# Teaching "Principles of Programming Languages" through Design and Implementation of a Simple Programming Language

Lydia Sinapova
Department of Computer Science
Simpson College
sinapova@simpson.edu

## Abstract

Teaching "Principles of Programming Languages" at an undergraduate level in a small liberal arts college is very challenging. Most of the concepts such as grammars, syntax, semantics, are very difficult for students that know only one or two programming languages. This paper discusses a "learning-by-doing" approach used in the author's "Programming Languages" course since 2002. We present a term project for design and implementation of an interpreter for a simple programming language.

# Introduction

Teaching "Principles of Programming Languages" at an undergraduate level in a small liberal arts college is very challenging. Most of the concepts such as grammars, syntax, semantics, are very difficult for students that know only one or two programming languages. Homework assignments should help students acquire better understanding of the underlying concepts. However, if the assignments consist of unrelated problems, they fail to provide the cognitive framework which students need to organize their individual mental models about what a programming language does and how it is designed and implemented. This paper presents a term project for design and implementation of a simple programming language, used in the author's "Programming Languages" course since 2002.

## Background

The programming language class uses the textbook by Pratt and Zelkowitz, 2000 [2]. The textbook follows the concept based-approach in describing the principle of design and implementation of programming languages. It also describes several programming languages to give illustration of the basic concepts. The syntax and semantic issues are given fairly good consideration, but they are not in the focus of the textbook. Only two chapters are designated to these issues. The core chapters discuss the major programming concepts – data types, encapsulation, sequence control, subprogram control, and storage management. The last two chapters discuss distributed and network programming.

The course presents programming languages from four perspectives:

- Language constructs necessary for specifying data types and control structures
- Formalisms for describing language syntax and semantics
- Language translation and execution
- Models of computation and programming paradigms

The course discusses the four major programming paradigms, however students do not do actual programming in a paradigm different from what they already know. Instead, the course emphasizes three major issues that are present in all programming paradigms:

- What a language should consist of in order to serve as a tool for specifying a computation
- How to specify language syntax and semantics
- What actions should be performed in order to translate source code into an executable module

The challenge that most of the educators face is what kind of exercises should be assigned to illustrate the in-class discussions. Two basic approaches have been used aimed at providing "hands-on" experience for better understanding the key language concepts.

- short programming assignments in various languages to illustrate a particular concept and a particular paradigm
- implementation of certain interesting components of well known programming languages.[1]

The first approach is easier to implement, however it seems that it does not contribute too much to students' understanding.

The second approach has recently gained more popularity. Full implementation of a language by a compiler is generally difficult and the majority of undergraduate courses in programming languages avoid it. The usual practice in courses that use this approach is that at the beginning students learn a functional programming language, which is used later as a tool for writing interpreters for a subset of some existing programming language. The benefits of this approach are that the students understand the semantics issues. The disadvantages are that the students have to spend time learning a new language. While there is nothing wrong in learning a new language, in general the purpose of a programming language class is not to teach new programming languages. In particular considering the specifics of small liberal arts colleges where usually only one programming languages course is offered, such approach will distract the students from the key issues in programming language design and implementation, focusing on language semantics.

An alternative to using a functional programming language has been proposed in [1]. The approach consists in using a language known by the students to develop interpreters for data type specifications. The disadvantage of this approach is that the students are exposed to a restricted set of language development issues. They do not get practical experience related to all stages of language design and development.

Here we propose another approach that is aimed at giving the students the opportunity to take part in each step of language design and interpreter-based implementation. Students design a simple programming language, define the language using BNF grammar and develop an interpreter for their language. When their project is completed they can write and run programs in the language that they have designed.

Below we describe the sequence of homework assignments that constitute the project.

# Interpreter of a simple programming language

## The Language

The prototype of the language developed by the students was taken from the textbook [2], p. 42-43:

| | |
|---|---|
| a = b | Assign to the variable **a** the value of the variable **b** |
| a = a+1 | Add 1 to **a** |
| a = a-1 | Subtract 1 from **a** |
| if a = 0 then goto L | If a = 0, transfer control to statement labeled with L |
| if a > 0 then goto L | If a > 0, transfer control to statement labeled with L |
| goto L | Transfer control to statement labeled with L |
| halt | Stop execution |

For example, the following program computes the sum of two positive integers a + b can be given by:

```
L:    a = a+1
      b = b-1
      if b > 0 then goto L
      halt
```

Note that this language falls under the imperative programming paradigm. It is not typed and does not enforce modular programming.

The first assignment acquaints the students with the language. They have to write simple programs in that language, for example to find the sum of two integers. They also have to suggest minimal extensions of the language without changing its simple structure. The proposed extensions concern input/output operators, comments, and more elaborate arithmetic and relational expressions.

## Language Design

The second assignment requires the students to describe verbally the components of their language. By the time this assignment is given, the class has discussed a number of syntactic elements and how the choice of syntactic elements determines the syntactic style of a language. Thus, the assignment naturally fits the scheduled topics. In particular, the assignment requires verbal description of the following syntactic elements:

- **Alphabet**. List the allowable symbols.
- **Constants**. Describe the type and range of allowed constants.
- **Identifiers.** List the type of identifiers and describe their syntax.
- **Operator symbols.** List the operator symbols in the language.
- **Keywords**. List the keywords in the language.
- **Noise words.** Decide whether the language would contain such words.
- **Comments**. Decide whether comments are allowed. If yes - describe their syntax.
- **Delimiters**. List the delimiters and their specific function (if any).
- **Expressions**. Describe the syntax of allowed expressions.
- **Statements.** Describe the type of statements in the language and their syntax.

At this stage students usually suggest:

- using integers, real numbers and strings
- having arithmetic expressions such as x +/- b, where x is a variable and b is a constant.
- extending the relational expressions to x >a, x = a, and x < a, where x is a variable and a is a constant
- initializing a variable with a constant
- removing the noise word "then"
- using space as a delimiter and removing ':' as a delimiter

Since data types have not been discussed yet, usually no one suggests adding keywords for the type of the variables.

## Formal Definition of the Language Syntax

The third assignment requires the formal definition of the language syntax using BNF notation. This assignment is given immediately after the BNF topic discussion.

Here is an excerpt of such definition:

<program> ::= <statement><statement>*
<statement> ::= <assignment> | <control> | <input/output>
<assignment> ::= <variable> = <arithmetic expression>
<arithmetic expression> ::= <term>
<arithmetic expression> ::= <arithmetic expression> - <term>
<arithmetic expression> ::= <arithmetic expression> + <term>
<term> ::= <primary>
<primary> ::= <variable> | <integer>
<variable> ::= <identifier>
<identifier> ::= <letter> {<letter> | <digit>}*
…..

At this point most students fail to include the labels in the BNF description of their language. The obvious reason is that they are not used to programs that have labels.
When asked to present ideas how labels can be included, most of them come up with the idea of introducing a new category – <unlabeled_statement> resulting in the new definition of a statement:

<statement> ::= <label><unlabeled_statement> | < unlabeled_statement >
< unlabeled_statement> ::= <assignment> | <control> | <input/output>

This helps them understand better the role of the syntactic categories, and to understand that the categories are a result of our conceptualization, and not something given in advance.

## Tokenizer

The tokenizer is the next assignment that students have to complete. Students are not allowed to use the Java built-in tokenizer. The input to the tokenizer is a text file containing the program statements. At this point students usually have to revise again the verbal description of the language and reconsider their choice of delimiters. The issue here is whether the language allows one statement per line, allows more than one statement, or allows one statement to span two lines. They also understand that the design choices are affecting the implementation, and hence usually they choose the simpler implementation – one statement per line.

The implementation of the tokenizer is relatively easy. Conceptually the tokenizer is a finite state automaton acting as a recognizer. The students can relate the tokenizer with what they are studying at this time of the course - regular grammars and finite state machines. They can easily understand what kinds of objects are described by regular grammars and how regular grammars are recognized by finite state automata.

The result of the tokenizer is a table containing all tokens as strings. The tokenizer reports one type of error = "unrecognized symbol". Any errors caught at this level are reported.

## Lexical Analysis

The purpose of this assignment is to design and implement the lexical analyzer: a program that will examine each token in the token table built by the tokenizer, and will collect information about the tokens necessary for the syntax and semantics analysis.
The tasks of the lexical analyzer are:

1. Build a table of token types
2. Build symbol tables:
    a. Store all variable names (without repetition) in an array
    b. Store all label names (without repetition) in an array. For each label, record in a matching array the number of the statement to which the label is attached.
3. Create an address table that provides references to the variables, label, keywords, and has the numeric values of the numbers used in the source program
4. Report errors:
    a. two or more labels with the same name preceding statements,
    b. dangling labels – used after 'goto' but not attached to a statement

At this phase, the students have to make several design and implementation choices.

First of all, they have to decide how to represent the syntactic categories of the tokens. Obviously, if they use the names from the BNF description, their table will use too much memory. The obvious solution is to use names of one letter only, e.g. V for <variable>, N

for <number>. By necessity they introduce a category K for 'keyword', which however is not used in the BNF notation. Thus, they can appreciate the importance of having adequate documentation, since only in the documentation they will write about why they have used 'K' in the table of token types to represent any keyword.

Second, some of the students realize that it does not seem right to store in one table names of variables that may be of different type. They also realize that so far they have not thought about how to declare the type of the variables. Students start asking questions in class and thus everybody becomes aware of the 'type' issue. This helps understand the usage of the descriptors, studied at the same time. Generally, the majority of students choose to have only one type – integer, which simplifies the implementation, but they also realize that the decision significantly reduces the potential of the language. Students have two weeks to accomplish this assignment, because it is relatively more difficult than the other assignments constituting the project.

## Syntactic Analysis

The tasks of the syntactic analysis are:

- Determine the syntax structure of each statement
- Report errors if any
- Record in an array the index of the syntax pattern for each statement (to be used later by the semantic analysis and the synthesis of the executable table)

The syntactic analyzer is based on the BNF description of the language. The assignment however does not require the development of a full parser. This decision is based on the following considerations:

a. Building a parser that assigns parse trees is a task that exceeds the scope of this class

b. The language is simple enough so that the parse trees of each type of statements can be flattened, i.e. students can use syntactic patterns derived from the BNF description. The syntactic patterns are derived on paper and then embodied in the syntax module as an array of type string.

Examples of such patterns are:

```
rules[0] = "V=V+N";       // a = a+1, a = a-1
rules[1] = "V=V-N" ;      // a = a-1
rules[2] = "V=V";         // a = b
rules[3] = "V=N";         // a = 5
rules[5] = "KV=NKL";      // if a = 0 goto L
……
```

For a statement to be syntactically correct it must match some pattern and it must meet certain conditions specific for each pattern. The second requirement accounts for the

context-sensitive features of the language that cannot be represented in the context-free syntax patterns. For example a statement with syntax pattern "KV=NKL" is valid only if the first keyword is “if” and the second keyword is “goto”.

With this assignment the students understand the difference between context-free and context-sensitive language description.

## Semantic Analysis

In this project, semantic analysis is based on the operational semantics of each statement. The operational semantics is defined by a set of rules specifying how the state of the memory changes while executing the statement. The semantics of the program is defined as a composition of the semantics of each statement. The program memory is simulated by an array that serves as a storage for the values of the program variables.

The semantic analysis in this project consists in appropriate encoding of the semantics of each statement, which will be decoded when running the program.

The result of the semantic analysis is a table that contains the codes of the operations specified in the statements and references to their operands (if any). The codes are designed by the students. Depending on the language design, statements have two or three operands at most. For example, if addition and subtraction are used only in statements of the type “a = a + N”, the number of the operands is two. If the language allows statements of the type “a = b + N”, the operands will be three. The operands are represented by their ‘addresses’ – indexes in the memory array if the operands are variables, or as literals if they are numbers.

Conceptually, the resulting table can be viewed as containing machine-like instructions with their operands in a two- or three-operand machine. This representation is very convenient for decoding at the final step of the interpretation – executing the program.

With this assignment, students gain practical understanding of the operational semantics.

## Running the program

Students have to develop an engine that decodes the semantic representation of the source program and performs the specified operations. The engine needs only the semantic table and the memory table. It plays the role of a processor with an instruction register, a program counter and an environment pointer. The program counter is an index in the semantic table. The instruction register contains the code of the instruction to be performed, stored in the table row whose index is in the program counter. The environment pointer is an index in the memory array. Decoding the instruction means applying the corresponding actions to the elements in the memory array whose indexes

are indicated in the semantic table. Each instruction determines the next value of the program counter. The execution stops when encountering the "halt" instruction. The execution is terminated abnormally when the program counter receives a value not less than the number of rows in the semantic table.

Implementing the engine is straightforward. It re-enforces the knowledge that students have obtained in the "Computer Organization" class.

## Project Documentation

Throughout each stage in this project students are required to provide documentation. The last assignment concerns the structuring of the documentation in two separate documents: user manual and programmer manual. The user manual describes the language informally and provides a brief tutorial how to write and execute programs in that language. The programmer's manual contains the formal description of the language, design decisions and implementation solutions, algorithms and source code of the program files.

## Conclusions

Through a sequence of related assignments the students are able to build their own interpreter for a language designed by them. The project provides practical experience in using BNF notation to write grammars. Students become familiar with the details of the lexical and syntactic analysis. They understand the meaning of the semantic analysis and synthesis by designing the internal code representation and writing a program that translates the syntactic structures into machine-like code, which is executed by a simulation program. Moreover, the project requires development of a simple interface to enable the user to create, edit and run programs in the implemented language, thus exposing the students to each step in creating a programming language.

Throughout the design and implementation process, students have to make various decisions and thus they learn about the programmers' responsibility. They can actually see the payoff between language expressiveness, ease of use, and the required efforts for the implementation.

The project requires the preparation of full documentation, including a user manual for programmers that will use the language, and a programmer's manual, intended for programmers that will maintain the language implementation. The documentation is done at each stage of the development process and helps students improve their technical writing skills. The project is very suitable for teamwork and thus it is a very useful experience for traditional students.

# References

[1] Fossum, Timothy and Susan Haller. *Reinforcing Programming Language Concepts Through Implementation in a Concept-Based Course*. The Journal of Computing Sciences in Colleges. December 2003. Volume 19, Number 2. pp. 82-90.

[2] Pratt, Terrence and Marvin V. Zelkowitz. *Programming Languages: Design and Implementation*. Prentice Hall. New Jersey 2000.