`cl-jackc`'s Grammar Language

Lucas Vieira*

May 7, 2019

ABSTRACT

`cl-jackc`[1] is an implementation of a compiler for the Jack programming language. It uses its own grammar engine for $LL(1)$ grammars, written from scratch in Common Lisp. This article outlines the domain-specific language for specifying grammar rules, its key points, and my experience on building the matching engine which interprets the grammar language.

1 Introduction

Building programming languages are often an arduous, but fun task. Some of the work lies on the designing of the language and its syntactical elements *per se*. The rest of the work lies on building the tools to make the language work, namely the compiler toolchain. In this article, I discuss a fragment of the compiler toolchain for the Jack programming language, one of the projects proposed by Nisan and Schocken (2008) [2], which is the parser for the syntax analysis of the Jack programming language.

The related book provides a very clear grammar for Jack, and so the job of the reader is to simply write a compiler for the language, in two different projects[2]. Here I show some of my strategies to build the first of the projects, while having zero experience on writing compilers.

Nisan and Schocken (2008) [2] provide a blueprint for the project software architecture in the Java language, so that the job of the reader is sped up. This model, however, obeys certain principles of object-oriented design which describe a very rigid software architecture. Due to influence of Abelson and Sussman (1996) [1], when primarily thinking of what to do, I did not view the compiler architecture as a rigid hierarchy of software modules, interacting with each other. While the modularization is still maintained,

*lucasvieira@protonmail.com

[1] `https://github.com/luksamuk/cl-jackc`

[2] The compiler is actually built in more than two projects, as the rest of the toolchain (Assembler and VM Translator) is built in chapters prior to the compiler-related chapters.

mostly for file organization and for keeping some degree of similarity with the proposed implementation, the syntax analyzer was primarily designed as layers of software comprising of certain abstractions, as if each of these layers were *domain-specific languages* themselves that could be used to build upper abstraction layers.

This approach led me to build a de-facto *domain-specific language* to represent the grammar of the Jack programming language in the topmost level. It also acquired the intended effect of having an underlying robust, simple and well-organized codebase. The formalisms and operation of this topmost grammar language are detailed as follows.

2 Grammars

Newton (2006)[3] states that a grammar is a formalism originally conceived for defining formal languages. The fundamental elements of grammars are the *rules*, which demonstrate how to generate a language's *words*.

Let there be a grammar G, which is a tuple (V, Σ, R, P) where:

- V is a finite set of all language *variables*;
- Σ is the language's alphabet, where $V \cap \Sigma = \varnothing$;
- R is a finite set comprised of ordered pairs known as *rules*, and
- $P \in V$ is a variable known as a starting point.

A grammar G is valid when it is possible to generate *sentential forms* from its rules, such that these *forms* are not populated by any *variable*, but only with *terminals* ($\alpha \in \Sigma$). These special *forms* are often called *words*.The set of all *words* generated by a grammar is called $L(G)$, or *language of* G.

It is said that a *word* ω belongs to $L(G)$ if, and only if, ω can be derived in n steps from G's *rules*.

The *rules* of a grammar, as already stated, are ordered pairs. The first element of the pair states *what should be replaced*, and the second element states *what is the replacement*. For example, consider the following rules.

$$A \to aA \tag{1}$$
$$A \to B \tag{2}$$
$$B \to \lambda \tag{3}$$

These *rules* belong to a certain type of grammar which will be crucial to the incoming object of discussion. Given a starting point A, as per Rule 1, it is possible to infer that this language is capable of generating *words* with an arbitrary number of "a" *terminals*, plus a dangling "A" variable at

the end. After that, A can be transformed into B (Rule 2), and B is then transformed into the *empty word* λ (Rule 3), which acts as the stopping point for grammar generation, leaving only a *word* comprised of *terminals*. This language may also be represented as $\{a\}^*$.

The previous example is interesting because the left-hand part of all rules is comprised only of *variables*. This means that, for any given variable which may appear in a *sentential form*, that single variable must have one or more corresponding rules which replace it. Furthermore, it is only necessary to find and replace *variables* in the *sentential form*, and never to analyze the disposition of *terminals*, since there are no *rules* requiring this procedure.

A grammar for the Jack language [2] is written using exactly this model. Each of the *rules* are a simple pair, where the first component is a *variable* which may appear anywhere, and so it can be transformed by another given rule of the grammar. Using this simple theorem, it is possible to write an application which takes a *source code* and parses it, so that this analysis indicates whether the *code* is a *word* of the Jack *language* or not.

3 Compiler's grammar syntax

The compiler's grammar language is written using s-expressions, since the application itself is written in Common Lisp. The chosen language provides powerful abstractions for dealing with lists. The grammar is written as an abstract syntax tree, so that Lisp makes it easier to traverse each rule.

A grammar is a list of rules. Each rule is a pair of two elements: a *name* and a *match*. There must be at least one rule enumerating *terminals*, which are expected textual elements.

Names are Lisp keywords[3], and therefore are prepended with a double colon, so that they are agnostic to the namespace[4] they're in.

Figure 1 is a transcribed fragment of a single rule for matching with a class structure of the Jack programming language. The rule is provided for illustrative purposes.

```
(:class   ((:keyword "class") :class-name (:symbol "{")
           (:many :class-var-dec)
           (:many :subroutine-dec)
           (:symbol "}")))
```

Figure 1: Rule for matching a class definition in the Jack language.

[3]This choice was made due to Common Lisp's structure regarding *packages*, which are analogous to C libraries in some ways. A Lisp `KEYWORD` is agnostic to context, as it belongs to its own package, whereas the next obvious choice (a Lisp `SYMBOL`) is not.

[4]For better understanding, the word *namespace* was used, though not technically correct.

The *match* component of a rule is, in itself, a list, and so it carries its own meaning. The *match* may be populated by other *keyword names*, *quantified rules*, *precise rules* and *exact-matches*, which must also be specified in the grammar itself.

Any valid Jack source code begins with a class definition, and so `:CLASS` corresponds to the starting point for the matching engine. The code snippet at Figure 1 offers a way to expect a class definition in the Jack programming language (adapted from Nisan and Schocken (2005) [2]). To recognize a class, the grammar expects the match of a *precise* pattern, in the following order:

- An exact match with the keyword `"class"`;
- The class's name, whatever it may be;
- An exact match with the symbol `{`;
- Zero or more variable declarations, whatever it may be;
- Zero or more subroutine declarations, whatever it may be;
- An exact match with the symbol `}`.

The matching engine works recursively, as it is suggested from the *wishful thinking* approach (Abelson and Sussman (1996) [1]). These operations (as well as the *match* portion of the rule itself) require better clarification of their underlying structure.

A valid grammar, when written in the grammar language, is comprised of atomic rules, which enumerate most *primitives*, and of compound rules, which enumerate a single kind of *primitive* and some *means of combination* for the rest of the rules.

Atomic rules

A rule is atomic if it is comprised of a single keyword. These rules are of the same type of the rule *names*, as they are meant to be replaced by the body of another rule by the matching engine.

There are also built-in atomic rules which do not need to appear on the grammar, since they are primitive to the matching engine, and so they are enumerated as follows:

- `:IDENTIFIER`: Any name which does not start with a letter. Breaks before a `:SYMBOL` or any whitespace.
- `:STRING-CONSTANT`: Any text surrounded with quotes. Cannot have any line breaks.

- `:INTEGER-CONSTANT`: Any text comprised only of numbers and no `:SYMBOL` terminals.
- *Terminals*: Any text. The text can be written as a string constant for Lisp.

Since matching these rules require knowledge of what a `:SYMBOL` and a `:KEYWORD` are, these rules must always be defined for any grammar. They can also be compared to the alphabet Σ of a formal grammar (Vieira (2006) [3]). This aspect will be discussed later, in greater detail.

In a rule such as `(:KEYWORD "class")`, the element `"class"` is a *terminal*, as it is raw text expected to be at the matching source code position. However, the rule as a whole is not atomic, as will be further discussed in the following subsection.

Compound rules

Any list in the *match* element is a compound rule. Since the *match* element itself is a list comprised of several sub-rules, it is also considered as a compound rule, in accordance to one of the following archetypes.

Quantified rules

A *compound rule* is a *quantified rule* when its first element is a quantifier keyword. The quantifier changes the matching engine context for the elements it encloses, following the meaning of the quantifier keywords:

- `:OR`: A disjunction of rules. Attempts to match, in order, each of the sub-rules it encloses. Stops when one of the rules is matched, and does not check for the remaining rules.
- `:MAYBE`: Attempts to sequentially match the group of all enclosed sub-rules, but the matching is optional; failure on the matching process does not fail the rest of the grammar match (*zero-or-one*).
- `:MANY`: Attempts to match the group of all enclosed sub-rules exhaustively, and keeps collecting the matching results until the repeating match fails (*zero-or-more*).

A *quantified rule* such as `(:many :identifier)`, for example, will keep collecting identifiers until there are no more identifiers to be collected. When matching an identifier fails, then all previously matched identifiers are collected. If no identifier was matched, the match results in a neutral value, but never fails.

A rule such as `(:maybe :identifier :integer-constant)` will attempt to match an identifier and then an integer constant. If any of those structures are not matched, then the match results in a neutral value, but never fails.

The rule `(:or :identifier :integer-constant)` attempts to match an identifier. If the identifier is not found, it attempts to match an integer constant. If the integer constant is also not found, then the match results in failure.

Exact-matches

A *compound rule* is an *exact-match rule* when comprised of two elements, where its first element is an existing rule in the grammar, and the second element is an expected *terminal*.

The *terminal* element of an *exact-match rule* must belong to a disjunctively-quantified rule, where each element of the disjunction is a *terminal* text as well. This associated, disjunctive rule is not supposed to be used in matching time, though it is important for grammar verification.

Any *exact-match rule* which uses a non-existing disjunctive rule, or uses a *terminal* which does not belong to the associated disjunctive rule, is considered to be syntactically incorrect.

A rule such as `(:KEYWORD "class")` is a well-defined *exact-match rule*, if and only if the grammar contains a rule as exemplified in Figure 2.

```
(:KEYWORD   ((:OR "class" "constructor" "function" ...)))
```

Figure 2: Example of a supporting rule for an *exact-match rule*.

Precise rules

A *precise rule* is the commonest type of rule, as the *match* element of a rule definition often falls into it. Moreover, any non-disjunctively quantified rule ends up degenerating into a *precise rule*, only changing the context where such *precise rule* fails.

Any part of a *match* element of a rule, which is also *compound*, but does not fall into the previous categories, is a *precise rule*. In other words, such rules are surrounded by parenthesis, but their first element is not a quantifier, and they also do not fit the *exact-match rule* specification.

These *compound rules* are basically enclosings for sub-rules which must "travel" as a group, and so all of their sub-rules must always match. Since a rule definition enumerates how it works, it is advised that any *match* portion of a rule definition should be a *precise rule* itself.

A rule such as `(:IDENTIFIER (:SYMBOL "=") :INTEGER-CONSTANT)` is a valid *precise rule*. The grammar expects three sub-rules to be matched sequentially. Should any of them fail, then the whole group fails.

Obligatory rules

As the matching engine was implemented, it was discovered that the structure needed to assume that certain rules were to be always expected. These obligatory rules, however, are lexical elements which are common to most languages.

- `:KEYWORD`: A disjunctively-quantified rule enumerating all text *terminals* of the language which are *language keywords*.
- `:SYMBOL`: A disjunctively-quantified rule enumerationg all text *terminals* of the language which are *language symbols*.

Keywords and symbols enumerate nothing less than the alphabet Σ of a language, and therefore act as the primary *terminals* for any other grammar rules.

Figure 3 is a snippet showing the Jack language's keywords and symbols, based on the language's grammar specification (Nisan and Schocken (2008), pp. 208-209 [2]).

```
(:keyword  ((:or "class" "constructor" "function"
                 "method" "field" "static" "var"
                 "int" "char" "boolean" "void"
                 "true" "false" "null" "this" "let"
                 "do" "if" "else" "while" "return")))

(:symbol   ((:or "{" "}" "(" ")" "[" "]" "." ","
                 ";" "+" "-" "*" "/" "&" "|" "<"
                 ">" "=" "~")))
```

Figure 3: Keywords and symbols as defined for the Jack language.

Code commentary

Comments in any language are often surrounded by two tokens, which are equivalent to text *terminals*. Comment detection, however, is not considered part of the grammar rules, as they are treated as a responsibility of the tokenizer.

The matching engine's topmost structure is responsible for comparing tokens which are already assumed to be valid, and comment delimiters (plus the contents of any comment) are not supposed to be valid tokens. For that reason, the comments are defined using an outside structure, a list of pairs which enumerate the tokens enclosing any comments[5].

[5]This is mostly an arbitrary choice, and the comments may be easily incorporated in the grammar. However, comments should not relate to grammatical rules themselves.

Let us take the Jack programming language as an example, as the matching engine was primarily built for it. A pair, in Lisp notation, such as `("/*" . "*/")` shows that there may be comments in the source file, which begin with the token `/*` and end with the token `*/`. And so, the matching engine's head, which is supposed to point at every read character, will simply skip the undesired characters – both the comment tokens and the text inbetween.

As for comments which end at a line break, a simple definition such as `("//")` suffices. This definition can also be written in the form `("//" . NIL)`, and so it implies that, when an end-of-comment delimiter is not informed, then that delimiter must be a `#\Newline` character.

4 Rule composition

The grammar language fundamentals have been outlined. *Terminals*, *exact-matches* and *built-in rules* act as primitives, while other *compound rules* act as means of combination. At this point it is important to discuss the language's *means of abstraction*.

Abstraction implies the building of structures which would allow the creation of the *rules* themselves. Figure 1 already hints at what is possible to make of the grammar language. For a better understanding, we should take a simpler example, as described in Figure 4.

```
(:var-decl  (:type
             :identifier
             (:maybe (:symbol "=")
                     :integer-constant)
             (:symbol ";")))

(:type      ((:or (:keyword "int")
                  (:keyword "char")
                  (:keyword "bool"))))

(:var-decls ((:many :var-decl)))
```

Figure 4: Rule example for matching a variable declaration.

The example outlined in Figure 4 is by no means practical, since there are more sophisticated ways of matching a variable declaration, but it should be enough for a brief explanation. Additionally, the obligatory `:SYMBOL` and `:KEYWORD` rules were omitted, as they are potentially implied by context.

Suppose that the compiler reads a file which may contain many variable declarations, as exemplified in Figure 5. We begin by attempting to match the rule `:VAR-DECLS`.

```
int   foo        = 5;
char  downcase_a = 97;
bool  false_val  = 0;
float this_fails = 3;
```

Figure 5: Example of a potential input file for the matching engine.

The file will fail a match for a `:VAR-DECL` in line 4, since the `:TYPE` rule will not match the keyword `"float"`. It will, however, not fail the entirety of the match process, giving the results of the first three lines. This happens due to the `:MANY` quantifier.

Since this quantifier is enclosed in a *precise match* context, in the definition of `:VAR-DECLS`, if no `:VAR-DECL` were matched, the whole match process would fail, and raise a condition in the matching engine.

For the example given at Figure 5, the match process would not fail with a syntax error. This is an undesided grammar runtime bug. The programmer could mitigate this problem with a *terminal* after the quantified variable declarations.

5 Conclusion

Grammar engines are certainly of great interest when designing a language, and so they often shape the way languages work to ensure that the syntax analysis of said language can be done by such a engine. They also yield seemingly simple structures when implemented, which guarantees easier debugging.

The time I spent building the matching engine was satisfactory, and ended up producing the discussed grammar language due to the way my compiler's syntax analysis was designed. The language was first though on paper, then carefully modified so that a comprehensive matcher could operate on it.

The matching engine itself is comprised of two relevant parts: a tokenizer and a matcher, which were not discussed in this article. But they were implemented as layers of software, such that the tokenizer was a foundation for matching the built-in rules and specific text strings; the matcher was the middle layer which could interpret the grammar language, and then the language itself comes on top.

This structure ended up being very interesting, because the software itself is now robust, and not necessarily specific to the Jack programming language. Further exploration can be made to make it work with other languages, and maybe even to analyze itself. I intend to soon attempt to build a Lisp dialect by just swapping the rule set of this matching engine. Performing this new experiment would lay the foundation to one of my future projects.

References

[1] H. Abelson, G. J. Sussman, and J. Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, 1996.

[2] N. Nisan and S. Schocken. *The Elements of Computing Systems*. MIT Press, 2008.

[3] N. J. Vieira. *Introdução aos Fundamentos da Computação*. Cengage Learning.