreter in the usual way is still far from trivial and does not spare the students from non-essentials like scanning and parsing. One way to address this is to trivialize scanning and parsing by adopting a LISP-like syntax for all of the languages studied, and to provide students with ready-made interpreters for subsets of these languages. This approach has been used with some success by Samuel Kamin of UIUC in his book "Programming Languages - An Interpreter-Based Approach"[5].

My own approach is to make use of a compiler generating system which I have developed over the last few years. In this system, the scanner, parser and necessary scaffolding is generated from a single source file containing a BNF-like syntax definition. The rest of the specification is supplied in the form of Object-Oriented code written in a very high level language, constituting an interpreter for the language.

The structure of the system is such that students need not understand the scaffolding. Furthermore, since the semantics are written as Object-Oriented code, the similarities between different programming languages can be highlighted and exploited via code re-use between implementations.

4 Semantics for a little language

To illustrate my system, I shall present a specification based on the little language dealt with in Kamin's first chapter. The syntax is follows:

```
goal = {instruction | function_definition}
function_definition =
  'function' identifier '(' identifier {',' identifier} ')'
     {instruction}
  'end'
```

```
instruction =
  ‘if’ expression ‘then’ {instruction} ‘else’ {instruction} ‘end’ |
  ‘while’ expression ‘do’ {instruction} ‘end’ |
  ‘print’ expression |
  ‘return’ expression |
  identifier ‘:=’ expression
expression = simple_expression [operator simple_expression]
simple_expression =
  integer | identifier | ‘(’ expression ‘)’ | identifier ‘(’ {expression} ‘)’
operator = ‘+’ | ‘-’ | ‘/’ | ‘*’ | ‘=’ | ‘<’ | ‘>’
identifier = (Up_let | Low_let) {Up_let | Low_let | Digit | ‘_’ }
integer = Digit{Digit}

Tokens = identifier | integer
Ignorable = Eol | ‘ ’
```

To add semantics to this language, one considers the abstract syntax tree (AST) as an expression to be evaluated, rather than as a structure to be translated into another language.

For example, the source program:

```
a := 1
a := a + 2
print 3 + a
```

is seen as syntactic sugar for the expression:

```
[‘:=’(new_identifier(‘a’), new_integer(‘1’)),
 ‘:=’(new_identifier(‘a’), ‘+’(new_identifier(‘a’), new_integer(‘2’)),
 ‘print’(‘+’(new_integer(‘3’), new_identifier(‘a’)))]
```

which is, of course, simply the AST.

To understand what will happen when the expression is evaluated, one must refer to the definitions of classes Integer and Identifier (and any classes referred to by their methods). These classes, which can be written in an ordinary Object-Oriented programming language, thus constitute the semantics of the language being implemented.

In practice, students are not concerned with the mechanics of the process which turns the BNF+Class definitions into a compiler or interpreter. (I.e. translating the AST into executable code, or interpreting it directly.)

Some of the code for classes Identifier and Integer is presented in figures 1 and 2, respectively. It is written in Slim[11], a language developed by the author and closely related to SETL[8].

5 Messy details

The syntax and class definitions presented above, omit some messy details in the interest of clarity. The actual system used does not allow function names such as ‘+’ and ‘if’ and does not make an implicit connection between syntax rules and method invocations.

It is also necessary to indicate that the parameters to the ‘if’ method should not be eagerly evaluated and to distinguish between the ‘left-hand’ and ‘right-hand’ values of identifiers. Finally, it is necessary to indicate how repeated syntactic constructs are to be handled.

Figure 1: Partial class definition for Identifiers

```
import Scope, Variable, Debug
export Identifier, new_identifier

Identifier : set of Entity := {}       !declares a class
chars : map from Identifier to String  !declares an attribute

func new_identifier(chs : String) -> Identifier
  result := new(Identifier)    !allocates a new Identifier instance
  chars(result) := chs         !assigns identifier name to corresponding attribute

func `:='(self, other) -> Identifier
  var := scope.lookup(chars(self))
  if var = Undefined
    var := new_variable()
    scope.enter(chars(self), var)
  end
  return `:='(var, other)  !call the corresponding method of the Variable class

func value(self) -> Entity
  repeat
    var := scope.lookup(chars(self))
    return value(var) when var # Undefined
    debug.Message(`Identifier \`'+chars(self)+`\' used before definition')
    debug.Interact
  loop
```

Figure 2: Partial class definition for Integers

```
export Integer, new_integer

Integer : set of Entity := {}
value : map from Integer to integer !the latter is the built-in type

func new_integer(chs : String) -> Integer
  pre mknum(chs) in integer
  result := new(Integer)
  value(result) := mknum(chs)

func `+'(self, other : Integer) -> Integer
  result := new(Integer)
  value(result) := value(self) + value(other)

func `print'(self) -> Integer
  put value(self)
  return self

func `if'(self, thenpart, elsepart) -> Entity
  if value(self) /= 0
    return value(thenpart)
  else
    return value(elsepart)
  end
```

Nevertheless, even with all the messy details included, the specification should still be well within the grasp of the average student. This remains to be seen, however, since I have yet to present the course.

It is hard to imagine, however, that this approach will present the students with greater difficulties than they would encounter in the study of Kamin-style interpreters or the various other formalisms used to specify programming language semantics.

The full specification is available on-line as `http://www.cs.upe.ac.za/pldi/example`.

References

[1] H E Bal and D Grune, *Programming Language Essentials*, Addison-Wesley, 1994.

[2] H L Dershem, M J Jipping, *Programming Languages: Structures and Models*, Wadsworth, 1990.

[3] Gunter, *Semantics of Programming Languages*, MIT Press.

[4] M Hennessy, *The Semantics of Programming Languages – An Elementary Introduction using Structural Operational Semantics*, Wiley, 1990.

[5] S N Kamin, *Programming Languages – An Interpreter-Based approach*, Addison-Wesley, 1990.

[6] K C Louden, *Programming Languages – Principles and Practice*, PWS, Boston, 1993.

[7] B Meyer, *Introduction to the Theory of Programming Languages*, Prentice Hall, 1989.

[8] J T Schwartz, et al, *Programming with sets: an introduction to SETL*, Springer, 1986.

[9] R Sethi, *Programming Languages – Concepts and Constructs*, Addison-Wesley, 1989.

[10] K Slonneger and B. Kurtz, *Formal Syntax and Semantics of Programming Languages*, Addison-Wesley, 1995.

[11] B H Venter, *Slim*, `http://www.cs.upe.ac.za/slim`.

[12] D A Watt, *Programming Language Concepts and Paradigms*, Prentice Hall, 1990.

[13] L B Wilson and R G Clark, *Comparative Programming Languages*, Addison-Wesley, 1988.