How To Produce All Sentences from a Two-level Grammar

Dick Grune

Vrije Universiteit
Vakgroep Informatica
de Boelelaan 1081
1081 HV Amsterdam

ABSTRACT

Two-level grammars are very readable formalisms for generating Chomsky Type 0 languages. Teaching and understanding them is greatly aided by the presence of a sentence generator. This note shows how a usable sentence generator can be constructed. A working program is available from the author.

KEYWORDS: interpreters, programming languages, formal languages, VW-grammars

1. Introduction

Two-level grammars (also called VW-grammars) have long had the name of being arcane tools to be used only by off-world language designers [1]. The book "Grammars for Programming Languages" by Cleaveland and Uzgalis [2] has done much to dispel the fog, but still insight in the workings of a VW-grammar is best gained by working it out on paper, a laborious and error-prone activity at best. This note wants to attract attention to the fact that it is, however, quite possible to write a program that, given a VW-grammar, will produce *all* its terminal productions and that will even do so in a reasonably efficient way.

Such a program (sentence generator) is a great help in understanding VW-grammars: it allows a VW-grammar to be considered as a problem-solving statement. The program will then produce the terminal productions (i.e., solve the problem), thus emphasizing the analogy between problem solving and sentence generating. See the example in section 4.

2. VW-grammars

For those familiar with formal languages the following few words on VW-grammars will suffice; others are referred to the above book [2]. A VW-grammar consists (mainly) of a set of *metarules*, a set of *hyperrules* and a starting form. The metarules form a context-free system, the *metagrammar*, in which each rule defines a *metanotion*, indicated by a capital letter. A hyperrule looks and acts like a context-free rule except that it may be parametrized with one or more metanotions; the real context-free rule only evolves by consistently replacing each metanotion by one of its terminal productions. This mechanism can be used to express context conditions in a very readable way, and can in fact be shown to have the full power of a Turing machine [3]. The technique can be glimpsed from the following overly simple example; it is used in its full strength in the ALGOL 68 report [1].

```
T :: int; bool; char.                    \ a metarule, defining T
.                                        \ a separator
T constant expression: T, T constant.    \ a parametrized hyperrule
int constant: 0; 1.                      \ a non-parametrized hyperrule
bool constant: false; true.
char constant: 'a'; 'z'.
.                                        \ another separator
T constant expression.                   \ the starting form
.
```

This VW-grammar will produce for instance `int, 0` and `char, 'z'`, but not `int, 'z'` or `char, 0`, by virtue of the fact that the hyperrule

```
T constant expression: T, T constant.
```

expands into:

```
int constant expression: int, int constant.
bool constant expression: bool, bool constant.
char constant expression: char, char constant.
```

Notes on the syntax used: rules end with a period; alternatives are separated by semicolons; a left-hand-side is separated from a right-hand-side by one colon in a hyperrule and by two colons in a metarule, just to distinguish them.

Sometimes the need arises to have in a hyperrule two metanotions with the same name for which we do not want to require consistent substitution. Such metanotions can be denoted as `T1`, `T2`, etc. Note that if `T1` occurs more than once, it again must be replaced consistently. We shall need these rules in section 3.3.

3. The Sentence Generator

3.1. The context-free case

To generate terminal productions from a given context-free grammar one proceeds as follows. Intermediate results in the production consist of sets of so-called *sentential forms*. The first sentential form is the starting form. It is looked up among the left-hand-sides of the (hyper)rules; if it is found it is replaced by the alternatives obtained from the right-hand-side (its *direct productions*), otherwise it is a terminal production. Then the next sentential form is chosen and treated likewise, until the set of sentential forms is empty (if ever). The crux here lies in choosing the right sentential form from the set. A simple-minded recursive implementation will tend to choose the most recent one, with the bad consequence that the sentence generator may loose itself in an infinite branch of the production tree. This problem is conveniently solved by holding the sentential forms in a first-in-first-out queue: if there are *N* sentential forms in the queue at a given moment, each representing a sub-tree in the production tree, then after *N* turns each of them will have been processed, thus preventing any specific branch from monopolizing the attention. This is essentially breadth-first production; it implements a "fair" production process.

3.2. The two-level case

The above scheme, satisfactory for a context-free grammar, fails when applied to VW-grammars, for the simple reason that often an infinite number of left-hand-sides, all resulting from the same hyperrule `H`, must be examined to arrive at a point where progress is again possible.

A solution suggests itself when we reconsider why this infinite number of left-hand-sides is to be examined: we try to find such terminal productions of the metanotions in the left-hand-side of `H` that, after substitution, it matches the sentential form. In other words: we are trying to parse the sentential form according to the metagrammar, with the left-hand-side of `H` as the starting form. Once we have obtained a parsing, we can for each metanotion M_i in the left-hand-side infer what terminal (meta)production P_i has to be used to make the whole left-hand-side fit the sentential form. We can then substitute these terminal productions for the metanotions in the right-hand-side and add the result to the set of sentential forms.

Now, the metagrammar is context-free, and parsing according to a context-free grammar is a solved problem for which good solutions exist. Note that we need an algorithm that gives us all possible parsings. To obtain all possible direct productions of a given sentential form, parsing must be attempted successively with the left-hand-side of each hyperrule in the VW-grammar as a starting form.

This solution raises two problems and a minor worry. One is that, though the metagrammar is context-free, the left-hand-side of the hyperrule need not be so, due to the requirement of consistent substitution of equally-named metanotions. This requires a context check, which is, however, easily performed on each possible parsing as soon as it is yielded by the parser, or can be incorporated in the parser.

3.3 Free metanotions

The second problem is caused by metanotions that occur in the right-hand-side only, the so-called *free* metanotions. A free metanotion `F` may be replaced, again consistently, by any of its terminal productions; for the purpose of our program, i.e. generating all terminal productions of the given VW-grammar, it must eventually be replaced by *all* its terminal productions. Now the set of all the terminal productions of `F` may very well be infinite, which requires infinite time and infinite space. The solution is the same as with the VW-grammars

themselves: produce only one level at a time. In this case, however, that causes the presence of metanotions in the sentential forms. These must be handled with care: if a sentential form to be processed contains a free metanotion, it must be developed, rather than that an attempt be made to parse the sentential form. Now the developing of a free metanotion may bring in a new free metanotion which may happen to have the same name as a free metanotion already there: to this new free metanotion the rule of consistent substitution does not apply. This problem is solved by attaching to each new free metanotion an index that is higher than any already in use (see the last paragraph of section 2).

A small example will make this clearer. Suppose the sentential form under observation is

```
a P b P c
```

(in which both P's must be substituted consistently) and P is defined by the metarule

```
P :: x; P y P.
```

(in which both P's need not be substituted consistently). Now the development of P yields

```
a x b x c                and
a P1 y P2 b P1 y P2 c    .
```

The first one is ready to be parsed the next time its turn comes, in the second one P1 and P2 will each independently undergo consistent substitution, yielding:

```
a x y x b x y x c
a x y P3 y P4 b x y P3 y P4 c
a P3 y P4 y x b P3 y P4 y x c
a P3 y P4 y P5 y P6 b P3 y P4 y P5 y P6 c
```

(It is clear that a highly ambiguous rule like `P :: x; P y P.` will cause a proliferation of sentential forms, which will slow down, but not block the production process.)

One minor worry is hidden in the fact that when parsing a sentential form according to the left-hand-side of a given hyperrule, we need *all* parsings. For some pathological metagrammars there may be infinitely many of these (due to ambiguous parsing of the empty string). The production process will then try to add all these to the set of sentential forms and no progress will be made any more. Such grammars are rare and hardly useful, and will be disallowed.

4. An example.

We shall demonstrate the problem-solving power of a VW-grammar production process through the following toy problem, due to Prof. A. van Wijngaarden. For an extensive example see [5].

The bottom of a one meter wide mountain cleft is strewn with a neat array of boulders, each one meter long. Sheep are entering the cleft at both ends. The two flocks try to pass each other, under the following rules. A boulder can hold at most one sheep; a sheep can step on the next boulder, or, if that is occupied, jump over one sheep to the boulder beyond it, if that boulder is free; no sheep can go backward.

Given the initial constellation, how can the flocks pass each other? A left-going sheep is denoted by <, a right-going sheep by >, and an empty boulder by =. The metagrammar then defines a flock going left, L, by stating that an L is either a < followed by an L, or the empty string. Likewise, flocks going right, R, free space F and positions P are defined.

```
\ Sheep in Mountain Cleft

\ < is a left-going sheep, > wants right, = is an empty boulder.
L :: < L ; .            \ flock of sheep going Left
R :: > R ; .            \ flock of sheep going Right
F :: = F ; .            \ Free space
A :: < ; > ; = .        \ Any object
P :: A P ; .            \ Positions
.
P = < P1: P < = P1 print.       \ one sheep goes left
P > = P1: P = > P1 print.       \ one sheep goes right
P = > < P1: P < > = P1 print.   \ one sheep jumps left
P > < = P1: P = < > P1 print.   \ one sheep jumps right
L F R: L F R !.                 \ solution found
P print: P /, P.                \ print position and continue
.
> > = < <.                      \ initial constellation
.
```

The hypergrammar, whose rules are identified by a single colon between left-hand-side and right-hand-side, describes the allowed steps. We consider the problem solved when we have a constellation that consists of (from left to right) a flock going left, zero or more empty boulders, and a flock going right. The solution thus has the form `L F R` and according to the above grammar it produces a copy of itself with an exclamation-mark suffixed to it. Such a form will not match any left-hand-side, so it is a terminal production or a blind alley. The suffix `print` occurring in all transition rules forces each resulting sentential form to go through the printing rule which makes a copy of the constellation with a slash suffixed to it, which prevents further production, and continues with the original sentential form. In this way we obtain an account of all the intermediate steps in the sheep-moving process, rather than just the final constellation. The result of the program was:

```
0 sec: >><=</, >><<=/, >><<=.
0 sec: >=><</, =><</, =><<.
2 sec: >><=</, >=<></, ><=></, ><<>=/, ><<=>/, ><<=>.
2 sec: >><=</, >=<></, ><=></, =<>></, <=>></, <=>><.
3 sec: >><=</, >=<></, =><></, <>=></, <=>></, <=>><.
3 sec: >=><</, ><>=</, ><><=/, ><=<>/, ><<=>/, ><<=>.
3 sec: >=><</, ><>=</, ><=></, ><<>=/, ><<=>/, ><<=>.
3 sec: >=><</, ><>=</, ><=></, =<>></, <=>></, <=>><.
4 sec: >><=</, >=<></, =><></, <>=></, <><>=/, <><=>/,
                        <=<>>/, <<=>>/, <<=>>!.
4 sec: >=><</, ><>=</, ><><=/, ><=<>/, =<><>/, <=><>/,
                        <<>=>/, <<=>>/, <<=>>!.
```

Only the last two sentential forms are terminal productions, the others are blind alleys. The difference between a terminal production and a blind alley, however, is a matter of external definition. If we want the program to distinguish them, we shall have to supply a formal rule. In our program we have chosen not to do so, since it is our experience that the blind alleys often provide more insight in what is really going on than the terminal productions themselves.

5. The Program

The program employs a recursive-descent parser, rigged so as to produce *all* possible parsings, something a simple-minded recursive-descent parser does not do. Its advantage was that it was simple to write, its disadvantages are that it cannot handle left-recursive metagrammars and that it can take exponential time for some grammars (though it seldom does). Both flaws could be remedied by using an Earley-like parser [6], one in which a boolean array P is set up in which $P_{p,l,m}$ is true if and only if the string of length l at position p in the sentential form is a terminal production of the metanotion m.

The program is written in C [4] and is eighteen pages long, divided about equally between reading and checking the grammar, producing terminal productions, and displaying intermediate and resulting sentential forms. It can be ordered from the author at the above address, or by contacting him on USENET as ...decvax!mcvax!vu44!dick.

6. Literature

[1] A. van Wijngaarden et al. (Eds), Revised Report on the Algorithmic Language ALGOL 68, Acta Informatica, 5, p1-236, 1975; also MC Tract 50, Mathematical Centre, Amsterdam 1976; also Sigplan Notices, 12(5), p5-70, 1977.

[2] J. Craig Cleaveland & R.C. Uzgalis, Grammars for Programming Languages, Elsevier Scientific Publ. Co., Amsterdam, 1977.

[3] A. van Wijngaarden, The generative power of two-level grammars, in J. Loecks (Ed.), Automata, Languages and Programming, Lecture Notes in Computer Science 14, Springer Verlag, Berlin, 1974.

[4] B.W. Kernighan & D.M. Ritchie, The C Programming Language, Prentice Hall, 1978.

[5] A. van Wijngaarden, Languageless Programming, in J.K. Reid (Ed.), Proceedings of the IFIP/TC2/WG2.1 Working Conference on the Relations between Numerical Computation and Programming Languages, Boulder, North-Holland Publ. Comp., 1981.

[6] J. Earley, An Efficient Context-Free Parsing Algorithm, Comm. ACM, Vol 13, 2, p. 94-102, 1970.