Dynamic Language Syntax Keyword Customization Using a KSL

Abubakar Nur Khalil

April 2018

Abstract

Programming languages have traditionally been designed with a fixed set of rules that guide two things, syntax and semantics. As with all languages, whether intended for computers or humans, there is a set of predefined rules that guide the validity of grammatical structure and the meanings derived thereof.

Historically, at least to this point, changing rules such as the valid syntax keywords of a programming language required obtaining and tinkering with its source code, however, this activity remains a niche one carried out by individuals with the know-how.

For some programmers the aforementioned language restrictions sometimes affect their productivity, as it requires learning different rules for each language, though most languages come from a family of shared rules, e.g. the C family of programming languages [8], nevertheless, even if the programming language's interpreter or compiler was edited, it would require the programmer to (re-)build the language code or install it on each new system, making it difficult to port.

In this paper we introduce a novel approach to adding built-in support for customizing a language's syntax keywords, through the elaborate use of a data structure we have termed the *Keyword Syntax List* (or KSL for short), to extend both dynamically-typed and statically-typed languages.

Background

Programming language design and compiler theory have historically been niche fields that garner the attention of only a select few programmers. A common practice for such curious persons is to try and implement some programming language, whether it be a Domain Specific Language, an industry-standard language like Python [7] or a language they have designed themselves. A natural result from this tinkering is the tendency to change core aspects of the language's behaviour, such as syntax keywords.

Let us assume that the aim of the programmer is to change the language's syntax keywords in order to make it more comfortable to write code, in such a case they would have to install their version of the language's source code on each new machine, or worse, might need to re-implement their solution each time. A simple yet relatable problem, and one which influenced our search for a generic solution.

Introduction

Suppose hypothetically that the syntax keyword for defining functions in the Go Programming Language [1] was strangely *"baz"*, in order to define some function '*sum*' which takes in two '*integers*' and returns their sum, one would need to write the following:

```
baz sum(x, y) int {
  return x + y;
}
```

As a user (minimalist or otherwise) of such a language, one would not find this keyword '*baz*' intuitive. When such a situation is met, a typical user is faced with three (3) options:

- Get a copy of the language's compiler or interpreter source code and tweak the offending syntax.
- Accept the limitations.
- Or, select a different language to use all together.

The first option might be rather tedious unless one is up to the task. More generally, it is an option only if the source code of the language in question is, in fact, available, if not, the next few options become the only reality, at least until and unless one understands the language well enough to build it from scratch.

Nowadays we do tend to find ourselves in the habit of picking option three without a second thought, as the sheer number of programming languages are in the hundreds [9], however, there are some languages, e.g. Rust [4] & Nim [6], whose syntax keyword peculiarities are inconsequential

when compared to the compelling advantages they offer.

It seems this problem may have been overlooked by language designers, though admittedly for varying justified reasons that are beyond the scope of this paper. Consequently, we intend to propose a formal and non-destructive method for customizing a programming language's syntax keywords, by providing a data-structure which we shall call the '*Keyword Syntax List*' (*KSL*), to handle user-defined syntax keywords synthesis with the language's predefined keywords.

An existing implementation of the KSL can be found in the Rocket Programming Language [2], our toy programming language, and is available on Github [2] for those who want to experiment with or possibly extend the KSL.

1 Structure of the KSL

Note: *code snippets are all written in Python.*

The KSL itself is a single list containing two (2) Dictionaries in the Python sense or *Associative Arrays* in the Computer Science sense, we shall herein refer to them as the *word-to-keyword* dictionary (**wk_Dict**) and the *value-to-word* dictionary (**vw_Dict**).

The former, as the name implies, contains mappings of *plain words*, which we shall define shortly, to the abstract token representation of the language's syntax keywords. Assuming the language's token representation of the default keywords is in the form *"< TokenType {keyword} {enum value} >"*, then our **wk_Dict** can be defined as follows:

```
{
  "LOG" : <TokenType PRINT 0>,
  "DEF" : <TokenType FUNC 1>,
  "LET" : <TokenType VAR 2>,
  ...
}
```

Figure 1: Sample *word-to-keyword* dictionary (**wk_Dict**).

The latter is populated with mappings of unique values associated with each keyword token in the language to *plain words* defined in the **wk_Dict**:

```
{
  0 : "LOG", 1 : "DEF", 2: "LET", ...
}
```

Figure 2: Sample *value-to-word* dictionary (**vw_Dict**).

The KSL is intended to serve as an internal memory bank for the language's interpreter or compiler, to store the mappings of the new user-defined keywords to the default keyword tokens (i.e. **wk_Dict**) for the language's scanner as it builds the language tokens, and mappings of these keyword token values to the custom keywords (i.e. **vw_Dict**) for the parser and interpreter to use in semantic related evaluation and error reporting.

In short, the KSL can be thought of as a decoupled lookup table for a programming language's syntax keywords, which guides in keyword literal scanning, parsing, code interpreting, and error reporting.

2 Using the KSL in Source code

2.1 Scanning/Lexing

Let us consider some subset of the BASIC [3] programming language with just four (4) syntax keywords:

- PRINT
- IF
- VAR

- GOTO

We can then allow the user to write a small config file; this file should at the very least contain a list of user-defined keywords on the left and a list of the language's corresponding default keywords on the right like so:

```
bake    print
when    if
define  var
do      goto
```

Figure 3: Sample content of our KSL config file.

It is important to note that the method of receiving the user-defined keywords is subject to the discretion of the implementer, but we digress. Ideally, the language's scanner would flag scan errors when custom words are encountered, therefore, in order to allow these newly defined keywords to be recognized properly, it would need to interface with our KSL's **wk_Dict**. Specifically, we must first parse the config file to create mappings of the *plain words* their corresponding *TokenType*, which thhe scanner can then use this as its reference table for tokenizing the keywords.

Now, let's assume that our hypothetical user does not provide a config file and simply wants to use the language's defaults, would that mean our KSL would be unused? Not entirely, we would still use the KSL, however, the only difference would be that **wk_Dict** and **vw_Dict** would be filled with mappings of the language's default keywords in *plain words* to their corresponding *TokenTypes*, and mappings of these default keywords' *TokenType* values to their plain word representations respectfully.

Proceeding with our example language, the contents of our representation of the language's syntax keywords' *TokenType*, along with our **wk_Dict** and **vw_Dict** (similar to the content in **Figs**. 1 & 2) can be found below:

```
import enum as _enum

# C-like enum type
@_enum.unique
class TokenType(_enum.Enum):
  # Default keywords
  PRINT       = 0  # 'print'
  IF          = 1  # 'if'
  VAR         = 2  # 'var'
  GOTO        = 3  # 'goto'

kw_Dict = {
  "BAKE": TokenType.PRINT,
  "WHEN": TokenType.IF,
  "DEFINE": TokenType.VAR,
  "DO": TokenType.GOTO
}

vw_Dict = {
  0: "BAKE",
  1: "WHEN",
  2: "DEFINE",
  3: "DO"
}

KSL = [kw_Dict, vw_Dict]
```

Figure 4: Sample code for our language's TokenType and KSL.

As discussed, the KSL can still be used throughout the language's different pass levels even in the absence of any user-defined keywords. This would allow us to easily transition from customs to defaults without adding additional burden to the internals of the language. Another interesting edge case would be as follows: suppose the custom keywords provided by our user does not equal the total number of

default keywords in the language, what do we do then? The answer at this point should seem very obvious, that is, we simply fill the remainder of the unchanged keywords with the language's defaults.

2.2 Using the KSL for Error reporting

Now that we are through tokenizing we would have to consider error reporting. Our user would be confused if she was presented with the following error message even after providing the custom keyword "*BAKE*" to replace "*PRINT*":

```
ParseError: 'PRINT' expected ';' after expression.
```

To avoid this inconsistency we need to also ensure error reporting is handled with the appropriate current valid keyword, whether user-defined or default. This is where our **vw_Dict** comes in handy. We simply look up the value of the language's keyword *TokenType* that is reporting the error in our **vw_Dict**, which would, in turn, return the appropriate plain word representation of that keyword, thus providing the perfect illusion to the user that their custom keywords are truly integrated while maintaining a uniform internal structure in the language's code. This method could be adopted for all errors to be reported no matter the pass level.

To further shed light on this, let us provide a contrived example of some error reporting code in a language's parser:

```
def printStatement():
  # Get 'PRINT' plain word from KSL
  print_lexeme = KSL[1][TokenType.PRINT.value]

  # ... more parsing code

  # Assuming semicolon (';') is a token.
  consume(TokenType.SEMICOL, f"'{print_lexeme}'
                expected ';' after expression")
```

Figure 5: Sample Parse Error code using a KSL.

We can therefore properly display the customized keyword's new name in our error reporting, and hence provides us with the consistency we need.

Error reporting is just as easy as that, we simply index the **vw_Dict** with the current keyword's value being parsed, interpreted, or even compiled and we will have our plain word representation ready to be used.

Recommendations & Limitations

Perhaps in the future, we may be able to extend the KSL to support user-defined syntax operator symbols, i.e changing the binary operator for addition from '+' to '@' for instance.

Another interesting use-case would be support for custom syntax comment symbol definitions, i.e changing the single-line comment symbol from '#' to '%', or multi-line comments from '/*' & '*/' to '"""' & '"""'.

The following questions should also be considered going forward if we indeed intend to standardize this solution, or at the very least improve it:

- Explore other methods for accepting these user-defined customizations, e.g. in a config file as discussed or possibly a JSON file, in a REPL session, or at the beginning of a source file.
- Optimal methods for quickly scanning and parsing the KSL after a customization method has been chosen.
- Performance optimizations for, and potential trade-offs of including the KSL in a programming language.
- Pitfalls in our current design of the KSL.
- Other potential approaches towards providing built-in support for customizing language syntax keywords besides using a KSL.
- Potential code linters for KSL supported languages such as Rocket [2].

Conclusion

We barely scratched the surface in terms of what can potentially be achieved by simply organizing user-defined custom keywords using a KSL. There is still a lot to discuss regarding what we have put forward, nevertheless, we have achieved what we set out to do initially: to propose a formal and non-destructive method for adding built-in support for customizing a programming language's syntax keywords by using a data-structure we have termed the KSL.

This elaborate integration of a KSL is intended to give programmers more control and freedom over the syntactic restrictions of a language, such as its syntax keywords.

The use of a KSL, as can be inferred, solves the portability problem previously discussed, as it provides a platform-agnostic built-in syntax customization extension.

Acknowledgements

To Robert Nystrom, for writing the *craftinginterpreters* [5] Book, without which this work would not have been possible.

Glossary

- **KSL** - The data structure which holds two separate but related lookup tables with mappings of user-defined syntax keywords to the default syntax keywords, and vice-versa.
- **Plain word** - Any group of ASCII characters that represent a word, i.e *"0x68 0x65 0x6c 0x6c 0x6f"* → *"hello"*.
- **Pass level** - The stages that take place before a given "source code" is compiled or interpreted.

References

[1] Ken Thompson et al (The Go Authors). Go (programming language). https://wikipedia.org/wiki/Go_(programming_language), 2020.

[2] Abubakar Nur Khalil. Rocket (programming language). https://github.com/Zero-1729/rocket, 2020.

[3] John G. Kemeny & Thomas E. Kurtz. Basic (programming language). https://wikipedia.org/wiki/BASIC, 2020.

[4] Graydon Hoare (Mozilla). Rust (programming language). https://wikipedia.org/wiki/Rust_(programming_language), 2020.

[5] Robert Nystrom. Crafting interpreters. https://craftinginterpreters.com/, 2020.

[6] Andreas Rumpf. Nim (programming language). https://wikipedia.org/wiki/Nim_(programming_language), 2020.

[7] Guido van Rossum. Python (programming language). https://wikipedia.org/wiki/Python_(programming_language), 2020.

[8] Wikipedia. C-family programming languages. https://wikipedia.org/wiki/List_of_C-family_programming_languages, 2020.

[9] Wikipedia. List of programming languages. https://wikipedia.org/wiki/List_of_programming_languages, 2020.