Extending Prolog for Better Natural Language Analysis

Khaled Shaalan
Computer and Information Sciences Dept., Institute of Statistical Studies and Research (ISSR),
Cairo Univ., 5 Tharwat St., Orman, Giza, Egypt
E-mail: shaalan@esic.claes.sci.eg

Abstract–*Prolog supports natural language parsing with a clean semantics and additional constructs such as definite clause grammars (DCGs). While it provides excellent computational support, we claim it does not provide good notation that increases the readability and maintainability of natural language analysis programming. In this paper we explore an alternative solution: a general notational extension to Prolog programs that provides for concise expression of definitions. This notational extension results in a powerful and convenient logic programming language that fits into natural language analysis programming. Programs translate to Prolog in a way similar to DCGs. That is to say, they have a specific syntax and can be loaded and expanded to Prolog code. This expansion is transparent to the user. To demonstrate the language capabilities, we present an example for an Arabic morphological analyzer.*

Keywords–Prolog, logic grammar, natural language processing

1. Introduction

Natural language analysis programming is attracting a significant amount of attention from researchers aiming at developing applications that understand natural languages. Logic programming plays an essential role in natural language analysis process because it attempts to use logic to express grammar rules and to formalize the process of parsing [5]. A grammar specified this way is known as *logic grammar* since it represents rules as Horn clauses [4]. Logic grammars can be conveniently implemented in Prolog. Prolog-based grammars can be quite efficient in practice [1]. Prolog interpretation algorithm uses exactly the same search strategy as the depth-first top-down parsing algorithm, so all it is needed is a way to reformulate grammar rules as clauses in Prolog.

Researches have evolved through the years, motivated in such concerns as ease of implementation, further expressive power, a view towards a general treatment of some language processing problems, or towards automating some part of the grammar writing process, such as the automatic construction of parse trees and internal representations. Generality and expressive power seem to have been the main concerns underlying all these efforts.

In this paper we explore a new formalism, namely *threaded clause grammars (TCGs)*, which we believe to increase the readability and maintainability of natural language analysis programming. This formalism results in a powerful and convenient logic programming language. TCGs are logic grammars augmented with explicit multiple accumulators, called *threaded variables*, useful in particular to make input and output transparent to users and provide for an easier to understand concise expression of definitions. The new formalism can be considered as a language on top of Prolog. TCGs programs directly translate to Prolog code. Its semantics is described in terms of translation to Prolog. The translation is based on source-to-source transformations. For each transformation we give a definition of a rewrite rule. To demonstrate its capabilities in solving natural language analysis problems, we present an example for an Augmented Transition Network (ATN) for Arabic word morphology.

This paper is structured as follows. Section 2 gives the background and related work of this research. Section 3 presents the syntax and semantics of TCGs. An example that more clearly illustrates the potential of this formalism is an Arabic morphological analyzer. This is described in Section 4. Section 5 concludes the article.

2. Background and Related Work

In natural language analysis programming, we usually use arguments to accumulate and pass information around. In Prolog, if information is accumulated through out a computation, a pair of arguments– an accumulator– is needed: one for the information up to the call and one for the information following it. The reason is that Prolog is a single-assignment language where variables are not subject to update in-place. The successive use of these arguments constructs a chain of variables, having exactly two occurrences each.

We can see that in real programs, such as a language analyzer, there are typically much more information that needs to be modified and passed around. In a utilization of these argument pairs, the following difficulties concerning the readability and maintainability of programs can be noted:

- *Verbose description in data change.* Each clause must at the very least repeat the names of argument pairs in the head and/or the body of the predicate definition. When a clause has several variables, where only a few of them are accessed or changed, then the notation of Prolog programs becomes quite cumbersome. This is due to the need to specify explicitly all the variables that do not change in the clause body.
- *Proliferation of variables along the chain of argument pairs.* Different names need to be invented for the old and new incarnations of the variable that did change in the clause body.
- *A greater likelihood of program errors.* Unfortunately, this tangle of arguments causes several problems. It is easy to switch two arguments around, omit one, or even just misspell one. The most important consequence of having these errors is a greater reduction in the readability and maintainability of programs. Worse, having to type all of these variable names and to fix the inevitable errors reduces productivity.

In the following, the proposals that solve the above problem will be examined after a brief discussion of issues related to the design and implementation concerning the argument pairs in Prolog.

1. *Number of argument pairs*. If no restrictions is placed on the number of argument pairs, the capabilities of expressing an arbitrary number of multiple argument pairs is allowed.
2. *Transparency to users*. Transparency provides facility to access data bound to a variable without placing extra burdens on the programmers. In a sense, changes to data are visible to the program and the user.
3. *Locality of definitions*. If it is desirable to program within the pure logical subset of Prolog and to follow the Edinburgh tradition, then locality of definitions should be preserved. In that case a predicate's meaning depends only on its definitions, and not on any outside information, e.g. globally accessible variables.

4. *Non-logical data change*. This is based on the destructive modification of logic program through the `assert` and `retract` operators of Prolog which insert and delete a clause from the program respectively. There are many problems with `assert/retract` which motivate e.g. Warren [12] to talk about "The evils of assert" and which make programming using these operators a rather non-declarative effort. A more recent solution is proposed in [10] has introduced the new primitives `assumel` and `assumei` to allow backtrackable destructive assignment. These non-standard primitives are implemented at the WAM-level. Although in both cases these facts are removed on backtracking, the semantics of `assumei` requires that variables in the assumption be not shared with references. In the current implementation, this means that an "assumed" atom is copied and copied again on invocation.
5. *Extra declarations*. Sometimes implicit declarations do not come complete, we must have extra declarations to relate program entities with each other. Extra declarations strongly affect maintainability–they can weaken it considerably.
6. *Structure of data*. If the data represented by an argument pair is not restricted to be a list structure, highly generic code can be written.
7. *Concision of definitions.* This would reduce both the size of code and the possibility of errors. Very simply, the more code you write, the more likely you are to make a mistake.

In the last few years, a number of notational extensions have been proposed which extends Prolog to support natural language analysis programming. We distinguish between two main groups of these extensions: 1) Prolog extended with implicit argument pairs that are associated with definitions through default conventions, and 2) Prolog extended with explicit accumulators. Although argument pairs in its original form is written explicitly, any language extension that offers some form of transparency to users is considered to be explicit in this paper.

There is a long history of implicit data change in Prolog. Because Prolog has always had `assert` and `retract` built-ins, these provided an early way of passing hidden information. Because they do not undo their effect on backtracking, leaving an unwanted "residue", they make it more difficult to give a declarative reading to a program. More pragmatically, maintaining such a program is difficult, while debugging one cannot simply retry a goal to see what happens without being sure to reset any state that has been modified.

For more than a decade, definite clause grammars (DCGs) notation was developed as a result of research in natural language parsing and understanding [6]. DCGs allow one to write attributed grammar rules directly in Prolog, producing a simple recursive decent parser. Prologs that conform to Edinburgh standard [2] have DCGs as a part of their implementations. The DCGs notation implements a single implicit argument pair. Predicates that implement each grammar rule take two additional arguments to represent a segment of the string being parsed, i.e. a list structure. Even though DCGs is easy to implement it is highly restrictive within the number of pairs, transparency to users, and data structures.

Taura et al. [10,3] proposed a generalization of DCGs, called Assumption Grammars (AGs). AGs are logic grammars augmented with hidden multiple accumulators, and linear and intuitionistic assumptions scoped over the current AND-continuation, i.e.

having their assumptions available in future computations on the same resolution branch instead of keeping them local to the body of a clause. AGs allow backtrackable destructive assignment that adds temporarily a clause usable in later proof. This sort of extension is implemented by extending the Prolog system itself that greatly reduces portability.

More recently, Schachte [8] proposed another model of global variables. These variables are assigned to and referenced much like global variables in imperative languages. A global variable is implicitly "created" when it is first assigned a value, and exists until the end of the computation. Its value may be accessed at any time after it is first assigned; it is an error to attempt to access it before it has been assigned. The implementation avoids the copying of BinProlog's `assumei`. With the addition of global variables, conjunction is no longer commutative or absorptive.

Van Roy [11] developed an extension, called extended DCGs notation (EDCGs), a more closely comparable to the present work, which allows any number of argument pairs to be threaded through the program. This also works as a translator, but in this case the user declares each individual accumulator, specifying the goal to execute in order to accumulate a new term in it. The user then must give extra declarations for which predicates should get which accumulators as arguments. So, instead of explicitly adding all these argument pairs to a definition, EDCGs will use these declarations to add them for you.

We propose a simpler formalism, threaded clause grammars (TCGs), featuring threaded variables that enables easier capture of large natural language analysis programs. Threaded variables can be used for tasks other than language analysis, which is an acquired programming taste. As indicated by its name, threaded variables are needed to cater for the input and output version of some changing data that is passed along from one definition to another. A primary advantage of this formalism is its simplicity. Most logic programmers understand argument pairs technique from their use in programs. An important effect is that programs become unnecessarily complicated and difficult to read and maintain. In addition, the approaches to extend Prolog with globally accessible variables and implicit argument pairs are difficult to read because they hide information flow. On the contrary, the use of threaded variables is transparent and has more declarative reading. Nevertheless, the present work avoids the need for extra declarations as required by EDCGs that become difficult to maintain for large programs. We will also show that it is easy to give semantics to TCGs programs.

3. Definition of TCGs

3.1 The Syntax of Clauses and Goals

The (abstract) syntactic category pertaining to programs in TCGs is shown in Fig. 1. We use the following notations in describing the syntax of programs: letters between < and > are nonterminal symbols, letters between ′ and ′ are terminal symbols and the symbol | separates alternatives.

```
            <clause>   ::=  <head> ':-' <body>
              <head>   ::=  <Prolog term>
                         |  <threaded term>
      <threaded term>   ::=  <Prolog term> '-' <threaded variables>
<threaded variables>    ::=  <variable>
                         |  '('<variable> ',' <variable>')'
                         |  <variable> '-' <threaded variables>
                         |  '('<variable> ',' <variable>')' '-'
                            <threaded variables>
              <body>    ::=  <sequence of goals>
              <goal>    ::=  <Prolog goal>
                         |  <threaded goal>
      <threaded goal>   ::=  <threaded term>
                         |  <Prolog list> '-' <variable>
```

Fig. 1 Abstract syntax of threaded clauses.

A *program* is a finite set of clauses, satisfying the condition that every goal occurring in the programs has a definition.

3.2 Simple Examples

The expansion of clauses to Prolog code is transparent to the user. This is best clarified by examples. As a simple example, consider the clause `tree_acc/3` that accumulates elements of a tree in a list. This clause takes a tree and an empty list as its arguments and returns a list of elements in that tree.

```
tree_acc(Tree)-L:-
      prelude(Tree)-L,
      process(Tree)-L,
      postlude(Tree)-L.
```

The expression `-L` is a *syntactic variable* that indicates a pair of arguments: the initial empty list and the final list including items accumulated from the tree. As indicated by its name, threaded clauses couples variables that are needed to cater the input and output version of some changing data, a tree in this case, that is passed along from one definition to another. The above clause is expanded to Prolog as follows.

```
tree_acc(Tree,L0,L):-
      prelude(Tree,L0,L1),
      process(Tree,L1,L2),
      postlude(Tree,L2,L).
```

The variable `L0` is the initial empty list, `L1` and `L2` are lists containing items accumulated from `prelude/3` and `process/3`, and `L` is the final list containing items accumulated from `postlude/3`. The two arguments in each goal are needed to cater for the changing of data. It is worth noting that the new variable names in the above definition are distinct.

Notice that the above expansion is transparent to the user. For example, if it is desired to display the list accumulated from `process/3` using the definition `portray_list/1`, all we have to do is to write the goal `portray_list(L)` after `process/3` as follows.

```
tree_acc(Tree)-L:-
      prelude(Tree)-L,
      process(Tree)-L,
      portray_list(L), /* portray_list(L2) */
      postlude(Tree)-L.
```

This would expand to the following goal `portray_list(L2).` This means that threaded variables can be embedded in arbitrary goals and terms.

Furthermore, sometimes it is desired to use explicit input and output at some point in a chain of threaded variables. To do this you simply write (Arg_i, Arg_o) in place of the threaded variable. Example:

```
foo(A,B)-S-T :-
      bar1-S-T,
      bar2-S-(A,B),
      bar3-T.
```

is equivalent to:

```
foo(A,B,S0,S,T0,T) :-
      bar1(S0,S1,T0,T1),
      bar2(S1,S,A,B),
      bar3(T1,T).
```

3.3 Expansion to Prolog

The expansion (translation) of the threaded clauses to Prolog clauses is based on source-to-source transformations. The operational and declarative semantics of the threaded programs is given in terms of their translation to Prolog. So long as this translation is straightforward, threaded definitions are able to exploit the clean semantics of Prolog.

The expansion of a threaded clause into Prolog code is based on transformation rules. Each kind of transformation is defined by a rewrite rule, $D \Rightarrow D'$, which substitutes the definition D', on its right hand side, for the definition D, on its left hand side.

In the transformation rules that follows, the notation

$$Var_i^o \Rightarrow Var_i, Var_o$$

denotes a pair of arguments that are produced by threading the input argument Var_i with the output argument Var_o.

- Clause transformation rule

  $clause \Rightarrow clause'$

  Where `clause` is a TCGs clause and `clause'` is a Prolog clause.
- Head transformation rule

  $head \Rightarrow head'$

  Where `head` is a TCGs head and `head'` is a Prolog head.
- Threaded term transformation rule

  Let $a_1, \ldots, a_n$ represent the arguments in `term`, let $a_1, \ldots, a_{n+1}, \ldots, a_m$ represent the arguments in `term'`

- `term-ThreadedVars ⇒ term′`

  Where `term` and `term′` refers to Prolog terms. `Threadedvars` is transformed to $a_{n+1}, \ldots, a_m$ according to the *threaded variables transformation* rules. Each embedded threaded variable in `term` is replaced by an embedded variable in `term′` that corresponds to its closest transformation.

- Threaded variables transformation rules

– Threaded variable transformation rule

$Var_i^o \Rightarrow Var_i, Var_o$

Where Var°_i represents a threaded variable in TCGs, and Var_i and Var_o refer to Prolog variables.

- Explicit variables pair transformation rule

$(Var_i, Var_o) \Rightarrow Var_i, Var_o$

Where Var_i and Var_o refer to Prolog variables.

– Multiple Threaded variables transformation rule

$Var_i^o - ThreadedVars \Rightarrow Var_i, Var_o, ThreadedVars'$

Where Var°_i represents a threaded variable in TCGs, and Var_i and Var_o refer to Prolog variables. `Threadedvars` is transformed to `ThreadedVars′` according to the *threaded variables transformation rules*.

– Multiple explicit variable pairs transformation rule

$(Var_i, Var_o)\text{-}ThreadedVars \Rightarrow Var_i, Var_o, ThreadedVars'$

Where Var_i and Var_o refer to Prolog variables and `Threadedvars` is transformed to `ThreadedVars′` according to the *threaded variables transformation* rules.

- Body transformation rule

`body ⇒ body′`

Where `body` is a TCGs body and `body′` is a Prolog body.

- Goal transformation rule

`goal ⇒ goal′`

Where `goal` is a TCGs goal and `goal′` is a Prolog goal. Each embedded threaded variable in `goal` is replaced by an embedded variable in `goal′` that corresponds to its closest transformation.

- Threaded goal transformation rules

– General Threaded goal transformation rule

`threaded goal ⇒ goal′`

Where `threaded goal` is transformed to `goal′` according to the *threaded term transformation rule*.

– List transformation rule

$[T_1, T_2, \Lambda, T_n] - Var_i^o \Rightarrow Var_i = [T_1, T_2, \Lambda, T_n \mid Var_o]$

Where $T_1, T_2, \ldots, T_n$ are Prolog terms, Var°_i represents a threaded variable in TCGs, Var_i and Var_o refer to Prolog variables, and $[T_1, T_2, \ldots, T_n]$ and $[T_1, T_2, \ldots, T_n \mid Var_o]$ are Prolog lists.

– Empty list transformation rule

$[] - Var_i^o \Rightarrow Var_i = Var_o$

Where Var°_i represents a threaded variable in TCGs, Var_i and Var_o refer to Prolog variables, and `[]` is the empty list.

3.4 Discussion

Prolog is no different from any other language in its need for a good programming style to build and maintain real applications [9]. One basic concern in composing Prolog programs has been to make them as readable as possible to increase program

clarity. Coining a good readable and declarative notation for a logic language does not come easily. A program must be considered as a whole. Its readability is determined by its physical layout and by the choice of names appearing in it.

Composing a program is a cyclic activity in which names are constantly being reworked to reflect our improved understanding of our creation. Providing convenient notations that make it easier for people to develop good style enhances its maintainability.

In TCGs, a one major design issue is to make programs more readable and easier to understand through the notational extensions that highlight the significant variables. The advantage is that declarativeness is preserved since the coupling of variables is due to the single assignment nature of Prolog. In other words, the reading of clauses becomes declarative rather than procedural. This issue is relaxed by the virtue of transparency. Another design goal is to facilitate program maintenance by relying on the concision of definitions, which reduces the size of code. This has an important impact to program development. We often find in maintaining a program that adding new argument pairs is required in order to communicate information through the program whose need was unanticipated. Besides, the need for modifying or extending goals at some point in the clause body that involves a chain of argument pairs. Consequently, many variables have to be named, typed, and probably some others need to be fixed. As a matter of fact, programming with threaded variables is two fold. It provides for the concision of code and preserves data flow dependencies through threaded arguments.

The expansion described in this section is implemented using SICStus Prolog's `term_expansion` facility that allows users to define their own translations from the clauses and terms appearing in a source file into the actual clauses of the program. The translation is straightforward, as there is a one-to-one correspondence between threaded clauses and Prolog clauses. That is, we translate a program with threaded clauses into the Prolog program that the user would have written without these notations. Therefore, the efficiency of TCGs programs is not compromised.

4. An Arabic Morphological Analyzer

In [7] we described a morphological analyzer for inflected Arabic words. An augmented transition network (ATN) [13] technique was successfully used to represent the context-sensitive knowledge about the relation between a stem and inflectional additions. An exhaustive-search to traverse the ATN generates all the possible interpretations of an inflected Arabic word. The arcs of the ATN are augmented with rules containing conditions and actions. This section presents an alternative implementation of this morphological analyzer that more clearly illustrates the potential of extending Prolog with TCGs.

4.1 The Augmented Transition Network (ATN)

An ATN, which describes the relation of an Arabic stem and additions, is built according to Fig. 2. The ATN consists of arcs. Each of which is a link from a departure node to a destination node, called states.

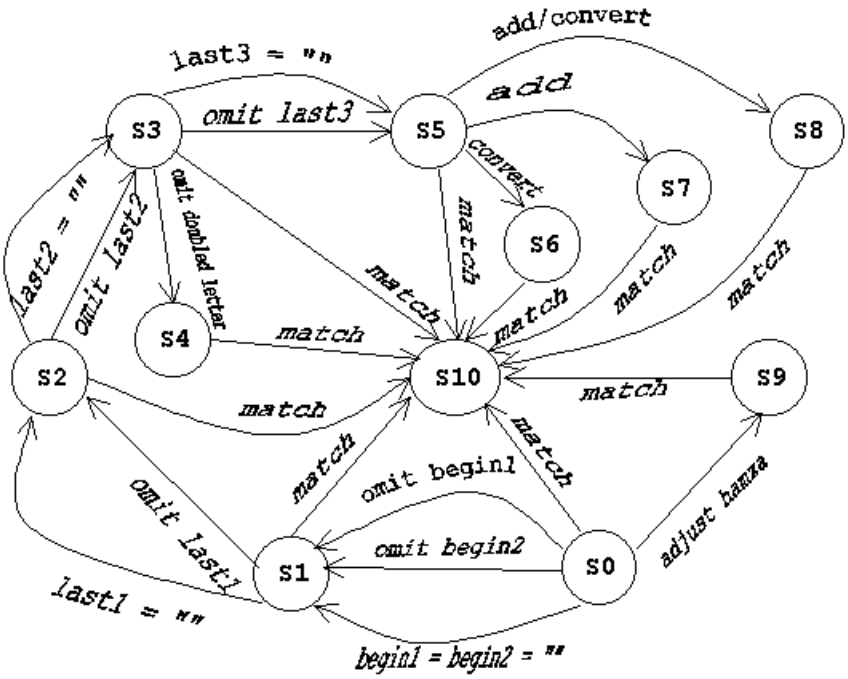

Fig. 2 ATN representing the relation between the additions and stem of an inflected Arabic word.

The implementation of the ATN is shown in Fig. 3. Each arc is associated with a set of registers. Five registers that correspond to the additions as classified in [7]. The sixth register is reserved for storage of the current assumed stem. For the sake of simplicity, in this article we will not include the implementation of the status flags that indicate the modifications made to the stem. The arcs contain conditions and actions that are implemented in Prolog as rules. As we traverse the ATN, the word is scanned for different additions, and what remains is the stem. This means that these registers are set to values by actions associated with each arc. Our program handles this, thanks to the ability of threaded variables to couple input-output arguments. Thus, a threaded variable is included for each register in order to collect this information.

```
arc(0,10)-B1R-B2R-L1R-L2R-L3R-Word:-
    match(Word).
arc(0,9)-B1R-B2R-L1R-L2R-L3R-Word:-
    adjust_hamza-Word.
arc(0,1)-B1R-B2R-L1R-L2R-L3R-Word:-
    omit_begin1(B1R)-Word.
arc(0,1)-B1R-B2R-L1R-L2R-L3R-Word:-
    omit_begin2(B2R)-Word.
arc(0,1)-B1R-B2R-L1R-L2R-L3R-Word.
arc(1,10)-B1R-B2R-L1R-L2R-L3R-Word:-
    match(Word).
arc(1,2)-B1R-B2R-L1R-L2R-L3R-Word:-
    omit_last1(L1R)-Word.
arc(1,2)-B1R-B2R-L1R-L2R-L3R-Word.
arc(2,10)-B1R-B2R-L1R-L2R-L3R-Word:-
    match(Word).
arc(2,3)-B1R-B2R-L1R-L2R-L3R-Word:-
    omit_last2(L2R)-Word.
arc(2,3)-B1R-B2R-L1R-L2R-L3R-Word.
arc(3,10)-B1R-B2R-L1R-L2R-L3R-Word:-
    match(Word).
```

```
arc(3,4)-B1R-B2R-L1R-L2R-L3R-Word:-
    omit_last_doubled(L2R)-Word.
arc(3,5)-B1R-B2R-L1R-L2R-L3R-Word:-
    omit_last3(L1R,L3R)-Word.
arc(3,5)-B1R-B2R-L1R-L2R-L3R-Word.
arc(4,10)-B1R-B2R-L1R-L2R-L3R-Word:-
    match(Word).
arc(5,10)-B1R-B2R-L1R-L2R-L3R-Word:-
    match(Word).
arc(5,6)-B1R-B2R-L1R-L2R-L3R-Word:-
    convert(B2R,L1R,L2R,L3R)-Word.
arc(5,7)-B1R-B2R-L1R-L2R-L3R-Word:-
    add(B2R,L1R,L2R,L3R)-Word.
arc(5,8)-B1R-B2R-L1R-L2R-L3R-Word:-
    add_or_convert(B2R)-Word.
arc(6,10)-B1R-B2R-L1R-L2R-L3R-Word:-
    match(Word).
arc(7,10)-B1R-B2R-L1R-L2R-L3R-Word:-
    match(Word).
arc(8,10)-B1R-B2R-L1R-L2R-L3R-Word:-
    match(Word).
arc(9,10)-B1R-B2R-L1R-L2R-L3R-Word:-
    match(Word).
```

Fig. 3 An implementation of ATN.

4.2 Traversing the ATN

The lexical analyzer traverses the ATN in a depth-first manner and backtracks to search exhaustively for all possible solutions, see Fig. 4.

```
recognize(Node)-B1R-B2R-L1R-L2R-L3R-Word:-
    final(Node).
recognize(Node)-B1R-B2R-L1R-L2R-L3R-Word:-
    arc-Node-B1R-B2R-L1R-L2R-L3R-Word,
    recognize(Node)-B1R-B2R-L1R-L2R-L3R-Word.
```

Fig. 4 Exhaustive depth first search of ATN.

A possible solution of the inflected word will fill the registers according to interpretation made by the lexical analysis. In order to keep all interpretations, we can use the built-in predicate, like `findall`, `setof` or `bagof`, that enables us to construct all the solutions of a Prolog goal in a list. Then, it can be used as the interface between the lexical-analysis phase and the subsequent phases.

5. Conclusions

This paper has been concentrated on issues in the design and implementation of a general notational extension to Prolog programs, called threaded clause grammars (TCGs). TCGs greatly increase the readability and maintainability of natural language analysis programs by allowing concision expression of definitions and avoiding problems associated with having definitions with many arguments. Furthermore, this formalism provides a syntactic convention that offers a consistent style of language analysis, as a wide range of language analyzers, e.g. morphological analyzers and parsers, can be expressed using TCGs notations. We have demonstrated this capabilities in terms of an example, an Arabic morphological analyzer. TCGs were implemented on top of SICStus Prolog. The semantics is specified at a very abstract

level in terms of transformation rules that translate TCGs programs into Prolog programs. Extending standard Prolog by TCGs can be very well incorporated in the future for better natural language analysis.

References

1. Allen, J. *Natural Language Understanding*, second edition, The Benjamin/Cummings Publishing Company, 1995.
2. Clocksin W. and Mellish C., *Programming in Prolog*, Springer-Verlag, 1981.
3. Dahl V., Tarau P., and Li R. Assumption Grammars for Processing Natural Language, *In proceedings of the fourteenth international conference on logic programming*, MIT press, pp. 256-269, 1997.
4. Dougherty R. *Natural Language Computing: An English Generative Grammar in Prolog*, Lawrence Erlbaum Associates, Inc., NJ, 1994.
5. Gazdar G. and Mellish C., *Natural Language Processing in Prolog: An Introduction to Computational Linguistics*, Addison-Wesley, 1990.
6. Pereira F., Shieber C, and David H, Definite Clause Grammars for language analysis–A Survey of the Formalism and a Comparison with Augmented Transition Networks, *In Readings in Natural Language Processing*, Grosz, B., Jones K., Webber B. (Eds.), pp. 24-101, Morgan Kuffmann, 1986.
7. Rafea A. and Shaalan K., Lexical Analysis of Inflected Arabic words using Exhaustive Search of an Augmented Transition Network, *Software Practice and Experience*, Vol. 23(6), pp. 567-588, John Wiley & sons, U.K., June 1993.
8. Schachte P., Global Variables in Logic Programming, *In proceedings of the fourteenth international conference on logic programming*, MIT press, pp. 3-17, 1997.
9. Sterling, L. and Shapiro E., *The Art of Prolog: Advanced Programming Techniques*, second edition, MIT Press, 1994.
10. Tarau P., Dahl V., and Fall A., Backtrackable State with Linear Assumptions, Continuations and Hidden Accumulators Grammars, Technical Report 95-2, Dèpartment d'Informatique, Université de Moncton, April 1995.
11. Van Roy P., A Useful Extension to Prolog's Definite Clause Grammar Notation, *ACM SIGPLAN Notices*, Vol. 24, No. 11, pp. 132-134, Nov. 1989.
12. Warren D. *Da*tabase updates in pure Prolog. *In Fifth Generation Computer Systems*, pp. 244-253, ICOT, 1984.
13. Woods W., Transition network grammar for natural language analysis, *comm. ACM*, Vol. 10, pp. 591-66, 1970.