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、高級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。

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。

??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。 **??**章では、 **??**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。 5 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、 **??**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。 3 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 **??**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。 4 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。 **??**章では、 4 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。 7 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　プロジェクトのオープンソース化が与える影響

本章では、本研究の対象とするプログラミング言語であるSwiftが最近行ったオープンソース化の影響についてまとめることで、本研究の着目する課題について整理する。

2.1　オープンソースプロジェクトの特徴

オープンソースプロジェクトというプロジェクト形態についてどのようなプロジェクトでなければならないという制約などはないが、特に有名なオープンソースプロジェクトの定義がOpen Source Initiativeによって与えられている [4]。表2.1はそのオープンソースの定義をまとめたものである。

表 2.1: オープンソースの定義

1	そのソフトウェア全体を含むソフトウェアの販売・頒布を使用料を課すなどして制限しない
2	ソースコードを利用・改変しやすい形で使用料などを課さずに公開し、その再頒布を制限しない
3	そのソフトウェアの改変・派生および改変・派生したものの同ライセンスでの頒布ができる
4	ソースコードとパッチファイルを共に頒布でき、変更されたソフトウェアの頒布が明確に認められていれば、改変されたソースコードの頒布を制限してもよい
5	特定の個人や団体を差別しない
6	利用する分野を制限しない
7	再頒布されたものであっても、追加の規約などなしにそのプログラムに付随する権利が認められる
8	そのプログラムのライセンスの範囲内で使用・頒布される場合は、プログラムの一部分だけであっても同じ権利が認められる
9	そのソフトウェアと共に頒布されるソフトウェアに対する制限は行わない
10	特定の技術やインターフェース形式に限定した条件を課さない

オープンソースプロジェクトはそのソースコードが公開されているために誰でも変更したものを公開できるが、誰もが元々のプロジェクトと全く異なる場所で独自に拡張や修正をしているだけではソフトウェアの発展につながりづらい。そのため、一般的に多くのオープンソースプロジェクトではその開発プロセス自体をオープンにし、誰もが本体プロジェクトのソフトウェアを拡張・修正できるようになっている。

表 2.2: オープンソースプロジェクトにおける開発フロー

順序	手順	具体的な内容の例
1	任意のユーザからのソフトウェアの拡張や修正提案・要求	オープンなチケット管理ツールやメーリングリストなどでの拡張・変更提案・バグ報告と議論
2	特定の開発者による拡張や修正の実装と提案	パッチファイルの投稿や PullRequest の作成
3	他の開発者による実装へのレビュー	コーディングスタイルや機能の必要性に関する指摘や議論
4	修正・拡張の本体ソフトウェアへの統合	拡張・修正後のソフトウェアをプロジェクトの最新バージョンとして置き換え

多くのオープンソースプロジェクトで採用されている開発フローを表 2.2 に示した。この開発フローは表 2.1 の 4 番目に該当するバッチファイルによって修正版を公開するような、少し特殊な形式のソフトウェア以外におけるものである。このように、オープンソースプロジェクトにおける開発では特定組織内のみでの開発とは異なった開発フローとなるという点に注意が必要である。

また、Open Source Initiative の提唱する定義に合致するオープンソースプロジェクトであっても、そのプロジェクトの形式はそのプロジェクトの方針を決定する主体をどこに置くかによって異なるということを Raymond が自身のエッセイ [5] で提唱している。表 2.3 に示した「伽藍」と「バザール」という 2 つの形式である。

伽藍形式のオープンソースプロジェクトは Linux 以前の多くのオープンソースプロジェクトで取られていた手法であり、その開発方針は特定の集団によって制御される。それに対して、Linux に代表されるバザール形式のオープンソースプロジェクトでは開発者コミュニティの意思によって自然と必要な開発が進められ、プロジェクトの主体となっている個人や団体が開発の各フローにおいて特別に大きな影響力を持たないことを良しとする。

本研究で対象とする Swift はプロジェクトへの多くの開発者の参加を期待しており、プロジェクトで進められるべき拡張や修正などのほとんどはオープンなメーリングリストやチケッティングツールで決められていくことを望んでいる [6]。このことから、Swift はバザール形式でのプロジェクト展開を目指していると取れるため、本論文では以後バザール形式のオープンソースプロジェクトについてのみ言及する。

表 2.3: Raymond によるオープンソースプロジェクトの分類

形式	特徴
伽藍形式	• 中央集権的に開発方針などを指揮し、主な開発を担当する特定集団が存在する • 拡張や修正は開発方針に従って行われ、各リリースはよくチェックされる
バザール形式	• 中心的な開発者が指揮者として与える影響は小さく、どの拡張や修正が実施されるかはコミュニティ全体の動向に左右される • リリースは変更が加わるごとに更新され、極端に言えばバグを含んだまま行われる

最後に、表 2.1 に示したオープンソースプロジェクトの定義と表 2.3 に示したバザール形式の特徴から考えられるバザール形式のオープンソースプロジェクトのそうでないプロジェクトと比較した際の特徴を表 2.4 に整理した。

これらの特徴がバザール形式のオープンソースプロジェクトにおいて、他のプロジェクトにはないメリットやデメリットを提供していると見ることができる。

表 2.4: バザール形式のオープンソースプロジェクトの特徴

特徴を作る要因	特徴
様々な人がプログラムを参照する	プログラムのあらゆる箇所に対して任意のタイミングで拡張提案・バグ報告が行われる
不特定多数のプログラマが開発に携わる	• プログラムのあらゆる箇所に対して任意のタイミングで拡張・修正が行われる • 様々なコーディングスタイルの拡張・修正実装が提案される
ソフトウェアを利用したソフトウェアが開発される	ソフトウェアの部分的な利用が試みられる
ソフトウェアの技術中立性が高まる	様々なプラットフォームへの移植が試みられる

2.2 クローズドプロジェクトのオープンソース化による変化

オープンソースプロジェクトは開発の開始当初からオープンソースプロジェクトとして進められているものと、開発の初期段階ではクローズドだったものが途中からオープン

ソース化されるものとに分けられる。初めからオープンソースとなっているプロジェクトでは開発の初期段階でその開発フローなどを徐々に構築していくことが可能となるが、途中からオープンソース化されるプロジェクトではプロジェクトの開発フローや慣習をオープンソース化に合わせて変更していく必要がある。

本説では、 2.1 節で述べたオープンソースプロジェクトの特徴とクローズドプロジェクトの特徴を比較することでプロジェクトのオープンソース化に伴う変化について整理する。

バザール形式でのオープンソース化を行うと、特に中央管理的に判断されていた開発方針などの事項が開発者コミュニティによって民主的に判断されるようになるという変化が最も大きく、それにともなっていくつかの重要な変化が起こる。表 2.5 はその具体的な変化についてまとめたものである。

表 2.5: バザール形式でのオープンソース化に伴う変化

起こる変化	クローズドプロジェクトでの状況	オープンソースプロジェクトでの状況
拡張や修正の要求・実装が行われるかどうかを決定する要因の変化	リソースが限られているため、一定の開発方針に従って拡張や修正の優先順位が決定する	ある拡張や修正がそのコストを差し引いても充分な興味を引くものであれば、充分なリソースによって実装されて統合される
コーディングスタイルの流動化	開発者へのコーディングスタイルの徹底が可能なので同じスタイルを維持することができる	過去のコードとの統一性も踏まえられるものの、その時々で支持されたコーディングスタイルが用いられるため、スタイルが変動する
ソフトウェアの各部分におけるモジュール化	ソフトウェアの利用先などをすべて把握することが可能なのでモジュール化する部分を限定できる	あらゆる箇所で様々な形でソフトウェアが部分的にも使用されうるため、様々な箇所で高いモジュール性が求められる
マルチプラットフォーム化の進行	対象とするプラットフォームを制限することができる	各プラットフォームの熱心なユーザによってマルチプラットフォーム化が進められる

オープンソースプロジェクトではどの拡張や修正を行うかがコミュニティ内の開発者の興味によって決定されるため、コストと照らして充分な価値のある変更には充分なリソースが当てられて取り組まれると期待できるが、それはつまり、拡張や修正のコストが高ければ実装を行うハードルも高くなってしまうということである。また、コーディングスタイルもクローズドなプロジェクトと比較すると流動化する可能性が高い。この場合、もちろんより最適なスタイルが選択されていく場合もあるが、ソフトウェア全体で統一したス

タイルを徹底することはより難しくなっていくだろう。クローズドなプロジェクトではそのメリットが限定的になりがちであるモジュール化やマルチプラットフォーム化がそれを所望する熱心な開発者によって進められていく可能性が高いという点は、オープンソースプロジェクトにおけるポジティブな変化と捉えていいだろう。特にマルチプラットフォーム化は、各プラットフォームの熱心なユーザにとってはその対象ソフトウェアを使用するための必要不可欠な第 1 歩となるため、特に活発化する可能性が高いとかんがえられる。

ここに挙げた中で最も注目すべき変化は表 2.5 中 1 番上の拡張・修正のコストがある機能を実装するか否かを判断する上で重要な変数となるという点である。プロジェクトをより活発化するためには、開発方針を明確化したりより多くのリソースを集められるように注力するのではなく、少しでもソフトウェアの拡張・修正コストを下げる必要が出てくるのである。

2.3 拡張・修正のコストと可読性

特に実用に用いられる大規模なソフトウェアにおいて、それを拡張・修正するためのコストがソフトウェアのソースコードの複雑性に大きな影響を受けるということは Banker らによる調査などによって示されている [7] [8]。また、ソフトウェアの持つ特徴だけでなく、そのソフトウェアを拡張・修正しようとする開発者の知識にもそのコストは左右される [1]。そのため、本研究ではこれら 2 つの要因を複合したものを可読性とし、それらには表 2.6 に示すような関係性があることを前提とする。なお、ここで言及したソフトウェアの複雑性については、その要因から表 2.7 に示すように 3 つの側面に分解することができる [9]。

すなわち、例えばソフトウェアで使用されている手法をうまく改善すると、ソフトウェアの複雑性を改善することができ、結果としてソフトウェアの可読性の向上ひいては拡張・修正するためのコストの減少を達成することができる。

表 2.6: 可読性を決定する要因と可読性との関係

可読性を決定する要因	可読性との関係
ソフトウェアの複雑性	複雑性が高いと可読性が下がる
ソフトウェアの対象とする問題や使用している手法に対するプログラム読者の知識	知識が多いと可読性が上がる

2.4 本研究が着目する課題

本研究では、プログラミング言語 Swift がそのオープンソース化によって 2.2 節および 2.3 節で示したようにソフトウェアの拡張や修正のコストを下げるためにも可読性を向

表 2.7: ソフトウェアの複雑性に影響する要因

影響を与える要因	影響の大きさ	改善できる可能性
ソフトウェアが対象とする問題	大きい	とても低い
ソフトウェアに使用するプログラミング言語・モデリング手法・設計手法	比較して大きい	比較して低い
ソフトウェアの開発に参加する開発者の技術や知識	比較して小さい	充分に高い

上させるべき状況にあるという課題に着目する。

2.3 節で見たように、この課題を解決するためには可読性に影響を与える要因について何らかの改善を行う必要があるが、現在の Swift では未だ可読性の維持・向上のための明確な施策は打ち出されていない。また、現在のソースコードの可読性は丁寧なコードレビューを重ねることによって保たれているが、これはプロジェクトが大きくなることでレビュアーが多様化することによって成り立たなくなったり、プロジェクトの拡大にともなってそのレビュー自体が開発スピードのボトルネックとなる可能性があるため、好ましい状況であるとはいえない。

そこで、本研究はオープンソース化した Swift においては今まで以上にそのソースコードの可読性を重視し、少しでも可読性を向上していく必要があるという考えのもとに、この課題に取り組む。

第3章　ソースコード可読性の向上方法と評価手法

本章では、本研究が目的とするソフトウェアのソースコード可読性の向上を達成するためのアプローチについて整理し、そのアプローチによって可読性が向上したかどうかを検証するための既存手法についてまとめる。

3.1　可読性を向上するためのアプローチ

* 可読性を向上するためには:open-source:readability で述べた可読性を決定する各要因の内のどれかを改善する必要がある* ただし、一般的にソースコードの可読性が問題になるようなフェーズではソフトウェアの対象とする問題は充分に明確化されているであろうことから、問題を変更するようなアプローチについては対象としない 1. ソフトウェアの複雑性を下げる手法 1. 使用するプログラミング言語を理解しやすいものにする* より高級な言語を使用するようにするなど 2. 使用するモデリング手法を理解しやすいものにする* 問題の捉え方を変え、より単純なモデルへ落としこむようにするなど 3. 使用する設計手法を理解しやすいものにする* 使用するデータ構造やアルゴリズムをより単純なものにするなど 4. 参加する開発者がより理解しやすいプログラムを書くようにする* コーディング規約によってプログラムの書き方を制限する* コメントやテストの記述量に対して制限を設けるなど 2. 拡張・修正を行う開発者の知識を合致させる手法 1. 拡張・修正を行う開発者がよりその対象とする問題についてよく知っているようにする* 対象とする問題をより広く認知されている別の問題に置き換える* 問題について整理したドキュメントを用意するなど 2. 拡張・修正を行う開発者がよりソフトウェアで用いられている手法についてよく知っているようにする* より広く知られたプログラミング言語や手法を採用する* 手法について整理したドキュメントを用意するなど 3. ソフトウェアで使用する手法を減らす* 使用するプログラミング言語やモデリング手法、設計手法、ツールの種類を減らすなど* これらのアプローチのより具体的な方法や各方法の実現可能性は対象とする問題によって変化する* :open-source:readability で述べたように開発者よりもその手法の改善を行う方が複雑性を下げる効果は高い可能性が高い

3.2 ソースコード可読性の評価手法

* 拡張・修正を行う開発者の知識を合致させるような手法ではその具体的な方法に合わせて定性的な手法も含む評価手法を考える必要がある* あるいは、単に使用する手法の種類を減らす場合などは確かにその手法が使われておらず、新しい手法の導入などが行われていないことだけを確認すれば、必要な知識が減っていることは明らかである* ソフトウェアの複雑性についてはよく知られた 3 種類の古典的な手法がある (http://www3.nd.edu/ rsanteli/pate-sp12/readings/SurveyComplexity.pdf) * 各評価手法の概要とその特徴 1. Line of Code (LOC) * ソフトウェアの実行可能なソースコードの行数とバグの密度の間にある相関関係から行数を指標にソフトウェアの複雑性を比較する* 各行の複雑性の差は無視している* プログラムの構造は一切勘案されない* 対象とするプログラムや開発者の技術などの要因によって最もバグ密度の低いソースコードの行数は変化する 2. Halstead Complexity Metrics (HCM) * ソースコード中の演算子と被演算子の種類および数に基づく指標によってソフトウェアの複雑性を比較する* データフローに基づく複雑性のみで制御フローに基づく複雑性は勘案されないため、分岐が 1 つもないプログラムと無限の分岐が存在するプログラムでも指標は同じ値となりうる 3. Cyclomatic Complexity Metric (CCM) * ソースコード中の線形独立な経路の数を指標としてソフトウェアの複雑性を比較する* 制御フローに基づく複雑性のみでデータフローに基づく複雑性は勘案されないため、経路の数が同じであれば 1 文だけのプログラムと無限の文を持つプログラムでも指標は同じ値となり得る* これらの手法は全てソースコードの可読性と経験則的に繋がりがあるというだけ* 各値を様々な角度から考察して初めて意味を成す* また、関連するデータの数やそのデータとのトランザクションの数を元にした指標としてファンクションポイント法があるが、これは:open-source:readability で述べた複雑性の側面の内対象とする問題と手法の一部分しか対象としない

3.3 本研究のアプローチ

* :open-source:issue で述べたとおり、Swift コンパイラではその対象とする問題の大きさから既に可読性が他のソフトウェアと比較して低いと思われる* :open-source:change で述べたようにバザール形式のオープンソースプロジェクトでは方針の民主的な決定が大きな違いを生む* 普段の開発の流れに対して大きく介入するようなアプローチではオープンソース化によるメリットを潰してしまう可能性がある* :readability:approach で述べたようなコーディング規約などによる制約を課すことは避けたい* また、開発者に対する改善よりも手法などに対する改善の方が効果が高いと期待できる* :introduction:background で述べたように、他のコンパイラでよく行われている手法としてコンパイラを自分自身で書き直す Self-host 化がある* Self-host 化を行えばコンパイラの記述言語に関する知識をコンパイル対象言語に関する知識と統一化できるため、ソフトウェアで使用する手法を減らすことになる* 一般に後発の言語のほうが高級であることが多いため、Swift で記述されたコンパイラの方が現行の C++で書かれたものよりも優れた可読性を持っていると期待できる* Self-host 化した Swift コンパイラを実装し、:readability:evaluation で述べた古

典的な3つの手法のどれかでそのコンパイラの複雑性を現行のコンパイラと比較することでアプローチの正当性を考察する

第4章　提案手法において発生しうる副作用

本研究ではコンパイラの可読性を向上する目的で Self-host 化を用いるが、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 Self-host 化をした上で自分自身によるコンパイルを可能とする 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。

4.1　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 化だけでなく Bootstrap まで行った言語のみ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、線引が難しいためである。

4.1.1　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 [10] 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、 [10] 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 [11]。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 [12] 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。

4.1.2 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 [13] 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 [14]。

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 [15]。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。

4.1.3 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 [16]。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 [17]。

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 [18] 内で述べている。

4.2 Swift における Self-host 化の副作用

表 4.1 に 4.1 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 4.1: 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、 4.1.1 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 4.1.2 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。

4.1.3 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 4.2 に 4.1 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 4.2: 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、 4.1.3 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、 4.1.1 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、 4.1.1 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 4.1.3 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 4.1.1 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、 4.1.3 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。

4.3 性能の差の検証方法

第5章　Self-host化したSwiftコンパイラ「TreeSwift」の設計と実装

5.1　可読性の比較が可能な箇所の絞り込み

5.1.1　Swift コンパイラの構造

* Swift コンパイラはおおまかに 5 つのステップに分けられる* 構文解析* 意味解析* SIL(Swift Intermediate Language) 生成* 詳細解析* IR(LLVM-IR) 生成

5.1.2　可読性の比較が可能な箇所

* :open-source:readability で述べたように、扱う問題や手法が異なるとそのプログラムの複雑性は大きく変化する* 参照解決や型推論、型検査を行う意味解析と最適化などを行う詳細解析は同じ問題を対象として同じ手段を取っているということを確認するのが難しい* Self-host 化し、Swift で記述した場合は LLVM の C++ API を直接使用できないため、C API を使用することとなる* IR 生成部分は異なる API を使用することによる影響が大きく出ることが考えられる* そのため、:readability:evaluation で上げたような手法で比較できそうなのは構文解析部分と SIL 生成部分* ただ、SIL 生成部分は SIL の仕様がまとまって公開される前だったため、設計に含めることが難しかった* そのため、本研究では構文解析部分のみを設計対象とする

5.2　一般的なコンパイラの構文解析器の構造

:implementation:comperable:part でも言及したとおり、同じ問題を対象と出来ているかどうか、同じ手法を用いているかは比較の上で重要な注意点

5.2.1　構文解析器の要素

* 構文解析器は大きく分けて 3 つの機能からなる 1. ソースファイルの文字列を意味のある文字単位 (字句) に分ける字句解析 2. 字句を意味のある文章単位ごとに処理して AST を組み上げる構文解析 3. 構文の間違いを報告するエラー処理

5.2.2 構文解析器の扱う問題

* 大きく分けて 2 つの問題を解決するために構文解析器は存在する 1. 構文処理* 正しい構文で記述されたソースコードを正しく解析すること 2. エラー検知* 間違った構文で記述されたソースコードの間違いを指摘すること* この 2 つの面の少なくとも大きな点については一致している状態で比較する必要がある

5.2.3 構文解析器の手法

* エラー検知については構文解析器だけでなく、後の意味解析や詳細解析でも行うため、実装によってどの段階でどのエラーを検知するか、などは大きく異なる場合がある* 構文処理については文法に応じて 3 種類の手法がある (dragonbook) 1. 汎用構文解析 2. 下向き構文解析 3. 上向き構文解析

5.3 TreeSwift の設計

* 構文解析器のみに絞る* 構文処理は網羅された仕様書が作成できるので、実際に解析ができるかどうかを網羅的に調べることができるので、できるだけ現行のものと同じになるように実装する* エラー検知はどれだけ網羅できているかを判断しづらいため、比較の対象から外す* 同じ手法を徹底することは難しいが、構文解析器全体では同じ LL(k) 方式を用いる

5.4 実装の概要

* 字句解析には各リテラルの解析を行う決定性有限オートマトンを複数用いている* モジュール定義にはテキストファイルを使用する* エラー回復は実装されていない

第6章　評価

6.1　ソースコード可読性の向上評価

6.1.1　評価手法

* LOC をベースとする* エラー処理の違いが大きく出る可能性がある* 実行可能な行からさらに実行した行のみを取り出して比較する* 実行可能な行の内どれだけを対象にできているかを算出する* 分岐を決定する文を除いた場合の行数も算出する* モジュール定義ファイルのパースには大きな差がでる可能性がある* モジュールのパースやコンパイラオプションの処理などは対象に含めない

6.1.2　計測

6.1.3　考察

6.2　Self-host 化に伴う副作用の評価

6.2.1　評価手法

* 速度とメモリ使用量を計測する

6.2.2　計測

6.2.3　考察

第7章　結論

本章では、本研究の結論と今後の展望を示す。

7.1　本研究の結論

* すべての指標において可読性は Self-host 化したものが上回っていた* Self-host 化によって可読性の向上が期待できる* この可読性の向上によって実際の作業量がどれほど変化するかは判らない* Halstead の提唱する式では労力などに換算できるとされているが、批判的な意見が多い* Self-host 化によって性能については低下する可能性がある* 本研究も Swift がバザール形式のオープンソースとなったことによる成果の 1 つであり、実際に Self-host 化を進めるかどうかを判断するためにはもっと様々な角度から吟味される必要がある

7.2　今後の展望

7.2.1　構文解析器以外の比較

* SIL 解析は比較できる可能性がある* 意味解析や詳細解析についても機能の同じ部分だけを慎重に選べば比較できそう

7.2.2　継続的な比較

* アップデートの多い新しい言語なので今後の変更で値が変化する可能性がある* ソフトウェア・メトリクスに対してはソフトウェア完成後の比較しかできない点が弱いとされている* 同じ機能部分だけを取り出して比較し続けることで修正・拡張による可読性の変化を見ることはできそう* 性能については常に変化するものなので、より継続的に見る意味がある

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 open source definition. http://opensource.org/docs/osd, January 2016.

[5] Eric S. Raymond and 浩生 山形 (訳). 伽藍とバザール (the cathedral and the bazaar). http://www.tlug.jp/docs/cathedral-bazaar/cathedral-paper-jp.html#toc2, January 2016.

[6] Welcome to swift.org. https://swift.org, December 2015.

[7] Rajiv D. Banker, Srikant M. Datar, Chris F. Kemerer, and Dani Zweig. Software complexity and maintenance costs, 1993.

[8] Rajiv D. Banker, Gordon B. Davis, and Sandra A. Slaughter. Software development practices, software complexity, and software maintenance performance: A field study, 1998.

[9] Sheng Yu and Shijie Zhou. A survey on metric of software complexity, 1982.

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

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

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

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

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

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

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

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

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