1 Earley Parser

1.1 Introduction

An Earley parser is a parser that can handle all context-free grammars, even ambiguous ones. The worst-case complexity is $O(n^3)$ where n is the length of the input string. However, this worst-case behavior is rare, and for a large class of reasonable grammars the worst-case complexity is linear, or at most quadratic.

In addition, an Earley parser is ideal for grammars that may change dynamically. Earley parsing requires little or no preprocessing of the grammar, unlike LALR(k) parsers that must generate parser states at preprocessing time.

1.2 Terminology

The terminology we use is slightly different from that used by Earley.

An *input token* or *token* for short is an object that is produced by the *tokenizer* (also known as the *scanner*, but we shall use "tokenizer" in order to avoid confusion with the parser function called the *scanner* by Earley), such as the number 123, the identifier `hello` or a left parenthesis.

A *terminal symbol* or *terminal* for short, is the *type* of an input token such as *variable* or *number*. How to organize tokens into types is sometimes a matter of taste, and often a terminal has only one instance (such as left parenthesis).

A *nonterminal symbol* or *nonterminal* for short, is a symbol that appears on the left-hand side of a grammar rule.

A *grammar rule* or a *production* is defined as in the literature with a left-hand side in the form of a *nonterminal* and a right-hand side which is a (possibly empty) sequence of terminals and nonterminals.

An *item* is similar to a grammar rule, except that there is a dot either at the beginning, at the end, or between two symbols of the right-hand side.

1.3 How Earley parsing works

An Earley parser is defined on $n+1$ sequential *states* where n is the number of tokens of the input. Each token determines the transition from one state to the next one in the sequence.

For the purpose of this document, we shall name the states sequentially from 0, whereas in a real implementation, they would be identified with their unique pointers.

A *suffix* is a pair consisting of an item and a state number. The state number corresponds to the state in which the suffix was first introduced.

In the literature (and indeed by Earley himself), the transition from one state to the next one is described as the sequential invocation of three different parser functions, called respectively the *completer*, the *predictor*, and the *scanner*. As we shall see later on in this document, doing it this way requires a special treatment for so-called ϵ-rules, i.e., rules with an empty sequence of symbols in the right-hand side. We also give a better method that does not have this problem. In this section, however, we stick to the traditional method.

The completer is invoked only on suffixes where the dot of the corresponding item is at the end of the sequence of symbols. We call such an item *completer item* and the corresponding suffix *completer suffix*.

Its function is as follows. When it sees a completer suffix of the form $N \rightarrow a_1a_2 \ldots a_k \cdot, s$ where N is a nonterminal, $a_1a_2 \ldots a_k$ is a sequence of symbols and s is the state in which the suffix $N \rightarrow \cdot a_1a_2 \ldots a_k, s$ was introduced by prediction, it searches state s for suffixes of the form $M \rightarrow b_1b_2 \ldots b_i \cdot Nb_{i+2} \ldots b_l, t$. For each such suffix, it adds the suffix $M \rightarrow b_1b_2 \ldots b_iN \cdot b_{i+2} \ldots b_l, t$ to the current state, unless it is already present. Here, suffixes are considered equal if they have the same item and the same state of introduction.

The predictor is invoked only on suffixes where the dot of the corresponding item is followed by a nonterminal. We call such an item *predictor item* and the corresponding suffix *predictor suffix*.

Its function is as follows. When it sees a predictor suffix of the form $M \rightarrow b_1b_2 \ldots b_i \cdot Nb_{i+2} \ldots b_l, t$, it looks in the grammar for rules of the form $N \rightarrow a_1a_2 \ldots a_k$, and for each one, introduces the suffix $N \rightarrow \cdot a_1a_2 \ldots a_k, s$, where

s is the number of the current state, unless such a suffix is already present.

The scanner is invoked only on suffixes where the dot of the corresponding item is followed by a terminal. We call such an item *scanner item* and the corresponding suffix *scanner suffix*.

Its function is as follows. When it sees a scanner suffix of the form $M \rightarrow b_1 b_2 \ldots b_i \cdot T b_{i+2} \ldots b_l, t$, where T is the terminal between the current state and the state following the current one, it adds the suffix $M \rightarrow b_1 b_2 \ldots b_i T \cdot b_{i+2} \ldots b_l, t$ to the state following the current one.

For the purpose of this section, we shall use a typical expression grammar with operations for addition and multiplication:

```
E → T
E → E + T
T → F
T → T * F
F → V
```

where `E` stands for *expression*, `T` for *term*, `F` for *factor*, and `V` for *variable*.

Parsing starts in state 0 with an artificially created suffix with a predictor suffix having some arbitrary left-hand side, and a right-hand side that starts with the nonterminal we want the parser to recognize and that ends with some end-of-file marker such as `$`. The dot precedes the first symbol of the right-hand side. Here, we are going to use `% →·E $` as the artificial suffix.

Parsing stops in state n, which is detected by the fact that the tokenizer returns end-of-file. If in that state there exists a scanner suffix (assuming the EOF marker is a terminal) of the form `% → E·$`, then parsing was successful. If not, parsing failed.

Here, we represent parser states and tokens as rows of a table. The first row of table represents state 0. Thus, the initial state of the parser looks like this:

token/state	Predictor suffixes	Completer suffixes	Scanner suffixes
0	`% →·E $`		
`V`			

We always run first the completer, then the predictor, and finally the scanner. Here there are no completer suffixes, so we are done running the completer.

Running the predictor, we get two new suffixes from the rules with a left-hand side of `E`.

token/state	Predictor suffixes	Completer suffixes	Scanner suffixes

0	`% →·E $,0`
	`E →·T,0`
	`E →·E + T,0`		

`V`			

One of the newly introduced suffixes again has an `E` after the dot so no new suffixes are introduced from it. Predicting from the other one, we introduce suffixes with `T` on the left-hand side.

token/state	Predictor suffixes	Completer suffixes	Scanner suffixes

0	`% →·E $,0`
	`E →·T,0`
	`E →·E + T,0`
	`T →·F,0`
	`T →·T * F,0`		

`V`			

Finally, predicting from the suffix with a `F` after the dot, we introduce a scanner suffix (since the symbol after the dot is a terminal).

token/state	Predictor suffixes	Completer suffixes	Scanner suffixes

0	`% →·E $,0`
	`E →·T,0`
	`E →·E + T,0`
	`T →·F,0`
	`T →·T * F,0`
			`F →·V,0`

`V`			

We are now ready to run the scanner. The next input token is `V`, so the scanner suffix with a `V` after the dot generates a completer suffix in state 1.

token/state	Predictor suffixes	Completer suffixes	Scanner suffixes

0	`% →·E $,0`
	`E →·T,0`
	`E →·E + T,0`
	`T →·F,0`
	`T →·T * F,0`
			`F →·V,0`

`V`			
1		`F → V·,0`	
`+`			

We must now run the completer. We should look for suffixes with an `F` after the dot in state 0. There is one, namely `T →·F,0` which generates the completer suffix `T → F·,0` in state 1.

token/state	Predictor suffixes	Completer suffixes	Scanner suffixes

0	`% →·E $,0`
	`E →·T,0`
	`E →·E + T,0`
	`T →·F,0`
	`T →·T * F,0`
			`F →·V,0`

`V`			
1		`F → V·,0` `T → F·,0`	
`+`			

Completing from the newly introduced suffix generates another completer suffix, namely `E → T·,0` and a scanner suffix: `T → T·* F,0`.

token/state	Predictor suffixes	Completer suffixes	Scanner suffixes

0	`% →·E $,0`
	`E →·T,0`
	`E →·E + T,0`
	`T →·F,0`
	`T →·T * F,0`
			`F →·V,0`

`V`			

1		`F → V·,0`
		`T → F·,0`
		`E → T·,0`
			`T → T·* F,0`

`+`			

Running the completer again gives us the scanner suffixes `% → E·$,0` and `E`

$\rightarrow$ `E·+ T,0`

token/state	Predictor suffixes	Completer suffixes	Scanner suffixes

0	`% →·E $,0`
	`E →·T,0`
	`E →·E + T,0`
	`T →·F,0`
	`T →·T * F,0`
			`F →·V,0`

`V`			

1		`F → V·,0`
		`T → F·,0`
		`E → T·,0`
			`T → T·* F,0`
			`% → E·$,0`
			`E → E·+ T,0`

`+`			

There are no predictor suffixes in state 1, so we run the scanner. The input symbol is `+` so only the suffix `E → E·+ T,0` works, and generates the predictor suffix `E → E +·T,0` in state 2.

token/state	Predictor suffixes	Completer suffixes	Scanner suffixes

0	`% →·E $,0`
	`E →·T,0`
	`E →·E + T,0`
	`T →·F,0`
	`T →·T * F,0`
			`F →·V,0`

`V`			

1		`F → V·,0`
		`T → F·,0`
		`E → T·,0`
			`T → T·* F,0`
			`E → E·+ T,0`

`+`			
2	`E → E +·T,0`		
`V`			

There are no completer suffixes in state 2, so we go directly to running the predictor. Predicting from the nonterminal `T`, we get two more predictor suffixes, namely `T →·F,2` and `T →·T * F,2` as well as the scanner suffix `F →·V,2`.

token/state	Predictor suffixes	Completer suffixes	Scanner suffixes
0	% →·E $,0 E →·T,0 E →·E + T,0 T →·F,0 T →·T * F,0		F →·V,0
V			
1		F → V·,0 T → F·,0 E → T·,0	T → T·* F,0 E → E·+ T,0
+			
2	E → E +·T T →·F,2 T →·T * F,2		F →·V,2
V			

We now run the scanner. The input token is a V, so we get the following situation:

token/state	Predictor suffixes	Completer suffixes	Scanner suffixes
0	% →·E $,0 E →·T,0 E →·E + T,0 T →·F,0 T →·T * F,0		F →·V,0
V			
1		F → V·,0 T → F·,0 E → T·,0	T → T·* F,0 E → E·+ T,0
+			
2	E → E +·T,0 T →·F,2 T →·T * F,2		F →·V,2
V			
3		F → V·,2	
*			

Running the completer, we must look for predictor suffixes in state 2 with a F after the dot. We find T →·F,2, which generates the new completer suffix T → F·,2. Running it again on this new suffix generates suffixes E → E + T·,0 (completer) and T → T·* F,2 (scanner). The new completer suffix says we must go back to state 0 and look for predictor suffixes with an `E` following the dot. We find `% →·E $,0`, which will generate the scanner suffix `% → E·$,0` and `E →·E + T,0` which will generate the scanner suffix `E → E·+ T,0`

token/state	Predictor suffixes	Completer suffixes	Scanner suffixes

0	`% →·E $,0`
	`E →·T,0`
	`E →·E + T,0`
	`T →·F,0`
	`T →·T * F,0`
			`F →·V,0`

V			

1		`F → V·,0`
		`T → F·,0`
		`E → T·,0`
			`T → T·* F,0`
			`E → E·+ T,0`

+			

2	`E → E +·T,0`
	`T →·F,2`
	`T →·T * F,2`
			`F →·V,2`

V			

3		`F → V·,2`
		`T → F·,2`
		`E → E + T·,0`
			`T → T·* F,2`
			`% → E·$,0`
			`E → E·+ T,0`

*			

There are no predictor suffixes, so we run the scanner. The input is `*`, so only the scanner suffix `T → T·* F,2` works. This gives the following situation:

token/state	Predictor suffixes	Completer suffixes	Scanner suffixes
0	% $\rightarrow\cdot$E $,0 E $\rightarrow\cdot$T,0 E $\rightarrow\cdot$E + T,0 T $\rightarrow\cdot$F,0 T $\rightarrow\cdot$T * F,0		F $\rightarrow\cdot$V,0
V			
1		F $\rightarrow$ V$\cdot$,0 T $\rightarrow$ F$\cdot$,0 E $\rightarrow$ T$\cdot$,0	T $\rightarrow$ T$\cdot$* F,0 E $\rightarrow$ E$\cdot$+ T,0
+			
2	E $\rightarrow$ E +$\cdot$T,0 T $\rightarrow\cdot$F,2 T $\rightarrow\cdot$T * F,2		F $\rightarrow\cdot$V,2
V			
3		F $\rightarrow$ V$\cdot$,2 T $\rightarrow$ F$\cdot$,2 E $\rightarrow$ E + T$\cdot$,0	T $\rightarrow$ T$\cdot$* F,2 % $\rightarrow$ E$\cdot$$,0 E $\rightarrow$ E$\cdot$+ T,0
*			
4	T $\rightarrow$ T *$\cdot$F,2		
V			

There are no completer suffixes, so we run the predictor. We get:

token/state	Predictor suffixes	Completer suffixes	Scanner suffixes

0	`% →·E $,0`
	`E →·T,0`
	`E →·E + T,0`
	`T →·F,0`
	`T →·T * F,0`
			`F →·V,0`

`V`			

1		`F → V·,0`
		`T → F·,0`
		`E → T·,0`
			`T → T·* F,0`
			`E → E·+ T,0`

`+`			

2	`E → E +·T,0`
	`T →·F,2`
	`T →·T * F,2`
			`F →·V,2`

`V`			

3		`F → V·,2`
		`T → F·,2`
		`E → E + T·,0`
			`T → T·* F,2`
			`% → E·$,0`
			`E → E·+ T,0`

`*`			
4	`T → T *·F,2`		`F →·V,4`
`V`			

Now we run the scanner. The input it `V`, so we get the following situation:

token/state	Predictor suffixes	Completer suffixes	Scanner suffixes

0	`% →·E $,0`
	`E →·T,0`
	`E →·E + T,0`
	`T →·F,0`
	`T →·T * F,0`
			`F →·V,0`

V			

1		`F → V·,0`
		`T → F·,0`
		`E → T·,0`
			`T → T·* F,0`
			`E → E·+ T,0`

+			

2	`E → E +·T,0`
	`T →·F,2`
	`T →·T * F,2`
			`F →·V,2`

V			

3		`F → V·,2`
		`T → F·,2`
		`E → E + T·,0`
			`T → T·* F,2`
			`% → E·$,0`
			`E → E·+ T,0`

*			
4	`T → T *·F,2`		`F →·V,4`
V			
5		`F → V·,4`	
$			

We must now run the completer which generates the new completer suffix `T → T * F·,2`. Going to state 2 to look for a `T`, we generate `E → E + T·,0` and `T → T·* F,2`. Going to state 0 to look for `E` we generate `% → E·$,0` and `E → E·+ T,0`. We now have the following situation:

token/state	Predictor suffixes	Completer suffixes	Scanner suffixes
0	% →·E $,0 E →·T,0 E →·E + T,0 T →·F,0 T →·T * F,0		F →·V,0
V			
1		F → V·,0 T → F·,0 E → T·,0	T → T·* F,0 E → E·+ T,0
+			
2	E → E +·T,0 T →·F,2 T →·T * F,2		F →·V,2
V			
3		F → V·,2 T → F·,2 E → E + T·,0	T → T·* F,2 % → E·$,0 E → E·+ T,0
*			
4	T → T *·F,2		F →·V,4
V			
5		F → V·,4 T → T * F·,2 E → E + T·,0	T → T·* F,2 % → E·$,0 E → E·+ T,0
$			

We are now at end-of-file of the input. The suffix % → E·$,0 is present, so the parse was successful.

1.4 Dealing with ϵ-rules

1.5 Ambiguous grammars

Consider the grammar:

```
E → A B
A → x
A → x y
B → z
```

B $\rightarrow$ y z

This grammar is ambiguous, because the input x y z can be generated either as E $\rightarrow$ A B $\rightarrow$ x B $\rightarrow$ x y z or as E $\rightarrow$ A B $\rightarrow$ x y B $\rightarrow$ x y z.

Let us see what happens when we give the Earley parser an ambiguous grammar like that.

token/state	Predictor suffixes	Completer suffixes	Scanner suffixes
0	% →·E$ $		
x			

token/state	Predictor suffixes	Completer suffixes	Scanner suffixes
0	% →·E $,0 $ E →·A B,0		A →·x,0 A →·x y,0
x			
1	E → A·B,0	A → x·,0	A → x·y,0 B →·z, 1 B →·y z, 1
y			

token/state	Predictor suffixes	Completer suffixes	Scanner suffixes
0	% →·E $,0 $ E →·A B,0		A →·x,0 A →·x y,0
x			
1	E → A·B,0	A → x·,0	A → x·y,0 B →·z, 1 B →·y z, 1
y			
2	E → A·B,0	A → x y·,0	B → y·z, 1 B →·z, 2 B →·y z, 2
z			

token/state	Predictor suffixes	Completer suffixes	Scanner suffixes
0	`% →·E $,0 $` `E →·A B,0`		`A →·x,0` `A →·x y,0`
x			

1	`E → A·B,0`	`A → x·,0`	`A → x·y,0`
			`B →·z, 1`
			`B →·y z, 1`

y			

2	`E → A·B,0`	`A → x y·,0`	`B → y·z, 1`
			`B →·z, 2`
			`B →·y z, 2`

z			

3		`B → y z·, 1`
		`B → z·, 2`
		`E → A B·,0`
		`% → E·$,0`	

The parse is successful. Notice how the suffix `E → A B·,0` in state 3 was generated by the completer in two different ways, from `B → y z·, 1` and `B → z·, 2` respectively. This is the general pattern for ambiguities. Whenever a suffix in two or more different ways by the completer there is an ambiguity.

Ambiguities are bad because they give rise to multiple parse trees for a single grammar on a single input sequence. In fact, there can be exponentially many parse trees compared to the size of the grammar. To see that, consider the grammar:

```
E → N N N N N N N N
N → A
N → B
A → x
B → x
```

We can avoid the ambiguity problem by not trying to build *all* the parse trees, but only *one*.

1.6 Building the parse tree

⇒ `grammar` [*Class*]

⇒ `rule` [*Class*]

⇒ `rules` *grammar* [*Generic Function*]

Given a grammar, returns a list of the rules of the grammar. This function returns objects that reveal some internal state; do not modify those objects.

⇒ `target` *rule* [*Generic Function*]

Given a rule, returns the target symbol.