Top Down Operator Precedence Parsing

Marin Jovanović

Introduction

"Parsing is the process of structuring a linear representation in accordance with a given grammar" (Grune and Jacobs, 2008). Such broad abstraction of definition implies great application area, therefore it is of critical value to implement it in an optimal way into the code.

In 1973 Vaughan Pratt suggested his method for parsing. He called it *top down operator precedence*, today it is also known as *Pratt parser*.

His motivation was that in the process of language designing and implementing some preoccupy with syntax while others deny its utilization. His solution was compromise between those two; parser which when writing feels more like grammar then program. In other words, he embedded syntax directly into code (Pratt, 1973).

Related work

Automaton theory found great application in parsing. Diverse grammars are put in use of defining languages and automatons are used for construction of parsers (mostly LR, its derivatives and recursive descent).

Through years the need for parser generators has grown. Yacc[1] and Bison[2] are examples of parser generators but there are many more[3]. Such solution is adequate when in need for parser where optimization is not an issue or if the user is not in position to write his own parser. Drawback of generators is that they can not be fined tuned as hand written parsers, consequently, hand written parsers are sometimes more suitable.

Pratts method is one way to solve this problem. He stated that his method is great for error handling and that it is trivial to implement (Pratt, 1973).

Problem and solution: from recursive descent to Pratt parser

The algorithm is based on an operator precedence and recursive descent (Crockford, 2007). I will build it from pure recursive descent parser while using functions as main callable units.

---

[1] http://dinosaur.compilertools.net/yacc/index.html
[2] https://www.gnu.org/software/bison/
[3] refer http://catalog.compilertools.net/lexparse.html and https://java-source.net/open-source/parser-generators

Recursive descent parsers have a drawback when parsing left recursive grammars. For any production that has the following structure: **S → Sa | a**, algorithm for recursive parsing would look like this:

```
function S() {

  // try first production
  S();
  a();

  // if first production fails try second production else raise exception
  a();

}
```

code snippet 1 (Kladov, 2020)

The problem with this algorithm is that it will loop infinitely in recursion. This is usually solved by rewriting algorithm like this:

```
function S() {

  a();

  while (get_next_token() == a) {
    a();
  }

}
```

code snippet 2 (Kladov, 2020)

Pratt used a more general approach using recursion and loop (Kladov, 2020):

```
function expression(rbp) {

  t = token
  token = get_next_token()
  left = t.nud()

  while (rbp < token.lbp) {
    t = token
    token = get_next_token()
    left = t.led()
  }

  return left
}
```

code snippet 3 (Bendersky, 2010)

At the start of parsing function *expression* will be called with an argument equal to zero and it will return parsed data structure (this is regularly AST). Tokens with same properties will be part of the same class.

Pratt used the following syntax for variables and functions:

- lbp – left binding power
  Each token class has its own.

- rpb – right binding power
- nud() – null denotation
  Function which handles prefixes. This function is called when for a given token we can predict which tokens need to follow it and bind them (i.e., in most languages after *if* we can expect Boolean expression).
- led() – left denotation
  Function which handles infixes and suffixes. This function is called when for a given token we can bind tokens that are left to it and optionally right.

(Pratt, 1973)

Each token class can contain function *nud* and *led*. They return formatted token structure. If logically *nud* or *led* are not necessary, they can be used for error handling.

Parser driver is the following function:

When function is called it tries to match tokens for the given prefix *t* and assign a given value to variable *left*. Then it tries to bind tokens that are left to *token* and optionaly right to it for as long as the current *rpb* is lower than current tokens *lpb*.

The power of pratt parser is in this simplicity. Every token is passed through this (code snippet 3) function (interface). Every token class has the same form. From here we can agree with Pratt that algorithm feels more like writing grammar then code.

For full implementation of this algorithm I suggest the following article[4]. In this explanation I heavily relied on that article.

Experiments

There have been made many articles on the topic of *top down operator precendance* algorithm. One of the implementations using toy language is extensively explained in the article[4].

Pratt stated that his algorithm has been implemented in SCRATCH-PAD and in MACSYMA (Pratt, 1973).

Crockford exploited this algorithm for his application JSlint[5]. He also wrote a paper on how it works[6] and displayed code publicly[7].

Another interesting use of this method is for application Desmos[8]. They made an article[9] on how they put it to use. They stated that they encountered a memory overflow because of recursion calls and that there were no guarantee that parser will work intendedly for every case (Lantsman, 2018).

[4] https://eli.thegreenplace.net/2010/01/02/top-down-operator-precedence-parsing
[5] http://jslint.com/
[6] https://crockford.com/javascript/tdop/tdop.html
[7] https://github.com/douglascrockford/JSLint

Conclusion

Top down operator precedence is simple parsing method for hand written parsers. It resolves around operator precedence and recursive descent (Crockford, 2007).

It implements grammar directly into code (Pratt, 1973). It is best utilized when it is in service of dynamic and functional programming languages (Crockford, 2007). It is also a good method when error handling is of high priority (Pratt, 1973).

On the other hand, memory overflow is possible and it needs thorough testing, therefore sometimes other methods are more applicable (Lantsman, 2018).

References

AHO, Alfred V. et al. 1986. *Compilers: Principles, Techniques, & Tools*. Boston: Addison-Wesley

BENDERSKY, Eli. 2010. Top-Down operator precedence parsing. Available at: https://eli.thegreenplace.net/2010/01/02/top-down-operator-precedence-parsing (Accessed: 14 January 2021)

CROCKFORD, Douglas. 2007. *Top Down Operator Precedence*. Available at: https://crockford.com/javascript/tdop/tdop.html (Accessed: 13 January 2021).

GRUNE, Dick and Ceriel J.H. JACOBS. 2008. *Parsing Techniques*. New York: Springer Science+Business Media, LLC.

KLADOV, Aleksey. 2020. Simple but Poweful Pratt Parsing. Available at: https://matklad.github.io/2020/04/13/simple-but-powerful-pratt-parsing.html (Accessed: 13 January 2021)

LANTSMAN, Denis. 2018. *How Desmos uses Pratt Parsers*. Available at: https://engineering.desmos.com/articles/pratt-parser/ (Accessed: 13 January 2021).

PRATT, Vaughan R. 1973. "Top down operator precedences". *Proceedings of the 1st annual ACM SIGACTSIGPLAN symposium on Principles of programming languages*, 41-51

[8] https://www.desmos.com/
[9] https://engineering.desmos.com/articles/pratt-parser/