電子通信情報系コアテキストシリーズ C-1

実践コンパイラ構成法

滝本 宗宏
著

コロナ社

刊行のことば

産業革命には，エネルギーを基軸に段階分けする立場と，産業のインフラ要素から情報化を含めて段階分けする立場がある。1860年代から始まったとされる第2次産業革命はエネルギー源としての「電気」を基軸に置く議論が一般的である。ところが，明治政府はそのような分類学を超越し，電気の効能は通信にあると見切っていた。実際，明治4年（1871年）には東京・ロンドン間で電信網を完成させ，その開発・運用に必要な技術者養成を目指して，明治6年に工部省工学寮電信科を創設している。本シリーズのテーマである電気・電子，通信と情報に関する日本最初の学校である。東京に電灯が灯ったのが1882年であるから，その10年以上前に通信網を完成していたわけである。一方，ケンブリッジ大学は1871年に電磁気現象に物理学の未来を夢見てキャヴェンディッシュ研究所を設立している。今から考えると，どちらの「電気」の見方も正しかったが，より産業的な実利を得たのは日本だったといえよう。現代は第4次産業革命のただ中にあるといわれ，日本の立ち遅れを叱責する声が大きい。ただし，そのように外国がやっていることをただ真似るのだとしたら，明治政府は物理学研究所を作っていたはずである。彼らは，後世にいわれるほど西洋の物まねに機械的に熱中していたわけではない。彼らなりの戦略眼があったと見るべきである。

電気・電子，通信そして情報は，以来，工学の主要な分野を形作ってきたが，特に第2次世界大戦後は，電子工学に代表される工業製品や生産設備の刷新を経て，1990年代以降の情報通信社会を導いている。これはコンピュータの性能の急速な向上と，光通信に代表される通信網の急速な高速化に支えられたイ

ンターネットの出現に負うところが大きい。太平洋横断海底通信ケーブルを例にとると最初の光ケーブルだった TPC-3（1989 年）の 560 Mbps と比較して FASTER（2016 年）の 60 Tbps では約 11 万倍の高速化が達成されている。この結果，全世界の情報を一瞬に集め，これまでにない速度で処理する，いわゆるビッグデータの時代が到来している。今や，SNS (Social Networking Service) などに代表されるように，我々の活動は様々なディジタルメディアに書き込まれるようになっている。我々は梅棹忠夫が数十年前に予想した情報環境の中で生活するようになったともいえる。20 世紀までの歴史研究の「書物」がそうであったように，ディジタル・メディアに堆積された「情報」こそ，これからの歴史を語る際の基本資料であるといえる。

そこで今回，これから技術者を目指す電気・電子・情報系学部生また高専生向けの教科書シリーズ「電子通信情報系コアテキストシリーズ」を立ち上げた。本シリーズは，電気・電子分野（A），通信分野（B）そして情報分野（C）と三つの分野に分け，多くの大学で講義されている科目を厳選し，実際に講義を担当している先生を執筆者とし，これからの教育現場に合った教科書を目指している。

本シリーズで勉強した学生が，若者の目で，上記のような 2010 年代における価値観から技術を再整理する一助になれば幸いである。

2017 年 5 月

編集委員長　浅見　徹

まえがき

言語処理系の研究の歴史は，計算機科学の分野の中でも，特に古い部類に入る。その長い歴史の中でも，最近のコンパイル技術の発展には目を見張るものがある。プログラミング言語に新しい概念が導入されるたびに新たなコンパイル手法が提案されてきたのはもちろんであるが，コード最適化に代表されるコンパイラ特有の技術も目覚ましい発展を遂げてきた。

一方，コンパイラの理論と実装の間のギャップはさらに広がったように見える。なぜなら，コンパイラの理論は，長年かつ多岐にわたるアルゴリズムの集積によってさらに複雑化しており，その実装は，新たな原始言語あるいは目的機械から生ずる多くの例外を，統合するよう求められるからである。大学や大学院における授業のなかで，このギャップを埋めて実践的なコンパイラのイメージを伝えることは，さらに難しくなった。

それでも，すでにコンパイラの知識を身に付けた研究者にとって，自分が組み立てた理論や手法のコンパイラ上への実装は，容易になったように見える。それは，コンパイラ共通基盤のような，実装の難しさをモジュール化しながら，一部の改編によって研究成果を得ることができる多くのツールや方法が提供されるようになってきたからである。

同様な方法を，基礎を教える授業に適用するのは難しい。それは，アルゴリズムとその実装を直接結び付けて解説する必要性があることから，モジュール化をうまく利用できないからである。

そこで，従来から，自動生成系を導入することによって，その入力である宣言的な表現でコンパイラの一部を記述する試みがなされてきた。この試みは，授

業の理解を深めるとともに，授業内でのコンパイラの実装を容易にするのに役立った。しかしながら，依然として，自動生成系の意味動作や，コンパイラのほかの部分は，C 言語や Java で記述されることが多く，自動生成される部分とそれ以外の部分との間には大きなギャップが残されていた。

本書は，自動生成系を利用するとともに，コンパイラを記述するプログラミング言語に，OCaml という関数型言語を採用している。OCaml は，型付きの強い言語でありながら，型推論機構によって，型を指定する必要がほとんどない。さらに，組やリストといったデータ構造を表現する構文を備えており，新しいデータ構造も容易に定義することができる。この OCaml の勘弁な記述によって，例えば，アルゴリズムの説明を行う際にも，仮想言語を用いることなく，OCaml の記述を直接に見せて進めることもできるであろう。

そうはいっても，OCaml は，C 言語や Java 言語のように，多くの大学で教えられているプログラミング言語というわけではない。多くの場合，コンパイラの授業のために，わざわざ新しいプログラミング言語を学ぶということになるかもしれない。そのような場合のために，本書の最初には，プロジェクトを実施するために必要最低限の OCaml への入門を載せた。

本書のもう一つの特徴は，機械コードとして x64 コードを採用していることである。一般性を排し，特定の多くの PC で動作できるコードを採用することによって，具体性を高め，課題や自主開発を通じた独自の発展を促すことが狙いである。なるべく学生自身の PC で動作確認ができるように，Linux，Mac OS X，Cygwin で動作するように心がけた。

本書中で紹介したソースコード，また内容をよりよく理解するための練習問題は Web†で提供するので，是非活用していただきたい。

本書の内容は，著者がここ数年，東京理科大学，慶應義塾大学，東京工業大学の学部学生を対象に担当してきたコンパイラの講義資料をベースにしている。OCaml については，東京理科大学の情報科学科の学生が，演習をとおして使用

† コロナ社の書籍詳細ページ（http://www.coronasha.co.jp/np/isbn/9784339019339/）の関連資料からダウンロードのこと。

した経験があるだけであった。また，慶應義塾大学と東京工業大学では，関数型言語を使用するのも初めてという学生が多かったが，実装課題においては優れたレポートが多く見られたことを付け加えておきたい。

2017 年 5 月

滝本 宗宏

目　次

1章 はじめに

◆本章のテーマ

本書は，プログラミング言語で書かれたプログラムから，身近な機械のうえで動作可能な機械コードを生成するまでの過程を，直観的にかつ実践的に解説している。

コンパイラは，異なる表現に変換するフェーズという単位で構成されているが，本書の解説の流れも，フェーズの並びの順序にほぼ一致している。

本章では，以降の流れを，コンパイラの構成とともに概観する。コンパイラの実装プロジェクトを進めるうえで重要な，開発ツールと記述言語についても触れる。

◆本章の構成（キーワード）

1.1 言語処理系

1.2 コンパイラ

コンパイラの構成，開発ツールと記述言語

◆本章を学ぶと以下の内容をマスターできます

☞ コンパイラとインタプリタの違い

☞ コンパイラの構成

☞ コンパイラの開発ツール

1.1 言語処理系

プログラミング言語で書かれたプログラムを処理するソフトウェアを**プログラミング言語処理系**，あるいは単に**言語処理系**という。言語処理系は，コンパイラとインタプリタに大別できる。

プログラムを，機械コードやほかのプログラミング言語で書かれたプログラムに変換するシステムを，**コンパイラ** (compiler) という。なお，ほかのプログラミング言語で書かれたプログラムに変換するものを**トランスレータ** (translator) と呼んで区別する場合もある。

一方，プログラム中の式や文がどのような命令列に対応するかを実行中に解析して，即座に実行するシステムを**インタプリタ** (interpreter) という。コンパイラが，事前にプログラムから変換した機械コードを実行する形式なのに対して，インタプリタは，実行中に対応する命令列を解析するので，実行効率が悪いという傾向がある。

また，Java のように，プログラムを解析する手間がほとんどないバイトコードにコンパイルし，そのバイトコードを仮想機械で実行するものもある。仮想機械が仮想的な機械を実現したインタプリタであることを考慮すると，この実行形式は，コンパイラとインタプリタの両方の側面を持つといえる。すなわち，いったんバイトコードにコンパイルすれば，実行効率を大きく損なうことなく，どのような実行環境であっても，仮想機械さえ備わっていれば実行することができるのである。

このように，コンパイラとインタプリタとでは実行の仕方や特徴が異なるものの，プログラムを解析する部分は同じである。本書は，おもにコンパイラを念頭に解説するが，その多くの部分は，インタプリタの開発にも応用できる。

1.2 コンパイラ

コンパイラは，計算機上で実行可能なプログラムを自然言語に近い言葉で記

述したいという欲求から，プログラミング言語を機械コードへ変換するソフトウェアとして 1950 年代に誕生した。コンパイラの研究と開発は，当初，その実現に努力が注がれ，プログラミング言語の定式化と標準化とともに，実用的なコンパイラの実現に向けて基礎理論およびプログラム解析・生成技術が整備されてきた。現在では，一部を除いてほとんどの部分を自動生成することが可能になり，その自動生成ツールも広く使用されるようになっている。

一方，コンパイラが生まれた当時から，より優れた機械コードの生成を目指す研究も続けられてきた。この優れたコードを生成する技術は，**コード最適化** (code optimization)，あるいは単に**最適化** (optimization) という[1),2)†]。現在，その努力は，自動的にあるいは指示子を用いてプログラムを**並列化** (parallelization) する技術も含め，コンパイラ研究の中心課題の一つになっている。

1.2.1 コンパイラの構成

コンパイラは，入力として特定のプログラミング言語で書かれた**原始プログラム** (source program) を受け取り，プログラミング言語の構文と意味に基づいて，対応する**目的プログラム** (target program) を生成する。目的プログラムは，特定の機械コードや仮想機械コードであることが多いが，ほかのプログラミング言語で記述されたコードであってもよい。原始プログラムから目的プログラムが生成される過程は，**図 1.1** に示す一連の**フェーズ** (phase) からなり，個々のプログラム表現は，各フェーズによって，異なる表現に変換される。

これらのフェーズは，プログラムの解析を行う**フロントエンド** (front end) と，プログラムの合成を行う**バックエンド** (back end) に大別できる。

フロントエンドは，つぎの三つのフェーズで構成される。

① **字句解析** (lexical analysis) フェーズ

② **構文解析** (syntax analysis または parsing) フェーズ

③ **意味解析** (semantic analysis) フェーズ

† 肩付き数字は，巻末の引用・参考文献の番号を表す。

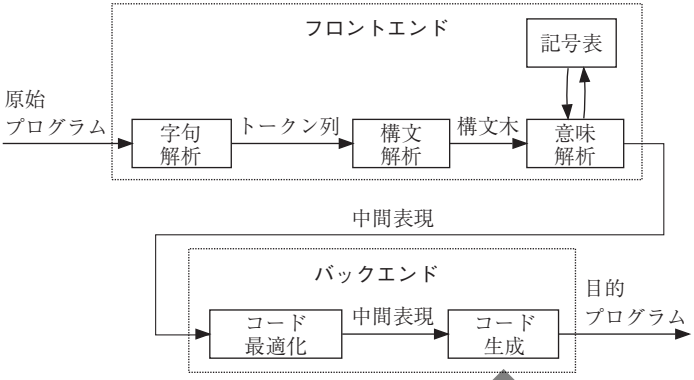

図 1.1　コンパイラの構成

字句解析フェーズは，文字の並びである原始プログラムを，意味のある文字の列，すなわち**トークン** (token) **列**へ変換する。つぎに構文解析フェーズは，構文規則に基づいて，トークン列から構文のパターンを見つけ，構文に対応する木表現である**構文木** (syntax tree) に変換する。構文解析フェーズが構文木を作成する際には，意味解析と連携して，字句有効範囲に基づく変数宣言の解決や型チェックなども行う。

バックエンドの入力には，構文木を用いるコンパイラも存在するが，原始プログラムを記述しているプログラミング言語に依存する部分が少ない**中間表現** (intermediate representation) を用いるのが一般的である。中間表現を用いることによって，フロントエンドとバックエンドの独立性を高めることができ，各部分の再利用性を高めることができる。

バックエンドは，大まかにいって，つぎの二つのフェーズで構成される。

① **コード最適化** (code optimization) フェーズ

② **コード生成** (code generation) フェーズ

コード最適化フェーズは，入力として受け取った中間表現に対して特別なプログラム変換を適用することによって，高速に実行できるコードを生成したり，メモリサイズの小さいコードを生成したりする役割を果たす。コード最適化フェーズは，複数のコード最適化ルーチンで構成されることが多く，中間表現による

プログラムは，各最適化ルーチンによって順に変換される。

最終的に，中間表現をコード生成部に送り，目的プログラムが生成される。目的プログラムは機械コードで書かれる場合と，アセンブリコードで書かれる場合とがある。アセンブリコードの場合には，機械コードへの変換のために，さらに**アセンブラ** (assembler) と**リンカ** (linker) による処理が必要である。

1.2.2 開発ツールと記述言語

コンパイラを構成するフェーズのうち，字句解析と構文解析は，**正規表現** (regular expression) と**文脈自由文法** (context–free grammar) によって定式化できることが知られている。そして，これらを入力として，それぞれのフェーズを自動生成するつぎのツールを利用することができる。

- **Lex**：正規表現から字句解析プログラムを生成する。
- **Yacc**：文脈自由文法から構文解析プログラムを生成する。

本書の前半では，これらの自動生成を可能にする理論を説明し，Lex と Yacc を用いて，フロントエンドを作成する方法を説明する。

Lex と Yacc における意味動作とバックエンドの記述には，関数型プログラミング言語の一つである OCaml を用いる。OCaml は，型推論機構と豊富なデータ構造を表現する構文を持つので，型を指定したり，自分でデータ構造を宣言することに煩わされることなく，本来のアルゴリズムの記述に集中することができる。

また，目的プログラムの記述には，具体性を高めるために，最近の多くの PC で動作させることができる x64 コードというアセンブリコードを用いる。

本書では，演習プロジェクトとして，Simple という単純なプログラミング言語を定義し，x64 コードを生成するコンパイラを作成する。Simple コンパイラの記述は，Lex と Yacc を用いて，OCaml で記述する。そこで，プロジェクトに必要な範囲で，OCaml の入門と，x64 コードの入門を与える。

索　　引

—— 著 者 略 歴 ——

1992年　慶應義塾大学理工学部計測工学科卒業
1994年　慶應義塾大学大学院理工学研究科前期博士課程修了（計算機科学専攻）
1999年　慶應義塾大学大学院理工学研究科後期博士課程単位取得退学（計算機科学専攻）
1999年　東京理科大学助手
2003年　博士（工学）（慶應義塾大学）
2004年　東京理科大学講師
2005～
2006年　カリフォルニア大学アーバイン校在外研究員
2010年　東京理科大学准教授
2013年　東京理科大学教授
　　現在に至る

実践コンパイラ構成法

Practical Compiler Construction Method

2017 年 7 月 25 日　初版第 1 刷発行

検印省略

著　者　滝本宗宏（たきもと むねひろ）
発 行 者　株式会社　コロナ社
　　代 表 者　牛来真也
印 刷 所　三美印刷株式会社
製 本 所　有限会社　愛千製本所

112–0011　東京都文京区千石 4–46–10
発 行 所　株式会社　コロナ社
CORONA PUBLISHING CO., LTD.
Tokyo Japan
振替 00140–8–14844・電話(03)3941–3131(代)
ホームページ　http://www.coronasha.co.jp

ISBN 978–4–339–01933–9　C3355　Printed in Japan　（新井）