Chapter 2
Left Recursion by Recursive Ascent

Roman R. Redziejowski

Abstract Recursive-descent parsers can not handle left recursion, and several solutions to this problem have been suggested. This paper presents yet another solution. The idea is to modify recursive-descent parser so that it reconstructs left-recursive portions of syntax tree bottom-up, by "recursive ascent".

2.1 Introduction

Recursive-descent parser is a collection of "parsing procedures" that call each other recursively, mimicking the derivation process. This simple scheme cannot be applied if the grammar is left-recursive: a parsing procedure may indefinitely call itself, every time facing the same input. Some solutions have been suggested, and they are outlined in Sect. 2.9 at the end. We present here yet another approach. We view parsing as the process of reconstructing the syntax tree of given string. For this purpose, we equip parsing procedures with "semantic actions" that explicitly produce that tree. Recursive descent reconstructs syntax tree starting from the top. We suggest to reconstruct left-recursive portions of the tree bottom-up, by a process that we call "recursive ascent". This is also done by procedures that call each other recursively and have their own semantic actions. These procedures can be regarded as new parsing procedures. We incorporate them into the recursive-descent parser, so that recursive ascent is at the end carried out as part of recursive-descent parser for a new grammar, referred to as the "dual grammar". The dual grammar is not left-recursive, and its parser produces syntax tree for the original grammar.

In Sect. 2.2 we recall the necessary notions: BNF grammar, derivation, and syntax tree. In Sect. 2.3 we recall the idea of recursive-descent parser and introduce "semantic actions" that reconstruct the syntax tree. After introducing the necessary concepts in Sect. 2.4, we introduce in Sect. 2.5 the idea of recursive ascent together

R. R. Redziejowski (✉)
Ceremonimästarvägen 10, 181 40 Lidingö, Sweden
e-mail: roman@redz.se
URL: https://www.romanredz.se

B. Schlingloff et al. (eds.), *Concurrency, Specification and Programming, Studies in Computational Intelligence 1091*, https://doi.org/10.1007/978-3-031-26651-5_2

with procedures to perform it. They are illustrated by two examples in Sect. 2.6. In Sect. 2.7 we point out that the procedures just defined are parsing procedures for dual grammar. Two Propositions state the essential properties of that grammar. Section 2.8 looks at implementation of choice expressions, Sect. 2.9 outlines other known solutions. Some unsolved problems are discussed in Sect. 2.10. Proofs of the Propositions are given in the Appendix.

2.2 The Grammar

We consider a BNF-like grammar $G = (\mathbb{N}, \Sigma, \mathbb{E}, \mathcal{E}, N_s)$ with set $\mathbb{N}$ of *non-terminals*, set Σ of *terminals*, set $\mathbb{E}$ of *expressions*, function $\mathcal{E}$ from non-terminals to expressions, and *start symbol* $N_s \in \mathbb{N}$.
An expression is one of these:

- $a \in \Sigma$ ("terminal"),
- $N \in \mathbb{N}$ ("non-terminal"),
- $e_1 \ldots e_n$ ("sequence"),
- $e_1 | \ldots | e_n$ ("choice"),

where each of e_i is an expression. The function $\mathcal{E}$ is defined by a set of rules of the form $N \rightarrow e$, where e is the expression assigned by $\mathcal{E}$ to non-terminal N. We often write $N \rightarrow e$ to mean $e = \mathcal{E}(N)$. In the following, expressions $a \in \Sigma$ and $N \in \mathbb{N}$ will be viewed as special cases of choice expression with $n = 1$. We do not include empty string ε among expressions. The problems caused by it and suggested solutions are discussed in Sect. 2.10.

Non-terminal $N \rightarrow e_1 \ldots e_n$ *derives* the string $e_1 \ldots e_n$ of symbols, while $N \rightarrow e_1 | \ldots | e_n$ derives one of $e_1, \ldots, e_n$. The derivation is repeated to obtain a string of terminals. This process is represented by syntax tree. The set of all strings derived from $N \in \mathbb{N}$ is called the *language* of N and is denoted by $\mathcal{L}(N)$. Figures 2.1 and 2.2 are examples of grammar G, showing syntax trees of strings derived from the start symbol.

2.3 Recursive Descent

The idea of recursive-descent parsing is that each terminal and each non-terminal is assigned its *parsing procedure*. The procedure either *consumes* some prefix of given input, or fails and leaves the input unchanged. Thus:

- Procedure for terminal a consumes a if the input begins with a. Otherwise it fails.
- Procedure for $N \rightarrow e_1 \ldots e_n$ calls the procedures $e_1, \ldots, e_n$, in this order, and fails if any of them fails.

Fig. 2.1 Example of grammar G and syntax tree of `'xabbay'`

$\mathbb{N}$ = {Z,A,A1,B,B1,B2}
Σ = {a,b,x,y}
N_s = Z

Z → x A y
A → A1 | a
A1→ B a
B → B1 | B2 | b
B1→ A b
B2→ B b

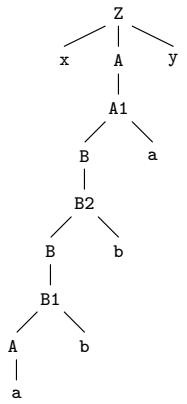
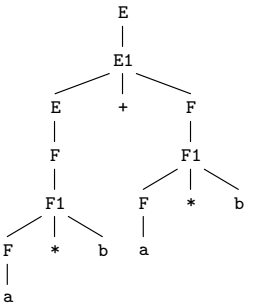


Fig. 2.2 Example of grammar G and syntax tree of `'a*b+a*b'`

$\mathbb{N}$ = {E,E1,F,F1}
Σ = {a,+,*}
N_s = E

E → E1 | F
E1→ E + F
F → F1 | a
F1→ F * b



- Procedure for $N \to e_1 | \ldots | e_n$ somehow selects and calls one of the procedures $e_1, \ldots, e_n$ that succeeds on the input. It fails if none of them can succeed. (The way of making this selection is discussed in Sect. 2.8.)

An important property of this method of parsing is close connection between the grammar and the parsing procedures. Namely, grammar rules $N \to e_1 \ldots e_n$ and $N \to e_1 | \ldots | e_n$ may be alternatively regarded as concise representations of parsing procedures, with N being the procedure's name and e_i representing a call to procedure e_i. The pattern of successful calls to parsing procedures mimics the derivation process, so successful call to procedure N consumes a prefix belonging to $\mathcal{L}(N)$. The information *how* the consumed string was derived is hidden in the pattern of procedure calls, and recorded in some unspecified way.

We begin our construction by making this recording explicit, and suggest that each successful parsing procedure returns syntax tree of the string it consumed. For this purpose, we add to each procedure its *semantic action*, executed after successful parsing. It is shown below enclosed in braces { }:

$$N \to e_1 \ldots e_n \ \{\texttt{return}\ [N \lhd [e_1] \ldots [e_n]]\} \tag{2.1}$$
$$N \to e_1 | \ldots | e_n \ \{\texttt{return}\ [N \lhd [e_i]]\} \tag{2.2}$$

We denote here by $[t]$ a syntax tree with top node t, and write $[t \lhd [t_1] \ldots [t_n]]$ to indicate that t has subtrees $[t_1], \ldots, [t_n]$ as children.

Each successfully called procedure e_i returns the syntax tree $[e_i]$. If e_i is for terminal a, it returns syntax tree $[a]$ of height 0. Procedure (2.1) returns a tree with top node N having as children the trees $[e_1], \ldots, [e_n]$ returned by called procedures. Procedure (2.2) returns a tree with top node N having as single child the tree $[e_i]$ returned by the selected procedure.

Parsing starts with invoking (2.1) or (2.2) for the start symbol. It results in recursive calls to subprocedures that eventually return syntax trees for the substrings they consume. At the end, procedure N_s builds and returns the final result.

Assuming that (2.2) makes its choice based only on input ahead, this does not work if the grammar contains left recursion. This is the case for both our examples. If `A` chooses `A1`, and `B` chooses `B1`, they are called again without any input being consumed in between, and thus bound to indefinitely make the same choices, never returning any result. The situation is similar for `E` and `F`.

2.4 Some Definitions

We need some definitions before suggesting how to handle left recursion.

For $N \in \mathbb{N}$ and $e \in \mathbb{N} \cup \Sigma$, define $N \xrightarrow{\text{first}} e$ to mean that procedure N called on some input may call that for e on the same input. We have thus:

- If $N \to e_1 \ldots e_n$, $N \xrightarrow{\text{first}} e_1$.
- If $N \to e_1 | \ldots | e_n$, $N \xrightarrow{\text{first}} e_i$ for $1 \le i \le n$.

Let $\xrightarrow{\text{First}}$ be the transitive closure of $\xrightarrow{\text{first}}$. Non-terminal $N \in \mathbb{N}$ is *(left) recursive* if $N \xrightarrow{\text{First}} N$. The set of all recursive non-terminals of G is denoted by $\mathbb{R}$. All non-terminals in Fig. 2.1 except `Z`, and all non-terminals in Fig. 2.2 are recursive.

Define relation between recursive N_1, N_2 that holds if $N_1 \xrightarrow{\text{First}} N_2 \xrightarrow{\text{First}} N_1$. This is an equivalence relation that partitions $\mathbb{R}$ into equivalence classes. We call them *recursion classes*. The recursion class of N is denoted by $\mathbb{C}(N)$. All non-terminals in Fig. 2.1 belong to the same class; the grammar of Fig. 2.2 has two recursion classes: `{E,E1}` and `{F,F1}`.

In syntax tree, the leftmost path emanating from any node is a chain of nodes connected by $\xrightarrow{\text{first}}$. Suppose N_1 and N_2 belonging to the same recursion class $\mathbb{C}$ appear on the same leftmost path. Any non-terminal N between them must also belong to $\mathbb{C}$, which follows from $N_1 \xrightarrow{\text{First}} N \xrightarrow{\text{First}} N_2 \xrightarrow{\text{First}} N_1$. It means that members of $\mathbb{C}$ appearing on the same leftmost path must form an uninterrupted sequence. We

call such sequence a *recursion path* of class $\mathbb{C}$. The only recursion path in Fig. 2.1 is the whole leftmost path from the first `A` without final `a`. The syntax tree in Fig. 2.2 has two recursion paths, one starting with `E` and another with `F`.

Let $N \rightarrow e_1 \ldots e_n$ be on a recursion path. The next item on the leftmost path is e_1, and it must belong to $\mathbb{C}(N)$ to ensure $N \xrightarrow{\text{First}} N$. It follows that the last item on a recursion path must be $N \rightarrow e_1 | \ldots | e_n$ where at least one of e_i is not a member of $\mathbb{C}(N)$. Such N is called an *exit* of $\mathbb{C}(N)$, and its alternatives outside $\mathbb{C}(N)$ are the *seeds* of $\mathbb{C}(N)$. In Fig. 2.1, both `A` and `B` are exits, and the seeds are `a` and `b`. In Fig. 2.2, `E` and `F` are exits of their respective classes, and the corresponding seeds are `F` and `a`.

A non-terminal that can be the start of a recursion path is called an *entry* of its recursion class. It is one of these:

- An element of $\mathcal{E}(N)$ for non-recursive N.
- One of $e_2, \ldots, e_n$ for recursive $N \rightarrow e_1 \ldots e_n$.
- A seed of another recursion class.
- The start symbol.

The recursion class in Fig. 2.1 has `A` as its entry. The recursion classes in Fig. 2.2 have `E` and `F` as their respective entries.

2.5 Recursive Ascent

To handle left recursion, we suggest that parsing procedure for entry E does not follow the pattern given by (2.1) and (2.2) above. It still returns syntax tree $[E]$, but builds this tree in a different way.

As noted in the preceding section, $[E]$ has recursion path starting with the entry node E and ending with the exit node followed by a seed. We start with the seed and ascend the recursion path, adding one node at a time. We reconstruct the side branches when the added node represents sequence expression. The tree being constructed is local to the invocation of procedure E. We call it "the plant", and denote it by $[\pi]$.

Sowing the Seed

Borrowing the way we used to represent parsing procedures, we outline the new entry procedure as:

$$E \rightarrow S\{\texttt{sow}\,[S]\}\;\texttt{grow}S\;\{\texttt{return}\,[\pi]\}\;. \tag{2.3}$$

Here, S represents call to parsing procedure for the seed S. If successful, it returns syntax tree $[S]$, and semantic action `sow` $[S]$ initializes $[\pi]$ with that tree. This is

followed by a call to new procedure `grow`S that continues parsing and grows the plant towards $[E]$. If it succeeds, the final semantic action returns the reconstructed tree. The whole procedure fails if S or `grow`S does.

In general, the recursion class $\mathbb{C}(E)$ may have more than one seed, so E has to choose one that matches the input. To represent this choice, we borrow the syntax of choice expression:

$$E \rightarrow (S_1\{\texttt{sow}\ [S_1]\}\ \texttt{grow}S_1 | \ldots | S_n\{\texttt{sow}\ [S_n]\}\ \texttt{grow}S_n)\ \{\texttt{return}\ [\pi]\}, \quad (2.4)$$

where $S_1, \ldots, S_n$ are all seeds of $\mathbb{C}(E)$.

Growing the Plant

The plant is grown by recursive procedures, each adding one node and then calling another procedure to continue growing. The procedure `grow`R that grows plant with top R can be sketched as follows:

$$\texttt{grow}R \rightarrow \texttt{add}P\ \texttt{grow}P. \quad (2.5)$$

It adds node P, and continues to grow the plant that has now P on top. The added node P is a predecessor of R in the recursion path, meaning $P \xrightarrow{\text{first}} R$. In general, R may have several predecessors P satisfying $P \xrightarrow{\text{first}} R$, so `grow`$R$ must choose one of them that matches the input. Again, we represent this in the way we used for choice expression:

$$\texttt{grow}R \rightarrow \texttt{add}P_1\ \texttt{grow}P_1 \mid \ldots \mid \texttt{add}P_n\ \texttt{grow}P_n \quad (2.6)$$

where $P_1, \ldots, P_n$ are all members of $\mathbb{C}(E)$ such that $P_i \xrightarrow{\text{first}} R$. We simplify this by introducing new procedure

$$\$P \rightarrow \texttt{add}P\ \texttt{grow}P, \quad (2.7)$$

so that (2.6) becomes:

$$\texttt{grow}R \rightarrow \$P_1 \mid \ldots \mid \$P_n. \quad (2.8)$$

The growing may stop when the plant reaches E. That means, (2.8) for E must have one more alternative, "do nothing", which we represent by ε:

$$\texttt{grow}E \rightarrow \$P_1 \mid \ldots \mid \$P_n \mid \varepsilon. \quad (2.9)$$

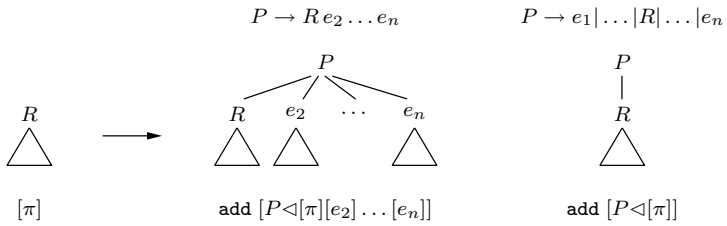


Fig. 2.3 add$P$

Adding a Node

Operation add$P$ is illustrated in Fig. 2.3. It adds node P to plant $[R]$.

If $P \to e_1 e_2 \ldots e_n$, the added node P will have the present plant as the first child. The other children are obtained by calling $e_2, \ldots, e_n$.
Operation add $[P \lhd [\pi][e_2] \ldots [e_n]]$ shown in the Figure adds node P with these children to the plant.

If $P \to e_1 | \ldots | e_n$, the added node P has only one child, namely the present plant.
Operation add $[P \lhd [\pi]]$ shown in the Figure adds such node P to the plant.

Inserting these operations in (2.7), we obtain:

$$\$P \to e_2 \ldots e_n \ \{\texttt{add}\ [P \lhd [\pi][e_2] \ldots [e_n]]\}\ \texttt{grow}P \quad \text{if } P \to e_1 \ldots e_n, \tag{2.10}$$

$$\$P \to \{\texttt{add}\ [P \lhd [\pi]]\}\ \texttt{grow}P \quad \text{if } P \to e_1 | \ldots | e_n. \tag{2.11}$$

Multiple Entries

The above assumed that each recursion class has only one entry. This is not true for many grammars; for example, the class of Primary in Java has four entries. Multiple entries can be handled because calls to entry procedures are nested, so one can keep track of which one is currently active. The exit alternative ε in grow$E$ must be applied only if E is the currently active one.

2.6 Examples

Example 1 Applying (2.1), (2.4), (2.8)–(2.9), (2.10)–(2.11) to the grammar of Fig. 2.1, replacing calls for "grow" by their definitions, and omitting unused procedures, we obtain the procedures shown in Fig. 2.4. We apply them to string 'xabbay', showing how they construct the syntax tree appearing in the Figure. Numbers on the left indicate top of the plant after each step. The procedures for

```
Z  → x A y {return [Z◁[x][A][y]]}
A  → a {sow [a]} $A {return [π]}
   | b {sow [b]} $B {return [π]}
$A → {add [A◁[π]]} ($B1 | ε)
$B → {add [B◁[π]]} ($A1 | $B2)
$A1→ a {add [A1◁[π][a]]} $A
$B1→ b {add [B1◁[π][b]]} $B
$B2→ b {add [B2◁[π][b]]} $B
```

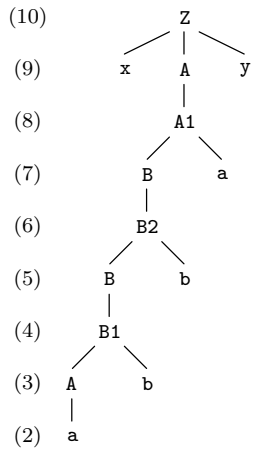


Fig. 2.4 Procedures and example of parsing for grammar of Fig. 2.1

choice expressions are assumed to always make correct choice using some oracle; this will be discussed later in Sect. 2.8.

1. The parsing starts with procedure for non-recursive `Z` that, in the usual way, calls the procedures `x`, `A`, and `y`. After consuming `'x'`, `Z` applies `A` to `'abbay'`.
2. Procedure `A` chooses its first alternative, applies `a` to `'abbay'`, leaving `'bbay'`, and initializes [π] with [`a`]. Then it calls `$A`.
3. Procedure `$A` adds node `A` on top of [`a`]. Then it chooses to call `$B1`.
4. Procedure `$B1` applies `b` to `'bbay'`, leaving `'bay'`, and creates node `B1` with children [`A`] and [`b`]. Then it calls `$B`.
5. Procedure `$B` adds node `B` on top of [`B1`]. Then it chooses to call `$B2`.
6. Procedure `$B2` applies `b` to `'bay'`, leaving `'ay'`, and creates node `B2` with children [`B`] and [`b`]. Then it calls `$B`.
7. Procedure `$B` adds node `B` on top of [`B2`]. Then it chooses to call `$A1`.
8. Procedure `$A1` applies `a` to `'ay'`, leaving `'y'`, and creates node `A1` with children [`B`] and [`a`]. Then it calls `$A`.
9. Procedure `$A` adds node `A` on top of [`A1`]. Then it chooses ε, and returns, causing the invoked `$A1, $B, $B2, $B, $B1, $A` to terminate one after another. Finally, it returns [`A`] to `Z`.
10. Procedure `Z` consumes `'y'` and returns the tree `[Z◁[x][A][y]]`.

Example 2 For example of Fig. 2.2, we obtain the procedures shown in Fig. 2.5. We apply them to string `'a*b+a*b'`, showing how they construct the syntax tree appearing in the Figure. Numbers indicate top of the plant after each step.

1. Procedure `E` calls parsing procedure for its seed `F`.
2. Procedure `F` calls parsing procedure for its seed `a`, which consumes `'a'`, leaving `'*b+a*b'`.
   Then it initializes its plant with `[a]` and calls `$F`.

```
E  → F {sow [F]} $E {return [π]}
$E → {add [E◁[π]]} ($E1 | ε)
$E1→ + F {add [E1◁[π][+][F]]} $E

F  → a {sow [a]} $F {return [π]}
$F → {add [F◁[π]]} ($F1 | ε)
$F1→ * b {add [F1◁[π][*][b]]} $F
```

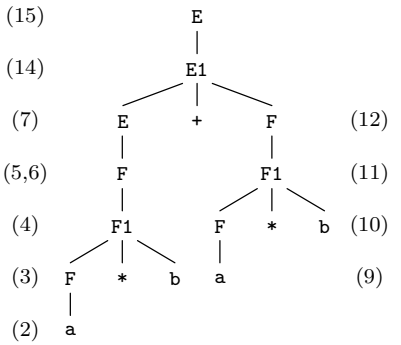


Fig. 2.5 Procedures and example of parsing for grammar of Fig. 2.2

3. Procedure `$F` adds `F` on top of `[a]` and chooses to call `$F1`.
4. Procedure `$F1` calls procedures `*` and `b` that consume `'*b'`, leaving `'+a*b'`. Then it adds `[F1 ◁ [F][*][b]]` on top of `[F]` and calls `$F`.
5. Procedure `$F` adds `F` on top of `[F1]` and chooses ε, causing the invoked `$F1` to terminate.
   Then it returns to procedure `F`.
6. Procedure `F` returns to `E` the syntax tree `[F]` of `'a*b'`.
   Procedure `E` initializes its plant with `[F]`. What has been the plant of `F` becomes now the seed of `E`. Procedure `E` calls then `$E`.
7. Procedure `$E` adds `E` on top of `[F]` and decides to call `$E1`.
8. Procedure `$E1` calls procedures `+` and `F`. The first consumes `'+'`, leaving `'a*b'`.
9. Procedure `F` calls parsing procedure for its seed `a`, which consumes `'a'`, leaving `'*b'`. It initializes its own plant with `[a]`. We have now two plants: that of procedure `E`, for the moment left waiting, and the new plant of `F`. Procedure `F` calls `$F`.
10. Procedure `$F` adds `F` on top of `[a]`. Then it chooses to call `$F1`.
11. Procedure `$F1` calls procedures `*` and `b` that consume the remaining `'*b'`. Then it adds `[F1 ◁ [F][*][b]]` on top of `[F]` and calls `$F`.
12. Procedure `$F` adds `F` on top of `[F1]` and chooses ε, causing the invoked `$F1` to terminate.
    Then it returns to procedure `F`.
13. Procedure `F` returns to `E` the syntax tree `[F]` of `'a*b'`.
14. Procedure `$E1` adds `E1 ◁ [E][+][F]` on top of `[E]`. The plant of `F` becomes now a branch of the plant of `E`. The procedure calls then `$E`.
15. Procedure `$E` adds `E` on top of `$E1` and chooses ε, causing the invoked `$E1` to terminate. Then it returns to procedure `E`.
16. Procedure `E` terminates, returning the constructed tree `[E]`.

2.7 The Dual Grammar

Consider now the procedures in Fig. 2.4 with their semantic actions made invisible:

```
Z   → x A y
A   → a $A | b $B
$A  → $B1 | ε
$B  → $A1 | $B2
$A1→ a $A
$B1→ b $B
$B2→ b $B
```

This is a new grammar with the set `{Z,A,$A,$B,$A1,$B1,$B2}` of non-terminals. Reversing what was said at the beginning of Sect. 2.3, we discover that procedures in Fig. 2.4 are parsing procedures for this grammar. It means that our parsing with recursive ascent is actually performed as recursive descent for a new grammar. This new grammar is in the following referred to as *dual grammar* for the grammar of Fig. 2.1. The procedures differ from (2.1)–(2.2) by having special semantic actions that construct syntax tree according to the original grammar. The dual grammar for grammar of Fig. 2.2 is:

```
E   → F $E
$E  → $E1 | ε
$E1→ + F $E
F   → a $F
$F  → $F1 | ε
$F1→ * a $F
```

One can see that dual grammar is in general constructed as follows:

- For each entry E create rule $E \rightarrow S_1\ \texttt{grow}S_1 \mid \ldots \mid S_n\ \texttt{grow}S_n$,
- Replace each recursive $R \rightarrow e_1 e_2 \ldots e_n$ by $\$R \rightarrow e_2 \ldots e_n\ \texttt{grow}R$,
- Replace each recursive $R \rightarrow e_1 \mid \ldots \mid e_n$ by $\$R \rightarrow \texttt{grow}R$,
- Replace $\texttt{grow}R$ by $\$P_1 \mid \ldots \mid \$P_m$ if $R \neq E$,
- Replace $\texttt{grow}E$ by $\$P_1 \mid \ldots \mid \$P_m \mid \varepsilon$,

where

- $S_1, \ldots, S_n$ are all seeds of $\mathbb{C}(E)$,
- $P_1, \ldots, P_m$ are all members of $\mathbb{C}(E)$ such that $P_i \xrightarrow{\text{first}} R$.

The dual grammar is an n-tuple $D = (\mathbb{N}_D, \Sigma, \mathbb{E}_D, \mathcal{E}_D, N_s)$. Its set $\mathbb{N}_D$ consists of:

- The non-recursive members of $\mathbb{N}$;
- The entries to recursion classes;
- The new non-terminals with $-names.

In the following, the set of all non-recursive members of $\mathbb{N}$ is denoted by $\overline{\mathbb{R}}$, and the set of all entries by $\mathbb{R}_E$. They appear as non-terminals in both G and D. The set $\overline{\mathbb{R}} \cup \mathbb{R}_E$ of these common symbols is denoted by $\mathbb{N}_C$. The language $\mathcal{L}_D(N)$ of $N \in \mathbb{N}_D$ is the set of all terminal strings derived from N according to the rules of D.

The two important facts about the dual grammar are:

Proposition 2.1 *The dual grammar is left-recursive only if the original grammar contains a cycle, that is, a non-terminal that derives itself.*

Proof is found in Appendix 1.

Proposition 2.2 $\mathcal{L}_D(N) = \mathcal{L}(N)$ *for all* $N \in \mathbb{N}_C$.

Proof is found in Appendix 2.

2.8 Implementing the Choices

One can see that the choices (2.4) and (2.8)–(2.9) became standard choice expressions in the dual grammar. The dual grammar contains also choice expressions inherited from the original grammar. So far, we did not say anything about how procedures for these expression decide which alternative to choose.

A safe way is to systematically check all choices, backtracking after each failure, until we find the correct one (or find that there is none). This, however, may take exponential time.

There exist shortcuts that give reasonable processing time. The problem is that each is applicable only to a restricted class of grammars. If D is outside this class, the parser will reject some strings of $\mathcal{L}_D(N_s)$.

One shortcut is to look at the next input terminal. This is very fast, and involves no backtracking, but requires that strings derived from different alternatives begin with different terminals. This is known as the LL(1) property. (As a matter of fact, both dual grammars in our examples have this property.)

Another shortcut, not much investigated, is to look at input within the reach of one parsing procedure. It requires that some procedures eventually called from different alternatives never succeed on the same input. This property of the grammar is suggested to be called LL(1p) [8].

Yet another one is limited backtracking, called "fast-back" in [5], page 57, and recently exploited in Parsing Expression Grammar [3]. Here the choice expression tries different alternatives and accepts the first successful one. Backtracking is limited to sequence expressions. Using the technique of "packrat parsing" [2], one can make it work in linear time.

Limited backtracking requires that all choices after first successful one must lead to failure. We say that grammar having this property is fast-back safe. A sufficient condition for fast-back safety is given in [9]. (Note that this property depends on the order of alternatives in choice expressions.)

Proposition 2.2 implies that if grammar D is safe for the chosen method (LL(1), LL(1p), respectively fast-back safe), the recursive-descent parser for D is a correct parser for G.

A subject of further research is how to translate the above properties of grammar D into equivalent properties of grammar G.

2.9 Previous Work

Our method is inspired by one outlined by Hill in [4]. His report describes only the algorithm, but it is clear that it follows from the idea of reconstructing the recursion path. Each entry of recursion class is represented by data structure called the "grower". The grower contains all seeds of its recursion class, and for each seed, the chains of its parents. Parsing procedure for the entry consists of interpreting the grower; it starts with the seeds and follows their parent chains.

The traditional way of eliminating left recursion is to rewrite the grammar so that left recursion is replaced by right recursion. It can be found, for example, in [1], page 176. It consists of replacing

$$\texttt{A} \rightarrow \texttt{A a} \mid \texttt{b} \tag{2.12}$$

by

$$\begin{aligned} \texttt{A} &\rightarrow \texttt{b A}' \\ \texttt{A}' &\rightarrow \texttt{a A}' \mid \varepsilon\,. \end{aligned}$$

Below is an example of this rewriting for the grammar given in [11] as an example of intersecting left-recursion loops:

```
                    E→ F n | n
E → F n | n         F → (n + x | G -) F'
F → E + x | G -     F'→ (n + x) F' | ε
G → H m | E         G → (H m | n | (n + x) F' n) G'
H → G y             G'→ (- F' n) G' | ε
                    H → (n G' y | (n + x) F' n G' y) H'
                    H'→ (m G' y) H' | ε
```

The process is cumbersome and produces large results. This can be compared with our dual grammar, which is obtained by a rather straightforward rewriting of syntax rules:

```
E  → n $E
$E → + x $F | $G | ε
$F → n $E
$G → - $F | $H
$H → y m $G
```

The main problem with the traditional rewrite is that it completely loses the original intent of the grammar. The same can be said about our dual grammar, but the difference is that our recursive descent for dual grammar produces correct syntax tree according to the original grammar.

In [13], Warth and al. suggested how to handle left recursion in Parsing Expression Grammar (PEG) by modifying the packrat parser. The principle of packrat parsing is that the result of each parsing procedure is saved in a "memoization table". The table is consulted before each new call, and if a result is found for the same position in input, it is used directly instead of calling the procedure.

The idea can be explained on the example grammar (2.12). Before the first call to `A`, we save "failed" as the result of `A` at the current position. Procedure `A` starts with the alternative `Aa` which requires calling `A` and then `a`. But the result of calling `A` is already in the table; as it is "failed", the alternative `Aa` fails, and `A` calls `b`. If it succeeds, we save `'b'` as the new result of `A`.

We backtrack and call `A` again. Alternative `Aa` obtains now `'b'` as the result of `A` and calls `a`. If it succeeds, we save `'ba'` as the new result of `A`. This is referred to as "growing the seed".

We repeat this as long as `a` in `Aa` succeeds. When it fails, we stop the repetition and are left with the result `'ba...a'` of `A` in the table.

In [12], Tratt indicated that this method does not require packrat parser, only a table for results of left-recursive procedures. Also, that left-recursive non-terminals can be detected by analyzing the grammar.

Medeiros et al. [7] introduced the notion of "bounded left recursion". The idea can again be explained on the example grammar (2.12). Procedure `A` is required to try alternative `Aa` exactly n times before choosing `b`. The string thus consumed is saved as $\mathtt{A}^n$. We call procedure `A` repeatedly with increasing values of the "bound" n, obtaining $\mathtt{A}^0$, $\mathtt{A}^1$, $\mathtt{A}^2$, etc..

Left part of Fig. 2.6 illustrates how $\mathtt{A}^2$ is obtained for input `'baax'`. The arrows represent successful calls. One can see from the Figure how this is obtained from saved $\mathtt{A}^1$ = `'ba'`, and this from $\mathtt{A}^0$ = `'b'`.

Right part of Fig. 2.6 shows what happens for $n = 3$. Dotted arrows represent failing calls. The `a` in the highest `Aa` encounters `'x'` and fails, and so does `Aa`. Procedure `A`, receiving failure from `Aa`, consumes `'b'`, giving $\mathtt{A}^3$ = `'b'`. This is the signal to stop repetition and return the saved $\mathtt{A}^2$.

The methods from [7, 12, 13] use memoization table; they repeatedly evaluate the left-recursive procedure in the process of growing the seed or incrementing the bound. None of these is the case with our approach.

All methods [4, 7, 12, 13] must handle situations more complex than the elementary example (2.12), like indirect and nested left recursion. They add a substantial amount of overhead to the mechanism of procedure call. This is not the case with our method, where all complexity is delegated to the task of constructing the dual grammar. The only overhead are the semantic actions of constructing the syntax tree, but they must also be invisibly included in the above methods to record the pattern of procedure calls.

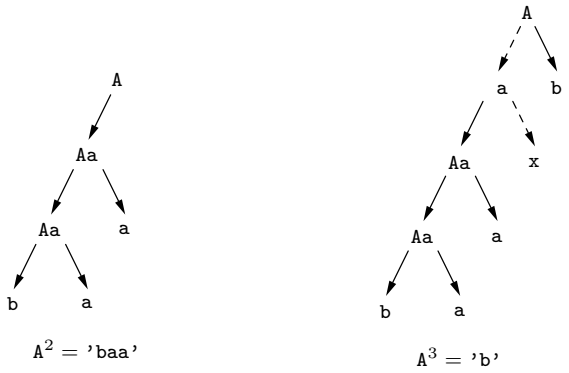


Fig. 2.6 Example of bounded left recursion applied to `'baax'`

In principle, we have a recursive-descent parser with simple calls to parsing procedures. If so desired, it can use limited backtracking implemented with packrat technology.

In addition, we offer a way to see if the chosen method of handling choice expressions results in correct parser. None of the cited solutions gives any hint on how to do it.

2.10 Unsolved Problems

We did not include empty word ε in our grammar. (It appeared in the dual grammar, but only with a special function.) The result of adding ε is that some expressions may derive empty string. These expressions are referred to as *nullable*, and can be identified as such by analyzing the grammar.

Nullable e_1 in $N \to e_1 \ldots e_n$ and nullable e_i in $N \to e_1 | \ldots | e_n$ extend the definition of $\xrightarrow{\text{first}}$ beyond the two rules given in Sect. 2.4. As the result, non-terminal may be recursive or not depending on input.

An example is $\mathtt{A} \to \mathtt{(a} \mid \varepsilon \mathtt{)Ab} \mid \mathtt{c}$ where `A` is left-recursive only if input does not start with `a`. It is known as "hidden" left recursion. This specific case can be solved by redefining the grammar as

```
A  →  (Ab|c)  |  (aAb|c),
```

but we do not provide a general solution. One can always allow ε under condition that it does not affect the set of recursive non-terminals, which can be checked by examining the grammar.

Nullable seed and nullable $e_2 \ldots e_n$ in recursive $N \to e_1 e_2 \ldots e_n$ invalidate the proof of Proposition 2.1. That means the dual grammar can be left-recursive even if G does not have a cycle. It requires a specific check for D being left-recursive.

Another kind of unsolved problem is `E` → `E+E | n`, which results in a right-recursive parse for `E`. This is so because the second `E` invoked by (2.10) gobbles up all input before the ascent can continue.

The problem with `E` → `E+E | n` was signalled by Tratt in [12]. In fact, the right-recursive result is correct because the grammar is ambiguous. Tratt explains why the left-recursive parse is desirable, and offers a solution for direct left recursion.

A solution proposed in [7] extends the grammar by assigning "priorities" to different occurrences of `E`. This opens new possibilities: a simple way of defining operator precedence. It is further exploited in [6].

Acknowledgements This version is the result of comments from two anonymous referees and comments to presentation at CS&P'2021 workshop [10].

2.11 Appendix 1: Proof of Proposition 2.1

For $N \in \mathbb{N}_D$ and $e \in \mathbb{N}_D \cup \Sigma$, define $N \xrightarrow{\text{firstD}} e$ to mean that parsing procedure N may call parsing procedure e on the same input:

(a) For $N \in \overline{\mathbb{R}}$, $\xrightarrow{\text{firstD}}$ is the same as $\xrightarrow{\text{first}}$.
(b) For $N \in \mathbb{R}_E$, $N \xrightarrow{\text{firstD}} S$ for each seed S of $\mathbb{C}(N)$.
(c) For $\$R \to e_2 \ldots e_n \texttt{grow}R$, $\$R \xrightarrow{\text{firstD}} e_2$.
(d) For $\$R \to \texttt{grow}R$, $\$R \xrightarrow{\text{firstD}} \$P_i$ for each $P_i \in \mathbb{C}(R)$ such that $P_i \xrightarrow{\text{first}} R$.

Suppose D is left-recursive, that is, there exist $N_1, N_2, \ldots, N_k \in \mathbb{N}_D$ where $N_1 \xrightarrow{\text{firstD}} \ldots N_2 \xrightarrow{\text{firstD}} \ldots \xrightarrow{\text{firstD}} N_k$ and $N_k = N_1$. We start by showing that none of them can be in $\mathbb{N}_C$.

Assume that $N_1 \in \mathbb{N}_C$. That means N_1 is either in $\overline{\mathbb{R}}$ or in $\mathbb{R}_E$.

Suppose $N_1 \in \overline{\mathbb{R}}$. Because N_1 is the same in both grammars, we have $\mathcal{E}_D(N_1) = \mathcal{E}(N_1)$, so N_2 is also in $\mathbb{N}$, and thus in $\mathbb{N}_C$. From $N_1 \xrightarrow{\text{firstD}} N_2$ follows $N_1 \xrightarrow{\text{first}} N_2$.

Suppose $N_1 \in \mathbb{R}_E$. According to (b), N_2 is a seed of $\mathbb{C}(N_1)$, which is either non-recursive or an entry, and thus is in $\mathbb{N}_C$. For a seed N_2 of $\mathbb{C}(N_1)$ holds $N_1 \xrightarrow{\text{First}} N_2$.

The above can be repeated with $N_2, \ldots, N_{k-1}$ to check that $N_i \in \mathbb{N}_C$ for all $1 \le i \le k$ and $N_i \xrightarrow{\text{First}} N_{i+1}$ for $1 \le i < k$. As $N_1 = N_k$, the latter means that none of N_i belongs to $\overline{\mathbb{R}}$, so they must all be in $\mathbb{R}_E$. Moreover, they are all in the same recursion class, $\mathbb{C}(N_1)$. According to (b), N_2 is a seed of $\mathbb{C}(N_1)$, that cannot be a member of $\mathbb{C}(N_1)$. This is a contradiction, so $N_1 \notin \mathbb{N}_C$, and this holds for the remaining $N_2, \ldots, N_{k-1}$.

It follows that each N_i is a \$-non-terminal of D. It cannot be that listed under (c) because then N_{i+1} is e_2 which belongs to $\mathbb{N}_C$.

Hence, each N_i must have the form listed under (d). It follows that if $N_i = \$R_i$ for some R_i, N_{i+1} must be $\$R_{i+1}$ such that $R_{i+1} \xrightarrow{\text{first}} R_i$. Thus, exists in grammar

G a sequence of non-terminals $R_k \xrightarrow{\text{first}} \ldots \xrightarrow{\text{first}} R_2 \xrightarrow{\text{first}} R_1$ with $R_k = R_1$. Each of them has choice statement as $\mathcal{E}(R_i)$ and can derive itself in $k-1$ steps. □

2.12 Appendix 2: Proof of Proposition 2.2

We show that each string derived from $N \in \mathbb{N}_C \cup \Sigma$ according to grammar G can be derived according to grammar D and vice-versa. We say that derivation has height h to mean that its syntax tree has height h. The proof is by induction on that height.

Induction base is the same in both directions: derivation of height 0. We consider terminal to be derived from itself with syntax tree of height 0. The G- and D-derivations are identical.

The following two lemmas provide induction step for the two derivations.

Lemma 2.1 *Assume that each string having G-derivation of height $h \geq 0$ from $N \in \mathbb{N}_C \cup \Sigma$ has a D-derivation from N. Then the same holds for each string with G-derivation of height $h+1$.*

Proof Take any string w having G-derivation from $N \in \mathbb{N}_C$ of height $h+1$.

(Case 1) $N \in \overline{\mathbb{R}}$.

If $N \rightarrow e_1 \ldots e_m$, all e_j are in $\mathbb{N}_C$ and $w = w_1 \ldots w_m$ where each w_j is G-derived from e_j by derivation of height h or less. We D-derive $e_1 \ldots e_m$ from N, and then, according to induction hypothesis, D-derive $w = w_1 \ldots w_m$ from $e_1 \ldots e_m$.

If $N \rightarrow e_1 | \ldots | e_m$, w is G-derived from one of e_j by derivation of height h or less. We D-derive e_j from N, and then, according to induction hypothesis, D-derive w from e_j.

(Case 2) $N \in \mathbb{R}_E$.

The G-derivation of w is shown in the left part of Fig. 2.7. One can see that $w = w_0 \ldots w_n$ where w_0 is G-derived from a seed and for $1 \leq i \leq$, w_i is either G-derived from $e_2 \ldots e_m$ (if $R_i \rightarrow e_1 \ldots e_m$) or is empty word ε (if $R_i \rightarrow e_1 | \ldots | e_m$). All these G-derivations are from members of $\mathbb{N}_C$ and have height h or less. Thus, by induction hypothesis, exist corresponding D-derivations of $w_0, \ldots, w_n$. The D-derivation of w using these results is shown as the right part of Fig. 2.7. □

Lemma 2.2 *Assume that each string having D-derivation of height $h \geq 0$ from $N \in \mathbb{N}_C \cup \Sigma$ has a G-derivation from N. Then the same holds for each string with D-derivation of height $h+1$.*

Proof (The proof is a mirror image of the proof of Lemma 2.1, but we spell it out.) Take any string w having D-derivation from $N \in \mathbb{N}_C$ of height $h+1$.

(Case 1) $N \in \overline{\mathbb{R}}$.

If $N \rightarrow e_1 \ldots e_m$, all e_j are in $\mathbb{N}_C$ and $w = w_1 \ldots w_m$ where each w_j is D-derived from e_j by derivation of height h or less. We G-derive $e_1 \ldots e_m$ from N, and then, according to induction hypothesis, G-derive $w = w_1 \ldots w_m$ from $e_1 \ldots e_m$.

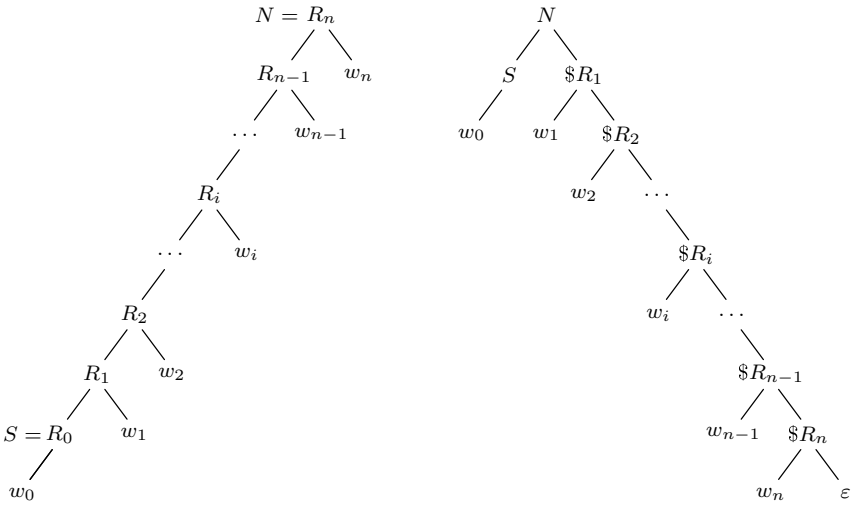


Fig. 2.7 G-derivation and D-derivation from E. w_0 is derived from the seed. Otherwise w_i is derived from $e_2, \ldots, e_m$ for sequence R_i or ε for choice R_i

If $N \to e_1 | \ldots | e_m$, w is D-derived from one of e_j by derivation of height h or less. We G-derive e_j from N, and then, according to induction hypothesis, G-derive w from e_j.
(Case 2) $N \in \mathbb{R}_E$.

The D-derivation of w is shown in the right part of Fig. 2.7. One can see that $w = w_0 \ldots w_n$ where w_0 is D-derived from a seed and for $1 \leq i \leq$, w_i is either D-derived from $e_2 \ldots e_m$ (if $R_i \to e_1 \ldots e_m$) or is empty word ε (if $R_i \to e_1 | \ldots | e_m$). All these D-derivations are from members of $\mathbb{N}_C$ and have height h or less. Thus, by induction hypothesis, exist corresponding G-derivations of $w_0, \ldots, w_n$. The G-derivation of w using these results is shown as the left part of Fig. 2.7. □

References

1. Aho, A.V., Sethi, R., Ullman, J.D.: Compilers, Principles, Techniques, and Tools. Addison-Wesley (1987)
2. Ford, B.: Packrat parsing: simple, powerful, lazy, linear time, functional pearl. In: Wand, M., Jones, S.L.P., (eds.) Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming (ICFP '02), Pittsburgh, Pennsylvania, USA, October 4–6, 2002, pp. 36–47. ACM (2002)
3. Ford, B.: Parsing expression grammars: A recognition-based syntactic foundation. In: Jones, N.D., Leroy, X., (eds.) Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2004, pp. 111–122. ACM, Venice, Italy (2004)
4. Hill, O.: Support for Left-Recursive PEGs (2010). https://github.com/orlandohill/peg-left-recursion
5. Hopgood, F.R.A.: Compiling Techniques. MacDonald/Elsevier (1969)

6. Laurent, N., Mens, K.: Parsing expression grammars made practical. In: SLE 2015: Proceedings of the 2015 ACM SIGPLAN International Conference on Software Language Engineering, pp. 167–172 (2015)
7. Medeiros, S., Mascarenhas, F., Ierusalimschy, R.: Left recursion in parsing expression grammars. Sci. Comput. Program. **96**, 177–190 (2014)
8. Redziejowski, R.R.: From EBNF to PEG. Fund. Inf. **128**, 177–191 (2013)
9. Redziejowski, R.R.: More about converting BNF to PEG. Fund. Inf. **133**(2–3), 177–191 (2014)
10. Redziejowski, R.R.: Left recursion by recursive ascent. In: Schlingloff, H., Vogel, T., (eds.), Proceedings of the 29th International Workshop on Concurrency, Specification and Programming (CS & P 2021), pp. 72–82 (2021). http://ceur-ws.org/Vol-2951
11. Sigaud, P.: Left recursion (2019). https://github.com/PhilippeSigaud/Pegged/wiki/Left-Recursion
12. Tratt, L.: Direct left-recursive parsing expression grammars. Tech. Rep. EIS-10-01, School of Engineering and Information Sciences, Middlesex University (2010)
13. Warth, A., Douglass, J.R., Millstein, T.D.: Packrat parsers can support left recursion. In: Proceedings of the 2008 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation, PEPM 2008, San Francisco, California, USA, January 7-8, pp. 103–110 (2008)