4章　文法

スライド1

　プログラミング言語は，厳格に定められた文法によってその形式を表現することができます．ここでは，まず初めに文法と言語との関係について勉強します．そして，コンパイラのなかで言語が果たす役割について，特に構文解析のなかで果たす役割について勉強します．さらに，文法を記述するBNF（バッカス・ナウア記法：Backus Naur Form）とプログラムがもつ構文の表現方法について勉強します．文法はそれぞれの言語の特徴を簡潔に表現することができる一つの数学的な道具でもあります．この章の勉強を終えたとき，文法がもつ広がりと奥行きのすばらしさをきっと感じ取ることができると期待しています．

スライド2

　私たちはこれまでにコンパイラを開発するにあたっては，まず初めにそれぞれのプログラミング言語にそって書かれた文字列からトークンを抽出する字句解析が必要であることを学んできました．そして，次に，それらのトークンの間の計算順序を含む関係を表す構文解析が必要であることを学んできました．そして，最後に，その関係にデータの型等も含めた意味付けを行う意味解析が必要であることを学んできました．コンパイラにおいては，このような一連の解析を終えたのち，その結果をもとにオブジェクトコードの合成が始まります．この章で勉強する文法は，この構文解析を進めるための重要な数学的な道具です．

スライド3

　言うまでもなく，プログラミング言語は厳密に定義されていなければなりません．例えば，C言語で書かれたa=b+c が実行されたとき，どのコンピュータを使っても，どのOSを使っても，その結果が同じでなくてはなりません．したがって，そこには，プログラムの書き方を厳格に定め，その意味を正しく理解することが必要となります．これらを規格といいます．この規格には，このスライドが示すように，文字の並びを規定する構文規則（syntax rule）と，文字の並びが持つ意味を定める意味規則（semantics rule）があります．しかし，先ほども言ったように，規格はプログラムの実行規則を規定しますが，その実行方法までを，例えばどのコンピュータやOSを使うか，どの程度の実行時間を要するか，どのような識別子を使って変数を記述するか，などまでを規定するものではありません．

スライド4

　構文規則自体もそれぞれのプログラミング言語の考え方の違いやコンピュータで使用できる文字の制限などで，大きく違います．例えば，代表的な演算の例である代入や比較についても言語によって大きく違っています．また，同じプログラミング言語であっても，ビット数の違いによって整数の範囲や文字コードが違うこともあります．したがって，これらを最小限に統一するための規範や基準を定めることもあります．

スライド5

　これは，同じC言語のプログラムであっても処理系によって結果が異なることを表す一例です．プログラム1およびプログラム2を実行させたときの結果は一般にスライドに書かれているようになります．しかし，一部の処理系では，スライドの下部にあるような結果が生じることもあります．これは，インクリメント演算子++が一つの式のなかで同一変数として複数個所に使用されたとき，変数の値が更新される時期までが規格で定められていないためです，したがって，このようなことは決して望ましいことではありません．

　このようにプログラミング言語の規格を定めることは，コンパイラを構築する上で極めて重要なことです．

スライド6

　次にプログラミング言語の文法を厳密に定義することを考えます．

スライド7

　文法Gは形式言語とも呼ばれ，このような4項組

G=(Σ,T,P,S)

で表されます．なお，本によってはその定義を

G=(N,T,P,S) N:非終端記号の集合

とするものも多くありますが，その内容は全く同じものであります．

　生成規則の一例として，スライドに示すような生成規則から構成される文法を考えてみましょう．これは括弧をもつ四則演算式を表す一つの文法でもあります．

　以下ではこれらの規則を見易く簡略化するため，例えば，

の3つの規則をまとめて，

と書くこととします．ここで，記号｜は のいずれか1つを選択することを表しています．

スライド8

　次にこの生成規則を使って，開始記号Sから記号を書き換えていくことを考えてみましょう．この書き換えていく操作を導出と言います．この導出については，この授業の初めに勉強した言語理論の基礎においても出てきたはずです．

スライド9

　導出は，生成規則の左辺に含まれるどれか一つの非終端記号に，その非終端記号を左辺にもつ規則を使って，それを書き換えていく操作を表します．すなわち，記号の並びのなかのいずれか一つの非終端記号に規則を使って新しい記号の並びを作り出すものであります．そして，これを繰り返すことによって，最終的に終端記号のみからなる記号列を作り出すことが導出の目的です．簡単な導出例をこのスライドに示します．この規則によって，中央にzが，その左にxが，右にyがそれぞれ同数だけ並んだ無限個の記号列を作り出すことができることは容易に分かるはずです．

スライド10

このスライドに式を，+，-，\*，/と( , )からなる四則演算の式を導出する生成規則の一例を示します．スライドに記載されている導出例は3変数の和を表す記号列を導出したものです．ここで，Eは式（Expression），Tは項（Term），Fは因子（Factor）と呼ばれている非終端記号を，vは変数（Variable），nは整数（Number）と呼ばれている終端記号です．このスライドに一つの導出例を示します．これより，v+v+n（変数＋変数＋整数）という型の式がこの文法によって導出できることが分かります．

これらの規則によって導出できる記号列の例と，これらの規則からは導出できない記号列の例をスライドに示します．これより，この文法が四則演算を定義するものであれば，プログラムのなかに右のような記号列があればそこでエラーが発生することとなります．

スライド11

　アルファベットと数字および下線から構成される識別子を定義してみましょう．アルファベットか下線で始まる識別子はこのスライドにある生成規則で定義できることとなります．

浮動小数点も同様に規則を使って定義することができます．浮動小数点の定義は以前に紹介したようになかなか厄介なものです．例えば，.25のような記述を許すのか許さないのかなどのような難しいことは考えず，一般的なものを定義したものをこのスライドに示します．ここで，εは空の文字列を表しています．

このように文法を使って対象を定義するにあたっては，そこに本来あってはならないものが含まれていないかどうか，逆に必要なものがすべて含まれているかどうか，を慎重に検討することが必要となります．

スライド12

　以上で構文解析に必要な文法を凡そ見てきました．それでは次に構文解析結果をどう表現するかについて考えてみましょう．

スライド13

　字句解析によって抽出されたトークンが織りなす計算順序を含む構文解析の結果は，構文木と呼ばれる木によって表現することができます．そこでは，文法の解釈にあいまいさがなければ，導出過程にかかわらず，その構文木は一意に決まります．このことは非常に重要なことです．これについて，例を使って見てみましょう．

スライド14

構文木作成の一例として，このような生成規則からなる文法を考えてみましょう．開始記号Sから始めて，規則を適用するたびにそれぞれの規則の左辺の記号（非終端記号）を親ノードに，右辺の記号（非終端記号と終端記号にかかわらず）を子ノードにもつ木を繰り返し作っていきましょう．例として，スライドにある規則のもとで，スライドの右上のような導出過程，すなわちそこに記載された順序で規則を適用していったとき作られる木は図4.1(a)のようになります．このような木を構文木と呼んでいます．

この文法では，これとは違う導出過程を適用したときの構文木もやはり図4.1(a)のようになります．これについては，各自で試しておいてください．このことは先ほども示したように非常に重要なことです．一般に，文法の書き方にあいまいさがなければ，導出過程が違っていても同じ構文木が得られることは，数学的にも証明されてもいます．

　なお，(a)の構文木から非終端記号のノードで子が1つのものを取り除くと(b)のような木となります．さらに非終端記号のノードを子ノードで置き換えると(c)のような木となります．この木はまさに，2+1\*3の計算順序を表しています．すなわち，この木は，先に1\*3を計算し，次に2とその結果を加えることを示しています．このように文法は単に文字列を指定するだけでなく，導出過程を木で表すことによってその構造も表していることとなります．

スライド15

　では，どのような文法であっても，導出過程によらず構文木は一意に定まるのでしょうか．試しに，スライドに示したこのような生成規則を考えてみます．このとき，スライドにあるような2つの異なる導出過程(a)と(b)を考えると，これらから作られる構文木はそれぞれスライドに描かれたような構文木となります．ここで，(a)の構文木は先に掛け算\*を行った後，足し算+を実行することを示しています．一方，(b)の構文木は先に足し算+を行った後，掛け算\*を実行することを示しています．しかし，前の例でも示したように，本来ならば構文木はその導出過程が違っていても同じものになるはずでした．

　では，どうしてこのような導出過程の違いが異なる構文木を生み出したのでしょうか．それは，この文法の解釈にあいまいさがあったためです．今，2つの規則 に注目してみましょう．この文法のなかでは，この2つの規則は同等に扱われています．したがって，この生成規則では，「掛け算が足し算に優先される」という計算の根本が表現しきれていない，あいまいな解釈が残っていることが分かります．一方，先ほどのスライドにある生成規則では，「掛け算が足し算に優先される」ことが明確に謳われています．

この例のように，プログラミング言語の記述にあたっては，このようなあいまいさは極めてまずい，望ましくないこととなります．このように，文法を示すにあたっては，複数の解釈ができる記述は避けなければなりません．

スライド16

　通常のプログラミング言語ではこれまで示してきた記法ではなく，バッカス・ナウア記法あるいはそれを拡張した形式でその構文が定義されます．バッカス・ナウア記法はジョン・バッカスとピーター・ナウアがALGOL60の文法定義のために考案したプログラミング言語のための記法です．

スライド17

　バッカス・ナウア記法は，文脈自由文法，これは以前に勉強したように，生成規則の左辺がただ一つの非終端記号からなり，右辺が非終端記号と終端記号から構成される記号列からなる規則で定義される文法でしたが，この文脈自由文法の定義に用いられるメタ言語（言語そのものやその言語がもつ情報を定義するための言語）の一つでり，今ではプログラミング言語の定義に利用されています．

スライド18

　式をBNFで定義した例をこのスライドに示します．ここで，規則の左辺に書かれている記号はすべて非終端記号であり，これら以外の\*,/,+/-(,)letter,digit　は終端記号です．このスライドにある上から11個の規則は，この章の初めで勉強した生成規則そのものであります．また，下から5個の規則は，変数や定数を表す識別子と数を定義したものです．

　この文法から生成できる式の例と，生成できない式の例をスライドに示します．たったこれだけの規則で，括弧をもつ四則演算のどのような式も定義できることは驚くべきことでしょう．あっぱれ，文法！

スライド19

　この章の最後に，括弧を使わずに簡単な形式で，それでいて演算子の優先順位を保存した形で構文木を記述することを考えてみましょう．このことは，コンピュータで構文木を保存する上で，極めて重要なことです．

スライド20

　今，一つの例として，このスライドにあるような

　　　w=(a+b/c)\*(d-e)

を考えてみましょう．この式の構文木はスライドに示すようになります．この式は演算子を変数の中間において式を記述しています．このような記述を中置記法と言います．構文木は木の構造自体が処理の手順を示しているため，木に従って計算を進めることができます．そこで，今，構文木を根からたどって，括弧を使わずにすべてのノードを印刷することを考えてみましょう．

　まず初めに演算子の前後にオペランドを置く中置記法です．この構文木を中置記法で括弧を使わずに書くとこのスライドのように，w=a+b/c\*d-eとなります．しかし，この中置記法で書かれた文字列から元の構文木を必ずしも一意に復元することはできません．これについては皆さんで試してみてください．したがって，中置記法は決して望ましいものではありません．

　そこで次に，先に自分自身のノードを表し，次に左の子，そして右の子を表す記法，すなわち最初に演算子，そして次に2つのオペランドを表す前置記法でこの構文木を表すとこのように，=w\*a/bc-de となります．この文字列からは元の構文木を一意に復元することができます．したがって，この前置記法を使えば正しい計算順序を保存することができます．

　最後に，構文木を最初に左の子，次に右の子，最後に自分自身の順にノードを並べた後置記法を考えてみましょう．これはオペランド，オペランド，演算子の順でノードを並べたものです．スライドにある構文木をこれで表現すると，スライドのように，wabc/+de-\*= となります．この文字列からも元の構文木を復元することが可能です．

　このように，前置記法や後置記法を使えば括弧なしで処理順序を一意に定めることが可能となります．ただし，二項演算子と単項演算子で同じ+や-の記号を使うと区別ができなくなるため，異なる記号を使うことが必要となります．

　この章ではプログラミング言語が簡単な文法で記述できること，および構文解析がその文法を使った導出過程によって表現できることを見てきました．文法は単にプログラミング言語の形式を規定しているだけでなく，その導出過程を調べることによって計算順序を含む構文解析ができることが分かってもらえたことと思います．まさに文法の底力を見たような気がします．

次の章では，いよいよコンパイラの中心課題を，この構文解析を機械的に，文法から自動的に構文解析を行うことを考えてみましょう．