A Prolog Toolkit for Formal Languages and Automata

Michel Wermelinger
Computing Department
The Open University
Walton Hall, Milton Keynes MK7 6AA, UK
http://mcs.open.ac.uk/mw4687

Artur Miguel Dias
Departamento de Informática
Universidade Nova de Lisboa
2829-516 Caparica, Portugal
amd@di.fct.unl.pt

ABSTRACT

This paper describes the first version of P$^\flat$ (read "P flat"), a collection of Prolog predicates that aims to provide a pedagogical implementation of concepts and algorithms taught in Formal Languages and Automata Theory (FLAT) courses. By "pedagogical implementation" we mean on the one hand that students should be able to easily map the implementation to the mathematical definitions given in lectures, and on the other hand that the toolkit should provide a library for students to implement further concepts and algorithms. In both cases the goal is to make students more confident in defining and manipulating the various kinds of languages and automata at a level beyond the one provided by visual simulators of automata. As such, P$^\flat$ is not intended to replace but rather complement existing graphical tools. We believe the declarative, non-deterministic, and interactive nature of Prolog helps in building an executable specification of FLAT concepts and definitions that can be actively extended and explored by students, in order to achieve the stated goal.

Categories and Subject Descriptors

F.1.1 [**Computation by Abstract Devices**]: Models of Computation—*Automata*; F.4.3 [**Mathematical Logic and Formal Languages**]: Formal Languages—*Classes defined by grammars or automata, Operations on languages*; K.3.1 [**Computers and Education**]: Computer Uses in Education—*Computer-assisted instruction*; K.3.2 [**Computers and Education**]: Computer and Information Science Education—*Computer science education*

General Terms

Algorithms, Languages, Theory

1. INTRODUCTION

Formal languages and automata theory (FLAT) lies at the very core of Computer Science. It provides the foundations for defining and processing all sorts of languages (programming languages, command languages like bash, query languages like SQL, data description languages like XML, etc.), for text searches based on regular expressions, for modelling communication protocols, and for model-checking such specifications, to name just a few widely used applications. It is therefore essential for students to have a good understanding of the concepts and algorithms provided by FLAT. However, the mathematical nature of the subject often constitutes a hurdle to students.

Several simulators for different kinds of automata (like finite automata, push-down automata and Turing machines) have been developed and described in the literature (see [1] for a survey). Although their exact functionality and available features differ, they all aim at providing a visual environment in which students can define automata and watch their step-by-step execution for a given input. These tools greatly help students to understand how the automata work, and aid them in developing and debugging automata for accepting a given language. However, all these tools are mainly "black boxes" that provide a fixed set of automata and operations (like minimization) upon them. The student is not expected to study or extend the source code.

We believe that a further (and possibly deeper) understanding of FLAT could be obtained if the tool to be provided adhered to the following pedagogical principles:

Adaptability Due to the minor variations among textbooks, instructors should be able to easily adapt the tool to the notation and formal definitions they have adopted for their classes.

Extensibility To reinforce learning by doing, students should be able to extend the tool by implementing further parts of FLAT.

Adequacy The implementation of the various FLAT concepts and operations should follow their formal definitions as closely as possible.

Flexibility The tool should allow for an unconstrained exploration of the various FLAT concepts.

The adaptability and extensibility principles entail that the tool's source code should be available. The flexibility principle can be best implemented by providing a library of FLAT concepts, their definitions and operations, instead of a "closed" application with a fixed user interface, because a library allows students to combine operations in unforeseen ways to solve exercises or implement new functionality.

ITiCSE'05, June 27–29, 2005, Monte de Caparica, Portugal.

Finally, we believe the adequacy principle, which supports adaptability and extensibility, is best achieved through an executable specification. Being a specification, it would be close enough to the abstract mathematical and formal concepts so that it could be studied by students. Being executable means that students would also have the ability to execute the automata, helping them solve FLAT exercises. An executable specification could therefore help students make the bridge between the abstract mathematical and formal concepts and their realization into the definition and automatic processing of various kinds of languages.

We have started to develop such an executable specification, choosing Prolog as the implementation language. Crookes [2] already argued for the use of Prolog to implement automata for educational purposes:

- The specification can be directly executed, acting as a simulator.
- The tracing facilities of Prolog can be used to single-step the execution without any additional programming effort.
- Prolog predicates only express logical relationships between their arguments, often allowing for a "reversible execution" that can be used for multi-purpose simulators, e.g., to find the starting state that would lead to a given end state.
- The conciseness of the implementation in Prolog (when compared to a procedural or object-oriented version) allows one to present the complete code to students, rather than just an outline.

Note that the first and last arguments are related to the adequacy principle, and the third one is related to the flexibility principle. We add the following advantages of using Prolog for our purposes:

- Non-deterministic automata can be implemented in a straightforward way, further shortening the gap between the mathematical definitions and their implementation.
- The interactive nature of Prolog interpreters allows users to quickly execute operations or query for certain relationships and obtain immediate feedback, thus supporting the active exploration of FLAT concepts by students.
- The ability to define new operators provides support for syntactic constructs that are close to the notations used in FLAT.
- The ability to construct and unify complex term structures supports a recursive style of programming that is close to the structural induction style of definitions and proofs used in FLAT.

Note that the third argument supports the flexibility principle, while the others support the adequacy principle.

We have started to develop P$^\flat$ (read: "P flat"), the Prolog Formal Languages and Automata Toolkit, a library of predicates to define and manipulate various kinds of languages and automata. The current version handles regular languages and push-down automata, but the latter are not described in the paper due to space constraints. P$^\flat$ is available from the first author's webpage. The distribution includes the Prolog source, a tutorial, some examples taken from [4], and the exercises suggested in this paper. P$^\flat$ is implemented in SWI-Prolog (www.swi-prolog.org), a freely available implementation of Prolog with pre-built binaries for Linux, Windows and Macintosh. However, the toolkit should be portable with little or no modification to other Prolog interpreters, as care has been taken to avoid predicates that are specific to SWI-Prolog.

It should be noted that there is a very sophisticated Prolog implementation of finite state automata [6], including their visual rendering in a variety of formats. However, it was built for research in natural language processing, and its size and complexity make it hardly usable for education purposes. Moreover, our ultimate goal is to cover most of the topics in typical FLAT courses, not just finite automata. On the other end of the scale, very simple implementations of some kinds of automata are readily available in Prolog textbooks and the computer science education literature [2, 3, 5], but they are mostly intended for illustration or miss some important characteristics for education purposes (like checking that the automaton is well-defined).

We also would like to point out that P$^\flat$ is not intended to compete with or replace existing visual automata simulators, but rather to complement them. In fact, it has been argued that using "*multiple* simulators with *contrasting* views for the *same* automaton can provide profound insights into FLAT concepts"[1] [1]. We believe that adding P$^\flat$ to the toolchest of FLAT educators can provide additional insights into the concepts and algorithms used to define and operate on formal languages and automata. One possible effective usage of such tools is to introduce automata informally via simulators and then present the mathematical concepts and formal definitions with the help of P$^\flat$.

The following sections provide an introduction to P$^\flat$, presenting its main predicates and their rationale, and suggest some exercises. The running example is the definition of the language of binary words with an even number of 1's. The file given in the appendix is part of P$^\flat$'s distribution and will be referred throughout the text. We assume the reader is moderately familiar with Prolog and FLAT.

2. SYMBOLS AND ALPHABETS

Making a chemical analogy, symbols are the atomic constituents of languages. As such, in P$^\flat$ symbols are represented by atomic terms (atoms, strings and numbers). An alphabet is a finite set of symbols. Usually, sets are represented in Prolog by lists. However, in the spirit of shortening the gap between the formal concepts and their implementation, P$^\flat$ adopts the usual curly braces. This also facilitates the distinction between alphabets and words (see the next section), contributing to the overall clarity of the executable specification.

An alphabet is defined by a fact `alphabet/2`, as illustrated in line 1 of the appendix. The first argument must be a unique Prolog ground term and the second argument is a set expression (rather convoluted in the example). Predicate `alphabet_symbols/2` computes the set of symbols corresponding to an arbitrary set expression. In particular, this can be used to check if the alphabets were defined as intended. For our example, line 2 outputs `{0,1}`.

[1]Words already emphasized in the original.

Students can also define total orders over alphabets' symbols, by asserting facts of the form `alphabet_order(Name, Alphabet, List)` stating that order *Name* defines the symbols denoted by expression *Alphabet* to be ordered as given by *List*. Lines 4 and 5 of the appendix provide examples. Notice that different orders (with distincts names) can be defined for the same alphabet, the aim being to allow students explore the impact of different alphabetic orders on the ordering of words (shown in the next section).

The toolkit provides predicate `check_declaration/1` to check for mistakes in declarations, writing the error messages to the screen. A quick way to check all declarations is to leave the argument uninstantiated as done in line 26. In general, the predicate will check that the declared name is a unique ground term. Additionally, for alphabet declarations, P$^\flat$ will check that the expression in the second argument of `alphabet/2` is indeed a set expression and it warns the user if the expression denotes the empty set, which is not a very useful alphabet. As for alphabet order declarations, `check_declaration/1` verifies that the second argument is an alphabetic expression and that all its symbols appear in the given list. For the example in the appendix, line 6 causes message `Error in wrong: [1, 2] is not a permutation of all symbols`.

3. WORDS

A word is a sequence of symbols, and is therefore very naturally represented in Prolog by a list, the empty list denoting the empty word. In FLAT, a symbol is also considered to be a word (of length 1), and therefore P$^\flat$ allows s to be a shorthand for `[`s`]`, if s is a symbol.

P$^\flat$ provides two operators on words: *W1* `*` *W2* for the product (concatenation) of two words and *W*`^`*N* for the N-th power of word W (i.e., its N-fold concatenation). If N is -1 then W is reversed. Predicate `compute_word/2` calculates the value (i.e., word) of such an expression. For example, lines 8-9 output `[0,1,0,1,0,1,1]`. Notice that the operators are Prolog's pre-defined multiplication and exponentiation operators for integers; as such, their associativity and precedence allows us to process word expressions without any additional programming effort to parse them:

```
compute_word(_ ^ 0, []).
compute_word(E ^ N, Wn) :- N > 0, M is N-1,
  compute_word(E, W), compute_word(W * W^M, Wn).
```

These two clauses follow directly the recursive mathematical definition ($w^0 = \epsilon, w^{n+1} = w \cdot w^n$, where ϵ is the empty word), making the implementation easy to understand.

The ability to interact with the Prolog interpreter allows the use of `compute_word` and `alphabet_symbols` as "calculators" for word and alphabet expressions with immediate feedback, helping students to understand the involved concepts (alphabets, words, intersection, concatenation, etc.) and gain confidence with the corresponding notations. Instructors can change P$^\flat$ to adapt it to the notational conventions used in their courses. Fundamental changes (like not using lists for words) will obviously require modifications throughout the toolkit, but small changes (like using a term of the form `rev(W)` instead of *W*`^(-1)`) can be easily accomodated. Another possible change is to require the second argument of `compute_word/2` to be instantiated: this means that the predicate could only be used to check if the student's answer is correct, not to compute the answer on the student's behalf.

Programming exercises involving the implementation of new or the extension of existing predicates are also a good way to foster the understanding of the mathematical concepts and their formal definitions. For example, instructors could ask students to extend the power operator to arbitrary negative integer exponents, with the intuitive meaning $w^{-n} = (w^n)^{-1}$, or to implement a predicate to check whether two word expressions are equivalent.

P$^\flat$ includes a predicate `word_alphabet/2` that, like many others in the toolkit, can be used in two ways: on the one hand, it can check whether a given word is over a given alphabet (i.e., is formed from symbols of that alphabet); on the other hand, if the word argument is a variable, the predicate can systematically generate through backtracking all words over the given alphabet, in increasing length. For example, `word_alphabet(W, bits)` succeeds successively with `W=[]`, `W=[0]`, `W=[1]`, `W=[0,0]`, etc.

Given a total order on symbols of an alphabet, predicate `lexically_ordered/3` can check whether two given words are ordered according to the lexical order. The lexical order is the generalization of the usual dictionary order to arbitrary alphabets and symbol orderings. The predicate can also be used in a generative mode. For example,

```
?- lexically_ordered(W, [1,0], O).
W = [], O = _ ;
W = [1], O = _ ;
W = [1,0], O = _ ;
W = [1,1|_], O = down;
W = [0|_], O = up;
No
```

The first three answers reflect the fact that $w \leq w'$ if w is a prefix of w', whatever the total order of the symbols. The fourth answer says that any word starting with 11 comes before 10 if bits are ordered in descending way, and the last answer says that when the order of bits is $0 < 1$ then any word beginning with 0 comes before 10.

Finally, P$^\flat$ provides predicates to deal with the concepts of prefix, suffix and subword. Those predicates can be used also in checking and generative modes. For example, `prefix(W, [0,1,1])` will successively generate all prefixes of 011.

We believe that the ability, due to unification, for several predicates to check given answers or generate answers, not only is helpful for students but can also be useful for instructors to have more dynamic lectures, in which the exposition of a topic is mingled with the execution of various examples, seeking answers from students before showing P$^\flat$'s results. In particular, students can be stimulated in lectures or through exercises to explore concrete examples that will lead them to answer generic questions like: is there a shortest or longest suffix for any given word? Do any two given words always have a common prefix? When is the lexical ordering independent of the symbol order?

4. LANGUAGES

A language is a set of words over the same alphabet. The user fact `language(Name, Alphabet)` declares *Name* to be a language over the given alphabet. User facts of the form `in_language(Word, Name, Boolean)` state if *Word* belongs to language *Name* or not, depending on the *Boolean* value.

Lines 11-13 illustrate these predicates. The (counter-) examples provide a way to test actual definitions for language *Name*. P$^\flat$ supports the Extreme Programming principle of writing tests first, because we believe that starting with (counter-)examples helps thinking about special cases right from the beginning, leading to a better understanding of the informal language description given in the exercise and to language definitions (automata, regular expressions, etc.) that are more likely to be correct.

Predicate `check_declaration/1` can be used to detect various mistakes: the alphabet of the language is not a valid alphabet expression; there is not at least one (counter-)example, in particular the special case of the empty word; the test word is not over the language's alphabet; the third argument of `in_language` is not a boolean; the first argument is not a word. For the example, line 26 will display for lines 12-14:

```
Warning in evenL: no test case for the empty word
Error in evenL: no is not a boolean test result
Error in evenL: 10 is not over given alphabet
```

The suggested next step is for the student to provide one or more direct definitions of the language, by writing unary predicates that succeed only if the given word (over the language's alphabet) belongs to the language. The user fact `predicate(`*`Name, Alphabet`*`)` asserts that predicate *Name/1* takes as argument a word over *Alphabet*. Lines 16-17 provide an example, where `occurs/3` is a P$^\flat$ predicate to check or compute the number of times a symbol occurs in a word.

Calling `test_definition(`*`Definition, Language`*`)` tests *Definition* with the (counter-)examples for *Language*. It will report any positive test words that have been rejected and any negative test words that have been accepted. For the example, the call in line 18 doesn't report any errors. The student can try out further words, by calling `accept(`*`Definition, Word`*`)`.

It is possible to write expressions over languages. The basic operands are literal sets of words, names of alphabets (an alphabet is a language because a symbol is a word of length one) and of language definitions. Operators include the set operators (because languages are sets), the Kleene star (`^*`), the positive closure (`^+`), the product and power operators. P$^\flat$ enforces a clear distinction between a definition (like a predicate, regular expression or automaton) and the language accepted by such a definition. Therefore, in language expressions one must write `lang(D)` and not simply `D` to represent the language accepted by `D`.

Predicate `word_language/2` can be used to check if a given word belongs to a language given by an expression, or to generate all such words. In the latter case, the predicate calls `language_alphabet/2` to compute the alphabet expression corresponding to the given language expression, and then generates and tests all words over the obtained alphabet using `word_alphabet/2`. This means that `word_language/2` will enter an infinite loop even for finite languages, as it doesn't know when to stop generating further words. For example, `word_language(W, -lang(evenP)/\bits^2)` will generate all binary words of length two not accepted by `evenP/1`. This is a finite language, but after succeeding with `W=[1,0]` and `W=[0,1]`, the predicate will enter an infinite loop if a further answer is asked for.

5. REGULAR EXPRESSIONS AND FINITE AUTOMATA

Besides writing language definitions directly in Prolog, one can define a regular language by a regular expression or a finite automaton.

P$^\flat$ allows the usual notation of regular expressions built from symbols, the empty word, the empty set, and the union, concatenation and closure operators. Regular expressions can be given a name (lines 20-21), which can then be used in further expressions or as an argument to predicates (line 22). As usual, `check_declaration/1` can be called to detect mistakes in the declaration. Regular expressions can be tested with `accept/2` and `test_definition/2`, too, reducing the number of predicate names the student has to memorise. In our example, line 22 will display "`Error in evenRE1: [1, 0, 1, 0, 1, 1] should be accepted`" because `evenRE1` requires the second 1 of a pair to be followed by the first 1 of the next pair.

Predicate `re_simplify/2` simplifies a regular expression according to simple equations like $E+E = E$, $E\cdot E^*+\epsilon = E^*$ or $E \cdot \emptyset = \emptyset$. These can be straightforwardly translated to clauses, using Prolog's term unification:

```
re_simplify(E + E, F) :- re_simplify(E, F).
re_simplify(E*E^* + [], F) :- re_simplify(E^*,F).
re_simplify(_ * {}, {}).
```

A possible exercise is to add further simplifications, e.g., $E + s = E$ if symbol s is accepted by expression E. One might also ask students to extend the notation in order to allow for words within regular expressions, e.g., to write `[a,b,c]^*` instead of `(a * b * c)^*`.

A finite automaton is declared by a fact `fa(`*`Name, I, Ts, Fs`*`)`, where I is the name (a Prolog ground term) of the initial state, *Fs* is the set of final state names, and *Ts* is the set of transitions. A transition from state *S1* to state *S2* via symbol S is given by a term of the form *`S1/S/S2`*. In non-deterministic automata it is possible for S to be the empty word `[]` and to have multiple transitions for the same state/symbol pair. We do not require transitions for all pairs of state and symbol, i.e., the "error state" and its transitions can be left out. Notice that the states and alphabet of an automata are not explicitly given, to speed up the writing of automata definitions, but they can be computed by auxiliary P$^\flat$ predicates.

Lines 24 and 25 illustrate an automaton definition. The call in line 26 will check if there is no transition for the initial state, not all states are reachable, there are no final states, or there is no transition for a given state/symbol pair. In this case, the messages are:

```
Warning in evenFA: no final states
Warning in evenFA: undefined transition for old/0
Warning in evenFA: undefined transition for odd/1
Error in evenFA: unreachable states {old}
```

As for alphabets, languages, and regular expressions, it is possible for the user to build expressions over finite automata. The allowed operators over automata are union, complement, intersection, closure, minimisation and determinisation (which is implicitly called by the minimisation and complement operators). This helps students to quickly build complex automata from simpler ones. Predicate `re_fa/2` translates a regular expression to an expression (with the

same structure) over automata, replacing each symbol of the regular expression by an automaton that just accepts that symbol. The predicate `compute_fa/2` evaluates an automata expression and returns the term representing the resulting automaton. This predicate generates new states whenever necessary, their names being `sX` where X is a unique integer obtained by calling Prolog's `gensym/2` predicate. The computations of the deterministic and minimal automata don't introduce any new names. Instead, the names of the resulting states are sets of names of the original states, thus showing clearly which states of the original non-deterministic or non-reduced automaton have been merged. If a more succint representation is sought, one can use `rename_fa/2` to rename all states of an automaton using fresh atomic names.

Predicates `accept/2` and `test_definition/2` can also be used with finite automata. To see which states are gone through for a particular word, predicate `fa_consume(F, S1, W, S2, P)` succeeds if automaton F goes through path P (a list of states) from state *S1* to state *S2* to consume word W. Notice that *S1* doesn't have to be the initial state of F. In fact, only F and W have to be instantiated upon calling the predicate: it will return through backtracking all possible paths that consume the word. To avoid any infinite loops, there should be no cycles in the automaton that don't consume any symbol.

To illustrate the use of the previously mentioned predicates, and to show how a library of operations can support the flexibility and extensibility principles, the appendix shows how to implement a predicate `mpp/3` that computes the maximally processed prefix of a word that failed to be accepted by a regular expression. Calling `mpp(evenRE1, [1,0,1,0,1,1], P)` succeeds with `P=[1,0,1,0]`, showing that the problem is in accepting the second pair of 1's. Predicate `mpp/3` first converts the regular expression to a minimal automaton and obtains its initial state. Through backtracking, it seeks for a prefix P of the test word that can be consumed starting in the initial state (but not necessarily leading to a final state) but such that there is no other prefix $P_2 > P$ that can also be consumed.

6. CONCLUDING REMARKS

This paper introduced P$^\flat$, a library of Prolog predicates to help students understand FLAT concepts in a complementary way to the visual obervation of automata behaviour, as supported by existing graphical automata simulators [1]. P$^\flat$ aims at being an intuitive executable specification of FLAT concepts that is close to the mathematical notation and formal definitions used in FLAT texts, thus helping students to understand the theory and its application to defining and processing languages. Furthermore, being a library of predicates instead of a closed application, P$^\flat$ supports the active exploration and extension by students (or instructors, to adapt P$^\flat$ to the particular notation or automata variants used in their courses). We have chosen Prolog because its declarative, non-deterministic, interactive nature, and the adequacy provided by unification and user-defined operators suit the above mentioned aims, as illustrated in the paper. However, there hardly is a perfect programming language for any given purpose, and Prolog is no exception. For some definitions, a functional approach might lead to even clearer specifications.

In future work we will extend P$^\flat$ with further FLAT concepts (like context-free grammars and Turing machines) and provide predicates that convert P$^\flat$'s to FSA's representation of automata to use its graphical output capabilities [6]. We also plan to test P$^\flat$ on other Prolog platforms, and to report on classroom experience in a future paper.

We thank the anonymous reviewers for their insightful and challenging comments that helped us improve the paper.

7. REFERENCES

[1] C. I. Chesñevar, M. L. Cobo, and W. Yurcik. Using theoretical computer simulators for formal languages and automata theory. *SIGCSE Bulletin*, 35(2):33–37, 2003.

[2] D. Crookes. Using Prolog to present abstract machines. *SIGCSE Bulletin*, 20(3):8–12, 1988.

[3] J. L. Hein. A declarative laboratory approach for discrete structures, logic, and computability. *SIGCSE Bulletin*, 25(3):19–25, 1993.

[4] L. Monteiro. Formal languages and automata. 298 slides, Univ. Nova de Lisboa, 2003. In Portuguese.

[5] L. Sterling and E. Shapiro. *The Art of Prolog*. MIT Press, 2nd edition, 1994.

[6] G. van Noord. FSA utilities: A toolbox to manipulate finite-state automata. In *Automata Implementation*, LNCS 1260. Springer Verlag, 1997.

APPENDIX

```
1  alphabet(bits, {}+{0,1}-{2}).
2  :- alphabet_symbols(bits, S), writeln(S).
3
4  alphabet_order(up, bits, [0,1]).
5  alphabet_order(down, bits, [1,0]).
6  alphabet_order(wrong, bits, [1,2]).
7
8  :- compute_word([0,1]^2*[1,1,0]^(-1), W),
9     writeln(W).
10
11 language(evenL, bits).
12 in_language([1,0,1,0,1,1], evenL, true).
13 in_language([1], evenL, false).
14 in_language(10, evenL, no).
15
16 predicate(evenP, bits).
17 evenP(W) :- occurs(1, W, N), 0 is N mod 2.
18 :- test_definition(evenP, evenL).
19
20 regexp(evenRE1, 0^* * (1 * 0^* * 1)^* * 0^*).
21 regexp(evenRE2, (0^* * 1 * 0^* * 1)^* * 0^*).
22 :- test_definition(evenRE1, evenL).
23
24 fa(evenFA, even, { even/0/even, even/1/odd,
25    old/1/even, odd/0/odd }, {}).
26 :- check_declaration(_).
27
28 mpp(RE, W, P) :- re_fa(RE, FA),
29    compute_fa(min(FA), MFA),
30    fa_initial(MFA, I), prefix(P, W),
31    fa_consume(MFA, I, P, _, _), \+ (
32    prefix(P2, W), prefix(P, P2), P \= P2,
33    fa_consume(MFA, I, P2, _, _) ).
```