Notes on Recursive Descent Grammars

J.R.B. Cockett
Department of Computer Science, University of Calgary,
Calgary, T2N 1N4, Alberta, Canada

January 28, 2016

1 Introduction

Intuitively a (context free) grammar is a *recursive descent grammar* if it can be used *directly* to generate a recursive descent parser. The direct translations allows for a function corresponding to each nonterminal which has a pattern matched clause corresponding to each production of the nonterminal. A production which does not require that any token on the input be matched is treated as a *default* clause and only is applied after all the other rules have been tried: of course having two default clauses will spell trouble!

Here is an example of how a grammar (the expression grammar) gets translated into a recursive descent parser:

```
exp -> term mterm.                    exp(ins) = mterm(term(ins))
term -> factor mfactor.               term(ins) = mfactor(factor(ins))
factor -> NUM                         factor(NUM:ins) = ins
        | VAR.                        factor(VAR:ins) = ins
mterm -> ADD term mterm   becomes code mterm(ADD:ins) = mterm(term(ins))
       |.                             mterm(input) = input
mfactor -> MUL factor mfactor         mfactor(MUL:ins) = mfactor(factor(ins))
       |.                             mfactor(ins) = ins
```

Notice how each production gets turned into code: each nonterminal becomes a function, each production rule becomes a clause of the function which is preconditioned on matching the initial sequence of terminals of the production. Subsequent nonterminals of the production are turned into a sequence of applications of their corresponding functions to the nonterminals on the leftover input.

The immense convenience of generating a recursive descent parser raises the question: which grammars can be translated, in this manner, into recursive descent parsers? The code resulting from this translation should recognize precisely the language of the grammar. Often it will be the case that the translation gives nice terminating code but does not recognize all the strings of the grammar. When the translation produces code which terminates and recognizes precisely the language specified by the grammar we shall say the translation is **complete**, otherwise we shall say it is incomplete.

It turns out that the answer to when the translation is complete is a little more complex than one might have at first suspected: this is largely because of the "defaulting" behaviour of the pattern matching which must not, by a premature match of an earlier rule, block a possible parse using a later rule.

In what follows we shall always assume that the grammar satisfies all the basic sanity checks: that is that the grammar is both *reachable* and *realizable*.

We present the following thee notions:

- Recursive descent form
- Pattern determined grammars
- Recursive descent grammars.

The second notion, pattern determined grammars, is an intermediate concept in which patterns are treated essentially as single symbols and, thus, the process of determining whether one has one of these grammars devolves into a first set and follow set calculation.. The reader, therefore, may skip the middle section (Section 3) without loss: the last section contains the key algorithm for determining whether a grammar is recursive descent.

2 Recursive descent form

A production is in **recursive descent form** if it is of the form:

$$x \to \beta\alpha$$

where β is a string of terminals (possibly empty) and α is a string of nonterminals (also possibly empty). A grammar is in recursive descent form when each of its productions is in recursive descent form.

The string of terminals, β, which start a production in recursive descent form is called the **pattern** of the production. A production in recursive descent form has an **empty pattern** when β is the empty string and has a nonempty pattern otherwise.

The point of the recursive descent form is that it can always be directly translated into code ... even if the code does not actually have the desired effect! For example the production

$$\texttt{f} \; -> \; \texttt{A B h k}$$

is in recursive descent form (here `A` and `B` are terminals) and gets translated into the pattern matched code phrase

$$\texttt{f}(\texttt{A} : \texttt{B} : \texttt{ins}) = \texttt{k}(\texttt{h}(\texttt{ins}))$$

which matches the `A` and `B` at the front of the input list (the pattern of the production) and then calls `h` then `k` on the remaining input to act recursively on the remaining input.

Of course, a recursive descent parser produced in this manner from a grammar whose productions are all in recursive descent form will not always work let alone be complete. This can be illustrated by taking the reverse of the expression grammar above. This is not in recursive descent form but can be made so by breaking up the production rules by adding some intermediate nonterminals:

```
exp -> mterm term.
term -> mfactor factor.
factor -> NUM
        | VAR.
mterm -> mterm term ADD
       |.
mfactor -> mfactor factor MUL
       |.
```

becomes

```
exp -> mterm term.
term -> mfactor factor.
factor -> NUM
        | VAR.
mterm -> mterm term add
       |.
add -> ADD.
mfactor -> mfactor factor mul
       |.
mul -> MUL.
```

This illustrates how one can easily turn *any* grammar into one which is in recursive descent form by the simple step of breaking up the productions at terminal strings (which are not patterns). We can then translate any grammar whose rules are in recursive descent form into code:

```
exp -> mterm term.                          exp(ins) = term(mterm(ins))
term -> mfactor factor.                     term(ins) = factor(mfactor(ins))
factor -> NUM                               factor(NUM:ins) = ins
        | VAR.                              factor(VAR:ins) = ins
mterm -> mterm term add      becomes code   mterm(ins) -> add(term(mterm(ins)))
       |.                                   mterm(ins) = ins
add -> ADD.                                 add(ADD:ins) = ins
mfactor -> mfactor factor mul               mfactor(ins) = mul(factor(mfactor(ins)))
       |.                                   mfactor(ins) = ins
mul -> MUL.                                 mul(MUL:ins) = ins
```

As we suspected, there are a number of worrisome things about the resulting program! To start with `mterm` and `mfactor` both implement bottomless recursions – so the program will not terminate (this will usually cause the recursion stack to overflow!). To avoid this problem, clearly, we must require that the grammar has no left-recursion. However, notice that there is also another problem: both of the `mterm` rules are now "default" rules ... but which should we choose? Putting one ahead of the other will make the later one unreachable ... and, thus, potentially certain important parsing actions may become unreachable in the code. Thus, simply not being left-recursive is not enough to make the parsing decisions determined by a recursive descent grammar complete.

Below we shall explore two notions. The first notion, a **pattern determined grammar**, is described in Section 3. We point out immediately that there is a rather unsatisfactory aspect of these grammars as they require certain pattern sets to be disjoint. This is a strong requirement which we shall *not* require of recursive descent grammars. Pattern determined grammars are a straightforward generalization of an LL(1) grammar. It generalizes the now familiar "first set" calculation to a "first pattern set" calculation. To write a top-down parser for a pattern determined grammar one uses the first pattern set associated with the productions of the current nonterminal and expands using that production only if the input (at that stage) matches one of the patterns of the production. As all patterns are disjoint this completely determines the top-down parse.

Of course, this is not exactly how a recursive descent parser works! The only pattern it should be necessary to check is the pattern of the production itself! Our second notion, described in 4, uses a rather different method and does allows for overlapping patterns: this time we capture precisely what we intend a *recursive descent grammar* to be. A recursive descent grammar is precisely a grammar for which the translation discussed above is complete provided the translation places all rules with *more specific* patterns before all those with *less specific patterns*.

A pattern β is more specific than β' whenever β' is a prefix of β (that is $\beta' \ll \beta$. This allows one to preorder the productions associated with each nonterminal according to the specificity of their pattern. One requirement of a recursive descent grammar is that this preorder is actually a partial order: equivalently this means that is no two productions associated to a nonterminal have the same pattern. The program then must always try the more specific productions before the less specific productions.

Thus, a production associated to a nonterminal is chosen only when its first pattern matches the current input *and* all the earlier productions (which must include all the productions with more specific patterns) have been tried and have failed to match the input. In particular, this certainly means that there can be at most one production with no pattern: as above, it will act as the default "catch all" rule. However, it also means there can be a more subtle form of defaulting to rules with less specific patterns (which is not possible in pattern determined grammars). To determine whether

a grammar is a recursive descent grammar one must ensure that no string which is recognized by a more specific production can also be recognized by a later less specific production. Thus, unlike a pattern determined grammar, the first patterns (associated to a nonterminal) need no longer be "disjoint" because the less specific rules can now be used to catch cases which more specific productions fail to match.

Section 4 describes a basic procedure for determining whether a grammar is recursive descent. It is based on a technique for determining whether a pattern can be recognized by a production.

3 Pattern Determined Grammars

Our first objective is to look at how the first set and follow set calculations and the theorem for LL(1) grammars can be generalized to give conditions for being a "pattern determined" grammar. This relies on the rather strong assumption, discussed more below, that the patterns associated the productions of each nonterminal be disjoint.

Those simply interested in recursive descent grammars should skip this section ...

3.1 Calculating the pattern sets

An important calculation which we shall require is to be able to tell which patterns a nonterminal can "eat". This would be an easy calculation, however, the presence of production rules with empty patterns makes it much tougher!

A pattern determined grammar should have its rules in recursive descent form. The objective is to be able to perform a top-down parse by associating with each nonterminal a set of patterns which allow one to determine a unique production to expand at each state of the pushdown automata by simply checking whether one of its patterns matches the input.

A pattern can be associated with a nonterminal x if it is the first pattern of a production with head x. However, in addition, if there are rules associated to x with an empty pattern a rule with a pattern can be called indirectly. Thus, given a rule of the form `x -> f g`, where `f` is a nonterminal, we may use any patterns which derive from f and indeed if `f` is nullable any patterns derived from `g`. Thus, we have to take into account the effect of the nullable nonterminals.

The calculation we consider then follows a pattern which should be quite familiar. It is broken into three steps: calculating the *immediate* first pattern relation, constructing a *propagation* relation, and then using these to get the complete first pattern sets – by composing the reflexive transitive closure of the propagation relation with the immediate first pattern relation.

3.2 Calculating the first pattern sets

Here is the scheme for calculating the first pattern sets:

- First calculate the **immediate pattern sets** of each nonterminal:

$$\mathsf{IFirstPat}(x) = \{\beta | \beta \neq \varepsilon \ \ \& \ \ x \to \beta\ \alpha\}$$

  that is the first patterns associated with the nonterminal x are the nonempty patterns of the productions with head x.

- Next build the **propagation graph** for the first pattern calculations: this has nodes the nonterminals and edges defined by:

$$x \rightarrowtail y \Leftrightarrow x \to \alpha^{\cdot}\ y\ \gamma \ \ \& \ \ \mathsf{nullable}(\alpha^{\cdot})$$

  Notice immediately this propagation only uses the the rules with an empty first pattern.

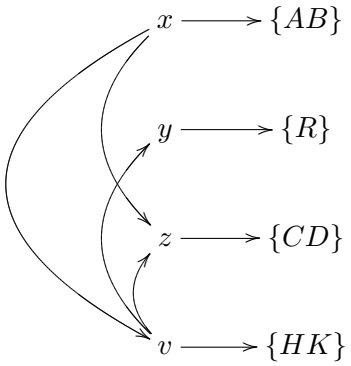
- Letting $x \rightarrowtail^{*} y$ denote the transitive reflexive closure of the the propagation relation then

$$\beta \in \mathsf{FirstPat}(x) \Leftrightarrow \exists y.\ x \rightarrowtail^{*} y \quad \& \quad \beta \in \mathsf{IFirstPat}(y)$$

Here is an example:

propagate FirstPat

```
x -> A B y
   | z v.
z -> C D
   |.
v -> H K
   | z y.
y -> R.
```



This means $\mathsf{FirstPat}(x) = \{AB, CD, HK, R\}$.

We may extend the definition of this first pattern set algorithm to arbitrary sentential forms in recursive descent form as follows:

$$\mathsf{FirstPat}(\beta\ \alpha) = \begin{cases} \{\beta\} & \beta \neq \varepsilon \\ \bigcup_{\substack{\alpha = \gamma z \alpha' \\ \gamma \text{ nullable}}} \mathsf{FirstPat}(z) & \beta = \varepsilon \end{cases}$$

This says that a first pattern set of a sentential form may be arrived at by skipping over a nullable nonterminal sequence and then using the first pattern set of the next nonterminal etc.

3.3 Being pattern determined

In order for a grammar to be pattern determined, we must know exactly which production to apply given its first pattern sets. To achieve this we must require the following rules:

[PD.1] For each nonterminal x its distinct productions

$$x \to \beta_i\ \alpha_i \qquad x \to \beta_j\ \alpha_j$$

have bodies (RHSs) with disjoint first pattern sets (what this means is explained below):

$$\mathsf{FirstPat}(\beta_i\ \alpha_i) \sqcup \mathsf{FirstPat}(\beta_j\ \alpha_j).$$

[PD.2] Every nonterminal, x, has at most one nullable production (null unambiguous).

[PD.3] For each nullable nonterminal x the first pattern set of x must be disjoint from the follow pattern set of x:

$$\mathsf{FirstPat}(x) \sqcup \mathsf{FollowPat}(x).$$

Here two pattern sets P_1 and P_2 are **disjoint** in case there is no string of terminals which matches a pattern from both sets. Of course to say a string of terminals, t, **matches** a pattern p simply means that p is a prefix of t, written $p \ll t$. Two patterns β_1 and β_2 match the same string if and only if either $\beta_1 \ll \beta_2$ or $\beta_2 \ll \beta_1$: that is one pattern is a prefix of the other. When $\beta_1 \ll \beta_2$ we say that β_1 is **less specific** that β_2. Thus two pattern sets are disjoint only if it happens that a pattern picked from one set is never a prefix of (i.e. less specific than) a pattern picked from the other set.

Clearly [**PD.1**] is necessary. Also as, a first pattern determined grammar must be unambiguous we know it must be null unambiguous that is [**PD.2**] must hold. The next example illustrates why [**PD.1**] is necessary. Consider the following grammar and corresponding code:

```
x -> A B y                          x(A:B:ins) = y(ins)
   | B z                            x(B:ins) = z(ins)
   | w.                             x(ins) = w(ins)
w -> A x          becomes code      w(A:ins) = x(ins)
   |.                               w(ins) = ins
z -> D E.                           z(D:E:ins) = ins
y -> F.                             y(F:ins) = ins
```

A string which is recognized by this grammar is `A B D E` as

$$x \to w \to \texttt{A}\ x \to \texttt{A B}\ z \to\ \texttt{A B D E}$$

but notice that the code fails on this as

$$\begin{aligned} x(\texttt{A B D E}) &= y(\texttt{D E}) \\ &= \text{... fail} \end{aligned}$$

Of course this grammar does not satisfy [**PD.1**]. The problem is that the wrong rule is used on the initial `A B` as the first phrase of the function x matches this initial sequence and the "default" production (which leads to the parse) is only tried after the other productions have failed but also has a first pattern for this (which is just `A`)!

3.4 Calculating follow patterns

The last condition, [**PD.3**], ensures if a nonterminal is nullable then it must be the case that any pattern which will be matched *following* that nonterminal must be disjoint from anything which can be matched by the nonterminal itself. Let us start by illustrating the problem with an example:

```
x -> A H                            x(A:H:ins) = ins
   | B G                            x(B:G:ins) = ins
   | w v.                           x(ins) = v(w(ins))
w -> C x          becomes code      w(C;ins) = x(ins)
   |.                               w(ins) = ins
v -> C D                            v(C:D:ins) = ins
   | C E.                           v(C:E:ins) = ins
```

Here the grammar recognizes `C E` by:

$$x \to w\ v \to v \to \texttt{C E}$$

However, the code has the following effect:

$$\begin{aligned} x(\texttt{C E}) &= v(w(\texttt{C E})) \\ &= v(x(\texttt{E})) \\ &= \text{... fail} \end{aligned}$$

The problem is that the default action of nulling w gives another matching opportunity which the code will not exercise. Notice that `C` is in the first pattern of w but it also is a pattern which can follow w (see below) and w is nullable. This means **[PD.3]** is not satisfied.

To calculate the follow patterns for a grammar in recursive descent form:

- First calculate the **immediate follow patterns**:

$$\mathsf{IFollowPat}(x) = \left\{ \beta \;\middle|\; \begin{array}{l} z \to \beta'\alpha x\alpha' \\ \beta \in \mathsf{FirstPat}(\alpha') \end{array} \right\}$$

- Next calculate the **follow propagation graph**: the nodes of the graph are the nonterminals and the edges are given by:

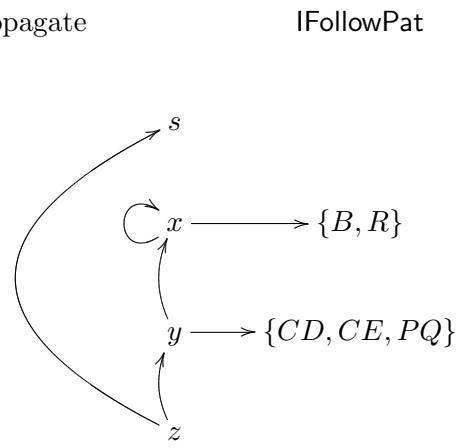
$$x \longrightarrow y \Leftrightarrow y \to \beta\ \alpha\ x\ \alpha' \ \ \&\ \ \mathsf{nullable}(\alpha')$$

- Then

$$\beta \in \mathsf{FollowPat}(x) \Leftrightarrow \exists y.\, x \overset{*}{\longrightarrow} y \ \ \&\ \ \beta \in \mathsf{IFollowPat}(y)$$

Here is an example of this calculation:

```
s -> A x y
   | y z.
y -> B z
   | R.
z -> C D
   | C E.
x -> P Q y x
   |.
```



This means $\mathsf{FollowPat}(z) = \{CD, CE, PQ, B, R\}$.

This grammar is pattern determined as its first pattern sets for production bodies with the same head are disjoint. Each terminal has at most one "default" production (so at most one can be nullable). Finally the first pattern set is disjoint from the follow pattern set for each nullable nonterminal.

4 Recursive Descent Grammars

An unsatisfactory aspect of pattern determined grammars is that the pattern sets must be disjoint so that one can always make a unique choice of which production to fire. A satisfying aspect of pattern determined grammars is it is easy to check whether one is holding one!

However, being pattern determined is altogether too strong requirement for general recursive descent grammars. To illustrate this consider what happens when patterns overlap. A variant of the first example of the previous section is:

```
x -> A B y
   | w z.
w -> A x
   |.
z -> D E.
y -> F.
```

This grammar does not satisfy **[PD.1]** ... but the pattern clash is on `AB` on the first rule of `x` and `A` being in the first pattern set of `w`. However, `A B` cannot be the prefix of anything recognized by the production `x -> w z`. Thus, although the first sets are not disjoint when the prefix is `A B`, using second production for `x` never results in a parse. So this does not cause a problem and the translation to recursive descent code *is* complete.

However, notice to determine the completeness we needed a new sort of calculation: we needed to be able to show that the sentential form `wz` *cannot* recognize the terminal prefix `AB`. In general, whenever $\beta_1 \gg \beta_2$ and $x \to \beta_1\gamma_1$ and $x \to \beta_2\gamma_2$ are two rules we must check that the terminal prefix β_1 is not the prefix of something which can be recognized by $\beta_2\gamma_2$ in any context where `x` occurs.

Now we could augment the above first pattern set calculation to check, whenever there are overlapping patterns, that any more general patterns associated with a nonterminal cannot also recognize the more specific pattern. However, some thought makes one realize that it is much more sensible and simpler to directly check that later rules can never recognize the patterns on which earlier rules are "fired".

This then begs the question of how one determines that a production cannot recognize a pattern. This is answered below but first we need to discuss the order in which productions should be considered.

4.1 Conditions for being recursive descent

Let us revisit the translation of the productions into code in a little more detail. For the program the order in which the productions of the grammar are placed is clearly absolutely critical. However, in a grammar the production order does not matter. However, notice that when the productions are in recursive descent form we may always preorder them according to the *specificity* of the patterns of the rule. A production $x \to \beta\alpha$ is *more specific* than another production $x \to \beta'\alpha'$ in case $\beta' \gg \beta$, that is β' is a prefix of β.

When the productions associated to a nonterminal, x, are translated into a program we require that the more specific productions are always placed above less specific productions (so they are matched first). Any order which satisfies this gives an acceptable translation. All programs which

result from acceptable translations will be equivalent provided the specificity preorders is actually a partial orders (i.e. no two productions are equally specific – that is have the same pattern). Clearly placing a more general production above a more specific production would immediately make the more specific production unreachable in the code and thus the translation would immediately be incomplete (whenever the grammar is realizable). Notice, also that one must not have two rules which are *equally* specific as, again, putting the one above the other would cause the latter to be blocked. In particular, notice the general requirement that no two productions associated to a given nonterminal have the same pattern means there can be at most one rule with an empty pattern.

The preorder induced by the patterns of the productions associated with any nonterminal, in a grammar in recursive descent form, is called the **specificity preorder**. The above discussion shows that this must actually be a partial order (i.e it satisfies antisymmetry $x \leq y \;\&\; x \leq y \Rightarrow x = y$) if it is to be a recursive descent grammar.

However, even if the specificity preorder is a partial order, and an acceptable translation to code is taken, it is still possible that a more specific production could recognize something that a later less specific production can also recognize: this would make the translation incomplete. Thus we need to be able to check whether less specific productions can recognize strings with the more specific pattern as a prefix. We shall show how this is done below: we shall say that the production $x \to \gamma$ **recognizes the pattern** β if there is an input string with prefix β which is recognized after expanding this production.

This now allows us to state the rules that a recursive descent grammar must satisfy:

[RD.1] The grammar must be in recursive descent form.

[RD.2] The productions for each non-terminal must be partially ordered by specificity.

[RD.3] The grammar must be left recursion free (i.e. leftmost expansion must always terminate).

[RD.4] For any two rules $x \to \beta_1\alpha_1$ and $x \to \beta_2\alpha_2$ in which the first is more specific than the second the production $x \to \beta_2\alpha_2$ must not recognize the pattern β_1.

It is clear that these rules are necessary: they are also sufficient as at each stage in a recursive descent parse they mean a unique production will lead to a parse.

This leaves the problem of determining whether a pattern can be recognized by a production.

4.2 Determining whether a pattern can be recognized by a sentential form

Let us assume that the grammar is not left recursive and is in recursive descent form such that the productions associated to each nonterminal are partially ordered by specificity. Suppose we are sitting at a rule $x \to \gamma$ and we want to determine whether a pattern, that is a string of terminals, β can be the prefix of a recognized string which results from "firing" this rule. How do we do this?

First we shall make the question more precise: we shall say that the pattern β is recognized by the production $x \to \gamma$ if there are derivations (for some sentential forms δ,δ', and δ''):

$$\mathsf{start} \to^* \delta x \delta' \quad \text{and} \quad \gamma\delta' \to^* \beta\delta''.$$

To determine whether this is true we use a nondeterministic search which starts with the triple:

$$(\beta, x, \gamma)$$

and generate triples from it by the rules described below. β will be recognized by the production $x \to \gamma$ if a triple of the form $(\varepsilon, z, \gamma')$ can be generated using the following replacement rules:

- Terminal reduction:
$$(T\beta, x, T'\gamma) \Rightarrow \begin{cases} \{(\beta, x, \gamma)\} & \text{if } T = T' \\ \{\} & \text{otherwise} \end{cases}$$

- Left nonterminal expansion:
$$(\beta, x, y\gamma) \Rightarrow \{(\beta, x, \delta\gamma) | y \to \delta \text{ is a production}\}$$

- Follow expansion:
$$(\beta, x, \varepsilon) \Rightarrow \{(\beta, y, \delta) | y \to \gamma x \delta \text{ is a production}\}$$

Note that if we apply these rules, because the grammar has no left recursion, eventually either (β, x, γ) will reduce β to the empty string (in which case we are done) or reduce the sentential form β to the empty sentential form. Thus, if we do not find a way of matching the pattern we will reach a stage at which a follow expansion must be performed.

The triples of the form (β, x, ε) are called **complete** triples.

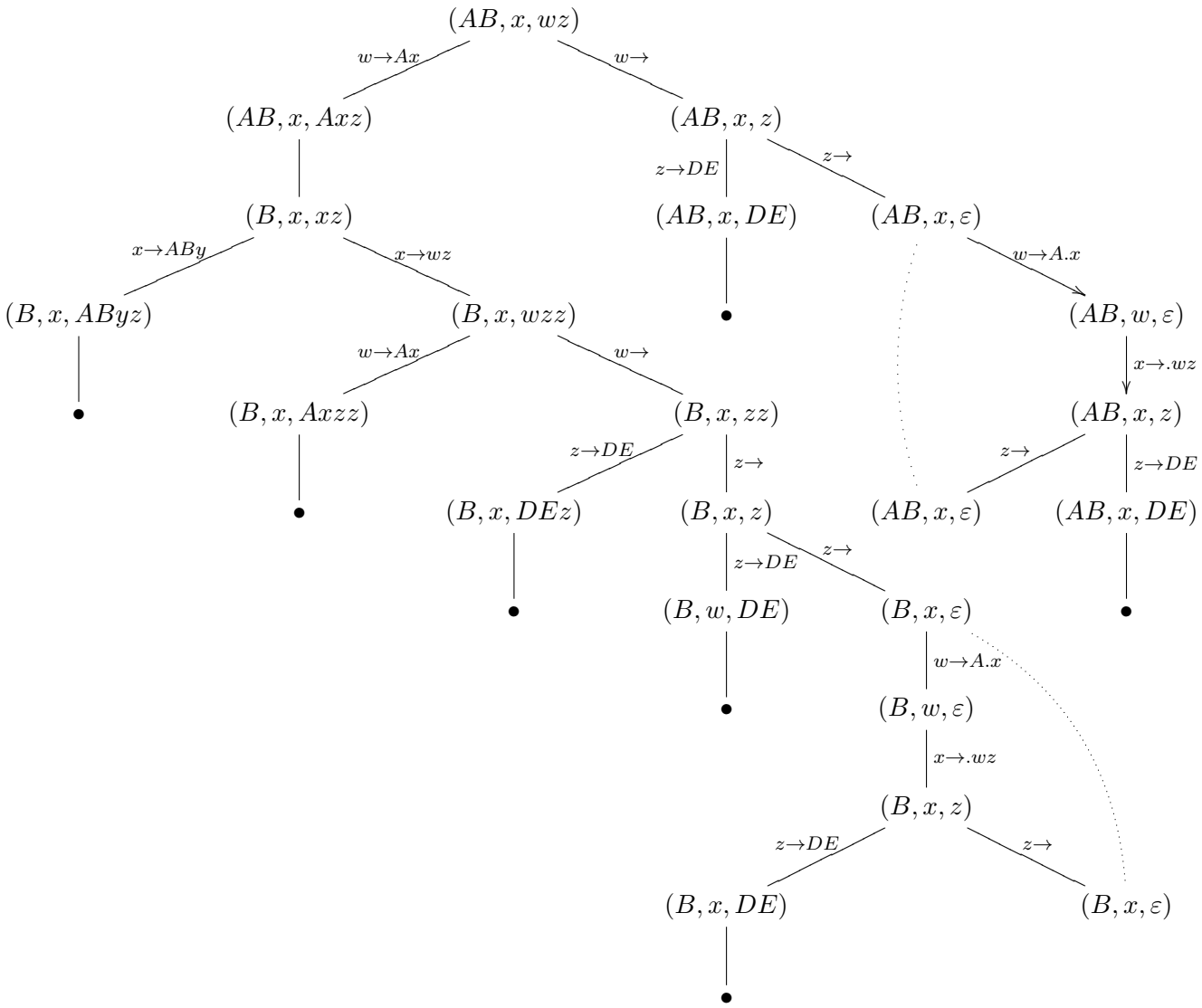
When a triple is complete, and β is non empty, the only option to continue is to apply the follow expansion rule. It is possible using the follow expansion rules to repeatedly generate the same completed triple. To avoid getting into an infinite search it is necessary to remember which completed triples have been generated so as to avoid a repetition in the search.

To illustrate the search we consider the grammar:

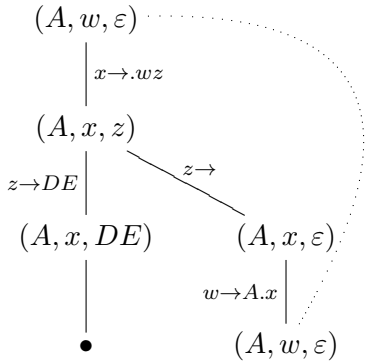
```
x -> A B y
   | w z.
w -> A x
   |.
z -> D E
   |.
y -> F.
```

which is a slight variant of the grammar above. Clearly the production satisfy the specificity requirement and the grammar is null unambiguous. Thus, we must check that less specific rule cannot recognize more specific patterns. The first thing we must check is that the pattern `AB` cannot

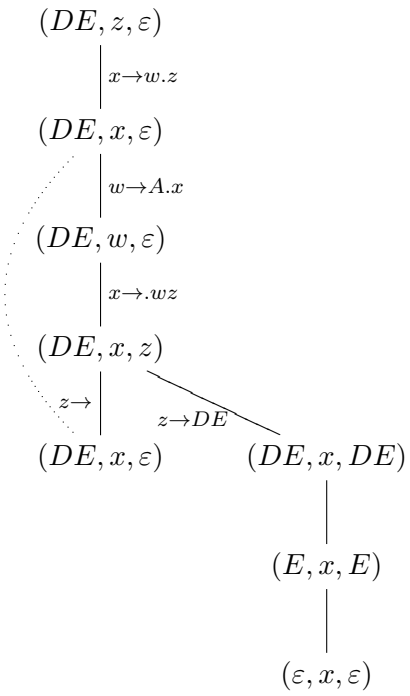
be recognized by the production $x \to wz$ here is the computation:



We must also check that `A` cannot be recognized by $w \to \varepsilon$, here is that calculation:



Finally we must check that DE cannot be recognized by $z \to \varepsilon$, here is that calculation:



So in fact this grammar is *not* recursive descent because DE *can* be recognized by $z \to \varepsilon$! Can you reconstruct the derivations involved from the search? Can you show that the grammar with which we started the section *is* recursive descent?

5 Epilogue

5.1 General remarks

1. First it is worth reiterating that most grammars are not recursive descent (or even in recursive descent form)! However, just because a grammar not in this form does not mean that it cannot be transformed in a recursive descent grammar ... and in fact all LL(k) grammars can be so transformed and this makes recursive descent parsing – with some transformation – widely applicable.

2. Recursive descent parsers are *efficient*: they require linear time on the size of the input (with a constant which depends on the maximum depth of recursion before one hits a pattern). Being top-down they also provide good error reporting.

3. A recursive descent grammar is always unambiguous. This is because the rules ensure that there is only one match which can be applied at any stage. This means that there can be at most one parse tree corresponding to a string of terminals.

4. Note there is no requirement of "no left-recursion" in a pattern determined grammar. As an exercise for the reader: can you show that they actually do imply that there cannot be a left-recursion?

5. What is the complexity of determining whether a grammar is recursive descent? We have argued informally that the search involved is finite but the reader rightly may be skeptical of the informal argument! Below we discuss this question further: the procedure does terminate BUT the complexity is, in the worst case, pretty bad! Of course, for most grammars this search is really quite limited.

5.2 Discussion of complexity

We shall assume the grammar $\mathcal{G} = (N, T, P, p, s)$ is already in recursive descent form, that it has no left recursion, and the productions associated with each nonterminal are partially ordered by specificity. The remain source of complexity is the cost of doing a search to find out whether a less specific rule could recognize a pattern. This is determined by the number of "reduced" search states $S_{\mathcal{G}}$ which are possible for a grammar and, must allow for the cost of:

(a) Remembering and checking whether complete states have been encountered before.

(b) The cost of reducing an unreduced state.

Both these costs are completely dominated by the cost of doing a search so we shall ignore them.

Recall a state is a triple: a pattern, a nonterminal, and a sentential form. The first component is always a postfix of the pattern, β_0, on which the search is initiated. The initiating patterns are, in turn, determined by the rules, however, we shall concentrate on bounding the search on an initial pattern β_0.

The number of different nonterminals possible in the second component of a state is, of course, bounded by $|N|$. The main difficulty is to determine the number of different sentential forms it is possible to have with a given pattern. To simplify this calculation we shall restrict attention to states whose the sentential form in the last component has no starting pattern, that is states

whose third component consists of a string of nonterminals. These are called are **reduced states**. Notice that all the states generated by the machine have in their third component a sentential form $\beta\alpha$ where β is a pattern and α is a list of nonterminals one never gets a general sentential form in the last component. A state whose last component has a nonempty pattern will be reduced deterministically by doing terminal reductions: the cost of reducing a state (or failing the search) is bounded by the length of the pattern in the first coordinate so can be ignored (as mentioned above). This leaves us to consider reduced states, that is those states whose third component is a string of nonterminals.

The only possible action at a reduced state is a nonterminal expansion of the leftmost nonterminal. Expanding by a production with a non-empty pattern will produce a non-reduced state: reducing that state will also reduce the pattern so it will produce (if it produces anything) a state with a shorter pattern in the first component. We first consider the number of possible states which can be generated without changing the pattern. To obtain another reduced state from a reduced state the expansion of the leftmost nonterminal must use a production whose pattern is empty: in other words it must be an expansion by the default rule. Recall – crucially – that for each nonterminal there is *at most one default rule* and the lack of left recursion means that the length of this expansion sequence is bounded by the number of nonterminals, $|N|$. This because because, starting at any sentential form – as there is no left recursion – the first nonterminals in such an expansion sequence must always be distinct. Thus, the number of reduced states which can be generated by leftmost default expansions from a given string of nonterminals α is also bounded by $|N|$.

Once one has completely developed all the "default expansions" there are two possibilities for states:

(a) The third component has been completely reduced to the empty string. The only rule which now applies is the follow expansion. To account for this we simply count all the states which have a nonterminal postfix of a righthand side of a production as being in the search. That is all the states:

$$\mathsf{Starts}(\beta_0) = \{(\beta, x, \alpha) | \beta \ll \beta_0, x \to \beta'\alpha' \in \mathcal{G}, \alpha \ll \alpha'\}$$

This set conveniently includes the initial states of all the searches we may have to do on β_0.

(b) The third component is not yet reduced but there is an expansion by a rule $x \to \beta\alpha$ where the pattern β is nonempty. This introduces a branching factor bounded by the number of productions in the grammar, $|P|$, as we must expand a leading nonterminal and each of these occurs first in a default expansion at most once.

However, having done the branching in (b) notice that the size of the pattern $\beta \ll \beta_0$ being tested is strictly reduced. Thus the number of states produced from a state in $(\beta, x, \alpha) \in \mathsf{Starts}(\beta_0)$ not allowing transitions back to $\mathsf{Starts}(\beta_0)$ is bounded by $|P|^{|\beta|}$. This means the number of states generated by $\mathsf{Starts}(\beta_0)$ is bounded by:

$$\sum_{(\beta,x,\alpha)\in\mathsf{Starts}(\beta_0)} |P|^{|\beta|} \leq |\mathsf{Starts}(\beta_0)| \cdot |P|^{|\beta_0|}$$

which we may further simplify by observing

$$|\mathsf{Starts}(\beta_0)| \leq |\beta_0| \cdot |\{(x, \alpha) | x \to \beta\alpha'\alpha \in \mathcal{G}\}| \leq |\beta_0| \cdot |\mathcal{G}|.$$

This gives a simplified bound on the search from β_0 to be:

$$|\beta_0| \cdot |\mathcal{G}| \cdot |P|^{|\beta_0|}.$$

In the worst case we may have to initiate a search over all patterns in the grammar. So the overall cost will be bounded by:

$$\sum_{\beta \in G} |\beta_0| \cdot |\mathcal{G}| \cdot |P|^{|\beta_0|}$$

If the maximum length of a pattern is M_β and noting that the number of patterns is bounded by the number of productions, we can simplify this bound further to:

$$|P| \cdot M_\beta \cdot |\mathcal{G}| \cdot |P|^{|M_\beta|} = M_\beta \cdot |\mathcal{G}| \cdot |P|^{|M_\beta|+1}$$

This is exponential in the maximum size of the patterns – but for a fixed M_β is polynomial.

Or course, it must be emphasized that this exercise was aimed at providing an upper bound: the performance on an actual grammar will usually be significantly better.