卒業論文 2015 年度 (平成 27 年)

Self-host 化による Swift コンパイラのソースコード可読性の向上

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)

Self-host 化による Swift コンパイラのソースコード可読性の向上

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、ソフトウェアの複雑性を低くできる可能性が高いからである。しかし、Swift においては現行のコンパイラを記述している C++も様々な特徴を持つ高級汎用言語であるため、その複雑性における優位点は明らかではない。

そこで本研究では、Swift で記述された Swift コンパイラの構文解析器を実装し、現行の Swift コンパイラの構文解析器とそのソースコードの行数を幾つかの部分に分けて比較することで、Self-host 化によって Swift コンパイラの可読性が向上することを検証した。ただし、Swift の Self-host 化による可読性の向上には Swift 独自の構文が関わっているため、他の言語においては既に高級言語で記述されているコンパイラを Self-host 化したとしても、同じ結果が得られるとは限らない。

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

Improvement of the Readability of Swift Compiler's Source Code by Self-hosting

When a software project changes from closed to open source, as both the number of joining developers & the extension/modification of program increase, the readability of the source code becomes an important factor for the success of the project.

In December 2015, the Swift programming language project, which is led by Apple, made its souce code public and was faced to face with such a change. In the current process, the readability of Swift compiler is maintained just by the effort of habitually following the coding style of existing code and its strict review. Therefore, when the new code grows as a fraction of the software, its readability may become out of control.

On the other hand, there is a technique called self-hosting, which is adopted in many general-purpose high-level programming languages. Self-hosting is a technique for writing the compiler in its own source programming language. Although there are multiple reasons to adopt the self-hosting style, the most attractive advantage is that developers can reduce their effort for understanding the software. When the compiler is self-hosted, its developers are not required to learn another language other than the source language. Furthermore, as the source language is newer than its compiler's description language in the early years, the self-hosting has an effect to change the compiler's source code less complex with the new and expressive grammars of source language.

In this research, I implemented a new Swift compiler which is written in Swift and compare its parser with the parser of Apple's Swift compiler in order to verify the readability enhancement of Self-hosting. As the result of this research, we could confirm the positive effect of self-hosting on the readability. But at the same time, as the effect depends on some characteristics of the grammar of Swift, we are considering that some other languages whose compilers are written in other high-level languages may not benefit from the effect.

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。

TODO: 表 *1.1* の情報に参考文献を付ける

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、我々が Swift のオープンソース化以前より実装していた Swift で記述した Swift コンパイラを用い、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。

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。

TODO: 内容を現在の章立てに合わせる

??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。 6 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。 8 章では本研究の結論と今後の展望についてまとめる。

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	×	-

第2章　オープンソース化によってSwiftが抱える課題

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、特に多くのプロジェクトが従っているオープンソースプロジェクトの持つべき性質がOpen Source Initiativeによって定義されている [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

2.3 節ではソフトウェアのソースコード可読性を向上するために表 2.6 に示した要因について適切な改善を行う必要があることと、その内の複雑性については表 2.7 に示した要因に分解できることを説明した。本説ではそのために考えられるアプローチについて整理するが、そのより具体的な手法が適用可能かどうかはアプローチを適用するソフトウェアの対象とする問題に依存するため、ここでは手法の例を挙げるに留める。

表 3.1 がその考えられるアプローチを列挙したものである。なお、表 2.7 でもソフトウェアが対象とする問題の変更などによる複雑性の解消が難しいことについては言及していたが、そもそも本研究で対象としているような拡張・修正が主体となるような開発フェーズではその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場合がほとんどであるため、問題を変更するようなアプローチについては取り上げていない。

2.3 節で述べたようにソフトウェアの複雑性についての改善を行う場合はアプローチする要因によって複雑性に対する影響が異なるため、表 3.1 に挙げたソフトウェアの複雑性に対するアプローチの中では上半分にあるものの方が下半分のものよりも結果的に可読性の向上に対してより高い効果を得られると期待できる。

3.2　ソースコード可読性の評価手法

3.1 節で述べたような可読性を向上するためのアプローチを取ったとしても、それが必ずしも可読性の向上に繋がるとは限らないため、確かに可読性が向上したことを確認するための評価手法が必要となる。

表 3.1 に挙げた中でも下半分の拡張・修正を行う開発者の知識を合致させるようなアプローチでは、場合によっては定量的な数値に落としこむことが難しいため、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。それに対して、ソフトウェアの複雑性については定量的に評価するための手法が多く考案されている。本節で

表 3.1: 可読性を向上するためのアプローチ

アプローチする可読性の要因	アプローチ	具体例
ソフトウェアの複雑性	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
	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
	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
	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
拡張・修正を行う開発者の知識	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
	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
	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

は、その中でも特によく取り上げられる 4 種類の手法について整理する [9] [10]。なお、これら 4 つの手法は全て、その値が経験則的にソフトウェアの複雑性と結びついているとされているだけであるため、実際に使用する際にはその結果について慎重に考察する必要がある。

LOC (Line of Code)

LOC では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などと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算出する。

LOC は最も古典的でありながらもその扱いの容易さからよく用いられているが、特にプログラムの構造がその結果に一切反映されていないという点に注意する必要がある。

FP (Function Point)

FP は 1979 年に Allan J. Albrecht によって提案された。ソフトウェアへ入出力されるデータとその入出力操作を列挙して重み付けを行い、その総和を指標にソフトウェアの複雑性を算出する。

FP は特に商用ソフトウェアの工数見積などの目的でよく使用されているが、算出ソフトウェア内のデータ処理による複雑性は経験則的に決定される重み付けのみによって反映される。

HCM (Halstead Complexity Metrics)

HCM は 1977 年に Maurice H. Halstead によって提案された。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算出する。

HCM ではデータフローに基づく複雑性を算出することができるとされているが、制御フローに基づく複雑性は一切勘案されていないため、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など注意が必要である。

CCM (Cyclomatic Complexity Metric)

CCM は 1976 年に Thomas J. McCabe によって提案された。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算出する。

CCM では制御フローに基づく複雑性を算出することができるとされているが、データフローに基づく複雑性は一切勘案されないため、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うるなど注意が必要である。

3.3 本研究のアプローチ

2.2 節で述べたように、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他の形式のプロジェクトとの違いを生む主要因となっている。そのため、その民主的な開発を促すために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をとってしまうと、オープンソース化によるメリットを潰してしまい、結果として逆効果となってしまう可能性すら考えられる。その視点からすると、 3.1 節で例として述べたようなコーディング規約などによって制約を課すようなアプローチではその期待できる効果の

薄さの割に逆効果となるリスクまで背負うこととなり、あまり良いアプローチであるとは考えられない。

そこで、本研究では 1.1 節でも他のコンパイラでよく用いられている手法として挙げた Self-host 化を可読性を向上するためのアプローチとして使用する。

Self-host 化は表 2.7 で示したソフトウェアの複雑性に影響する要因の内でも比較して大きな影響を持つ、使用するプログラミング言語の変更であるため、ソフトウェアの複雑性を大きく解消できる可能性がある。さらに、Self-host 化を行うと、コンパイラの記述言語とコンパイル対象言語が同じものになるため、コンパイラを記述している言語のために特別な知識を要さず、結果としてソフトウェアで使用する手法を減らすことができる。これは表 3.1 で一番下に挙げているアプローチに他ならない。つまり、Self-host 化ではソフトウェアの複雑性とソフトウェア開発者に要求される知識の両面から可読性の向上に寄与すると期待できるのである。

ただし、 3.2 節で述べたとおり、実際に Self-host 化によって Swift コンパイラの複雑性が解消されるかどうかを決定するには具体的にその複雑性の比較を行う必要がある。特に Swift コンパイラにおいては現行の実装が既に高級言語である C++によって記述されているため、これを Swift で書き直すことによって充分な可読性の向上を得られるかどうかをそれら言語の仕様など飲みから判断することは難しい。

そのため、本研究では Self-host 化した Swift コンパイラを実装し、 3.2 節で述べた手法のどれかを用いることでそのコンパイラの複雑性を現行のコンパイラと比較し、このアプローチの正当性を評価・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。

また、Bootstrap においては新しいバージョンのコンパイラのソースコードをどのようにしてまだ存在していないその新しいバージョンのコンパイラ自身でコンパイルするか、という Bootstrap における極めて一般的な問題が存在している。そのため本説では、各事例について Bootstrap を行う目的と Bootstrap を行った結果に加え、どのようにして Bootstrap を行ったかについてもまとめることで、Swift が将来的に Bootstrap まで行った場合に発生しうる問題についても考察するための情報を提供する。

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 [11] 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、 [11] 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 [12]。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 [13] 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 [14] 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 [15]。

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 [16]。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 [17]。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 [18]。

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 [19] 内で述べている。

4.2 Swift における Self-host 化の副作用

4.2.1 可読性の向上以外の副次的なメリット

まず、表 4.1 に 4.1 節で述べた事例から 3.3 節で述べた可読性の向上に繋がる点以外の Bootstrap における以外の利点をまとめ、Swift が Bootstrap を行った場合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
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	×
コンパイラの各フローをライブラリとして提供できる	×

4.1.1 節や 4.1.2 で見たように、各事例でも記述やデバッグが容易になったりより柔軟になり拡張性が高くなったりしているという評価があることは本研究のアプローチにおいても可読性を向上できると期待できる大きな根拠となる。また、 4.1.2 節では Self-host 化を行っても必ずしも必要となるプログラミング言語の知識が減るとは限らないことが分かったが、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て差し支えない。

一方で、 4.1.1 節で上がっていたような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 と現行の Swift コンパイラ記述言語である 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未だ不十分なため、C++の方に分配が上がる可能性が高い。

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。

4.2.2 Self-host 化によって想定されるデメリット

次に、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 が Bootstrap を行った場合にそれらの課題が問題となるかどうかをまとめた。なお、ここでは Self-host 化に限らず Bootstrap までを行った場合にのみ発生する課題についても考察しているが、これはコンパイラの基幹機能を Self-host 化したコンパイラでは一般的に Boostrap まで行っており、Swift でもそうなることが予想されるため、意図的に Self-host 化だけを行った場合の課題にのみ絞り込まないようにしているためである。

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。

他の低級言語への依存は、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。

最後に、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。

本研究では可読性の向上のみを目的としているためにこれらのデメリットに対する評価などは行わないが、実際に Self-host 化を行う際にはこうしたデメリットを考慮し、目的に即した手段を取れているかどうかを充分に確認する必要があるだろう。

第5章　Swiftコンパイラの構成と各部分における複雑性の比較難易度

コンパイラは 4.1 節で見た事例だけからでも分かるように、同じプログラミング言語を対象としたものでもその設計手法やそもそも重視する問題が異なる場合がある。かつ、2.3 節で挙げたように、ソフトウェアが対象とする問題やソフトウェアに使用される基幹的な手法はそのソフトウェアの複雑性を決定する上で重要な変数となっている。そのため、3.2 節で述べたような手法を用いて複雑性を計測・比較して得た変化が確かに Self-host 化によるものであると考えるためには、対象とする問題と提案手法以外の基幹的な設計手法が比較対象において同一である必要がある。また、複雑性の比較を行う際にはどの手法を用いるにしても、巨大なソフトウェアであるコンパイラの全体について一度に比較してしまうと、各部分における複雑性の差が打ち消し合い、Self-host 化が与えている要因に対する考察を行う上で評価の結果が扱いづらくなってしまうことが考えられる。

そこで本章では、現行の Swift コンパイラをその機能によっていくつかの部分に切り分けた上で、各部分における目的と使用されている手法についてまとめ、その部分を Self-host 化した場合でも同じ目的のもとに同様の手法を取ることが可能かどうか考察することで、実際に本研究で実装を行って比較を行う部分を選択するための判断材料をつくる。

5.1　Swiftコンパイラの構成

Swift コンパイラの大まかな構成を図 5.1 に示した。この図は Swift の中心的開発者である Groff と Lattner による Swift コンパイラについての説明資料 [20] 中の図を訳し簡略化したものである。Swift コンパイラは図中矩形で表されている 5 つの処理によって図中円形で表されている 4 つのデータ形式を順に読み込み・生成する。データ形式から処理を貫いて他のデータ形式まで伸びる矢線はデータ形式がどの処理によってどの順に生成されるかを表している。

この図が表現しているのは、Swift コンパイラのデータ構造を媒介とした各処理ステップの独立性である。Swift コンパイラでは各処理ステップで扱うデータ形式を原則として

図 5.1: Swift コンパイラの構成

Swift、AST、SIL、LLVM-IR のいずれかに限定することで処理ステップ間の結合を疎にしている。

以下では Swift コンパイラで行われる各処理ステップについて、それぞれの目的とそのために使用されている手法について概説する。なお、本論文内の Swift コンパイラについての説明は swift-2.2-SNAPSHOT-2015-12-31-a のリリースにおけるソフトウェアの内容に基づいている。

5.1.1 構文解析

構文解析は次の 2 つの目的のために行われる。

1. ただの文字列である Swift プログラムを解析して構造を持った抽象構文木 (AST) に変換する
2. プログラムの構文的な誤りを検知して報告する

1 つ目の目的を達成するために、Swift では LL(k) クラスの再帰下降構文解析を手作業で構築して用いている。この手法についての理解を深めるために、以下では近代的なプログラミング言語に用いられるもう 1 つの構文解析手法と共にこの構文解析手法について解説する [21]。

下向き構文解析

下向き構文解析では、プログラム全体を表現する非終端記号の文法から順により詳細な構成要素に深さ優先で分解していくことによってプログラムを解析する。Swift で用いられている LL(k) クラスの再帰下降構文解析は下向き構文解析の 1 種である。

TODO: dragonbook P.234 のような図を追加

再帰下降構文解析は関数によって表現された手続きの集合として構成され、最初の文法を表現する手続きからより詳細な構成要素のための文法を表現する手続きを再帰的に呼び出すことで処理が行われる。そのため、再帰下降構文解析では文法と実装が明確な対応関係を持っている事が多く、手作業で構文解析器を作成することが比較的簡単である。

上向き構文解析

上向き構文解析では、プログラムの字句を先頭から順に読み、具体的な字句をより抽象的な非終端記号へと還元していくことで当てはまる文法を決定していくことでプログラムを解析する。

TODO: dragonbook P.252 のような図を追加

上向き構文解析において最もよく用いられているのは LR(1) クラスの構文解析器である。LR(1) クラスの解析器は多くのプログラミング言語で用いられる文法の殆どを解析できる上、Yacc に代表されるような古くからよく用いられている構文解析器の自動生成ソフトウェアが存在する。ただし、逆に LR(1) クラスの解析器を手作業で記述するのは容易ではない。なお、LL(k) クラスの解析器についても構文解析器を自動生成するソフトウェアは存在する [22] が、LR(1) クラスの解析器ほど頻繁に使用されてはいない。

このように構文解析器には複数の設計手法が存在しており、その内の幾つかの手法においては同等の解析能力を保持しているが、どの手法を採用するかによって構文解析器のプログラムは大きく変化する。また、構文解析器の 2 つ目の目的であるエラー検知については、その構文解析器が自動生成されるような場合には自動生成されたプログラムの用意するインターフェースによって検知タイミングや検知の難易度が変化する場合がある。そのため、構文解析器について複雑性を比較する場合には、少なくとも比較を行うコンパイラについても現行の Swift コンパイラと同じように LL(k) クラスの再帰下降構文解析を手作業で構築して利用されている必要があると考えられる。

これに加えて、Swift では Swift で作成されたライブラリを読み込むために C や C++ におけるヘッダファイルと同じような役割を担う独自形式のモジュールファイルをライブラリ作成時に自動生成しており、構文解析器はこれを解析する役割も担っている。モジュールファイルの解析は通常のプログラムの解析とは別の関数をエントリ・ポイントとして実行されるために分離して考えることも可能だが、モジュールファイルの解析についても比較を行う場合は、解析にかかる作業量が大きく異なることを避けるために、同様の形式のモジュールファイルを使用している必要があると考えられる。

5.1.2 意味解析

意味解析の目的は次のとおりである。

1. AST 中の変数や関数、型を互いに結びつけ、プログラム全体の整合性を確認する
2. 開発者によって省略されている情報を補足する

1 つ目の目的は主に参照解決と型検査、2 つ目の目的は主に型推論を行うことによって達成される。以下では意味解析の中で行われるこれら 3 つの処理について Swift で使用されている手法と複雑性の比較を行うために統一されるべき設計手法について述べる。

参照解決

プログラム中で使用される変数や関数が正確にどこで宣言された変数や関数を指しているかを知ることができるタイミングはその変数や関数がどんな種類であるかに依存する。

例えば、プログラム 5.1 のような Swift プログラムでは 2 行目で参照される変数 x が直前の行で宣言されているため、構文解析を行いながら宣言された変数の情報を蓄積しておけば、最短で 2 行目の x を読んだ直後に変数 x の参照を解決できる。それに対してプログラム 5.2 のように少し複雑化すると、例えばプログラム中 5 行目の変数 s.x の参照を解決するためにはいくつかのステップを踏む必要が出てくる。このプログラムで変数 s.x の参照を解決するためには、まず変数 s の参照を解決し、その型が明示されていないためにこれを推論し、クラス Sample のメンバリストから変数 x の宣言を探し出すという長大なステップを踏まなければならない。

プログラム 5.1: 直ちに変数解決が可能である例

```
1 let x: Int = 1
2 print(x)
```

プログラム 5.2: 変数解決までに複数の処理が必要となる例

```
1 class Sample {
2     let x: Int = 1
3 }
4 let s = Sample()
5 print(s.x)
```

このように、参照解決のタイミングはプログラミング言語の仕様によって制限されるが、特に最短のタイミングで解決を行わないといけないということもないため、実装時の簡潔性などを勘案した上で各実装ごとに決定される。実際、現行の Swift コンパイラでも参照解決のタイミングはその変数や関数の参照のされ方によってまちまちである。

そのため、比較対象のコンパイラにおいても全く同じタイミングで同じように参照解決が行われるように設計することは容易ではない。ただし、そのタイミングが異なったとしても比較対象のコンパイラが同等の参照解決能力を持っていることを示すことは不可能ではない。その場合は、型検査と型推論も含む意味解析部分全体をまとめて比較対象とした上で、意味解析全体を経て最終的に同じように参照解決がなされていることを確認するという形になる。

型検査・型推論

Swift は Haskell や ML などの関数型言語と同様の非常に強力な型システムを備えており、その型検査と型推論は Hindley と Milner による型推論アルゴリズムを拡張したアルゴリズムによって行われている [23] [24]。

Hindley と Milner による型推論アルゴリズム自体は非常によく知られている上に様々な言語で採用されているが、現行の Swift コンパイラで行われている独自の拡張は他の型推論を用いる関数型言語と比較して Swift に特徴的なパラダイムであるオブジェクト指向プログラミングや型パラメータ多相を実現するためのものである。

そのため、型検査と型推論についてコンパイラの比較を行う場合には、Hindley と Milner による型推論アルゴリズムを元にしたアルゴリズムを用いた設計にした上で、Swift が独自に拡張している型についても同等の型推論・型検査能力を備えていることを示す必要がある。ただし、参照解決の項で述べたように、特定の変数や関数の参照解決を行うために型推論が必要となる場合があるため、型検査や型推論と参照解決を分離して評価することは難しく、どちらかを比較する際には両方を同時に比較せざるを得ないという点には注意する必要がある。

5.1.3 SIL 生成

SIL 生成は、AST を分析してより抽象度の低い独自中間言語である Swift Intermediate Language(SIL) に変換することで最適化などの準備を行うためのステップである。SIL は LLVM-IR をベースとしてループやエラー処理、オブジェクト指向プログラミングのためのクラスの概念などを拡張し、プログラムの意味に基づく詳細なエラー検出などを行い易くした言語である [20]。

SIL 生成のステップは特に一般的な手法などに基づいた処理が行われているわけではなく、完全に AST と SIL の設計に依存した処理となる。そのため、SIL 生成部分を比較するためには比較するコンパイラにおける AST の設計を現行の Swift コンパイラと合わせる必要がある。ただし実際には、AST の構造はプログラミング言語の仕様と採用する構文解析手法に依存して決定するため、構文解析手法についても同一のものを選択する必要があるだろう。

5.1.4 詳細解析

詳細解析は、SIL を分析して AST のような抽象度の高い状態や LLVM-IR のような抽象度の低い状態では処理しづらい最適化やエラー検出を行うためのステップである。

現行の Swift コンパイラでは SIL が包含するコールグラフやクラス階層といった情報を元に最適化やエラー検出を行っており、それらは手法ごとにモジュール化されている。また、Swift ではメモリ管理の手法として各オブジェクトの作成が要求された際に自動的にメモリを割り当て、不要になったタイミングを検知してメモリを開放する Automatic Reference Counting 方式を用いており [25]、そのための処理の挿入などもこの詳細解析で行っている。

詳細解析は複数のアルゴリズムの組み合わせによって構成されているため、比較するコンパイラでもそれぞれの処理について同一のアルゴリズムを採用することで同一の手法を使用した詳細解析を構築することが可能である。

5.1.5 IR 生成

IR 生成は SIL を分析して LLVM-IR を生成することで、低抽象度での最適化や機械語生成を行うための準備を行うためのステップである。

LLVM は統一された中間言語である LLVM-IR に対する最適化や LLVM-IR から様々なプラットフォーム上で動作可能な機械語へのコンパイルを行うツール群であり [26]、現行のコンパイラはこの LLVM を用いることでコンパイラのバックエンドをフロントエンドから完全に分離している。

LLVM のツールセットは実行可能なソフトウェアや用意された API を通して使用されており、現行のコンパイラでは C++で記述された API を用いている。この API には他の言語向けのものも存在するが、LLVM のツールは主に C++で記述されているために C++の API が最も充実しており、現行の Swift コンパイラは C++の API のみで実装されている高度なインターフェースを利用している。そのため、C++以外の言語で実装された Swift コンパイラでは、現行のコンパイラと同じ手法で同じ LLVM-IR を生成するように構成することは困難である。

第6章　Self-host化したSwiftコンパイラ「TreeSwift」の実装

本章では、 5 章での考察を元に本研究で評価する Swift コンパイラの部分について説明し、その Self-host 化した実装について述べる。

なお、本研究で実装した Swift コンパイラには「TreeSwift」というコードネームが付与されている [27]。本論文でも現行の Apple 社による Swift コンパイラと簡単に区別するため、本研究で実装した Swift コンパイラを指して TreeSwift と記述する。

6.1　評価を行う Swift コンパイラの部分

5 章のはじめに述べたように、ソースコードの複雑性を巨大なソフトウェアであるコンパイラ全体で比較すると、部分部分の差が打ち消しあった結果的な値しか見えず、複雑性に差をつくることとなった原因を究明することも容易ではなくなる。本研究では、 5.1 節で述べた Swift コンパイラの部分の内、構文解析器についてのみ注目し、これを実装することで比較評価を行う。 5.1 節で述べたように、各構成要素はそれぞれに比較が困難になる可能性のある箇所があるが、構文解析器はコンパイラ内で始めに実行される箇所であるために他の処理から分離しやすく、かつあるプログラムをコンパイルするために必要な箇所のみを抜き出すことで、更に小さい構成に分解できるためである。

6.2　実装の概要

本節では実装した Swift コンパイラ中の構文解析部分について、その実装を紹介する。なお、TreeSwift では完全ではないものの、構文解析器以外の部分についても実装を行い、構文解析器が充分機能するものであることを確認している。

6.2.1　構文解析器の実装

構文解析器は Swift プログラムを構文を構成する字句に分解する字句解析、字句を構文にまとめ上げる構文解析、通常のプログラムとは異なるライブラリ用のモジュールファイルを解析するモジュール解析、およびそれらで発生するエラーを処理するエラー検出からなる。本節では TreeSwift の構文解析器におけるそれらの実装について説明する。

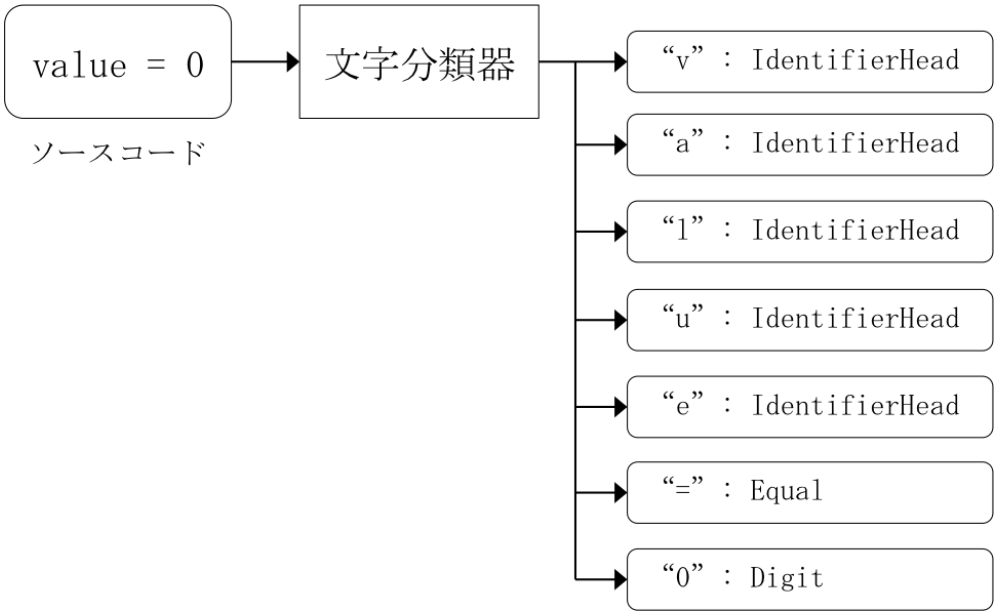

図 6.1: 文字分類器の仕組み

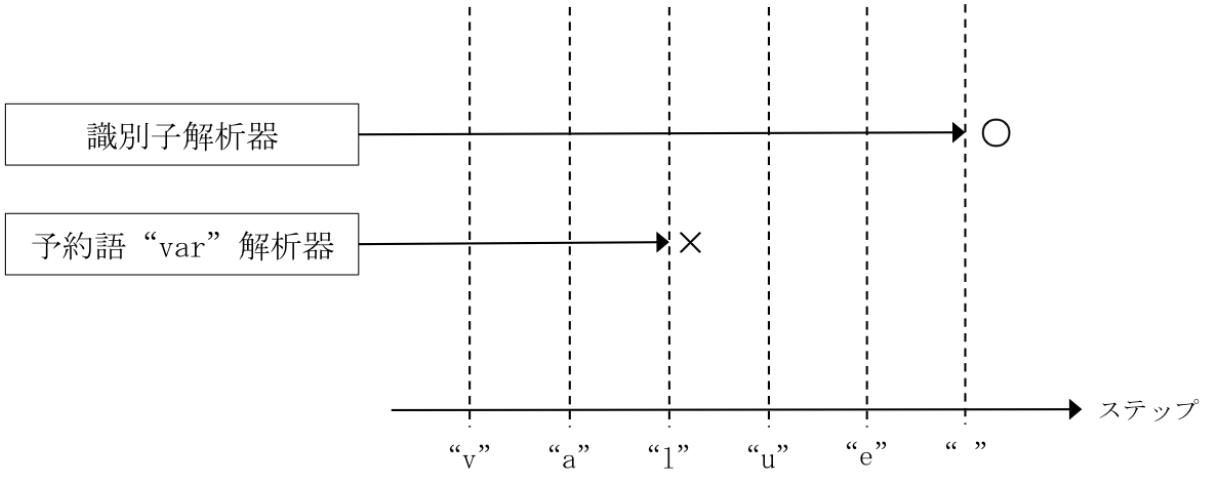

図 6.2: 字句解析器の仕組み

字句解析

TreeSwift では字句解析を文字分類器と字句解析器の 2 つのモジュールによって行う。文字分類器は図 6.1 のようにソースコードを 1 文字ずつ読み込み、各文字に所属する字句グループに関する注釈をつけることで、字句解析での文字の判断を補助する役割を担う。字句解析器は注釈付きの文字を順に読み込んで、Swift で用いられる各字句が生成できるか判断する小さなオートマトンの集まりからなっており、図 6.2 のようにそれらを平行して動かすことで、後戻りなしに注釈付きの文字を字句に分割する。

構文解析

TreeSwift の構文解析は 5.1.1 節で述べた現行の Swift コンパイラと同じ LL(k) クラスの再帰下降構文解析を用いている。

また、AST に用いるデータ構造には Swift の特徴を活かして列挙体とクラスを使い分けることで、AST を分解する際に実行時の型情報に基づく動的型変換の使用を避けられるように設計している。

モジュール解析

TreeSwift では標準ライブラリを含む外部ライブラリに定義されている変数や関数、型などに関する情報をモジュールファイルとしてコンパイラに供給することで、外部ライブラリ内で定義されている要素を import 文で取り込み、使用できるようになる。現行の Swift コンパイラは同じ働きをするモジュールファイルを独自形式で提供しているが、TreeSwift では Swift に近い構文を持つテキストファイルを用いている。

エラー検出

TreeSwift の構文解析器はもちろん誤った構文に対してエラーを報告するが、エラー回復を行ってできるだけ多くのエラーを検出するような動作は行っていないため、基本的には 1 つのエラーを検出した時点で停止してしまう。

第7章　評価

本章では、 6 章で述べた TreeSwift を用いて、現行の Swift コンパイラとその構文解析器の複雑性に関する比較評価を行う。

7.1　評価手法

6.1 節で述べたように、本評価では各コンパイラの複雑性を比較した上で比較結果の原因となった箇所を特定することを目的とする。そのために、コンパイラの構文解析器のみに着目し、特定のプログラムをコンパイルするために必要な箇所のみ抜き出してその複雑性を評価する。

表 7.1 に、 3.2 節の各手法について、構文解析器のみを対象とし (条件 A)、さらにその中の特定の実行パスだけを抜き出したものを対象とする (条件 B) という 2 つの条件下でも考察するに値する結果が得られる見込があるかをまとめた。LOC はその計測の容易さから部分的なプログラムでも正確に計測を行い、考察に繋げることができると考えられるが、他の手法はソフトウェア全体の複雑性を包括的に捉えることを目的としているため、本評価で用いるには適切とはいえない。

そのため、本研究では LOC に基づいて、表 7.4 に示す手順で計測を行う。この手順によって、図 7.1 のように特定のプログラムを実行した場合のコードだけを対象として計測を行うことができる。この手順で計測された値は通常の LOC とは異なるため、これ以降区別するために実行部分 LOC と呼ぶ。

現行の Swift コンパイラについては、もともと用意されている構文解析のみを実行するオプションをつけるだけでなく、通常そのオプションと共に有効化される AST の表示機能をコメントアウトすることで、構文解析のみが確実に実行されるように調整している。また、コンパイラを読み込むデバッガは両コンパイラ共に LLDB のバージョン 340.4.119 を使用している。具体的に評価で使用している Swift コンパイラおよびその実行時オプションの詳細は表 **??**にまとめている。なお、 6.2.1 節で述べたように TreeSwift と現行の Swift コンパイラではモジュールファイルの形式が異なっており、その点が結果に影響する可能性があるため、計測時にはコンパイラへ標準ライブラリのパスを与えず、モジュールファイルの解析に関する部分が計測対象に含まれないようにしている。

計測時にコンパイラに読ませる Swift プログラムには Apple 社が提供するプログラミング言語 Swift のチュートリアル [28] で用いられている 7 つのサンプルプログラムを使用する。ただし、これらのサンプルプログラム中に含まれていた文字リテラル中への式の埋め込みとクロージャを引数に取る関数呼び出しでの括弧の省略のみについては TreeSwift が

表 7.1: 各複雑性計測手法の評価目的との合致性

評価手法	(条件 A)	(条件 B)	理由
LOC	○	○	行数は対象に関わらず計測が可能であり、同じように加工された 2 つのプログラムに対してであればそれらを比較することもできる。
FP	×	×	比較する 2 つのコンパイラはデータの入出力に関して全く同じ動作をするため、比較にならない。
HCM	○	△	プログラムの一部分だけを対象とした場合に HCM の値がどのような意味を示すかを慎重に考える必要がある。
CCM	○	×	プログラムの一部分だけを取り出すと、分岐の片方だけが実行されたプログラムでは分岐があるにも関わらず異なる実行パスが存在しないので、CCM の値の意味が変化してしまう。

表 7.2: 実行時 LOC の計測手順

順番	手順
1	計測対象のコンパイラを構文解析のみを実行するように変更し、デバッグ情報付きでコンパイルする
2	コンパイラをデバッガで読み込み、ソースコード中のすべての行にブレークポイントを追加する
3	ブレークポイントは対応する機械語の存在する行でのみ設定されるため、その数を数える
4	1 で用意したコンパイラを評価用の Swift プログラムに対して実行し、1 度でもヒットしたブレークポイントに印をつける
5	印のついたブレークポイントの数を数える

表 7.3: 評価に使用した Swift コンパイラの詳細情報

バージョン	swift-2.2-SNAPSHOT-2015-12-31-a から AST の表示をコメントアウトしたもの
実行時に使用するオプション	-frontend -dump-parse

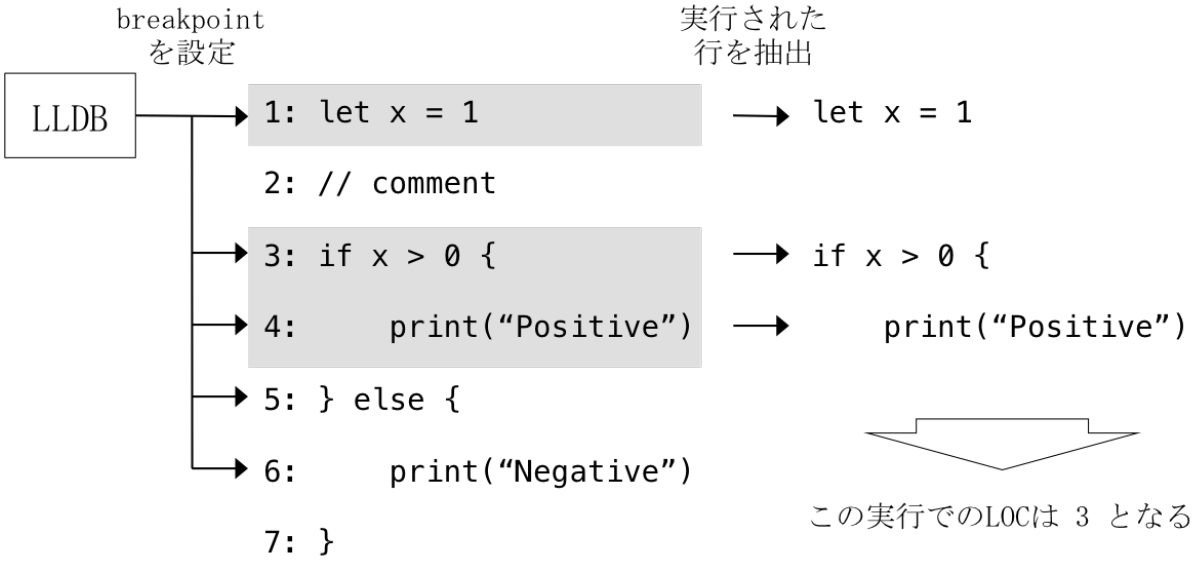

図 7.1: 実行部分 LOC の計測手順

対応していないため、同じ意味を持つ明示的な文字列への変換および文字の結合と括弧付きの呼び出しに変更している。各プログラムの概要を表 ??にまとめた。これらのプログラムはチュートリアルという特性上 Swift の代表的な機能を充分網羅している上、プログラムで使用される変数や型に使用する名前や値、処理の構造には実際のソフトウェアを想定したような現実的なものが選ばれているため、コンパイラの機能を満遍なく使いつつも特に実際のソフトウェアでもよく用いられるであろう箇所にフォーカスした計測ができると期待できる。

7.2 計測

まず、表 7.4 の各プログラムをコンパイルした場合それぞれについて 7.1 節で説明した方法を用いて実行部分 LOC を計測した。表 7.5 がその結果および現行の Swift コンパイラの結果 (表中 Swift) を TreeSwift の結果 (表中 TreeSwift) で割った値の一覧である。単純に計測結果だけを見ると、プログラムのコンパイルに要した行数は現行の Swift コンパイラの方が $2\tilde{3}$ 倍近く多い。ただし、TreeSwift では Swift コンパイラとして必要な構文解析以外の処理が全て実装されているわけではないため、Swift コンパイラにおいては AST の検査など後の処理のための準備が行われており、実行部分 LOC が大きく算出されている可能性がある。

そのため、次に各コンパイラの結果をソースコードのファイルごとに分けた上で、構文解析の本体を構成するファイル (構文解析本体ファイル) と AST を構成するファイル (AST ファイル) を選び、それらについて実行部分 LOC の合計を算出した。なお、構文解析本体ファイルと AST を構成するファイル以外のファイルはコンパイラ内の各所から共通で

表 7.4: 計測に使用する Swift プログラム

ファイル名	プログラムの概要
simple_values.swift	変数の定義と使用、文字・数値・配列・辞書リテラルの使用について説明されている。
control_flow.swift	分岐や繰り返しについて、Swift に特徴的なオプショナル値やパターンマッチと組み合わせた例を中心に説明されている。
functions_and_closures.swift	関数の定義と使用について、基本的なものから可変長引数やクロージャを用いた高階関数の呼び出しなどの応用的なものまでが説明されている。
objects_and_classes.swift	クラスとそのインスタンス化、継承、関数のオーバーライドなどの Swift におけるオブジェクト指向プログラミングについて説明されている。
enumerations_and_structures.swift	構造体と Swift に特徴的な関数や関連型などを持つ事のできる列挙体についてその宣言方法と使い方が説明されている。
protocols_and_extensions.swift	存在型や型拡張によるクラスや構造体の分類および拡張について説明されている。
generics.swift	型パラメータ多相を用いた関数や列挙体の宣言方法とその使用方法について説明されている。

使用されるユーティリティや AST から間接的に使用される構文解析後の処理ステップ用のデータ構造などから構成されている。

表 7.6 および表 7.7 は各コンパイラごとに表 7.5 の数値を構文解析本体ファイル、AST ファイル、その他のファイル毎に分割した算出結果であり、図 7.2 は構文解析本体ファイル、図 7.3 は AST ファイルの実行部分 LOC を現行の Swift コンパイラと TreeSwift で比較したグラフである。また、構文解析本体ファイルについては各コンパイラの実行部分 LOC の差と比を表 7.8 にまとめた。

7.2.1 考察

まず、表 7.5 の全体における実行部分 LOC で両コンパイラに異様に大きな差を与えていた原因は表 7.6 の実行部分 LOC を各ファイル種別ごとに分割して算出した結果から、構文解析本体ファイルではなく、その後の処理のための準備を行うためのプログラム群にあると考えられる。実際、現行の Swift コンパイラはその後に控える SIL 生成のために AST を走査して正当性の確認を行ったり、Swift プログラムの構文から SIL 生成用の付帯情報が構文解析時に生成されるようにしており、それらの TreeSwift に存在しない機能を

表 7.5: 各プログラムをコンパイルした各コンパイラの実行部分 LOC

対象プログラム	Swift	TreeSwift	比
simple_values.swift	4188	1928	2.172
control_flow.swift	5347	2226	2.402
functions_and_closures.swift	5819	2187	2.661
objects_and_classes.swift	5937	2122	2.798
enumerations_and_structures.swift	5762	2258	2.552
protocols_and_extensions.swift	5598	2132	2.626
generics.swift	5887	2233	2.636

表 7.6: Swift における各ファイル種別ごとの実行部分 LOC

対象プログラム	構文解析本体ファイル	AST ファイル	その他のファイル
simple_values.swift	1148	1263	1777
control_flow.swift	1569	1816	1962
functions_and_closures.swift	1783	1982	2054
objects_and_classes.swift	1934	1970	2033
enumerations_and_structures.swift	1858	1886	2018
protocols_and_extensions.swift	1781	1851	1966
generics.swift	1897	1934	2056

表 7.7: TreeSwift における各ファイル種別ごとの実行部分 LOC

対象プログラム	構文解析本体ファイル	AST ファイル	その他のファイル
simple_values.swift	376	1457	95
control_flow.swift	433	1698	95
functions_and_closures.swift	448	1644	95
objects_and_classes.swift	437	1590	95
enumerations_and_structures.swift	505	1658	95
protocols_and_extensions.swift	493	1544	95
generics.swift	481	1657	95

提供するためのプログラムが実行部分 LOC の大きな差を生んでいると見られる。なお、表 7.7 において TreeSwift での構文解析本体ファイルおよび AST ファイル以外のファイルの実行部分 LOC が同じ値となっているのは、TreeSwift ではそうした追加情報の生成や AST の正当性確認を行っておらず、その他のファイルがコンパイラ全体で使用される

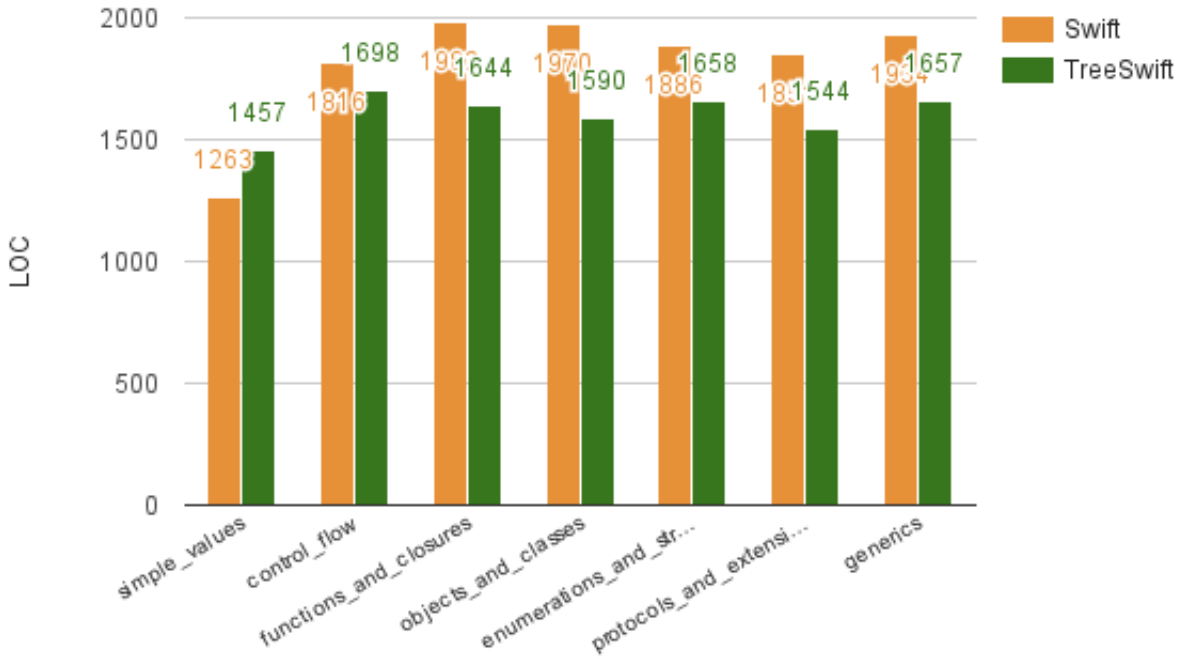

図 7.2: 構文解析本体ファイルに対象を絞った実行部分 LOC

表 7.8: 構文解析本体ファイルに対象を絞った実行部分 LOC の差と比

対象プログラム	差	比
simple_values.swift	-194	0.867
control_flow.swift	118	1.069
functions_and_closures.swift	338	1.206
objects_and_classes.swift	380	1.239
enumerations_and_structures.swift	228	1.138
protocols_and_extensions.swift	307	1.199
generics.swift	277	1.167

ファイル IO の仕組みなどを提供する基本ツールだけで構成されており、構文にかかわらず同じように使用されるからである。

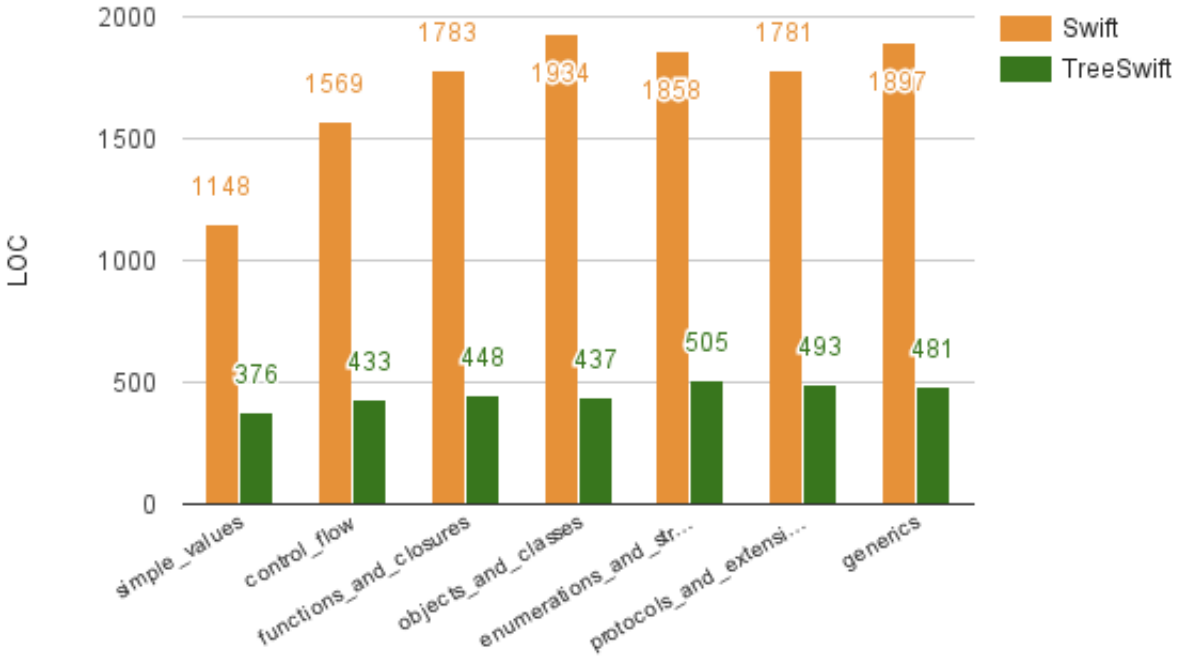

図 7.3: AST ファイルに対象を絞った実行部分 LOC

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

* 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

8.2　今後の展望

8.2.1　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

8.2.2　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

.1　計測に使用したプログラムの全文

??節における計測に使用したプログラムの全文をプログラム ??に示す。

謝辞

参考文献

[1] James L. Elshoff and Michael Marcotty. Improving computer program readability to aid modification. *Communications of the ACM 25.8*, pages 512–521, 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 山形 浩生 (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. *Communications of the ACM 36.11*, pages 81–94, 1993.

[8] Rajiv D. Banker, Gordon B. Davis, and Sandra A. Slaughter. Software development practices, software complexity, and software maintenance performance: A field study. *Management science 44.4*, pages 433–450, 1998.

[9] Sheng Yu and Shijie Zhou. A survey on metric of software complexity. *Information Management and Engineering (ICIME), 2010 The 2nd IEEE International Conference on. IEEE*, pages 512–521, 1982.

[10] Charles R. Symons. Function point analysis: Difficulties and improvements. *Software Engineering, IEEE Transactions on 14.1*, pages 2–11, 1988.

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

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

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

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

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

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

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

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

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

[20] Joe Groff and Chris Lattner. Swift intermediate language. http://llvm.org/devmtg/2015-10/slides/GroffLattner-SILHighLevelIR.pdf, January 2016.

[21] A. V. エイホ, M. S. ラム, R. セシィ, J.D. ウルマン, and 原田 賢一 (訳). **コンパイラ *[第 2 版]* —原理・技法・ツール—**. 株式会社 サイエンス社, 2009.

[22] T. J. Parr and R. W. Quong. Antlr: A predicated-ll(k) parser generator. *Software: Practice and Experience 25.7*, pages 789–810, 1995.

[23] Type checker design and implementation. https://github.com/apple/swift/blob/master/docs/TypeChecker.rst, January 2016.

[24] Benjamin C. Pierce, 住居 英二郎 (監訳), 遠藤 侑介, 酒井 政裕, 今井 敬吾, 黒木裕介, 今井 宜洋, 才川 隆文, and 今井 健男 (共訳). **型システム入門 プログラミング言語と型の理論**. 株式会社 オーム社, 2013.

[25] The swift programming language (swift 2.1): Automatic reference counting. https://developer.apple.com/library/ios/documentation/Swift/Conceptual/Swift_Programming_Language/AutomaticReferenceCounting.html, January 2016.

[26] The llvm compiler infrastructure. http://llvm.org, January 2016.

[27] Treeswift. https://github.com/demmys/treeswift, January 2016.

[28] The swift programming language (swift 2.1): A swift tour. https://developer.apple.com/library/ios/documentation/Swift/Conceptual/Swift_Programming_Language/GuidedTour.html, January 2016.