Expressions and Equations

exploring languages through application development

Narkizian, Jake hnarkizi@ucsc.edu

March 12, 2015

The goal of this project was to compare and contrast the development process of two imperative programming languages by implementing a program in both. Our program can evaluate mathematical expressions and solve for a single unknown in an equation. Through this process we achieved a deeper understanding of how imperative programming has changed over time, and how modern features affect programming.

Languages

The languages we compared were, ANSI C, and Apple's Swift. Despite minor differences between the implementations of our project (mostly due to languages-specific limitations or features) the basic data structures and algorithms used remain largely the same. This allowed us to compare C and Swift much more precisely.

It is worth noting that while both of us have worked extensively with C, neither of us had any decent experience in Swift prior to this project.

Swift

After the success of the iPhone and its SDK in 2008, almost all Objective-C code being written was for Apple's products[3]. Today Swift is no different—released just shy of 5 months ago on October 22, 2014[1], its aim is to replace Objective-C as the language

used for app development on the company's iOS and OS X platforms. The replacement is Apple's way of eliminating "the baggage of C". The languages itself is far more advanced than its predecessor. It includes features introduced in almost every other modern programming language, and features much cleaner syntax.

Why Swift?

A major strength of Apple's iOS platform is its large app offering. We believe that Apple has designed Swift with a broad feature set because it aims to attract more developers to its platform. Many of the features are borrowed from popular languages that most developers already know. For example, it includes type inference from functional programming languages such as Haskell, optional semicolons and cleaner syntax from scripting languages like Python, and closures from web development languages like JavaScript[2]. In many ways Swift is the culmination of the best programming techniques and paradigms of the past decade packed in one place.

Another important facet of the language is its focus on *safer* code[2]. Code safety is a common issue when programming in older imperative languages. Swift avoids issues such as memory access errors, null pointer exceptions, and garbage data with the use of *optionals*. This is For these reasons Swift is an excellent candidate to examine for our project.

ANSI C

While development on Swift began in 2010, Dennis Richie started work on The C Programming Language 41 years earlier in 1969[4]. In the past four decades C has become one of the most widely used programming languages in the world. C's dominance not only allowed it to be relevant today, but its impact and legacy have influenced many modern languages.

The influence of the C programming language can be seen in almost every modern imperative programming languages. Many languages, including Java, JavaScript, C++, Objective-C, and C# even borrow directly from C's syntax.

Why C?

After many decades of use, C has matured into a relatively lightweight general purpose programming language. Today it represents a perfected version of the ideas, philosophies, and practices of the last generations of programming. If Swift is the child of a modern times, then C is its elderly forefather. Its stark contrast with For this reason it is the perfect standard to compare to Swift.

Architecture

In this section we will first define the models in the Operations ADT, and then a general overview of the algorithms used. Finally an explanation of the driver programs we used.

Operations ADT

The majority of the code we wrote for both implementations of this project were in an ADT called *Operations*. The ADT defines two models and their associated methods. These models are the *Operation* and the *OpList*. An OpList can model a first degree polynomial or a simple mathematical expression by representing a collection of indexed Operations.

Operation

An Operation models a single element mathematical expression that, when *valid*, evaluates to a number. There are three types of Operations.

The most common form, the *Identity-Operation*, represents a real decimal or integer number. It evaluates exactly to that number, and is always considered valid.

The second most common form, the *Operator-Operation*, represents a standard arithmetic operation (e.g. ^,*, /, +, -). This Operation contains links to two valid Operations—these represent the left and right operands respectively (see figure 1). When evaluated the specified operator performs the operation using the numbers represented in its left and right operations as operands. This operation is considered valid if and only if it contains two valid operands.

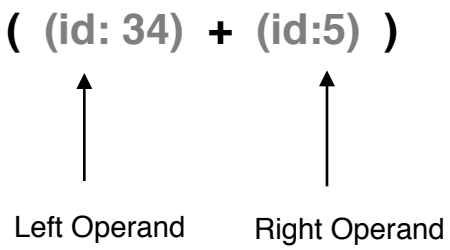

figure 1: Plus operation with two Id operations as operands.

An Operator-Operation can hold any kind of valid operation including other Operator-Operations. For example a subtraction Operation complete with two Id-Operations as operands can be the left operand for an addition operation.

The third Operation type is the *Variable-Operation*. This contains no value, and represents an unknown in an equation. In our ADT it functions as placeholder for our program to plug in a value as x. Evaluating this requires the passing of a rational real decimal or integer number to plug in, and the operation evaluates to the passed number. This Operation is always considered valid.

OpList

An OpList is a collection of operations ordered from left to right. If the OpList is modeling an equation it must contain another Id-Operation to serve as the equals value. If an OpList contains an equals value, it must also contain exactly one Variable-Operation.

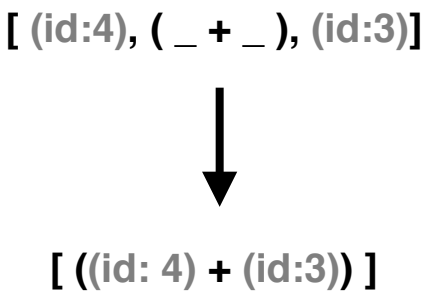

figure 2: Invalid OpList reduced to a valid OpList.

An OpList is considered valid if and only if it contains exactly one valid operation. Only valid OpLists

can be evaluated. The evaluation of a valid OpList is equal to the value of the evaluation of the OpList's only operation. In practice this single operation is typically a nesting of many different operations.

An invalid OpList can be *reduced* to a valid OpList by setting operands to the values immediately left and right of every invalid Operation(see figure 2) in the order of *precedence*. The operands are then removed from the list. This reduction process requires that each invalid operation is preceded by and followed by a valid operation prior to it becoming valid. Thus an invalid reducible OpList can only contain an alternating series of valid and invalid Operations, beginning and ending with a valid Operation. This implies that a reducible OpList can only have an odd number of operations, as there cannot exist any number of consecutively valid or invalid Operations (see figure 3).

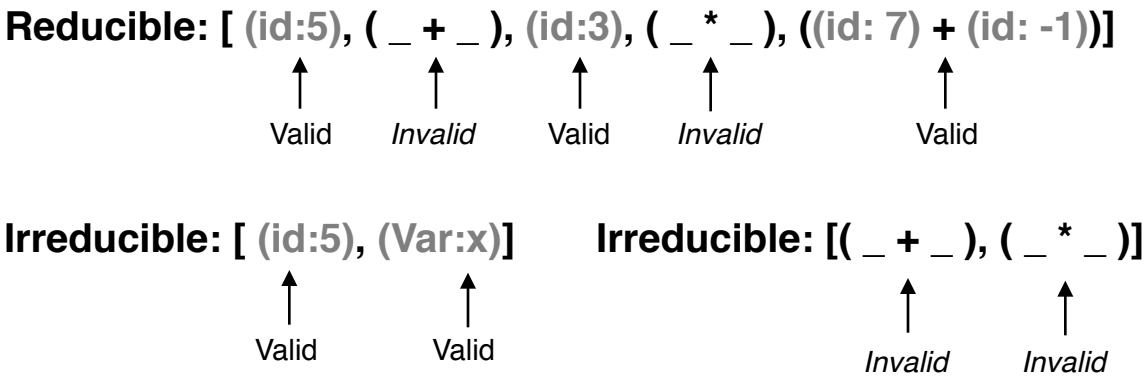

figure 3: Examples of reducible and irreducible OpLists

Precedence Ranking

The order by which Operator-Operations absorb their operands during reduction is determined by the precedence rank of each operator. This is a number that represents the importance or precedence of the operation. The precedence ranking of all operators is set in the scheme: { ^: 5 , *:4 , /:3 , +:2, -:1} by default. This default number is multiplied by the operations *parenthesis level* during reduction. This is the number of parentheses within which an Operation exists. The entire OpList is set to have a parenthesis level of one by default.

During reduction The highest ranked Operation has the valid Operations to its left and right set as operands first. This process then continues to the next highest ranked Operation until the OpList reduces to one Operation and becomes valid.

Evaluation Algorithms and Pseudo Code

Operation Evaluation

An Operation can be evaluated if it is valid. Id Operations return their associated number value. Variable Operations return the number passed to be plugged in for the variable. Operator Operations compute the value of their operands down to values, and then perform a predefined operation (corresponding to the operator) on those values. The result is returned.

```
// this pseudo code only shows addition for simplicity

Operation Op = some valid addition Operation
if Op is Id-Operation
      return Op.number

return (Op.left.evaluate()) + (Op.right.evaluate())
```

OpList Reduction

An invalid OpList is reduced by absorbing its left and right operands until it becomes valid.

```
OpList O = some reducible OpList

While O is invalid
      next_op = next highest ranked Operation in O
      next_op.leftOperand  = O[next_op.index - 1]
      next_op.rightOperand = O[next_op.index]

return O // see above algorithm
```

Expression Evaluation

To evaluate an expression the OpList is simply reduced, and its only function is then evaluated.

```
OpList O = some reducible OpList

O.reduce()

return O.evaluate() // see above algorithm
```

Equation Evaluation

To evaluate an equation for an unknown value the OpList must be made valid. Next we will find the x value by searching for it between the maximum x value and the minimum x value with binary search. It is worth noting that in both implementations of our project entering an equation with an x value above or below the max will result in an infinite loop. However the bounds are pretty broad and we do not anticipate this to be a problem for the purposes of this project.

```
OpList O = some OpList

if O.valid is false
     make O valid  // see algorithm above

try_value = (upper bound + lower bound) / 2

While O.evaluate(try_value) != O.equals_value
     if try_value was too small
          try_value = (upper b + try_value) / 2
     else
          try_value = (lower b + try_value) / 2

return try_value
```

Parsing Algorithm

This algorithm parses a String of mathematical operations into an OpList

Valid Strings

A string is valid if it contains an odd series of numbers (variables are considered numbers) and basic arithmetic operators. A string can also contain parentheses, however for every opening parenthesis, there must be a closing parenthesis. The main loop uses truth values to keep track of the last Operation appended to the OpList. This allows for error checking.

Driver Program

The Driver Program takes a String from the user, parses it into an OpList and (depending on some mechanism in the driver) solves for x or evaluates the OpList.

Swift Implementation

This implementation utilizes a simple OS X program to input strings and output values or errors.

C Implementation

This implementation is a command line tool with an option for entering an equation. All errors and results are printed to the command line.

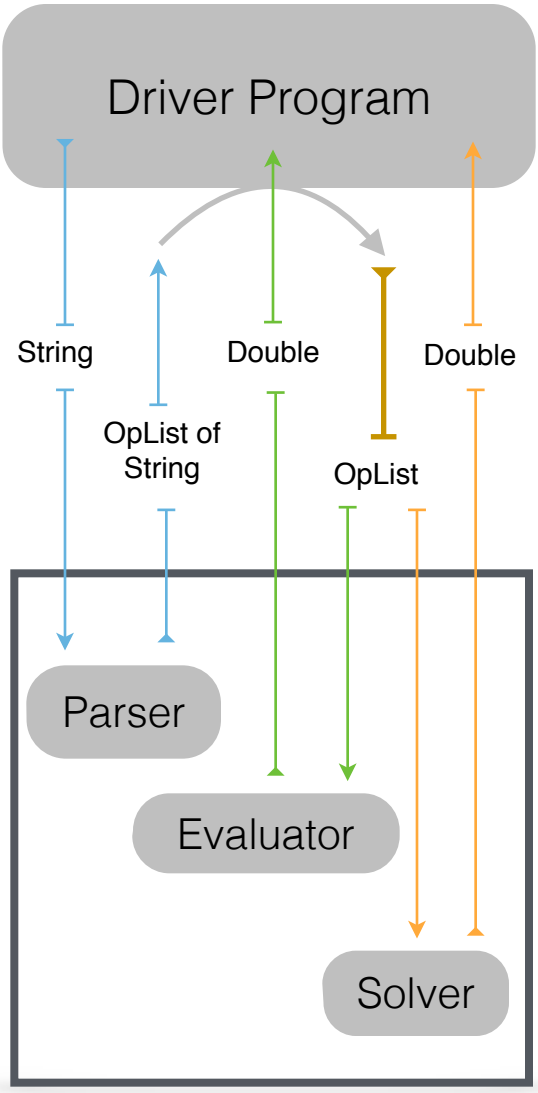

figure 4: Driver Program overview

Analysis

As we predicted the development time for the swift implementation of our project was shorter than that of the C counterpart. What we did not predict was the factor of this difference. It took almost twice as long to complete the C version. It is important to consider that the C version was the first implementation. This may have made it easier to complete the Swift implementation, however we planned out most of the data structures, algorithms, and required methods of our program in (very informal) pseudo code before we began coding. To further mitigate this risk we did not look at the C implementation while coding in Swift. We also generally assigned ourselves different tasks in each language (e.g. Cancik implemented the C version of the parse algorithm, while Narkizian implemented the Swift version). Another thing we did not predict was the ease at we were able to pick up Swift. It took us about an hour of internet browsing, and practicing to start the implementation.

Implementing Models In ADT

This was a very difficult aspect of the C implementation. When we began development, we found ourselves having to start over many times in C, as it was difficult to keep track of how a function and a struct pointer relate to one another. Without classes or OOP of any sort the code seemed to get more complicated and less readable as we added features. For each method a reference to the model needs to be passed before the operation is performed. While this aspect did not affect much of the program logic, it provided for some very annoying error-prone syntax. Another drawback of this approach was that we had to check for NULL references for each reference in each function before performing any operations. Because of the manual memory management we wrote constructor and destructor functions that allocate and free memory appropriately.

In contrast, during the swift implementation, it took us about 5 minutes to write an OpList class with a nested Operations class as no destructor is needed and the required “init” function of every class was very simple. When writing the functions we did not

have to deal with any possible NULL references. The syntax was also much cleaner over all. Calling a function ([Object Name].[Method]()) is far cleaner and much easier to read. This prevented lots of time wasted on finding and correcting unintentional syntax errors—as was the case many times in C.

Optionals

One of the best things we learned about the Swift programming language was its use of *optionals*. Rather than setting a property value to NULL as you would do in C, You must declare the property as *optional* with he "*?*" operator to explicitly designate that the variable either contains a value or nil (nil is the Swift equivalent of NULL). If an optional property is used it must be *unwrapped* with the ("!") operator.

We found this approach far easier to work with and far safer than working with pointers in C. If a nil variable is accidentally unwrapped we know exactly where it happened, and when it happened. Further, an optional value cannot be used without !, thus the programmer is explicitly reminded of weather a value can possibly be nil or not. This reminding led to much fewer errors as we often found ourselves far more aware of how we intended each variable to be used. One drawback of optionals, however, is that readability was slightly hurt. It is often difficult to see a small exclamation point in a large method call, and while this is something an experienced Swift programmer would likely not have, it was significant enough to slow us down a few times.

Method Implementations

This aspect made by far the biggest difference in terms of development time. Just to illustrate this point, our Operations.c file contains about 1175 lines of code, our Operations.swift files contains just over 750 lines. This is largely due to the fact that Swift has a rich set of easy to use data structures and methods for those structures. In contrast C has primitive data types and pointers—the rest is up to you.

In our implementation we found that an ordered collection of unbounded size was the best data structure for the OpList implementation. This does not exist in C's standard

libraries, and thus each Operation in the OpList also had to be implemented as a de facto node in a linked list of Operations. Swift includes arrays of mutable size. It also provides many useful methods such as count(), which returns the length of the array, and properties such as last, which points to the last element in the array. These methods and properties had to be manually implemented and maintained in the C ADT. Thus Swift's more comprehensive data structures allowed us to shorten the development time for the ADT by about a third.

Type Inference

In Swift you can declare a variable of any type with the "var" keyword, and a constant of any type with the "let" keyword. This, like optionals, allowed us to avoid trivial mistakes such as modifying incorrect variables or returning incorrect types. In C we often had to deal with silly and difficult to understand type errors during compilation that were avoided in Swift. This also contributed to the shorter development time.

Xcode 6

C code can be written in any number of text editors or IDEs, however Swift is meant to be used specifically with Xcode 6, Apple's IDE for OS X and iOS development. While this does not say much about our C implementation, it is worth noting the tremendous difference that Xcode made during the testing and debugging of our program. Xcode includes a feature called "Playground" which allows you to see what your code is doing while you type it. For example, it will display the current value of a variable on the right after you modify it. This was incredibly useful when we ran into infinite loops while solving for an unknown.

Conclusions

Implementing a program in C requires a lot carefulness and diligence. Implementing a program in Swift requires less of both. This is really the most significant difference between these two languages. From a theoretical perspective there is really no

functionality that Swift has that cannot be compensated for in C with some extra time and patience. Every segmentation fault will eventually be found if you look long enough, and every missing method will be eventually be implemented if you work hard enough. The basic concepts of procedural programming, such as control flow for instance, exists just as they did decades ago—and they will likely persist for decades to come. By implementing features that minimize risks and maximize organization and intervenes, Apple has essentially crafted a more convenient C (and a more convenient Java, Objective-C, C++...).

Theoretically the development process in C and Swift isn't that different. Practically, it is. Reflecting upon our work its clear how much more taxing the C version was than the Swift version. Simple things like finding the source of a segmentation fault took hours, while finding an exception in Swift took only a few seconds. As a result, we found ourselves almost unsatisfied with our Swift implementation. We expected to have had to climb a much steeper hill. Swift demonstrated the importance of convenience and intuitiveness in fostering the logical thinking required for programming. In the end this not only allows one to maximize efficiency, but to essentially write better code.

References

[1] Apple Inc. Xcode 6 Release Notes. https://developer.apple.com/library/ios/releasenotes/DeveloperTools/RN-Xcode/Chapters/xc6_release_notes.html , accessed 03-09-15

[2] Timmer, John. A fast look at Swift. http://arstechnica.com/apple/2014/06/a-fast-look-at-swift-apples-new-programming-language/1, accessed 03-09-15

[3] Apple inc. App Store Review Guidelines. https://developer.apple.com/app-store/review/guidelines/, accessed 03-09-15

[4] Ritchie, Dennis M. The Development of The C Langauge*. cm.bell-labs.com/cm/cs/who/dmr/chist.html. accessed 03-09-15