卒業論文 2015 年度 (平成 27 年)

Bootstrap に向けた Swift による Swift 構文解析器の設計と実装

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)

Bootstrap に向けた Swift による Swift 構文解析器の設計と実装

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 Bootstrap が行われている。Bootstrap を行うことによるメリットはいくつかあるが、度々モチベーションとしてあげられるのは、現存するプログラミング言語よりも Bootstrap を行おうと考えているプログラミング言語のほうが後発のものであるため、より表現力が高く開発しやすいという点である。

しかし、近年開発されている汎用プログラミング言語に至っては、その言語自体だけでなく最初にコンパイラを記述する言語も高級なものとなっており、対象のコンパイラを記述する上でどちらの方がより高い表現力や性能を持つかを簡単に判断することはできなくなってきている。

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 もそのメリットとデメリットを明確に評価することができず、Bootstrap するべきか否かの判断を下せていない汎用プログラミング言語の 1 つである。現在最も有名な Swift のコンパイラ実装は C++ で記述されており、コンパイラの核となる構文解析においても C++の特徴的な機能を駆使して、より低級な言語ではボイラープレートとなるコードを排除している。Swift はその可読性の高さと実行速度の速さを謳った言語であるが、その性能が Swift コンパイラという大規模なソフトウェアにおいて C++を相手としても通用するものであるかどうかを形式的に議論することは容易ではない。

そこで本研究では、Swift で記述した Swift の構文解析器を実装し、その性能とソースコードの行数を現行の Swift コンパイラ中の構文解析器と比較することで、Swift が Bootstrap を行うための判断材料を収集・考察する。本論文では、Swift で構文解析器を書き換えることによって可読性につながりうるソースコードの行数の削減は実現できるが、性能の面においては未だ Swift 自体が充分な性能を持っていない可能性があることを示し、その結果から Swift が Bootstrap を行うならば必要になるであろうステップについて考察を行っている。

キーワード:
1. コンパイラ・ブートストラップ, 2. 構文解析, 3. 構文解析器の実装,
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

Design and Implementation of Swift Parser Written in Swift for Bootstrapping

English abstract here.

Keywords :
1. Bootstrap a Compiler, 2. Parser, 3. Implementation of Parser,
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 社が予てより同社の提供する 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として提供していたプログラミング言語 Swift をオープンソース化し、Linux を中心としたさまざまなプラットフォームにおけるソフトウェアを開発するための拡張を開始した。これによりプログラミング言語 Swift は Objective-C の担ってきた iOS や Mac OS X などにおけるソフトウェアの開発だけでなく、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た。

Swift はオブジェクト指向や型パラメータ多相の導入、関数の第一級オブジェクト化、Hindly と Milner による型推論アルゴリズムの採用など、現在多くのプログラマに使用されている他の汎用高級言語が持つ様々な特徴を持っているが、まだその特徴を採用するか否かがよく議論されていないものもある。その内の 1 つが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 Bootstrap プロセスの採用である。

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 [1] の 2015 年 12 月版において上げられている言語の内、汎用言語であるものだけを上位から 20 言語抽出し、それらの主要なコンパイラがその言語自体で記述されているかを示したものである。この 20 言語の内だけでも Bootstrap を行っているものが 8 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 Bootstrap されていることが分かる。しかし、現在 Swift は C++を用いて開発されており、Bootstrap を行っていない。

開発者のメーリングリスト [2] では、現在の Swift においては未だ多くの機能が不足しているために他の問題を優先していること、Swift コンパイラのバックエンドとして採用されている LLVM の API が C++で提供されており、その Swift への移植が簡単ではないことから、Swift の Bootstrap について議論を行うためにはもう少し時間が必要だという意見も上がっている。

しかし、 2.2 節で述べるように Bootstrap を行うことで得られる利益があることが他の言語の事例によって示されている以上、十分な議論なしに現状の Swift に Bootstrap が不要であると判断するのは早計であるといえる。

1.2 本研究が着目する課題

Swift コンパイラが抱える他の問題との優先度や使用しているフレームワークとのつなぎ込みに関する問題が解決したとしても、Swift コンパイラの Bootstrap を行うかどうかという判断を下すにはより根源的な課題がある。それは、現在 Swift を記述している C++ 言語が Swift と比較しても高い性能と表現力を持っているために、Bootstrap を行うことで得られるメリットや、そもそも現行のコンパイラで使用されている手法や現行のコンパイラが持っている性能を維持したまま Bootstrap を行うことが可能であるか否かが自明でないというものである。

この現状から、Swift が Bootstrap を行った時に得られる実利益があるのかどうかは実際に Swift で Swift コンパイラを実装してみなければわからない点が多い。

1.3 本研究の目的

本研究では、Swift コンパイラが Bootstrap することによって得られるメリットと被るデメリットの内、実装がなければ判断しがたい部分を定量的に示し、Bootstrap を行うべきか否かを判断する上で有用な情報を収集することを目的とする。そのためのアプローチとして、Swift コンパイラの基幹的機能である構文解析器を Swift によって実装し、その実行時間と実行時のメモリ使用量、ソースコードの行数を現行の Swift コンパイラの構文解析器と比較する。また、この独自の構文解析器は現行の Swift コンパイラと基本的な設計手法において同じものを採用するだけで、完全に独立させたものとして実装する。

この方法により、現在の Swift コンパイラの開発状況などの影響を一切受けずに Bootstrap のための評価が可能となり、またその評価が Bootstrap の可能性に対して有意義な知見を与えることを提示する。

1.4 本論文の構成

本論文の構成は次の通りである。

第 2 章では本研究の考察対象であるコンパイラの Bootstrap について Swift 以外の言語の事例からそのメリットについてまとめ、Swift の Bootstrap における利点と課題について整理する。第 3 章では本研究が着目するプログラミング言語 Swift の特徴とそのコンパイラ実装の基幹部分における特徴について説明する。第 4 章では現行の Swift コンパイラとの比較対象となる Swift で記述した Swift コンパイラ「TreeSwift」の構成について述べ、Bootstrap を行うべきか否かを判断するために必要な要件を満たしていることを確認する。第 5 章では現行の Swift コンパイラと TreeSwift の構文解析器についてその実行速度と実行時のメモリ使用量、ソースコードの行数を比較し、その結果について考察する。第 6 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	コンパイラ名	Bootstrap 状況
1	Java	javac	N (C, C++?)
1	Java	OpenJDK	N (C++, Java)
2	C	gcc	N (C++)
2	C	clang	N (C++)
3	C++	gcc	Y
3	C++	clang	Y
3	C++	Microsoft Visual C++	Y
4	Python	CPython	N (C)
4	Python	PyPy	Y
5	C#	Microsoft Visual C#	N (C++)
5	C#	.NET Compiler Platform	Y
6	PHP	Zend Engine	N (C)
7	Visual Basic .NET	Visual Studio	N (C++, C#)
7	Visual Basic .NET	.NET Compiler Platform	Y
8	JavaScript	SpiderMonkey	N (C, C++)
8	JavaScript	V8	N (C++, JavaScript)
9	Perl	perl	N (C)
10	Ruby	Ruby MRI	N (C)
12	Visual Basic	Visual Studio	N (C++, C#)
13	Delphi/Object Pascal	Delphi	N (?)
13	Delphi/Object Pascal	Free Pascal	Y
14	Swift	swift	N (C++)
15	Objective-C	clang	N (C++)
15	Objective-C	gcc	N (C++)
17	Pascal	Free Pascal	Y
17	Pascal	GNU Pascal	N (C, Pascal)
20	COBOL	GnuCOBOL	N (C, C++)
21	Ada	GNAT	Y
22	Fortran	GNU Fortran	N (C, C++)
22	Fortran	Absoft Fortran Compiler	?
23	D	DMD	Y
24	Groovy	groovy	N (Java, Groovy)

第2章　コンパイラのBootstrap

本章では、これまでにBootstrapを行ってきた高級汎用言語の事例を紹介し、それらの例からSwiftのBootstrapにおける利点と課題について整理する。

2.1　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。

2.1.1　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[3]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、 [3] 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 [4]。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 [5] 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。

2.1.2 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 [6] 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 [7]。

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 [8]。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。

2.1.3 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 [9]。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 [10]。

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 [11] 内で述べている。

2.2 Swift における Bootstrap の利点

表 2.1 に 2.1 節で述べた事例から Bootstrap における利点をまとめ、Swift の Bootstrap 時にそれらの利点が得られる可能性があるか否かをまとめた。

表 2.1: Swift でも享受できる可能性のある Bootstrap の利点

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

Swift で記述されたプログラムは現行のコンパイラで使用されている C++で記述されたものと比較して、記述やデバッグが容易になる可能性が高い。詳細は 3 章で述べているが、メモリの管理を自動的に行う ARC 方式や強力な型推論と型検査は C++で必要となる冗長な記述を省略し、多くのエラーをコンパイル時に発見する。ただし、飽くまでもこれはプログラムの記述の容易さだけに関する議論であり、 2.1.1 節で見たように、より簡単な記述で同等の性能が得られるとは限らないという点に注意されたい。

コンパイラの開発を行うために必要な知識が減るという点は 2.1.2 節で述べた PyPy のような形でなければほとんど全ての Bootstrap に対して有効である。Swift はコンパイラ型言語であるため、よほど特殊な方法を採用しない限りはこのメリットを享受できると考えられる。

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

2.1.3 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。

最後に、言語の柔軟性と拡張性だが、これについては Swift と C++の言語仕様を見比べているだけでは判断できない。採用しているプログラミングパラダイムや構文の種類であれば C++は Swift を上回っているが、ボイラープレートが多く 1 つの変更によって修正しなくてはならない箇所が増えたり、逆にボイラープレートを無くすために特殊な構文を駆使することで拡張性を損なったりすることもある。それに対して利用可能な機能の種類が少ない Swift では、そもそも C++では可能な拡張を取り入れられない可能性がある。

この点について完全な結論を得ることは難しいが、コンパイラの一部分でも実装を比較することができれば、より具体的な予想を行うことは可能である。本研究では実際に実装された Swift による Swift コンパイラと現行の C++による Swift コンパイラのソースコードの行数の比較から、このメリットを得られるかどうかに関するある程度の判断材料が得られることを示す。

2.3 Swift における Bootstrap の課題

表 2.2 に 2.1 節で述べた事例から Bootstrap において発生しうる課題をまとめ、Swift の Bootstrap 時にそれらの課題が問題となるかどうかをまとめた

表 2.2: 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、 2.1.3 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、 2.1.1 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、 2.1.1 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 2.1.3 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、 2.2 節でも述べたように Swift 自体がコンパイラ型言語であるため、起こりづらいと考えられる。

そして最後に、コンパイル速度の低下は 2.1.1 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、 2.1.3 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、この点を明らかにするために Swift で書かれた Swift コンパイラと現行の C++による Swift コンパイラの実行速度と実行時のメモリ使用量を比較し、Bootstrap を行った際にコンパイラの性能がどう変化し、どういった箇所に性能の低下や向上が起こりうるかを示す。

第3章　プログラミング言語Swift

本章では、本研究の対象であるプログラミング言語 Swift の言語的特徴と Swift コンパイラの構成について述べる。

3.1　Swift の特徴

2.2 節および 2.3 節で述べたように、Swift は他の言語と比較しても多くの特徴を備えており、それがこのコンパイラの複雑性を増しているために Bootstrap における費用対効果の試算を困難にする原因となっている。

本節ではそうした Swift の特徴について概説する。

3.1.1　マルチパラダイム

プログラミング言語が採用するプログラミングパラダイムによって、その言語のコンパイラの設計、特に構文解析器とコード生成部分の設計は大きな影響を受ける。Swift は近年の汎用言語に採用されている多くのプログラミングパラダイムを取り入れているマルチパラダイムプログラミング言語であり、以下のようなプログラミングパラダイムを採用している。

関数型プログラミング

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

Swift における関数型プログラミングの例をプログラム 3.1 に挙げる。関数用の型が矢印演算子によって提供されており、関数自体を変数に代入して使用できていることがわかるだろう。

プログラム 3.1: Swift における関数型プログラミングの例

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

このパラダイムにより、コンパイラではC や Java、C++と比較して関数のために無名関数や部分適用といったより多くの構文を用意し、一般的に関数を第一級オブジェクトとして扱わない事が多いアセンブリ言語へは関数ポインタなどを使用することでそれらの構文を翻訳する必要がある。

オブジェクト指向プログラミング

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

Swift におけるオブジェクト指向プログラミングの例をプログラム 3.2 に挙げる。この例では継承関係のあるクラスから生成されたオブジェクトに対し、クラスで定義された関数のメンバを呼び出している。

プログラム 3.2: Swift におけるオブジェクト指向プログラミングの例

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

このパラダイムを実現するために、コンパイラでは各複合型ごとのスコープの管理が必要になり、継承関係のある型同士を部分型として扱った上で、仮想関数テーブルなどを用いて実行時に呼び出すメンバを動的に決定できる仕組みを生成する必要がある。

パターンマッチ

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

Swift におけるパターンマッチの例をプログラム 3.3 に挙げる。現在の Swift ではこの例内の 6 行目のような列挙体やタプルの値について特に柔軟なパターンマッチを提供している。

プログラム 3.3: Swift におけるパターンマッチの例

```
1  enum Sample {
2      case X, Y(Int)
```

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

このパラダイムの実現には、コンパイラで左辺値のパターンが表す構造と型を解釈し、右辺値の値をより詳細な構造に分解してより単純な同値性を確認する演算や変数の宣言の集まりに変換する必要がある。

3.1.2 強力な型システム

強力な型システムはプログラマのミスを発見する有効な手立てとなり得るが、一般的にコンパイラの型推論や型検査にかかる時間との間でトレードオフの関係となる。以下では Swift で採用されている型システムの特徴について述べる。

型パラメータ多相

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

コンパイラでは全称型を持つ複合型や関数をその文脈によって決定された型で具体的に定義しなおし、それらの具体的に定義された型を同一の型として扱う必要がある。この機能の実装は後述する型推論において型の解決が可能かどうかを決定するためのステップを大幅に増加させてしまうため、コンパイル時間との兼ね合いを考えてより絞り込んだ制約を要求するなど、状況に合わせたサポート機能の組み込みが要求される。

関数のオーバーロード

Swift は引数の数が異なる関数や引数の型が異なる関数を同じ名前で定義できる、関数のオーバーロード機能を有している。

この機能によってコンパイラでは、どの関数が呼びだされているかを決定する前にその関数の型を決定するステップを追加しなければならなくなる。ただし、型推論による型の決定には関連する項の参照が解決されて型注釈などが把握されている必要があるため、関数呼び出し以外の項については型を決定する前に参照を解決する必要があり、オーバーロードの機能を追加すると関数呼び出しと他の項を同一の手順で参照解決・型解決できなくなるという複雑性を含まざるを得なくなる。

型推論

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

Hindly と Milner によるアルゴリズムは ML や Haskell など多くの関数型プログラミングを採用する言語で採用されているが、前述のとおりオブジェクト指向や型パラメータ多相、関数のオーバーロードを許すという特徴から、Swift コンパイラではそれを大きく拡張した形で実装しなくてはならない。

3.1.3 高い可読性

Swift がマルチパラダイム性と強力な型システムの他にその言語の特徴として大きく掲げているのが、高い可読性である。Swift では未定義値を安全に取り扱うためのオプショナル型や無名関数を定義するためのクロージャ、複数の値をまとめて扱えるタプル、自由なデータ構造を記述可能な列挙体などについてその可読性を高めるための糖衣構文を用意している。

また、糖衣構文などを提供するために構文の曖昧性を残しているのも Swift の特徴である。例えば、プログラム 3.4 のような構文は Swift の構文定義に則った全く正しいコードだが、一般的なアプローチでは正しく解析することができない。x の後に続く { が x の引数のクロージャの始まりなのか if 文の本体の始まりなのかは、x が引数としてクロージャを取る関数であることを解析機が知っていなければ決定できず、かつその x はこれから解析を行う他のファイルなどに宣言されている可能性があるために、全てのファイルの解析を完了するまで x の参照を解決することはできない可能性があるためである。

プログラム 3.4: 曖昧な構文を持った正しい Swift コード

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

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

3.2 Swift コンパイラの構成

3.1 節で述べた特徴を実現するために、Swift コンパイラではその詳細な設計において多くの工夫が行われている。本節ではその詳細な工夫について本研究で評価の対象としている構文解析器のみについて説明し、その他の箇所については概要を述べるだけに留める。

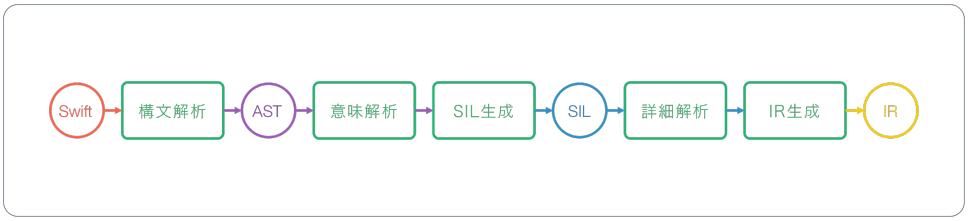

図 3.1: Swift コンパイラの構成

3.2.1 Swift コンパイラの概要

Swift コンパイラは大まかに図 3.1 のような構造を持つ [12]。

構文解析の詳細については 3.2.2 節で詳しく述べるが、Swift コンパイラは構文解析時に可能な限り変数や型などの参照を解決し、続く意味解析で不明だった参照の解決と明示されていない型の推論、型の整合性の検査を行う。

また、コンパイラの構成で特に特徴的なのは意味解析と LLVM-IR のコード生成との間に SIL(Swift Intermediate Language) という独自の中間表現を挟んでいる点である。Swift コンパイラでは、この SIL 形式で AST(Abstruct Syntax Tree) では解析しづらいコールグラフやクラス階層に基づくエラーの検出やメモリ管理用コードの挿入、最適化などを行っている。

SIL 形式が複雑な解析部分を担っているために Swift コンパイラの構文解析器と AST は意味解析より後のステップで更に改変されること無く、完全に分離されている。そのため、コンパイラ中の構文解析部分のみを抽出して他の実装と比較評価を行うことが可能となっている。

3.2.2 Swift コンパイラの構文解析器

3.1 節で述べたように Swift は多くの特徴を持っているため、その構文は他の言語と比較しても複雑なものとなっており、構文解析には高い柔軟性が求められる。以下では Swift コンパイラの構文解析器における各機能についてその特徴を述べる。

LL(k) 方式構文解析

現行の Swift コンパイラでは再帰下降構文解析の一種である LL(k) 方式を構文解析に採用しており、その構文解析器は C++の関数として書き下されている。

LL(k) 方式では解析する対象の構文によって解析器が先読みを行うトークンの数を変更することで、各構文に必要な先読みの数を最小限に抑えながらも、 3.1.3 節で取り上げたような完全な曖昧性を持つ構文以外は必ず解析対象の構文を決定することができる。

また、単なる LL(k) 方式の構文解析器が自動生成が可能であることは [13] などで知られているが、Swift の場合は先述の曖昧な構文に対する対処やより詳細なエラーの報告を行うために、構文解析器を全て手動で書き下している。

参照解決

現行の Swift コンパイラでは構文解析時に可能な限り変数や型の参照を解決し、他のファイルで宣言されている可能性のあるものなど、その時点で解決不可能なものについては解決不可能としてマークして AST に組み入れる。解決が不可能であった参照は全てのファイルを走査し終えた後の意味解析の最初のステップで解決される。

AST

曖昧な構文の解析

Swift コンパイラの構文解析器では、構文定義的に参照の解決などが行われた後でなければどの構文なのかを判断することができない曖昧な構文については、ひとまず特定の構文として解析して意味解析時に適切な形式でなかった場合にのみそれを変換する。

第4章　TreeSwiftの設計と実装

本章では、 3 章で述べた現行の Swift コンパイラの特徴から比較対象となる Swift で記述した Swift コンパイラ TreeSwift の構文解析器が満たすべき特徴を整理し、それがどのように実装されているかについて述べる。

4.1　構文解析器が満たすべき特徴

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

LL(k) 方式の採用

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

エラー文の分離

エラー文の書き方によるソースコードの分量の変化を防ぐため、エラー文自体は構文解析器自体と別のファイルに定義し、それを参照する形を取る。

構文解析以外の部分の分離

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

4.2　実装の概要

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

4.2.1 構文解析機能

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

字句解析

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

構文解析

解析に使用する構文は Apple 社の提供する Swift の公式ドキュメント [14] にある構文をベースとするが、本資料は多くの誤りを含んでおりかつ LL(k) 形式では解析不可能な左再帰を含んでいるため、それらを修正した独自の構文定義を使用している。

また、TreeSwift では 3.2.2 節で述べていたような現行の Swift コンパイラが判断できていない構文についても、独自の構文定義の追加によって構文的に分類・区別している。そのため、TreeSwift では全ての構文について構文解析が完了した時点で曖昧性が排除されている。

4.2.2 参照解決機能

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

参照解決のタイミング

TreeSwift では 3.2.2 節で述べた現行の Swift コンパイラの参照解決とは異なり、構文解析中には一切の参照解決を行わない。そのため、構文解析が完了して参照解決を行う際にはすでにすべての参照解決が可能であることが保証されており、参照解決を構文解析の本体から分離することが可能となっている。

スコープ解決

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

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

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

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

4.2.3 構文解析器のその他の機能

エラーの処理

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

ライブラリの扱い

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

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

4.2.4 構文解析以外の実装

4.3 解析可能な構文の検証

本節では、TreeSwift を用いて実際に構文の解析を行った結果によって TreeSwift が十分な構文解析機能を持っていることを示す。

4.3.1 検証するプログラム

4.3.2 検証の結果

4.3.3 検証のまとめ

第5章　評価

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

5.1　評価概要

評価の目的は 2.2 節および 2.3 節で述べた Swift における Bootstrap のメリットと課題の内、言語仕様からだけでは判断のつかなかった点を考察する材料を集め、有意義な考察を得ることである。そのために、本研究では表 5.1 に示す 3 種類の比較を行う。

表 5.1: 評価のための比較項目

構文解析器のソースコードの行数	各コンパイラのソースコードの内、ファイルを読み込み、字句解析および構文解析を行い、AST を生成するまでに実行されるプログラムを記述した部分からエラー分を定義している箇所以外を抜き出し、その行数を計測・比較する。
構文解析の実行時間	Swift の異なる構文を使用した複数のプログラムを各コンパイラでコンパイルし、その構文解析にかかる時間を各コンパイラの構文解析開始箇所および終了箇所に埋め込まれた時間取得プログラムの返す時間の差分によって測定・比較する。
構文解析時の最大使用メモリ量	Swift の異なる構文を使用した複数のプログラムをプロファイラによってプロファイリングしながら各コンパイラでコンパイルし、その構文解析終了時点でプログラムを強制終了することで、構文解析完了までの間で同時に確保したメモリの最大量を測定・比較する。

本研究で実装した TreeSwift は、 4.3.2 節で示したとおり現行の Swift コンパイラと同等の構文解析機能を保持しており、 4.1 節で述べたとおりそれらの機能が同様の手法で実現されている。そのため、TreeSwift と現行の Swift コンパイラにおいては構文解析器のソースコードの行数がより少ない方が、ある機能を修正・拡張する時に変更しなければならないソースコード中の行数は少なくなると考えられる。このことから、TreeSwift の方が構文解析器のソースコードの行数が少ないかどうかを評価することにより、 2.2 節で述べた Bootstrap によって柔軟性と拡張性を得られるかどうかを判断する。

2.3 節で述べた Bootstrap によってコンパイラの性能の低下が起こるかどうかについては、構文解析の実行時間と構文解析時の最大使用メモリ量を比較することで判断する。一般にプログラムの実行速度とメモリ使用量はトレードオフの関係にあるため、それらの両方について TreeSwift と現行のコンパイラの間に大きな差が見られない場合か両方について TreeSwift の方がうわ待っている場合に限り、Bootstrap による性能の低下はないと考えることができる。

5.2 計測

5.2.1 構文解析器のソースコードの行数

5.2.2 構文解析の実行時間

5.2.3 構文解析時の最大使用メモリ量

5.3 考察

第6章　結論

6.1　本研究のまとめ

6.2　今後の展望

謝辞

参考文献

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

[2] [swift-dev] bootstrapping swift compiler. https://lists.swift.org/pipermail/swift-dev/2015-December/000004.html, December 2015.

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

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

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

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

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

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

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

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

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

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

[13] T. J. Parr and R. W. Quong. Antlr: A predicated-ll(k) parser generator, 1995.

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