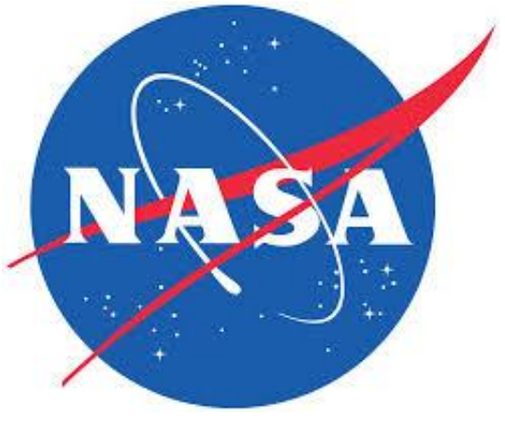
Converting ADQL's Grammar from BNF to PEG

Walter Landry



What?

- PEG (Parsing Expression Grammar), like BNF (Backus-Naur Form), is another way of specifying grammars
- It is naturally aligned with how recursive descent parsers work
- Unlike BNF, it is never ambiguous

But

- In practice, this means debugging left recursion problems rather than shift-reduce problems
- Tool support is different
  - Many libraries (PEGTL, Boost::Spirit, PEGjs)
  - But not the old familiar tools (e.g yacc, bison, antlr)

Translating

- In principle, there are BNF grammars that have no PEG equivalent.
- In practice, there is always an equivalent.
- But this means that there is no automatically convert a BNF grammar into PEG.

Already done?

- I sort of already created a PEG grammar for ADQL when I wrote my ADQL parser library
- I used boost::spirit, which uses operator overloading to emulate a PEG DSL in C++.

libadql

- https://github.com/Caltech-IPAC/libadql
- Comes with 195 tests
- The generated parser is fast
- Compiling it is slooooooooow
  - and uses lots of memory
  - I had to split up the long list of keywords to get the memory use under a GB.
- Template errors can be intimidating.

libadql vs BNF

- 2639 lines of code according to cloc
- 1/3 is the grammar
- 2/3 are all of the various types for the AST
- Does not have a full AST. Just the parts that I cared about.
- In comparison, the BNF has 512 non-blank lines.

Only Mostly Done

- The DSL is close to PEG, but not the same
  - e.g. prefix '+' rather than postfix '+'
- Proper backtracking is finicky, requiring some hacks.
- Loading the code properly into an AST can require even more hacks.
- About 2 days of work to convert.

Spaces Cause Trouble

- The BNF is ambiguous about when spaces are part of the grammar.

```
[sign] <unsigned_integer>                          Prohibited
ACOS '(' <numeric_value_expression> ')'  Allowed
<search_condition> OR <boolean_term>       Required
```

- But it is all implicit. So I needed to discern the intent of the rule. The tests were really helpful.

Reserved Keywords Aren't

- The BNF defines a large number of reserved words, but never uses them in the grammar.
- I ended up making reserved words illegal for identifiers
- I also reserved TAP_UPLOAD. Not sure if I really needed to.

Fixing Left Recursion is Usually Easy

- rules like

  ```
  a | (a b+)
  ```

  become

  ```
  (a b+) | a
  ```

- recursive joins was the only tough one

Missing PEG Capabilities?

- Case insensitive match
- Expectation Parser
  - The entire parse fails if a rule does not match. Not allowed to backtrack and try something else.
  - Useful if, for example, you are in the middle of parsing a function and do not find the closing braces.
  - Really useful for comprehensible errors

Results

- Fully (?) working PEG grammar
- 583 non-blank lines
- Still rough. Could be simplified further.
- I fed it into PEGjs and it passed all of my 195 tests.
- Retains some quirks from my implementation: SQL 99 arrays, hard coded reference frames, limited geometry support, CAST operator.

Useless Rules?

- Some rules complicate the grammar without significant benefit.
- No one (?) actually evaluates the queries. We only rewrite them for our back ends.
- For example, there is a special rule for UDF's, but the regular function syntax already covers them.

No Math?

- Similarly, checking airity on math functions can be done by the back end.
- The ADQL spec can require the existence of these functions, but it does not have to be in the grammar.
- Implementations might want to check it in the grammar to give better error messages, but it would not be required.