\English

Connor Davey

09/18/2015

Contents

1 Introduction

In the 1940s in the newly emerging field of computer science, the idea of programming a generalized computer lead to the idea of a programming language. Since then, not only has the idea of programming languages become reality, but the idea of what a programming language is or should be has changed and developed and hundreds of different programming languages have been created based on these changing theories and convictions. \English considers a large number of the problems with programming language design and seeks to tackle some of the most relevant and important of them as best as possible:

- readability
- understandability
- extensibility
- efficiency (esp. runtime)

While all practical programming languages attempt to address all of these problems to some extent, more often than not, some of these important properties are sacrificed in order to improve some of the remaining properties. Generally, in more recent decades of computer science, runtime efficiency has been sacrificed more and more in favor of readability and understandability. As computers became more widely used and programming professions and projects became more commonplace, programming languages became easier to read, learn, and use. However, this often came at a great cost to efficiency. For example, Python is a high-level language often revered as one of the simplest to read and understand[**?**] with modern day usage in many areas, especially concerning web development (see Django as an example). However, Python sacrifices programming efficiency in order to achieve much of this readability[**?**][**?**][**?**].

\English seeks to unite extreme simplicity and readability with high orders of efficiency using everything from high-level high order functionality to low-level Assembly programming. The grammar of the language is intended to closely relate to ordinary English (hence the name of the language) while maintaining and extending a few of the most basic, widely used programming language syntactic structures. Meanwhile, the ability to write "safe" Assembly code integrated into a highly functional language allows for powerful extensibility and high potential for runtime efficiency.

2 Language Overview

Most unique about \English is the fact that its syntax is programmatically defined; the entire language is made up of operators declared with a context-free grammatical description of their syntaxes. The code is parsed as a context-dependent grammar using a dynamic set of rules defined by \English's most basic operator:

```
define <<type>> <declaration>:  <body>;
```

where "`<type>`" is the return type of the operator, "`<declaration>`" is the context-free grammatical description of the operator complete with typed arguments and punctuation, and "`<body>`" is the set of operations that the operator will perform.

\English is comprised completely of operators even more so than Haskell is comprised of functions. Every part of the language works on this paradigm, including code blocks like loops or conditional statements and the type system. This not only unifies the language and improves parsing simplicity, but also improves the versatility of the language by removing unnecessary restrictions on usage of the operators. Even variables are only a special type of operator that takes no arguments and stores its return value in memory instead of evaluating an expression each time it is called. Operators' true function comes from the "asm" operator, which allows the use of Assembly-level NASM commands to describe functions. This Assembly level integration works with the standard library to not only allow ease of use and powerful low-level operation, but also to ensure the restriction of potentially unsafe operations that may hinder the language's optimizability and safety, e.g. the use of arbitrary pointers.

Because types, too, are defined using operators, types are parameterizable not only with other types like Java type parameterization or C++ templates, but with any kind of term, allowing types to not only include such data as the type of items contained in a list, but also the range in which a number may lie or the number of elements in a list. This adds structure and readability to the code and improves error finding and debugging.

\English is also built for optimization. It is a pure compiled language like C and, like C, includes the ability to implement Assembly instructions. In addition, like Haskell, \English is built to accomodate functional optimizations such as referential integrity preprocessing referred to in \English as "finalization". This optimization theoretically allows the compiler to determine which operators preserve referential integrity (always return the same value given the same arguments) and run any process composed of only final operators at compile time using basic Assembly emulation.

Because \English has mutable variables unlike Haskell, this "finality" is categorized into two types: "return-finality", which means that the return value is consistent given consistent arguments, and "enviro-finality", which means that the operator does not modify the value of any variables outside of its own body. \English is built to be able to classify such instances of either of these at the Assembly level using the "safe" asm directive, allowing an optimizing compiler to potentially ignore all operations concerning the return value of an operator if that return value is not used or lazily evaluate operations on variables outside of the operator.

A great example of the use of this finalization would be in processing regex strings. \English allows the capture of literal characters in operator definitions, allowing programmers to create special types of srings; one such case in the standard library is a regex string that represents a regular expression. In languages like Java or C++, such functionality would be achieved by passing a plain string into a method or function to process the string's grammar into a regex object. That object represents the structure of the regular expression and can then be used for matches against strings. This system processes these regexes at runtime, which is not only less efficient than preprocessing at compile time, but creates the potential for runtime errors caused by misspellings or regex grammatical mistakes. In \English, regex string processing is final because it takes in only one argument: literal static code characters. This means that not only can \English run the code once at compile time instead of every time the operator is called at runtime, but it also means that any errors in the given regular expression are guaranteed to be thrown at compile time rather than possibly thrown at runtime whenever (if ever) that particular section of the code is run during software testing.

Finally, because of \English's dynamically defined completely malleable syntax, it can be defined to look and work exactly like other programming languages in a form that the \English compiler can understand. This means that by simply "turning off" \English standard operators selectively, the \English compiler can understand other programming languages in the same terms that it understands \English. Using a simple system built into the compiler, then, \English code can be seamlessly integrated with other libraries or projects written in other programming languages. Tool libraries like C++ GUI libraries or SQL query libraries can be made available without rewriting the entire library in \English; the code, whether compiled or interpreted normally, can be compiled along with \English; and all (or most) of the same optimizations used in the \English compiler can be applied to other languages.

In summary, \English attempts to provide solutions to the difficult problem of readability and the even more elusive problem of understandability without sacrificing optimizability, efficiency, and accuracy of the language. In fact, \English attempts to make strides in all of these areas: readability via a powerful syntax allowing inlaid words and punctuation; understandability with short, elegant statements and code bodies; optimizability via techniques such as finalization; efficiency via consolidation of Assembly level and highly functional coding paradigms; and accuracy via strong, static, limitlessly parameterizable typing. With the translational library system, this language also pursues the unification of languages and the reuse of existing libraries and tools for extension of the abilities of the language/project.

3 Project Goals

This project seeks to satisfy the following goals.

3.1 Concrete Goals

3.2 Abstract Goals

4 Tools

Resources to be used in building the project include, but is not limited to, the following open-source free to use tools.

4.1 Compiler Tools

4.1.1 The NASM Assembler

4.1.2 L.D.

4.2 Production Tools

4.2.1 G++

References

[1] "Pythons elegant, and not overly cryptic, syntax emphasizes support for common programming methodology and promotes code readability and thus maintainability." – Sanner, Michel F. *Python: a programming language for software integration and development.* J Mol Graph Model 17.1 (1999): 57-61.

[2] http://benchmarksgame.alioth.debian.org/u64q/python.html

[3] http://benchmarksgame.alioth.debian.org/u64q/compare.php?lang=python3&lang2=gcc

[4] "It is fair to say that the core of the Python programming language is not particularly suitable for scientific applications involving intensive computations. This is mainly due to slow execution of long nested loops and the lack of efficient array data structures." – Cai, Xing, Hans Petter Langtangen, and Halvard Moe. *On the performance of the Python programming language for serial and parallel scientific computations.* Scientific Programming 13.1 (2005): 31-56.