

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

電子通信情報系コアテキストシリーズ  
編集委員会

編集委員長

博士（情報理工学） 浅見 徹（東京大学）

編集委員

（五十音順）

博士（理学） 河野 健二（慶應義塾大学）

博士（情報学） 五島 正裕（国立情報学研究所）

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

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ジタルメディアに書き込まれるようになってきている。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

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。

そうはいつても、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<sup>†</sup> 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

---

<sup>†</sup> 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

早稲田大学

## 1 章 はじめに

---

- 1.1 言語処理系 2
- 1.2 コンパイラ 2
  - 1.2.1 コンパイラの構成 3
  - 1.2.2 開発ツールと記述言語 5

## 2 章 記述言語

---

- 2.1 コンパイラの記述と OCaml 7
- 2.2 OCaml の基本 7
  - 2.2.1 実行と基本型 7
  - 2.2.2 変数束縛 11
  - 2.2.3 関数定義 12
- 2.3 複雑な型の利用 13
  - 2.3.1 構造を持つ型 13
  - 2.3.2 パターンマッチング 15
  - 2.3.3 便利な高階関数 15
- 2.4 型の定義 17
  - 2.4.1 レコード 17
  - 2.4.2 バリエント 19
- 2.5 そのほかの重要構文 20
  - 2.5.1 `let-rec-and` と `type-and` 20



	目	次
2.5.2	参 照 型	21
2.5.3	例 外	22
2.6	インタプリタの作成	23
2.6.1	プログラムの木表現	23
2.6.2	環 境	25
2.6.3	意 味 関 数	26
2.6.4	インタプリタの完成	27

## 3 章 字 句 解 析

---

3.1	字句解析の概観	30
3.2	トークンの指定	31
3.2.1	正 規 表 現	32
3.2.2	Lex のトークン指定	33
3.3	有限オートマトンによる実現	35
3.3.1	有限オートマトン	35
3.3.2	DFA とその利用	36
3.3.3	正規表現から NFA への変換	39
3.3.4	NFA から DFA への変換	41
3.3.5	状態の最小化	44
3.4	Lex を用いた字句解析器の実現	45
3.5	Simple コンパイラの子句解析器	48

## 4 章 構 文 解 析

---

4.1	構文の指定	51
4.2	文脈自由文法	52
4.2.1	導 出	52
4.2.2	解析木と曖昧な文法	53
4.2.3	曖昧でない文法への変換	55
4.3	予測型構文解析	57

4.4	FIRST 集合と FOLLOW 集合	59
4.4.1	FIRST 集合を求めるアルゴリズム	60
4.4.2	FOLLOW 集合	62
4.5	予測型構文解析器の実現	64
4.5.1	予測型構文解析表	64
4.5.2	左再帰の除去	65
4.5.3	左くり出し	66
4.5.4	エラー回復	69
4.6	LR 構文解析	70
4.6.1	LR(0) 構文解析器の実現	73
4.6.2	SLR 構文解析	78
4.6.3	LR(1) 構文解析	79
4.6.4	LALR(1) 構文解析	83
4.6.5	文法クラスの関係	85
4.7	Yacc と Simple 言語の構文解析器の実現	85
4.7.1	OCamlyacc の概観	86
4.7.2	曖昧な文法の利用	90
4.7.3	OCamlyacc のエラー回復	93
4.7.4	OCamllex との連携	94
4.8	抽象構文木	96
4.9	Simple コンパイラの構文解析	101
4.9.1	文	103
4.9.2	宣言	104
4.9.3	左辺値	105
4.9.4	式	105
4.9.5	構文解析の実現	106

## 5 章 意味解析

5.1	記号表	110
-----	-----	-----

5.1.1	有効範囲と記号表	110
5.1.2	記号表の実現	112
5.1.3	記号表の登録情報	118
5.2	型 検 査	119
5.2.1	型	119
5.2.2	式の型検査	122
5.2.3	宣言の処理	125

## 6 章 実行時環境

---

6.1	x64 アセンブリ言語	131
6.1.1	x64 アセンブリコードの概観	131
6.1.2	メモリの構成	133
6.1.3	x64 の 命 令	135
6.2	関数呼出しと駆動レコード	140
6.2.1	高階関数	140
6.2.2	スタックフレーム	142
6.2.3	呼出し規約	144
6.2.4	非局所データの参照	149

## 7 章 コード生成

---

7.1	コード生成の準備	155
7.2	式のコード生成	157
7.2.1	定数と変数	157
7.2.2	算術演算	160
7.2.3	関数呼出し	163
7.3	文のコード生成	163
7.3.1	代入文	164
7.3.2	C ライブラリを呼び出す仮想関数	164
7.3.3	return文	168

7.3.4	手続き呼出し	168
7.3.5	関係演算と分岐	170
7.3.6	ブロックのコード生成	174
7.4	宣言の処理	175
7.4.1	型宣言の処理	175
7.4.2	関数のコード生成	176
7.5	プログラムのコード生成	177
7.6	Simple コンパイラの完成	177
7.7	コード最適化	179
7.7.1	冗長な命令の削除	180
7.7.2	制御フローの最適化	182
付録	Simple 言語	186
A.1	言語マニュアル	186
A.1.1	プログラム	186
A.1.2	字句	186
A.1.3	宣言	186
A.1.4	文	187
A.1.5	式	188
A.2	Simple 言語のプログラム例	190
A.2.1	ユークリッドの互除法	190
A.2.2	再帰を用いた単純ソート	191
A.3	Simple コンパイラプログラム	192
	引用・参考文献	203
	索引	204

# 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<sup>1),2)</sup>†。現在、その努力は、自動的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。

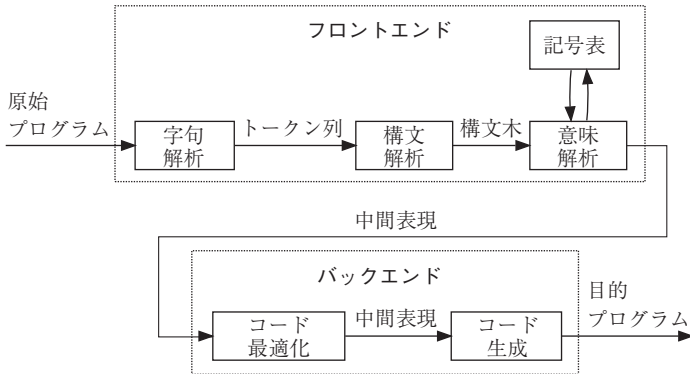


図 1.1 コンパイラの構成

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

【あ】	【か】	【く】
アセンブラ assembler 5, 131	開始記号 start symbol 52	空導出可能 nullable 60
アセンブリ言語 assembly language 131	開始状態 start state 35	具象構文木 concrete syntax tree 97
後入れ先出し LIFO 140	解析木 parse tree 53	駆動レコード activation record 142
アドレッシングモード addressing mode 136	型 type 9	組 tuple 14
アプリケーションバイナリ インタフェース application binary interface, ABI 145	型検査 type checking 109	【け】
アルファベット alphabet 32	型構成子 type constructor 119	決定性 deterministic 36
【い】	型式 type expression 119	決定性有限オートマトン deterministic finite automaton 36
意味解析 semantic analysis 3	型推論 type inference 7, 12	言語 language 32, 36
因子 factor 56	型変数 type variable 13	言語処理系 2
インタプリタ interpreter 2	可変長引数 variable argument 146	原始プログラム source program 3
【う】	環境 environment 25	【こ】
右辺値 rvalue 104	還元 reduce 70, 72	項 term 56
【え】	関数型 functional 7	高階関数 higher order function 141
エピローグ epilogue 133, 144	【き】	構造等価 structural equivalence 121
エラー回復 error recovery 69	記号 symbol 32	構文解析 syntax analysis または parsing 3, 50
【お】	記号表 symbol table 109	構文解析器 parser 50
オフセット offset 136	競合 conflict 64	構文解析器生成系 parser generator 86
	局所変数 local variable 12, 140	構文木 syntax tree 4, 97
	拒否 reject 36	

コード最適化  
code optimization 3, 4, 179

コード生成  
code generation 4, 154

コード生成器  
code generator 154

ゴミ集め  
garbage collection 135

コンパイラ  
compiler 2

**【さ】**

最右導出  
rightmost derivation 53

再帰下降型構文解析器  
recursive descent parser 57

最左導出  
leftmost derivation 53

最長一致 35

最適化  
optimization 3, 179

最適化器  
optimizer 180

先読み  
lookahead 57

左辺値  
lvalue 103

**【し】**

式  
expression 56

識別子  
identifier 30

字句  
lexeme 30

字句解析  
lexical analysis 3, 29

字句解析器  
lexical analyzer  
または lexer 29

字句解析器生成系  
lexical analyzer generator 46

字句有効範囲  
lexical scope 110

四則演算言語 23

実行スタック  
execution stack 134

シフト  
shift 72

シャドウ領域  
shadow space 147

終端記号  
terminal symbol 52

出力引数  
outgoing argument 144

受理状態  
accept state 35

状態  
state 35

**【す】**

スタックフレーム  
stack frame 142

スタックポインタ  
stack pointer 142

**【せ】**

正規表現  
regular expression 5, 31

生成規則  
production rule 52

静的リンク  
static link 150

遷移  
transition 35

**【そ】**

属性  
attribute 31

即値  
immediate 136

**【た】**

多重定義  
overloading 124

**【ち】**

チェイン法  
chaining 112

中間表現  
intermediate  
representation 4

抽象構文木  
abstract syntax tree 96

**【て】**

定数リテラル 30

ディスプレイ  
display 151

手続き間最適化  
interprocedural  
optimization 180

手続き内大域最適化  
intraprocedural global  
optimization 180

手続き呼出し  
procedure call 103

**【と】**

動作  
action 34, 47

導出  
derivation 52

トークン列  
token 4, 30, 46

トランスレータ  
translator 2

**【な】**

名前等価  
name equivalence 120

**【に】**

二分探索木  
binary search tree 115

入力引数  
incoming argument 143

## 【の】

のぞき穴最適化  
peephole optimization 180

## 【は】

パターンマッチング  
pattern matching 7

バックエンド  
back end 3

ハッシュ関数  
hash function 112

ハッシュ表  
hash table 112

バリエーション  
variant 19

## 【ひ】

非決定性  
non-deterministic 36

非決定性有限オートマトン  
non-deterministic finite  
automaton 36

非終端記号  
non-terminal symbol 52

左くり出し  
left factor 66

左再帰  
left recursion 59

## 【ふ】

フェーズ  
phase 3

副作用  
side effect 23

物理アドレス  
physical address 133

ぶらさがり else  
dangling else 92

プログラミング言語処理系 2

プログラム解析  
program analysis 180

ブロック構造  
block structure 149

プロローグ  
prologue 133, 144

フロントエンド  
front end 3

文  
statement 56

文法  
grammar 52

文脈自由文法  
context-free grammar 5, 51

## 【へ】

閉包  
closure 74

並列化  
parallelization 3

変数束縛  
variable binding 11

## 【も】

目的プログラム  
target program 3

戻りアドレス  
return address 143

## 【ゆ】

有限オートマトン  
finite automaton 35

有効範囲  
scope 110

優先規則 35

## 【よ】

予測型構文解析  
predictive parsing 50

予測型構文解析器  
predictive parser 57

予測型構文解析表  
predictive parsing table 64

呼び出され側  
callee 147

呼び出され側保存レジスタ  
callee-save register 148

呼出し側  
caller 147

呼出し側保存レジスタ  
caller-save register 148

呼出し規約  
calling convention 144

予約語  
keyword 30

## 【り】

リスト  
list 14

リンカ  
linker 5

## 【れ】

例外処理  
exception handling 22

レコード  
record 14, 17

## 【ろ】

論理アドレス空間  
logical address space 134

	<b>[B]</b>	Lex	5	OCamllex	33
BNF		LR		ocamlopt	8
Backus-Naur form	87	left-to-right	70	OCamlyacc	86
	<b>[D]</b>	LR 構文解析			
		LR parsing	50, 70	<b>[S]</b>	
DFA		LR(0)	73	Simple 言語	186
deterministic finite automaton	36	LR(0) 項		SLR	
		LR(0) item	74	Simple LR	79
	<b>[E]</b>	LR(0) 構文解析表		System V AMD64 ABI	145
error トークン		LR(0) parsing table	76		
error token	93	LR(1)	80	<b>[Y]</b>	
	<b>[F]</b>	LR(1) 項		Yacc	
FIRST 集合	59	LR(1) item	80	Yet another compiler compiler	5, 86
FOLLOW 集合	62				
	<b>[G]</b>	<b>[M]</b>			
gcc		Microsoft x64	145	<b>【ギリシャ文字】</b>	
GNU Compiler Collection	131	<b>[N]</b>		$\epsilon$ 閉包	
	<b>[L]</b>	NFA		$\epsilon$ -closure	42
LALR(1)		non-deterministic automaton	36		
look-ahead LR(1)	83	<b>[O]</b>			
		ocamlc	8		

—— 著者略歴 ——

1992年 慶應義塾大学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

© Munehiro Takimoto 2017

2017年7月25日 初版第1刷発行

検印省略

著者 たもと 滝 もと 本 むねひろ 宗 ひろ 宏  
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)



**JCOPY** < 出版者著作権管理機構 委託出版物 >

本書の無断複製は著作権法上での例外を除き禁じられています。複製される場合は、そのつど事前に、出版者著作権管理機構（電話 03-3513-6969, FAX 03-3513-6979, e-mail: info@jcopy.or.jp）の許諾を得てください。

本書のコピー、スキャン、デジタル化等の無断複製・転載は著作権法上での例外を除き禁じられています。購入者以外の第三者による本書の電子データ化及び電子書籍化は、いかなる場合も認めていません。落丁・乱丁はお取替えいたします。