Jerly: a Java implementation of Earley's algorithm

Laszlo Szathmary

szathmar@loria.fr
szathml@delfin.unideb.hu

1. Introduction

In this paper we present a Java implementation of Earley's efficient parsing algorithm for λ-free context-free grammars. This algorithm will also compute a recognition matrix for each input word. The cells of the Earley matrix contain production rules in a dotted form, where the position of the dot represents the progress made by the parsing process from left to right. Each cell will contain a set of *items* of the form

$$A \rightarrow X.Y$$

where $A \in V_N$, $X,Y \in (V_N \cup V_T)^*$, and $A \rightarrow XY$ is a production rule of the given grammar.

The recognition matrix is a triangular matrix. Its cells are supplied with indices as shown in Figure 1 for an input word of length n.

0,0	0,1	0,2				0,n
	1,1	1,2				1,n
		2,2				2,n
						n,n

Table 1

The recognition matrix has one more row and one more column than the length of the input word. During the execution of the algorithm the bottom right cell (n,n) will not be used. An input word belongs to the given language if and only if the top right cell (0,n) has a dotted rule of the form $S \rightarrow U.$ at the end of the execution of the algorithm (where $U \in (V_N \cup V_T)^*$).

2. The Earley algorithm

2.1 Formal description of the algorithm

Given a λ-free context-free grammar $G = (V_N, V_T, S, H)$ and a word $P = a_1...a_n \in V_T^+$, the matrix F will be computed column by column from left to right with each column being computed bottom up except for the cells $F_{i,i}$ in the main diagonal, which are to be computed at last in each column. Initially the cells of F are supposed to be empty and they will be filled up by the following algorithm:

1) $j \leftarrow 0$
   $S \rightarrow W \in H$
   $\Rightarrow \quad S \rightarrow .W \in F_{0,0}$
2) $B \rightarrow U \in H$
   $k \leq j \qquad A \rightarrow X.BY \in F_{k,j}$
   $\Rightarrow \quad B \rightarrow .U \in F_{j,j}$
   (where $B \in V_N$; $U \in (V_N \cup V_T)^+$; $X,Y \in (V_N \cup V_T)^*$)
3) if $j = n$, then the computation is finished
   if $j < n$, then: $j \leftarrow j + 1$, $i \leftarrow i - 1$
4) $A \rightarrow X.a_jY \in F_{i,j-1}$
   $\Rightarrow A \rightarrow Xaj.Y \in F_{i,j}$
5) $i \leq k < j$: $A \rightarrow X.BY \in F_{i,k}$
   $B \rightarrow U. \in F_{k,j}$
   $\Rightarrow A \rightarrow XB.Y \in F_{i,j}$
   (where $U \in (V_N \cup V_T)^*$)
6) if $i > 0$: $i \leftarrow i - 1$, goto 4
   if $i = 0$: goto 2

If the top right cell $(0,n)$ has a dotted rule of the form $S \rightarrow U.$ at the end of the execution of the algorithm (where $U \in (V_N \cup V_T)^*$), then the input word can be generated with the grammar. Otherwise the word does not belong to the language.

2.2 Human readable rewrite of the algorithm

Now let us see what the previous steps actually mean. We will refer to the rules of the grammar as „input rules", and to the word as „input word".

Step 1

The original $S \rightarrow$ something rules are written to cell (0,0) as $S \rightarrow$ .something

Step 2

loop while $k \leq j$ (i.e. in column j we are stepping downwards cell by cell)
{
In cell $F_{k,j}$ we are looking for rules where there is a non-terminal after the dot. Let this non-terminal be B (we memorize it in a variable). Do we have original rules of the following form: $B \rightarrow$ something? If yes, then add $B \rightarrow$ .something to $F_{j,j}$ ($F_{j,j}$ is the bottom cell of the column).
}

Step 3

If $j = n$, then the computation is finished. If $j < n$, then increase j and decrease i.

The variable j points to the current column. With j we are stepping one column right. The variable i points to the current row. It will point one cell above the bottom cell, and we will move upwards with i in the next steps.

Step 4

One cell left from cell (i,j) do we have such a rule that has the j^{th} letter of the input word after the dot? If yes, then we copy this rule here (to cell (i,j)), and we move its dot one position right.

Step 5

We store the cell at position (k,j) in variable E. We store the cell at position (i,k) in variable G. Do we have such a rule in E that its dot is at the end (like $B \rightarrow$ something.)? If yes, then we memorize B, and we look in G if there is a rule there with B on the right side of the dot. If yes, then we copy this rule to cell (i,j), but we move its dot one position right. After this we step with E one cell downwards, and with G one cell to the right, and

we start again. The loop stops when we reach with F the cell, which is next to (i,j). If we wrote some new rules to $F_{i,j}$, then we have to pair cells $(0,j)$ and $(0,0)$ again!

Step 6

If $i > 0$, then decrease i and goto step 4.

If $i = 0$, then goto step 2.

It means: if we haven't arrived yet to the top with i, then we step with i one position upwards, and we jump to step 4. If we are at the top with i, then jump to step 2.

2.3 The Earley algorithm without GOTOs

Using GOTOs is not a good programming practice. Let us see the following, cleaner version of the original algorithm presented in Section 2.1:

```
n ← length of the input word;
step_1();
while (j < n)
{
   step_2(j);
   ++j;
   i=j-1;
   while (i >= 0)
   {
      step_4(i,j);
      step_5(i, j);
      --i;
   }
}
```

This is the controller block of the program.

3. Implementation

The program was implemented entirely in Java. The current version 0.0.2 (22 November 2004) consists of 15 classes. This version is a standalone console application that waits 2 parameters. The first file contains the rules of the language, the second file contains the word that we want to analyze. In the description of a language capital letters are considered to be non-terminals, while all the other characters are treated as terminal symbols. The program produces two kinds of output. The first one is written to the standard output, while the second one is written to an HTML file. The HTML file shows

the whole recognition matrix, allowing an easy understanding of the execution of the algorithm.

4. Example

Let us consider the following example:

rules.txt:

```
S->S+A
S->A
A->AxB
A->B
B->(s)
B->a
```

In the input file a rule has the following form: on the left side there can and there must be one non-terminal. Non-terminals are capital letters. Any other letters are considered as terminals.

question.txt:

```
axa+a
```

Execution of the program:

```
$ ./jerly rules.txt question.txt
```

Result on the standard output:

```
The word 'axa+a' can be generated with the grammar.
```

HTML result of the recognition matrix:

Result:

the word 'axa+a' can be generated with the grammar.

Jerly by *Jabba Laci*	0	1	2	3	4	5
0	0,0 B->.(s) S->.A A->.AxB A->.B B->.a S->.S+A	0,1 S->A. A->A.xB A->B. B->a. S->S.+A	0,2 A->Ax.B	0,3 S->A. A->A.xB A->AxB. S->S.+A	0,4 S->S+.A	0,5 S->S.+A S->S+A.
1		1,1	1,2	1,3	1,4	1,5
2			2,2 B->.(s) B->.a	2,3 B->a.	2,4	2,5
3				3,3	3,4	3,5
4					4,4 B->.(s) A->.AxB A->.B B->.a	4,5 A->A.xB A->B. B->a.

Let us see a negative example too, when the input word is 'a+b':

Result:

the word 'a+b' canNOT be generated with the grammar.

Jerly by *Jabba Laci*	0	1	2	3
0	0,0 B->.(S) S->.A A->.AxB A->.B B->.a S->.S+A	0,1 S->A. A->A.xB A->B. B->a. S->S.+A	0,2 S->S+.A	0,3
1		1,1	1,2	1,3
2			2,2 B->.(S) A->.AxB A->.B B->.a	2,3

5. Conclusion

In this paper we presented our Java implementation of the Earley algorithm, called Jerly. The program is an open source software, thus anyone can freely reuse it. As it is written in Java in a nice OO-manner, reuse and integration of the program cannot cause any problem. If you have any questions, ideas about the program, please feel free to contact the author.

6. Bibliography

Révész, György E.: "Introduction to Formal Languages", 1983, pp. 143–150.