Yaccscript: A Platform for Intersecting High-Level Languages

John Healey
Emory University
Suite W401
400 Dowman Drive
Atlanta, Georgia 30322

jpheale@emory.edu

ABSTRACT

Programming paradigms are often skewed towards a particular domain of problems, thus one effective way to utilize them is through a multiparadigm approach to software development. One way to achieve this goal is to compile multiple languages to a single platform that can support a variety of processing models. This paper describes Yaccscript, an extensible platform for compiling languages to a common framework, and demonstrates its effectiveness at providing interoperability between object-oriented, functional, and logic programming through implementations of Python, Haskell, and Prolog.

Categories and Subject Descriptors

D.3.2 [**Programming Languages**]: Language Classifications—*Extensible Languages, Multiparadigm Languages*;
D.3.4 [**Programming Languages**]: Processors—*Interpreters, Translator writing systems and compiler generators*

General Terms

Languages

Keywords

Multiparadigm Programming, Lisp

1. INTRODUCTION

The study of programming paradigms has lead to contrasting opinions. Often, there are disagreements on which paradigm is superior to the others or how best to express a particular paradigm. These opinions have lead language designers to construct a multitude of languages with widely varying syntax, data, and processing models. Although these languages can all express the same algorithms,[1] they are often somewhat domain-limited, solving a certain problems more easily than others.

Since the domain-specificity of a language is often closely tied to its underlying paradigm, several language designers have suggested constructing languages to support multiple paradigms. One approach to this task is to create a single all-encompassing syntax that is capable of supporting the different paradigms. Another approach, which is used by Yaccscript, is the creation of a common language framework that allows multiple languages to interoperate, in their native syntax, using a common platform.

The specific approach used for Yaccscript is to combine two existing ideas from the field of computer science. As the name implies, Yaccscript is the fusion of a compiler-compiler[2] and a scripting engine. Thus, it supports multiple languages by compiling compilers for those languages dynamically. To make the system even simpler, the target platform for these compilers is Common Lisp with an API to ease interoperability between languages.

2. PRIOR WORK

2.1 Leda

Leda is a language developed at Oregon State University to explore the unification of existing paradigms into a single syntax. It supports imperative, object-oriented, functional and logic programming [3]. Leda does succeed in supporting these paradigms with a concise syntax. A shortcoming of this approach is that its single syntax is not extensible, so it potentially lacks the ability to incorporate new paradigms as they are discovered.

2.2 .NET

.NET is a framework for implementing multiple languages developed by Microsoft. The platform supports multiple languages by compiling them to a Common Language Runtime, which offers a unified type system as well as a byte-code interpreter. The task of porting any given language to the .NET platform involves porting or writing a compiler which uses the Microsoft Intermediary Language (MSIL) as the target [9].

[1]Except for languages that are not Turing complete.
[2]Yaccscript is not actually compatible with YACC, but the name still seems to fit as it is Yet Another Compiler-Compiler.

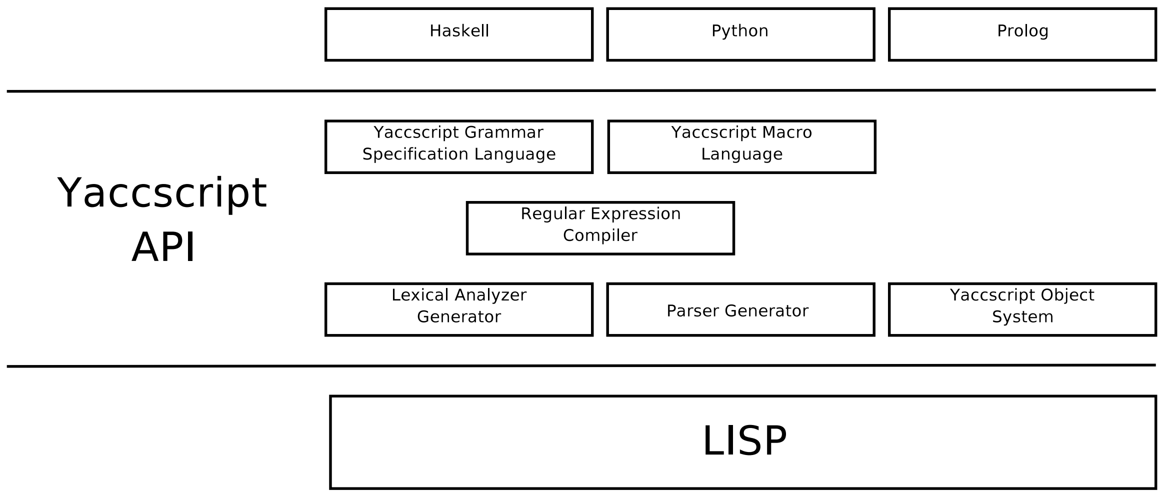
Haskell
Python
Prolog
Yaccscript API
Yaccscript Grammar Specification Language
Yaccscript Macro Language
Regular Expression Compiler
Lexical Analyzer Generator
Parser Generator
Yaccscript Object System
LISP

Figure 1: An Overview of the Yaccscript System

All of the languages that were implemented on top of Yaccscript have also been implemented for the .NET platform. By using a sequence of compilers, Haskell code can be directly compiled to .NET byte code [10]. The .NET platform also hosts *IronPython*, an implementation of the Python language which actually out-performs the standard implementation of Python [6]. Although both of these exist, there have not been any published studies on how the two languages can be used together on the .NET platform.

2.3 Multiparadigm Programming In Lisp

Lisp has also been known for its ability to support multiparadigm programming. It has sometimes been referred to as a language without syntax, as Lisp code takes the form of an abstract syntax tree. Because Lisp code is constructed out of Lisp data, it can be easily restructured through the use of macros, allowing programmers to build abstractions in the way that code is processed. Although its roots are functional, Common Lisp has constructs to enable imperative and object oriented [1] programming. Lisp has also been shown to be flexible enough to support non-deterministic logic programming [11]. It is because of Lisp's flexibility in paradigm expression that it was chosen to serve as the underlying execution engine for Yaccscript.

Lisp also supports syntactic extensions through dispatch macro characters. These functions are shortcuts that correspond to Lisp macros. Some examples of these used in Common Lisp are the syntax for creating vectors and the hash-quote used to access functions. Although they do serve to expand the syntactic potential of Lisp, they are still based on prefix notation. It should also be noted, that the original specification for Lisp referred to meta expressions (M-expressions) [8], which were an alternate notation for representing the more common symbolic expressions (S-expressions). This syntax never caught on and has not been supported since very early Lisp implementations. Although Yaccscript seems to revive the idea of M-expressions, it does so with the added benefit of making the alternate syntax extensible.

3. OVERVIEW OF THE SYSTEM

There are several components to the Yaccscript system, as depicted by **Figure 1**. The underlying execution engine is GNU Clisp [5]. On top of that is the Yaccscript API, which provides a lexer and parser generator and the Yaccscript Object System. The lexer and parser generators were used to create the Yaccscript Grammar Specification Language, a syntax for the creation of lexical analyzers and parsers. There is also a macro language which is used to abstract and shorten code written for the lexer and parser generators. Finally, there is the Yaccscript executable, a program that can be run in two different modes. The first mode can be given a program written in a language that Yaccscript can compile. The second mode allows a read-eval-print-loop (REPL) to be executed for any given Yaccscript language.

3.1 The Lexer and Parser Generators

The lexer generator compiles a Lisp list into a block of code that can be used to process regular expressions[3]. It can be used to process a single regular expression, or more often, be used to tokenize a string of text based on a series of regular expressions. When used as a tokenizer, it returns the longest matching string, and in the event of two tokens of the same length, it chooses whichever regular expression the token matched first.

The parser generator compiles Lisp lists into a backtracking recursive descent parser. For each expression in the parser, it matches the first series of tokens that conform to the expression and returns a string containing Lisp code to be parsed and compiled. It also supports the ability to define simple macros to abstract and simplify the return strings.

Since both the lexer and parser generator compile Lisp data, they can be used to compile data that they can then process. This allowed the creation of a simple regular expression compiler that takes a string to be compiled to Lisp data for processing by the lexer generator. The lexer generator, parser, and regular expression compiler were combined to create the Yaccscript Grammar Specification Language.

[3]The rules for matching are consistent with Lex.

```
LANGUAGE yslang.cl
IMPORT: "stdmacros.ys"

LEXER:

number : "([0-9]+)"
symbol : "([a-z]+)"
assign : "\x3d"
IGNORE : "(\x20|\x09|\x0a)"

PARSER:

REPL -> {expression} : 0;

expression -> {assignment}  : 0
           |  symbol        : 0
           |  number        : 0
           ;

assignment -> symbol assign
              {expression}  : setvar(0 2);
```

Figure 2: A Simple Language Definition

3.2 The Yaccscript Object System

The Yaccscript Object System provides data consistency between different languages that are to be compiled to Lisp. All data used in Yaccscript programs is represented as an object. This ensures that even the most fundamental data types are the same. It also allows for every piece of data to be abstracted into different types. Each class has methods associated with it for general use as well as specific methods to facilitate operator overloading.

There are a handful of data types that are built into the Yaccscript. The most basic objects are ints, floats, symbols, and boolean values. Using those data types in conjunction with data types built into Common Lisp, support for lists, tuples, and hash tables was integrated into the system. There is also a string type which is a subtype of the list type. Finally, there is a data type for functions, which are first-class objects in Yaccscript.

There are functions and macros associated with the Yaccscript Object System. Aside from the functions used to define data types and instantiate them, there are functions that access the fields and methods of the data types. There are also macros that are used to facilitate iteration through the abstract data types or perform calculations on them. For instance, there is a common iteration macro that can be used to iterate through lists, strings, or even hashes, which it iterates through as it would a list of tuples.

3.3 The Yaccscript Language

The Yaccscript Grammar Specification Language, also referred to simply as the Yaccscript Language, is a custom language for mapping syntactic rules to Lisp code. Files constructed with the language have two parts. The first part uses a series of regular expressions to define the accepted (as well as ignored) tokens. The second part defines the grammar rules. Each rule has one or more mappings that matches a series of tokens (terminals) and rules (non-terminals) to the Lisp code that it evaluates to.

Figure 2 demonstrates a simple language that supports symbols, numbers, and an assignment operator and can be utilized through a REPL. The `LANGUAGE` line specifies the file that describes the language used to compile this file. In this case, it is "yslang.cl", the file that describes the compilation rules for the Yaccscript Grammar Specification Language. The `LEXER` section describes 3 tokens as well as a set of whitespace characters that should be ignored. The `PARSER` section has three rules: "REPL", "expression", and "assignment". Each consists of one or more syntactic patterns composed of tokens, referred to by name, and other rules, denoted by their name surrounded by "{" and "}". Each of these syntactic patterns has a corresponding compilation rule, which specifies the code that is to be generated when the pattern is matched. The numbers in the compilation rules correspond to the tokens and rules matched on the left side of the rule.

The first rule specifies that this language is to be accessed via the REPL interface.[4] The second is used to compile expressions that consist of an assignment, a symbol, or a number. Compilation rules consisting solely of the number 0 specify that the tokens and rules should be emitted as they are processed. Finally, the third expression compiles assignment expressions. The compilation rule used here is slightly more complicated. When the assignment rule sees a symbol, an assignment operator, and an expression, it compiles them using the `setvar` macro described in "`stdmacros.ys`" taking the symbol (0) and the expression (2) as arguments. If this language were to be given the expression "`a = b = 10`" to compile, it would compile it to the Lisp code "`(setf a (setf b 10))`".

As mentioned above, the Yaccscript Language was defined using the lexer and parser generators. There are two practical results of this design. First, it simplified the creation of the language. It also broadened the potential use of the system; the Yaccscript language can be used to parse structured data into Lisp data for processing by Yaccscript or Common Lisp programs. The final benefit of this design is that the syntax for the Yaccscript language is extensible to the point of being completely replaceable. Thus, the syntax for defining lexical analyzers and parsers can be modified or rewritten using Lisp or the Yaccscript Language itself.

Table 1: Language Implementation Size (Lines)

Language	Lisp	Yaccscript	Total
Python	78	418	496
Haskell	252	326	578
Prolog	360	107	467

4. LANGUAGES CONSTRUCTED WITH YACCSCRIPT

To demonstrate Yaccscript's potential for supporting and integrating different programming paradigms, portions of three existing languages were implemented. The languages were chosen to represent the three common paradigms, object-oriented, functional, and logic programming. Although the implementations are incomplete, they do represent large subsets of the corresponding languages. **Table 1** lists the amount

[4]Another rule could be created to add support for entire files written in this language, but that would most likely be coupled with a rule to compile a series of expressions.

Figure 3: Utilizing Haskell Code in Python

Euclid's GCD Implemented in Haskell

```
euclid _ 1 = 1
euclid m n =
    if m % n == 0
        then n
        else euclid n (m % n)
```

The Python REPL

```
Python> euclid(5,25)
5
Python> euclid(14)(42)
14
Python> e12 = euclid(12)
FUNCTION: CURRIED-EUCLID
Python> e12(66)
6
Python> e12(200)
4
```

of code (Lisp and Yaccscript) that was needed to implement each of these language subsets. Its worth noting that Python, which has probably the most verbose syntax of the three languages, required the most Yaccscript code, but the least Lisp. On the other end of the scale is Prolog, which has a very minimalist syntax, but a computational model that required a large amount of Lisp to accommodate.

4.1 Python

Python was chosen to demonstrate Yaccscript's ability to represent object-oriented programming languages. Python was a natural choice as it uses objects to represent all data and has syntax that works by operator overloading. It also contains powerful facilities for optional function arguments, and the ability to unpack tuples into variables[12]. The Python implementation is also probably the most complete of the three languages, since it naturally mapped onto the Lisp processing model and Yaccscript data types.

4.2 Haskell

Haskell is a functional language that allows for the easy creation of purely functional code. A particularly interesting feature of Haskell — and of many modern functional languages — is currying, the ability to create new functions by filling in some of the parameters of an existing function [7]. This was implemented in Yaccscript by overloading the existing function class to return a new function when called with less parameters than are expected. By enabling currying as a property of functions designed in Haskell, the ability remains with those functions when they are utilized in different languages. Although this implementation captures almost all of Haskell's syntax, it lacks support for lazy evaluation.

4.3 Prolog

Prolog is a widely known logic programming language — a declarative programming paradigm based, in principle, on non-deterministic evaluation of logical rules. Prolog allows a programmer to define a series of predicates which can be evaluated in two different ways. If a predicate is evaluated with literal values, it will return either true or false. However, a predicate can also be passed variables, in which case it will return the possible values of those variables[2]. This was implemented in Yaccscript by having Prolog predicates return true, false, or a list of hash tables that map variables to their values.

5. INTERACTION BETWEEN LANGUAGES IN YACCSCRIPT

Code in Yaccscript is divided up based on packages. The different languages ultimately intersect through functions. Thus, the ability for a language and its underlying paradigm to be integrated into Yaccscript is based on the ability for the functions of a language to be called by different languages.

5.1 Python and Haskell

Despite their apparent differences, Python and Haskell work quite well together in Yaccscript. A function written in Haskell can be easily accessed in Python and even retains its currying abilities. A good example of a function that is easily implemented in Haskell is Euclid's GCD algorithm, demonstrated in **Figure 3**. When imported into a Python REPL, the function becomes available. Since the function was implemented in Haskell, it still supports currying. The function `e12`, is defined much like it could be in Haskell, simply by passing a single value to the `euclid` function.

Python functions can be imported into Haskell, as can objects that implement the `__call__` method. A simple class that demonstrates this is the `Accumulator`, which is defined in **Figure 4**. The accumulator is created with an initial value and can then be called like a function, taking a single number as an argument and adding that to the initial value. Although it lacks the ability to be curried, it can still be used with Haskell's built-in function composition abilities. When composed with a square-function, a new function that squares the argument before accumulating it is created.

5.2 Python and Prolog

The Yaccscript implementation of Prolog treats predicates in a way that allows them to be used as functions. If passed literal values, the Prolog predicate will return a boolean value. However, it can also be passed symbols, in which case it will return a list of hashes that can easily be accessed. **Figure 5** demonstrates a simple implementation of factorial in Prolog. When imported into a Python interpreter, the factorial predicate can be accessed two different ways. It can be given two literal values, in which case it will return a boolean value. If given a symbol[5] and a literal value, it returns a list of hashes that match the symbol to the matching values.

Incorporating a Python function into Prolog is somewhat less trivial since general function calls are not easily supported by Prolog's computation model that treats functions

[5]Symbols in Python are not syntactically available, but can be easily synthesized.

Figure 4: Utilizing the Python Accumulator in Haskell

An Accumulator in Python

```
class Accumulator:
    def __init__(self,val):
        self.val = val
    def __call__(self,inc):
        self.val = self.val + inc
        return self.val
```

The Haskell REPL

```
Haskell> acc = Accumulator 100
#<OBJECT #x207EDD4E>
Haskell> acc 10
110
Haskell> acc 20
130
Haskell> square x = x * x
FUNCTION: SQUARE
Haskell> accsq = acc . square
FUNCTION: COMPOSED-ACCUMULATOR-SQUARE
Haskell> accsq 10
230
```

Figure 5: Utilizing the Prolog Factorial in Python

Factorial in Prolog

```
factorial(X,Y) :- Z is X - 1,
                  X > 0,
                  factorial(Z,A),
                  Y is A * X.
factorial(X,Y) :- X < 1,
                  Y is 1.
```

The Python REPL

```
Python> factorial(5,120)
True
Python> symx = sym("X")
X
Python> fac = factorial(5,symx)
[{X:120,},]
Python> fac[0][symx]
120
```

Table 2: Breakdown of Pancake Flipping

Language	Number of Lines
Python	130
Haskell	31
Prolog	46
Total	207

as uninterpreted symbols. Thus, the task of incorporating Python functions into Prolog usually requires a customized function or function wrapper that takes symbols and literals as arguments and returns a boolean value or list of hashes.

5.3 Haskell and Prolog

When incorporating drastically different languages, certain paradigmic inconsistencies are not easily resolved. Providing interaction between Haskell and Prolog proved to be challenging, since Haskell lacks built-in support for Hash tables. However, for this particular application, the hash tables are only needed for iteration, which can be synthesized using a list of tuples. So for the purpose of interoperability, both the Haskell and Prolog implementations were designed with the ability to mix hashes with lists of tuples for this purpose.

6. YACCSCRIPT IN ACTION: PANCAKE FLIPPING

An example of an algorithm that can take advantage of multiple-paradigm programming is the Gates-Papadimitriou algorithm for sorting by prefix reversal [4]. The problem, estimating the number of prefix reversals needed to sort a list of integers, is also known as the "pancake-flipping" problem, because of a real-life metaphor of sorting a stack of pancakes from largest to smallest.

This particular algorithm was chosen because it can very clearly be broken down into imperative, functional, and logic programming tasks. The algorithm uses Prolog to decide which state the stack of pancakes is in, which determines what series of flips should be applied. It uses Haskell to compose and perform the actual flips. Python is used to optimize certain operations that Prolog could only easily perform with copious, unwieldy recursion, as well as iterate through the main loop of the program.

The breakdown of the program into the individual languages is displayed in **Table 2**. The bulk of the code was written in Python, since it was used both for simple functions that were utilized by the Prolog code as well as in the main loop. However, the Haskell and Prolog files are also much shorter because they are exceptionally well suited for their respective portions of the application. All of the flip sequences in the Haskell code were single-line functions. The Prolog code wasn't much longer, as predicates defining the different "states" of the pancake stack were defined in 2-4 lines of code each. The 207 lines of code in the three languages compiled to 590 lines of Common Lisp. As would be expected, the code generated by the compiler is not as readable or concise as a hand-written code. For purpose of comparison, it should be noted that an earlier implementation of this algorithm in Common Lisp by the same author took 280 lines of code.

7. DISCUSSION

7.1 Potential Applications of Yaccscript

A possible application that would make use of the multiple paradigms would be a web-based genealogical database. Haskell's ability to process recursive data as well as its ability to curry and compose functions makes it ideal for the generation of html. Prolog is adept at processing the genealogical data, as relationships can be described in terms of each other, so that very little source data about each person is needed to describe an entire family tree. Python could be used as an iterative processing engine for the data that is stored on the hard drive. Although there isn't currently a library for parsing HTTP GET and POST data, it would be possible to build one using the Yaccscript Grammar Specification Language.

However, since Yaccscript is currently lacking the support libraries that would make it practical for large applications. At the moment it is mostly useful to language designers and researchers for creating prototype implementations of languages. Much of the functionality of an interpreter is already taken care of by Common Lisp, thus most of the work in creating a language in Yaccscript is in simulating the language's processing model in Lisp. However, there are several other features offered by Yaccscript to ease the creation of languages. For instance, grammar rules can be traced, so that their evaluation can be monitored for debugging. Also, for languages designed with interoperability in mind, testing frameworks can be constructed in a known working language to find errors.

7.2 Future Work

At this point, Yaccscript is still heavily research oriented and lacks many features that would be expected from a production-use scripting language. The error handling capabilities are inadequate and the platform itself would benefit from having additional libraries. Some of the more core features, such as sockets, would need to be implemented in Common Lisp to map existing functionality in Lisp to the Yaccscript Object System. However, as the hosted languages achieve more complete compatibility with their standardized counterparts, it becomes more feasible to port existing libraries from these languages to the system.

The particular language implementations could also use some work. In particular, the Haskell and Prolog implementations are lacking support for lazy evaluation. Rather than rewriting each implementation to utilize this processing model, it would be preferable to create a general framework for adding lazy evaluation to Yaccscript-implemented languages, especially since the feature seems common in declarative programming languages.

The final issue is portability. The system was designed using GNU Clisp [5] and as a result is dependent on some features unique to it. With a little bit of effort, Yaccscript could be ported to work with more Common Lisp implementations. Several implementations are capable of compiling to native code, which could provide a speed boost. Also, better cross-Lisp support would correspond to better cross-platform support.

8. CONCLUSION

Despite the multitude of languages that have been developed, it is rare to find one that is flexible enough to elegantly represent all of the common paradigms. Similarly, it is rare to find any one paradigm that succeeds in solving all problems with equal beauty and efficiency. Yaccscript's ability to host drastically different languages and facilitate interoperability makes it an interesting tool for researching the implications of multi-paradigm design on software engineering. The development and maintenance of larger applications in Yaccscript should provide some insight as to the scalability of multi-language design.

9. REFERENCES

[1] D. Bobrow, L. DeMichiel, and R. Gabriel. Common lisp object system specification. *SIGPLAN Notices*, 23, 1988.

[2] I. Bratko. *Prolog Programming for Artificial Intelligence*. Addison Wesley, 3 edition, 2001.

[3] T. Budd and R. Pandey. Never mind the paradigm, what about multiparadigm languages. *SIGCSE Bulletin*, 27(2):25–40, June 1995.

[4] W. H. Gates and C. H. Papadimitriou. Bounds for sorting by prefix reversal. *Discrete Mathematics*, (27):47–57, 1979.

[5] B. Haible and S. Steingold. Gnu clisp, 2005. Available At: http://clisp.cons.org/.

[6] J. Hugunin. Ironpython: A fast python implementation for .net and mono. In *PyCon*. Python Software Foundation, March 2004. Available at http://www.python.org/pycon/dc2004/papers/9/IronPython_PyCon2004.html.

[7] S. P. Jones and L. Augustsson. *Haskell 98 Language and Libraries, The Revised Report*, December 2002. Available at: http://haskell.org/definition.

[8] J. McCarthy. Recursive functions of symbolic expressions and their computation by machine. *Communications of the ACM*, 3(1):184–195, 1960.

[9] B. Meyer. The significance of dot-net. *Software Development*, 8(11), November 2000.

[10] L. O'Boyle. Making haskell .net compatible, 2002. Available at http://coscweb2.cosc.canterbury.ac.nz/research/reports/HonsReps/2002/hons_0206.pdf.

[11] J. Siskind and D. McAllester. Nondeterministic lisp as a substrate for constraint logic programming. In *AAAI-93*, pages 133–138, 1993.

[12] G. van Rossum. *Python Documentation, version 2.3.5*, February 2005. Available at: http://www.python.org/doc/2.3.5/.