「コンパイラ」講義ノート

国島丈生

2011 年 4 月 7 日

第 1 章

導入

本講義では、コンパイラの内部構成について、理論面から実際のプログラムに至るまでを扱う。

C 言語のコンパイラは演習・実験等でよく使っているはずである。このソフトウェアは、C 言語のプログラムを機械語のプログラムや実行形式ファイル（a.out など）に変換してくれる。一般に**コンパイラ**（compiler）とは、「あるプログラミング言語で書かれたプログラムを読み込んで、別のプログラミング言語による等価なプログラムに翻訳するソフトウェア」である。翻訳前のプログラミング言語を**原始言語**（source language）、翻訳後のプログラミング言語を**目的言語**（target language）という。また、翻訳前のプログラムを**原始プログラム**（source program）、翻訳後のプログラムを**目的プログラム**（target program）という。C 言語のコンパイラは、原始言語が C、目的言語が機械語であるようなコンパイラ[*1]である。またコンパイラは、原始プログラムを解析し、その中に文法的な誤りがあれば、そのことをユーザに通知してくれる。

コンパイラの歴史は 1950 年代にさかのぼる。当時は、コンパイラは実装にひどく手間のかかるソフトウェアであった。しかし、その後 1980 年代頃までに、コンパイラ設計に関する理論的かつ体系的な手法がいろいろ考案され、コンパイラ実装の多くの部分は自動で行えるようになった。

したがってコンパイラの設計技法は、情報科学、ハードウェア、ソフトウェアの 3 分野が交差するところにあり、関連科目も多岐に渡る。

先行科目　情報処理学、離散数学、プログラミング言語 I、プログラミング言語 II、計算機アーキテクチャ A

後続科目　形式言語理論（修士）

このほか、この講義ではプログラム例の記述に C 言語を用いる。そのため、講義全般を通して C 言語の知識が必要になる。

この講義は [2][1] [4] を基にしている。ただし、分量が膨大であり、かつ難解な部分も多いので、基礎的な部分を抜粋して構成している[*2]。その他、[7][3][6] などを適宜参照している。

[*1] gcc は、C 以外の言語（C++ など）も原始言語として扱える。

[*2] 本テキスト中の例についてはこれらの書籍からの引用がかなり多く、学外関係者に流布すると著作権上問題がある。本資料の利用は学内関係者のみにとどめておいていただきたい。

```
1           .text
2   .globl _main
3   _main:
4           pushl   %ebp
5           movl    %esp, %ebp
6           subl    $24, %esp
7           movl    $1, -12(%ebp)
8           movl    -12(%ebp), %eax
9           addl    $20, %eax
10          movl    %eax, -16(%ebp)
11          movl    %ebp, %esp
12          popl    %ebp
13          ret
14          .subsections_via_symbols
```

図 1.1 i386 機械語命令の例

1.1 コンパイラの構成

今後の講義の流れを知っておくために、例を用いてコンパイラの内部構成を簡単に説明することにしよう。次の C プログラム（simple.c）を考える。

```
int main()
{
  int x;
  int y = 1;
  x = y + 20;
}
```

コンパイラは simple.c の内容を解析し、例えば図 1.1 のような機械語列を生成する[*3]。なお、本当の機械語列は、それぞれのシンボルを 0/1 の列で置き換えたものになる。

各機械語命令は、CPU によって直接解釈実行することができる。simple.c をコンパイルして得られた実行形式ファイル（a.out など）を実行すると、この機械語命令列が主記憶上にロードされ、CPU がそれを順次解釈して実行していく。各行の意味はだいたい次のようになっている。

- 3 行目…_main という目印（ラベル）づけ。main 関数に対応している。
- 4-6 行目…main 関数の呼び出し処理。このとき、-16(%ebp) 番地から 4 バイトの領域に局所変数 x が、-12(%ebp) 番地から 4 バイトの領域に局所変数 y がそれぞれ割り当てられる。
- 7 行目…y の領域に$1、すなわち定数 1 がセットされる。
- 8 行目…y の領域の値がレジスタ %eax にセットされる。
- 9 行目…レジスタ %eax の値に定数 20 が加えられる。
- 10 行目…レジスタ %eax の値が x の領域にセットされる。

[*3] i386 アーキテクチャを仮定している。

- 11-13 行目…main 関数の終了処理[*4]。

この講義で扱う内容を整理すると次のようになる。

1. 原始プログラム（上の例では C プログラム）から目的プログラム（上の例では機械語列）を生成するアルゴリズム
2. 1 を実現するプログラム
3. 2 を自動的に生成するアルゴリズム/手法

1 と 3 の違いに充分留意されたい。

さて、コンパイラは大きく分けて以下の 2 つの部分から構成される。

解析部 原始プログラムを解析し、コンパイラ内部で処理するためのデータ構造（**中間表現**）を作る。文字列としての原始プログラムからプログラムの最小意味単位（**トークン**, token）を切り出す**字句解析**（lexical analysis）部、トークン列からプログラムの意味構造を解析する**構文解析**（syntax analysis）部、型の検査など構文解析部では行えないプログラム解析を行う**意味解析**（semantic analysis）部の 3 つの部分からなる。また、先の例の main, x, y など、ユーザが定義した変数や関数の列挙もこの段階で行われる。

合成部 解析部で作った中間表現を基に、目的プログラムを合成する。場合によっては、実行速度を上げるために、合成したプログラムをさらに加工することもある（**最適化**, optimization）。

コンパイラは、最初は原始プログラムを単なる文字の並びとして扱う。先の例で言えば、次のような文字の並びとして扱う（\n は改行文字を表す）。

```
int main()\n{\n int x;\n int y = 1;\n x = y + 20;\n}\n
```

ここから C 言語として意味のある文字列を切り出すのが字句解析部である。結果は次のような文字列の並びになる[*5]。空白や改行文字など、C 言語として意味のない文字列は出力されていないことに注意しておいてほしい。

```
int, main, (, ), {, int, x, ;, int, y, =, 1, ;, x, =, y, +, 20, ;, }
```

この並びを、プログラミング言語の文法に従ってグループ化し、意味のまとまりを解析するのが構文解析部である。例えば、セミコロン ; は C 言語の文の区切りであり、セミコロンで囲まれた次の部分は一つのまとまりになる。

```
x, =, y, +, 20
```

これはさらに x, =, y + 20 という 3 つの部分にグループ化でき、さらに最後の部分は y, +, 20 という 3 つの部分にグループ化できる。このように、グループ化した結果、つまり構文解析の結果は通常グループの入れ子になり、木で表現するのが一般的である（図 1.2）。

[*4] 11-12 行目は leave という 1 つの機械語命令にまとめられていることもある。

[*5] 後で述べるように、実際には、出力するのはそれぞれの文字列を抽象化した記号、すなわちトークンである。また、一度にこのような並びをまとめて出力するのではなく、字句解析部に要求があるたびに一つずつ切り出して出力する。

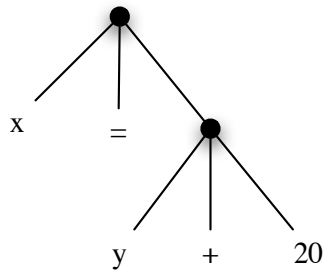


図 1.2　x = y + 20 を表す木構造

論理的には、これが構文解析部の出力データ構造になる。

この木には、型の情報など、プログラム実行のための重要な情報がいくつか含まれていない。これを補うのが意味解析部である。定数や変数の型変換などは意味解析部で適宜処理される。その結果、構文解析部の出力の木に修正が加えられることもある。

ここまでで得られた木構造を基にして、目的プログラムが合成される。その際、いきなり目的プログラムを合成するのではなく、いったん擬似的な機械語列を生成してから実際の目的プログラムに変換するほうが考えやすい。例えば、x = y + 20 に対して、新たな変数 temp1, temp2 を導入し、次のような擬似機械語列を考える。

```
temp1 = y
temp2 = temp1 + 20
x = temp2
```

すると、先に示した目的プログラムとの対応がとりやすくなることが分かるだろう。この擬似機械語列に対して、変数 temp1, temp2 にレジスタ %eax を、x に番地-16(%ebp) を、y に番地-12(%ebp) をそれぞれ割り当てると、先に示した目的プログラムが生成できる。レジスタや番地の割り当て、関数呼び出し時の処理コードの追加なども合成部で行われる。

なお、ここで述べたコンパイラの構成はもっとも基本的なものである。実際のコンパイラ設計ではこの構成に厳密には従わないことも多々ある。

1.2　練習問題

問題 1.1　gcc は、-s オプションをつけることで機械語命令列を出力させることができる。簡単な C 言語プログラムを各自で書き、どのような機械語命令列が出力されるか、試してみよ。

問題 1.2　次の C 言語の文を、C 言語として意味のある文字列に分解せよ。

```
x = 0; while (i < 100) { x += i; i++; }
```

第 2 章

正則表現と有限オートマトン

コンパイラの技法のうち、字句解析と構文解析は言語理論と関連が深い。ここでは言語理論で用いられる用語を簡単に説明し、次いで字句解析で用いられる正則表現と有限オートマトンについて述べる。構文解析で用いられる文脈自由文法については 4 章で述べる。

2.1 アルファベットと記号列

普段われわれが「文字」と呼んでいるものを、言語理論では**記号**（symbol）といい、また、記号の有限集合を**アルファベット**（alphabet）という[*1]。

例 2.1 $\{a, b, \cdots, z\}$ や $\{0, 1\}$ はアルファベットである。□

記号が「文字」を表さないこともある。例えば、コンパイラの字句解析部では記号を「文字」と考えてよいが、構文解析部では、記号は原始言語で意味のある文字列、すなわちトークンである。

アルファベット Σ[*2]中の記号からなる有限列を、Σ 上の**記号列**（string）という[*3]。Σ 上の記号列 s について、s に現れる記号の数を s の**長さ**といい、$|s|$ で表す。

例 2.2 *main* や *exercise* はアルファベット $\{a, b, \cdots, z\}$ 上の記号列であり、長さはそれぞれ $4, 8$ である。また $\epsilon, 0, 11, 01011$ はいずれもアルファベット $\{0, 1\}$ 上の記号列であり、長さはそれぞれ $0, 1, 2, 5$ である。□

長さ 0 の記号列、つまり 1 つも記号を含まない記号列を**空列**（empty string）といい、ϵ という記法で表す。$|\epsilon| = 0$ である。

2.2 言語

言語理論での**言語**（language）とは、日本語や英語、プログラミング言語などを抽象化したものである。

アルファベット Σ 上の**言語**（language）とは、Σ 上の記号列の集合である。言語に含まれる記号列は可算無限でも構わない。

[*1] 日常生活で英文字を表すときに用いる「アルファベット」とは意味が異なる。

[*2] アルファベットはしばしば Σ と表される。

[*3] 文字列、あるいは語（word）ということもある。

例 2.3　$\{0\}$, $\{00, 01, 10, 11\}$, $\{1, 11, 111, 1111, \cdots\}$ はいずれもアルファベット $\{0, 1\}$ 上の言語であり、それぞれ記号列 0、長さ 2 のすべての記号列、記号 1 のみからなるすべての記号列を表す。□

言語は記号列の集合であるから、空集合 $\emptyset$ や、空列しか含まない集合 $\{\epsilon\}$ も言語である。この 2 つの言語は全く異なるものであることに充分注意しておいてほしい。$\emptyset$ には要素は 1 つも含まれないが、$\{\epsilon\}$ には ϵ という要素が 1 つ含まれている。

2.3　言語に対する演算

言語は（記号列の）集合であるので、$\cup$, $\cap$, $^-$（補集合）などの集合演算はもちろん適用できる。その他に、記号列に由来する重要な演算がいくつかある。

2.3.1　連接

まず、**連接**という演算を紹介しよう。そのために、記号列に対する連接を定義しておく。2 つの記号列 x, y について、x の後ろに y が続く記号列を x と y の**連接**（concatenation）といい、$x \cdot y$ または xy と書く。

例 2.4　$00 \cdot 11 = 0011$, $111 \cdot \epsilon = \epsilon \cdot 111 = 111$ である。□

任意の記号列 x に対して $\epsilon \cdot x = x \cdot \epsilon = x$ が成り立つ。つまり、ϵ は連接に関する単位元である。

記号列 x 自身を n 個連接した記号列を x^n と表し、特に $x^0 = \epsilon$ と定義しておく。この結果、$i \geq 0$, $j \geq 0$ に対して $x^{i+j} = x^i \cdot x^j$ が成立する。

例 2.5　$(01)^2 = 0101, (01)^3 = 010101$ である。□

さて、2 つの言語 L, M に対する連接 $\cdot$ を次のように定義する。

$$L \cdot M = \{s \cdot t \mid s \in L, t \in M\}$$

記号列の連接と同様、$\cdot$ を省略して LM と書くこともできる。

例 2.6　$L = \{0, 1\}, M = \{0, 01, 111\}$ とすると、$L \cdot M = \{00, 001, 0111, 10, 101, 1111\}$ である。また $L \cdot \{\epsilon\} = \{\epsilon\} \cdot L = L$, $M \cdot \emptyset = \emptyset \cdot M = \emptyset$ である。□

任意の言語 L に対し $L \cdot \{\epsilon\} = \{\epsilon\} \cdot L = L$ が成り立つ。また $L \cdot \emptyset = \emptyset \cdot L = \emptyset$ が成り立つ。つまり、言語の連接では、$\{\epsilon\}$ が単位元、$\emptyset$ が零元である。

言語 L を n 個連接して得られる言語を L^n と表し、特に $L^0 = \{\epsilon\}$ とする。これにより $i \geq 0$, $j \geq 0$ について $L^{i+j} = L^i L^j$ が成り立つ。

2.3.2　閉包

言語 L について、次の式で定義される言語 L^* を L の **Kleene 閉包**（Kleene closure）という。

$$L^* = L^0 \cup L^1 \cup L^2 \cup \cdots = \bigcup_{i=0}^{\infty} L^i$$

また次の式で定義される言語 L^+ を L の**正閉包**（positive closure）という。

$$\begin{aligned} L^+ &= L^1 \cup L^2 \cup L^3 \cup \cdots = \bigcup_{i=1}^{\infty} L^i \\ &= L^* - \{\epsilon\} \end{aligned}$$

例 2.7　$L = \{a\}$ とすると

$$\begin{aligned} L^0 &= \{\epsilon\} \\ L^1 &= \{a\} \\ L^2 &= \{aa\} \\ L^3 &= \{aaa\} \\ &\cdots \end{aligned}$$

であり、$L^* = \{\epsilon, a, aa, aaa, \cdots\}$, $L^+ = \{a, aa, aaa, \cdots\}$ である。

また $L = \{a, b\}$ とすると

$$\begin{aligned} L^0 &= \{\epsilon\} \\ L^1 &= \{a, b\} \\ L^2 &= \{aa, ab, ba, bb\} \\ &\cdots \end{aligned}$$

であり、$L^* = \{\epsilon, a, b, aa, ab, ba, bb, \cdots\}$, $L^+ = \{a, b, aa, ab, ba, bb, \cdots\}$ である。この例から分かるように、各々の L から異なる要素を選んでも構わない。□

特に、アルファベット Σ の Kleene 閉包 Σ^* は Σ 上の長さ 0 以上のすべての記号列からなる集合である。

$$\Sigma^* = \Sigma^0 \cup \Sigma^1 \cup \Sigma^2 \cup \cdots$$

また Σ の正閉包 Σ^+ は Σ 上の長さ 1 以上のすべての記号列からなる集合である。

$$\begin{aligned} \Sigma^+ &= \Sigma^1 \cup \Sigma^2 \cup \cdots \\ &= \Sigma^* - \{\epsilon\} \end{aligned}$$

例 2.8　$\Sigma = \{0, 1\}$ とするとき、$\Sigma^* = \{\epsilon, 0, 1, 00, 01, 10, 11, 000, \cdots\}$ である。また $\Sigma^+ = \{0, 1, 00, 01, 10, 11, 000, \cdots\}$ である。□

2.4 正則表現

正則表現[*4]（regular expression）は、記号列の集合を簡潔に表現する記法である。まず記法の定義を示そう。

定義 2.1　アルファベット Σ 上の正則表現とは、以下の規則から再帰的に構成される式である。

1. $\emptyset$ は正則表現である。

[*4] **正規表現**とも言う。

2. ϵ は正則表現である。
3. a（ただし $a \in \Sigma$）は正則表現である。
4. r, r' を正則表現とするとき、$r \mid r'$ は正則表現である。
5. r, r' を正則表現とするとき、$r \cdot r'$ は正則表現である。
6. r を正則表現とするとき、r^* は正則表現である。
7. r を正則表現とするとき、(r) は正則表現である。

□

例 2.9　$0 \cdot (0 \mid 1)^*$ はアルファベット $\{0,1\}$ 上の正則表現である。なぜなら:

1. 定義 2.1 の 3 により 0, 1 はそれぞれ正則表現である。
2. 定義 2.1 の 4 により $0 \mid 1$ は正則表現である。
3. 定義 2.1 の 7 により $(0 \mid 1)$ は正則表現である。
4. 定義 2.1 の 6 により $(0 \mid 1)^*$ は正則表現である。
5. 定義 2.1 の 5 により $0 \cdot (0 \mid 1)^*$ は正則表現である。

□

アルファベット Σ 上の正則表現は、Σ 上のある言語を表している。

定義 2.2　アルファベット Σ 上の正則表現 R の表す言語 $L(R)$ は、以下のように再帰的に定義される。

1. $L(\emptyset) = \emptyset$
2. $L(\epsilon) = \{\epsilon\}$
3. $L(a) = \{a\}$
4. r, r' の表す言語をそれぞれ $L(r)$, $L(r')$ とするとき、$L(r \mid r') = L(r) \cup L(r')$
5. r, r' の表す言語をそれぞれ $L(r)$, $L(r')$ とするとき、$L(r \cdot r') = L(r) \cdot L(r')$
6. r の表す言語を $L(r)$ とするとき $L(r^*) = L(r)^*$
7. r の表す言語を $L(r)$ とするとき $L((r)) = L(r)$

□

例 2.10　例 2.9 の正則表現は言語 $\{0, 00, 01, 000, 001, 010, 011, 0000, \cdots\}$、つまり、0 の後ろに 0 と 1 が 0 個以上続くような記号列全体を表している。なぜなら:

$$\begin{aligned}
L(0 \cdot (0 \mid 1)^*) &= L(0) \cdot L((0 \mid 1)^*) \\
&= L(0) \cdot L((0 \mid 1))^* \\
&= L(0) \cdot L(0 \mid 1)^* \\
&= L(0) \cdot (L(0) \cup L(1))^* \\
&= \{0\} \cdot \{0,1\}^* \\
&= \{0\} \cdot \{\epsilon, 0, 1, 00, 01, 10, 11, \cdots\} \\
&= \{0, 00, 01, 000, 001, 010, 011, \cdots\}
\end{aligned}$$

□

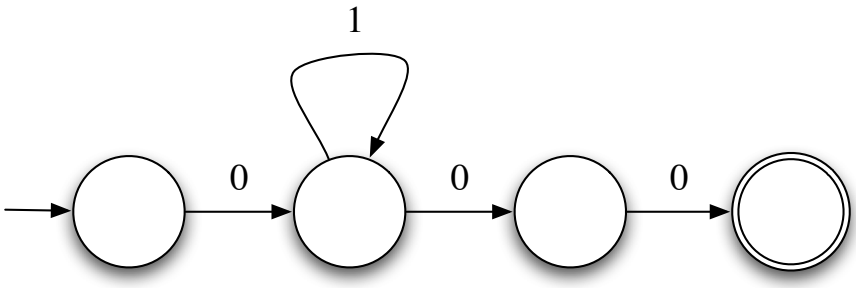


図 2.1 有限オートマトン

正則表現に用いられる演算子 $*$, $\cdot$, $|$ の優先順位は、$*$ が最も高く、ついで $\cdot$、最も低いのが $|$ となる。また $r \cdot r'$ は rr' と書いてもよい。

以降、正則表現に関して次の略記法を用いることがある。

$$r^+ \equiv rr^*$$
$$r? \equiv r \mid \epsilon$$
$$[abc] \equiv a \mid b \mid c$$
$$[a\text{-}z] \equiv a \mid \cdots \mid z$$

例 2.11 C 言語の 10 進定数は 0 以外の数字で始まり、その後に 0 個以上の数字が続くので、正則表現 [1 - 9] [0 - 9]* で表せる（練習問題 2.3, 2.4, 2.5 参照）。□

2.5 有限オートマトン

正則表現は、**有限オートマトン**（finite automaton）という計算モデルと極めて関連が深い。

直観的には、有限オートマトンはいくつかの**状態**（state）と状態間の**遷移**（transition）からなる有向グラフで表される（図 2.1）。状態には 1 つの**初期状態**（initial state）、複数個の**受理状態**（accepting state）があり、また、各遷移にはアルファベット中の 1 文字（ラベル）が付けられている。最初、有限オートマトンは初期状態に制御がある。そして、入力を 1 文字読むごとに、入力を現在制御が置かれている状態から伸びる遷移のラベルと照合し、合致する遷移先の状態に制御を移す。このように状態間の遷移を繰り返し、入力を読み切ったときに受理状態に制御があれば、その入力を受理する。図 2.1 の有限オートマトンは、0 で始まり、次に 1 が 0 個以上続き、00 で終わるような記号列をすべて受理する。なお、以下では、ϵ をラベルに持つ遷移はなく、かつ各状態について、入力となる記号と照合される遷移は常に 1 つだけであると仮定する[*5]。

定義 2.3 （決定性）有限オートマトン A は 5 つ組 $(Q, \Sigma, \delta, q_0, F)$ である。ここで、

- Q：状態の有限集合
- Σ：入力記号の有限集合

[*5] すなわち、ここで考える有限オートマトンは**決定性**（deterministic）であると仮定する。

- δ：遷移関数（transition function）。状態と入力記号を与えると状態 1 つを返す
- $q_0 \in Q$：初期状態
- $F \subseteq Q$：受理状態の集合

□

実は、正則表現と有限オートマトンは等価[*6]であり、任意の正則表現から等価な有限オートマトンを構成したり、任意の有限オートマトンから等価な正則表現を構成することができる。手法の詳細は別の書籍を参照されたい（例えば [7]）。この講義で必要になるのは、極めて簡単な形式の正則表現から有限オートマトンを構成することだけなので、これについて直観的な説明をするにとどめる。

基本的な正則表現について、対応する有限オートマトンは図 2.2 のようになる。これを再帰的に用いて、有限オートマトンを順に構成し、最後に初期状態と受理状態を定めればよい。ただし ϵ の扱いは注意が必要である。$\epsilon \cdot a = a \cdot \epsilon = a$ などの性質を用いて、あらかじめ正則表現から ϵ を除去し、有限オートマトンを構成しなければならない[*7]。

2.6 練習問題

問題 2.1　次の言語を表す正則表現を示せ。

1. アルファベット $\{0, 1\}$ 上の記号列のうち、0 から始まり 1 が 0 回以上続くもの全体からなる言語
2. アルファベット $\{a, b, c\}$ 上の記号列のうち、1 個以上の a と 1 個以上の b を含むもの全体からなる言語
3. アルファベット $\{0, 1\}$ 上の記号列のうち、0 と 1 が交互に出現するもの全体からなる言語

問題 2.2　正則表現 $(0 \mid 1)^*0(0 \mid 1)(0 \mid 1)$ が表す言語は何か。言葉で説明せよ。

問題 2.3　C 言語の空白記号は、空白（␣）、タブ（`\t`）、改行（`\n`）が 1 個以上並んだ文字列である。これを正則表現で表せ。

問題 2.4　C 言語の 16 進定数は 0x あるいは 0X で始まり、数字および a, b, c, d, e, f, A, B, C, D, E, F が 1 個以上続く。これを正則表現で表せ。

問題 2.5　C 言語の識別子は A〜Z, a〜z, 0〜9, 下線（_）からなる文字列である。ただし、最初の文字に数字を使うことはできない。これを正則表現で表せ。

問題 2.6　アルファベットを $\{0, 1\}$ とするとき、次の言語を受理する有限オートマトンを示せ。

1. 00 で終わる記号列全体

[*6] 任意の正則表現に対して、その表す言語を受理する有限オートマトンが存在する。また、任意の有限オートマトンに対して、それが受理する言語を表す正則表現が存在する。

[*7] ϵ については、この方法ではうまくいかない場合があるかもしれない。厳密には、正則表現 $\rightarrow$ ϵ 遷移つき非決定性有限オートマトン $\rightarrow$ 非決定性有限オートマトン $\rightarrow$ 決定性有限オートマトンの順に変換を行わなければならない。詳細は [7] などを参照されたい。

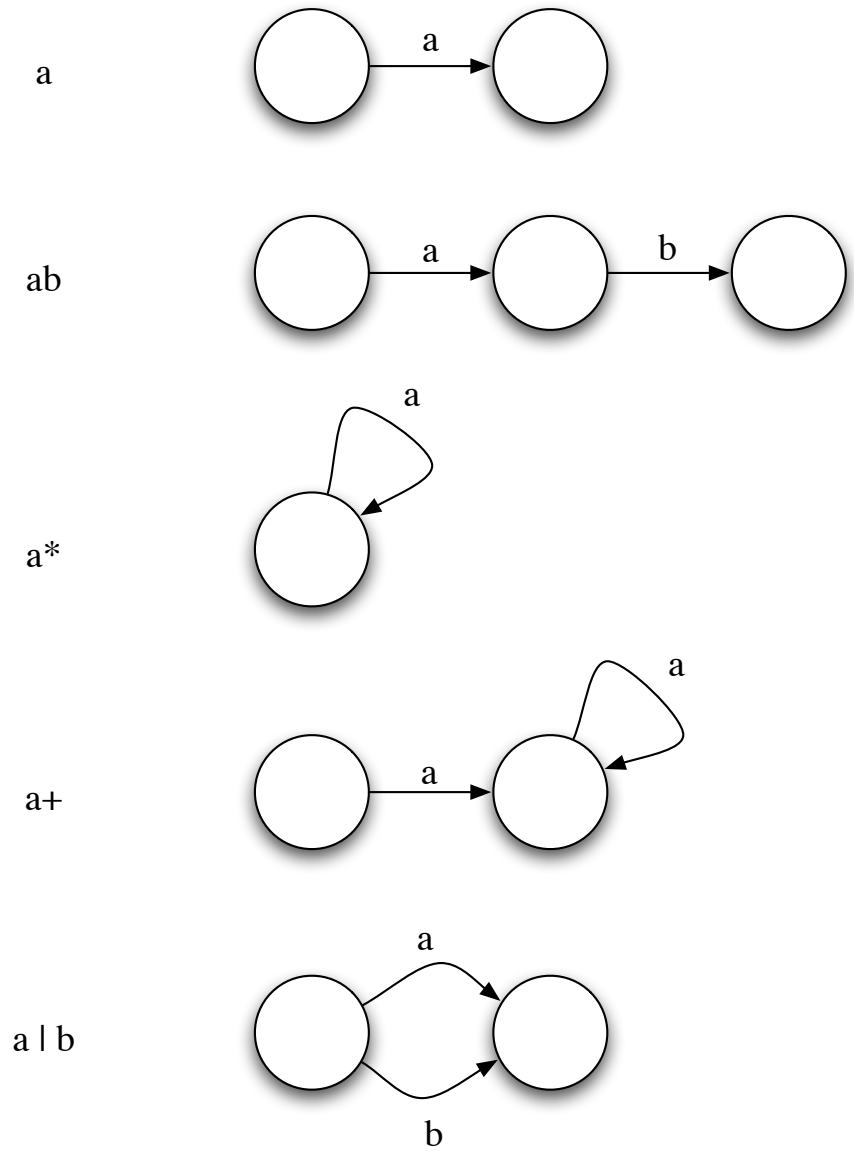


図 2.2 正則表現と対応する有限オートマトン

2. 011 を途中に含む記号列全体

問題 2.7 正則表現 $00(0 \mid 1)^*$ に対応する有限オートマトンを構成せよ。

第 3 章

字句解析

3.1 用語

説明を始める前に、この章を通して用いられる用語を整理しておくことにしよう。

1.1 節で述べたように、字句解析の目的は、文字列としての原始プログラムから意味のある文字列を切り出すことである。ここで言う「(原始言語で) 意味のある文字列」のことを**字句要素** (lexeme) という。例えば、C 言語の字句要素には次のようなものがある。

- 変数名、関数名などの**識別子** (identifier)
- int, while など、C 言語プログラム中で特別な意味を持ち、他の用途に用いることのできない語 (**予約語**, keyword)
- データの値を表す**定数** (constant)。整数定数、浮動小数点定数、文字定数などがある。
- 文字列を表す**文字列リテラル** (string literal)
- +, *, =, ==などの**演算子** (operator)
- 括弧、セミコロンなどの**区切り記号** (separator)

空白、タブ、改行文字、コメントなどは C 言語のプログラムの意味には影響を与えないので、字句解析の際に捨てられる。

実際の字句解析では、切り出した字句要素をそのまま構文解析部に渡すことはせず、字句要素を**トークン** (token) というデータに変換してから渡す。トークンには以下のような情報が含まれる。

- 字句要素の種類。例えば ID (識別子)、IF (予約語 if)、RELOP (比較演算子) など[*1]。
- 付加情報。例えば、識別子 argc には、その識別子の名前、すなわち “argc” が付加情報として付けられる。また整数定数 453 には、その定数の値、すなわち 453 という整数が付加情報として付けられる。付加情報のない場合もある。

[*1] 以降、特に断らない限り、英大文字と数字からなるゴシック文字列を字句要素の種類として用いる。

3.2　素朴な字句解析とその問題点

字句解析アルゴリズムの前提として、原始プログラムを読む回数は 1 回に押さえたい。つまり、原始プログラムのファイルを初めから終わりまで順に 1 回読み込むだけで字句解析を終わらせるようにしたい。これは、CPU の計算に比べて、ファイル（二次記憶）との入出力はずっと時間がかかるからである。

また、コンパイラ中に巨大な文字配列を確保し、そこに原始プログラムを全部読み込んでから字句解析をするという方法も好ましくない。巨大な配列を確保すると、コンパイラの使用メモリ量が増えてしまう。コンパイラは比較的重いプログラムなので、なるべく使用メモリ量を節約して設計したい。

すると、素朴な字句解析プログラムは次のようになるであろう。原始プログラムファイルから getc() 関数で 1 文字読み、その種類によって場合分けを行う。これをファイルの終わりまで繰り返す。

```
#include <stdio.h>
/* 後述するように、字句要素は整数で表現する */
#define IF 1
/* 以下、字句要素の定義が続く */

/* 次の字句要素の番号を返す */
int simple_lexical_analysis()
{
  char c;
  /* file: 原始プログラムファイル */
  while((c = getc(file)) != EOF) {
    /* 予約語 if の処理 */
    if (c == 'i') {
      c = getc(file);
      if (c == 'f') {
        return IF;
      }
    }
    /* 以下、他の字句要素の処理が続く */
  }
}
```

しかし、このプログラムには問題がある。字句要素ごとの処理が複雑になってしまうのである。C 言語には、i で始まる予約語は if の他に int もある。またユーザが i で始まる識別子を使う可能性も高い。このような可能性をすべて網羅して場合分けを書くのは大変である。ましてやこのプログラムを自動的に生成するのは難しい。

以下の節では、正則表現を用いて字句解析部を自動的に生成する手法を述べる。

3.3　正則表現を基にした字句解析

字句解析の手法では、次のことを考える必要がある。

1. 字句要素を定義する方法

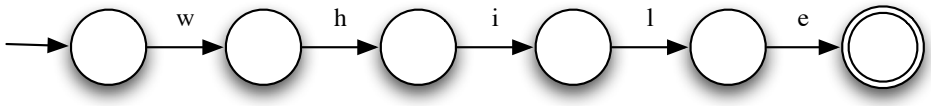


図 3.1 while に対応する有限オートマトン

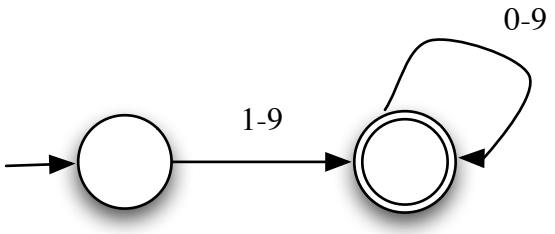


図 3.2 10 進数に対応する有限オートマトン

2. 1 で定義した字句要素を検出する方法
3. 字句要素を検出したときに何らかの動作を行う方法（整数定数の字句要素を検出したときにその整数の値を求める、など）

ここで述べる字句解析手法では、各字句要素を正則表現で表し、対応する有限オートマトンを構成して字句要素を受理させる。上に述べた 1 の実現方法として正則表現を、2 の実現方法として有限オートマトンを利用するのである。例えば、予約語 while に対応する正則表現は $while$ であり、この正則表現と等価な有限オートマトンは図 3.1 のようになる。また、C 言語の 10 進定数は 0 以外の数字から始まる数字の列であるので、これに対応する正則表現は [1-9][0-9]* であり、これと等価な有限オートマトンは図 3.2 のようになる。

このような有限オートマトンをすべての字句要素に対して作り、並列に動作させる。つまり、原始プログラムファイルから 1 文字読み込んで、それを入力記号としてすべての有限オートマトンを動作させる。これを繰り返して、ある有限オートマトンが受理状態に到達すれば、字句要素が見つかったことになる。例えば ‘w’, ‘h’, ‘i’, ‘l’, ‘e’ の順に原始プログラムファイルから文字を読み込んだとすると、図 3.1 の有限オートマトンは受理状態に到達し、一方、それ以外の有限オートマトン（例えば図 3.2 のオートマトン）は受理状態に到達しない。従って、字句要素 while が見つかったことになる。字句要素が 1 つ見つかれば、すべての有限オートマトンをリセットし（初期状態に戻して）、再び原始プログラムファイルからの文字の読込みを始める。

ただし、この方法ではまだ問題点が残っている。

1. 一つの文字列が 2 つ以上の有限オートマトンで受理されることがある。例えば文字列 ‘while’ は図 3.1 の有限オートマトンで受理されるが、C 言語の識別子を受理する有限オートマトン（図 3.5、練習問題 2.5 を参照）でも受理される。この場合、C

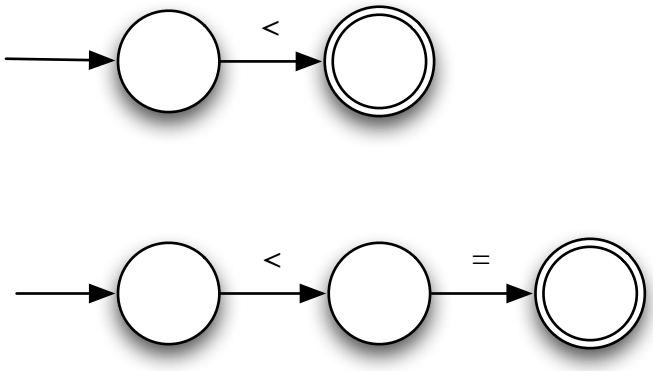


図 3.3　比較演算子に対する有限オートマトン

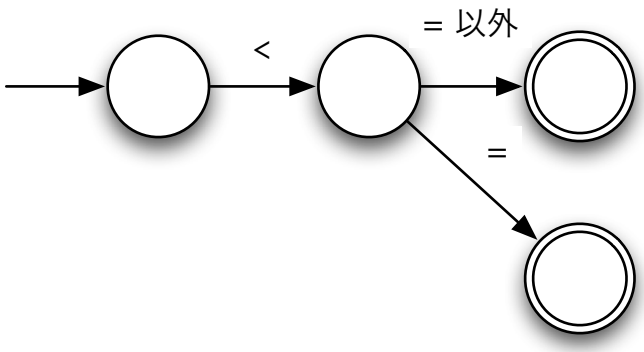


図 3.4　比較演算子に対する有限オートマトン（改良版）

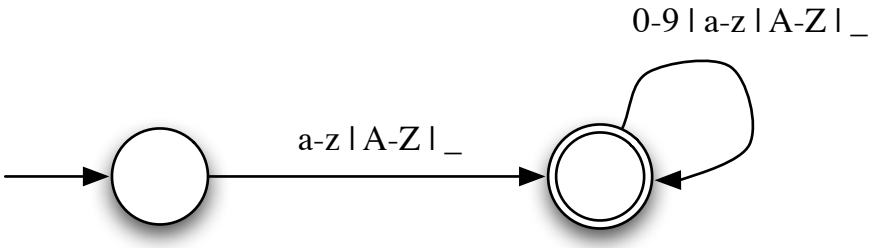


図 3.5　識別子に対する有限オートマトン

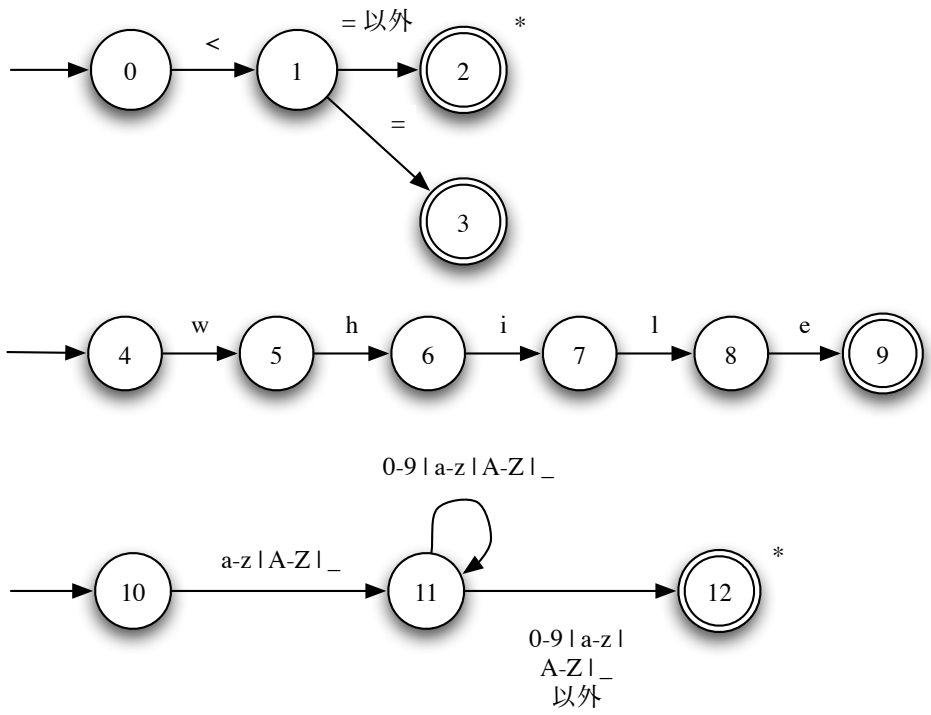


図 3.6　字句解析のための有限オートマトン群

言語では予約語として解釈しなければならない。すなわち、図 3.1 の有限オートマトンを優先して処理しなければならない。

2. (最長一致の原則) 字句要素を認識するときは、最も長いものを採用しなければならない。例えば<と<=という 2 つの字句要素を考えよう。これらの字句要素を受理する有限オートマトンを図 3.3 に示す。‘<’ を読み込んだとき、上のオートマトンは受理状態に到達する。しかし、もしこの次の文字が ‘=’ であれば、<ではなく<=という字句要素と解釈しなければならない。つまり、下のオートマトンのみで受理させなければならない。

2 に挙げた図 3.3 の問題点を改良する一つの案を図 3.4 に示す。ポイントは、‘<’ を受理する有限オートマトンを改良し、‘<’ の次に ‘=’ 以外の文字を読み込んだときに受理する、とした点である。一般に、最長一致の原則はこれと同様の方法で解決できる。

ただし、注意しなければならない点がある。上のオートマトンで字句要素 ‘<’ を受理したとき、本来の字句要素 ‘<’ よりも 1 文字余計に読み込んでしまっている。この余計に読み込んだ文字は、次の字句要素を構成する文字であるかもしれないため、‘<’ を受理した後、読み込みを破棄しておかなければならない。

3.4　字句解析プログラムの実現

有限オートマトンを基礎とする字句解析プログラムの実現について説明を行う。なお、説明のため、ここでは認識すべきトークンを<, <=, while, それに C 言語の識別子のみとす

る。また、これらのトークンを認識する有限オートマトンの状態に番号を付けておく（図 3.6）。

3.4.1 ファイルとの入出力

これから作成する字句解析プログラムは、原始プログラムファイルから 1 文字ずつ文字を読み、処理する、という操作をファイルの終わりまで繰り返す。ただし、3.3 節で述べたように、字句によっては、字句を認識した後に読み込みを破棄しておかなければならない。

したがって、原始プログラムファイルからの読み込みは getc() 関数を、読み込みの破棄は ungetc() 関数をそれぞれ用いればよい[*2]。ただし、この部分は後で改良を試みたいので、直接 getc(), ungetc() を使うのではなく、次のような関数を通して使うことにする。

```
int nextchar(FILE* infile)
{
   return getc(infile);
}

void backchar(int c, FILE* infile)
{
   ungetc(c, infile);
}
```

3.4.2 データの表現

コンパイラでは通常、原始言語のトークンに通し番号を付け、コンパイラの内部でもその番号（整数）でそれぞれのトークンを表す。次の例では、<, <=, while, 識別子にそれぞれ 0, 1, 2, 3 という番号を割り当てている。

```
#define LT  0
#define LE  1
#define WHILE 2
#define ID  3
```

すると次節で示すように、字句解析プログラムはトークンを表す番号を返り値とする関数となり、したがって int 型の値を返す関数となる。

ただし識別子については、実際の識別子名（i, main などの変数名や関数名）も字句解析の出力とする必要がある。これは大域変数 char yytext[80]; に格納されるものとする[*3]。

字句解析プログラムでは、ほかに次のような大域変数を用いる。

- int state;…有限オートマトンの現在の状態を表す。初期値は 0。
- int initial_state;…現在処理中の有限オートマトンの初期状態を表す。初期値は 0。
- FILE *file;…原始プログラムのファイルポインタ。

[*2] ungetc() 関数は今まで使ったことがないと思われるので、オンラインマニュアルなどでどういう関数か調べておくこと。getc() 関数を知らない、もしくは忘れている場合もオンラインマニュアルなどで調べておくこと。

[*3] 字句解析部の後に続く構文解析部から識別子名を参照できるように、大域変数を用いている。

3.4.3 字句解析プログラム

次に、字句解析プログラムの本体である nexttoken() 関数のコードを示す。この関数は、構文解析部から必要に応じて呼び出されることを想定しており、次のトークンを表す番号を返り値とする。また、トークンが識別子の場合には、大域変数 yytext にその識別子名を格納する。

```
 1  #include <ctype.h>
 2
 3  int fail(char *s, int i)
 4  {
 5    int j;
 6    for (j = 0; j < i; j ++) {
 7      backchar(s[j], file);
 8    }
 9    switch (start) {
10    case 0:
11      start = 4;
12      break;
13    case 4:
14      start = 10;
15      break;
16    case 10:
17      /* どのトークンも認識失敗 */
18      break;
19    }
20    return start;
21  }
22
23  int nexttoken()
24  {
25    int c; /* 現在処理中の文字 */
26    char s[80]; /* 初期状態から現在までに処理した文字列 */
27    int i = 0;
28
29    /* 大域変数の初期化 */
30    state = 0;
31    start = 0;
32    yytext[0] = '\0';
33
34    while (1) {
35      switch (state) {
36      case 0:
37        c = nextchar(file);
38        if (c == '␣' || c == '\t' || c == '\n')
39          state = 0; /* 空白の読み飛ばし */
40        else if (c == '<') {
41          state = 1;
42          s[i++] = c;
43        } else /* 1 つ目のオートマトンは失敗 */
44          state = fail(s, i);
45        break;
46      case 1:
```

```
47        c = nextchar(file);
48        if (c == '=') {
49          state = 2;
50          s[i++] = c;
51        } else
52          state = 3;
53        break;
54      case 2:
55        return LE;
56      case 3:
57        backchar(c, file); /* 1 文字破棄 */
58        return LT;
59      case 4:
60        c = nextchar(file);
61        s[i++] = c;
62        if (c == 'w')
63          state = 5;
64        else
65          state = fail(s, i);
66        break;
67      case 5:
68        c = nextchar(file);
69        s[i++] = c;
70        if (c == 'h')
71          state = 6;
72        else
73          state = fail(s, i);
74        break;
75      case 6:
76        c = nextchar(file);
77        s[i++] = c;
78        if (c == 'i')
79          state = 7;
80        else
81          state = fail(s, i);
82        break;
83      case 7:
84        c = nextchar(file);
85        s[i++] = c;
86        if (c == 'l')
87          state = 8;
88        else
89          state = fail(s, i);
90        break;
91      case 8:
92        c = nextchar(file);
93        s[i++] = c;
94        if (c == 'e')
95          state = 9;
96        else
97          state = fail(s, i);
98        break;
99      case 9:
100       return WHILE;
101     case 10:
```

```
102         c = nextchar(file);
103         s[i++] = c;
104         if (isalpha(c) || c == '_')
105           state = 11;
106         else
107           state = fail(s, i);
108         break;
109       case 11:
110         c = nextchar(file);
111         s[i++] = c;
112         if (isalpha(c) || isdigit(c) || c == '_')
113           state = 11;
114         else
115           state = 12;
116         break;
117       case 12:
118         backchar(c, file);
119         s[i] = '\0'; /* 読み込みすぎた c を \0 で上書き */
120         strcpy(s, yytext);
121         return ID;
122       }
123     }
124   }
```

大まかには、現在の状態 (state) と次の文字 (c) から次の状態を決定し、state の値を更新し、状態による分岐を行う (35 行目)。その結果受理状態に到達すれば、認識したトークンの番号を返す。受理状態に到達できなかった場合は、fail() 関数を呼び出し、初期状態 start の値を更新して次のオートマトンによる受理を試す。これをオートマトンの数だけ繰り返していく。どのオートマトンでも受理できなかった場合は、原始プログラムに誤りがあったことになるので、エラー処理を行う[*4]。

図 3.6 で * を付けた状態は、認識したトークンよりも 1 文字先まで読み込んでしまっていることを示す。例えばトークン '<' は、トークンの読み込み自体は状態 1 で終了しているが、'<' か '<=' かを決定するために 1 文字先まで読み込み、状態 2 に至る必要がある。識別子も、識別子に含まれない文字が出てきた時点で初めて、識別子の終わりが認識できるため、状態 12 に到達したときには 1 文字先まで読んでしまっていることになる。そのため、状態 2 と状態 12 については、backchar() 関数により、1 文字分読み込みを破棄している。

入力の破棄はオートマトンを切り替える際にも発生する。例えば状態 6 で、文字 'a' を読み込んでトークン while の認識に失敗した場合、すでに読み込んだ 'w', 'h', 'a' を破棄してから次のオートマトンに切り替えなければならない。そのために、既に読み込んだ文字を蓄える配列 s、読み込んだ文字数を保持する変数 i を局所変数として用意している。fail() を呼び出すときには s と i を引数として渡し、fail() 中で入力の破棄を行っている。

また状態 12 では、現在の s の内容、すなわち認識した識別子名を大域変数 yytext にコピーしている。

[*4] isalpha(c) 関数、isdigit(c) 関数はそれぞれ、c がアルファベットか、c が数字かを判定する関数である。詳細はオンラインマニュアルなどを参照のこと。

3.4.4　字句解析部自動生成プログラム lex

3.4 節で示した字句解析プログラムは各状態で行うべき処理がかなりパターン化されており、図 3.6 のオートマトンが与えられれば、3.2 節で示したプログラムに比べて自動的に生成しやすい。

実は、字句解析部を自動生成するプログラムはいくつもある。これらはいずれも、認識すべきトークンの正則表現と、トークンに対して行うべき処理を指定すると、字句解析プログラムを自動生成してくれる。内部的には、正則表現から図 3.6 に相当するオートマトンを自動生成し（アルゴリズムは例えば [7] を参照)、さらに 3.4 節に示したような字句解析プログラムを自動生成している。

例として、Unix に標準装備されている lex プログラムを取り上げる。3.4 節のプログラムと同等の字句解析プログラムを自動生成するには、以下のようなルールを書き、lex で処理すればよい。

```
%{
#define LT 0
#define LE 1
#define WHILE 2
#define ID 3
%}

%%
[ \t\n]+         {}
"<"              {return LT;}
"<="     {return LE;}
while    {return WHILE;}
[A-Za-z][0-9A-Za-z_]+    {return ID;}
%%
```

‘%{’ から ‘%}’ までがトークンの番号指定、‘%%’ に挟まれた部分が認識すべきトークンを表す正則表現と、そのときに行われる処理（C 言語プログラム）である。識別子名は自動的に yytext という変数に格納される。

lex やその他の字句解析部自動生成ソフトウェアの詳細は、ここでは省略する。各自で適宜参考資料にあたられたい。

3.5　入力のバッファリング

本章の最後に、nextchar() 関数、backchar() 関数の改良について考えよう。

基本となる考え方は、プログラム中に**バッファ**（buffer）と呼ばれる領域を用意しておく、というものである。あらかじめ、原始プログラムから一定量の文字をバッファに取り込んでおき、nextchar() と backchar() はバッファから 1 文字読み込んだり、バッファに文字を戻したりするようにする（図 3.7)。これにより、ungetc() 関数を使う必要はなくなる[*5]。

[*5] 本章の例では示していないが、原始プログラムからバッファへの読み込みも、getc() を用いて 1 文字ずつ読み込むのではなく、gets() などにより複数の文字をまとめて読み込むことができ、やはり効率が良く

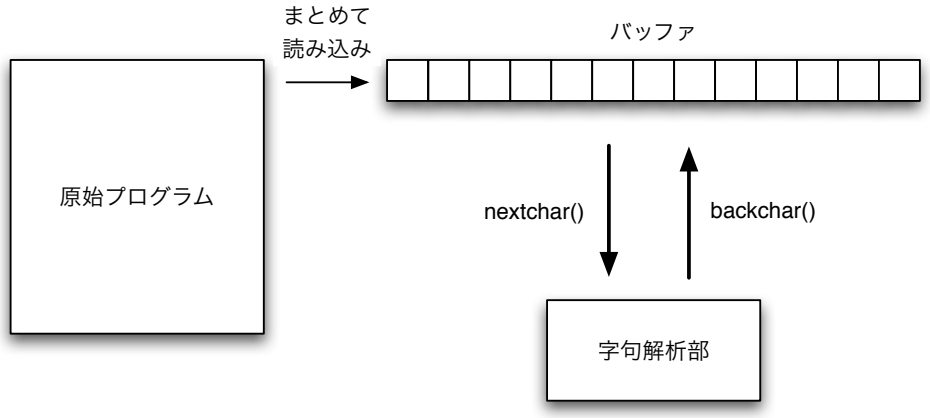


図 3.7　入力のバッファリング

バッファを用いた改良版 nextchar()、backchar() を次に示す。この例では、長さ 2048 のバッファ（char 型の配列）を用意し、前半分と後ろ半分を交互に用いている。変数 forward は大域変数で、現在処理している文字の添字を保持している。

```
int buffer[2048];
/* 前半分と後ろ半分に分けて使う。初期化はしてあると仮定 */

int nextchar()
{
  int i;
  if (forward == 1023) { /* 前半分の終わりだったら */
    for (i = 1024; i < 2048; i++) {
      buffer[i] = getc(file);
    }
    forward ++;
  }
  else if (forward == 2047) {
    /* 後ろ半分の終わりだったら */
    for (i = 0; i < 1024; i++) {
      buffer[i] = getc(file);
    }
    forward = 0;
  }
  else forward ++;
  return buffer[forward];
}

void backchar()
{
  if (forward == 0) {
    forward = 2047;
  } else {
    forward --;
```

なる。

```
  }
}
```

字句解析部でバッファを用いると、入力の破棄に関してもう一つ利点が生まれる。3.4 節の字句解析プログラムでは、backchar() の呼び出しが発生したときに備え、現在処理中の文字列を nexttoken() 中に保持しておく必要があった（変数 s)。しかし、バッファを用いると、現在処理中の文字列は必ずバッファ中に残っている。これを利用すると、現在処理中の文字列を nexttoken() 中で保持しておく必要がなくなる。改良版の nexttoken()、fail() を次に示す。変数 beginning は、現在処理しているオートマトンが初期状態であったときの forward の位置、すなわち現在処理中のトークンの先頭の添字を保持している。

```
#include <ctype.h>

int fail()
{
  forward = beginning;
  switch (start) {
  case 0:
    start = 4;
    break;
  case 4:
    start = 10;
    break;
  case 10:
    /* どのトークンも認識失敗 */
    break;
  }
  return start;
}

int nexttoken()
{
  int c; /* 現在処理中の文字 */

  /* 大域変数の初期化 */
  state = 0;
  start = 0;
  yytext[0] = '\0';

  while (1) {
    switch (state) {
    case 0:
      c = nextchar();
      if (c == '␣' || c == '\t' || c == '\n') {
        state = 0; /* 空白の読み飛ばし */
        beginning ++;
      }
      else if (c == '<')
        state = 1;
      else /* 1 つ目のオートマトンは失敗 */
        state = fail();
      break;
    case 1:
```

```
    c = nextchar();
    if (c == '=')
      state = 2;
    else
      state = 3;
    break;
  case 2:
    return LE;
  case 3:
    backchar(); /* 1 文字破棄 */
    return LT;
  case 4:
    c = nextchar();
    if (c == 'w')
      state = 5;
    else
      state = fail();
    break;
  case 5:
    c = nextchar();
    if (c == 'h')
      state = 6;
    else
      state = fail();
    break;
  case 6:
    c = nextchar();
    if (c == 'i')
      state = 7;
    else
      state = fail();
    break;
  case 7:
    c = nextchar();
    if (c == 'l')
      state = 8;
    else
      state = fail();
    break;
  case 8:
    c = nextchar();
    if (c == 'e')
      state = 9;
    else
      state = fail();
    break;
  case 9:
    return WHILE;
  case 10:
    c = nextchar();
    if (isalpha(c) || c == '_')
      state = 11;
    else
      state = fail();
    break;
```

```
    case 11:
      c = nextchar();
      if (isalpha(c) || isdigit(c) || c == '_')
        state = 11;
      else
        state = 12;
      break;
    case 12:
      backchar();
      strncpy(&buffer[beginning],
              forward - beginning + 1,
              yytext);
      return ID;
    }
  }
}
```

第 4 章

文脈自由文法

構文解析部の処理では、何らかの形で原始プログラム言語の構文をコンピュータで扱わなければならない。2 章で取り上げた正則表現は、原始プログラム言語で扱う字句要素を定義することはできるが、それらをどのように組み合わせれば文法的に正しいのか、については定義することができない。

本章では、プログラミング言語の構文を扱うための基本的な理論である文脈自由文法について説明し、構文解析に必要ないくつかの重要な性質について述べる。

なお、本章でも 2 章で述べた言語理論に関する用語を用いる。

4.1 文脈自由文法

4.1.1 定義

プログラミング言語の構文を定義するには、通常、**文脈自由文法**（context free grammar）を用いる。例えば C 言語の if 文の構文は、文脈自由文法では次のように書くことができる。

$$
\begin{aligned}
&\mathit{selection\_statement} \to \mathsf{if}\ (\ \mathit{expression}\ )\ \mathit{statement} \\
&\mathit{selection\_statement} \to \mathsf{if}\ (\ \mathit{expression}\ )\ \mathit{statement}\ \mathsf{else}\ \mathit{statement}
\end{aligned}
$$

矢印 $\to$ とその両辺からなる式を**生成規則**（production rule）、もしくは略して**規則**（rule）という。生成規則の左辺は記号 1 つ、右辺は記号列から成っている。記号は**非終端記号**（nonterminal symbol）と**終端記号**（terminal symbol）に分けられる。この例では、*selection_statement*, *expression*, *statement* が非終端記号、if, (,), else が終端記号である。生成規則の左辺には非終端記号しか出現できない。右辺には非終端記号、終端記号ともに出現することができる。非終端記号のうち 1 つを特に**開始記号**（start symbol）という。

もう 1 つ例として「数字、+、− からなる式」を表す文脈自由文法を示そう。

$$
\begin{aligned}
\mathit{list} &\to \mathit{list} + \mathit{digit} \\
\mathit{list} &\to \mathit{list} - \mathit{digit} \\
\mathit{list} &\to \mathit{digit} \\
\mathit{digit} &\to \mathsf{0} \mid \mathsf{1} \mid \mathsf{2} \mid \mathsf{3} \mid \mathsf{4} \mid \mathsf{5} \mid \mathsf{6} \mid \mathsf{7} \mid \mathsf{8} \mid \mathsf{9}
\end{aligned}
\tag{4.1}
$$

この例では *list*, *digit* が非終端記号、数字と +, − が終端記号である。また、最後の規

則の | は「または」の意味である。なお、この例から分かるように、左辺の非終端記号が同じ生成規則の右辺に出現してもかまわない。

まとめると、文脈自由文法 G は次のような構成要素からなる。

1. 非終端記号の有限集合 V
2. 終端記号の有限集合 T
3. 生成規則の有限集合 P
4. 開始記号 $S \in V$。非終端記号のうち 1 つ

記号を 1 つも含まない列もある。これを**空列**（empty string）とよび、ϵ という記号で表す。また、プログラミング言語の構文を表す表現として、ほかに **BNF 記法**（Backus Naur Form）というものがあるが、文脈自由文法とほぼ同じと考えてよい。BNF 記法で上の文法を書くと次のようになる。

$$
\begin{aligned}
\mathit{list} &::= \mathit{list} + \mathit{digit} \\
\mathit{list} &::= \mathit{list} - \mathit{digit} \\
\mathit{list} &::= \mathit{digit} \\
\mathit{digit} &::= 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9
\end{aligned}
$$

以降の説明では、次のように記法を決めておくことにする。

1. 前のほうの英小文字 $(a, b, c, \cdots)$、演算子記号 $(+, -, \cdots)$、括弧やコンマなどの区切り記号、数字、太字の記号列は終端記号を表す。
2. 前のほうの英大文字 $(A, B, C, \cdots)$、S (開始記号)、英小文字のイタリック体の名前 (*expr*, *stmt* など) は非終端記号を表す。
3. 後ろのほうの英大文字 $(X, Y, Z, \cdots)$ は非終端記号または終端記号を表す。
4. 後ろのほうの英小文字 $(w, x, y, z, \cdots)$ は終端記号列を表す。
5. 小文字のギリシャ文字 $(\alpha, \beta, \gamma, \cdots)$ は、記号列 (終端記号と非終端記号が混ざっていてよい) を表す。
6. 最初に現れる生成規則の左辺を開始記号とする。また、S を開始記号とすることが多い。

4.1.2　文脈自由文法の意味

文脈自由文法 $G = (V, T, P, S)$ は、T、すなわち終端記号の集合をアルファベットとする言語を表している。では、具体的にどのような言語を表しているのだろうか。

文法の言語の定義方法にはいくつかあるが、その 1 つに「生成規則を書き換え規則と考える」というものがある。つまり、開始記号 S を S-生成規則の右辺の記号列 α で置き換え、さらに α に含まれる非終端記号を、その記号の生成規則の右辺で置き換える。この操作を、記号列が終端記号のみになるまで繰り返す。このようにして得られる終端記号列すべての集合を、その文法の**言語**（language）という。

例 4.1 (4.1) の文法

$$list \to list + digit \tag{4.2}$$
$$list \to list - digit \tag{4.3}$$
$$list \to digit \tag{4.4}$$
$$digit \to 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \tag{4.5}$$

について、(4.2) を用いると、開始記号 $list$ を右辺の記号列 $list + digit$ に置き換えることができる。これを繰り返すと、以下のように終端記号列 $9 - 5 + 2$ を得ることができる。

$$\begin{aligned}
\underline{list} &\Rightarrow \underline{list} + digit && \text{(4.2) を適用}\\
&\Rightarrow \underline{list} - digit + digit && \text{(4.3) を適用}\\
&\Rightarrow \underline{digit} - digit + digit && \text{(4.4) を適用}\\
&\Rightarrow 9 - \underline{digit} + digit && \text{(4.5) を適用}\\
&\Rightarrow 9 - 5 + \underline{digit} && \text{(4.5) を適用}\\
&\Rightarrow 9 - 5 + 2 && \text{(4.5) を適用}
\end{aligned}$$

もう少し形式的に書くと次のようになる。生成規則 $A \to \gamma$ を記号列 $\alpha A \beta$ に適用して A を γ に置き換えると $\alpha\gamma\beta$ となる。このとき、$\alpha A \beta$ から $\alpha\gamma\beta$ が**導出される**（derive）といい、$\alpha A\beta \Rightarrow \alpha\gamma\beta$ と表す。α_1 に 0 回以上置き換えを適用して α_n が導出されるとき、$\alpha_1 \overset{*}{\Rightarrow} \alpha_n$ と書く。$S \overset{*}{\Rightarrow} \alpha$ のとき、α を**文形式**（sentential form）という。とくに α が終端記号のみからなるとき、**文**（sentence）という。

文法 G の言語 $L(G)$ とは、$L(G) = \{w \mid S \overset{*}{\Rightarrow} w\}$ である。ただし S は G の開始記号である。すなわち、G の言語とは、G の開始記号から導出される文の集合である。

4.2 解析木と導出

文脈自由文法の開始記号にどのように生成規則が適用され、終端記号列が生成されたのかを図示したものを**解析木**（parse tree）という。解析木は、導出の様子を図示したものであると考えられ、文の “構文的な構造” を表したものだと言える。例えば、(4.1) に示した文脈自由文法について、式 $9 - 5 + 2$ の解析木は図 4.1 のようになる。

形式的には、解析木は次のような木である。

1. 根は開始記号をラベルとして持つ。
2. 葉は終端記号または ϵ をラベルとして持つ。
3. 内部の節は非終端記号をラベルとして持つ。
4. 非終端記号 A をラベルに持つ内部の節の子が左から順に $X_1, X_2, \cdots, X_n$ とすると、$A \to X_1 X_2 \cdots X_n$ は生成規則である。$A \to \epsilon$ に対しては、節 A は ϵ をラベルとする子を持つ。

解析木の葉を左から右に読んでいくと、終端記号列が得られる。これが開始記号から生成される終端記号列、すなわち文である。これを解析木の**結果**（result）という。

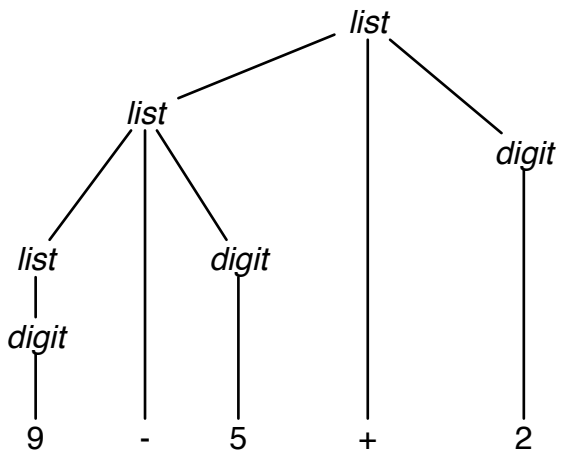


図 4.1　解析木の例

4.2.1　解析木の意義

解析木は、式の値を計算したりプログラムを翻訳する際にいろいろ役立つ。例えば、先に述べた数式の値を計算することを考えよう。各非終端記号が属性 v を持ち、各生成規則に次のようなプログラム片が付けられているとする。なお、左辺と右辺に同じ非終端記号が出現する場合には、便宜上、それぞれ l と r という添字を付けて区別している。

$$
\begin{aligned}
list &\to list + digit \quad \{list_l.v = list_r.v + digit.v;\} \\
list &\to list - digit \quad \{list_l.v = list_r.v - digit.v;\} \\
list &\to digit \quad \{list.v = digit.v;\} \\
digit &\to 0 \quad \{digit.v = 0;\} \\
digit &\to 1 \quad \{digit.v = 1;\} \\
&\cdots \\
digit &\to 9 \quad \{digit.v = 9;\}
\end{aligned}
$$

解析木の節点の属性 v の値を、その節点に対応する生成規則に付けられたプログラム片に基づいて計算すると、その解析木の結果である式の値が求められる。式 $9-5+2$ の解析木による例を図 4.2 に示す。この計算は、解析木を深さ優先で探索し、帰りがけにプログラム片を実行することに相当している。

4.3　最左導出、最右導出

文法 G の開始記号 S から導出される文形式には、非終端記号が複数個含まれることがある。このうち、どの非終端記号から置き換えを行っていくかは任意である。どこから置き換えを行っても、最終的に得られる文は等しく、対応する解析木も等しい。

例えば、文法 $E \to E + E \mid E * E \mid (E) \mid -E \mid \mathbf{id}$ において、文 $-(\mathbf{id} + \mathbf{id})$ を導出す

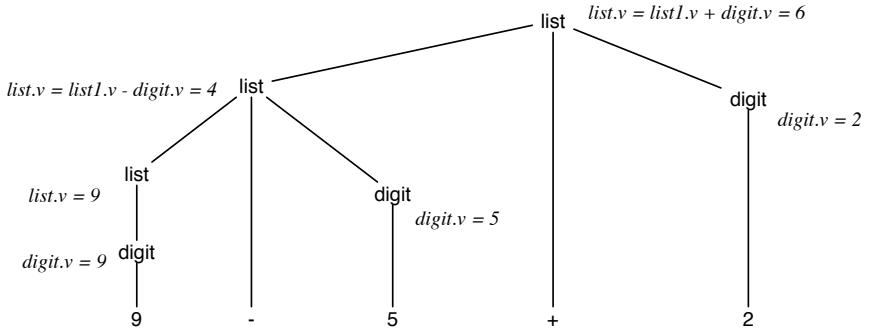


図 4.2　解析木による式の値の計算

る方法は次のようなものがある。

$$E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(E+E) \Rightarrow -(\mathbf{id}+E) \Rightarrow -(\mathbf{id}+\mathbf{id})$$

$$E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(E+E) \Rightarrow -(E+\mathbf{id}) \Rightarrow -(\mathbf{id}+\mathbf{id})$$

この例では 2 通りだけだが、一般にはもっと多くの導出があり得る。これらの導出の中で、“標準形” と言えるものを定めておくことにしよう。必ず文形式の一番左の非終端記号を置き換えるような導出を**最左導出**（leftmost derivation）、一番右の非終端記号を置き換えるような導出を**最右導出**（rightmost derivation）という。上の例では、1 番目のものが最左導出、2 番目のものが最右導出である。

構文解析では、最左導出が非常に重要な役割を果たす[*1]。

4.4　曖昧な文法

1 つの文に対して 2 つ以上の解析木が作れる文法を**曖昧である**（ambiguous）という。構文解析では 2 つ以上の解析木ができてしまうのは困るので、曖昧でない文法に変換したり、ある種の規則を設けて複数の解析木から 1 つを選ぶようにしたりする。

曖昧さを解消する方法は、あまり体系的にはなっていない。問題 4.1 の文法は曖昧であるが、(4.1) の文法に変形すれば曖昧さは解消できる。

もう 1 つ例を挙げよう。

$$\begin{aligned} stmt \rightarrow\ & \mathbf{if}\ (expr)\ stmt \\ & \mid \mathbf{if}\ (expr)\ stmt\ \mathbf{else}\ stmt \\ & \mid S_1 \mid S_2 \mid S_3 \\ expr \rightarrow\ & E_1 \mid E_2 \end{aligned}$$

上に示したのは if 文を表す文法である。しかし、図 4.3 で分かるように、この文法は次の文に対して 2 つの解析木を生成するので、曖昧である。

$$\mathbf{if}\ (E_1)\ S_1\ \mathbf{else\ if}\ (E_2)\ S_2\ \mathbf{else}\ S_3$$

[*1] 本講義で触れなかった上向き構文解析という手法では、最右導出が重要な役割を果たす。

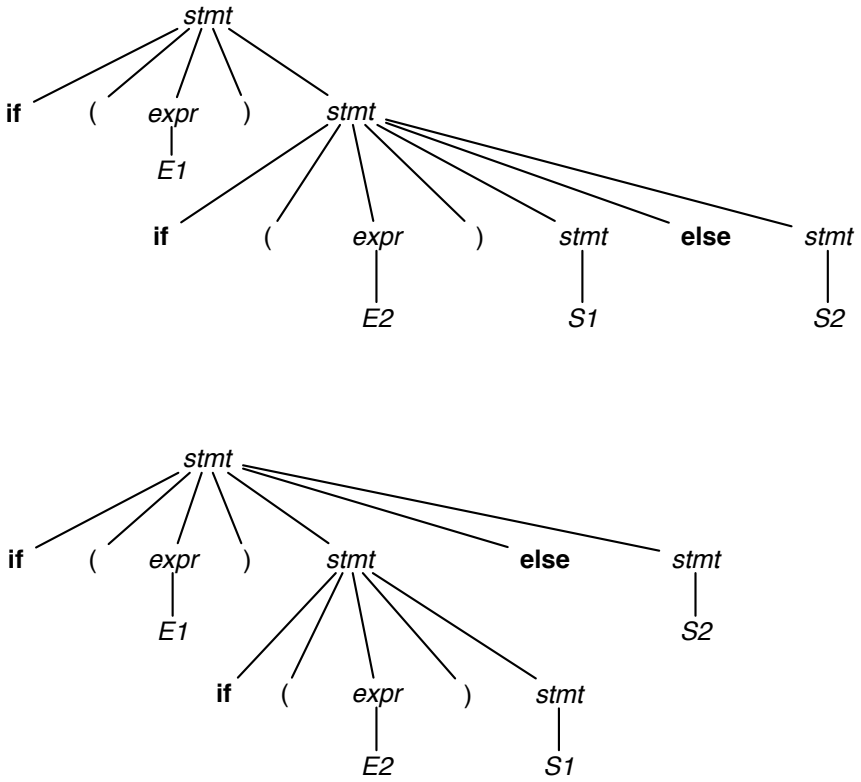


図 4.3　曖昧な文法に対する 2 つの解析木

このような条件文の文法では普通、「各 else は前のほうにある未対応の then 文のうち、もっとも近いものと対応する」という規則を設け、曖昧性を解消する。つまり図 4.3 の解析木のうち、上のほうを採用する。次のように文法を書き換えることで、曖昧さが解消できる。

$$
\begin{aligned}
\mathit{stmt} &\to \mathit{matched\_stmt} \\
&\mid \mathit{unmatched\_stmt} \\
\mathit{matched\_stmt} &\to \textbf{if}\ (\mathit{expr})\ \mathit{matched\_stmt}\ \textbf{else}\ \mathit{matched\_stmt} \\
&\mid S_1 \mid S_2 \mid S_3 \\
\mathit{unmatched\_stmt} &\to \textbf{if}\ (\mathit{expr})\ \mathit{stmt} \\
&\mid \textbf{if}\ (\mathit{expr})\ \mathit{matched\_stmt}\ \textbf{else}\ \mathit{unmatched\_stmt} \\
\mathit{expr} &\to E_1 \mid E_2
\end{aligned}
$$

4.5　左再帰

$A \stackrel{*}{\Rightarrow} A\alpha$ という導出が存在するとき、この文法は**左再帰**であるという。後述するが、左再帰の文法は構文解析では扱いがやっかいである。特に本講義で述べる手法では、構文解析が停止しない状況が起こってしまう。

幸い、左再帰文法は、それと等価（同じ言語を生成する）で左再帰を含まない文法に必

ず変形することができる。ここでは、$A \to A\alpha$ という形の生成規則を持つ左再帰文法を、左再帰ではない文法に変換する方法のみ示す。左再帰の文法の A-生成規則は、一般に

$$A \to A\alpha_1 \mid A\alpha_2 \mid \cdots \mid A\alpha_m \mid \beta_1 \mid \beta_2 \mid \cdots \mid \beta_n$$

と書ける。ここで β_i は A 以外の記号で始まる記号列である。このとき、この A-生成規則を次のように変形する。

$$\begin{aligned} A &\to \beta_1 A' \mid \beta_2 A' \mid \cdots \mid \beta_n A' \\ A' &\to \alpha_1 A' \mid \alpha_2 A' \mid \cdots \mid \alpha_m A' \mid \epsilon \end{aligned}$$

この文法は、変形前のものと同じ言語を生成し、かつ左再帰を含まない。

4.6 文脈自由文法の限界

言語 $L = \{wcw \mid w \in (a \mid b)^*\}$ は「c の前後に同じ文字列 w が出現するような言語」であり、プログラミング言語での「変数を使うときには、それより前に宣言をしておかなければならない」という決まりを抽象化したものと言える。ところが、実は L は文脈自由文法では表現できない。つまり、変数を使う前に宣言されているか検査するのは、構文解析部では行うことができない。実際、この検査は、意味解析部で行われている。

このように、プログラミング言語の制限には構文解析部で検査できないものがあり、それらは意味解析部で検査されることが多い。

練習問題

問題 4.1 次の文脈自由文法について、トークン列 8 - 2 + 6 の解析木を示せ。可能なすべての場合を示し、自然な式の意味になるものはどれか示せ。

$$string \to string + string \mid string - string \mid 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$$

問題 4.2 文脈自由文法 $S \to 0S1 \mid 01$ はどのような言語を表すか述べよ。

問題 4.3 次の文法を考える。

$$\begin{aligned} S &\to A1B \\ A &\to 0A \mid \epsilon \\ B &\to 0B \mid 1B \mid \epsilon \end{aligned}$$

この文法に対し、文字列 00101 の最左導出、最右導出を示せ。

問題 4.4 算術式に対する文法

$$\begin{aligned} E &\to E + T \mid T \\ T &\to T * F \mid F \\ F &\to (E) \mid \mathbf{id} \end{aligned}$$

を、左再帰でないように変形せよ。

第 5 章

構文解析

5.1 構文解析の手法

構文解析部は、字句解析部から出力されたトークン列を読み込み、原始プログラミング言語の文法に適合しているか解析し、適合している場合はトークンを意味のまとまりを反映した入れ子構造にグループ化したものを出力する。

ところで、原始プログラム言語の文法が（トークンを終端記号とするような）文脈自由文法で記述されている、としよう。すると、構文解析部の出力は、トークン列を結果とするような解析木に他ならない。つまり構文解析とは、文脈自由文法をもとにトークン列を導出し、解析木を構成する作業そのものである。ただし、考えておかなければならないことがいくつかある。

1 つ目は、構文解析の計算量（計算時間）である。どんな文脈自由文法でも、それに基づいた構文解析ルーチンは作れるのだが、一般には、長さ n のトークン列の構文解析に $O(n^3)$ の時間が必要となってしまうのである。対象とするトークン列が原始プログラムであり、サイズが大きくなりうることを考えると、これでは計算時間がかかりすぎると言える。そこで、コンパイラの構文解析では、文法に制限を加え、$O(n)$ 程度の時間で構文解析を可能にするのが普通である。

2 つ目は、構文解析プログラムの設計の方針である。対象とするトークン列の長さが事前には分からず、かつ大きくなりうるため、トークン列をあらかじめ全部読み込んでから解析を行う、というプログラム設計は好ましくない。そこで、この講義では、字句解析部から構文解析部に 1 トークンずつ渡しながら解析を行う、という手法をとる。このため、見かけ上、文脈自由文法での導出とは異なるアルゴリズムになっている。

構文解析の手法には、解析木を作る方向によって大きく 2 種類に分けられる。根から葉に向けて作る手法を**下向き構文解析**（top-down parsing）、葉から根に向けて作る手法を**上向き構文解析**（bottom-up parsing）という。下向き構文解析のほうが手法としては易しく、手作業で構文解析ルーチンを作れるが、文法に対する制限は大きい。ただし、それでもほとんどのプログラミング言語の構文解析には十分である。そこで、この講義では、下向き構文解析に絞って解説する。

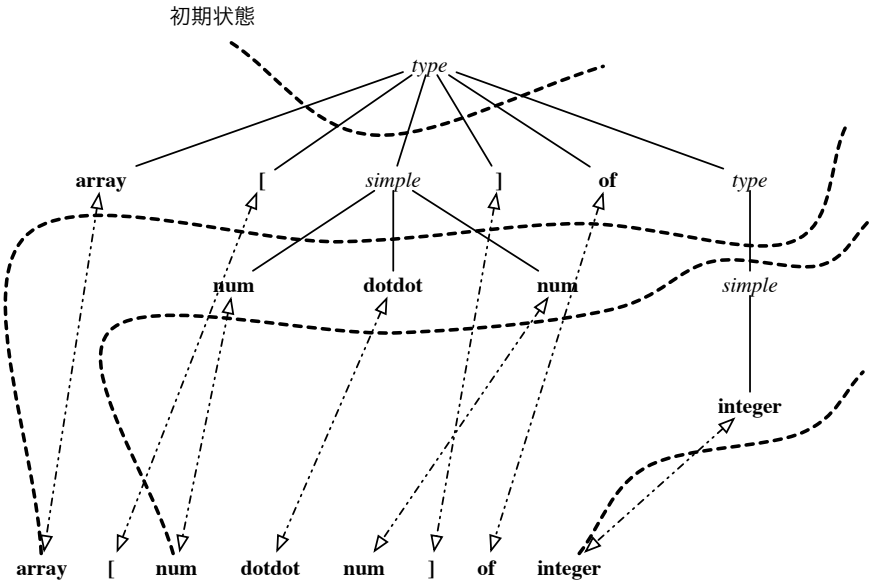


図 5.1　解析木の構成過程

5.2　予測型構文解析

次の文脈自由文法を例にとり、下向き構文解析を説明しよう。この文法は Pascal というプログラミング言語の型宣言の一部である。

$$\begin{aligned} type &\rightarrow simple \mid \hat{}\ id \mid \mathsf{array}\ [\ simple\ ]\ \mathsf{of}\ type \\ simple &\rightarrow \mathsf{integer} \mid \mathsf{char} \mid \mathsf{num\ dotdot\ num} \end{aligned} \tag{5.1}$$

この文法でトークン列 array [num dotdot num] of integer の解析木を作る過程を図 5.1 に示す。

最初、節点は開始記号 *type* だけであり、制御はこの節点にある。字句解析部から先頭のトークン array が渡されると、*type* を左辺に持つ生成規則のうち、array を先頭とする記号列を生成する可能性のある規則を探す。この場合は $type \rightarrow \mathsf{array}[\ simple\ ]\mathsf{of}\ type$ のみが該当する。そこで、この規則の右辺の各記号を、順に *type* の子節点とし、制御を子節点の先頭、すなわち array とラベル付けされた節点に移す。ラベル array は終端記号なので、現在のトークン array と照合する。照合に成功すれば、次のトークンを読み込み、また、現在の解析木で深さ優先順に次の節点、つまり [とラベル付けされた節点に制御を移して、同様の処理を続ける。

この処理から分かるように、上の文法では、トークン列を左から右に 1 回走査するだけで下向きに解析木を構成することができる。このような（下向き）構文解析手法を特に**予測型構文解析**（predictive parsing）という。「予測」という言葉は、次に作ろうとしている節が非終端記号の場合、現在のトークン 1 つだけで、どの生成規則を適用して導出を進めればよいか、予測できるという事実に由来している。

後述するように、予測型構文解析ルーチンは非常に簡単に実装できる。しかし、どんな文法でも予測型構文解析が行えるわけではない。かなり制限された形の文法でしか、この

解析手法は使えない。それでも、実用上は、予測型構文解析が行える程度の文法で充分な場合がかなり多い。

文法 G について予測型構文解析が可能であるためには、G が次のような制限を満たしていなければならない。

1. G は曖昧でなく、かつ左再帰ではない。
2. 生成規則 $A \to \alpha_1 \mid \alpha_2 \mid \cdots \mid \alpha_n$ について、現在の入力記号 a のみからどの生成規則で A を書き換えればよいか、一意に決定できなければならない。

以下で対象とするのは、曖昧でなく、左再帰ではない文法とする。もし構文解析を行いたい文法が曖昧であったり左再帰であったりする場合は、4.4 節や 4.5 節の手法を用いて、曖昧でなく左再帰でない文法に変形しておかなければならない。

5.3 LL(1) 文法

予測型構文解析の可能な文法として、**LL(1) 文法**（LL(1) grammar）という文法を紹介する。多くのプログラミング言語の文法は LL(1) 文法に収まっていることが知られている。

5.3.1 FIRST と FOLLOW

まず FIRST と FOLLOW という 2 つの終端記号集合を導入しておこう。

任意の記号列 α に対し、α から導出される記号列の中で先頭に現れる終端記号の集合を $\mathsf{FIRST}(\alpha)$ という。ただし $S \stackrel{*}{\Rightarrow} \epsilon$ の場合は ϵ も $\mathsf{FIRST}(\alpha)$ に含める。

非終端記号 A に対し、文形式の中で、A のすぐ後ろに現れる可能性のある終端記号の集合を $\mathsf{FOLLOW}(A)$ とする。A が文形式の一番後ろになることがある場合には、記号列の末尾を表す特殊な終端記号 \$ を $\mathsf{FOLLOW}(A)$ に含める。

例 5.1　次の（曖昧でなく左再帰でない）文法を考える。

$$
\begin{aligned}
E &\to TE' \\
E' &\to +TE' \mid \epsilon \\
T &\to FT' \\
T' &\to *FT' \mid \epsilon \\
F &\to (E) \mid \mathbf{id}
\end{aligned}
$$

この文法に対し、FIRST を求めると次のようになる。

$$\begin{aligned}
&\mathsf{FIRST}(E) = \mathsf{FIRST}(T) = \mathsf{FIRST}(F) = \{(, \mathbf{id}\} \\
&\mathsf{FIRST}(E') = \{+, \epsilon\} \\
&\mathsf{FIRST}(T') = \{*, \epsilon\} \\
&\mathsf{FIRST}(TE') = \mathsf{FIRST}(T) = \{(, \mathbf{id}\} \\
&\mathsf{FIRST}(\epsilon) = \{\epsilon\} \\
&\mathsf{FIRST}(+TE') = \{+\} \\
&\mathsf{FIRST}(FT') = \mathsf{FIRST}(F) = \{(, \mathbf{id}\} \\
&\mathsf{FIRST}(*FT') = \{*\} \\
&\mathsf{FIRST}((E)) = \{(\} \\
&\mathsf{FIRST}(\mathbf{id}) = \{\mathbf{id}\}
\end{aligned}$$

また FOLLOW を求めると次のようになる。

$$\begin{aligned}
&\mathsf{FOLLOW}(E) = \mathsf{FOLLOW}(E') = \{), \$\} \\
&\mathsf{FOLLOW}(T) = \mathsf{FOLLOW}(T') = \{+,), \$\} \\
&\mathsf{FOLLOW}(F) = \{+, *,), \$\}
\end{aligned}$$

□

5.3.2　FIRST の計算方法

1. 終端記号 a について $\mathsf{FIRST}(a) = \{a\}$
2. ϵ について $\mathsf{FIRST}(\epsilon) = \{\epsilon\}$
3. 生成規則 $X \to a\alpha$ について、a を $\mathsf{FIRST}(X)$ に加える。
4. 生成規則 $X \to \epsilon$ について、ϵ を $\mathsf{FIRST}(X)$ に加える。
5. 生成規則 $X \to Y_1 Y_2 \cdots Y_n$ について
   - Y_1 が ϵ を導出しないならば $\mathsf{FIRST}(Y_1)$ を $\mathsf{FIRST}(X)$ に加える
   - $Y_1 \overset{*}{\Rightarrow} \epsilon$ であれば、$\mathsf{FIRST}(Y_1) - \{\epsilon\}$ を $\mathsf{FIRST}(X)$ に加える。さらに Y_2 が ϵ を導出しないならば、$\mathsf{FIRST}(Y_2)$ を $\mathsf{FIRST}(X)$ に加え、$Y_2 \overset{*}{\Rightarrow} \epsilon$ ならば $\mathsf{FIRST}(Y_2) - \{\epsilon\}$ を $\mathsf{FIRST}(X)$ に加える。(以下同様)
   - $Y_1 Y_2 \cdots Y_n \overset{*}{\Rightarrow} \epsilon$ ならば、ϵ を $\mathsf{FIRST}(X)$ に加える
6. $\mathsf{FIRST}(X_1 X_2 \cdots X_n)$ は 5 と同様に計算

規則 5 がやや分かりにくいかもしれない。例えば、生成規則 $X \to Y_1 Y_2$ について、ある終端記号 x が $\mathsf{FIRST}(X)$ に含まれるのは、次のいずれかの場合に限られる。

1. $x \in \mathsf{FIRST}(Y_1)$
2. $Y_1 \overset{*}{\Rightarrow} \epsilon$ かつ $x \in \mathsf{FIRST}(Y_2)$

つまり Y_1 から終端記号が何も生成されない可能性があるならば、$\mathsf{FIRST}(Y_2)$ の要素も $\mathsf{FIRST}(X)$ に含まれる、ということである。規則 5 はこれを一般的にしたものである。

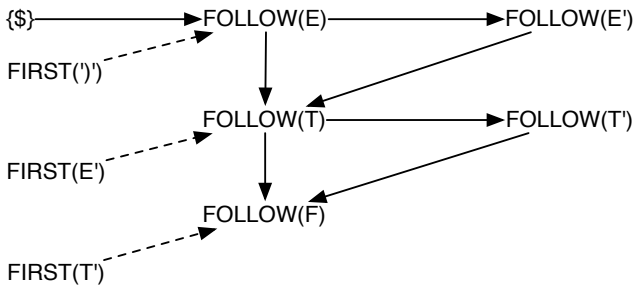


図 5.2 FOLLOW の値の伝搬を表すグラフ

5.3.3 FOLLOW の計算方法

1. FOLLOW(S) に \$ を加える。
2. どの FOLLOW にも記号が追加されなくなるまで、以下を繰り返す。
   - $A \to \alpha B \beta$ について、FIRST(β) $- \{\epsilon\}$ を FOLLOW(B) に加える。
   - $A \to \alpha B \beta$、かつ FIRST(β) が ϵ を含んでいるとき、FOLLOW(A) を FOLLOW(B) に加える。
   - $A \to \alpha B$ のとき、FOLLOW(A) を FOLLOW(B) に加える。

やや分かりにくいアルゴリズムなので、少し整理してみよう。FOLLOW() の値が変化するのは、次の 3 通りの場合しかない。

- 開始記号 S について、無条件に \$ が加えられる。
- $A \to \alpha B \beta$ の形の規則について、FIRST(β) が FOLLOW(B) に加えられる。(β から導出される記号列の先頭の文字は B の直後に現れるから)
- $A \to \alpha B$、もしくは $A \to \alpha B \beta$ で β から ϵ が導出される (β が消えてしまう) 場合について、FOLLOW(A) が FOLLOW(B) に加えられる。(A の直後に現れる文字は B の直後にも現れるから)

そこで、生成規則を見ながら、値が伝搬していく様子を有向グラフにしてみる。例えば $E \to TE'$ から FIRST(E') $- \{\epsilon\}$ が FOLLOW(T) に伝搬することが分かるので、FIRST(E') から FOLLOW(T) へ有向枝を加える。このとき、FIRST(E') から $\{\epsilon\}$ を引かなければならないので、枝を破線にしておく。また $E' \to +TE' \mid \epsilon$ より $E' \Rightarrow \epsilon$ であるので、FOLLOW(E') が FOLLOW(T) に伝搬することが分かる。そこで、FOLLOW(E') から FOLLOW(T) へ有向枝を加える。

このようにすべての値の伝搬に対して有向枝を加えると、図 5.2 のようなグラフが得られる。このグラフを用いて FOLLOW を計算することができる。つまり、有向枝 $X \to Y$ がある場合は X の要素をすべて Y に加える。また $X \cdots > Y$ がある場合は X の要素のうち ϵ 以外をすべて Y に加える。図 5.2 を基に FOLLOW を計算すると、例 5.1 に示した通りになることを確かめよ。

5.3.4　LL(1) 文法

DIRECTOR

上で述べた FIRST と FOLLOW を用いて、DIRECTOR という終端記号集合を定義する。生成規則 $A \to \alpha$ に対して DIRECTOR(A, α) は次のような集合である。

$$\mathsf{DIRECTOR}(A,\alpha) = \begin{cases} \mathsf{FIRST}(\alpha) & \alpha \overset{*}{\Rightarrow} \epsilon \text{ でないとき} \\ \mathsf{FIRST}(\alpha) \cup \mathsf{FOLLOW}(A) & \alpha \overset{*}{\Rightarrow} \epsilon \text{ のとき} \end{cases}$$

直観的には、DIRECTOR(A, α) とは、生成規則 $A \to \alpha$ を適用したときに入力トークン列の先頭に現れる可能性のある終端記号の集合である。普通は FIRST(α) に一致するのだが、A から ϵ が導出される可能性のある場合に限り、A の次に出てくる可能性のある終端記号、すなわち FOLLOW(A) が加えられる。

例 5.2　例 5.1 の文法に対して

$$\begin{aligned}
&\mathsf{DIRECTOR}(E, TE') = \mathsf{FIRST}(T) = \{(, \mathbf{id}\} \\
&\mathsf{DIRECTOR}(E', +TE') = \mathsf{FIRST}(+TE) = \{+\} \\
&\mathsf{DIRECTOR}(E', \epsilon) = \mathsf{FOLLOW}(E') = \{), \$\} \\
&\mathsf{DIRECTOR}(T, FT') = \mathsf{FIRST}(F) = \{(, \mathbf{id}\} \\
&\mathsf{DIRECTOR}(T', *FT') = \mathsf{FIRST}(*FT') = \{*\} \\
&\mathsf{DIRECTOR}(T', \epsilon) = \mathsf{FOLLOW}(T') = \{+,), \$\} \\
&\mathsf{DIRECTOR}(F, (E)) = \mathsf{FIRST}((E)) = \{(\} \\
&\mathsf{DIRECTOR}(F, \mathbf{id}) = \mathsf{FIRST}(\mathbf{id}) = \{\mathbf{id}\}
\end{aligned}$$

□

LL(1) **文法**

文法 G の生成規則のうち、$A \to \alpha \mid \beta$ の形のものについて常に

$$\mathsf{DIRECTOR}(A,\alpha) \cap \mathsf{DIRECTOR}(A,\beta) = \emptyset \tag{5.2}$$

であるとき、G を **LL(1) 文法**という。与えられた文脈自由文法が LL(1) 文法であれば、予測型構文解析プログラムが必ず作れる。

例 5.3　例 5.1 の文法では

$$\begin{aligned}
&\mathsf{DIRECTOR}(E', +TE') \cap \mathsf{DIRECTOR}(E', \epsilon) = \emptyset \\
&\mathsf{DIRECTOR}(T', *FT') \cap \mathsf{DIRECTOR}(T', \epsilon) = \emptyset \\
&\mathsf{DIRECTOR}(F, (E)) \cap \mathsf{DIRECTOR}(F, \mathbf{id}) = \emptyset
\end{aligned}$$

したがって、この文法は LL(1) 文法である。□

5.4　予測型構文解析ルーチンの設計

予測型構文解析は、解析木を構成しつつ深さ優先にたどることを思い出そう。したがって、予測型構文解析のプログラムは、各非終端記号に対して 1 つの関数を用意し、生成規

則の右辺にならって対応する関数呼び出しを行えばよい。同じ左辺 A を持つ生成規則が複数ある場合は、DIRECTOR に基づいて場合分けを行う。

文法 (5.1) に対する構文解析ルーチンの実装例を以下に示す。lookahead は現在走査中のトークンを保持する変数、nexttoken() は次のトークンを取得する関数である。また、各トークンは token 型の値であるとする[*1]。

各関数は生成規則と正確に対応しており、体系的に実装することが可能である。

```
int lookahead; /* 現在のトークン */
int nexttoken(); /* 次のトークンを返す */

/* 節点 n にトークン t を子として追加 */
void addChild(node n, int t);
/* 節点 n の最初の子を返す */
node firstChild(node n);
/* 節点 child の次の弟を返す */
node nextSibling(node n);

void match(int t)
{
  if (lookahead == t) {
    lookahead = nexttoken();
  } else {
    /* 構文エラー */
  }
}

void type(node n)
{
  node child;
  if (lookahead == INTEGER || lookahead == CHAR
      || lookahead == NUM) {
    addChild(n, simple);
    child = firstChild(n);
    simple(child);
  } else if (lookahead == HAT) {
    addChild(n, HAT);
    addChild(n, ID);
    child = firstChild(n);
    match(HAT, child);
    child = nextSibling(child);
    match(ID, child);
  } else if (lookahead == ARRAY) {
    addChild(n, ARRAY);
    addChild(n, LPAREN);
    addChild(n, simple);
    addChild(n, RPAREN);
    addChild(n, OF);
    addChild(n, type);

    child = firstChild(n);
    match(ARRAY, child);
```

[*1] 実際に C 言語で構文解析部を設計する場合は、各トークンを整数で表し、#define で別名を付けることが多い。

```
    child = nextSibling(child);
    match(LPAREN);
    child = nextSibling(child);
    simple(child);
    child = nextSibling(child);
    match(RPAREN);
    child = nextSibling(child);
    match(OF);
    child = nextSibling(child);
    type(child);
  } else {
    /* 構文エラー */
  }
}

void simple(node n)
{
  node child;
  if (lookahead == INTEGER) {
    addChild(n, INTEGER);
    child = firstChild(n);
    match(INTEGER, child);
  } else if (lookahead == CHAR) {
    addChild(n, CHAR);
    child = firstChild(n);
    match(CHAR, child);
  } else if (lookahead == NUM) {
    addChild(n, NUM);
    addChild(n, DOTDOT);
    addChild(n, NUM);
    child = firstChild(n);
    match(NUM);
    child = nextSibling(child);
    match(DOTDOT);
    child = nextSibling(child);
    match(NUM);
  } else {
    /* 構文エラー */
  }
}
```

5.5　LL(1) 文法と予測型構文解析の関連

文法 G が予測型構文解析可能である条件を思い出そう。

1. G は曖昧でなく、かつ左再帰ではない。
2. 生成規則 $A \to \alpha_1 \mid \alpha_2 \mid \cdots \mid \alpha_n$ について、現在の入力記号 a のみからどの生成規則で A を書き換えればよいか、一意に決定できなければならない。

これらが満たされないとき、なぜ予測型構文解析ができないのか考えてみる。

5.5.1 曖昧さ

曖昧な文法に対して予測型構文解析がうまくいかないのは明らかであろう。「曖昧である」ということは、あるトークン列を考えると、解析木の可能性が 2 つ以上あるということである。どちらが選択されるかは運次第である[*2]。つまり、同じプログラムなのに、コンパイルするたびに動作が変わってしまうということである。これでは使い物にならない。これは 4.4 節で述べたように文法を変形したり、規則間に優先度を設けることで解決できることがある。

例 5.4 C 言語の if 文を一般化した次の文法を考える。

$$S \to iES \mid iESeS \mid a$$
$$E \to b$$

これに対し左端の括り出しを行うと、次のような文法が得られる。

$$S \to iESS' \mid a$$
$$S' \to eS \mid \epsilon$$
$$E \to b$$

これでもまだ LL(1) 文法ではない。実はこの文法は、どう変形しても LL(1) にはならないことがわかっている。しかし、$S' \to eS$ を $S' \to \epsilon$ より優先して適用することにすれば、予測型構文解析が行える。S' に対応する関数は次のようになる。

```
void S_dash() { match(e); S(); }
```

□

上で示した優先度は、4.4 節で述べた規則「各 else は前のほうにある未対応の then 文のうち、もっとも近いものと対応する」と同じ働きをしている。

5.5.2 左再帰

左再帰である文法で予測型構文解析ルーチンを書くと、再帰呼び出しが無限に発生してしまう。例えば次の生成規則を考える。

$$expr \to expr \ + \ term$$
$$term \to 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$$

これに対応する予測型構文解析ルーチンを書くと、関数 expr() は次のような再帰関数になる（節点の生成等のコードは省いている。また型も無視している）。

```
expr() { expr(); match(PLUS); term(); }
```

これは再帰関数であるが、基底が存在せず、しかも関数 expr() の内部で直ちに再帰呼び出しが起こっている。これでは明らかに停止しない。

[*2] 情報数学の用語で言うなら**非決定的**（nondeterministic）である。

5.5.3　DIRECTOR の条件

条件 2 は、LL(1) 文法であるための条件 (5.2) に他ならない。この条件を満たさない文法に対して予測型構文解析を行うと、解析に失敗したときに記号列をいくらか戻し、処理をやり直さなければならないことがある。後戻りは一般に処理コストが高く、なるべく避けるのが望ましい。

例 5.5　文法 $S \to aBd, B \to b \mid bc$ は明らかに LL(1) 文法ではない（DIRECTOR(B, b) と DIRECTOR(B, bc) を求めよ）。この文法に対する予測型構文解析ルーチンを作ると、次のようになる。

```
void S() { match(a); B(); match(d); }
void B() { match(b); または { match(b); match(c); } }
```

「または」と書いた部分は「match(b) が失敗したら match(b); match(c);」という意味である[*3]。

この文法により記号列 $abcd$ を構文解析する過程は次のようになる。

1. S() を呼び出す。
2. S() の中から B() を呼び出す。
3. B() の中から match(b) を呼び出す。この結果、文 abd が得られるが、これは $abcd$ と一致しないため、構文解析は失敗であり、入力記号 b を読む前の状態に後戻りし、B() に戻る。
4. B() の中から match(b); match(c); を呼び出す。この結果、文 $abcd$ が得られ、構文解析は成功する。

この文法では、左端の括り出しを行うことで後戻りが回避できる。つまり文法を $S \to aBd, B \to b(c \mid \epsilon)$ と書き直す。すると構文解析プログラムは次のようになり、後戻りが発生しない。

```
void S() { match(a); B(); match(d); }
void B() { match(b); { if (lookahead == 'c') match(c); } }
```

後戻りの起こりうる文法は、左端の括り出し以外に、LR(1) 構文解析などの上向き構文解析法によっても構文解析できることがある。本講義では詳細は省略する。

[*3] 通常はこのような後戻りを含むプログラムは作らないため、分かりやすさを重視して、あえて日本語で示した。

練習問題

問題 5.1 次の文脈自由文法中の各非終端記号について FIRST と FOLLOW を計算せよ。

$$
\begin{aligned}
&S \to D \\
&A \to aA \mid \epsilon \\
&B \to bAC \mid AcD \\
&C \to bC \mid Ac \\
&D \to BA \mid d
\end{aligned}
$$

問題 5.2 問題 5.1 の文法は LL(1) 文法か。

問題 5.3 次の文法が LL(1) 文法であることを示せ。

$$
\begin{aligned}
&S \to AaAb \mid BbBa \\
&A \to \epsilon \\
&B \to \epsilon
\end{aligned}
$$

第 6 章

意味解析

プログラミング言語の構文上の制約には、文脈自由文法の枠から外れるものがいくつかある。ここまでにもところどころで述べてきたが、整理してみよう。

- 変数の宣言は、その変数を使用するより先に出現しなくてはならない。4.6 節でも述べた通り、これは文脈自由文法では表現できない。もちろん LL(1) 文法でも表現できない。
- 型の整合性。例えば C 言語の int x = 10.0 + 2; という文は、int x = (int)(10.0 + (float)2);、つまり、(1) 右辺の 2 が float 型に変換され (2) 10.0 + 2.0 を計算し（結果は float 型）(3) int 型に変換されて変数 x に代入される、というように計算される。これらの型変換は、C 言語の文法では表現されておらず、字句解析部や構文解析部では扱えない。

このように、文脈自由文法から外れる制約の検査や解析は、構文解析部の後で行われる。これを**意味解析**（semantic analysis）という。意味解析は、構文解析の出力である解析木を入力とし、その解析木を変形したり、別のデータ構造を作成したりする。本章では、意味解析の基礎となる技術である翻訳スキームについて述べ、いくつかの制約の検査について実例により説明する。

6.1 記号表

原始プログラム中でなんらかの名前がついている構成要素（変数名、関数名、構造体、構造体のメンバなど）を特徴づける情報には次のようなものがある。

- 構成要素の種類（変数、関数など）
- 構成要素の名前
- 型情報（変数の型、関数の引数の個数、個々の引数の型、返り値の型など）
- 属性（static など）
- 構成要素が割り当てられている番地

これらの情報はコンパイラの各段階で頻繁に参照されるので、何らかの形のデータ構造で表現し、コンパイラ中で保持・管理しなければならない。コンパイラでは通常、これらの情報を**記号表**（symbol table）という表で管理する。

6.1.1　記号表の実装

記号表を実装する最も簡単な方法は、構造体の配列を用いるものである。簡単のため、記号表に格納される情報を、識別子の文字列（lexname、最大長さ MAXLEN)、識別子のトークンの種類（lextoken）の 2 つであるとし、記号表に登録できるエントリの最大数を MAXENT としておこう。すると、記号表を表すデータ構造は次のようになる。

```
typedef struct table_ent {
  char lexname[MAXLEN];
  int lextoken;
};
table_ent lextable[MAXENT];
```

記号表に対する主な操作は次の 2 つである。

- int insert(char *s, int t)…文字列 s, トークン t のエントリを新たに記号表に登録し、そのエントリの番号（添字）を返す。すでに登録されていれば（登録できないとして）-1 を返す。
- int lookup(char *s)…文字列 s のエントリが記号表に登録されているか調べる。あればそのエントリの番号（添字）を返し、なければ-1 を返す。

上記の構造体の配列による記号表に対し、これらの操作を C 言語で実装することは容易であろう。各自で試みられたい。

実際のコンパイラでは、上で示した記号表の実装は次のような問題がある。

1. 登録できる文字列の長さに制限がある。つまり、プログラムで使える識別子の長さに制限がある。
2. 記号表のエントリ数に制限がある。つまり、プログラムで使える識別子の数に制限がある。
3. 記号表を検索したり、挿入しようとするエントリがあるかどうか調べるのに、最悪、記号表のエントリをすべて調べなければならない。つまり、insert や lookup の処理の効率が悪い。

1 は、lexname を char の固定長配列ではなく、malloc() などにより動的確保するようにすればよい。また 2 は、配列ではなく連結リストなどを用いれば解決可能である。しかし、3 についてはこのような工夫では解決できず、**ハッシュ**法などの手法が必要になる[*1]。

6.1.2　記号表による意味解析

記号表を用いると、本章の最初で述べた構文上の制約「変数の宣言は、その変数を使用するより先に出現しなくてはならない」は容易に検査できる。すなわち、次のようにすればよい。

1. 変数や関数、構造体などの宣言が出現したときに、名前、型などの情報を記号表に

[*1] ハッシュ法については「アルゴリズム論」で取り上げられているので、説明は省略する。詳細は「アルゴリズム論」の教科書 [5] を参照されたい。

登録する。
2. 予約語以外の識別子がプログラム中で出現したときに、記号表を参照し、その識別子がすでに登録されているか調べる。登録されていなければ、識別子の宣言をせずに使用されているので、誤りである。

ここで注意をしておかなければならないのが、変数や関数などの**有効範囲**（scope)、すなわちその変数や関数を参照することのできるプログラムの範囲である。C 言語の識別子には、有効範囲に関して次のような制限がある。

1. 大域変数（global variable)、関数、構造体、構造体のメンバなどは、それが宣言された場所からファイルの終わりまでが有効範囲である。
2. 関数の引数は、その関数の内部が有効範囲である。
3. 関数内で宣言された変数（局所変数、local variable）は、宣言された場所からその関数の終わりまでが有効範囲である。
4. 同じ識別子が大域変数と局所変数・引数としてともに用いられていた場合、局所変数・引数が優先する。

記号表の管理も、この有効範囲を考慮して行わなければならない。

簡単なのは、1 のタイプの識別子と、2, 3 のタイプの識別子を別の記号表で管理する、という方法である。記号表の管理は次のように修正される。ただし以下では、それぞれの記号表を大域記号表、局所記号表と呼んでいる。

1. 大域変数、構造体、構造体のメンバ、extern 文による（ファイル外部の変数、関数などの）宣言などが出現したときには、その名前、型などの情報を大域記号表に登録する。
2. 関数宣言が出現したときには、関数名とその型などの情報を大域記号表に、引数の名前、型などの情報を局所記号表にそれぞれ登録する。
3. 局所変数の宣言が出現したときには、その変数の名前、型などの情報を局所記号表に登録する。
4. 予約語以外の識別子がプログラム中で出現したときに、局所記号表、大域記号表の順に参照し、その識別子がすでに登録されているか調べる。どちらの記号表にも登録されていなければ、識別子の宣言をせずに使用されているので、誤りである。
5. 関数宣言の終わりに到達すると、局所記号表の内容をすべて消去する。

なお、C 言語では、有効範囲として（中括弧で囲まれた）ブロックも考慮しなければならない。また、goto 文の飛び先を指定するラベルも、上に述べたものとは異なる有効範囲を持っている。このような理由により、実際には、より複雑な記号表管理が必要になる。また、「計算機言語 I」で取り上げた ML や「計算機言語 II」で取り上げる予定の Java など、C より複雑な有効範囲を持つプログラミング言語でも、より複雑な記号表管理が必要である。

6.2 型検査と型変換

次に、本章の最初に挙げた構文上の制約のうち、型の整合性について述べる。

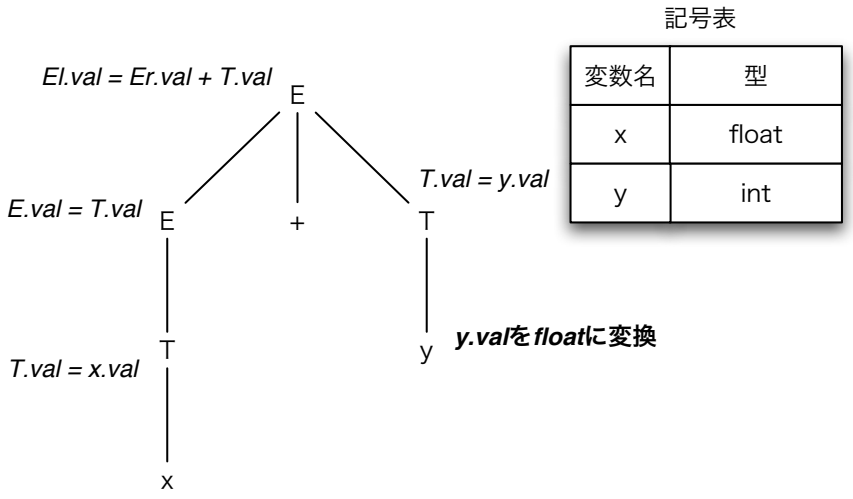


図 6.1　解析木への型変換情報の付加

「計算機言語 I」で取り上げた ML では、演算数に対する型の制限が非常に強い。例えば、算術式 x div y であれば、x, y はいずれも整数型（int）でなければならない。この制約は、4.2 節の例と同様、以下のようなプログラム断片付き生成規則で表現することができる。

$$
\begin{aligned}
list \rightarrow list \text{ div } digit \ \{ & list_l.type = \\
& \text{if } (list_r.type == int \ \&\& \ digit.type == int) \\
& \text{then } int \text{ else 型エラー } \} \\
list \rightarrow digit \ & \{list.type = digit.type; \} \\
digit \rightarrow 0 \ & \{digit.type = int; \} \\
& \cdots \\
digit \rightarrow 9 \ & \{digit.type = int; \}
\end{aligned}
$$

4.2 節の算術式の値の計算と同様、この型検査は、プログラム断片付き解析木を作り、それを深さ優先でたどりながらプログラム断片を実行することで、処理できる。

C 言語の場合も上の例と同様に型検査を行うことができる。ただし、原子型の種類が多く（int, signed int, unsigned int, long, signed long, unsigned long, short, $\cdots$）、かつ型の整合性に関するルールが複雑である。例えば $x + y$ において、x が float 型で y が int 型であっても誤りではない。この場合、y が float 型に変換されてから加算が行われ、結果は float 型になる[*2]。このとき、y に関する型変換はプログラム実行時に行われることに注意しなければならない。つまり、コンパイラは、x と y の加算の機械語命令に加え、y を float に変換する機械語命令を生成しなければならない。そのため、意味解析部では、y を float に変換する必要があるという情報を解析木に追加する（図 6.1）。

[*2] 算術演算における演算数の型変換については [6] の A6.5 節を参照のこと。

6.3 解析木を用いた意味解析

解析木を用いると、さまざまな実用的な処理が体系的に行えるようになる。6.2 節で挙げた型検査や 4.2 節で述べた算術式の値の計算がその例である。本節では、このような解析木を用いた意味解析の基礎となる理論として、翻訳スキームという考え方を紹介する。

まず、コンパイラの動作に近い例として、中置記法で書かれた算術式を後置記法に変換することを考えよう。

6.3.1 中置記法と後置記法

我々が普段目にする 2 項演算子は、たいてい、2 つの演算数の間に演算子を書く。32 + 9, 4 / 2 といった具合である。これを**中置記法**（infix notation）という。しかし、算術式の書き方はこれだけではない。演算子を 2 つの演算数の前に置く**前置記法**（prefix notation）、2 つの演算数の後ろに置く**後置記法**（postfix notation）という書き方もある。ここでは後置記法のみもう少し詳しく説明することにしよう。

定義 6.1 中置記法による式 E の後置記法は次のように再帰的に定義される。

1. E が変数または定数であれば、E の後置記法は E である。
2. 任意の 2 項演算子 op について、$E_1 \,\mathrm{op}\, E_2$ の後置記法は $E_1'\ E_2'\ \mathrm{op}$ である。ここで E_1', E_2' はそれぞれ E_1, E_2 の後置記法である。
3. (E) の後置記法は E' である。ここで E' は E の後置記法である。

□

例 6.1 $32 + 3$ の後置記法は 32 3 + である。$9 + 4 * 3$ の後置記法は 9 4 3 * + である。$(1 + 2) * 4$ の後置記法は 1 2 + 4 * である。□

後置記法で書かれた算術式は、スタック 1 つで計算できるという特徴を持つ。式を左から右に 1 トークンずつ読み、次の動作を行えばよい。

1. トークンが数であれば、その数をスタックに積む（push）。
2. トークンが演算子 op であれば、スタックの上 2 つの要素（x, y とする）をおろし（pop）、$x \,\mathrm{op}\, y$ を計算し、結果をスタックに積む。

これを式の終わりまで行い、最後にスタックに残った数が式の計算結果である。例 6.1 の後置記法で確かめてみよ。

6.3.2 中置記法から後置記法への変換

中置記法で書かれた算術式を後置記法に変換する方法の 1 つに、解析木を用いるやり方がある。分かりやすくするために、算術式のための文法として、次のような（左再帰を含

む）ものを考える。

$$
\begin{aligned}
&E \to E + E \\
&E \to E - E \\
&E \to E * E \\
&E \to E \ / \ E \\
&E \to (E) \\
&E \to 0 \\
&\cdots \\
&E \to 9
\end{aligned}
\tag{6.1}
$$

(6.1) の規則それぞれにプログラム断片を付けてみる。ここで *putchar* は、引数を印字する関数である。

$$
\begin{aligned}
&E \to E + E && \{putchar('+');\} \\
&E \to E - E && \{putchar('-');\} \\
&E \to E * E && \{putchar('*');\} \\
&E \to E \ / \ E && \{putchar('/');\} \\
&E \to (E) && \\
&E \to 0 && \{putchar('0');\} \\
&\cdots && \\
&E \to 9 && \{putchar('9');\}
\end{aligned}
$$

(6.1) の文法に基づいて解析木を作るとき、導出に用いた規則にプログラム断片が付いていれば、その断片を左辺の非終端記号に対応する節点に付けておく。例えば、式 $(3+4)*5$ に対してプログラム断片を付けながら解析木を作ると、図 6.2 のようになる。

得られた解析木を深さ優先でたどり、節点にプログラム断片が付いている場合は帰りがけにその断片を実行する。すると図 6.2 の例では 3 4 + 5 * となり、後置記法の式が正しく得られる。

6.4　翻訳スキーム

図 6.2 の例ではプログラム断片を帰りがけに実行したが、一般には、行きがけや通りがけにプログラム断片を実行したいこともある。そこで、(6.1) のプログラム断片付きの文法を少し拡張しよう。これが**翻訳スキーム**と呼ばれるものである。

定義 6.2　文脈自由文法 G において、G の各生成規則の右辺にプログラム断片を埋め込んで得られる（拡張された）文法を**翻訳スキーム**（translation scheme）という。翻訳スキームの基になる文脈自由文法 G を**基底文法**（underlying grammar）、埋め込まれたプログラム断片を**意味動作**（semantic action）とそれぞれ言う。□

例 6.2　次に示すのは、基底文法を (6.1) とし、中置記法の式を後置記法に変換する翻訳

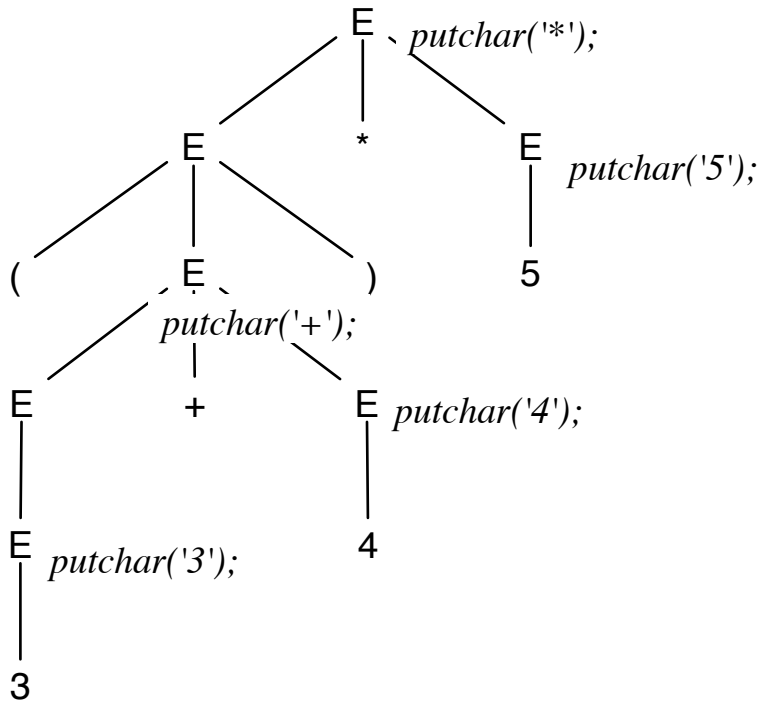


図 6.2 $(3+4)*5$ に対するプログラム断片付き解析木

スキームである。

$$
\begin{aligned}
&E \to E + E \ \{putchar('+');\} \\
&E \to E - E \ \{putchar('-');\} \\
&E \to E * E \ \{putchar('*');\} \\
&E \to E \ / \ E \ \{putchar('/');\} \\
&E \to (E) \\
&E \to 0 \ \{putchar('0');\} \\
&\cdots \\
&E \to 9 \ \{putchar('9');\}
\end{aligned}
\tag{6.2}
$$

□

意味動作を特殊な終端記号とみなすと、翻訳スキームは文脈自由文法と見ることができ、(意味動作を含んだ) 解析木を生成したり、左再帰の除去などの文脈自由文法の変形手法をそのまま適用したりできる。

例 6.3 (6.2) の翻訳スキームを用いて式 $4+3*5$ の (意味動作を含んだ) 解析木を生成すると図 6.3 のようになる。この解析木を深さ優先にたどり、意味動作の節点ではその意味動作を実行すると、式 4 3 5 * + という後置記法の式が得られる。

例 6.4 (6.2) の翻訳スキームに 4.5 の手法を適用して左再帰を除去すると、次のような翻

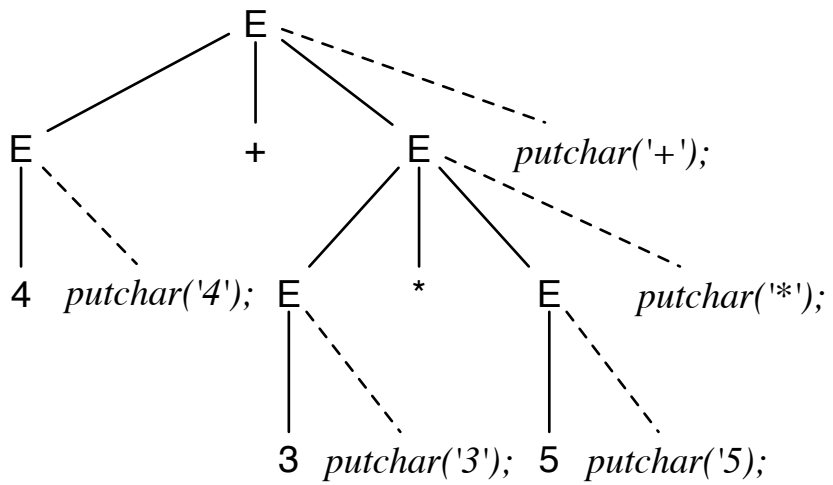


図 6.3　翻訳スキームによる $4+3*5$ の解析木

訳スキームに変形される。

$$
\begin{aligned}
E \rightarrow & (E)E' \\
& \mid 0\ \{putchar('0');\}\ E' \\
& \mid \cdots \\
& \mid 9\ \{putchar('9');\}\ E' \\
E' \rightarrow & +E\ \{putchar('+');\}\ E' \\
& \mid -E\ \{putchar('-');\}\ E' \\
& \mid *E\ \{putchar('*');\}\ E' \\
& \mid /E\ \{putchar('/');\}\ E' \\
& \mid \epsilon
\end{aligned}
$$

これを用いて式 $3*(1+2)$ の解析木を生成すると図 6.4 のようになる。

6.4.1　意味動作の形式

以降の節の準備として、意味動作の形式についてもう少し詳しく述べる。これまで、意味動作はプログラム断片だとしてきたが、どんな形のプログラムでも認められるわけではない。使える変数に関して制限がある。

プログラム断片中で使える変数は $X.v$ という形式に限られる。ここで X は、基底文法に出現する非終端記号または終端記号である。v は翻訳スキーム内で適当に決めて構わない。v を X の**属性**（attribute）という。

6.5　性質のよい翻訳スキーム

前節で述べた翻訳スキームによる計算は、常に効率よく行えるわけではない。計算効率のポイントの一つが「深さ優先に 1 回たどるだけで計算できるか」である。解析木は一般

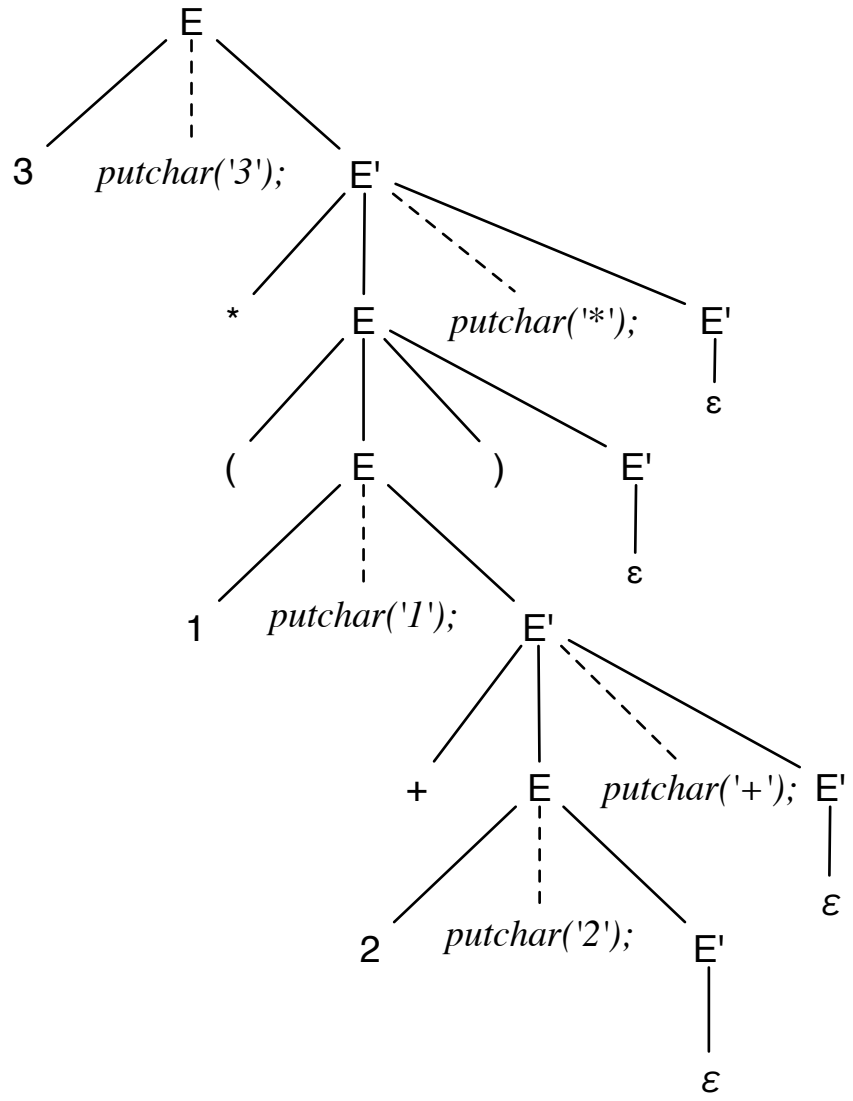


図 6.4 (6.2) による $3 * (1 + 2)$ の解析木

に大きくなりがちなので、深さ優先探索の回数が増えるほど、計算効率が悪くなることが予想されるからである。

そこでこの節では、翻訳スキームの意味動作に制限を加え、深さ優先に 1 回たどるだけで計算が行える翻訳スキーム（S 属性定義、L 属性定義）を導入する。

6.5.1 合成属性と相続属性

もともとの翻訳スキームでは、基底文法に出現する非終端記号や終端記号の属性はすべて意味動作内で用いることができる。しかし、これでは意味動作の右辺値（$v = E$ の形の文の E）を計算するのに、解析木のあちこちを参照しなくてはならず、深さ優先に 1 回たどるだけでは到底計算が終わらない。

そこで、性質のよい属性を 2 つ定義しよう。

定義 6.3　解析木の節点 A について、A の属性 x が $A.x = f(c_1, c_2, \cdots, c_n)$ と計算できるとする。$c_1, c_2, \cdots, c_n$ がすべて A の子節点の属性であるとき、x を A の**合成属性**（synthesized attribute）という。□

定義 6.4　解析木の節点 A について、A の属性 x が $A.x = f(c_1, c_2, \cdots, c_n)$ と計算できるとする。$c_1, c_2, \cdots, c_n$ がすべて A の親節点または兄弟節点の属性であるとき、x を A の**相続属性**（inherited attribute）という。□

生成規則 $A \to \alpha$ に基づいて説明すると、次のようになる。

- A の属性 x が α 中の非終端記号や終端記号の属性のみから計算できるなら、x は合成属性
- α 中の記号 B の属性 y が、A や α 中の他の記号の属性のみから計算できるなら、y は相続属性

例 6.5　次に示す翻訳スキームは合成属性のみからなる。添字の l, r は、それぞれ左辺に出現する記号、右辺に出現する記号を表している。

$$
\begin{array}{ll}
E \to E + T & \{E_l.val = E_r.val + T.val;\} \\
E \to T & \{E.val = T.val;\} \\
T \to T * F & \{T_l.val = T_r.val * F.val;\} \\
T \to F & \{T.val = F.val;\} \\
F \to (E) & \{F.val = E.val;\} \\
F \to digit & \{F.val = digit.lexval;\}
\end{array}
$$

□

例 6.6　次に示す翻訳スキームのうち、非終端記号 L の属性 in は相続属性である。

$$
\begin{array}{ll}
D \to T\ L & \{L.in = T.type;\} \\
T \to int & \{T.type = integer;\} \\
T \to float & \{T.type = float;\} \\
L \to L, id & \{L_r.in = L_l.in; addtype(id.entry, L_l.in);\} \\
L \to id & \{addtype(id.entry, L.in);\}
\end{array}
$$

□

合成属性や相続属性は、解析木の深さ優先探索と相性が良い。合成属性の計算は、帰りがけに行うとたどりを 1 回で済ませることができる。また、相続属性の一部も、1 回のたどりで計算を行うことができる。

6.5.2　S 属性定義

定義 6.5　翻訳スキームのうち、次の条件を満たすものを **S 属性定義**（S-attributed definition）という。

1. 意味動作中に現れる変数はすべて合成属性である。

2. すべての意味動作は、生成規則の右辺の末尾にしか現れない。

□

例 6.7 例 6.5 の翻訳スキームは S 属性定義である。□

定義より、S 属性定義の意味動作は、解析木を 1 回深さ優先にたどるだけですべて計算することができる。

6.5.3 L 属性定義

定義 6.6 翻訳スキームのうち、次の条件を満たすものを **L 属性定義**（L-attributed definition）という。

1. 意味動作中に現れる変数はすべて合成属性か相続属性である。
2. 基底文法の生成規則 $A \rightarrow X_1X_2\cdots X_n$ について、X_j の相続属性は、$X_1, X_2, \cdots, X_{j-1}$ の属性と A の相続属性だけに依存する。
3. 基底文法の生成規則 $A \rightarrow X_1X_2\cdots X_n$ について、X_j の相続属性を左辺に持つ文を含む意味動作は、X_j の直前に出現する。
4. 基底文法の生成規則 $A \rightarrow X_1X_2\cdots X_n$ について、A の合成属性はこの規則の末尾に出現する。

□

例 6.8 次の翻訳スキームは L 属性定義である。ここで、*type* は合成属性、*in* は相続属性である[*3]。また *entry* は識別子を表す終端記号 *id* の属性であり、その識別子の名前を格納しているとする。*entry* の値は意味解析以前にすでに分かっているものとする。関数 $addtype(id.entry, L.in)$ は、$id.entry$ の型として $L.in$ を登録する、という動作をする。

$$
\begin{aligned}
&D \rightarrow T\ \{L.in = T.type;\}\ L\\
&T \rightarrow \mathsf{int}\ \{T.type = integer;\}\\
&T \rightarrow \mathsf{float}\ \{T.type = float;\}\\
&L \rightarrow \{L_r.in = L_l.in;\}\ L\ ,\ \mathsf{id}\ \{addtype(\mathsf{id}.entry, L_l.in);\}\\
&L \rightarrow \mathsf{id}\ \{addtype(\mathsf{id}.entry, L.in);\}
\end{aligned}
$$

□

定義 6.6 の 1 と 4 から分かるように、もし L 属性定義に相続属性が一つも含まれなければ、S 属性定義そのものである。すなわち、S 属性定義は必ず L 属性定義になる。

S 属性定義と同様に、L 属性定義の翻訳スキームも解析木を 1 回深さ優先にたどるだけで、すべての意味動作を計算することができる。

例 6.9 例 6.8 の翻訳スキームに基づいて、トークン列 int id, id, id から生成した意味動作付き解析木を図 6.5 に示す。なお、非終端記号 L、終端記号 id が複数現れるので、便宜上添字を付けて区別している。

[*3] この翻訳スキームは、int i, j; のように int 型の変数を複数同時に宣言するときに、i, j の型を int と定めるものである。

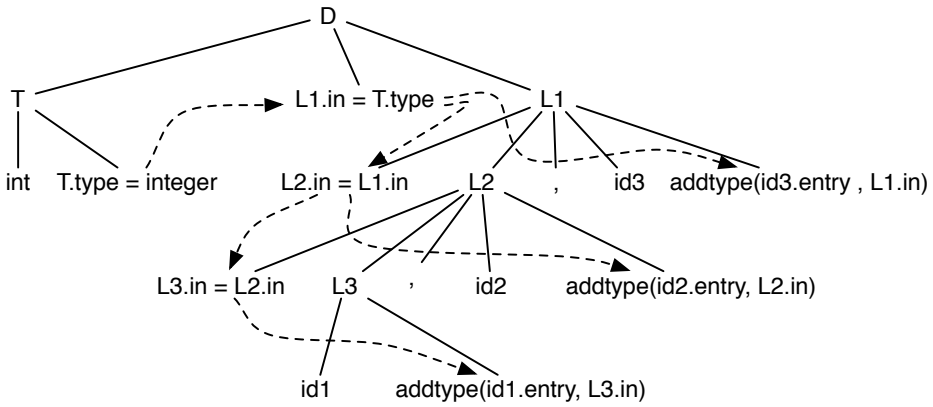


図 6.5　L 属性定義に基づく解析木

図中の破線は、属性の依存関係を示している。例えば、意味規則 $T.type = integer$; で決定した $T.type$ の値を $L_1.in = T.type$; で参照する、といった具合である。この解析木を深さ優先でたどっていくと、破線で示した属性の依存関係の順序を守りながら意味規則が計算されていくことを確かめよ。□

第 7 章

実行時環境

以降の章では、意味解析部までで得られた解析木をもとに機械語命令列を生成する部分（**コード生成部**）について説明を行う。

実際に生成される機械語命令列は、対象とする CPU によって異なる。したがって、コード生成部の詳細は、対象とする CPU によっても異なってくる。ここでは、具体性を持たせるために、Pentium プロセッサを対象とし、また gcc コンパイラで生成されるアセンブリ言語を例として用いる。ただし、説明する手法は特定の CPU に依存しない汎用的なものである。

C や ML, Java のように、我々が通常プログラミングを行う言語を高級言語ということがある。これは、アセンブリ言語や機械語に比べて論理的に高度な概念が扱えることに由来する用語である。例えば、関数、型、構造体などはすべて高級言語にしかない「高度な概念」と言える。

本講義を受講している諸君は、すでに C などの高級言語によるプログラミングは充分習得していることであろう。また、アセンブリ言語や、それを CPU がいかにして実行するかも学習していることであろう[*1]。しかし、この 2 つのプログラム実行の間には、論理的に大きな差がある。例えば、C 言語の関数の呼び出し、実行、返り値の処理などが、一体どういう機械語列になり、どのように実行されているのか、想像がつくだろうか。このギャップを埋め、コード生成の理解のための準備をするのが、本章の目的である。

7.1 予備知識

7.1.1 メモリ領域

我々が今日用いる汎用コンピュータは、プログラムの機械語列やデータをメモリ上に置き、その機械語列を CPU が実行していく、という形式のものがほとんどである。プログラムの機械語列もメモリ上に置かれているため、メモリ上の機械語列を置き換えるだけで、違うプログラムが実行できる[*2]。メモリは通常 1 バイト単位で読み書きが可能であり、各バイトには一意に定まる番号が振られている。この番号のことを**番地**（アドレス、address）という。番地は 0 番から始まり、例えば 32 ビット計算機であれば $2^{32} - 1$ 番地までである。

[*1] 計算機アーキテクチャ A

[*2] このようなコンピュータのことを**フォン・ノイマン型計算機**という。

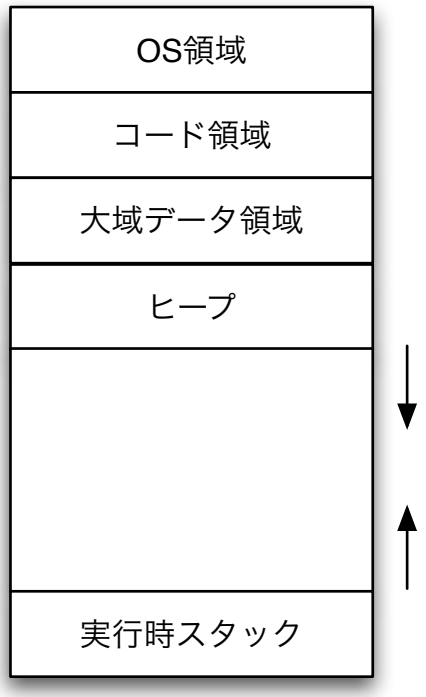


図 7.1　メモリ領域の分割例

メモリはいくつかの連続した領域（**メモリ領域**）に分割して使用される。

- OS 専用の領域。個々の計算機、OS ごとに大きさは固定されている。
- **ユーザ領域**…通常のプログラムを実行するのに使用される領域
  - **コード領域**…実行するプログラムを格納する領域。いったんプログラムがメモリ上にロードされれば、変更されない。すなわち、読み取り専用の領域である。
  - **データ領域**…プログラム実行に必要なデータを格納する領域。プログラムの実行とともに内容が変化する。すなわち読み書き可能な領域である。
    - **大域データ領域**…大域変数を割り当てる領域
    - **実行時スタック**…局所変数や関数への実引数を割り当てる領域
    - **ヒープ**…malloc() などにより実行時に動的に生成されるデータを割り当てる領域

ユーザ領域のうち、コード領域と大域データ領域の大きさはコンパイル時に決定され、プログラム実行時には変化しない。一方、実行時スタックとヒープの大きさは実行時に変化する。このような性質を満たすようなメモリ領域の使い方の一例を図 7.1 に示す。この例では、メモリの下位番地から OS 領域、コード領域、大域データ領域、ヒープを割り当て、実行時スタックは逆にメモリの上位番地から割り当てる。

大域変数が割り当てられる番地はコンパイル時に決定される[*3]。一方局所変数や実引数

[*3] 現代のコンピュータでは、複数のプログラムを同時に実行させることができる。そのため、実際には、コード領域や大域データ領域についても、コード領域の先頭からの相対番地のみ決定しておき、プログラ

は、プログラム実行中に関数が呼び出されたとき、実行時スタック上に割り当てられるので、それまで番地は決定できない。また malloc() などにより動的に確保された領域も、実行時に実際にヒープに割り当てられるまで、番地は決定できない。

7.1.2 CPU とレジスタ

メモリ上に配置された機械語プログラムは CPU（中央演算処理装置）が実行する。具体的には、機械語プログラムを順に 1 命令ずつ読み出し、その命令に書かれた指示に従ってメモリ領域やレジスタから値を読み出して算術演算や比較演算を行い、結果をメモリ領域やレジスタに格納する。これを繰り返す。

Pentium プロセッサには、次のようなレジスタが用意されている。

1. **汎用レジスタ**（general-purpose register）…算術演算や比較演算などの演算数として用いることのできるレジスタ。本講義での機械語命令には、%eax, %ebx, %ecx, %edx などのレジスタの他、実行時スタックを操作するのに用いられる**ベースポインタ**（base pointer）%ebp と**スタックポインタ**（stack pointer）%esp がある。
2. **条件フラグ**（condition flag）…比較命令の結果を格納するレジスタ。ゼロフラグ %zf や符号フラグ %sf などがある。本講義で扱う機械語命令では、比較命令が自動的に設定し、条件付きジャンプ命令が適切なフラグを自動的に参照する。したがって、機械語命令に明示的に現れることはない。
3. **命令ポインタ**（instruction pointer）…**プログラムカウンタ**（program counter）とも言う。CPU が次に実行すべき命令の格納番地を指すレジスタ。%eip という記法を用いる。この値も自動的に更新されるため、命令ポインタが機械語命令に明示的に現れることはない。

7.1.3 機械語命令

以降の説明を理解するのに必要な機械語命令について、簡単にまとめておく。なお、本講義では Pentium プロセッサの機械語を記述するのに、AT&T 形式と呼ばれるアセンブリ言語を用いている。

個々の機械語命令は、次のような書式をしている。[] は、その部分が省略可能であることを表している。

[ラベル:] 命令名 第 1 演算数 [, 第 2 演算数, ···, 第 n 演算数]

ラベルは、コード領域中でこの機械語命令が置かれている番地を表すためのシンボルであり、先頭が _ であるような文字列を用いる。関数に相当する機械語命令列の先頭には、**_関数名**というラベルが必ず付けられる。なお、図 1.1 の 3 行目のように、ラベルだけが 1 行に書かれていても構わない。

演算数として用いることのできるのは、汎用レジスタ、メモリ番地、整数定数のみである[*4]。メモリ番地の指定は、以下の 2 通りの方法がある。

ム（機械語命令列）をメモリ上のどこにも配置できるようにしておくことが多い。このような機械語命令列のことを**再配置可能**（relocatable）であると言う。

[*4] 整数定数以外の定数（文字列、浮動小数点数など）は、コード領域や大域データ領域などに値を格納して

1. その番地に付けられたラベル名
2. 汎用レジスタの値を番地と見なし、その番地からの相対番地を指定する。例えば、(レジスタ R の値) 番地を $(\%R)$、(レジスタ R の値 $+ n$) 番地を $n(\%R)$ というように記述する。括弧がつくことで、レジスタそのものではなく、レジスタの値の番地という意味になることに注意してほしい[*5]。

また整数定数の先頭には$を付ける。

本講義では、実際の Pentium プロセッサの機械語のうちごく一部だけを用いる。

2 項算術命令　演算数を 2 つとる算術演算。addl（加算）, subl（減算）, imull（乗算）などがある。結果は第 2 演算数に格納される。例えば add %ebx,%eax は、レジスタ %ebx の値をレジスタ %eax の値に加え、結果をレジスタ %eax に格納する。

単項算術命令　演算数を 1 つとる算術演算。neg（符号の反転）、dec（1 引く）、inc（1 足す）などがある。結果は第 1 演算数に格納される。例えば inc %eax は、レジスタ %eax の値に 1 を加え、結果を %eax に格納する。

移動命令　演算数を 2 つとり、第 1 演算数の値を第 2 演算数に格納する。本講義では movl のみ扱う。例えば movl %eax,%ebx は、レジスタ %eax の値をレジスタ %ebx に格納する。

比較命令　演算数を 2 つとり、第 1 演算数と第 2 演算数を比較し、結果を条件フラグに設定する。定数は第 1 演算数でのみ使える。本講義では cmpl のみ用いる。例えば cmpl $4,%eax は、定数 4 とレジスタ %eax の値とを比較し、次のように条件フラグを設定する。

$\mathsf{zf} = 0, \mathsf{sf} = 0$	$4 <$ %eax のとき
$\mathsf{zf} = 1, \mathsf{sf} = 0$	$4 =$ %eax のとき
$\mathsf{zf} = 0, \mathsf{sf} = 1$	$4 >$ %eax のとき

無条件ジャンプ命令　jmp 命令。第 1 演算数にラベルをとり、そのラベルの番地にジャンプする。実際には、ラベルの番地を命令ポインタ %eip に格納する、という動作をする。これにより、ラベルの番地の機械語命令が次に実行されることになる。

条件ジャンプ命令　第 1 演算数にラベルをとる。必ず比較命令と対にして使われる。現在の条件フラグの値を参照し、ジャンプ条件と一致すればラベルの番地にジャンプする。一致しなければ次の命令に進む。例えば jge という条件ジャンプ命令は、$\mathsf{sf} = 0$ のとき、すなわち直前の比較命令で 第 1 演算数 $\leq$ 第 2 演算数 であったときにジャンプする。条件ジャンプ命令と、そのジャンプ条件を以下にまとめておく。

おき、その番地をラベルで参照する。

[*5] 間接参照という。

(x, y はそれぞれ第 1 演算数、第 2 演算数を表す)

jg	$\mathsf{zf} = 0 \wedge \mathsf{sf} = 0$	$(x < y)$
jge	$\mathsf{sf} = 0$	$(x \leq y)$
je	$\mathsf{zf} = 1$	$(x = y)$
jne	$\mathsf{zf} = 0$	$(x \neq y)$
jl	$\mathsf{sf} = 1$	$(x > y)$
jle	$\mathsf{zf} = 1 \vee \mathsf{sf} = 1$	$(x \geq y)$

実行時スタック操作命令 実行時スタックの一番上の番地はスタックポインタ %esp に格納されている。この値を変更し、実行時スタックに要素を積んだり（push 命令）、先頭の要素をおろしたり（pop 命令）する命令である。図 7.1 で分かる通り、番地の下位方向に実行時スタックが伸びることに注意しておいてほしい。したがって、例えば push %eax は、レジスタ %eax の値を実行時スタックの一番上に積み、%esp の値を 1 要素分、すなわち 4 減らす[*6]。逆に pop %ebx は、実行時スタックの先頭の要素の値をレジスタ %ebx に格納し、%esp の値を 4 増やす。

関数呼び出し命令 call 命令。第 1 演算数に関数を表すラベルを指定し、その関数を呼び出す。次節で詳しく説明する。

関数からの戻り命令 ret 命令。関数の実行を終了し、関数呼び出し前の番地に制御を移す。次節で詳しく説明する。

7.2 関数の呼び出し

関数呼び出しの際には、考慮しなければならない点がいくつもある。説明のため、main 関数から foo 関数を呼び出すとする。

- foo を呼び出す前に、実引数を設定しなければならない。これは main 側で行われる作業であり、かつ実行時スタックを用いなければならない。
- foo の終了時に返り値を設定し、main に渡さなければならない。
- foo の実行を始める前に、局所変数の領域を実行時スタックに確保し、必要なら初期化を行わなければならない。
- foo を呼び出す前と foo を呼び出した後で、レジスタや実行時スタックの状態は変化してはならない（返り値の設定を除く）[*7]。つまり、foo を呼び出す前に汎用レジスタ、命令ポインタ、スタックポインタ、ベースポインタなどの内容を実行時スタックに退避し、foo の終了時に退避した内容を適切に復帰しなければならない。

実際の機械語命令を見ながら、関数呼び出しのときに何が起こるか説明する。次のような非常に簡単な C プログラムを考える。

```
int foo(int x)
{
  int y = x*x;
```

[*6] ここでは 1 要素が 32 ビット、すなわち 4 バイトであると仮定している。
[*7] 大域データ領域やヒープは変化しても構わない。

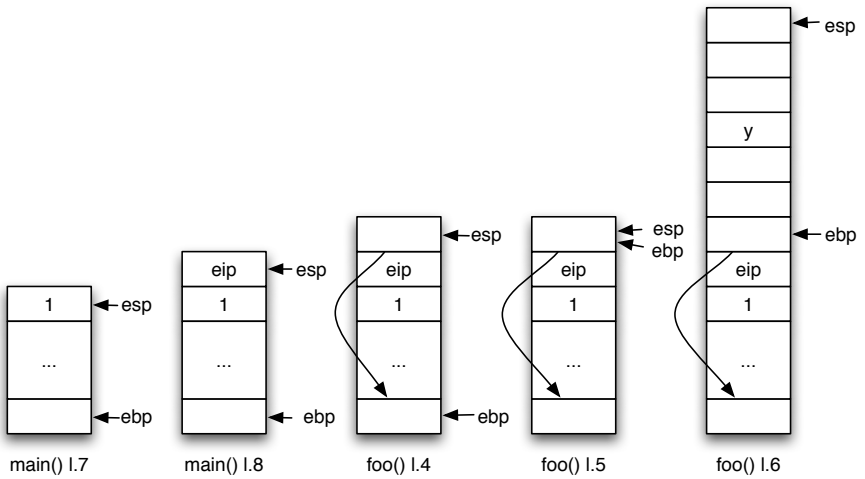


図 7.2　関数 foo 呼び出し時の実行時スタック管理

```
  return y+2;
}
```

この関数を main から呼び出す場合、次のような C プログラムになるだろう。

```
int main()
{
  foo(1);
}
```

さて、関数 foo に対応する機械語命令列は次のようになる（必要部分のみ抜粋)。

```
1  _foo:
2          pushl   %ebp
3          movl    %esp, %ebp
4          subl    $24, %esp
5          movl    8(%ebp), %eax
6          imull   8(%ebp), %eax
7          movl    %eax, -12(%ebp)
8          movl    -12(%ebp), %eax
9          addl    $2, %eax
10         movl    %ebp, %esp
11         popl    %ebp
12         ret
```

また、main 側で foo を呼び出す部分は、次のような機械語命令列になる（必要部分のみ抜粋)。

```
1          movl    $1, (%esp)
2          call    _foo
```

foo を呼び出すとき、foo の終了時それぞれについて、上に示した機械語命令を追っていこう。まず foo の呼び出し時である（図 7.2)。

1. main の 7 行目: 実引数の設定。(%esp)、すなわちスタックポインタの指し示している実行時スタック領域（実行時スタックの先頭）に 1 を格納する。これが x に対応する実引数になる[*8]。
2. main の 8 行目: foo 関数の呼び出し。call 命令では、(命令ポインタ %eip の値 +1) が実行時スタックにプッシュされ（%eip の退避)、_foo ラベル、すなわち関数 foo の 3 行目の番地が %eip に格納される。したがって、次に実行される機械語命令は foo の 3 行目になる。
3. foo の 3 行目: ラベルだけなので、何も行われない。
4. foo の 4 行目: ベースポインタ %ebp の現在の値、すなわち foo 呼び出し前のベースポインタの値を実行時スタックにプッシュする。(%ebp の退避)
5. foo の 5 行目: スタックポインタ %esp の現在の値を %ebp に格納する。これにより、foo の実行に入る前のスタックポインタの値がベースポインタ %ebp に退避されたことになる。(%esp の退避)
6. foo の 6 行目: スタックポインタ %esp の値を 24 減らす。これは、実行時スタックに要素を 6 個積んだことに相当している。この中には、局所変数 y の領域[*9]なども含まれている。
7. foo の 7-9 行目: 局所変数 y の初期化。実引数 x の領域は 8(%ebp)、つまりこの時点でのベースポインタから 2 要素分下の領域であることに注意せよ。

次に関数 foo の終了時の処理を追ってみる（図 7.3)。

1. foo の 10-11 行目: 返り値の設定。Pentium プロセッサでは通常、関数からの返り値は汎用レジスタ %eax を使って、呼び出し元に受け渡される。ここでも、$y + 2$ の計算結果が %eax に格納されている。
2. foo の 12 行目: 現在のベースポインタ %ebp の値をスタックポインタ %esp に格納する。呼び出し時の処理 5 で、呼び出し前のスタックポインタの値を %ebp に退避していたことに注意せよ。つまり、ここで %esp を呼び出し前の状態に復帰している。
3. foo の 13 行目: 実行時スタックの先頭の要素をポップし、ベースポインタ %ebp に格納している。呼び出し前の処理 4 で、呼び出し前のベースポインタの値を実行時スタックに退避していたことに注意せよ。つまり、ここで %ebp を呼び出し前の状態に復帰している。
4. foo の 14 行目: foo 関数からの戻り。ret 命令では、実行時スタックの先頭の要素をポップし、命令ポインタ %eip に格納する。呼び出し前の処理 2 で、呼び出し前の %eip の値を実行時スタックに退避していたことに注意せよ。つまり、ここで %eip を呼び出し前の状態に復帰している。したがって、次の機械語命令は call 命令の番地 +1、すなわち call 命令の次の機械語命令になる。

関数呼び出し時、および関数終了時には、常に上記のような処理が行われる。ただし実際には、他の汎用レジスタの退避・復帰のための機械語命令列も加えられることがある。

[*8] ここでは示していないが、6 行目以前では、実行時スタックの先頭には何も値が格納されていない。したがって、実行時スタックの先頭の値が上書きされることはない。

[*9] -12(%ebp)、すなわちこの時点でのベースポインタから 3 要素分上の領域である。

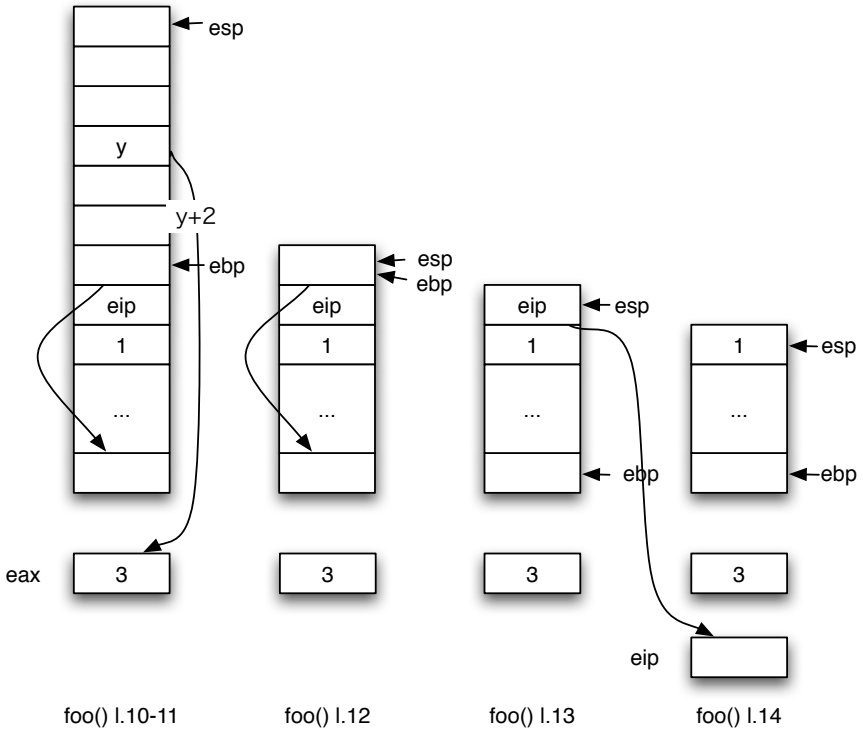


図 7.3　関数 foo からの戻り処理

第 8 章

目的プログラム生成

本講義の最後として、目的プログラムの生成について述べる。ただし、C 言語の構文すべてを扱うと内容が複雑になるので、ここでは概略を述べるにとどめる。また 7 章に引き続き、具体性を持たせるために、Pentium プロセッサの AT&T 形式アセンブリ言語を目的言語とするが、本章で述べる内容は汎用的なものである。

まず初めに、C 言語の代表的な文について、どのような機械語コードを生成すればいいのか述べる。次に、そのいくつかについて、解析木からコードを生成する方法について検討する。

8.1 C 言語の文に対する機械語コード

8.1.1 複文

C 言語の複文は次のような形式である。

{ 宣言文$_1$; $\cdots$ 宣言文$_n$; 文$_1$; $\cdots$ 文$_m$; }

「宣言文」とは変数の宣言、「文」とは宣言文以外の C 言語の文である。C 言語では、複文の先頭に変数の宣言を置くことができ、複文内で宣言された変数は、その複文内でのみ有効な局所変数となる。

これに対応する機械語コードは、次のような構造になる。

宣言文$_1, \cdots,$宣言文$_n$ に対応する変数領域の確保
宣言文$_1$ の初期化（もしあれば）
$\cdots$
宣言文$_n$ の初期化（もしあれば）
文$_1$ の実行
$\cdots$
文$_m$ の実行
局所変数領域の解放

局所変数領域の確保は、具体的には、実行時スタック上に局所変数分の領域を（push 命令などで）積むことに相当する。ただし、変数の初期化や 文$_1, \cdots,$文$_m$ で変数の参照が起こるため、それぞれの局所変数の領域がどこに確保されたのか、複文のコード生成中、記

憶しておく必要がある。これは通常、確保された領域の番地を記号表に書き込むことで対処する。例えば、

int x = 10;

という宣言文に対して x の領域を実行時スタック上に（例えば-16(%ebp) に）確保するとすると、-16(%ebp) を記号表の x の項目に書き込む。次いでこの初期化として、次のような機械語を生成する。

```
movl   -16(%ebp), 10
```

複文の中で確保した局所変数領域は、複文終了時に（pop 命令などで）実行時スタックから下ろすようにする。

8.1.2 条件文

C 言語の条件文の典型的な形は次のようになる。

if (条件式) 文$_1$ else 文$_2$

これに対応する機械語コードは次のようになる。

条件式を計算し、結果をレジスタ R に格納
cmpl R, 0
je L_1
文$_1$ の実行
jmp L_2
L_1: 文$_2$ の実行
L_2:

C 言語では、条件式が 0 以外なら真（文$_1$ が実行）、0 なら偽（文$_2$ が実行）と見なされることに注意されたい。

8.1.3 繰り返し文

ここでは繰り返し文のうち while 文のみを取り上げる。while 文の形式は次のようになる。

while (条件式) 文

これに対する機械語コードは次のようになる。

L_1: 条件式を計算し、結果をレジスタ R に格納
cmpl R, 0
je L_2
jmp L_1
L_2:

もし while 文の中に break 文があれば、(while 文から脱出するということであるから) jmp L_2 を挿入すればよい。また continue 文があれば (次の繰り返しに進むということであるから) jmp L_1 を挿入すればよい。

8.1.4 式文

式文は、C 言語の代入文や関数呼び出しの総称である[*1]。関数呼び出しがどのようなコードになるかは 7.2 節で述べたので、ここでは代入文について取り上げる。

変数 v への代入文 v = e'; に対する機械語コードは次のようになる。ただし $loc(v)$ は、変数 v が割り当てられている番地を表す。

e' の計算を行い、結果をレジスタ R に格納
movl R, $loc(v)$

したがって、i = foo(); のように e' が関数呼び出しの場合には、次のようなコードが生成される。関数からの返り値はレジスタ %eax に格納されることを思い出しておくこと。

call _foo
movl %eax, $loc(i)$

e' が算術式の場合、基本的なコード生成の手順は次のようになる。二項算術演算子 $\circ$ に対応して、機械語の算術命令 $inst$ が用意されているならば、算術式 $e_1 \circ e_2$ のための機械語コードは次のようになる。

e_1 の値をレジスタ R にロード
$inst$ e_2 **の値**, R

例えば式 $x + y$ に対しては次のようなコードが生成される。

movl $loc(x)$, R
addl $loc(y)$, R

また式 $a * b + y$ に対しては次のようなコードが生成される。

movl $loc(a)$, R
imul $loc(b)$, R
movl R, R'
addl $loc(y)$, R'

$a * b$ の計算結果はレジスタ R に残っているので、3 行目ではレジスタ R の値をレジスタ R' にロードする、というコードを生成している。

[*1] 正確には、式 e を実行するだけの文である。e に変数への代入が含まれる場合には、副作用によって変数に計算結果の値が残るとみなされる。プログラム実行の副作用については「計算機言語 I」の講義内容を参照のこと。

8.1.5　コード生成の問題

ここまでの節で述べてきた機械語コードは原則的なもので、実際はさらに工夫を行う必要がある。特に、レジスタの割り当てと無駄なコードの除去は充分注意しなければならない。

例として、上で挙げた式 $a*b+y$ に対する次のコードを考える。

movl $loc(a)$, R
imul $loc(b)$, R
movl R, R'
addl $loc(y)$, R'

このコードでは、$a*b$ の計算結果をレジスタ R に、最終結果をレジスタ R' にそれぞれ割り当てている。しかし、この例では最終結果も R に割り当てても構わないはずである。すると次のようなコードになり、3 つの機械語命令で済み、また使用するレジスタも 1 つで済む。

movl $loc(a)$, R
imul $loc(b)$, R
addl $loc(y)$, R

汎用レジスタの個数は少ないので、無駄なく汎用レジスタを使い回すことは、速いコードを生み出すのに大変重要である。

また機械語命令の数を減らすことも重要である。「たかが機械語命令 1 つ減らしたところで、大して速くなるわけではない」と思うかもしれないが、これが 100 万回のループの中のコードだったらどうだろう。40 秒かかった計算が 30 秒で済むかもしれない。

いったん生成したコードを変形し、より速く効率よいコードに改良することを**最適化**（optimization）という。本講義ではこれ以上深入りしないが、最適化を行うには、解析木からいきなり目的プログラムの機械語命令を生成するのではなく、次節で述べるように、中間的な形式のコード（中間コード）を生成するほうがよい。最適化のためのコード解析が、機械語命令に対してよりも中間コードに対するほうが行いやすいからである。

8.2　翻訳スキームに基づくコード生成

この節では、翻訳スキームに基づいて解析木から中間コードを生成する方法について述べる。ただし、簡単のため、取り上げるのは算術式と while 文にとどめる。

8.2.1　三番地コード

ここでは中間コードとして**三番地コード**（three-address code）というものを用いる。三番地コードの文のうち、本節の説明に必要なものを以下に示す。なお、三番地コードの全命令は [2] の 8.1 節にある。

1. 代入文 x = y op z。ここで op は二項算術演算子である。

2. 代入文 x = op y。ここで op は単項演算子（符号の反転を表す − など）である。
3. 代入文 x = y。y の値を x に代入する。
4. 無条件分岐 goto L。L をラベルに持つ文に分岐する。
5. 条件付き分岐 if x op y goto L。op は比較演算子である。x op y が成立すれば L をラベルに持つ文に分岐する。成立しなければ次の文に制御を移す。

これらから分かるように、三番地コードでは、文の右辺には演算子をたかだか 1 つしか許さない。2 つ以上演算子を含む算術式については、一時変数を用いて、複数の文の列に翻訳しなければならない。たとえば $x + y * z$ という原始言語の式は、次のような文の列になる。

t1 = y * z
t2 = x + t1

8.2.2 三番地コードを生成する翻訳スキーム

多くの場合、三番地コードは 6.4 節で述べた S 属性定義や L 属性定義を用いて、解析木から生成することができる。

まず、右辺に算術式を持つ代入文を考えよう。生成規則は次のようになる。

$$
\begin{aligned}
S &\to \mathsf{id} = E \\
E &\to E + E \\
E &\to E * E \\
E &\to -E \\
E &\to (E) \\
E &\to \mathsf{id}
\end{aligned}
$$

この文法に基づき、代入文 id = id + id * id の解析木を作ると図 8.1 のようになる。

この代入文に対する三番地コードには、さまざまな可能性がある。ここでは、解析木中の内部節点（非終端記号）E それぞれに一時変数を割り振るものとしよう（図 8.1 中の斜字体）。すると、対応する三番地コードは次のようになる[*2]。コード中、関数 $val(id)$ は識別子 id の値を表す。

$$
\begin{aligned}
t_4 &= val(id_2) && E_4 \\
t_1 &= val(id_3) && E_1 \\
t_2 &= val(id_4) && E_2 \\
t_3 &= t_1 * t_2 && E_3 \\
t_5 &= t_4 + t_3 && E_5 \\
id_1 &= t_5 && S
\end{aligned}
$$

明らかに各文はそれぞれ右に示した内部節点に対応しており、かつその節点の子節点の一時変数のみに依存している。しかも、対応する内部節点の出現順は、解析木を帰りがけ順にたどった順に一致している。

[*2] 明らかに冗長な文が含まれているが、これは最適化によって改善するものとし、ここでは問題にしない。

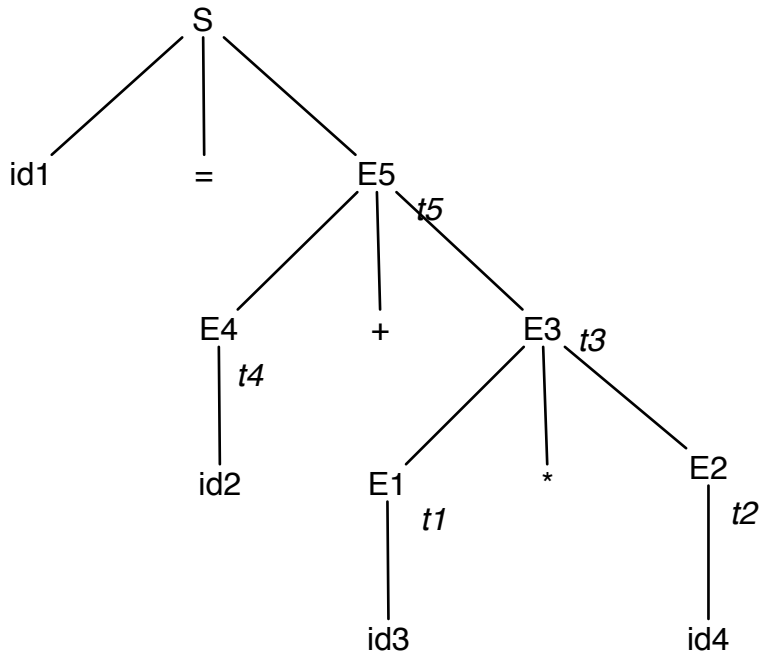


図 8.1　代入文 id = id + id * id に対する解析木

したがって、上記の三番地コードは S 属性定義によって生成することが可能である。以下に三番地コードを生成する S 属性定義を示す。ここで、非終端記号の属性 $place$ は、その非終端記号に対応する一時変数名を保持している。また関数 $newtemp()$ は新たな一時変数を生成する関数、$gencode()$ は引数から三番地コードを生成する関数である。

$$
\begin{aligned}
S \rightarrow & \mathsf{id} = E \ \{gencode(\mathsf{id} = E.place);\} \\
E \rightarrow & \{E.place = newtemp();\} \\
& E_1 + E_2 \ \{gencode(E.place = E_1.place + E_2.place);\} \\
E \rightarrow & \{E.place = newtemp();\} \\
& E_1 * E_2 \ \{gencode(E.place = E_1.place * E_2.place);\} \\
E \rightarrow & \{E.place = newtemp();\} \\
& -E_1 \ \{gencode(E.place = -E_1.place);\} \\
E \rightarrow & \{E.place = newtemp();\} \\
& (E_1) \ \{gencode(E.place = E_1.place);\} \\
E \rightarrow & \{E.place = newtemp();\} \\
& \mathsf{id} \ \{gencode(E.place = val(\mathsf{id}));\}
\end{aligned}
$$

もう一つ、翻訳スキームを用いた三番地コード生成の例として、while 文を取り上げる。while 文の生成規則は次のようになる。

$$S \rightarrow \mathsf{while} \ (E) \ S$$

これに対応する三番地コードは次のようになる。添字の r は右辺に出現する記号であることを表す。

L_1: E に対するコード
if $E.place = 0$ goto L_2
S_r に対するコード
goto L_1
L_2:

これも代入文と同様、S 属性定義を用いて解析木から生成することが可能である。以下にこの三番地コードを生成する S 属性定義を示す。ここで属性 $begin, end$ はそれぞれ L_1, L_2 に相当するラベルを保持する。また関数 $newlabel()$ は、新たなラベルを生成する関数である。

$S \rightarrow \{S.begin = newlabel();\ S.end = newlabel();\}$
$\{gencode(S.begin:);\}$
while (E) $\{/* \text{ここまでで } E \text{ のコードが生成されている} */\}$
$\{gencode(\text{if } E.place = 0 \text{ goto } S.end);\}$
S $\{/* \text{ここまでで } S_r \text{のコードが生成されている} */\}$
$\{gencode(\text{goto } S.begin);\}$
$\{gencode(S.end:);\}$

参考文献

[1] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers - Principles, Techniques, & Tools (Second Edition)*. Addison Wesley, 2007.

[2] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers - Principles, Techniques, and Tools*. Addison-Wesley, 1986. 日本語訳: 「コンパイラ I・II」（原田賢一訳、サイエンス社）.

[3] 中田育男. コンパイラの構成と最適化. 朝倉書店, 1999.

[4] 湯浅太一. コンパイラ. 昭晃堂, 2001.

[5] A. V. エイホ, J. E. ホップクロフト, J. D. ウルマン. アルゴリズムの設計と解析 I. サイエンス社, 1977.

[6] B. W. カーニハン, D. M. リッチー. プログラミング言語 C 第 2 版. 共立出版, 1989.

[7] J. ホップクロフト, R. モトワニ, 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.

練習問題の解答

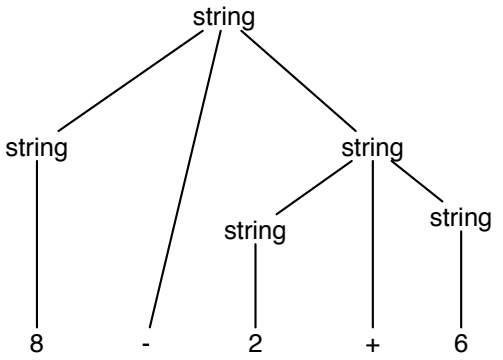
1 章

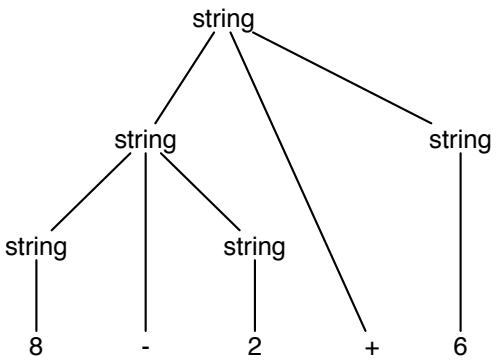
■1.1　略

■1.2　`x, =, 0, ;, while, (, i, <, 100, ), {, x, +=, i, ;, i, ++, ;, }`

4 章

■4.1





+, − は左結合の演算子であり、括弧がなければ式の左から計算していくのが正しい意味になる。したがって、下側の解析木のほうが自然である。このように、文法によっては、同じ記号列に対して解析木が 2 通り以上考えられる場合がある。このような文法を**曖昧である**という。

■4.2　0 が n 個 $(n \geq 1)$ 並んだ後に 1 が n 個並んだような文字列すべてからなる言語。$0, 1$ をそれぞれ $\{, \}$ に置き換えてみると、これは対応関係のとれた括弧の入れ子を表す。この言語は正則言語ではない例として有名である。つまり、この言語は正則表現では書けず、またこの言語を受理する有限オートマトンも作れない。

■4.3

$$S \Rightarrow A1B \Rightarrow 0A1B \Rightarrow 00A1B \Rightarrow 001B \Rightarrow 0010B \Rightarrow 00101B \Rightarrow 00101$$

$$S \Rightarrow A1B \Rightarrow A10B \Rightarrow A101B \Rightarrow A101 \Rightarrow 0A101 \Rightarrow 00A101 \Rightarrow 00101$$

■4.4

$$\begin{aligned} E &\to TE' \\ E' &\to +TE' \mid \epsilon \\ T &\to FT' \\ T' &\to *FT' \mid \epsilon \\ F &\to (E) \mid \mathbf{id} \end{aligned}$$