Teaching computer languages and elementary theory for mixed audiences at university level

Henning Christiansen

Roskilde University, Computer Science Dept.
P.O.Box 260, DK-4000 Roskilde, Denmark
E-mail: henning@ruc.dk

Abstract. Theoretical issues of computer science are traditionally taught in a way that presupposes a solid mathematical background and are usually considered more or less unaccessible for students without this. An effective methodology is described which has been developed for a target group of university students with different backgrounds such as natural science or humanities. It has been developed for a course that integrates theoretical material on computer languages and abstract machines with practical programming techniques. Prolog used as meta-language for describing language issues is the central instrument in the approach: Formal descriptions become running prototypes that are easy and appealing to test and modify, and can be extended into analyzers, interpreters, and tools such as tracers and debuggers. Experience shows a high learning curve, especially when the principles are extended into a learning-by-doing approach having the students to develop such descriptions themselves from an informal introduction.

1 Introduction

The advanced studies in Computer Science at Roskilde University are offered for students with a variety of different backgrounds such as natural science, humanities, and social science basic studies. A distinct characteristic at Roskilde is that 50% of the students' work consists of project work which means that the nominal time for lectures in the advanced studies is rather small compared with most other universities and there is an obvious danger that the education may concentrate on practical and application oriented aspects without the penetration of theoretical insight and understanding that is expected from a university degree.

This situation has motivated the development of a new methodology for teaching theoretical aspects of computer science which has proven its effectiveness and that we believe can be inspiring in other teaching contexts where a high and uniform mathematical knowledge cannot be expected. Recent educations that integrate humanities with information technology and elements of computer science seem to be a field where methods such as those we have developed seem appropriate.

The methodology has been developed for a course that integrates theoretical material on computer languages and abstract machines with practical programming and language processing techniques. The notion of computer languages is meant in a wide sense that covers a range of technologies and phenomena such as user interface languages, software packages, programming languages, (abstract and real) machine languages, and the language of Turing machines.

Experimentation and learning-by-doing are important in the approach but instead of using specialized language processing tools we have relied on the logical programming language Prolog: It is used as a meta-language that combines formality and high expressibility with an interactive environment for incremental testing of specifications. Prolog is experienced by students as a conceptually simple, transparent, and uniform framework which they can use as a practical tool also outside the course. Prolog is also an interesting object of study by itself, being a refreshing alternative to object-oriented languages such as Java which in Roskilde and many other universities is the students' first programming language.

The approach shows a high learning curve in computer science topics usually considered to be difficult and complicated; we can mention examples such as implementation of recursion in programming languages which is taught by having the students to implement an interpreter, and Turing completeness and the halting problem where the students experiment with an interpreter consisting of very few lines of Prolog.

Finally, traditional methods for language processing are introduced such as lexical analyzers and parsers, and the use of interpreters and translators as components in practical systems' development.

1.1 The conditions for teaching computer science at Roskilde University

The study for a Master's degree at Roskilde University takes five years and starts with two years of basic studies in one of Natural Science, Humanities, or Social Science. Following, each student chooses two different advanced topics, each corresponding to one and a half year of study, and one of which can be Computer Science. The student writes a Master's Thesis in one of his or her advanced topics, and there is also an option of integrating the two topics into an extended thesis work. A distinct characteristic of all studies at Roskilde is that 50% of the students' work is problem-oriented project work, thus leaving a net 3/4 year for course work at the advanced studies in Computer Science.

It is a challenge to design the overall study plan and individual courses: In the same class, some students may have strong mathematical qualifications while others have a very superficial, if any, relation to mathematics. Students are supposed to have spent about one half year of their basic studies with introductory computer science topics, including elementary programming, but this is often not the case. Other students may have several programming projects in their portfolio when arriving for the advanced studies. Some students have their full involvement in computer science and produce a Master's thesis here, while for others, computer science is simply their "secondary" advanced topic.

Certain issues that are important for all sorts of teaching become extra critical in this context. First of all, the presentation needs to be appealing and fruitful for every single student in this heterogeneous audience. Secondly, extreme care must be made in the selection of topics in order to provide a coherent course with a reasonable covering, considering that each course has few nominal hours. Finally, each course must be designed as a component of a full education comparable with any other five-year university education with computer science as a major subject.

The students' uneven mathematical background is another big challenge in the teaching of these inherently mathematical and logical topics. In practice, however, this problem is solved by the applied combination of rigorous structure and hands-on experience: It appeals to the humanities students' trained ability to think at a high conceptual level which to a large extent can replace mathematical experience. The mathematically inclined students can easily fill in any remaining detail and, on the basis of this course, approach the hardcore literature on the topics by themselves.

The course we refer to corresponds to one quarter of a semester (7.5 ECTS points), which is implemented as 10 full course days plus home work. In the appendix, a sketch of a course schedule is given.

An earlier version of the course was integrated in a full semester course with the theme "Language and Technology" which included also machine and system architecture, relational databases, introduction to computer networks, and a project work.

1.2 Background and related work

A fundamental idea in our approach is to see each computer language as the representative of an abstract machine in the sense introduced in the fundamental paper by Dijkstra [13]. Dijkstra applies the notion for program modules, noticing that each such introduces its own (small) nomenclature and semantics so that programming is seen as a matter of building and combining abstract machines. The same principle has been applied under the name of virtual machine to capture the different layers of hardware and software that comprise a modern computer in Tanenbaum's incomparable book [32] "Structured Computer Organization" (editions 1976, 1984, 1990, 1999).

Our style of semantic description in Prolog is a descendant of approaches to formal descriptions of programming languages initiated by Hoare's axiomatic semantics [17], denotational semantics [21], and similar approaches. There is, in fact, a strong similarity between our defining interpreters and Plotkin's operational semantics [27] which also is described in [38], the main difference is that our version is written in an executable language. Another interesting tool is Mosses' SIS system [23] that executes denotational semantic definitions specified in the lambda-calculus.

The book by Abelson and Sussman [1] applies the functional programming language Scheme for executable specifications in their advanced textbook that goes much more in depth with programming language semantics than we do. This book is intended exclusively for a mathematically highly competent and inclined audience and so is the textbook by Slonneger and Kurtz [29] who use Prolog as their language for implementing denotational and axiomatic semantics. Although their goals are slightly different, their conclusion is similar to ours: "Prolog proved to be an excellent tool for illustrating the formal semantics of programming languages. ... these laboratory exercises were highly successful in motivating students. The hands-on experience helped demystify the learning of formal semantics." At the technical level, there is a minor difference as they use Definite Clause Grammars in contrast to our interpreters and compilers that work directly on abstract syntax trees conveniently written using Prolog's operator notation. In comparison, our direct use of abstract syntax trees decomposed by unification (as opposed to lists of tokens decomposed by syntactic patterns) provides simpler descriptions that emphasize the structural aspects of language as syntactic peculiarities are removed completely.

The recent textbook by Tucker and Noonan [34] takes an interesting approach by motivating the formal study of syntax, type systems, and denotational semantics using a small imperative language and a set of implementation tools and student projects written in Java. In comparison with our use of the declarative Prolog language for executable specifications, Tucker and Noonan's use of Java and its object-oriented facilities focuses the students' attention more directly towards the development of practical and robust language processors. In addition, this textbook gives a comprehensive covering of different programming language paradigms and concepts, including event-driven and concurrent programming.

The remarkable book "On Pascal Compilers" [15] must also be mentioned. It is written as an introduction to compiling using a Pascal compiler written in Pascal as case study. The entire source text is shown and explained in the different chapters of the book, and the author has produced a piece of art in terms of readable and well-structured code built around a recursive-descent parser.

It may be possible that our approach can be transferred to a functional programming setting using instead, say, Haskell [3] or ML [22] as meta-language. These languages may offer many of the same advantages as Prolog but the idea still needs to be tested in practice.

Compiler writing in Prolog has been considered by several authors, e.g., [36, 24]. Semantics of programming languages specified in Prolog or Prolog-like languages (often implemented on top of Prolog) is not uncommon, e.g., [4, 6, 12, 26]. The close relation between attribute grammars [19] and Prolog has also been inspiring for the referenced works, evident in the notation applied by [37] and formally spelled out by [11].

Courses that go further into compiler construction often involve an assignment for the students to produce a running compiler for some small language, and dedicated tools can be used such as the classical Unix tools, Lex and Yacc [18], and Tanenbaum's ACK tool kit [33]. See also [39] who reports a project-oriented approach to teaching of compiler construction.

Finally we mention a few excellent textbooks on programming language semantics [30, 38] which are inspiring background for any advanced student and teacher of such topics, but which are inaccessible for the major part of the student community we address with the present approach.

1.3 Overview

In the following we start explaining the properties of Prolog which made it the obvious choice for our purposes. Next, we describe the abstract machine model that underlies our teaching methodology; defining interpreters written in Prolog are introduced as part of this model. A precise understanding of sequential machine languages with their extensions into imperative and procedural programming languages is important knowledge for any computer scientist; we explain the Prolog models used to describe these phenomena and how they are used in the teaching: Semantics of machine language and while-programs by defining interpreters; a compiler from while-programs to machine language which also serves as a general introduction to compiling; type checking and semantics for procedural languages with recursion. The following section shows other applications of Prolog as meta-language, including for describing a LISP-system, Turing machines, and a self-interpreter for Prolog which is extended into a simple tracer and debugger; relational algebra is introduced by having the students produce an evaluator for it. It is explained briefly how traditional methods for syntax analysis are fitted into a course based on our principles. A final section is included with conclusions and perspectives; an appendix gives an example of a course schedule.

Our methodology is currently documented in a locally printed textbook [8] in Danish, which still needs to be matured into an internationally publishable edition.

2 Why the strong emphasis of Prolog

Prolog is an obvious second programming language that shows the diversity of the field for students brought up with a language such as Java. Prolog is a typeless language in which any data structure has a denotation and with no need for constructors and selection methods as these are embedded in Prolog's unification. Java, on the other hand, requires the programmer to produce large collections of classes, interfaces, methods, and a test main method before anything can be executed. The conflict between flexibility and conciseness on the one hand, and security and robustness on the other is so obviously exposed in this comparison. In our teaching we have used the first part of [5] as a textbook that we also recommend to the reader who needs an introduction to Prolog.

Prolog appeals to an interactive and incremental type of program development that is in strong contrast to Java and an object-oriented methodology. In

Prolog one may start to write and test a program in order to achieve an understanding of some domain in question which seems not to be legitimate or practical in the typed and objected-oriented Java view of the world. This is also the spirit in which Prolog is presented to the students and the way it is applied subsequently in the course.

A study of Prolog motivates also considerations about the notion of a meta-language: `assert` and `retract` take arguments that represent program text, the same goes for Prolog's approximation to negation-as-failure which essentially is a meta-linguistic device within the language (see [16, 10] for discussion). The problematic semantics of these features gives rise to a discussion of what requirements should be made to a meta-linguistic representation. Operator definitions in Prolog comprise syntactic meta-language within the language, and are also a perfect point of departure for a detailed treatment of priority and associativity in programming language syntax.

When using Prolog as general purpose meta-language for computer language notions (including programming languages), we rely on the following properties.

– Prolog terms with operator definitions provide an immediate representation of abstract syntax trees in a textually pleasing form; see the following expression which with one operator definition for "`:=`" is a legal Prolog term:
  ```
  while( x<y, (x:= x+y ; y:= y+1))
  ```
– Structurally inductive definitions are expressed straightforwardly in Prolog by means of rules and unification, e.g.,
  ```
  statement(while(C,S),···):- condition(C,···),
        statement(S,···), ···.
  ```
– Data types for, say, symbol tables and variable bindings, are easily implemented by Prolog structures and a few auxiliary predicates.
– Last but no least: Prolog appears as an easily accessible framework compared with, say, set and domain theory. Specifications are directly executable and can be monitored in detail using a tracer; they can be developed and tested incrementally and interactively. Students can easily modify or extend examples and test their solutions.

Typically, Prolog is introduced in two full course days in a workshop setting: Short introductions combined with practical work at the computers. A striking feature of Prolog (as compared to any other programming language that we are aware of) is that it is possible to get around the whole language in such a short time. Students will have obtained an understanding of the core of Prolog and have written their own programs, and they have also tried to use the more tricky parts, including cut and assert/retract. At this point, Prolog can be considered a tool at hand for the students with their practical experience growing throughout the course.

The teaching of Prolog, in addition to the straight matters of how to use it, should also consider Prolog as an independent object of study, an instance of the phenomenon of a general programming language, to be compared with, say, Java along the lines discussed in the introduction to this section. General notions can be stressed such as abstract and concrete syntax, syntactic sugar, details about operator precedence, semantics and pragmatic issues. This "exterior" view of Prolog is complemented by presenting and having the students to dissect a running system written in Java for a subset of Prolog which includes a detailed implementation of Prolog's unification.

3 A basic model of abstract machines and languages

As mentioned in the introduction, the notion of abstract machines is used to characterize computer languages in the wide sense and distinguish them from other sorts of languages.

The model is unsophisticated and quickly introduced to the students. A particular *abstract machine* is characterized by its *input language* which is a collection of phrases or sentences, a *memory* which at any given time contains a value from some domain of values, and finally a *semantic function* mapping a phrase of the input language and memory state into a new memory state. For simplicity, output is not explicit part of the definition but considered as part of the "transparent" memory whenever needed.

Examples to show the versatility of the definition include machine language, programming languages, interfaces of program modules, user interface languages, and in order to stress the notion of state, a simple calculator and a database.

The model includes a general notion of *implementation* of one machine in terms of another, and three different modes are defined, *interpretation*, *translation* and use of *abstraction mechanisms* in standard programming languages. Details are straightforward and left out from the present paper.

Abstract and concrete syntax are introduced and distinguished in an informal way, and the representation of abstract syntax trees by Prolog terms is emphasized. The set of abstract syntax trees for a context-free language can be characterized by a recursive Prolog program consisting of rules of the form

$$cat_0\texttt{(}op\texttt{(T}_1\texttt{,..., T}_n\texttt{)):-}\ cat_1\texttt{(T}_1\texttt{),...,}cat_n\texttt{(T}_n\texttt{).}$$

where *op* names an operator combining phrases of syntactic categories $cat_1, \ldots, cat_n$ into a phrase of category cat_0.

Syntax-directed definitions can be specified by adding more arguments corresponding to the synthesized as well as inherited attributes of an attribute grammar [19]. Consistent with our abstract machine model, we introduce what we call a *defining interpreter* which to each syntax tree associates its *semantic relation* of tuples $\langle s_1, \ldots, s_k \rangle$ by predicates of the form

$$cat_i\texttt{(}syntax\text{-}tree\texttt{,}s_1\texttt{,...,}s_k\texttt{)}$$

Notice that we deviate slightly in the terminology from the abstract machine model referring here to semantic relation instead of a function; this is to conform with the semantics of Prolog although we do not actually use the inherent generalization to nondeterministic languages. As an example, a defining interpreter for an imperative language may associate with each statement a relation between variable state before and after execution, which for a statement such as "`x:= x+1`" contains among others the following tuples.

```
         ⟨[x=7] , [x=8]⟩
       ⟨[x=666] , [x=667]⟩
    ⟨[x=1,y=32] , [x=2,y=32]⟩
⟨[a=17,x=1,y=32] , [a=17,x=2,y=32]⟩
```

This is basically the systematic framework in which different programming language fragments are studied, and we show examples below that have been used in our course.

Interpreters (for toy languages at least) written in this fashion are typically brief and concise; they can be understood and tested rule by rule and are easy to extend and modify. We show also how compilation and type checking can be treated in similar ways.

4 Imperative and procedural languages

The semantics of the different layers of sequential machine language and of imperative and procedural languages, as well as their interrelation, are essential topics for an understanding of how computers and information technology work.

In the following we show how these phenomena are characterized in our Prolog-based style, putting emphasis on the spirit in which it is communicated concretely in the teaching. We proceed by introducing a defining interpreter for a simple machine-like language giving a continuation-style semantics for jumps and control points. This serves the dual purposes of making the semantics of such languages explicit and of introducing continuations as programming technique and as semantic principle. Continuations are often considered a very difficult concept to grasp but in the way presented here it appears quite obvious and natural. Next is shown a defining interpreter for while-programs, formulated in a straightforward recursive fashion without explicit continuations, thus emphasizing how it differs from the previous. Compilation of while-programs into machine language is easily formalized as a syntax-directed translation written in Prolog.

Through all these steps, students solve problems involving extensions and modification of the Prolog programs involved. We have tried different ways to teach topics related to type checking and implementation of recursive procedures; the most successful that we sketch below has been to have the students by themselves develop type checker and interpreter for a simple Pascal-like language from a brief and informal introduction to the notions.

4.1 A defining interpreter for a machine language

A machine language is characterized as sequences of simple state transformations executed in their textual order, however broken by jump instructions whose meaning depends on the labels in the current program. In order to provide a formal presentation to the students, we introduce a simplified machine language by means of the following sample program represented as a Prolog list. It is assumed that a computer with a running Prolog session attached to a projector is available in the lecture room.

The (yet) uncommented example, by the names chosen for the instructions, is intended to trigger the intuition of the existence of some abstract machine.

```
[     push(2),
      store(t),
   7, fetch(x),
      push(2),
      add,
      store(x),
      fetch(t),
      push(1),
      subtract,
      store(t),
      fetch(t),
      push(0),
      equal,
      n_jump(7)]
```

Without any further introduction, the teacher can execute this program by hand on the blackboard in a dialogue with the students. The semantics of such programs assumes a stack (that we can represent as a Prolog list) and a storage of variable bindings (represented conveniently as lists of "equations", e.g., `[a=17,x=1,y=32]`. Two auxiliary predicates are introduced in order to work with stores.

`store(`*VariableID*`,` *Value*`,` *Store*`,` *UpdatedStore*`)`
`fetch(`*VariableID*`,` *Value*`,` *Store*`)`

The necessary Prolog code to implement these are shown to the students with the behaviour tested in the lecture room or by having the students to do small exercises themselves.

The central predicate in a defining interpreter is the following. The first argument represents a sequence of instructions (a continuation) to be executed and the second one passes the entire program around to all instructions to give the contextual meaning of labels.

`sequence(`*Sequence*, *WholeProgram*`,` *CurrentStack*`,` *CurrentStore*`,`
FinalStack`,` *FinalStore*`)`

Before giving the details of this, we set up a definition for a whole program as follows.

```
machine_program(Prog, FinalStack, FinalStore):-
      sequence(Prog,Prog,[],[],FinalStack,FinalStore).
```

The meaning of simple statements that transform the state is given by tail-recursive rules such as the following: Do whatever state transition is indicated by the first instruction and give the resulting state to the continuation.

```
sequence([push(N)|Cont], Prog, S0, L0, S1, L1):-
      sequence(Cont, Prog, [N|S0], L0, S1, L1).

sequence([fetch(Var)|Cont], Prog, S0, L0, S1, L1):-
      fetch(Var,X,L0),
      sequence(Cont, Prog, [X|S0], L0, S1, L1).

sequence([add|Cont], Prog, [X,Y|S0], L0, S1, L1):-
      YplusX is Y + X,
      sequence(Cont, Prog, [YplusX|S0], L0, S1, L1).
```

These rules are tested on the computer and compared with drawings on the blackboard if needed. The similar rule for subtraction is a good point for discussing the order of the arguments:

```
sequence([minus|Rest], Prog, [X,Y|S0], L0, S1, L1):-
      YminusX is Y - X,
      sequence(Rest, Prog, [YminusX|S0], L0, S1, L1).
```

At this stage, the audience is warmed up for the more interesting cases. The unconditional jump instruction is defined as follows; it is assumed that the diverse usages of the `append` predicate have been exercised thoroughly with the students at an earlier stage.

```
sequence([jump(E)|_], P, S0, L0, S1, L1):-
     append(_, [E|Cont], P),
     sequence(Cont, P, S0, L0, S1, L1).
```

Executing a few examples, perhaps complemented by a drawing on the blackboard — and within a few minutes the students have grasped the principle of a continuation and continuation semantics.

The following two rules defining conditional jumps serve as an immediate repetition of the principle.

```
sequence([n_jump(E)|_], P, [0|S0], L0, S1, L1):-
     append(_, [E|Cont], P),
     sequence(Cont, P, S0, L0, S1, L1).

sequence([n_jump(_)|Cont], P, [1|S0], L0, S1, L1):-
      sequence(Cont, P, S0, L0, S1, L1).
```

Now we need only provide the rules for skipping over labels in a sequence and for stopping a run.

```
sequence([Label|Rest], P, S0, L0, S1, L1):-
     integer(Label),
     sequence(Rest, P, S0, L0, S1, L1).

sequence([],_,S,L,S,L).
```

The entire defining interpreter is now finished and can be tested on the sample program shown as introduction (and the students are receptive to the teachers proclamations concerning precise language definitions at the one hand and the implementation of various languages by an interpreter on the other).

As a final piece of candy, the following rule is added as the first one to the interpreter:

```
sequence([Inst|_],_,_,_,_,_):-
     write(Inst), write(' '), fail.
```

This turns the specification into a functioning tracer, thus emphasizing the advantages of having formal specifications integrated in a general purpose, interactive programming environment.

The following exercises are given to the students in order to provide a hands-on feeling and to give them an impression of the power of being able to design and implement themselves new language constructions and facilities.

- Extend language and interpreter with instructions for subroutines: `jump_sub(`*to-label*`,` *return-to-label*`)` and `return`. Provide an interesting sample program in the extended language for testing the solution.
- Write a Prolog program checking that labels are used in a consistent way (part of exercise is to define what that means). Prize is given for the most elegant solution.
- Examples are shown of how subsequences of instructions can be replaced by other and more efficient ones. Write a Prolog predicate that performs such optimizations.
- Design and implement an extension of the tracer so it becomes a debugger with possibility to change variables, affect outcomes of conditional jumps and (optionally) allow arbitrary number of undo's of execution steps. (The last topic is perfect training for those students who want to master the powerful control device provided by Prolog's backtracking.)

Moving up to the meta-pedagogical level, we conclude that this (part of a) lecture with exercises, built around a seemingly innocent example, in a compact but digestive way has established important pieces of knowledge and methodology that otherwise may be quite an obstacle for many students.

4.2 A defining interpreter for while-programs

The detailed comments to the previous examples have indicated the spirit in which we communicate an understanding of computer languages to the students; the following examples are given in a more compact way. Now we consider while-programs of which the following sample, representing Euclid's algorithm for greatest common divisor, is a prototypical example.

```
a:= 221 ; b:= 493 ;
while( a =\= b,
         if( a>b, a:= a-b, b:= b-a))
```

Notice that this expression, with its straightforward textual appearance, is a syntactically correct Prolog term representing an abstract syntax tree. A defining interpreter consists of the following predicates.

`program(`*program*`,` *final-storage*`)`
`statement(`*statement*`,` *storage-before*`,` *storage-after*`)`
`expression(`*expression*`,` *storage*`,` *integer*`)`
`condition(`*condition*`,` *storage*`,` {`true`, `false`}`)`

Some of the rules of this interpreter are shown in the following; the most important ones are for the `if` and `while` statements. Notice that we reuse the storage structure and auxiliaries from the previous example.

```
program(P, Storage) :- statement(P, [], Storage).

statement((Var := Expression), L1, L2):-
      expression(Expression, L1, Value),
      store(Var,Value,L1,L2).

statement( (S1 ; S2), L1, L3):-
      statement(S1, L1, L2),
      statement(S2, L2, L3).

statement( if(Cond, Smt1, Stm2), L1, L2):-
      condition(Cond, L1, Value),
      (Value = true -> statement(Stm1, L1, L2)
                              ; statement(Stm2, L1, L2)).

statement( while(Cond, Stm), L1, L2):-
      condition(Cond, L1, Value),
      (Value = true -> statement(
                (Stm ; while(Cond, Stm)),L1,L2) ; L1=L2).

expression(Variable, L, V):- atom(Variable),
      fetch(Variable,V,L).

expression( Int, _, Int):- integer(Int).

expression( (Exp1 + Exp2), L, Res):-
      expression( Exp1, L, V1),
      expression( Exp2, L, V2),
      Res is V1 + V2.

condition( true, _, true).

condition( false, _, false).
condition( (Exp1 = Exp2), L, Res):-
      expression( Exp1, L, V1),
      expression( Exp2, L, V2),
      (V1 = V2 -> Res = true ; Res = false).
```

Exercises are provided so that the students can experiment with and extend this defining interpreter in order to get a deeper understanding of how it works.

– Run a sample program which includes a while loop with Prolog's debugger switched on as to record all primitive actions involved.
– Extend the language with an expression of the form `result_is(` *statement*, *variable*) with the intended semantics that the *statement* is executed and then the value of the variable defines the value of the expression. Special attention should be paid to possible side-effects on other variables.
– Extend the language with a `for` loop.
– Extend the interpreter with a simple tracing facility.

In case the students have been presented earlier for Hoare logic, for instance in a previous programming course, there is another good exercise in formalizing this as an interpreter in Prolog.

4.3 A compiler for while-programs

The structure of our defining interpreters can also be adapted to describe compilers. Above, we considered a semantics for while-programs defined in terms of state transformations and now we consider an alternate semantics capturing meanings by means of sequences of machine instructions.

We introduce two auxiliary predicates, one to generate new unused target language labels and another one providing syntactic sugar for putting together sequences of instruction sequences and single instructions. They are illustrated in the following example query.

```
?- new_label(L1), new_label(L2), C1 = [push(1),add],
   C2 <- L1 + push(7) + L2 + C1.

L1 = 117
L2 = 118
C1 = [push(1),add]
C2 = [117,push(7),118,push(1),add]
```

Depending on the level of their experience in Prolog programming, the students may be given as an exercise to program these auxiliaries or the definitions are presented in the lecture. Now a simple, non-optimizing compiler for while-programs can be presented as follows (selected rules only).

```
program(P, K):- statement(P, K).

statement((S1 ; S2), C):-
      statement(S1, C1),
      statement(S2, C2),
      C <- C1 + C2.

statement((Var := Exp), C):-
      expression(Exp, C1),
      C <- C1 + store(Var).

statement( if(Cond, Stm1, Stm2), C):-
      condition(Cond, CondC),
      statement(Statement1, C1),
      statement(Statement2, C2),
      new_label(L2), new_label(L_end),
      C <-    CondC +
              n_jump(L2) +
              C1 +
              jump(L_end) +
         L2 + C2 +
         L_end.

statement( while(Cond, Stm), C):-
      condition(Cond, CondC),
      statement( Stm, C1),
      new_label(Lstart), new_label(Lend),
      C <-  Lstart + CondC +
                     n_jump(Lend) +
                     C1 +
                     jump(Lstart) +
            Lend.

expression(Number, C):-
      integer(Number),
      C <- push(Number).

expression( Variable, C):-
     atom(Variable),
     C <- fetch(Variable).

expression((Exp1 + Exp2), C):-
      expression(Exp1, C1),
      expression(Exp2, C2),
      C <- C1 + C2 + add.
```

These rules can be tested one after one during the lecture as they are introduced. Finally, we can combine the compiler with the defining interpreter for machine programs as follows.

```
?- program( ( a:= 221 ; b:= 493 ;
              while( a =\= b,
                     if( a > b,
                         a:= a-b,
                         b:= b-a))), C),
     machine_program( C, _, L).

C = [  stack(221),store(a),stack(493),store(b),
     2,fetch(a),fetch(b),not_equal,n_jump(3),
       fetch(a),fetch(b),greater,n_jump(0),
       fetch(a),fetch(b),minus,store(a),jump(1),
     0,fetch(b),fetch(a),minus,store(b),
     1,jump(2),
     3],
L = [a=17,b=17]
```

Exercises given to the students consist of adapting this compiler to the extensions proposed earlier in exercises related to the defining interpreter. The optimizer for machine programs considered in an earlier exercise can be applied at different level of granularity.

The purpose of presenting this little compiler to the students is manyfold: It illustrates the notions of a compiler and of syntax-directed translation and it makes the distinction between interpretation and compilation clear. It shows how standard imperative constructs are mapped into machine language and may serve as an appetizer for more serious studies of compilers, for example [2]. Finally, it serves as an introduction to the larger learning-by-doing exercise described in the following section.

4.4 Do-it-yourself recursive procedures

Instead of always presenting ready solutions to the students, it is also motivating, once they have become familiar with the principles, to let them work out nontrivial examples by themselves.

In the following, we sketch a larger exercise in which a class of students had to produce a type checker and an interpreter for a Pascal-like language with arrays and side-effects. The following, recursive quicksort program is prototypical; notice that an "^" operator is used for array-indexing.

```
program(
    (var(n,int); var(a, int_array(4))),
    declare_proc( qsort, left, right,
       (var(i,int); var(j,int); var(x,int); var(w,int)),
       (i:= left; j:= right; x:= a^( (left+right)//2) ;
        repeat( (while(a^i<x, i:= i+1) ;
                 while(x<a^j, j:=j-1) ;
                 if(i=<j, (w:=a^i; a^i:= a^j; a^j:= w;
                           i:= i+1; j:= j-1))),
        % until
                 i > j); % end repeat
        if( left<j, proc_call(sort,left, j)) ;
        if( i < right, proc_call(sort,i,right)) )
     ), % end proc qsort

     % main program:
     (n:= 4; a:= [30,10,40,20];
      proc(qsort,1,n); write(a)))
```

The syntax, including scope and type principles, and the semantics of the language were described informally to the students and their task was to produce type checker and interpreter to be tested on a number of sample programs, including the one shown above.

The students had programming experience in advance with this sort of language but the presentation of this assignment was their first systematic introduction to types and type checking. In order to simplify their work, they were given auxiliary predicates for working with symbol tables and runtime stacks, but with only a sketchy explanation of how to use them. So the students' task was to put together the whole machinery and test it.

The prescribed time for the work was one week on half time, including writing a small report documenting their solutions; they could work in groups of up to three students. The most experienced students had type checker and interpreter running after four or five hours, and all students in a class of some 30 students solved the task within the prescribed time. All solutions were acceptable and there was no obvious difference between those produced by students with a mathematical background and by those without. In general, the students characterized this exercise as a difficult and challenging one, but also that it had been one of the most interesting ones from which they had learned quite a lot.

We show some fragments of a possible solution. Let us make precise some assumptions about the language. Procedures take always two integer parameters, and variable declarations may introduce integer and array variables. There are no local procedures, so the runtime stack can be organized as a list of stack frames, each being a list of bindings; looking up a variable can be done by looking first in the topmost frame and if not found here, in the bottom frame. Recursive calls are possible within a procedure and to previously declared procedures.

The type checker can be defined by a predicate `tc_`*cat*`(`*tree*`,` *current-table*`,` *updated-table*`)` for those syntactic categories *cat* whose phrases are intended to introduce new nomenclature, and with fewer arguments for other syntactic categories. A sufficient type checker rule for a single procedure declaration is the following.

```
tc_proc_decl(
  declare_proc(ProcId,ParId1,ParId2,LocalVarDecls, Stm),
  Table1, Table2):-
    tc_identifier(ProcId),
    Table2 = [(ProcId, procedure2) | Table1],
    tc_identifier(ParId1), tc_identifier(ParId2),
    Table3 = [(ParId2, int),(ParId1, int)|Table2],
    tc_var_decl(LocalVarDecls,Table3,Table4),
    tc_statement(Stm,Table4).
```

Correct typing of a procedure call is expressed in the following way.

```
tc_statement( proc_call(ProcId, Exp1, Exp2),Table):-
    symbol_tabel_find(ProcId, Table, procedure2),
    tc_expression(Exp1,Table,int),
    tc_expression(Exp2,Table,int).
```

For the interpreter, we give the flavour of a solution by showing the most complicated rule which is the one for procedure calls. Each statement is executed relative to a table of procedure closures plus a runtime stack and produces an updated runtime stack. The procedural meaning of local variable declarations is to extend a current stack frame with “locations” for the variables as to produce a new frame.

```
statement(proc_call(ProcId, Exp1, Exp2),
                  ProcTable, Stack1, Stack2):-
    member( proc(Id,ParId1, ParId2, LocalVarDecls, Stm),
                  ProcTable),
    expression(Exp1, Stack1, ParValue1),
    expression(Exp2, Stack1, ParValue2),
    var_decl(LocalVarDecls,
      [(ParId2,ParValue2),(ParId1,ParValue1)], StackFrame),
    statement(Stm, ProcTable, [StackFrame|Stack1], [_|Stack2]).
```

Although the overall structure of rules as the one above is simple, the correct positioning of the logical variables is a difficult task. Here Prolog’s short cycle of incremental program development and testing is a great advantage. Having completed this rule, the student has gained a very clear understanding of what goes on when a procedure is activated in a call-by-value language.

5 Other course elements involving Prolog as meta-language

We sketch briefly a number of other examples which have been used in our course. Logic circuits modelled in Prolog is a standard example used in many Prolog text books and is obvious to apply in our context due to the meta-linguistic aspects (modelling the language of logic circuits); we refrain from giving details. The following examples show different aspects of Prolog as well as other languages and programming tools.

5.1 LISP modelled with assert-retract

This example goes definitely to the limit of our paradigm of using Prolog as a logical specification language. In this way, the presentation becomes a bit provocative and can initiate discussions about what requirements should be made in general to a specification and to a meta-language.

By means of Prolog's `assert` and `retract` facilities, we define an interpreter for a small LISP-like language with function definitions and variable assignments modelled as side-effects. This way we provide a model of an interactive LISP environment [20]. Here follow a few rules that show the principle; notice also the pragmatic aspect present in error messages in some rules.

```
lisp([quote,X], X).

lisp( [plus,X,Y], Value):-
    lisp(X, Xvalue),
    lisp(Y, Yvalue),
    Value is Xvalue + Yvalue.

lisp([car,X], Value):-
    lisp(X, Xvalue),
    (Xvalue = [Value | _] -> true
     ;
     nl, write('CAR of non-list: '),
     write(Xvalue), abort).

lisp([setq,Var,X], Xvalue):-
    lisp(X, Xvalue),
    asserta((lisp(Var, Xvalue):- !)).
```

The last rule gives rise to a discussion of binding times and a critique of Prolog for the lack of indication of different binding times for `Var`, `X` and for `Xvalue`; this is another way of showing the problems inherent in the nonground representation of Prolog in itself.

The following rule for function definitions with a single parameter emphasizes the problem but shows also many interesting programming language aspects such as extensibility, parameter transmission and, again, different binding times.

```
lisp([defun, F, Param, Body], F):-
    asserta((lisp( [F, Arg], Value):- !,
                lisp(Arg, ArgValue),
                asserta((lisp(Param, ArgValue):- !)).
                lisp( Body, Value),
                retract((lisp(Param, ArgValue):- !)) )).
```

This rule gives the teacher a good reason to criticize the nonground representation for very practical reasons: It makes the specification almost unreadable. This suggests the design of a new syntax (from [7]) for a ground representation with one or more prefix asterisks to indicate binding time for represented variables.

```
lisp([defun, F, Param, Body], F):-
    new_asserta(
        (lisp([F, *arg], *value):- !,
           lisp(*arg, *argValue),
           new_asserta((lisp(Param, *argValue):- !)).
           lisp(Body, *value),
           new_retract((lisp(Param,*argValue):- !)))).
```

The exercises given to the students are the following:

– Extend the interpreter to handle the `eval` function and test it on given examples.
– Implement a version with call-by-name parameters.
– Examine the given interpreter and add complete error messages; what does "complete" mean?
– Analyze the interpreter to figure out what happens when formal parameter names are `setq`ed inside the body of a functions. Discuss different possible semantics and test them.

As options for the more advanced students having a special interest in Prolog programming, the following extra tasks are proposed.

– Write a definition of a nullary predicate `run_lisp` that adds a read-eval-print loop upon the interpreter.
– Add a debugging facility to the interpreter.
– Implement the suggested `new_asserta` and `new_retract` predicates.
– Write program transformers that can apply to the body of function definitions, e.g., getting rid of explicit parameter references and using substitution by means of Prolog variables instead.

5.2 Turing machines

Most computer science educations at university level do not leave much time for the study of computability and decidability issues. This may be because other and more practical issues are given higher priority and the fact that an in-depth study of these matters is not considered so relevant for a majority of students.

However, we find it important for historical and philosophical reasons that students are given an introduction so they know that such issues exist and have some idea of what they mean, so to speak giving a qualified impression of what computers can and can't.

Again, a running implementation of an interpreter in Prolog is an excellent way to illustrate a Turing-machine and to provide a truly dynamic model, especially when a tracing facility is added. The definition of such an interpreter is straightforward and not shown here. The existence of the interpreter shows that Prolog is Turing-complete, and playing with it warms up the student for the proof of undecidability of the halting problem. Exercises consist of writing small Turing-machines (including a "copy machine" often used in the mentioned proof) and extending the interpreter to handle multi-tape machines.

5.3 Playing with Vanilla and Prolog source-to-source compilation

The familiar Vanilla self-interpreter for Prolog [35] is a perfect example to illustrate the notion of a self-interpreter.

```
solve(true).
solve((A,B)):- solve(A), solve(B).
solve(A):- clause(A,B), solve(B).
```

It may appear a bit absurd and useless to the students until we begin modifying it into a tracer by adding the following material to its last rule.

```
solve(A):-
    trace_code((write('Enter '), write(A), nl),
               (write('Fail '), write(A), nl)),
    clause(A,B),
    trace_code((write('Try '), write((A:- B)), nl ),
               (write('Drop '), write((A:- B)), nl)),
    solve(B),
    trace_code((write('Succeed '), write(A), nl)).

trace_code( Forwards, Backwards):-
    Forwards ; Backwards, fail.
```

Further extensions make it into a debugger which allows the user to affect program execution similarly to standard Prolog debuggers.

Efficiency measuring of programs can also be incorporated in the interpreter, but we can also use source-to-source compilation instead (and thus provide an opportunity to show this phenomenon). Half a page of Prolog code can implement a translator that `retract`s each clause of the form

Head :- *Body*

and `assert`s another one of the form

Head :- *CountClauseEntranceAndBacktrack* , *Body* ,
CountClauseExitAndRe-entrances

where the added pieces of code maintain global counters for each clause.

This is an entertaining and systematic way to study and characterize programming tools, and interesting exercises can be given to the students of extending or implementing similar tools.

5.4 Do-it-yourself relational algebra

In a section above we showed how type checking and implementation of recursive procedures can be taught by having the students to develop an implementation in Prolog. We have used the same methodology in a broader scoped course for an introduction to relational algebra, and we may suggest it be applied also in a standard database course.

A small example of a database is informally introduced with the notions of a relational schema (with named, untyped attributes) and database tuples, and operations `union`, `intersect`, `where` ⟨*simple-condition*⟩, and `join` where the latter is defined in terms of coinciding attribute names. A representation of tabelled relations is shown with schema and tuples given as Prolog facts, e.g.:

```
schema(costumer, [costumer_no, costumer_name, costumer_city]).
tuple( costumer, [k17, jensen, roskilde]).
tuple( costumer, [k29, hansen, copenhagen]).
```

The students' task is now to complete the definitions for the `schema` and `tuple` predicates so that an arbitrary relational expression can be interpreted as first argument.

The conditions were the same as for the task on recursive procedures described above, one week on half time, including writing a small report documenting the solutions. This task has been given to several classes of students and all students usually succeed in producing an acceptable solution, although `join` often causes problems. The students' comments on working with this task are usually very positive.

6 Syntax analysis

Traditional methods for lexical analysis and parsing are integral components of our course. Prolog is used as a ready-at-hand tool for the students to implement finite state machines, deterministic as well as nondeterministic. Top-down parsing is illustrated perfectly by Prolog's built-in Definite Clause Grammars [25], and bottom up-parsers by an analogous grammar formalism CHRG [9] developed on top of Constraint Handling Rules [14] which is a recent extension to some

Prolog versions (e.g., SICStus Prolog [31]) that provides a natural paradigm for bottom-up evaluation. A better choice may be to stay within a Prolog framework also when considering bottom-up parsing; we refer to [28] who show how this can be done.

Now a quick and effective introduction can be given to standard implementation principles for finite state machines (table controlled or compiled into control structures) and for top-down parsers in the shape of an LL(1) recursive-descent parser.

As a conclusion of the course, the students are shown a full implementation in Java of a subset of Prolog, including lexical analysis, parsing, representation of abstract syntax trees in an object-oriented language, and an interpreter which exposes a detailed implementation of Prolog's unification procedure.

7 Conclusion and perspectives

Theoretical computer science issues can be taught in an entertaining and concrete way which, unlike traditional approaches, appeals to a wide range of students for which a uniform mathematical background cannot be taken for granted.

We have explained a methodology based on a combination of a simple, underlying model of abstract machines and the use of Prolog as general definition and implementation language. A combination of qualities of Prolog makes it well suited for the purpose: Conceptual simplicity and high expressibility with a core language consistent with a subset of first-order logic; syntactic extensibility that allows a direct notation for abstract syntax trees in a textually acceptable form; a rule-based structure that fits perfectly with an inductive style of definition. Last but not least: Prolog is an interactive language that appeals to incremental development, testing, and experimentation with an extremely short turn-around time from idea $\rightarrow$ implementation $\rightarrow$ observation $\rightarrow$ revision or extension of idea.

Another approach to teaching these concepts is to apply a collection of different tools, for instance specialized compiler writing tools and animated tools with fancy graphics for, say, Turing machines, machine languages, etc.; several such tools can be found on the Internet. However, such a collection of tools lacks transparency seen from the student's mind. Prolog as we have used it here, has declarative and operational semantics that are fairly easy to grasp for a start, and its use as primary tool throughout the course strengthens this understanding.

With the methodology we have described, we want to stress that the concepts treated are not just theory for its own sake, but contain valuable understanding about highly practical matters. Lexical analysis and parsing are indispensable for any program developer, and interpretation and translation are powerful tools for the development of larger systems. Even the technique exposed in the toy-like defining interpreters provides a very powerful tool for experimental prototype development.

The methodology has been applied with good success at Roskilde University and looking back, it appears to have served two purposes: Providing a solid minimum of knowledge that every computer science student needs indifferently of specialization, and secondly as an appetizer and introduction for those wanting to specialize in compiler construction and other language implementation techniques or deeper studies in theoretical computer science.

8 Appendix: A sample course schedule

The following table shows the schedule for a version of a course designed according to our methodology as it was given in spring 2001. The actual course may change slightly from semester to semester so not all items mentioned in the present paper are found in this particular schedule. The course corresponds to 25% of a student's work in one semester and is concentrated on 10 full course days. When nothing else is indicated below, each course day consists of lectures and practical problem solving related to the day's lecture. A considerable amount of homework is expected from the students.

1	Introduction: Abstract and concrete syntax, semantics, pragmatics, language and meta-language. Prolog workshop I: The core language, incl. structures.
2	Prolog workshop II: Lists, operators, assert/retract, cut, negation-as-failure.
3	Abstract machines: Definitions of a.m., interpreter, translator, etc. Prolog workshop II contd.
4	Language and meta-language, Prolog as meta-language. Semantics of sequential and imperative languages; defining interpreters and a small compiler.
5	Declarations, types, type checking, context-dependencies, recursive procedures.
6	Introduction to and practical work with large exercise: do-it-yourself recursive procedures, interpreter and type checker.
7	Conclusion and comments to large exercise. Turing-machines, decidability and computability, Turing universality, the halting problem, Turing machines in Prolog.
8	Constraint logic programming: Introduction to CLP(R) and CHR; CHR Grammars for bottom-up parsing.
9	Syntax analysis: Lexical analysis and parsing; recursive-descent parsing
10	Overview of phases in a traditional compiler. Dissection of an implementation of Prolog in Java. Evaluation of the course.

Acknowledgment: The do-it-yourself relational algebra task has been developed in collaboration with my colleague at Roskilde University, Troels Andreasen. Thanks to Niels Jørgensen for helpful comments to an early version of this paper. This research is supported in part by the IT-University of Copenhagen.

References

1. Harold Abelson, Gerald Jay Sussman, and Julie Sussman. *Structure and Interpretation of Computer Programs.* MIT Press, USA, 6 edition, 1985.
2. A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers, Principles, Techniques and Tools.* Prentice-Hall, 1986.
3. Richard Bird. *Introduction to Functional Programming using Haskell.* Prentice-Hall, Second edition, 1998.
4. P. Borras, D. Clement, T. Despeyroux, J. Incerpi, and G. Kahn. CENTAUR: The System. In *Proceedings of the Third ACM SIGSOFT '88 Symposium on Software Development Environments*, pages 14–24, November 1988. Published as SIGSOFT Software Engineering Notes, volume 13, number 5.
5. Ivan Bratko. *Prolog Programming for Artificial Intelligence.* Addison-Wesley, Reading, Massachusetts, third edition, 2001.
6. B. R. Bryant and A. Pan. Rapid Prototyping of Programming Languages Semantics using Prolog. In *Proc. of COMPSAC 89, Int'l Computer Software and Applications Conference*, pages 439–446. IEEE, 1989.
7. Henning Christiansen. Declarative Semantics of a Meta-Programming Language. In M. Bruynooghe, editor, *Proceedings of the Second Workshop on Metaprogramming in Logic, April 4–6, 1990, Belgium*, pages 159–168, 1990.
8. Henning Christiansen. Sprog og abstrakte maskiner, 3. rev. udgave [in Danish; eqv. "Languages and abstract machines"]. Datalogiske noter 18, Roskilde University, Roskilde, Denmark, 2000.
9. Henning Christiansen. CHR Grammars. *Int'l Journal on Theory and Practice of Logic Programming*, 2005. To appear.
10. Henning Christiansen and Davide Martinenghi. Symbolic constraints for meta-logic programming. *Journal of Applied Artificial Intelligence*, pages 345–368, 1999.
11. Pierre Deransart and Jan Maluszynski. Relating Logic Programs and Attribute Grammars. *Journal of Logic Programming*, 2(2):119–155, 1985.
12. Thierry Despeyroux. TYPOL: A Formalism to Implement Natural Semantics. Research Report 94, INRIA, Rocquencourt, France, March 1988.
13. Edsger W. Dijkstra. Notes on Structured Programming. In E. Dijkstra, Ole-J. Dahl, and C. A. R. Hoare, editors, *Structured Programming*, pages 1–82, New York, NY, 1972. Academic Press.
14. Thom Frühwirth. Theory and Practice of Constraint Handling Rules, Special Issue on Constraint Logic Programming. *Journal of Logic Programming*, 37(1–3):95–138, October 1998.
15. Per Brinch Hansen. *On Pascal Compilers.* Prentice-Hall, Upper Saddle River, NJ 07458, USA, 1985.
16. P. M. Hill and J. Gallagher. *Meta-Programming in Logic Programming*, volume 5, pages 421–498. Oxford University Press, January 1998.
17. C. A. R. Hoare. An Axiomatic Basis of Computer Programming. *Communications of the ACM*, 12:576–580, 1969.

18. S. C. Johnson. Yacc: Yet Another Compiler-Compiler. Technical report, Bell Laboratories, Murray Hill, NJ, 1978.
19. Donald Knuth. Semantics for Context-Free Languages. *Mathematical Systems Theory*, 2:127–145, 1968.
20. John McCarthy, Paul W. Abrahams, Daniel J. Edwards, Timothy P. Hart, and Michael I. Levin. *LISP 1.5 Programmer's Manual.* MIT Press, Cambridge, Mass., 1962.
21. R. E. Milne and C. Strachey. *A Theory of Programming Language Semantics (part a, part b).* Chapman & Hall, London, 1976.
22. Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised).* The MIT Press, 1997.
23. Peter Mosses. SIS — Semantics Implementation System, Reference Manual and User Guide. DAIMI Report MD-30, DAIMI, University of Århus, Denmark, 1979.
24. J. Paakki. Prolog in Practical Compiler Writing. *The Computer Journal*, 34(1):64–72, February 1991.
25. F. C. N. Pereira and D. H. D. Warren. Definite clause grammars for language analysis — a survey of the formalism and a comparison with augmented transition networks. *Artificial Intelligence*, 13:231–278, 1980.
26. M. Pettersson. RML — A New Language and Implementation for Natural Semantics. *Lecture Notes in Computer Science*, 844:117–131, 1994.
27. G. D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Computer Science Department, Aarhus University, Aarhus, Denmark, September 1981.
28. Stuart M. Shieber, Yves Schabes, and Fernando C. N. Pereira. Principles and implementation of deductive parsing. *Journal of Logic Programming*, 24(1–2):3–36, 1995.
29. Kenneth Slonneger and Barry L. Kurtz. *Formal Syntax and Semantics of Programming Languages.* Addison-Wesley, New York, NY, 1995.
30. Joseph E. Stoy. *The Scott-Strachey Approach to Programming Language Theory.* MIT Press, Cambridge, Massachusetts, 1977.
31. Swedish Institute of Computer Science. SICStus Prolog user's manual, Version 3.10. Most recent version available at `http://www.sics.se/isl`, 2003.
32. A. S. Tanenbaum. *Structured Computer Organization.* Prentice Hall, Englewood Cliffs, 4 edition, 1999.
33. A. S. Tanenbaum, H. van Staveren, E. G. Keizern, and J.W. Stevenson. A Practical Tool Kit for Making Portable Compilers. *Communications of the ACM*, 26:654–660, 1983.
34. Allen Tucker and Robert Noonan. *Programming Languages, Principles and Paradigms.* McGraw-Hill, 2002.
35. D. H. D. Warren. Implementing Prolog - Compiling Predicate Logic Programs – Volumes 1 & 2. D.A.I. Research Report 39, 40, University of Edinburgh, May 1977.
36. David H. D. Warren. Logic Programming and Compiler Writing. *Software—Practice and Experience*, 10(2):97–125, February 1980.
37. David A. Watt and Ole Lehrmann Madsen. Extended Attribute Grammars. *The Computer Journal*, 26(2):142–153, May 1983.
38. Glynn Winskel. *The Formal Semantics of Programming Languages: An Introduction.* MIT Press, Cambridge, Massachusetts, 1993.
39. A. Zaring. CCL: A subject language for compiler projects. *Computer Science Education*, 11(2):135–163, 2001.