構文解析のしくみ

水島宏太

2025 年 6 月

目次

第 1 章

第 1 章構文解析の世界へようこそ

皆さん、はじめまして！ この本は「構文解析」というテーマについて扱った、一風変わった本です。これまで「構文解析」については、コンパイラや言語処理系を扱った書籍の一部で触れられる程度でした。「Parsing Techniques」という英語の書籍があるものの、英語圏ですら広く知られているとは言い難いですし、内容も網羅的ではあるものの平易とは言えません。

私がはじめて「構文解析」の入口に立ったのは高校三年生のときでした。当時、私はプログラミング言語に興味を持ち始め、いつかは自分のプログラミング言語を作ってみたいと思っていました。しかし、高校の図書館にあった「プログラミング言語を作る」本を借りて読んでみたものの、とても難しく当時の私には理解できませんでしたし、コンパイラ開発者のバイブルともよばれる「ドラゴンブック」[*1]も同様でした。

何がきっかけかは覚えていないのですが、いきなり大きな言語を作ろうとしても難しいということでまずは括弧を含む算術式を計算できる「電卓」アプリを作ることにしました。非常に単純なもので

```
(1 + 2) * (3 + 4)
```

のようにテキストフィールドに入力して「計算」ボタンを押すと 21 と表示される、ただそれだけの代物です。今振り返れば、この電卓アプリはごく初歩的な「構文解析」と「インタプリタ」を実装したものとも言えます。

ただし、当時の私は言語処理系を作る際に必須の知識がほとんどなく、抽象構文木も典型的な構文解析のアルゴリズムも知らず、足りない知恵を振り絞って括弧の対応や演算子の優先順位を自力で計算したのでした。それは本当に拙いものでしたが、それでもなんとか動くものを作ることができたのです。

それから二十年余り。筆者は現在、研究職ではないものの構文解析の手法の一つである、Parsing Expression Grammar（PEG）を専門として、時折関連する論文の査読を引き受けたりしています。

現在の私は研究に未練を残しつつちょっとアカデミアに関わりがある微妙な立場ですが、そんな中でも折に触れて思うことがありました。構文解析はなぜ、一般の技術書であまり取り上げられないのだろうか、ということです。

[*1] 翻訳: ISO/IEC 14977:1996 Information technology — Syntactic metalanguage — Extended BNF https://hazm.at/mox/lang/meta-language/ebnf/iso-iec-14977-extended-bnf/index.html

言語処理系を扱う書籍の一部として構文解析が取り上げられることは珍しくありません。しかし、その扱いはあくまで「おまけ」であって、言語を作る上での通過点としてしか位置づけられていません。これについては、言語の本質は構文解析の「後」にあるのであり、構文は本質ではないというのが大きいでしょう。筆者もこの点に異論はありません。ただ、構文（正確には具象構文）はほんとうに「おまけ」なのでしょうか。ときどき疑問に思います。

プログラミング言語に携わる研究者の間では、プログラミング言語の「意味」は抽象構文木に対して定められるものであり、構文は「ガワ」に過ぎないという考え方の人が多いように思いますし、その考えも間違ってはいないと思います。

しかし、現実に我々が読み書きするのはプログラムの抽象構文木ではなく、プログラムの文字列です。構文は UI であると言えます。多くのアプリケーションにおいて UI が果たす役割は非常に大きく、UI 自体が専門分野として存在しているくらいです。その UI たる構文と UI を適切な内部構造に変換する構文解析は軽視されていいのだろうか。筆者が構文解析の本を書こうと思い立ったのはそんな「こだわり」からでした。

本書の読者の皆さんには、構文解析の世界を少しでも楽しんでいただければと思います。構文解析は非常に奥深いテーマであり、その奥深さを一冊の本で完全に網羅することはできません。しかし、本書を通じて構文解析の基礎を学び、その面白さを感じていただければと思います。

1.1　構文解析の歴史的背景

構文解析の歴史は、コンピュータサイエンスの黎明期にまで遡ります。1950 年代から 1960 年代にかけて、初期のコンパイラ開発者たちは、人間が書いたプログラムをコンピュータが理解できる形に変換する方法を模索していました。当時、プログラミング言語の文法を厳密に定義し、それを機械的に処理する方法は確立されていませんでした。

転機となったのは、言語学者ノーム・チョムスキーが 1956 年に発表した形式言語理論でした。チョムスキーは言語を 4 つの階層（チョムスキー階層）に分類し、その中で「文脈自由文法」（第 4 章で解説）がプログラミング言語の記述に適していることが明らかになりました。この理論的基盤により、構文解析は科学的なアプローチが可能な分野となったのです。

構文解析アルゴリズムの発展において重要な貢献をした人物たちがいます。1965 年、Donald Knuth（ドナルド・クヌース）は LR 法（第 5 章で解説）を発表しました。これは「Left-to-right, Rightmost derivation」の略で、効率的な構文解析を可能にする画期的なアルゴリズムでした。一方、LL 法（Left-to-right, Leftmost derivation）（第 5 章で解説）は、1960 年代後半に Philip M. Lewis II（フィリップ・ルイス）と Richard E. Stearns（リチャード・スターンズ）によって体系化されました。LL 法は再帰下降構文解析の理論的基礎を提供し、より単純で理解しやすい手法として普及しました。これらのアルゴリズムは、現在でも多くの構文解析器の基礎となっています。

初期のプログラミング言語の開発においても、構文解析は重要な役割を果たしました。John Backus（ジョン・バッカス）は 1950 年代に FORTRAN の開発を主導し、Peter Naur（ピーター・ナウア）とともに BNF（Backus-Naur Form）（第 2 章で解説）という文法記述法を確立しました。BNF は、プログラミング言語の文法を形式的に記述するための標準的な記法となり、現在でも広く使われています。また、Grace Hopper（グレース・ホッパー）は初期のコンパイラ開発のパイオニアとして、人間が理解しやすい言語から機械語への変換技術の基礎を築きました。

1970 年代に入ると、Stephen C. Johnson（スティーブン・ジョンソン）によって Yacc（Yet Another Compiler Compiler）が開発され、Mike Lesk（マイク・レスク）と Eric Schmidt（エリック・シュミット）によって Lex（Lexical Analyzer Generator）が開発されました。これらのツールは、文法定義から自動的に構文解析器を生成することを可能にし、コンパイラ開発の民主化に大きく貢献しました。現在でも、これらのツールの思想は多くのパーサージェネレータに受け継がれています。

現代では、ANTLR（ANother Tool for Language Recognition）のような、より強力で使いやすいツールが登場し、構文解析はますます身近なものになっています。また、Parsing Expression Grammar（PEG）（第 5 章で解説）のような新しい形式文法も提案され、構文解析の世界は今なお進化を続けています。

1.2　構文解析とは何か

このテーマについては別の立場からの意見もあるかもしれませんが、筆者は構文解析を以下のように定義しています。

構文解析とは、入力された文字列が特定の文法に従っているかどうかを判断し、その構造を抽出するプロセスです。プログラミング言語においては、ソースコードを解析して文法に従っているかを真偽値で返したり、抽象構文木（AST: Abstract Syntax Tree）を生成することが一般的な目的です。抽象構文木については第 2 章の後半で詳しく説明します。

文法に従っているかを真偽値で返す場合、構文解析のためのメソッドは Java では以下のように表現できます。

```
boolean parse(String input);
```

文法に従っているかを真偽値で返すだけではなく、構文解析の結果として抽象構文木を生成する場合、メソッドは以下のように表現できます。ここで、記述を簡便にするため、Java 17 から正式導入された sealed interface と Java 16 から正式導入された record を使っています。

```
interface Tree {}
sealed interface ParseResult
  permits ParseResult.Success, ParseResult.Failure {
  record Success(Tree ast) implements ParseResult {}
  record Failure(String errorMessage) implements ParseResult {}
}
ParseResult parse(String input);
```

インタフェース `Tree` は抽象構文木の基底クラスであり、`ParseResult` は構文解析の結果を表すクラスです。`ParseResult.Success` は構文解析が成功した場合の結果を表し、抽象構文木を含みます。一方、`ParseResult.Failure` は構文解析が失敗した場合の結果を表し、エラーメッセージを含みます。

構文解析によって得られる抽象構文木を処理することで、さまざまな操作が可能になります。例えば、抽象構文木を使ってコードの最適化や変換、静的解析、コード生成などを行うことができます。

皆さんが何かしらの言語（プログラミング言語に限りません。HTML や Markdown、JSON なども含みます）で書かれたテキストを扱っているとき、構文解析器は必ずどこかで動いています。例えば、JavaScript のコードをブラウザで実行する際、ブラウザはまず JavaScript のコードを構文解析し、抽象構文木を生成してから実行します。また、JSON を扱う API では、JSON 文字列を構文解析して抽象構文木に変換し、その後の処理に利用します。

1.3　構文解析の身近な応用例

構文解析は、皆さんが日常的に使っているツールの中で活躍しています。いくつか具体例を見てみましょう。

IDE の機能: Visual Studio Code や IntelliJ IDEA などの統合開発環境では、コードを書いている最中にリアルタイムで構文解析が行われています。コード補完（入力途中で候補を表示）、コードの構造表示やナビゲーション、エラーの即座の検出などは構文解析によって実現されています。

リンターとフォーマッター: コードの品質チェックや自動整形を行うツールが昨今一般的に使われています。たとえば、ESLint や Prettier などがあります。これらも内部で構文解析を行っています。コードを抽象構文木に変換し、その構造を解析することで、潜在的な問題を検出したり、一貫したスタイルに整形できるのです。

マークダウンパーサー: ドキュメントを執筆する際に使われる Markdown も、構文解析の対象です。****太字****や***斜体***、見出しやリストなどの記法を解釈し、HTML に変換するためには、Markdown の文法に従った構文解析が必要です。

設定ファイルの解析: `.env` ファイルの環境変数、`.gitignore` のパターン、`package.json` の依存関係など、様々な設定ファイルも構文解析の対象です。

データベースクエリ: SQL クエリを実行する際、データベースエンジンはまずクエリを構文解析し、実行計画を立てます。`SELECT * FROM users WHERE age > 18` のようなクエリも、構文解析を経て初めて実行可能になります。

正規表現エンジン: 正規表現そのものも、実は小さな言語です。`/^[a-zA-Z0-9]+@[a-zA-Z0-9]+\.[a-zA-Z]+$/`のようなパターンを解釈するには、正規表現の文法に従った構文解析が必要です。

これらの例からわかるように、構文解析は特別な分野の技術ではなく、私たちの開発を支える基盤技術なのです。

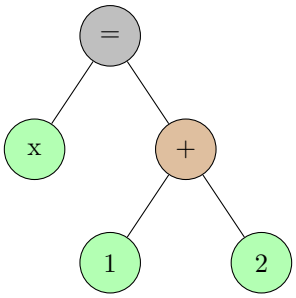
1.4　構文解析の奥深さ

皆さんがもっとも身近に使っている言語の 1 つである JavaScript を例にとって、構文解析の奥深さについて説明してみます。

以下の JavaScript プログラムを構文解析することにします。

```
x = 1 +
  2
```

このコードは、JavaScript エンジンによって `x = 1 + 2;` と解釈され、直観的には以下のような抽象構文木が結果として返ってくるのが正しいように思われます（抽象構文木の説明は、いったんおいておきます）。

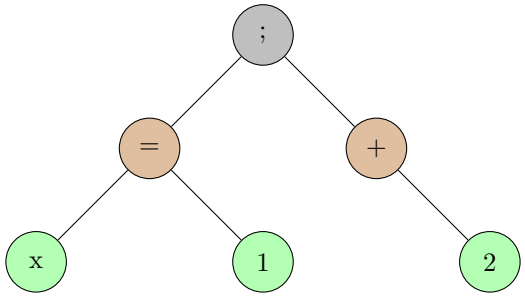


では、次の JavaScript プログラムはどうでしょうか？

```
x = 1
+ 2
```

おそらく多くの方は先ほどと同じ構文木になることを期待するのではないでしょうか。

しかし、実際には JavaScript の自動セミコロン挿入（ASI: Automatic Semicolon Insertion）というルールにより、`x = 1; +2;` と解釈され、結果として以下のような 2 つの文として扱われます。



`+` の後で改行するか、あるいは `+` の前に改行するかという一見**ささいな違い**によって、構文解析の結果が変わってしまうのです。JavaScript の ASI は、特定のルールに基づいて行末にセミコロンが自動的に挿入される機能ですが、そのルールは時として直感に反する結果を生むことがあります。例えば、`return` 文の直後で改行すると、

```
return
a + b;
```

これは ASI により `return; a + b;` と解釈され、意図しない結果（`undefined` が返される）になることがあります。ASI の主なルールには、「行終端文字の後に続くトークンが、現在の文法的文脈で許されない場合、行終端文字の位置にセミコロンを挿入する」といったものがありますが、例外も存在します。このような複雑さが、構文解析器の実装を難しくする一因となります。ASI による予期せぬ挙動を避けるために、行頭にセミコロンを記述するスタイルや、式の途中では改行しないといったコーディング規約を採用する開発者もいます。

2000 年以降に登場して普及した言語は、このような特徴を持っていることが多いです。たとえば、Go、Swift、Kotlin、Scala の文法はこのような特徴を持っています。より古い言語である Ruby や Python も同じ特徴を持っています。このように、ちょっとした違いで構文解析結果が変わってしまう例は珍しくありません。

このような文法の進化の背景には、プログラマにとって「書きやすく読みやすい」文法を提供するという意図があります。このような「改行に敏感な」文法があると、構文解析器装が複雑になりがちです。プログラミング言語の設計者にとっては考慮すべき点が増えるので、このような文法を嫌う言語設計者もいますが、利用者にとってはセミコロンの入力を省略できるなどの利便性があるため、広く採用されています。

しかし「構文解析が複雑になる」というのは一体どういうことでしょうか。ほとんどの方はピンとこないのではないでしょうか。この本では、このような問いに対して一定の答えを提示することを目指します。「構文解析の複雑さ」と一言で言っても、その要因は様々です。例えば、文法が曖昧であるか（一つの文に対して複数の解釈が可能か）、解析にバックトラック（試行錯誤）が必要か、どれだけの先読みトークン（入力の一部を先に見る数）が必要か、といった要素が複雑さに関わってきます。

本書では、これらの概念に触れながら、構文解析の奥深さを探求していきます。これらの複雑さを理解する上で重要な役割を果たすのが、「文脈自由文法」という考え方です。文脈自由文法とは、簡単に言えば、プログラムの構造を定義するための一連の書き換え規則のことです。例えば、「数値は式である」や「式は、式と演算子と式からなる」といったルールを形式的に記述するものです。詳細については第 4 章で詳しく解説しますが、この文脈自由文法という道具立てがあることで、構文解析の様々な側面を体系的に議論できるようになります。

読者の皆さんは既存の言語に歯がゆさを感じたことはないでしょうか。たとえば、昨今は JSON や YAML を使って領域特化言語（DSL:Domain Specific Language）を提供することが一般的になっています。Amazon Web Services（AWS）の CloudFormation や Google Cloud Platform（GCP）の Deployment Manager などがその例です。これらの DSL は、JSON や YAML という既存のデータ形式を利用して、特定のドメインに特化した言語を提供しています。

しかし、これらの DSL は JSON や YAML の枠に収めるために無理をしており、どうしても「不自然」な文法になってしまいます。JSON や YAML は世界中に普及していますから、JSON や YAML を使った DSL を提供する合理性はあるもの、場合によっては JSON や YAML に縛られない DSL の文法を考案することが求められることもあります。そのためには構文解析の知識が必要です。

構文解析は単に実用的であるにとどまらず非常に「楽しい」テーマです。読者の皆さんには本書を通じて、是非「構文解析の面白さ」を感じていただければと思います。

本書は次のような構成になっています。

第 2 章では簡単な算術式を例題にして、構文解析とはどういう処理かを体感してもらいます。定義を天下り式に提示するような本もありますが、構文解析については「まずは書いてみる」のが手っ取り早いというのが筆者の持論です。

第 3 章では JSON の構文解析器を書いてもらいます。JSON は世界中で使われている非常に実用的なデータ形式（言語）でありながら、非常にシンプルです。そのシンプルな JSON を通して、実用的な言語の構文解析の基礎を学んでいただければと思います。

第 4 章では文脈自由文法（や言語）について解説します。文脈自由文法やそれに関する理論は構文解析の基礎になっています。括弧の対応を表現する言語を `Dyck` 言語と言いますが、`Dyck` 言語は文脈自由言語を特徴づけるものです。この章を理解することで文脈自由文法の直観的な理解が得られます。現代の構文解析は文脈自由文法を基盤としていますから、文脈自由言語の概念を理解することは非常に重要です。

第 5 章では現在の構文解析で広く採用されているアルゴリズムのうち、主だったものを解説します。特に LL(1) や LR(1) といったアルゴリズムについてできるだけ平易にかつ詳しく説明します。また、PEG や Packrat Parsing という比較的新しい構文解析アルゴリズムについても詳しく説明します。

第 6 章では構文解析器生成系について解説します。Yacc のような古典的なものに留まらず、ALL(*) アルゴリズムを基盤にした ANTLR や PEG を基盤にしたパーサーコンビネータなど、最新の構文解析器生成系について説明します。さらに、第 6 章では簡単なパーサーコンビネータを自作します。パーサーコンビネータは元々は関数型プログラミング言語から出てきたテクニックですが昨今では色々な言語でパーサーコンビネータライブラリがあります。パーサーコンビネータを自作する体験を通じて「文の定義から構文解析器を生成する」とはどういうことか理解してもらえるのではないかと思います。

第 7 章では従来の言語処理系についての本が取り扱わなかった「現実の構文解析」の話をします。従来の書籍に書かれている構文解析の世界はとても「綺麗」なものです。しかし、Ruby でも Python でもあるいは JavaScript でも良いですが、現実の構文解析は必ずしもそこまで綺麗にはいかないものです。この章を通して現実の言語における構文解析はとても泥臭いものであること、その泥臭さを通して「書きやすく読みやすい」文法が実現されていることを実感してもらえるのではないかと思います。

第 8 章は締めくくりとしてこれまでの章を振り返りつつ、今後、みなさんが構文解析を学ぶにあたって参考になりそうな文献や資料について紹介します。この本は構文解析のみを取り扱った珍しい本ですが、それでも構文解析の世界は広く、本書で取り扱わなかったテーマも多々あります。この章を読んで、構文解析の世界により深く興味を持っていただければ幸いです。

1.5　本書で扱うプログラミング言語とツール

本書では、構文解析を学ぶために、以下のプログラミング言語とツールを使用します。

Java: 本書の主要な実装言語として Java を選びました。Java は静的型付けであり、Java 21 のような最近のバージョンであれば、抽象構文木のような複雑なデータ構造を表現するのにも不足はありません。そのため、本書で利用する Java のバージョンは 21 を基本とします。

パーサージェネレータ: 第 6 章では以下のツールを扱います：- ANTLR4：現代的で強力なパーサージェネレータ - JavaCC：Java 専用の伝統的なパーサージェネレータ - Yacc/Lex：C 言語用の古典的ツール（歴史的理解のため）

開発環境: 特定の IDE は必須ではありませんが、IntelliJ IDEA Community Edition や Eclipse を推奨します。これらの IDE は、Java の開発に適しており、構文解析器の実装やテストを効率的に行うことができます。Visual Studio Code のようなエディタに慣れている場合は、それでも構いませんが、筆者としては IDE を使うことをお勧めします。特に、IntelliJ IDEA は Java の開発に非常に適しており、コード補完やリファクタリング機能が充実しています。

すべてのサンプルコードは GitHub リポジトリから入手可能です。

1.6　学習の道筋

本書は様々な背景を持つ読者を想定しています。ここでは、目的に応じた読み進め方を提案します。

初心者の方へ

- まず第 2 章で構文解析の基本的な考え方を理解してください
- 次に第 3 章で JSON パーサーを実装し、実践的な経験を積みます
- その後、第 6 章でツールを使った構文解析器の生成を学びます
- 理論的な内容（第 4 章、第 5 章）は後回しにしても構いません

理論もしっかり学びたい方へ

- 第 2 章→第 3 章→第 4 章→第 5 章の順番で読み進めてください
- 第 4 章の文脈自由文法は構文解析の理論的基礎です
- 第 5 章のアルゴリズムは少し難しいですが、じっくり取り組んでください
- 第 6 章で理論の実装を、第 7 章で実践を学びます

すぐに使える知識が欲しい方へ

- 第 2 章→第 3 章→第 6 章→第 7 章の順で読むことをお勧めします
- 第 7 章では現実に遭遇する泥臭い構文解析の問題を扱います

各章の難易度の目安

- 第 2 章：★（入門）
- 第 3 章：★★（実践基礎）
- 第 4 章：★★★（理論基礎）
- 第 5 章：★★★★（理論応用）
- 第 6 章：★★（ツール活用）
- 第 7 章：★★（実践応用）
- 第 8 章：★（まとめ）

最初から最後まで読み通してもらえれば、最終的には構文解析の基礎から応用まで幅広く理解できるようになります。

さて、第 1 章を読み終えた皆さんは、構文解析というテーマに対してどのような印象を持たれたでしょうか。もしかしたら、「普段何気なく書いているプログラムの裏側では、こんな複雑なことが行われているのか」と驚かれたかもしれません。あるいは、「自分の手で言語のルールを定義し、それを解釈するプログラムを作るのは面白そうだ」と感じた方もいるかもしれません。あなたがこれまでに触れたことのある言語で、構文が原因で混乱した経験はありますか？ もしあれば、それはなぜだったのか、この本を読み進める中でヒントが見つかるかもしれません。

次の第 2 章では、いよいよ具体的な構文解析の世界に足を踏み入れ、簡単な算術式を題材に、手を動かしながら構文解析の第一歩を体験していただきます。お楽しみに！

2025 年 6 月、自室にて記す。

水島宏太

1.7　コラム：現代の構文解析の課題

構文解析技術は長い歴史を持ちますが、現代においては新たな課題に直面しています。

エラーリカバリー: 従来の構文解析器は、文法エラーに遭遇すると即座に処理を停止していました。これは、最初の文法エラーが後続の文法エラーを引き起こす上に、後続の文法エラーが実態を表していないことが多かったための措置でした。しかし、IDE では入力途中のコードを常に解析する必要があります。そのため、エラーがあっても可能な限り解析を続行し、意味のあるエラーメッセージを提供する「エラーリカバリー」技術が重要になっています。

インクリメンタルパージング: ファイル全体を毎回解析し直すのは非効率です。変更された部分だけを再解析する「インクリメンタルパージング」により、大規模なファイルでもリアルタイムな解析が可能になります。

Language Server Protocol（LSP）: Microsoft が提唱した LSP は、言語サーバーとエディタ間の通信プロトコルです。実質的には IDE のための基盤プロトコルと言っても良いでしょう。構文解析結果を効率的に共有することで、一つの言語サーバーが複数のエディタに対応できるようになりました。

AI との融合: GitHub Copilot や ChatGPT のような AI ツールは、コードの構造を理解する必要があります。構文解析によって得られた抽象構文木は、AI がコードを**理解**するための重要な手がかりとなっています。

これらの課題は、構文解析が単なる「文字列から木構造への変換」以上の役割を担っていることを示しています。

第 2 章

第 2 章構文解析の基本

この章からいよいよ構文解析についての説明を始めたいと思います。とはいっても本書を手に取った皆様は構文解析についてまだ馴染みがないかと思います。そこで、まずは簡単な算術式の構文解析を例にして、構文解析の基本について学ぶことにしましょう。

算術式という言い方は堅苦しいですが、要は数式のことです。たとえば、`1 + 2` や `(1 + 2) * 3`、`(3 * 3) * (2 * 3)` のようなものです。プログラマーの皆さんなら普段、日常的にプログラムの一部として算術式を書いているでしょう。算術式は非常にシンプルな「言語」なので、構文解析の題材にするのに最適なのです。

2.1　算術式の文法

ただ「算術式」といっただけだと人によってかなりイメージするものが異なります。本書では以下の条件を満たすものを算術式とします。

- 四則演算ができる
  - 足し算は `x+y`
  - 引き算は `x-y`
  - 掛け算は `x*y`
  - 割り算は `x/y`
- 優先順位は掛け算・割り算が高く、足し算・引き算が低い
  - `1+2*3` は `1+(2*3)` と解釈される
  - `7-6/2` は `7-(6/2)` と解釈される
- 掛け算・割り算の優先順位は同じで、足し算・引き算の優先順位も同じ
- 優先順位が同じ演算子は左から右に結びつく
  - `1+2-3` は `(1+2)-3` と解釈される
- 値は整数のみ
  - `1+2.0` のような式はエラーになる
- 式の優先順位を変えるために括弧を使うことができる
  - `1+2*3` は `1+(2*3)` の意味になるが、`(1+2)*3` と書くことで意味が変わる

- スペースは使えない
  - 1+2 は OK だが、1 + 2 はエラーになる

最後の「スペースは使えない」という制限はあくまで単純化のためです。特に、字句解析（文字列を意味のある単位、トークンに分割する処理）を単純化し、構文解析の核となる考え方に集中するために設けています。実際の算術式ではスペースも使えて無視されるのが一般的ですが、その処理は後の章で触れることにします。

この定義に従う算術式には以下のようなものが含まれます。

```
100  # 100
1+2*3  # 7
(1+2)*3  # 9
3*(1+2)  # 9
12/3  # 4
1+3*4/2  # 7
```

皆さんは何かしらのプログラミング言語を使ってプログラムを書いているはずですから、上のような算術式は馴染みが深いはずです。

しかし、上のような日本語を使った定義だけでは算術式の文法を表現するのには不十分です。たとえば「式の優先順位を変えるために括弧を使うことができる」といっても、なんとなくはわかるものの、定義としては曖昧です。もちろん、日本語で詳しく記述して曖昧さを少なくしていくこともできますが、いたずらに長くなるだけです。

曖昧さがない形で文法を表現するために、特別な記法を使うことが一般的です。これを**形式文法**と呼びます。「形式文法」という名前は堅苦しく聞こえるかもしれませんが、要は「プログラミング言語の文法を書くための、決まった書き方」のことです。

たとえば、皆さんが Java のメソッドを書く時にも決まった形式があるように（例：`public void methodName() { ... }`）、プログラミング言語の文法そのものを記述する時にも決まった形式があるのです。

次節では、そのような形式文法の中でも最も広く使われている BNF（Backus-Naur Form、バッカス・ナウア記法）を使って、算術式の文法を表現してみましょう。BNF は、プログラミング言語の仕様書やドキュメントでよく見かける記法です。

2.2 算術式の BNF

プログラミング言語の文法自体を表現する文法（メタ文法といいます）の一つに、BNF があります。BNF は、プログラミング言語の文法をはじめ、インターネット上でのメッセージ交換フォーマットなど、様々な文法を表現するのに使われています。本書の読者の方には BNF に馴染みのない方も多いと思うので、算術式の BNF の前に BNF について説明します。本書では、記述を簡潔にするため、ISO/IEC 14977 で仕様が策定された EBNF[*1]で用いられる繰り

[*1] 翻訳: ISO/IEC 14977:1996 Information technology — Syntactic metalanguage — Extended BNF https://hazm.at/mox/lang/meta-language/ebnf/iso-iec-14977-extended-bnf/index.html

返し（{}）、オプション（[]）、グループ化（()）といった拡張記法を一部取り入れ、これを広義の BNF として扱います。BNF には歴史的に多くの方言が存在するため、本書で用いる記法について事前に説明しておきます。

2.2.1 BNF の概要

BNF は Fortran の開発者でもある、John Backus（ジョン・バッカス）らが開発した記法です。BNF は「プログラミング言語」そのものの文法を記述するために開発されました。基本情報技術者試験でも出題されるので、そこで知った方もおられるかもしれません。

本書で用いる広義の BNF（EBNF の要素を取り入れたもの）の主要な記法は以下の通りです。

- `=`：左辺の非終端記号と右辺の定義を区切ります。
- `|`：選択を表します。例えば `A | B` は「A または B」を意味します。
- `{}`：0 回以上の繰り返しを表します。例えば `A {B}` は「A の後に B が 0 回以上続く」ことを意味します。
- `[]`：0 回または 1 回の出現（オプション）を表します。例えば `[A] B` は「A があるかもしれないし、ないかもしれないが、その後に B が続く」ことを意味します。
- `()`：グループ化を表します。例えば `(A | B) C` は「A または B の後に C が続く」ことを意味します。
- `'...'` または`"…"`：終端記号（リテラル文字列）を表します。
- 非終端記号（例: `expr`, `term`）：他の規則で定義される記号を表します。

では早速、BNF の具体例を見て行きましょう。以下の例は 2.1 で出てきた算術式を元に、

- 扱える数値は一桁だけ

のように単純化した BNF です。

```
expr = term { ('+' | '-') term };
term = factor { ('*' | '/') factor };
factor = NUMBER | '(' expr ')';
NUMBER = '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9';
```

たくさんの記号が出てきて戸惑われたかもしれませんが、一つずつ丁寧に見ていきましょう。

2.2.2 expr

BNF では、以下のような**生成規則**の集まりによって、文法を表現します。

```
expr = term { ('+' | '-') term };
```

=の左側である expr が**左辺**（この規則で定義される非終端記号）で、右側が**右辺**（その定義）になります。

ここでは

```
term { ('+' | '-' ) term }
```

が本体になります。

本体の中に出てくる、他の規則を参照する部分（ここでは term) を**非終端記号**と呼びます。これは「まだ展開の余地がある記号」という意味で、同じ BNF 内で定義されている他の非終端記号と一致する必要があります。Java でいえば、メソッド呼び出しのようなものです。

一方、'+' や'-' のように' で囲まれた記号や文字を**終端記号**と呼びます。これは「これ以上展開できない、最終的な文字」という意味で、実際にプログラムに書かれる文字そのものを表します。Java でいえば、+ 演算子や; セミコロンのようなものです。

BNF において{}で囲まれたものは、その中の要素が 0 回以上繰り返して出現することを示しています。したがって、

```
term { ('+' | '-') term }
```

という記述は、まず term が出現し、その後に「+ または-とそれに続く term」が 0 回以上繰り返して出現することを示しています。0 回以上という言い方が重要です。というのは、term が 1 回だけ出現してその後は何も出現しない場合もあるからです。たとえば、1 という式はこの規則にマッチしますが、{}で囲まれた部分は出現していません。

この生成規則を日本語で表現すると「expr という非終端記号は、右辺の定義 term { ('+' | '-') term } によって定義される」と読むことができます。

なお、本来の（ISO/IEC）BNF では

```
expr = term, { ('+' | '-'), term };
```

のように要素間をカンマで区切りますが、本書では可読性を考慮してスペースで区切るようにしています。

2.2.3 term

term は算術式の中で、掛け算や割り算を含んだ式を表す規則です。factor という規則を参照しています。

```
term = factor { ('*' | '/') factor };
```

この規則によって term は、factor が*または/を挟んで 0 回以上繰り返して出現することを示しています。

2.2.4 factor

factor は算術式の中で、数値や括弧で囲まれた式を表す規則です。

```
factor = NUMBER | '(' expr ')';
```

この規則によって factor は、

- NUMBER
- （と）に挟まれた expr

のどちらかであることを示しています。

2.2.5 NUMBER

NUMBER は数値を表す規則です。ここでは単純化のために 1 桁の整数のみを扱っています。

```
NUMBER = '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9';
```

NUMBER は 0 から 9 のどれか 1 文字であることを示しています。

2.3 BNF を使った解析プロセス

算術式の文法を規則 expr を含む BNF によって表現することができました。この記述には**曖昧さがなく**、スペースを許さないことや演算子の優先順位などの概念も含まれています。

この規則 expr を使うと、式 1+2 はどのように解析できるのでしょうか？ ここでは、その概要について簡単に説明します。

この BNF を使って解析を行う方法は一意ではなく、様々なやり方（手続き）があります。その個々の手続きがまさに構文解析アルゴリズムとなります。個々の構文解析アルゴリズムについては第 5 章で詳しく説明しますが、ここでは、BNF を使って**素直に**算術式を解析する方法の概要を説明します。ちなみに、構文解析に限った話ではありませんが、そのような**素直な**方法のことをナイーブ（naive）な方法と呼ぶことがあります。

まず最初に BNF の規則を関数とみなします。たとえば、規則 expr は関数 expr に対応し、規則 term は関数 term に対応します。これらの規則に対応する関数は、解析したい文字列を引数に取り、解析の結果を返します。

この関数の宣言を Java ライクな擬似コードで表すと、以下のようになります。

```
ParseResult expr(String input);
```

次に関数の呼び出しを構文解析とみなします。引数として解析したい文字列を渡すと、構文解析結果が返ってくるものとします。

たとえば、expr("1+2") という関数呼び出しは、文字列"1+2"を解析することを示します。まだ expr の本体がありませんが、結果としては、解析が成功した旨の情報がかえってきて欲しいでしょう。一方、expr("1+") という関数呼び出しは、解析が失敗した旨の情報がかえってきて欲しいはずです。

関数の呼び出し結果は、

- 解析に成功した場合：成功したことを表す情報と、まだ解析していない残りの文字列。これを (SUCCESS, remaining_string) と表す
- 失敗した場合：失敗を表す定数 FAIL を返す

のどちらかであるとします。

以上の前提を元に、実際に expr("1+2") がどのように解析されるか、主要なポイントを追ってみましょう。詳細な

内部動作は省略し、重要な部分に焦点を当てて説明します。

2.3.1 解析の流れ

`expr("1+2")` の解析がどのように進むか、主要なポイントを見てみましょう：

1. 最初の数字「1」の解析

- `expr` は `term` を呼び出し、`term` は `factor` を、`factor` は `NUMBER` を呼び出します
- `NUMBER` は先頭の`"1"`がマッチするので、`1` を消費して（`SUCCESS, "+2"`）を返します
- この結果が `factor`、`term` を経由して `expr` まで戻ってきます

2. 演算子「+」の処理

- `expr` の定義 `term { ('+' | '-') term }` に従い、`+` が `('+' | '-')` にマッチします
- `+` を消費して、残りは`"2"`になります

3. 次の数字「2」の解析

- 再び `term` → `factor` → `NUMBER` と呼び出されます
- `NUMBER` は`"2"`にマッチし、`2` を消費して（`SUCCESS, ""`）を返します
- 入力が空になったので、解析は成功です

2.3.2 重要なポイント

この解析プロセスで注目すべき点は：

- **再帰的な構造**: `expr` → `term` → `factor` → `NUMBER` という呼び出しの連鎖
- **文字の消費**: マッチした文字を消費しながら、残りの文字列を次の解析に渡す
- **繰り返しの処理**: `{ }`で囲まれた部分は 0 回以上繰り返される
- **優先順位の実現**: 文法の階層構造により、自然に演算子の優先順位が守られる

このような再帰的な解析手法は「再帰下降構文解析」と呼ばれ、BNF の構造をそのままプログラムに反映できる直感的な方法です。

2.4 なぜこの BNF で演算子の優先順位を表現できるのか

BNF を使った素直な解析プロセスを理解したところで、重要な疑問に答えましょう。「なぜこのように書くと、掛け算・割り算が足し算・引き算より優先されるのか？」ということです。

その答えは、**文法の階層構造**にあります。BNF を改めて見てみましょう：

```
expr = term { ('+' | '-') term };
term = factor { ('*' | '/') factor };
```

```
factor = NUMBER | '(' expr ')';
```

この文法では、次のような階層関係があります：

1. expr（最上位）：足し算・引き算を扱う
2. term（中間層）：掛け算・割り算を扱う

3. factor（最下位）：数値や括弧で囲まれた式を扱う

2.4.1 優先順位が実現される仕組み

式 1+2*3 を例に、なぜ * が + より先に処理されるかを見てみましょう。

1. expr から解析を開始

- expr は term を呼び出す
- この時点では + や-を探すが、まだ term の解析中

2. term の解析

- term は factor（ここでは 1）を解析
- 次の文字が + なので、*や/ではない
- したがって term の解析は 1 で終了し、expr に戻る

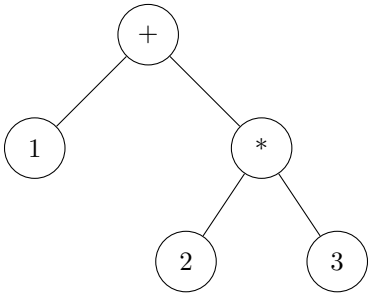
3. expr の続き

- + を見つけて消費
- 次の term（2*3 の部分）を解析

4. 2 番目の term の解析

- factor（2）を解析
- 次が*なので、term の繰り返し部分が適用される
- *を消費して、次の factor（3）を解析
- term 全体として 2*3 がひとまとまりになる

この結果として、1+(2*3) という構造が得られます。下位の規則で*や/が先に処理されることで、演算子の優先順位が自然に表現されているわけです。



2.4.2 もし文法の階層が逆だったら？

以下のような**誤った**文法を書いたとしましょう：

```
// 誤った文法（優先順位が逆）
expr = factor { ('*' | '/') factor };
factor = term { ('+' | '-') term };
term = NUMBER | '(' expr ')';
```

この場合、1+2*3 は (1+2)*3 として解析されてしまいます。なぜなら、以下のような解析手順になるからです：

1. expr が最初に*や/を探す
2. factor で 1+2 全体が先に処理される
3. その後で*3 が処理される

2.4.3 文法設計の原則

演算子の優先順位を正しく表現するための原則は以下のようになります：

1. 優先順位の低い演算子ほど上位の規則で扱う

- +, -は最上位の expr で
- *, /は中間の term で

2. 同じ優先順位の演算子は同じ規則内で扱う

- + と-は同じ expr 内
- *と/は同じ term 内

3. 左結合性は繰り返しで表現

- {...}による繰り返しが左から右への結合を実現

このような階層構造により、パーサーは自然に演算子の優先順位を守りながら解析を進めることができるのです。

2.5 抽象構文木

算術式の文法を BNF で表現し、実際に算術式を表す文字列と照合することができました。しかし、これでは文字列が与えられた BNF にマッチするかどうかの判定しかできません。

皆さんおなじみの JSON がそうであるように、実用的には解析した結果を何らかのデータ構造に変換して格納しておく必要があります。そのためのデータ構造が**抽象構文木**（Abstract Syntax Tree）です。

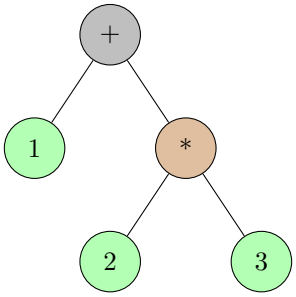
抽象構文木とは何でしょうか？ 今、私達が欲しいのは

- 空白や括弧といった余分な情報が含まれず
- 演算子の優先順位を表現できる

ような構造です。**抽象構文木**（Abstract Syntax Tree）は、そのようなニーズを満たすデータ構造です。

抽象構文木は一般には任意個の子を持つ多分木として表現されます。

たとえば、`1 + 2 * 3` という算術式の抽象構文木は以下のようになります。



抽象構文木の各ノードは、プログラムの構造を表現するためのデータ構造です。たとえば、`+` ノードは足し算を表し、`1` ノードは整数の `1` を表します。

抽象構文木（AST）では、各ノードはプログラムの構成要素を表し、親子関係によって演算子とその演算対象（オペランド）の関係性を示します。例えば、`1 + 2` という式では、`+` が演算子、`1` と `2` がオペランドです。抽象構文木には次のようなノードがあります。

2.5.1　内部ノード

演算子や制御構造など、他のノードを子として持つノードです。先程の例でいうと、`+` ノードと`*`ノードが内部ノードです。

BNF の観点から言えば、内部ノードは主に生成規則の右辺における記号の 2 つ以上の並びに対応します。

2.5.2　葉ノード

子を持たないノードです。例えば、`1` や `2`、`3` といった数値は葉ノードとなります。一般的には、リテラルや変数など、それ以上分解できない最小単位のノードが葉ノードとなります。

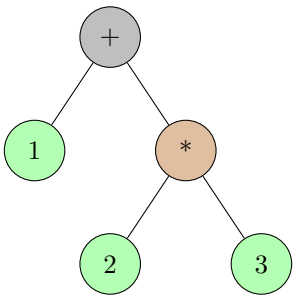
BNF の観点から言えば、葉ノードは終端記号（`'1'`、`'+'` など）に対応します。終端記号はそれ以上分解できない最終的な要素であり、それが抽象構文木の葉ノードになるわけです。

2.5.3　根ノード

抽象構文木の最上位に位置するノードです。抽象構文木の根となるノードであり、プログラム全体を表します。数式の例でいえば、全体を表す `+` ノードが根ノードとなります。内部ノードと根ノードは排他でないことに注意してください。たとえば、`+` ノードは内部ノードであり、同時に根ノードでもあります。

2.5.4 優先順位

改めて、先程の抽象構文木を見てみましょう。



この木構造では、+ がルートノードであり、その左の子が 1、右の子が*です。*ノードの子として 2 と 3 が配置されています。これにより、演算の優先順位が明確に表現されるわけです。

この抽象構文木を見れば、1 + (2 * 3) という演算順序が表現されており、(1 + 2) * 3 のような異なる解釈にはならないことが直感的に理解できるでしょう。

2.5.5 抽象構文木を Java で表現する

抽象構文木は視覚的な表現としても便利ですが、その真価はプログラム上で表現・処理することにあります。ここでは Java で表現する方法を紹介します。

```
// 式を表すインタフェース
sealed interface Exp permits Add, Sub, Mul, Div, Num {}
// 加算を表すレコード
// lhs: left-hand side (左辺)、rhs: right-hand side (右辺)
record Add(Exp lhs, Exp rhs) implements Exp {}
// 減算を表すレコード
record Sub(Exp lhs, Exp rhs) implements Exp {}
// 乗算を表すレコード
record Mul(Exp lhs, Exp rhs) implements Exp {}
// 除算を表すレコード
record Div(Exp lhs, Exp rhs) implements Exp {}
// 数値を表すレコード
record Num(int value) implements Exp {}
```

インタフェース Exp は、抽象構文木のノードを表すインタフェースです。

Add、Sub、Mul、Div、Num は、それぞれ加算、減算、乗算、除算、数値を表すレコードです。これによって、抽象構文木を Java で表現することができます。

試しに `1 + 2 * 3` の抽象構文木を Java で表現してみましょう。

```
// 1 + 2 * 3
Exp exp = new Add(
  new Num(1), new Mul(new Num(2), new Num(3))
);
```

各クラスのインスタンスを作成することで、抽象構文木を表現することができます。

2.5.6 抽象構文木を評価する

構文解析が完了して抽象構文木が得られると、その先にはどのような可能性が広がるのでしょうか。実は、抽象構文木さえ作ってしまえば、それを活用して様々なことができるようになります。その一例として、ここでは抽象構文木を「評価」して計算結果を得る方法を紹介します。

抽象構文木を作成するだけでは、プログラムの実行結果を得ることはできません。抽象構文木を評価するためには、再帰的な処理を行うことが必要です。抽象構文木を辿りながら、各ノードの演算を行うことで、プログラムの実行結果を得ることができます。

以下は、算術式の抽象構文木を評価する Java のコード例です。

```
int eval(Exp e) {
  return switch(e){
    case Num t -> t.value();
    case Add t -> eval(t.lhs()) + eval(t.rhs());
    case Sub t -> eval(t.lhs()) - eval(t.rhs());
    case Mul t -> eval(t.lhs()) * eval(t.rhs());
    case Div t -> {
      int divisor = eval(t.rhs()); // 先に右辺を評価
      if(divisor == 0) { // 評価結果でゼロチェック
        throw new ArithmeticException("division by zero");
      }
      yield eval(t.lhs()) / divisor; // 左辺を評価して除算
    }
  };
}
```

ノードの種類に応じて switch 式で処理を分岐しています：

- `Num` ノードの場合: 格納されている数値 `t.value()` をそのまま返します。これが再帰のベースケースとなります。

- Add ノードの場合: 左の子 t.lhs() と右の子 t.rhs() をそれぞれ再帰的に eval し、その結果を足し算します。
- Sub、Mul ノードの場合: Add と同様に、左右の子を再帰的に評価し、それぞれの演算を行います。
- Div ノードの場合: まず右の子（除数）を eval し、その結果が 0 であれば ArithmeticException をスローします。0 でなければ、次に左の子（被除数）を eval し、除算の結果を返します。ゼロ除算チェックは、実際の計算を行う前に行うことが重要です。また、eval(t.rhs()) を 2 回呼び出すのを避けるため、一度評価した結果を変数に格納しています。

この eval メソッドの動作を 1 + 2 * 3 の抽象構文木 new Add(new Num(1), new Mul(new Num(2), new Num(3))) で追ってみましょう。

- eval(new Add(new Num(1), new Mul(new Num(2), new Num(3)))) が呼び出される。
  - Add ケースにマッチ。
  - eval(t.lhs()) つまり eval(new Num(1)) が呼び出される。
    - Num ケースにマッチ。1 を返す。
  - eval(t.rhs()) つまり eval(new Mul(new Num(2), new Num(3))) が呼び出される。
    - Mul ケースにマッチ。
    - eval(t.lhs()) つまり eval(new Num(2)) が呼び出される。
      - Num ケースにマッチ。2 を返す。
    - eval(t.rhs()) つまり eval(new Num(3)) が呼び出される。
      - Num ケースにマッチ。3 を返す。
    - 2 * 3 の結果である 6 を返す。
  - 1 + 6 の結果である 7 を返す。

このように抽象構文木を再帰的に辿ることで、式の評価が実現されます。

Div の場合のゼロ除算エラー処理は、プログラムの実行時エラーを防ぐために不可欠です。他の演算（Add, Sub, Mul）では、Java の整数演算がオーバーフローする可能性はありますが、ArithmeticException のような実行時例外を直接引き起こす「不正な演算」は（ゼロ除算ほど明確には）定義されていないため、ここでは特にエラー処理を加えていません（もちろん、より堅牢な電卓を作る場合はオーバーフロー検知なども考慮に入れるべきです）。

eval メソッドを使うことで、次のように抽象構文木を評価することができます。

```
// 1 + 2 * 3
Exp exp = new Add(
  new Num(1), new Mul(new Num(2), new Num(3))
);
eval(exp); // 7
```

抽象構文木をデータとして表現することで、プログラムの構造を簡単に解析することができるのです。

2.6　まとめ

この章では、算術式の文法を例題として BNF について紹介し、BNF に基づいて算術式を解析する方法の概要について説明しました。また、抽象構文木についても紹介し、抽象構文木を Java で表現する方法と、抽象構文木を評価する方法について説明しました。

しかし、今のままでは BNF に基づく「構文解析器」は与えられた文字列が文法にマッチするかどうかを判定するだけで、抽象構文木を生成することができません。次章では、ここで学んだ BNF の考え方を応用して、実際に Java で動作する JSON の構文解析器を実装し、JSON の抽象構文木を生成する方法について詳しく説明します。

第 3 章

第 3 章 JSON の構文解析

2 章で構文解析に必要な基本概念について学ぶことができました。この章では JSON という実際に使われている言語を題材に、より実践的な構文解析のやり方を学んでいきます。

3.1 JSON の概要

JSON（JavaScript Object Notation）は、Web サービスにアクセスするための API で非常に一般的に使われているデータフォーマットです。また、企業内サービス間で連携するときにも非常によく使われます。皆さんは何らかの形で JSON に触れたことがあるのではないかと思います。

JSON は元々は、JavaScript のサブセットとして、オブジェクトに関する部分だけを切り出したものでしたが、現在は ECMA-404[*1]で標準化されており、色々な言語で JSON を扱うライブラリがあります。また、JSON はデータ交換用フォーマットの中でも非常にシンプルであるという特徴があり、そのシンプルさ故か、同じ言語でも JSON を扱うライブラリが乱立する程です。今の Web アプリケーション開発に携わる開発者にとって JSON は避けて通れないといってよいでしょう。

以降では簡単な JSON のサンプルを通して JSON の概要を説明します。

3.1.1 オブジェクト

以下は、二つの名前/値のペアからなる**オブジェクト**です。

```
{
  "name": "Kota Mizushima",
  "age":  41
}
```

[*1] 翻訳: ISO/IEC 14977:1996 Information technology — Syntactic metalanguage — Extended BNF https://hazm.at/mox/lang/meta-language/ebnf/iso-iec-14977-extended-bnf/index.html

この JSON は、name と"Kota Mizushima"という文字列の**ペア**と、age と 41 という数値の**ペア**からなる**オブジェクト**であることを示しています。

なお、用語については、ECMA-404 の仕様書に記載されているものに準拠しています。名前/値のペアは、属性やプロパティと呼ばれることもあるので、適宜読み替えてください。日本語で表現すると、このオブジェクトは、名前が Kota Mizushima、年齢が 41 という人物一人分のデータを表していると考えることができます。オブジェクトは、{} で囲まれた、"name":value の対が, を区切り文字として続く形になります。後述しますが、name の部分は**文字列**である必要があります。

3.1.2 配列

別の例として、以下の JSON を見てみます。

```
{
  "kind":"Rectangle",
  "points": [
    {"x":0,   "y":0  },
    {"x":0,   "y":100},
    {"x":100, "y":100},
    {"x":100, "y":0  },
  ]
}
```

この JSON は、

- "kind"と"Rectangle"のペア
- "points"と [...] のペア

からなるオブジェクトです。さらに、"points"に対応する値が**配列**になっていて、その中に以下の 4 つの要素が含まれています。

- オブジェクト：{"x":0,   "y":0}
- オブジェクト：{"x":0,   "y":100}
- オブジェクト：{"x":100, "y":100}
- オブジェクト：{"x":100, "y":0}

配列は、[] で囲まれた要素の並びで、区切り文字は, です。

このオブジェクトは、種類が四角形で、それを構成する点が (0, 0), (0, 100), (100, 100), (100, 0) からなっているデータを表現しているとみることができます。

3.1.3 数値

これまで見てきたオブジェクトと配列は複合的なデータでしたが、既に出てきているように、JSON にはこれ以上分解できないデータもあります。先ほどから出てきている数値もそうです。数値は、

```
1
10
100
1000
1.0
1.5
```

のような形になっており、整数または小数です。JSON での数値の解釈は特に規定されていない点に注意してください。たとえば、0.1 は 2 進法での小数だと解釈しても良いですし、10 進法での小数と解釈しても構いません。つまり、特に、IEEE の浮動小数点数である、といった規定はありません。

3.1.4 文字列

JSON のデータには文字列もあります。

```
"Hello, World"
"Kota Mizushima"
"hogehoge"
```

のように、""で囲まれたものが文字列となります。JSON の仕様では、オブジェクトのキーは必ずダブルクォーテーションで囲まれた文字列でなければなりません。たとえば、以下は **JavaScript** のオブジェクトリテラルとしては許容される場合がありますが（キー name が識別子の命名規則に合致するため）、**JSON** の定義には従っていません。

```
{
  name: "Kota Mizushima", // name がダブルクォーテーションで囲まれていない！
  age:  41
}
```

このような形式は、多くの JSON パーサーではエラーとして扱われます。JavaScript のオブジェクトリテラルと JSON の構文には違いがある点に注意してください。

3.1.5　真偽値

JSON には、多くのプログラミング言語にある真偽値もあります。JSON の真偽値は以下のように、`true` または `false` の二通りです。

```
true
false
```

真偽値も解釈方法は定められていませんが、ほとんどのプログラミング言語で、該当するリテラル表現があるので、おおむねそのような真偽値リテラルにマッピングされます。

3.1.6　null

多くのプログラミング言語にある要素ですが、JSON には `null` もあります。多くのプログラミング言語の JSON ライブラリでは、無効値に相当する値にマッピングされますが、JSON の仕様では `null` の解釈は定められていません。`null` に相当するリテラルがあればそれにマッピングされる事も多いですが、`Option` や `Maybe` といったデータ型によって `null` を表現する言語では、そのようなデータ型にマッピングされる事が多いようです。

3.1.7　JSON の全体像

ここまでで、JSON で現れる 6 つの要素について説明しましたが、JSON で使える要素は**これだけ**です。このシンプルさが、多くのプログラミング言語で JSON が使われる要因でもあるのでしょう。JSON で使える要素について改めて並べてみます。

- オブジェクト
- 配列
- 数値
- 文字列
- 真偽値
- `null`

次の節では、この JSON の**文法**が、どのような形で表現できるかについて見ていきます。

3.2　JSON の BNF

前の節で JSON の概要について説明し終わったところで、いよいよ JSON の文法について見ていきます。JSON の文法は ECMA-404 の仕様書に記載されていますが、ここでは、それを若干変形した BNF で表現された JSON の文法を見ていきます。

JSON の BNF による定義を簡略化したものは以下で全てです。特に小数点以下の部分は煩雑になる割に本質的でな

いので削除しました。また、文字列のエスケープシーケンス（\n、\t、\"など）も本章では扱いません。これらは構文解析の本質を理解する上では必須ではないためです。

```
json = ws value;
value = true | false | null | number | string | object | array;
object = LBRACE RBRACE | LBRACE pair {COMMA pair} RBRACE;
pair = string COLON value;
array = LBRACKET RBRACKET | LBRACKET value {COMMA value} RBRACKET ;
string = ("\"\"" | "\"" {CHAR} "\"") ws;
number = INT ws;
true = "true" ws;
false = "false" ws;
null = "null" ws;

COMMA = "," ws;
COLON = ":" ws;
LBRACE = "{" ws;
RBRACE = "}" ws;
LBRACKET = "[" ws;
RBRACKET = "]" ws;

ws = {" " | "\t" | "\n" | "\r"} ;
CHAR = "a" | "b" | "c" | "d" | "e" | "f" | "g" | "h" | "i" | "j" | "k" | "l" | "m" |
       "n" | "o" | "p" | "q" | "r" | "s" | "t" | "u" | "v" | "w" | "x" | "y" | "z" |
       "A" | "B" | "C" | "D" | "E" | "F" | "G" | "H" | "I" | "J" | "K" | "L" | "M" |
       "N" | "O" | "P" | "Q" | "R" | "S" | "T" | "U" | "V" | "W" | "X" | "Y" | "Z" |
       "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" |
       " " | "!" | "#" | "$" | "%" | "&" | "'" | "(" | ")" | "*" | "+" | "," |
       "-" | "." | "/" | ":" | ";" | "<" | "=" | ">" | "?" | "@" | "[" | "]" |
       "^" | "_" | "`" | "{" | "|" | "}" | "~" ;
INT = ["-"] ("0" | (NONZERO {DIGIT})) ;
DIGIT = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" ;
NONZERO = "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" ;
```

これまで説明したJSONの要素と比較して見慣れない記号が出てきましたが、一つ一つ見て行きましょう。

なお、規則名の大文字・小文字の使い分けについてですが、COMMA、LBRACEのような大文字で書かれた規則は、後の字句解析の節で出てくる「トークン」に対応します。これらは単一の記号や固定の文字列を表すもので、構文解析の際には一つの単位として扱われます。一方、json、valueのような小文字の規則は、より複雑な構造を表します。

3.2.1 json

一番上から読んでいきます。2 章の復習になりますが、BNF では、

```
json = ws value;
```

のような**規則**の集まりによって、文法を表現します。=の左側である json が**左辺**で、右側（ここでは ws value）が**本体**になります。さらに、本体の中に出てくる、他の規則を参照する部分（ここでは value や ws) を非終端記号と呼びます。非終端記号は同じ BNF で定義されている規則の左辺と一致する必要があります。

この規則を日本語で表現すると、「json という名前の規則は、ws の後に value が続く」と読むことができます。value は、JSON の値を表しているので、json という規則は ws（空白文字）の後に JSON の値が続くものを表しています。

3.2.2 object

object は JSON のオブジェクトを表す規則で、定義は以下のようになっています。

```
object = LBRACE RBRACE | LBRACE pair {COMMA pair} RBRACE;
```

pair の定義はのちほど出てきますので心配しないでください。

この規則によって object は

- ブレースで囲まれたもの（LBRACE RBRACE) である
  - LBRACE は Left-Brace（開き波カッコ）の略で{を示しています
  - RBRACE は Right-Brace（閉じ波カッコ）の略で}を示しています
- LBRACE が来た後に、pair が 1 回出現して、さらにその後に、「COMMA（カンマ）とそれに続く pair」というペアが 0 回以上繰り返して出現した後、RBRACE が来る

のどちらかであることを表しています。

具体的な JSON を当てはめてみましょう。以下の JSON は LBRACE RBRACE にマッチします。

```
{}
```

以下の JSON は LBRACE pair {COMMA pair} RBRACE にマッチします。

```
{"x":1}
{"x":1,"y":2}
{"x":1,"y":2,"z":3}
```

しかし、以下のテキストは、object に当てはまらず、エラーになります。{COMMA pair}とあるように、カンマは後ろにペアを必要とするからです。

```
{"x":1,} // , で終わっている
```

3.2.3 pair

pair（ペア）は、JSON のオブジェクト内での"x":1 に当たる部分を表現する規則です。value の定義については後述します。

```
pair = string COLON value;
```

これによってペアは:（COLON）の前に文字列リテラル（string) が来て、その後に JSON の値（value）が来ることを表しています。pair にマッチするテキストとしては、

```
"x":1
"y":true
```

などがあります。一方で、以下のテキストは pair にマッチしません。JavaScript のオブジェクトと JSON が違う点です。

```
x:1 // 文字列リテラルでないといけない
```

3.2.4 COMMA

COMMA は、カンマを表す規則です。カンマそのものを表すには、単に","と書けばいいのですが、任意個の空白文字が続くことを表現したいため、規則 ws（後述）を参照しています。

```
COMMA = "," ws;
```

3.2.5 array

array は、JSON の値の配列を表す規則です。

```
array = LBRACKET RBRACKET | LBRACKET value {COMMA value} RBRACKET ;
```

LBRACKET は開き大カッコ（[）を、RBRACKET は閉じ大カッコ（]) を表しています。value の定義については後述します。

これによって array は、

- 大カッコで囲まれたもの（LBRACKET RBRACKET) である
- 開き大カッコ（LBRACKET）が来た後に、value が 1 回あらわれ、さらにその後に、「COMMA（カンマ）とそれに続く value」というペアが 0 回以上繰り返してあらわれた後、閉じ大カッコが来る（RBRACKET)

のどちらかであることを表しています。よく見ると、先程の object と同様の構造を持っていることがわかります。

array についても具体的な JSON を当てはめてみましょう。以下の JSON は LBRACKET RBRACKET にマッチします。

```
[]
```

また、以下の JSON は LBRACKET value {COMMA value} RBRACKET にマッチします。

```
[1]
[1, 2]
[1, 2, 3]
["foo"]
```

しかし、以下のテキストは、array に当てはまらず、エラーになります。{COMMA pair}とあるように、カンマは必ず後ろに value を必要とするからです。

```
[1,] // , で終わっている
```

3.2.6 value

```
value = true | false | null | number | string | object | array;
```

value は JSON の値を表現する規則です。これは、JSON の値は、

- 真（true）
- 偽（false）
- ヌル（null）
- 数値（number）
- 文字列（string）
- オブジェクト（object）
- 配列（array）

のいずれかでなければいけない事を示しています。JSON を普段使っている皆さんにはお馴染みでしょう。

3.2.7 true

true は、真を表すリテラルを表す規則です。

```
true = "true" ws;
```

文字列 true が真を表すということでそのまますね。

3.2.8 false

false は、偽を表すリテラルを表す規則です。構造的には、true と同じです。

```
false = "false" ws;
```

3.2.9 null

`null` は、ヌルリテラルを表す規則です。構造的には、`true` や `false` と同じです。

```
null = "null" ws;
```

`null` は、ヌル値があるプログラミング言語だと、その値にマッピングされますが、ここではあくまでヌル値は `null` で表されることしか言っておらず、**意味は特に規定していない**ことに注意してください。

3.2.10 number

`number` は、数値リテラルを表す規則です。

```
number = INT ws;
```

整数（`INT`）に続いて、`ws` が来るのが `number` であるということを表現しています。

3.2.11 string

`string` は文字列リテラルを表す規則です。

```
string = ("\"\"" | "\"" CHAR+ "\"") ws;
```

`"`で始まって、`CHAR` で定義される文字が 0 個以上続いて、`"` で終わります。`CHAR` の定義は BNF 中に含まれており、ダブルクォーテーションとバックスラッシュを除く印字可能文字を表しています。なお、本章ではエスケープシーケンスは扱わないため、バックスラッシュを含む文字列は処理できません。

3.2.12 JSON の BNF まとめ

JSON の BNF は、非常に少数の規則だけで表現することができます。読者の中には、あまりにも簡潔過ぎて驚かれた方もいるのではないでしょうか。しかし、これだけ単純であるにも関わらず、JSON の BNF は**再帰的に定義されている**ため、非常に複雑な構造も表現することができます。たとえば、

- 一要素の配列があり、その要素はオブジェクトであり、キー`"a"`に対応する要素の中に配列があって、その配列は空配列である

といったことも、JSON では以下のように表現することができます。

```
[{"a":[]}]
```

再帰的な規則は、構文解析において非常に重要な要素なので、これから本書を読み進める上でも念頭に置いてください。

3.3　JSON の抽象構文木

JSON の定義と、文法について見てきました。構文解析器を実装する前に、まず JSON の抽象構文木（AST: Abstract Syntax Tree）を Java でどのように表現するかを定義しましょう。

抽象構文木は、2 章でも説明したとおり、プログラムの構造を表現するためのデータ構造です。重要な点は、抽象構文木では元の文字列に含まれていた「構文上のノイズ」が取り除かれることです。例えば：

- 空白文字（`ws`）は抽象構文木には含まれません
- カンマ（`,`）やコロン（`:`）などの区切り文字も、構造として表現されるため個別のノードにはなりません
- 括弧（`{}`、`[]`）も同様に、オブジェクトや配列という構造で表現されます

以下の `JsonAst` の定義を見ると、JSON の各要素を Java のクラスとして表現していることがわかります。これは、JSON という「文字列」を、Java のオブジェクトという「構造化されたデータ」に変換するための定義です。

```
public interface JsonAst {
    // value
    sealed interface JsonValue permits
        JsonNull, JsonTrue, JsonFalse,
        JsonNumber, JsonString,
        JsonObject, JsonArray {}

    // NULL
    record JsonNull() implements JsonValue {
        @Override
        public String toString() {
            return "null";
        }
    }

    // TRUE
    record JsonTrue() implements JsonValue {
        @Override
        public String toString() {
            return "true";
        }
    }
```

```
// FALSE
record JsonFalse() implements JsonValue {
    @Override
    public String toString() {
        return "false";
    }
}

// NUMBER
record JsonNumber(double value) implements JsonValue {
    @Override
    public String toString() {
        return "JsonNumber(" + value + '}';
    }
}

// STRING
record JsonString(String value) implements JsonValue {
    @Override
    public String toString() {
        return "JsonString(\"" + value + "\")";
    }
}

// object
record JsonObject(List<Pair<JsonString, JsonValue>> properties)
    implements JsonValue {
    @Override
    public String toString() {
        return "JsonObject{" + properties + '}';
    }
}

// array
record JsonArray(List<JsonValue> elements)
    implements JsonValue {
    @Override
```

```
        public String toString() {
            return "JsonArray[" + elements + ']';
        }
    }
}
```

各クラスが BNF の左辺に対応しているのがわかるでしょうか。ただし、前述のとおり、ws や区切り文字（,、:）などは抽象構文木には現れません。これらは構文解析の過程で消費されますが、最終的なデータ構造には含まれないのです。

3.4 JSON の構文解析器

この節では、BNF を元に、JSON のデータを**構文解析**するプログラムを実装していきます。以下のようなインタフェース JsonParser インタフェースを実装したクラスを「JSON の構文解析器」と考えることにします。

```
package parser;
interface JsonParser {
    public ParseResult<JsonAst.JsonValue> parse(String input);
}
```

クラス ParseResult<T>は以下のようなジェネリックなクラスになっています。value は解析結果の値です。これは任意の型をとり得るので、T としています。また、input は「構文解析の対象となる文字列」を表します。

```
public record ParseResult<T>(T value, String input) {}
```

インタフェース JsonParser は parse() メソッドだけを持ちます。parse() メソッドは、文字列 input を受け取り、ParseResult<JsonAst.JsonValue>型を返します。

3.4.1 構文解析の戦略

実装に入る前に、これから作る JSON パーサーの解析戦略について説明しておきます。

1. トップダウン解析: BNF の規則を上から順番に適用していく方式を採用します。例えば、value 規則から始めて、それぞれの選択肢を試していきます。
2. バックトラック: 解析に失敗した場合、前の状態に戻って別の選択肢を試します。これにより、複数の可能性がある場合でも正しい解析結果を見つけることができます。
3. 順序付き選択（Ordered Choice）: BNF の | で区切られた選択肢を、左から右へ順番に試します。例えば、value = true | false | null | number | string | object | array の場合、まず true を試し、失敗したら false を試す、という具合です。この順序は重要で、より具体的なパターンを先に配置することで、

正しい解析を保証します。実装の `parseValue()` メソッドでも、BNF の定義順序と同じ順番で各選択肢を試すようにしています。

4. 例外による失敗の表現: 構文解析の失敗は、`ParseException` という例外で表現します。これにより、深くネストした解析処理から簡潔に失敗を伝播させることができます。

なお、クラス名を `PegJsonParser` としているのは、この実装が後の章で説明する PEG（Parsing Expression Grammar）の考え方に基づいているためです。PEG は順序付き選択とバックトラックを特徴とする文法形式で、本実装もこれらの特徴を持っています。

3.4.2 構文解析器の全体像

それでは、JSON の構文解析器の実装を見ていきましょう。構文解析を行う各メソッドについては省略して、後ほど解説することにします。以下が `PegJsonParser` クラスの全体像です。

```
package parser;

import java.util.ArrayList;
import java.util.List;

public class PegJsonParser implements JsonParser {
    private int cursor;
    private String input;

    private static class ParseException extends RuntimeException {
        public ParseException(String message) {
            super(message);
        }
    }

    public ParseResult<JsonAst.JsonValue> parse(String input) {
        this.input = input;
        this.cursor = 0;
        var value = parseValue();
        return new ParseResult<>(value, input.substring(this.cursor));
    }

    /**
     * 指定された文字列リテラルが現在のカーソル位置にあることを確認し、
```

```
     * マッチした場合はカーソルを進めます。
     * @param literal 認識すべき文字列リテラル
     * @throws ParseException リテラルがマッチしない場合
     */
    private void recognize(String literal) {
        if(input.substring(cursor).startsWith(literal)) {
            cursor += literal.length();
        } else {
            String substring = input.substring(cursor);
            int endIndex = cursor +
              (literal.length() > substring.length() ?
                substring.length() : literal.length());
            throw new ParseException(
                "expected: " + literal +
                ", actual: " + input.substring(cursor, endIndex)
            );
        }
    }


    /**
     * 空白文字（スペース、タブ、改行、キャリッジリターン）を
     * スキップし、カーソルを次の非空白文字まで進めます。
     */
    private void skipWhitespace() {
        OUTER:
        while(cursor < input.length()) {
            char currentCharacter = input.charAt(cursor);
            switch (currentCharacter) {
                // 空白文字
                case ' ':       // Space
                case '\t':      // Horizontal Tab
                case '\n':      // Line Feed
                case '\r':      // Carriage Return
                    cursor++;
                    continue OUTER;
                default:
```

```
                break OUTER;
            }
        }
    }

    /**
     * JSON の値（value）を解析します。
     * BNF: value = true / false / null / number / string / object / array
     * @return 解析された JSON 値
     * @throws ParseException 有効な JSON 値が見つからない場合
     */
    private JsonAst.JsonValue parseValue() {
        // 後で解説
    }



    /**
     * true 値を解析します。
     * BNF: true = "true" ws
     * @return JSON の true 値
     * @throws ParseException "true"がマッチしない場合
     */
    private JsonAst.JsonTrue parseTrue() {
        // 後で解説
    }

    // false = "false" ws;
    private JsonAst.JsonFalse parseFalse() {
        // 後で解説
    }

    // null = "null" ws;
    private JsonAst.JsonNull parseNull() {
        // 後で解説
    }

    // LBRACE = '{' ws;
```

```
    private void parseLBrace() {
        // 後で解説
    }

    // RBRACE = '}' ws;
    private void parseRBrace() {
        // 後で解説
    }

    // LBRACKET = '[' ws;
    private void parseLBracket() {
        // 後で解説
    }

    // RBRACKET = ']' ws;
    private void parseRBracket() {
        // 後で解説
    }

    // COMMA = ',' ws;
    private void parseComma() {
        // 後で解説
    }

    // COLON = ':' ws;
    private void parseColon() {
        // 後で解説
    }

    // string = ('"""' | '"' {CHAR} '"') ws;
    private JsonAst.JsonString parseString() {
        // 後で解説
    }

    // number = INT ws;
    private JsonAst.JsonNumber parseNumber() {
        // 後で解説
```

```
    }

    // pair = string COLON value;
    private Pair<JsonAst.JsonString, JsonAst.JsonValue> parsePair() {
        // 後で解説
    }

    // object = LBRACE RBRACE | LBRACE pair {COMMA pair} RBRACE;
    private JsonAst.JsonObject parseObject() {
        // 後で解説
    }

    // array = LBRACKET RBRACKET | LBRACKET value {COMMA value} RBRACKET;
    public JsonAst.JsonArray parseArray() {
        // 後で解説
    }
}
```

このクラス PegJsonParser で重要なことは、クラスがフィールドとして以下を保持していることです。

```
public class PegJsonParser implements JsonParser {
    private int cursor;
    private String input;
    // ...
}
```

構文解析器を実装する方法としては、同じ入力文字列を与えれば同じ解析結果が返ってくるような関数型の実装方法と、今回のように、現在どこまで読み進めたかによって解析結果が変わる手続き型の方法があるのですが、手続き型の方が説明しやすいので、本書では手続き型の実装方法を採用しています。

また、skipWhitespace() メソッドと recognize() メソッドを定義して、空白文字の読み飛ばしと、特定の文字列の認識を行います。さらに、構文解析中にエラーが発生した場合は ParseException を投げることで、どこでどのような問題が発生したかを呼び出し元に伝えます。

cursor フィールドは現在の読み取り位置を、input フィールドは解析対象の文字列を保持します。エラーが発生した場合は ParseException を投げることで、どこでどのような問題が発生したかを呼び出し元に伝えます。

次からはいよいよ構文解析の各メソッドを見ていきます。

3.4.3　value の構文解析メソッド

まずは、parseValue() メソッドから始めましょう。

```
    private JsonAst.JsonValue parseValue() {
        int backup = cursor;
        try {
            return parseTrue();
        } catch (ParseException e) {
            cursor = backup;
        }

        try {
            return parseFalse();
        } catch (ParseException e) {
            cursor = backup;
        }
        // parseNull(), parseNumber(), parseString(),
        // parseObject(), parseArray() と続く...
    }
```

parseValue() の実装は、BNF の value = true | false | null | number | string | object | array という定義に対応しています。

|は「または」を意味するので、「value は、true または false または null または...」という意味になります。このコードでは、それぞれの可能性を順番に試していきます。

1. まず parseTrue() を試し、失敗したらカーソルを元に戻す
2. 次に parseFalse() を試し、失敗したらカーソルを元に戻す
3. 以下同様に続く...

このように、失敗したら元の位置に戻って別の可能性を試すことを「バックトラック」と呼びます。

ちなみに、この順番は重要です。value の例では右辺がそれぞれ排他的なので問題ありませんが、順番を変えると結果が変わってしまうことがあります。

3.4.4　true の構文解析メソッド

true の構文解析は、次のような parseTrue() メソッドとして定義します。

```
private JsonAst.JsonTrue parseTrue() {
    recognize("true");
    skipWhitespace();
    return new JsonAst.JsonTrue();
}
```

このメソッドで行っていることを見ていきましょう。このメソッドでは、入力である input の現在位置が"true"という文字列で始まっているかをチェックします。もしそうなら、**JSON の true** をあらわす JsonAst.JsonTrue のインスタンスを返します。もし、先頭が"true"でなければ、構文解析は失敗なので例外を発生させますが、これは recognize() メソッドの中で行われています。recognize() の内部では、入力の現在位置と与えられた文字列を照合して、マッチしない場合例外を投げます。

次に、skipWhitespace() メソッドを呼び出して、「空白の読み飛ばし」を行っています。

3.4.5 false の構文解析メソッド

false の構文解析は、次のような parseFalse() メソッドとして定義します。

```
private JsonAst.JsonFalse parseFalse() {
    recognize("false");
    skipWhitespace();
    return new JsonAst.JsonFalse();
}
```

これも、parseTrue() とほぼ同じですので、特に説明の必要はないでしょう。

3.4.6 null の構文解析メソッド

null の構文解析は、次のような parseNull() メソッドとして定義します。

```
private JsonAst.JsonNull parseNull() {
    recognize("null");
    skipWhitespace();
    return new JsonAst.JsonNull();
}
```

これも、parseTrue() や parseFalse() とほぼ同じです。固定の文字列を解析するという点で、これらのメソッドは共通の処理になっています。

3.4.7　数値の構文解析メソッド

数値の構文解析は、次のような parseNumber() メソッドとして定義します。

```
    private JsonAst.JsonNumber parseNumber() {
        // 本書の実装では、JSON の数値型を整数のみに限定して扱います。
        int start = cursor;
        char ch = 0;
        // オプショナルなマイナス記号
        if (cursor < input.length() && input.charAt(cursor) == '-') {
            cursor++;
        }

        // 数字の連続（0, または 1-9 で始まり数字が続く）
        int digitsStart = cursor;
        if (cursor < input.length()) {
            ch = input.charAt(cursor);
            if (ch == '0') {
                cursor++;
            } else if ('1' <= ch && ch <= '9') {
                cursor++;
                while (cursor < input.length()) {
                    ch = input.charAt(cursor);
                    if (!('0' <= ch && ch <= '9')) break;
                    cursor++;
                }
            } else {
                // 数字で始まらない場合はエラー
                throw new ParseException(
                    "expected: digit actual: " +
                    (cursor < input.length() ? input.charAt(cursor) : "EOF")
                );
            }
        } else {
             throw new ParseException("expected: digit actual: EOF");
        }
```

```
        if (digitsStart == cursor) {
            // 数字が一つも読まれなかった場合（"-"のみなど）
            throw new ParseException(
                "expected: digit after '-' actual: " +
                (cursor < input.length() ? input.charAt(cursor) : "EOF")
            );
        }

        String numberStr = input.substring(start, cursor);
        try {
            double value = Double.parseDouble(numberStr);
            skipWhitespace();
            return new JsonAst.JsonNumber(value);
        } catch (NumberFormatException e) {
            throw new ParseException("invalid number format: " + numberStr);
        }
    }
```

parseNumber() メソッド（PegJsonParser 内）では、入力文字列から数値部分を読み取り、Double.parseDouble を用いて数値に変換しています。この実装は、ECMA-404 で定義される JSON の数値型の完全な仕様（小数部、指数部を含む）には対応しておらず、整数のみを扱えるように単純化されています。

3.4.8 文字列の構文解析メソッド

文字列の構文解析は、次のような parseString() メソッドとして定義します。

```
    private JsonAst.JsonString parseString() {
        if(cursor >= input.length()) {
            throw new ParseException("expected: \"" + " actual: EOF");
        }
        char ch = input.charAt(cursor);
        if(ch != '"') {
            throw new ParseException("expected: \"" + "actual: " + ch);
        }
        cursor++;
        var builder = new StringBuilder();
        OUTER:
```

```
        while(cursor < input.length()) {
            ch = input.charAt(cursor);
            switch(ch) {
                case '\\':
                    throw new ParseException(
                        "escape sequences are not supported in this parser"
                    );
                    break;
                case '"':
                    cursor++;
                    break OUTER;
                default:
                    builder.append(ch);
                    cursor++;
                    break;
            }
        }

        if(ch != '"') {
            throw new ParseException("expected: " + "\"" + " actual: " + ch);
        } else {
            skipWhitespace();
            return new JsonAst.JsonString(builder.toString());
        }
    }
```

`while` 文の中が若干複雑になっていますが、一つ一つ見ていきます。

まず、最初の部分では、

```
        if(cursor >= input.length()) {
            throw new ParseException("expected: \"" + " actual: EOF");
        }
        char ch = input.charAt(cursor);
        if(ch != '"') {
            throw new ParseException("expected: \"" + "actual: " + ch);
        }
        cursor++;
```

- 入力が終端に達していないこと
- 入力の最初が"であること

をチェックしています。文字列は当然ながら、ダブルクォートで始まりますし、入力の終端にも達していないはずですから、それらの条件が満たされなければ例外が投げられるわけです。

while 文の中では、各文字を読み込んで文字列を構築していきます。switch 文の中で最も重要なのは default ケースで、ここで通常の文字を StringBuilder に追加して文字列を構築しています。ダブルクォート（"）が現れたら文字列の終端として処理し、バックスラッシュ（\）が現れた場合はエラーとして処理します。実際の JSON パーサーではエスケープシーケンスの処理が必要ですが、構文解析の本質を理解する上では必須ではないため、本書では省略しています。

while 文が終わったあとで、

```
        if(ch != '"') {
            throw new ParseException("expected: " + "\"" + " actual: " + ch);
        } else {
            skipWhitespace();
            return new JsonAst.JsonString(builder.toString());
        }
        throw new RuntimeException("never reach here");
```

というチェックを入れることによって、ダブルクォートで文字列が終端している事を確認した後、空白を読み飛ばしています。

3.4.9　配列の構文解析メソッド

配列の構文解析は、次のような parseArray() メソッドとして定義します。

```
    public JsonAst.JsonArray parseArray() {
        int backup = cursor;
        try {
            parseLBracket();
            parseRBracket();
            return new JsonAst.JsonArray(new ArrayList<>());
        } catch (ParseException e) {
            cursor = backup;
        }

        parseLBracket();
```

```
        List<JsonAst.JsonValue> values = new ArrayList<>();
        var value = parseValue();
        values.add(value);
        try {
            while (true) {
                backup = cursor;
                parseComma();
                value = parseValue();
                values.add(value);
            }
        } catch (ParseException e) {
            cursor = backup;
            parseRBracket();
            return new JsonAst.JsonArray(values);
        }
    }
```

この parseArray() は多少複雑になります。まず、先頭に"["が来るかチェックする必要があります。これをコードにすると、以下のようになります。

```
parseLBracket();
```

parseLBracket() は以下のように定義されています。

```
 private void parseLBracket() {
    recognize("[");
    skipWhitespace();
}
```

recognize() で、現在の入力位置が [と一致しているかチェックをした後、空白を読み飛ばしています。

[の次には任意の JsonValue または"]"が来る可能性があります。この時、まず最初に、] が来ると**仮定**するのがポイントです。

```
        int backup = cursor;
        try {
            parseLBracket();
            parseRBracket();
```

```
            return new JsonAst.JsonArray(new ArrayList<>());
        } catch (ParseException e) {
            cursor = backup;
        }
```

仮定が成り立たなかった場合、ParseException が throw されるはずですから、それを catch して、バックアップした位置に巻き戻します。

] が来るという仮定が成り立たなかった場合、再び最初に [が出現して、その次に来るのは任意の JsonValue ですから、以下のようなコードになります。

```
        parseLBracket();
        List<JsonAst.JsonValue> values = new ArrayList<>();
        var value = parseValue();
        values.add(value);
```

ローカル変数 values は、配列の要素を格納するためのものです。

配列の中で、最初の要素が読み込まれた後、次に来るのは、, か] のどちらかですが、ひとまず、, が来ると仮定して while ループで

```
parseComma();
value = parseValue();
values.add(value);
```

を繰り返します。この繰り返しは、1 回ごとに必ず入力を 1 以上進めます。失敗した時は、テキストが正しい JSON なら、] が来るはずなので、

```
parseRBracket();
return new JsonAst.JsonArray(values);
```

とします。もし、テキストが正しい JSON でない場合、parseRBracket() から例外が投げられるはずですが、その例外は**より上位の層が適切にリカバーしてくれると期待して**放置します。JSON のような再帰的な構造を解析する時、このような、「自分の呼び出し元が適切にやってくれるはず」（何故なら、自分はその呼び出し元で適切に catch しているのだから）という考え方が重要になります。

多少複雑になりましたが、parseArray() の定義が、EBNF における表記

```
array = LBRACKET RBRACKET | LBRACKET {value {COMMA value}} RBRACKET ;
```

に対応していることがわかるでしょうか。読み方のポイントは、|の後を、例外をキャッチした後の処理ととらえることです。

3.4.10　オブジェクトの構文解析メソッド

オブジェクトの構文解析は、次のような parseObject() メソッドとして定義します。

```
    private JsonAst.JsonObject parseObject() {
        int backup = cursor;
        try {
            parseLBrace();
            parseRBrace();
            return new JsonAst.JsonObject(new ArrayList<>());
        } catch (ParseException e) {
            cursor = backup;
        }

        parseLBrace();
        List<Pair<JsonAst.JsonString, JsonAst.JsonValue>> members =
            new ArrayList<>();
        var member = parsePair();
        members.add(member);
        try {
            while (true) {
                backup = cursor;
                parseComma();
                member = parsePair();
                members.add(member);
            }
        } catch (ParseException e) {
            cursor = backup;
            parseRBrace();
            return new JsonAst.JsonObject(members);
        }
    }
```

この定義を見て、ひょっとしたら、

「あれ？ これ、parseArray() とほとんど同じでは」

と気づかれた読者の方が居るかも知れません。実際、parseObject() がやっていることは parseArray() と非常に類似しています。

最初に、

```
parseLBrace();
parseRBrace();
return new JsonAst.JsonObject(new ArrayList<>());
```

としている箇所は、{}という形の空オブジェクトを読み取ろうとしていますが、これは、空配列 [] を読み取るコードとほぼ同じです。

続くコードも、対応する記号が{}か [] の違いこそあるものの、基本的に同じです。唯一の違いは、オブジェクトの各要素は、name:value というペアなため、parseValue() の代わりに parsePair() を呼び出しているところくらいです。

そして、parsePair() は以下のように定義されています。

```
    private Pair<JsonAst.JsonString, JsonAst.JsonValue> parsePair() {
        var key = parseString();
        parseColon();
        var value = parseValue();
        return new Pair<>(key, value);
    }
```

これは EBNF における以下の定義にそのまま対応しているのがわかるでしょう。

```
pair = string COLON value;
```

3.4.11 構文解析における再帰

配列やオブジェクトの構文解析メソッドを見るとわかりますが、

- parseArray() -> parseValue() -> parseArray()
- parseArray() -> parseValue() -> parseObject()
- parseObject() -> parseValue() -> parseObject()
- parseObject() -> parseValue() -> parseArray()

のような再帰呼び出しが起こり得ることがわかります。このような再帰呼び出しでは、各ステップで必ず構文解析が 1 文字以上進むため、JSON がどれだけ深くなっても（スタックが溢れない限り）うまく構文解析ができるのです。

3.4.12 構文解析と PEG

このようにして JSON の構文解析器を実装することができました。

実は、ここで作った構文解析器には、以下のような特徴があります：

1. **バックトラック**：失敗したら元の位置に戻って別の可能性を試す
2. **順序付き選択**：|で区切られた選択肢を左から順番に試す
3. **文字列を直接解析**：特別な前処理なしに、入力文字列をそのまま解析する

このような構文解析の手法を **PEG（Parsing Expression Grammar、解析表現文法）**と呼びます。PEG は 2004 年に提案された比較的新しい手法で、プログラミング言語のような「曖昧さがない」言語の解析に適しています。最近では Python（バージョン 3.9 以降）も PEG ベースの構文解析器を使っています。

厳密には PEG 自体は BNF と異なる文法の定義方法（形式文法）ですが、PEG 自体が「どのように解析されるか」を定義するため、ここでは PEG 自体を構文解析の手法として扱います。この点については第 5 章で詳しく解説します。

PEG は直感的でシンプルなので、最初に学ぶのに適しています。ただし、従来から使われている別の構文解析手法も重要です。次の節では、その伝統的な手法について解説します。

3.5　古典的な構文解析器

前節では、PEG という手法を使って構文解析器を作りました。しかし、伝統的な構文解析の手法では、少し違ったアプローチをとります。

伝統的な手法では、構文解析を以下の 2 つのステップに分けます：

1. **字句解析（トークナイザー）**：文字列を「トークン」という単位に分割する
2. **構文解析（パーサー）**：トークンの列から構造を組み立てる

たとえば、以下の英文があったとします。

```
We are parsers.
```

我々は構文解析器であるというジョーク的な文ですが、それはさておき、この文は

```
[We, are, parsers]
```

という三つのトークン（単語）に分解すると考えるのが字句解析の発想法です。

古典的な構文解析の世界では、字句解析が必須とされていましたが、それは後の章で説明される構文解析アルゴリズムの都合に加えて、空白のスキップという処理を字句解析で行えるからでもあります。

前節で出てきた JSON の構文解析器では `skipWhitespace()` の呼び出しが頻出していましたが、字句解析器を使う場合、空白を読み飛ばす処理を先に行うことで、構文解析器では空白の読み飛ばしという作業をしなくてよくなります。

この点はトレードオフがあって、たとえば、空白に関する規則がある言語の中でブレがある場合には、字句解析という前処理はかえってしない方が良いということすらあります。ともあれ、字句解析という前処理を通すことには一定のメリットがあるのは確かです。

以下では字句解析器を使った構文解析器の全体像を示します。ここでは、コアとなるアイデアに絞って説明します。完全なソースコードは巻末の付録を参照してください。

3.5.1 JSON の字句解析器

JSON の字句解析器（トークナイザー）の基本構造は次のようになります。

```
public class SimpleJsonTokenizer implements JsonTokenizer {
    private final String input;
    private int index;

    // すべての入力を一度にトークン化
    public List<Token> tokenizeAll() {
        List<Token> tokens = new ArrayList<>();
        while (moveNext()) {
            tokens.add(current());
        }
        tokens.add(new Token(Token.Type.EOF, null));
        return tokens;
    }

    // 次のトークンを読み取る
    private boolean moveNext() {
        skipWhitespace();

        if (index >= input.length()) {
            return false;
        }

        char ch = input.charAt(index);
        switch (ch) {
            case '"':
                return tokenizeStringLiteral();
            case '{':
                accept("{", Token.Type.LBRACE, "{");
                return true;
            case '[':
                accept("[", Token.Type.LBRACKET, "[");
```

```
                return true;
            // 他のトークンも同様に処理...
        }
    }

    // 文字列トークンの読み取り（エスケープシーケンス非対応）
    private boolean tokenizeStringLiteral() {
        if(input.charAt(index) != '"') return false;
        index++;
        var builder = new StringBuilder();
        while(index < input.length()) {
            char ch = input.charAt(index);
            if(ch == '"') {
                fetched = new Token(Token.Type.STRING, builder.toString());
                index++;
                return true;
            }
            builder.append(ch);
            index++;
        }
        return false;
    }
}
```

このコードは SimpleJsonTokenizer クラスの基本構造を示しています。主な特徴は以下の通りです：

- tokenizeAll()：
  - 入力文字列全体を一度に処理してトークン列を生成
  - moveNext() を繰り返し呼び出してトークンを収集
  - 最後に EOF（End Of File）トークンを追加
- moveNext()：
  - まず空白文字をスキップ
  - 現在の文字を見て、トークンの種類を判定
  - switch 文で各文字に応じた処理を実行
    - " → 文字列リテラルの処理
    - {、[→ 括弧トークンの生成
    - その他の文字も同様にパターンマッチング
- tokenizeStringLiteral() メソッド：

- ダブルクォートで囲まれた文字列を読み取る
- 閉じクォートが見つかるまで文字を収集
- エスケープシーケンス非対応

この設計により、字句解析の責任が明確になり、構文解析器は純粋にトークンの構造解析に専念できます。

3.5.2 JSON の構文解析器

この字句解析器を使った構文解析器の基本構造を示します。トークナイザーが一度にすべてのトークンを生成し、パーサーはそのトークン列を処理します。

```
public class SimpleJsonParser implements JsonParser {
    private List<Token> tokens;
    private int index;

    public ParseResult<JsonAst.JsonValue> parse(String input) {
        // 入力を一度に完全にトークン化
        var tokenizer = new SimpleJsonTokenizer(input);
        this.tokens = tokenizer.tokenizeAll();
        this.index = 0;
        var value = parseValue();
        return new ParseResult<>(value, "");
    }

    private Token current() {
        if (index < tokens.size()) {
            return tokens.get(index);
        }
        return new Token(Token.Type.EOF, null);
    }

    private boolean moveNext() {
        if (index < tokens.size() - 1) {
            index++;
            return true;
        }
        return false;
    }
```

```
    // トークン列から抽象構文木を構築
    private JsonAst.JsonValue parseValue() {
        // 中身は後で解説
    }

    // オブジェクトの解析（トークンベース）
    private JsonAst.JsonObject parseObject() {
        // 中身は後で解説
    }

    // 配列の解析（トークンベース）
    private JsonAst.JsonArray parseArray() {
        // 中身は後で解説
    }
    // 他のメソッドも同様に実装
}
```

このアプローチの重要な点は、**字句解析と構文解析の責任が明確に分離されている**ことです。`tokenizeAll()` メソッドが最初に入力全体をトークン列に変換し、パーサーはそのトークン列を処理します。これにより：

1. 関心の分離: 字句解析器はトークンの認識に、構文解析器は構造の構築に専念できます
2. デバッグの容易さ: トークン列を事前に確認できるため、問題の切り分けが簡単です
3. 実装の簡潔さ: 各コンポーネントが単一の責任を持つため、コードがシンプルになります

構文解析器の `parseXXX()` メソッドを見ると、文字列の代わりにトークン列を処理していることがわかります。また、この構文解析器には空白の読み飛ばしに関する処理が入っていません。これは、字句解析器が空白を処理済みだからです。

PEG 版と異なり、途中で失敗したら後戻り（バックトラック）するという処理も存在しません。トークン列が事前に確定しているため、より決定的な解析が可能になります。

以下では構文解析のための各メソッドの詳細を説明します。残りのコードは、巻末の付録に掲載しています。

3.5.3 parseValue

`parseValue()` メソッドは、JSON の値を解析するためのメソッドです。これは、BNF で定義された `value = true | false | null | number | string | object | array` に対応しています。実装は以下のようになります。

```
    private Ast.JsonValue parseValue() {
        var token = current();
        switch(token.type) {
            case TRUE:
                return parseTrue();
            case FALSE:
                return parseFalse();
            case NULL:
                return parseNull();
            case INTEGER:
                return parseNumber();
            case STRING:
                return parseString();
            case LBRACE:
                return parseObject();
            case LBRACKET:
                return parseArray();
        }
        throw new RuntimeException("cannot reach here");
    }
```

current() メソッドは、現在のトークンを取得するためのメソッドで、トークン列から現在の位置のトークンを返します。switch 文では、先頭のトークンの種類に応じて適切な解析メソッドを呼び出しています。バックトラックが必要ないため、各トークンの種類に応じて直接対応するメソッドを呼び出す形になっています。

3.5.4 parseObject

parseObject() メソッドは、規則 object に対応するメソッドで、JSON のオブジェクトリテラルに対応するものを解析するメソッドでもあります。実装を示すと以下のようになります：

```
    private JsonAst.JsonObject parseObject() {
        if(current().type != Token.Type.LBRACE) {
            throw new ParseException(
                "expected `{`, actual: " + current().value
            );
        }
```

```
        moveNext();
        if(current().type == Token.Type.RBRACE) {
            return new JsonAst.JsonObject(new ArrayList<>());
        }

        List<Pair<JsonAst.JsonString, JsonAst.JsonValue>> members =
            new ArrayList<>();
        var pair= parsePair();
        members.add(pair);

        while(moveNext()) {
            if(current().type == Token.Type.RBRACE) {
                return new JsonAst.JsonObject(members);
            }
            if(current().type != Token.Type.COMMA) {
                throw new ParseException(
                    "expected: `,`, actual: " + current().value
                );
            }
            moveNext();
            pair = parsePair();
            members.add(pair);
        }

        throw new ParseException("unexpected EOF");
    }
```

まず、最初の if 文で、次のトークンが{であることを確認した後に、

- その次のトークンが}であった場合：空オブジェクトを返す
- それ以外の場合：`parsePair()` を呼び出し、`string:value` のようなペアを解析した後、以下のループに突入：
  - 次のトークンが}の場合、集めたペアのリストを引数として、`JsonAst.JsonObject()` オブジェクトを作って返す
  - それ以外で、次のトークンが`,`でない場合、構文エラーを投げて終了
  - それ以外の場合：次のトークンをフェッチして来て、`parsePair()` を呼び出して、ペアを解析した後、リストにペアを追加

のような動作を行います。実際の JSON のオブジェクトと対応付けてみると、より理解が進むでしょう。

3.5.5 parseArray

`parseArray()` メソッドは、規則 `array` に対応するメソッドで、JSON の配列リテラルに対応するものを解析するメソッドでもあります。実装を示すと以下のようになります：

```
    private JsonAst.JsonArray parseArray() {
        if(current().type != Token.Type.LBRACKET) {
            throw new ParseException(
                "expected: `[`, actual: " + current().value
            );
        }

        moveNext();
        if(current().type == Token.Type.RBRACKET) {
            return new JsonAst.JsonArray(new ArrayList<>());
        }

        List<JsonAst.JsonValue> values = new ArrayList<>();
        var value = parseValue();
        values.add(value);

        while(moveNext()) {
            if(current().type == Token.Type.RBRACKET) {
                return new JsonAst.JsonArray(values);
            }
            if(current().type != Token.Type.COMMA) {
                throw new ParseException(
                    "expected: `,`, actual: " + current().value
                );
            }
            moveNext();
            value = parseValue();
            values.add(value);
        }

        throw new ParseException("unexpected EOF");
    }
```

まず、最初の if 文で、次のトークンが [であることを確認した後に、

- その次のトークンが] であった場合：空の配列 (JsonAst.JsonArray) を返す
- それ以外の場合：parseValue() を呼び出し、value を解析した後、以下のループに突入：
  - 次のトークンが] の場合、集めた values のリストを引数として、JsonAst.JsonArray() オブジェクトを作って返す
  - それ以外で、次のトークンが, でない場合、構文エラーを投げて終了
  - それ以外の場合：次のトークンをフェッチして来て、parseValue() を呼び出して、value を解析した後、リストに value を追加

のような動作を行います。実際の JSON の配列と対応付けてみると、より理解が進むでしょう。

parseArray() のコードを読めばわかるように、ほとんどのコードは、parseObject() と共通のものになっています。もしこれが気になるようであれば、共通部分をくくりだすことも出来ます。他の割愛したメソッドも同様に、トークンを読み取って期待される構文を解析するという流れになっています。

3.5.6 字句解析器と構文解析器の連携

字句解析器を使った構文解析の流れを、{"key": "value"} を例に説明します：

1. **字句解析フェーズ**: tokenizeAll() が入力全体を一度にトークン列に変換

入力: {"key": "value"}
↓
トークン列: [LBRACE, STRING("key"), COLON, STRING("value"), RBRACE, EOF]

2. **構文解析フェーズ**: パーサーがトークン列を走査しながら抽象構文木を構築

- index == 0: LBRACE を見て parseObject() を呼び出し
- index == 1: STRING("key") を読み取り
- index == 2: COLON を確認
- index == 3: STRING("value") を parseValue() で処理
- index == 4: RBRACE で終了を確認
- 最終的にオブジェクトの抽象構文木を生成

この方式ではトークン列が事前に確定しているため、以下の利点があります：

- パーサーは純粋に構造の解析に集中できる
- トークン列をログ出力してデバッグが容易

3.6 PEG ベースの構文解析器と伝統的な構文解析器の違い

前節の PEG ベースの構文解析器と、この節の字句解析器を使った構文解析器の主な違いは以下の通りです：

1. 関心の分離: 字句解析と構文解析が明確に分離されている
2. 空白の処理: 字句解析器が空白を処理するため、構文解析器は空白を意識しない
3. バックトラック: 字句解析器を使う方式では通常バックトラックを行わない
4. 性能: 一般的に字句解析器を使う方式の方が高速

PEG ベースの構文解析器にはいいとこなしのように見えますが、PEG ベースの構文解析器は、以下のような利点もあります：

1. シンプルな実装: PEG は直感的で、構文解析のロジックが大幅に簡潔に表現できる
2. 再帰的な構造の自然な表現: 再帰的な文法を自然に扱えるため、特にネストされた構造の解析が容易

PEG ベースの構文解析器は、トークン列を生成する必要がないため、文字列補間のような、一見トークン化が難しいケースでも、構文解析器の中で直接文字列を解析できるという利点があります。

3.7 まとめ

この章では、JSON の構文解析や字句解析を実際に作ってみることを通して、構文解析の基礎について学んでもらいました。特に、

- JSON の概要
- JSON の BNF
- JSON の構文解析器（PEG 版）
- 古典的な構文解析器
- JSON の字句解析器
- JSON の構文解析器

といった順番で、JSON の定義から入って、PEG による JSON パーサー、字句解析器を使った構文解析器の作り方について学んでもらいました。この書籍中で使った JSON は ECMA-404 で定義されている正式な JSON のサブセットになっています。たとえば、浮動小数点数が完全に扱えないという制限がありますが、構文解析器全体から見ればささいなことなので、この章を理解出来れば、JSON の構文解析について理解できたと思って構いません。

次の章では、文脈自由文法（Context-Free Grammar, CFG）の考え方について学んでもらいます。文脈自由文法は、現在使われているほとんどの構文解析アルゴリズムの基盤となっている概念であって、CFG の理解なくしては、その後の構文解析の理解もおぼつかないからです。

逆に、CFG の考え方さえわかってしまえば、個別の構文解析アルゴリズム自体は、それほど難しいとは感じられなくなって来るかもしれません。

演習問題

1. コメントのサポート:*

- JSON の BNF 定義を拡張し、// から行末までの単一行コメントと、/* から */ までの複数行コメントをサポートするようにしてください。
  - PegJsonParser と SimpleJsonTokenizer の両方を修正し、これらのコメントを正しく無視するように実装してください。
  - ヒント: PegJsonParser では skipWhitespace にコメントスキップのロジックを追加するか、各解析メソッドの適切な箇所でコメントを読み飛ばす処理を挟みます。SimpleJsonTokenizer では moveNext の switch 文にケースを追加し、そこからコメントの種別を判定して読み飛ばす処理を実装します。

2. 数値型の拡張:

- PegJsonParser の parseNumber メソッドと、SimpleJsonTokenizer の tokenizeNumber メソッドを修正し、ECMA-404 仕様に準拠した数値型（小数部、指数部 e または E を含む）を正しく解析できるようにしてください。
  - JsonAst.JsonNumber の value フィールドの型を double から java.math.BigDecimal に変更し、精度が失われないように対応してください。
  - テストケースとして、123, -0.5, 1.2e3, 0.4E-1 のような多様な数値表現を試してみてください

第 4 章

第 4 章文脈自由文法の世界

第 3 章では、JSON の構文解析器を記述することを通して、構文解析のやり方を学びました。構文解析器についても、PEG 型の構文解析器および字句解析器を使った 2 通りを作ってみることで、構文解析器といっても色々な書き方があるのがわかってもらえたのではないかと思います。

この第 4 章では、現代の構文解析を語る上で必須である、文脈自由文法という概念について学ぶことにします。「文脈自由文法」というと、一見、堅くて難しそうな印象を持つ方も多いかもしれません。

しかし、実は皆さんは既に文脈自由文法を使っているのです。Java の `if` 文やメソッド定義、JSON の入れ子構造など、プログラミングで日常的に扱っている「構造」はすべて文脈自由文法で表現されています。この章では、そんな身近な例から始めて、徐々に文脈自由文法の概念を理解していきましょう。

4.1　身近な例から始める文脈自由文法

まず、皆さんが普段書いている Java コードを見てみましょう。以下はシンプルな if 文です。

```
if (x > 0) {
    System.out.println("正の数です");
}
```

この if 文の構造を言葉で説明すると「`if` の後に条件式を括弧で囲み、その後に文のブロックが来る」となります。さらに、if 文の中に if 文を書くこともできます。

```
if (x > 0) {
    if (x > 100) {
        System.out.println("100 より大きい");
    }
}
```

この「入れ子にできる構造」こそが、文脈自由文法の本質なのです。

この入れ子構造は、プログラミング以外の場面でも頻繁に現れます。

- **HTML の要素**：

```
<div>
    <p>段落の中に<strong>強調</strong>があり、
       さらに<a href="...">リンク</a>も含まれる</p>
</div>
```

- **数式の括弧**：

```
((2 + 3) × (4 − 1)) ÷ 5
```

- **日本語の引用**：

彼は「『明日は晴れる』と言った」と述べた

- **JSON のオブジェクト**：

```
{
  "user": {
    "profile": {
      "name": "太郎"
    }
  }
}
```

これらの例に共通するのは、ある構造の中に同じような構造を含むことができ、その深さに制限がないという点です。このような構造を正確に記述し、解析するために文脈自由文法が必要になるのです。この問題は、次節で述べる「括弧の対応」という根本的な問題に帰着します。

4.2　括弧の対応という根本問題

形式言語で最も基本的で重要な構造の一つが「括弧の対応」です。大抵のプログラミング言語（JSON や YAML のようなものも含む）は開き括弧と閉じ括弧に対応するルールを持っています。たとえ括弧そのものでなくても、`begin/end` や`{}`などが任意の深さだけ入れ子にできる場合は同じことです。

かなり原始的なプログラミング言語についてすら、数式を入力できる機能があり、かつ数式の中で括弧を使うことができる以上、プログラミング言語の構文解析において括弧の対応は避けて通れない問題です。

さて、ここで括弧の対応とは何かを考えてみましょう。括弧の対応が取れているとは、開き括弧と閉じ括弧が正しくペアになっていることを意味します。たとえば、以下は正しい括弧の対応です：

```
()          // 1 組の括弧
```

```
(())        // 入れ子になった括弧
(()())      // 入れ子と並列の組み合わせ
()()()      // 並列に並んだ括弧
```

一方、以下は正しくない例です：

```
)(          // 順序が逆
(()         // 閉じ括弧が不足
())         // 開き括弧が不足
```

このような「括弧の釣り合いが取れた文字列の並び」を表すものを Dyck 言語（「ディック言語」と読みます）と呼びます。「Dyck」は数学者の名前に由来しています。この、一見単純に見える問題が、構文解析において極めて重要な位置を占めているのです。

4.3　BNF で括弧の構造を表現する

この括弧の対応をどのように文法として表現すればよいでしょうか？ まずは既に皆さんに学んでもらった BNF を使って考えてみましょう。

括弧の構造には以下の 2 つのパターンがあります：

1. 空文字列（括弧なし）
2. (+ 内側の括弧構造 +) + 続きの括弧構造

これを BNF で書くと以下のようになります：

```
P = "(" P ")" P | "";
```

P は「括弧のパターン」を表します。この定義は再帰的になっていることに注目してください。P の定義の中に P 自身が現れています。これこそが、入れ子構造を表現する鍵なのです。

この BNF は上記の 2 つのパターンと直接対応しています：- ""：空文字列（パターン 1）- "(" P ")" P：開き括弧、内側の構造、閉じ括弧、続きの構造（パターン 2）

4.4　BNF から文脈自由文法へ

前節の BNF は既に文脈自由文法の一種です。ただし、文脈自由文法の議論をする際には、より標準的な記法を使います。段階的に変換してみましょう。

- ステップ 1: 選択を分離

まず、|で区切られた選択肢を別々の規則に分けます：

```
P = "(" P ")" P;
P = "";
```

同じ P が 2 つの規則で定義されていますが、これは「P は 2 つのパターンのどちらかになる」という意味です。

- ステップ 2: 記号の変更

次に、記法を数学的な標準形に近づけます：

- = を →（生成規則を表す矢印）に変更
- "" を ε （イプシロン：空文字列を表す記号）に変更
- 文字を囲む引用符を削除

```
P → ( P ) P
P → ε
```

これで文脈自由文法の標準的な記法になりました！

4.4.1 用語の整理

ここで重要な用語を整理しておきましょう：

- 生成規則：P → (P) P のような矢印で結ばれた規則
- 非終端記号：P のような、さらに展開される記号（変数のようなもの）
- 終端記号：(や) のような、これ以上展開されない記号（実際の文字）
- 開始記号：文法の起点となる非終端記号（この例では P）

つまり、文脈自由文法とは以下のようにして表現できます：

- 生成規則の集まり
  - 各生成規則は「非終端記号 → 記号の並び」の形
  - 記号の並びは終端記号と非終端記号の組み合わせ（空文字列ε も可）

これだけのシンプルなルールで、プログラミング言語の複雑な構文を表現できるのです。

4.5 実例で理解する「言語」の概念

前の節で定義した Dyck 言語の文法をもう一度見てみましょう。

```
D → P
P → ( P ) P
P → ε
```

この文法は括弧の対応をあらわすもの、という風にぼかして来ました。実のところ、これまで出てきた文法はすべて「形式言語」を定義するものです。でも、ちょっと待ってください。「形式言語」って何でしょうか？

私たちが普段「プログラミング言語」と呼んでいる Java や Python と、この 3 行の規則で表される「括弧の言語」は、どう関係しているのでしょうか？ 実は、どちらも同じ「形式言語」という概念で理解できるのです。

4.5.1 「形式言語」とは

私たちが普段使う「言語」という言葉は、自然言語（日本語や英語など）を指すことが多いですが、理論の世界では「形式言語」という概念が重要です。形式言語とは、**文字列の集合**として定義される言語のことです。

ここで言う集合とは数学の集合のことです。形式言語の世界では文字列をひたすら集めていった、その集合が「言語」と呼ばれるのです。

さて、この観点から「Java 言語」や「JavaScript 言語」という時、それは何を指しているのでしょうか？

4.5.2 言語を「文字列の集合」として考える

形式言語の立場からは、プログラミング言語を**その言語で書ける正しいプログラムすべての集合**として定義することができます。「正しいプログラム」といっても、構文解析ができるもの、型検査を通るもの、実行時にエラーが出ないものなど、いくつかの基準がありますが、ここでは「構文解析が通るプログラム」と考えます。

具体例で考えてみましょう。以下はすべて正しい JavaScript プログラムです：

```
console.log("Hello, World!");
```

```
console.log(3);
```

```
console.log(3 + 5);
```

これらを集めていくと、JavaScript という言語（以後 JS と表記）は次のような文字列の集合として表現できます：

```
JS = {
  "console.log(\"Hello, World!\");",
  "console.log(3);",
  "console.log(3 + 5);",
  "const x = 10;",
  "function add(a, b) { return a + b; }",
  ...（無限に続く）
}
```

JS で書ける正しいプログラムは無限にあるので、この集合は**無限集合**（正確には可算無限個の集合）になります。

4.5.3 Dyck 言語も集合として理解する

同様に、Dyck 言語（括弧の対応が取れた文字列）も集合として表現できます：

```
Dyck = {
  "()",
  "(())",
  "((()))",
  "(()())",
  "()()",
  "()()()",
  ...（無限に続く）
}
```

Dyck 言語も無限集合です。基本的に私達が扱いたい言語＝集合は、ほとんどが無限集合です。なぜなら、基本的にはその言語のテキストがいくら長くなってもいいようにしたいからです。言語であらかじめ「長さの制限」を設けることは、実用的ではありません。

4.5.4 集合として言語を扱うメリット

言語を集合として扱うと、数学の集合論で使う記号が使えるようになります。これには実用的なメリットがあります。

まず、基本的な記号の意味を説明します：

- ∈：「含まれる」を意味します
- ∉：「含まれない」を意味します
- ⊂：「部分集合」を意味します（左の集合が右の集合に完全に含まれる）
- ∩：「共通部分」を意味します（両方の集合に含まれる要素）
- 例 1：ある文字列が言語に含まれるかの判定

```
"()" ∈ Dyck          // "()"は Dyck 言語に含まれる
")(" ∉ Dyck          // ")("は Dyck 言語に含まれない
```

- 例 2：言語の後方互換性

Java 8 が Java 5 の後方互換であることを、集合の包含関係で表現できます：

```
Java5 ⊂ Java8       // Java 5 で書けるプログラムはすべて Java 8 でも書ける
```

- 例 3：言語の共通部分

たとえば、「JS でも TS でも有効なプログラム」は以下のように表現できます：

```
JS ∩ TS = {
  "console.log(123);",
  "const x = 5;",
```

```
  "function f() { return 1; }",
  ...
}
```

このように、言語を集合として扱うことで、言語間の関係を明確に表現できるようになります。これは単なる理論ではなく、言語設計や互換性の議論において非常に役立ちます。

4.6 文脈自由文法の数学的定義

ここまでふわっとした説明をしてきましたが、文脈自由文法の定義はもう少し厳密に行うことができます。本書では厳密な数学的扱いよりも実用的な理解を重視するため、あくまで厳密にはこう定義されるという話であって、この定義を直接使うことはありません。興味のない方は読み飛ばしても構いません。

文脈自由文法（Context-Free Grammar, CFG）は、形式的には 4 つ組 $G = (V, \Sigma, P, S)$ として定義されます：

- V：非終端記号（Non-terminal symbols）の有限集合
- Σ：終端記号（Terminal symbols）の有限集合（$V \cap \Sigma = \emptyset$）
- P：生成規則（Production rules）の有限集合
  - 各規則は $A \to \alpha$ の形（$A \in V$、$\alpha \in (V \cup \Sigma)^*$）
- S：開始記号（Start symbol）（$S \in V$）

例えば、Dyck 言語の文法は数学的には以下のように表せます：

- $V = \{P\}$
- $\Sigma = \{(,)\}$
- $P = \{P \to (P)P, P \to \varepsilon\}$
- $S = P$

ここで ε は空文字列を表します。

文法 G が生成する言語 $L(G)$ は、開始記号 S から生成規則を繰り返し適用して導出できるすべての終端記号列の集合として定義されます：

$$L(G) = \{w \in \Sigma^* \mid S \Rightarrow^* w\}$$

ここで、$\Rightarrow^*$ は「0 回以上の導出ステップ」を表します。

もう少し噛み砕いて説明すると：

1. 開始記号 S から出発する
2. 生成規則を適用して、非終端記号を置き換えていく
3. すべてが終端記号になったら、それが言語 $L(G)$ の要素の一つ

例えば、Dyck 言語の文法では以下のようにになります。

- $P \Rightarrow (P)P \Rightarrow ()P \Rightarrow ()$ という導出により、() が $L(G)$ に含まれる
- $P \Rightarrow (P)P \Rightarrow ((P)P)P \Rightarrow (()P)P \Rightarrow (())P \Rightarrow (())$ により、(()) も $L(G)$ に含まれる
- このようにして、すべての「正しく対応した括弧列」が $L(G)$ の要素となる

この数学的定義を「きっちりと」理解することは、理論的には重要ですが、構文解析器の実装では、直感的な理解で問題ありません。BNF などの記法は数学的定義と本質的に同じものを、より読みやすい形で表現しています。

4.7　正規表現の限界と文脈自由言語

ここまでで文脈自由文法の基本的な概念と言語を集合として扱う考え方を学びました。BNF と文脈自由文法の表現能力が実質的に等価であることはなんとなく理解できたかと思います。しかし、ここで一つ重大な疑問が浮かび上がります。それは、私たちにとって馴染み深い正規表現との関係です。正規表現も文脈自由文法と同じように言語＝文字列の集合を表現するためのツールですが、正規表現と文脈自由文法にはどのような違いがあるのでしょうか？

4.7.1　身近な正規表現から考える

皆さんは日常的に正規表現を使っているでしょう。ファイル検索、テキスト処理、入力値の検証など、様々な場面で活躍しています。

たとえば、Java で電話番号の形式をチェックする時、以下のような正規表現を使うことがあります：

```
String phonePat = "\\d{3}-\\d{4}-\\d{4}";  // 例：090-1234-5678
if (phone.matches(phonePat)) {
    // 有効な電話番号
}
```

あるいは、郵便番号の簡易チェックでは以下のような正規表現を使います：

```
String postalCodePat = "\\d{3}-\\d{4}";  // 例：123-4567
if (postalCode.matches(postalCodePat)) {
    // 有効な郵便番号
}
```

正規表現は、特定のパターンにマッチする文字列を簡潔に表現できる強力なツールです。また、正規表現も無限集合を扱うことができます。上記の正規表現はあくまで有限個のパターンしか表せませんが、以下のパターンでは自然数を無限に表現できます：

```
String natPat = "0|[1-9][0-9]*";  // 0以上の自然数
if (number.matches(natPat)) {
    // 有効な自然数
```

```
}
```

このとき、natPat は以下のような無限集合を表します：

{'0', '1', '2', '3', ...}

注意して欲しいのは、これはあくまで文字列としての自然数の集合であり、数値としての自然数ではないということです。正規表現は文字列のパターンを表現するためのツールであり、数値そのものを扱うわけではありません。

4.7.2　正規表現でできないこと

しかし、正規表現には決定的な限界があります。それは**括弧の対応が取れているかチェックできない**ということです。

試しに、既にでてきた Dyck 言語をを正規表現で判定することを考えてみましょう：

OK:　(), (()), (()(()))
NG:　)(, (()), ())

どんなに工夫しても、任意の深さの括弧の対応を正規表現で表現することはできません。なぜでしょうか？ 次からは正規表現の基本的な仕組みとその限界について詳しく見ていきましょう。

4.7.3　正規表現の構成要素を詳しく見る

正規表現は、実は非常にシンプルな 3 つの基本演算から構成されています。現代の正規表現エンジンは多くの便利な記法を提供していますが、理論的にはすべて以下の 1-3 の基本演算に帰着できます（ただし、後方参照など「非正規表現」とでも呼ぶべき機能を除く）。

4.7.3.1　1. 連接（Concatenation）

2 つの正規表現を続けて書くことを**連接**と呼びます。

正規表現: ab
マッチする文字列: "ab"
マッチしない文字列: "a", "b", "ba", "abc"

連接は最も基本的な演算で、「次に」という順序関係を表現します。

4.7.3.2　2. 選択（Alternation）

|記号を使って、複数の選択肢を表現します。

正規表現: a|b
マッチする文字列: "a", "b"
マッチしない文字列: "ab", "c", ""

より複雑な例：

正規表現：（ab|cd）
マッチする文字列："ab"，"cd"
マッチしない文字列："ac"，"bd"，"abcd"

4.7.3.3　3. 繰り返し（Kleene Star）

*記号は、直前の要素を 0 回以上繰り返すことを意味します。この演算は数学者 Stephen Cole Kleene（ステファン・クリーネ）にちなんで「Kleene Star」と呼ばれます。

正規表現：a*
マッチする文字列：""，"a"，"aa"，"aaa"，...
マッチしない文字列："b"，"ab"

4.7.3.4　派生的な演算子

現代の正規表現では、便利のために多くの派生的な演算子が追加されています：

a+	1 回以上の繰り返し（aa*と同等）
a?	0 回または 1 回（a|ε と同等）
a{n}	ちょうど n 回の繰り返し
a{n,m}	n 回以上 m 回以下の繰り返し
[a-z]	文字クラス（a|b|c|...|z と同等）
.	任意の 1 文字

しかし、これらはすべて基本の 3 演算で表現できます。たとえば：

- a+ は aa*
- a? は a|ε （ε は空文字列）
- [abc] は a|b|c

4.7.4　正規表現の裏側にある仕組み

皆さんは正規表現を使っているとき、その裏側でどのような処理が行われているか考えたことはありますか？ 実は、正規表現の限界を理解するには、その背後にある「オートマトン」という仕組みを知ることが役立ちます。「オートマトン」という名前は難しそうですが、要は「文字列を一文字ずつ読みながら状態を変えていく機械」のことです。

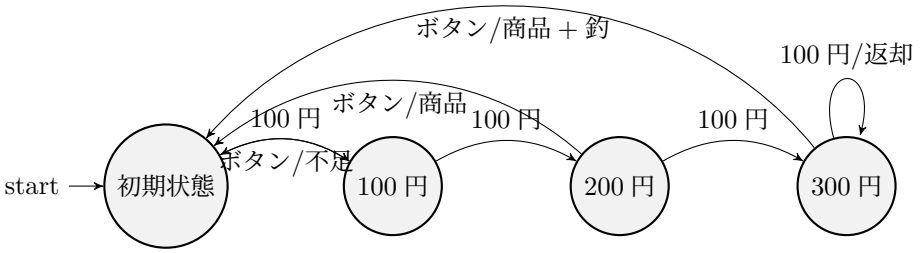
4.7.4.1　オートマトンを自動販売機で考える

オートマトンは「有限個の状態を持つ機械」です。難しく聞こえるかもしれませんが、自動販売機を例にするとわかりやすくなります。ここで、全ての商品は 100 円とし、投入可能な上限金額は 300 円とします：

- 初期状態：お金が入っていない

  - 100 円投入 → 100 円状態へ遷移
  - 商品ボタンを押す → 初期状態のままで、お金が不足している旨を表示
- 100 円状態
  - 100 円投入 → 200 円状態へ遷移
  - 商品ボタンを押す → 商品を出して初期状態へ遷移
- 200 円状態
  - 100 円投入 → 300 円状態へ遷移
  - 商品ボタンを押す → 商品と、お釣りをだして初期状態へ遷移
- 300 円状態
  - 100 円投入 → 300 円状態のまま、100 円を返却
  - 商品ボタンを押す → 商品と、お釣りをだして初期状態へ遷移

図にすると以下のようになります：



この図の見方を説明します：

- **ノード（状態）**：
  - 円で表現される各ノードが「状態」を表します
  -「start」と書かれた矢印が指す「初期状態」から開始します
  - 各状態には現在の投入金額が記載されています
- **矢印（遷移）**：
  - 矢印は状態間の「遷移」を表します
  - 矢印上のラベルは「入力/出力」の形式です
    - 「100 円」：100 円硬貨を投入
    - 「ボタン/商品」：商品ボタンを押すと商品が出る
    - 「ボタン/不足」：商品ボタンを押すがお金が不足
    - 「ボタン/商品 + 釣」：商品ボタンを押すと商品とお釣りが出る
    - 「100 円/返却」：100 円を投入するが満杯なので返却される
- **動作の例**：
  1. 初期状態から 100 円を 2 回投入 → 200 円状態へ
  2. 200 円状態で商品ボタンを押す → 商品が出て初期状態へ戻る

この図は、自動販売機の「状態遷移」を完全に表現しています。どの状態からどの入力を受けたらどの状態に遷移するかが一目でわかります。

このように、オートマトンは「状態」と「入力」によって遷移を決定します。自動販売機では、状態はお金の投入状況で、入力はお金の投入や商品ボタンの押下です。

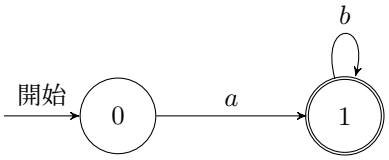
4.7.4.2 簡単な例：ab*を認識するオートマトン

このオートマトン（有限状態機械）を使うと正規表現で表現される言語を認識することができます（正規表現で認識可能な言語は全てオートマトンで表現可能と言い換えててもいいです)。ここでは、正規表現の例として `ab*` を認識するオートマトンを考えてみましょう。

この正規表現は以下のような文字列にマッチします：

- `a`（b が 0 個）
- `ab`（b が 1 個）
- `abb`（b が 2 個）
- `abbb`（b が 3 個）
- …

この正規表現を認識するオートマトンは以下のようになります：



- 状態 0（開始状態）：最初の状態
- 状態 1（受理状態）：この状態で入力が終わればマッチ成功
- 矢印：文字を読んだときの状態遷移

このオートマトンは以下のように動作します：

- 入力 “a”：状態 0 →状態 1（**受理**）
- 入力 “ab”：状態 0 →状態 1 →状態 1（**受理**）
- 入力 “abb”：状態 0 →状態 1 →状態 1 →状態 1（**受理**）
- 入力 “b”：状態 0 で’b’ を読めない（**非受理**）

4.7.4.3 2 種類のオートマトン

オートマトンには大きく分けて 2 種類あります。難しい名前ですが、それぞれの特徴を簡単に説明します：

- **NFA（Non-deterministic Finite Automaton、非決定性有限オートマトン）**
  - 「非決定性」とは、同じ文字を読んだときに複数の選択肢がある
  - 文字を読まずに状態を移動できる（イプシロン遷移）
  - 正規表現から比較的簡単に作れる
- **DFA（Deterministic Finite Automaton、決定性有限オートマトン）**

– 「決定性」とは、各状態で各文字に対して次の状態が一意に決まる
– 文字を読まずに移動することはできない
– 実装が簡単で、実行速度が速い

この 2 つのオートマトンは正規表現（正確には後で説明する正規言語）を認識するモデルであり、能力的には等価です。どちらも正規表現を認識できますが、実装の複雑さや速度に違いがあります。

ちなみに、これまで見てきたオートマトンは全て状態遷移が一意に決まる DFA です。DFA は状態遷移が一意に決まるため、実装が非常にシンプルで高速です。

一方で、NFA はそのままでは実装が複雑で遅くなりますが、正規表現からの変換が容易であるという利点があります。

4.7.5 なぜ正規表現では括弧の対応が扱えないのか

ここまで、正規表現がオートマトンという仕組みで動作することを学びました。では、なぜオートマトン（正規表現）では括弧の対応がチェックできないのでしょうか？ その理由を理解するために、簡単な例から考えてみましょう。

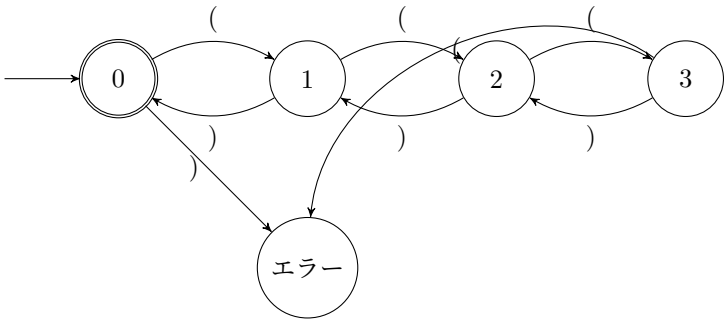
括弧の対応を確認するには、以下のような処理が必要です：- (を読んだら、「今までに読んだ開き括弧の数」を 1 増やす -) を読んだら、「対応する開き括弧があるか」確認して、1 減らす - 最後に、カウントが 0 であることを確認する

ここで問題になるのは、括弧は何重にでもネスト（入れ子）できるということです。たとえば以下のようなケースを扱える必要があります：- (()) : 2 重 - ((())) : 3 重
- (((()))) : 4 重 - …無限に続く

4.7.5.1 オートマトンの限界：有限の状態しか持てない

オートマトンは「有限個の状態」しか持てません。仮に 10 個の状態を持つオートマトンを作ったとしても、11 重にネストした括弧は扱えません。どんなに状態数を増やしても、それを超える深さのネストが存在します。

これを具体的に理解するために、簡単なオートマトンで考えてみましょう。仮に「最大 3 つまでの開き括弧を数える」オートマトンを作ると：



このオートマトンは 3 つまでの開き括弧なら正しく数えられますが、4 つ目の開き括弧が来るとエラー状態に遷移してしまいます。状態数を 100 個に増やしても、101 個の開き括弧には対応できません。

4.7.6　言語の階層：正規言語 < 文脈自由言語 < …

ところで、実は、形式言語で表現できる言語の能力には階層があります。

- 正規言語（RL: Regular Language）：
  - 正規表現で表現できる**言語の集合**
  - 例：電話番号、メールアドレス、識別子
- 文脈自由言語（CFL: Context-Free Language）：
  - 文脈自由文法で表現できる**言語の集合**
  - 例：プログラミング言語の構文、JSON、XML
  - 正規言語をすべて含む（RL ⊂ CFL）

ここで重要なのは、正規言語や文脈自由言語といったとき、実際には言語ではなく言語の集合について語っていることです。このような言語の集合のことを言語クラスと呼びます。

文脈自由文法は正規表現の上位互換で、正規表現で表現できることはすべて文脈自由文法でも表現でき、加えて括弧の対応のような複雑な構造も扱えます。

4.7.7　実用上の意味

この違いは実用上極めて重要です。以下のような違いがあるからです：

- 正規表現で十分な例：
  - URL の検証
  - 電話番号のフォーマットチェック
  - 単純なトークンの切り出し
- 文脈自由文法が必要な例：
  - プログラミング言語の構文解析
  - JSON や YAML、XML のパース
  - 数式の評価（括弧を含む）

だからこそ、構文解析器を作る時には正規表現だけでなく、文脈自由文法の理解が必要になるのです。

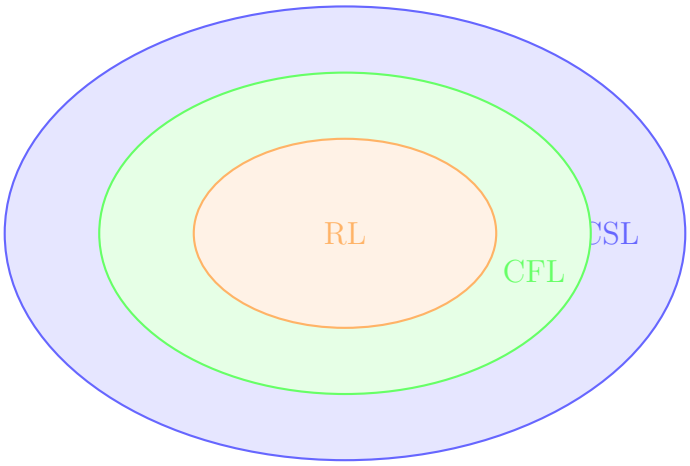
4.8　さらに上の階層

実は言語の階層はさらに続きます。ここでは簡単に紹介します：

- 文脈依存言語（CSL: Context-Sensitive Language）：
  - 例：`a^n b^n c^n`（a が n 個、b が n 個、c が n 個ずつが同じ数だけ並ぶ）
  - より複雑な制約を表現できます

- 帰納的可算言語（Recursively Enumerable Language）：
  - コンピュータで計算できるすべてのもの
  - Java や Python などのプログラミング言語の計算能力はこのレベル

階層を図にすると次のようになります：



プログラミング言語の**構文**は文脈自由文法で記述できますが、プログラミング言語自体の**計算能力**はチューリング完全（最上位）です。この違いを理解することが重要です。

4.9　文法から文字列を作る：導出の仕組み

文脈自由文法は「生成規則」の集まりでした。では、この「生成」とは何でしょうか？

Dyck 言語の文法をもう一度見てみましょう：

```
D → P
P → ( P ) P
P → ε
```

これらの規則は「置き換えルール」として読むことができます：

- `D → P`：「D を見たら P に置き換えてよい」
- `P → ( P ) P`：「P を見たら (P) P に置き換えてよい」
- `P → ε`：「P を見たら空文字列に置き換えてよい」

4.9.1　実際に文字列を生成してみる

() という文字列を生成する過程を追ってみましょう：

```
D                  // 開始記号から始める
→ P                // D → P を適用
→ ( P ) P          // P → ( P ) P を適用
→ ( ε ) P          // 最初の P に P → ε を適用
```

→ ( ) P　　// ε は空文字列なので消える

→ ( ) ε　　// 2 番目の P に P → ε を適用

→ ( )　　// ε は空文字列なので消える

このように、規則を順番に適用して文字列を作ることを**導出**と呼びます。生成すると言い換えても良いでしょう。

4.9.2 複数の導出方法

同じ文字列を導出する方法は複数あります。簡単な文法で考えてみましょう：

S → AB

A → a

B → b

この文法から ab を導出する時、どちらから先に展開するかで 2 通りの方法があります：

- 方法 1（左から展開）：

S → AB → aB → ab

- 方法 2（右から展開）：

S → AB → Ab → ab

4.9.3 最左導出と最右導出

このように、好きな順番で非終端記号を展開できるのですが、導出の過程を統一するために 2 つの標準的な方法が定義されています。

- **最左導出**：常に一番左の非終端記号を展開
- **最右導出**：常に一番右の非終端記号を展開

より複雑な例で見てみましょう。以下の文法は「1 個以上の a」の後に「1 個以上の b」が続く文字列を表します：

S → AB

A → aA | a

B → bB | b

aabb を導出する場合：

- **最左導出：**

S

→ AB　　(S → AB)

→ aAB　　(A → aA)

→ aaB　　(A → a)

→ aabB　　(B → bB)

→ aabb　　(B → b)

- **最右導出：**

S

→ AB　　(S → AB)

→ AbB　　(B → bB)

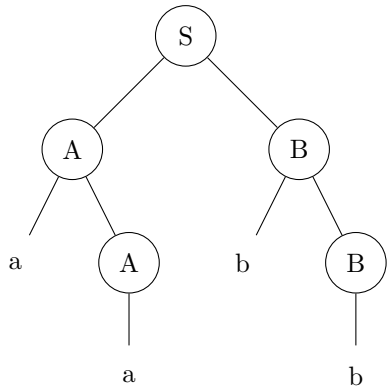
→ Abb　　(B → b)

→ aAbb　　(A → aA)

→ aabb　　(A → a)

4.9.4　解析木との関係

どちらの導出方法でも、最終的に同じ**解析木**が得られます：



最左導出は木を左から右へ構築し、最右導出は右から左へ構築するイメージです。ここで得られた解析木は具体的な文字列の情報を含んでおり、抽象構文木と対比する形で具象構文木と呼ばれることもあります。

4.9.5　なぜ 2 つの導出方法が重要か

実はこれらは構文解析の 2 大手法に対応しています：

- **最左導出 → 下向き構文解析**（トップダウン）
- **最右導出の逆 → 上向き構文解析**（ボトムアップ）

第 3 章で作った JSON パーサーは下向き構文解析の一種でした。次章では、これらの手法についてより詳しく学んでいきます。

4.10　まとめ

この章では、文脈自由文法について学びました。最初は難しく感じたかもしれませんが、実は私たちが普段書いているプログラムと深く関わっている概念です。この章で学んだことを整理します。

1. 文脈自由文法の基礎
    - Java の if 文や JSON の構造など、プログラミングの「入れ子構造」は文脈自由文法で表現される
    - BNF から文脈自由文法への変換は記法の違いに過ぎない
    - 生成規則、非終端記号、終端記号という基本要素で構成される
2. 言語を集合として理解する
    - プログラミング言語は「正しいプログラムの集合」として定義できる
    - 集合論の記法を使って言語間の関係を厳密に議論できる
    - 後方互換性なども集合の包含関係として表現可能
3. 正規表現とオートマトン
    - 正規表現は正規言語を表現するための強力なツール
    - NFA（非決定性有限オートマトン）と DFA（決定性有限オートマトン）の違い
    - 正規表現は有限の状態しか持てないため、括弧の対応などの複雑な構造を扱えない
4. 言語の階層
    - 正規表現（正規言語）< 文脈自由文法（文脈自由言語）< ... 帰納的可算言語
    - プログラミング言語の構文解析には文脈自由文法が必要
5. 導出の仕組み
    - 生成規則を適用して文字列を生成する過程が導出
    - 最左導出と最右導出という 2 つの標準的な方法がある
    - これらは下向き/上向き構文解析に対応する

文脈自由文法の理解は、以下の場面で役立ちます：

- **構文解析器の実装**：第 5 章以降で学ぶ様々な構文解析アルゴリズムの基礎
- **言語設計**：新しい DSL やプログラミング言語を設計する際の指針
- **エラーメッセージの理解**：コンパイラのエラーメッセージがなぜそう言っているかの理解
- **ツールの選択**：正規表現で十分か、パーサーが必要かの判断

次章ではこの文脈自由文法を基に、実際の構文解析アルゴリズムについて詳しく見ていきます。

第 5 章

第 5 章構文解析アルゴリズム古今東西

第 4 章で学んだ文脈自由文法は、構文解析の理論的な基礎です。特に Dyck 言語（ディック言語）を例に、括弧の対応という根本的な問題を通して、再帰的な構造を文法でどう表現するかを学びました。この章では、いよいよその文法を実際に解析するための「アルゴリズム」について深く掘り下げていきます。

5.1　本章で学ぶこと

「構文解析アルゴリズム」と聞くと難しそうに感じるかもしれません。しかし、実は皆さんはすでに 2 つの構文解析アルゴリズムを実装しているのです。

第 3 章を思い出してください。最初に実装した `PegJsonParser` は、**PEG（Parsing Expression Grammar）**という手法の素朴な実装でした。次に実装した `SimpleJsonParser` は、実は **LL(1)**（「エルエルワン」と読みます）と呼ばれる手法に近い**再帰下降構文解析器**だったのです。

この章では、これらのアルゴリズムがどのような仕組みで動いているのか、他にどのようなアルゴリズムが存在するのかを体系的に学びます。具体的には：

1. **予測型下向き構文解析（Predictive Top-down Parsing）**：文法の開始記号から始めて、入力文字列に向かって解析を進める方法
   - 先読みを使って適用する規則を予測する
   - LL(1)
2. **予測型上向き構文解析（Predictive Bottom-up Parsing）**：入力文字列から始めて、文法の開始記号に向かって解析を進める方法
   - シフトと還元を使って解析を進める
   - LR(0)、SLR(1)、LR(1)、LALR(1)
3. **下向き構文解析 + バックトラック**：先読みの代わりにバックトラックを使う方法
   - PEG（Parsing Expression Grammar）、Packrat Parsing

それぞれのアルゴリズムには得意・不得意があり、実際のプログラミング言語の構文解析器では、言語の特性に応じて最適なアルゴリズムが選ばれています。

5.2　構文解析器生成系との関係

各アルゴリズムには、対応する**構文解析器生成系**（パーサージェネレータ）が存在します（自作することもできます）：

- **Yacc/Bison**：
  - LALR(1) を採用
  - C 言語向け
- **JavaCC**：
  - LL(1) を採用
  - Java 向け
- **ANTLR**：
  - ALL(*) を採用（拡張された LL 系を採用）
  - 多言語対応
- **各種 PEG パーサジェネレータ**：PEG を採用

これらのツールについては第 6 章で詳しく説明しますが、本章でアルゴリズムを理解することで、各ツールがなぜそのアルゴリズムを選んだのかが見えてくるはずです。

5.3　下向き構文解析と上向き構文解析 - 2 つの世界観

構文解析アルゴリズムの世界には、大きく分けて 2 つの「世界観」があります。それが**下向き（Top-down）**と**上向き（Bottom-up）**です。この 2 つは、同じ構文解析という仕事を、まったく逆のアプローチで実現します。

5.3.1　なぜ 2 つのアプローチが必要なのか？

第 4 章で学んだ Dyck 言語を例に考えてみましょう。文法は以下でした：

```
D → P
P → ( P ) P
P → ε
```

入力文字列（()）を解析する際、2 つの考え方ができます：

1. **下向き（予測的）**：「D から始まって、どうすれば (()) が導出できるか？」
2. **上向き（還元的）**：「(()) から始めて、どうすれば D に辿り着けるか？」

これは、迷路を解くときに「入口から出口を探す」か「出口から入口を探す」かの違いに似ています。どちらも正解に辿り着きますが、効率や適用できる問題が異なるのです。

5.4　下向き構文解析の基本原理

下向き構文解析は、文法の開始記号から出発し、入力文字列に向かって「上から下へ」解析を進めます。

5.4.1　Dyck 言語で学ぶ予測型下向き構文解析

第 4 章で学んだ Dyck 言語（括弧の対応が取れた文字列を表す言語）を例に、具体的に動作を追ってみましょう。説明のために、文法に入力の開始・終了を表す$（ドル記号）を追加します：

```
D -> $ P $
P -> ( P ) P
P -> ε
```

入力文字列（()）に対して、この文法が受理するかどうかを判定してみましょう。

5.4.2　スタックを使った解析の追跡

予測型下向き構文解析では、**スタック**を使って解析の進行状況を管理します。スタックには、「今どの規則のどの位置を解析しているか」という情報を記録します。

ドット記号「・」（中黒）を使って、規則内の現在位置を表します。例えば：

- `D -> ・$ P $`：まだ何も解析していない状態
- `D -> $・P $`：$を解析済み、次は P を解析する
- `D -> $ P $・`：すべて解析完了

このドット記号は、Java でいえばデバッガーでステップ実行する時の「現在の実行位置」のようなものだと考えてください。

では、実際に解析を開始しましょう：

入力: `$ ( ( ) ) $`
スタック: `[ D -> ・$ P $ ]`

1. 最初の記号 $ を入力から読み込みます。スタックトップの規則が期待する記号と一致するので、ドットを進めます：

入力: `( ( ) ) $`
スタック: `[ D -> $・P $ ]`

2. 非終端記号 P を解析する必要があります。ここで「予測」（あるいは先読み）、が必要になります。

非終端記号とは、第 2 章で学んだように「まだ展開の余地がある記号」のことでした。Java でいえばメソッド呼び出しのようなものです。

次の入力文字（先読み文字）は (です。P の規則は 2 つあります：

- P -> (P) P：(で始まる
- P -> ε ：空文字列（何もない）

ここで、どちらを選ぶかを事前に決める必要があります。先読み文字が (でかつ P は常に (で始まるので、P -> (P) P を選択します。この規則をスタックに追加します：

入力: (( )) $
スタック: [P -> ・(P) P, D -> $・P $]

3. 入力の先頭の (を読み込み、スタックトップの規則のドットを進めます：

入力: ( )) $
スタック: [P -> (・P) P, D -> $・P $]

4. 非終端記号 P を解析するため、再び先読み文字を見ます。

次の先読み文字は (です。先ほどと同様に P -> (P) P を選択して、この規則をスタックに追加します：

入力: ( )) $
スタック: [P -> ・(P) P, P -> (・P) P, D -> $・P $]

5. 入力の先頭の (を読み込み、スタックトップの規則のドットを進めます：

入力:)) $
スタック: [P -> (・P) P, P -> (・P) P, D -> $・P $]

6. 非終端記号 P を解析するため、先読み文字を見ます。

次の先読み文字は) です。今度は、P -> ε を選択します。空文字列にマッチするので、ドットを進めるだけで、入力からは何も消費しません：

入力:)) $
スタック: [P -> (P ・) P, P -> (・P) P, D -> $・P $]

7. 入力の先頭の) を読み込み、スタックトップの規則のドットを進めます：

入力:) $
スタック: [P -> (P) ・P, P -> (・P) P, D -> $・P $]

8. 非終端記号 P を解析するため、先読み文字を見ます。

次の先読み文字は) です。ここで、P -> ε を選択します。空文字列にマッチするので、ドットを進めるだけで、入力からは何も消費しません：

入力:) $
スタック: [P -> (P) P ・, P -> (・P) P, D -> $・P $]

9. スタックトップの規則のドットが最後まで進んだので、スタックからその規則を除去すると同時に、新たなスタックトップのドットを進めます：

入力: ） $
スタック: [P -> （ P ・） P, D -> $・P $]

10. 入力の先頭の ） を読み込み、スタックトップの規則のドットを進めます：

入力: $
スタック: [P -> （ P ） ・ P, D -> $・P $]

11. 非終端記号 P を解析するため、先読み文字を見ます。

次の先読み文字は $ です。ここで、P -> ε を選択します。空文字列にマッチするので、ドットを進めるだけで、入力からは何も消費しません：

入力: $
スタック: [P -> （ P ） P ・, D -> $ ・ P $]

12. スタックトップの規則のドットが最後まで進んだので、スタックからその規則を除去すると同時に、新たなスタックトップのドットを進めます：

入力: $
スタック: [D -> $ P ・ $]

13. 入力の先頭の $ を読み込み、スタックトップの規則のドットを進めます：

入力:
スタック: [D -> $ P $ ・]

14. ここでスタックトップの規則のドットが最後まで進んだので、スタックからその規則を除去します。スタックが空になり、入力もすべて消費されたので、受理されます。

入力:
スタック: []

このように、「先読み文字を見て、次に適用すべき規則を予測する」のが予測型下向き構文解析の特徴です。

5.4.3 予測型下向き構文解析のアルゴリズム

予測型下向き構文解析の動作パターンをまとめると以下のようにになります：

1. 先読み文字の取得：
   - 現在の入力位置から 1 文字先を見る
2. 非終端記号の展開：
   - スタックトップが非終端記号の場合、先読み文字に基づいて適用する規則を「予測」

- 選択した規則をスタックに追加
- 適切な規則がない場合はエラー

3. 終端記号のマッチング：
   - スタックトップが終端記号の場合、入力文字と比較
   - 一致すれば入力を消費してドットを進める
   - 一致しなければエラー
4. 規則の完了：
   - 規則の最後まで進んだら、スタックからその規則を除去
   - 一つ上の規則の解析を続行
5. 受理判定：
   - 入力がすべて消費され、スタックが空になれば受理
   - それ以外は拒否

5.5　予測型再帰下降構文解析 - 下向き構文解析の Java 実装

スタックを使った下向き構文解析の動作を理解したところで、これを Java で実装してみましょう。多くの場合、スタックを明示的に使わずに**再帰呼び出し**を使って実装できます。第 3 章で実装した `SimpleJsonParser` は、予測型再帰下降構文解析の一種で、実装もほぼ同じです。

```
// 文法規則：
// D -> P
// P -> ( P ) P
// P -> ε （イプシロン：空文字列）
public class Dyck {
    private final String input;
    private int pos;

    public Dyck(String input) {
        this.input = input;
        this.pos = 0;
    }

    public boolean parse() {
        boolean result = D();
        return result && pos == input.length();
    }
```

```
    private boolean D() {
        return P();
    }

    private boolean P() {
        // 先読み文字が '(' の場合
        if (pos < input.length() && input.charAt(pos) == '(') {
            // P -> ( P ) P
            pos++; // '(' を読み進める
            if (!P()) return false;
            if (pos < input.length() && input.charAt(pos) == ')') {
                pos++; // ')' を読み進める
                return P();
            } else {
                return false;
            }
        } else {
            // P -> ε
            return true;
        }
    }
}
```

5.5.1 コードの解説

この実装では、各非終端記号に対応するメソッドを作成しています：

- D() メソッド：文法の開始記号 D に対応
  - 規則 D -> P をそのまま実装：P() を呼び出すだけ
- P()：非終端記号 P に対応
  - 最初に P -> (P) P を試す：先読み文字が (か確認
  - 適用できない場合は P -> ε を適用：常に true を返す
- 文字の消費：pos をインクリメントすることで実現
- バックトラックのない予測型：一度選択した規則で失敗したら、即座に false を返す

5.5.2　文法規則とコードの対応関係

BNF 規則と Java コードの対応を見てみましょう：

文法規則： `D -> P`

Java コード：

```
private boolean D() { return P(); }
```

文法規則： `P -> ( P ) P | ε`

Java コード：

```
private boolean P() {
    if (先読み文字 == '(') {
        // P -> ( P ) P を適用
    } else {
        // P -> ε  を適用
    }
}
```

- 各非終端記号にメソッドが対応
- 非終端記号の参照はメソッド呼び出しに変換
- 選択は if 文で実現
- 連接は順次実行で実現

と対応しています。

5.5.3　「再帰」「下降」という名前の由来

この実装方法が「再帰下降」と呼ばれる理由は：

1. 再帰：メソッドが自分自身または他のメソッドを呼び出す
2. 下降：文法の開始記号から「下」の規則へと解析が進む

実際に `parse("(())")` を実行すると、以下のような呼び出しの連鎖が発生します：

```
parse() -> D() -> P() -> P() -> P() -> ...
```

この呼び出しスタックが、先ほどスタックで説明した状態と対応しているのです。

5.6　上向き構文解析の基本原理

下向き構文解析を理解したところで、次は**上向き構文解析**を見ていきましょう。こちらは下向きとは全く逆のアプローチを取ります。

5.6.1　シフト還元構文解析の基本アイデア

ナイーブな（素直な）上向き構文解析は**シフト還元構文解析**とも呼ばれます。この名前は、2 つの基本操作から来ています：

- シフト（Shift）：入力から 1 文字読み込んでスタックに積む
- 還元（Reduce）：スタック上の記号列が某規則の右辺と一致したら、左辺に置き換える

下向き構文解析が「文法から入力へ」と進むのに対し、上向き構文解析は「入力から文法へ」と進みます。入力文字列を徐々に「縮めて」いって、最終的に開始記号に辿り着けるかを確認するのです。

比喩としては、テトリスのブロックを積み上げていくようなイメージです。入力文字列を 1 文字ずつ積み上げ、規則にマッチしたらその部分を消すことで、最終的に開始記号だけが残るようにします。

5.6.2　例で学ぶシフト還元構文解析

同じ Dyck 言語を例にしますが、ナイーブな上向き構文解析では空文字列規則（ε 規則）が扱いづらいので、等価な別の文法を使います：

```
D -> $ P $
D -> $ ε  $
P -> P X
P -> X
X -> ( X )
X -> ()
```

この文法の特徴は以下の通りです：

- X は内側の括弧ペアを表す
- P は括弧ペアの列を表す
- D は全体を表す

では、入力文字列（()）に対してシフト還元構文解析を行ってみましょう。

1. 開始記号$をシフト

スタック: [$]
入力: (( )) $

2. 最初の (をシフト

入力: ( )) $
スタック: [$, ( ]

3. さらに (をシフト

入力:)) $
スタック: [$, (, ( ]

4. さらに) をシフト

入力:) $
スタック: [$, (, (,)]

5. スタックの末尾 (,) が規則 X -> () の右辺と一致。2 つの記号を X に**還元**:

入力:) $
スタック: [$, (, X]

6. 次の) をシフト

入力: $
スタック: [$, (, X,)]

7. スタックの末尾 (, X,) が規則 X -> (X) の右辺と一致。3 つの記号を X に**還元**:

入力: $
スタック: [$, X]

8. スタックの末尾 X が規則 P -> X の右辺と一致。1 つの記号を P に**還元**:

入力: $
スタック: [$, P]

9. $をシフト

入力:
スタック: [$, P, $]

9. スタックの末尾$, P, $が規則 D -> $ P $の右辺と一致。3 つの記号を D に**還元**:

入力:
スタック: [D]

入力が空になり、スタックに開始記号 D だけが残ったので、受理されます。

5.6.3 シフト還元構文解析のアルゴリズム

シフト還元構文解析の動作パターンをまとめると：

1. シフト操作：

- 入力から 1 文字読み込んでスタックに積む

- 常に左から右へ順番に読む

2. 還元操作：

- スタック上端の記号列が、ある規則の右辺と一致したら
- その記号列を取り除いて、規則の左辺の非終端記号を積む

3. アクションの選択：

- シフトと還元のどちらを行うかを決定する必要がある
- この決定方法が LR(0)、SLR(1)、LR(1) などの違いになる

4. 受理判定：

- 入力をすべて読み、スタックが開始記号だけになれば受理
- それ以外は拒否

5.6.4　下向きと上向きの違い

ここまでの例からわかるように：

- **下向き**：`D` から始めて `(())` を「生成」しようとする
- **上向き**：`(())` から始めて `D` に「還元」しようとする

上向き構文解析の還元操作は、第 4 章で学んだ「最右導出」の逆操作に相当します。つまり、導出の過程を逆再生しているのです。

5.7　シフト還元構文解析の Java 実装

シフト還元構文解析を Java で実装してみましょう。予測型下向き構文解析とは異なり、明示的にスタックを使い、規則をデータ構造として扱います。

5.7.1　必要なデータ構造

まず、記号（終端記号・非終端記号）を表すクラスと、文法規則を表すクラスが必要です：

```
// 記号を表すインターフェース
interface Element {
    char value();
}

// 終端記号（実際の文字：'(', ')', '$' など）
```

```
record Terminal(char value) implements Element {}

// 非終端記号（文法記号：'D', 'P', 'X' など）
record NonTerminal(char value) implements Element {}
```

次に、文法規則を表す Rule クラスを定義します：

```
import java.util.List;
import java.util.ArrayList;

public record Rule(char lhs, List<Element> rhs) {
    // 可変長引数コンストラクタ（便利のため）
    public Rule(char lhs, Element... rhs) {
        this(lhs, List.of(rhs));
    }

    // スタックの上端がこの規則の右辺と一致するか判定
    public boolean matches(List<Element> stack) {
        if (stack.size() < rhs.size()) return false;

        // スタックの上から rhs.size() 個の要素を比較
        for (int i = 0; i < rhs.size(); i++) {
            Element elementInRule = rhs.get(i);
            Element elementInStack = stack.get(
                stack.size() - rhs.size() + i
            );
            if (!elementInRule.equals(elementInStack)) {
                return false;
            }
        }
        return true;
    }
}
```

5.7.2　シフト還元構文解析器の実装

上記のデータ構造を使って、実際のシフト還元構文解析器を実装します：

```
import java.util.List;
import java.util.ArrayList;

public class DyckShiftReduce {
    private final String input;
    private int position;
    private final List<Rule> rules;
    private final List<Element> stack = new ArrayList<>();

    public DyckShiftReduce(String input) {
        this.input = input;
        this.position = 0;

        // 文法規則の定義
        this.rules = List.of(
            // D -> $ P $
            new Rule('D',
                new Terminal('$'),
                new NonTerminal('P'),
                new Terminal('$')
            ),
            // D -> $ $ (空文字列の場合)
            new Rule('D',
                new Terminal('$'),
                new Terminal('$')
            ),
            // P -> P X
            new Rule('P',
                new NonTerminal('P'),
                new NonTerminal('X')
            ),
            // P -> X
            new Rule('P',
                new NonTerminal('X')
            ),
            // X -> ( X )
            new Rule('X',
```

```
                new Terminal('('),
                new NonTerminal('X'),
                new Terminal(')')
            ),
            // X -> ()
            new Rule('X',
                new Terminal('('),
                new Terminal(')')
            )
        );
    }

    public boolean parse() {
        // 開始記号$をスタックに積む
        stack.add(new Terminal('$'));

        // メインループ：シフトと還元を繰り返す
        while (true) {
            // まず還元を試みる
            if (!tryReduce()) {
                // 還元できない場合、シフトを試みる
                if (position < input.length()) {
                    char c = input.charAt(position);
                    stack.add(new Terminal(c));
                    position++;
                } else {
                    // シフトもできないので終了
                    break;
                }
            }
        }

        // 終端記号$をスタックに積む
        stack.add(new Terminal('$'));

        // 最後に可能な限り還元を繰り返す
        while (tryReduce()) {
```

```
            // 還元ができなくなるまで続ける
        }

        // スタックが [D] のみになったら受理
        return stack.size() == 1 &&
               stack.get(0).equals(new NonTerminal('D'));
    }

    private boolean tryReduce() {
        for (Rule rule : rules) {
            if (rule.matches(stack)) {
                // マッチしたら右辺の長さ分スタックから削除
                for (int i = 0; i < rule.rhs().size(); i++) {
                    stack.remove(stack.size() - 1);
                }
                // 左辺の非終端記号をスタックに追加
                stack.add(new NonTerminal(rule.lhs()));
                return true; // 還元成功
            }
        }
        return false; // 還元できる規則がなかった
    }
}
```

5.7.3　実装のポイント

この実装の重要な点は以下の通りです：

1. シフトより還元を優先：まず `tryReduce()` で還元を試み、できない場合のみシフト
2. 単純な還元ルール：すべての規則を順番にチェックし、最初にマッチしたものを使う
3. 明示的なスタック操作：`List<Element>`をスタックとして使い、要素の追加・削除を明示的に行う

この実装は最も単純なシフト還元構文解析で、実用的なパーサでは洗練されたアルゴリズム（LR(0)、SLR(1)、LR(1)、LALR(1)）が使われます。

5.8　下向き構文解析と上向き構文解析の比較

ここまで下向き構文解析と上向き構文解析の具体例を見てきました。両者にはそれぞれ得意不得意があります。

5.8.1　下向き構文解析の利点と欠点

利点：

1. 直感的な実装

- 文法規則とメソッドが 1 対 1 に対応
- 手書きの構文解析器が書きやすい
- 多くのプログラミング言語のコンパイラが手書き再帰下降を採用

2. 文脈依存の扱いやすさ

- メソッドの引数で情報を渡せる
- 解析中に状態を管理しやすい
- エラー回復やエラーメッセージの生成が容易

3. デバッグのしやすさ

- 通常の関数呼び出しなのでデバッガで追える
- 構文解析の過程が理解しやすい

欠点：

1. 左再帰の問題

たとえば、以下の BNF は上向き型だと普通に解析できますが、工夫なしに下向き型で実装すると無限再帰に陥ってスタックオーバーフローします。

```
A -> A "a" | ε  // 左再帰を含む文法の例（ε は空文字列）
```

この文法は「A は、A の後に a が続くか、何もないか」という意味ですが、A を解析するためにまず A を解析する必要があるので無限ループになります。

このような問題を下向き型で解決する方法も存在します。例えば、以下のように等価な右再帰の文法に書き換えることで除去できます。

```
A -> "a" A | ε
```

この変換により、下向き構文解析で問題となる無限再帰を避けることができます。ただし、文法の書き換えは常に簡単とは限りません。

5.8.2　上向き構文解析の利点と欠点

利点：

1. 左再帰を自然に扱える

2. より広いクラスの文法を扱える
3. 効率的な構文解析表による高速化が可能

欠点：

1. 実装が複雑
2. 文脈依存の処理が難しい

たとえば、それまでの文脈に応じて構文解析のルールを切り替えたくなることがあります。最近の言語によく搭載されている文字列補間などはその最たる例です。

たとえば、"の中は文字列リテラルとして特別扱いされますが、その中で`#{`が出てきたら（Ruby の場合）、通常の式を構文解析するときのルールに戻る必要があります。

このように、文脈に応じて適用するルールを切り替えるのは下向き型が得意です。もちろん、上向き型でも実現できないわけではありません。実際、Ruby の構文解析機は Yacc の定義ファイルから生成されるようになっていますが、Yacc が採用しているのは代表的な上向き構文解析法である `LALR(1)` です。

5.9　LL(1) - 代表的な下向き構文解析アルゴリズム

ここからは、具体的な構文解析アルゴリズムについて詳しく見ていきましょう。まずは下向き構文解析の代表格である **LL(1)** から始めます。

5.9.1　LL(1) とは？

LL(1) という名前は以下の意味を持ちます：

1. 最初の L：**L**eft-to-right（左から右へ入力を読む）
2. 2 番目の L：**L**eftmost derivation（最左導出を行う）
3. (1)：1 トークン先読み

つまり「1 トークン先を見て、次に適用すべき規則を一意に決定できる」という制約を持つ下向き構文解析法です。

この「LL」という名前の由来ですが、構文解析の研究が盛んだった 1960 年代に、様々な手法を分類するために考案された命名規則です。同様に、後で出てくる「LR」も「Left-to-right, Rightmost derivation」を表します。

5.9.2　LL(1) の直感的な理解

身近な例で考えてみましょう。Java の if 文と while 文の解析を例にします：

```
if (age < 18) {
    System.out.println("18 歳未満です");
}
```

```
while (count > 0) {
    count--;
}
```

プログラマがこのコードを見たとき、最初のトークンだけで文の種類を判断できます：- `if` で始まる → if 文だ！ - `while` で始まる → while 文だ！

LL(1) は、まさにこの「最初の 1 トークンで判断」というアイデアをアルゴリズム化したものです。

しかし、すべての構文がこのように単純ではありません。例えば、以下の単純な算術式を考えてみましょう。`Num` は終端記号とします：

```
E  → Num
E  → ( E )
E  → - E
```

この場合、`E` は以下のいずれかで始まる可能性があります：

- Num（例：`42`）
- 左括弧（例：`(3 + 5)`）
- マイナス記号（例：`-10`）

つまり、`E` という非終端記号は複数のトークンで始まる可能性があるのです。

5.9.3　なぜ FIRST 集合と FOLLOW 集合が必要か

LL(1) を実現するには、以下の 2 つの問題を解決する必要があります：

1. 複数のトークンで始まる構文

上の算術式の例のように、一つの非終端記号が複数のトークンで始まる場合があります。これを体系的に扱うために **FIRST 集合**（その非終端記号から始まる可能性のあるトークンの集合）が必要です。

2. 省略可能な要素

Java の if 文には、else 節があるものとないものがあります：

```
// else節なし
if (condition) { ... }

// else節あり
if (condition) { ... } else { ... }
```

else 節は「あってもなくてもいい」要素です。このような場合、else 節の後に何が来るかを知る必要があります。これが **FOLLOW 集合**（その非終端記号の後に来る可能性のあるトークンの集合）です。

5.9.4 FIRST 集合と FOLLOW 集合

LL(1) 構文解析を実現するには、これらの集合を計算して利用します：

5.9.4.1 FIRST 集合

ある非終端記号から導出される文字列の「最初に現れうるトークンの集合」を **FIRST 集合**と呼びます。

正式には、非終端記号 A に対して以下のように定義されます：

`FIRST(A) = { a | A ⇒* aβ であるような終端記号 a }`

ここで、`A ⇒* aβ` は、非終端記号 A が何らかの導出を経て、終端記号 a で始まる文字列に変換できることを意味します。

例えば、先ほどでてきた以下の算術式の文法を考えます。

```
E → Num
E → ( E )
E → - E
```

この場合、FIRST 集合は次のようになります：

`FIRST(E) = { '(', '-', Num }`

5.9.4.2 nullable - 空文字列を生成できるか

LL(1) 構文解析で重要な概念として、**nullable**（ヌラブル）があります。これは、ある非終端記号が空文字列（ε ）を生成できるかどうかを示すブール値です。

nullable という名前は Java の `null` とは関係ありません。「空にできる」つまり「何も生成しないことができる」という意味です。

例えば、以下の文法を考えます：

```
A -> $ a B C $
B -> b | ε
C -> d C | ε
```

この場合：

- `nullable(A) = false`（A は必ず a と$を含むため）
- `nullable(B) = true`（`B -> ε` という規則があるため）
- `nullable(C) = true`（`C -> ε` という規則があるため）

何故 nullable が重要かというと、**FIRST 集合**及び **FOLLOW 集合**を計算するためです。**FIRST 集合**の計算では、非終端記号が空文字列を生成できるかどうかを確認する必要があります。また、**FOLLOW 集合**の計算でも、非終端記号が空文字列を生成できる場合、その後に来るトークンを正しく扱うために nullable が必要です。

5.9.4.3 FOLLOW 集合

非終端記号の後に現れうるトークンの集合を **FOLLOW 集合**と呼びます。これは、**nullable** な非終端記号（空文字列規則を持つ非終端記号）を扱うために特に重要です。

例えば、以下の文法を考えてみます：

```
A -> $ a B C $
B -> b | ε
C -> c C | ε
```

この場合、FOLLOW 集合は次のようになります：

```
FOLLOW(A) = { $ }      // A は開始記号
FOLLOW(B) = { c, $ }   // B の後には c または入力の終端が来る
FOLLOW(C) = { $ }      // C の後には入力の終端が来る
```

5.9.5 FIRST 集合、FOLLOW 集合、nullable の計算方法

これらの集合を実際に計算する方法を見ていきましょう。

5.9.5.1 nullable の計算アルゴリズム

1. すべての非終端記号について `nullable = false` に初期化
2. `A -> ε` の形の規則があれば `nullable(A) = true`
3. `A -> X  X  ... X` について、すべての `X` が nullable なら `nullable(A) = true`
4. `false` -> `true` への変化がなくなるまで繰り返す

例：

```
E → T E'
E' → + T E' | ε
T → F T'
T' → * F T' | ε
F → ( E ) | id
```

計算過程：

- 初期化：すべて `false`
- `E' → ε` があるので `nullable(E') = true`

- T' → ε があるので nullable(T') = true
- 他の非終端記号はε 規則を持たず、右辺も nullable でないので false のまま

結果：

- nullable(E) = false
- nullable(E') = true
- nullable(T) = false
- nullable(T') = true
- nullable(F) = false

5.9.5.2 FIRST 集合の計算アルゴリズム

各規則 A → α について次の手順で計算します：

1. α = X1 X2 ... Xn とする
2. X1 が終端記号なら、FIRST(A) に X1 を追加
3. X1 が非終端記号なら：
   - FIRST(X1) - {ε }を FIRST(A) に追加
   - X1 が **nullable** なら X2 も確認（以下同様）
4. すべての X1 が **nullable** なら、ε を FIRST(A) に追加

例の文法での計算は以下のようになります：

1. F → (E) | id
   - FIRST(F) = { '(', id }（両方の規則の最初の終端記号）
2. T → F T'
   - FIRST(T) = FIRST(F) = { '(', id }（F は非終端記号なので FIRST(F) を使用）
3. E → T E'
   - FIRST(E) = FIRST(T) = { '(', id }
4. E' → + T E' | ε
   - FIRST(E') = { '+', ε }（最初の規則から +、2 番目の規則からε ）
5. T' → * F T' | ε
   - FIRST(T') = { '*', ε }

5.9.5.3 FOLLOW 集合の計算アルゴリズム

1. すべての非終端記号の FOLLOW を空集合に初期化
2. 開始記号 S に対して FOLLOW(S) = {$}（入力終端）
3. 規則 B → α A β に対して：
   - FIRST(β) - {ε }を FOLLOW(A) に追加
   - β が nullable なら FOLLOW(B) を FOLLOW(A) に追加

4. 規則 B → α A（末尾に A）に対して：
    - FOLLOW(B) を FOLLOW(A) に追加
5. 変化がなくなるまで繰り返す

例の文法での計算は以下のようになります：

1. 初期状態：
    - FOLLOW(E) = {$}（開始記号として）
2. 規則 F → (E) から：
    - E の後に) が来るので、FOLLOW(E) = {$, ')'}
3. 規則 E → T E' から：
    - T の後に E' が来る。FIRST(E') = {'+', ε }
    - E' が **nullable** なので、FOLLOW(E) も FOLLOW(T) に追加
    - FOLLOW(T) = {'+', $, ')'}
4. 規則 E → T E' から：
    - E' は末尾なので、FOLLOW(E') に FOLLOW(E) を追加
    - FOLLOW(E') = {$, ')'}

同様に計算を続けると以下のようになります：

- FOLLOW(E) = {$, ')'}
- FOLLOW(E') = {$, ')'}
- FOLLOW(T) = {'+', $, ')'}
- FOLLOW(T') = {'+', $, ')'}
- FOLLOW(F) = {'*', '+', $, ')'}

5.9.6 LL(1) 構文解析と FIRST 集合・FOLLOW 集合の関係

6 章でててくる構文解析器生成系では、FIRST 集合と FOLLOW 集合を使って、LL(1) 構文解析表を作成します。これは「非終端記号」と「先読みトークン」の組み合わせから、適用すべき規則を決定する表です。ただし、手書きで LL(1) 構文解析器を実装する場合は、表を明示的に作成せず、FIRST 集合と FOLLOW 集合に相当するものを手で（あるいは頭の中で）作って、適用すべき規則を決定します。

たとえば、以下のような文法があったとします：

A → α
A → β

このとき、FIRST 集合と FOLLOW 集合を使って、どちらの規則を適用するかを決定します：

- もし FIRST(α) と FIRST(β) に共通の要素がなければ、先読みトークンを見てどちらの規則を適用するか判断できます
  - このとき、共通の要素があればそれは衝突が発生しており、LL(1) 構文解析は不可能です
- もしα またはβ が nullable（空文字列を生成可能）な場合は、FOLLOW 集合も考慮する必要があります

次に、FOLLOW 集合が必要になる具体例を見ていきましょう。以下の文法を考えます：

S → A B
A → a | ε　　（A は空文字列も生成可能）
B → b

A を解析する際、先読みトークンが a なら A → a を適用しますが、b の場合はどうでしょうか？

- FIRST(A) = {a}（ε は含まない）
- nullable(A) = true（A → ε があるため）
- FOLLOW(A) = {b}（A の後に B が来て、FIRST(B) = {b}）

先読みトークンが b の場合：

- b ∉ FIRST(A) なので、通常なら A → a は適用できない
- しかし nullable(A) = true かつ b ∈ FOLLOW(A) なので、A → ε を適用すべき

つまり、LL(1) 構文解析では以下のような戦略をとります：

- 先読みトークンが FIRST(α) に含まれる → A → α を適用
- nullable(α) = true かつ先読みトークンが FOLLOW(A) に含まれる→ A → α を適用（α は空文字列を生成）

FOLLOW 集合を適用する場合でも先程と同じように衝突が発生する可能性があります。例えば、以下のような文法を考えてみましょう：

S → A B | C D
A → a | ε
C → c | ε
B → b
D → b

この文法では FIRST と FOLLOW は次のようになります：

- FIRST(A) = {a}, nullable(A) = true
- FIRST(C) = {c}, nullable(C) = true
- FOLLOW(A) = {b}（A の後に B が来る）
- FOLLOW(C) = {b}（C の後に D が来る）

ここで、先読みトークンが b の場合、次のような問題が発生します：

- S → A B を選ぶなら、A → ε を適用して B → b に進む
- S → C D を選ぶなら、C → ε を適用して D → b に進む

どちらの場合も b ∈ FOLLOW(A) かつ b ∈ FOLLOW(C) なので、S の段階でどちらの規則を選ぶべきか判断できません。この場合は、文法が LL(1) ではない、つまり LL(1) での解析が不可能であることを示しています。

構文解析表を明示的に作らなくても、FIRST 集合と FOLLOW 集合の概念を理解していれば、直感的に LL(1) パーサを実装できるのです。

5.9.7 LL(1) の問題点と限界

LL(1) はシンプルで実用的ですが、いくつかの限界があります。

1. 共通前置辞問題

LL(1) では、同じ非終端記号から始まる複数の規則がある場合、先読みトークンだけではどの規則を適用すべきか判断できません。

```
S -> a B
S -> a C
```

両方の規則が a で始まるため、1 文字先読みでは判断できません。この問題は**左因子化**（Left Factoring）で解決できます：

```
S  -> a S'
S' -> B
S' -> C
```

左因子化は直観的には、共通の前置きを抽出して新しい非終端記号を導入することで、先読みトークンだけで規則を一意に決定できるようにする変換操作です。

2. 左再帰の問題

LL(1) の最大の欠点は、**左再帰**を扱えないことです。

```
E -> E + T
E -> T
```

この問題は**左再帰の除去**で解決できます：

```
E  -> T E'
E' -> + T E'
E' -> ε
```

しかし、この変換により文法の直感性が失われ、抽象構文木の構築も複雑になります。

ここまでで下向き構文解析の基本的な考え方と LL(1) について学びました。次は、より効率的な上向き構文解析の代表格である **LR 構文解析**を見ていきましょう。

5.10　LR 構文解析 - 素朴なシフト還元の効率化

最初に見た素朴なシフト還元構文解析には、大きな問題がありました：

1. **いつシフトして、いつ還元するか？** - 毎回スタック全体を調べる必要がある
2. **どの規則で還元するか？** - すべての規則を順番に試す必要がある

これらの判断を効率的に行うのが **LR 構文解析**です。

5.10.1　素朴な方法の問題点

先ほどの `DyckShiftReduce` の実装を思い出してください：

```
private boolean tryReduce() {
    for (Rule rule : rules) {  // すべての規則を試す！
        if (rule.matches(stack)) {  // スタック全体をチェック！
            // 還元処理
        }
    }
    return false;
}
```

入力が長くなると、この方法は非効率的です。LR 構文解析は、この問題を**オートマトン**で解決します。

5.10.2　LR 構文解析のアイデア

素朴なシフト還元構文解析の問題点は、毎回スタック全体を見て、どの規則が適用できるかを総当たりで確認する必要があることでした。しかし、ここで重要な観察があります：

スタックの「状態」は有限個に分類できる

第 4 章で学んだ有限オートマトンを思い出してください。有限オートマトンは、有限個の状態と状態間の遷移で表現される計算モデルでした。LR 構文解析の革新的なアイデアは、シフト還元構文解析の過程を有限オートマトンとして表現できることです。

具体的には：

1. スタックに積まれた記号列の「パターン」を状態として管理
2. シフトや還元の操作を状態遷移として表現

3. 各状態で「次に何をすべきか」を事前に計算して表に保存

これにより、構文解析時は単に「現在の状態」と「次の入力」から表を引くだけで、次の動作（シフトか還元か）が決定できるようになります。これが **LR 構文解析表**と呼ばれるものです。

5.10.3 LR(0) 項 - 解析の進行状況を表す

LR 構文解析の中核となる概念が **LR(0) 項**です。これは、構文解析の進行状況を表現する方法です。

文法規則の中にドット（・）を挿入して、「どこまで認識したか」を表します：

```
A → ・α β      （まだ何も認識していない）
A → α ・β      （α まで認識した）
A → α β ・     （すべて認識した = 還元可能）
```

例えば、規則 `E → T + E` に対する LR(0) 項目は：

- `E → ・T + E`：まだ何も認識していない
- `E → T・+ E`：T まで認識した
- `E → T +・E`：T + まで認識した
- `E → T + E・`：すべて認識した（還元可能）

の 5 つとなります。

5.10.4 LR(0) 項集合 - 同じ「状況」をまとめる

ここで重要な概念が **LR(0) 項集合**です。構文解析の過程で、複数の LR(0) 項が同時に「有効」になることがあります。これらをまとめて 1 つの「状態」として扱います。

例えば、以下のような状況を考えてみましょう：

```
現在の状態 = {
    E → T・+ E    （T を認識済み、次は + を期待）
    E → T・       （T を認識済み、ここで還元も可能）
}
```

この状態では、2 つの可能性が同時に存在しています：

1. 次に + が来れば、最初の規則に従って解析を続ける
2. ここで `E → T` として還元することもできる

LR(0) 項集合は、このような「同じ状況で有効な項目の集まり」を表現します。オートマトンの各状態は、実は LR(0) 項の集合なのです。

なぜ集合として扱うのでしょうか？ それは、構文解析の過程で複数の可能性を同時に追跡する必要があるからです。

例えば、if 文の解析中に、else 節があるかないかの両方の可能性を考慮する必要があるような場合です。

5.10.5 閉包 - 項目の展開

LR(0) 項集合を構築する上で最も重要なのが、LR(0) 項の**閉包**（Closure）です。これは、ある LR(0) 項から「論理的に導かれる全ての項目」を表す概念です。

例えば、項目 S → ・E $ があるとします。これは「これから E を認識したい」という状況を表しています。しかし、E を認識するためには、E から始まる規則も考慮する必要があります。このために閉包を求めます。

S → ・E $　　（これから E を認識したい）
↓
E を認識するには、E から始まる規則も必要：
E → ・T
E → ・E + T

さらに、T を認識するためには：

T → ・id

このように、ドットの直後に非終端記号がある場合、その非終端記号から始まる全ての規則を追加していく必要があります。

5.10.5.1 LR(0) 項の閉包を求めるアルゴリズム

```
closure(I) {
    J = I
    repeat {
        for（各項目 A → α ・B β  in J）{
            for（各規則 B → γ ）{
                J = J ∪ {B → ・γ }
            }
        }
    } until J に変化がなくなる
    return J
}
```

5.10.6 拡張開始記号の必要性

LR 構文解析では、文法に**拡張開始記号**（augmented start symbol）を追加するのが一般的です。元の開始記号が S だとすると、新しい開始記号 S’ を追加し、規則 S' → S $ を追加します。

なぜこれが必要なのでしょうか？ 主な理由は以下の通りです：

1. **受理状態の明確化**：構文解析が「いつ終了するか」を明確にするため
2. **還元の一意性**：開始記号への還元を他の還元と区別するため
3. **実装の簡潔性**：特別な終了処理を不要にするため

例えば、以下の文法を考えてみましょう：

```
E → T
E → E + T
T → id
```

この文法で入力 `id` を解析する場合、`T → id` で還元した後、`E → T` で還元して E になりますが、「ここで解析終了」という明確な信号がありません。

拡張開始記号を追加すると、以下のようになります：

```
S' → E $
E → T
E → E + T
T → id
```

`S' → E $・`という項目に到達したときに「入力を全て消費し、開始記号に還元できた」ことが明確になります。

5.10.7　閉包からオートマトンへ

閉包を理解したところで、これをどのようにオートマトン構築に活用するかを見ていきましょう。

LR(0) オートマトンは以下の要素から構成されます：

1. 状態：LR(0) 項集合（閉包を取った後の項目の集合）
2. 遷移：ある状態から記号を読んで次の状態へ移る関係
3. 初期状態：拡張開始記号から始まる項目の閉包
4. 受理状態：`S' → E $・`を含む状態

構築の基本的な流れは次のようになります：

1. 初期項目 `S' → ・E $` から始める
2. その閉包を計算して初期状態とする
3. 各状態から可能な遷移を計算
4. 新しい状態が生まれたら、その閉包を計算
5. 新しい状態が生まれなくなるまで繰り返す

このプロセスはグラフの幅優先探索に似ています。各ノード（状態）から到達可能な全てのノードを探索していくのです。

以下の文法に対して LR(0) オートマトンを構築してみましょう：

```
S' → E $      （拡張開始記号）
E → T
E → E + T
T → id
```

5.10.8　ステップ 1：初期状態（I0）の作成

初期状態は、拡張開始記号から始まる項目のクロージャです：

```
I0 = closure({S' → ・E $})
```

クロージャを計算すると：

1. `S' → ・E $` がある
2. ドットの後に E があるので、E から始まる規則を追加：

- `E → ・T`
- `E → ・E + T`

3. `E → ・T` のドットの後に T があるので、T から始まる規則を追加：

- `T → ・id`

結果：

```
I0 = {
    S' → ・E $
    E → ・T
    E → ・E + T
    T → ・id
}
```

5.10.9　ステップ 2：GOTO 関数による遷移の計算

GOTO 関数は、ある状態から記号を読んだときの次の状態を計算します：

```
GOTO(I, X) = closure({A → α X・β  | A → α ・X β  ∈ I})
```

I0 からの各遷移を計算してみましょう：

GOTO(I0, E) = I0 でドットの後に E がある項目に対して状態 I1 を求める：

1. `S' → ・E $` のドットの後に E があるので、`S' → E・$` を得る
2. `E → ・E + T` のドットの後に E があるので、`E → E・+ T` を得る
3. 上記の項目集合の閉包をとって状態 I1 を得る：

```
I1 = closure({S' → E・$, E → E・+ T})
   = {
       S' → E・$
       E → E・+ T
   }
```

GOTO(I0, T) = I0 でドットの後に T がある項目に対して状態 I2 を求める：

1. `E → ・T` のドットの後に T があるので、`E → T・`を得る
2. 上記の項目集合の閉包をとって状態 I2 を得る：

```
I2 = closure({E → T・})
   = {E → T・}
```

GOTO(I0, id) = I0 でドットの後に id がある項目に対して状態 I3 を求める：

1. `T → ・id` のドットの後に id があるので、`T → id・`を得る
2. 上記の項目集合の閉包をとって状態 I3 を得る：

```
I3 = closure({T → id・})
   = {T → id・}
```

5.10.10　ステップ 3：全ての状態を探索

I1 からの遷移：

GOTO(I1,) = $I1$でドットの後に がある項目に対して状態 I4 を求める：

1. `S' → E・$` のドットの後に$があるので、`S' → E $・`を得る
2. これは受理状態なので、I4 は受理状態として定義

```
I4 = {S' → E $・}  （受理状態）
```

GOTO(I1, +) = I1 でドットの後に + がある項目に対して状態 I5 を求める：

1. `E → E・+ T` のドットの後に + があるので、`E → E +・T` を得る
2. `T` から始まる規則も追加するため、`T → ・id` を得る
3. 上記の項目集合の閉包をとって状態 I5 を得る：

```
I5 = closure({E → E +・T, T → ・id})
   = {
       E → E +・T
       T → ・id
   }
```

I5 からの遷移：

GOTO(I5, T) = I5 でドットの後に T がある項目に対して状態 I6 を求める：

1. E → E +・T のドットの後に T があるので、E → E + T・を得る
2. 上記の項目集合の閉包をとって状態 I6 を得る：

I6 = {E → E + T・}

GOTO(I5, id) = I5 でドットの後に id がある項目に対して状態 I3 を求める：

1. T → ・id のドットの後に id があるので、T → id・を得る
2. これは既存の状態 I3 と同じなので、新しい状態は作成しない
3. つまり、I3 を再利用する

I3 = {T → id・}

5.10.11　完成した LR(0) オートマトン

状態：

- I0: {S' → ・E $, E → ・T, E → ・E + T, T → ・id}
- I1: {S' → E・$, E → E・+ T}
- I2: {E → T・}
- I3: {T → id・}
- I4: {S' → E $・}
- I5: {E → E +・T, T → ・id}
- I6: {E → E + T・}

遷移：

- I0 --E--> I1
- I0 --T--> I2
- I0 --id--> I3
- I1 --$--> I4
- I1 --+--> I5
- I5 --T--> I6
- I5 --id--> I3

5.10.12　LR(0) 構文解析表の構築

オートマトンから構文解析表を作成します。表には 2 種類の情報を記録します：

1. ACTION 表：終端記号に対するアクション

- s n：シフトして状態 n へ遷移

- `r n`：規則 n で還元
- `acc`：受理

2. GOTO 表：非終端記号に対する遷移

5.10.13　構築規則

1. 項目 `A → α ・a β` が状態 i にあり、GOTO(i, a) = j なら：

- ACTION[i, a] = s j

2. 項目 `A → α ・` が状態 i にある（還元可能）なら：

- 全ての終端記号に対して ACTION[i, *] = r (A →α)

3. 項目 `S' → E $・` が状態 i にあるなら：

- ACTION[i, $] = acc

4. GOTO(i, A) = j なら：

- GOTO[i, A] = j

5.10.13.1　完成した解析表

状態	ACTION			GOTO	
	id	+	$	E	T
0	s3			1	2
1		s5	acc		
2	r1	r1	r1		
3	r3	r3	r3		
4	acc				
5	s3				6
6	r2	r2	r2		

規則番号：

- 0: S’ → E $
- 1: E → T
- 2: E → E + T
- 3: T → id

5.10.14 LR(0) 構文解析の実行

では、構築した構文解析表を使って、入力 `id + id $` を解析してみましょう：

- ステップ 1：
  - スタック: [0]
  - 入力: id + id $
  - アクション: ACTION[0, id] = s3
  - 実行: id をシフトして状態 3 へ
- ステップ 2：
  - スタック: [0, 3]
  - 入力: + id $
  - アクション: ACTION[3, +] = r3 (T → id)
  - 実行: T → id で還元
    - スタックから 1 つ削除: [0]
    - GOTO[0, T] = 2
    - 状態 2 をプッシュ: [0, 2]
- ステップ 3：
  - スタック: [0, 2]
  - 入力: + id $
  - アクション: ACTION[2, +] = r1 (E → T)
  - 実行: E → T で還元
    - スタックから 1 つ削除: [0]
    - GOTO[0, E] = 1
    - 状態 1 をプッシュ: [0, 1]
- ステップ 4：
  - スタック: [0, 1]
  - 入力: + id $
  - アクション: ACTION[1, +] = s5
  - 実行: + をシフトして状態 5 へ
- ステップ 5：
  - スタック: [0, 1, 5]
  - 入力: id $
  - アクション: ACTION[5, id] = s3
  - 実行: id をシフトして状態 3 へ
- ステップ 6：
  - スタック: [0, 1, 5, 3]
  - 入力: $

  - アクション: ACTION[3, $] = r3 (T → id)
  - 実行: T → id で還元
    - スタックから 1 つ削除: [0, 1, 5]
    - GOTO[5, T] = 6
    - 状態 6 をプッシュ: [0, 1, 5, 6]
- ステップ 7：
  - スタック: [0, 1, 5, 6]
  - 入力: $
  - アクション: ACTION[6, $] = r2 (E → E + T)
  - 実行: E → E + T で還元
    - スタックから 3 つ削除: [0]
    - GOTO[0, E] = 1
    - 状態 1 をプッシュ: [0, 1]
- ステップ 8：
  - スタック: [0, 1]
  - 入力: $
  - アクション: ACTION[1, $] = acc -実行: 受理 !

このように、LR(0) 構文解析は：

1. 現在の状態と入力記号から表を引く
2. シフトか還元かを決定
3. 還元の場合は、どの規則を使うかも一意に決まる

という単純な操作の繰り返しで、効率的に構文解析を実現します。

5.10.15　LR(0) の限界：コンフリクト

LR(0) は効率的ですが、重要な限界があります。以下の算術式の文法を考えてみましょう：

```
E → E + T
E → T
T → id
```

この文法で LR(0) オートマトンを構築すると、ある状態で以下の項目を含むことになります：

```
状態 X = {
    E → T・        （還元準備完了）
    E → E・+ T     （+ を期待）
}
```

この状態で問題が発生します。何故なら：

- 入力が + なら → シフトして + を読むべき
- 入力が$（終端）なら → E → T で還元すべき

しかし、LR(0) は**次の入力を見ない**ので、どちらを選ぶか決められません。これを**シフト/還元コンフリクト**と呼びます。

5.10.16　なぜコンフリクトが起きるのか

素朴なシフト還元では、実は暗黙的に「次の文字」を見ていました。しかし、LR(0) では「状態だけ」で判断しようとするため、情報が不足するのです。

5.11　SLR(1)、LR(1)、LALR(1) - 先読みによる改良

LR(0) のコンフリクトを解決するため、**先読み**（次の入力を見る）を導入した手法が開発されました。

5.11.1　SLR(1) - FOLLOW 集合による改良

SLR(1)（Simple LR(1)）は、LL(1) で学んだ FOLLOW 集合の考え方を使います。

基本アイデア：

「ある非終端記号 A の後に来うる記号」が分かれば、還元すべきタイミングが分かる

先ほどのコンフリクトの例を考えてみましょう：

```
状態 X = {
    E → T・        （還元準備完了）
    E → E・+ T     （+ を期待）
}
```

この状態で、次の入力が + か$かによって、どのアクションを取るべきかが変わります。SLR(1) では、以下のように判断します：

- FOLLOW(E) = {+, $}（E の後には + か$が来る）
- 次の入力が + なら：
  - シフト（E → E・+ T のため）
  - 還元も可能（+ ∈ FOLLOW(E)）だが、シフトを優先
- 次の入力が$なら：
  - 還元のみ（$ ∈ FOLLOW(E)）

これで多くのコンフリクトが解消されます。

5.11.1.1 SLR(1) 構文解析表の構築

SLR(1) のアクション表の構築は、LR(0) と以下の点で異なります：

LR(0) の還元ルール（問題あり）：- 項目 `A → α ・`が状態 i にあるなら、**全ての終端記号**に対して還元

SLR(1) の還元ルール（改善版）：- 項目 `A → α ・`が状態 i にあるなら、**FOLLOW(A) 内の終端記号**に対してのみ還元

つまり、「A の後に来る可能性がある記号」の時だけ還元することで、不要な還元を防ぎます。

5.11.2 SLR(1) で解決できるコンフリクトの例

先ほどの文法で SLR(1) がどのようにコンフリクトを解決するか見てみましょう：

```
S' → E $
E → E + T
E → T
T → id
```

LR(0) では状態 2 で以下のコンフリクトが発生していました：

状態 2 = `{E → T・}`

LR(0) では、この状態で全ての終端記号（id, +, $）に対して `E → T` で還元しようとしました。しかし、これは過剰です。

SLR(1) では、FOLLOW(E) を計算します：

1. `S' → E $` より、E の後に$が来る可能性がある
2. `E → E + T` より、E の後に + が来る可能性がある

よって `FOLLOW(E) = {+, $}`

これにより、状態 2 では以下のようになります：

- + に対して：`E → T` で還元
- $に対して：`E → T` で還元
- `id` に対して：**何もしない**（エラー）

このように、SLR(1) は無駄な還元を防ぎ、より正確な構文解析表を構築できます。

5.11.3 SLR(1) 構文解析表の完成版

状態	ACTION			GOTO	
	id	+	$	E	T
0	s3			1	2
1		s5	acc		
2		r1	r1		
3		r3	r3		
4	acc				
5	s3				6
6		r2	r2		

LR(0) との違いは、状態 2 と状態 3、状態 6 の id 列が空欄（エラー）になっていることです。これにより、不適切な還元を防いでいます。

5.11.4 SLR(1) の限界

SLR(1) は多くの実用的な文法を扱えますが、以下のような場合にはまだコンフリクトが残ります：

```
S → L = R | R
L → * R | id
R → L
```

この文法は「代入文」を表現しています（*p = q のような）。この文法では、ある状態で以下の項目が含まれます：

```
状態 X = {
    R → L・      （L で還元準備完了）
    S → L・= R   （=を期待）
}
```

FOLLOW(R) = {=, $}なので、入力が=の時に：

- シフト（S → L・= R のため）
- 還元（R → L、かつ= ∈ FOLLOW(R) のため）

両方が可能となり、SLR(1) でもコンフリクトが解決できません。このような場合には、より強力な LR(1) が必要になります。

5.11.5 LR(1) - より精密な先読み

SLR(1) でも解決できないコンフリクトがあります。**LR(1)** は、各項目に「その項目に到達した文脈での先読み記号」を付加します：

```
[E → T・, +]    （この還元の後には + が来る）
```

[E → T・, $]　（この還元の後には$が来る）

同じ E → T・でも、文脈によって異なる先読み記号を持つことで、より精密な判断が可能になります。

5.11.6 LR(1) 項目の構造

LR(1) 項目は、LR(0) 項目に**先読み記号**（lookahead symbol）を追加したものです：

[A → α ・β , a]

ここで、a は先読み記号で、「この項目で還元した後に期待される記号」を表します。

例えば：- [E → T・, +]：T を認識済み、還元後に + が来ることを期待 - [E → T・, $]：T を認識済み、還元後に入力終端が来ることを期待

同じ E → T・でも、文脈によって異なる先読み記号を持つため、より精密な判断が可能です。

5.11.7 LR(1) 閉包の計算

LR(1) の閉包計算は、先読み記号の伝播を考慮する必要があります：

```
closure(I) {
    J = I
    repeat {
        for（各項目 [A → α ・B β , a] in J）{
            for（各規則 B → γ ）{
                for（FIRST(β a) 内の各終端記号 b）{
                    J = J ∪ {[B → ・γ , b]}
                }
            }
        }
    } until J に変化がなくなる
    return J
}
```

重要なのは、FIRST(β a) の計算です：

- β が空でない場合：FIRST(β) を使う
- β が空または nullable な場合：a も含める

5.11.8 LR(1) の具体例

SLR(1) で問題となった代入文の文法で、LR(1) がどのように動作するか見てみましょう：

```
S' → S $
S → L = R | R
L → * R | id
R → L
```

5.11.8.1 初期状態 I0 の構築

I0 = closure({[S' → ・S $, $]})

閉包を計算すると：

```
I0 = {
    [S' → ・S $, $]
    [S → ・L = R, $]
    [S → ・R, $]
    [L → ・* R, =]     // 注目：先読みは=
    [L → ・id, =]      // 注目：先読みは=
    [R → ・L, $]
    [L → ・* R, $]     // 注目：先読みは$
    [L → ・id, $]      // 注目：先読みは$
}
```

ここで重要なのは、同じ `L → ・* R` や `L → ・id` が、異なる先読み記号（=と$）を持って複数存在することです。

5.11.8.2 問題となる状態の解析

GOTO(I0, L) を計算すると：

```
I2 = {
    [S → L・= R, $]    // =を期待
    [R → L・, $]       // $を期待（還元可能）
}
```

この状態で、次の入力が=の場合、以下のようになります：

- `[S → L・= R, $]` により、シフトする
- `[R → L・, $]` は、先読みが$なので還元しない

つまり、LR(1) では**コンフリクトが解決**されます！

5.11.9 LR(1) と SLR(1) の違い

同じ状態を SLR(1) で見ると：

```
状態 X = {
    S → L・= R
    R → L・
}
```

SLR(1) では FOLLOW(R) = {=, $}なので、=に対して還元も可能となり、コンフリクトが発生します。

LR(1) の優位性は以下の点です：

- 文脈を考慮：項目がどのような経路で到達したかを記憶
- 精密な先読み：FOLLOW 集合ではなく、実際の文脈での先読み記号を使用
- より多くの文法を扱える：SLR(1) で発生するコンフリクトの多くを解決

5.11.10 LR(1) の欠点

LR(1) は強力ですが、以下の重大な欠点があります：

1. **状態数の爆発**

- 同じコア（ドットの位置まで同じ項目）でも、先読み記号が異なれば別状態
- 実用的な文法では、状態数が数千〜数万になることも

2. **構文解析表のサイズ**

- 状態数 × 記号数のテーブルが必要
- メモリ使用量が問題となる場合がある

3. **構築時間**

- 状態数が多いため、オートマトン構築に時間がかかる

例えば、プログラミング言語の文法では次のようになります：

- LR(0)：数百状態
- SLR(1)：数百状態（LR(0) と同じ）
- LR(1)：数千〜数万状態
- LALR(1)：数百状態（後述）

この問題を解決するために開発されたのが LALR(1) です。

5.11.11 LALR(1) - 実用的な妥協点

LALR(1)（Look-Ahead LR(1)）は、LR(1) の強力さと LR(0) のコンパクトさを両立させた「実用的な妥協点」です。

5.11.12　LALR(1) の基本アイデア

LR(1) の問題は状態数の爆発でした。しかし、よく観察すると多くの状態が「ほぼ同じ」であることがわかります：

```
状態 10: {
    [E → T・, +]
    [T → T・* F, +]
}

状態 25: {
    [E → T・, $]
    [T → T・* F, $]
}
```

これらの状態は、先読み記号以外は全く同じです。このような状態の「コア」（先読み記号を除いた部分）は同一です：

```
コア: {
    E → T・
    T → T・* F
}
```

LALR(1) は、同じコアを持つ LR(1) 状態をマージすることで、状態数を削減します。

5.11.13　LALR(1) 状態のマージ

マージのプロセスを具体的に見てみましょう：

```
LR(1) 状態 A: {
    [A → α ・β , a]
    [B → γ ・δ , b]
}

LR(1) 状態 B: {
    [A → α ・β , c]
    [B → γ ・δ , d]
}
```

これをマージすると、以下のようになります：

```
LALR(1) 状態: {
    [A → α ・β , a/c]
    [B → γ ・δ , b/d]
```

```
}
```

先読み記号が和集合になることに注目してください。

5.11.14　LALR(1) の構築方法

LALR(1) オートマトンの構築には主に 2 つの方法があります：

5.11.14.1　方法 1：LR(1) からの変換

1. 完全な LR(1) オートマトンを構築
2. 同じコアを持つ状態を識別
3. それらの状態をマージ（先読み記号は和集合）
4. 遷移関係を調整

この方法は概念的に分かりやすいですが、一時的に大きな LR(1) オートマトンを構築する必要があります。

5.11.14.2　方法 2：直接構築

1. LR(0) オートマトンを構築
2. 各状態に対して、効率的に先読み記号を計算
3. 先読み伝播グラフを使って先読み記号を伝播

実用的な構文解析器生成系（Yacc、Bison）は、効率的な方法 2 を採用しています。

5.11.15　LALR(1) で新たに生じるコンフリクト

状態のマージにより、LR(1) では解決できていたコンフリクトが再発することがあります。以下の文法を考えてみましょう：

```
S' → S $
S → a A d | b B d | a B e | b A e
A → c
B → c
```

この文法は、前半の記号（a か b）と後半の記号（d か e）の組み合わせで、中間の A か B を決定する必要があります。

LR(1) では、以下のような状態が別々に存在します：

```
状態 X: { [A → c・, d] }  // a で始まり d で終わる文脈
状態 Y: { [A → c・, e] }  // b で始まり e で終わる文脈
状態 Z: { [B → c・, d] }  // b で始まり d で終わる文脈
状態 W: { [B → c・, e] }  // a で始まり e で終わる文脈
```

LALR(1) では、これらがマージされます：

マージ後の状態: {

[A → c・, d/e]

[B → c・, d/e]

}

この状態で、先読みが d または e の時、A と B のどちらで還元すべきか決定できません（reduce/reduce コンフリクト）。

5.11.16 LALR(1) と LR(1) の実用的な比較

実際のプログラミング言語の文法での比較：

特性	LR(1)	LALR(1)
状態数	数千〜数万	数百（LR(0) と同程度）
構文解析表サイズ	非常に大きい	実用的なサイズ
構築時間	遅い	高速
解析能力	最も強力	わずかに劣る
実用性	メモリ制約で困難	十分実用的

5.11.17 なぜ Yacc は LALR(1) を選んだのか

1975 年に Stephen C. Johnson が Yacc を開発した際、LALR(1) を採用した理由は以下のようなものです：

1. **メモリ制約**：当時のコンピュータでは、LR(1) の巨大な解析表は非現実的
2. **十分な表現力**：ほとんどのプログラミング言語の文法は LALR(1) で記述可能
3. **効率的な実装**：DeRemer の効率的なアルゴリズムが利用可能

5.11.18 LALR(1) の実用例

実際のプログラミング言語の構文解析では LALR(1) が使われていることが多いです：

- **C 言語**：元々 Yacc で記述された LALR(1) 文法
- **Ruby**：Bison を使用（LALR(1)）
- **PostgreSQL**：SQL 文法を Bison で解析

5.11.19 LALR(1) の限界を超えて

LALR(1) でもコンフリクトが解決できない場合には以下のような対処法があります：

1. **文法の書き換え**：多くの場合、等価な文法に変換可能
2. **GLR（Generalized LR）**：非決定的な解析を許容

3. **優先順位と結合性**：Yacc/Bison の `%left`、`%right`、`%prec`
4. **意味的な処理**：構文解析後に意味解析で解決

ちなみに、GLR は Bison などでサポートされていますが、実装が複雑になるため、一般的には LALR(1) で十分な場合が多いです。

5.11.20　まとめ：素朴な方法からの進化

1. **素朴なシフト還元**：毎回すべての規則をチェック
2. **LR(0)**：オートマトンで効率化、ただし先読みなし
3. **SLR(1)**：FOLLOW 集合で簡単な先読み
4. **LR(1)**：文脈ごとの先読みで精密化
5. **LALR(1)**：実用的なバランス

この流れを理解することで、各手法の存在意義が明確になります。

5.12　Parsing Expression Grammar(PEG) - 新しいアプローチ

2004 年に Bryan Ford が提案した **Parsing Expression Grammar（PEG）**は、全く違うアプローチを取ります。これまでの構文解析手法は大前提として、文脈自由文法を元にしていましたが、PEG は文脈自由文法に一見よく似た、それでいて異なる新たな形式文法です。異なる形式文法であるため、CFG と表現能力はことなります。

正確にいうと、PEG 自体を構文解析アルゴリズムというのは多少不適切なのですが、PEG 自体に標準的な操作的意味論（実行の方法くらいの意味）が定義されているため、ここでは PEG を構文解析アルゴリズムとして扱います。

5.12.1　PEG の基本アイデア

PEG の最大の特徴は以下の通りです：

1. 順序付き選択とバックトラック：`A / B` は「まず A を試し、失敗したら B を試す」
2. 字句解析が不要：文字列レベルで直接解析
3. いくつかの文脈依存言語を扱える：たとえば、`a^n b^n c^n` も扱える
4. 全ての LR(1) 文法を表現可能
5. 非常に単純：アルゴリズムが非常に単純で、実装が容易

5.12.2　第 3 章で実装した PEG

思い出してください。第 3 章で最初に実装した `PegJsonParser` がまさに PEG の実装でした：

```
public Ast.JsonArray parseArray() {
    int backup = cursor;
    try {
        // LBRACKET RBRACKET
        parseLBracket();
        parseRBracket();
        return new Ast.JsonArray(new ArrayList<>());
    } catch (ParseException e) {
        cursor = backup;  // バックトラック！
        // LBRACKET value {COMMA value} RBRACKET
        parseLBracket();
        // ... 省略 ...
    }
}
```

この「バックトラック」が PEG の核心です。LL にせよ LR にせよ、構文解析は「一度決めたら戻れない」ので、バックトラックができません。しかし、PEG では「失敗したら前の状態に戻って別の選択肢を試す」ことができます。LL や LR が非常に複雑なのはひとはえに「バックトラックができない」からでもあります。PEG はこのバックトラックを行うことで、非常にシンプルな構文解析を実現しています。

5.12.3　PEG の形式的定義

PEG は以下の 8 つの構成要素から成り立ちます：

1. 空文字列：ε
2. 終端記号：t
3. 非終端記号：N
4. 連接：e1 e2（e1 の後に e2）
5. 選択：e1 / e2（e1 または e2）
6. 0 回以上の繰り返し：e*
7. 肯定述語：&e（e にマッチするが消費しない）
8. 否定述語：!e（e にマッチしないことを確認）

特に 7 番と 8 番は文脈自由文法にはない、PEG 特有の機能です。

5.12.4　PEG と CFG の違い

特徴	CFG	PEG
選択演算子	|（非決定的）	/（順序付き）

特徴	CFG	PEG
曖昧性	あり得る	常に一意
字句解析	必須	不要
左再帰	問題なし（LR 系）	無限ループ
表現力	文脈自由言語	LR(k) + 一部の文脈依存言語

ちなみに、言語クラスとしてみたとき、明らかに PEG ⊆ CFG はなりたちません。それは CFG で表現できない `a^n b^n c^n` のような言語を PEG は表現できるからです。一方、CFG ⊆ PEG が成り立つかは不明（open problem）ですが、こちらも成り立たないと予想されています。何故かと言うと、以下のような明らかに不自然な状態があるからです。

- 回文を表現できない可能性が高い：`A → aAa | bAb | ε` のような文法に相当する文法を PEG で記述すると、PEG では 2 の累乗 - 2 の長さの回文という非常に不自然な表現になります。さらに、現状、PEG では回文を表現するための文法が発見されていません。
- 線形時間で解析が可能：PEG は後述する Packrat Parsing により、線形時間で解析が可能です。CFG では線形時間の解析は知られていません。

5.13 Packrat Parsing - PEG の線形時間化

PEG の最大の弱点は、バックトラックにより最悪の場合に指数関数時間がかかることです。**Packrat Parsing** は、メモ化（memoization）を使ってこの問題を解決します。

5.13.1 メモ化とは？

メモ化（memoization）は、「同じ引数で関数を呼んだら、同じ結果が返る」ことを利用して、一度計算した結果をキャッシュする技法です。

「メモ化」という名前は「memoize」（記憶する）から来ています。「メモリ化」ではなく「メモ化」であることに注意してください。

5.13.2 フィボナッチ数で学ぶメモ化

フィボナッチ数の素朴な実装を考えてみます：

```
public static long fib(long n) {
    if(n == 0) return 0L;
    if(n == 1) return 1L;
    else return fib(n - 1) + fib(n - 2);
```

```
}
```

この実装の問題点は、同じ値を何度も計算してしまうことです。メモ化を適用すると次のようになります：

```
private static Map<Long, Long> cache = new HashMap<>();

public static long fib(long n) {
    Long value = cache.get(n);
    if(value != null) return value;

    long result;
    if(n == 0) {
        result = 0L;
    } else if (n == 1) {
        result = 1L;
    } else {
        result = fib(n - 1) + fib(n - 2);
    }
    cache.put(n, result);
    return result;
}
```

メモ化の効果はこのようなケースで絶大です。具体的には：

- 時間計算量：`O(2^n)` → `O(n)`
- 空間計算量：`O(1)` → `O(n)`

メモ化により、同じ計算を何度も行わず、結果をキャッシュすることで大幅に計算量を削減できます。

5.13.3　PEG パーサのメモ化

PEG パーサにもメモ化を適用できます。単に全ての規則に対してメモ化を行ったものを **Packrat Parsing** と呼びます。Packrat Parsing では、各規則のパース結果をキャッシュし、同じ位置で同じ規則を再度パースする際にキャッシュを利用します。このようにすることで、PEG のバックトラックによる指数関数的な計算量を線形時間に抑えることができます。

Packrat Parsing の基本的な実装は以下のようになります：

```
public class PackratParser {
    private Map<Integer, Map<String, MemoEntry>> memo;

    static class MemoEntry {
        boolean success;
        int endPos;
    }

    private boolean memoized(String ruleName, Supplier<Boolean> parser) {
        // メモ化テーブルをチェック
        Map<String, MemoEntry> posCache = memo.get(pos);
        if (posCache != null) {
            MemoEntry entry = posCache.get(ruleName);
            if (entry != null) {
                // キャッシュヒット！
                pos = entry.endPos;
                return entry.success;
            }
        }

        // キャッシュミス：実際にパース
        int startPos = pos;
        boolean success = parser.get();
        int endPos = pos;

        // 結果をキャッシュに保存
        // ... 省略 ...

        return success;
    }
}
```

ここでのポイントは memo というマップを使って、各位置（pos）と規則名（ruleName）に対するパース結果をキャッシュすることです。これにより、同じ位置で同じ規則を再度パースする際に、キャッシュを利用して高速化できます。

5.13.4 Packrat Parsing の特徴

利点：

- 線形時間：O(n)（n は入力長）
- 左再帰対応：工夫により可能
- 実装が単純（単なるメモ化）

欠点：

- メモリ使用量：O(n×m)（m は文法規則数）
- キャッシュミスのオーバーヘッド
- 並列化が困難

Packrat Parsing は、PEG のバックトラックによる指数関数的な計算量を線形時間に抑えるための強力な手法です。しかし、理論上は線形時間であるものの、その代償としてメモリ使用量が大きくなるため、注意が必要です。

5.14　構文解析アルゴリズムの計算量と表現力

各アルゴリズムの計算量をまとめると：

アルゴリズム	時間計算量	空間計算量	備考
LL(1)	O(n)	O(|N| × |T|)	|N|: 非終端記号数, |T|: 終端記号数
LL(k)	O(n)	O(|N| × |T|^k)	k が大きくなると表サイズが指数的に増大
SLR(1)	O(n)	O(状態数 × (|T|+|N|))	状態数は LR(0) と同じ
LR(1)	O(n)	O(状態数 × (|T|+|N|))	状態数が多い
LALR(1)	O(n)	O(状態数 × (|T|+|N|))	状態数は LR(0) と同程度
PEG	O(2^n) (最悪)	O(n)	バックトラック用スタック
Packrat	O(n)	O(n × |P|)	|P|: 規則数

この表の「O 表記」は、アルゴリズムの計算量を表す記法で、O(n) は「入力の長さに比例した時間」、O(n^2) は「入力の長さの 2 乗に比例した時間」を意味します。

注目すべきは、PEG を除くすべての手法が線形時間で解析できることです。また、Packrat Parsing を使えば PEG も線形時間になります。

5.15　まとめ

この章では、文脈自由文法を実際に解析するための様々なアルゴリズムを学びました。

5.15.1　学んだアルゴリズムの整理

下向き構文解析（トップダウン）

- 予測的構文解析：Dyck 言語を例に、スタックを使った実装

- LL(1) 構文解析：FIRST 集合と FOLLOW 集合による効率化

上向き構文解析（ボトムアップ）

- シフト還元構文解析：葉から根に向かって構文木を構築
- LR(0)、SLR(1)、LR(1)、LALR(1)：段階的に強力になる手法

その他の手法

- PEG：順序付き選択による決定的な構文解析
- Packrat Parsing：メモ化による PEG の線形時間化

これらについて知っていることは、以下のような実践的なスキルにつながります：

1. **構文解析器生成系の選択**：各ツールの特性を理解して選択できる

- 例：ANTLR はエラーリカバリーが優れているので IDE 向き
- Yacc/Bison は高速だがエラーメッセージが分かりにくい

2. **エラーメッセージの理解**：コンフリクトの意味がわかる

- 「shift/reduce conflict」→ シフトと還元のどちらを選ぶか決まらない
- 「reduce/reduce conflict」→ 複数の還元規則から選べない

3. **DSL の設計**：パースしやすい文法設計ができる

- シンプルな設定ファイルなら LL(1) で十分
- 複雑な表現が必要なら PEG や LALR(1) を検討

4. **パフォーマンスの理解**：適切な最適化を選択できる

- 大量のデータ処理なら線形時間が保証される手法を選ぶ
- IDE のリアルタイム解析ならインクリメンタル対応を考慮

構文解析は一見難しく感じるかもしれませんが、基本的なアイデアは「文字列を構造化する」というシンプルなものです。この章で学んだ各手法の特徴を理解しておけば、実際の開発で構文解析が必要になった時に、適切な選択ができるはずです。

次章では、ここで学んだアルゴリズムを実装したパーサジェネレータ（構文解析器生成系）について具体的に見ていきます。

第 6 章

第 6 章構文解析器生成系の世界

第 5 章では現在知られている構文解析手法について、アイデアと概要を説明しました。しかし、構文解析の世界ではよく知られていることなのですが、第 5 章で説明した手法は毎回プログラマが手で実装する必要はありません。

というのは、CFG（文脈自由文法）や PEG（その類似表記も含む）によって記述された文法定義から特定の構文解析アルゴリズムを用いた構文解析器を生成する**構文解析器生成系**（Parser Generator、パーサージェネレータ）というソフトウェアがあるからです。

簡単に言えば、「文法の定義を書いたら、対応したパーサーのコードを自動生成してくれるツール」です。これにより、複雑な構文解析アルゴリズムを知らなくても、文法定義さえ書ければ高性能なパーサーを作れるのです。

構文解析器生成系でもっとも代表的なものは Yacc（ヤック）あるいはその互換実装である GNU Bison（バイソン）でしょう。Yacc は LALR(1) 法を利用した C 言語の構文解析器を生成してくれるソフトウェアです。

ちなみに「Yacc」は「Yet Another Compiler Compiler」（また別のコンパイラコンパイラ）の略で、「コンパイラを作るためのコンパイラ」という意味です。

この章では構文解析器生成系という種類のソフトウェアの背後にあるアイデアからはじまり、PEG から実際に対応する構文解析器を作る方法、多種多様な構文解析器生成系についての紹介を行います。

また、この章の最後ではある意味構文解析生成系の一種とも言える**パーサーコンビネータ**の実装方法について踏み込んで説明します。

パーサーコンビネータとは、簡単に言えば「プログラミング言語に備わったファーストクラス関数やオブジェクトを組み合わせてパーサーを構築する方法」です。通常の構文解析器生成系が「文法定義ファイル」から「Java コード」を生成するのに対し、パーサーコンビネータでは Java のメソッドを組み合わせて直接パーサーを作ります。単なる手書きパーサーと違い、パーサーコンビネータは規則や文法に対応した関数やオブジェクトがライブラリとして備わっているため、再利用性が高く、柔軟な構文解析が可能です。

構文解析器生成系を作るのは骨が折れますが、パーサーコンビネータであれば、いわゆる「ラムダ式」を持つほとんどのプログラミング言語で比較的簡単に実装できます。本書で利用している Java でも同様です。というわけで、本章を読めば皆さんもパーサーコンビネータを明日から自前で実装できるようになります。

6.1　Dyck 言語の文法と PEG による構文解析器生成

これまで何度も登場した Dyck 言語（括弧の対応が取れた文字列を表す言語）は明らかに LL(1) 法でも LR(1) 法でも PEG によっても解析可能な言語です。しかし、手書きでパーサーを実装しようとすると、退屈な繰り返しコードが多くなりがちです。

実際のところ、Dyck 言語を表現する文法があって、構文解析アルゴリズムが PEG ということまで分かれば対応する Java コードを**機械的に生成する**ことも可能そうに見えます。特に、構文解析はコード量が多いわりには退屈な繰り返しコードが多いものですから、文法から Java コードを生成できれば劇的に工数を削減できそうです。

このように「文法と構文解析手法が決まれば、後のコードは自動的に決定可能なはずだから、機械に任せてしまおう」という考え方が構文解析器生成系というソフトウェアの背後にあるアイデアです。

早速ですが、以下のように Dyck 言語を表す文法（第 4 章で示した `P -> ( P ) P | ε` に近いもの）が PEG で与えられたとして、構文解析器を生成する方法を考えてみましょう。

6.1.1　Dyck 言語の PEG

PEG の記法では、/は「順序付き選択」を表し、BNF の|とは異なり「最初の選択肢がマッチしたらそれを採用し、2 番目以降は試さない」という意味になります。

```
D <- P;
P <- "(" P ")" P / ""; // "" は空文字列ε （イプシロン）を表す
```

この PEG は P が (P)P の形であるか、または空文字列であることを示します。PEG では非終端記号の呼び出しは関数呼び出しとみなすことができますから、この定義に対応するパーサー関数（メソッド）のスケルトンを考えると、次のようなコードのイメージになります。

```
// 概念的なスケルトン
public boolean parseD() {
    return parseP(); // D は P に委譲
}

public boolean parseP() {
    // P <- "(" P ")" P / "";
    // 最初に "(" P ")" P を試す：
    //   "(" にマッチするか？
    //   マッチしたら、P を再帰的に呼び出す（1 回目）
    //   P の呼び出しが成功したら：
    //     ")" にマッチするか？
```

```
    //      マッチしたら、P を再帰的に呼び出す（2回目）
    //      P の呼び出しが成功したら：
    //        全体として成功
    // もし途中で失敗したら：
    //   バックトラックして "" を試す
    //   "" は常に成功（空文字列を消費）
    // どちらかが成功すれば parseP は成功
}
```

次の項ではこのスケルトンを実際の Java コードに変換して、Dyck 言語の構文解析器を生成する方法を見ていきます。

6.1.2 Dyck 言語の構文解析器を生成する

PEG から Java コードへの機械的な変換は、以下の規則に従います：

- **非終端記号** `N`：`parseN()` という名前のメソッドを生成
- **文字列リテラル** `str`：`match(str)` の呼び出しに変換
- **連接** `e1 e2`：`e1` の中身を再帰的に変換した結果と `e2` の中身を再帰的に変換した結果を連結
- **順序付き選択** `e1 / e2`：`e1` を試すコードと、失敗したらバックトラックするコード、`e2` を試すコードを連結する
  - ここではバックトラックのために例外を使う
- **繰り返し**：ループ構造に変換

```
public class DyckParser {
    private String input;
    private int pos;

    public static class Failure extends RuntimeException {
        public Failure(String message) {
            super(message);
        }
    }

    public DyckParser(String input) {
        this.input = input;
        this.pos = 0;
    }
```

```
    // D <- P
    public void parseD() {
        parseP();
    }

    // P <- "(" P ")" P / ""
    public void parseP() {
        // 現在位置を保存（バックトラック用）
        int saved = pos;

        try {
            match("(");  // "(" にマッチするか？
            parseP();  // P を再帰的に呼び出す（1 回目）
            match(")");  // ")" にマッチするか？
            parseP();  // P を再帰的に呼び出す（2 回目）
            return;  // 成功
        } catch (Failure e) {
            // 失敗したらバックトラック
            pos = saved;
            // 空文字列 "" を試す（常に成功）
            return;
        }
    }

    // 文字列のマッチングを行うヘルパーメソッド
    private void match(String str) {
        if (pos + str.length() <= input.length() &&
            input.startsWith(str, pos)) {
            pos += str.length();
            return;
        }
        throw new Failure("Expected '" + str + "' at position " + pos);
    }

    // パース実行メソッド
    public boolean parse() {
        parseD();
```

```
        // 例外が起きず、全ての文字が消費されていれば成功
        if (pos == input.length()) {
            return true;
        } else {
            return false;
        }
    }
}
```

このようにして、比較的シンプルに PEG から Dyck 言語の構文解析器を生成することができました。例外を使ってバックトラックを実装していますが、これは単純化のためです。Java で例外を投げるのはそこそこ重い処理なので、単純に if 文で分岐する方法を使った方が効率的です。

6.1.3 構文解析器生成系の実装

実際の構文解析器生成系では、以下のようなステップで自動生成を行います：

1. **PEG 文法の解析**：PEG 記法で書かれた文法定義自体を解析
2. **中間表現の生成**：文法規則を内部的なデータ構造に変換
3. **コード生成**：テンプレートを使って Java コードを出力

以下は中間表現が生成された後のコード生成の例です。ここでは、PEG の規則を表すクラス `Rule` と、各種 PEG 式を表す `Expr` クラスを定義し、Java コードを生成しています。

```
import java.util.*;
// 簡単な構文解析器生成系の例
public class PEG2Java {
    // 文法規則を表すレコード
    public record Rule(String name, Expr body) {}

    // PEG 式を表す抽象クラス
    public abstract static class Expr {
        abstract String generate(String indent);
    }

    // 文字列リテラル
    public static class Lit extends Expr {
        public final String value;
```

```
        public Lit(String value) {
            this.value = value;
        }
        String generate(String indent) {
            return indent + "match(\"" + value + "\");";
        }
    }

    // 非終端記号
    public static class NT extends Expr {
        public final String name;
        public NT(String name) {
            this.name = name;
        }
        // 非終端記号のコード生成
        String generate(String indent) {
            return indent + "parse" + name + "();";
        }
    }

    // 連接
    static class Seq extends Expr {
        public final List<Expr> exprs;
        public Seq(Expr... exprs) {
            this.exprs = Arrays.asList(exprs);
        }

        // 連接のコード生成。単純に N 個の式を順に生成
        String generate(String indent) {
            StringBuilder code = new StringBuilder();
            for (Expr e : exprs) {
                code.append(e.generate(indent));
                code.append("\n");
            }
            return code.toString();
        }
    }
```

```
    // 順序付き選択
    public static class Choice extends Expr {
        public final Expr alt1, alt2;
        public Choice(Expr alt1, Expr alt2) {
            this.alt1 = alt1;
            this.alt2 = alt2;
        }

        // 順序付き選択のコード生成
        // try-catch を使ってバックトラックを実装
        String generate(String indent) {
            StringBuilder code = new StringBuilder();
            code.append(indent + "int saved = pos;\n");
            code.append(indent + "try {\n");
            code.append(alt1.generate(indent + "    "));
            code.append(indent + "} catch (Failure e) {\n");
            code.append(indent + "    pos = saved;\n");
            code.append(alt2.generate(indent + "    "));
            code.append("\n");
            code.append(indent + "}\n");
            return code.toString();
        }
    }

    // コード生成のメインメソッド
    public static String generateParser(List<Rule> rules) {
        StringBuilder code = new StringBuilder();
        code.append("public class Parser {\n");
        // 例外クラスの定義
        code.append("    public static class Failure ");
        code.append("extends RuntimeException {\n");
        code.append("        public Failure(String message) {\n");
        code.append("            super(message);\n");
        code.append("        }\n");
        code.append("    }\n\n");
        // フィールドの定義
```

```
        code.append("    private String input;\n");
        code.append("    private int pos;\n");
        // コンストラクタの定義
        code.append("    public Parser(String input) {\n");
        code.append("        this.input = input;\n");
        code.append("        this.pos = 0;\n");
        code.append("    }\n\n");
        // match メソッドの定義
        code.append("    public void match(String str) {\n");
        code.append("        if (pos + str.length() <= input.length() &&\n");
        code.append("            input.startsWith(str, pos)) {\n");
        code.append("            pos += str.length();\n");
        code.append("            return;\n");
        code.append("        }\n");
        code.append(
            "        throw new Failure(\"Expected '\"" +
            " + str + \"' at pos \" + pos);\n"
        );
        code.append("    }\n\n");

        // 各規則に対してメソッドを生成
        for (Rule rule : rules) {
            code.append("    public void parse" + rule.name + "() {\n");
            code.append(rule.body.generate("        "));
            code.append("\n");
            code.append("        return;\n");
            code.append("    }\n\n");
        }

        code.append("}\n");
        return code.toString();
    }
}
```

先ほど作った Dyck 言語の PEG をちょっと変形したものをこの構文解析器生成系に与えます。利用コードは次のようになります。

```
// D <- P;
// P <- "(" P ")" P / "()";
List<Rule> rules = Arrays.asList(
    new Rule("D", new NT("P")),
    new Rule("P",
        new Choice(
            new Seq(
                new Lit("("),
                new NT("P"),
                new Lit(")"),
                new NT("P")
            ),
            new Lit("()")
         )
    )
);
// 構文解析器のコードを生成
String parserCode = PEG2Java.generateParser(rules);
System.out.println(parserCode);
```

次のような Java コードが生成されます。

```
public class Parser {
    public static class Failure extends RuntimeException {
        public Failure(String message) {
            super(message);
        }
    }
    public Parser(String input, int pos) {
        this.input = input;
        this.pos = pos;
    }

    private String input;
    private int pos;
    public void match(String str) {
        if (pos + str.length() <= input.length() &&
```

```
            input.startsWith(str, pos)) {
            pos += str.length();
            return;
        }
        throw new Failure("Expected '" + str + "' at pos " + pos);
    }

    public void parseD() {
        parseP();
        return;
    }

    public void parseP() {
        int saved = pos;
        try {
            match("(");
            parseP();
            match(")");
            parseP();
        } catch (Failure e) {
            pos = saved;
            match("()");
        }

        return;
    }
}
```

このコードは実際に Dyck 言語の変形版を解析することができます。`parseD()` メソッドを呼び出すことで、Dyck 言語の文字列が正しいかどうかをチェックできます。

このように、PEG の文法定義から機械的に Java コードを生成することができます。本体のコードだけでいえばわずか 140 行程度です。これで PEG の構文解析器生成系が実装できるのです。もちろん、実用的には抽象構文木（AST）の生成や、エラーメッセージの改善などが必要になりますが、コアの部分はこのようにシンプルに実装できます。

6.2　構文解析器生成系の分類

構文解析器生成系は 1970 年代頃から研究の蓄積があり、数多くの構文解析器生成系がこれまで開発されています。

基本的には、構文解析器生成系は採用している構文解析アルゴリズムによって分類されます。たとえば：

- JavaCC は LL(1) 構文解析器を出力するため、**LL(1) 構文解析器生成系**
- yacc/bison は LALR(1) 構文解析器を出力するため、**LALR(1) 構文解析器生成系**

一般的な構文解析器生成系の処理フローは、おおむね以下のようになります。ただし、PEG を採用している構文解析器生成系では、字句解析器の部分が丸ごと不要になります。

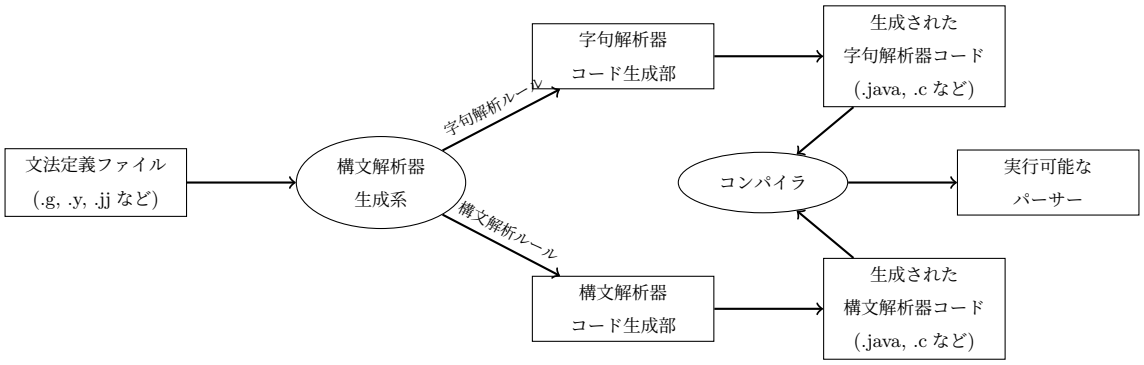


図 6.1　一般的な構文解析器生成系の処理フロー

Yacc/Lex のように、字句解析器生成系（Lex）と構文解析器生成系（Yacc）が別々のツールとして提供され、連携して動作するケースもあります。

ただし、GNU Bison は Yacc と違って、LALR(1) より広い GLR（Generalized LR：一般化 LR）構文解析器も生成できるので、GLR 構文解析器生成系であるとも言えます。GLR は、通常の LR 構文解析ではコンフリクト（シフト/還元コンフリクトなど）が発生するような曖昧な文法も扱えるように拡張された手法です。Yacc/Bison を使う場合、ほとんどは LALR(1) 構文解析器を出力するので、GLR について言及されることは少ないですが、知っておいても損はないでしょう。

より大きなくくりでみると、下向き構文解析（LL 法や PEG）と上向き構文解析（LR 法など）という観点から分類することもできますし、ともに文脈自由文法ベースである LL 法や LR 法と、PEG など他の形式言語を用いた構文解析法を対比してみせることもできます。

以下に、本書で紹介する代表的な構文解析器生成系の比較をまとめます。

特徴項目	JavaCC	Yacc/Bison	ANTLR
採用アルゴリズム	LL(k) (デフォルトは LL(1))	LALR(1) (Bison は GLR も可)	ALL(*)
生成コードの言語	Java	C, C++ (Bison は Java なども限定的にサポート)	Java, C++, Python, JavaScript, Go, C#, Swift, Dart, PHP (多言語対応)
左再帰の扱い	不可 (文法書き換えが必要)	直接左再帰を扱える	直接・間接左再帰を扱える (v4 以降)

特徴項目	JavaCC	Yacc/Bison	ANTLR
曖昧性解決	先読みトークン数 (k) の調整、意味アクション	演算子の優先順位・結合規則指定、%prec などで対応	意味アクション、構文述語、ALL(*) による自動解決
エラー報告/リカバリ機能	基本的	error トークンによる限定的なリカバリ	高度なエラー報告、柔軟なエラーリカバリ戦略
学習コスト	Java ユーザーには比較的容易	やや高め (C 言語と連携の知識も必要)	機能が豊富で強力な分、やや高め
その他特徴	Java に特化、構文が Java ライク	C 言語との親和性が高い、歴史と実績がある	強力な解析能力、豊富なターゲット言語、優れたツールサポート (GUI など)

6.3 JavaCC：Java の構文解析生成系の定番

1996 年、Sun Microsystems（当時）は、Jack という構文解析器生成系をリリースしました。その後、Jack の作者が自らの会社を立ち上げ、Jack は JavaCC に改名されて広く知られることとなりましたが、現在では紆余曲折の末、javacc.github.io の元で開発およびメンテナンスが行われています。現在のライセンスは 3 条項 BSD ライセンスです。

JavaCC という名前は「Java Compiler Compiler」の略で、yacc 同様「コンパイラを作るためのコンパイラ」という意味です。

JavaCC は LL(1) 法を元に作られており、構文定義ファイルから LL(1) パーサーを生成します。以下は四則演算を含む数式を計算できる電卓を JavaCC で書いた場合の例です。

```
options {
  STATIC = false;
  JDK_VERSION = "21";
}

PARSER_BEGIN(Calculator)
package com.github.kmizu.calculator;
public class Calculator {
  public static void main(String[] args) throws ParseException {
    Calculator parser = new Calculator(System.in);
    parser.start();
  }
}
```

```
PARSER_END(Calculator)

SKIP : { " " | "\t"  | "\r" | "\n" }
TOKEN : {
  <ADD: "+">
| <SUBTRACT: "-">
| <MULTIPLY: "*">
| <DIVIDE: "/">
| <LPAREN: "(">
| <RPAREN: ")">
| <INTEGER: (["0"-"9"])+>
}

public int expression() :
{int r = 0;}
{
  r=add() <EOF> { return r; }
}

public int add() :
{int r = 0; int v = 0;}
{
    r=mult() ( <ADD> v=mult() { r += v; }| <SUBTRACT> v=mult() { r -= v; })* {
        return r;
    }
}



public int mult() :
{int r = 0; int v = 0;}
{
   r=primary() (
     <MULTIPLY> v=primary() { r *= v; }
   | <DIVIDE> v=primary() { r /= v; })* { return r; }
}

public int primary() :
```

```
{int r = 0; Token t = null;}
{
(
  <LPAREN> r=expression() <RPAREN>
| t=<INTEGER> { r = Integer.parseInt(t.image); }
) { return r; }
}
```

この SKIP から TOKEN までの部分はトークン定義になります。トークンとは、構文解析の基本単位となる要素で、「+」「123」「(」などのことです。第 3 章の JSON パーサーでも「文字列」「数値」「中括弧」などをトークンとして扱いましたね。ここでは、7 つのトークンを定義しています。トークン定義の後が構文規則の定義になります。ここでは、

- expression()
- add()
- mult()
- primary()

の 4 つの構文規則が定義されています。各構文規則は Java のメソッドに酷似した形で記述されますが、実際、ここから生成される.java ファイルには同じ名前のメソッドが定義されます。

expression() が add() を呼び出して、add() が mult() を呼び出して、mult() が primary() を呼び出すという構図は第 2 章で既にみた形ですが、第 2 章と違って単純に宣言的に各構文規則の関係を書けばそれで OK なのが構文解析器生成系の強みです。

ちなみに、{int r = 0;}のような部分は Java コードの埋め込みで、構文解析中に実行される処理を記述しています。これを「セマンティックアクション」（意味的動作）と呼びます。セマンティックアクションを使うことで、単に文法が正しいかチェックするだけでなく、解析しながら計算や抽象構文木の構築などの処理を行うことができます。

このようにして定義した電卓プログラムは次のようにして利用することができます。

```
package com.github.kmizu.calculator;

import jdk.jfr.Description;
import org.junit.jupiter.api.Test;
import com.github.kmizu.calculator.Calculator;
import java.io.*;

import static org.junit.jupiter.api.Assertions.assertEquals;

public class CalculatorTest {
```

```
    @Test
    @Description("1 + 2 * 3 = 7")
    public void test1() throws Exception {
        Calculator calculator = new Calculator(
            new StringReader("1 + 2 * 3"))
        ;
        assertEquals(7, calculator.expression());
    }

    @Test
    @Description("(1 + 2) * 4 = 12")
    public void test2() throws Exception {
        Calculator calculator = new Calculator(
            new StringReader("(1 + 2) * 4")
        );
        assertEquals(12, calculator.expression());
    }

    @Test
    @Description("(5 * 6) - (3 + 4) = 23")
    public void test3() throws Exception {
        Calculator calculator = new Calculator(
            new StringReader("(5 * 6) - (3 + 4)")
        );
        assertEquals(23, calculator.expression());
    }
}
```

この CalculatorTest クラスでは JUnit5 を使って、JavaCC で定義した Calculator クラスの挙動をテストしています。空白や括弧を含む数式を問題なく計算できているのがわかるでしょう。

このようなケースでは先読みトークン数が 1 のため、JavaCC のデフォルトで構いませんが、定義したい構文によっては先読み数を 2 以上に増やさなければいけないこともあります。以下のようにして先読み数を増やすことができます：

```
options {
  STATIC = false;
  JDK_VERSION = "21";
```

```
  LOOKAHEAD = 2
}
```

LOOKAHEAD = 2 というオプションによって、先読みトークン数を 2 に増やしています。

先読みトークン数が 2 ということは、「次の 2 つのトークンを見て、どの規則を適用するか決定できる」という意味です。LOOKAHEAD は固定されていれば任意の正の整数にできるので、JavaCC はデフォルトでは LL(1) だが、オプションを設定することによって LL(k) になるともいえます。

JavaCC は構文定義ファイルの文法がかなり Java に似ているため、生成されるコードの形を想像しやすいというメリットがあります。JavaCC は Java の構文解析生成系の中では最古の部類の割に今でも現役で使われているのは、Java ユーザにとっての使いやすさが背景にあるように思います。

6.4 Yacc (GNU Bison)：構文解析器生成系の老舗

Yacc は 1970 年代に AT&T のベル研究所にいた Stephen C. Johnson（スティーブン・ジョンソン）によって作られたソフトウェアで、非常に歴史がある構文解析器生成系です。第 5 章で学んだ LALR(1) 法の構文解析生成系を実用化した最初のツールでもあります。

現在広く使われている GNU Bison は Yacc の GNU による再実装で、GNU プロジェクトの一部として開発されています。GNU Bison は Yacc と互換性があり、Yacc の機能を拡張したものです。

Yacc を使って、四則演算を行う電卓プログラムを作るにはまず字句解析器生成系である Lex（正確には GNU による再実装である Flex）用の定義ファイルを書く必要ががあります。Lex の定義ファイル token.l は次のようになります：

```
%{
#include "y.tab.h"
%}

%%
[0-9]+      { yylval = atoi(yytext); return NUM; }
[-+*/()]    { return yytext[0]; }
[ \t]       { /* ignore whitespce */ }
"\r\n"      { return EOL; }
"\r"        { return EOL; }
"\n"        { return EOL; }
.           { printf("Invalid character: %s\n", yytext); }
%%
```

%% から %% までがトークンの定義です。これはそれぞれ次のように読むことができます。

- `[0-9]+ { yylval = atoi(yytext); return NUM; }`

  0-9 までの数字が 1 個以上あった場合は数値として解釈し（`atoi(yytext)`）、トークン `NUM` として返します。

- `[-+*/()] { return yytext[0]; }`

  -,+,*,/,(,) についてはそれぞれの文字をそのままトークンとして返します。

- `[ \t] { /* ignore whitespce */ }`

  タブおよびスペースは単純に無視します

- `"\r\n" { return EOL; }`

  改行は EOL というトークンとして返します

- `"\r" { return EOL; }`

  前に同じ

- `"\n" { return EOL; }`

  前に同じ

- `. { printf("Invalid character: %s\n", yytext); }`

  それ以外の文字が来たらエラーを吐きます

このトークン定義ファイルを入力として与えると、flex は `lex.yy.c` という形で字句解析器を出力します。

次に、yacc の構文定義ファイルを書きます：

```
%{
#include <stdio.h>
int yylex(void);
void yyerror(char const *s);
int yywrap(void) {return 1;}
extern int yylval;
%}

%token NUM
%token EOL
%left '+' '-'
%left '*' '/'

%%
```

```
input : expr EOL
      { printf("Result: %d\n", $1); }
      ;

expr : NUM
      { $$ = $1; }
      | expr '+' expr
      { $$ = $1 + $3; }
      | expr '-' expr
      { $$ = $1 - $3; }
      | expr '*' expr
      { $$ = $1 * $3; }
      | expr '/' expr
      {
        if ($3 == 0) { yyerror("Cannot divide by zero."); }
        else { $$ = $1 / $3; }
      }
      | '(' expr ')'
      { $$ = $2; }
      ;



void yyerror(char const *s)
{
  fprintf(stderr, "Parse error: %s\n", s);
}

int main()
{
  yyparse();
}
```

flex の場合と同じく、%% から %% までが構文規則の定義の本体です。実行されるコードが入っているので読みづらくなっていますが、それを除くと以下のようになります：

```
% {
%token NUM
%token EOL
%left '+' '-'
```

```
%left '*' '/'
}

%%
input : expr EOL
      { printf("Result: %d\n", $1); }
      ;

expr : NUM
      | expr '+' expr
      | expr '-' expr
      | expr '*' expr
      | expr '/' expr
      | '(' expr ')'
      ;
%%
```

だいぶ見やすくなりましたね。入力を表す input 規則は expr EOL からなります。expr は式を表す規則ですから、その後に改行が来れば input は終了となります。

次に、expr 規則ですが、ここでは yacc の優先順位を表現するための機能である %left を使ったため、優先順位のためだけに規則を作る必要がなくなっており、定義が簡潔になっています。ともあれ、こうして定義された文法定義ファイルを yacc に与えると y.tab.c というファイルを出力します。

flex と yacc が出力したファイルを結合して実行ファイルを作ると、次のような入力に対して：

```
1 + 2 * 3
2 + 3
6 / 2
3 / 0
```

それぞれ、次のような出力を返します。

```
Result: 7
Result: 5
Result: 3
Parse error: Cannot divide by zero.
```

Yacc はとても古いソフトウェアの一つですが、Ruby の文法定義ファイル parse.y は Yacc 用ですし、未だに各種言語処理系では現役で使われてもいます。

6.5　ANTLR：多言語対応の強力な下向き構文解析生成系

1989 年に Purdue Compiler Construction set(PCCTS) というものがありました、ANTLR はその後継というべきもので、これまでに、LL(k) -> LL(*) -> ALL(*) と構文解析アルゴリズムを拡張し、取り扱える文法の幅を広げつつアクティブに開発が続けられています。作者は Terence Parr という方ですが、構文解析器一筋と言っていいくらいに、ANTLR にこれまでの時間を費やしてきている人です。

それだけに、ANTLR の完成度は非常に高いものになっています。また、一時期は LR 法に比べて LL 法の評価は低いものでしたが、Terence Parr が LL(k) を改良していく過程で、LL(*) や ALL(*) のような LR 法に比べてもなんら劣らない、実用的にも使いやすい構文解析法が発明されました。

ANTLR は Java、C++ などいくつもの言語を扱うことができますが、特に安心して使えるのは Java です。以下は先程と同様の、四則演算を解析できる数式パーサーを ANTLR で書いた場合の例です。

ANTLR では構文規則は、**規則名 : 本体 ;** という形で記述しますが、LL パーサー向けの構文定義を素直に書き下すだけで OK です。

```
grammar Expression;

expression returns [int e]
    : v=additive {$e = $v.e;}
    ;

additive returns [int e = 0;]
    : l=multitive {$e = $l.e;} (
      '+' r=multitive {$e = $e + $r.e;}
    | '-' r=multitive {$e = $e - $r.e;}
    )*
    ;

multitive returns [int e = 0;]
    : l=primary {$e = $l.e;} (
      '*' r=primary {$e = $e * $r.e;}
    | '/' r=primary {$e = $e / $r.e;}
    )*
    ;

primary returns [int e]
```

```
    : n=NUMBER {$e = Integer.parseInt($n.getText());}
    | '(' x=expression ')' {$e = $x.e;}
    ;

LP : '(' ;
RP : ')' ;
NUMBER : INT ;
fragment INT :   '0' | [1-9] [0-9]* ; // no leading zeros
WS  :   [ \t\n\r]+ -> skip ;
```

規則 expression が数式を表す規則です。そのあとに続く returns [int e] はこの規則を使って解析を行った場合に int 型の値を返すことを意味しています。これまで見てきたように構文解析をした後には抽象構文木をはじめとして何らかのデータ構造を返す必要があります。returns ... はそのために用意されている構文です。名前が全て大文字の規則はトークンを表しています。

数式を表す各規則についてはこれまで書いてきた構文解析器と同じ構造なので読むのに苦労はしないでしょう。

規則 WS は空白文字を表すトークンですが、これは数式を解析する上では読み飛ばす必要があります。[\t\n\r]+ -> skip は

- スペース
- タブ文字
- 改行文字

のいずれかが出現した場合は読み飛ばすということを表現しています。

ANTLR は下向き型の構文解析が苦手とする左再帰もある程度扱うことができます。先程の定義ファイルでは繰り返しを使っていましたが、これを左再帰に直した以下の定義ファイルも全く同じ挙動をします。

```
grammar LRExpression;

expression returns [int e]
    : v=additive {$e = $v.e;}
    ;

additive returns [int e]
    : l=additive op='+' r=multitive {$e = $l.e + $r.e;}
    | l=additive op='-' r=multitive {$e = $l.e - $r.e;}
    | v=multitive {$e = $v.e;}
    ;
```

```
multitive returns [int e]
    : l=multitive op='*' r=primary {$e = $l.e * $r.e;}
    | l=multitive op='/' r=primary {$e = $l.e / $r.e;}
    | v=primary {$e = $v.e;}
    ;

primary returns [int e]
    : n=NUMBER {$e = Integer.parseInt($n.getText());}
    | '(' x=expression ')' {$e = $x.e;}
    ;

LP : '(' ;
RP : ')' ;
NUMBER : INT ;
fragment INT :   '0' | [1-9] [0-9]* ; // no leading zeros
WS  :   [ \t\n\r]+ -> skip ;
```

左再帰を使うことでより簡単に文法を定義できることもあるので、あると嬉しい機能だと言えます。

さらに、ANTLR は ALL(*) というアルゴリズムを採用しているため、通常の LL パーサーでは扱えないような文法定義も取り扱うことができます。以下の「最小 XML」文法定義を見てみましょう。

```
grammar PetitXML;
@parser::header {
    import static com.github.asciidwango.parser_book.ch5.PetitXML.*;
    import java.util.*;
}

root returns [Element e]
    : v=element {$e = $v.e;}
    ;

element returns [Element e]
    : ('<' begin=NAME '>' es=elements '</' end=NAME '>' {
          $begin.text.equals($end.text)
      }?
      {$e = new Element($begin.text, $es.es);})
```

```
    | ('<' name=NAME '/>' {$e = new Element($name.text);})
    ;

elements returns [List<Element> es]
    : { $es = new ArrayList<>();} (element {$es.add($element.e);})*
    ;

LT:    '<';
GT:    '>';
SLASH: '/';
NAME:  [a-zA-Z_][a-zA-Z0-9]* ;

WS :   [ \t\n\r]+ -> skip ;
```

PeitXML の名の通り、属性やテキストなどは全く扱うことができず、<a>や<a/>、<a>といった要素のみを扱うことができます。規則 element が重要です。

```
element returns [Element e]
    : ('<' begin=NAME '>' es=elements '</' end=NAME '>' {
            $begin.text.equals($end.text)
       }?
      {$e = new Element($begin.text, $es.es);})
    | ('<' name=NAME '/>' {$e = new Element($name.text);})
    ;
```

ここで空要素（<e/>など）に分岐するか、子要素を持つ要素（<a>など）に分岐するかを決定するには、<に加えて、任意の長さになり得るタグ名を先読みしなければいけません。通常の LL パーサーでは何文字（何トークン）先読みしているのは予め決定されているのでこのような文法定義を取り扱うことはできません。しかし、ANTLR の ALL(*) アルゴリズムとその前身となる LL(*) アルゴリズムでは任意個の文字数を先読みして分岐を決定することができます。

ANTLR では通常の LL パーサーと違い文法を記述する上での大きな制約がなく、非常に強力です。理論的な意味でも ALL(*) アルゴリズムは任意の決定的な文脈自由言語を取り扱うことができます。

また、ALL(*) アルゴリズム自体とは関係ありませんが、XML のパーサーを書くときには開きタグと閉じタグの名前が一致している必要があります。この条件を記述するために PetitXML では次のように記述されています。

```
'<' begin=NAME '>' es=elements '</' end=NAME '>' {
  $begin.text.equals($end.text)
}?
```

この中の`{$begin.text.equals($end.text)}?`という部分は semantic predicate（意味的述語）と呼ばれ、プログラムとして書かれた条件式が真になるときにだけマッチします。この例では、開きタグの名前（`begin`）と閉じタグの名前（`end`）が一致しているかを Java のコードで確認しています。semantic predicate のような機能はプログラミング言語をそのまま埋め込むという意味で、正直「あまり綺麗ではない」と思わなくもないですが、実用上は semantic predicate を使いたくなる場面にしばしば遭遇します。

ANTLR はこういった実用上重要な痒いところにも手が届くように作られており、非常によくできた構文解析機生成系といえるでしょう。

6.6 SComb

手前味噌ですが、拙作のパーサーコンビネータである SComb も紹介しておきます。これまで紹介してきたものはすべて構文解析器生成系です。つまり、独自の言語を用いて作りたい言語の文法を記述し、そこから**対象言語**（C であったり Java であったり様々ですが）で書かれた構文解析器を生成するものだったわけですが、パーサーコンビネータは少々違います。

パーサーコンビネータでは対象言語のメソッドや関数、オブジェクトとして構文解析器を定義し、演算子やメソッドによって構文解析器を組み合わせることで構文解析器を組み立てていきます。「コンビネータ」という名前は「組み合わせる（combine）」から来ており、小さなパーサーを組み合わせて大きなパーサーを作るという意味です。パーサーコンビネータではメソッドや関数、オブジェクトとして規則自体を記述するため、特別にプラグインを作らなくても IDE による支援が受けられることや、対象言語が静的型システムを持っていた場合、型チェックによる支援を受けられることがメリットとして挙げられます。

SComb で四則演算を解析できるプログラムを書くと以下のようになります。先程述べたように SComb はパーサーコンビネータであり、これ自体が Scala のプログラム（`object` 宣言）でもあります。

```
object Calculator extends SCombinator {
  // root <- E
  def root: Parser[Int] = E

  // E <- A
  def E: Parser[Int] = rule(A)

  // A <- M ("+" M / "-" M)*
  def A: Parser[Int] = rule(chainl(M) {
```

```
    $("+").map { op => (lhs: Int, rhs: Int) => lhs + rhs } |
    $("-").map { op => (lhs: Int, rhs: Int) => lhs - rhs }
  })

  // M <- P ("*" P / "/" P)*
  def M: Parser[Int] = rule(chainl(P) {
    $("*").map { op => (lhs: Int, rhs: Int) => lhs * rhs } |
    $("/").map { op => (lhs: Int, rhs: Int) => lhs / rhs }
  })

  // P <- "(" E ")" | N
  def P: Parser[Int] = rule{
    (for {
      _ <- string("("); e <- E; _ <- string(")")} yield e) | N
  }

  // N <- [0-9]+
  def N: P[Int] = rule(set('0'to'9').+.map{ digits => digits.mkString.toInt})

  def parse(input: String): Result[Int] = parse(root, input)
}
```

Scala は Java とは異なるプログラミング言語ですが、JVM 上で動く言語なので、Java ユーザーの方でも比較的理解しやすいと思います。

各メソッドに対応する BNF による規則をコメントとして付加してみましたが、BNF と比較しても簡潔に記述できているのがわかります。Scala は記号をそのままメソッドとして記述できるなど、元々 DSL（ドメイン特化言語）に向いている特徴を持った言語なのですが、その特徴を活用しています。`chainl` というメソッドについてだけは見慣れない読者の方は多そうですが、これは

```
// M ::= P ("+" P | "-" P)*
```

のような二項演算を簡潔に記述するためのコンビネータ（メソッド）です。パーサーコンビネータの別のメリットとして、BNF（あるいは PEG）に無いような演算子をこのように後付で導入できることも挙げられます。構文規則からの値（意味値）の取り出しも Scala の for 式（Java の for 文とは異なり、値を生成する式です）を用いて簡潔に記述できています。

筆者は自作言語 Klassic の処理系作成のために SComb を使っていますが、かなり複雑な文法を記述できるにも関わらず、SComb のコア部分はわずか 600 行ほどです。それでいて高い拡張性や簡潔な記述が可能なのは、Scala という言語の能力と、SComb がベースとして利用している PEG という手法のシンプルさがあってのものだと言えるで

しょう。

6.7　パーサーコンビネータ JComb を自作しよう！

コンパイラについて解説した本は数えきれないほどありますし、その中で構文解析アルゴリズムについて説明した本も少なからずあります。しかし、構文解析アルゴリズムについてのみフォーカスした本は Parsing Techniques ほぼ一冊といえる現状です。その上でパーサーコンビネータの自作まで踏み込んだ書籍はほぼ皆無と言っていいでしょう。読者の方には「さすがにちょっとパーサーコンビネータの自作は無理があるのでは」と思われた方もいるのではないでしょうか。

しかし、驚くべきことに、現代的な言語であればパーサーコンビネータを自作するのは本当に簡単です。きっと、多くの読者の方々が拍子抜けしてしまうくらいに。この節では Java で書かれたパーサーコンビネータ JComb を自作する過程を通じて皆さんにパーサーコンビネータとはどのようなものかを学んでいただきます。

パーサーコンビネータと構文解析器生成系の関係は、次のように考えることができます：

- 構文解析器生成系：文法定義ファイル → ツール → Java コード
- パーサーコンビネータ：Java コードで直接文法を表現

つまり、パーサーコンビネータは「生成」のステップを省き、プログラミング言語の機能を活用して直接パーサーを組み立てる手法なのです。

まず復習になりますが、構文解析器というのは文字列を入力として受け取って、解析結果を返す関数（あるいはオブジェクト）とみなせるのでした。これはパーサーコンビネータ、特に PEG を使ったパーサーコンビネータを実装するときに有用な見方です。この「構文解析器はオブジェクトである」を文字通りとって、以下のようなジェネリックなインタフェース `JParser<R>`を定義します。

```
interface JParser<R> {
  Result<R> parse(String input);
}
```

ここで構文解析器を表現するインタフェース `JParser<R>`は型パラメータ `R` を受け取ることに注意してください。Java の型パラメータ（ジェネリクス）は、「このインタフェースは何かの型 `R` に対して動作しますが、具体的な型は使用時に決まります」という意味です。

一般に構文解析の結果は抽象構文木になりますが、インタフェースを定義する時点では抽象構文木がどのような形になるかはわかりようがないので、型パラメータにしておくのです。`JParser<R>`はたった一つのメソッド `parse()` を持ちます。`parse()` は入力文字列 `input` を受け取り、解析結果を `Result<R>`として返します。

`JParser<R>`の実装は一体全体どのようなものになるの？ という疑問を脇に置いておけば理解は難しくないでしょう。次に解析結果 `Result<V>`をレコードとして定義します。

```
record Result<V>(V value, String rest){}
```

レコード Result<V>は解析結果を保持するクラスです。value は解析結果の値を表現し、rest は解析した結果「残った」文字列を表します。たとえば、"123abc"という文字列から数値部分"123"を解析した場合、value は 123（整数値）、rest は"abc"（残りの文字列）となります。

このインタフェース JParser<R>は次のように使えると理想的です。

```
JParser<Integer> calculator = ...;
Result<Integer> result = calculator.parse("1+2*3");
assert 7 == result.value();
```

パーサーコンビネータは、このようなどこか都合の良い JParser<R>を、BNF（あるいは PEG）に近い文法規則を連ねていくのに近い使い勝手で構築するための技法です。前の節で紹介した SComb もパーサーコンビネータでしたが基本的には同じようなものです。

この節では最終的に上のような式を解析できるパーサーコンビネータを作るのが目標です。

6.7.1 部品を考えよう

これからパーサーコンビネータを作っていくわけですが、パーサーコンビネータの基本となる「部品」を作る必要があります。

まず最初に、文字列リテラルを受け取ってそれを解析できる次のような string() メソッドは是非とも欲しいところです。

```
assert new Result<String>("123", "").equals(string("123").parse("123"));
```

これは BNF で言えば文字列リテラルの表記に相当します。

次に、解析に成功したとしてその値を別の値に変換するための方法もほしいところです。たとえば、123 という文字列を解析したとして、これは最終的に文字列ではなく int に変換したいところです。Java のラムダ式（無名関数）を使えば、このような変換を簡潔に書けます。このようなメソッドは、ラムダ式で変換を定義できるように、次のような map() メソッドとして提供したいところです。

```
<T, U> JParser<U> map(JParser<T> parser, Function<T, U> function);
assert (new Result<Integer>(123, "")).equals(
    map(string("123"), v -> Integer.parseInt(v)).parse("123")
);
```

これは構文解析器生成系でセマンティックアクションを書くのに相当すると言えるでしょう。つまり、単に文法を

チェックするだけでなく、解析結果を使って何か計算をしたり、データ構造を構築したりする部分です。

BNF で `a | b`、つまり選択を書くのに相当するメソッドも必要です。これは次のような `alt()` メソッドとして提供します。

```
<T> JParser<T> alt(JParser<T> p1, JParser<T> p2);
assert (new Result<String>("bar", "")).equals(
    alt(string("foo"), string("bar")).parse("bar")
);
```

同様に、BNF で `a b`、つまり連接を書くのに相当するメソッドも必要ですが、これは次のような `seq()` メソッドとして提供します。

```
record Pair<A, B>(A a, B b){}
<A, B> JParser<Pair<A, B>> seq(JParser<A> p1, JParser<B> p2);
assert (new Result<>(new Pair<>("foo", "bar"), "")).equals(
    seq(string("foo"), string("bar")).parse("foobar")
);
```

最後に、BNF で `a*`、つまり 0 回以上の繰り返しに相当する `rep0()` メソッド

```
<T> JParser<List<T>> rep0(JParser<T> p);
```

や `a+`、つまり 1 回以上の繰り返しに相当する `rep1()` メソッドもほしいところです。

```
<T> JParser<List<T>> rep1(JParser<T> p);
assert (new Result<List<String>>(List.of("a", "a", "a"), "")).equals(
  rep1(string("a")).parse("aaa")
);
```

この節ではこれらのプリミティブなメソッドの実装方法について説明していきます。

6.7.2 `string()` メソッド

まず最初に `string(String literal)` メソッドで返す `JLiteralParser<String>`の中身を作ってみましょう。`JLiteralParser` クラスはただ一つのメソッド `parser()` をもつので次のような実装になります。

```
class JLiteralParser implements JParser<String> {
  private String literal;
  public JLiteralParser(String literal) {
```

```
    this.literal = literal;

  }
  public Result<String> parse(String input) {
    if(input.startsWith(literal)) {
      return new Result<String>(literal, input.substring(literal.length()));
    } else {
      return null;
    }
  }
}
```

このクラスは次のようにして使います。

```
assert new Result<String>("foo", "").equals(new JLiteralParser("foo").parse("foo"));
```

リテラルを表すフィールド literal が input の先頭とマッチした場合、literal と残りの文字列からなる Result<String>を返します。そうでない場合は返すべき Result がないので null を返します。簡単ですね。

startsWith メソッドは、文字列がある文字列で始まるかを判定する Java の標準メソッドです。substring メソッドは、文字列の一部を切り出すメソッドです。

あとはこのクラスのインスタンスを返す string() メソッドを作成するだけです。なお、使うときの利便性のため、以降では各種メソッドはクラス JComb の static メソッドとして実装していきます。

```
public class JComb {
  public static JParser<String> string(String literal) {
    return new JLiteralParser(literal);
  }
}
```

使う時は次のようになります。

```
JParser<String> foo = string("foo");
assert new Result<String>("foo", "_bar").equals(foo.parse("foo_bar"));
assert null == foo.parse("baz");
```

6.7.3 alt() メソッド

次に二つのパーサーを取って「選択」パーサーを返すメソッド alt() を実装します。先程のようにクラスを実装してもいいですが、メソッドは一つだけなのでラムダ式（Java 8 から導入された無名関数）にします。

```
public class JComb {
    // p1 / p2（PEG の順序付き選択に対応）
    public static <A> JParser<A> alt(JParser<A> p1, JParser<A> p2) {
        return (input) -> {
            var result = p1.parse(input);//(1) p1 を試す
            if(result != null) return result;//(2) p1 が成功したらその結果を返す
            return p2.parse(input);//(3) p1 が失敗したら p2 を試す
        };
    }
}
```

ラムダ式について復習しておくと、これは実質的には以下のような匿名クラスを書いたのと同じになります。

```
public class JComb {
    public static <A> JParser<A> alt(JParser<A> p1, JParser<A> p2) {
        return new JAltParser<A>(p1, p2);
    }
}

class JAltParser<A> implements JParser<A> {
  private JParser<A> p1, p2;
  public JAltParser(JParser<A> p1, JParser<A> p2) {
    this.p1 = p1;
    this.p2 = p2;
  }
  public Result<A> parse(String input) {
    var result = p1.parse(input);
    if(result != null) return result;
    return p2.parse(input);
  }
}
```

この定義では、

1. まずパーサー p1 を試す
2. p1 が成功した場合は p2 を試すことなく値をそのまま返す
3. p1 が失敗した場合、p2 を試しその値を返す

という挙動をします。これは BNF の選択 | とは異なり、PEG の「順序付き選択 /」に対応します。

実は今ここで作っているパーサーコンビネータである JComb は（SComb と同様に）PEG をベースとしたパーサーコンビネータだったのです。もちろん、PEG ベースでないパーサーコンビネータを作ることも出来るのですが実装がかなり複雑になってしまいます。PEG の挙動をそのままプログラミング言語に当てはめるのは非常に簡単であるため、今回は PEG を採用しましたが、もし興味があれば BNF ベース（文脈自由文法ベース）のパーサーコンビネータも作ってみてください。

6.7.4 seq() メソッド

次に二つのパーサーを取って「連接」パーサーを返すメソッド seq() を実装します。これは PEG の連接 e1 e2 に対応します。先程と同じくラムダ式にしてみます。

```
record Pair<A, B>(A a, B b){}
// p1 p2 (PEG の連接に対応)
public class JComb {
    public static <A, B> JParser<Pair<A, B>> seq(
        JParser<A> p1, JParser<B> p2) {
        return (input) -> {
            var result1 = p1.parse(input); //(1-1) p1 を試す
            if(result1 == null) return null; //(1-2) p1 が失敗したら全体も失敗
            var result2 = p2.parse(result1.rest()); //(2-1) p1 の残り入力で p2 を試す
            if(result2 == null) return null; //(2-2) p2 が失敗したら全体も失敗
            // 両方成功したら、結果をペアにして返す
            return new Result<>(
                new Pair<A, B>(
                    result1.value(), result2.value()
                ), result2.rest()
            );//(2-3)
        };
    }
}
```

先程の alt() メソッドと似通った実装ですが、p1 が失敗したら全体が失敗する（1-2）のがポイントです。p1 と p2 の両方が成功した場合は、二つの値のペアを返しています（2-3）。

6.7.4.1 rep0(), rep1() メソッド

残りは 0 回以上の繰り返し（PEG の p*）を表す rep0() と 1 回以上の繰り返し（PEG の p+）を表す rep1() メソッドです。

まず、rep0() メソッドは次のようになります。

```
public class JComb {
    public static <A> JParser<List<A>> rep0(JParser<A> p) {
        return (input) -> {
            var result = p.parse(input); // (1)
            if(result == null) return new Result<>(List.of(), input); // (2)
            var value = result.value();
            var rest = result.rest();
            var result2 = rep0(p).parse(rest); //(3)
            if(result2 == null) return new Result<>(List.of(value), rest);
            List<A> values = new ArrayList<>();
            values.add(value);
            values.addAll(result2.value());
            return new Result<>(values, result2.rest());
        };
    }
}
```

(1) でまずパーサー p を適用しています。ここで失敗した場合、0 回の繰り返しにマッチしたことになるので、空リストからなる結果を返します（(2)）。そうでなければ、1 回以上の繰り返しにマッチしたことになるので、繰り返し同じ処理をする必要がありますが、これは再帰呼出しによって簡単に実装できます（(3)）。シンプルな実装ですね。

rep1(p) は意味的には seq(p, rep0(p)) なので、次のようにして実装を簡略化することができます。

```
public class JComb {
    public static <A> JParser<List<A>> rep1(JParser<A> p) {
        JParser<Pair<A, List<A>>> rep1Sugar = seq(p, rep0(p));
        return (input) -> {
            var result = rep1Sugar.parse(input);//(1)
            if(result == null) return null;//(2)
            var pairValue = result.value(); // result から値を取得
            var values = new ArrayList<A>();
            values.add(pairValue.a()); // Pair の最初の要素
```

```
            values.addAll(pairValue.b()); // Pairの２番目の要素（List）
            return new Result<>(values, result.rest()); //(3)
        };
    }
}
```

実質的な本体は (1) だけで、あとは結果の値が Pair なのを List に加工しているだけですね。

6.7.5 map() メソッド

パーサーを加工して別の値を生成するためのメソッド map() を実装してみましょう。map() は JParser<R>のメソッドとして実装するとメソッドチェインが使えて便利なので、インタフェースの default メソッドとして実装します。

```
interface JParser<R> {
    Result<R> parse(String input);

    default <T> JParser<T> map(Function<R, T> f) {
        return (input) -> {
            var result = this.parse(input);
            if (result == null) return null;
            return new Result<>(f.apply(result.value()), result.rest()); (1)
        };
    }
}
```

(1) で f.apply(result.value()) として値を加工しているのがポイントです。

6.7.6 lazy() メソッド

パーサーを遅延評価するためのメソッド lazy() も導入します。

```
public class JComb {
    public static <A> JParser<A> lazy(Supplier<JParser<A>> supplier) {
        return (input) -> supplier.get().parse(input);
    }
}
```

lazy メソッドの必要性について補足します。Java は先行評価（関数が呼ばれたらすぐに評価）を行う言語であるため、再帰的な文法規則を直接メソッド呼び出しで表現しようとすると、パーサーオブジェクトの構築時に無限再帰が

発生し StackOverflowError となることがあります。例えば、算術式の文法で expression が additive を呼び出し、additive が primary を呼び出し、primary が括弧表現の中で再び expression を呼び出すような相互再帰構造を考えてみましょう。

```
// JComb を使った算術式のパーサ定義（簡略版・lazy なしのイメージ）
// 仮に直接代入しようとすると...
public static JParser<Integer> expression() {
    return additive();
}
public static JParser<Integer> additive() {
    return primary();
}
public static JParser<Integer> primary() {
    return alt(
        number,
        seq(string("("), expression(), string(")"))
    );
}
```

上記のように単純にメソッド呼び出しでパーサーを組み合わせようとすると、expression の初期化時に additive が必要になり、その additive の初期化に primary が、さらにその primary の初期化に expression が必要となり、循環参照によって初期化が終わらなくなります。

lazy は Supplier<JParser<A>> を引数に取ることで、実際の JParser<A> オブジェクトの取得（supplier.get()）を、そのパーサが実際に parse() メソッドで使われるときまで遅延させます。Supplier は Java の関数型インタフェースで、「引数なしで値を返す関数」を表します。これにより、相互再帰するパーサ定義でも、オブジェクト構築時の無限再帰を避けることができます。算術式の例では、expression の定義内で additive を呼び出す部分を lazy(() -> additive()) のように記述することで、この問題を解決します。

6.7.7 regex() メソッド

せっかくなので正規表現を扱うメソッド regex() も導入してみましょう。

```
public class JComb {
    public static JParser<String> regex(String regex) {
        return (input) -> {
            var matcher = Pattern.compile(regex).matcher(input);
            if(matcher.lookingAt()) {
                return new Result<>(
```

```
                    matcher.group(), input.substring(matcher.end())
                );
            } else {
                return null;
            }
        };
    }
}
```

引数として与えられた文字列を Pattern.compile() で正規表現に変換して、マッチングを行うだけです。matcher.lookingAt() は文字列の先頭から正規表現にマッチする部分があるかを確認するメソッドです。これは次のようにして使うことができます。

```
var number = regex("[0-9]+").map(v -> Integer.parseInt(v));
assert (new Result<Integer>(10, "")).equals(number.parse("10"));
```

6.7.8 算術式のインタプリタを書いてみる

ここまでで作ったクラス JComb と JParser などを使っていよいよ簡単な算術式のインタプリタを書いてみましょう。仕様は次の通りです。

- 扱える数値は整数のみ
- 演算子は加減乗除（+|-|*|/）のみ
- () によるグルーピングができる

実装だけを提示すると次のようになります。

```
public class Calculator {
    // expression は加減算を担当（左結合）
    // PEG: expression <- multitive ( ( "+" / "-" ) multitive )*
    public static JParser<Integer> expression() {
        return seq( // multitive と (( "+" / "-" ) multitive )* の連接
            lazy(() -> multitive()), // 左辺の multitive（乗除の項）
            rep0( // 0回以上の繰り返し
                seq( // ( "+" / "-" ) と multitive の連接
                    // "+" または "-"
                    alt(string("+"), string("-")),
                    lazy(() -> multitive()) // 右辺の multitive
                )
```

```
            )
        ).map(p -> { // 解析結果を処理するラムダ式
            var left = p.a(); // 初期値（最初の項）
            var rights = p.b(); // 残りの演算子と項のペアのリスト
            for (var rightPair : rights) { // 各 Pair<String, Integer> について
                var op = rightPair.a(); // 演算子（"+" または "-"）
                var rightValue = rightPair.b(); // 項の値
                if (op.equals("+")) {
                    left += rightValue;
                } else { // op.equals("-")
                    left -= rightValue;
                }
            }
            return left; // 計算結果
        });
    }

    // multitive は乗除算を担当（左結合）
    // PEG: multitive <- primary ( ( "*" / "/" ) primary )*
    public static JParser<Integer> multitive() {
        return seq( // primary と ( ( "*" / "/" ) primary )* の連接
            lazy(() -> primary()), // 左辺の primary
            rep0( // 0回以上の繰り返し
                seq( // ( "*" / "/" ) と primary の連接
                    alt(string("*"), string("/")), // "*" または "/"
                    lazy(() -> primary()) // 右辺の primary
                )
            )
        ).map(p -> { // 解析結果を処理するラムダ式
            var left = p.a(); // 初期値（最初の因子）
            var rights = p.b(); // 残りの演算子と因子のペアのリスト
            for (var rightPair : rights) {
                var op = rightPair.a(); // 演算子（"*" または "/"）
                var rightValue = rightPair.b(); // 因子の値
                if (op.equals("*")) {
                    left *= rightValue;
                } else { // op.equals("/")
```

```
                    if (rightValue == 0)
                        throw new ArithmeticException("Division by zero");
                    left /= rightValue;
                }
            }
            return left; // 計算結果
        });
    }

    // primary <- number / "(" expression ")"
    public static JParser<Integer> primary() {
        return alt( // number または "(" expression ")" の選択
            number, // 数値パーサ
            seq(
                string("("), // 開き括弧
                // 括弧内の式（expression を再帰呼び出し）
                lazy(() -> expression())
            ).flatMap(p1 ->
                // p1 は Pair<String, Integer>型 ("(" と expression の結果)
                seq(
                    p1.b(), // expression の結果（Integer）を次の seq の左側にする
                    string(")")  // 閉じ括弧
                // p2は Pair<Integer, String>型、その最初の要素（Integer）を返す
                ).map(p2 -> p2.a())
            );
        );
    }

    // number <- [0-9]+ （PEG の正規表現リテラルに対応）
    private static JParser<Integer> number =
        regex("[0-9]+").map(Integer::parseInt);
}
```

表記は冗長なもののほぼ PEG に一対一に対応しているのがわかるのではないでしょうか？‘

これに対して JUnit を使って以下のようなテストコードを記述してみます。無事、意図通りに解釈されていることがわかります。

```
assertEquals(
    new Result<>(7, ""), Calculator.expression().parse("1+2*3")
); // テストをパス（実際には multitive で処理される）
assertEquals(
    new Result<>(0, ""), Calculator.expression().parse("1+2-3")
); // テストをパス
```

6.7.9 自作パーサーコンビネータのススメ

DSL（ドメイン特化言語）に向いた Scala に比べれば冗長になったものの、手書きで再帰下降パーサーを組み立てるのに比べると大幅に簡潔な記述を実現することができました。しかも、JComb 全体を通しても 500 行にすら満たないのは特筆すべきところです。Java がユーザ定義の中置演算子（+ や*のような演算子を自分で定義できる機能）をサポートしていればもっと簡潔にできたのですが、そこは向き不向きといったところでしょうか。

パーサーコンビネータを使うと、手書きでパーサーを書いたり、あるいは、対象言語に構文解析器生成系がないようなケースでも、比較的気軽にパーサーを組み立てるための DSL（Domain Specific Language）を定義できるのです。また、それだけでなく、特に Java のような静的型付き言語を使った場合ですが、IDE による支援も受けられますし、BNF や PEG にはない便利な演算子を自分で導入することもできます。

パーサーコンビネータはお手軽なだけあって各種プログラミング言語に実装されています。たとえば、Java 用なら jparsec があります。しかし、筆者としては、パーサーコンビネータが動作する仕組みを理解するために、是非とも**自分だけの**パーサーコンビネータを実装してみてほしいと思います。

6.8 まとめ

この章では、構文解析器生成系（パーサージェネレータ）という、文法定義から構文解析器を自動生成するソフトウェアについて学びました。

まず、PEG から構文解析器を生成する基本的な仕組みを、Dyck 言語を例に具体的に見ていきました。文法規則を機械的に Java コードに変換することで、手書きの煩雑さから解放され、文法定義に集中できることを確認しました。

次に、代表的な構文解析器生成系として以下を紹介しました：

- **JavaCC**：LL(k) 法を採用し、Java に特化した構文が Java ユーザーに親しみやすい
- **Yacc/Bison**：LALR(1) 法を採用する歴史ある構文解析器生成系で、C 言語との親和性が高い
- **ANTLR**：ALL(*) アルゴリズムにより強力な解析能力を持ち、左再帰も扱える多言語対応ツール

最後に、パーサーコンビネータという、プログラミング言語の機能を活用して直接パーサーを組み立てる手法を学び、実際に Java でパーサーコンビネータ「JComb」を実装しました。わずか 500 行に満たないコードで、PEG に対応した実用的なパーサーコンビネータを作ることができることを示しました。

構文解析器生成系とパーサーコンビネータは、それぞれ異なる強みを持ちます。構文解析器生成系は宣言的で可読性

が高く、パーサーコンビネータはプログラミング言語の型システムや IDE の支援を受けられ、柔軟な拡張が可能です。どちらを選ぶかは、プロジェクトの要件や開発チームの好みによって決めることになるでしょう。

第 7 章

第 7 章現実の構文解析

ここまでで、LL 法や LR 法、Packrat Parsing といった、これまでに知られているメジャーな構文解析アルゴリズムを一通り取り上げてきました。これらの構文解析アルゴリズムは概ね文脈自由言語あるいはそのサブセットを取り扱うことができ、一般的なプログラミング言語の構文解析を行うのに必要十分な能力を持っているように思えます。

しかし、構文解析を専門としている人や実用的な構文解析器を書いている人は直感的に理解していることなのですが、実のところ、既存の構文解析アルゴリズムだけではうまく取り扱えない類の構文があります。一言でいうと、それらの構文は文脈自由言語から逸脱しているために、文脈自由言語を取り扱う既存の手法だけではうまくいかないのです。

このような、既存の構文解析アルゴリズムだけでは扱えない要素は多数あります。たとえば、C の typedef はその典型ですし、Ruby や Perl のヒアドキュメントと呼ばれる構文もそうです。他には、Scala のプレースホルダ構文や C++ のテンプレート、Python のインデント文法など、文脈自由言語を逸脱しているがゆえに人間が特別に配慮しなければいけない構文は多く見かけられます。

また、これまでの章では、主に構文解析を行う手法を取り扱っていましたが、現実問題としては抽象構文木をうまく作る方法やエラーメッセージを適切に出す方法も重要になってきます。

この章では、巷の書籍ではあまり扱われない、しかし現実の構文解析では対処しなくてはならない構文や問題について取り上げます。皆さんが何かしらの構文解析器を作るとき、やはり理想どおりにはいかないことが多いと思います。そのような現実の構文解析の**泥臭さ**に遭遇する読者の方々の助けになれば幸いです。

7.1　トークンが「文脈自由」なケース

最近の多くの言語は文字列補間 (String Interpolation) と呼ばれる機能を持っています。

たとえば、Ruby では以下の文字列を評価すると、`"x + y = x + y"`ではなく`"x + y = 3"`になります。

```
x = 1
y = 2
"x + y = #{x + y}" # "x + y = 3"
```

#{と}で囲まれた範囲を Ruby の式として評価した結果を文字列として埋め込んでくれるわけです。

Scala でも同じことを次のように書くことができます。

```
val x = 1; val y = 2
s"x + y = ${x + y}" // "x + y = 3"
```

Swift だと次のようになります。

```
let x = 1
let y = 2
"x + y = \(x + y)"
```

同様の機能は Kotlin、Python、JavaScript、TypeScript など様々な言語に採用されています。比較的新しい言語や、既存言語の新機能として採用するのが普通になった機能と言えるでしょう。

文字列補間は便利な機能ですが、構文解析という観点からは少々やっかいな存在です。文字列リテラルは従来、字句解析で扱うトークンでした。そして、字句解析では処理を単純化するためにトークンを正規言語の範囲に収まるようにするのがベストプラクティスでした。字句解析と構文解析を分離し、かつ、字句解析を可能な限り単純化するという観点で言えばある意味当然とも言えますが、文字列補間は正規表現で表現できたもの（正規言語に収まるもの）を文脈自由文法で取り扱わなければいけない存在にしてしまいました。

たとえば、Ruby では以下のように#{}の中にさらに文字列リテラルを書くことができ、その中には#{}を……といった具合に無限にネストできるのです。これまでの章を振り返ればわかるように、これは Dyck 言語的な構造があり、明らかに正規言語では扱えず、文脈自由言語の能力が必要なものです。

```
x = 1
y = 2
"expr1 (#{"expr2 (#{x + y})"})" # "expr1 (expr2 (3))"
```

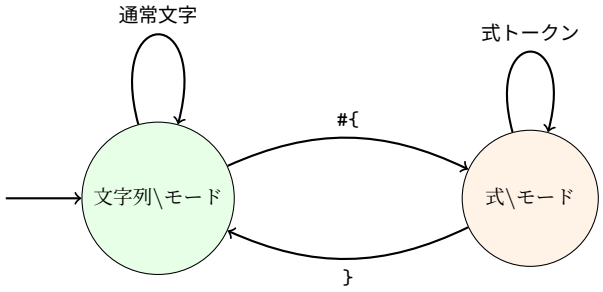
しかし、従来の手法では文字列リテラルはトークンとして扱わなければいけないため、各言語処理系の実装者は ad hoc な形で構文解析器に手を加えています。たとえば、Ruby の構文解析器は Bison を使って書かれていますが、字句解析器に状態を持たせることでこの問題に対処しています。文字列リテラル内に#{が出現したら状態を式モードに切り替えて、その中で文字列リテラルがあらわれたら文字列モードに切り替えるといった具合です。

一方、PEG では字句解析と構文解析が分離されていないため、特別な工夫をすることなく文字列補間を実装することができます。以下は Ruby の文字列補間と同じようなものを PEG で記述する例です。

```
string <- "\"" ("#{" expression "}" / .)* "\""
expression <- 式の定義
```

文字列補間を含む文字列リテラルは分解可能という意味で厳密な意味ではトークンと言えないわけですが、PEG は字句解析を分離しないおかげで文字列リテラルをことさら特別扱いする必要がないわけです。

PEG の利用例が近年増えてきているのは、言語に対してこのようにアドホックに構文を追加したいというニーズがあるためではないかと筆者は考えています。



例: `"abc#{1+2}def"` の 解 析

1. 文字列モード: `"abc"` を消費
2. `#{` を認識 → 式モードへ遷移
3. 式モード: `1+2` を式として解析
4. `}` を認識 → 文字列モードへ復帰
5. 文字列モード: `def"` を消費

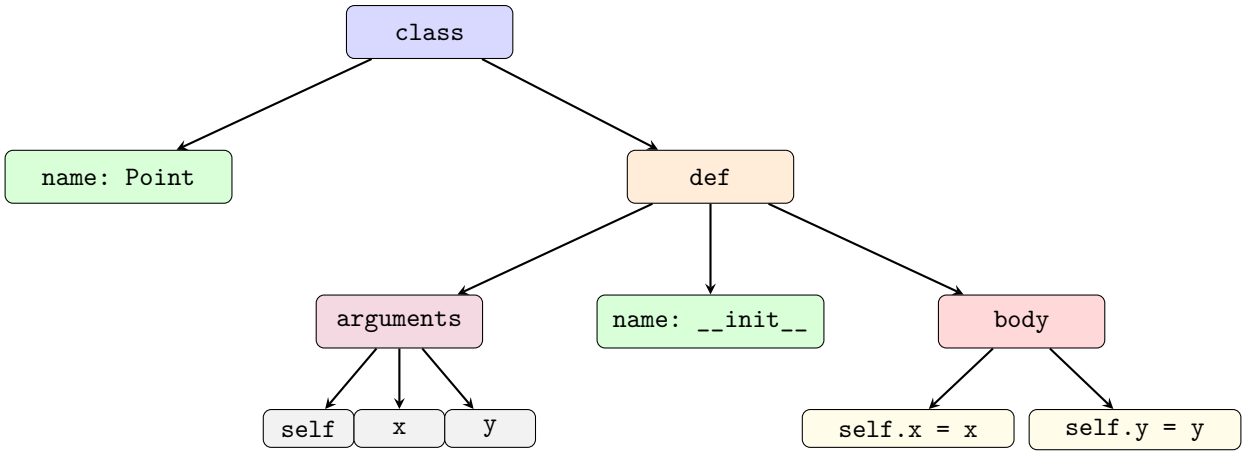
図 7.1: 文字列補間の解析状態遷移

7.2 インデント文法

Python ではインデントによってプログラムの構造を表現します。たとえば、次の Python プログラムを考えます。

```
class Point:
  def __init__(self, x, y):
    self.x = x
    self.y = y
```

この Python プログラムは次のような抽象構文木に変換されると考えられます。



インデントによってプログラムの構造を表現するというアイデアは秀逸ですが、インデントによる構造の表現は明

らかに文脈自由言語を超えます。インデントによってブロックや文法の構造を表現することは、Haskell や Scala 3、YAML などでも採用されており、広く使われているアイデアです。

Python では字句解析のときにインデントを<INDENT>（インデントレベル増加）、インデントを「外す」のを<DEDENT>（インデントレベル減少）という特別なトークンに変換します。これにより、構文解析器自体はこれらのトークンをブロックの開始と終了のように扱うことができ、文脈自由文法の範囲で処理しやすくなります。

例えば、先の Point クラスの定義は、字句解析後には（簡略化すると）以下のようなトークン列として扱われるイメージです。

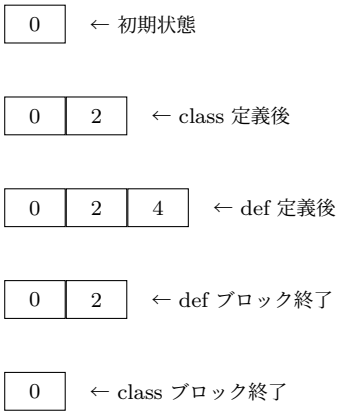
```
<CLASS> <NAME:Point> <COLON> <NEWLINE>
  <INDENT>
    <DEF> <NAME:__init__>
    <LPAREN> <NAME:self> <COMMA> <NAME:x> <COMMA> <NAME:y> <RPAREN>
    <COLON> <NEWLINE>
    <INDENT>
      <NAME:self> <DOT> <NAME:x> <ASSIGN> <NAME:x> <NEWLINE>
      <NAME:self> <DOT> <NAME:y> <ASSIGN> <NAME:y> <NEWLINE>
    <DEDENT>
<DEDENT>
```

（<NAME:A>は識別子 A、<COLON>はコロン、<NEWLINE>は改行、<ASSIGN>は代入演算子を表すトークンとします。実際にはさらに詳細なトークン分割が行われます。このように、インデント/デデントトークンがブロック構造を示すため、構文解析器は括弧の対応付けと同様の形で処理できます。

しかし、よくよく考えればわかるのですが、<INDENT>トークンと<DEDENT>トークンを切り出す時点で、文脈自由ではありません。つまり、字句解析時に<INDENT>と<DEDENT>トークンを切り出すために特殊な処理をしていることになります。<DEDENT>トークンは<INDENT>トークンとスペースの数が対応していなければいけないため、切り出すためには正規表現でも文脈自由文法でも手に余ります。

```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y
```

インデントスタック



スタックを使っていることから一見文脈自由文法で処理できるように思えるかもしれませんが、スタックの各要素が数字であり、この数字が対応してない時点でエラーになるので、文脈自由文法ではありません。Python の字句解析

器はこのようなインデント/デデントトークンをプログラムで処理していますが、その字句解析器は明らかに文脈自由でない言語を処理しています。

いずれにせよ、インデント文法が文脈自由言語に収まらないために、字句解析器あるいは構文解析器のどちらかにしわ寄せがきます。インデント文法を処理できるもっと強力な形式文法はありますが、プログラムで ad hoc に対応する方が早いため、そのような方式が一般的です。

7.3 ヒアドキュメント

ヒアドキュメントは複数行に渡る文字列を記述するための文法で、従来は bash などのシェル言語で採用されていましたが、Perl や Ruby もヒアドキュメントを採用しました。たとえば、Ruby で HTML の文字列をヒアドキュメントで以下のように書くことができます。

```
html = <<HTML
<html>
  <head><title>Title</title></head>
  <body><p>Hello</p></body>
</html>
HTML
```

特筆すべきは、<<HTML と HTML のように対応している間だけが文字列として解釈されることです。これだけなら文脈自由言語の範囲内です。実際には問題はもっと複雑です。ヒアドキュメントは**ネストが可能**なのです。たとえば、以下のようなヒアドキュメントは正しい Ruby プログラムです。

```
here = <<E1 + <<E2
ここは E1 です
E1
ここは E2 です
E2
```

これは以下の内容の文字列として解釈されます。

```
ここは E1 です
ここは E2 です
```

ヒアドキュメント内では文字列補間が使えるのでさらに複雑です。以下のようなヒアドキュメントも OK なのです。

```
a = 100
b = 200
here = <<A + <<B
```

```
a は#{a}です
A
b は#{b}です
B
```

これは次の文字列として解釈されます。

a は 100 です
b は 200 です

読者の方々はおそらく「確かに凄いけど、普通はこのような書き方をすることはほぼないのでは」と思われたのではないでしょうか。実際問題そうなのですが、Ruby はこのような複雑怪奇なプログラムもうまく構文解析できなければいけないのも事実です。

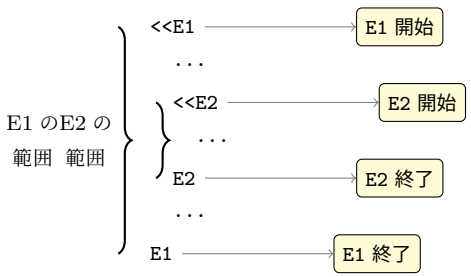
Ruby のヒアドキュメントを適切に構文解析するには HTML や XML におけるタグ名の対応付けと同じ処理が必要になりますが、これは文脈自由言語の範囲を超えています。Ruby のヒアドキュメントがこのような振る舞いをすることを初めて知ったのは筆者が大学院生の頃ですが、あまりに予想外の振る舞いに目眩がする思いだったのを覚えています。

Ruby のヒアドキュメントが実際にどのように実装されているかはさておき、筆者はかつて中田育男先生が実装された ISO Ruby の Scala による試験的な構文解析器の実装のお手伝いをした際に、ヒアドキュメントの扱いに非常に苦労しました。

実装の詳細は、中田先生の ruby_scala リポジトリで確認できますが、ヒアドキュメントの開始デリミタを記憶し、対応する終了デリミタが現れるまでを特別に処理する、といった複雑なロジックが必要でした。

このときは Scala のパーサーコンビネータを使ってヒアドキュメントを再現したのですが、引数を取ってコンビネータを返すメソッドを定義することで問題を解決しました。形式言語の文脈でいうのなら、PEG の規則が引数を持てるように拡張することでヒアドキュメントを解釈できるようになったと言うことができます。

PEG を拡張して規則が引数を持てるようにするという試みは複数ありますが、筆者も Macro PEG という PEG を拡張したものを提案しました。ヒアドキュメントという当たり前に使われている言語機能ですら、構文解析を正しく行うためには厄介な処理をする必要があるのです。



7.4 改行終端可能文法

C、C++、Java、C#などの言語では、解釈・実行の基本単位は**文** (Statement) と呼ばれるものになります。また、文はセミコロンなどの終端子と呼ばれるもので終わるか、区切り文字で区切られるのが一般的です。セミコロンが終端子でなく文の区切りになるのが Pascal などの言語です。厳密には違いますが、関数型プログラミング言語 Standard ML のセミコロンも似たような扱いです。

Java では次のように書くことで、A、B、C を順に出力することができます。

```
System.out.println("A");
System.out.println("B");
System.out.println("C");
```

このように文が終端子（Terminator) で終わる文法には、文の途中に改行が挟まっても単なるスペースと同様に取り扱えるという利点があります。先程のプログラムを次のように書き換えても意味は変わりません。

```
System.out.println(
  "A");
System.out.println(
  "B");
System.out.println(
  "C");
```

大抵の場合、文は一行で終わるのですから、毎回セミコロンをつけなければいけないのも面倒くさいものです。そういったニーズを反映してか、Scala、Kotlin、Swift、Go などの比較的新しい言語では（Scala の初期バージョンが 2003 ですから、そこまで新しいのかという話もありますが)、文はセミコロンで終わることもできるが、改行でも終わることができます。より古い言語でも Python、Ruby、JavaScript も改行で文が終わることができます。

たとえば、先程の Java プログラムに相当する Scala プログラムは次のようになります。

```
println("A")
println("B")
println("C")
```

見た目にもすっきりしますし、改行を併用するコーディングスタイルが大半であることを考えても、無駄なタイピングが減るしでいいことずくめです。Scala ではそれでいて、次のように文の途中で改行が入っても問題なく解釈・実行することができます。

```
println(
  "A")
println(
  "B")
println(
  "C")
```

Scala でも一行の文字数が増えれば分割したくなりますから、このような機能があるのは自然でしょう。

ここで一つの疑問が湧きます。「文は改行で終わる」という規則なら改行が来たときに「文の終わり」とみなせばよいですし、「文はセミコロンで終わる」という規則なら、セミコロンが来たときに「文の終わり」とみなせば問題ありません。しかしながら、このような文法を実現するためには「セミコロンが来れば文が終わるが、改行で文が終わることもある」というややこしい規則に基づいて構文解析をしなければいけません。

このような文法を実現するのは案外ややこしいものです。Java の `System.out.println("A");` という文は正確には「式文」と呼ばれますが、この式文は次のように定義されます。

```
expression_statement ::= expression <SEMICOLON>
```

<SEMICOLON>はセミコロンを表すトークンです。では、Scala 式の文法を「改行でもセミコロンでも終わることができる」と考えて次のように記述しても大丈夫でしょうか。

```
expression_statement = expression (<SEMICOLON> | <LINE_TERMINATOR>)
```

<LINE_TERMINATOR>は改行を表すトークンです。プラットフォームによって改行コードは異なるので、このように定義しておくと楽でしょう。このような規則でうまく先程の例全てをうまく取り扱えるかといえば、端的に言って無理です。たとえば、次のようなコードを考えてみましょう。

```
println(
  "A")
```

(の後に改行が来ていますが、ここでは単なる空白文字として扱って無視して欲しいわけです。しかし、上記の規則では改行が来た時点で文が終わってしまいます。つまり、上のような式文は「式文が改行で終わる」という規則に従っていないため、構文解析エラーになってしまいます。

C 系の言語では原則的には、字句解析時に改行もスペースも同じ扱いで処理しているので、このような「式の途中で改行が来る」ケースも特に工夫する必要がありませんでした。しかし、Scala などの言語における「改行」は式の途中では無視されるが文末では終端子にもなり得るという複雑な存在です。

言い換えると「文脈」を考慮して改行を取り扱う必要がでてきたのです。これ自体は文脈自由文法の範囲で取り扱う事が可能ですが、字句解析器が素朴に空白文字を無視するだけでは対応できません。なぜなら、通常の字句解析器は文脈を持たないからです。字句解析器は入力をトークンに分解するだけのもので、文脈情報を持たないため、改行が文の終わりかどうかを判断できません。

このような文法はどのようにすれば取り扱えるでしょうか。なかなか難しい問題ですが、大きく分けて二つの戦略があります。

一つ目は字句解析器に文脈情報を持たせる方法です。たとえば、「式」モードでは改行は無視されるが、「文」モードだと無視されないという風にした上で、式が終わったら「文」モードに切り替えを行い、式が開始したら「式」モードに切り替えを行います。この方式を採用している典型的な言語が Ruby で、C で書かれた字句解析器には実に多数の文脈情報を持たせています。

```
// https://github.com/ruby/ruby/blob/v3_2_0/parse.y#L161-L181
/* examine combinations */
enum lex_state_e {
#define DEF_EXPR(n) EXPR_##n = (1 << EXPR_##n##_bit)
    DEF_EXPR(BEG),
    DEF_EXPR(END),
    DEF_EXPR(ENDARG),
    DEF_EXPR(ENDFN),
    DEF_EXPR(ARG),
    DEF_EXPR(CMDARG),
    DEF_EXPR(MID),
    DEF_EXPR(FNAME),
    DEF_EXPR(DOT),
    DEF_EXPR(CLASS),
    DEF_EXPR(LABEL),
    DEF_EXPR(LABELED),
    DEF_EXPR(FITEM),
    EXPR_VALUE = EXPR_BEG,
    EXPR_BEG_ANY  =  (EXPR_BEG | EXPR_MID | EXPR_CLASS),
    EXPR_ARG_ANY  =  (EXPR_ARG | EXPR_CMDARG),
    EXPR_END_ANY  =  (EXPR_END | EXPR_ENDARG | EXPR_ENDFN),
    EXPR_NONE = 0
};
```

「改行で文が終わる」以外にも Ruby はかなり複雑な構文解析を行っているため、このように多数の状態を字句解析器に持たせる必要があります。Ruby の文法は私が知る限り**もっとも複雑なものの一つ**なのでややこれは極端ですが、字句解析器に状態を持たせるアプローチは他の言語も採用していることが多いようです。

別のアプローチとして PEG を使うという方法があります。PEG では字句解析という概念自体がありませんから、式の途中に改行が入るというのも構文解析レベルで処理できます。Python では 3.9 から PEG ベースのパーサーが導入され、3.10 以降も継続して使用されています。これにより、Python のパーサーは改行の扱いなど、柔軟な構文規則を以前よりも直接的に記述しやすくなりました。

また、拙作のプログラミング言語 Klassic では次のようにして式の合間に改行を挟むことができるようにしています。

```
//add ::= term {"+" term | "-" term}
lazy val add: Parser[AST] = rule{
  chainl(term)(
    (%% << CL(PLUS)) ^^ { location => (left: AST, right: AST) =>
        BinaryExpression(location, Operator.ADD, left, right)
    }
  | (%% << CL(MINUS)) ^^ { location => (left: AST, right: AST) =>
        BinaryExpression(location, Operator.SUBTRACT, left, right)
    }
  )
}
```

関数 CL() は次のように定義されます。

```
def CL[T](parser: Parser[T]): Parser[T] = parser << SPACING
```

Klassic の構文解析器は自作のパーサーコンビネータライブラリで構築されているので少々ややこしく見えますが、要約すると、CL() は引数に与えたものの後に任意個のスペース（改行）が来るという意味で、キーワードである PLUS や MINUS の後にこのような規則を差し込むことで「式の途中での改行は無視」が実現できています。

現在ある言語で採用されているかはわかりませんが、GLR 法のようにスキャナレス構文解析と呼ばれる他の手法を使う方法もあります。スキャナレスということは字句解析が無いということですが、字句解析器を別に必要としない構文解析法の総称を指します。PEG も字句解析器を必要としませんから、スキャナレス構文解析の一種と言えます。

ともあれ、私達が普通に使っている「改行で文が終わる」ようにできる処理一つとっても厄介な問題だということがポイントです。

7.5 C の typedef

C 言語の typedef 文は既存の型に別名をつける機能です。C 言語や C++ 言語のプログラムをバリバリ書いているプログラマならお馴染みの機能です。typedef は

- 移植性を高める
- 関数ポインタを使ったよみづらい宣言を読みやすくする

といった目的で使われますが、この typedef 文が意外に曲者だったりします。以下は i を int 型の別名として定義するものですが、同時にローカル変数 i を i 型として定義しています。

```
typedef int i;
int main(void) {
        i i = 100; // OK
        int x = (i)'a'; // ERROR
        return 0;
}
```

現実にこのようなコードの書き方をするかはともかく、

```
i i = 100;
```

は明らかに OK な表現として解析してあげなければいけません。一方で、

```
int x = (i)'a';
```

は構文解析エラーになります。`i i = 100;` という宣言がなければこの文も通るのですが、合わせて書けば構文解析エラーです。それ以前の文脈である識別子が typedef されたかどうかで構文解析の結果が変わるのですからとてもややこしいです。

C 言語ではこのようなややこしい構文を解析するために、構文解析器が typedef した識別子のテーブルを持っておいて、それを使って構文解析を行うという手法を採用しています。

7.6 Scala での「文頭に演算子が来る場合の処理」

7.4 で改行で文を終端する文法について説明しましたが、Scala はさらにややこしい入力を処理できなければいけません。たとえば、以下のような文を解釈できる必要があります。

```
val x = 1
      + 2
println(x) // 3
```

ここで、x を 3 とちゃんと解釈するには `val x = 1` が改行で終わったから「文が終わった」と解釈せず、次のトークンである `+` まで見てから文が終わるか判定する必要があります。これまで試した限り、同じことができるのは JavaScript くらいで、Ruby、Python、Kotlin、Go、Swift などの言語ではエラーになるか、`x = 1` で文が終わったと解釈され、`+ 2` は別の文として解釈されるケースばかりでした。

この処理について、Scala 言語仕様内の 1.2 Newline Characters に関連する記述があります。

> Scala is a line-oriented language where statements may be terminated by semi-colons or newlines. A newline in a Scala source text is treated as the special token “nl” if the three following criteria are satisfied: 1. The token immediately preceding the newline can terminate a statement. 2. The token immediately

following the newline can begin a statement. 3. The token appears in a region where newlines are enabled. The tokens that can terminate a statement are: literals, identifiers and the following delimiters and reserved words:

これを意訳すると、通常の場合は Scala の文はセミコロンまたは改行で終わることができるが、次の三つの条件**全て**を満たしたときのみ、特別なトークン `nl` として扱われることになる、ということになります。

1. 改行の直前のトークンが「文を終わらせられる」ものである場合
2. 改行の直後のトークンが「文を始められる」ものである場合
3. 改行が「利用可能」になっている箇所にあらわれたものである場合

たとえば、以下の Scala プログラムについていうと、最初の改行は `nl` トークンになりませんが、何故かというと条件 1 が満たされても条件 2 が満たされないからです。

```
val x = 1
      + 2
```

ちなみに、調査を開始する時点では Scala の文法の基本文法を継承した Kotlin でも同じようになっていると思っていたのですが、一行目で文が終わると解釈されてしまいました。

```
val x = 1
      + 2 // Kotlin では + 2 は単独の式として解釈されてしまう
println(x) // 1
```

これらに対する体系的な調査は行っていませんが、Scala は文法上、ところどころにトリッキーな機能があり、改行の扱いも特殊なものになっています。

7.7　C++ のテンプレート構文

C++ のテンプレートは型パラメータを使ったジェネリックプログラミングを可能にする強力な機能ですが、その構文は構文解析の観点から見ると興味深い問題を含んでいます。特に有名なのが「>>の曖昧性」です。

Java のジェネリクスでいうと `List<List<String>>` のような書き方に相当しますが、C++ では歴史的な経緯から特別な問題がありました。

C++03 までは、次のようなネストしたテンプレートの宣言はコンパイルエラーになりました：

```
std::vector<std::vector<int>> matrix;  // C++03 ではエラー
```

何故かというと、>>が右シフト演算子として解釈されてしまうからです。そのため、C++03 では次のようにスペースを入れる必要がありました：

```
std::vector<std::vector<int> > matrix;  // C++03 では OK
```

この問題は単純な字句解析の問題のように見えますが、実はそうではありません。次の例を考えてみましょう：

```
template<int N>
struct A {
    static const int value = N;
};

// これは A<1>::value >> 2（右シフト）
int x = A<1>::value >> 2;

// これは vector<vector<int>>（テンプレートの終端）
std::vector<std::vector<int>> v;
```

同じ>>という文字列が、文脈によって右シフト演算子として解釈されたり、テンプレート引数リストの終端記号 2 つとして解釈されたりする必要があるのです。これは明らかに文脈自由文法の範囲を超えており、構文解析器は現在の文脈（テンプレート引数リストの中にいるかどうか）を追跡する必要があります。

C++11 以降では、この問題に対処するため、構文解析器がテンプレート引数リストの文脈では>>を> >として解釈するように標準が変更されました。しかし、これは構文解析器の実装をより複雑にします。例えば、次のような入れ子になったケースも正しく処理する必要があります：

```
std::map<int, std::vector<std::vector<int>>> nested;  // C++11 以降は OK
```

さらに複雑なのは、>>=（右シフト代入演算子）や>>>（一部の拡張で使われる）なども考慮する必要があることです。構文解析器は、テンプレート引数リストの文脈を正確に追跡し、適切にトークンを分割する必要があります。

7.8　正規表現リテラルの曖昧性

JavaScript には正規表現リテラルという便利な記法があります。Java では `Pattern.compile("[a-z]+", Pattern.CASE_INSENSITIVE)` のように書くところを、JavaScript では以下のように簡潔に書けます：

```
const pattern = /[a-z]+/i;  // 大文字小文字を区別しない英字のパターン
```

しかし、この/記号は除算演算子としても使われるため、構文解析上の曖昧性が生じます：

```
// これは正規表現リテラル
const regex = /ab+c/;

// これは除算演算
const result = a / b + c / d;
```

さらに厄介なのは、次のようなケースです：

```
// これは何でしょう？
return /regex/i.test(str);  // 正規表現リテラル
return a / b / c;           // 除算演算
```

この曖昧性を解決するには、構文解析器は現在の文脈を理解する必要があります。JavaScript の仕様では、正規表現リテラルが現れることができる位置と、除算演算子が現れることができる位置を厳密に定義しています。

一般的なルールとしては以下のようになります：

- 式の開始位置では/は正規表現リテラルの開始
- 式の途中では/は除算演算子

しかし、実際にはもっと複雑で、前のトークンが何であるかによって判断する必要があります：

```
// 前のトークンが ) の場合は除算
func() / 2

// 前のトークンが = の場合は正規表現
const x = /pattern/

// 前のトークンが return の場合は正規表現
return /pattern/g
```

JavaScript の構文解析器は、前のトークンの種類を記憶し、それに基づいて/の解釈を切り替える必要があります。これは字句解析と構文解析が密接に連携する必要があることを示す好例です。

7.9　エラーリカバリ

構文解析の途中でエラーが起きることは普通にあります。構文解析中のエラーリカバリについては多くの研究があるものの、コンパイラの教科書で構文解析アルゴリズムでのエラーリカバリについて言及されることは稀です。推測ですが、構文解析において２つ目以降のエラーは大抵最初のエラーに誘発されて起こるということや、どうしても経験則に頼った記述になりがちなため、教科書で言及されることは少ないのでしょう。また、大抵の言語処理系で構文解

析中のエラーリカバリについては大したことをしていなかったという歴史的事情もあるかもしれません。

しかし、現在は別の観点から構文解析中のエラーリカバリが重要性を増してきています。それは、IDE あるいはテキストエディタの拡張としての IDE 類似の機能を提供することが一般的になったため、「構文解析エラーになるが、それっぽくなんとか構文解析をして欲しい」という強いニーズがあるからです。

ユーザーがコードを書いている最中は、一時的に文法的に正しくない状態になることが頻繁にあります。IDE がそのような状況でも構文ハイライト、コード補完、リアルタイムのエラー表示などの機能を提供し続けるためには、エラーが発生しても即座に解析を中断するのではなく、可能な限り解析を継続し、後続のエラーも検出できるような仕組み、すなわちエラーリカバリ機構が不可欠です。

エラーリカバリにはいくつかの代表的な手法があります。

- **パニックモード (Panic Mode):** 最も単純な手法の一つです。エラーを検出したら、セミコロン（;）や閉じ波括弧（}）のような、文やブロックの区切りとなる「同期トークン」が見つかるまで、入力トークンを読み飛ばします。同期トークンが見つかったら、そこから解析を再開します。

  C 言語風のコード `x = a + * b; y = c;` で、`*` が予期しないトークンとしてエラーになった場合、パニックモードでは`*` と `b` を読み飛ばし、次の同期トークンである`;` を見つけて解析を再開します。これにより、`y = c;` の解析は行われますが、‘* b‘ に関するエラーの詳細は失われる可能性があります。

- **フレーズレベルリカバリ (Phrase-Level Recovery):** エラー箇所の周辺で、局所的な修正を試みる手法です。例えば、不足しているセミコロンを補ったり、予期しないトークンを削除したり、期待されるトークンに置き換えたりします。

```
x = a + b y = c;
```

というコードで、`b` と `y` の間にセミコロンが欠落している場合、パーサは `y` が予期しないトークンであると判断します。フレーズレベルリカバリでは「文の終わりにはセミコロンが期待される」という知識に基づき、`b` の後にセミコロンを挿入して `x = a + b; y = c;` として解析を試みるかもしれません。

あるいは、Java のメソッド呼び出しで

```
myObject.method(arg1 arg2)
```

のようにカンマが抜けている場合、`arg1` と `arg2` の間にカンマを補って `myObject.method(arg1, arg2)` として解釈を試みる、といった具合です。

JSON の配列 `[1, , 2]` で余分なカンマがある場合、削除して `[1, 2]` として解析を続けるかもしれません。パニックモードよりは洗練されていますが、どのような修正を行うかの判断が難しく、実装が複雑になりがちです。

- **エラー生成規則 (Error Productions):** 文法にあらかじめよくあるエラーパターンに対応する生成規則を追加しておく手法です。例えば、「`if (condition) statement`」という正しい規則に加えて、「`if condition) statement`」（開き括弧が欠落）のようなエラー用の規則を定義しておきます。これにより、特定のエラーを

「受理」し、解析を継続できます。

Yacc のようなツールでは、`error` トークンを使ってエラー規則を定義できます。

```
statement: IF '(' expr ')' statement
         | IF error ')' statement { yyerror("Missing opening parenthesis in if statement"); }
         | /* ... other rules ... */
         ;
```

この例では、`if` の後に開き括弧 `(` がない場合に `error` トークンがマッチし、エラーメッセージを出力しつつ、`)` 以降の解析を継続しようとします。多くのエラーパターンを網羅しようとすると文法が複雑になりますが、特定のエラーに対しては効果的です。

- **グローバルコレクション (Global Correction):** 理論的には最も強力な手法で、入力文字列全体に対して、最小限の修正（挿入、削除、置換）で文法的に正しい文字列に変換する方法を探します。

  `if x > 0) { ... }` という入力に対し、グローバルコレクションは開き括弧 `(` を挿入するのが最小の修正であると判断するかもしれません。しかし、入力全体を考慮して最適な修正を見つけるのは計算量的に非常に困難であり、実用的なコンパイラや IDE でこの手法が全面的に採用されることは稀です。

IDE のような環境では、これらの手法を組み合わせたり、部分的な構文木（エラー箇所を含むかもしれないが、解析できた部分）を構築したり、インクリメンタルな解析（変更箇所だけを再解析する）を行ったりすることで、ユーザーが編集中でも可能な限り正確な情報を提供しようと試みています。

例えば、Java のクラスを書いている途中で

```
class MyClass {
  public void myMethod() {
    ...
  }
// ここで閉じ括弧 `}` を入力し忘れた場合
```

と入力し、最後の閉じ括弧 `}` を入力し忘れている場合でも、IDE は `myMethod` の本体部分についてはある程度解析を試み、メソッド内の変数に対するコード補完や型チェックを行おうとします。これは、エラーリカバリ機構が、エラー箇所を特定しつつも、それ以外の部分については解析を継続し、部分的な構文情報を抽出しているためです。

エラーリカバリは、単にエラーを見つけるだけでなく、その後の解析をどう継続し、ユーザーにどのようなフィードバックを与えるかという、実践的で奥深い問題領域なのです。

7.10　まとめ

7 章では現実の構文解析で遭遇する問題について、いくつかの例を挙げて説明しました。筆者が大学院博士後期課程に進学した頃「構文解析は終わった問題」と言われたのを覚えていますが、実際にはその後も ANTLR の `LL(*)`、さ

らにそれに続く ALL(*) アルゴリズムのような革新が起きていますし、細かいところでは今回の例のように従来の構文解析法単体では取り扱えない部分をアドホックに各プログラミング言語が補っている部分があります。

このような問題が起きるのは結局のところ、当初の想定と違って「プログラミング言語は文脈自由言語では表せなかった」ということです。もちろん、文脈自由言語の範囲に納めるように文法を制限することは可能ですが、便利な表記を許していくとどうしても文脈自由言語から「はみ出て」しまうということでしょう。このような「現実のプログラミング言語の文脈依存性」については専門の研究者以外には案外知られていなかったりしますが、ともあれこのような問題があることを知っておくのは、既存言語の表記法を取り入れた新しい言語を設計するときにも有益でしょう。

第 8 章

第 8 章 おわりに

ここまでで構文解析の世界を概観してみましたがいかがでしたか？ 構文解析、特に非自然言語の構文解析というのは地味なもので、パーサージェネレータの発展などもあり、20 世紀末には「構文解析はもう終わった問題だ」という人もいました。ただ、その一方で 2000 年代以降になっても PEG の発明（再発見）があり、Python の構文解析器に採用されるまでにつながりましたし、`LL(*)` や `ALL(*)` のような革新的なアルゴリズムが生み出されています。それも、どちらかといえば主流であった上向き型の構文解析でなく下向き型の構文解析で、です。

とはいえやはり地味なものは地味であり、プログラミング言語処理系を構成するコンポーネントという観点から言っても「脇役」という印象は否めません。ただ、わたしたちはプログラミング言語を書いているときは、コンパイラの内部表現や抽象構文木と対話しているわけではありません。プログラマーが直接対話する相手はプログラミング言語の具象構文であり、具象構文はプログラミング言語の「UI」を担当する部分といえるでしょう。通常のアプリケーションで UI が軽視されるべきでないのと同様にやはり具象構文も軽視されるべきでないと私は思いますし、よりよい具象構文の設計には構文解析の知識が助けになると信じています。

ところで、ここまで、構文解析の基盤を支える「形式言語」の世界についてはあえてはしょった説明に留めました。何故なら構文解析を学ぶという点からすると本筋から外れ過ぎてしまいますし、何より形式言語理論を学ぶのは骨が折れる作業でもあるからです。

とはいえ、せっかくなので、この章では形式言語理論のほんの導入だけでも紹介したいと思います。形式言語理論は、言語の構造を数学的に研究する分野であり、構文解析の理論的基盤となっています。「数学的」というと難しそうに聞こえますが、要は「プログラミング言語の文法を厳密に扱うための理論」と考えてください。

例えば、第 4 章で触れた「正規言語」や「文脈自由言語」といった言語クラスは、それぞれ異なる計算モデル（機械）によって認識できることが知られています。本書でも多少触れましたが、改めて簡単にまとめておきます。

- **正規言語:** 有限オートマトンという、有限個の状態しか持たない単純な機械で認識できます。正規表現は正規言語を記述するための便利な記法です。括弧の対応のように無限のネスト構造を持つものは扱えません。
- **文脈自由言語:** プッシュダウンオートマトンという、有限オートマトンにスタック（無限の深さを持つメモリ）を追加した機械で認識できます。括弧の対応のような再帰的な構造を扱えるようになります。本書で紹介した多くの構文解析アルゴリズムは、この文脈自由言語を対象としています。
- **文脈依存言語:** 文脈自由言語よりも強力なクラスで、例えば `a^n b^n c^n`（n 個の a、n 個の b、n 個の c がこ

の順で並ぶ文字列の集合）のような言語を記述できます。プッシュダウンオートマトンでは認識が不可能です。

- **帰納的可算言語（チューリングマシンが認識する言語）：**最も強力な言語クラスで、私たちが普段使う Java や Python などのプログラミング言語で書けるアルゴリズム（計算可能な問題）が認識できる言語の範囲に対応します。

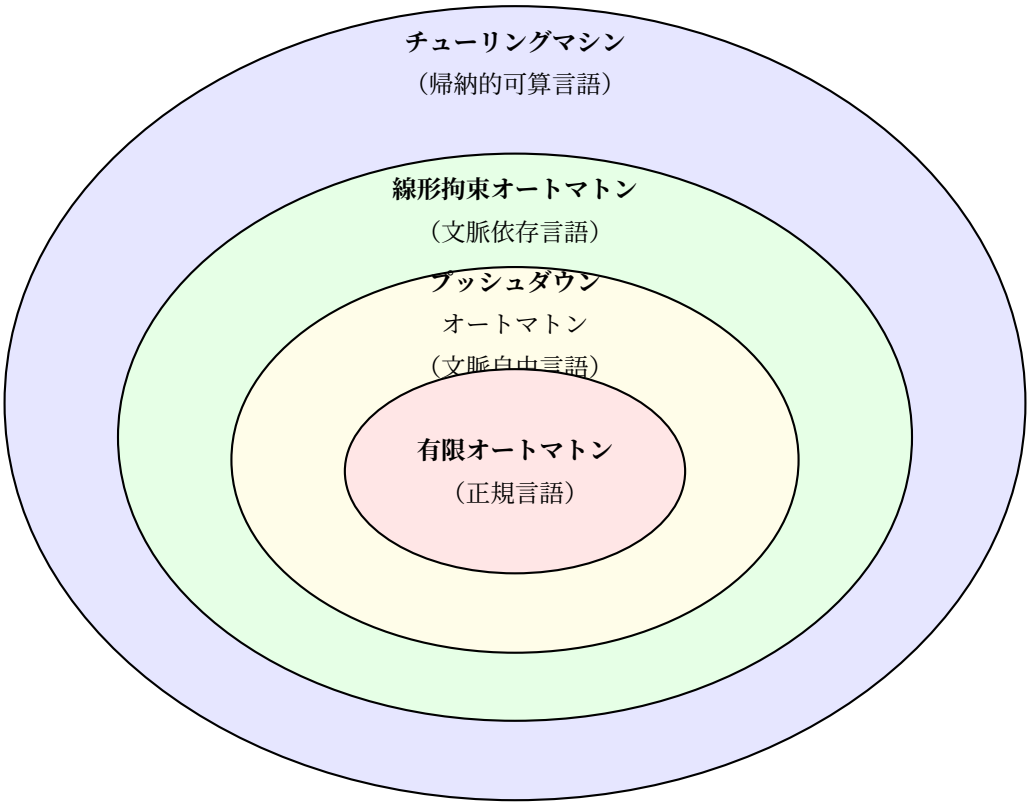


図 8.1　オートマトンの階層と言語クラスの対応（各内側の集合は外側の集合に含まれる）

形式言語理論を学ぶと、「なぜある種のパターンは正規表現で書けるのに、別のパターンは書けないのか？」や「なぜ `a^n b^n` は文脈自由言語なのに `a^n b^n c^n` はそうではないのか？」といった疑問に、より深いレベルで答えることができるようになります。これらの問いは、計算モデルの能力の限界と深く関わっています。

幸い、形式言語理論を学ぶための良質な教科書はいくつもあります。もしこの本を読み終えて、言語の理論的な側面にさらに興味を持った方は、ぜひ専門書を手に取ってみてください。以下でいくつかの参考文献を紹介します。

8.0.1　古典的名著・専門書

- A.V. エイホ、R. セシィ、M.S. ラム、J.D. ウルマン,『コンパイラ - 原理・技法・ツール』(第 2 版)、サイエンス社、2009 年（通称ドラゴンブック）
  - コメント: コンパイラ構築に関する標準的な教科書。字句解析、構文解析（LL、LR)、意味解析、コード生成など、コンパイラの全般的なトピックを網羅。理論的背景もしっかり解説されています。中級者以上向け。
- J. ホップクロフト、J. ウルマン、『オートマトン　言語理論　計算論　I』(第 2 版) 、サイエンス社、2003 年
  - コメント: オートマトンと形式言語の理論に関する大学生レベル以上向け教科書。正規言語、文脈自由言語など、計算理論の基礎をしっかり学べます。数学的な厳密さを求める方向け。

- J. ホップクロフト、J. ウルマン、『オートマトン　言語理論　計算論　II』(第 2 版) 、サイエンス社、2003 年
  - コメント: 同上。IIではチューリングマシン、決定不能性、計算複雑性などを取り扱っています。IIは計算理論のより深い部分に踏み込んでいます。
- Dick Grune, Ceriel J.H. Jacobs. *Parsing Techniques: A Practical Guide (2nd Edition)*
  - コメント: 書名どおり、様々な構文解析技術に特化した書籍。LL、LR だけでなく、アーリー法、GLR、CYK 法など、より高度なアルゴリズムや曖昧性のある文法の扱いについても詳しい。構文解析を専門的に深めたい方向け。

8.0.2 特定の技術に関する論文・資料

- Bryan Ford, “Parsing Expression Grammars: A Recognition-Based Syntactic Foundation”, 2004
  - コメント: PEG を提案したオリジナルの論文。PEG の形式的な定義、操作的意味論、Packrat Parsing について解説。理論的な背景を深く理解したい方向け。(オンラインで検索すれば見つかるはずです)
- Terence Parr, *The Definitive ANTLR 4 Reference*
  - コメント: ANTLR v4 の作者自身による解説書。ANTLR の文法定義、使い方、ALL(*) アルゴリズムの概要、実践的なパーサー構築のテクニックが豊富。ANTLR を使いこなしたいなら必読。
- ANTLR 公式サイト、https://www.antlr.org/
  - コメント: ANTLR のドキュメント、チュートリアル、文法リポジトリなど。最新情報やコミュニティのサポートも得られます。
- JavaCC 公式サイト、https://javacc.github.io/javacc/
  - コメント: JavaCC のドキュメント、チュートリアル、FAQ など。
- GNU Bison マニュアル、https://www.gnu.org/software/bison/manual/
  - コメント: Bison (Yacc 互換) の詳細なマニュアル。LALR(1) や GLR パーサーの生成方法、文法定義の書き方などが学べます。

これらの資料を通じて、構文解析の世界への探求をさらに深めていただければ幸いです。

8.1 まとめ

本書を通じて、構文解析の基本的な考え方や、様々なアルゴリズム、現実のプログラミング言語が抱える課題の一端に触れていただきました。

よりよい具象構文の設計には構文解析の知識が助けになるという趣旨のことを冒頭で述べましたが、これは例えば、あなたが新しいドメイン固有言語（DSL）を設計する際に、利用者が直観的に理解しやすく、かつパーサーが効率的に解析できるような構文（例: 演算子の優先順位、予約語の選択、ブロック構造の表現方法など）を選ぶ上で、本書で学んだ LL/LR の特性や PEG の柔軟性といった知識が役立つでしょう。

たとえば、JSON のようなシンプルな設定ファイル形式を拡張したいとき、コメント機能を追加するにしても「行コメント」にするか「ブロックコメント」にするか、あるいは Python のような文字列リテラル内のドキュメント方式にするかで、構文解析の難易度は変わってきます。こうした選択を適切に行えるようになることが、本書で得られる

実践的な知識の一つです。

また「構文解析は終わった問題ではない」という点も改めて強調しておきたいと思います。プログラミング言語は進化を続けており、async/await のような非同期処理の構文、パターンマッチングの高度化、型システムの進化に伴う構文の複雑化など、依然として構文解析技術に新たな課題を提示し続けています。特に、言語の進化の過程で機能追加をする場合、本書でも出てきた衝突（コンフリクト）が起こることは珍しくありません。これらの課題に取り組む上で、本書で得た知識が何らかの形で皆さんの力になることを願っています。

構文解析の世界は奥深く、そして面白いものです。この本が、その面白さを少しでも伝えることができたなら、著者としてこれ以上の喜びはありません。

2025 年 6 月、自室にて。水島宏太

第 9 章

参考文献

9.1 日本語書籍

- A.V. エイホ、R. セシィ、M.S. ラム、J.D. ウルマン,『コンパイラ - 原理・技法・ツール』(第 2 版) サイエンス社、2009 年
- 新屋良磨、鈴木勇介、高田謙、正規表現技術入門——最新エンジン実装と理論的背景、技術評論社、2015 年

9.2 英語書籍

- Andrew W. Appel『Modern Compiler Implementation in ML』Cambridge University Press, 2008 年
- Terence Parr『Language Implementation Patterns』Pragmatic Bookshelf, 2009 年
- Terence Parr『The Definitive ANTLR 4 Reference』Pragmatic Bookshelf, 2013 年

9.3 重要論文

- Donald E. Knuth「On the Translation of Languages from Left to Right」Information and Control, Vol. 8, No. 6, pp. 607-639, 1965 年
- Frank DeRemer「Simple LR(k) grammars」Communications of the ACM, Vol. 14, No. 7, pp. 453-460, 1971 年
- Bryan Ford「Parsing expression grammars: a recognition-based syntactic foundation」POPL '04, pp. 111-122, 2004 年
- Terence Parr, Kathleen Fisher「LL(*): The Foundation of the ANTLR Parser Generator」PLDI '11, pp. 425-436, 2011 年

9.4 仕様書・標準

- ECMA-404 The JSON data interchange syntax
- ISO/IEC 14977:1996 EBNF

9.5 Web リソース

- Crafting Interpreters by Robert Nystrom
- Let's Build a Compiler by Jack Crenshaw
- Writing An Interpreter In Go by Thorsten Ball