Earley parser

In computer science, the **Earley parser** is an algorithm for parsing strings that belong to a given context-free language, though (depending on the variant) it may suffer problems with certain nullable grammars. The algorithm, named after its inventor, Jay Earley, is a chart parser that uses dynamic programming; it is mainly used for parsing in computational linguistics. It was first introduced in his dissertation (and later appeared in abbreviated, more legible form in a journal).

Earley parsers are appealing because they can parse all context-free languages*Talk:Earley parser#*, unlike LR parsers and LL parsers, which are more typically used in compilers but which can only handle restricted classes of languages. The Earley parser executes in cubic time in the general case $O(n^3)$, where *n* is the length of the parsed string, quadratic time for unambiguous grammars $O(n^2)$, and linear time for almost all LR(k) grammars. It performs particularly well when the rules are written left-recursively.

Earley Recognizer

The following algorithm describes the Earley recognizer. The recognizer can be easily modified to create a parse tree as it recognizes, and in that way can be turned into a parser.

The algorithm

In the following descriptions, α, β, and γ represent any string of terminals/nonterminals (including the empty string), X and Y represent single nonterminals, and *a* represents a terminal symbol.

Earley's algorithm is a top-down dynamic programming algorithm. In the following, we use Earley's dot notation: given a production $X \to \alpha\beta$, the notation $X \to \alpha \bullet \beta$ represents a condition in which α has already been parsed and β is expected.

Input position 0 is the position prior to input. Input position *n* is the position after accepting the *n*th token. (Informally, input positions can be thought of as locations at token boundaries.) For every input position, the parser generates a *state set*. Each state is a tuple ($X \to \alpha \bullet \beta$, *i*), consisting of

- the production currently being matched ($X \to \alpha\ \beta$)
- our current position in that production (represented by the dot)
- the position *i* in the input at which the matching of this production began: the *origin position*

(Earley's original algorithm included a look-ahead in the state; later research showed this to have little practical effect on the parsing efficiency, and it has subsequently been dropped from most implementations.)

The state set at input position *k* is called S(*k*). The parser is seeded with S(0) consisting of only the top-level rule. The parser then repeatedly executes three operations: *prediction*, *scanning*, and *completion*.

- **Prediction**: For every state in S(*k*) of the form ($X \to \alpha \bullet Y\ \beta$, *j*) (where *j* is the origin position as above), add ($Y \to \bullet\ \gamma$, *k*) to S(*k*) for every production in the grammar with Y on the left-hand side ($Y \to \gamma$).
- **Scanning**: If *a* is the next symbol in the input stream, for every state in S(*k*) of the form ($X \to \alpha \bullet a\ \beta$, *j*), add ($X \to \alpha\ a \bullet \beta$, *j*) to S(*k*+1).
- **Completion**: For every state in S(*k*) of the form ($X \to \gamma \bullet$, *j*), find states in S(*j*) of the form ($Y \to \alpha \bullet X\ \beta$, *i*) and add ($Y \to \alpha\ X \bullet \beta$, *i*) to S(*k*).

It is important to note that duplicate states are not added to the state set, only new ones. These three operations are repeated until no new states can be added to the set. The set is generally implemented as a queue of states to process, with the operation to be performed depending on what kind of state it is.

Pseudocode

Adapted from by Daniel Jurafsky and James H. Martin

```
function EARLEY-PARSE(words, grammar)
    ENQUEUE((γ → •S, 0), chart[0])
    for i ← from 0 to LENGTH(words) do
        for each state in chart[i] do
            if INCOMPLETE?(state) then
                if NEXT-CAT(state) is a nonterminal then
                    PREDICTOR(state, i, grammar)         // non-terminal
                else do
                    SCANNER(state, i)                    // terminal
            else do
                COMPLETER(state, i)
        end
    end
    return chart

procedure PREDICTOR((A → α•B, i), j, grammar)
    for each (B → γ) in GRAMMAR-RULES-FOR(B, grammar) do
        ADD-TO-SET((B → •γ, j), chart[ j])
    end

procedure SCANNER((A → α•B, i), j)
    if B ⊂ PARTS-OF-SPEECH(word[j]) then
        ADD-TO-SET((B → word[j], i), chart[j + 1])
    end

procedure COMPLETER((B → γ•, j), k)
    for each (A → α•Bβ, i) in chart[j] do
        ADD-TO-SET((A → αB•β, i), chart[k])
    end
```

Example

Consider the following simple grammar for arithmetic expressions:

```
 ::= S      # the start rule
<S> ::= <S> "+" <M>|<M>
<M> ::= <M> "*" <T>|<T>
<T> ::= "1" | "2" | "3" | "4"
```

With the input:

```
2 + 3 * 4
```

This is the sequence of state sets:

```
(state no.) Production (Origin) # Comment
----------------------------------------
```

S(0): • 2 + 3 * 4

```
(1)  P → • S          (0)    # start rule
(2)  S → • S + M      (0)    # predict from (1)
(3)  S → • M          (0)    # predict from (1)
(4)  M → • M * T      (0)    # predict from (3)
(5)  M → • T          (0)    # predict from (3)
(6)  T → • number     (0)    # predict from (5)
```

S(1): 2 • + 3 * 4

```
(1)  T → number •  (0)    # scan from S(0)(6)
(2)  M → T •      (0)    # complete from (1) and S(0)(5)
(3)  M → M • * T      (0)    # complete from (2) and S(0)(4)
(4)  S → M •      (0)    # complete from (2) and S(0)(3)
(5)  S → S • + M      (0)    # complete from (4) and S(0)(2)
(6)  P → S •      (0)    # complete from (4) and S(0)(1)
```

S(2): 2 + • 3 * 4

```
(1)  S → S + • M      (0)    # scan from S(1)(5)
(2)  M → • M * T      (2)    # predict from (1)
(3)  M → • T          (2)    # predict from (1)
(4)  T → • number     (2)    # predict from (3)
```

S(3): 2 + 3 • * 4

```
(1)  T → number •  (2)    # scan from S(2)(4)
(2)  M → T •      (2)    # complete from (1) and S(2)(3)
(3)  M → M • * T      (2)    # complete from (2) and S(2)(2)
(4)  S → S + M •  (0)    # complete from (2) and S(2)(1)
(5)  S → S • + M      (0)    # complete from (4) and S(0)(2)
(6)  P → S •      (0)    # complete from (4) and S(0)(1)
```

S(4): 2 + 3 * • 4

```
(1)  M → M * • T      (2)    # scan from S(3)(3)
(2)  T → • number     (4)    # predict from (1)
```

S(5): 2 + 3 * 4 •

```
(1)  T → number •  (4)    # scan from S(4)(2)
(2)  M → M * T •  (2)    # complete from (1) and S(4)(1)
(3)  M → M • * T      (2)    # complete from (2) and S(2)(2)
(4)  S → S + M •  (0)    # complete from (2) and S(2)(1)
(5)  S → S • + M      (0)    # complete from (4) and S(0)(2)
(6)  P → S •      (0)    # complete from (4) and S(0)(1)
```

The state (P → S •, 0) represents a completed parse. This state also appears in S(3) and S(1), which are complete sentences.

Citations

Other Reference Materials

- Aycock, John; Horspool, R. Nigel (2002). "Practical Earley Parsing". *The Computer Journal* **45** (6). pp. 620–630. doi: 10.1093/comjnl/45.6.620 (http://dx.doi.org/10.1093/comjnl/45.6.620).
- Leo, Joop M. I. M. (1991), "A general context-free parsing algorithm running in linear time on every LR(*k*) grammar without using lookahead", *Theoretical Computer Science* **82** (1): 165–176, doi: 10.1016/0304-3975(91)90180-A (http://dx.doi.org/10.1016/0304-3975(91)90180-A), MR 1112117 (http://www.ams.org/mathscinet-getitem?mr=1112117).
- Tomita, Masaru (1984). "LR parsers for natural languages". *COLING*. 10th International Conference on Computational Linguistics. pp. 354–357.

External links

C Implementations

- 'early' (http://cocom.sourceforge.net/ammunition-13.html) An Earley parser C -library.
- 'C Earley Parser' (https://bitbucket.org/abki/c-earley-parser/src) An Earley parser C. Wikipedia:Link rot

Java Implementations

- PEN (http://linguateca.dei.uc.pt/index.php?sep=recursos) A Java library that implements the Earley algorithm.
- Pep (http://www.ling.ohio-state.edu/~scott/#projects-pep) A Java library that implements the Earley algorithm and provides charts and parse trees as parsing artifacts.
- (http://www.cs.umanitoba.ca/~comp4190/Earley/Earley.java) A Java implementation of Earley parser.

Perl Implementations

- Marpa::R2 (https://metacpan.org/module/Marpa::R2) and Marpa::XS (https://metacpan.org/module/Marpa::XS), Perl modules. Marpa (http://jeffreykegler.github.com/Marpa-web-site/) is an Earley's algorithm that includes the improvements made by Joop Leo, and by Aycock and Horspool.
- Parse::Earley (https://metacpan.org/module/Parse::Earley) A Perl module that implements Jay Earley's original algorithm.

Python Implementations

- Charty (http://www.cavar.me/damir/charty/python/) a Python implementation of an Earley parser.
- NLTK (http://nltk.org/) a Python toolkit that has an Earley parser.
- Spark (http://pages.cpsc.ucalgary.ca/~aycock/spark/) an Object Oriented "little language framework" for Python that implements an Earley parser.
- earley3.py (http://github.com/tomerfiliba/tau/blob/master/earley3.py) A stand-alone implementation of the algorithm in less than 150 lines of code, including generation of the parsing-forest and samples.

Common Lisp Implementations

- CL-EARLEY-PARSER (http://www.cliki.net/CL-EARLEY-PARSER) A Common Lisp library that implements an Earley parser.

Scheme/Racket Implementations

- Charty-Racket (http://www.cavar.me/damir/charty/scheme/) A Scheme / Racket implementation of an Earley parser.

Resources

- The Accent compiler-compiler (http://accent.compilertools.net/Entire.html)

Article Sources and Contributors

Earley parser *Source*: http://en.wikipedia.org/w/index.php?oldid=576537591 *Contributors*: 1&only, AlexChurchill, Architectual, Borsotti, Brynosaurus, Cadr, Chentz, ChrisGualtieri, Clément Pillias, Conversion script, David Eppstein, Derek Ross, DixonD, EnTerr, Fimbulvetr, Frap, Idmillington, JYOuyang, Jamelan, Jason Quinn, Jeffreykegler, John of Reading, Jonsafari, Khabs, Kimiko, Kwi, Limited Atonement, Luqui, MCiura, Macrakis, Mkartic me, Opaldraggy, Paul Foxworthy, Peak, RA0808, Rfc1394, Simon_J_Kissane, Two Bananas, UKoch, Woogyun, Zacchiro, 71 anonymous edits

License