5章　下向構文解析

スライド1

　オートマトン理論と言語理論はコンパイラの合理的な設計・開発を支える重要な基盤技術であり，そこにはいかなる不確かさ・曖昧さも入る余地はありません．この章では，一連のコンパイラの設計・開発のなかでも最も理解が困難である構文解析について，その厳密さよりもむしろその考え方が直感的に分かることをめざして勉強を進めていきましょう．構文解析は理解が困難と言っても，その本質は常識的であり自然な考え方に基づくものであります．途中であきらめることなく，前に進みましょう．

スライド2

　例によって例の図です．コンパイラ全体のなかでのこの章の位置を，まずは確かめておきましょう．ソースプログラムから機械語で書かれたオブジェクトコードを生成するコンパイラを実現するあたっては，まず初めに入力された文字列からトークンを抽出することが必要でした．この字句解析は，原理的にはオートマトンを使って行うことができることをこれまでに見てきました．また，前章では，それぞれのプログラミング言語がもつ規格は，言語によって厳格に与えられることを見てきました．これらをもとに，この章ではコンパイラの中心をなす構文解析について勉強していきます．そこでは，対象とするプログラミング言語を規定する文法のもとで，それの開始記号から目標とするトークン列に至る導出の過程を見つけ出すことによって，言い換えれば開始記号からどの生成規則をどのような順序で適用していくとめざすトークン列に至るかを調べることによって，計算順序等を含む構文解析が行われることとなります．以下では，これについて考えてみましょう．

スライド3

　最初にいくつかの用語を定義しておきます．

　まず初めに，あらためて構文解析とは，与えられたトークン列のもとで，開始記号から終端記号に至る導出過程を決定することと定義します．第4章では，いくつかの導出例を見てきました．そこでは，まず初めに，開始記号から対象とするトークン列に至る導出過程があれば，そのトークン列はそのときの文法が示すプログラミング言語の規格を満たしていることを，そうでなければそのトークン列はそのプログラミング言語の文法に沿っておらずエラーとなることを示してきました．さらに，開始記号からめざすトークン列に至る導出過程が存在するならば，その導出過程を表す構文木を通じて，各トークンの処理順序等が解析できることも示してきました．これらについては，もう一度，この章で見直すこととします．

　次に，構文解析を行うにあたり，構文木をの根に相当する開始記号からそれの葉に相当する終端記号に向かって，すなわち，構文木を上から下に向かって構築していく解析方法を下向構文解析と言います．これとは逆に，下から上に向かって構文解析を行う解析方法を上向構文解析と言います．これについては第7章で勉強します．

　さらに，下向構文解析のなかで，構文解析プログラムが再帰的手続きで構成されたものを再帰下降法と言います．まずは，この再帰下降法について紹介します．

スライド4

　今，式を定義する文法が与えられたとして，式を解析する手順を考えてみましょう．ただし，この構文解析の段階においては，文字列は字句解析によって既にトークンに分割されているものとします．したがって，以下では，このトークンの列が与えられたとのもとで，開始記号からそれに至る導出の過程を求めることとします．

例として，このスライドにある生成規則を考えます．人は演算子の優先順位などを考え，どこが先に処理されるべきかなどを容易に認識することができます．しかし，コンピュータにとってはこのことはかなり難しい問題となります．そこで，単純に文法に従って，何が欲しいかを要求しながら解析する手順を考えてみましょう．今，この文法において，Eが何をすべきかを考えると，　の2つの規則から， が構成できることが分かります．したがって，Tを認識したら，その後に＋が続く限りTの処理を続ければよいこととなります．このため，Eはスライド中の“Eの処理”に記載されているような処理を行うこととなります．同様に，　の2つの規則から， が構成できるため，TはFを認識したら，その後に\*が続く限りFを処理すればよいこととなります．一方，Fを左辺にもつ3つの規則 から，それぞれの規則を適用すると最初の記号はn,　－,　( のいずれかとなるため，次のトークンを見てどの規則を適用するのがよいかが分かることとなります．したがって，その際，これら以外のトークンが来れば，入力に誤りがあり文法に違反していることとなります．以上より，Fの処理はこのスライドのようになります．

このようにこの生成規則では，Eの処理がTを呼び，Tの処理がFを呼び，Fの処理がEを呼び出すこととなります．したがって，この解析は，解析手順が再帰的手続きで構成された再帰下降法であることが分かります．

スライド5

　以上のような考察をもとに，先の生成規則をもつ文法の構文解析は，これまでに述べてきた手順をプログラム化すれば実現できることになります．こうして作成された構文解析プログラムを，このスライドと次のスライドに示します．ここで，このスライドが示すプログラムは第3章で紹介した字句解析ツールlexを使って作られたlexプログラムを，次のスライドにあるプログラムは前のスライド4に示した内容をC言語を使って書いたものです．

　このスライドにある字句解析部分では，6行目以降に正規表現に準じたlexの記述形式に従って，それぞれのトークンが定義されています．これによって，入力された文字列からそれぞれのトークンが抽出されることとなります

スライド6

抽出されたトークンは，このスライドに示したプログラム中の関数get\_tokenによって取り出されるものとします．すると，先ほどのE,Fの処理はそれぞれこのようなCプログラムとして実現されることとなります．ここで，青い部分がEの処理を，赤い部分がFの処理を，緑の部分がTの処理を，橙色の部分がBの処理を表しています．また，Eの処理のなかの＋Tの繰り返しは加算代入演算子+=で，Tの処理のなかの\*Fの繰り返しは乗算代入演算子\*=で実行されます．

この例は字句解析ツールlexで抽出されたトークン列をもとに，それが+のトークンであればEの処理を，\*のトークンであればTの処理を，値nのトークンであればFの処理を行う形での構文解析の手順をCプログラムで実現したものです．

スライド7

今の例が示すように，再帰下降法では，人が文法を解釈して構文解析を行いました．しかし，これでは作成されたコンパイラが正しいかどうかが不明確となり，誤りも起きやすくなります．そこで，次に文法から直接コンパイラを生成する方法について紹介します．が，その前に，今の例では演算子が+と\*の2つでしたが，より多くの優先順位をもった演算子があるときの扱いについて見ておきましょう．

スライド8

　正しい演算子の優先順位を保存した構文木を作成するにあたっては，一般にこれらの例が示すように前後の演算子の優先順位を考えながら木を構成しなければなりません．

例えば，スライドのなかの(1)の場合（＾2+3\*5$）において，演算子＊を受け取ったときを考えてみましょう．演算子＊の直前の演算子＋の優先順位は今受け取った＊よりも優先順位が低いため，先にこの＊を処理しなければなりません．このためには＊を含む新しい木（赤い木）を構成することが必要となります．

　一方，(2)の場合（＾2\*3+5$）において，演算子＋を受けとった場合を考えると，これの直前の演算子＊の優先順位は＋より高いため，これを先に処理した後＋を処理できるように新しい部分（赤い箇所）をスライドにあるように既にあった木（黒い箇所）にまとめることとなります．

　このように構文木の作成にあたっては，前後の演算子の優先順位を考慮することが重要となります．

スライド9

優先順位が異なる多くの演算子が混在するなかで，このような対応を機械的に行うにあたっては，このスライドの左の表にあるように，優先順位の強さを表す整数をあらかじめ定めておくことが現実的です．構文木はこの優先順位を使って作成されることとなります．

しかし，プログラミング言語によっては，このような単純な優先順位付けでは正しく処理できないこともあります．例えば，

　　　a=b=0 bに0を代入しそのｂをaに代入したい

を場合によっては

　　　(a=b)=0

と解釈してしまうこともあります．これは通常の演算子は左結合ですが，代入やべき乗は一般に右結合であることによるためです．このため，演算子が左にきたときと右にきたときを区別して，2つの優先順位を設けることが考えられます．このスライド中の右の表はこのような考えに基づいて定義されたものです．このように，演算子の優先順位は意外と難しいものです．

スライド10

　それでは，いよいよこの章の中心部分である文法から直接コンパイラを生成する方法について考えてみましょう．

スライド11

　再帰下降法では人が文法の意味を解釈し，それに合わせて処理を記述することが必要でした．しかし，この方法では，解釈の誤りやあいまいさによって，コンパイラが意図しない動作をしてしまう危険性があります．さらに，さまざまな言語に対して，構文解析して構文解析システムを作成することが必要となります．このため，文法から直接コンパイラを生成する解析方法が考え出されました．これを直構文解析と言います

　直構文解析の代表的なものとして，LL構文解析があります．これの考え方は分かり易いものであり，次に入力するトークンである終端記号を見ながら，その終端記号を先頭に来させる可能性のある生成規則を順次選ぶことによって導出過程を決定しようとするものです．この方法は，生成規則と，各規則ごとに予め求めておいたその規則を適用したとき先頭にくる可能性のある終端記号の集合，これを先頭記号集合と言いますが，この生成規則と先頭記号集合を使いながら，適用する規則を順次決定していこうというものです．これについては，後で例を使って説明します．

　なお，この解析法をLL構文解析と言いますが，その由来は入力文字列を左（Left）から構文解析していき，左端から導出（Leftmost Derivation）を行うことにあります．

スライド11

　LL構文解析の一例として，このスライドにある生成規則Pを考えてみましょう．ただし，この生成規則Pにおいては，例えば，のように規則の右辺にすぐに自分自身が現れると，これを左再帰と言いますが，この左再帰が自分自身の定義を繰り返し呼び出すため，このままでは解析ができなくなります．したがって，このような場合は，PをPと同じ言語を生成する左再帰でない生成規則P’に予め変換しておくことが必要となります（P’では，どの規則も左辺と右辺に同じ記号を持っていないことに注意）．以下では，Pと同等な左再帰でない生成規則P’を考えてみましょう．

スライド13

　次に，P’に含まれる生成規則ごとに，先ほど説明した先頭記号集合を求めておきます．例えば，規則 の先頭記号集合 は，この規則から導出できるすべての終端記号の並びのなかで先頭にくる可能性のある記号から作られる集合を表しています．例えば，この例では，

より，

となることが分かります．生成規則P’に含まれる10個のそれぞれの規則の先頭記号集合はスライドのようになります．これについては，各自で確かめておいてください．

　なお，これの機械的な求め方については，教科書P.57の「5.3.2先頭記号集合の作成手法」を見ておいてください．

スライド14

以上によって，解析のための準備は整いました．

例として，

　　入力列（トークン列）：　2+3\*4+5

の構文解析を進めてみましょう．

①最初に開始記号Sを左辺にもつ規則が，適用可能性のある規則の候補として選ばれます．ここで，入力列2+3\*4+5の先頭文字の2は

であることより，規則を選ぶと最初に2がくる可能性があることが分かります．

パワーポイントをクリック

②次にEを左辺にもつ規則を次に適用する規則の候補とします．この規則の先頭記号集合は

を満たすため，次に適用できる規則として，が選ばれます．

パワーポイントをクリック

以下，同様に先頭記号集合を見ながら2がそれに含まれる規則の選択を続けていきます．

パワーポイントをクリックを続ける

このような手続きを進め，それ以上選択できる規則がなくなったならば，次の終端記号+（2+3\*4+5の2つ目の記号）に移ります．そして，同じ手続きを進めます．構文木はこのような手順を経て自動生成されていきます．

　このスライドの最後（パワーポイントをクリックをすべて完了したとき）に，最終的に得られる構文木を示しておきます．このようにLL構文解析は，生成規則と先頭記号集合を繰り返し使いながら進められていきます．

スライド15

　このスライドには，今，説明した導出過程がたどってきた経緯が表形式で書かれています．前のスライドと並行しながら（全く同じことを表しています）あらためてLL構文解析の手順を確認しておいてください．

先頭にくる記号を左から順にたどりながら，その記号を先頭に来させる可能性のある規則を順に調べていくLL構文解析の考え方は，極めて自然で理解し易いものと思われます．もう一度，今の例を追いながら，その考え方を確かめておいてください．このことが第5章の最重要ポイントです．

スライド16

LL構文解析では，次にくる一文字だけに注目して規則を選んできました，では，次にくる文字を2文字まで配慮したらどうなるでしょうか．この章の最後に，これについて考えてみましょう．

スライド17

　生成規則と，先頭からk個の終端記号の列からなる先頭記号集合を用いる解析法をLL(k)構文解析と言います．先ほどまで勉強していたLL構文解析は先頭1個のみの終端記号を扱っていたため，これをLL(1)構文解析と呼ぶこともあります．

スライド18

LL(k)構文解析の詳細は教科書にもほとんど記載されていないため，その内容は省略します．しかし，LL(1)とLL(k)の比較の要点についてはスライドにまとめておきます．その内容は次のようです．

(0) LL(1)では解析できないが，LL(k), k>1 では解析できる言語が存在する．

しかし，

(1) しかし，LL(k)で解析できる言語は，文法を修正することによってLL(1)でも解析できる．

(2) 文法の定義はLL(k)の方がLL(1)よりも容易である．

　この章では，文法からコンパイラを直接生成するための基本的な考え方について勉強しました．教科書第5章は，初めてこれを学ぶ皆さんにとってはかなり分かりづらい内容であったと思います．しかし，そのもとになっている考え方は極めて常識的であり自然な考え方であると考えられます．この種の内容は，細部に陥ることなく，まずは全体を大まかにつかむことが必要です．

第6章は特別な構文解析処理系についての内容であるため，章全体を省略します．

次は第7章です．