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

Swift がオープンソース化されたオープンソースプロジェクトでは開発者のモチベーションを向上する必要があるそのためには、可読性を向上する必要がある

可読性を表す指標として特定のプログラムを実行するために必要なプログラムの行数から算出される実行部分 LOC を用いる

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によって、そのソースコードの可読性が拡張性や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、これらの方法のみ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性は明らかではない。

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 source 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

1.1.1　オープンソースプロジェクトにおけるモチベーション向上の必要性

2015 年 12 月、Apple 社が 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る Swift のコンパイラをオープンソース化した。このプロジェクトのオープンソース化は、単にソースコードを公開することだけを指すのではない。Open Source Initiative が定義 [1] しているように、ソースコードを公開した上でそのソフトウェアの改変版や派生版の販売・頒布を許可するのが一般的なオープンソースプロジェクトのスタイルである。しかし、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が困難である。そのため、オープンソース化は単にソースコードの管理方法を変えるだけでなく、ソフトウェアの開発フローそのものの変更を伴う場合が多い。

クローズドプロジェクトではプロジェクトに関わる開発者の人数がはっきりしているため、そのリソースに見合った明確な開発方針がマネージャーによって決定され、それに従って役割分担された人員によって実装やレビューが進められる。それに対してオープンソースプロジェクトでは、プロジェクトに関わる開発者やユーザから成るコミュニティでの議論を通して現在のプロジェクトで行うべき修正や拡張が提案・検討され、それに対して必要性を感じた開発者が自発的に実装やレビューを行う。すなわち、クローズドプロジェクトではそのプロジェクトに割ける人員やその時間といったリソースが開発を進めるための最も重要な因子となっているのに対して、オープンソースプロジェクトではその開発に必要な労力を差し引いても開発者が十分なモチベーションを持てるようにしていく必要がある [2]。

1.1.2　ソースコード可読性向上の必要性

Swift コンパイラについて言えば、そのプロジェクト自体に対する開発者の関心は非常に高いといえる。2001 年以来 10 年以上にわたって Apple 社が提供する Cocoa フレームワークのコア言語となっていた Objective-C [3] を完全に置き換える言語として発表された Swift は Objective-C の膨大なユーザをそのまま受け継ぐ形となり、インターネット上の情報量を元にしたプログラミング言語の人気度格付け [4] [5] においても過去に類を見ない早さで上位にランキングされるようになっているからだ。

しかし一方で、開発を行う上でのコストにはまだ改善の余地があり、開発に必要なリソースが開発者の興味・関心に応じて充分に割り当てられるかという点については疑問が残る。ソフトウェアの修正や拡張といったメンテナンスのためのコストには、ソフトウェアのソースコードの可読性が大きな影響力を持つことが Elshoff [6] や Banker [7] [8] によって言及されているが、Swift コンパイラにおいてそのソース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のみ保たれている。このような方法では可読性が向上していく可能性が低いだけでなく、新しく修正・追加されたコードがオープンソース化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可読性を維持することすらできなくなる。

つまり、Swift プロジェクトをさらに活発なものとするためには、Swift コンパイラのソースコード可読性を何かしらの方法で向上していく必要がある。

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 プロジェクトにおいてコンパイラのソースコード可読性を向上する必要があるという課題に着目する。

もちろん、現在も Swift プロジェクトでは可読性が低くなってしまっている箇所のリファクタリングなども盛んに行われているが、そうした部分的な修正だけでは、プロジェクト全体の開発コストの削減にはつながりづらい。そのため、本研究では Swift コンパイラ全体における可読性の向上を課題として考える。

1.3 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、Swift コンパイラの可読性をより具体的に表す数値指標について説明し、本研究の目的を明確化する。 3 章では、本研究において Swift コンパイラの可読性を向上するために用いる Self-host 化というアプローチについて説明する。 4 章では、現行の Swift コンパイラと比較評価を行うために本研究で実装する Self-host 化された Swift コンパイラの設計について説明し、その実装を概説する。 5 章では、Self-host 化された Swift コンパイラの可読性を 2 章で述べる指標を用いて現行の Swift コンパイラのものと比較し、その結果について考察する。 6 章では本研究のまとめと今後の課題についてまとめる。

第2章　本研究の目的

本研究ではソフトウェアの可読性を示す複数の指標の中から LOC(Line of Code) を対象とし、Swift コンパイラにおいてその LOC をベースとした実行部分 LOC を改善することを目的とする。本章では、ソフトウェアの可読性に影響を及ぼす要因を分類し、特に本研究で重要視する種類の要因を定量化することで、その実行部分 LOC の算出方法と正当性について述べる。

2.1　ソースコード可読性中の改善すべき要因

ソースコードの可読性に影響を与える要因には様々なものがあるが、本研究では可読性に関する過去の研究 [6] [7] [8] [9] [10] に挙げられている要因を参考に、それらの要因を表 2.1 に上げる 3 種類に分類する。

このように要因を分類することによって、 1.1 節で述べた Swift コンパイラの可読性の低下がどのような種類の要因によって起こりうるのかを考察できる。

まず、2 つ目の説明性に関する要因については、コメントや識別子、インデントなどのプログラム中の明確に分離される部分が対象となっているため、既存のコードにそのスタイルを合わせたり、レビューで違いを指摘することが比較的容易である。次に、3 つ目の知識量に関する要因については、ソフトウェアの読者が誰であるかによって左右されるため、プロジェクトの発展に従って変化するようなものではない。それらに対して、ソフトウェアの処理の複雑性は比較してレビューやコーディングスタイルへの追従によって保持することが難しく、今後 Swift コンパイラの可読性が低下する主要因となると考えられる。

そのため、本研究ではこれら 3 つの側面のうち特にソフトウェアの処理の複雑性を下げることによって得られる可読性の向上を重要視し、以降はそのための方法について論じる。

2.2　ソフトウェアの処理の複雑性を定量化する手法

本研究が重要視するソフトウェアの処理の複雑性を定量化する手法は既存研究によって多く提案されている。

本節では、その中でも特によく取り上げられる LOC、FP、HCM、CCM という 4 種類の手法 [11] [12] について整理する。なお、これら 4 つの手法における値とソフトウェアの複雑性との関連性は全て経験則的に示されているのみである。そのため、実際に使用する

表 2.1: ソースコード可読性に要因を与える要因の分類

種類	分類される要因の例	可読性との関係
ソフトウェアの処理の複雑性に関する要因	• ソフトウェアに 使用されるデータ構造やアルゴリズムの複雑さ • 制御構造などによるソフトウェアの実行パスの複雑さ • 使用されるプログラミング言語の表現力	複雑性が低いと可読性が上がる
ソフトウェアの説明性に関する要因	• コメントの充実度 • 変数や関数、型などの名前の適切さ • インデントの適切さ	説明性が高いと可読性が上がる
ソフトウェアの読者に求められる知識量に関する要因	• ソフトウェアが解決する問題の難易度 • ソフトウェアが使用する手法の難易度 • ソフトウェアに使用される手法の数	求められる知識量が少ないと可読性が上がる

際には単に値の大小を比べるだけでなく、その原因の究明までを含めた充分な考察を行うことが求められる。

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を指標にソフトウェアの複雑性を算出する。一般的に充分大きなソフトウェアでは行数の多いソフトウェアの方が複雑性が高くなるとされているが、小さなソフトウェアについては行数と複雑性の間の明確な相関は認められていないため、注意が必要である。

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点については考慮する必要がある。

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開発にかかる工数の見積などの目的でよく使用されているが、対象とするソフトウェア内でのデータ処理による複雑性は経験的に決定される重み付け以外には一切反映されないため、データ処理中心のソフトウェアでは有意義な値にならない点に注意が必要である。

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一部のソフトウェアでは有意義な計測とならない点に注意が必要である。

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一部のソフトウェアでは有意義な計測とならない点に注意が必要である。

このように、ソフトウェアの処理の複雑性を定量化する手法では各手法によって対象とすることのできるソフトウェアの種類が異なる。そのため本研究においても、まずは各手法が本研究の対象とする Swift コンパイラに対して有効であるかを検討する必要がある。

2.3 本研究において Swift コンパイラの可読性を表す指標

本節では、 2.2 節で述べた手法のうちどれをどのように使用することで本研究が対象とする Swift コンパイラの複雑性を定量化できるかについて述べる。

Swift コンパイラに各手法を適用する上で最も気をつけなければならないのは、Swift コンパイラは様々な手法が用いられている大きなソフトウェアであるという点である。どの手法であっても、コンパイラのように大きなソフトウェアの全体を対象として評価を行うと、部分部分では複雑性の高い箇所と低い箇所がそれぞれ存在しているにもかかわらず、それらが打ち消し合って平均的な結果のみしか算出できなくなってしまい、適切な考察に結びつかない。

また、コンパイラのソースコードの中にはコンパイラとしての基本的な機能を担う箇所と、特定の条件下のみで必要な機能を担う箇所が混在している。本研究ではプロジェクト

における拡張や修正の活発化に繋がるコンパイラの可読性を向上することが目的であるため、特によく読まれるであろう基本的な機能の複雑性から指標を算出するようにしたい。

これらの条件を満たすために、本研究では Swift コンパイラのソースコード中の基本的な一部の構文をコンパイルするために必要な箇所のみを対象に測定手法を用いることを考える。しかし、Swift の基本的な構文を満たすいくつかのサブセットを定義し、その構文をコンパイルするために必要な関数などだけをソースコードから抜き出すようなことは到底簡単ではない。そこで本研究においては、コンパイラ中の特定の構文を対象としている箇所を静的に決定するのではなく、その構文を実際にコンパイルした際に実行される箇所を動的に把握することで、特定の構文をコンパイルするために必要な箇所だけを対象とした複雑性の計測を行う。

表 2.2 に、 2.2 節の各手法について、Swift コンパイラを対象とし (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 つの条件下でも得られる値が充分に複雑性を表した指標となり得るかをまとめた。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、本研究で用いるには適切とはいえない。

表 2.2: 各複雑性計測手法の本研究における条件との合致性

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る。
FP	×	×	コンパイラは入出力ではなく内部的な処理が中心となるため、FP の値はコンパイラの複雑性を充分表現できない。
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	○	×	プログラムの一部の実行パスだけを取り出すと、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ために CCM に反映されず、CCM の値の意味が変化してしまう。

そのため、本研究では LOC に基づいて図 2.1 のようにデバッガを用いることで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い、それを本研究における Swift コンパイラの可読性を表す指標として用いる。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。

つまり、本研究の目的は Swift の基本的なプログラムを対象とした際の Swift コンパイラの実行部分 LOC を下げることであり、それが可読性の向上ひいてはプロジェクトの活性化に繋がるものであると考える。

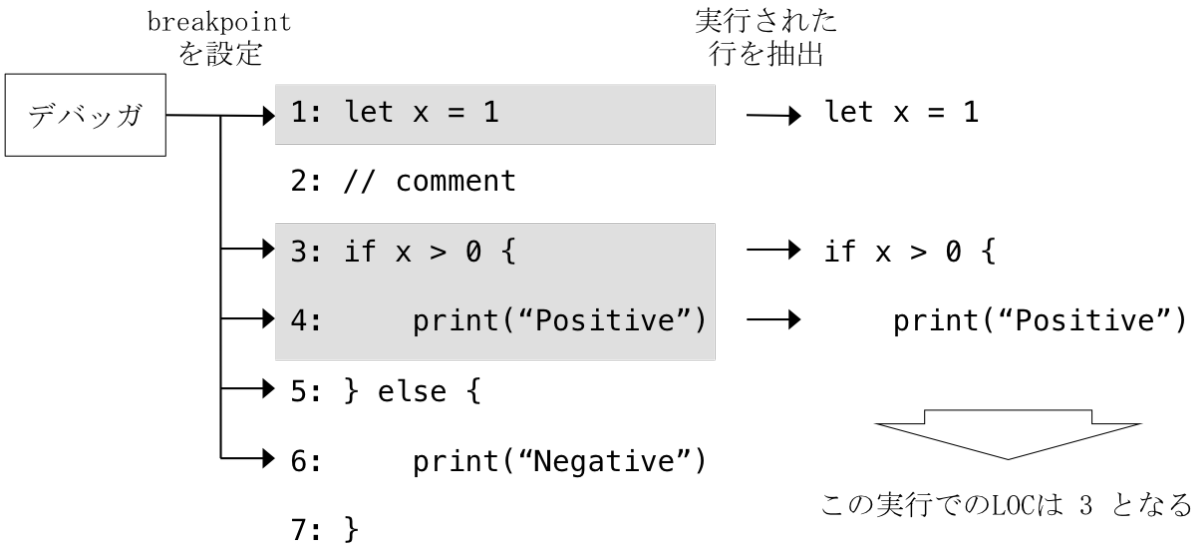

図 2.1: 実行部分 LOC の計測手順

第3章　本研究のアプローチ

本章では、 2.3 節で述べた Swift コンパイラの実行部分 LOC を下げるために本研究で用いるアプローチについて述べる。

3.1　可読性の向上のために用いる手段

ソフトウェアの実行部分における行数を削減し可読性を向上するための手法としては、ソフトウェアで用いられるデータ構造やアルゴリズムを変更したり、より根本的に用いるプログラミングパラダイムを変更したりするようなアプローチが考えられる。しかし、こうしたアプローチが適用可能かどうかは対象とするソフトウェアが解決しようとしている問題やそのために使用している手法によってそもそも適用が不可能である場合すらある。

そこでまず、他のプログラミング言語において可読性を向上することを目的として行われているアプローチについて見ると、Swift 以外の汎用言語のコンパイラにおいては、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 3.1 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 [4] 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 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 3.1 では Bootstrap しているかどうかについてまとめている。

表 3.1 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。

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、一部の構文をその構文字体を用いて解釈することができ、特定の構文をコンパイルするために必要な処理をより簡単に記述できる可能性がある。また、特に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、より高い表現力によってより短い行数でコンパイラを書き直す事ができる可能性が高い。

さらに、他の手法とは異なるメリットとして、Self-host 化を行うとコンパイラを記述する上で必要なプログラミング言語の知識を 1 つの言語に関するもののみに限定できるようになる、という点がある。これはコンパイラを開発する上で使用される手法が減少していることにほかならず、 2.1 節で述べた 3 つの要因の分類の内、3 つ目の知識量に関する要因の改善を行うことに値する。これはもちろん本研究が最も注目する点ではないが、目標とする実行部分 LOC の低減を達成した上で更に他の点においてもソースコード可読性を向上できる可能性があるのであれば、その手法を採用しない手はない。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性は明らかではない。

そこで、本研究では Self-host 化した Swift コンパイラを実装し、現行の Swift コンパイラとその実行部分 LOC を比較することで、Self-host 化が可読性の向上に有用であることを示す。

なお、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 [21] 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。

3.2 Self-host 化の事例

3.1 節では、現在の Swift コンパイラが Self-host 化を行っていない理由として、可読性以外におけるデメリットの発生が考えられていることを述べた。このように、本研究のアプローチとして用いる Self-host 化はそのソースコードを書き直すために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る可能性がある。そこで本節では、特に近年多くのユーザに使用されており、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 Self-host 化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 つの事例から、Self-host 化の持ちうる様々な効果について紹介する。なお、本節で述べる事例はすべて Self-host 化だけでなく Bootstrap まで行っている。一般にコンパイラの基幹的な機能について Self-host 化を行った言語では最終的に Bootstrap を目的とすることが多いため、本節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。

3.2.1 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。 [22]

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 [23] 内に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容易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、 [23] 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 の持つ構文が他の高級言語と比べて少ない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 [24]。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 [25] 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。

3.2.2 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。 [16]

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 [26] 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 [27]。

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 [28]。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。

3.2.3 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。 [17]

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 [29]。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 [30]。

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 [31] 内で述べている。

3.3 Self-host 化による可読性の向上以外のメリット

まず、表 3.2 に 3.2 節で述べた事例から 3.1 節で述べた可読性の向上に繋がる点以外の Bootstrap における利点をまとめ、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。

3.2.1 節や 3.2.2 節で見たように、各事例でも記述やデバッグが容易になったり、柔軟性や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、 3.2.2 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、 3.2.1 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。

3.2.3 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。

3.4 Self-host 化によって想定されるデメリット

次に、表 3.3 に 3.2 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Bootstrap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。

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、 3.2.3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、 3.2.1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、 3.2.1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3.2.3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3.2.1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、 3.2.3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。

最後に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。

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。

表 3.1: 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 [13], gcc [14], Microsoft Visual C++ [15]
4	Python	○	PyPy [16]
5	C#	○	.NET Compiler Platform [17]
6	PHP	×	-
7	Visual Basic .NET	○	.NET Compiler Platform [17]
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 [18]
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 [19]
22	Fortran	×	-
23	D	○	DMD [20]
24	Groovy	×	-

表 3.2: 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	×

表 3.3: 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	○

第4章　Self-host化したSwiftコンパイラの設計と実装

本章では、本研究で現行のコンパイラとその実行部分 LOC を比較する Self-host 化した Swift コンパイラに必要な設計について述べ、実装について紹介する。

4.1　実装箇所の絞り込みの必要性

2.3 節でも述べたように、Swift コンパイラは非常に大きなソフトウェアであるため、評価結果から適切な考察を行うためには、これをより小さな部分に分け、部分ごとに評価を行う必要がある。そのために同節では本研究における可読性を表す指標として実行部分 LOC を使用することを説明したが、それによって特定のプログラムを対象とした時に必要となるコンパイラの部分だけを取り出したとしても、その中に含まれる処理には多数のステップがあり、まだ適切な考察を行うために十分対象を切り分けられているとはいえない。

また、Self-host 化した Swift コンパイラと現行の Swift コンパイラの実行部分 LOC を比較した際に、その差が Self-host 化によって生まれたものであると考えるためには、2 つのコンパイラの比較箇所において Self-host 化以外の大きな実装上の差があってはならない。しかし、Self-host 化した Swift コンパイラではそもそも実装に同様の手法を用いることができない処理ステップも存在しうる。

そこで本研究では、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を考察した上で、特に比較を行う上で有意義だと思われる一部分についてのみ、現行の Swift コンパイラと同等の機能を持つように実装する。

4.2　現行の Swift コンパイラの構成

Swift コンパイラの大まかな構成を図 4.1 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 [32] 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

図 4.1: Swift コンパイラの構成

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 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構成に基づいている。

4.2.1 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 [33]。

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。

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、プログラムを解析する。

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 [34] 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。

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。

4.2.2 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 4.1 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 4.2 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 print 関数に渡されている関数 s.f 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関数 f の 2 つの宣言を探した上で、s.f に与えられている引数の型を調べ、それが Int 型であるために 2 つの宣言のうち Int 型を引数に取る方を選ぶ、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 4.1: 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 4.2: 変数解決までに複数の処理が必要となる例

```
1  class Sample {
2      func f(a: Int) {
3          print("int")
4      }
5      func f(a: String) {
6          print("string")
7      }
8  }
9  let s = Sample()
10 print(s.f(0))
```

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。

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 [35] [36]。

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、関数のオーバーロードを実現するためのものであり、これらを強力な型推論アルゴリズムと共に採用している言語は極めて少ない。

4.2.3 SIL 生成

SIL 生成は、AST を分析してより抽象度の低い Swift 独自の中間言語である 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 [32]。

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。

4.2.4 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 [37]、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。

4.2.5 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 [38]、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。

なお、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。

4.3 実装を行う Swift コンパイラの部分

本研究では、 4.2 節における Swift コンパイラの構成要素の観察から、構文解析器についてのみ注目し、これを実装して比較評価を行う。同節で述べたように、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、用いられている手法も明確に定義されている一般的なものである。そのため、本研究での比較評価には構文解析器を用いる。

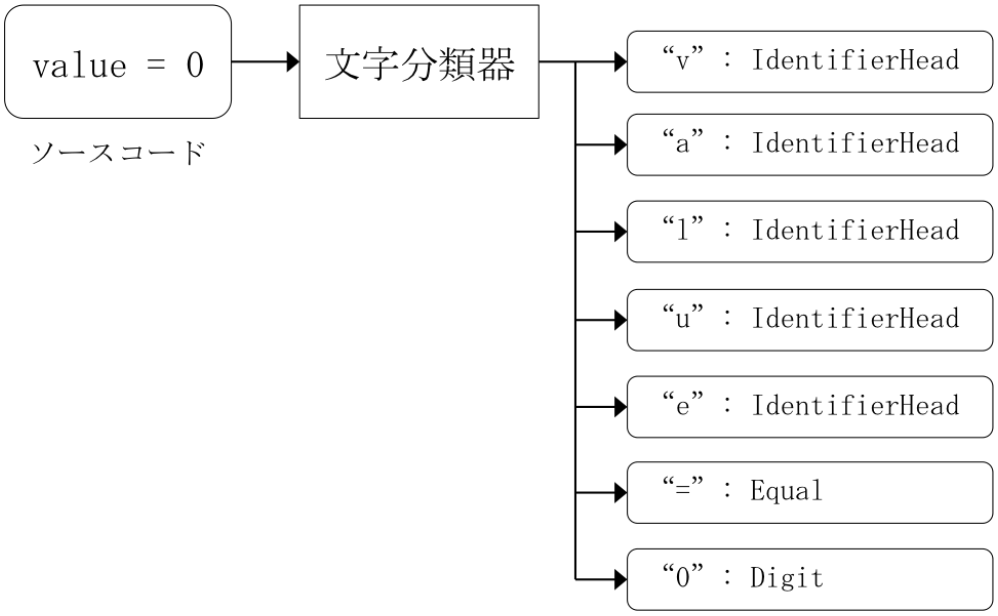

図 4.2: 文字分類器の仕組み

4.4 実装の概要

本節では実装した Swift コンパイラの構文解析器について、Swift プログラムから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の 4 つの処理機能に分けてその実装を紹介する。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 [39]。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、以降では本研究で実装した Swift コンパイラを指して TreeSwift と記述する。また、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。

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 4.2 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 4.3 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。

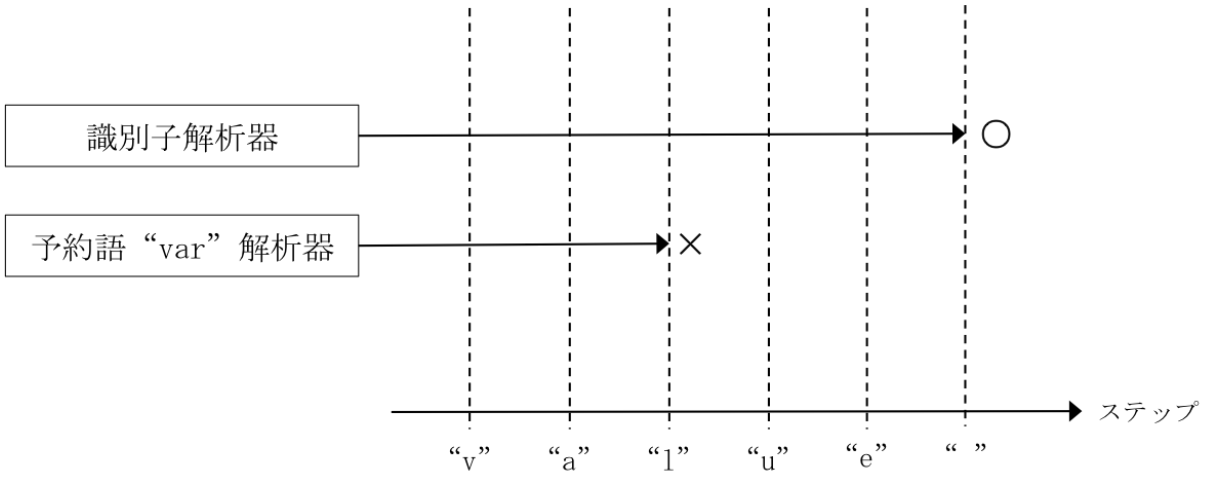

図 4.3: 字句解析器の仕組み

構文解析

TreeSwift の構文解析は 4.2.1 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、使用できるようにしている。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5章　評価

本章では、 4 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実行部分 LOC の比較評価を行う。

5.1　評価手法

本節では、現行の Swift コンパイラと TreeSwift についてその構文解析器の実行部分 LOC を評価するための具体的な方法について述べる。

まず、現行の Swift コンパイラでは、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する。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 5.1 にまとめている。なお、 4.4 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。

表 5.1: 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

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 [40] 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における括弧の省略という 2 つの構文については TreeSwift が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という異なる構文に変更している。

各プログラムの概要を表 5.2 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それぞれ異なる Swift の基本的な機能を使用している上、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現実的なものが選ばれているため、様々な構文について別々にかつ実際のソフトウェアで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。

表 5.2: 計測に使用する Swift プログラム

ファイル名	コメントや空行を除いた行数	プログラムの概要
simple_values.swift	25	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	67	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	69	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	82	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	62	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	33	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	25	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。

5.2 計測

まず、表 5.2 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 5.1 節で説明した方法を用いて実行部分 LOC を計測した。表 5.3 がその結果および現行の Swift コンパイ

ラの結果 (表中 Swift) と TreeSwift の結果 (表中 TreeSwift) の比の一覧である。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 2~3 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 Self-host 化による差だけでなく、その差によって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実行部分 LOC をその行が書かれている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群 (構文解析ファイル群)、AST を構成するファイル群 (AST ファイル群) およびその他のファイル群ごとに集計しなおした。なお、その他のファイル群にはコンパイラ内の各所から共通で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が含まれている。

表 5.4 および表 5.5 はコンパイラごとにそのファイル群ごとの再集計を行った結果であり、図 5.1 および図 5.2 はそれらの値の内、構文解析ファイル群と AST ファイル群それぞれの実行部分 LOC について現行の Swift コンパイラのものと TreeSwift のものを同じグラフ上にプロットしたものである。また、構文解析ファイル群については後の考察のために各コンパイラにおける実行部分 LOC の差と現行の Swift コンパイラの実行部分 LOC に対する差の割合を表 5.6 にまとめた。

表 5.3: 各コンパイラにおける実行部分 LOC の計測結果

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

5.2.1 考察

まず、表 5.3 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 5.4 の実行部分 LOC をファイル群ごとに再集計した結果から、構文解析ファイル群ではなく、それ以外のファイル群に含まれるプログラム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 5.5 においてその他

表 5.4: 現行の Swift コンパイラについてファイル群ごとに実行部分 LOC を再集計した結果

対象プログラム	構文解析ファイル群	AST　ファイル群	その他のファイル群
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 5.5: TreeSwift についてファイル群ごとに実行部分 LOC を再集計した結果

対象プログラム	構文解析ファイル群	AST　ファイル群	その他のファイル群
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

表 5.6: 各コンパイラの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	Swift からの LOC 減少率
simple_values.swift	-194	-16.9 %
control_flow.swift	118	7.52 %
functions_and_closures.swift	338	19.0 %
objects_and_classes.swift	380	19.6 %
enumerations_and_structures.swift	228	12.3 %
protocols_and_extensions.swift	307	17.2 %
generics.swift	277	14.6 %

のファイル群の実行部分 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

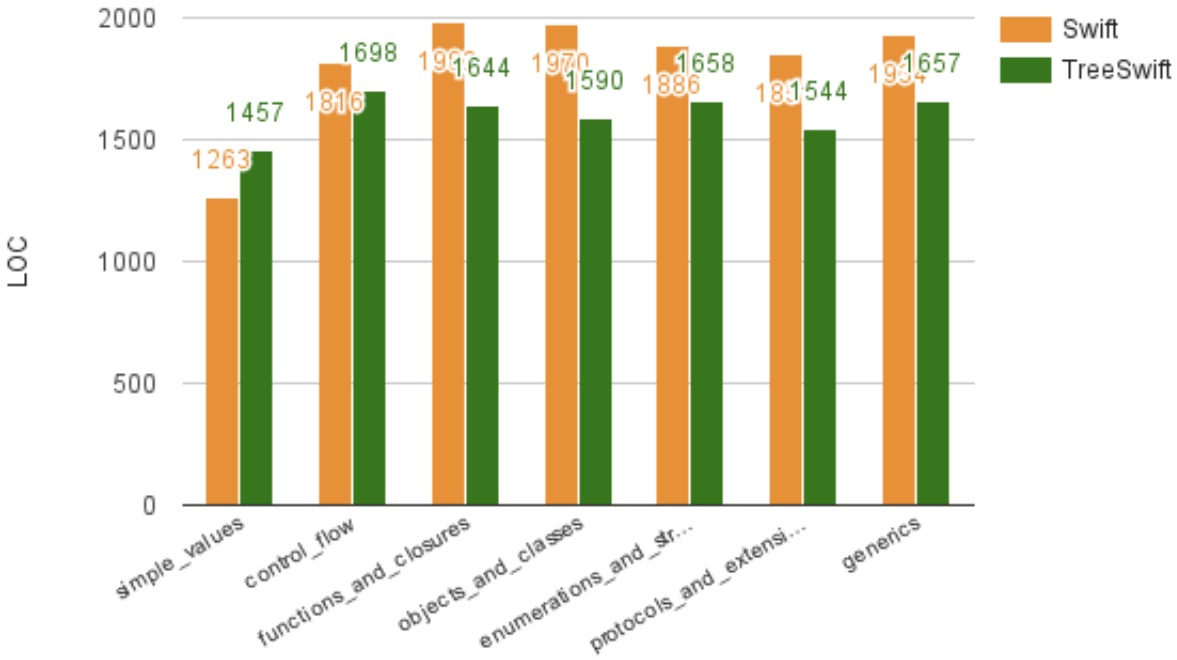

図 5.1: 両コンパイラの構文解析ファイル群の実行部分 LOC の比較

用されるファイル I/O 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ているからである。

しかし、図 5.1 にあるそうした部分を差し引いた構文解析ファイル群の結果のみについて見ても、simple_values.swift 以外のプログラム全てで TreeSwift の方が低い実行部分 LOC を記録している。特にこの差を作り出していると考えられる原因の 1 つが、両コンパイラにおける分岐構文の使用傾向の差異である。表 5.7 および表 5.8 は構文解析ファイル群の実行部分 LOC の算出元となっている行の内、分岐構文の使用されている行がどれだけ存在するかを調べたものである。なお、分岐構文としては現行の Swift コンパイラでは if 文と switch 文と assert 文、TreeSwift では if 文と switch 文と guard 文をそれぞれ対象としている。

これらの表に見られる通り、現行の Swift コンパイラにおいては switch 文に対して if 文が多く使用されているのに比べて、TreeSwift では if 文の使用量が減り、代わりに switch 文の使用量が増加している。これは、Swift の持つ強力なパターンマッチや列挙型によって、C++では異なる分岐文で別個に処理されていたものを同一の switch 文で使用できるようになったためだと考えられる。Swift のパターンマッチは条件判定と同時に列挙型やタプルなどの内部構造を分解して変数に束縛する役割を果たすため、switch 文への移行は複数の条件文をまとめた上で変数の宣言や代入に関する行を削減することに繋がり、結果として実行部分 LOC に減少につながっていたと考えられる。

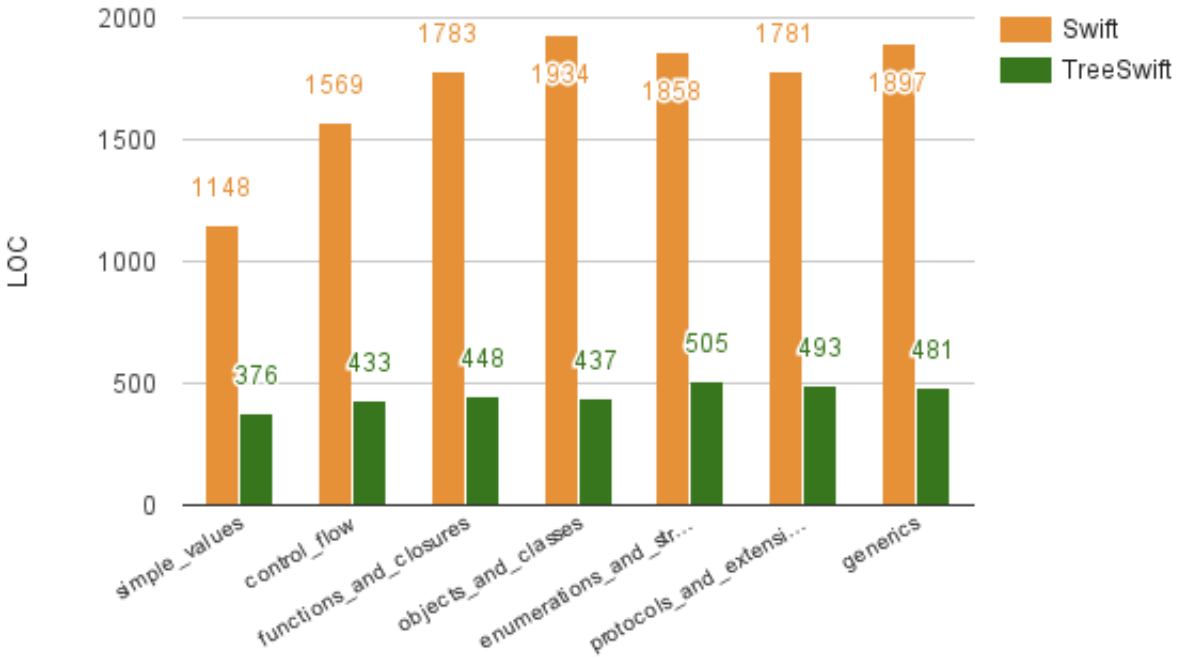

図 5.2: 両コンパイラの AST ファイル群の実行部分 LOC の比較

また、simple_values.swift のみにおいて TreeSwift の実行部分 LOC が大きくなっていたのは、 4.4 節で述べたように、TreeSwift では構文解析のステップを簡略化することを目的として、現行の Swift コンパイラのものと比較して複雑なアルゴリズムを字句解析器に採用しているためだと思われる。これは、7 つの対象プログラムについて Swift の構文解析本体ファイルのみを対象とした実行部分 LOC の標本分散は約 53874 であるのに対して、TreeSwift では約 5930 と低く、TreeSwift の方が全てのプログラムで共通して実行されている部分が多くの部分を占めていることからも考察できる。

以上の考察より、表 5.6 中の結果は Swift コンパイラを Self-host 化したことによる効果を充分表しており、評価に用いたプログラムにおいては Self-host 化によって Swift コンパイラの実行部分 LOC を平均 10.47%減少させることができたことがわかる。

表 5.7: Swift における構文解析ファイル群の実行部分中の分岐構文の行数

対象プログラム	if 文	switch 文	assert 文
simple_values.swift	266	22	33
control_flow.swift	392	28	43
functions_and_closures.swift	432	28	46
objects_and_classes.swift	449	28	37
enumerations_and_structures.swift	407	26	38
protocols_and_extensions.swift	407	27	35
generics.swift	436	26	43

表 5.8: TreeSwift における構文解析ファイル群の実行部分中の分岐構文の行数

対象プログラム	if 文	switch 文	guard 文
simple_values.swift	193	60	19
control_flow.swift	223	68	22
functions_and_closures.swift	206	69	21
objects_and_classes.swift	208	69	23
enumerations_and_structures.swift	219	71	26
protocols_and_extensions.swift	201	64	25
generics.swift	214	70	26

第6章　結論

本章では、本論文のまとめと今後の課題を示す。

6.1　本研究のまとめ

本研究では、Swift コンパイラのソースコード可読性を Self-host 化によって向上させることを目的として、Self-host 化した Swift コンパイラを実装し、その構文解析器について可読性を表す指標である実行部分 LOC を現行の Swift コンパイラと比較した。Swift のチュートリアル中で用いられている 7 つのサンプルプログラムを解析する場合の実行部分 LOC を測定した結果、特に Swift の持つパターンマッチなどの機能によって Self-host 化したコンパイラでは平均して 10.47%の実行部分 LOC の減少を達成することができていた。このことから、Self-host 化を行うことによって Swift コンパイラのソースコード可読性を向上させられることが分かった。

ただし、今回計測できた可読性の向上は Swift の構文が主要因となっている可能性が高く、他の既に高級言語でコンパイラが記述されている言語においては同様の手法を用いたとしても同じような結果を得られるとは限らないことには注意が必要である。

6.2　今後の課題

6.2.1　構文解析器以外の比較

構文解析以降の処理ステップについては、その処理が 1 つ前のステップの影響を受けるため、5.2 節で AST ファイル群について可読性が向上しているかどうかを判断できなかったように、本研究の手法ではうまく可読性の比較を行えない可能性が高い。

しかし、コンパイラの他の処理ステップには 4.2 節で述べたように構文解析器とは異なる様々なアルゴリズムやデータ構造が使用されており、その中には構文解析器のように Swift の構文による可読性の向上が充分に発揮されない箇所が存在する可能性もある。

そのため、実際に Swift コンパイラの全体を Self-host 化するか否かを判断するためには、他の処理ステップについても何らかの手法を用いて評価を行っていく必要があるだろう。

6.2.2 継続的な比較

Swift は現在も破壊的な変更を含む活発な仕様変更の相次ぐ言語であり、今後のバージョンの仕様で比較した場合は、可読性についての評価結果も変化する可能性がある。そのため、確かに Swift に対して Self-host 化が十分な可読性の向上をもたらすことを判断するためには、今後のバージョンでも継続的に同様の比較を行い、その変化を見る必要がある。

付 録A　TreeSwiftの実装における留意点

本研究で現行のSwiftコンパイラとの比較に用いたTreeSwiftは実際には本論文の動機として紹介されているSwiftのオープンソース化よりもずっと以前から実装が進められていた。つまりTreeSwiftはApple社の実装の詳細を知ること無くSwiftコンパイラを実装していたにも関わらず、結果的に構文解析器において同じ手法を採用していたということである。

本付録では、本論文の論旨とは一切関係ないものの、そうしたTreeSwiftの設計や実装を行う上で見つけた幾つかの気付きとそこから得た知見について、TreeSwiftの実装を解説しながら紹介する。本付録における情報は、本論文で提案しているようにSwiftコンパイラをSelf-host化する際はもちろん、Swiftを含む近代的な言語のコンパイラを設計する際などにも役立つだろう。

A.1　SwiftでSwiftコンパイラを記述する上での障壁

Apple社のSwiftコンパイラとTreeSwiftとの一番大きな差は実装開始当初から変わらず、Self-host化されているという点にある。しかし、Swiftはもともと Apple社の提供するフレームワークを用いた特定のプラットフォーム向けのアプリケーションを記述するためのドメイン特化言語のように発表されていた。そのため、そもそもSwiftにはコンパイラを実装するために十分な機能が用意されていない可能性があり、まずはSwiftコンパイラの各ステップで行われている処理を明らかにして、それらの処理がSwift言語でも実現できるかどうかを考察する必要があった。

実際にそうした考察を行うと、Swiftコンパイラで行われている処理の中にはSwiftだけでは実現が困難な特に大きな2つの処理が存在していた。本節ではそれらについて、その処理の実装が難しかった原因とTreeSwiftで取った解決方法を紹介する。

A.1.1　ファイル入出力

まず、最初に問題となるのはファイルの入出力である。コンパイラでは言わずもがな、プログラムのソースファイルを入力し、中間言語ファイルや場合によっては特定の形式のバイナリファイルなどを出力する必要がある。また、ほとんどのコンパイラではソースファイルにエラーや警告が見つかった場合にその原因箇所をユーザに知らせることを目的

として、字句や構文を表現する各オブジェクトにそのオブジェクトが対応するソースコード中の場所を保持させる。その情報から実際のコードをスムーズに参照しエラーなどと一緒に表示するには、ファイル内の特定箇所に直接アクセスするインターフェースが必要となる。しかし、これらの機能は Swift の標準ライブラリでは提供されていないか不十分であった。

そのため、TreeSwift では Apple 社が Swift コンパイラと一緒に提供している Darwin をベースとした環境の C 言語の標準ライブラリを Swift のライブラリであるかのように利用できる Darwin モジュールを用い、fread や fseek などの関数をラッピングしたクラスを作ってファイルの入出力処理を行っている。

A.1.2　LLVM の使用

次に大きな問題となるのは、Swift のバックエンドを担う LLVM の使用である。もちろん、LLVM はコンパイラのバックエンドを実現するための 1 つの方法でしか無いので、TreeSwift でもこれを採用しなければならないということはない。しかし、Swift の文法について詳細に見ると、その中には LLVM を使用することが前提であるかのように感じるものがいくつかあることに気づく。その典型的な例が、Swift のアクセスレベル修飾子と LLVM-IR のリンケージタイプとの類似性である。

表 A.1 に Swift の 3 つのアクセスレベル修飾子と LLVM の 3 つのリンケージタイプおよびそれらの意味を似ている物同士で並べて示した [41]。Swift におけるファイルを LLVM におけるモジュール、Swift におけるモジュールを LLVM におけるオブジェクトファイルと読み替えれば、それぞれのアクセスレベル修飾子とリンケージタイプが明らかな対応関係にあることが分かる。つまり、Swift のアクセスレベル修飾子は LLVM をバックエンドに用いた際にその実装が簡単になるように設計されていると見れる。

こうした類似点が他にも多数存在しているため、TreeSwift においてもそのバックエンドに LLVM 以外のプラットフォームを用いたり、バックエンドまで自作するような考え方はなかった。ただそうすると、LLVM のメインとなる API が C++で提供されており、C や Python 向けに提供されている API がその機能面において C++のものよりも劣っているという点が問題になる。

TreeSwift では最初、この問題に対して LLVM の C++用 API をラップした Objective-C のライブラリを作成し、そのライブラリを Swift から読み込むことで対処しようとした。Objective-C には C++と、Swift には Objective-C との相互運用性があるため、これらを組み合わせることで問題を克服できると考えていたからである。しかし、実際には Objective-C のカプセル化機構ではバックグラウンドで C++のライブラリ内に定義された C++のオブジェクトを用いながらも、それらを一切含まない純粋な Objective-C だけのインターフェースを備えたラッパークラス群を定義することが難しく、そのコストの高さからこのアプローチは不採用に終わった。

結果として、現在の TreeSwift では LLVM の C 言語用 API を直接用いて Swift から LLVM を使用している。LLVM の C 言語用 API において Swift コンパイラを設計する上で充分な機能が提供されているか、という点には未だ不安が残っているが、TreeSwift の実装と

表 A.1: Swift のアクセスレベル修飾子と LLVM のリンケージタイプの類似性

Swift のアクセスレベル修飾子		LLVM のリンケージタイプ	
名前	意味	名前	意味
private	修飾された要素は同一ファイル内からのみアクセス可能	private	修飾されたグローバル値は同一モジュール内からのみアクセス可能
internal	修飾された要素は同一モジュール内からのみアクセス可能	internal	修飾されたグローバル値は同一オブジェクトファイル内からのみアクセス可能
public	修飾された要素は外部モジュールからでもアクセス可能	available_externally	修飾されたグローバル値はどこからでも参照可能

同時期に作られた Ruby 言語に似たコンパイラ型言語である Crystal が Bootstrap するためにこの API を使用しているという例もある [42] ため、根本的な欠陥はないであろうと考え、実装に使用している。

A.2　Swift の文法と実装に対する示唆

Swift の文法はオープンソース化以前より公式ドキュメント上で BNF に近い形式の構文で公開されており、それを参照することで Swift のユーザは文法の詳細までを知れるようになっている [43]。しかし、そこで公開されている文法は単に Swift の文法を解説するためのドキュメントでしかなく、実装されている文法とは細部で異なる部分が多々存在している。そのため、TreeSwift を実装するにあたってはまずドキュメントの文法を全て書き写し、文法中の間違いや不適切な点を修正する作業を行う必要があった。

ただ、この修正作業のために Swift の文法を細かく確認したことで得られた Swift の文法に関する知見は多く、結果としてそれらの知見が構文解析器を実装する上で重要な幾つかの判断を助けることとなった。本節ではそうした知見の中から特に TreeSwift の設計に影響を与えた 2 つの例を紹介する。

A.2.1　文法の曖昧性と構文解析器の文法クラス

Swift の基本的な文法は C や C++、Objective-C などの俗に C-Family と呼ばれるプログラミング言語に類似しているが、それらの言語とは異なり、Swift の文法はプログラマにコンパイラが構文解析を行うための記述を強いないように注意深く設計されている。Swift では文の終わりを示す”;”や制御構文の条件式の前後に”(”と”)”を付けることを強制しな

いし、順に実行される手続きの集まりは制御構文内・関数定義内・無名関数内などのどこに書かれるかにかかわらず、”{”と”}”で囲うことで表される。しかし、こうしたプログラマ中心の設計は当然のことながら、他の C-Family 言語のコンパイラでは問題になっていなかった箇所で構文の曖昧性を生み、コンパイラの仕事を増やしている。

例えば、プログラム A.1 は Swift の文法に則った全く正しいプログラムだが、このプログラムを曖昧性無く解析するのは非常に難しい。2 行目の if から始まる文を解析するためには、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を判断するために、x の参照を解決した上でその型を知る必要がある。このプログラムにおいては直前に x の宣言があるために x の参照を解決することはできるが、x の型は明示的に注釈されていないため、これを推論する必要が出てくる。

プログラム A.1: 曖昧な構文を持った正しい Swift プログラム

```
1 let x = { (f: () -> Bool) in f() || true }
2 if x { print("closure␣argument") } {
3     print("if␣body")
4 }
```

ではここで、制御構文の条件式の後に”{”と”}”で囲まれた手続きの集まりが 2 度以上現れるような曖昧性のある構文に当たった場合だけ、条件式の型推論を先に行うように構文解析器を設計したとする。その設計を前提に考えると、今度はプログラム A.2 のような複数のファイルにプログラムが分かれている多少複雑な例を解析する際に問題が生じる。

プログラム A.2: 特定のアプローチではエラーの検出に多大な計算量を要する誤った Swift プログラム

```
1  // class.swift
2  class Sample {
3      let a = true
4  }
5
6  // function.swift
7  func f() {
8      let s = Sample()
9      if s.a {
10         print("if␣body")
11     } {
12         print("missing␣else␣keyword␣body")
13     }
14 }
15
16 // main.swift
17 f()
```

この例では 2~4 行目、7~14 行目および 17 行目がそれぞれ class.swift、function.swift、main.swift という 3 つのファイルに分かれているものとしているが、その 11 行目、function.swift 内の 5 行目に if 文に続く else 文に必要なキーワード”else”を書き忘れているというバグがある。そのため、実際にプログラムの解析を始めると、このバグにより現在

想定している構文解析器は 9~13 行目の文を曖昧性のある制御構文とみなし、s.a の型を推論しようと試みる。しかし、この変数 s のクラス Sample は class.swift という別ファイルで宣言されているため、そのクラスのインスタンス変数である a の型を解決するためには、function.swift よりも class.swift を先に解析しておく必要があり、ここで問題が生じる。Swift において main.swift と命名されたファイル以外のファイルについて名前の規定はなく、それらの解析順を誘導することはできないからである。つまり、現在想定している構文解析器においてこのプログラムから正しくエラーを検出するためには、曖昧性の原因となっている式とその式に関連するすべての項を正しく型付けできるか型付けできないことが明らかになるまで、main.swift 以外の全ファイルを繰り返し解析し直す必要があるのだ。この解決方法が解決したい問題に対して非常に複雑であり、かつ構文解析器の処理時間を必要以上に引き伸ばしてしまうことは明らかである。

そのため、こうした Swift の曖昧な構文では正確に解析することを諦め、曖昧性のある構文としてエラーにするか、どちらかの構文として決め打った解析のみを行うように構文解析器を設計する必要がある。ただ、こうした設計を実現するためには、構文解析器で採用できる実装手法が限られてくる。

まず、先述のような曖昧性のある文法 BNF などで記述し、その構文解析器を完全に自動生成することは現実的ではない。現在主に用いられている構文解析器の自動生成ソフトウェアは解析対象の言語に曖昧な文法が含まれず、最悪の場合でもプログラム中の全ての字句を先読みすれば適用する文法が決定できるという前提のもとに設計されているからである [34]。

さらに、たとえ一部であっても構文解析を手作業で書き下すとなると、 4.2.1 節でも述べたように、上向き構文解析を用いることは難しい。

そのため、近年のコンパイラで用いられているようなアルゴリズムを用いるのであれば、Swift の構文解析器は下向きかつ任意個の字句を先読みする LL(k) などに代表される文法クラスのものとなり、かつそれを完全に手作業で書き下すか、一部のみについて自動生成を用いる形となる。

そうした Swift の文法の持つ性質から考えて TreeSwift では LL(k) クラスの再帰下降構文解析を手作業によって書き下すこととなったので、Apple 社の Swift コンパイラにおいても全く同じアプローチの構文解析器が実装されていたという事実は驚くに値しないかもしれない。しかし、Apple 社の提供するドキュメントでは Swift の文法が下向き構文解析で解析不可能な左再帰を含んで記述されているところを見ると、文法の持つ性質を注視せずに構文解析器を設計し始めていた場合にはここで示したような問題と真正面からぶつかり、以上に非効率な構文解析器を実装してしまっていた危険性も十分にあったと考えられるのである。

A.2.2　変数の宣言とスコープ

A.3　参照解決・型推論・メンバ呼び出しの相互依存性

現時点の TreeSwift においてまだうまく実装ができていない Swift の機能として、意味解析のステップにおける相互に依存する参照・型・メンバ呼び出しの解決がある。

4.2.2 節でも言及したように、Swift ではオブジェクト指向プログラミングを採用し、型の省略と関数のオーバーロードを同時に許しているため、特定の関数の参照を解決するためには適切な順で型推論やメンバ呼び出しの解決、関連する他の参照の解決を先に行う必要がある。さらに、その内の型推論だけに限って見ても Swift はプロトコルと呼ばれる存在型やジェネリクスと呼ばれる全称型を利用できるようにしており、型同士の関係性を解決を行うプロセスまでもが非常に複雑化している。

これは、他の言語と比較してもかなり欲張った設計である。よく使われている言語の例で言えば、C++や Java、Scala は関数のオーバーロードを許すオブジェクト指向言語であるが Swift のように強力な型推論を提供していないし、OCaml や Rust といった関数型言語の色が強いオブジェクト指向言語では強力な型推論を提供している代わりに、完全な関数のオーバーロードを許してはいない。さらに比較して近年に開発された言語であっても、2005 年に登場した Haxe や 2011 年に登場したばかりの Ceylon は Ocaml などと同様に強力な型推論と引き換えに関数のオーバーロードをその仕様に含めていない [44] [45]。

そこで本節では、この意味解析の設計を Swift コンパイラを実装する上で特に難易度の高い箇所であると捉え、TreeSwift を実装する上で遭遇したこの意味解析における難題に繋がる 2 つの設計上の分岐点について紹介する。

A.3.1　参照解決のタイミング

A.3.2　メンバ呼び出しの型解決を行うタイミング

謝辞

本論文の作成にあたり、ご指導いただきました慶應義塾大学 環境情報学部教授 村井純博士、同学部教授 中村修博士、同学部准教授 Rodney D. Van Meter III 博士、同学部准教授 三次仁博士、同学部准教授 楠本博之博士、同学部准教授 植原啓介博士、同学部准教授 中澤仁博士、政策・メディア研究科特任准教授 鈴木茂哉博士、SFC 研究所 上席所員 (訪問) 斉藤賢爾博士に感謝致します。

研究について日頃からご指導頂きました政策・メディア研究科博士課程 松谷健史氏、政策・メディア研究科特任助教 空閑洋平氏に感謝致します。研究室に所属したばかりの頃から本研究に至るまで、特定の分野にこだわらない広い視点で何年生の時であっても妥協のない姿勢で向かい合い、絶えず多くのご指導をいただきました。本研究を卒業論文としてまとめることができたのも両氏のおかげです。重ねて感謝申し上げます。

研究テーマについて折に触れてご相談をさせていただいた政策・メディア研究科特任教授 (非常勤) 宮地恵美氏に感謝いたします。

研究室を通じた生活の中で多くの示唆を与えてくれた高橋俊成氏、原雅彦氏、沖幸太朗氏、山中勇成氏、河口綾摩氏、安井瑛男氏、黒米祐馬氏、および Arch 研究グループの皆様に感謝します。また、徳田・村井・楠本・中村・高汐・バンミーター・植原・三次・中澤・武田合同研究プロジェクトの皆様に感謝致します。

最後に、私の研究を支えてくれた両親と姉・義兄をはじめとする親族、多くの友人・知人に感謝し、謝辞と致します。

参考文献

[1] The open source definition. http://opensource.org/docs/osd, January 2016.

[2] 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.

[3] Objective-c - wikipedia. https://ja.wikipedia.org/wiki/Objective-C#.E6.AD.B4.E5.8F.B2, January 2016.

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

[5] Programming languages - tecosystems. https://redmonk.com/sogrady/category/programming-languages/, January 2016.

[6] James L. Elshoff and Michael Marcotty. Improving computer program readability to aid modification. *Communications of the ACM 25.8*, pages 512–521, 1982.

[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] Ted Tenny. Program readability: Procedures versus comments. *Software Engineering, IEEE Transactions on 14.9*, pages 1271–1279, 1988.

[10] Richard J. Miara, Joyce A. Musselman, Juan A. Navarro, and Ben Shneiderman. Program indentation and comprehensibility. *Communications of the ACM 26.11*, pages 861–867, 1983.

[11] 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.

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

[13] llvm-mirror/clang: Mirror of official clang git repository located at http://llvm.org/git/clang. `https://en.wikipedia.org/wiki/GNU_Compiler_Collectio://github.com/llvm-mirror/clang`, January 2016.

[14] Gnu compiler collection - wikipedia, the free encyclopedia. `https://en.wikipedia.org/wiki/GNU_Compiler_Collection`, January 2016.

[15] Visual c++ - wikipedia, the free encyclopedia. `https://en.wikipedia.org/wiki/Visual_C%2B%2B`, January 2016.

[16] Pypy - wikipedia, the free encyclopedia. `https://en.wikipedia.org/wiki/PyPy`, January 2016.

[17] .net compiler platform - wikipedia, the free encyclopedia. `https://en.wikipedia.org/wiki/.NET_Compiler_Platform`, January 2016.

[18] Free pascal - wikipedia, the free encyclopedia. `https://en.wikipedia.org/wiki/Free_Pascal`, January 2016.

[19] Gnat - wikipedia, the free encyclopedia. `https://en.wikipedia.org/wiki/GNAT`, January 2016.

[20] D-programming-language/dmd: dmd d programming language compiler. `https://github.com/D-Programming-Language/dmd`, January 2016.

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

[22] Go (programming language) - wikipedia, the free encyclopedia. `https://en.wikipedia.org/wiki/Go_(programming_language)`, January 2016.

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

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

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

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

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

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

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

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

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

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

[33] 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.

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

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

[36] 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.

[37] 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.

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

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

[40] 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.

[41] Llvm language reference manual - llvm 3.9 documentation. http://llvm.org/docs/LangRef.html, January 2016.

[42] crystal: The crystal programming language. https://github.com/manastech/crystal, January 2016.

[43] 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, January 2016.

[44] Method overloading - google グループ. https://groups.google.com/forum/#!topic/haxelang/feujdbvrrrQ, January 2016.

[45] Ceylon: Language design frequently asked questions. http://ceylon-lang.org/documentation/faq/language-design/#overloading, January 2016.