2 簡単な1パス コンパイラ

この章は，3章から8章までの導入部にあたり，中置式を後置形に翻訳するCプログラムの作成を例にとって，基本的なコンパイル技法を解説する．この章では，コンパイラのフロントエンド，すなわち字句解析，構文解析，および中間コード生成の3つの部分について述べる．コード生成とコード最適化はそれぞれ9章と10章で述べる．

2.1 概　　説

プログラム言語は，プログラムがどのような形をしているか(言語の**構文**)，そして，それがなにを意味するのか(言語の**意味**)，を記述することによって定義される．構文の規定には，文脈自由文法あるいは**BNF**(Backus-Naur Formの略)とよぶ記法が広く用いられている．意味は，現在使用できるどんな記法を用いても，構文よりずっと記述が困難である．したがって，言語の意味を規定するには，非形式的な記述や暗示的な例を使わなければならない．

文脈自由文法は，言語の構文を規定するだけでなく，プログラムの翻訳の仕方も示唆する．**構文主導翻訳**とよぶ文法指向のコンパイル技法は，コンパイラのフロントエンドを作る上で大きな助けになる．この章ではおもにこの技法を用いる．

構文主導翻訳を説明するために，中置式を後置形に翻訳するプログラムの作成を考える．ここで，後置形とは演算子をその演算数のあとに書いたものをいう．たとえば，式9−5+2の後置形は95−2+である．すべての演算をスタックで行う計算機に対しては，後置記法からそのコードへ直接，変換ができる．はじめに，正符号または負符号で区切った数字の式を考え，それを後置形に翻訳する簡単なプログラムから作ることにしよう．基本的な考え方を理解したあと，このプログラムを拡張し，プログラム言語のもっと一般的な構文要素も処理できるようにする．その際，各翻訳プログラムは，前に作ったものを系統的に拡張していく．

これから作ろうとするコンパイラの**字句解析ルーチン**は，入力文字のストリームをトークンのストリームに変換する．このストリームは，図2.1に示すように，次のフェーズへの入力となる．図の中の“構文主導翻訳”は構文解析ルーチンと中間コード生成ルーチンとを合わせたものである．はじめに数字と演算子だけからなる式を扱うのは，字句解析を簡単にするためである．この場合には，各入力文字が1つのトークンになる．次にこの言語を拡張して，数，識別子，およびキーワードといった字句要素を加える．そして，そのように拡張した言語のために，連続する入力文字を適当なトークンにまとめる字句解析ルーチンを作成する．なお，字句解析ルーチンの構成法は3章で詳しく述べる．

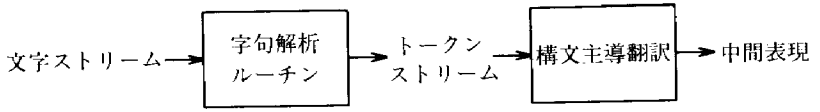


図2.1　コンパイラのフロントエンドの構成

2.2 構 文 の 定 義

この節では，言語の構文を規定するために文脈自由文法(以下，たんに**文法**)とよぶ概念を導入する．コンパイラのフロントエンドの仕様を示すのに，本書では，その仕様の一部として文法を使用する．

文法はプログラム言語における構文要素の階層構造を自然な形で記述する．たとえば，C言語のif-else文は

　　if (式) 文 **else** 文

という形，すなわちif-else文は，キーワード**if**，左かっこ，式，右かっこ，文，キーワード**else**，そして文を順に並べたものからなる(Cには**then**というキーワードはない)．式を表わすために変数*expr*，文を表わすために変数*stmt*を用いると，上の構成規則は次のように表現できる．

　　stmt → **if** (*expr*) *stmt* **else** *stmt*　　(2.1)

ここで，“→”は“…は，次の形式である”，と読めばよい．このような規則を**生成規則**とよぶ．生成規則の中のキーワードifやかっこのような字句要素を**トークン**とよぶ．トークンの列を表わす*expr*や*stmt*のような変数を**非終端記号**とよぶ．

文脈自由文法は次の4つの要素からなる.

1. トークンの集合. 以下, トークンのことを**終端記号**ともよぶ.
2. 非終端記号の集合
3. 生成規則の集合. 各生成規則は, 生成規則の**左辺**とよぶ非終端記号, →, そして生成規則の**右辺**とよぶ終端記号や非終端記号の列からなる.
4. **開始記号**とよぶ1つの非終端記号

文法は生成規則の並びによって定め, 一般的な約束に従って, 開始記号に対する生成規則は並びの先頭に置く. 数字, <=のような符号, **while**のような太字の文字列は終端記号とする. また, イタリック体の名前は非終端記号とし, それ以外の字体の名前や記号は終端記号†とする. 生成規則を簡潔に示すために, 左辺に同じ非終端記号をもつ生成規則は, 各規則の右辺をそれぞれ記号 | で区切って1つに書き表わしてもよい. "|" は "または" と読めばよい.

例2.1 この章の例には, 数字, 正符号および負符号からなる式を用いる. たとえば, 9−5+2, 3−1や7である. 正符号と負符号は数字のあいだに書かなければならないから, この式は "正符号または負符号で区切った数字の並び" と表現できる. 次に示す文法はそのような式の構文を記述するものである.

$$list \rightarrow list + digit \tag{2.2}$$
$$list \rightarrow list - digit \tag{2.3}$$
$$list \rightarrow digit \tag{2.4}$$
$$digit \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \tag{2.5}$$

上の生成規則において, 左辺に非終端記号 *list* をもつ3つの規則は, 次のようにまとめて記述することができる.

$$list \rightarrow list + digit \mid list - digit \mid digit$$

上の約束から, この文法におけるトークンは

† イタリック体の1文字の記号は, 4章で文法の詳しい説明をするときに, 別の目的に使用する. たとえば, *X*, *Y*, *Z* は1つの終端記号または非終端記号を表わす記号として用いる. しかし, その場合でも, 2文字以上のイタリック体の名前は非終端記号を表わす.

+ − 0 1 2 3 4 5 6 7 8 9

である. 非終端記号はイタリック体の *list* と *digit* であり, *list* の生成規則が先頭にあるので, 開始記号は *list* になる. □

各生成規則について, その規則は左辺の非終端記号に**対する**ものである, という. また, トークンを0個以上並べたものを**トークン列**という. トークンを1つも含まないトークン列を**空列**とよび, ϵ で表わす.

開始記号から出発して, 非終端記号をそれに対応する右辺で置き換える操作を繰り返していくと, 文法からトークン列が得られる. 開始記号から導出できるトークン列が, その文法で定義する**言語**を形成する.

例2.2 例2.1の文法で定義した言語は, 正符号または負符号で区切った数字の並びからなる.

非終端記号 *digit* に対する10個の生成規則によって, *digit* は, トークン0, 1, ... , 9の中の任意のものを表わす. 生成規則(2.4)から, 数字1つだけでも数字の並びである. 生成規則(2.2)と(2.3)は, 任意の並びのあとに正符号または負符号が続き, そのあとに数字を続けたものも並びになることを表わす.

要するに, (2.2)から(2.5)までの生成規則があれば, いま考えている言語は完全に定義できる. たとえば, 9−5+2が *list* であることは, 次のように説明できる.

a) 9は *digit* であるから, 生成規則(2.4)によって, 9は *list* である.
b) 9は *list* であり, 5は *digit* であるから, 9−5は生成規則(2.3)によって *list* である.
c) 9−5は *list* であり, 2は *digit* であるから, 9−5+2は生成規則(2.2)によって *list* である.

このような説明の仕方は, 図2.2のように木を使って表現できる. 木の各節点には文法記号がラベルとして書き込んである. 内部節点とその子が1つの生成規則に対応する. 内部節点は生成規則の左辺に, また子はその右辺に対応する. この木を**解析木**といい, 次の項で説明する. □

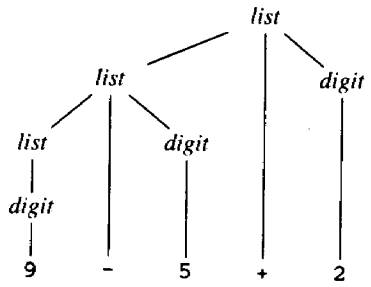


図2.2　例2.1の文法による，9－5＋2に対する解析木

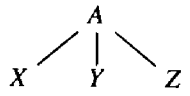
例2.3　いままでの並びとは少し異なるものに，Pascalのbegin-endブロックに現われる，セミコロンで区切った文並びがある．それは，トークン **begin** と **end** のあいだは空の文並びであってもよい，ということである．begin-endブロックに対する文法として，次の生成規則を考えてみよう．

block → **begin** *opt_stmts* **end**
opt_stmts → *stmt_list* | ε
stmt_list → *stmt_list* ; *stmt* | *stmt*

opt_list（"optional statement list"，任意の文の並びの意）に対する右辺には，空列を表わすεがある．そこで，*opt_list* を空列で置き換えると，*block* は2つのトークンだけからなる **begin end** になる．*stmt_list* に対する生成規則は例2.1の *list* に対する生成規則とよく似ている．それは，算術演算子をセミコロンで置き換え，*digit* を *stmt* で置き換えてみればわかる．この例では，*stmt* に対する生成規則は示していないが，*stmt* はif文や代入文などのいろいろな文に対する生成規則を表わすものと考えればよい．それらの文の生成規則はあとのほうで説明する．　□

解析木

解析木は，文法の開始記号から，言語の記号列がどのようにして導出されるかを図示したものである．非終端記号 A に対する生成規則を $A \to XYZ$ とすると，その解析木は A を内部節点とし，X, Y, Z を左から順に A の子としたものからなる．



形式的にいうと，文脈自由文法が与えられたとき，その**解析木**は次の性質をもつ．

1. 根は開始記号をラベルとしてもつ．
2. 各葉はトークンまたはεをラベルとしてもつ．
3. 各内部節点は1つの非終端記号をラベルとしてもつ．
4. 内部節点にラベルとして付けた非終端記号を A とし，A の子を左から順に X_1, X_2, ..., X_n とすると，$A \to X_1\ X_2 \cdots X_n$ は生成規則である．ここで，X_1, X_2, ..., X_n は終端記号または非終端記号を表わす．特殊な例として，$A \to \epsilon$ であれば，A の節点は，ε をラベルとする子をもつ．

例2.4　図2.2で，根には，例2.1の文法における開始記号 *list* が付いている．根の子には，左から右へ順に，*list*, ＋および *digit* というラベルが付いている．上の性質4から，例2.1の文法を見てみると，たしかに

list → *list* ＋ *digit*

は生成規則である．根の左側の子を見ても，－についての同じパターンの繰返しである．また，*digit* というラベルをもつ3つの節点は，それぞれ数字をラベルとする子を1つずつもつ．　□

解析木の葉だけを左から右へ読んでいくと，木の**結果**が得られる．これは解析木の根に付けられた非終端記号から**生成**される，あるいは**導出**される記号列である．図2.2から生成される列は9－5＋2である．この図では，すべての葉を一番下に示してあるが，これからは，葉を必ずしも揃えて書くとはかぎらない．ある親の子を a, b とし，a は b の左側にあるものとすると，その親から生成される記号列の中で，a の子孫は，必ず b の子孫たちよりも左に現われる．この性質から，どのような木でも，葉には左から右へという自然な順序が付く．

文法から生成される言語は，解析木から生成できる記号列の集合である，と定義してもよい．与えられた記号列に対して，その解析木を求める過程を**構文解析**という．

曖昧さ

文法に従って，記号列の構造を考えるときには，次の点によく注意しないといけない．解析木からは，葉だけを順に読み取っていけば，1つの記号列が得られるけれども，文法からは，同じ記号列を生成する解析木が2つ以上できてしまう場合がある．このとき，その文法は**曖昧**であるという．文法が曖昧であることを示すには，2つ以上の解析木をもつ記号列を見つければよい．1つの記号列に対して，解析木がいくつも存在すれば，その記号列は，通常，何通りかの意味をもつので，そのような記号列を取り扱うには，曖昧さのない文法を作るか，あるいは曖昧さを解決するために別の規則を設けるかしなければならない．

例2.5　例2.1で数字も並びも区別しなかったとして，次のような文法を作ったとしよう．

string → *string* + *string* | *string* − *string* | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

digit は *list* の特殊な場合なので，*digit* と *list* の概念を合わせてしまって，上の生成規則のように非終端記号 *string* としても一見よさそうである．

しかし，そうすると，式 9−5+2 については図2.3に示すように 2 通りの解析木が考えられる．これらの木に対応する式を，かっこを使って示すと，それぞれ(9−5)+2 と 9−(5+2)の意味になる．もとの式の値は，ふつう 6 になるのに，2 番目の式では 2 になってしまう．図2.1の文法に従えば，このような解釈は生じない．　□

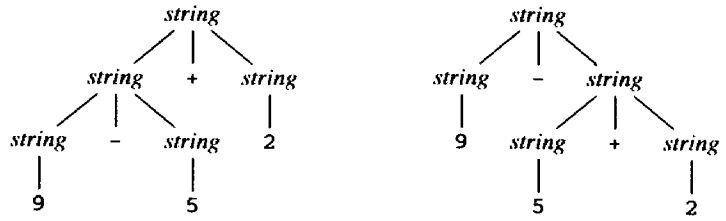


図2.3　9−5+2に対する2つの解析木

演算子の結合性

慣習で，9+5+2 は(9+5)+2 と等価であり，9−5−2 は(9−5)−2 と等価である．これらの式の中の 5 のように演算数の両側に演算子があるとき，どちらの演算子がその演算数をとるかを約束しておかなければならない．演算数の両側に演算子＋があるとき，その演算数は左側の演算子と結び付くので，＋は**左結合**であるという．ほとんどのプログラム言語で，加減乗除の 4 つの演算子は左結合である．

演算子のなかには，べき乗演算のように右結合のものもある．また，C 言語の代入演算子 = も右結合である．たとえば，式 a = b = c は a =(b = c)と同じに扱われる．

a = b = c のような右結合の演算子からなる並びは，次の文法で生成できる．

right → *letter* = *right* | *letter*

letter → a | b | ⋯ | z

−のような左結合の演算子の解析木と，= のような右結合の演算子の解析木との対比を図2.4に示す．ここで，9−5−2 の解析木は左さがりに，a = b = c の解析木は右さがりになる点に注意してほしい．

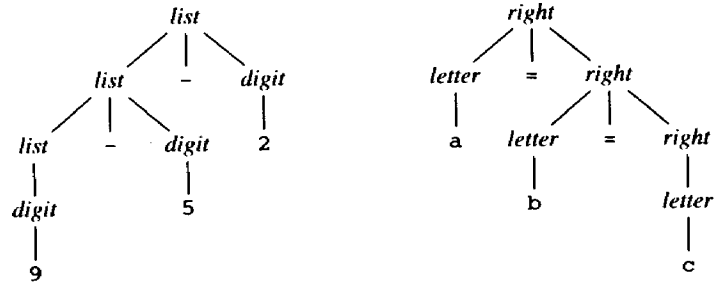


図2.4　左結合と右結合の演算子に対する解析木

演算子の順位

式 9+5 * 2 は，(9+5) * 2 と 9+(5 * 2)の 2 通りの解釈が考えられる．この曖昧さは＋と*のそれぞれの結合性では解決ができない．これを解決するには，各演算子について，それらのあいだでの相対的な順位を決める必要がある．

＋よりも*が先に演算数をとるとき，*は＋よりも**順位が高い**という．数学では，乗除のほうが加減よりも順位が高いのが常識である．したがって，9+5 * 2 と 9 * 5+2 はどちらも*が演算数 5 をとる．すなわち，これらの式はそれぞれ 9+(5 * 2)と(9 * 5)+2 と等価である．

式の構文　算術式に対する文法は，演算子の結合性と順位とを示す表から作ることができる．そこで，よく用いる 4 つの算術演算子について，その順位表をもとに文法を作ってみよう．順位表は次のように，演算子の順位の低いものから示し，同じ順位のものは同じ行に示す．

左結合：　+ −
左結合：　* /

2つのレベルの順位に対して，それぞれ非終端記号 *expr*(式)と *term*(項)を用い，さらに式の基本単位を生成するために非終端記号 *factor*(因子)を用いる．この基本単位は，いま考えている例では，数字，あるいはかっこでくくった式である．

factor → **digit** | (*expr*)

次に，一番順位の高い二項演算子*と/を考える．これらの演算子は左結合であるから，その生成規則はそれらの並びを左側に結合したものとなる．

term → *term* * *factor*
　　| *term* / *factor*
　　| *factor*

同様に，*expr* は項を加減演算子で区切った並びを生成する．

expr → *expr* + *term*
　　| *expr* − *term*
　　| *term*

以上から，次の文法が得られる．

expr → *expr* + *term* | *expr* − *term* | *term*
term → *term* * *factor* | *term* / *factor* | *factor*
factor → **digit** | (*expr*)

この文法は，+か−で区切った項の並びを式と定義し，項はさらに*か/で区切った因子の並びと定義している．かっこでくくった式も因子であるから，かっこを使えば，任意の深さの式(あるいは，任意の高さの木)が作れる．

文の構文　たいていの言語はキーワードから文の認識ができる．Pascal の文は，代入文と手続き呼出しをのぞいて，必ずキーワードで始まる．Pascal の文を定義する文法の例を以下に示す．ただし，トークン **id** は識別子を表わすものとする．

stmt → **id** := *expr*
　　| **if** *expr* **then** *stmt*
　　| **if** *expr* **then** *stmt* **else** *stmt*
　　| **while** *expr* **do** *stmt*
　　| **begin** *opt_stmts* **end**

非終端記号 *opt_stmts* は，例2.3の生成規則で使用したが，文をセミコロンで区切った並びであり，空であってもよい．

2.3　構文主導翻訳

プログラム言語の構成要素を翻訳するには，各要素に対するコード以外に，多くの情報を保持しておかなければならない．たとえば，構成要素の型，目的コードにおける最初の命令の格納場所，あるいは生成した命令の個数などが必要になる．これらの情報は抽象化して，その構成要素の**属性**とよぶ．属性は，たとえば，型，文字列，メモリの記憶場所など任意の数量を表わすものでよい．

この節では，プログラム言語における構成要素の翻訳を規定するために，構文主導定義とよぶ定式化を説明する．構文主導定義は，構文要素に属性を結び付け，その属性を用いて翻訳の仕方を規定しようというものである．あとの章では，この定義を用いて，コンパイラのフロントエンドにおけるいろいろな翻訳処理を規定する．

さらに，この定義のほかに，翻訳スキームとよぶ手続き的な考え方も導入する．この章では，中置式から後置記法の式への翻訳に翻訳スキームを用いる．構文主導定義とその実現は 5 章で詳しく述べる．

後置記法

式 E の後置記法は次のように帰納的に定義できる．

1. E が変数または定数であれば，E の後置記法は E 自身である．
2. 任意の二項演算子を *op* としたとき，E が E_1 *op* E_2 という形の式であれば，E の後置記法は，E_1' E_2' *op* である．ここで，E_1' と E_2' はそれぞれ，E_1 と E_2 の後置記法である．
3. E が(E_1)という形の式であれば，E の後置記法は E_1 の後置記法そのものである．

後置記法では，演算子の位置と項数(引数の個数)とによって，中置式の解釈が一意に定まるから，かっこは不要になる．たとえば，(9−5)+2 の後置記法は 95−2+に，9−(5+2)の後置記法は 952+−となる．

構文主導定義

構文主導定義は，入力の構文構造の規定に文脈自由文法を使用する．この定義では，文法記号に属性の集合を結び付け，各生成規則に意味規則の集合を結び付ける．意味規則は，生成規則の中に現われる記号について，属性値の計算の仕方を定める規則である．このとき，構文主導定義は文法と意味規則の集合とによって構成される．

翻訳は入力から出力への変換である．入力 x に対する出力は次のように規定する．最初に，x に対する解析木を作成する．解析木の節点 n の文法記号を X とし，その節点における X の属性 a の値を $X.a$ で表わす．n における $X.a$ の値を求めるには，n で用いた X の生成規則について，その規則と結び付けられている属性 a の意味規則を用いる．解析木の各節点における属性値を表示した木のことを**属性付き解析木**という．

合成属性

解析木の節点における属性の値が節点の子の属性値から定まれば，その属性を**合成属性**という．合成属性には，解析木を上向きに1回巡回するだけで評価ができるという，ありがたい性質がある．この章では合成属性だけを用い，相続属性とよぶ属性は 5 章で説明する．

例2.6　＋または−で区切られた数字の式に対して，それを後置記法へ変換するための構文主導定義を図2.5に示す．各非終端記号には，文字列を値とする属性 t が結び付けてある．この属性は，解析木中の非終端記号から作り出される式の後置記法を表わす．

数字の後置形は数字自身である．たとえば，生成規則 $term \rightarrow 9$ の意味規則から，解析木の節点にこの生成規則を適用すれば，$term.t$ はつねに 9 となる．生成規則 $expr \rightarrow term$ を適用するときは，$term.t$ の値が $expr.t$ になる．

生成規則 $expr \rightarrow expr_1 + term$ は演算子＋を含む式を生成する(1つの生成

生成規則	意味規則
$expr \rightarrow expr_1 + term$	$expr.t := expr_1.t \parallel term.t \parallel '+'$
$expr \rightarrow expr_1 - term$	$expr.t := expr_1.t \parallel term.t \parallel '-'$
$expr \rightarrow term$	$expr.t := term.t$
$term \rightarrow 0$	$term.t := '0'$
$term \rightarrow 1$	$term.t := '1'$
...	...
$term \rightarrow 9$	$term.t := '9'$

図2.5　中置形から後置形への翻訳のための構文主導定義

規則の中に $expr$ が 2 回現われるので，右辺の $expr$ には添字を付け，左辺の $expr$ と区別する)．＋の左側の演算数は $expr_1$ によって与えられ，右側の演算数は $term$ によって与えられる．この生成規則に付けた意味規則

$$expr.t := expr_1.t \parallel term.t \parallel '+'$$

から，$expr.t$ の値は，後置形 $expr_1.t$ と $term.t$ とを連結し，それに＋を連結したものになる．意味規則の中の演算子 ∥ は文字列の連結を表す．

図2.2の木に対する属性付き解析木を図2.6に示す．各節点での属性 t の値は，その節点に用いた生成規則をもとに，それと結合されている意味規則に従って計算したものである．この解析木から生成される列の後置形全体は，根の属性値として得られる．□

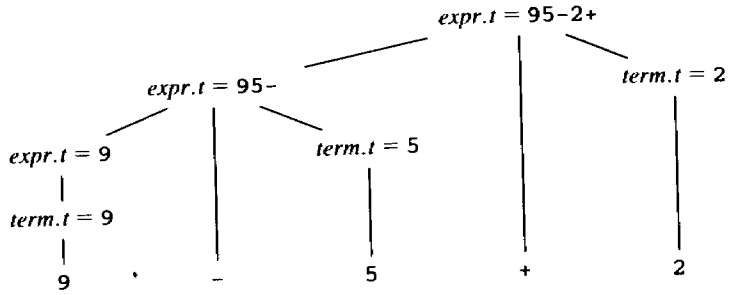


図2.6　解析木の各節での属性値

例2.7　ロボットを考える．このロボットは 1 つの命令で，現在の位置から東西南北のどちらかの方向に一歩だけ移動する．ロボットに対する命令列は次の

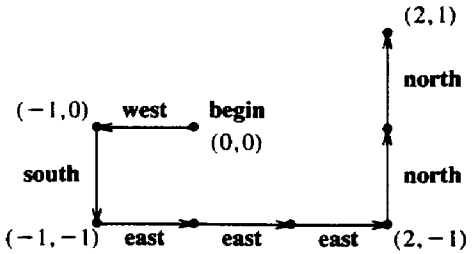


図2.7 ロボットの位置

文法によって定義する.

$seq \rightarrow seq\ instr \mid \mathbf{begin}$

$instr \rightarrow \mathbf{east} \mid \mathbf{north} \mid \mathbf{west} \mid \mathbf{south}$

次の入力が与えられたとき，ロボットの位置は図2.7のように変化する.

begin west south east east east north north

この図で，(x, y)はロボットの位置を表わす．ただし，x と y はそれぞれ，出発点からの東と北への歩数である．x が負ならば，ロボットは出発点よりも西側に位置し，y が負ならば，ロボットは出発点よりも南側に位置する.

命令列をロボットの位置に翻訳する構文主導定義を作ってみよう．非終端記号 *seq* の命令列から得られる位置を記録するために，2つの属性 *seq.x* と *seq.y* を用いる．*seq* は最初に **begin** を生成し，*seq.x* と *seq.y* をそれぞれ 0 に初期設定する．これは，命令列 **begin west south** に対する解析木で考えると，図2.8の一番左下の内部節点にあたる.

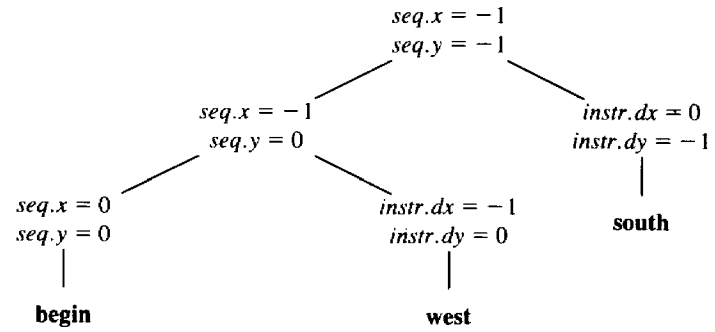


図2.8 **begin west south** に対する属性付き解析木

instr から導出される各命令によってロボットの位置が変化するとき，その変化分は属性 *instr.dx* と *instr.dy* に得るようにする．たとえば，*instr* が **west** を導出するときには，$instr.dx = -1$, $instr.dy = 0$とする．命令列 *seq* として，命令列 seq_1のあとに新しい命令 *instr* が続く生成規則を考えると，*seq* を実行したときのロボットの現在位置は次の意味規則で与えられる.

$seq.x := seq_1.x + instr.dx$

$seq.y := seq_1.y + instr.dy$

命令列をロボットの位置に翻訳するための構文主導定義を図2.9に示す. □

生成規則	意味規則
$seq \rightarrow \mathbf{begin}$	$seq.x := 0$ $seq.y := 0$
$seq \rightarrow seq_1\ instr$	$seq.x := seq_1.x + instr.dx$ $seq.y := seq_1.y + instr.dy$
$instr \rightarrow \mathbf{east}$	$instr.dx := 1$ $instr.dy := 0$
$instr \rightarrow \mathbf{north}$	$instr.dx := 0$ $instr.dy := 1$
$instr \rightarrow \mathbf{west}$	$instr.dx := -1$ $instr.dy := 0$
$instr \rightarrow \mathbf{south}$	$instr.dx := 0$ $instr.dy := -1$

図2.9 ロボットの位置の構文主導定義

深さ優先巡回

構文主導定義は，構文木の属性を評価するときの順序はとくに規定していない．属性 a を評価しようとするときに，a が依存する属性がすべてそれ以前で求められていれば，評価順序は任意でよい．一般には，解析木を辿りながら，節点をはじめて訪れるときに評価しなければならない属性もあれば，節点の子を訪れたあとで，あるいはそのあいだで，評価しなければならない属性もある．属性評価の最適な順序は5章で詳しく述べる.

この章で扱う翻訳は，解析木の属性評価を実現する場合に，どれも意味規則

の評価順序があらかじめ決められるものに限る．木の**巡回**は根から出発し，一定の順序に従って，各節点を訪れる．この章では，図2.10に示す深さ優先の巡回によって，意味規則を評価する．それには，図2.11に示すように，根から出発して，再帰的に節点の子たちを左から右に訪れる．各節点における意味規則は，その節点の子孫の巡回がすべて終わったあとで，1回だけ評価する．このような巡回の仕方は，まだ巡回の終わっていない節点を先に訪れ，その結果，根からできるだけ離れた節点を訪れようとするので，深さ優先の巡回とよばれる．

```
procedure visit(n: 節点);
begin
    for nの各子供mを左から右へ順に do
        visit(m);
    節点nの意味規則を評価
end
```

図2.10　木の深さ優先巡回

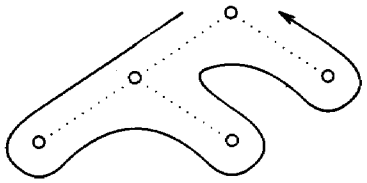


図2.11　木の深さ優先巡回の例

翻訳スキーム

この章では，これ以降，翻訳を定義するのに手続き的仕様を用いる．**翻訳スキーム**とは，文脈自由文法において，各生成規則の右辺に**意味動作**とよぶプログラムのコードを埋め込んだものである．翻訳スキームのもとになる文法を**基底文法**とよぶ．翻訳スキームは，意味規則の評価順序を明確に指定する点をのぞけば，構文主導定義と変わりはない．動作は生成規則の右辺の中に書き，実行したい位置にその動作を中かっこでくくって示す．たとえば，次のように書く．

$$rest \rightarrow + \ term \ \{ print('+') \} \ rest_1$$

基底文法から生成される文を x とすると，翻訳スキームは，x の解析木を深さ

優先巡回しながら，その途中で出会う動作を順に実行していく．たとえば，上の例で，*rest* というラベルをもつ解析木を考えよう．この場合の動作 $\{print('+')\}$ は，*term* の部分木の巡回が終わり，$rest_1$ の巡回に移る直前で実行される．

翻訳スキーム用の解析木を描くには，動作用の葉を新しく作り，その葉を生成規則の左辺の節点と破線で結ぶ．たとえば，上に示した生成規則と動作とを表わす解析木は図2.12のようになる．意味動作に対する節点は子をもたないから，意味動作はその節点を最初に訪れた時点で実行される．

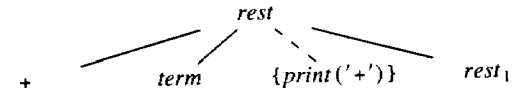


図2.12　意味動作のために作られた葉

翻訳結果の出力

この章では，翻訳スキームの中に現われる意味動作として，翻訳結果をファイルに書き出す操作だけを考える．1回の動作では1つの文字列または1つの文字を出力する．たとえば，9−5+2の翻訳は，部分式の翻訳結果をどこにも格納しないで，95−2+の各文字を1文字ずつ印字し，95−2+に変換する．このように，出力を細かく分けて作り出していくときには，文字の出力順序が重要な問題になる．

前に述べた構文主導定義には次の性質がある．各生成規則の左辺に現われる非終端記号について，その翻訳結果の文字列は，途中に適当な文字列を挿入しながら，右辺の非終端記号の翻訳結果を順に連結したものになる．このような性質をもつ構文主導定義を**単純構文主導定義**という．たとえば，図2.5の構文主導定義で，最初の生成規則と意味規則を考えてみよう．

生成規則	意味規則
$expr \rightarrow expr_1 + term$	$expr.t := expr_1.t \parallel term.t \parallel '+'$

(2.6)

この意味規則は，*expr* の翻訳 *expr.t* が，$expr_1$ と *term* の翻訳結果を連結し，そのあとに記号＋を続けた文字列であることを表わす．生成規則の右辺で，$expr_1$ が *term* よりも前に現われる点に注意してほしい．

次の例では，$term.t$ と $rest_1.t$ のあいだに文字列が入っている．

生成規則	意味規則
$rest \rightarrow + \ term \ rest_1$	$rest.t := term.t \parallel '+' \parallel rest_1.t$ (2.7)

しかし，この例も，生成規則の右辺で非終端記号 $term$ が $rest_1$ よりも前に現われているから，単純構文主導定義である．

単純構文主導定義は次のようにして翻訳スキームを利用すれば実現ができる．意味規則の中に直接現われる文字定数は動作によって印字するようにし，その動作は，意味規則における文字定数の出現順序を考慮して，生成規則の必要な箇所に挿入する．次の生成規則の動作は，(2.6)と(2.7)の意味規則の中の文字をそれぞれ印字するものである．

$$expr \rightarrow expr_1 + term \ \{print('+')\}$$
$$rest \rightarrow +\,term \ \{print('+')\} \ rest_1$$

例2.8　図2.5では，中置式を後置形に翻訳するための単純構文主導定義を示した．その定義から導出した翻訳スキームを図2.13に示し，9−5+2 に対する動作付きの解析木を図2.14に示す．図2.6と図2.14は入力から出力への同じ変換を規定しているけれども，出力の仕方が異なる．図2.6は結果を解析木の根に付加するのに対し，図2.14の方は結果を直接，印字している．

図2.14の根は図2.13の最初の生成規則を表わす．深さ優先巡回において，根の最左端の部分木を訪れると，最初に，$expr$ の左側の演算数に対する部分木の動作が実行される．次に，葉＋を訪れるが，そこではなにも動作は行われない．つづいて，右辺の演算数 $term$ に対する部分木の動作が実行され，最後に，あとから付加した節点で意味動作 $\{print('+')\}$ が実行される．

```
expr  →  expr + term   { print('+') }
expr  →  expr - term   { print('-') }
expr  →  term
term  →  0             { print('0') }
term  →  1             { print('1') }
   ...
term  →  9             { print('9') }
```

図2.13　式を後置記法に翻訳するための動作

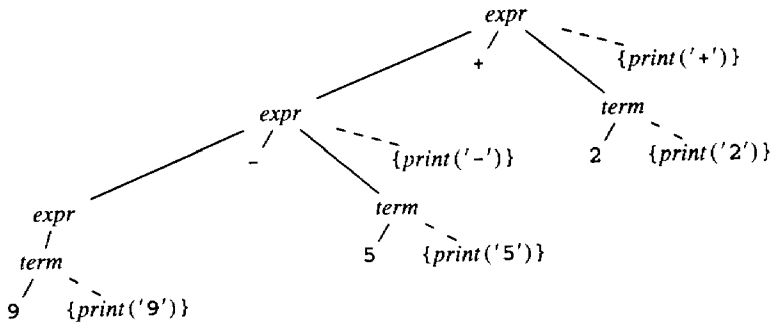


図2.14　9−5+2を95−2+に翻訳するときの動作

$term$ の生成規則には右辺に1桁の数字しかないから，その動作によって数字が印字されるだけである．生成規則 $expr \rightarrow term$ は動作の必要がなく，演算子を印字するのは最初の2つの生成規則だけである．解析木の深さ優先巡回によって，図2.14の動作は以上のようにして実行され，最終的に 95−2+ という印字結果が得られる．□

構文解析の方法は，一般に，“欲張り法”とよばれる特徴をもつアルゴリズムがほとんどであり，入力を左から右へ走査しながら，次のトークンを読み込む前にできるだけ大きな解析木を作ろうとする．単純翻訳スキーム(単純構文主導定義から導かれる)の動作も左から右への実行なので，単純翻訳スキームは，構文解析をしながら意味動作を実行する形の実現ができる．この方法を用いれば，解析木を作らなくてすむ．

2.4 構 文 解 析

構文解析は，トークン列が文法から生成できるかどうかを決定する処理である．この問題は，コンパイラが実際に解析木を作らなくても，解析木を使って考えると理解しやすい．構文解析ルーチンは，実際に解析木を作り出さなくても，その能力は備えていなければならない．そうでないと，翻訳の正しさは保証できない．

この節では，構文主導翻訳ルーチンの作成に応用できる解析法を紹介する．

次の節では，図2.13の翻訳スキームの実現を考え，C言語の完全なプログラムを示す．また，別の有力な方法に，ソフトウェア ツールを利用して，翻訳スキームから直接，翻訳ルーチンを生成する方法もある．これについては，4.7節を参照してほしい．このツールを用いれば，図2.13の翻訳スキームには手を加えないで，そのままの形で実現ができる．

どんな文法についても，構文解析ルーチンを作成することはできる．ただし，実際には限られた形式の文法しか使用しない．任意の文脈自由文法に対しては，n個のトークンの列を解析するのに，$O(n^3)$の時間を必要とする構文解析ルーチンも存在する．しかし，計算時間がnの3乗に比例するのでは，コストがかかりすぎて実用的ではない．プログラム言語については，一般に，構文解析が速くできるような文法を作ることが可能である．実際の言語の構文解析はどれも本質的に，線形の計算時間のアルゴリズムですむ．たいていは入力を左から右へ1回走査するだけですみ，しかもその間，トークンは1つ先読みするだけである．

構文解析法はほとんどが，**上向き**（または**上昇型**），および**下向き**（または**下降型**），とよぶ2つのクラスのどちらかに属す．これらの用語は，解析木の節点を作っていくときの方向を意味する．下向き解析は根から葉に向かうのに対し，上向き解析は葉から根に向かって木を作成する．下向き構文解析法は，効率の良いルーチンが手作業で簡単に作成できるので，広く普及している．しかし，構文解析で扱える文法と翻訳スキームのクラスとしては，上向きのほうが大きく，文法から直接，構文解析ルーチンを生成するツールは，この方法を用いる傾向が強い．

下向き構文解析

ここでは，下向き構文解析に適した文法を考え，その方法を紹介する．下向き構文解析ルーチンの一般的な作り方はこの節の後半で述べる．次の文法は，Pascalにおける型宣言の一部である．この文法では，文字列 ".." が1つの単位になることを強調するために，トークンとして**dotdot**を使用している．

$$
\begin{aligned}
\mathit{type} &\rightarrow \mathit{simple} \\
&\mid \uparrow \textbf{id} \\
&\mid \textbf{array}\ [\mathit{simple}]\ \textbf{of}\ \mathit{type} \\
\mathit{simple} &\rightarrow \textbf{integer} \\
&\mid \textbf{char} \\
&\mid \textbf{num dotdot num}
\end{aligned}
\tag{2.8}
$$

解析木を下向きに作成するには，まず，開始記号をラベルにもつ根を作り，以下，次の2つのステップを繰り返す（例は図2.15を参照）．

1. 非終端記号Aの付いた節点nについて，Aの生成規則を1つ選び出し，その規則の右辺の記号に対してnの子を作る．

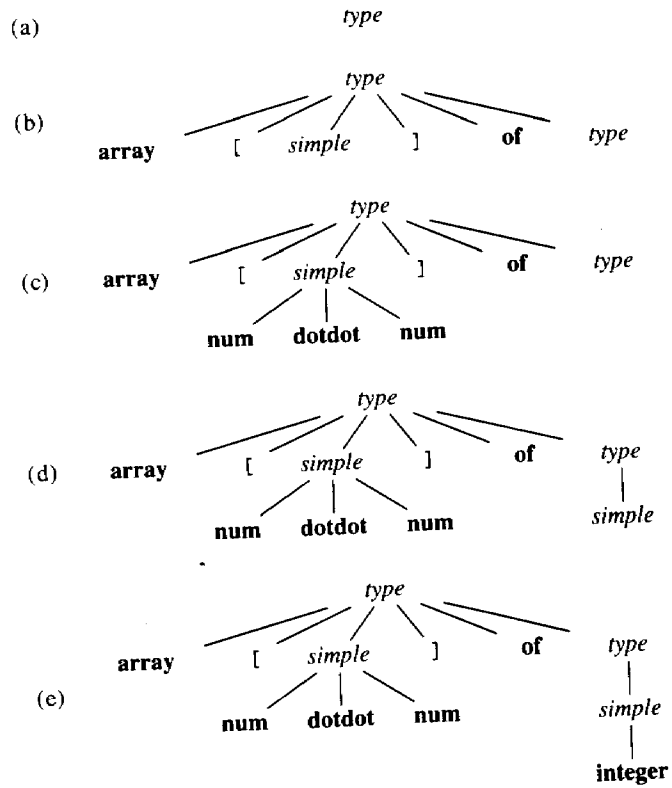


図2.15　解析木を下向きに作るときのステップ

2.　次に部分木を作ろうとする節点を見つける．

文法によっては，入力記号列を左から右へ１回走査するだけで，上のステップが実現できてしまうものもある．入力記号列の中で現在，走査中のトークンを，**先読み記号**とよぶ．処理を開始する時点での先読み記号は入力の先頭のトークン，すなわち最左端のトークンである．次の入力記号列に対する構文解析の過程を図2.16に示す．

array [num dotdot num] of integer

最初，トークン **array** が先読み記号となる．このとき，解析木についてわかっていることは，根が開始記号 *type* をもつ，という事実だけである．このときの状況を図2.16(a)に示す．このあとの目標は，解析木から生成される列と入力記号列とが一致するように残りの部分を作ることである．

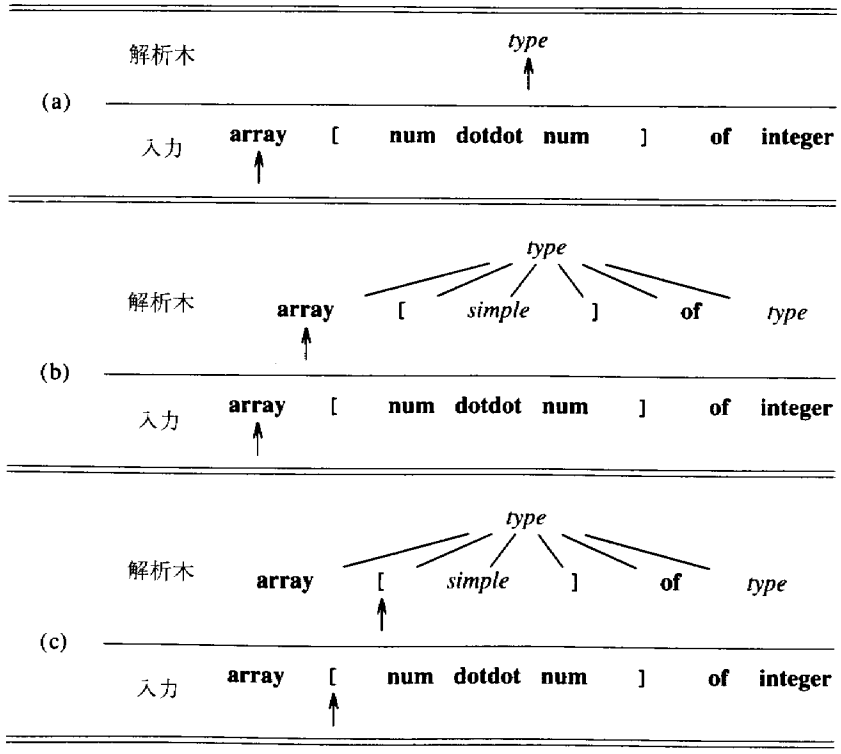


図2.16　左から右への入力の走査による下向き構文解析

それには，図2.16(a)の非終端記号 *type* から，先読み記号 **array** で始まる記号列が導出できなければならない．(2.8)の文法で，そのような記号列が導出できる *type* の生成規則は１つしかないので，それを選ぶ．そして，右辺の記号をもつ子を根の下に作る．

図2.16の３つの図では，先読み記号と，これから作ろうとしている解析木の節点とをそれぞれ矢印で示す．子の節点を作ったら，次に，その中の一番左にある子を考える．図2.16(b)は，根に子を作っただけの状態なので，一番左の子 **array** を次に取りあげる．

いま作ろうとしている節点が終端記号であって，その記号が先読み記号と一致すれば，解析木の中の節点と入力記号の両方を1つ先に進める．このときには，次の入力トークンが先読み記号となり，解析木の中の次の子が次に考える節点となる．図2.16(c)では，解析木の矢印が，根の次の子に移り，入力の矢印が次のトークン [に進む．このあと，解析木の矢印は非終端記号 *simple* の付いた子に移る．非終端記号の付いた子を考えるときには，その記号に対する生成規則を選択するために，いままでと同じ処理を繰り返す．

一般に，非終端記号に対する生成規則の選択は，試行錯誤が必要になる場合もある．すなわち，１つの生成規則を試みて，それがうまくいかなければ，もとに戻って，別の生成規則をふたたび試みなければならないことがある．１つの生成規則を使ってみて，それで入力記号列と一致する木ができなければ，その生成規則は不適当である．しかし，文法のなかには，予測型構文解析とよんで，後戻りの起らない解析ができる特別なものもある．

予測型構文解析

再帰下降構文解析は下向き解析の一種で，文法で用いる各非終端記号に対して１つの手続きを用意しておき，それらの手続きの実行によって入力記号列を処理する．ここでは，再帰下降構文解析のうちでも，予測型構文解析とよぶ特別な方法について述べる．この方法が使用できるのは，先読み記号によって，次にどの非終端記号に対する手続きを呼び出せばよいかが一意に定まる言語に限られる．入力に対する解析木は，実際に呼び出される手続きの系列によって暗に定義される．

```
procedure match(t:トークン);
begin
    if lookahead = t then
        lookahead := nexttoken
    else error
end;

procedure type;
begin
    if lookahead は { integer, char, num } のどれか then
        simple
    else if lookahead = '↑' then begin
        match('↑'); match(id)
    end
    else if lookahead = array then begin
        match(array); match('['); simple; match(']'); match(of); type
    end
    else error
end;

procedure simple;
begin
    if lookahead = integer then
        match(integer)
    else if lookahead = char then
        match(char)
    else if lookahead = num then begin
        match(num); match(dotdot); match(num)
    end
    else error
end;
```

図2.17 予測型構文解析ルーチンの擬似コード

図2.17の予測型構文解析ルーチンは，文法(2.8)の非終端記号 *type* と *simple* に対する手続き，および補助的な手続き *match* からなる．*match* は，引数 t と先読み記号とが一致すれば，先読み記号を次の入力トークンまで進める手続きであり，*type* と *simple* のコードを簡単にするために使用する．現在，走査中の

入力トークンは変数 *lookahead* に保持し，その変数の値は *match* によって更新される．

構文解析は，開始記号，いま考えている文法では *type* に対する手続きの呼出しで始まる．図2.16と同じ入力を使うと，*lookahead* には最初，先頭のトークン **array** を入れておく．手続き *type* は次のコードを実行する．

match(**array**); *match*('['); *simple*; *match*(']'); *match*(**of**); *type* (2.9)

これは生成規則

type → **array** [*simple*] **of** *type*

の右辺に対応する．右辺の終端記号は先読み記号との照合に，また右辺の非終端記号は手続きの呼出しになる点に注意してほしい．

図2.16の入力で，トークン **array** と [までの照合が終わったとすると，次の先読み記号は **num** である．この時点で，手続き *simple* を呼び出すと，その本体では次のコードが実行される．

match(**num**); *match*(**dotdot**); *match*(**num**)

先読み記号は，どの手続きを呼び出すかを決める働きをもつ．生成規則の右辺がトークンで始まっているときは，先読み記号とそのトークンとを照合し，両者が一致していれば，その規則を使用する．では，次のように右辺が非終端記号で始まる場合はどうだろう．

type → *simple* (2.10)

先読み記号が *simple* から生成できるものであれば，この規則を用いることができる．たとえば，(2.9)のコードの実行中に，制御が手続き *type* の呼出しに達し，先読み記号がトークン **integer** であったとする．このとき，トークン **integer** で始まる *type* の生成規則は存在しないけれども，**integer** で始まる *simple* の生成規則なら存在する．そこで，先読み記号の **integer** に対しては，(2.10)の生成規則を採用して，*type* から手続き *simple* を呼び出す．

予測型構文解析は，生成規則の右辺から生成される記号列のうちで，先頭がどのような記号で始まるか，という情報を手掛りに解析を行う．もう少し詳しく説明するために，非終端記号 A に対する生成規則の右辺を α としよう．また，α が生成する記号列をすべて考え，先頭の記号として現われるトークンの集合を FIRST(α)とする．α が ϵ であれば，あるいは α から ϵ が生成されるこ

とがあれば，ϵ も FIRST(α)に含める†．FIRST の例を以下に示す．

FIRST(*simple*) = { **integer**, **char**, **num** }

FIRST(↑**id**) = { ↑ }

FIRST(**array** [*simple*] **of** *type*) = { **array** }

実際には，生成規則の右辺はトークンで始まることが多いので，FIRST の集合を求めるのは簡単である．FIRST を計算するためのアルゴリズムは4.4節で述べる．

非終端記号 A に対して，2 つの生成規則 $A \to \alpha$ と $A \to \beta$ があったら，FIRST の集合を考慮しなければならない．後戻りのない再帰下降構文解析では，FIRST(α)と FIRST(β)が互いに素でなければならない．もしそうなら，先読み記号によって，どちらの生成規則を使用すればよいかが決定できる．すなわち，先読み記号が FIRST(α)の要素であれば，α を使用し，FIRST(β)の要素であれば，β を使用する．

ϵ 生成規則の使用

右辺に ϵ をもつ生成規則は特別な取扱いが必要である．再帰下降構文解析では，どのような生成規則も使用できないときにかぎって，省略時解釈として，ϵ 生成規則を使用する．例で考えよう．

stmt → **begin** *opt_stmts* **end**

opt_stmts → **stmt_list** | ϵ

opt_stmts の解析において，先読み記号が FIRST(*stmt_list*)になければ，ϵ 生成規則を使用してもよいだろう．ただし，この選択が正しいのは，先読み記号が **end** の場合だけである．そうでなければ，*stmt* の解析で誤りとなる．

予測型構文解析ルーチンの設計

予測型構文解析ルーチンは，すべての非終端記号に対する手続きの集りから

† 右辺に ϵ をもつ生成規則があると，非終端記号から生成される記号列の中から，先頭の記号を求めるのが面倒になる．たとえば，$A \to BC$ という生成規則があって，非終端記号 B が空列を生成することがあれば，FIRST(A)には，C から生成される記号列の先頭の記号を含めなければならない．さらに，C も ϵ を生成することがあれば，ϵ は FIRST(A)にも FIRST(BC)にも含まれる．

なる．各手続きでは次の2つの処理をする．

1. 先読み記号を調べて，どの生成規則を使用するかを決める．生成規則の右辺を α とすると，先読み記号が FIRST(α)にあれば，その規則を使用する．1つの先読み記号に対して，2つの右辺が適用できる文法では，この解析法を用いることができない．先読み記号がどんな右辺の FIRST にもなければ，ϵ 生成規則を使用する．
2. 次に，その生成規則を用いて，次のように右辺の処理をする．右辺に現われる非終端記号については，それに対応する手続きを呼び出し，トークンについては，先読み記号との照合を行う．そして，その照合に成功すれば，次の入力トークンを読み込み，そうでなければ，誤りを宣告する．以上の規則を文法(2.8)に適用すると，図2.17が得られる．

以前に文法を拡張して翻訳スキームを作ったが，それと同じように，この予測型構文解析ルーチンを拡張すると，構文主導翻訳ルーチンが作成できる．そのためのアルゴリズムは5.4節で述べる．この章で用いる翻訳スキームでは，非終端記号に属性を結び付けるという扱い方はしないので，いまのところ次のようにして，翻訳スキームから予測型構文解析ルーチンを実現する．

1. 生成規則中の動作は考えないで，まず予測型構文解析ルーチンを作成する．
2. 翻訳スキームの動作を次のようにして，そのルーチンの中に書き写す．翻訳スキームの中で，動作が文法記号 X のあとに現われれば，X を実現するためのコードのあとに，その動作を複写する．また，動作が生成規則の先頭にあれば，生成規則を実現するコードの直前に，その動作を書き写す．

次の節では，このような方法を使って，翻訳ルーチンを作成する．

左再帰

文法の性質によっては，再帰下降構文解析ルーチンが無限ループになる危険性がある．たとえば，次の左再帰の生成規則を考えてみれば明らかである．

expr → *expr* + *term*

この生成規則は，左辺の非終端記号と右辺の最左端の記号とが一致している．いま，*expr* の手続きの中で，この規則を適用することに決めたとする．右辺は *expr* で始まるから，*expr* の手続きを再帰的に呼び出す．その結果，構文解析ル

ーチンは無限ループに陥ってしまう．ここで，注意してほしいのは，再帰下降構文解析ルーチンにおいて，先読み記号が変わるのは，右辺の終端記号が入力のトークンと一致したときだけ，ということである．上の生成規則は非終端記号 *expr* で始まるので，再帰呼出しのあいだで入力が変わることがないので，このような結果になってしまう．

左再帰を含む生成規則は，規則の書換えによって，左再帰を除くことができる．次のような2つの生成規則をもつ非終端記号 A を考えてみよう．

$$A \rightarrow A\alpha \mid \beta$$

ここで，α と β は終端記号および非終端記号からなる記号列であり，A では始まらないものとする．たとえば，

$$expr \rightarrow expr + term \mid term$$

において，$A = expr$，　$\alpha = +\,term$ で，$\beta = term$ である．

生成規則 $A \rightarrow A\alpha$ は，右辺の最左端の記号が A であるから，非終端記号 A は**左再帰**である．この生成規則を繰り返し用いると，A からは，A の右側に記号列 α をいくつか並べた記号列が得られる．最後に，先頭の A を β で置き換えると，結局，この規則からは，図2.18(a)に示すように，β のあとに0個以上の α を並べた記号列が得られる．

A に対する生成規則は次のように書き換えても，図2.18(b)のように，同じ記号列が得られる．

$$\begin{aligned} A &\rightarrow \beta R \\ R &\rightarrow \alpha R \mid \epsilon \end{aligned} \qquad (2.11)$$

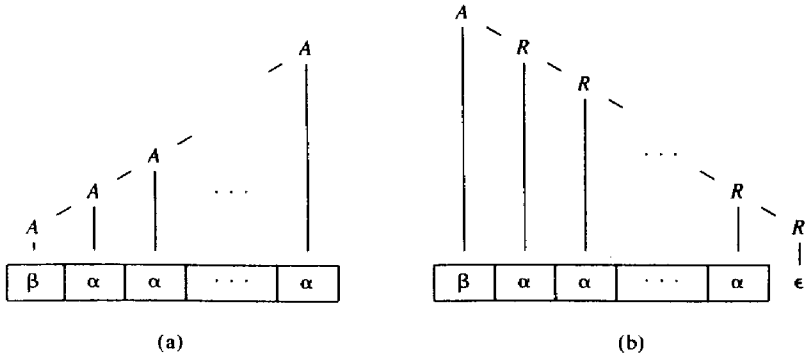


図2.18　左再帰と右再帰による記号列の生成の仕方

ここで，R は新しく導入した非終端記号である．生成規則 $R \rightarrow \alpha R$ は，右辺の最後の記号として R 自身が現われるので，右再帰である．右再帰の生成規則では，木が図2.18(b)のように，右下方向に成長していく．右さがりの木は，負符号のような左結合の演算子を含んでいると，式の翻訳が面倒になる．しかし，次の節で述べるが，右再帰文法でも，注意深く翻訳スキームを設計すれば，式を後置記法に正しく翻訳することができる．

4章では，左再帰のさらに一般的な形を考え，文法から左再帰を除去する方法について述べる．

2.5　簡単な式の翻訳プログラム

この節では，いままでの3つの節で説明してきた技法を使って，算術式を後置形に変換するための構文主導翻訳プログラムをCで作成する．最初は，プログラムが簡単になるように，+か−で区切った数字の式だけを考える．次の2つの節では，これを拡張して，数，識別子，あるいはほかの演算子も扱えるようにする．式は言語の構成要素としてもっとも一般的なものであり，その翻訳の仕方を詳しく学んでおいてほしい．

構文主導翻訳スキームは翻訳プログラムの仕様としても役に立つ．ここでも翻訳の定義として，図2.19(図2.13と同じもの)のスキームを使用する．スキームが与えられたときに，それを予測型構文解析ルーチンとして実現しようとすると，もとの文法を変更しなければならないことがよくある．たとえば，図2.19の文法は左再帰であり，前節で述べたように，その文法から左再帰を除去しなければ，予測型構文解析ルーチンで扱うことができない．

```
expr  →  expr + term    { print('+') }
expr  →  expr - term    { print('-') }
expr  →  term
term  →  0              { print('0') }
term  →  1              { print('1') }
  ...
term  →  9              { print('9') }
```

図2.19　中置式から後置式への翻訳の最初の仕様

抽象構文と具象構文

入力記号列の翻訳は，**構文木**，正確には**抽象構文木**とよぶ木から考え始めるといい．この木は，各節点が演算子を表わし，その節点の子が演算数を表わす．抽象構文木に対して，いままで使ってきた解析木のことを**具象構文木**という．また，基底文法をその言語の**具象構文**という．構文木は翻訳に不要な節点が省略され，意味のある節点だけからなるという特徴をもち，解析木とは異なる．

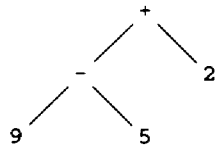


図2.20　9－5＋2の構文木

たとえば，9－5＋2 の構文木を図2.20に示す．＋と－は順位が同じで，同じ順位の演算子は左から右に評価するので，この木では，9－5 が部分式になる．図2.20とそれに対応する図2.2の解析木を比較してみると，解析木では演算子が子になるのに対し，構文木では演算子が内部節点となる．

翻訳スキームの基底文法は，解析木が構文木にできるだけ近い形になるものが望ましい．図2.19の文法での部分式のまとめ方は，構文木における部分木のまとめ方とよく似ている．しかし，図2.19の文法は左再帰であり，予測型構文解析には向いていない．これは，解析しやすい文法を求める一方で，同時に，翻訳しやすいまったく別の文法を求める，という点で矛盾しているようにも思われる．この問題の解決は，左再帰の除去にある．それには次の例で示すように十分な注意が必要である．

例2.9　次の文法は，図2.19の文法とまったく同じ言語を生成し，再帰下降構文解析法が使用できても，式を後置形に翻訳するのには向いていない．

$$
\begin{aligned}
&expr \rightarrow term\ rest\\
&rest \rightarrow + \ expr \mid - \ expr \mid \epsilon\\
&term \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9
\end{aligned}
$$

$rest \rightarrow + \ expr$ および $rest \rightarrow - \ expr$ による記号列の生成を考えると，この生成規則では，演算子に対する演算数がわからなくなってしまう，という問題がある．この生成規則を使って，仮に *expr.t* の翻訳から *rest.t* の翻訳を作ったと

すると，次の2通りの記述が考えられる．どちらにしても正しい翻訳はできない(ここには，－についての生成規則と意味規則だけしか示していない)．

$$rest \rightarrow - \ expr \quad \{rest.t := '-' \parallel expr.t\} \tag{2.12}$$

$$rest \rightarrow - \ expr \quad \{rest.t := expr.t \parallel '-'\} \tag{2.13}$$

9－5 の正しい翻訳は 95－である．しかし，(2.12)の動作を実行すると，*expr.t* よりも前に，－が現われるから，9－5 を翻訳しても 9－5 となり，誤りになる．

一方，(2.13)の規則を使い，＋についても同じような翻訳規則を作ったとすると，それらの意味規則は演算子を右端に移動させるだけなので，9－5＋2 は 952＋－という誤った翻訳になる．正しい翻訳は 95－2＋である．　□

翻訳スキームの適用

図2.18に示した左再帰を除去するための技法は，意味動作を含む生成規則にも適用できる．5.4 節では，その技法の拡張として，合成属性を考慮したときの変形の仕方を述べる．前の技法を使うと，生成規則 $A \rightarrow A\alpha \mid A\beta \mid \gamma$ は次のように変形できる．

$$
\begin{aligned}
&A \rightarrow \gamma R\\
&R \rightarrow \alpha R \mid \beta R \mid \epsilon
\end{aligned}
$$

この変形を行うときには，生成規則の中に埋め込んである意味動作も一緒にして扱う．いま，$A = expr$，　$\alpha = + \ term \ \{ print('+') \}$，$\beta = - \ term \ \{ print('-') \}$，$\gamma = term$ とすると，上の変形からは(2.14)の翻訳スキームが得られる．図2.19の *expr* の生成規則は，(2.14)において，*expr* および新しい非終端記号 *rest* に対する生成規則に変形してある．*term* の生成規則は図2.19と同じである．(2.14)の基底文法は例2.9の文法とは異なり，別の文法を用いているために，翻訳が可能になっている．

$$
\begin{aligned}
&expr \rightarrow term\ rest\\
&rest \rightarrow + \ term \ \{ print('+') \} \ rest \mid - \ term \ \{ print('-') \} \ rest \mid \epsilon\\
&term \rightarrow 0 \ \{ print('0') \}\\
&term \rightarrow 1 \ \{ print('1') \}\\
&\quad \cdots\\
&term \rightarrow 9 \ \{ print('9') \}
\end{aligned}
\tag{2.14}
$$

この文法によって，9－5＋2 がどのように翻訳されるかを図2.21に示す．

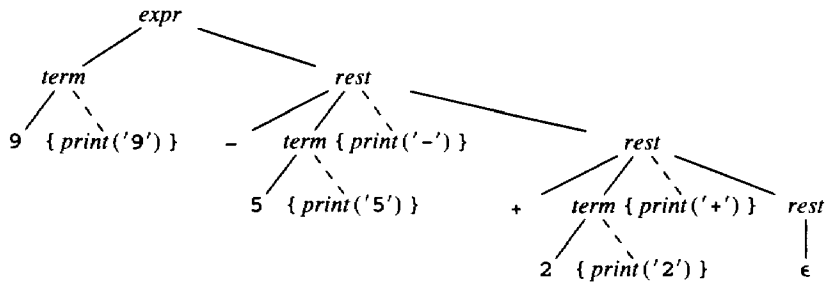


図2.21　9−5+2から95−2+への翻訳

非終端記号 *expr*, *term* および *rest* の手続き

(2.14)の構文主導翻訳スキームを使って，翻訳プログラムをC言語で実現してみよう．この翻訳プログラムの中心になるのは，関数expr，termおよびrestのCコードである．それらは(2.14)の非終端記号に対応するもので，図2.22に示す．

ここで，関数matchは，あとに示すとおり，図2.17のコードをC用に書き換えたものにすぎない．この関数は，先読み記号とトークンとを照合し，入力を1つ先に進める働きをもつ．いま考えている言語では，各トークンが1つの文字だけからなるので，matchは文字の比較と読込みで実現できる．

これからもC言語を使うので，Cをよく知らない人のために，PascalのようなAlgol系の言語とCとの大きな違いを述べておこう．Cのプログラムは，関数定義の並びからなる．実行は，mainと名付けた関数から始まる．関数定義は入れ子にはできない．関数は引数をもたなくても，引数並びをくくるかっこを必ず書かなければならない．そこで，図2.22では，expr(), term(), およびrest()という書き方をしている．関数は，“値による”引数渡し，またはすべての関数に対する大域的なデータのアクセスによって，他の関数とデータをやりとりする．たとえば，関数term()とrest()は，大域名lookaheadを用いて，先読み記号を調べている．


```
expr()
{
     term(); rest();
}

rest()
{
     if (lookahead == '+') {
         match('+'); term(); putchar('+'); rest();
     }
     else if (lookahead == '-') {
         match('-'); term(); putchar('-'); rest();
     }
     else ;
}

term()
{
     if (isdigit(lookahead)) {
         putchar(lookahead); match(lookahead);
     }
     else error();
}
```

図2.22　非終端記号 *expr*, *rest* および *term* に対する関数

CとPascalとでは代入と同値の判定に次の記号を用いる．

操作	C	Pascal
代入	=	:=
同値の判定	==	=
非同値の判定	!=	<>

非終端記号に対する関数は，生成規則の右辺に従って処理を行う．たとえば，生成規則 *expr* → *term rest* は，関数expr()の中でterm()とrest()の呼出しによって実現する．

別の例をあげると，関数rest()は，先読み記号が+であれば，(2.14)の *rest* に対する生成規則のうちで最初のものを，また先読み記号が−であれば，2番目の規則を，そうでなければ，省略時解釈として生成規則 *rest* → ϵ を用いる．*rest*

の最初の生成規則は図2.22の中で，最初の if 文で実現している．先読み記号が＋であれば，match('+')の呼出し，＋との照合に成功する．term()の呼出しのあと，＋を印字する意味動作は，Cの標準ライブラリ ルーチン putchar ('+')を用いて実現する．*rest* に対する 3 番目の生成規則は，右辺が ϵ であるので，rest()の最後の else ではなにもしない．

term に対する10個の生成規則はそれぞれの数字を生成する．図2.22において，isdigt は先読み記号が数字かどうかを判定する関数である．term では，数字の判定に成功したら，その数字を印字し，そのあとで match を呼び出す．もし，先読み記号が数字でなければ，誤りとして処理する(match は先読み記号を変更するので，印字は，数字との照合の前に行わなければならない)．完全なプログラムを示す前に，図2.22のコードを改良し，もっと高速の翻訳を工夫してみよう．

翻訳プログラムの最適化

再帰呼出しは，繰返しによって置換えができることがある．手続き本体の最後の文で，その手続き自身を再帰的に呼び出していれば，その呼出しは**尾部再帰**である，という．たとえば，関数 rest()の 4 行目と 5 行目にある rest()の呼出しは，そのあと制御がどちらも関数の終りに達するから，尾部再帰である．

尾部再帰を繰返しで置き換えれば，プログラムの実行時間が短縮できる．引数をもたない手続きであれば，尾部再帰呼出しは，たんに手続きの先頭への飛越しで置き換えてよい(引数付きの手続きの場合は，5.6節を参照)．その結果，rest のコードは次のようになる．

```
rest()
{
L:    if (lookahead == '+') {
          match('+'); term(); putchar('+'); goto L;
      }
      else if (lookahead == '-') {
          match('-'); term(); putchar('-'); goto L;
      }
      else ;
}
```


手続き rest では，先読み記号が＋または－であるあいだ，符号との照合を行い，次に数字との照合のために term を呼び出す処理を繰り返す．この繰返しは，符号と数字とが交互に現われる列についてだけ実行される．図2.22を上のように変更したとすると，あとに残る rest の呼出しは expr の中だけになる(3行目)．これらの2つの関数は，図2.23に示すように 1 つにまとめることができる．Cでは，条件式としての 1 はつねに真であるから，

```
while(1) stmt
```

と書けば，*stmt* を繰返し実行することができる．ループからの脱出には break 文を用いる．図2.23のような形でコードを書いておくと，あとから演算子を追加するのが簡単になる．

```
expr()
{
      term();
      while(1)
            if (lookahead == '+') {
                  match('+'); term(); putchar('+');
            }
            else if (lookahead == '-') {
                  match('-'); term(); putchar('-');
            }
            else break;
}
```

図2.23 図2.22 の関数 expr と rest の置換え

完全なプログラム

これまで考えてきた翻訳プログラムの完全なものを図2.24に示す．#include で始まる先頭の行は，関数 isdigit のコードを含むヘッダーファイル ctype.h の挿入を表わす．なお，このファイルには isdigit 以外にほかの標準ライブラリ ルーチンが含まれている．

1 文字のトークンを読み込むのには，入力ファイルから次の文字を読み込む標準ライブラリ ルーチン getchar を利用する．lookahead は，あとの節で扱うように，トークンが単一の文字でない場合を考慮し，図2.24の2行目で整数と定義している．lookahead は，どの関数よりも外側で宣言しているので，3 行

```
#include <ctype.h> /* 関数 isdigitを含むファイルをロード */
int lookahead;

main()
{
     lookahead = getchar();
     expr();
     putchar('\n'); /* 終りに改行文字を付ける */
}

expr()
{
     term();
     while(1)
          if (lookahead == '+') {
               match('+'); term(); putchar('+');
          }
          else if (lookahead == '-') {
               match('-'); term(); putchar('-');
          }
          else break;
}

term()
{
     if (isdigit(lookahead)) {
          putchar(lookahead);
          match(lookahead);
     }
     else error();
}

match(t)
     int t;
{
     if (lookahead == t)
          lookahead = getchar();
     else error();
}

error()
{
     printf("syntax error\n"); /* 誤りメッセージの印字 */
     exit(1);       /* そして，実行打ち切り*/
}
```

図2.24 中置式を後置形へ翻訳するCプログラム

目以降で定義する関数に対しては大域変数となる．

関数matchはトークンを検査する．すなわち，与えられたトークンと先読み記号とが一致すれば，次の入力トークンを読み込み，そうでなければ，誤り処理ルーチンerrorを呼び出す．

関数errorは，標準ライブラリ ルーチンprintfを使って，メッセージ“syntax error”(構文誤り)を印字し，別の標準ライブラリ ルーチンexit(1)を呼び出して，実行を終了する．

2.6 字 句 解 析

この節では，前の節で作成した翻訳プログラムに字句解析ルーチンを追加してみる．この字句解析ルーチンは入力からトークン ストリームへの変換を行い，構文解析ルーチンはそのストリームを入力として解析を行う．2.2節の文法の定義において，言語の文（sentence）はトークン列からなる，と述べた．1つのトークンを形成する入力文字の列を字句という．字句解析ルーチンを作ると，構文解析ルーチンでは，トークンの字句表現を考えなくてもすむ．まず，字句解析ルーチンで処理してほしい機能をあげてみよう．

空欄と注釈の除去

前の節に示した式の翻訳プログラムでは，入力に現われる文字をすべて式の一部として処理していたので，空白のような余計な文字があると，誤りになる．たいていの言語は，トークンの前後に“空欄”（空白，タブ，改行)を許している．同様に，注釈も，構文解析ルーチンや翻訳プログラムで処理しなくてもすむように空欄として扱う．

空欄を字句解析ルーチンで除いてしまえば，構文解析ルーチンではその処理を考えなくてよい．文法を変更して，空欄を構文に含めるのも1つの案ではあるが，その実現はそれほど簡単ではない．

定数

式の中に1桁の数字が現われるのであれば，そこには任意の整定数が現われてもよいのでは，と考えるのが自然である．整定数は数字の列であるから，整定数が使えるようにするには，式の文法にそのための生成規則を追加するか，

それともそのような定数のトークンを作るかすればよい．数は翻訳での1つの単位になるので，数字を集めて整数にするのは，字句解析ルーチンの仕事とするほうがよい．

そこで，整数を表わすトークンを **num** としよう．入力ストリームに数字の列が現われたら，字句解析ルーチンは **num** を構文解析ルーチンに渡す．そのとき，整数の値は，トークン **num** の属性として一緒に引き渡す．字句解析ルーチンは，論理的に，つねにトークンとその属性とを一緒にして，その両者を構文解析ルーチンに渡す．トークンとその属性の組を＜＞で囲んで表わすと，入力

```
31 + 28 + 59
```

は次のような組の列に変換される．

〈**num**, 31〉 〈+, 〉 〈**num**, 28〉 〈+, 〉 〈**num**, 59〉

トークン＋には属性はない．各組の2番目の要素，すなわち属性は，構文解析では使わないが，あとの処理で必要となる．

識別子とキーワードの認識

言語では，変数，配列，関数などの名前に識別子を用いる．識別子は文法の中でトークンとして扱うことが多い．そのような文法にもとづく構文解析ルーチンは，識別子が入力に現われるたびに，それらをすべて同じトークン，たとえば **id** として扱う．たとえば，字句解析ルーチンは入力

```
count = count + increment ;          (2.15)
```

を次のようなトークン ストリームに変換する．

id = **id** + **id** ; (2.16)

構文解析はこのトークン ストリームを処理する．

(2.15)の入力行についての字句解析を考えると，トークン **id** と，**id** の**インスタンス**(具体例)にあたる字句 count および increment とは区別して扱ったほうがよい．(2.16)しか与えられないとすると，翻訳ルーチンは，字句 count が最初の2つの **id** のインスタンスであり，字句 increment が3番目の **id** のインスタンスであることがわからなくなり，構文解析のあとの処理ができなくなってしまう．

入力の中に識別子を形成する字句があれば，それが以前に現われたものであるかどうかを調べる必要がある．それには，1章で述べたように，記号表を使う．字句は記号表に格納し，そのエントリへのポインタをトークン **id** の属性として用いる．

プログラム言語では，区切り記号として，また特定の構文要素を識別する目的で，begin, end, if のような決まった文字列をよく用いる．**キーワード**とよぶ文字列は一般に識別子の構成規則に従うので，どんなときに字句がキーワードとなり，どんなときに識別子となるかを判定する必要がある．キーワードが**予約済み**になっていて，識別子としての使用が禁止されていれば，この問題の解決は簡単である．そうなっていたら，キーワード以外の文字列を識別子とすればよい．

Pascal の< ,<= および< >のように，1つの文字が何種類かの字句に現われると，トークンを区別する問題が生じる．そのようなトークンを効率よく認識するための技法は3章で述べる．

字句解析ルーチンのインタフェース

字句解析ルーチンを構文解析ルーチンと入力ストリームとのあいだに置く場合には，図2.25のように両者とのやりとりが必要になる．字句解析ルーチンは入力から文字を読み込み，それを字句にまとめ，字句を形成するトークンとその属性値とを，あとの処理に引き渡す．場合によっては，構文解析ルーチンにトークンを返す前に，何文字かの先読みが必要になることもある．たとえば，Pascal では，文字＞が現われたら，その先の文字を読み込まなければならない．そして，次の文字が＝であれば，文字列＞＝は，“より大きいか，等しい”を判定する演算子の字句となる．そうでなければ，“より大きい”を判定する演算子の字句であり，字句解析ルーチンは1文字余計に文字を読んでしまったことになる．読みすぎた文字は次の字句の先頭の文字である可能性があるから，入力に戻しておかなければならない．

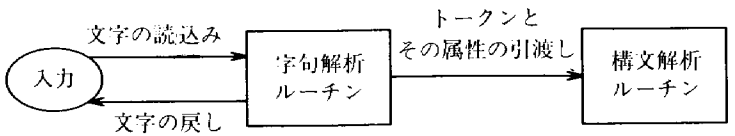


図2.25 入力と構文解析ルーチンのあいだに置かれた字句解析ルーチン

字句解析ルーチンと構文解析ルーチンの関係は生産者と消費者の関係である．字句解析ルーチンはトークンを生産し，構文解析ルーチンはそれを消費する．作り出したトークンは，それが使われるまで，トークン バッファの中に保持しておく．バッファが一杯だと，字句解析ルーチンは処理を進めることができないし，バッファが空だと構文解析ルーチンは処理ができないので，両者のあいだでのやりとりは，バッファの大きさだけによって制約される．一般に，バッファは1つのトークンを保持できればよい．その場合には，字句解析ルーチンを手続きとし，構文解析ルーチンから呼ばれるたびに，トークンを返すようにすればよい．

文字の読込みと文字をもとに戻す操作を実現するときには，ふつう入力バッファを使用する．バッファには 1 回に何文字かのブロックを入れておき，解析の終わった部分をポインタで指させるすようにする．文字をもとに戻すには，ポインタを前に戻せばよい．誤りがあったときは，入力テキストでどの部分が誤りであるかを指摘するために，入力文字をしまっておかなければならない場合もある．入力文字をバッファリングするのは，効率を上げるためであり，文字は一般に 1 文字ずつ読み込むよりも，ブロック単位で読み込むほうが効率が良いからである．入力のバッファリングの技法は3.2節で述べる．

字句解析ルーチン

2.5節の式の翻訳プログラムのために基本的な字句解析ルーチンを作ってみよう．ここでは，字句として，式の中に空欄や数が現われてもよいようにしたい．識別子については次の節で考える．

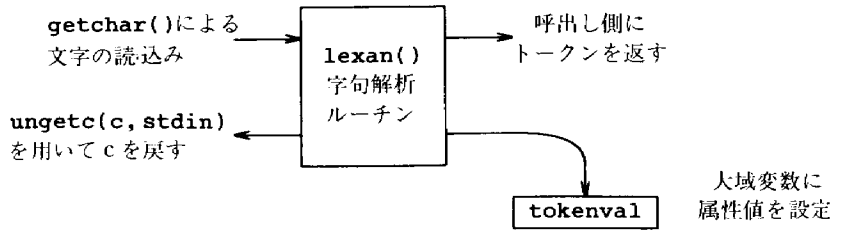


図2.26 図2.25におけるデータのやりとりの実現

図2.26は，字句解析ルーチンを C 言語の関数 lexan とし，図2.25におけるデータのやりとりの仕方を示す．getchar と ungetc は，標準のヘッダーファイル stdio.h で定義されているライブラリ関数であり，入力バッファに関する処理をする．lexan は入力文字の読込みと文字をもとに戻す操作のために，それぞれ，これらの関数を呼び出す．c を文字型の変数とすると，次の2つの文を実行しても，入力ストリームは変化しない．

```
c = getchar(); ungetc(c, stdin);
```

getchar は次の入力文字を c に代入し，ungetc は c の値を標準入力 stdin に戻す．

コンパイラ記述言語が，関数から構造型のデータを返す機能をもっていなければ，トークンとその属性とは別々に引き渡さなければならない．そこで，関数 lexan はトークンをコード化して整数を返すようにする．1 文字のトークンは文字コードそのものを整数として返す．**num** のようなトークンは，文字コードよりも大きな整数，たとえば256以上の整数にコード化する．ここでは，コード化が簡単に変更できるように，トークン **num** のコードを記号定数 NUM で表わす．Pascal では，定数宣言によって，NUM とコードとの結合ができる．C では，次の define 文によって，NUM は256を表わす名前と定義すればよい．

```
#define  NUM  256
```

関数 lexan は，入力の中に数字の列があると，NUM を返し，同時にその値を大域変数 tokenval に代入する．たとえば，入力で 7 のすぐ次に 6 がくると，tokenval には整数値76を代入する．

式の中に数を許すには，図2.19の文法を変更しなければならない．それには，各数字を非終端記号 *factor* で置き換え，次の生成規則と意味動作を加えればよい．

factor → (*expr*)
　　| **num** { *print*(**num**.*value*) }

図2.27に示す *factor* に対する C のコードは，上の生成規則を直接，実現したものである．lookahead が NUM であれば，属性 **num**.*value* の値は大域変数 tokenval に与えられる．この値の印字には標準ライブラリ関数 printf を用いる．この関数の最初の引数は，そのあとの引数に対する書式の指定であり，書式を表わす文字列を引用符で囲んで与える．この文字列の中に%d が現われる

```
factor()
{
     if (lookahead == '(') {
          match('('); expr(); match(')');
     }
     else if (lookahead == NUM) {
          printf(" %d ", tokenval); match(NUM);
     }
     else error();
}
```

図2.27 数も演算数とするための *factor* のCコード

と，次の引数の値が10進数表示で印字される．したがって，図2.27の printf は，1つの空白のあとに，tokenval の値を10進数表示で印字し，そのあとにもう1つ空白を印字する．

関数 lexan の実現を図2.28に示す．8～28行目までの while 文の本体を実行するたびに，9行目で，文字を t に読み込む．その文字が空白やタブ('\t')であれば，たんに while ループをまわるだけで，それらを構文解析ルーチンに返すことはしない．また，t が改行文字('\n')あれば，入力の行数を管理するために，大域変数 lineno の値を増やす．このときも，なにもトークンを返さない．行番号は，誤りメッセージの中で，誤りの箇所を指摘するのに用いる．

数字列を読み込むコードは14～23行目である．14行目と17行目では，入力文字 t が数字かどうかを判定するために，ヘッダーファイル type.h で定義されているライブラリ関数 isdigit(t) を使用する．ASCII または EBCDIC 文字集合を用いる計算機では，t が数字だとすると，t に対応する整数値は式 t − '0' によって得られる．そのほかの文字集合を用いる場合には，別の変換が必要になることもある．2.9節では，この字句解析ルーチンを式の翻訳プログラムの中に組み入れる．

2.7 記号表の組込み

一般に，原始プログラムのいろいろな構成要素に関する情報は，記号表とよぶデータ構造の中に格納しておく．それらの情報はコンパイラの解析フェーズで収集し，合成フェーズで目的コードを生成するのに使用する．たとえば，字句解析では，識別子を形成する文字列あるいは字句を記号表のエントリに入れ


```
(1)  #include <stdio.h>
(2)  #include <ctype.h>
(3)  int  lineno = 1;
(4)  int  tokenval = NONE;
(5)  int  lexan()
(6)  {
(7)       int t;
(8)       while(1) {
(9)            t = getchar();
(10)           if (t == ' ' || t == '\t')
(11)                ;     /* 空白とタブを取り除く */
(12)           else if (t == '\n')
(13)                lineno = lineno + 1;
(14)           else if (isdigit(t)) {
(15)                tokenval = t - '0';
(16)                t = getchar();
(17)                while (isdigit(t)) {
(18)                     tokenval = tokenval*10 + t-'0';
(19)                     t = getchar();
(20)                }
(21)                ungetc(t, stdin);
(22)                return NUM;
(23)           }
(24)           else {
(25)                tokenval = NONE;
(26)                return t;
(27)           }
(28)      }
(29) }
```

図2.28 空欄を除き，数を取り出す字句解析ルーチンのCコード

る．コンパイラのあとのフェーズは，識別子の型，用途(手続き，変数，ラベルなど）および記憶場所といった情報をそのエントリに加えていく．コード生成フェーズはこの情報を使って，変数に値を格納したり，アクセスしたりするためのコードを作り出す．記号表の実現と使い方は7.6節で詳しく述べる．この節では，前の節の構文解析ルーチンが記号表とどのようなやりとりをするかについて，例で説明する．

記号表のインタフェース

記号表ルーチンのおもな仕事は字句の格納と検索である．字句を格納するときは，一緒にそのトークンも格納しておく．記号表に関する操作には次の2つがある．

insert(s, t)：文字列 s とトークン t に対する新しいエントリを作り，その添字を返す．

lookup(s)：文字列 s のエントリの添字を返す．s が見つからなければ，0を返す．

字句解析ルーチンは，記号表の中に字句のエントリが作られているかどうかを知るために表引きを行う．もし，そのようなエントリが存在していなければ，挿入操作 insert を使って，エントリを作る．ここでは，記号表エントリの構造が外部に明らかにされているものとして，字句解析ルーチンと構文解析ルーチンの実現方法を述べる．

予約語の処理

上に述べた記号表ルーチンを使用すれば，予約語の処理ができる．たとえば，字句 div と mod について，それぞれのトークン **div** と **mod** を考えよう．この場合には，次の呼出しによって，記号表を初期設定しておけばよい．

```
insert("div", div);
insert("mod", mod);
```

このあとの呼出し lookup("div")はトークン **div** を返すので，div は識別子として使用することができない．

このような初期設定の仕方をすれば，どのような予約語でも正しく取り扱うことができる．

記号表の実現

記号表のデータ構造の一例を図2.29に示す．識別子を形成する字句を保持する際に，同じ大きさの空間を画一的に使用するという方式はとりたくない．そのようにしても，非常に長い識別子は収めることができないし，i のように短かい識別子があると，空間の無駄使いになってしまう．そこで，図2.29のように，別の配列 lexemes を使って，識別子を形成する文字列はすべてその中に入れる

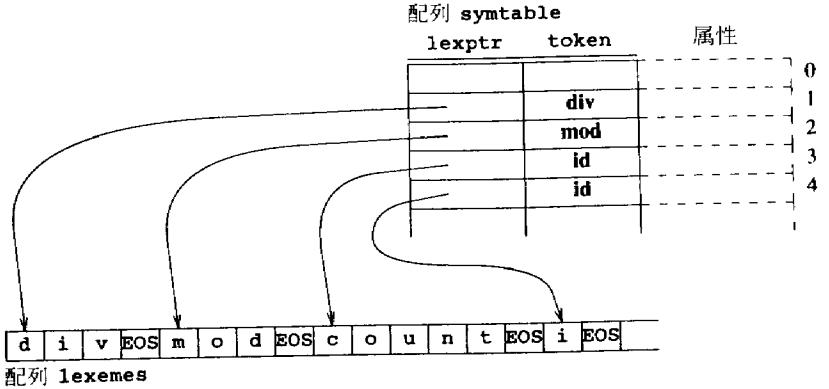


図2.29　記号表と文字列を格納するための配列

ようにする．各文字列の終りには文字列終了文字 EOS を付ける．EOS には，識別子の中にけっして現われることがない文字を使用する．記号表の配列 symtable の各エントリは2つの欄からなるレコードである．欄の1つは字句の先頭へのポインタ lexptr，もう1つは token である．必要ならば，欄を追加し，そのほかの属性値をもたせてもよい．

lookup は与えられた文字列に対するエントリがないと0を返すので，図2.29の0番目のエントリは空のままにしてある．次の2つは予約語 div と mod のエントリである．そのあとは，識別子 count と i のエントリを示す．

識別子を処理するための構文解析ルーチンの擬似コードを図2.30に示す．C 言語による実現は2.9節に示す．空欄と整定数の処理は前の節の図2.28と同じである．

いま考えている字句解析ルーチンでは，英字を読み込むと，それをまず，バッファ lexbuf に格納し，そのあとに現われる英数字を lexbuf のあとの部分に格納していく．lexbuf に集めた文字列が記号表にあるかどうかは，lookup 操作によって調べる．記号表は，最初，図2.29のように予約語 div と mod のエントリが作ってあるので，lexbuf が div か mod であれば，表引きによって，そのエントリが見つかる．もし，lexbuf の文字列が見つからなければ，すなわち lookup から0が返ってくれば，はじめて現われた識別子を意味する．そのときは，insert を用いて，そのためのエントリを作る．エントリを作り終わったときの p は，lexbuf の文字列に対する記号表エントリを指している．そこで，

```
function lexan: integer;
var   lexbuf:   array [0..100] of char;
      c:        char;
begin
    loop begin
        文字を c に読み込む;
        if c は空白またはタブ then
            何もしない
        else if c は改行文字 then
            lineno := lineno + 1
        else if c は数字 then begin
            c とそのあとの数字からなる数の値を tokenval に代入;
            return NUM
        end
        else if c は英字 then begin
            c とそのあとの英数字を lexbuf に 代入;
            p := lookup(lexbuf);
            if p = 0 then
                p := insert(lexbuf, ID);
            tokenval := p;
            return 表のエントリ p の token 欄
        end
        else begin  /* トークンは 1 つの文字 */
            tokenval に NONE を代入;  /* 属性はない */
            return 文字 c のコードに相当する整数
        end
    end
end
```

図2.30 字句解析ルーチンの擬似コード

この値を tokenval に入れ，そのエントリ中の欄 token の値を構文解析ルーチンに返す．

c が空欄や英数字以外の文字であれば，c に入っている文字コードを整数化し，その値をトークンとして返す．1 文字からなるトークンは属性をもたないから，tokenval には NONE を代入する．

2.8 抽象スタック機械

コンパイラのフロントエンドは，原始プログラムの内部表現を作り，バックエンドは，その内部表現から目的プログラムを生成する．中間表現でもっとも一般的なものは抽象スタック機械のコードである．1 章で述べたように，コンパイラをフロントエンドとバックエンドの 2 つに分ける理由は，コンパイラを新しい機械の上で実行させようとしたときに，コンパイラの変更を容易にするためである．

この節では，抽象スタック機械を解説し，コードの生成法を述べる．この機械は命令とデータとをメモリの別の場所に格納し，算術演算はすべてスタック上の値を対象とする．命令は非常に限られたものしかなく，整数演算，スタック操作，および制御の流れの 3 種類とする．図2.31にこの機械の構成を示す．ポインタ *pc* は実行しようとする命令を指す．各命令の意味を以下に簡単に説明する．

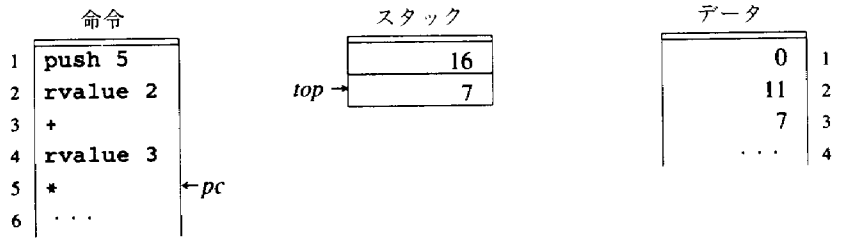


図2.31 最初の4 つの命令を実行し終わったときの抽象機械の状態

算術命令

抽象機械は中間言語の各演算子を実現しなければならない．加減算のような基本演算は抽象機械で直接，実行できる．しかし，複雑な演算は抽象機械の命令列として実現しなければならないこともある．ここでは，機械の説明を簡単にするために，中間言語の各算術演算子には，そのための命令が用意されているものとする．

算術式に対する抽象機械のコードは，スタックを使いながら，その後置表現の評価をシミュレートしていく．評価は，後置表現を左から右に処理していき，演算数に出会ったら，その値をスタックに積む(プッシュする)．*k* 項演算子に出会ったときには，その *k* 個の引数が左から順にスタックに積まれ，一番右側の引数はスタックの最上段に積まれている．このとき，スタックの上段にある *k* 個の値に対して演算を行ったら，それらをスタックから取り去り(ポップし)，

演算結果を再びスタックに積む．たとえば，後置式１３＋５＊を評価するときの動作は次のとおりである．

1. 1をスタックに積む．
2. 3をスタックに積む．
3. スタックの最上段にある2つの要素を加えたら，それらの要素をスタックから取り去り，結果4をスタックに積む．
4. 5をスタックに積む．
5. スタックの最上段にある2つの要素を掛けたら，それらの要素をスタックから取り去り，結果20をスタックに積む．

式全体の値は，最後の動作を終了したときのスタックの最上段の値(この例では20)である．

中間言語では，値はすべて整数とし，論理値のfalseは0として，trueは0以外の整数として扱う．論理演算子andとorは左右の演算数を必ず評価するものとする．

左辺値と右辺値

識別子は代入文の左辺に現われる場合と，右辺に現われる場合とで，異なる意味をもつ．代入文

```
i := 5 ;
i := i+1 ;
```

において，右辺は整数値の指定であるのに対し，左辺に現われるiは値の格納先の指定である．同様に，pとqを文字へのポインタとすると，

```
p↑:= q↑;
```

において，右辺のq↑は文字の指定であるのに対し，p↑はその文字の格納先の指定である．そこで，代入文の左辺と右辺とで，それぞれが意味する値を，**左辺値**，**右辺値**とよんで区別する．右辺値はふつう値と考えているものであり，左辺値は格納場所を表わす値である．

スタック操作

いままで，スタックに整定数を積んだり，最上段の値を取り去ったりする命令を用いてきたが，このほかにスタック上のデータをアクセスする命令として

次のものがある．

push v	vをスタックに積む．
rvalue l	データの格納場所lの内容を積む．
lvalue l	データの格納場所lの番地を積む．
pop	スタックの最上段の値を捨て去る．
:=	最上段にある右辺値を，その下の左辺値の示す場所に入れ，両方の値を取り去る．
copy	最上段の値をスタックに複写して積む．

式の翻訳

式を評価するためのスタック機械のコードと，後置記法のあいだには深い関係がある．式$E+F$の後置形は，定義から，Eの後置形とFの後置形のあとに，＋を連結したものである．これと同様に，$E+F$を評価するためのスタック機械のコードは，Eを評価するコードのあとに，Fを評価するコードを続け，そのあとに両者の値を加え合わせるための命令を続けたものとなる．したがって，式からスタック機械のコードへの翻訳には，2.6節と2.7節で述べた翻訳ルーチンを用いることができる．

式に対するスタック機械のコードの生成を考えよう．ここでは，データの格納場所の番地に記号を用いる(識別子に対する記憶場所の割付けは7章で述べる)．式a+bは次のように翻訳される．

```
rvalue a
rvalue b
+
```

これを言葉で説明すると，格納場所aとbの内容をスタックに積んだら，スタックの最上段にある2つの値の和を計算し，次に両者の値をスタックから取り去り，最後に結果をふたたびスタックに積む，という操作になる．

代入文からスタック機械コードへの翻訳は次のようにすればよい．代入しようとする識別子の左辺値をまずスタックに積み，式を評価したら，その右辺値を識別子に代入する．たとえば，代入文

```
day := (1461 * y) div 4 + (153 * m + 2) div 5 + d        (2.17)
```

を翻訳すると図2.32のコードが得られる．

```
lvalue day          push 2
push 1461           +
rvalue y            push 5
*                   div
push 4              +
div                 rvalue d
push 153            +
rvalue m            :=
*
```

図2.32 day := (1461 * y) div 4 + (153 * m + 2) div 5 + d の翻訳

以上の説明を定式化すると，次のようになる．

stmt → **id** := *expr* { *stmt.t* := 'lvalue' ‖ **id**.*lexeme* ‖ *expr.t* ‖ ' := ' }

ここで，非終端記号 *stmt* および *expr* の属性 *t* はそれぞれの翻訳結果を保持し，**id** の属性 *lexme* は識別子の文字列表現を保持する．

制御の流れ

スタック機械は，条件付きあるいは無条件命令による飛越しが起きないかぎり，書かれた順に命令を実行していく．飛先の指定には，一般に，次のように何通りかの方法が考えられる．

1. 命令の演算数として飛先を与える．
2. 命令の演算数として正または負の相対位置を指定する．
3. 飛先を記号で指定する．すなわち，機械自身がラベルを直接，処理する．

最初の2つの方法では，スタックの上段から演算数を取り出すという別の機能が必要になる．

ここでは，飛越し命令が簡単に作れる 3 番目の方法を用いる．また，飛先に記号番地を用いておくと，抽象機械のコードを生成したあと，そのコードの改良のために，命令を挿入したり，削除したりしても，飛先は変更しなくてすむ．

スタック機械の制御の流れに関する命令は次のとおりである．

label *l* 飛先 *l* を示す．それ以外の効果はない．

goto *l* 次は，ラベル *l* をもつ命令から実行を続ける．

gofalse *l* スタックの最上段から値を取り去り，その値が 0 なら飛越しをする．

gotrue *l* スタックの最上段から値を取り去り，その値が 0 以外の値なら飛越しをする．

halt 実行を停止する．

文の翻訳

図2.33は，条件文および while 文に対する抽象機械のコードの概要を示したものである．ここでは，ラベルの生成について述べる．

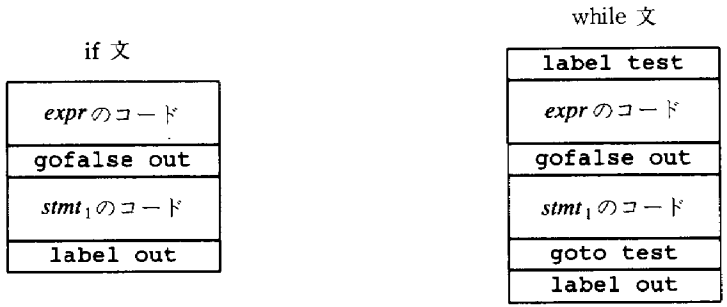


図2.33 条件文と while 文に対するコードの一般的な形

図2.33の if 文のコードを考えよう．label out という命令は原始プログラムの翻訳結果の中で，どこか 1 箇所にしか現われないはずである．もし，同じ label 命令が何箇所にも現われるとしたら，goto 命令によって制御を移すときに混乱が起きてしまう．そこで，out は，同じものにならないように，if 文を翻訳するたびに，一意なラベルで置き換えなければならない．

そのために，ラベルを作り出す手続きを *newlabel* とし，*newlabel* は呼び出されるたびに新しいラベルを返すものとする．次の意味動作では，*newlabel* の呼出しによって得られるラベルを，局所変数 *out* に代入するようにしている．

stmt → **if** *expr* **then** $stmt_1$ { *out* := *newlabel* ;
stmt.t := *expr.t* ‖
'gofalse' *out* ‖ (2.18)
$stmt_1.t$ ‖
'label' *out* }

翻訳結果の出力

2.5節の式の翻訳ルーチンでは，印字用の文を使って，式の翻訳結果を少しずつ分けて生成していった．ここでも，文の翻訳結果を出力するのに，同じように印字用の文を使用してもよいが，印字動作の詳細を隠すために，出力専用の手続き *emit* が用意されているものとしよう．そうすれば，抽象機械の命令を行に分けて表示するかどうか，といった問題は *emit* の中だけで考えればよくなる．この手続きを用いると，(2.18)は次のように書くことができる．

stmt → **if**
　　expr　{ *out* := *newlabel*; *emit*('gofalse', *out*)}
　　then
　　$stmt_1$　{ *emit*('label', *out*)}

意味動作が生成規則の中に現われる場合でも，生成規則の右辺の要素は，左から右へ順に処理されるものとする．上の生成規則における意味動作は次の順序で実行される．*expr* の構文解析における動作を実行したあと，*newlabel* から得られた値を *out* に代入し，gofalse 命令を出力する．そして，$stmt_1$の構文解析に移り，その中での動作を実行したあと，最後に label 命令を出力する．非終端記号 *expr* と $stmt_1$の構文解析の中で実行される動作が，それぞれの非終端記号に対するコードを出力するものと仮定すると，図2.33は上の生成規則によって実現できる．

代入文および条件文を翻訳するための擬似コードを図2.34に示す．変数 *out* は手続き *stmt* に対して局所的であるから，その値は，手続き *expr* や *stmt* の呼出しによる影響を受けない．ここで，ラベルの生成に注意しないといけない．ラベルは擬似コードにおいて，L のあとに整数を付けて L1，L2，... という形で出力するものと仮定する．ラベルのもとになる *out* は整数と宣言してあり，*newlabel* が返す整数が *out* の値となるから，*emit* はその整数からラベルを作り出して，印字しなければならない．

図2.33の while 文についても if 文の場合と同じように，翻訳のための擬似コードを書くことができる．文並びの翻訳結果は，並びの順序に従って各文の翻訳結果をたんにつなぎ合わせたものにすぎない．この実現は自分で試してみなさい．

1つの入口と1つの出口をもつ構文要素の翻訳は，ほとんどが while 文の翻訳と似たものになる．それを式における制御の流れの例で示そう．


```
procedure stmt;
var test, out: integer;    /* ラベル用 */
begin
    if lookahead = id then begin
        emit('lvalue', tokenval); match(id); match(':='); expr; emit(':=')
    end
    else if lookahead = 'if' then begin
        match('if');
        expr;
        out := newlabel;
        emit('gofalse', out);
        match('then');
        stmt;
        emit('label', out)
    end
        /* ほかの文に対するコードはここに入れる */
    else error;
end
```

図2.34　文を翻訳するための擬似コード

例2.10　2.7節の字句解析ルーチンでは次の形の条件文を用いた．

　　if *t*=**blank or** *t*=**tab then** ...

もし，*t* が空白であったとすると，*t* がタブであるかどうかの判定は不要である．その理由は，最初の同値性が成り立てば，この条件式全体が真になることを意味するからである．そこで，式

　　$expr_1$ **or** $expr_2$

は次のような実現ができる．

　　if $expr_1$ **then true else** $expr_2$

次のコードは演算子 **or** の実現を示す．この実現が正しいことを自分で確かめてみなさい．

```
    expr1のコード
    copy            /* expr1の値を複写 */
    gotrue out
```

```
    pop             /* exprの値をポップ */
    expr2のコード
    label out
```

gotrue と gofalse 命令は，条件文や while 文に対するコード生成を簡単にするために，スタックの最上段にある値を取り去る点に注意してほしい．そこで，$expr_1$の値を複写しておき gotrue 命令で飛越しが起っても，スタックの最上段に真が積まれていることを保証する．　□

2.9　コンパイル技法の応用

この章では，コンパイラのフロントエンドの構成のために構文主導の技法をいくつか解説してきた．この節では，それらの技法のまとめとして，セミコロンで区切られた式の並びからなる言語を考え，中置形の各式を後置形に翻訳するCプログラムを作ってみよう．式は，数，識別子および演算子+，−，*，/，div，mod からなる．翻訳プログラムからの出力は，各式の後置表現である．この翻訳プログラムは2.5〜2.7節で説明した翻訳ルーチンの拡張にすぎない．Cプログラムの完全なリストはこの節の終りに示す．

翻訳プログラムの説明

翻訳プログラムは図2.35の構文主導翻訳スキーマを用いて設計する．トークン **id** は英字で始まる英数字の非空の列，**num** は数字列，**eof** はファイル終了文字を表わす．トークンは空白，タブ文字，改行文字("空欄")の列によって区切られる．トークン **id** の属性 *lexeme* はそのトークンを形成する文字列を保持し，トークン **num** の属性 *value* は **num** が表わす整数を保持する．

翻訳プログラムのコードは7つのモジュールに分け，それぞれを1つのファイルに格納する．実行はモジュール main.c から開始する．このモジュールは，初期設定のために init( ) を呼び出し，そのあと翻訳のために parse( ) を呼び出す．そのほかのモジュールの関係は図2.36に示すとおりである．この翻訳プログラムでは，これ以外に複数のモジュールに共通な定義を含むヘッダーファイル global.h を使用する．そのために，どのモジュールも先頭に

```
#include "global.h"
```

を置き，このヘッダーファイルを自分のモジュールの一部として取り込むようにする．実際の翻訳プログラムのコードを示す前に，各モジュールとその構成を簡単に説明しておこう．


```
start → list eof
 list → expr ; list
      | ε
 expr → expr + term        { print('+') }
      | expr - term        { print('-') }
      | term
 term → term * factor      { print('*') }
      | term / factor      { print('/') }
      | term div factor    { print('DIV') }
      | term mod factor    { print('MOD') }
      | factor
factor → ( expr )
      | id                 { print(id.lexeme) }
      | num                { print(num.value) }
```

図2.35　中置形から後置形への翻訳プログラムの仕様

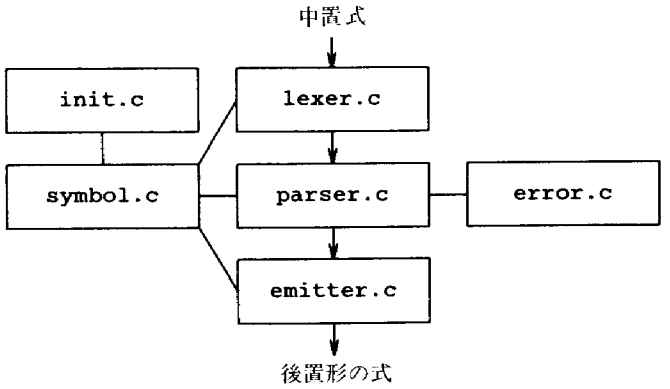


図2.36　中置形から後置形への翻訳プログラムのモジュール 構成

字句解析モジュール　lexer.c

字句解析ルーチン lexan( ) は，トークンを見つけるために構文解析ルーチンから呼び出される．このルーチンは，図2.30の擬似コードを実現したもので，文字を1つずつ読み込みながら，トークンが見つかったらそれを構文解析ルーチ

ンに返す．トークンの属性値は大域変数 tokenval に代入する．

構文解析ルーチンは次のトークンを処理する．

＋　－　＊　/　DIV　MOD　(　)　ID　NUM　DONE

ここで，ID は識別子を，NUM は数を，DONE はファイル終了文字を表わす．空欄は字句解析ルーチンの中で読み飛ばしてしまう．原始言語の字句について，字句解析ルーチンが作り出すトークンと属性値を図2.37の表に示す．

字句	トークン	属性値
空欄		
数字列	NUM	数字列の値
div	DIV	
mod	MOD	
上記以外の英字で始まる英数字の列	ID	記号表の添字
ファイル終了文字	DONE	
そのほかの文字	その文字	NONE

図2.37　トークン一覧

字句解析ルーチンは，同じ識別子の字句が以前に現われたかどうかを判定するために記号表ルーチン lookup を使用し，新しい字句を記号表に格納するためにルーチン insert を使用する．字句解析中に改行文字が出てきたら，大域変数 lineno の値を 1 増やす．

構文解析モジュール　parser.c

構文解析ルーチンは2.5節の技法を用いて作成する．最初に，再帰下降構文解析ルーチンによって，基底文法が解析できるように，図2.35の翻訳スキームから左再帰を除去する．その結果を図2.38に示す．

非終端記号 *expr*，*term* および *factor* に対する関数は，図2.24に示したようにして作成する．関数 parse( )は開始記号 start の解析を実現したものである．この関数では，新しいトークンが必要になると，lexan を呼び出す．構文解析ルーチンは，翻訳結果を出力するのに関数 emit を使用し，構文誤りを表示するのに関数 error を使用する．

$$
\begin{aligned}
start &\rightarrow list\ \textbf{eof}\\
list &\rightarrow expr\ ;\ list\\
&\mid \epsilon\\
expr &\rightarrow term\ moreterms\\
moreterms &\rightarrow +\ term\ \{\ print('+')\ \}\ moreterms\\
&\mid -\ term\ \{\ print('-')\ \}\ moreterms\\
&\mid \epsilon\\
term &\rightarrow factor\ morefactors\\
morefactors &\rightarrow *\ factor\ \{\ print('*')\ \}\ morefactors\\
&\mid /\ factor\ \{\ print('/')\ \}\ morefactors\\
&\mid \textbf{div}\ factor\ \{\ print(\texttt{'DIV'})\ \}\ morefactors\\
&\mid \textbf{mod}\ factor\ \{\ print(\texttt{'MOD'})\ \}\ morefactors\\
&\mid \epsilon\\
factor &\rightarrow (\ expr\ )\\
&\mid \textbf{id}\ \{\ print(\textbf{id}.lexeme)\ \}\\
&\mid \textbf{num}\ \{\ print(\textbf{num}.value)\ \}
\end{aligned}
$$

図2.38　左再帰を取り除いた導文主導翻訳スキーム

出力モジュール　emitter.c

この出力モジュールは，トークン t とその属性値 tval を出力する関数 emit(t , tval)だけからなる．

記号表モジュール　symbol.c と init.c

記号表モジュール symbol. c は2.7節の図2.29に示したデータ構造を実現したものである．配列 symtable のエントリは，配列 lexemes へのポインタとトークンを表わす整数との対からなる．操作 insert(s, t)は，トークン t および t を形成する字句 s を記号表に登録し，そのエントリの添字を返す．関数 lookup(s)は字句 s が symtable にあれば，そのエントリの添字を返し，そうでなければ，0を返す．

モジュール init.c は記号表にキーワードを前もって登録しておくのに用いる．キーワードの字句とトークン表現は，すべて symtable と同じ型の配列 keywords に格納しておく．関数 init( )は，配列 keywords の中の要素を順に調べながらキーワードを取りだし，関数 insert を用いてそれを記号表に登録

する．こうしておくと，キーワードの変更があっても，トークン表現の変更は keywords の修正だけですむ．

誤り処理モジュール　error.c

このモジュールは誤りについての処理を一括して行い，非常に基本的な働きをする．ここに示す翻訳プログラムは構文誤りを検出すると，現在処理中の行に誤りがあったことを知らせるメッセージを表示し，実行を停止する．しかし，できれば，次のセミコロンまで入力文字を読み飛ばし，構文解析を続けたほうがよい．この改良は自分で試してみなさい．さらにすぐれた誤り回復の技法は 4 章で述べる．

翻訳プログラムの作成

以上の各モジュールのコードは，それぞれファイル lexer.c，parser.c, emitter.c, symbol.c, init.c, error.c および main.c に入れておく．C プログラムの主ルーチンは，ファイル main.c に納め，そのルーチンから init( ) を呼び，次に parse( ) を呼び，処理が正しく終了したときには，exit(0) を呼び出す．

UNIX オペレーティング システムのもとでは，次のコマンドによって翻訳プログラムを作り出すことができる．

```
cc lexer.c parser.c emitter.c symbol.c init.c error.c main.c
```

あるいは，各ファイル

```
cc -c filename.c
```

によって，別々にコンパイルしておき，それによって得られた各ファイル *filename*.o を

```
cc lexer.o parser.o emitter.o symbol.o init.c error.o main.o
```

によって，リンク編集してもよい．どちらの場合でも，cc コマンドによって，翻訳プログラムを含むファイル a.out が作り出される．翻訳プログラムを実行するには a.out と入力し，あとは翻訳したい式を入力すればよい．たとえば，

```
2+3 * 5;
12 div 5 mod 2;
```

のようにデータを入れる．

プログラム リスト

以下に，翻訳プログラムを実現した C プログラムのリストを示す．はじめに，大域ヘッダーファイル global.h の内容を示し，そのあとに 7 つの原始ファイルの内容を示す．ここでは，わかりやすいように，プログラムは C の基本的なスタイルで記述してある．

```
/****   global.h   ********************************************/

#include <stdio.h>      /* 入出力ルーチンをロード */
#include <ctype.h>      /* 文字検査ルーチンをロード */

#define BSIZE  128      /* バッファの大きさ */
#define NONE   -1
#define EOS    '\0'

#define NUM    256
#define DIV    257
#define MOD    258
#define ID     259
#define DONE   260

int tokenval;      /*  トークンの属性値  */
int lineno;

struct entry {     /*  記号表エントリの形式  */
     char *lexptr;
     int  token;
};

struct entry symtable[];   /*  記号表  */

/****   lexer.c   *********************************************/

#include "global.h"

char lexbuf[BSIZE];
int  lineno = 1;
int  tokenval = NONE;

int  lexan()       /*  字句解析ルーチン  */
{
     int t;
     while(1) {
          t = getchar();
          if (t == ' ' || t == '\t')
               ;  /*  空欄を取り除く  */
          else if (t == '\n')
               lineno = lineno + 1;
```

```
        else if (isdigit(t)) {        /* tは数字 */
             ungetc(t, stdin);
             scanf("%d", &tokenval);
             return NUM;
        }
        else if (isalpha(t)) {        /* tは英字 */
             int p, b = 0;
             while (isalnum(t)) {     /* tは英数字 */
                  lexbuf[b] = t;
                  t = getchar();
                  b = b + 1;
                  if (b >= BSIZE)
                       error("compiler error");
             }
             lexbuf[b] = EOS;
             if (t != EOF)
                  ungetc(t, stdin);
             p = lookup(lexbuf);
             if (p == 0)
                  p = insert(lexbuf, ID);
             tokenval = p;
             return symtable[p].token;
        }
        else if (t == EOF)
             return DONE;
        else {
             tokenval = NONE;
             return t;
        }
    }
}

/****   parser.c   ****************************************/

#include "global.h"

int lookahead;

parse()     /*  式の並びを構文解析し，翻訳する  */
{
     lookahead = lexan();
     while (lookahead != DONE ) {
          expr(); match(';');
     }
}

expr()
{
```



```
     int t;
     term();
     while(1)
          switch (lookahead) {
          case '+': case '-':
               t = lookahead;
               match(lookahead); term(); emit(t, NONE);
               continue;
          default:
               return;
          }
}

term()
{
     int t;
     factor();
     while(1)
          switch (lookahead) {
          case '*': case '/': case DIV: case MOD:
               t = lookahead;
               match(lookahead); factor(); emit(t, NONE);
               continue;
          default:
               return;
          }
}

factor()
{
     switch(lookahead) {
          case '(':
               match('('); expr(); match(')'); break;
          case NUM:
               emit(NUM, tokenval); match(NUM); break;
          case ID:
               emit(ID, tokenval); match(ID); break;
          default:
               error("syntax error");
     }
}

match(t)
     int  t;
{
     if (lookahead == t)
          lookahead = lexan();
     else error("syntax error");
}
```

```
/****   emitter.c   ****************************************/

#include "global.h"

emit(t, tval)     /*  出力の生成  */
     int  t, tval;
{
     switch(t) {
          case '+': case '-': case '*': case '/':
               printf("%c\n", t); break;
          case DIV:
               printf("DIV\n"); break;
          case MOD:
               printf("MOD\n"); break;
          case NUM:
               printf("%d\n", tval); break;
          case ID:
               printf("%s\n", symtable[tval].lexptr); break;
          default:
               printf("token %d, tokenval %d\n", t, tval);
     }
}

/****   symbol.c   *****************************************/

#include "global.h"

#define STRMAX 999    /*  配列  lexemes の大きさ*/
#define SYMMAX 100    /* symtable の大きさ  */

char lexemes[STRMAX];
int  lastchar = - 1;  /* lexemes で最後に使用した位置  */
struct entry symtable[SYMMAX];
int  lastentry = 0;   /* symtable で最後に使用した位置  */

int  lookup(s)        /* s のエントリの位置を返す  */
     char s[];
{
     int p;
     for (p = lastentry; p > 0; p = p - 1)
          if (strcmp(symtable[p].lexptr, s) == 0)
              return p;
     return 0;
}

int  insert(s, tok)   /* s のエントリの位置を返す  */
     char s[];
     int  tok;
{
     int len;
     len = strlen(s); /* strlen は s の長さを計算する関数*/
     if (lastentry + 1 >= SYMMAX)
          error("symbol table full");
```



```
     if (lastchar + len + 1 >= STRMAX)
          error("lexemes array full");
     lastentry = lastentry + 1;
     symtable[lastentry].token = tok;
     symtable[lastentry].lexptr = &lexemes[lastchar + 1];
     lastchar = lastchar + len + 1;
     strcpy(symtable[lastentry].lexptr, s);
     return lastentry;
}

/****   init.c   *******************************************/

#include "global.h"

struct entry keywords[] = {
     "div", DIV,
     "mod", MOD,
     0,     0
};

init()     /*  symtable にキーワードを登録  */
{
     struct entry *p;
     for (p = keywords; p->token; p++)
          insert(p->lexptr, p->token);
}

/****   error.c   ******************************************/

#include "global.h"

error(m)     /*  すべての誤りメッセージを生成  */
     char *m;
{
     fprintf(stderr, "line %d: %s\n", lineno, m);
     exit(1);     /*  失敗したときの終了  */
}

/****   main.c   *******************************************/

#include "global.h"

main()
{
     init();
     parse();
     exit(0);     /*  正常な終了  */
}

/****************************************************************/
```