Using Prolog for a Compilers Course

Salvador Abreu[1,*,†]

[1]University of Évora and NOVA LINCS, Évora, Portugal

Abstract

In this paper we describe our experience in using Prolog in compiler construction courses, in the context of an undergraduate degree in Computer Science. We show some of the most emblematic usages of Prolog or other Logic Programming languages, to solve problems in compiler design and implementation. Established language extensions such as constraints or more experimental ones, such as contextual logic programming are also drawn upon to tackle specific aspects of the design of a compiler. We also briefly discuss how this approach fares with students.

Keywords

Computer Science Education, Logic Programming, Compiler Writing

1. Introduction

In the early 1980s, the computing division of LNEC, the National Laboratory for Civil Engineering in Lisbon, hosted a DECsystem-10 mainframe computer.

At the time, there was a project at LNEC whose goal was to build FORVER, a tool which could read in entire FORTRAN programs and analyze them, looking for and reporting on questionable or dangerous programming practices, a common trait with the local user community. In order to build FORTRAN, some additional tools were built to help with parsing. Material from Aho and Ullmann's original *Dragon book* [1] was the main influence in the design and development of a lexical analyzer and a parser generator, which were loosely inspired in `lex` and `yacc`. One major difference, though, was that the implementation language was to be Pascal.

In the directory where C. Hedrick's "system" Pascal compiler sources should have been, the PAS: pseudo-device, one came across an unexpected and very intriguing artifact: instead of Pascal programs, there was a compiler, some documentation and loads of examples for something called *Prolog*, a programming language. Upon further investigation, I ended up reading texts about Prolog, including the ASCII-formatted text of the seminal *How to solve it with Prolog* [2], a manuscript that was later extended and formally published in [3]. In the process, I came across an article called *Logic Programming and Compiler Writing*, by David H. D. Warren [4], which discussed techniques for writing a Prolog compiler.

Using Prolog or variations thereof in an higher-education context poses the question of *how* it is introduced, especially to those who already have a computer science background. Besides the pedagogical approach, the importance of the actual tools being used cannot be overstressed: lightweight and agile compilers or interpreters, together with their runtime support is crucial for the effective adoption of the programming paradigm. Recent Prolog systems incorporate playground-like sample programming environments, which may be online as in Swish [5] or the CIAO Prolog playground [6], and these make the Prolog programming environments very accessible.

The *ACM/IEEE-CS/AAAI Computer Science Curricula*, presently at its 2023 edition [7], has long shaped the way computing topics are structured and presented in higher education. Over the years, Logic Programming was introduced, then demoted and more recently slightly rehabilitated, although

PEG26 - Prolog Education Workshop 2026, July 19, 2026, Lisbon, Portugal

*Corresponding author.

spa@uevora.pt (S. Abreu)

0000-0002-1613-4631 (S. Abreu)

© 2026 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

CEUR Workshop Proceedings ceur-ws.org ISSN 1613-0073

published 2026-11-03

CEUR-WS.org/Vol-4311/PEG_regular_08.pdf

in a minor role when compared with, say, Functional Programming. Institutions that carry Logic Programming throughout their curricula have become scarce, yet some remain.

This article briefly discusses the use of Logic Programming techniques and tools in the context of a Computer Science curriculum, as taught at the University of Évora, with a particular focus on its application in an initial Compilers undergraduate course. The remainder of the paper starts by contextualizing the teaching of compilers in section 2. In section 3 we discuss how Prolog fits into this scheme. Section 4 briefly reflects on how students responded to using Logic Programming tools for building a compiler. In section 5 we discuss the experience and point at ideas for future improvement.

2. Compiler courses in a CS undergrad curriculum

Several compilers courses begin with parsing programming languages, introducing tools which perform this aspect, with various degrees of automation and integration. Most of these tools are fairly evolved, some of them are at least partly embedded in interactive development environments, including effective graphical user intefaces components.

The compilers course at the University of Évora has evolved along the years, retaining its integrating character. In its early configurations, the course was capping a sequence of courses on programming languages covering multiple paradigms, computer architecture, algorithms and data structures, formal languages and automata, and finally one on programming language runtime support and memory management. While this largely remains true, the current instance is now a simpler, elective course, drawing a smaller set of students that when it was required. It is also structured to fit in the standard European Bologna 3-year undergrad framework, thereby undergoing some simplification.

3. Integrating Prolog into a Compiler Workflow

From the start, our approach was to take some of the better established compiler construction tools and devise ways in which a Prolog component could be interfaced, made effective and convenient. One explicit goal was to steer the students away from having to design and implement any non-trivial data structures in C.

3.1. Lexical and Syntax Analysis

Having determined that the student compiler writer would use `flex` and `Bison` [8] as their base tools, one only needs to provide a (hopefully simple) means whereby the information from the tokenizer and parser may be passed on to the next stage, which would be implemented in Prolog.

The purpose of such an interface is to lead the Prolog end of the compiler to construct an abstract parse tree (APT) from messages sent from the parsing stage. We settled on a stack-based approach, in which the C part emits stack instructions to be parsed by a Prolog program, which incrementally builds the APT.

The way we designed this is to interpret the text being printed from the semantic actions on the `bison` parser side as a sequence of stack operations, which are to be performed on the Prolog side. Each action is understood to reflect the current and next states of the stack, so that a sequence of such operations ends up leaving the stack in a state which contains the entire APT, expressed as a Prolog term.

The stack operation instructions are expressed as a term of the form `BEFORE - AFTER.`, where `BEFORE` is unified with the state of the stack before the action is carried out, and `AFTER` becomes the stack as it should be, after the action has been carried out. A sample set of instructions is shown in table 1, it should be noted that this is not exhaustive as the actual patterns depend on the language being parsed as well as the encoding chosen for the APT.

Consider, for example, a simple calculator, for which an excerpt of the `Bison` grammar is shown in figure 1. This grammar does not produce or otherwise handle attributes: it relies solely on the

Pattern	Description
`_ - [].`	Reset the stack.
`S - [X|S].`	Push `X` onto the stack.
`[_|S] - S.`	Pops the top element from the stack.
`[A,B|S] - [sub(B,A)|S].`	Pop `A`, `B` and push `sub(B, A)`.

Table 1
Sample Stack Operations

(bottom-up) clause matching which leads to reductions which trigger the execution of the semantic action – which always consists of a simple print-like statement. Convenience macros are available, which include a simple *push* operation with provision for the data format and a *push-after-pop* where two patterns are given: one for the top of the stack before the operation, and another for what goes in after. Figure 1 illustrates the two use-cases. Observe the order in which the `A` and `B` (Prolog) variables occur, which denote the stack order. It should be noted that this mechanism can accomodate all sorts of data structures, not necessarily only lists interpreted as stacks.

```
1    expr : expr '+' expr   { APT ("B, A", "A+B"); }
2         | expr '-' expr   { APT ("B, A", "A-B"); }
3         | expr '*' expr   { APT ("B, A", "A*B"); }
4         | expr '/' expr   { APT ("B, A", "A/B"); }
5         | '(' expr ')'    {}
6         | NUM             { PUSH ("%d", $1); }
7         ;
8
```

Figure 1: Sample `Bison` grammar

On the Prolog end, a simple program to read in the APT does the job, as illustrated in figure 2, where predicate `make_apt/1` is invoked with a single unbound argument and then succeeds with the entire abstract parse tree for a value.

```
1    make_apt(T) :- make_apt([] - [], [], T).
2
3    make_apt(end_of_file, T, T).
4    make_apt(I - T, I, F) :- read(X), make_apt(X, T, F).
5
```

Figure 2: Prolog-side APT loader

3.2. Dictionaries

An essential data structure in a compiler is the symbol table, or dictionary. This needs to flexible in order to encompass the different use-cases which occur in the context of parsing and interpreting a programming language.

There are several ways in which dictionaries may be implemented in Prolog, we opted for a *single incomplete term*, i.e. a deep term with unbound variables as places for future growth. Dictionaries are also intended to behave logically, meaning that they are backtrackable.

Moreover, the dictionary is being put forward as a kind of "universal" data structure, in which we make use of the key/value association to represent values which would have otherwise been represented

as a regular Prolog compound term. To prevent this generalized use to collide with the primary intended purpose of collecting information about names, the latter are stored with a key of the form `id(NAME)`.

Dictionaries may be nested. One way of achieving this is by including a spectial key, up, whose value is the enclosing dictionary. Search operations using such dictionaries are performed iteratively, until the key has been found or there is no further enclosing dictionary.

3.3. Semantic Analysis

Once the APT has been loaded into Prolog, as a single term, a compiler needs to inspect and extract relevant information from it. Such is the role of the two phases *name* analysis and *type* analysis.

Name Analysis is a process in which we are concerned with finding out what names stand for: essentially we want to replace occurrences of a name by whatever it refers to, this is the *dereferencing* process. This is naturally performed during a traversal of the APT, with an appropriate dictionary in tow.

Type Analysis is another traversal of the APT, but one in which we are concerned with validating the occurrences of bits of program in context. This action can be performed by resorting to unification to ensure type correctness. Ill-typed programs will fail type analysis; this situation may be handled with different levels of sophistication: the simplest will just fail, with no indication of what happened.

3.4. Code Generation

Code generation is one place where Prolog really shines, as we can use Definite Clause Grammars (DCGs) "in reverse", by providing a segment of the internal structure of the program, for instance a rewritten APT, as extra arguments to drive the grammar, in order to produce a string as output. This sort of technique stems from DCGs being a simple syntactic sugar for Prolog, in which programs inherit the underlying language features, most notably its handling of nondeterminism and its treatment of data.

For example, considering code generation for arithmetic expressions on a simple stack machine, as seen in figure 3. Assume all operations implicitly pop their arguments off the stack and push the result. Constants are directly pushed, while variable names are understood to denote the corresponding memory address and must undergo an indirection in order to fetch the value.

```
1    expr(A+B) --> expr(A), expr(B), [ add ].
2    expr(A-B) --> expr(A), expr(B), [ sub ].
3    expr(A*B) --> expr(A), expr(B), [ mul ].
4    expr(A/B) --> expr(A), expr(B), [ div ].
5    expr(X) --> { constant(X) }, [ push(X) ].
6    expr(X) --> { variable(X) }, [ push(X), load ].
7
```

Figure 3: DCG for arithmetic expression code generation

The generation of labels, the linking of code blocks and other aspects are easily handled with a dictionary, as previously discussed in section 3.2.

3.5. Incorporating in-Development Language Features

Using Prolog in a compilers course provided a captive audience, with which further experiments could be carried out. One such was the use of Contextual Logic Programming (CxLP) [9], as implemented

inside the GNU Prolog compiler [10]. Using CxLP allowed us to create significantly simpler and more reusable code, when compared with regular Prolog.

Consider for instance the simple dictionary `lookup/2` operation and a straightforward implementation thereof in figure 4. Recall that CxLP allows programs to share *unit-wide variables*, i.e. variables whose scope is the entire unit, not just the clause they occur in. In this particular case, we are using the context to carry the (incomplete) term which represents the dictionary data structure. In order to

```
:- unit dic(ST).

lookup(K, V) :- lookup(ST, K, V).

lookup(D, _, _) :- var(D), !, fail.
lookup([K=V | _], K, V).
lookup([_ | D], K, V) :- lookup(D, K, V).

```

Figure 4: Dictionary lookup using CxLP

provide the functionality required of the hierarchical dictionary, a single clause may be *added* to the `lookup/2` predicate, as shown in figure 5. This assumes that the dictionary hierarchy is reflected in the context structure, in the form of a stack of dictionaries. Let the context be of the form `CX=[U1, U2, ··· Un]`, where `Ui` is the topmost unit of the form `dic(ST)` and `Uj` is the next unit of the same form. If we evaluate the goal `CX :< lookup(K, V)`, the base goal (`lookup(K, V)`) will be tried in units `U1`, `U2` ···. The first unit whose `ST` argument contains an entry which unifies with key `K` will match, thereby providing a proper nested dictionary mechanism.

```
lookup(K, V) :- :^ lookup(K, V).

```

Figure 5: Hierarchical dictionary with Contexts

4. Student Response

As students we already exposed to Prolog in earlier courses, they needed no introduction to Prolog. They did, however, need to adapt to tools which are not generally available and therefore had to be steered towards a specific implementation of Prolog.

The students were given the choice as to the language which they would use to implement the compiler project in, and most of them opted to use Prolog.

5. Concluding Remarks

Using Prolog to write compilers has a long history, to which we gave continuity. The reception by students has been generally good, as several aspects of a compiler are naturally implemented as Logic Programs.

One aspect that we retain is that Prolog and its derivatives are easiest to use as rapid prototyping tools. When one needs to work out the actions necessary in a production environment, the advantage over other approaches is not as pronounced.

Acknowledgments

This work is supported in part by UID/04516/NOVA Laboratory for Computer Science and Informatics (NOVA LINCS) with the financial support of FCT.IP.

Declaration on Generative AI

The author has not employed any Generative AI tools.

References

[1] A. V. Aho, J. D. Ullman, Principles of Compiler Design (Addison-Wesley series in computer science and information processing), Addison-Wesley Longman Publishing Co., Inc., 1977.
[2] H. Coelho, J. C. Cotta, L. M. Pereira, How to Solve it with Prolog, (No Title) (1982).
[3] H. Coelho, J. C. Cotta, Prolog by example: how to learn, teach and use it, Springer Science & Business Media, 2012.
[4] D. H. D. Warren, Logic programming and compiler writing, Softw. Pract. Exp. 10 (1980) 97–125. URL: https://doi.org/10.1002/spe.4380100203. doi:10.1002/SPE.4380100203.
[5] J. Wielemaker, T. Lager, F. Riguzzi, SWISH: swi-prolog for sharing, CoRR abs/1511.00915 (2015). URL: http://arxiv.org/abs/1511.00915. arXiv:1511.00915.
[6] J. F. Morales, S. Abreu, D. Ferreiro, M. V. Hermenegildo, Teaching prolog with active logic documents, in: D. S. Warren, V. Dahl, T. Eiter, M. V. Hermenegildo, R. A. Kowalski, F. Rossi (Eds.), Prolog: The Next 50 Years, Lecture Notes in Computer Science, Springer, 2023, pp. 171–183. URL: https://doi.org/10.1007/978-3-031-35254-6_14. doi:10.1007/978-3-031-35254-6_14.
[7] A. N. Kumar, R. K. Raj, S. G. Aly, M. D. Anderson, B. A. Becker, R. L. Blumenthal, E. Eaton, S. L. Epstein, M. Goldweber, P. Jalote, D. Lea, M. Oudshoorn, M. Pias, S. Reiser, C. Servin, R. Simha, T. Winters, Q. Xiang, Computer Science Curricula 2023 (CS2023), Association for Computing Machinery, New York, NY, USA, 2024. URL: https://doi.org/10.1145/3664191. doi:10.1145/3664191, ACM/IEEE-CS/AAAI Joint Task Force on Computing Curricula.
[8] J. Levine, Flex & Bison: Text Processing Tools, " O'Reilly Media, Inc.", 2009.
[9] S. Abreu, D. Diaz, Objective: In minimum context, in: C. Palamidessi (Ed.), Logic Programming, 19th International Conference, ICLP 2003, Mumbai, India, December 9-13, 2003, Proceedings, Lecture Notes in Computer Science, Springer, 2003, pp. 128–147. URL: https://doi.org/10.1007/978-3-540-24599-5_10. doi:10.1007/978-3-540-24599-5_10.
[10] D. Diaz, S. Abreu, P. Codognet, On the implementation of GNU prolog, CoRR abs/1012.2496 (2010). URL: http://arxiv.org/abs/1012.2496. arXiv:1012.2496.