卒業論文 2015 年度 (平成 27 年)

Swift コンパイラの Bootstrap 化による可読性の向上

慶應義塾大学 環境情報学部
出水 厚輝

卒業論文要旨 - 2015 年度 (平成 27 年度)

Swift コンパイラの Self-host 化による可読性の向上

オープンソースとなったソフトウェアにおいては、その開発に関わるプログラマの増加とそれに伴うプログラムの修正や機能追加の増加によって、そのソースコードの可読性が拡張やそれに対するレビューの容易さに影響し、プロジェクト自体の成否に大きく関わる場合がある。

2015 年 12 月にオープンソースとなった Apple 社が中心となって開発しているプログラミング言語 Swift もそうした可能性の分岐点に立つソフトウェアの 1 つである。現在 Swift コンパイラの可読性は既存コードのコーディングスタイルへの習慣的な追従とレビューの徹底によって保たれているが、この形だけでは新しいコードの増加やプロジェクトメンバーの交代などによってその可読性が保てなくなる可能性が高い。

一方で、Swift では行われていないものの、現在利用されている多くの高級な汎用プログラミング言語では、コンパイル対象となる言語自体でそのコンパイラを記述する Self-host 化がよく行われている。Self-host 化を行うことによるメリットはいくつかあるが、たびたびモチベーションとしてあげられるのは、その可読性における優位点である。コンパイラを記述する言語とその対象言語が同じになれば開発者はより少ない知識でコンパイラのコードを読むことができる上、初期のコンパイラにおいては、それを記述している言語よりもそのコンパイル対象となっている言語のほうが必ず後発のものであるため、多くの場合により表現力が高く、可読性においてもより高い水準となるからである。

しかし、Swift の開発者チームでは C++で記述されたそのコンパイラを Swift で書き直すことよりも現在検討されている仕様の追加を行うことなどを優先しており、Self-host 化について具体的な議論を開始するまでには至っていない。ただ、オープンソースとなったことで Swift においてより高い可読性が要求されるようになってきている以上、Self-host 化が可読性の向上に対して十分な成果を与えるのであれば、それを見当する価値は十分にあると考えられる。

そこで本研究では、Swift で記述した Swift コンパイラを実装し、Self-host 化によって Swift コンパイラの可読性を向上させられることを検証した。検証には、実装した Swift コンパイラの構文解析を行うコードの行数と現行の Swift コンパイラにおいて同様の機能を担う箇所のコードの行数を比較した結果を用いている。検証の結果として、本論文では Swift コンパイラの Self-host 化によってその可読性が向上する可能性が十分にあることを示した。だだし、同時に行った Self-host 化に伴うデメリットの考察により、Self-host 化がコンパイラに対して与える他の影響を鑑みると、実際の適用に際してはより慎重にならなければいけないということもわかった。

キーワード:
1. コンパイラ, 2. Self-host 化, 3. プログラムの可読性,
4. プログラミング言語 Swift

慶應義塾大学 環境情報学部

出水 厚輝

Abstract of Bachelor's Thesis - Academic Year 2015

Improve the Readability of Swift Compiler by Self-hosting

English abstract here.

Keywords :
1. Compiler, 2. Self-hosting, 3. Readability of Program,
4. Swift Programming Language

Keio University, Faculty of Environment and Information Studies
Atsuki Demizu

目 次

図 目 次

表 目 次

第1章　序論

1.1　背景

2015 年 12 月、Apple 社が予てより Cocoa および Cocoa Touch フレームワークを用いたソフトウェアの開発用プログラミング言語として提供していた Swift をオープンソース化し、同時に Linux を中心としたさまざまなプラットフォーム上でのソフトウェア開発に使用するための拡張を開始した。これにより Swift は、Objective-C の担ってきた iOS や Mac OS X などの特定プラットフォームに向けたソフトウェアだけでなく、C++や Java など他の汎用プログラミング言語が担ってきたソフトウェアの開発においてもそれらの代替となり得る可能性を持つこととなっており、今後はこれまで以上に様々な拡張と修正が行われていくと予想される。それに加え、オープンソースソフトウェアにおいては多くのプログラマが開発に関わるようになる都合から、拡張や修正のためのコードに対するレビューのプロセスがバグを事前に防ぐためにより重要となる。

このような状況の変化によって、Swift コンパイラにはこれまで強く求められていなかったある特徴が求められるようになっている。それは、Swift コンパイラのソースコード自体における高い可読性である。

プログラムに対する拡張や修正の効率が、そのプログラムの可読性に大きな影響を受けることは Elshoff [1] で言及されている。また、コードレビューのようなプロセスでは、プログラムにおける実行速度などの性能の高さではなくコードの読みやすさによってその効率が左右されることが明らかである。これらのことから、以前は Swift コンパイラについてよく知る極小数のメンバーによって開発が行われていたために顕著化していなかった高い可読性に対する要求が、今後は高まっていくであろうと考えられる。

一方、Swift 以外の汎用言語のコンパイラにおいてはそのソースコードの可読性を高めることを目的の内の 1 つとして、コンパイラをそのコンパイル対象の言語自体で開発する Self-host 化を行っている例がよく見られる。

表 1.1 は Web 検索エンジンにおけるクエリヒット数からプログラミング言語の知名度を格付けした TIOBE Index [2] の 2015 年 12 月版において上げられている言語の内、汎用言語であるものだけを上位から 20 言語抽出し、それらの主要なコンパイラにおいて Bootstrap 化されているものがあるかをまとめたものである。なお、Bootstrap とは Self-host 化によってコンパイラのソースコードから他言語への依存を排除し、以降の新しいバージョンのコンパイラをそのコンパイラ自身でコンパイルできるようにすることを指す。単に Self-host 化だけが行われている場合はコンパイラ中のコンパイル対象言語で記述された箇所がごく僅かである場合もあるため、表 1.1 では Bootstrap 化しているかどうかについてまとめている。

表 1.1 中の 20 言語の内だけでも Bootstrap を採用しているものが 7 言語あり、 その中に性能の問題からコンパイラ用の言語として採用されづらいインタプリタ型言語なども含まれていることを考慮すれば、かなりの言語が Self-host 化されていることが分かる。

しかし、Swift の Self-host 化について Swift コンパイラのレポジトリ内に記載されている FAQ [3] では、Self-host 化した際に言語環境を用意するプロセスが煩雑化すること、現時点では Swift にコンパイラ開発用の特徴を追加するよりも汎用言語としての特徴追加を優先したいことから、短期的には Self-host 化を行う予定はないと述べられている。

表 1.1: 知名度の高いプログラミング言語の Bootstrap 状況

順位	言語名	Bootstrap されているか	Bootstrap されている主要コンパイラ
1	Java	×	-
2	C	×	-
3	C++	○	clang, gcc, Microsoft Visual C++
4	Python	○	PyPy
5	C#	○	.NET Compiler Platform
6	PHP	×	-
7	Visual Basic .NET	○	.NET Compiler Platform
8	JavaScript	×	-
9	Perl	×	-
10	Ruby	×	-
11	Assembly Language	(高級言語でないため除外)	
12	Visual Basic	×	-
13	Delphi/Object Pascal	○	Free Pascal
14	Swift	×	-
15	Objective-C	×	-
16	MATLAB	(汎用言語でないため除外)	
17	Pascal	×	-
18	R	(汎用言語でないため除外)	
19	PL/SQL	(汎用言語でないため除外)	
20	COBOL	×	-
21	Ada	○	GNAT
22	Fortran	×	-
23	D	○	DMD
24	Groovy	×	-

1.2 本研究が着目する課題

本研究では、 1.1 節で述べたように Swift コンパイラの可読性に対する要求が高まっているという点に着目する。

現在の Swift コンパイラにおけるコードの可読性は既存コードのコーディングスタイルへの追従やレビューの徹底などによって保たれているが、これはコミュニティベースで修正・追加されたコードがオープンソースとなる以前のコードの量を上回ったり、レビュアーが多様化することによって持続できなくなる。また、これを持続する単純なアプローチとしてコーディング規約などの策定を行うことも考えられるが、詳細な規約はそれを把握するためだけでも多くの労力を要し、結果として可読性向上のための試みが当初の目的であったはずのプログラムの拡張や修正のコストを増大させてしまう可能性がある。

1.3 本研究の目的とアプローチ

本研究では Swift コンパイラの可読性を現在のものよりも向上させることを目的とする。そのアプローチとして、Swift で記述した Swift コンパイラを実装し、そのソースコードの可読性を現行の Swift コンパイラと比較することで、Swift コンパイラの Self-host 化が可読性の向上に有効であることを示す。

コンパイラの Self-host 化には大きく分けて 3 つの可読性を向上させうる特徴がある。まず 1 つ目は、コンパイラのコードをコンパイル対象の言語で記述できるという点である。これにより、ソースコードを読むために必要な知識が減るだけでなく、そのコードは間違いなくそのコンパイラの全開発者にとってより馴染み深いものとなる。次に 2 つ目は、対象言語が最初にそのコンパイラを記述した言語よりも必ず後発のものとなるために、より高級な構文や機能を保つ場合が多いという点である。より高級な構文や機能は開発者の理解を助け、可読性を向上させる可能性が高い。最後に 3 つ目は、言語仕様中の一部の機能や構文について実際にその機能や構文を用いて実装することができるという点である。これもその直感的な記述から、一般的に可読性が高まると期待できる。

さらに、Self-host 化ではこれらのメリットをコンパイル対象の言語でそのコンパイラを記述し続ける限り継続的に享受できる。そのため Self-host 化によって、 1.2 節で述べた時間経過に伴って可読性が維持できなくなっていくという問題や追加の手順によって開発者のコストを増大させてしまうという懸念なく、単純に可読性を向上させ、コンパイラの拡張や修正のコストを減少させられると期待することができる。

しかし、可読性を向上させうる特徴の内、後者 2 つについては必ずしも可読性を向上させるとは限らない点に注意する必要がある。実際、Swift においては現行のコンパイラが既に多くの機能を持つ高級言語である C++で記述されているために、それらの特徴による可読性の向上が自明ではない。

そのため、本研究では実際に Self-host 化した Swift コンパイラを実装した上で、Swift と C++の違いからそれぞれのコンパイラのソースコードにおける可読性を決定する要因について慎重に検討し、各コンパイラのソースコードの行数に基づく定量的な指標を用いてその可読性を比較する。

また、コンパイラの Self-host 化はコンパイラのプログラムを全て書き換えてしまうために、可読性の向上だけでない様々な影響を与えうる。その点について、本論文ではいくつかの事例と実装したコンパイラにおける性能の評価から考察し、実際に Self-host 化を行うにあたっての注意事項としてまとめる。

1.4 本論文の構成

本論文における以降の構成は次の通りである。

2 章では、本研究が着目するプログラミング言語 Swift とそのコンパイラの記述言語 C++それぞれの特徴についてまとめ、可読性の差が生まれる原因となりうる差異について考察する。 3 章では、 2 章で述べる言語の差異を根拠に Swift と C++で記述されたコンパイラの可読性の差を検証するための方法について説明する。 4 章では、本研究で実装した Self-host 化された Swift コンパイラの設計について述べ、 3 章で示した検証を行うために十分な機能を有していることを示す。 5 章では、実際に現行のコンパイラと Self-host 化されたコンパイラについて 3 章で述べる可読性の比較を行い、その結果について考察する。 6 章では、Self-host 化がもたらしうる可読性の向上以外の影響について他の言語における事例から考察し、特にその影響が自明でない性能の低下があるかどうかを知るための方法について述べる。 7 章では、 6 章で述べる方法で現行のコンパイラと Self-host 化されたコンパイラの性能の差異を比較し、その結果について考察する。 8 章では本研究の結論と今後の展望についてまとめる。

第2章　SwiftのおよびC++の特徴と差異

本研究ではSelf-host化によってSwiftコンパイラの可読性を向上させることを目的としているが、その考えはSwiftで記述したSwiftコンパイラと現行のC++で記述されたコンパイラとで可読性に差が出るという仮説に基づいている。本章では、その仮説が確からしいと考えている根拠となる各言語の差異について、それらの特徴を比較して説明する。

2.1　比較する特徴

本章ではSwiftとC++について以下3つの視点から特徴を述べる。

1. 言語パラダイム
2. 型システム
3. 制御構文

まず、プログラミング言語が採用するプログラミングパラダイムはその言語に必要となる機能を決定し、たとえ新しい言語であったとしてもその構文はそれら機能の慣習的な表記法の影響を受ける。そのため、プログラミングパラダイムはその言語の可読性を左右する一つの大きな要因となっており、本章で取り上げるべき特徴であるといえる。

次に、SwiftやC++のような静的型付言語では採用する型システムによってプログラムに対する意味付けの柔軟性が大きく左右される。うまく意味付けのできないプログラムは可読性を低下させうるため、型システムについても本章で取り上げることとする。

最後に、分岐やループを作る制御構文はプログラムの流れを決定する役割を担うため、プログラムの流れの把握し易さにその設計が大きな影響を及ぼす。プログラムの流れを把握することがコードを読む大方の目的であるため、これも可読性に影響を与える1つの要因として本章で取り上げる。

2.2　Swiftの特徴

2.2.1　言語パラダイム

Swiftは近年の汎用言語に採用されている多くのプログラミングパラダイムを取り入れているマルチパラダイムプログラミング言語であり、以下のようなプログラミングパラダ

イムを採用している。

関数型プログラミング

Swift では関数を第一級のオブジェクトとして扱い、2 つの型から成る関数型を用意することで、ML や Haskell などの言語と同様のラムダ計算に近しい表記方法を行う関数型プログラミングが可能となっている。

Swift における関数型プログラミングの例をプログラム 2.1 に挙げる。関数用の型が矢印演算子によって提供されており、関数自体を変数に代入して使用できていることがわかるだろう。

プログラム 2.1: Swift における関数型プログラミングの例

```
1 let f: Int -> Int = { x in x + 1 }
2 print(f(1)) // 標準出力に 2 と表示する
```

このパラダイムにより、関数を引数として受け取る高階関数を用いるなどして関数をより抽象化し、ボイラープレートを防ぐことができるようになる。また、一般に関数の型はその表記が複雑化しやすいという問題があるが、Swift では型推論や型の別名定義などの手法によってこれを簡単に記述できるようになっている。

オブジェクト指向プログラミング

Swift が提供する複合型であるクラス、構造体、列挙体、プロトコルでは継承関係を定義することができ、外部の手続きから呼び出し可能な値や他の型定義などのメンバを持つことができるように設計されていることで、オブジェクト指向プログラミングを可能としている。

Swift におけるオブジェクト指向プログラミングの例をプログラム 2.2 に挙げる。この例では継承関係のあるクラスから生成されたオブジェクトに対し、クラスで定義された関数のメンバを呼び出している。

プログラム 2.2: Swift におけるオブジェクト指向プログラミングの例

```
1  class Parent {
2      func f() { print("parent") }
3  }
4
5  class Child : Parent {
6      override func f() { print("child") }
7  }
8
9  let x: Parent = Child()
10 x.f() // 標準出力に child と表示する
```

継承関係を持つことができる複合型はプログラムのカプセル化の礎を築きプログラムの再利用を容易にする他、関数や変数をメンバとして保持することでそれらの関係性をより明確に定義できるようになる。

パターンマッチ

Swift は特定の構造を持つ値についてその一般的なパターンを定義し、変数を含む左辺値と変数を含まない右辺値を比較することで左辺値中の変数の型と値を決定するパターンマッチの機構を持っている。

Swift におけるパターンマッチの例をプログラム 2.3 に挙げる。現在の Swift ではこの例内の 6 行目のような列挙体やタプルの値について特に柔軟なパターンマッチを提供している。

プログラム 2.3: Swift におけるパターンマッチの例

```
1 enum Sample {
2     case X, Y(Int)
3 }
4 let x = Sample.Y(1)
5
6 if case Sample.Y(let v) = x {
7     print(v) // 標準出力に 1 と表示する
8 }
```

パターンマッチでは値の構造を元にした処理の分岐を端的に記述することができ、パターンマッチ中の変数束縛や変数定義式でのパターンマッチは複数の値を同時に扱う自然な方法を提供する。

2.2.2 型システム

Swift の型システムはその柔軟性によってプログラマの様々な意味を表現できるだけでなく、型推論によって煩雑な型の省略を可能にし、プログラムの可読性を向上している。以下では Swift で採用されている型システムの特徴について述べる。

型パラメータ多相

Swift では各複合型や関数内で用いられている型を全称型を持つ変数で記述することにより、複数の型を対象とすることができる型パラメータ多相を採用している。

型パラメータ多相により任意の型を包む高階型を定義することで、モナドなどのより抽象化されたオブジェクトの実装が可能となり、うまく実装を行うことで類似の処理について各型毎に繰り返し記述する必要を無くすことができる。

関数のオーバーロード

Swift の型システムでは、引数の数が異なる関数や引数の型が異なる関数を同じ名前で定義する関数のオーバーロードを許している。

この機能により、演算子を他の関数と同じように扱い、演算子の定義を特定の型についてオーバーロードした実装を与えることで、各型毎に適切な振る舞いを行うように演算子をプログラマがカスタマイズすることができる。

型推論

Swift では関数型プログラミングを可能とする多くの言語と同様に Hindly と Milner によるアルゴリズムを採用した強力な型推論を備えている。

この型推論アルゴリズムにより、Swift ではかなり多くの式について型推論が可能となっており、プログラマはほとんどの場合に明示的な型注釈を行うかどうかを自分で選択できる。

2.2.3 制御構文

Swift の制御構文には他の汎用言語と比較しても多くの種類と機能がある。また、その条件分岐にパターンマッチを使用できる点も Swift の制御構文における特徴の 1 つである。以下では特徴的な構文について述べる。

For-in 構文

For-in 構文は配列の各要素に対する繰り返しを提供する。プログラム 2.4 のように配列要素の束縛にはパターンマッチを使用できるため、配列要素に対する繰り返しのボイラープレートを無くすことができる。

プログラム 2.4: For-in 制御構文

```
let list = ["a", "b", "c"]

for (i, e) in list.enumerate() {
    print("\\(i): \\(e)")
}
/* 標準出力に
0: a
1: b
2: c
と表示する */
```

Guard 構文

Guard 構文は現在実行しているプログラムの継続をやめ、フローを脱出するような条件分岐を記述するために使用できる。プログラム 2.6 は数の階乗を求めるプログラムだが、関数 factorial は引数 n に負数が与えられた場合や 0 が与えられた場合に限った別の操作を行うために Guard 構文を利用している。

プログラム 2.5: Guard 制御構文

```
1 func factorial(n: Int) -> Int? {
2     guard n > 0 else {
3         return nil
4     }
5     guard n != 0 else {
6         return 1
7     }
8     return n * factorial(n - 1)
9 }
```

Defer 構文

Defer 構文は現在のスコープを脱出する際に実行する処理を指定するために使用できる。プログラム **??**ではファイルのクローズが readFromFile 関数の終了時に必ず実行されるように Defer 構文を利用している。

プログラム 2.6: Guard 制御構文

```
1  func readFromFile(file: String, size: Int) -> String? {
2      let fp = fopen(name, "r")
3      defer {
4          fclose(fp)
5      }
6      let buffer = [CChar](count: size, repeatedValue: 0)
7      let ret = fread(&buffer, sizeof(CChar), size - 1, fp)
8      guard ret != 0 else {
9          return nil
10     }
11     guard let str = String.fromCString(&buffer) else {
12         return nil
13     }
14     return ret
15 }
```

2.3 C++の特徴

2.3.1 言語パラダイム

2.3.2 型システム

2.3.3 制御構文

2.4 Swift と C++の差異

2.4.1 機能的特徴の差異

2.4.2 構文的特徴の差異

第3章　可読性の差の評価方法

本章では、Self-host 化した Swift コンパイラと現行の Swift コンパイラとの間で可読性の差を評価する方法について述べる。

3.1　可読性の比較に使用する指標

本研究で実装した TreeSwift は、 ??節で示したとおり現行の Swift コンパイラと同等の構文解析機能を保持しており、 4.1 節で述べたとおりそれらの機能が同様の手法で実現されている。そのため、TreeSwift と現行の Swift コンパイラにおいては構文解析器のソースコードの行数がより少ない方が、同じ意味を把握するために参照しなければならない行が少なく、より可読性が高いと考えることができる。

3.2　比較対象

比較に用いるコンパイラの要素としては構文解析器以外にも意味解析部分、コード生成部分、最適化部分が考えられるが、これらは次の理由により本研究の目的に見合った単純な比較ができないため対象としない。

まず、コード生成部分については現行の Swift コンパイラと Swift で記述する Swift コンパイラの間で使用する LLVM の API が異なってしまう点で問題がある。現行の Swift コンパイラは C++で記述しているために C++用の LLVM API を使用するが、実装するコンパイラでは Swift に C++との相互運用性がないため、C 用の LLVM API を使用している。これらの API 間ではその設計が大きく異なり、その差によって結果に大きな影響が出てしまう可能性があるため、コード生成部分は比較の対象としない。

次に最適化部分については、その実装が言語の仕様によって制限されないという点に問題がある。仕様に制限されないために各コンパイラでのサポート状況によって大きくその実装が異なってきてしまうため、比較の対象としない。

最後に、意味解析部分には主に型推論、型検査、型決定後の参照解決、詳細なエラー検出、言語レベルでの最適化などが含まれるが、このステップ中の各項目は密接に関わりあっており、コンパイラから切り出した際の単位が大きくなってしまうという問題がある。評価する対象の単位が大きいと、その評価の結果は複数の機能間で相互に打ち消し合い、実際には各機能によっていい結果と悪い結果が混在しているにも関わらず、押し並べた結果のみが表出してしまう可能性がある。そのため、意味解析部分についても比較の対象とはしない。

3.3 比較するプログラムの同一性

3.4 本研究における可読性の差の検証方法

各コンパイラのソースコードの内、ファイルを読み込み、字句解析および構文解析を行い、AST を生成するまでに実行されるプログラムを記述した部分からエラー分を定義している箇所以外を抜き出し、その行数を計測・比較する。

第4章　Self-host化したSwiftコンパイラ「TreeSwift」の設計と実装

本章では、??章で述べた現行のSwiftコンパイラの特徴から比較対象となるSwiftで記述したSwiftコンパイラTreeSwiftの構文解析器が満たすべき特徴を整理し、それがどのように実装されているかについて述べる。

4.1　構文解析器が満たすべき特徴

Bootstrapを行う上でBootstrap後のコンパイラが満たすべき要件は、基本的にはその言語の仕様を満たしていることと、以前よりも性能が改善されていること以外にはない。しかし、本研究では現行のSwiftコンパイラと比較し考察することが目的であるため、評価を行うために必要な同一の特徴を保持している必要がある。以下では、その特徴について述べる。

LL(k)方式の採用

現行のコンパイラの構文解析器と同等の解析力とエラー検出力、拡張性を保持していることを保証するため、TreeSwiftでもLL(k)方式の構文解析器をSwiftで書き下すことによって実装を行う。

構文解析以外の部分の分離

現行のコンパイラとの比較時に比較箇所が行う処理の差が出ないようにするため、型の推論や検査など構文解析以外の部分を構文解析器から分離する。

4.2　実装の概要

本節では4.1節で示した特徴を満たしながら、TreeSwiftがどのように実装されているかを説明する。

4.2.1 構文解析機能

ソースコードの構文を解析し、AST にまとめ上げる処理は字句解析と構文解析の 2 つのステップに分けることができる。

字句解析

TreeSwift では図のように、入力となるソースコードの文字を 1 字ずつ読みながら分類し、リテラルや識別子、予約語といった文字数の定まっていないトークンについてはそれぞれに専用の状態遷移機械によって生成する。

構文解析

解析に使用する構文は Apple 社の提供する Swift の公式ドキュメント [4] にある構文をベースとするが、本資料は多くの誤りを含んでおりかつ LL(k) 形式では解析不可能な左再帰を含んでいるため、それらを修正した独自の構文定義を使用している。

また、TreeSwift では **??**節で述べていたような現行の Swift コンパイラが判断できていない構文についても、独自の構文定義の追加によって構文的に分類・区別している。そのため、TreeSwift では全ての構文について構文解析が完了した時点で曖昧性が排除されている。

4.2.2 参照解決機能

変数や型などの参照を解決する機能の実現には、スコープを管理し、解決しなければならない。また、参照解決を行うタイミングもその言語仕様に合わせた選択が必要となる。

参照解決のタイミング

TreeSwift では **??**節で述べた現行の Swift コンパイラの参照解決とは異なり、構文解析中には一切の参照解決を行わない。そのため、構文解析が完了して参照解決を行う際にはすでにすべての参照解決が可能であることが保証されており、参照解決を構文解析の本体から分離することが可能となっている。

スコープ解決

TreeSwift におけるスコープの解決は構文解析時に AST とは別に構築されるスコープツリーを用いて行う。

スコープツリーでは分岐やループ、複合型の宣言などに伴って切り替わるスコープの入れ子構造を木として表現し、木の各ノードがそのスコープ内で宣言された変数や関数、型

などをメンバとして保持する。また、木の各ノードは後の参照解決のためにスコープ内で参照された変数や関数、型などの情報も保持する。

この構造により、参照解決時には木の各ノードを走査し、各参照情報についてその直近の親ノードから順に参照の対象がメンバとして含まれているかを遡って確認していけば、参照解決を行うことができるようになる。

曖昧な構文の解決

また、糖衣構文などを提供するために構文の曖昧性を残しているのも Swift の特徴である。例えば、プログラム 4.1 のような構文は Swift の構文定義に則った全く正しいコードだが、一般的なアプローチでは正しく解析することができない。x の後に続く { が x の引数のクロージャの始まりなのか if 文の本体の始まりなのかは、x が引数としてクロージャを取る関数であることを解析機が知っていなければ決定できず、かつその x はこれから解析を行う他のファイルなどに宣言されている可能性があるために、全てのファイルの解析を完了するまで x の参照を解決することはできない可能性があるためである。

プログラム 4.1: 曖昧な構文を持った正しい Swift コード

```
1 let x: (() -> ()) -> Bool = { f in true }
2
3 if x { print("closure") } { print("if") }
```

こうした構文を一切の曖昧性なしに解析するためにはコンパイル対象となるプログラムを参照の解決に必要な回数だけ走査しなくてはならなくなるが、それは少なく見積もっても大変非効率であり、現実的ではない。現状の Swift コンパイラではこうした曖昧な構文は可能性のあるどちらかの構文に決め打ちして実装されており、もう一方を意図して記述しているプログラマに対しては全く見当はずれのエラーメッセージを提示することとなっている。

4.2.3 構文解析器のその他の機能

エラーの処理

TreeSwift の実行中に発見されたソースコードのエラーは構文解析ステップを通して同じ 1 つのインスタンスによって処理される。構文解析器からはエラーを発見した時点でエラー文の参照と解析対象に関する情報をそのインスタンスに渡す。そのエラーが致命的なものであった場合や、エラーを管理するインスタンスに設定された既定値よりも多くのエラーが報告されていた場合は、その時点で Swift のエラーハンドリング機能を用いて解析中の構文の関数を抜け、エラー表示などの処理を行う。

ライブラリの扱い

TreeSwift では独自のバイナリ形式を使用している現行の Swift コンパイラとは異なり、標準ライブラリを含むライブラリ内に定義されている変数や関数、型などの情報を保持するモジュール情報ファイルをテキストファイルで管理している。それらのライブラリは import 文によって要求された時か、標準ライブラリの場合はプログラマがインプットした全てのソースコードの解析前に一度だけ解析され、一般的なソースコードと同様の AST を形成する。

ただし、ライブラリのモジュール情報ファイルでは複数のファイルにまたがったグローバル変数などの定義は存在しないため、構文解析中に即座に参照解決を行う。

4.2.4 構文解析以外の実装

第5章　可読性の向上に関する評価

本章では、 4 章で説明した TreeSwift と現行の Swift コンパイラとの比較評価の方法とその結果について述べる。

5.1　評価概要

評価の目的は Self-host 化によって Swift コンパイラの可読性が向上していることを検証することである。

5.1.1　比較対象

本研究では、各コンパイラの構文解析を行う箇所にのみ注目して比較評価を行う。

比較に用いるコンパイラの要素としては他に意味解析部分、コード生成部分、最適化部分が考えられるが、これらは次の理由により本研究の目的に見合った単純な比較ができないため対象としない。

まず、コード生成部分については現行の Swift コンパイラと TreeSwift との間で使用する LLVM の API が異なってしまう点で問題がある。現行の Swift コンパイラは C++で記述しているために C++用の LLVM API を使用しているが、実装するコンパイラでは Swift に C++との相互運用性がないため、C 用の LLVM API を使用している。これらの API 間ではその設計が大きく異なり、その差によって結果に大きな影響が出てしまう可能性があるため、コード生成部分は比較の対象としない。

次に最適化部分については、その実装が言語の仕様によって制限されないという点に問題がある。仕様に制限されないために各コンパイラでのサポート状況によって大きくその実装が異なってきてしまい、同等の機能が比較されていることの確認が難しいため、ここでは比較の対象としない。

最後に、意味解析部分には主に型推論、型検査、型決定後の参照解決、詳細なエラー検出、言語レベルでの最適化などが含まれるが、このステップ中の各項目は密接に関わりあっており、コンパイラから切り出した際の単位が大きくなってしまうという問題がある。評価する対象の単位が大きいと、その評価の結果は複数の機能間で相互に打ち消し合い、実際には各機能によっていい結果と悪い結果が混在しているにも関わらず、押し並べた結果のみが表出してしまう可能性がある。そのため、意味解析部分についても比較の対象とはしない。

5.1.2 評価方法

評価は各コンパイラが特定の Swift プログラムを構文解析する際に実行したコードの行数を比較することで行う。

より具体的には、評価に用いる Swift プログラムをコンパイラがロードし、字句解析並びに構文解析を行って、メモリ内にその AST を構築するまでに実行された機械語に対応するソースコードをコンパイラの全ソースコードから抽出し、その行数が少ない方をより可読性が高いとする。この際、コンパイラオプションの解析などのコンパイルのための下準備を行っている箇所は対象としない。また、標準ライブラリや依存モジュールの構文解析については、TreeSwift では特に独自の形式を定義していないためモジュールの定義にテキストファイルを使用しているのに対して、現行の Swift では独自のファイル形式を使用しており、そこで大きな差が生まれてしまうことを避けるために対象としない。

本研究で実装した TreeSwift は、 4 章で示したとおり現行の Swift コンパイラと同様の手法で構文解析器が実現されている。そのため、実行されたコードの行数を比較することで、同じ Swift プログラムを同様に解析できる構文解析器の内容を把握するために参照しなければならない行が少なく、より可読性が高いと考えることができる。

5.2 計測

5.2.1 計測内容

計測には

- 実用的な複数のベンチマークプログラム
- 構文解析器中の実行可能なコードの 90%以上を実行する 1 つのプログラム
- 構文解析器中の実行可能なコードの 90%以上を実行する複数のプログラム

を使用する。また、計測が Swift の構文を充分に網羅していることを示すため、各プログラムのコンパイル時に実行されたソースコードの、現行の Swift コンパイラの構文解析器中の実行可能なコード全体に占める割合も計算する。

5.2.2 計測結果

- 各プログラムで行数は少なくなる
- 実行されたコードの割合についても記述

5.3 考察

- コードの割合から充分に網羅されていることを確認
- 行数が少なくなっているので可読性の向上はできている
- 行数が少なくなっている原因を幾つかの箇所をピックアップして考察する
- 行数が少なくなっている原因の 1 つとして、コンパイラのチューニングが甘いという可能性がある (C++ではポインタやインライン関数など様々な機能でチューニングできる)
- Self-host 化はコンパイラのコード全体に影響をあたえるので、他の影響が出ていないかを調べる必要がある
- 他の言語の事例を上げて性能以外のデメリットについてもまとめて問題がないか確認する

第6章　事例からみるSelf-host化の副作用

5.3節で述べたように、Self-host化はコンパイラに対して可読性の向上以外の様々な影響を与えている可能性がある。そこで本章では、これまでにBootstrapを行ってきた高級汎用言語の事例を紹介し、それらの例からSwiftのSelf-host化において発生しうる可読性の向上以外の作用について整理する。

6.1　Bootstrapの事例

BootstrapはFortranやLispのような比較的古い言語からGoやF#のような比較的新しい言語まであらゆる時代の言語で行われており、その目的は様々である。本節では、その中から特に近年よく使用されており、先に他の言語による実装が十分な機能を持ってリリースされているにもかかわらずBootstrapを行った高級汎用言語である、Goのgo、PythonのPyPy、C#の.NET Compiler Platformの3つの事例について紹介する。なお、本節でSelf-host化ではなくBootstrap化を行った言語を対象としているのは、1.1節で述べたのと同様にSelf-host化だけを行っている言語にはコンパイラの大部分を書き換えているものと一部分だけを書き換えているものだけがあり、千疋が難しいためである。

6.1.1　Go - go

Goは2009年にGoogle社より発表された、構文の簡潔さと効率の高さ、並列処理のサポートを中心的な特徴とする静的型付コンパイラ型言語である。発表から6年を経た2015年にリリースされたバージョン1.5でBootstrapが行われ、それまでCで記述されていたコンパイラは完全にGoへと書き換わった。

Bootstrapの目的

GoコンパイラのBootstrapの目的はCとGoの比較という形で[5]内で以下のように述べられている。

- It is easier to write correct Go code than to write correct C code.
- It is easier to debug incorrect Go code than to debug incorrect C code.

- Work on a Go compiler necessarily requires a good understanding of Go. Implementing the compiler in C adds an unnecessary second requirement.
- Go makes parallel execution trivial compared to C.
- Go has better standard support than C for modularity, for automated rewriting, for unit testing, and for profiling.
- Go is much more fun to use than C.

主に Go を用いたコンパイラの開発が C を用いた場合よりも正確かつ楽になるという点を強調していることから、Go コンパイラの Bootstrap の目的は主にコンパイラ開発フローの改善にあったということができるだろう。

Bootstrap の方法

Go コンパイラにおいては、[5] に詳細な Bootstrap のプロセスが記述されている。これによれば、Bootstrap はおおまかに以下の流れで行われた。

1. C から Go へのコードの自動変換器を作成する。
2. 自動変換機を C で書かれた Go コンパイラに対して使用し、新しいコンパイラとする
3. 新しい Go で書かれたコンパイラを Go にとって最適な記法へと修正する
4. プロファイラの解析結果などを用いて Go で書かれたコンパイラを最適化する
5. コンパイラのフロントエンドを Go で独自に開発されているものへと変更する

この手法では、特に C から Go への自動変換器を作成したことで、C で書かれたコンパイラの開発を止めること無く Go への変換の準備を行うことができた、という点が優れている。ただしこの手法を取れたのは、Go が C に近い機能を多く採用していたこと、C と Go が共に他の高級汎用言語と比べて少ない構文しか持っていなかったことに依るところが大きい。

また、バージョン 1.5 以降の Go コンパイラにおいてはまず Go のバージョン 1.4 を用いてコンパイルし、その後にそのコンパイラを再度自分自身でビルドすることによって最新バージョンのコンパイラでコンパイルした最新バージョンのコンパイラを得る [6]。この多少複雑な形によりバージョン 1.5 以降でもコンパイラを C から独立させることができるが、その代わりに Go のバージョン 1.4 に依存し続ける点には注意しなくてはならない。

Bootstrap の結果

Bootstrap が行われた結果、Go コンパイラのコンパイル速度が低下したことがバージョン 1.5 のリリースノート [7] で言及されている。これについて同リリースノートでは C から Go へのコード変換が Go の性能を十分に引き出せないコードへの変換を行っているためだとしており、プロファイラの解析結果などを用いた最適化が続けられている。

6.1.2 Python - PyPy

Python は 1991 年に発表されたマルチパラダイムの動的型付けインタプリタ型言語である。Python の最も有名な実装である CPython は C 言語で書かれているが、その CPython と互換性があり、Bootstrap された全く別のコンパイラが 2007 年に PyPy という名前でリリースされている。

この PyPy では CPython と比べて JIT コンパイル機能を備えている点が最も大きな違いとなっている。

Bootstrap の目的

PyPy が Bootstrap を行った目的はそのドキュメントである [8] 内の以下の記述から、特に Python という言語の持つ柔軟性と、それによる拡張性の高さを利用するためであると読み取れる。

> This Python implementation is written in RPython as a relatively simple interpreter, in some respects easier to understand than CPython, the C reference implementation of Python. We are using its high level and flexibility to quickly experiment with features or implementation techniques in ways that would, in a traditional approach, require pervasive changes to the source code.

Bootstrap の方法

PyPy のインタプリタは、PyPy と同時に開発されている RPython という Python のサブセット言語で実装されており、RPython は RPython で書かれたプログラムを C などのより低レベルな言語に変換する役割を担う [9]。

そのため、RPython の実行時の性能は PyPy 自体の性能に一切関与せず、例えば Python で記述されている RPython が PyPy で実行されているか CPython で実行されているかは PyPy の性能に何ら影響を与えない。

この RPython という変換器による仲介と、既存実装である CPython との互換性が PyPy の Bootstrap を可能にしている。

Bootstrap の結果

PyPy については多くのベンチマークにおいて互換性のある CPython のバージョンに対してその実行速度が向上していることが示されている [10]。これは PyPy が単に Bootstrap を行っただけでなく、RPython によって PyPy インタプリタをネイティブコードにコンパイルできるよう仲介した上で、JIT コンパイル機能を付加したためである。

このように、Bootstrap を行ったインタプリタを直接そのインタプリタで実行するのではなく、より高速に動作する形へ変換して実行することで、性能の低下を免れられ、それどころか独自の拡張によってそれまでの実装よりもより高い性能を得られる場合がある。

ただし、この手法を取った場合は PyPy における C のような他の低級言語への依存が残ってしまい、その可搬性に制限が生じてしまう可能性があるという点には注意する必要がある。

6.1.3 C# - .NET Compiler Platform

C#は 2000 年に Microsoft 社より.NET Framework を利用するアプリケーションの開発用に発表された、マルチパラダイムの静的型付けコンパイラ型言語である。2014 年に同社は C++で記述されていたコンパイラの Bootstrap を行い、Visual Basic .NET と合わせてコンパイラ中の各モジュールを API によって外部から利用できるようにした.NET Compiler Platform をプレビュー版としてリリースした。その後 2015 年には Visual Studio 2015 における標準の C#コンパイラとして.NET Compiler Platform を採用するようになっている。

Bootstrap の目的

.NET Compiler Platform はコンパイラの構文解析や参照解決、フロー解析などの各ステップを独立した API として提供している [11]。これにより、例えば Visual Studio などの IDE はこれらの API を使用することで、いちから C#のパーサを構築すること無くシンタックスハイライトや定義箇所の参照機能を提供できる。そうしたライブラリ的機能をスムーズに利用できるようにするためには、.NET Compiler Platform 自体がそれを利用する Visual Studio の拡張などと同様の言語で提供されている必要があった。

その結果、.NET Compiler Platform は C#コンパイラを C#、Visual Basic .NET を Visual Basic .NET で記述する Bootstrap の形式で開発することとなっている。

Bootstrap の方法

.NET Compiler Platform は Visual Studio 2013 以前に使用されていた Visual C#とは独立して開発され、Visual Studio 2013 に採用されていた C# 5 の次期バージョンである C# 6 の実装となっていた。そのため、.NET Compiler Platform の開発において Visual C#に対する大きな変更などは行われておらず、全ての新機能を.NET Compiler Platform のみに対して適用するだけで事足りている。

また、.NET Compiler Platform の最新版は Visual Studio の最新版とともにバイナリ形式で配布されることが前提となっており、公開されているソースコードからビルドを行う場合でも配布されているコンパイラを使用する。そのため、配布されている Visual Studio が実行可能なプラットフォーム以外で.NET Compiler Platform を使用するためにはそれを実行可能な環境でクロスコンパイルするか.NET Compiler Platform 以外の C#コンパイラを用いてコンパイルする以外に方法がないが、その明確な手立ては示されていない [12]。

Bootstrap の結果

Microsoft 社では Bootstrap を行うにあたってその性能に対して非常に注力しており、結果として Bootstrap 後も想定していた充分によい性能が発揮できていると [13] 内で述べている。

6.2 Swift における Self-host 化の副作用

表 6.1 に 6.1 節で述べた事例から Bootstrap における利点をまとめ、Swift の Bootstrap 時にそれらの利点が得られる可能性があるか否かをまとめた。

表 6.1: Swift でも享受できる可能性のある Bootstrap の利点

利点	Swift での享受
プログラムの記述やデバッグがより容易になる	◯
コンパイラの開発を行うために必要な知識が減る	◯
並列化やモジュール化、テスト、プロファイリングなどにおいて高いサポートが得られる	×
より柔軟で拡張性が高くなる	?
コンパイラの各フローをライブラリとして提供できる	×

Swift で記述されたプログラムは現行のコンパイラで使用されている C++で記述されたものと比較して、記述やデバッグが容易になる可能性が高い。詳細は **??**章で述べているが、メモリの管理を自動的に行う ARC 方式や強力な型推論と型検査は C++で必要となる冗長な記述を省略し、多くのエラーをコンパイル時に発見する。ただし、飽くまでもこれはプログラムの記述の容易さだけに関する議論であり、 6.1.1 節で見たように、より簡単な記述で同等の性能が得られるとは限らないという点に注意されたい。

コンパイラの開発を行うために必要な知識が減るという点は 6.1.2 節で述べた PyPy のような形でなければほとんど全ての Bootstrap に対して有効である。Swift はコンパイラ型言語であるため、よほど特殊な方法を採用しない限りはこのメリットを享受できると考えられる。

並列化やモジュール化、テスト、プロファイリングに対するサポートは Swift と C++の間で同等か、Swift は特に Mac OS X 以外のプラットフォームにおける各機能のサポートが不十分なため、C++の方に分配が上がる可能性が高い。

1.1 節でも述べたように、言語の柔軟性および拡張性にはその可読性が大きく影響することが [1] によって示されているため、これが Swift においても利点となるかどうかは可読性が向上するかどうかという本研究の評価によって判断することができる。

6.1.3 節で挙げた.NET Compiler Platform のようにコンパイラの各フローをライブラリとして提供することは設計次第で可能だろう。しかし、現状の Swift には C#に対する Visual Studio のようなその言語で記述された IDE や言語のためのツールなどは存在して

おらず、その API を Swift で提供したとしても、特筆すべきほどの利点にはなり得ない可能性が高い。

表 6.2 に 6.1 節で述べた事例から Bootstrap において発生しうる課題をまとめ、Swift の Bootstrap 時にそれらの課題が問題となるかどうかをまとめた。

表 6.2: Swift の Bootstrap 時に発生しうる課題

課題	Swift の Bootstrap 時に問題となるか
現行のコンパイラの開発に対する影響	◯
過去のバージョンに対する依存	◯
コンパイル速度の低下	?
他の低級言語への依存	×
他のプラットフォームへの移植の煩雑化	◯

Swift の言語仕様は日々メーリングリストで改善のための議論が行われており、既に次期バージョンである 3.0 での破壊的な変更も定まっているように、まだまだ安定していない。そのため、 6.1.3 節で挙げた C#のように全く別のプロジェクトとして Bootstrap された Swift コンパイラを作成する場合には、現行のコンパイラと Bootstrap しているコンパイラの両方のメンテナンスコストが非常に高くなってしまう。これを防ぐためには Bootstrap のプロセスにおいてコンパイラへの大きな仕様変更などを行わないようにしなくてはならず、現行のコンパイラの開発に対する影響は避けられなくなってしまう。なお、 6.1.1 節で述べた Go の例のように C++から Swift への変換器を作成する形式は現実的ではない。なぜならば、C++と Swift は互いに言語仕様が複雑かつ多様であり、かつ Swift の言語仕様は先述の通り破壊的に変更されているため、その変換器を作成するためのコストは Bootstrap によって得られるメリットよりも格段に大きくなる可能性が高いからである。

過去のバージョンに対する依存は、 6.1.1 節で述べた Go のような方法を取れば避けられない課題である。かつ、現状の Swift においてはバージョン間で破壊的な変更があるため、それがコンパイラの機能を大きく制限してしまう可能性が高い。この問題は 6.1.3 節で述べた.NET Compiler Platform のようにコンパイラの配布の基本をバイナリ形式とすることで解決できる。ただし、その場合は.NET Compiler Platform と同様に他のプラットフォームへの移植の煩雑化という問題とのトレードオフに陥る点には注意しなくてはならない。

コンパイル速度の低下は 6.1.1 節にある Go のように変換器を使用しなかったとしても起こりうる可能性があるが、 6.1.3 節にある.NET Compiler Platform の事例のようにそのパフォーマンス改善に注力することで十分な性能を得られる可能性も等しくある。この結果を形式的な議論から導くことは難しいが、本研究で可読性の評価に用いる構文解析器は 1.3 節で示した構文解析器の比較が可読性が向上したかどうかの検証に使用できるということと同じ理由で、その実行速度やメモリ使用量を比較し、その変化を判断するために

使用できる。

他の低級言語への依存は、 **??**節でも述べたように Swift 自体がコンパイラ型言語であるため、起こりづらいと考えられる。

6.3 性能の差の検証方法

第7章　性能の低下に関する評価

7.1　評価概要

- 性能の低下がないかを確認するために実行速度と実行時の最大メモリ使用量を比較する
- 行数の比較と同様に特定のプログラムをコンパイルした時の構文解析の性能だけを評価する

7.2　計測

7.3　考察

第8章　結論

本章では、本研究の結論と今後の展望を示す。

8.1　本研究の結論

本研究では、Bootstrap 化によって Swift コンパイラの可読性を向上することを目的として、実際に Swift で記述した Swift コンパイラを実装し、その構文解析器について現行のコンパイラとの比較を行った。実装したコンパイラの構文解析器はソースコードの行数から可読性の変化を判断するために、現行のコンパイラと同様の手法で設計した上で同等の機能を有していることを示した。

ソースコードの行数を比較した結果から、Swift コンパイラを Bootstrap することによりその可読性を向上できる可能性が高いことが分かった。

ただし、可読性の検証に用いた構文解析器で性能の評価も行うことにより、Bootstrap したコンパイラが現行のコンパイラと全く同じコード生成機能を有したと仮定すると、Bootstrap によって可読性の向上と共にコンパイルの実行速度の低下とコンパイル中の最大使用メモリ量の増加を招く可能性が高いことも分かった。

これらの結果に加えて **??**節で事例から検討した Swift の Bootstrap における課題を考慮にいれると、Bootstrap によって可読性が向上するからといってすぐにそれを適用するのではなく、本当に現在の Swift に必要な機能を今後の方向性から検討しなおし、実際の適用にあたっては慎重に判断していく必要があると考えられる。

8.2　今後の展望

8.2.1　構文解析器以外の比較

8.2.2　継続的な比較

謝辞

参考文献

[1] James L. Elshoff and Michael Marcotty. Improving computer program readability to aid modification, 1982.

[2] Tiobe software: Tiobe index. http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html, December 2015.

[3] Frequently asked questions about swift. https://github.com/apple/swift/blob/2c7b0b22831159396fe0e98e5944e64a483c356e/www/FAQ.rst, December 2015.

[4] The swift programming language (swift 2.1): Summary of the grammar. https://developer.apple.com/library/ios/documentation/Swift/Conceptual/Swift_Programming_Language/zzSummaryOfTheGrammar.html#//apple_ref/doc/uid/TP40014097-CH38-ID458, December 2015.

[5] Russ Cox. Go 1.3+ compiler overhaul. https://docs.google.com/document/d/1P3BLR31VA8cvLJLfMibSuTdwTuF7WWLux71CYD0eeD8/edit, December 2015.

[6] Russ Cox. Go 1.5 bootstrap plan. https://docs.google.com/document/d/1OaatvGhEAq7VseQ9kkavxKNAfepWy2yhPUBs96FGV28/edit, December 2015.

[7] Go 1.5 release notes. https://golang.org/doc/go1.5#performance, December 2015.

[8] Goals and architecture overview — pypy 4.0.0 documentation. http://doc.pypy.org/en/latest/architecture.html, December 2015.

[9] Goals and architecture overview — rpython 4.0.0 documentation. http://rpython.readthedocs.org/en/latest/architecture.html, December 2015.

[10] Pypy's speed center. http://speed.pypy.org, December 2015.

[11] .net compiler platform ("roslyn") - documentation. https://roslyn.codeplex.com/wikipage?title=Overview&referringTitle=Documentation, December 2015.

[12] roslyn/cross-platform.md. https://github.com/dotnet/roslyn/blob/master/docs/infrastructure/cross-platform.md, December 2015.

[13] Roslyn performance (matt gertz) - the c# team. http://blogs.msdn.com/b/csharpfaq/archive/2014/01/15/roslyn-performance-matt-gertz.aspx, December 2015.