

Notes on Recursive Descent Grammars

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

March 8, 2012

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 matching phrases 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* and only is applied after all the other rules have been tried.

Here is how the expression grammar gets translated into a recursive descent parser:

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

Notice how each production gets turned into code: each nonterminal becomes a function, each production rule becomes a phrase 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” behavior 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*.

2 Recursive descent form

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

$$x \rightarrow \beta\alpha$$

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 **first pattern** of the production. A production in recursive descent form has an **empty first pattern** when β is the empty string (and has a nonempty first 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

$$f \rightarrow A B h k$$

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

$$f(A : B : ins) = k(h(ins))$$

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

Of course we know the recursive descent parser produced by this translation will not always work (that is the translation is incomplete). 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:

<pre>exp -> mterm term. term -> mfactor factor. factor -> NUM VAR. mterm -> mterm term ADD . mfactor -> mfactor factor MUL .</pre>	becomes	<pre>exp -> mterm term. term -> mfactor factor. factor -> NUM VAR. mterm -> mterm term add . add -> ADD. mfactor -> mfactor factor mul . mul -> MUL.</pre>
---	----------------	---

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 production rules. We can then translate any grammar whose rules are in recursive descent form into code:

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

As we expected, there are a number of worrisome things about this code. To start with `mterm` and `mfactor` both implement bottomless recursion – so the code will not terminate (actually this will usually cause an overflow of the recursion stack!). Of course, we know what the problem is in this case: we must avoid having any left-recursion in the grammar. However, notice that there is also another problem: both of the `mterm` rules are now “default” rules ... but which should we now choose? Putting one ahead of the other will make the later one unreachable ... and, thus, potentially certain important parsing actions may become, in the code, unreachable. Thus, simply not being left-recursive is not enough to make the parsing decisions determined by a recursive descent grammar complete.

To make progress we should explain more carefully what we are trying to achieve. In fact, below we shall consider two notions: the first we shall call a **pattern determined grammar** and the second a **recursive descent grammar**. It is the latter in which we are particularly interested as these are precisely the grammars for which the above translation (when properly circumscribed) is complete.

The first notion is a relatively straight forward generalization of the notion of an LL(1) grammar except that one uses instead of tokens the first patterns¹ in the rules for look ahead. Thus, it is somewhat like an LL(k) grammar in which the k varies and, in fact, the only sequences allowed are those which already appear as first patterns in the (recursive descent form) rules. In particular, in a first pattern determined grammar it does not matter in which order the rules are written: what rule to apply is determined entirely by the current nonterminal and the first pattern (amongst those associated with the nonterminal) which matches the input – thus there is a finite action table.

One thing to note is that if there is at most one rule with an *empty* first pattern associated to each terminal then, when the grammar is first pattern determined, that rule can be treated as a default which is applied only when all others fail. Thus in creating the code all one has to do is to place this “default” rule last to make the above translation complete. This means we have partially solved our original problem.

A recursive descent grammar is more general as the order in which the patterns are matched now matters. It is precisely a grammar for which the above translation is complete when the translation places all rules with more specific first patterns first. This means a rule associated to a nonterminal is chosen only when its first pattern matches the current input *and* all the earlier rules have failed to match the input. In particular, this means there can certainly be at most one rule with no first pattern and, when such is present, it will act as the default “catch all” rule.

¹It is worth remarking that one cannot simply make these patterns into the new tokens to achieve the same effect. This is because one pattern can be a prefix of another causing some unexpected interactions.

The difficulty in determining whether a grammar is a recursive descent grammar arises as one wants no string which is recognized by a more specific rule to also match a less specific rule. Thus, unlike a pattern determined grammar, the first patterns (associated to a nonterminal) need no longer be “disjoint” because the less specific rules can be used to catch cases “by default”. This is a little more subtle as changing the order of the rules (placing a less specific rule above a more general rule) will make the corresponding recursive descent parser incomplete (i.e. fail to recognize some strings which should be recognized).

Below we describe necessary and sufficient conditions for a grammar to be pattern determined and to be recursive descent.

3 Pattern Determined Grammars

Our objective is to show 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.

3.1 Calculating the first patterns

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

A first pattern can be associated with a nonterminal x if it is the first pattern of a production with head x . However, if there are rules associated to x with an empty first pattern a rule with a first patter can be called indirectly. Thus if such a rule is $x \rightarrow f g$ we may apply any pattern which f can apply. However, also if f is nullable then we may apply any pattern g applies. Thus, we have to take careful account of the nullable nonterminals.

The calculation 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 relation.

Here is the calculation:

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

$$\text{IFirstPat}(x) = \{\beta | \beta \neq \varepsilon \ \& \ x \rightarrow \beta \alpha\}$$

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

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

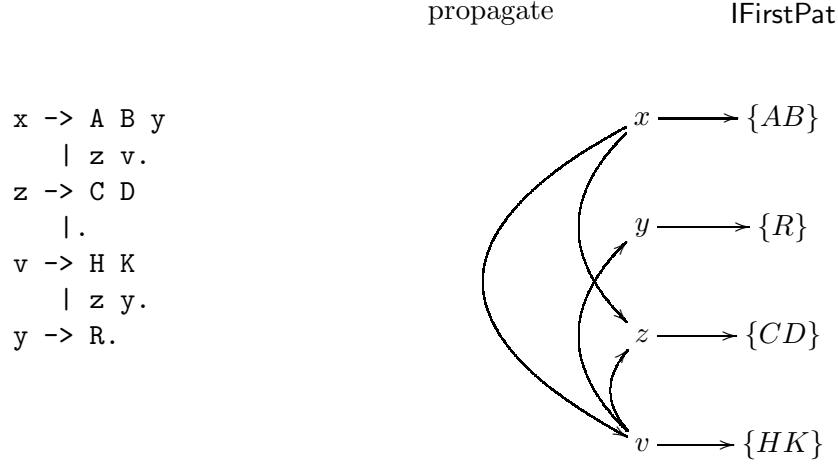
$$x \rightarrow y \Leftrightarrow x \rightarrow \alpha \cdot y \alpha \ \& \ \text{nullable}(\alpha)$$

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

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

$$\beta \in \text{FirstPat}(x) \Leftrightarrow \exists y. x \rightarrow^* y \ \& \ \beta \in \text{IFirstPat}(y)$$

Here is an example:



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

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

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

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

3.2 Conditions for being pattern determined

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

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

$$x \rightarrow \beta_i \alpha_i \quad x \rightarrow \beta_j \alpha_j$$

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

$$\text{FirstPat}(\beta_i \alpha_i) \sqcap \text{FirstPat}(\beta_j \alpha_j).$$

[PD.2] Every nonterminal, x , has at most one nullable production.

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

$$\text{FirstPat}(x) \sqcap \text{FollowPat}(x).$$

Here two pattern sets P_1 and P_2 are **disjoint** in case there is no string of terminals which matches a pattern from each set. 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. Thus two pattern sets are disjoint only if it happens that a pattern picked from one set is never a prefix of 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 last condition [PD.3] is discussed below, and before we turn to that consider the following example which illustrates why [RD.1] is necessary. Consider the following grammar and corresponding code:

$x \rightarrow A B y$ $\quad B z$ $\quad w.$ $w \rightarrow A x$ $\quad .$ $z \rightarrow D E.$ $y \rightarrow F.$	becomes code	$x(A:B:ins) = y(ins)$ $x(B:ins) = z(ins)$ $x(ins) = w(ins)$ $w(A:ins) = x(ins)$ $w(ins) = ins$ $z(D:E:ins) = ins$ $y(F:ins) = ins$
--	---------------------	--

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

$$x \rightarrow w \rightarrow A x \rightarrow A B z \rightarrow A B D E$$

but notice that the code fails on this as

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

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)!

3.3 Calculating follow patterns

The remaining problem is that 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:

$x \rightarrow A H$ $\quad B G$ $\quad w v.$ $w \rightarrow C x$ $\quad .$ $v \rightarrow C D$ $\quad C E.$	becomes code	$x(A:H:ins) = ins$ $x(B:G:ins) = ins$ $x(ins) = v(w(ins))$ $w(C;ins) = x(ins)$ $w(ins) = ins$ $v(C:D:ins) = ins$ $v(C:E:ins) = ins$
---	---------------------	---

Here the grammar recognizes $C E$ by:

$$x \rightarrow w \ v \rightarrow v \rightarrow C E$$

However, the code has the following effect:

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

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**:

$$\text{IFollowPat}(x) = \left\{ \beta \mid \begin{array}{l} z \rightarrow \beta' \alpha x \alpha' \\ \beta \in \text{FirstPat}(\alpha') \end{array} \right\}$$

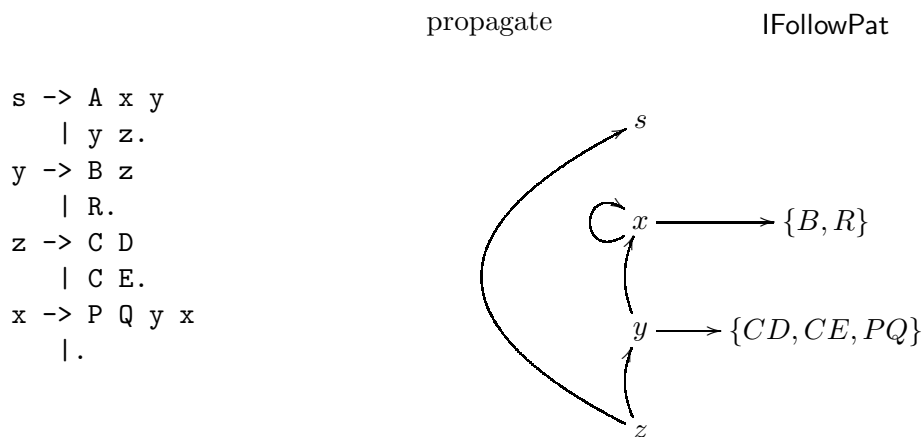
- Next calculate the **follow propagation graph**: the nodes of the graph are the nonterminals and the edges are given by:

$$x \rightarrow y \Leftrightarrow y \rightarrow \beta \alpha x \alpha' \ \& \ \text{nullable}(\alpha')$$

- Then

$$\beta \in \text{FollowPat}(x) \Leftrightarrow \exists y. x \rightarrow^* y \ \& \ \beta \in \text{IFollowPat}(y)$$

Here is an example of this calculation:



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

This grammar is recursive descent as its first pattern sets for production bodies with the same head are disjoint. Each terminal has at most one “default” production. Finally the first pattern set is disjoint from the follow pattern set of the nullable nonterminal.

4 Recursive Descent Grammars

An interesting variant of the first examples 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, AB cannot be the prefix of anything recognized by the production $x \rightarrow wz$. Thus, although the first sets are not disjoint when the prefix is AB this last production never results in a parse. So this does not cause a problem and the translation is complete.

4.1 Conditions for being recursive descent

Before we proceed we should discuss the translation procedure in a little more detail. In the program the order in which the rules are placed is clearly critical. However, in a grammar the rules actually do not have any order. However, when the rules are in recursive descent form we may preorder the rules according to the first patterns of the rule. A rule $x \rightarrow \beta\alpha$ is more specific than another rule $x \rightarrow \beta'\alpha'$ in case $\beta' \succ \beta$, 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 the less specific productions. Any order which satisfies this is an acceptable translation, and, indeed, the resulting program will be equivalent. Clearly to translate a more general production above a more specific production would simply make the more specific production unreachable in the code and thus the translation would be incomplete. Notice, however, that two rules cannot be *equally* specific otherwise putting the one above the other would cause the other to be blocked. In particular, there can be but one rule with an empty first pattern.

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

However, even if we are able to order the rules in the implementation suitably, it is still possible that a more specific production could be something that a later less specific production can also recognize. 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 $x \rightarrow \gamma$ **recognizes the pattern** β if there is a string continuing – recall that to reach the call of x already some of the input will have been eaten – with β which can be recognized by using that 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 the specificity pre-ordering.

[RD.3] The grammar must be left recursion free.

[RD.4] For any two rules $x \rightarrow \beta_1\alpha_1$ and $x \rightarrow \beta_2\alpha_2$ in which the first is more specific than the second the production $x \rightarrow \beta_2\alpha_2$ 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 prefix can be recognized

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 $x \rightarrow \gamma$ 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 $x \rightarrow \gamma$ if there are derivations (for some sentential forms δ, δ' , and δ''):

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

To achieve this we write a nondeterministic program which starts with the triple:

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

and generate triples from it. β will be recognized by the production $x \rightarrow \gamma$ 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:

$$(\beta, x, y\gamma) \Rightarrow \{(\beta, x, \delta\gamma) \mid y \rightarrow \delta \text{ is a production}\}$$

- Follow expansion:

$$(\beta, x, \varepsilon) \Rightarrow \{(\beta, y, \delta) \mid y \rightarrow \gamma x \delta \text{ is a production}\}$$

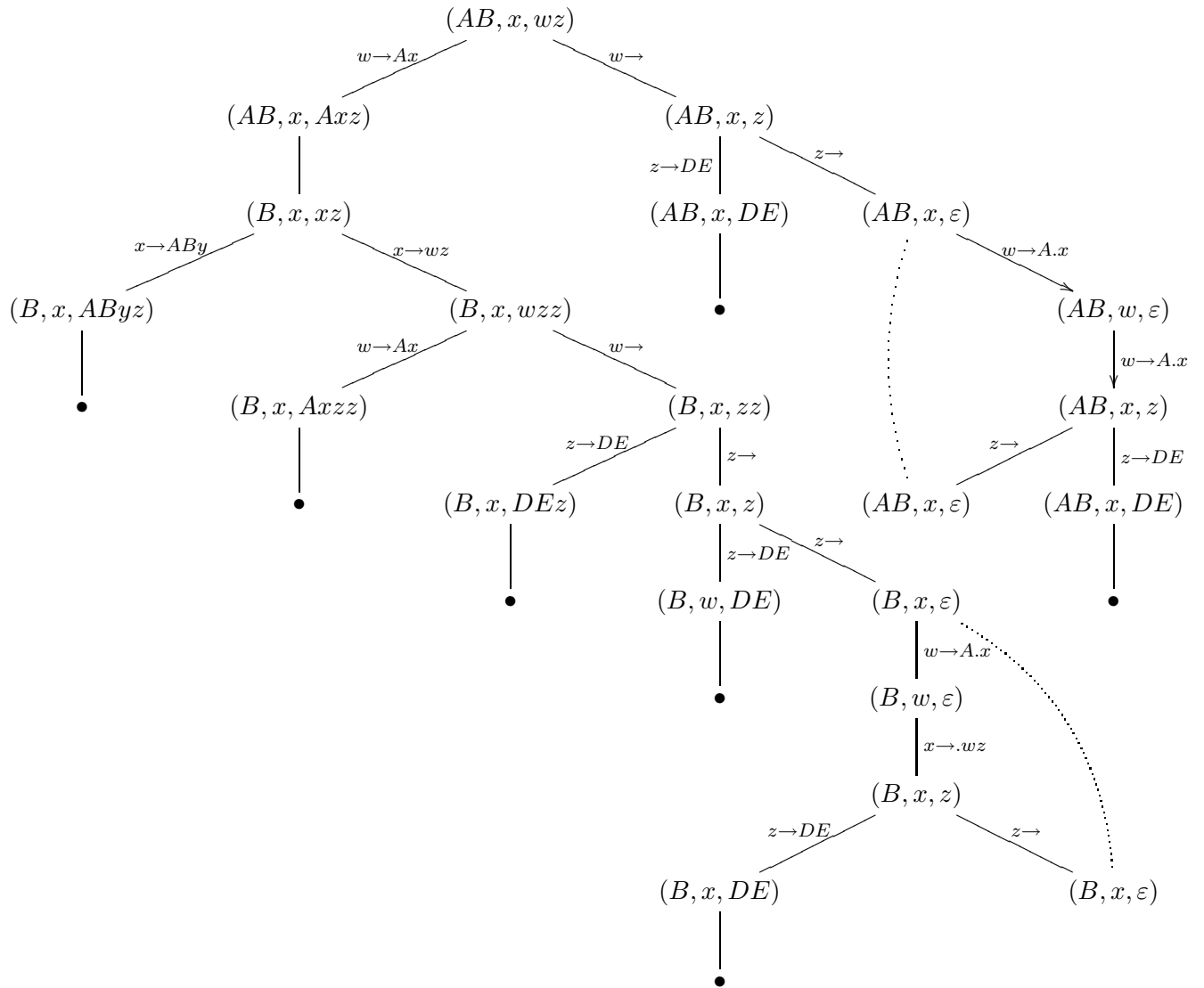
Note that if we apply these rules, because the grammar has no left recursion, eventually either (β, x, γ) will reduce β to the empty string (in which case we are done) or reduce the sentential form β to the empty sentential form. Thus, if we do not find a way of matching the pattern we will reach a stage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

When a triple is completed 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 this it is necessary to remember which completed triples have been generated so as to avoid repetition.

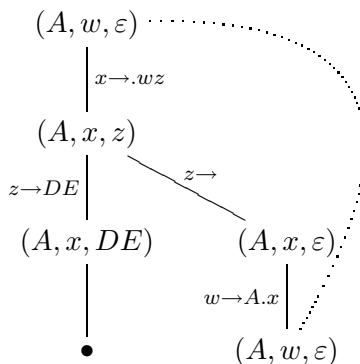
We may apply this to the grammar:

$x \rightarrow A B y$
 $\quad | w z.$
 $w \rightarrow A x$
 $\quad |.$
 $z \rightarrow D E$
 $\quad |.$
 $y \rightarrow F.$

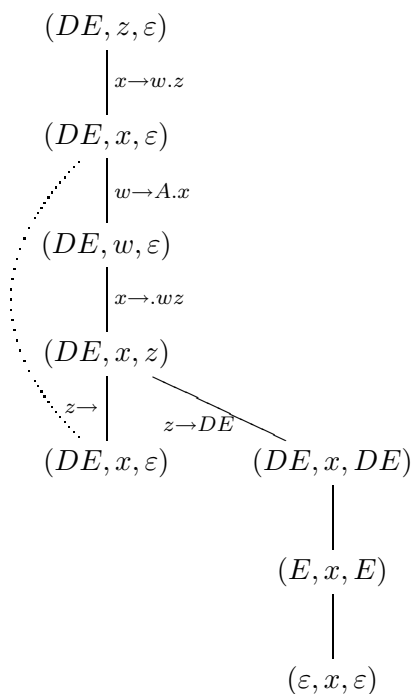
which is a slight variant of the grammar above. The first thing we must check is that the pattern AB cannot be recognized by the production $x \rightarrow wz$ here is the computation:



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



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



So in fact this grammar is *not* recursive descent because DE *can* be recognized by $z \rightarrow \varepsilon$! 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

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).
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. What happened to “no left-recursion” in a pattern determined grammar? The conditions we have supplied make no mention of left recursion! 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? The answer is possibly very bad!

Here is an assessment. Assuming the grammar is in recursive descent form the main source of complexity is the cost of doing a search to find out whether a rule could recognize a pattern. This is determined by the number of search states S_G which are possible for a grammar and, to allow for the cost of remembering and retrieving states for which it has been determined whether they recognize their pattern, requires order $S_G \log S_G$. A state is a triple a pattern, a nonterminal, and a sentential form. The patterns are postfixes of patterns which start the productions: the number of these is (generously) bounded by the size of the grammar itself. The sentential form is initially and after a follow expansion a postfixes of the righthand sides of a production associated with the nonterminal: thus, the number of these pairs is also bounded by the size of the grammar. However, one can also expand first nonterminals: as there is no left recursion one can do this at most a number of times bounded by the number of nonterminals before producing an initial terminal. In fact, at each step in order to *not* produce a terminal one must choose the unique production without a pattern. All this does means that the size of the sentential form is bounded. However, the expansion rules, it seems, can cause the search space to become exponential.