Left recursion in *Mouse*

Roman R. Redziejowski

1 Recursive Ascent

1.1 Basic concepts

Expressions, rules, syntax trees

The part of *Mouse* grammar involved in left-recursion can be seen as a BNF grammar over alphabet Σ of *terminals*. It is a set of named *expressions* defined by rules of the form $A = e$ where e is an expression and A is its name. An expression is one of these:

- $a \in \Sigma$ ("terminal"),
- $e_1 \dots e_n$ with $n > 1$ ("sequence"),
- $e_1/ \dots /e_n$ with $n \geq 1$ ("choice"),

where each of e_i is either an expression or expression name. The rule states that A is synonymous with the expression e.

(To simplify the picture, we omit for the moment the empty-word expression.)

The grammar is a mechanism for generating strings of terminals. Such string is generated from an expression e by repeated application of these operations:

- Replace sequence expression $e = e_1 \dots e_n$ by sequence of expressions $e_1, \dots, e_n$.
- Replace choice expression $e = e_1/ \dots /e_n$ by one of the expressions $e_1, \dots, e_n$.

This is carried out until no more replacements are possible. The process is represented by a *syntax tree*. It is an ordered tree where the nodes are expressions and the leaves are terminals. The branch(es) emanating from node e represent replacement of e by one or more expressions. The set of terminal strings that can be generated from expression e is called the *language* of e.

Shown in Figure 1 are two grammars and examples of strings generated by them, together with their syntax trees. (The capitals are expression names, small letters are terminals, and `Ab`, `EF`, etc. are temporary names for inner expressions.)

Recursive expressions and recursion classes

Let $e \xrightarrow{\text{first}} e'$ mean that node e in the syntax tree may have node e' as its leftmost (or only) branch:

- If $e = e_1 \dots e_n$, we have $e \xrightarrow{\text{first}} e_1$.
- If $e = e_1/ \dots /e_n$, we have $e \xrightarrow{\text{first}} e_i$ for $1 \leq i \leq n$.

Let $e \xrightarrow{\text{First}} e'$ means that node e' in the syntax tree may be reached from node e via a *leftmost path* - one consisting of leftmost branches. In other words, that $e \xrightarrow{\text{first}} e_1 \xrightarrow{\text{first}} \dots \xrightarrow{\text{first}} e_n \xrightarrow{\text{first}} e'$ for some $e_1, e_2, \dots, e_n, n \geq 0$.

Define expression e to be *left-recursive*, (shortly: *recursive*) to mean that $e \xrightarrow{\text{First}} e$. We denote the set of all recursive expressions by $\mathbb{R}$. We say that recursive expressions e_1, e_2 belong to the same *recursion*

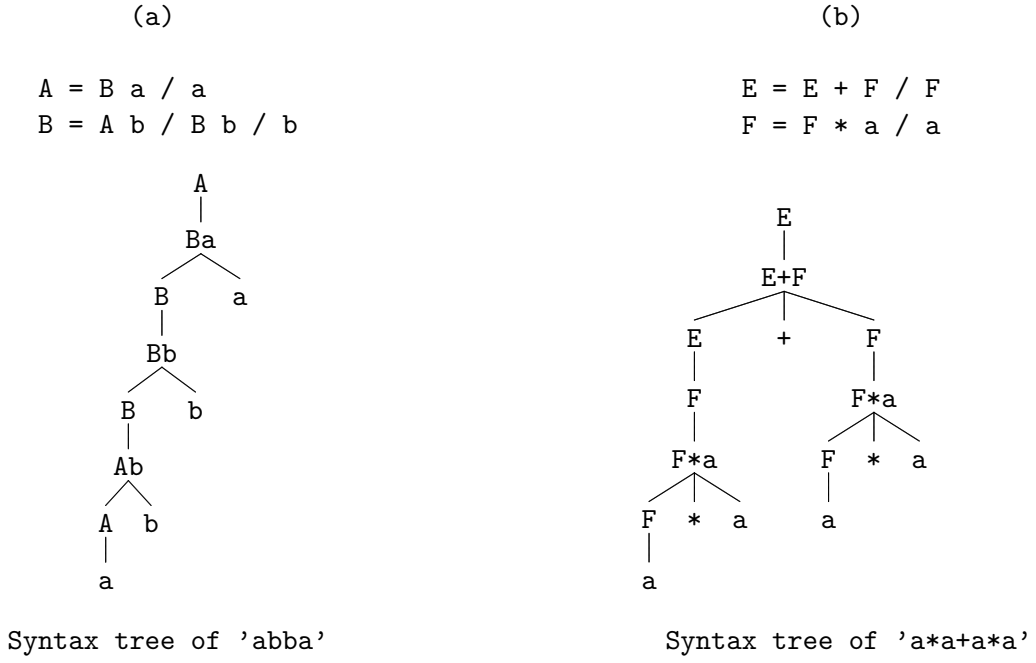

Figure 1: Examples of left-recursive grammars

class if $e_1 \xrightarrow{\text{First}} e_2 \xrightarrow{\text{First}} e_1$. The recursion classes are disjoint and form a partition of $\mathbb{R}$. The recursion class of expression e is denoted by $\mathbb{C}(e)$.

The grammar of example (a) in Figure 1 has {`A`,`Ab`,`B`,`Ba`,`Bb`} as the set of recursive expressions. They all belong to the same class. Example (b) has two recursion classes: {`E`,`E+F`} and {`F`,`F*a`}.

Recursion path: entry, exit, seed

In syntax tree, the leftmost path emanating from any expression is a chain of expressions connected by $\xrightarrow{\text{first}}$. Suppose expressions e_1, e_2 belonging to the same recursion class $\mathbb{C}$ appear on the same leftmost path. Any expression e between them must also belong to $\mathbb{C}$, because we have $e_1 \xrightarrow{\text{First}} e \xrightarrow{\text{First}} e_2 \xrightarrow{\text{First}} e_1$. It means that all expressions from $\mathbb{C}$ appearing on the same leftmost path must form an uninterrupted sequence. We call such sequence a *recursion path* of class $\mathbb{C}$.

In example (a), there is one such path: the whole leftmost path without final `a`. In example (b), we have recursion path (`E`,`E+F`,`E`) of the class containing `E` and two paths (`F`,`E*F`,`F`) of the class containing `F`.

Let $e = e_1 \dots e_n$ be on recursion path. The next expression on the path must be e_1, and it must belong to $\mathbb{C}(e)$, because $e \xrightarrow{\text{First}} e$ requires $e \xrightarrow{\text{first}} e_1 \xrightarrow{\text{First}} e$. It follows that the last expression on recursion path can only be $e = e_1/\dots/e_n$ where at least one of e_i is not be a member of $\mathbb{C}(e)$.

The choice expression in $\mathbb{C}$ with alternatives outside $\mathbb{C}$ is called an *exit* of $\mathbb{C}$, and these alternatives are the *seeds* of $\mathbb{C}$. In example (a), both `A` and `B` are exits, and the seeds are `a` and `b`. In example (b), `E` is the exit of class {`E`,`E+F`} with seed `F`, and `F` is the exit of class {`F`,`F*a`} with seed `a`.

An expression that may appear as first in recursion path of class $\mathbb{C}$ is an *entry* of that class. The class in example (a) has entry `A`; the classes in example (b) have `E` respectively `F` as their entries.

To simplify the discussion, we assume for the moment that each recursion class has only one entry expression.

1.2 Parsing

Parsing is the process of reconstructing the syntax tree for a given string. *Mouse* uses for this purpose the method known as "recursive descent". It reconstructs the syntax tree top-down.

Each expression has its procedure to construct the subtree having that expression as its root. If it succeeds, it "consumes" the corresponding input. The procedure for a terminal just checks for that terminal. The procedure for $e_1 \dots e_n$ calls in turn the procedures for $e_1, \dots, e_n$, that should appear as the subtrees. The procedure for $e_1/ \dots /e_n$ must choose one of the procedures for $e_1, \dots, e_n$. *Mouse* does this by trial-and-error: calls the procedures one after another and backtracks until one of them succeeds.

This process does not work for left-recursive expressions. An attempt to reconstruct recursion path would mean a procedure for such ecpression calling itself indefinitely on the same input. The solution applied by *Mouse* is that procedure for $e \in \mathbb{R}$ encountered in a recursive descent does not continue the descent. Instead, it reconstructs the syntax tree bottom up, by a process that we call "recursive ascent". The recursion path startng with e in recursion class $\mathbb{C}$ must end with a seed of $\mathbb{C}$. The process starts by trying procedures for the seeds of $\mathbb{C}$. The procedure for e identifies the seed an then grows the syntax tree like a plant towards e.

Example of recursive ascent

The process is illustrated on the grammar (a) from Figure 1 for input `'abba'`. It is the action of procedure for `A` and its goal is to reconstruct syntax tree with root `A`.

- The seeds are `a` and `b`. Apply `a` to `'abba'`. It consumes `'a'`. Put `a` as start of the plant.
- The only possible predecessor of `a` in the syntax tree is `A`. As `a` is an alternative of `A`, there is no need to call the procedure `A`. It must consume `'a'`. Add `A` on top of the plant.
- The only possible predecessor of `A` is `Ab`. We already know that procedure `A` consumed `'a'`. Apply procedure `b` to the remaining `'bba'`. It consumes `b`, so `Ab` consumed `'ab'`. Add `Ab` on top of the plant.
- The only possible predecessor of `Ab` is `B`. As `Ab` is an alternative of `B`, there is no need to call the procedure `B` . It must consume `'ab'`. Add `B` on top of the plant.
- The possible predecessors of `B` are `Ba` and `Bb`. We already know that `B` consumed `'ab'`, leaving `'ba'`. Apply procedure `a` to `'ba'`. Fails. Backtrack.
- Apply procedure `b` to `'ba'`. It consumes `b`, so `Bb` consumes `'abb'`. Add `Bb` on top of the plant.
- The only possible predecessor of `Bb` is `B`. As `Bb` is an alternative of `B`, there is no need to call the procedure `B`. It must consume `'abb'`. Add `B` on top of the plant.
- The possible predecessors of `B` are `Ba` and `Bb`. We already know that `B` consumes `'abb'`, leaving `'a'`. Apply procedure `a` to `'a'`. It consumes `a`, so `Ba` consumes `'abba'`. Add `Ba` on top of the plant.
- The only possible predecessor of `Ba` is `A`. As `Ba` is an alternative of `A`, there is no need to call the procedure `A`. It must consume `'abba'`. Add `A` on top of the plant.
- The only possible predecessor of `A` is `Ab`. We already know that `A` consumed `'abba'`. Applying `b` to the remaining empty input fails. Backtrack.

We have thus reconstructed syntax tree with root `A`. We already had a tree with root `A` after the second step, but did not stop there. This is the "greedy" behavior of *Mouse*: it stops only when no continuation is posible.

Note that the described process may be nested. In example (b), identifying the seed for `E` means invoking construction of recursion path for `F`. The same applies to the step that tries `E+F`.

1.3 The dual grammar

The process illustrated above can be carried out as recursive descent process for another grammar.

In the repetitive step of the ascent we first complete processing of expression R that was initiated by the preceding step, and add R to the plant. Then we select a possible predecessor P of R and repeat the same for P. Apart from growing the plant, this is exactly the recursive descent process for expression

$$\$R = \texttt{rest}(R)\texttt{pre}(R). \tag{1}$$

Here $\texttt{rest}(R)$ is the sequence of procedures to be called after the first one of R succeeded:

- $\texttt{rest}(e_1 \dots e_n) = e_2 \dots e_n$;
- $\texttt{rest}(e_1 | \dots | e_n) = \varepsilon$.

The $\texttt{pre}(R)$ represents the choice of next procedure to call:

- $\texttt{pre}(R) = \$P_1 / \dots / \$P_n$, or
- $\texttt{pre}(R) = \$P_1 / \dots / \$P_n / \varepsilon$ if R is the entry of $\mathbb{C}(R)$,

where P_i are all members of $\mathbb{C}(R)$ such that $P_i \xrightarrow{\text{first}} R$.

(ε is the procdure for empty word - always succeeds without consuming anything.)

The procedure corresponding to (1) is referred to as an *ascent procedure*. The whole process is started by identifying the seed and exploring its predecessor. This is exactly the action of procedure for expression

$$E = S_1\, \texttt{pre}(S_1) / \dots / S_n\, \texttt{pre}(S_n) \tag{2}$$

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

Summarizing, the recursive ascent can be performed by recursive descent in a "dual grammar" obtained as follows:

- Assign names to all unnamed inner expressions.
- For $A \in \mathbb{R}$ replace $A = e$ by $\$A = \texttt{rest}(A)\texttt{pre}(A)$;
- For each entry E add $E = S_1\, \texttt{pre}(S_1) / \dots / S_n\, \texttt{pre}(S_n)$.

The dual grammars for our examples are shown in Figure 2.

(a)

```
A   = a $A / b $B
$A  = $Ab / ε
$Ab = b $B
$B  = $Ba / $Bb
$Ba = a $A
$Bb = b $B
```

(b)

```
E   = F $E
$E  = $EF / ε
$EF = + F $E
F   = a $F
$F  = $Fa / ε
$Fa = * a $F
```

Figure 2: Dual grammars for examples from Figure 1.

It can be verified that dual grammar is free from left-recursion as long as the original grammar does not have cycles. (A cycle is when procedure calls itself without consuming input.)

It can also be verified that the language of every non-recursive expression and every entry E is unchanged in the dual grammar.

Formal proofs are found in a separate paper.

2 Implementation in *Mouse*

The parser generated by *Mouse* is a PEG parser for the dual grammar outlined above. As one can easily see, procedures for expressions not involved in left recursion remain unchanged.

2.1 There is more in *Mouse*

Expressions other than sequence and choice

The full *Mouse* grammar contains expressions other than sequence and choice: syntactic predicates, ”query”, ”star”, ”plus”, ”star-plus”, and ”plus-plus”. They are not supported as left-recursive.

Nullable expressions

The predicates, query, and star expressions are ”nullable”: their language contains empty word. This is also the case for some expressions constructed from them. As they are not supported as left-recursive, they remain unchanged in the dual grammar.

There is, however, one restriction: the first element in a recursive sequence expression must not be nullable.

Multiple entries

The real-life grammars may have recursion classes with multiple entries. For example, the recursion class of Primary in Java has four entries. The problem is that the terminating alternative ε appears in $\texttt{pre}(R)$ only if the actual ascent was entered via R. This would require, in principle, a separate set of ascent procedures for each entry. *Mouse* solves this problem using the fact that ascents are nested. It maintains a stack of ascents and each $\$R$ procedure for entry R consults the top of stack to check if the current ascent was entered by R.

The expression (2) is the same for each entry to the same class C. To avoid repetition, it is implemented as subroutine named $$C that is used by each entry.

2.2 Semantics

This section describes the working of parser generated by *Mouse*. It is referred to as *Parser*. It operates, as described before, by recursive descent with procedures corresponding to expressions. The purpose of this process is to reconstruct the syntax tree of given input, usually referrd to as its ”abstract syntax tree” (AST). Most parsers deliver their result in the form of an AST encoded in some way. Normally, this AST is in the next step further processed to yield something useful, such as a compiler or result of a computation. It is here the referred to as the *semantic value* of the AST. The *Parser* skips this step to directly produce the semantic value. This result is defined by the *Parser*’s constructor by means of *semantic actions* attached to expressions.

The AST being reconstructed remains behind the scenes in a more or less virtual form. This approach hides irrelevant details of the AST. They are, in particular, abundant if the grammar defines lexical structures.

Phrase objects

Each parsing procedure represnts its result by a Java object of class Phrase. The object contains identification of the corresponding expresssion, the consumed input string and the semantic value - another Java object. In addition, it points to Phrases created by the sub-procedures.

This may be represented as follows:

```
[E] -> [e1] ... [en]
```

where [E] is the Phrase of procedure for expression E and [e1] ... [en] are Phrases of procedures called from it. This may be viewed as a part of the reconstructed AST, with node [E] and branches from it to [e1] ... [en].

Semantic actions

When semantic action is called, it has access to the structure shown above, and has at its disposal service methods for accessing [E] and each of [ei]. At that moment, the semantic values of [e1] ... [en] are already computed, and the action uses them to compute the value for [E]. Alternatively, the action may compute the value for [E] directly from the string consumed by E.

For all expressions E other than choice, the semantic action is associated with the expression. For a choice expression E, each alternative has its own semantic action, and *Parser* selects it depending on which alternative was recognized.

Semantics of recursive ascent

The way of creating Phrase objects is described in a separate document.

The problem with recursive ascent is that recursive expressions do not have corresponding procedures. Instead for procedure for a recursive expression R, the parser has the ascent procedure $R that constructs its Phrase [$R].

A typical set of Phrases for recursive ascent may appear like this:

```
  [E] ->
[C$S] -> [s1]...[sn]
 [$X] ->
 [$R] -> [r2]...[rm]
    ...
 [$Z] -> ...
 [$E] -> [e2]...[ek]
```

Here procedure for the entry expression E started recursive ascent by calling C$S that identified seed S with components [s1]...[sn]. Then C$S succeeded with $X that is part of pre(S). For the exit expression X, rest(X) is empty, and $X succeded with $R that is part of pre(X). Then $R succeeded with its rest(R) part that consisted of [r2]...[rm] and followed to an expression from pre(R). This continued until $E completed its rest(E) part and arrived at the empty alternative of pre(E).

This is hidden for the constructor of *Parser* that writes its semantic actions for S, X, R, .., Z, E in the same way as for non-recursive expressions. In order to apply these actions, the *Parser* constructs behind the scenes a number of simulated Phrases that in the above case will appear like this:

```
[S] -> [s1]...[sn]
[X] -> [S]
[R] -> [X][r2]...[rm]
   ...
[Z] -> ...
[E] -> [Z][e2]...[ek]
```

after which it invokes the semantic actions for S, X, R, .., Z, and E. (The semantic values of [si], [ri], etc. have been computed before.)

The picture above is, in fact, the reconstructed AST for the ascent.

Choice of semantic action

There is a problem with recursive choice expression `A = .../B/...`. Suppose a recursion path contains `A` that calls `B`. The *Parser*, ascending the path, will first call `$B`, creating the Phrase `[$B]`. Then, it will call `$A` as part of `pre(B)`, resulting in the Phrase `[$A]`:

```
[$B] -> ...
[$A] ->
```

The simulation described above gives

```
[B] -> ...
[A] -> [B]
```

and now one is supposed to call semantic action of `A`. It has to be the action for alternative `B`. And this follows from the fact that `$A` is called from `pre(B)`.

This is solved so that `$A` receives its action as a parameter, in the form of lambda-expression, and stores it in `[$A]` from where it is retrieved when needed.

For uniformity, this is applied to all recursive expressions, even if they have only one semantic action.