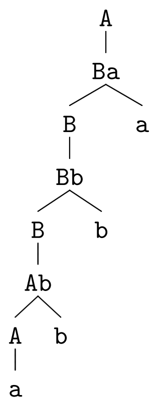
Left Recursion in *Mouse*

Roman R. Redziejowski

Basic concepts

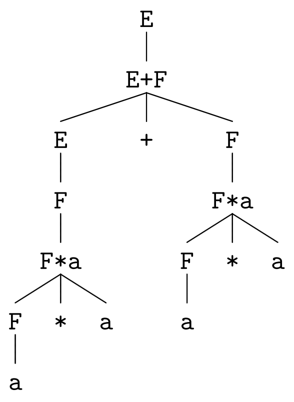
The following are examples of left-recursive grammars and strings parsed according to them.
To simplify the picture, we omitted double quotes and represented terminals by small letters.

```
(1)

A = B a / a ;
B = A b / B b / b ;
```



Syntax tree of 'abba'

```
(2)

E = E + F / F ;
F = F * a / a ;
```



Syntax tree of 'a*a+a*a'

Figure 1: Examples of left-recursive grammars

For expressions e_1 and e_2, we write $e_1 \xrightarrow{\text{first}} e_2$ to mean that e_1 may call e_2 on the same input. *Mouse* restricts support for left-recursion to Choice expressions $e_1/\ldots/e_n$ and Sequence expressions $e_1 \ldots e_n$ where e_1 is not nullable. We have thus:

- For $e = e_1/\ldots/e_n$, $e \xrightarrow{\text{first}} e_i$ for $i = 1, \ldots n$.
- For $e = e_1 \ldots e_n$, $e \xrightarrow{\text{first}} e_1$.

We write $e_1 \xrightarrow{\text{First}} e_n$ to mean $e_1 \xrightarrow{\text{first}} e_2 \xrightarrow{\text{first}} \ldots \xrightarrow{\text{first}} e_n$ and say that e is *left-recursive* (shortly: *recursive*) to mean that $e \xrightarrow{\text{First}} e$. All nonterminal expressions in the above examples are recursive.

One can divide recursive expressions into disjoint classes by defining that e_1 and e_2 belong to the same class if $e_1 \xrightarrow{\text{First}} e_2 \xrightarrow{\text{First}} e_1$. All recursive expressions in the first example belong to the same class; in the second we have two classes, namely {`E`, `E+F`} and {`F`, `F*a`}.

Choose any recursive expression in the syntax tree and follow the leftmost path emanating from it. The path follows the $\xrightarrow{\text{first}}$ links and all expressions in it are applied to the same input.

The leftmost path may contain sequences of expressions from the same class; we refer to them as *recursion paths*.

There is only one recursion path in example (1); it consists of the entire leftmost path without the final `a`. In example (2), we have `(E, E+E, E)` and two `(F, F*F, F)`.

A recursion path of class $\mathbb{C}$ must end with a Choice $e = e_1/\dots/e_n$ where at least one of e_i does not belong to $\mathbb{C}$. Such e is called an *exit* of $\mathbb{C}$, and each such e_i is a *seed* of $\mathbb{C}$. The exits in example (1) are both `A` and `B`, and the seeds are `a` and `b`. In example (2), class $\{$`E, E+F`$\}$ has exit `E` and seed `F` while class $\{$`F, F*a`$\}$ has exit `F` and seed `a`.

A recursion path of class $\mathbb{C}$ starts with an expression e that is either the start of the grammar or is called by an expression outside $\mathbb{C}$. Such e is an *entry* of $\mathbb{C}$. The entry in example (1) is `A`. The classes in example (2) have `E` and `F` as their respective entries.

Recursive Ascent

The procedure for a recursive expression E is not implemented by calling procedures for its sub-expressions. Instead, it starts the process of reconstructing the recursion path that begins with E. The process begins with a seed and builds the path upwards. It is referred here as the *recursive ascent*.

The process starts with identifying the seed. It means calling the expressions for seeds of the class of E until one of them succeeds on the input string. The successful expression becomes then a candidate for the end of recursion path.

The success of seed expression starts a recursive process. Suppose an expression e succeeded, consuming prefix x of the input, and was placed on top of the recursion path being grown. The next expression, e', above it must be such that $e' \xrightarrow{\text{first}} e$, which means trying them in turn.

If $e' = e_1/\dots/e_n$, e is one of e_i and it is not necessary to try e' as it would also consume exactly x. We put it on top of the candidate path being grown.

If $e' = e_1 \dots e_n$, e the same as e_1. As we know that $e_1 = e$ consumes x, we have to try $e_2 \dots e_n$ on the input following x. If they succeed in consuming y, e' consumes xy, and we put e' on top of the candidate path.
If they fail, we backtrack to try another alternative.

We illustrate this process on the grammar from example (1). Figure 2 is the scheme for our search. Expression at each node is followed by the next expressions to try.

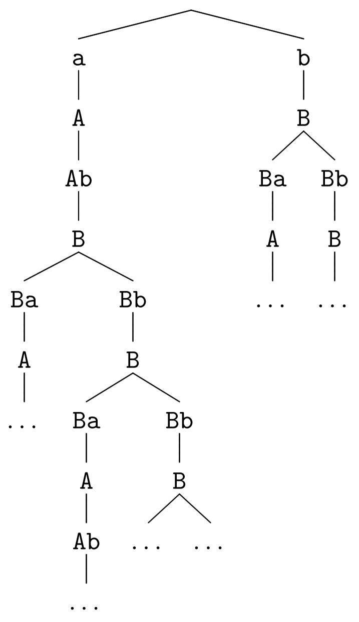


Figure 2: Search tree for example (1)

Assume the input is 'abba'. The process is as follows.

- Apply expression for seed a. Succeeds and consumes 'a'.
- No need to try A. We know that it succeeds and consumes 'a'. Add A on top of tentative path.
- Try Ab. As A already consumed 'a', apply b to 'bba'. Succeeds and consumes b, so Ab consumes 'ab'. Add Ab on top of tentative path.
- No need to try B. We know that it succeeds and consumes 'ab'. Add B on top of tentative path.
- Try Ba. As B already consumed 'ab', apply a to 'ba'. Fails. Backtrack to B.
- Try Bb. As B already consumed 'ab', apply b to 'ba'. Succeeds and consumes b, so Bb consumes 'abb'. Add Bb on top of tentative path.
- No need to try B. We know that it succeeds and consumes 'abb'. Add B on top of tentative path.
- Try Ba. As B already consumed 'abb', apply a to 'a'. Succeeds and consumes a, so Ba consumes 'abba'. Add Ba on top of tentative path.
- No need to try A. We know that it succeeds and consumes 'abba'. Add A on top of tentative path.
- Try Ab. As A already consumed 'abba', apply b to ''. Fails. Backtrack to A.

At this moment, we could backtrack and try an unexplored branch. But, we are after a recursion path starting with A. And we have just found it, so we can terminate the process. But, to do this, we have to answer two questions.

Question 1: already after the second step, we had recursion path starting with A. why not stop there? The answer is that we apply an arbitrary policy of being "greedy": consume as much as we can. The failure of Ab indicates that we do not find A by going farther down in the search tree.

Question 2: why not backtrack farther down to find another A the consumes more of input (not the case here, but possible in general). The answer is that we apply an arbitrary policy of "limited backtracking": accept the first good result even if we may fail to accept the input later on.

Note that the described process may be nested. In example (2), identifying the seed for E means invoking construction of recursion path for F. The same applies to the step that tries E+F.

Parsing Procedures

The process illustrated above is carried out by a set of parsing procedures.

The procedure for entry expression E implements the expression

$$E = C\$S_1 \ / \ \ldots \ / \ C\$S_n$$

where C is the name of recursion class of E and $S_1, \ldots S_n$ are names of seeds of that class.

The "seed procedure" $C\$S$ is supposed to reconstruct the recursion sequence starting with E and ending with seed S. It implements the expression

$$C\$S = S \ (\$R_1 \ / \ \ldots \ / \ \$R_n)$$

where S is the seed and $\$R_1, \ldots \$R_n$ are "ascent procedures" for recursive expressions $R_1, \ldots, R_n$ to be tried after success of S.

The ascent procedure $\$R$ depends on expression R. The ascent procedure for Choice R implements the expression

$$\$R = \$R_1 \ / \ \ldots \ / \ \$R_n$$

where $\$R_1, \ldots \$R_n$ are ascent procedures for recursive expressions $R_1, \ldots, R_n$ to be tried after success of R.

The ascent procedure for Sequence $R = e_1 e_2 \dots e_m$ implements the expression

$$\$R = e_2 \dots e_m(\$R_1 \ / \ \dots \ / \ \$R_n)$$

where, as before, $\$R_1, \dots \$R_n$ are ascent procedures for recursive expressions $R_1, \dots, R_n$ to be tried after success of R.

In case where R is the same as the entry expression E, the ending Choice is different, and becomes

$$\$R_1 \ / \ \dots \ / \ \$R_n \ / \ \varepsilon$$

which results in success when all of $\$R_i$ fail.

There is an extra complexity when the recursion class has more than one entry. In that case the expression with ε in the ending Choice depends on which expression was used as the entry. Instead of having a different set of procedures for each entry, the procedures carry entry identification as a parameter. There is also an intermediate procedure $\$\$C$ to provide the common entry point.

Another complexity arises from handling of semantic values. This requires a special version of ascent procedure for Rules to apply semantic actions.