Scheme as an Expository Language for Liberal Arts Students

Aaron Konstam
Department of Computer Science
715 Stadium Drive
San Antonio, Texas 78212-7200
(210) 736-7484
FAX: (210) 736-7477
akonstam@shrew.cs.trinity.edu

John E. Howland
Department of Computer Science
715 Stadium Drive
San Antonio, Texas 78212-7200
(210) 736-7480
FAX: (210) 736-7477
jhowland@ariel.cs.trinity.edu

Abstract

The Scheme dialect of Lisp is being used as an expository notation in introductory courses for liberal arts students at Trinity University. Terminology from natural language identifying parts of speech, such as verb, noun, pronoun, adverb, etc., is used to present Scheme syntax and semantics to non programmers. Simple working models of various computer science topics are described. Experiences from two Trinity computer science courses are presented.

Keywords: Scheme, exposition.

Introduction

Throughout the period that we have been teaching computer science courses at Trinity we have tried to be consistent in choosing a programming language environment for our beginning courses. Not consistent in that we always have used the same programming language but rather choosing the currently available language that best met the same consistent set of goals.

First, we would like our language to be in some sense a *lingua franca* [Kon74,21]. That is, a language that allows us to teach the art of programming as well construct models of the computer science concepts we wish to get across and be pervasively usable in a wide variety of our courses. In addition, the language should support an expository style of expression in a concise manner and it should be possible to verbalize "sentences" in the language so that students and faculty may easily communicate. The language should have a simple enough syntax and a straight forward enough semantics to prevent the necessity of the instructor spending the large fraction of the course discussing the arcane intricacies of the language. We would rather spend the time in our courses focusing on the programming and computer science principles we would like the students to learn.

For students who are not intending to become computer scientists, the programming environment used in such a course should be applicable directly and easily to problems it their chosen disciplines, As with natural languages the proof of the programming language pudding is in the using. No one can become interested in a programming language that can not efficiently support him in doing something he or she decides they want to do.

A further characteristic of this programing environment that can be derived directly form the considerations discussed above is that the programming environment that is introduced to students in these first courses should be generally available on the variety of computer systems they will most likely have available to them during their tenure in the college environment, The language we chose that met these criteria is Scheme. Twenty years ago one of the authors attended an ACM conference where the suggestion that Lisp be taught as a first language was met with both hostility and disbelief. Now the Scheme dialect of Lisp is used in the first courses taught at some of the most prestigious universities in our country. What brought this change about?

A primary factor was the development of the Scheme programming language and the availability of a number of Scheme systems; many with out charge. But more important is the realization that this language contains the basic tools to model the essential principles of computer science and technology in a way unprecedented in previous languages. Scheme can be used directly as a language of exposition for the concepts that we want to teach in ways that are more direct and less cluttered by arcane syntax then trying to do the same exposition in Basic, C, Pascal or Fortran.

It is well understood that Scheme is an ideal notation of exposition for computer science topics when dealing with computer science and engineering students. [Abel 85], [Spri 89] and [Frie 92]. The focus of this paper is to argue that Scheme is an ideal notation for describing the important ideas of computer science to liberal arts students. Recently, the authors have experimentally used Scheme in two introductory computer science courses which are designed primarily for liberal arts and humanities students rather than science or engineering students.

The first course, **Computers and Society**, has approximately 1/3 content of problem solving, algorithms and programming; 1/3 content of a variety of computer applications including topics of what computers can and cannot do and 1/3 content studying the impact of computers technology on society.

The second course is a laboratory science course, **Great Ideas in Computer Science,** consisting of lectures lasting one week each on twelve core computer science topics ranging from computer organization to artificial intelligence. This course has a co-requisite contained laboratory course where students perform 13 prepared laboratory experiments.

An Approach for Students In the Liberal Arts.

Our experience has been that students in the liberal arts are more than slightly awed by the computer. They have had to accept that computer based word processing is a necessary activity that they have to perform, but, in their minds, they are using what to them is a complex typewriter. The thought that they could ever share with the computer some common procedural task fills them with fear and apprehension. If they are going to be introduced to the principles of computers and computing it will have to be on their own terms. One can not start by bombarding them with strange terms like byte and mips which seems to be evidence of the computer scientist's inability to spell correctly.

Rather, we find computing must be introduced by emphasizing that it is merely a reapplication of things they already know how to do. They already know how to explain to to a friend how a task is to be accomplished. Well, now they are going to try to explain a task that they already understand to a computer in a similar way. Similar in that such explanations are done in both instances in a language that the other (whether a human or computer other) understands.

Just as different people understand different languages it should not be to big a leap of faith for a liberal arts student to accept that different computers understand different languages or that (and this may be even more relevant) that the same computer or person may understand different languages. Students are well aware that languages are used to communicate and such languages are characterized by a describable syntax. This syntax tells us whether a unit in that language, usually called a sentence, is properly formed. Properly formed sentences are made up of standard parts of speech such as nouns, verbs, pronouns, adverbs, etc.

Another characteristic of languages is semantics. It should be possible to assign a meaning to the syntactically correct sentence with its component parts. There is the possibility, of course, that a sentence, though properly formed from a syntactic point of view, is meaningless or ambiguous.

In teaching liberal arts students about programming, we find it helpful to emphasize the similarities between what one might call the essential or no frills structure of programming languages and the natural languages with which they are already familiar. In describing the syntax and semantics of Scheme, this analogy seems especially worthwhile and palatable to the students we have had contact with in our classes.

What we call sentences in Scheme (and what others might call expressions) are a list of elements enclosed in parentheses. In a properly formed sentence the first item in this list is a verb. Following the verb, in a sentence, can be nouns, pronouns, verbs or subordinate sentences which are the objects of the sentence verb.

```
(<verb> <object> ...)
```

As one would expect, a simple sentence would not contain subordinate sentences.

For example,

```
(+ 2 3)
(* 2 (+ 2 3))
```

The latter is an example of a compound sentence.

A noun, from out point of view, is a self defining item whose meaning never changes. It might be a number, a list or a string. A pronoun is an item that is used in place of the noun. The meaning of a pronoun must be determined form the context of the sentence. It is what one would normally call a variable or a parameter.

A verb phrase in Scheme can either be a simple verb (or what we might more commonly call a function) or it might be what we describe as adverbial phrase, that is a sentence whose meaning results in a verb.

It is possible (and even useful when communicating with liberal arts students) that the verb of a sentence is, itself, the result of a sentence. Such compound sentences do not seem to occur in English, however, the authors have not found this a problem because this kind of compound sentence may be explained as being similar to the way an adverbial phrase modifies a verb deriving a new verb.

Why do we go to some much trouble to give these things such un-computerese names? Because we are dealing with people for whom use of common used computer science terms are so foreign that they serve as a sometimes impenetrable barrier to the learning process.

Scheme as the Language

If one accepts the suggestion of ephasizing to novice students the analogy between the physical structure (i,.e., the syntax) of programming languages and natural languages, Scheme provides another advantage as the language of choice to be used in this way. That is, that its syntax is so simple. Students are aware that it is hard work to become familiar enough with a natural language which is not their native language before one is confident of being able to construct syntactically correct sentences in that language. It can be pointed out very quickly that to learn the structure of a syntactically correct sentence in Scheme is a job of no more than a few minutes. They may not yet know what the sentences mean but they can become very quickly "experts" in detecting incorrectly formed sentences in Scheme.

It might be more difficult to convince beginning students that the semantics of scheme is also easy to learn. On one level, the semantics of Scheme is readily grasped. How verbs or verb phrases are applied to the nouns and pronouns in their scope can, we believe, be readily grasped. The order in which verbs are applied in sentences with more than one verb is a little more difficult but we believe can also be readily absorbed by students by relying on an analogy with what they already know about the language of function application in high school algebra.

However, if we focus on the semantics of scheme on a different level we have more of a problem. We all know that in any language there may be a difference between knowing what the words mean individually and knowing what the sentence means as a totality. In natural language we rarely resort to recursive definitions when we explain to someone how to accomplish a task. While in a language like Scheme recursion is a major tool used in carrying out algorithms and iterative or enumerated processes. Therefore, in our experience the semantics of recursion has to be approached very carefully with liberal arts students (and even many hard-core science students).

The semantics of special forms is another topic that must be carefully approached with these students as an embodiment of an idiosyncratic syntax which follows special rules. Here we are fortunate to be able to fall back on analogies with the many idiosyncratic semantic examples from the students own natural language.

One, semantic related problem with some scheme implementations should be noted in passing. We, as computer scientists, feel comfortable with the notion that there are some expressions that return values, some that do not and also some expressions whose return values are unspecified because they are being evaluated for a side effect. There is at least one implementation of Scheme that arranges for the third type of sentence to return the result, <unspecified>. From our observation the novice student finds this choice of return value some what unsettling. For example, to specify a value for a symbol they write:

```
(define a 10)==> <unspecified>
```

Which seems to imply, perhaps, that something was wrong with the specification. This is an illustration of how we need to be very sensitive to aspects of the semantics of a programming language that has special meaning to us as technologists but serves as a barrier for understanding among a more general audience.

References

[Abel 85] Abelson, Harold and Sussman, Gerald with Sussman, Julie. *Structure and Interpretation of Computer Programs*, MIT Press, 1985.

[Spri 89] Springer, George and Friedman, Daniel. *Scheme and the Art of Programming*, MIT Press, 1989.

[Kon74] Konstam, Aaron and Howland, John E. "APL as a Lingua Franca in the Computer Science Curriculum", SIGCSE Bulletin 6 (1), February 1974.

[Frie 92] Friedman, Daniel, Wand, Mitchell and Haynes, Christopher. *Essentials of Programming Languages*, MIT Press, 1992.