2020年度「プログラミング言語」配布資料 (7)

五十嵐 淳

2022年10月02日

プログラミング言語の仕様は，なんとなくプログラムを書いている間はわざわざ目を通す機会はあまりないが，その言語についての正確な知識を得るためには有益である．その意味で言語仕様は辞書のようなもの[1]である．また，言語処理系を作る者にとっては，言語仕様を読む技能は必要不可欠といえる．言語仕様には大きくわけて，次の2つのことが書いてある．

- どんな記号列がその言語のプログラムたりえるか，という**構文論 (syntax)** について
- プログラムがどんな動作をするか，という **意味論 (semantics)** について

これに加えて多くの言語仕様では，処理系が提供すべき標準ライブラリについての記述が加えられていることも多い．この章では，いくつかの言語の仕様を具体的に見て，特に言語の構文についての記述を概観していく．

1 プログラミング言語の構文論

プログラミング言語の構文論は多くの現代的な言語の場合, **字句文法 (lexical grammar)** と **構文文法 (syntactic grammar)** にわけて与えられる．字句文法は，どういった文字の並びがその言語の「単語」になりうるか (この「単語」を**トークン (token)** ということがある) を与えているもので，構文文法は，どういったトークンの並びがプログラムたりうるかを規定している．この二段構えの構成はコンパイラの構造と密接に関連している．コンパイラは通常，入力ファイルを，**字句解析 (lexical analysis)** 処理によってトークン列に変換し，それを，**構文解析 (syntactic analysis)** 処理によって**抽象構文木 (abstract syntax tree)** と呼ばれる，プログラムの構成を表現するデータに変換する．(その後，様々な処理を経てコンパイラであればターゲットプログラムが，インタプリタであればプログラムの実行結果が出力されるが，それはこの講義の範囲ではない．) また，どういった文字列がプログラムたりうるか，という意味では型に関する制約も構文論の一部といえるが，通常のコンパイラの実装では，構文解析と型検査は別フェーズの処理であることもあり，構文に関する仕様とは見做されないことも多い．(古典的なコンパイラの教科書では型検査は**意味解析 (semantic analysis)** と呼ばれる．)

字句文法や構文文法を定義する際には通常，**BNF (Backus-Naur form)** またはその拡張が使われる．これは，本質的には文脈自由文法の生成規則であるが，生成規則の右辺に，選択を表す`|`，グループ化をするための括弧 (`(...)`)，一回以上の繰り返しを表す`+`，0回以上の繰り返しを表す`*`，0または1回，すなわちオプショナルな出現を表す `?` や `[...]` などの記法を使う．例えば，以下は OCaml における識別子を定義する BNF である．(OCaml では {} で0回以上の繰り返しを，$\{\}^+$ で1回以上の繰り返しを表す，という (やや標準的ではない？) 記法を採用している．)

$$letter ::= \texttt{A} \ldots \texttt{Z} \mid \texttt{a} \ldots \texttt{z}$$

[1] 初心者が調べるのに使ってみても，説明に使われている言葉の意味がわからなくて結局なんだかわからないという点も含めて．

$$ident ::= (letter \mid \texttt{\_})\{letter \mid \texttt{0}\ldots\texttt{9} \mid \texttt{\_} \mid \texttt{'}\}$$

これで識別子 (*ident*) が，英文字かアンダースコア (“_”) で始まり，英数字かアンダースコアかアポストロフィ (“'”) が続く文字列であることがわかる．

1.1 字句文法

字句文法では，以下のことが定義される．

空白文字 **空白文字 (blank characters)** は，トークン間の区切りを明示する役割は果たすが，構文解析の段階では無視する (トークンとして現れない) ことが多い．この講義で扱った言語では，スペース，タブ，改行文字は一括して空白文字として扱われ全て構文解析の段階では無視されるので，改行をスペースに置き換えたり，その逆をしてもプログラムとしては全く同じく扱われる．(字句・構文解析技術が成熟していなかった頃の) 古いプログラミング言語では，行が特殊な意味を持つものも多い．また，逆に最近の言語 (Haskell, Python など) では，ひと固まりの処理 (C 言語, Java での {} で表されるブロック) などを表すために，括弧などの記号ではなく，字下げ (インデント, indent)，つまり行頭からの空白の長さを利用するものもあり，そういった言語ではスペースと改行は厳密に区別される．また，エディタ上では何も表示されていないのに空白文字ではない，という文字 (典型的にはいわゆる全角スペース) もありうるので注意が必要である．

コメント コメントも通常，空白と同じく構文解析の段階では無視される．コメントには，C 言語や Java の `//` のように，ある記号から行末までをコメントとして扱ういわゆる一行コメントと，C 言語や Java の `/* ... */`，OCaml の `(* ... *)` のように開始記号と終端記号の間を全てコメントとして扱うブロックコメントがある．ブロックコメント中に再びコメント開始記号が現れるような，いわば入れ子の扱いは言語によって異なるので注意した方がよい．例えば C 言語では入れ子コメントはなく`/* ABC /* DEF */ GHI */` は `... DEF */` までしかコメントとして扱われない．そのため `GHI` の後の`*/` を読み込んだ時点でエラーが検知される．一方，OCaml は開始記号と同じ数の終端記号が現れるまでコメントとして扱うため，`(* ABC (* DEF *) GHI *)` 全体がコメントとなる．

識別子と予約語 **識別子 (identifier)** はプログラム中に登場する「もの」(変数や型，クラス，コンストラクタ) の名前として使える記号列である．識別子で使える記号は英数字が中心だが，それ以外の記号については言語によってわりとまちまちである．また一部の文字列は**予約語 (reserved word)** といって，識別子として使うことはできないよう指定されている．また，文法とは別に，長さ，大文字小文字の使い分け，などについての慣習・ローカルルールが**命名規則 (naming convention)** として設けられていることも多い．識別子は英語などの自然言語から由来することも多いので，英語 (のように空白が単語の区切りになる言語) で複数の単語にわたるフレーズを識別子とする場合，単語をアンダースコア (`_`) でつなぐか，ハイフン (`-`) でつなぐか，単語の先頭を大文字にして (空白なしで) つなぐか，などが (言語仕様の外で) 決められていることも多い．これらのつなぎ方はそれぞれスネークケース (`snake_case`)，チェインケース (`chain-case`)，キャメルケース (`camelCase`) などと呼ばれる．

各種定数 整数，浮動小数点数，文字，文字列などの定数の表記を定める．これらは種類毎に整数定数のトークン，浮動小数点数のトークン，文字定数のトークン，文字列定数のトークンとなる．

その他記号 その他，括弧や `==` などといった記号列でトークンになるものを定める．

トークンの区切り方に複数の可能性がある場合，多くの場合，(ファイルの先頭から見て) できるだけ長く文字列をとってトークンになるように区切る，という方法が採用される．例えば C 言語では `&&` も `&` も演算子だが，`&&` を`&` がふたつ並んだものではなく，`&&` でひとつのトークンとして扱う．`&&&` が現れている場合，`&&` に

続いて `&` が並んだものとして扱うだろう．このような区切り方を「longest match をとる」といったりする．

1.1.1 予約語とキーワード

既に述べたように予約語は識別子として使えない例外的な文字の並びである．例えば Java, C 言語, OCaml では `for` は予約語であり，変数の名前として使うことができない．予約語と関連する用語として**キーワード (keyword)** がある．これは，言語の構文を表すための特別な文字列で，これらの言語で `for` はキーワードでもある．多くの言語で予約語とキーワードはほとんど一致するが，いくつか例外もある．

- Java では (今のところ) `goto` 構文があるわけではないが識別子として使えない．上の意味では，予約語ではあるがキーワードではないということになるのだが，Java 言語仕様の用語では，ここでいう予約語も全て「キーワード」と呼んでいる．
- FORTRAN のようにキーワードが識別子として使える言語もある．例えば `FOR` 文がある一方で，変数 `FOR` に数を代入したりできる．キーワードが識別子になることができると構文解析がややこしくなるので，現代的な言語では避けられることが多い．

1.1.2 ケーススタディ: Java

Java 言語仕様 3 章 Lexical Structure を見てみよう．

1.1.3 ケーススタディ: OCaml

OCaml reference manual 7.1 節 Lexical convention を見てみよう．

1.1.4 ケーススタディ: C

言語仕様ドラフト N1256 の 6.4 Lexical elements を見てみよう．

1.1.5 ケーススタディ: Scheme

言語仕様第五版 R5RS の Chapter 2 Lexical Conventions と Section 7.1.1 Lexical structure を見てみよう．

1.2 構文文法

構文文法が与えるのは，どういったトークンの列が，式，文，関数定義などといったプログラムの構成要素となるかの規則である．また，構文・演算子の結合順位 (例えば `1 + 2 * 3` が「`1` と，`2` と `3` に演算子 `*` を施した結果に `+` を施したもの」であることがなど) が決められているのも構文文法である．

まず，算術演算子 `+` や `*`，(C 言語の) `&` などの演算子は以下のように分類される．

前置演算子 (prefix operator) 式の前に置いて，より大きな式を構成するような演算子．(C 言語の `&` など．)
中置演算子 (infix operator) ふたつの式の間に置いて，より大きな式を構成するような演算子．(`+` など．)
後置演算子 (postfix operator) 式の後に置いて，より大きな式を構成するような演算子．(C 言語，Java の`++` など．)

ミックスフィックス演算子 (mixfix operator) ふたつ以上の記号列からなる演算子．Java や C では〈式 1〉の値が真なら〈式 2〉の値，偽なら〈式 3〉の値になる，という OCaml の `if` 相当の〈式 1〉`?`〈式 2〉`:`〈式 3〉という構文があり，三項演算子 (ternary operator) としばしば呼ばれるが，これは，`?` と `:` を組み合わせたミックスフィックス演算子の一例である．

C 言語の `*` や `&` のように，ひとつの記号が前置演算子になったり中置演算子になる場合もある．Java や C 言語の `++` は前置演算子にもなるし，後置演算子にもなる．

先程の `1 + 2 * 3` のように，ひとつの式 (ここでは 2) の前後に演算子が置かれた場合に，どちらを優先させて結合するかを決めるのが演算子の**優先順位 (precedence)** または **結合の強さ**である．強いものほど優先させて結合する．「中置演算子 `*` は中置演算子 `+` よりも結合が強い」という．また，C 言語での`*x++` という表現は，* 変数 `x` にポインタ参照 (`*`演算子) を施した先のオブジェクトの中を 1 増やす (`++`演算子)，すなわち `(*x)++` と読むのではなく，* 変数 `x` を 1 増やして (`++`演算子) から，(増やす前の) `x` の指す先を参照する (`*`演算子)，すなわち `*(x++)` と読む表現である．このとき「後置演算子`++`は前置演算子 `*` よりも結合が強い」という．

ここまで，演算子は何らかの記号・文字列であることを暗黙のうちに仮定していて話を進めてきたが，言語によっては，専用のトークンが存在しない演算 (子) もありえる．典型例が，ふたつの式を並置する OCaml の関数適用 (`f x` など) である．この場合でも，`f x + 1` が `f (x + 1)` のことなのか，`(f x) + 1` のことなのかを決定するために優先順位を定める必要がある．

複数の演算子が等しい優先順位を持つこともある．同じ (優先順位の) 中置演算子が，ある式の前後に置かれた場合に，どちらを優先するかを決定するのが，**結合性 (associativity)** であり，**右結合 (right associative)** と **左結合 (left associative)** のふたつがある．右結合の場合〈式 1〉〈演算子 1〉〈式 2〉〈演算子 2〉〈式 3〉は〈式 1〉〈演算子 1〉(〈式 2〉〈演算子 2〉〈式 3〉) と解釈され，左結合の場合〈式 1〉〈演算子 1〉〈式 2〉〈演算子 2〉〈式 3〉は (〈式 1〉〈演算子 1〉〈式 2〉)〈演算子 2〉〈式 3〉と解釈される．例えば`+` と `-` は (C 言語でも Java でも OCaml でも) 左結合演算子で，`1 + 2 - 4 + 5` は `((1 + 2) - 4) + 5` のことである．

いくつかの言語では，プログラマが新しい演算子を優先度・結合性とともに定義することができる．

優先順位は演算子の文脈で説明されることが多いが，その範囲が構文の形式だけからははっきりしない/曖昧であるような構文要素同士についても優先順位を考える必要がある．「範囲が曖昧」というのは，例えば OCaml の `if ... then ... else ...` や `let x = ... in ...` のように `else` や `in` の後どこまでが `if` 式，`let` 式なのか明らかでない，という意味である．一方，OCaml の `begin ... end` は `end` がこの構文の終わりを示すので「範囲が構文だけからはっきりしている」構文要素である．前置・中置・後置演算子も範囲のはっきりしない構文要素である．(前置なら終わりがはっきりしない，後置なら始まりがはっきりしない，中置は始まりも終わりもはっきりしていない．) 一般には終わりがはっきりしない構文要素の中で始まりがはっきりしない構文要素が現れると優先順位の問題が発生する．例えば `let x = y in 1 + y` は `(let x = y in 1) + y` のことなのか `let x = y in (1 + y)` のことなのかを優先順位で決める必要がある．

以降のケーススタディでは，式の構文がどのように指定されているかを中心に見ていくが，優先順位や結合性の指定の仕方の違いに注目しよう．また，仕様では，構文と同時に，意味 (動作) も織り交ぜて記述されていることが多い．

1.2.1 ケーススタディ: OCaml

OCaml reference manual 7.7 節 Expressions を見てみよう．

OCaml の場合，全ての種類の式が，ひとつの非終端記号 (expr) に関する生成規則で記述されており，優先順位や結合性は別途表になっている．また，OCaml でも新しい演算子を定義することができるが，優先度・結合性は演算子の先頭の記号の種類で最初から決まっていて，プログラマが決められない．融通が利かないようにも思えるが，優先度をプログラマが決められるようにすると，同じ字面の式でもプログラム毎に式の意味が変わったりするので，それはそれで紛らわしい．

1.2.2 ケーススタディ: Java

Java 言語仕様 15 章 Expressions を見てみよう．Java では，沢山の非終端記号を導入して生成規則を工夫することで優先順位や結合性を表現している．

以下は，乗算・除算と加算・減算についての文法規則の抜粋である．UnaryExpression は別に定義されている前置演算子式のための非終端記号の名前である．

```
MultiplicativeExpression:
  UnaryExpression
  MultiplicativeExpression * UnaryExpression
  MultiplicativeExpression / UnaryExpression
  MultiplicativeExpression % UnaryExpression

AdditiveExpression:
  MultiplicativeExpression
  AdditiveExpression + MultiplicativeExpression
  AdditiveExpression - MultiplicativeExpression
```

ここから，*, /, % が +, - よりも優先度が高く，左結合になっていることが (よく考えると) 読み取れる．このことは，以下のような文脈自由文法を使って，開始記号 (A) からどんな文字列がどのように導出されるか導出木を考えてみるとよいだろう．

$G = (V, \Sigma, R, A)$

$V = \{A, M, U\}$

$\Sigma = \{0, 1, (,), +, *\}$

$R = \{A \to M, A \to A + M, M \to U, M \to M * U, U \to 0, U \to 1, U \to (A)\}$

A から $(0 + 1) * 0 * 1$ を導出する: $A \to M \to M * U \to M * 1 \to M * U * 1 \to M * 0 * 1 \to U * 0 * 1 \to (A) * 0 * 1 \to (A + M) * 0 * 1 \to ... \to (0 + 1) * 0 * 1$

1.2.3 ケーススタディ: C

言語仕様ドラフト N1256 の 6.5 Expressions を見てみよう．

C 言語でも，優先順位や結合性は，沢山の非終端記号を導入して生成規則を工夫することで与えている．

1.2.4 ケーススタディ: Scheme

言語仕様第五版 R5RS の Chapter 4 Expressions と Section 7.1.2〜7.1.6 を見てみよう．

Scheme の場合は，演算子の優先度などの概念がそもそもない．構文毎に syntax (プリミティブ, 4.1 節), library-syntax (プリミティブではなく，マクロと呼ばれる機能によって別の構文に展開されて処理される構文で，derived expression とも呼ばれる．4.2 節) に分かれている．

2 エラー

間違ったプログラム (そもそもプログラムでないような入力も含めて) についてできるだけ漏れなく挙げ，その場合に何が起こるべきかを記述するのも言語仕様の重要な役割である．エラーにも，以下にあげるようないくつかの種類がある．

大きくわけて，プログラムを言語処理系に与えてから実行が始まる前に検出されるエラー，いわゆる**コンパイル時エラー (compile-time error)**[2]と，実行の最中に検出されるエラー (**実行時エラー (run-time error)**) に大別される．

2.1 構文エラー

字句解析と構文解析の段階で発生するコンパイル時エラーである．言語処理系について学ぶとわかるが，大抵のプログラミング言語の構文解析は，前から順に読んでいき，行き詰まったら即座に (その先は読まずに) 止まってしまうアルゴリズムで作られていることもあり，エラーメッセージは不親切であることが多い (つまり，「ここで期待するトークン・記号が来なかった」程度しか教えてくれない．)

2.2 型と型に関するエラー

Java, OCaml, C では，

- 数値でないものに対する算術演算[3]
- 関数でないものを関数として呼び出そうといている
- メソッド呼び出しをしようとしたところ，そのメソッドがオブジェクトにない

といった，プログラム中でやりとりされるデータの種類に関する不整合がないかどうかをコンパイル時に検査する．データの種類を表すための `int` や `double` といった情報を **型 (type)** と呼び，データの種類に関する不整合がないかの検査を **型検査 (type checking)** と呼ぶ．型検査をコンパイル時 (すなわちプログラムの実行前) に行なう言語と，プログラムを実行しながら行う言語があり，前者を**静的に型付けされる言語 (statically typed language)**，後者を**動的に型付けされる言語 (dynamically typed language)** という．Scheme, Python, JavaScript といった言語は動的に型付けされる言語である．

静的型エラー (static type error) は，コンパイル時の型検査 (これを**静的型検査 (static type checking)** という) において発見されるエラーのことである．

[2]コンパイラを使う場合は翻訳段階で発生するのでコンパイル時エラーでよいが，実行にコンパイラを使わない場合にも発生するので他の用語がほしいところだ．

[3]言語によっては，数値への自動変換を試みて，それが成功すればエラーとはならない (もしくは，どんな値でも数値に変換できてしまう) ものもある．

一方，実行時に行われる型検査に起因するエラーを**実行時型エラー (run-time type error)** と呼ぶ．Java, OCaml では，静的型検査に通ったプログラムについては，実行時型エラーは発生しないことが理論的に保証されている (ことになっている)．このような，「一部の実行時エラーを実行前に全て検知できる型システムを持つ」言語を**強く型付けされる言語 (strongly typed language)** という．C 言語も実行前に型検査を行うが，検査がザルなので何かのエラーが防げる保証があるわけではない．

ただし，強く型付けされる言語でも，

- 除算を行おうとしたところ，除数がゼロだった．
- メソッド呼び出しをしようとしたところ，メソッド呼び出しの対象となるオブジェクトを表す式の値が `null` ポインタだった．

といった，その言語における型の分類能力を越えたところで発生するエラーを防ぐことはできない．強く型付けされる言語は，バグを仕込まないための強力なサポートをしてくれるが，型の表現力に相対的な能力しかないことには注意したい．(型の表現力をあげてより多くのエラーを防ごう，という試みはプログラミング言語研究の一分野である．)

2.3 実行時エラー

実行時エラーは，その名の通り，実行時に発生するエラーの総称である．実行時型エラー，メモリ不足，ゼロ除算，`null` ポインタ参照などが典型的な実行時エラーである．

実行時エラーが発生した場合に，実際に何が起こるかは言語によってまちまちであるが，典型的にはプログラム実行が異常終了する．モダンな言語では例外処理と呼ばれる実行時エラーから回復するための言語機構が用意されていることも多い．C 言語では以下に述べる未定義動作 (undefined behavior) として定義[4]している．また，同じ原因のエラーでも言語によって対処がまちまちなこともある．例えば整数演算のオーバーフローなどは C では未定義だが，OCaml では全く無視される．

2.4 未定義動作と未指定動作

通常，「動作が未定義」とは言語仕様にぬけがあって，プログラムがどのように実行されるかが不明な場合をさすが，一部の言語 (C 言語が代表) では，**未定義動作 (undefined behavior)** という用語をわざわざ導入している．C 言語における未定義動作の定義は何でもあり—anything goes—であり，無理矢理実行を続けてもよい (その後どうなるかは，神，もとい，コンパイラ・ＯＳ・ハードウェアのみぞ知る) し，実行時エラーとして実行を中断してもよい[5]．

未定義動作がどういう場合に発生するかわかるくらいなら，きちんと実行を中断させるようにしろ，というのは尤もな指摘だが，エラーを検知するためのコストを払いたくない場合もある．例えば配列アクセスで添字が大きすぎる場合，C 言語では未定義動作[6]だが，それを検知するためには配列にはそのサイズ情報を付加し，読み書きの際には添字チェックを行う必要がある．最適化である程度省くことができるかもしれないが，そのような余計なコストがかかる可能性があるくらいなら，効率を重視して無視 (これも未定義動作のうちである) し

[4]「未定義として定義」とはこれいかに．

[5]極端な場合，プログラムを実行したユーザのファイルを全て削除されたとしても文句はいえない．(そんな C コンパイラしかなかったら，とても気をつけてプログラムを書くでしょうね．)

[6]Java では `ArrayIndexOutOfBoundsException` という例外が発生すると，Java 言語仕様 10.4 Array Access や 15.10.4 RunTime Evaluation of Array Access Expressions に規定されている．添字のチェックをするために各配列オブジェクトは長さの情報を持っている．

てしまえ，というのが C 言語系の言語の設計思想である．ここはトレードオフでもあるが，C 言語がよく使われていた時代に比べ，未定義動作のリスクも大きくなってきている．特に不正な領域を読み書きできてしまうことによって発生するセキュリティの問題は深刻である．(効率はある程度諦めて実行時にエラーを検出しよう，という研究や，型システムを工夫して未定義動作が発生するのを未然に防ごう，という研究が 1990 年代後半から盛んになった．Rust などの新しい低水準言語は，そういった研究の賜物である．)

未定義動作と紛らわしいのが **未指定動作 (unspecified behavior)** である．これは，いくつかの可能性があるがどれを選ぶかは処理系次第，というもので，例えば C, OCaml, Scheme では，関数呼び出しで実引数が複数ある場合に，それらの評価順序は未指定動作となっていて，どの順番で実行されるかわからない．ということで，特定の評価順序に依存したプログラムは書くべきではない．

ちなみに，Java はどのプラットフォームでも同じように動作することを目指して設計されていることもあって，基本的に未定義動作や未指定動作は (意図しない仕様の記述漏れを除いて) 存在しない．

2.5 ケーススタディ

- Java は言語仕様 Chapter 1 のわりと早いうちに「コンパイル時エラー」についての記述がある．実行時のエラーは例外機構に組込まれており Chapter 11 に記述されている．
- OCaml はエラーについてのまとまった記述はあまりない．
- C 言語については 3.4.3 undefined behavior, 3.4.4 unspecified behavior に定義があるので，一度読んでみるとよいだろう．
- Scheme についても R5RS の 1.3.2 Error situations and unspecified behavior という節がある．