Better Earley then never

Saturday, October 25, 2014

I wrote nearley working on course materials for a Berkeley CS course, but it quickly spiralled into a pretty big project. Perhaps more than parsing, I learned how to manage an open-source project with multiple contributors, and how to take concepts written in math-heavy notation and convert them to ideas (and code!).

There aren't many tutorials about Earley parsing, because Earley parsing has been shadowed by the recursive descent or lookahead parsers that everyone uses. (The only significant Earley project out there is Marpa; I got some help from Marpa's creator, Jefferey Kegler.) But Earley parsers are awesome, because they will parse *anything* you give them. Depending on the algorithm specified, popular parsers such as lex/yacc, flex/bison, Jison, PEGjs, and Antlr will break depending on the grammar you give it. And by break, I mean infinite loops caused by left recursion, crashes, or stubborn refusals to compile because of a "shift-reduce error".

Here's my mini-tutorial that explains Earley parsing, with an emphasis on de-emphasizing notation. It's adapted from a file that used to live in the git repo for nearley.

Primer: Backus-Naur Form

The Earley algorithm parses a string (or any other form of a stream of tokens) based on a grammar in Backus-Naur Form. A BNF grammar consists of a set of **production rules**, which are expansions of **nonterminals**. This is best illustrated with an example:

```
expression ->
     number # a number is a valid expression
```

```
      | expression "+" expression # sum
      | expression "-" expression # difference
      | "(" expression ")" # parenthesization

number -> "1" | "2" # for simplicity's sake, there are only 2 numbers
```

This small language would let you write programs such as `(1+2+1+2)-1-2-1`.

`expression` and `number` are *nonterminals*, and `"+"` and `"-"` are *literals*. The literals and nonterminals together are **tokens**.

The **production rules** followed the `->`s. The `|`s delimited different expansions. Thus, we could have written

```
number -> "1"
number -> "2"
```

and it would be an identical grammar.

For the rest of this guide, we use the following simple, recursive grammar:

```
E -> "(" E ")"
   | null
```

this matches an arbitrary number of balanced parentheses: `()`, `(())`, etc. It also matches the empty string . Keep in mind that for a parsing algorithm, this is already very powerful, because you cannot write a regular expression for this example.

Algorithm

Earley works by producing a table of partial parsings.

(Warning: some notation is about to ensue.)

The nth column of the table contains all possible ways to parse `s[:n]`, the first *n* characters of *s*. Each parsing is represented by the relevant production rule, and a **marker** denoting how far we have parsed. This is represented with a dot • character placed after the last parsed token.

Consider the parsing of this string `()` with the grammar `E` above. Column `0` of the table looks

like:

```
# COL 0
1. E -> • "(" E ")"
2. E -> • null
```

which indicates that we are expecting either of those two sequences.

We now proceed to process each entry (in order) as follows:

1. If the next token (the token after the marker •) is `null`, insert a new entry, which is identical excpept that the marker is incremented. (The `null` token doesn't matter.) Then re-process according to these rules.
2. If the next token is a nonterminal, insert a new entry, which expects this nonterminal.
3. If there is no expected token (that is, the marker is all the way at the end), then we have parsed the nonterminal completely. Thus, find the rule that expected this nonterminal (as a result of rule 1), and increment its marker.

Example!

Following this procedure for Column 0, we have:

```
# COL 0 [processed]
1. E -> • "(" E ")"
2. E -> • null
3. E -> null •
```

Now, we consume a character from our string. The first character is `"("`. We bring forward any entry in the previous column that expects this character, incrementing the marker. In this case, it is only the first entry of column 0. Thus, we have:

```
# COL 1, consuming "("
1. E -> "(" • E ")" [from col 0 entry 1]
```

Processing, we have (you can read the comments top-to-bottom to get an idea of how the execution works):

```
# COL 1
```

```
# brought from consuming a "("
1. E -> "(" • E ")" [from col 0 entry 1]

# copy the relevant rules for the E expected by
# the first entry
2. E -> • "(" E ")" [from col 1 entry 1]
3. E -> • null [from col 1 entry 1]

# increment the null rule
4. E -> null • [from col 1 entry 3]

# entry 4 is completed, so we increment entry 1
5. E -> "(" E • ")" [from col 0 entry 1]
```

Notice how we must keep track of where each entry was added so that we know which entry to increment when it is completed.

Next, we consume a `")"`, the second (and last) character of our string. We have:

```
# COL 2, consuming ")"

# brought from consuming a ")"
1. E -> "(" E ")" • [from col 0 entry 1]
```

Nothing further can be done, so the parsing is complete. We now find entries that are complete and created from an entry in column 0. (That means we have a parsing from the beginning of the string to the end). Since we have such an entry in column 2, this represents the parsing.

Finale

Nearley parses using the above algorithm, but giving each entry “baggage”, namely the parsed data as a tree structure. When we finish an entry (and are about to process it with rule 3), we apply the postprocessor function to the baggage. Once we determine a parsing, we can reveal—with a flourish—the postprocessed data to be used by the user.

Parting words

If we had multiple entries that worked in the end, there would be multiple parsings of the grammar. This means the grammar is **ambiguous**, and this is generally a very bad sign. It can lead to messy programming bugs, or exponentially slow parsing.

It is analogous to the confusion generated when one says

> *I'm really worried Christopher Nolan will kill a man dressed like a bat in his next movie. (The man will be dressed like a bat, I mean. Christopher Nolan won't be, probably.)*