STUDIA UNIV. BABEŞ–BOLYAI, INFORMATICA, Volume **LV**, Number 1, 2010

DCGS FOR PARSING AND ERROR HANDLING

TIBOR ÁSVÁNYI

Abstract. The available texts on standard Definite Clause Grammars (DCGs) are just about analysing correct input texts and generating the appropriate outputs. Therefore, their example programs simply say *no* on texts containing syntactical faults.

In this paper we concentrate on the parsing problem, handling the possible formal deficiencies of the input texts, using standard DCGs (shortly: DCGs). We describe a methodology, in order to develop DCGs parsing input texts while giving appropriate messages, if syntax errors are found in them. In our approach, we give the semantics of the correct input texts, and the semantics of the erroneous input texts together.

In this paper we suppose that the reader is familiar with the Prolog language, especially with standard DCGs, and has some practical knowledge, how to write readable and effective programs in Prolog.

1. Introduction

In the logic programming community, logic grammars are widely used while solving compiler writing or natural language processing problems. Some logic granmmars (like DCGs [5]) support top-down, others (like AID [2]) support bottom-up parsing of the input. Some of them support deterministic parsing and have built-in facilities to handle the deficiencies of the input [4], others (like DCGs) support indeterministic parsing and do not have built-ins to handle errors.

Received by the editors: 15. February 2011.

2010 *Mathematics Subject Classification.* 94-02, 94-04.

1998 *CR Categories and Descriptors.* D. Software [**D.1 PROGRAMMING TECHNIQUES**]: D.1.6 Logic Programming – *Logic Grammars*; F. Theory of Computation [**F.3 LOGICS AND MEANINGS OF PROGRAMS**]: F.3.2 Semantics of Programming Languages – *Program analysis*; F. Theory of Computation [**F.4 MATHEMATICAL LOGIC AND FORMAL LANGUAGES**]: F.4.2 Grammars and Other Rewriting Systems – *Parsing*.

Key words and phrases. logic grammar, Definite Clause Grammar, DCG, parsing, context-free description, context-dependent information, error hadling, error semantics, full semantics.

DCGs [5] are a popular extension of the logic programming language Prolog [1]. It adds the power of Prolog to context-free grammars. This means that DCGs provide top-down, left-to-right parsing, in accordance with the backtracking mechanism of Prolog [6].

Through their parameters, the rules may pass structural, context-dependent, and semantic information. Ambiguities and different interpretations of input texts can be expressed in an elegant and natural way. DCGs can be used for analysing and/or for generating sentences. Therefore, translators and compilers can be implemented without unnecessary efforts: The parser DCG may build an inner representation of the input, and the generator DCG may produce the required output.

According to our interest, we develop a method, in order to write parsers handling correct and erroneous input texts as well. For case of simplicity, the semantics of both kinds of input texts will be defined by Prolog terms.
The semantics of the erroneous input texts will be called *error semantics*.

We suppose that we are over the tokenizing process, and our input text is represented by a proper list [3] of lexical elements (tokens) [7].

Now, the main phases of constructing a parser are the following:

(1) Describe the set of correct input texts using a DCG.
(2) Add cuts in a safe manner – "as early as possible" –, exactly into the positions where it is already known that the containing DCG rule is responsible for analysing the actual part of the input.
(3) Add default rules for error handling and error recovery.

The first "to do" – describing the set of correct input texts – has the following steps:

(1) Give a partial description by a context-free grammar.
(2) Add context-dependent, syntactical information.
(3) Add semantical information.

We can add the cuts during this process when it is convenient; but we add the rules for error handling normally after.

We will present our method through a small example, and structure this paper according to the phases and steps of completing the parser. (The parser was tested in SICStus Prolog 4.1.3. [8])

2. The example language

Let us have a series of definitions of integer constants. The later may depend on the earlier. Each constant is defined by a simple binary operation

depending on two predefined (symbolic or literal) constants. The possible operations are: addition, substraction, multiplication, integer divison, modulus, *that is* `+,-,*,//,mod`. Each constant definition is terminated by a dot.
For example:

```
a   = 1+1.
a_1 = a-1.
b   = 3*a.
c3  = b//a.       %% integer division
d   = b mod 4.

u = u*4.    %% u depends on undefined constant (not allowed)
d = a+1.    %% d is redefined constant (not allowed)
```

3. Describing correct inputs

In this section we detail the first "to do" outlined in the Inroduction. First we give a partial description of our example language by a context-free grammar. Next we extend this description with the context-dependent, syntactical properties. At last we add the appropriate semantical information.

3.1. **Partial, context-free desription.** First we give a context-free desription. In this step, we are unaware of the context dependent properties: undefined and redefined constants are still allowed.

```
constants --> constdef, !, constants.
constants --> [].

constdef --> const, [=], expr, ['.'].
expr --> operand, operator, operand.

operator --> [Op], { op(Op) }.
op(+).    op(-).    op(*).    op(//).    op(mod).

operand --> [Y], { integer(Y), ! }.
operand -->  id.

const --> id.
id --> [Id], {is_id(Id)}.

is_id(Id) :-
    atom(Id), atom_codes(Id,[C1|Cs]),
    lower_case_letter_code(C1),
    \+ ( member(K,Cs), \+id_code(K) ).
```

```
id_code(K) :-
    ( lower_case_letter_code(K) -> true
    ; upper_case_letter_code(K) -> true
    ; digit_code(K) -> true
    ; K == 0'_
    ).

lower_case_letter_code(K) :- 0'a =< K, K =< 0'z.
upper_case_letter_code(K) :- 0'A =< K, K =< 0'Z.
digit_code(K) :- 0'0 =< K, K =< 0'9.
```

One can see, that the necessary cuts – in the "as early as possible" style – have already been inserted.

3.2. **Adding context-dependent, syntactical information.** In order to handle undefined and redefined constants, we have to store the list of constant names defined up till now – in a parameter:

```
constants --> constants([]).

% Bs is the list of constant identifiers defined up till now.
% C is the identifier of the first constant described
% by this very rule:
constants(Bs) --> constdef(Bs,C), !, constants([C|Bs]).
constants(_Bs) --> [].

constdef(Bs,Id) --> const(Bs,Id), [=], expr(Bs), ['.'].

expr(Bs), ['.'] --> operand(Bs), operator, operand(Bs), ['.'].

operator --> [Op], { op(Op) }.
op(+).    op(-).    op(*).    op(//).    op(mod).

operand(_Bs) --> [Y], { integer(Y), ! }.
% If an operand of an expression is an id,
% it must have been defined before:
operand(Bs) -->  id(Y), { member(Y,Bs) -> true }.

% The constant should not have been defined before:
const(Bs,Id) --> id(Id), { \+member(Id,Bs) }.

id(Id) --> [Id], {is_id(Id)}.  % Predicate is_id/1 is unchanged.
```

The grammar still simply says `yes`, if the series of constant definitions is syntactically correct, or `no` otherwise.

3.3. **Adding semantical information.** The semantics will be a proper list of equalities of the form `Id=Value`, that is, the values of the different constants are given – in the original order:

```
% Constants is the list representing the semantics.
constants(Constants) -->
    constants([],Cs), {reverse(Cs,Constants)}.

:- use_module( library(lists), [reverse/2] ).


% List Bs represents the semantics of the constants
% defined up to this point -- in reversed order, and
% C is the semantics of the actual constant.
constants(Bs,Cs) --> constdef(Bs,C), !, constants([C|Bs],Cs).
constants(Bs,Bs) --> [].

constdef(Bs,Id=Val) -->
    const(Bs,Id), [=], expr(Bs,Expr), ['.'],
    { catch( Val is Expr, _, fail ) }.
    % In case of zero divisor or arithmetic overflow,
    % this rule must fail.


expr(Bs,Expr), ['.'] -->
    operand(Bs,X), operator(Op), operand(Bs,Y), ['.'],
    { Expr =.. [Op,X,Y] }.
% X and Y are the values of the operands.


operator(Op) --> [Op], { op(Op) }.
op(+).    op(-).    op(*).    op(//).    op(mod).

operand(_Bs,X) --> [Y], { integer(Y), !, X = Y }.
operand(Bs,X) -->  id(Y), { member(Y=Z,Bs) -> X = Z }.


const(Bs,Id) --> id(Id), { \+member(Id=_Z,Bs) }.
```

```
id(Id) --> [Id], {is_id(Id)}.  % Predicate is_id/1 is unchanged.
```

Using this DCG, we can receive the semantics of a series of constants, that is, the values of those constants, for example:

```
| ?- _Tokens = [a,=,1,+,1,., a_1,=,a,-,1,., b,=,3,*,a,.,
                c3,=,b,//,a,., d,=,b,mod,4,.],
     constants(Defs,_Tokens,[]).
Defs = [ a=2, a_1=1, b=6, c3=3, d=2 ]
```

But, if there is some error, we receive not much information:

```
| ?- _Tokens = [a,=,1,+,a,.], constants(Defs,_Tokens,[]).
no
```

Using our method of step-by-step refinement of the DCG, we arrived at the point where the known texts about standard DCGs stop. We believe, this method have made the way easier.

4. Adding error semantics

At last we add the error alternatives, that is, rules for error handling.

In general, if L is a language over an alphabet, then the *error semantics* is the semantics of the complementer of L. We propose that the error semantics should be represented by Prolog terms following a notational convention developed for L. We will call them *error terms*.

In our example, the semantics of a list of input tokens is a list of equalities, and of error terms. If a correct constant definition is found, then its semantics is an equality of the form learnt in the previous section. If no more constant definition is found, but the appropriate token list is still nonempty, then the last element of the list representing the semantics have this form: `error(constants,RemainingTokens)`.

In general, an error term has the form `error(Id,Context)`: `Id` refers to the type and level of the error, and `Context` refers to the textual context of the error.

If an erroneous constant definition is found, then `Id` refers to the source(s) of the error(s). Provided that the equality can be recognised on the token list, `Context` is an equality, where the correct components are given as usually, but the erroneous components are given with error terms. If the equality cannot be recognised on the token list, `Context` is just a token list, closed by the dot sign finishing the constant definition.

```
constants(Constants) -->
    constants([],Cs), {reverse(Cs,Constants)}.

:- use_module( library(lists), [reverse/2] ).

constants(Bs,Cs) --> constdef(Bs,C), !, constants([C|Bs],Cs).
constants(Bs,Cs) -->
    nonempty(Tokens), !, {Cs=[error(constants,Tokens)|Bs]}.
constants(Bs,Bs) --> [].

nonempty([T|Ts]) --> [T], anything(Ts).
anything([T|Ts]) --> [T], !, anything(Ts).
anything([]) --> [].
```

In `constdef(Bs,C)` below, the condition `atom(Id)` means that `Id` is not an error term, that is, it can be defined as a constant. If `Expr` contains an error term, its evaluation raises an exception. [Otherwise numerical errors (zero divisor, arithmetic overflow) are still possible, but here is only the general `error(expression,Id=Expr)` notation for them, in order to reduce complexity.]

```
constdef(Bs,C) -->
    const(Bs,Id), [=], expr(Bs,Expr), ['.'], !,
    { atom(Id), catch( Val is Expr, _, fail ) -> C = (Id=Val)
    ; atom(Id) -> C = error(expression,Id=Expr)
    ; catch( Val is Expr, _, fail ) ->
          C = error(constant,Id=Val)
    ; C = error(constant_and_expression,Id=Expr)
    }.

% Basic problem with the constant definition:
constdef(_Bs,error(constdef_syntax,Ts)) -->
    constdef_error(Ts).

constdef_error(['.']) --> ['.'], !.
constdef_error([X|Xs]) --> [X], constdef_error(Xs).
```

As usually, at the level of `expr(Bs,Expr)`, only the basic structure of the exression is checked:

```
expr(Bs,Expr), ['.'] -->
    operand(Bs,X), operator(Op), operand(Bs,Y), ['.'], !,
    { atom(Op) -> Expr =.. [Op,X,Y]  % Op is a legal operator
    ; Expr = [X,Op,Y]            % Op is not a legal operator
    }
% Basic problem with the struct of the expression:
expr(_Bs,error(expression_syntax,Ts)) -->
    expression_error(Ts).

expression_error([]), ['.']   --> ['.'], !.
expression_error([X|Xs]) --> [X], !, expression_error(Xs).
expression_error([]) --> [].

operator(Operator) --> [Op],
    { op(Op) -> Operator = Op
    ; Operator = error(operator,Op)
    }.

op(+).    op(-).    op(*).      op(//).    op(mod).

operand(_Bs,X) --> [Y], { integer(Y), !, X = Y }.
operand(Bs,X) -->  id(Y), !,
    { member(Y=Z,Bs) -> X = Z
    ; X = error(undefined,Y)
    }.
operand(_Bs,X) --> [Y], { X = error(operand_syntax,Y) }.
```

At last we describe the analysis of the name of the constant being defined.

```
const(Bs,Id) --> id(Y), !,
    { member(Y=_Z,Bs) -> Id = error(redefined,Y)
    ; Id = Y
    }.
const(_Bs,error(non_id_constant_symbol,X)) --> [X].

id(Id) --> [Id], {is_id(Id)}.  % Predicate is_id/1 is unchanged.
```

One finds each kind of error terms in the answer to the following Prolog question:

```
| ?- _Tokens = [a,=,1,+,1,., 'A1',=,a,-,1,., b,=,3,*,'A',.,
                c3,=,b,//,a,., a,=,b,rem,4,., 6,=,b,.],
     constants(Defs,_Tokens,[]).

Defs = [a=2,
        error(constant,error(non_id_constant_symbol,'A1')=1),
        error(expression,b=3*error(operand_syntax,'A')),
        error(expression,c3=error(undefined,b)//2),
        error(constant_and_expression,
              error(redefined,a)=
              [error(undefined,b),error(operator,rem),4]),
        error(constant_and_expression,
              error(non_id_constant_symbol,6)=
              error(expression_syntax,[b]))]
```

5. Conclusion

We have seen general guidelines on how to develop a DCG, in order to parse an input text regarding a language L, while we give the internal, semantical representation of this input, if it is correct or not.

Surely, this is a programming methodology – a kind of structured programming – specialised for DCGs.

Our achievements can be summarised as follows.

- Step-by-step refinement of a DCG makes its development easier.
- Error handling with DCGs is natural if we add error alternatives to the rules describing the language.
- Provided that T is an *abc*, and $L \subseteq T^*$ is the language to be described; when we add error alternatives to the original rules describing L, we augment the semantics of L with the semantics of $T^* \setminus L$, which is called the *error semantics* of L.

At last, the semantics of the whole T^* is given. It can be called the *full semantics* of L over T. The author believes, this is a novel approach to writing parsers in general.

There is a much more complex example here:

`http://aszt.inf.elte.hu/~asvanyi/pl/cm/pas/`

Future work could show the usefulness of this approach through its application to practical programming languages. Another possibility is to give the *full semantics* of languages using other approaches to semantics.

6. Acknowledgement

The European Union and the European Social Fund have provided financial support to the project under the grant agreement no. TÁMOP 4.2.1./B-09/KMR-2010-0003.

References

[1] P. Deransart, A.A. Ed-Dbali, L. Cervoni, *Prolog: The Standard (Reference Manual)*, Springer-Verlag, 1996.
[2] Ulf Nilsson, *AID: An Alternative Implementation of DCGs*, New Generation Computing, No 4, Vol 4 (1986), pp. 383-399.
[3] Richard O'Keefe, *The Craft of Prolog*, The MIT Press, 1990.
[4] Jukka Paakki, *A practical implementation of DCGs*, Lecture Notes in Computer Science, Volume 477/1991, Springer(1991), pp. 224-225.
[5] F. Pereira, D. Warren, *Definite clause grammars for language analysis*, Artificial Intelligence, Elsevier, 1980.
[6] L. Sterling, E. Shapiro, *The Art of Prolog* (Second Edition), The MIT Press, London, England, 1994.
[7] David H. D. Warren, *Logic programming and compiler writing*, Software: Practice and Experience, Volume 10 Issue 2, Copyright: 2010 John Wiley & Sons, Ltd. (Published Online: 27 Oct 2006) pp. 97-125.
[8] *Documentation for SICStus Prolog 4*, Swedish Institute of Computer Science, Kista, Sweden, 2010.
(`http://www.sics.se/isl/sicstuswww/site/documentation.html`)

Faculty of Informatics, Eötvös Loránd University, Budapest XI. Pázmány Péter sétány 1/C, H-1117, HUNGARY
E-mail address: `asvanyi@inf.elte.hu`