7-2019

An Introduction to Declarative Programming in CLIPS and PROLOG

Jack L. Watkin
University of Nebraska - Lincoln

Adam C. Volk
University of Nebraska - Lincoln

Saverio Perugini
University of Dayton, sperugini1@udayton.edu

An Introduction to Declarative Programming in CLIPS and PROLOG

Jack L. Watkin, Adam C. Volk[1], and Saverio Perugini[2]
[1]Department of Mathematics, University of Nebraska Lincoln, Lincoln, Nebraska, USA
[2]Department of Computer Science, University of Dayton, Dayton, Ohio, USA

Abstract— *We provide a brief introduction to* CLIPS*—a declarative/logic programming language for implementing expert systems—and* PROLOG*—a declarative/logic programming language based on first-order, predicate calculus. Unlike imperative languages in which the programmer specifies how to compute a solution to a problem, in a declarative language, the programmer specifies what they what to find, and the system uses a search strategy built into the language. We also briefly discuss applications of* CLIPS *and* PROLOG*.*

Keywords: CLIPS, declarative programming, first-order predicate calculus, logic programming, production system, PROLOG

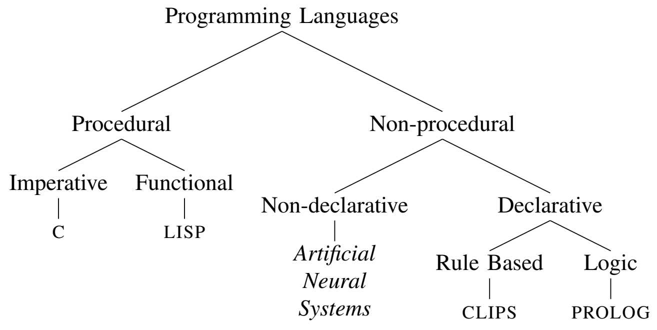


Fig. 1: A hierarchy of programming paradigms and languages (adapted from [6]).

1. Introduction

CLIPS[1] is a declarative/logic programming language for implementing expert systems. Originally called NASA's Artificial Intelligence Language (NAIL), CLIPS started as a tool for creating expert systems at NASA in the 1980s. An *expert system* is a computer program capable of modeling the knowledge of a human expert [1]. CLIPS stands for C Language Integrated Production System. In artificial intelligence, a production system is a computer system which relies on facts and rules to guide its decision making. Programming in a declarative, rule-based language like CLIPS is fundamentally different than programming in more traditional programming languages like C or Java and more resembles programming in PROLOG [2], [3], [4], [5]—a declarative/logic language to which we compare CLIPS throughout this paper.[2] Fig. 1 situates CLIPS (and PROLOG) in relation to other programming paradigms and languages [6].

2. Declarative/Logic Programming

In the declarative paradigm of programming, rather than describing *how* to compute a solution as done when programming in, e.g., Java, the programmer describes *what* a solution to a problem looks like through the declaration of facts and rules describing it. While CLIPS and PROLOG are both declarative programming language and, thus, involve the declaration of facts and rules, they each use fundamentally different search strategies.

2.1 Horn Clauses

Logic programming in PROLOG is based on first-order, predicate calculus—a formal system of logic which uses variables, predicates, quantifiers, and logical connectives to produce propositions involving clauses. *Clausal form* is a standard (and simplified) form for propositions: $B_1 \vee B_2 \vee \cdots \vee B_n \subset A_1 \wedge A_2 \wedge \cdots \wedge A_m$. The As and Bs are *terms*. The left hand side is called the *consequent*, while the right hand side is called the *antecedent*. The interpretation is: if all of the As are true, then at least one of the Bs must be true. Advantages of representing propositions in clausal form are (i) existential quantifiers are unnecessary; (ii) universal quantifiers are implicit in the use of variables in the atomic propositions; (iii) no operators other than conjunction and disjunction are required; and (iv) all predicate calculus propositions can be converted to clausal form. "When propositions are used for resolution,[—a rule of inference discussed below—]only a restricted kind of clausal form called a Horn clause can be used, which further simplifies the resolution process" [7]. Horn clauses conform to one of the three forms detailed in Table 1 [8]. All expressions in PROLOG match one of these three forms for Horn clauses.

[1]`http://www.clipsrules.net/`

[2]PROLOG, which stands for PROgramming in LOGic, is a declarative/-logic programming language developed in the early 1970s for artificial intelligence applications. The PROLOG interpreter assumed in this paper is SWI-PROLOG—`http://www.swi-prolog.org/`.

Cite as: Watkin, J.L., Volk, A.C., & Perugini, S. (2019). An introduction to declarative programming in CLIPS and PROLOG. In Arabnia, H.R., Deligiannidis, L., Grimaila, M.R., Hodson, D.D., & Tinetti, F.G. (Eds.), *Proceedings of the International Conference on Scientific Computing (CSC)*, 105–111. USA: CSREA Press. (Publication of the World Congress in Computer Science, Computer Engineering, and Applied Computing (CSCE). Available at `https://csce.ucmss.com/cr/books/2019/LFS/CSREA2019/CSC2488.pdf`.)

Table 1: Types of Horn clauses.

Form	Horn Clause Type	PROLOG Name
$\{\} \subset B_1 \wedge ... \wedge B_n, n \geq 1$	Headless	Goal
$A \subset \{\}$	Headed	Fact
$A \subset B_1 \wedge ... \wedge B_n, n \geq 1$	Headed	Rule

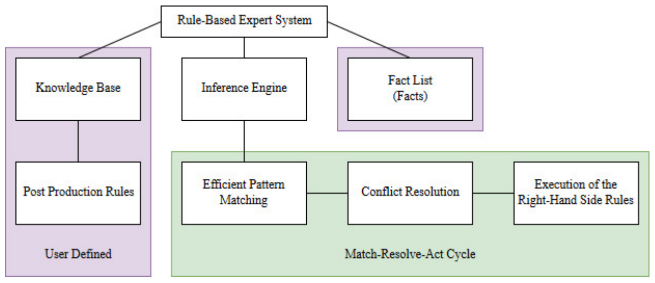


Fig. 2: Architectural design of CLIPS (adapted from [6]).

2.2 Asserting Facts and Rules

Like PROLOG, in CLIPS expert systems, knowledge is represented as *facts* and *rules* and, thus, a CLIPS or PROLOG program consists of facts and rules. A *fact* is an axiom that is asserted as true. A *rule* is a declaration expressed in the form of an `IF THEN` statement. For example, a fact may be 'It is raining.' In CLIPS this fact is written as:

```
(assert (weather raining))
```

The `assert` keyword defines facts, which are inserted in FIFO order into the `fact-list`. Facts can also be added to the `fact-list` with the `deffacts` command. An example rule is 'If it is raining, then I carry an umbrella':

```
(defrule ourrule
   (weather raining)
   =>
   (assert (carry umbrella)))
```

The following is the general syntax of a rule [1][3]:

```
(defrule rule_name
   (pattern_1) ; IF Condition 1
   (pattern_2) ; And Condition 2
   .
   .
   (pattern_N) ; And Condition N
   => ; THEN
   (action_1) ; Perform Action 1
   (action_2) ; And Action 2
   .
   .
   (action_N)) ; And Action N
```

The CLIPS shell can be invoked in UNIX-based systems with the `clips` command. From within the CLIPS shell, the user can `assert` facts, `defrules`, and `(run)` the inference engine. When the user issues the `(run)` command, the inference engine pattern matches facts with rules. If all patterns are matched within the rule, then the actions associated with that rule are fired. To load facts and rules from an external file, use the `-f` option (e.g., `clips -f database.clp`). Table 2 is a summary of commands accessible from within the CLIPS shell and usable in CLIPS scripts.

[3]Note that `;` begins a comment.

Table 2: Essential CLIPS shell commands.

Command	Function
`(run)`	Run the inference engine.
`(facts)`	Retrieve the current fact-list.
`(clear)`	Restores CLIPS to start-up state.
`(retract n)`	Retract fact `n`.
`(retract *)`	Retract all facts.
`(watch facts)`	Observe facts entering or exiting memory.

For purposes of comparison, we can declare the proposition 'It is raining' in PROLOG with the fact:

```
weather(raining).
```

Similarly, we can declare the rule 'If it is raining, then I carry an umbrella' in PROLOG with the rule.

```
carry(umbrella) :- weather(raining).
```

Additionally, consider the following set of facts and rule in PROLOG:

```
color(red).
color(yellow).
color(blue).

lightcolor(X) :- color(yellow).
```

These facts assert that red, yellow, and blue are colors. The rule declares that yellow is a light color.

2.3 Resolution and Unification

The *Match-Resolve-Act* cycle is the foundation of the CLIPS inference engine which performs pattern matching between rules and facts through the use of the *Rete Algorithm*. Once the CLIPS inference engine has matched all applicable rules, conflict resolution occurs. Conflict resolution is the process of scheduling rules that were matched at the same time. Once the actions have been performed, the inference engine returns to the pattern matching stage to search for new rules that may be matched as a result of the previous actions. This process continues until a fixed point is reached.

In PROLOG, however, the user gives the inference engine a *goal* that it then sets out to satisfy (i.e., prove) based on the knowledge base of facts and rules. To run a program, the user supplies one or more goals, each in the form of a headless Horn clause (see Table 1). The activity of supplying a goal can be viewed as asking questions of the program or querying the system as one does with a database system. When a goal is given, the inference engine attempts to match the goal with the head of a headed Horn clause, which can be either a fact or a rule. PROLOG works backward from the goal using a rule of inference called *resolution* to find a series of facts and rules which can be used to prove the goal. This approach is called *backward chaining* because the system works backward from a goal to find a path to prove that the goal is true. CLIPS, on the other hand, works in the opposite direction. CLIPS takes asserted facts and attempts to match them to rules to make inferences. This process is known as *forward chaining*. The concept of a goal does not exist in CLIPS. The diagram shown in Fig. 2 illustrates the overall architecture of the CLIPS system [8]. If PROLOG cannot prove a goal, it assumes the goal to be false—called the *closed-world assumption*. In either case, the task of satisfying a goal is left to the inference engine.

For purposes of comparison, we consider resolution in PROLOG. Given the color fact base above, we can submit the following queries:

```
1 ?- color(red).
2 true.
3 ?- color(X).
4 X = red;
5 X = yellow;
6 X = blue;
7 false.
8 ?- color(pink).
9 false.
```

Note that the case of the first letter of a term indicates whether it is interpreted as data (lowercase) or as a variable (uppercase). Variables must begin with a capital letter or an underscore. Thus, the goal `color(X).` (on line 3) returns as many values as we request for which the query is true.[4] "This process of determining useful values for variables is called *unification*. The temporary assigning of values to variables to allow unification is called *instantiation*" [7].

The following is another example of a set of facts (sometimes called a database) that describe a binary tree. Notice also a `path` predicate which defines a path between two vertices, with two rules, to be either an edge from `X` to `Y` (on line 6) or a path from `X` to `Y` (on line 7) through some intermediate vertex `Z` such that there is also an edge from `X` to `Z` and a path from `Z` to `Y` [8].

```
1 edge(a,b).
2 edge(a,c).
3 edge(b,d).
4 edge(c,e).
5
6 path(X,Y) :- edge(X,Y).
7 path(X,Y) :- edge(X,Z),path(Z,Y).
```

The user can then query the program by expressing goals to determine whether the goal is true or to find all instantiations of variables which make the goal true. For instance, the goal `path(a,c)` asks if there exists a path between vertices `a` and `c`.

```
?- path(a,c).
true .
```

To prove that goal, PROLOG uses resolution, which includes unification. When the goal `path(a,c)` is given, PROLOG performs its resolution algorithm through the following steps:

1. {} :- path(a, c).
2. path(X, Y) :- edge(X, Y).
3. path(a, c) :- edge(a, c).
4. edge(a, c) :- {}.
5. path(a, c) :- {}.
6. {} :- {}

[4]Additional solutions are requested with a 'n' or ';' keystroke.

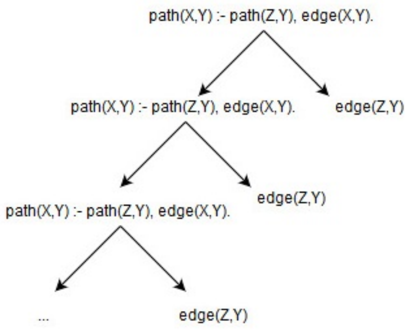

path(X,Y) :- path(Z,Y), edge(X,Y).
path(X,Y) :- path(Z,Y), edge(X,Y).
edge(Z,Y)
path(X,Y) :- path(Z,Y), edge(X,Y).
edge(Z,Y)
...
edge(Z,Y)

Fig. 3: Resolution graph illustrating an infinite expansion of the `path` predicate.

At step 3, PROLOG unifies the clauses `path(a,c)` and `path(X,Y)`, which involves instantiating `X` and `Y` with the values `a` and `c`, respectively. On steps 4 and 5, the fact `edge(a,c)` is resolved with the clause `path(a,c) :- edge(a,c)` to deduce that `path(a,c)` is true. The goal `path(a,E)` returns all the values of `E` that satisfies this goal.

```
?- path(a,E).
E = b ;
E = c ;
E = d ;
E = e ;
false.
```

We can also query for all the paths with the goal `path(X,Y).`:

```
?- path(X,Y).
X = a,
Y = b .

?- path(X,Y).
X = a,
Y = b ;
X = a,
Y = c ;
X = b,
Y = d ;
X = c,
Y = e ;
X = a,
Y = d ;
X = a,
Y = e ;
false.
```

Consider the following rewrite of the `path` predicate:

```
path(X,Y) :- edge(X,Y).
path(X,Y) :- path(Z,Y), edge(X,Z).
```

While there is no inherent order to clauses in an antecedent in a proposition in first-order predicate calculus, an implemented system like PROLOG must pursue sub-goals in the antecedent of a proposition in a deterministic order. In PROLOG, during resolution, clauses in the antecedent of a proposition are evaluated from left to right. This, of course, is problematic with predicates defined in a left-recursive fashion as is the case with the rewrite of the `path` predicate above. Specifically, since PROLOG clauses are evaluated from left to right, Z will never be bound to a value.

```
?- path(X,Y).
X = a,
Y = b ;
X = a,
Y = c ;
X = b,
Y = d ;
X = c,
Y = e ;
X = a,
Y = d ;
X = a,
Y = e ;
ERROR: Stack limit (1.0Gb) exceeded
ERROR:  Stack sizes: local: 1.0Gb, global: 17Kb, trail: 2Kb
ERROR:  Stack depth: 12,200,621, last-call: 0%, Choice points: 5
ERROR:  Probable infinite recursion (cycle):
ERROR:    [12,200,620] user:path(_4520, d)
ERROR:    [12,200,619] user:path(_4540, d)
```

The resolution tree in Fig. 3 illustrates this idea. The left-to-right evaluation strategy of clauses causes this tree to be searched in a depth-first fashion, which leads to an infinite expansion of the rule. Rules are also pursued in a top-down fashion. Thus, if we reverse the two rules defining the path predicate given above, the stack overflow occurs immediately without any output:

```
path(X,Y) :- path(Z,Y),edge(X,Z).
path(X,Y) :- edge(X,Y).


?- path(X,Y).
ERROR: Stack limit (1.0Gb) exceeded
ERROR:  Stack sizes: local: 1.0Gb, global: 16Kb, trail: 2Kb
ERROR:  Stack depth: 6,710,347, last-call: 0%,
        Choice points: 6,710,342
ERROR:  In:
ERROR:    [6,710,347] user:path(_4384, _4386)
ERROR:    [6,710,346] user:path(_4404, _4406)
ERROR:    [6,710,345] user:path(_4424, _4426)
ERROR:    [6,710,344] user:path(_4444, _4446)
ERROR:    [6,710,343] user:path(_4464, _4466)
ERROR:
ERROR: Use the --stack_limit=size[KMG]
         command line option or
ERROR: ?- set_prolog_flag(stack_limit, 2_147_483_648).
           to double the limit.
```

Thus, it is important to ensure that variables can be bound to values during resolution before they are used recursively.

3. Going Further in CLIPS

We briefly discuss three programming language concepts that are helpful in CLIPS programming.

3.1 Variables

Variables in CLIPS are prefixed with a ? (e.g., ?x). Variables need not be declared explicitly. However, variables must be bound to a value before they are used. Consider the following program that computes a factorial:

```
(defrule factorial
  (factrun ?x)
  =>
  (assert (fact ?x 1)))

(defrule facthelper
  (fact ?x ?y)
  (test (> ?x 0))
  =>
  (assert (fact (- ?x 1) (* ?x ?y))))
```

When the facts for the rule facthelper are pattern matched, ?x and ?y are each bound to a value. Next, the bound value for ?x is used to evaluate the validity of the fact (test (> ?x 0)). When variables are bound within a rule, that binding exists only within that rule. For persistent global data, defglobal should be used as follows:

```
(defglobal ?*var* = "" )
```

Assignment to global variables is done with the bind operator.

3.2 Templates

Templates are used to associate related data (e.g., facts) in a single package—similar to structs in C. Templates are containers for multiple facts, where each fact is a slot in the template. Rules can be pattern matched to templates based on a subset of a template's slots. Below is a demonstration of the use of pattern matching to select specific data from a database of facts.

```
(deftemplate car
  (slot make
    (type SYMBOL)
    (allowed-symbols
      truck compact)
    (default compact))
  (multislot name
    (type SYMBOL)
    (default ?DERIVE)))
(deffacts cars
  (car (make truck)
      (name Tundra))
  (car (make compact)
      (name Accord))
  (car (make compact)
      (name Passat)))

(defrule compactcar
  (car (make compact)
       (name ?name))
  =>
  (printout t ?name crlf))
```

3.3 Conditional Facts in Rules

Pattern matching need not match an exact pattern. Logical operators—*or* (|), *and* (&), and *not* (~)—can be applied to pattern operands to support conditional matches. The following rule demonstrates the use of these operators:

```
(defrule walk
  (light ~red&~yellow) ; if the light
                       ; is not yellow and
                       ; is not red
  (cars none|stopped) ; no cars or stopped
  =>
  (printout t "Walk" crlf))
```

4. Applications of CLIPS and PROLOG

We briefly introduce some applications of CLIPS and PROLOG.

4.1 Decision Trees

An application of CLIPS is decision trees. More generally, CLIPS can be applied to graphs that represent a human decision-making process. Facts can be thought of as the edges of these graphs, while rules can be thought of as the actions or states associated with each vertex of the graph. An example of this decision-making process is an expert system that emulates a physician in treating, diagnosing, and explaining diabetes [9]. The patient asserts facts about herself including eating habits, blood-sugar levels, and symptoms. The rules within this expert system match these facts and provide recommendations about managing diabetes in the same way a physician may interact with a patient.

4.2 Graphs

We can model graphs in PROLOG using a list whose first element is a list of vertices and whose second element is a list of directed edges, where each edge is a list of two elements—the source and target of the edge. Using this list representation of a graph, a sample graph is: `[[a,b,c,d],[[a,b],[b,c],[c,d],[d,b]]]`.

Consider the following definitions of the primitive `append` and `member` predicates, which we use in subsequent examples, with sample queries and outputs:

```
append([], X, X).
append([X|L1], L2, [X|L12]) :- append(L1, L2, L12).

member(X, List) :- append(_, [X|_], List).

?- append([a,b,c], [d,e,f], Y).
Y = [a, b, c, d, e, f].

?- member(4, [2,4,6,8]).
true.
```

Using the `append` and `member` predicates (and others not defined here, e.g., `flatten` and `makeset`), we define a `graph` predicate:

```
graph([Vertices,Edges]) :- checkDuplicateEdge(Edges),
   flatten(Edges, X), makeset(X, Y), subset(Vertices, Y).

edge([Vset,Eset], Edge1) :-
   graph([Vset,Eset]), member(Edge1, Eset).

vertex([Vset,Eset], Vertex1) :-
   graph([Vset,Eset]), member(Vertex1, Vset).
```

The `graph` predicate tests whether a given input represents a valid graph by checking if there are no duplicate edges and that the defined edges do not use vertices which are not included in the vertex set. The `edge` predicate takes a graph and an edge and returns `true` if the graph is valid and the edge is a member of that graph's edge set, and `false` otherwise. The `vertex` predicate serves the same purpose for vertices. These predicates serve as building blocks from which we can construct more interesting predicates. For example, we can check if one graph is a subgraph of another one. We can also check whether or not a graph has a cycle in general or a cycle containing a given vertex. (A *chain* is a list of vertices such that each vertex is adjacent to the next vertex in the list. A *cycle* is a chain in which the final vertex is adjacent to the first.) Consider the following PROLOG predicates:

```
subgraph([Vset1,Eset1], [Vset2,Eset2]) :-
   graph([Vset1,Eset1]), graph([Vset2,Eset2]),
   subset(Vset1,Vset2), subset(Eset1,Eset2).

has_cycle(Graph, Vertex) :- chain(Graph, Vertex, Vertex, _).

cycle_vertices(G, [V1|Vset]) :-
   has_cycle(G, V1); cycle_vertices(G, Vset).

has_cycle([[V1|Vset], Eset]) :-
   cycle_vertices([[V1|Vset],Eset], [V1|Vset]).
```

Note that the above predicates make use of the `chain` predicate which checks if there is a path from a start vertex to an end vertex in a graph. As stated above, a *cycle* is a path where the start vertex and end vertex are the same vertex. Note that here we model edges of the graph as a list of lists. If edges are modeled as facts, a different `cycle` predicate must be defined:

```
edge(a,b).
edge(b,a).
edge(a,c).
edge(c,d).
edge(d,a).

cycle(Start, Visited) :-
   cycle(Start, Start, [Start], Visited).

cycle(Orig, Start, Path, Visited) :-
   edge(Start,Orig), reverse([Orig|Path], Visited).

cycle(Orig, Start, Path, Visited) :-
   edge(Start, Next), \+ member(Next, Path),
   cycle(Orig, Next, [Next|Path], Visited).
```

As final examples, we illustrate predicates for identifying a graph with no edges (or an independent set) and a complete graph. (An *independent set* is a graph with no edges, or a set of vertices with no edges between them. A *complete graph* is a graph in which each vertex is adjacent to every other vertex.) These two classes of graphs are complements of each other. To identify an independent set, we must check if the edge set is empty. On the other hand, a complete directed graph has no loops, but all other possible edges. A complete directed graph with n vertices has exactly $n \times (n-1)$ edges. Thus, we can check if a graph is complete by verifying that a valid graph has no self-edges and that the number of edges satisfies this condition. The following are `independent` and `complete` predicates, and helper `count` and `proper` predicates, for implementing these tests:

```
/* count number of elements in a list */
count([],0).
count([_|X], Y) :- count(X, Z), Y is Z+1.

/* for graph with X vertices , checks if X(X-1) edges */
proper(Y, X) :- Z is Y - X*(X-1), Z == 0.

/* checks if graph is an independent set */
independent([Vset, []]) :- graph([[Vset], []]).

/* checks if graph is complete ( infinite recursion problem) */
complete([Vset,Eset]) :-
   graph([Vset,Eset]), not(member([X,X], Eset)),
   count(Vset, X), count(Eset, Y), proper(Y, X).
```

Note that infinite recursion may arise. In particular, in some cases, when a goal is pursued, the program correctly outputs `true`. However, if the user presses the semicolon key rather than the period key, the system will return `true`, what appears to be, an infinite number of times. Consider the following sample goals and corresponding output.

```
?- graph([[a,b,c],[[a,b],[b,c],[d,a]]]).
false.
?- edge([[a,b,c],[[a,b],[b,c], [d,a]]], [a,b]).
true.
?- has_cycle([[a,b,c,d],[[a,b],[b,c],[c,d],[d,b]]]).
true.
?- has_cycle([[a,b,c,d],[[a,b],[b,c],[c,d],[d,b]]], a).
false.
?- has_cycle([[a,b,c,d],[[a,b],[b,c],[c,d],[d,b]]], d).
true.
?- subgraph([[a,b,c],[[a,b],[a,c],[b,c]]], [[a,b,c],[[a,b],[a,c]]]).
true.
?- complete([[],[]]).
true.
?- complete([[a,b,c],[[a,b],[a,c],[b,a], [b,c],[c,a],[c,b]]]).
true.
```

4.3 Natural Language Processing

One application of PROLOG is *natural language processing* [10], [11]—the search engine used by PROLOG naturally functions as a recursive-descent parser. One could conceive facts as terminals and rules as non-terminals or production rules. Consider the following simple grammar:

<sentence>	::=	*<noun phrase> <verb phrase>*
<noun phrase>	::=	*<determiner> <adj noun phrase>*
<noun phrase>	::=	*<adj noun phrase>*
<adj noun phrase>	::=	*<adj> <adj noun phrase>*
<adj noun phrase>	::=	*<noun>*
<verb phrase>	::=	*<verb> <noun phrase>*
<verb phrase>	::=	*<verb>*

Using this grammar, a PROLOG program can be written to verify the syntactic validity of a sentence. Note that the candidate sentence is represented as a list where each element is a single word in the language (e.g., `sentence(["The","dog","runs","fastly"])`).

```
sentence(S) :-
   append(NP, VP, S), noun_phrase(NP), verb_phrase(VP).

noun_phrase(NP) :-
   append(ART, NP2, NP), det(ART), noun_phrase_adj(NP2).

noun_phrase(NP) :- noun_phrase_adj(NP).

noun_phrase_adj(NP) :- append(ADJ, NPADJ, NP),
                       adjective(ADJ), noun_phrase_adj(NPADJ).

noun_phrase_adj(NP) :- noun(NP).

verb_phrase(VP) :- append(V, NP, VP), verb(V), noun_phrase(NP).

verb_phrase(VP) :- verb(VP).
```

One drawback of using PROLOG to implement a parser is that left-recursive grammars cannot be implemented for the same reasons discussed in § 2.3. Other applications of PROLOG include Petri nets, recommender systems, and deep learning [10].

5. Conclusion

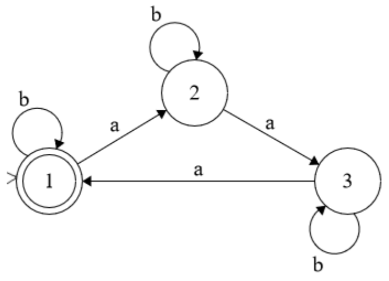
The declarative nature of programming in CLIPS and PROLOG sets the languages apart from other programming languages. Problems are solved by specifying a description of the solution, not a series of instructions to compute a solution—a PROLOG “state of mind” [12]. CLIPS operationalizes this motif with the Rete Algorithm; PROLOG operationalizes this motif through resolution and unification. CLIPS is a language which allows for the implementation of ‘rule of thumb’ knowledge similar to that of a human expert. These forward-chaining systems help a computer make heuristic decisions by considering the facts (or lack of facts) of a situation and drawing conclusions. Because CLIPS can handle arbitrarily large data, it has an advantage over human experts because humans are limited in their ability to consider many facts at once. For YouTube videos of CLIPS and PROLOG presented and recorded by the authors, visit `https://www.youtube.com/watch?v=XX8Fxze6Np8&t=2s` (CLIPS) and `https://www.youtube.com/watch?v=SfXOgyOl3LU` (PROLOG).

Appendix

A CLIPS Programming Exercises

The following are programming exercises that involve essential CLIPS concepts:

1) Build a finite state machine using CLIPS that accepts a language L consisting of strings in which the number of a’s in the string is a multiple of three over an alphabet {a,b}. Use the following state machine for L:



Examples:

```
CLIPS> (run)
String? aaabba
Rejected

CLIPS> (reset)
CLIPS> (run)
String? aabbba
Accepted

CLIPS>
```

2) Rewrite the factorial program in § 3.1 so that only the fact with the final result of the factorial rule is stored in the fact list. Note that `retract` can be used to remove facts from the fact list.

Examples:

```
CLIPS> (assert (fact_run 5))
CLIPS> (run)
CLIPS> (facts)
f-0 (fact_run 5)
f-1 (fact 0 120)

CLIPS>
```

B PROLOG Programming Exercises

The following are programming exercises that involve essential PROLOG concepts:

1) A *multiplexer* is a device that selects one of many inputs to output based on a select line input. Define a PROLOG predicate that acts as a 4-input 2-bit *multiplexer*.
Examples:

```
?- mux("1", "2", "3", "4", 1, 1, Output).
Output = "4".

?- mux("1", "2", "3", "4", 0, 1, Output).
Output = "2".
```

2) Define a PROLOG predicate that takes two cities and a route and determines if that route is a valid. Excluding fact declarations, this program should be approximately 15 lines of code. The roads need not be implicitly bi-directional.
Sample list of cities:

```
road(paris, rouen).
road(paris, lyon).
road(lyon, marseille).
road(marseille, nice).
road(paris, bordeaux).
road(paris, caen).
road(bordeaux, madrid).
road(madrid, cuenca).
```

Examples:

```
?- route(paris, caen, [paris, caen]).
true.

?- route(paris, cuenca, Route).
Route = [paris, bordeaux, madrid, cuenca].
```

3) Define a PROLOG predicate which takes an infix arithmetic expression and an integer and determines whether the integer results from evaluating the expression. The predicate need not handle a divide by 0 error. Use the following grammar:

$$\begin{array}{lrcl} (r_1) & <expression> & ::= & <number>\ <op>\ <expression> \\ (r_1) & <expression> & ::= & <number>\ <op>\ <number> \\ (r_3) & <op> & ::= & + \ \mid\ - \ \mid\ * \ \mid\ / \end{array}$$

Examples:

```
?- expr([3,*,39,+,3], 120).
true.

?- expr([3,*,39,+,3], 39).
false.

?- expr([3,*,39,+,3], X).
X = 120.
```

References

[1] J. C. Giarratano, *CLIPS User's Guide*. Cambridge, MA: The MIT Press, 2008.

[2] P. Brna, "Prolog programming: A first course," 2001, available from https://courses.cs.washington.edu/courses/cse341/03sp/brna.pdf [Last accessed: 22 May 2019].

[3] W. Clocksin and C. Mellish, *Programming in Prolog*, 5th ed. Berlin, Germany: Springer-Verlag, 2003.

[4] M. Kifer and Y. Liu, Eds., *Declarative Logic Programming: Theory, Systems, and Applications*. New York, NY: ACM and Morgan & Claypool, 2018.

[5] F. Pereira, "A brief introduction to Prolog," *ACM SIGPLAN Notices*, vol. 28, no. 3, pp. 365–366, 1993.

[6] J. Giarratano and G. Riley, *Expert systems principles and programming*. Boston, MA: PWS Publishing Company, 1998.

[7] R. Sebesta, *Concepts of Programming Languages*, 9th ed. Boston, MA: Addison-Wesley, 2010.

[8] P. Lucas and L. van der Gaag, *Principles of expert systems*. Boston, MA: Addison-Wesley, 1991.

[9] M. Garcia, T. Gandhi, J. Singh, L. Duarte, R. Shen, M. Dantu, S. Ponder, and H. Ramirez, *Esdiabetes (an Expert System in Diabetes)*. Consortium for Computing Sciences in Colleges, 2001.

[10] J. Eckroth, *AI Blueprints: How to build and deploy AI business projects*. Packt Publishing, 2018.

[11] C. Matthews, *An introduction to natural language processing through Prolog*. London, United Kingdom: Longman, 1998.

[12] J. Eckroth, "AI education matters: Biductive computing with Prolog," *AI Matters*, vol. 5, no. 1, pp. 14–16, 2019.