1 A Very Simple Compiler

In this section we examine another sort of metaprogram, namely one which takes a program as input and translates it into another program. In this case, we will take a "program" written in the format of a context-free phrase structure grammar and translate it into a Prolog program which can then be used to recognize the strings which are in the language generated by the input grammar.

We have already seen that there is a very close correspondence between the production rules of a context-free grammar and definite clauses. For example, a rule like

$$S \longrightarrow NP \;\; VP$$

corresponds to a definite clause of the form

```
s(L0,L):-
      np(L0,L1),
      vp(L1,L).
```

This clause states that the string which is the difference between `L0` and `L` belongs to category `s` if its first part can be categorized as an `np` and its second part can be categorized as a `vp`. Likewise, a rule expanding a nonterminal symbol into a terminal symbol, like

$$NP \longrightarrow John$$

corresponds to a definite clause of the form

```
np(L0,L):- L0 = [john|L].
```

Then a string `L` is generated by a grammar just in case the goal

```
?- s(L,[]).
```

succeeds.

```
*/
%
% cfdcg.pl
%

:- op(1200,xfx,==>).
/*
```

We begin by defining an operator '`==>`'/2, which we will use as our production arrow (the more obvious choice '`-->`' is already taken by Prolog). Grammar rules will then look like

```
s ==> np,vp.
```

in the case of expansion into nonterminal categories, and, in the case of expansion into terminal elements,

```
np ==> [john].
```

Note that anything which marks terminal elements as different from nonterminals would work here. We use this strategy because that is how Prolog's own compiler for "Definite Clause Grammars" represents terminal elements. Here is a complete grammar in this format. /*

```
*/
%
% g1.pl
%

s ==> np,vp.
s ==> np,aux,vp.

np ==> det,n.
np ==> [john].
np ==> [mary].
np ==> [he].
np ==> [him].
np ==> [she].
np ==> [her].

det ==> [the].
det ==> [a].
det ==> [every].
det ==> [some].

n ==> [lawyer].
n ==> [politician].
n ==> [dog].

aux ==> [will].
aux ==> [can].
aux ==> [wont].

vp ==> v.
vp ==> v,np.
vp ==> v,s.

v ==> [thinks].
v ==> [see].
v ==> [sees].
v ==> [think].
```

```
v ==> [sneeze].
v ==> [sneezes].

/*

*/
```

This grammar generates strings like

john sneezes

and

some dog thinks every politician will see mary sneeze

among (too many) others.

Note that, once we define '`==>`' as an operator of the appropriate sort, this grammar is a legitimate Prolog program, and can be loaded into Prolog with the usual commands for consulting or compiling programs. However, it cannot be "run" as is. One thing we could do would be to write an interpreter which could call a subgoal like

```
(LeftSide ==> RightSide)
```

to nondeterministically select instances of the '`==>`'/2 predicate for further processing.

Here, however, we will do something different. We will tell Prolog to consult the grammar file in a special way, so that when it reads in an expression whose primary functor is `==>`/2, it will translate that expression into a normal Prolog clause (i.e., an expression whose primary functor is `:-`/2), so that we can then "run ther grammar" directly, without an interpreter.

SICStus Prolog provides a special predicate called `expand_term`/2, which is called on every term in a file when that file is either consulted or compiled. The first thing that `expand_term`/2 does is to look and see whether the user has defined a special "hook" predicate called `term_expansion`/2. If any clauses for `term_expansion`/2 exist, then these are called and their output becomes the output of `expand_term`/2. Otherwise, `expand_term`/2 does nothing. So the simplest way to construct a compiler for programs in an object language of our own design is to provide a definition for `term_expansion`/2 which does the right thing.

```
/*
term_expansion((A ==> Bs),Clause):-
        cat2atom(A,L0,L,Head),
        Clause = (Head:- Body),
        build_body(Bs,Body,L0,L).
/*
```

Here we have defined `term_expansion`/2 so that it will recognize grammar rules as they are read in from a file. (Note that the file can contain ordinary Prolog clauses as well: if `expand_term`/2 calls `term_expansion`/2 and it fails, then expand_term/2 just defaults to its normal behavior.) Each such rule is translated to a definite clause Clause with head Head and body Body. The head is constructed by a call to cat2atom/4 and the body is constructed by build_body/4, both of which we will define below.

The essential problem is twofold. We must construct Prolog atoms out of category symbols, by making the category symbol the main functor and adding two argument positions, and then we must fill the argument positions with the appropriate difference-list arguments. That is, we have to make sure that, whenever two categories are adjacent in the right hand side of a rule, then the last argument of the first category must be identical to the first argument of the second category. In this way we enforce the condition that the second category begin where the first one ends.

The predicate cat2atom/4 expects to be given the proper list arguments when it is called. All it does is attach them to the category symbol to make a Prolog term.

```
*/
cat2atom(Category,L0,L,Atom):-
        functor(Atom,Category,2),
        arg(1,Atom,L0),
        arg(2,Atom,L).
% or
%       Head =.. [A,L0,L].
/*
```

Then build_body/4 just calls cat2atom/4 for every category in the right hand side of the grammar rule, making sure to keep proper track of the difference lists.

```
*/
build_body([Word],BAtom,L0,L):-
        BAtom = (L0=[Word|L]).
build_body(B,BAtom,L0,L):-
        \+ B = (_,_),
        cat2atom(B,L0,L,BAtom).
build_body((B1,Bs),(B1Atom,Rest),L0,L):-
        cat2atom(B1,L0,L1,B1Atom),
        build_body(Bs,Rest,L1,L).
/*
```

The first clause handles terminal productions. If the right hand side of a rule is a terminal element, that is, a word enclosed in square brackets, then it is converted into an atom with functor =/2 which will act to consume a word from the input string.

The second clause is the base clause for the case where the right hand side of the rule is a sequence of nonterminal symbols. Note that the right hand sides of rules are "round lists" like the bodies of Prolog clauses and goals in general.

There is no way to recognize the end of a round list except by the negated goal `\+B=(_,_)`, so we put this in as a guard on clause 2. In this case we simply call `cat2atom/4` one last time to return the last subgoal of the body we are building. Finally, in the recursive clause we call `cat2atom/4` on the first category in the rule's right hand side, passing the output difference list argument `L1` to the recursive call to `build_body/4` so that it will be used as the *input* difference list argument when the next body-atom is built.

Once we consult this code (which I have named `cfdcg.pl`), all subsequent calls to `consult` or `compile` will make use of the newly defined clauses for `term_expansion/2`. So, given the grammar `g1.pl` defined above, the following goal

```
?- [cfdcg],[g1].
```

(or just "`?-[cfdcg,g1].`" for short) will cause `g1.pl` to be translated into Prolog clauses. For example, after issuing the above goal, if you then issue the goal

```
?- listing.
```

you will see something like the following.

```
v(A, B) :-
        A=[thinks|B].
v(A, B) :-
        A=[see|B].
v(A, B) :-
        A=[sees|B].
v(A, B) :-
        A=[think|B].
v(A, B) :-
        A=[sneeze|B].
v(A, B) :-
        A=[sneezes|B].

s(A, B) :-
        np(A, C),
        vp(C, B).
s(A, B) :-
        np(A, D),
        aux(D, C),
        vp(C, B).

n(A, B) :-
        A=[lawyer|B].
n(A, B) :-
        A=[politician|B].
n(A, B) :-
```

```
        A=[dog|B].

np(A, B) :-
        det(A, C),
        n(C, B).
np(A, B) :-
        A=[john|B].
np(A, B) :-
        A=[mary|B].
np(A, B) :-
        A=[he|B].
np(A, B) :-
        A=[him|B].
np(A, B) :-
        A=[she|B].
np(A, B) :-
        A=[her|B].

det(A, B) :-
        A=[the|B].
det(A, B) :-
        A=[a|B].
det(A, B) :-
        A=[every|B].
det(A, B) :-
        A=[some|B].

aux(A, B) :-
        A=[will|B].
aux(A, B) :-
        A=[can|B].
aux(A, B) :-
        A=[wont|B].

vp(A, B) :-
        v(A, B).
vp(A, B) :-
        v(A, C),
        np(C, B).
vp(A, B) :-
        v(A, C),
        s(C, B).
```