7 How to implement a recursive descent parser

A parser is a program which processes input defined by a context-free grammar. The translation given in the previous section is not very useful in the design of such a program because of the non-determinism. Here I show how for a certain class of grammars this non-determinism can be eliminated and using the example of arithmetical expressions I will show how a JAVA-program can be constructed which parses and evaluates expressions.

7.1 What is a LL(1) grammar ?

The basic idea of a *recursive descent parser* is to use the current input symbol to decide which alternative to choose. Grammars which have the property that it is possible to do this are called LL(1) grammars.
First we introduce an end marker \$, for a given $G = (V, \Sigma, S, P)$ we define the augmented grammar $G^{\$} = (V', \Sigma', S', P')$ where

- $V' = V \cup \{S'\}$ where S' is chosen s.t. $S' \notin V \cup \Sigma$,
- $\Sigma' = \Sigma \cup \{\$\}$ where \$ is chosen s.t. $\$ \notin V \cup \Sigma$,
- $P' = P \cup \{S' \to S\$\}$

The idea is that

$$L(G^{\$}) = \{w\$ \mid w \in L(G)\}$$

Now for each nonterminal symbol $A \in V' \cup \Sigma'$ we define

$$\begin{aligned} \text{First}(A) &= \{a \mid a \in \Sigma \wedge A \Rightarrow^* a\beta\} \\ \text{Follow}(A) &= \{a \mid a \in \Sigma \wedge S' \Rightarrow^* \alpha A a \beta\} \end{aligned}$$

i.e. First(A) is the set of terminal symbols with which a word derived from A may start and Follow(A) is the set of symbols which may occur directly after A. We use the augmented grammar to have a marker for the end of the word. For each production $A \to \alpha \in P$ we define the set Lookahead($A \to \alpha$) which are the set of symbols which indicate that we are in this alternative.

$$\begin{aligned} \text{Lookahead}(A \to B_1 B_2 \dots B_n) = & \bigcup \{\text{First}(B_i) \mid \forall 1 \le k < i . B_k \Rightarrow^* \epsilon\} \\ & \cup \begin{cases} \text{Follow}(A) & \text{if } B_1 B_2 \dots B_k \Rightarrow^* \epsilon \\ \emptyset & \text{otherwise} \end{cases} \end{aligned}$$

We now say **a grammar G is LL(1)**, iff for each pair $A \to \alpha, A \to \beta \in P$ with $\alpha \neq \beta$ it is the case that Lookahead($A \to \alpha$) $\cap$ Lookahead($A \to \beta$) $= \emptyset$

7.2 How to calculate First and Follow

We have to determine whether $A \Rightarrow^* \epsilon$. If there are no ϵ-production we know that the answer is always negative, otherwise

- If $A \to \epsilon \in P$ we know that $A \Rightarrow^* \epsilon$.
- If $A \to B_1 B_2 \dots B_n$ where all B_i are nonterminal symbols and for all $1 \le i \le n$: $B_i \Rightarrow^* \epsilon$ then we also know $A \Rightarrow^* \epsilon$.

We calculate First and Follow in a similar fashion:

- First(a) = $\{a\}$ if $a \in \Sigma$.
- If $A \to B_1 B_2 \dots B_n$ and there is an $i \le n$ s.t. $\forall 1 \le k < i . B_k \Rightarrow^* \epsilon$ then we add First(B_i) to First(A).

And for Follow:

- \$ $\in$ Follow(S) where S is the original start symbol.
- If there is a production $A \to \alpha B \beta$ then everything in First(β) is in Follow(B).
- If there is a production $A \to \alpha B \beta$ with $\beta \Rightarrow^* \epsilon$ then everything in Follow(A) is also in Follow(B).

7.3 Constructing an LL(1) grammar

Let's have a look at the grammar G for arithmetical expressions again. $G = (\{E, T, F\}, \{(,), a, +, *\}, E, P)$ where

$$\begin{aligned} P = \{ & E \to T \mid E + T \\ & T \to F \mid T * F \\ & F \to a \mid (E) \end{aligned}$$

We don't need the Follow-sets in the moment because the empty word doesn't occur in the grammar. For the nonterminal symbols we have

$$\begin{aligned} \text{First}(F) &= \{\mathtt{a}, (\} \\ \text{First}(T) &= \{\mathtt{a}, (\} \\ \text{First}(E) &= \{\mathtt{a}, (\} \end{aligned}$$

and now it is easy to see that most of the Lookahead-sets agree, e.g.

$$\begin{aligned} \text{Lookahead}(E \to T) &= \{\mathtt{a}, (\} \\ \text{Lookahead}(E \to E + T) &= \{\mathtt{a}, (\} \\ \text{Lookahead}(T \to F) &= \{\mathtt{a}, (\} \\ \text{Lookahead}(T \to T * F) &= \{\mathtt{a}, (\} \\ \text{Lookahead}(F \to a) &= \{\mathtt{a}\} \\ \text{Lookahead}(F \to (E)) &= \{(\} \end{aligned}$$

Hence the grammar G **is not LL(1)**.
However, luckily there is an alternative grammar G' which defines the same language: $G' = (\{E, E', T, T', F\}, \{(,), a, +, *\}, E, P')$ where

$$\begin{aligned} P' = \{ & E \to TE' \\ & E' \to +TE' \mid \epsilon \\ & T \to FT' \\ & T' \to *FT' \mid \epsilon \\ & F \to a \mid (E) \end{aligned}$$

Since we have ϵ-productions we do need the Follow-sets.

$$\begin{aligned}\text{First}(E) = \text{First}(T) = \text{First}(F) &= \{\texttt{a}, \texttt{(}\}\\ \text{First}(E') &= \{\texttt{+}\}\\ \text{First}(T') &= \{\texttt{*}\}\\ \text{Follow}(E) = \text{Follow}(E') &= \{\texttt{)}, \$\}\\ \text{Follow}(T) = \text{Follow}(T') &= \{\texttt{+}, \texttt{)}, \$\}\\ \text{Follow}(F) &= \{\texttt{+}, \texttt{*}, \texttt{)}, \$\}\end{aligned}$$

Now we calculate the Lookahead-sets:

$$\begin{aligned}\text{Lookahead}(E \to TE') &= \{\texttt{a}, \texttt{(}\}\\ \text{Lookahead}(E' \to \texttt{+}TE') &= \{\texttt{+}\}\\ \text{Lookahead}(E' \to \epsilon) &= \text{Follow}(E') = \{\texttt{)}, \$\}\\ \text{Lookahead}(T \to \texttt{+}FT') &= \{\texttt{a}, \texttt{(}\}\\ \text{Lookahead}(T' \to \texttt{*}FT') &= \{\texttt{*}\}\\ \text{Lookahead}(T' \to \epsilon) &= \text{Follow}(T') = \{\texttt{+}, \texttt{)}, \$\}\\ \text{Lookahead}(F \to a) &= \{\texttt{a}\}\\ \text{Lookahead}(F \to (E)) &= \{\texttt{(}\}\end{aligned}$$

Hence the grammar G' **is LL(1)**.

7.4 How to implement the parser

We can now implement a parser - one way would be to construct a deterministic PDA. However, using JAVA we can implement the parser using recursion - here the internal JAVA stack plays the role of the stack of the PDA.
First of all we have to separate the input into *tokens* which are the terminal symbols of our grammar. To keep things simple I assume that tokens are separated by blanks, i.e. one has to type

```
( a + a ) * a
```

for `(a+a)*a`. This has the advantage that we can use `java.util.StringTokenizer`.
In a real implementation tokenizing is usually done by using finite automata.
I don't want to get lost in java details - in the main program I read a line and produce a tokenizer:

```
String line=in.readLine();
st = new StringTokenizer(line+" $");
```

The tokenizer `st` and the current token are static variables. I implement the convenience method `next` which assigns the next token to `curr`.

```
static StringTokenizer st;
static String curr;

static void next() {
```



```
    try {
        curr=st.nextToken().intern();
    } catch( NoSuchElementException e) {
        curr=null;
    }
}
```

We also implement a convenience method `error(String)` to report an error and terminate the program.
Now we can translate all productions into methods using the Lookahead sets to determine which alternative to choose. E.g. we translate

$$E' \to \texttt{+}TE' \mid \epsilon$$

into (using `E1` for E' to follow JAVA rules):

```
static void parseE1() {
    if (curr=="+") {
        next();
        parseT();
        parseE1();
    } else if(curr==")" || curr=="$" ) {
    } else {
        error("Unexpected :"+curr);
    }
```

The basic idea is to

- Translate each occurrence of a non terminal symbol into a test that this symbol has been read and a call of `next()`.
- Translate each nonterminal symbol into a call of the method with the same name.
- If you have to decide between different productions use the lookahead sets to determine which one to use.
- If you find that there is no way to continue call `error()`.

We initiate the parsing process by calling `next()` to read the first symbol and then call `parseE()`. If after processing `parseE()` we are at the end marker, then the parsing has been successful.

```
next();
parseE();
if(curr=="$") {
    System.out.println("OK ");
} else {
    error("End expected");
}
```

The complete parser can be found at
http://www.cs.nott.ac.uk/~txa/g51mal/ParseE0.java.
Actually, we can be a bit more realistic and turn the parser into a simple evaluator by

- Replace `a` by any integer. I.e. we use

```
Integer.valueOf(curr).intValue();
```

  to translate the current token into a number. JAVA will raise an exception if this fails.

- Calculate the value of the expression read. I.e. we have to change the method interfaces:

```
static int parseE()
static int parseE1(int x)
static int parseT()
static int parseT1(int x)
static int parseF()
```

  The idea behind `parseE1` and `parseT1` is to pass the result calculated so far and leave it to the method to incorporate the missing part of the expression. I.e. in the case of `parseE1`

```
static int parseE1(int x) {
    if (curr=="+") {
        next();
        int y = parseT();
        return parseE1(x+y);
    } else if(curr==")" || curr=="$" ) {
        return x;
    } else {
        error("Unexpected :"+curr);
        return x;
    }
}
```

Here is the complete program with evaluation
http://www.cs.nott.ac.uk/~txa/g51mal/ParseE.java.
We can run the program and observe that it handles precedence of operators and brackets properly:

```
[txa@jacob misc]$ java ParseE
3 + 4 * 5
OK 23
[txa@jacob misc]$ java ParseE
( 3 + 4 ) * 5
OK 35
```


7.5 Beyond LL(1) - use LR(1) generators

The restriction to LL(1) has a number of disadvantages: In many case a natural (and unambiguous) grammar like G has to be changed. There are some cases where this is actually impossible, i.e. although the language is deterministic there is no LL(1) grammar for this.
Luckily, there is a more powerful approach, called LR(1). LL(1) proceeds from top to bottom, when we are looking at the parse tree, hence this is called top-down parsing. In contrast LR(1) proceeds from bottom to top, i.e. it tries to construct the parse tree from the bottom upwards.
The disadvantage with LR(1) and the related approach LALR(1) (which is slightly less powerful but much more efficient) is that it is very hard to construct LR-parsers from hand. Hence there are automated tools which get the grammar as an input and which produce a parser as the output. One of the first of those *parser generators* was YACC for C. Nowadays one can find parser generators for many languages such as JAVA CUP for Java [Hud99] and Happy for Haskell [Mar01].