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".

Keywords

parsing, recursive descent, left recursion

1. Introduction

Recursive-descent parser is a collection of "parsing procedures" that correspond to different syntactic units and call each other recursively. Some of these procedures must choose what to call next, and the choice is made by looking at the input ahead, on depth-first basis. This process does not work if the grammar is left-recursive: the procedure may indefinitely call itself, facing the same input. One suggested solution [1] counts the number of recursive calls of each procedure and stops when it reaches a pre-set bound. The process is repeated with the bound starting with 1 and gradually increased. Another solution [2, 3] saves the input consumed by invocations of parsing procedures and tricks the parser to use the saved result instead of making recursive call. The third idea sees parsing as reconstruction of input's syntax tree. The classical process builds that tree top-down, but [4] suggests that portions of the tree involved in left recursion can be reconstructed starting from the bottom. We present here an approach based on this idea. To reconstruct portions of the tree bottom-up we use procedures that call each other recursively. These procedures can be regarded as new parsing procedures. We incorporate them into the recursive-descent parser, and the result is recursive-descent parser for a new grammar, referred to as the "dual grammar". The dual grammar is (normally) not left-recursive and defines the same language as the original one.

After the necessary definitions in Section 2, Section 3 gives an example of recursive ascent, and shows that the procedures performing it can be seen as parsing procedures for new non-terminals. Section 4 incorporates these new non-terminals into the original grammar, thus obtaining the dual grammar. Two Propositions state the essential properties of that grammar. Section 5 discusses handling choice expressions, and Section 6 discusses the elements omitted to simplify the presentation. Section 7 contains some final remarks: relation to other solutions and unsolved problems. Proofs of the Propositions are given in the Appendix.

29th international Workshop on Concurrency, Specification and Programming (CS&P'21), September 2021, Berlin, Germany

roman@redz.se (R. R. Redziejowski)
http://www.romanredz.se (R. R. Redziejowski)

2. Basic concepts

We consider a BNF-like grammar $G = (\mathbb{N}, \Sigma, \mathbb{E}, \mathcal{E}, N_s)$ with finite set $\mathbb{N}$ of *non-terminals*, finite set Σ of *terminals*, finite set $\mathbb{E}$ of *expressions*, function $\mathcal{E}$ from non-terminals to expressions, and the *start symbol* N_s.
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 $N \to e$, where e is the expression assigned by $\mathcal{E}$ to non-terminal N. We often write $N \to e$ to mean $e = \mathcal{E}(N)$. To simplify the presentation, we did not include the empty string ε among expressions. In the following, expressions $a \in \Sigma$ and $N \in \mathbb{N}$ will be viewed as special cases of choice expression with $n = 1$.

Non-terminal $N \to e_1 \ldots e_n$ *derives* the string $e_1 \ldots e_n$ of symbols, while $N \to e_1|\ldots|e_n$ 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.

Figures 1 and 2 are examples of grammar G, showing syntax trees of strings derived from the start symbol. The set of all strings derived from $N \in \mathbb{N}$ is called the *language* of N and is denoted by $\mathcal{L}(N)$.

$$\begin{aligned} \mathbb{N} &= \{\mathtt{Z,A,A1,B,B1,B2}\} \\ \Sigma &= \{\mathtt{a,b,x,y}\} \\ Ns &= \mathtt{Z} \end{aligned}$$

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

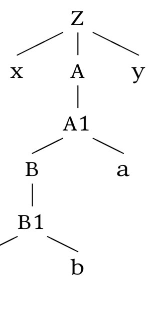

Figure 1: Example of grammar G and syntax tree of 'xabay'

For $N \in \mathbb{N}$ and $e \in \mathbb{N} \cup \Sigma$, define $N \xrightarrow{\text{first}} e$ to mean that parsing procedure for N 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 $1 \leq i \leq n$.

Let $\xrightarrow{\text{First}}$ be the transitive closure of $\xrightarrow{\text{first}}$. Non-terminal $N \in \mathbb{N}$ is *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 Figure 1 except Z, and all non-terminals in Figure 2 are recursive.

$$\begin{aligned} \mathbb{N} &= \{\texttt{E},\texttt{E1},\texttt{F},\texttt{F1}\} \\ \Sigma &= \{\texttt{a},\texttt{b},\texttt{x},\texttt{z}\} \\ Ns &= \texttt{E} \end{aligned}$$

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

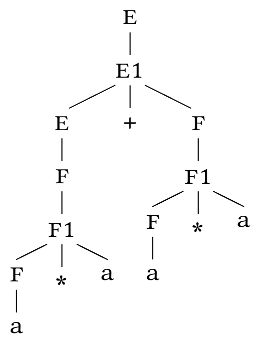

Figure 2: Example of grammar G and syntax tree of 'a*a+a*a'

Define relation between recursive $N_1, N_2 \in \mathbb{R}$ 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 Figure 1 belong to the same class; the grammar of Figure 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 the fact that $N_1 \xrightarrow{\text{First}} N \xrightarrow{\text{First}} N_2 \xrightarrow{\text{First}} N_1$. 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 Figure 1 is the whole leftmost path from the first A without final a. The syntax tree in Figure 2 has two recursion paths, one starting with E and another with F.

Let $N \to e_1 \dots e_n$ 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 $N \to e_1|\dots|e_n$ 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 Figure 1, both A and B are exits, and the seeds are a and b. In Figure 2, E and F are exits of their respective classes, and the corresponding seeds are F and a.

A recursive N that appears in expression for a non-terminal outside $\mathbb{C}(N)$, or is the start symbol, is an *entry* of its recursion class. It is the only non-terminal that can be first in a recursion path. The recursion class of Figure 1 has entry A, and recursion classes of Figure 2 have E and F as their respective entries.
To simplify presentation, we assume that each class has only one entry.

3. Recursive ascent

Recursive descent constructs the syntax tree implicitly, as the structure of its procedure calls. We assume that to serve any purpose, this tree has to be somehow registered. Thus, we assume that parsing procedures include "semantic actions" that actually build data structure representing the tree.

We suggest that parsing procedure for entry E builds its syntax tree in a special way, illustrated below by parsing the string `'xabay'` from Figure 1. The parsing starts with procedure for non-recursive Z that, in the usual way, calls the procedures for `'x'`, A, and `'y'`. After consuming `'x'`, Z applies A to `'abay'`. The procedure for entry A is not the usual choice between A1 and a; instead, it reconstructs the subtree A as follows:

1. The recursion path from A must end with a seed. The seeds are a and b. Decide to try a.
2. Apply expression a to `'abay'`. It consumes `'a'`, leaving `'bay'`. Use a as start of the tree.
3. The only possible parent of seed a in the syntax tree is its exit A. Add A on top of the tree, with a as child.
4. We have a tree for A, but decide to continue.
5. A appears only in B1$\rightarrow$A b, so B1 is the only possible parent of A.
6. The tree for A is already constructed. Complete B1 by applying expression b to the remaining `'bay'`. It consumes b, leaving `'ay'`. Add B1 on top of the tree, with A, b as children.
7. B1 appears only in B$\rightarrow$B1 | B2 | b, so B is the only possible parent of B1.
8. Add B on top of the tree, with B1 as child.
9. The possible parents of B are A1$\rightarrow$B a and B2$\rightarrow$B b. Decide for A1.
10. The tree for B is already constructed. Complete A1 by applying expression a to `'ay'`. It consumes `'a'`, leaving `'y'`. Add A1 on top of the tree with B, a as children.
11. The only possible parent of A1 is A$\rightarrow$A1.
12. Add A on top of the tree, with A1 as child.
13. We have a tree for A, and decide to stop. Return the tree of the consumed `'aba'` to Z, which continues to consume y and constructs the tree for `'xabay'`.

In general, the procedure for entry E chooses and executes one of procedures that correspond to different seeds. In the example, the choice is made in step 1, and the selected procedure consists of steps 2-13. The procedure for seed S starts with constructing the tree for S and follows by executing a procedure that adds node for the containing exit X; then it proceeds to grow the resulting tree towards E. These are the steps 2 respectively 3-13 in our example. This may be seen as parsing procedure that implements a new expression for E:

$$E \rightarrow S_1\,\$X_1|\ldots|S_n\,\$X_n \tag{1}$$

where S_i are all the seeds of $\mathbb{C}(E)$ and X_i are the exits containing them.

We can see $\$X$ as a special case of procedure $\$R$ that adds a new node R on top of previously constructed tree. This is the case in steps 6, 8, 10, and 12 of our example. The procedure then continues to build the tree, which is in each case done in the subsequent steps up to the step 13.

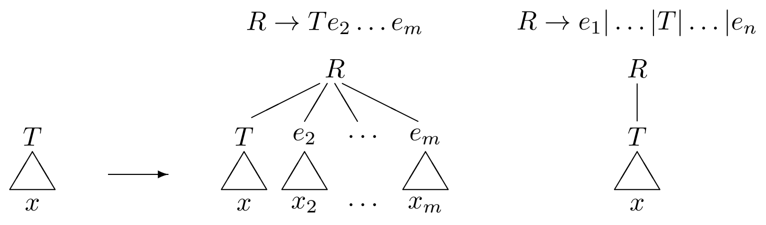

Figure 3: `Adding R`

The way of adding R on top of tree T depends on R as illustrated in Figure 3.

If $R \to e_1 e_2 \dots e_m$ (where $e_1 = T$), $\$R$ builds the trees for $e_2, \dots, e_m$, binds them with T into the tree for R. The whole procedure can be seen as parsing procedure for new non-terminal:

$$\$R \to e_2 \dots e_m \; \#R \tag{2}$$

where $e_2, \dots, e_m$ are procedures for these expressions and $\#R$ continues the growing.

If $R \to e_1|\dots|e_m$ (where $T = e_i$ for some i), the procedure just adds R on top of T and proceeds to grow that tree. This can be seen as parsing procedure for:

$$\$R \to \#R \; . \tag{3}$$

Procedure $\#R$ consists of choosing a possible parent of node R and adding that parent to the tree. This can be seen as parsing procedure for another new non-terminal:

$$\#R \to \$P_1|\dots|\$P_n \tag{4}$$

where P_i are all members of $\mathbb{C}$ such that $P_i \xrightarrow{\text{first}} R$.
In our example, the choice (4) is performed in steps 5, 7, 9, and 11.

If R is identical to E, $\#R$ can choose to terminate building of the tree. Therefore, $\#E$ must have an alternative to allow that:

$$\#E \to \$P_1|\dots|\$P_n|\$\varepsilon \tag{5}$$

where procedure $\$\varepsilon$ returns the tree constructed for E. It was not chosen in step 4, but terminated the process in step 13.

4. The dual grammar

As shown above, parsing procedure for an entry expression can be implemented by a set of procedures that reconstruct portion of syntax tree bottom-up, by "recursive ascent". These procedures can be seen as parsing procedures for new non-terminals. We can add these non-terminals to the original grammar. The result for the grammar of Figure 1 is shown in Figure 4. Here we replaced the expression for entry A by (1), and added the new non-terminals appearing in it, and in their expressions. The other left-recursive non-terminals are no longer used, so they are omitted. The non-recursive non-terminals, here represented by Z, are left unchanged. This

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

Figure 4: Dual grammar for grammar of Figure 1

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

Figure 5: Dual grammar for grammar of Figure 2

is the "dual grammar" of the grammar in Figure 1. Figure 5 shows the dual grammar obtained in a similar way for the grammar of Figure 2.

In general, the dual grammar is obtained as follows:

- For each entry E replace $\mathcal{E}(E)$ by $S_1\,\$X_1|\ldots|S_n\,\$X_n$
  where $S_1,\ldots,S_n$ are all seeds of $\mathbb{C}(E)$, and X_i is the exit containing S_i.
- Replace each recursive $R \to e_1e_2\ldots e_n$ by $\$R \to e_2\ldots e_n\ \#R$.
- Replace each recursive $R \to e_1|\ldots|e_n$ by $\$R \to \#R$.
- For each $R \in \mathbb{R}$ create:
  - $\#R \to \$P_1|\ldots|\$P_n$ if $R \neq E$,
  - $\#R \to \$P_1|\ldots|\$P_n|\$\varepsilon$ if $R = E$,

  where $P_1,\ldots,P_n$ are all members of $\mathbb{C}(R)$ such that $P_i \xrightarrow{\text{first}} R$, and E is the entry of $\mathbb{C}$.

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 $-expressions;
- The #-expressions.

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 $\mathbb{R}_\mathbb{E}$. They appear as non-terminals in both G and D. The set $\overline{\mathbb{R}} \cup \mathbb{R}_\mathbb{E}$ of these common symbols is denoted by $\mathbb{N}_c$. Functions $\mathcal{E}$ and $\mathcal{E}_d$ assign the same expressions to members of $\overline{\mathbb{R}}$.

The two important facts about the dual grammar are:

Proposition 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 the Appendix.

Proposition 2. $\mathcal{L}_D(N) = \mathcal{L}(N)$ *for all* $N \in \mathbb{N}_c$.

Proof is found in the Appendix.

The start symbol N_s is either non-recursive or an entry, so it appears in both grammars, and thus both grammars define the same language. It means that recursive-descent parser for the dual grammar is a correct parser for the original grammar, and its "semantic actions" build syntax tree for the original grammar.

5. Implementing the choices

The construction of dual grammar introduces a number of choice expressions. We assume that they are treated in the same way as those originally present in the grammar. If dual grammar has the LL(k) property, the choice can be made by looking at the next k input terminals. The dual grammar of Figure 4 is LL(1), so decisions in steps 1, 5, 10, and 14 of the example could well be made by looking at the next terminal.

If the dual grammar is not LL(k), the possible option is trial-and-error with backtracking. As this may result in exponential processing time, the practical solution is limited backtracking, recently being popular as the core of Parsing Expression Grammar (PEG) [5]. (As a matter of fact, the three solutions named in the Introduction are all suggested as modifications to PEG.)

The problem with limited backtracking is that it may fail to accept some legitimate input strings. For some grammars, which may be called "PEG-complete", limited backtracking *does* accept all legitimate strings. Thus, if the dual grammar is PEG-complete, it provides a correct parser for the original grammar.

There exist a sufficient condition that may be used to check if the dual grammar is PEG-complete [6].

The checks for LL(k) and PEG-completeness are carried on dual grammar. It is the subject of further research how they can be replaced by checks applied to the original grammar.

6. Towards full grammar

We made here two simplifying assumptions. First, we did not include empty string ε in the grammar. Second, we assumed a unique entry to each recursion class.

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 $N \to e_1 \ldots e_n$ invalidates the whole analysis in Section 2. Nullable $e_2 \ldots e_n$ in recursive $N \to e_1 e_2 \ldots e_n$ invalidates the proof of Proposition 1. Except for these two cases, our approach seems to work with empty string.

The assumption of unique entry per recursion class is not true for many grammars; for example, the class of Primary in Java has four entries.

The exit alternative $\$\varepsilon$ must appear in $\#E$ for the entry E that actually started the ascent. This can be solved by having a separate version of $\#E$ and $\$E$ for each entry, multiplying the number of these non-terminals by the number of entries.

A practical shortcut can exploit the fact that the ascents are nested. One can keep the stack of entries for active ascents and modify the procedure for $\#T$ to check if T is the entry to current ascent.

7. Final remarks

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 [7], page 177. The process is cumbersome and produces large results; most important, it loses the spirit of the grammar. Our rewriting is straightforward and produces correct syntax tree for the original grammar.

The methods described in [1, 2, 3] use memoization tables and special code to handle them. The procedures are repeatedly applied to the same input. Our approach is in effect just another recursive-descent parser.
Our idea of recursive ascent is in principle the same as that of [4], but this latter uses specially coded "grower" to interpret data structures derived from the grammar.

As indicated in Section 6, our method does not handle some cases of nullable expressions. Among them is the known case of "hidden" left recursion: $A \to e_1 e_2 \ldots e_n$ becomes left recursive when e_1 derives ε. Another unsolved problem is $E \to E{+}E \mid n$, which results in a right-recursive parse for E.

References

[1] S. Medeiros, F. Mascarenhas, R. Ierusalimschy, Left recursion in Parsing Expression Grammars, Science of Computer Programming 96 (2014) 177–190.

[2] A. Warth, J. R. Douglass, T. D. Millstein, 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, 2008, pp. 103–110.

[3] L. Tratt, Direct left-recursive parsing expression grammars, Technical Report EIS-10-01, School of Engineering and Information Sciences, Middlesex University, 2010.

[4] O. Hill, Support for Left-Recursive PEGs, 2010. `https://github.com/orlandohill/peg-left-recursion`.

[5] B. Ford, Parsing expression grammars: A recognition-based syntactic foundation, in: N. D. Jones, X. Leroy (Eds.), Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2004, ACM, Venice, Italy, 2004, pp. 111–122.

[6] R. R. Redziejowski, More about converting BNF to PEG, Fundamenta Informaticae 133 (2014) 177–191.

[7] A. V. Aho, R. Sethi, J. D. Ullman, Compilers, Principles, Techniques, and Tools, Addison-Wesley, 1987.

A. Proof of Proposition 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 $N \in \mathbb{R}_{\mathbb{E}}$, $N \xrightarrow{\text{firstD}} S$ for each seed S of $\mathbb{C}(N)$.
(c) For $\$R \to e_2 \dots e_n\#R$, $\$R \xrightarrow{\text{firstD}} e_2$.
(d) For $\$R \to \#R$, $\$R \xrightarrow{\text{firstD}} \#R$.
(e) $\#R \xrightarrow{\text{firstD}} \$P_i$ for each $P_i \in \mathbb{C}(R)$ such that $P_i \xrightarrow{\text{first}} R$.

Grammar D is left-recursive if $N \xrightarrow{\text{FirstD}} N$ for some $N \in \mathbb{N}_d$, where $\xrightarrow{\text{FirstD}}$ is the transitive closure of $\xrightarrow{\text{firstD}}$.

Suppose D is left-recursive, that is, exist $N_1, N_2, \dots, N_k \in \mathbb{N}_d$ such that $N_1 \xrightarrow{\text{FirstD}} N_2 \xrightarrow{\text{FirstD}} \dots \xrightarrow{\text{FirstD}} N_k$ and $N_k = N_1$. We start by showing that none of them can be in $\mathbb{N}_c$.

Assume, by contradiction, that $N_1 \in \mathbb{N}_c$. We show that in this case $N_2 \in \mathbb{N}_c$ and $N_1 \xrightarrow{\text{First}} N_2$.

(Case 1) $N_1 \in \overline{\mathbb{R}}$. Because $\mathcal{E}_d(N_1) = \mathcal{E}(N_1)$, N_2 is in $\mathbb{N}$, and thus in $\mathbb{N}_c$. We have $N_1 \xrightarrow{\text{first}} N_2$.

(Case 2) $N_1 \in \mathbb{R}_{\mathbb{E}}$. 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 $N_1 \xrightarrow{\text{First}} N_2$.

(Conclusion) The above can be repeated with $N_2, \dots, N_{k-1}$ to show that if any of N_i is in $\mathbb{N}_c$, then for all i, $1 \le i \le n$ $N_i \in \mathbb{N}_c$ and $N_i \xrightarrow{\text{First}} N_i$. Thus, all N_i belong to $\mathbb{R}$ and are all in the same recursion class. As none of N_i belongs to $\overline{\mathbb{R}}$, they must all be in $\mathbb{R}_{\mathbb{E}}$. Then, in particular, $N_1 \in \mathbb{R}_{\mathbb{E}}$. But, according to (b), N_2 is a seed of $\mathbb{C}(N_1)$, that cannot be a member of $\mathbb{C}(N_1)$. Thus, $N_1 \notin \mathbb{N}_c$.

It follows that each N_i is $\$R_i$ or $\#R_i$ for some $R_i \in \mathbb{R}$. If $R_i \to e_2 \dots e_n$, we have from (c) $\$R_i \xrightarrow{\text{firstD}} e_2$, and $e_2 \in \mathbb{N}_c$. That means all R_i are choice expressions. Thus, the sequence of N_i is a repetition of $\$R_i, \#R_{i+1}, \$R_{i+2}$ where, according to (d), $R_{i+1} = R_i$, and according to (e), $R_{i+2} \to \dots |R_{i+1}| \dots$. That means R_{i+2} derives R_i, and in $k/2$ steps derives itself. $\square$

B. Proof of Proposition 2

The proof is in terms of syntax trees. We write $e \lhd x$ for syntax tree of x with root e. We write $e \lhd [e_1 \lhd x_1 + \dots + e_n \lhd x_n]x$ to represent syntax tree with root e and children $e_1 \lhd x_1, \dots, e_n \lhd x_n$ where $x_1 \dots x_n = x$.

To distinguish between the trees according to grammar G (G-trees) and those according to grammar D (D-trees) we use the symbols $\lhd_G$ respectively $\lhd_D$.

Assuming that a terminal derives itself, we show that every string x derived from $e \in \mathbb{N}_c \cup \Sigma$ according to G can be also derived from e according to D and vice-versa. The proof is by induction on height of syntax trees.

(Induction base) Syntax tree of height 0. This is the syntax tree of a terminal. The terminals and their syntax trees are identical in both grammars.

(Induction step) Assume that the stated property holds for all strings w having syntax tree of height $h \geq 0$ or less. Lemmas 1 and 2 below show that it holds then for syntax trees of height $h+1$.

Lemma 1. *Assume that for each $e \lhd_G w$ of height $h \geq 0$ with $e \in \mathbb{N}_c \cup \Sigma$ exists $e \lhd_D w$. Then the same holds for each $e \lhd_G w$ of height $h+1$.*

Proof. Take any $N \lhd_G w$ of height $h+1$ where $N \in \mathbb{N}_c$.

(Case 1) $N \in \overline{\mathbb{R}}$. It means $N \rightarrow e_1 \dots e_n$ or $N \rightarrow e_1 | \dots | e_n$.
We have $N \lhd_G [e_1 \lhd_G w_1 + \dots + e_n \lhd_G w_n] w$ respectively $N \lhd_G [e_j \lhd_G w] w$. Each of the subtrees $e_i \lhd_G w_i$ has height h or less and each e_i is in $\mathbb{N}_c \cup \Sigma$. By assumption, exists $e_i \lhd_D w_i$ for each i. The required D-tree is $N \lhd_D [e_1 \lhd_D w_1 + \dots + e_n \lhd_D w_n] w$ respectively $N \lhd_D [e_j \lhd_D w] w$.

(Case 2) $N \in \mathbb{R}_{\mathbb{E}}$. The tree $N \lhd_G w$ has the leftmost path $R_n \xrightarrow{\text{first}} \dots \xrightarrow{\text{first}} R_1 \xrightarrow{\text{first}} R_0$ where $R_n = N$, $R_i \in \mathbb{C}(N)$ for $n \geq i > 0$, and R_0 is a seed of $\mathbb{C}(N)$. Define x_i to be the string derived by G from R_i, for $0 \leq i \leq n$. We have thus $w = x_n$.

Define w_0 be the string derived by G from R_0, so the subtree for R_0 is $R_0 \lhd_G w_0$.
Let $1 \leq i \leq n$. If $R_i \rightarrow e_1 e_2 \dots e_m$, the subtree for R_i is $R_i \lhd_G [R_{i-1} \lhd_G x_{i-1} + e_2 \lhd_G v_2 + \dots + e_m \lhd_G v_m] x_i$. Define $w_i = v_2 \dots v_m$, so $x_i = x_{i-1} w_i$.
if $R_i \rightarrow e_1 | \dots | e_m$, the subtree for R_i is $R_i \lhd_G [R_{i-1} \lhd_G x_{i-1}] x_i$ Define $w_i = \varepsilon$, so we have again $x_i = x_{i-1} w_i$.
One can easily see that $w = x_n = w_0 \dots w_n$.

Define y_i to be the string derived by D from $\$R_i$, and z_i to be the string derived by D from $\#R_i$, for $0 \leq i \leq n$.

For entry R_n exists the tree $\#R_n \lhd_D \varepsilon$. If $\$R_n \rightarrow \#R_n$, exists the tree $\$R_n \lhd_D [\#R_n \lhd_D \varepsilon] \varepsilon = w_n$. If $\$R_n \rightarrow e_2 \dots e_n \#R_n$, exists by assumption D-tree for each of e_j, deriving, respectively, v_j. Thus, exists the tree $\$R_n \lhd_D [e_2 \lhd_D v_2 + dots + e_m \lhd_D v_m + \#R_n \lhd_D \varepsilon] v_2 \dots v_m \varepsilon = w_n$. Define $y_n = w_n$.

Suppose that exists the tree $\$R_i \lhd_D y_i$. Then exists the tree $\#R_{i-1} [\lhd_D \$R_i \lhd_D y_i] y_i$. By construction similar to the above, w find the D-tree for $\$R_{i-1}$ deriving $y_{i-1} = w_{i-1} y_i$. At the end, we find the D-tree for $\$R_1$ deriving $y_1 = w_1 y_2$.
By assumption there exists D-tree for the seed R_0 deriving w_0. For entry R_n we have $R_n \rightarrow R_0 \$R_1$, which gives the D-tree $R_n \lhd_D [R_0 \lhd_D w_0 + \$R_i \lhd_D y_1] w_0 y_1$. One can easily see that $y_1 = w_1 \dots w_n$, so we have a D-tree for $N = R_n$ deriving $w = w_0 \dots w_n$. □

Lemma 2. *Assume that for each $e \lhd_D w$ of height $h \geq 0$ with $e \in \mathbb{N}_c \cup \Sigma$ exists $e \lhd_G w$. Then the same holds for each $e \lhd_D w$ of height $h+1$.*

Proof. Take any $N \lhd_D w$ of height $h+1$ where $N \in \mathbb{N}_c$.

(Case 1) $N \in \overline{\mathbb{R}}$. It means $N \rightarrow e_1 \dots e_n$ or $N \rightarrow e_1 | \dots | e_n$. We have $N \lhd_D [e_1 \lhd_D w_1 + \dots + e_n \lhd_D w_n] w$ respectively $N \lhd_D [e_j \lhd_D w] w$. Each of the subtrees $e_i \lhd_D w_i$ has height h or less and each e_i is in $\mathbb{N}_c \cup \Sigma$. By assumption, exists $e_i \lhd_G w_i$ for each i. The required G-tree is $N \lhd_G [e_1 \lhd_G w_1 + \cdots + e_n \lhd_G w_n] w$ respectively $N \lhd_G [e_j \lhd_G w] w$.

(Case 2) $N \in \mathbb{R}_{\mathbb{E}}$. The D-tree of N has root N, node S for seed of $\mathbb{C}(N)$,and nodes $\$R_i$, $\#R_i$ for $1 \leq i \leq n$. Denote $R_n = N$ and $R_0 = S$. Denote y_i the string derived by $\$R_i$ and z_i that derived by $\#R_i$. We have $z_i = y_{i+1}$ for $i < n$ and $z_n = \varepsilon$. Denote by w_0 the string derived by R_0 and by w one derived by $N = R_n$.
The D-tree for R_n is $R_n \lhd_D [R_0 \lhd_D w_0 + \$R_1 \lhd_D y_1] w_0 y_1 = w$.
If $\$R_i \to e_2 \dots e_n \#R_i$, we have $y_i = v_2 \dots v_m z_i$. where v_j is the string derived by e_j. Defining $w_i = v_2 \dots v_m$ by w_i, we have $y_i = w_i z_i$.
If $\$R_i \#R_i$, we have $y_i = z_i$. Defining $w_i = \varepsilon$, we have again $y_i = w_i z_i$.
We have $z_i = y_(i+1)$ for $i < n$ and $z_n = \varepsilon$, so $y_i = w_i y_(i+1)$ for $i < n$ and $y_n = w_n$. One can easily see that $w = w_0 \dots w_n$.

By assumption exists G-tree $R_0 \lhd_G w_0$. Suppose we have the G-tree for R_{i-1} where $1 \leq i \leq n$ that derives x_{i-1}.
Suppose $R_i \to R_(i-1) e_2 \dots e_m$. By assumption exist G-trees that derive $v_2 \dots v_m$ from $e_2 \dots e_m$, so exists G-tree for R_i deriving $x_i = x_(i-1) v_2 \dots v_m = x_(i-1) w_i$. Suppose $R_i \to R_(i-1)$. Then exists G-tree deriving $x_i = x_(i-1)\varepsilon = x_(i-1) w_i$. One can easily see that the string x_n derived by R_n is $w_0 \dots w_n = w$ $\square$