GPC – a tool for higher-order grammar specification

José João Dias de Almeida
Department of Computer Science
University of Minho
Campus de Gualtar
4700 Braga-Portugal
jj@di.uminho.pt

Abstract

In this text we describe GPC , a higher-order grammar specification prototype environment with automatic semantic rewriting implemented in prolog in a DCG-like way.

In GPC every non terminal symbol (or expression) have a synthesized semantic attribute (that can be calculated by a function (predicate) t_value) and can have other attributes. In recognition time, after any reduction, synthesized attribute rewriting of the lhs is activated.

In higher-order grammar specification design, the questions of semantic definitions are sometimes forgotten or difficult to do.

GPC meta-operator can take synthesized semantic attribute in account so that they can also take semantic decisions.

Some examples are presented along the text.

Contents

1 Introduction

GPC is a tool for higher-order grammar specification with automatic semantic rewriting, implemented in prolog, allowing definition of grammar styles, with general frame-works to be completed by parameterization thus revealing the underlying structures of language.

In the following we'll present some design features and some examples of higher-order specification of grammars in the GPC grammar prototyping environment. The higher-order specification of grammars has an extra problem coming from the fact that grammar objects are usually described as 2 level elements (the syntax and the semantics). This makes parameterization more complex.

In this text we'll use a DCG/prolog-like notations[7] in some example.

We'll assume that the reader is familiar with context free grammar concepts and notations[3], logic grammars [2, 4, 5, 1], DCG and prolog.

GPC extends the DCG's terminal symbols in many ways; in the text the notation `[Class-Sem]` means a terminal symbol with class "Class" and value "Sem" (ex: `[int-S]` to get any integer and collect is value in S).

We'll begin by describing the general semantic mechanism used in GPC ; after that we'll introduce some simple meta-operators and their semantic dealing versions. In next section we'll introduce some more realistic examples and some brief reflections about the definition of the syntax of grammar syntax.

2 Semantic rewriting

In GPC semantics is describe by a particular attribute (the semantic attribute) present in every non terminal(NT) symbols, and some terminal(T) symbols. In addition we can have other auxiliar attributes associated.

When we write a production rule:

```
a(f(S1,S2,Sn)) --->
              a1(S1),a2(S2),an(Sn)
```

internally we have:

```
a(Se)) ---> a1(S1), a2(S2), an(Sn),
         {rewrite(f(S1,S2,Sn),Se)}.
```

This means that before any reduction, we have a semantic rewriting step which allows us simplify, normalize and validate semantics before going any further.

Rewriting operators are controlled by the rewriting rules set which can also be conditional and contextual.

Alternatively semantics non-rewriting production rule definition is provided with the following syntax:

```
a(f(S1,S2,Sn)) -+->
                a1(S1),a2(S2),an(Sn)
```

This allows the definition of some meta-operators with special semantic evaluation.

Attributes can be manipulated in any of usual ways of logic grammars.

Example 2.1

Suppose that we are writing a compiler and we have a rule:

```
int(add(X,Y))--->int(X),"+",int(Y).
int(X)       --->[integer-X].
int(v(X))    --->[var-X].
```

which build a tree with integer or variables operands.

We can add some rewriting rules to simplify some parts of the tree and evaluate them when there are no variables involved:

```
add(X,Y) ==> Z :-
    number(X),number(Y), Z is X+Y.
add(0,X) ==> X.
add(X,0) ==> X.
```

In this way, the semantics of `1+3+a+0`
would be `add(4,a)` instead of
`add(add(add(1,3),a),0)`.

Rewriting rules are meant to insert a mechanism to product a normal form for the semantics, and can also be used to translate semantic representations for pragmatic reasons.

Rewriting rules can also be seen as production rules of type 0 grammars in Chomsky hierarchy which, for complexity reasons, can only be used in the semantic representation level.

3 Meta-Operators

In grammar specifications the use of Backus Naur form(BNF) is very popular. The extended Backus Naur form adds some meta-operator (optional and repetitive operators) to BNF and it is a more comfortable way for grammar specifications.

We can see the meta-operators as examples of functions over languages:

```
* : Lang -> Lang
*(A) --> []
      |  A *(A)
```

In traditional grammar descriptions we can not define new meta-operators; in fact we can't write productions that have NT symbols with symbols(T or NT) as parameters.

We will present some examples of the use of GPC to define meta-operators. In example 3.1 we have the case of meta-operators ?, *, permutation, frequently available as a predefined operator, treated without semantics; in example 3.2 we treat with GPC the same examples using semantics. In example 3.3 we have the case of a function with arguments that describe other arguments ("printf") that is tradicionaly treated in runtime and here expressed in terms of a higher-order grammar. The example 3.4 shows the use of meta-operators do simplify the description of expression by providing a basic framework to be completed by parameters.

Note that even if you find meta-operators difficult to write, you have to do it only once.

Example 3.1: ?, *, and permutation

The very well known meta-operators `?` and `*` are not possible in traditional parsers generators definitions nor in DCG[1].

In GPC we can define

```
; optional operator
?(X) --> X | []
```

or

```
; Kleene operator
*(X) --> X , *(X) | []
```

The previous `?(X)` elegant example is useless when X has semantics because it is not defined what the semantics of `?(X)` will be specially when X is absent.

Similar questions arise with *(X).

The simple (but useful) operator permutation. p(X1,...,Xn) means X1...Xn in any order. In GPC we can define

[1] in some implementations they are partially possible [6]

```
; permutation of 2 elements
p(x,y)   -+-> X,Y | Y,X

; permutation of 3 elements
p(X,Y,Z) -+-> X,p(Y,Z)
            | Y,p(X,Z)
            | Z,p(X,Y)
```

In this case there is no semantic problems.

Example 3.2: ? and * with semantics

In order to deal with semantics, we have to introduce a predicates(t_value) to separate the symbols from their semantic values and to construct terms with NT symbols and its semantics.

```
t_value(Symbol,Semantic,Term)
Term = Symbol - Semantic     (1)
Symbol - Semantic            (2)
```

1. used in the rhs of production rules to separate a term in its symbol and semantics.

2. used in production rules to build a term from its symbol and semantics

By default the semantic value of a term is the first attribute [2].
The optional operator could be written in the following way (decision: when absent, `semantics(?X) = nil`):

```
?X --> X=F-Se,
   ( X ;     /* we find X        */
    {Se=nil})/* no X; Sem. <- nil*/
```

The repetitive operator could be written (decision: `semantics(*X) is a LIST of semantics(X)`):

```
*X --> X=F-Se,
       ( {Se=[]};
         {Se=[H|T]}, F-H, *F-T)
```

Example 3.3: printf

In this example we'll describe a subset of "printf"-like C functions. "printf" has a first argument that is a string and that (between other things) describes the rest of the arguments (argument type list).

```
printf("%d%s%d\n", a, b, c)
```

[2] this can be redefined by val_meta predicate

The format `%d%s%d\n` would force the existence of 2 integer arguments a and c (`%d`) and a string value b (`%s`).

n GPC we'll begin by defining a meta-operator "list of X" `list(Val,X)` that accepts a list of symbols X (T or NT) and gives the list of their values Val; every 2 symbols will be separated by ","[3].

```
list([],[])          -+-> [].
list([S],[H])        -+-> H=F-S, H.
list([S|St],[H1,H2|T] -+->
      H1=F-S,H1,"," ,list(St,[H2|T]).
```

This way, if we have intexp (= integer expression) and strexp NT symbols we could "apply"
`list(S,[intexp(),strexp(),intexp()])`
to the string `' 1+2 , "example", 3" '`
to obtain something like `S=[add(1,2), "example", 3]`

The "printf" definition can be:

```
printf_exp(pf(F,L)) --->
    "printf(",
    strexp(F),  % get the format
    { extract_arg_list(F,La)},
    list(L,La), ")". % get arguments
```

where extract_arg_list is the extracting argument list predicate.

Example 3.4: expressions

We now present some useful simple examples related to expression definition for usual programming languages.

Generally we can say that a simple expression of type T can be:

- a valued terminal of that type (if possible) : `[T-S]`
- a parenthised expression of the type `"(" exp(T,S) ")"`
- a function call of a function returning T type
- a variable

So we can write a "sim_exp" meta-operator depending on the existence of a `exp var` symbols and a signature predicate:

```
 sim_exp(T,S) -+-> [T-S]          (1)
    | [var-S]                     (2)
    | "(" exp(T,S) ")"            (3)
```

[3] it is trivial to generalize the example to a generic separator that would be given as a parameter

```
   | [var-Fu],                  (4)
       {sig(Fu,T,Args,S1)},     (5)
       "(", list(Sl,Args), ")",(6)
       { S1 =*> S}.             (7)
```

1. valued terminal symbol of type T
2. variable
3. expression of type T between parenthesis
4. function ,
5. signature `Fu:Args --> T` and semantics S1 of the function
   Ex.: `sig(fact,int,[int-S],f(S)),`
6. meta-operator list to get the arguments,
7. semantic rewriting of S1

This meta-operator can be used in the definition of expression:

```
exp(S,T)  ---> i(exp(S,T,4),
        left(("/\"-and;"\/"-or))).
exp(S,T,4)---> i(exp(S,T,3),
        noass(("=="-eq;"!"-neq))).
exp(S,T,3)---> i(exp(S,T,2),
        left(("+"-add;"-"-sub))).
exp(S,T,2)---> i(exp(S,T,1),
        left(("*"-mul;"%"-rem))).
exp(S,T,1)---> i(exp(S,T,0),
        righ("^"-exp)).
exp(S,T,0)---> simp_exp(T,S).
```

The i meta-operator (whose code will not be comented) takes a list of expressions argument1 separated by the infix operators whose associativity and name are described.

```
i(Ex,Op)          -+-> Ex.
i(exp(SE,T,P),OP) -+-> exp(S1,T1,P),
            ++(OP,T1-S1,exp(SE,T,P)).

++(left(O-N),P,exp(SF,T1,Pr)) -+-> O,
     {sig(N,T,[P,T2-S2],Se)},
     exp(S2,T2,Pr),
     {Se=*>S},
     (  {SF=S,T=T1} ;
        ++(left(O),T-S,exp(SF,T1,Pr))).
```

The meta-operator also verifies the signatures and builds the parsing trees.

It is simple to implement a meta-operator to deal with generic operator-grammar definitions [3].

The complete code of the examples is available (contact author's email).

4 Redefinition of syntax definition syntax

In general we'll use meta-operators to define a style of grammar definition : the general meta-operators can be grouped in a style file and "included" before the grammar definition text. In order to obtain a more perfect (and ergonomic) product it is sometimes useful to add/remove some details of style.

A simple production like

```
intexp(add(X,Y)) -->
     intexp(X),"+",intexp(Y)
```

would be fine for someone coming from prolog-world but something like

```
add:intexp x intexp-->intexp
       using infix"+"
```

would be more natural for someone coming from algebraic world...

To deal with this question GPC has a programmable context dependent transformation mechanism that make it easy.

5 Conclusions and future work

We hope that this work can show the utility of GPC in the definition of general grammar frameworks and styles, and the possibility of taking semantic decisions in meta-operators.

This work is also intended to be a contribution to the development of higher-order grammar specification which have not been widely use until now and that will be a step to understand languages

The application of GPC to natural languages is also being done.

We are planning to implement GPC also with a flexible parser generator tool like PCCTS/sourcerer[**?**]

References

[1] Harvey Abramson. Metarules and an aproach to conjunction in definite clause translation grammars: Some aspects of gramatical metaprogramming. In R. Kowalski and K.Bowen, editors, *Proc.88 – Int. Symp. on Logic Programming*, pages 233–247. The MIT Press, 1988.

[2] Harvey Abramson and Veronica Dahl. *Logic Grammars*. Symbolic Computation - Artificial Intelligence. Springer-Verlag, 1989.

[3] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers Principles, Techniques and Tools*. Addison-Wesley, 1986.

[4] H. Lewis Chau. Narrowing grammars: A comparison with other logic grammars. In Ewing L. Lusk and Ross A. Overbeek, editors, *Proceedings of the North American Conference on Logic Programing*. The MIT Press, 1989.

[5] H. Lewis Chau and D.Stott Parker. Narrowing grammars: Theory implementation, and applications. *The Journal of Logic Programming*, 14:253–286, 1992.

[6] R.A. O'Keefe. *The Craft of Prolog*. Logic Programming. The MIT Press, 1990.

[7] F. C. N. Pereira and D. H. D. Warren. Definite clause grammars for language analysis: A survey of the formalism and comparison with augmented transistion networks. *Artificial Intelligence*, 13(3):231–278, 1980.

A Ac: Introduction

A.1 GPC a short overview

- prototyping environment
- automatic semantic rewriting
- logic grammar
- higher-order grammar specification:
  - symbols with parameters T,NT,...
  - expressions that produce NT
  - frame-works
- allowing definition of grammar styles
- implemented in prolog

A.2 Ac: Motivation

- abstraction
- understand underlying structures of language
- specification of basic constructors

- grammar reuse

Higher-order specification of grammars:

- symbols are not atomic
  - syntactic aspects
  - semantic aspects
  - ...
- meta-operator definition needs constructors and selectors of NT and T symbols

A.3 Ac: Semantic rewriting

```
a(f(S1,S2,Sn)) ---> a1(S1),a2(S2),an(Sn)
```

internally we have:

```
a(Se)) ---> a1(S1), a2(S2), an(Sn),
         {rewrite(f(S1,S2,Sn),Se)}.
```

a rewriting rule:

```
termo1 ==> termo2
```

motivation:

- simplify semantics as soon as possible
- semantic normal form
- rewrite in "bottom" change the way

Example: build a tree with integer or variables:

```
int(add(X,Y))--->int(X),"+",int(Y).
int(X)       --->[integer-X].
int(v(X))    --->[var-X].
```

```
[[1+3+a+0]] = add(add(add(1,3),v(a)),0)
```

rewriting rules to:

- simplify parts of the tree
- evaluate when no variables involved:

```
add(X,Y) ==> Z :-
    number(X),number(Y), Z is X+Y.
add(0,X) ==> X.
add(X,0) ==> X.
```

```
[[1+3+a+0]] = add(4,v(a))
```

B Ac: Examples

B.1 Example: permutation of...

```
; definition:
; permutation of 2 elements
p(X,Y)   -+-> X,Y | Y,X

; permutation of 3 elements
p(X,Y,Z) -+-> X,p(Y,Z)
            | Y,p(X,Z)
            | Z,p(X,Y)


; teste:
sent( s(N,V,C)) ---> p(np(N),vp(V),adv(V))

  [[Joao cantou ontem]] =
  [[cantou Joao ontem]] =
  [[ontem Joao cantou]] =
     s(Joao',cantar',ontem')
```

B.2 Ac: Example: optional and star

```
; definition 1
; optional operator
?(X) --> X | []

; Kleene operator
*(X) --> X , *(X) | []
```

Elegant but ...useless definitions:

- it is not defined semantics!!!

```
?int(S)    [[ ]] = ?
*int(S)    [[ 1 2 3 ]] = ?
```

Ac: Semantic constructors and selectors

- by default, the semantic values is the first attribute
- `[term-value]`
  - terminal of class `term`
  - semantics `value`

- predicates do separate symbols from their semantic values:

```
t_value(Symbol,Semantic,Term)
Term = Symbol - Semantic     (1)
Symbol - Semantic            (2)
```

(1) to separate a term in symbol and semantics
(2) to build a term from its symbol and semantics

B.3 Ac: Semantics in ? and *

```
; definition 2
; optional operator [[ ]] = nil

?X --> X=F-Se,          let X: symbol F semantic Se
       ( X ;            we find X                 or
         {Se=nil})      no X: Sem <- nil

; Kleene operator
; semantics: [[ x,y,...]] = [ [[x]], [[y]],...]

*X --> X=F-Se,
       ( {Se=[]};
         {Se=[H|T]}, F-H, *F-T)

;test
;
```

B.4 Ac: Examples: printf

```
printf("%d%s%d\n", a, b, c)
            a-int b-str c-int
```

Meta-operator "list of X" `list(Val,Xl)`:

- accepts a list of symbols X
- gives the list of their values Val
- separator ","

```
list([],[])           -+-> [].
list([S],[H])         -+-> H=F-S, H.
list([S|St],[H1,H2|T] -+-> H1=F-S, H1, "," , list(St,[H2|T]).

;teste:
list(S,[intexp(_),strexp(_),intexp(_)])
```

```
1+2 , "example", 3

S= [add(1,2), "example", 3]
```

The "printf" definition can be:

```
printf_exp(pf(F,L)) --->
    "printf","(",
    strexp(F),                  get the format
    { extract_arg_list(F,La)},
    list(L,La), ")".            get arguments
```

B.5 Ac: Example simple expressions

simple expression of type T can be:
(1) valued terminal symbol of type T
(2) variable
(3) "(" expression of type T ")"
(4) function of type T (and the write arguments)

```
 sim_exp(T,S) -+-> [T-S]         (1)
    | [var-S]                    (2)
    | "(" exp(T,S) ")"           (3)
    | [var-Fu],                  (4)
         {sig(Fu,T,Args,S1)},    (5)
         "(", list(_,Args), ")", (6)
         { S1 =*> S}.            (7)
```

(5) Ex.: `sig(fact,int,[int-S],f(S))`,
signature `Fu:Args --> T` and semantics S1 of the function
(6) meta-operator to get the arguments,
(7) semantic rewriting of S1

```
 sim_exp(int,S) applied to
 fact("a")              error
 fact(2,3)              error
 fact(1+2)              f(3)
```

B.6 Ac: Example: expression

```
exp(S,T)  ---> i(exp(S,T,4), left(("/\"-and; ...))).
exp(S,T,4)---> i(exp(S,T,3), noass(("=="-eq; ...))).
exp(S,T,3)---> i(exp(S,T,2), left(("+"-add; ...))).
exp(S,T,2)---> i(exp(S,T,1), left(("*"-mul; ...))).
exp(S,T,1)---> i(exp(S,T,0), righ("^"-exp)).

exp(S,T,0)---> sim_exp(T,S).
```

where **i** is a infix meta-operator to:

- get a list of expressions (arg1)
- separated by infix operator(arg2):
  - with a associativity and
  - name
- verifies the signatures
- builds the parsing tree

```
i(Ex,Op)           -+-> Ex.
i(exp(SE,T,P),OP)  -+-> exp(S1,T1,P),
                        ++(OP,T1-S1,exp(SE,T,P)).

++(left(O-N),P,exp(SF,T1,Pr)) -+-> O,
         {sig(N,T,[P,T2-S2],Se)},
         exp(S2,T2,Pr),
         {Se=*>S},
         (  {SF=S,T=T1} ;
            ++(left(O),T-S,exp(SF,T1,Pr))).
```