Compiler Construction

Lucas Franceschino

June 15, 2017

Contents

1 Choice of JavaScript

2 Lexer and Parser

2.1 Global design

I wanted to seperate the language definition and the mechanism of the lexer and parser. That's why I decided to develop theses bricks in a modulable way. In order to use the lexer or the parser, you need to initialize a class and configure it with some specifications about the language.

2.2 Lexer

The lexer does not works with streams (yet?): it process the whole file, build up a list of tokens, and eventually give it to the parser.

The lexer alone do not specify any language specific rules. When using the lexer, you need to provide a list of rules describing the authorized tokens in the language you want to tokenize. Theses rules are stored as an array of `LexerRule`.

2.2.1 Rules

A rule consist of an regular expression and a `TokenKind` (a string to classify the matched token). We also can specify an ignore flag: if so, the matched token is dropped immediately by the lexer. For example, some languages are not layout sensible, then you can specify a rule matching any space or tab with a ignore flag.

The order of the rules in the given array does matter. If a matching rule B follows an other matching rule A, then - without any consideration on the length of any matches - the match from A will be used.

2.2.2 Functionning

We start working on the caracter right after the last matched token. If there is no any matched token, then we start at the begining. Then, we try each regular expressions in the right order: if any is matched, we stop and we loop. Else, there is an error.

Each tokens keep trace of the orginal string and position from the program source.

2.2.3 Error handling

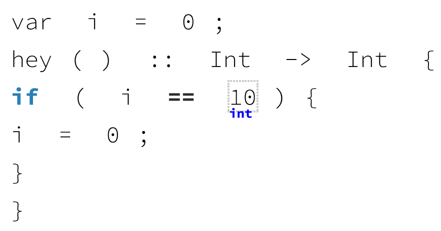
You can provide some "error rules" to the lexer. Theses rules are applied when an error occur. This make the lexer's user able to customize errors according to specific contexts. For example, we can detect mal-formed number: some digits followed by a letter for example.

An error consist of a position in the original program source, a length, and a message. This way, the Lexer and even the Parser can easily generate nice errors.

The output is colored: the line and number of caracter are given, the source line is quoted, with the conflicting part colored.

2.2.4 Debuging

The class lexer provide a `saveAsHtml` method that convert the token list into an HTML document rendering a syntax highlighting. If you drag your mouse over any token in this HTML document, it will show you its kind of token and its size.
The figure on the right show the HTML render of a simple program tokenized.



```
var  i  =  0 ;
hey ( )  ::  Int  ->  Int  {
if  (  i  ==  10 ) {
i  =  0 ;
}
}
```

2.3 Grammar

As explained previously, I wanted to split parser and language. That's why I made a tiny parser for a tiny grammar description language.

Sublime text highlight definition In order to work with the grammar confortably, I made a syntax definition for my favorite text editor, Sublime Text. You can find it in the tools directory.

2.3.1 Syntax

Each line is a rule. A rule looks like that : `Name` $\texttt{Step}_1$ $\texttt{Q}_1$, $\texttt{Step}_2$ $\texttt{Q}_2$, ..., $\texttt{Step}_n$ $\texttt{Q}_n$. The $\texttt{Step}_i$ will be described later. Indeed, a grammar rule consist of a series of tokens to match. $\texttt{Q}_1$ is optionnal; it can be `+` (ie match this step one or more time) `*` (ie match this step zero, one or more time), `?` (match this step zero or one time).

A `Step` is of the form : $\texttt{Pattern}_1$ `|` $\texttt{Pattern}_2$ `| ...|` $\texttt{Pattern}_m$. A pattern is either a name (unquoted) of a rule, or a string refering to a `TokenKind` (defined in one of the lexer's rules). A pattern can refer to a non yet existing rule. (circular references are then allowed - hopefully)

Tree manipulation You can use two modifier to perform basic tree manipulations: `#drop` and `#flattenIfNested`.

`#drop` operate on a node (let's denote it A) with only one child B: A is remplaced by B. Indeed, sometimes, since this tiny grammar language is not very permissive, you need to split some rules to match some more complicated patterns: you don't really want that to be reflected in the AST. Then this drop modifier make that "artificial" node invisible.

`#flattenIfNested` on a node A will flatten children of A whose type is A also. If you match something using a direct recursive rule, then this modifier will turn a tree like $[a_1, [a_2, [a_3]]]$ into $[a_1, a_2, a_3]$ (where a_i are nodes whose type is the same as A).

You can also provide a name for each step: for instance `Step* as ident`. By naming each steps, it is easier to, later, manipulate the tree. Instead of saying "the first matched step", you say "the step named *something*". This is particulary useful in the PrettyPrinter.

Check out the `grammar.gr` provided for a more comprehensive view of the syntax and the possibilities.

2.3.2 Parser

The parser is very simple and very specific for this tiny language: we divide each lines, we filter comments, and we "consume" each steps. Each rules is "compiled" into a closure and registred (at first as an empty rule) into a cache, then after all rules compiled, we execute all closure. This allow recursive references over rules. The result is a JavaScript object (ie kind of a dictionnary) containing the main rule.

»»»»»» **ICI - TODO** ««««««

2.4 Program generator

After defining the grammar and the lexer rules, we know everything about what is a correct program for the parser. Then, in order to find some eventual buggy constructions, I decided to make a program generator.

By going through the grammar, we can generate a random program. If a step is mandatory, then no randomness is possible, but each time there is a modifier, we can apply some randomness. This way we build a sort of tree of the future program.

The leafs are the token kind specified by the grammar. Theses token kinds are directly linked to regular expressions. RandExp, is a JavaScript library generating random strings matching a given regular expression. Using that library, I can generate any tokens.

The only issue is the size of the program generated: since it is random, I have no control over that. To solve that issue, I put some guardrail: after a certain (customizable) level of deepness, the algorithm try to stop as soon as possible to generate things (when the quantifier of a step is $*$ or ?, we skip it; when it is $+$, we apply it only once).

Also, after that limit, we avoid circular patterns used already as much as possible. For exemple, if we are in the expression $1+2+3+4$ (witch is in fact $\{1+\{2+\{3+4\}\}\}$ in the call stack), the circular rule managing the binary operators is used already 3 times, so we avoid it (if possible).

The result is that the length of the generated program can explode but is still reasonnable, witch is what I wanted.

State I'm running this program generator for hours against my lexer and parser to see if there are some bugs. I spotted many errors (both due to grammar design and lexer/parser bugs), that I fixed. Now, it looks like it is ok.

Issue I introduced misformed error detection rules (i.e. avoiding $===$), but that kinds of "excluding rules" are "bad": that can lead to inconsistencies and the program generator can't efficiently respect them. I think I will remove that from the lexer design.

Issue The generator do not take into account reserved words. Then, some errors can occurs. The solution

2.5 Parser

2.5.1 Optimization

By analysing the grammar, I optimize the Parser: for each possible rule, we detect the possible starting lexems. Then, for any steps, we build a function that associate a lexem with a list of possibl rules. Then we avoid loads of recursion detections.

Using this behaviour, the execution time was drastically reduced.

face