文脈自由文法と bison

Qua

2025 年 3 月 25 日

文法とは

形式言語 (以下言語という) は (ある条件を満たす) 記号列の集合として定義される。言語の要素である各記号列を語という。言語を定義する方法にはいくつかある。例えば、「0 と 1 からなる列で 1 が二つ以上連続して現れず、かつ末尾が 1 であるもの全体」のように条件を言葉で示したり、$(\mathbf{0}^*\mathbf{10})^*\mathbf{0}^*\mathbf{1}$ のように正規表現で示したり、図 1 のような有限オートマトン M に受理される語[*1] 全体として定義したりすることができる。

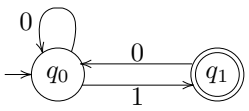


図 1　$(\mathbf{0}^*\mathbf{10})^*\mathbf{0}^*\mathbf{1}$ を受理する有限オートマトン

言語を定義する方法として次のようなことを考える。語 (記号列) を構成する記号を「終端記号」という。それ以外に抽象的な語の構成要素を表す文字である「非終端記号」を用意する。非終端記号のうちの 1 つは「開始記号」と呼ばれる。また、

$$\alpha \to \beta$$

という形の生成規則をいくつか用意する。ただし、α と β は終端記号と非終端記号からなる列であり、α には少なくとも 1 つの非終端記号が含まれるものとする。開始記号 1 つからなる文字列から始めて、文字列に含まれる α の部分を β で置き換えること (これを直接に導出するという) を、すべてが終端記号からなる文字列が得られるまで繰り返す。このような操作を導出または生成という。「導出によって得られる語全体の集合」として言語を定義できる。

形式文法 (以下、文法という) は $G = (N, \Sigma, P, S)$ で定義される。ここで、N は非終端記号の集合、Σ は終端記号の集合、P は生成規則の集合、S は開始記号である。文法 G で導出によって得られる語全体の集合を $L(G)$ で表す。

例 1

$$\begin{aligned}
N &= \{\langle\ \text{文}\ \rangle, \langle\ \text{主語}\ \rangle, \langle\ \text{述語}\ \rangle, \langle\ \text{動詞}\ \rangle, \langle\ \text{形容詞}\ \rangle\} \\
\Sigma &= \{\ \text{私}, \text{あなた}, \text{は}, \text{走る}, \text{食べる}, \text{強い}, \text{優しい}\ \} \\
P &= \{\langle\ \text{文}\ \rangle \to \langle\ \text{主語}\ \rangle\langle\ \text{述語}\ \rangle,\quad \langle\ \text{主語}\ \rangle \to \text{私 は},\quad \langle\ \text{主語}\ \rangle \to \text{あなた は},\quad \langle\ \text{述語}\ \rangle \to \langle\ \text{動詞}\ \rangle, \\
&\qquad \langle\ \text{述語}\ \rangle \to \langle\ \text{形容詞}\ \rangle,\quad \langle\ \text{動詞}\ \rangle \to \text{走る},\quad \langle\ \text{動詞}\ \rangle \to \text{食べる},\quad \langle\ \text{形容詞}\ \rangle \to \text{強い},\quad \langle\ \text{形容詞}\ \rangle \to \text{優しい}\ \} \\
S &= \langle\ \text{文}\ \rangle
\end{aligned}$$

は一つの文法である。この文法から

$$\langle\ \text{文}\ \rangle \to \langle\ \text{主語}\ \rangle\langle\ \text{述語}\ \rangle \to \text{私 は}\langle\ \text{述語}\ \rangle \to \text{私 は}\langle\ \text{動詞}\ \rangle \to \text{私 は 走る}$$

[*1] 短い $\to$ が付けられている初期状態から、与えられた語を左から 1 文字ずつ読んで、その文字が書かれている矢印をたどっていく。すべての文字を読み終わったときに二重丸で表された受理状態に至るとき、またそのときに限ってその語は受理される。

というように「私は走る」という文ができる。$L(G) = \{$ 私は走る, 私は食べる, 私は強い, 私は優しい, あなたは走る, あなたは食べる, あなたは強い, あなたは優しい $\}$ である。

例 2 文法

$$
\begin{aligned}
&N = \{\langle Q_0\rangle, \langle Q_1\rangle\} \\
&\Sigma = \{0, 1\} \\
&P = \{\langle Q_0\rangle \to 0\langle Q_0\rangle, \quad \langle Q_0\rangle \to 1\langle Q_1\rangle, \quad \langle Q_0\rangle \to 1, \quad \langle Q_1\rangle \to 0\langle Q_0\rangle\} \\
&S = Q_0
\end{aligned}
$$

で定義される文法によって生成される語全体からなる言語は図 1 の有限オートマトンが受理する言語と等しい。

問題 1 図 1 のオートマトンで 00101 が受理されることを確かめよ。また例 2 の文法で 00101 が生成されることを確かめよ。何かに気づくはず。

文脈自由文法

文法の生成規則の左辺がちょうど 1 つの非終端記号であるものに限られるとき、その文法を文脈自由文法という。例 1 の文法も文脈自由文法である。

生成規則の適用を、左辺を親、右辺の各記号を子として、木で表すことができる。これを導出木という。例 1 で示した導出は図 2 の導出木で表される。

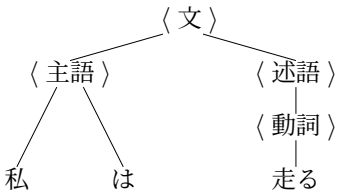


図 2 「私は走る」の導出木

とくにプログラミングに関わる文脈自由文法の表現法としてバッカス・ナウア記法 (BNF) がよく使われる。BNF では、左辺が共通の生成規則

$$A \to \beta_1, \quad A \to \beta_2, \quad \cdots, A \to \beta_n$$

を

$$A ::= \beta_1 \mid \beta_2 \mid \cdots \mid \beta_n$$

のように記述する。

例 3 例 1 で示した文法は BNF 記法によって以下の通りに表される。

〈 文 〉 ::= 〈 主語 〉〈 述語 〉
〈 主語 〉 ::= 私は | あなたは
〈 述語 〉 ::= 〈 動詞 〉 | 〈 形容詞 〉
〈 動詞 〉 ::= 走る | 食べる
〈 形容詞 〉 ::= 強い | 優しい

例 4 四則演算からなる式を表す文脈自由文法 $G = (N, \Sigma, P, S)$ は次のように定義できる。

$$
\begin{aligned}
N &= \{\langle E\rangle, \langle T\rangle, \langle F\rangle\} \\
\Sigma &= \{\mathbf{+}, \mathbf{-}, \mathbf{*}, \mathbf{/}, \mathbf{(}, \mathbf{)}, \mathbf{num}\} \\
P &= \{\langle E\rangle \to \langle E\rangle\ \mathbf{+}\ \langle T\rangle, \langle E\rangle \to \langle E\rangle\ \mathbf{-}\ \langle T\rangle, \langle E\rangle \to \langle T\rangle, \langle T\rangle \to \langle T\rangle\ \mathbf{*}\ \langle F\rangle, \langle T\rangle \to \langle T\rangle\ \mathbf{/}\ \langle F\rangle, \langle T\rangle \to \langle F\rangle, \\
&\quad \langle F\rangle \to \mathbf{(}\ \langle E\rangle\ \mathbf{)}, \langle F\rangle \to \mathbf{num}\} \\
S &= \langle E\rangle
\end{aligned}
$$

ここで、$\langle E\rangle$, $\langle T\rangle$, $\langle F\rangle$ はそれぞれ式、項、因子を表す非終端記号、**num** は数または変数を表す終端記号とする。これに対応する BNF は次のようになる。

$$
\begin{aligned}
&\langle E\rangle ::= \langle E\rangle\mathbf{+}\langle T\rangle \mid \langle E\rangle\mathbf{-}\langle T\rangle \mid \langle T\rangle \\
&\langle T\rangle ::= \langle T\rangle\mathbf{*}\langle F\rangle \mid \langle T\rangle\mathbf{/}\langle F\rangle \mid \langle F\rangle \\
&\langle F\rangle ::= \mathbf{(}\langle E\rangle\mathbf{)} \mid \mathbf{num}
\end{aligned}
$$

さて、この文法では、左辺と右辺に同じ非終端記号が現れている。例えば、

$$\langle E\rangle ::= \langle E\rangle\mathbf{+}\langle T\rangle \mid \langle E\rangle\mathbf{-}\langle T\rangle \mid \langle T\rangle$$

に注目してみよう。この規則のうち、$\langle E\rangle ::= \langle E\rangle\mathbf{+}\langle T\rangle$ を繰り返して適用し、最後に $\langle E\rangle ::= \langle T\rangle$ を適用すると、

$$\langle E\rangle \to \langle E\rangle\mathbf{+}\langle T\rangle \to \langle E\rangle\mathbf{+}\langle T\rangle\mathbf{+}\langle T\rangle \to \langle E\rangle\mathbf{+}\langle T\rangle\mathbf{+}\langle T\rangle\mathbf{+}\langle T\rangle \cdots \to \langle E\rangle\mathbf{+}\langle T\rangle\mathbf{+}\cdots\mathbf{+}\langle T\rangle \to \langle T\rangle\mathbf{+}\langle T\rangle\mathbf{+}\cdots\mathbf{+}\langle T\rangle$$

となる。$\langle E\rangle ::= \langle E\rangle\mathbf{-}\langle T\rangle$ も同様であり、合わせると、「式はいくつかの項の和または差である」ことを表している。

この文法に対して、$\mathbf{num} + \mathbf{num} * \mathbf{num}$ は次のように導出される。

$$
\begin{aligned}
&\langle E\rangle \to \langle E\rangle + \langle T\rangle \to \langle T\rangle + \langle T\rangle \to \langle F\rangle + \langle T\rangle \to \mathbf{num} + \langle T\rangle \to \mathbf{num} + \langle T\rangle * \langle F\rangle \to \mathbf{num} + \langle F\rangle * \langle F\rangle \\
&\quad \to \mathbf{num} + \mathbf{num} * \langle F\rangle \to \mathbf{num} + \mathbf{num} * \mathbf{num}
\end{aligned}
$$

この導出に対応する導出木は図 3 のとおりである。

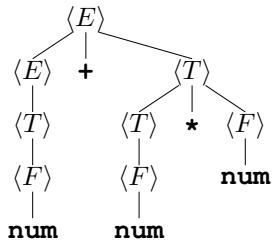


図 3　$\mathbf{num} + \mathbf{num} * \mathbf{num}$ の導出木

問題 2　$(\mathbf{num} - \mathbf{num})/\mathbf{num}$ の導出木を求めよ。

構文解析と意味値

ある言語を生成する文法と、その言語に属する語に対して、導出木を作ることを構文解析という。前節の式の例で、**num** は数値または変数を代表しているものとした。ここで、語に含まれるそれぞれの **num** が表す数値がわかれば、具体的な式としてその値を計算することができる。$3 + 4 * 5$ の計算をしてみよう。説明のため、図 3 で示した $\mathbf{num} + \mathbf{num} * \mathbf{num}$ の導出木のそれぞれの非終端記号と **num** とに番号をつけて区別できるようにしたものを図 4 に示す。また、生成規則の適用の順番を丸数字で表す。

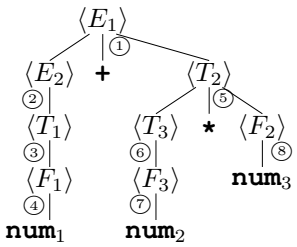


図 4　図 3 の導出木

$\mathbf{num}_1$, $\mathbf{num}_2$, $\mathbf{num}_3$ の値はそれぞれ 3, 4, 5 である。生成規則の適用を⑧, ⑦, ...,①のように葉から根に向かって見ていく。葉から根に向かって逆に見て生成規則の適用していく。生成規則の右辺が 1 つの記号だけのときには左辺の非終端記号の値は右辺の値そのものとする。また生成規則の右辺が演算子を含む計算式のときには、右辺の記号をその値に置き換えて計算を実行した結果を左辺の非終端記号の値とする。以下のようにして、根 $\langle E_1 \rangle$ に対応する数値 $3 + 4 * 5 = 23$ が得られる。

⑧で、$\langle F_2 \rangle$ の値が $\mathbf{num}_3 = 5$ になる。　⑦で、$\langle F_3 \rangle$ の値が $\mathbf{num}_2 = 4$ になる。
⑥で、$\langle T_3 \rangle$ の値が $\langle F_3 \rangle = 4$ になる。　⑤で、$\langle T_2 \rangle$ の値が $\langle T_3 \rangle * \langle F_2 \rangle = 4 * 5 = 20$ になる。
④で、$\langle F_1 \rangle$ の値が $\mathbf{num}_1 = 3$ になる。　③で、$\langle T_1 \rangle$ の値が $\langle F_1 \rangle = 3$ になる。
②で、$\langle E_2 \rangle$ の値が $\langle T_1 \rangle = 3$ になる。　①で、$\langle T_1 \rangle$ の値が $\langle E_2 \rangle + \langle T_2 \rangle = 3 + 20 = 23$ になる。

非終端記号のそれぞれに数値を対応させて計算を行う代わりに、部分木を対応させて数式に対応した構文木を作ることもできる。構文木と導出木の違いに注意すること。数式を表す構文木では、内点は演算子、葉は数値 (または変数) を表し、各内点の子は対応する演算子で演算される対象である。**num** はその値に対応する 1 点だけからなる木とする。導出規則の右辺が演算子を含む計算式のときには、左辺の非終端記号の値は、右辺に含まれる演算子を根、その他の終端・非終端記号に対応する導出木をその根の子とするような導出木とする。例えば、$\mathbf{num}_1$ の値は、3 に対応する 1 点である。$\langle F_1 \rangle$ は導出規則 $\langle F_1 \rangle \to \mathbf{num}_1$ に対応するので、その値は $\mathbf{num}_1$ の値とする。$\langle E_1 \rangle$ は導出規則 $\langle E_1 \rangle \to \langle E_2 \rangle$**+**$\langle T_2 \rangle$ に対応するので、$\langle E_1 \rangle$ の値は **+** を根として、$\langle E_1 \rangle$ に対応する木と $\langle T_2 \rangle$ に対応する木を根の左右の部分木としてもつような木となる。

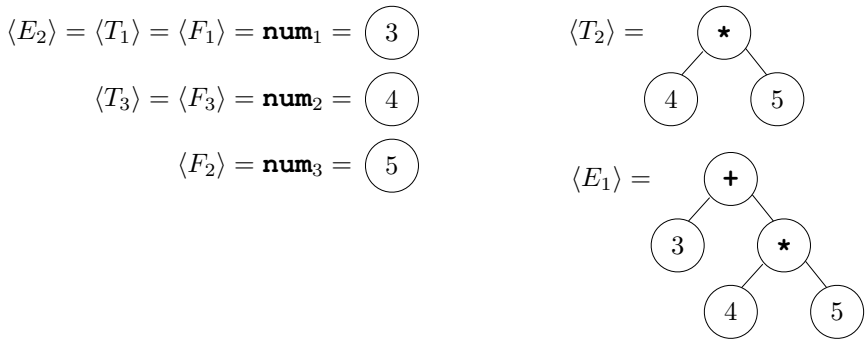


図 5　構文木を意味値とする場合

これらのような記号に対応した値 (前者では数値、後者では構文木) を、その記号の意味値と呼ぶ。

bison

bison は文法を入力して、その構文解析を行う C プログラム (構文解析器という) を出力するアプリケーションである。以下のリスト 1 に入力ファイルの例を示す。入力ファイルは `%%` で 3 つの部分に分けられている。最初の部分 (例の 1〜8 行目) が定義部、次の部分 (10〜22 行目) がルール部、最後の部分 (24 行目以降) がサブルーチン部である。定義部の `%{` と `%}` で囲まれた部分、および、サブルーチン部は C プログラムの一部として出力される。ここで、

`int yylex()`	構文解析器から呼び出され、解析すべき語の終端記号を 1 つずつ返す。 終端記号の意味値がある場合には、大域変数 `yylval` に代入する。
`int yyparse()`	構文解析を実行する
`int yyerror()`	構文解析中でエラーが発生した場合に呼び出される

定義部では、記号の定義などを行う。この例では、NUM という終端記号があること、開始記号が `line` であることを設定している。なお、開始記号を指定しない場合には、ルール部の最初に出てくる生成規則の左辺が開始記号 (この例では E) であるとみなされる。なお、bison では各非終端記号の意味値はとくに指定しなければ int 型であるとみなされる。

ルール部が文法の生成規則の記述になっている。BNF の`::=`を`:` に置き換え、最後にセミコロンを記している。各生成規則の後ろに`{ }`で囲まれている部分はアクションと呼ばれる。アクションの中には C プログラムを書くことができる。ただし、`$$`は左辺の非終端記号、$\$n$ は右辺の記号の並びのうち n 番目の記号を表す。ここで、`'+'` のような記号も数えなければならないことに注意する。アクションがない場合には、`$$=$1;` が書かれたものとみなされる。`$$`を計算することで意味値を得ていること、この例では前節で説明した計算式の値を計算していることに気づいてもらいたい。

リスト 1　電卓

```
1  %{
2  #include <stdio.h>
3  int yylex();
4  int yyerror( char * );
5  %}
6
7  %token NUM
8  %start line
9  %%
10 E    : E '+' T        { $$ = $1 + $3; }
11     | E '-' T        { $$ = $1 - $3; }
12     | T
13     ;
14 T    : T '*' F        { $$ = $1 * $3; }
15     | T '/' F        { $$ = $1 / $3; }
16     | F
17     ;
18 F    : '(' E ')'      { $$ = $2; }
19     | NUM
20     ;
21 line: E              { printf("%d\n", $1); }
22     ;
23 %%
24 int ntoken;
25 char **tokens;
26 int yylex(){
27     static int i=0;
28     if( ++i >= ntoken ) return 0;
29     if( sscanf(tokens[i], "%d", &yylval)==1 ) return NUM;
30     return tokens[i][0];
31 }
32 int yyerror( char *msg ){
33     fprintf( stderr, "yyerror(): \"%s\".\n", msg );
34     return 0;
35 }
36 int main( int argc, char **argv ){
37     ntoken = argc; tokens = argv;
38     yyparse();
39     return 0;
40 }
```

問題 3　リスト 1 のプログラムを calc.y として作成し、次の手順で実行ファイルを作成せよ。

```
bison calc.y
```

```
gcc -o calc calc.tab.c
```

できあがった実行ファイル calc は、引数として与えた式を計算する。数値、記号の間にはスペースを入れて区切って入力する。なお、*, (,) などのシェルにとって意味のある記号は、前に\を付けてエスケープすること。

```
./calc \( 3 + 5 \) \* 2
```

リスト 1 では整数しか計算できない電卓ができる。浮動小数点値を計算させるには、21 行目の %d を %f に、29 行目の %d を %lf に変えるとともに、定義部の先頭の %{と %}に囲まれた部分に#define YYSTYPE double を加える。これによって、意味値が double であることが指定される。

あいまい性

これまでの例で出てきた式を表す文法

$\langle E\rangle$ ::= $\langle E\rangle$ **+** $\langle T\rangle$ | $\langle E\rangle$ **−** $\langle T\rangle$ | $\langle T\rangle$
$\langle T\rangle$ ::= $\langle T\rangle$ ***** $\langle F\rangle$ | $\langle T\rangle$ **/** $\langle F\rangle$ | $\langle F\rangle$
$\langle F\rangle$ ::= **(** $\langle E\rangle$ **)** | **num**

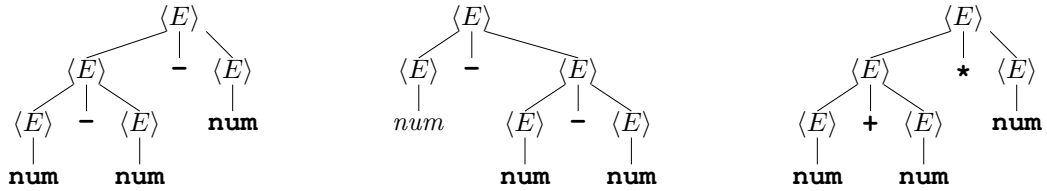
では、3 つの非終端記号があってややこしく見える。生成規則中に $\langle E\rangle$ ::= $\langle T\rangle$, $\langle T\rangle$ ::= $\langle F\rangle$ があるので、$\langle E\rangle$, $\langle T\rangle$, $\langle F\rangle$ を全て一緒にして、

$\langle E\rangle$ ::= $\langle E\rangle$ **+** $\langle E\rangle$ | $\langle E\rangle$ **−** $\langle E\rangle$ | $\langle E\rangle$ ***** $\langle E\rangle$ | $\langle E\rangle$ **/** $\langle E\rangle$ | **(** $\langle E\rangle$ **)** | **num**

とするのはどうだろうか。この方法では、次の 2 点で問題となる。まず、すぐにわかるように、すべての演算子が同等に扱われているので、掛け算、割り算が足し算、引き算よりも優先であることが記述されていない。もう一つ、後の例から分かるように計算の順序が一意に定まらないという問題もある。例えば、結合律が成り立たない引き算または割り算が連続するときにどちらを先にするかで結果が異なる。[*2]

この文法で **num** − **num** − **num** の導出木を求めると、以下の (a)(b) の 2 通りになる。(a) は (**num** − **num**) − **num** に、(b) は **num** − (**num** − **num**) にそれぞれ対応している。(a) が望ましい計算順序である。

このように 1 つの語に対して 2 通り以上の導出木ができる文脈自由文法はあいまいであるという。



(a) 左結合的　(b) 右結合的　(c) 和優先？

図 6　あいまいな文法の導出木

また、**num** + **num** ∗ **num** の導出木も 2 通りある。積は和よりも優先順位が高いので、和を先に計算することに相当する (c) は望ましくない導出木である。bison にはこのような問題を回避するための仕組みがある。

リスト 2　あいまいな文法

```
1 %{
2 #include <stdio.h>
3 int yylex();
```

[*2] なお、後置記法の式は $\langle E\rangle$ ::= $\langle E\rangle$ $\langle E\rangle$ **+** | $\langle E\rangle$ $\langle E\rangle$ **−** | $\langle E\rangle$ $\langle E\rangle$ ***** | $\langle E\rangle$ $\langle E\rangle$ **/** | **num** でこれらの問題を気にせずに記述可能である。

```
4  int yyerror( char * );
5  %}
6
7  %token NUM
8  %start line
9  %%
10 line: E              { printf("%d\n", $1); }
11     ;
12 E   : E '+' E        { $$ = $1 + $3; }
13     | E '-' E        { $$ = $1 - $3; }
14     | E '*' E        { $$ = $1 * $3; }
15     | E '/' E        { $$ = $1 / $3; }
16     | '(' E ')'      { $$ = $2; }
17     | NUM
18     ;
19 %%
20
21 int ntoken;
22 char **tokens;
23 int yylex(){
24     static int i=0;
25     if( ++i >= ntoken ) return 0;
26     if( sscanf(tokens[i], "%d", &yylval)==1 ) return NUM;
27     return tokens[i][0];
28 }
29 int yyerror( char *msg ){
30     fprintf( stderr, "yyerror(): \"%s\".\n", msg );
31     return 0;
32 }
33 int main( int argc, char **argv ){
34     ntoken = argc; tokens = argv;
35     yyparse();
36     return 0;
37 }
```

このファイルを bison にかけると、次のような警告が表示される。この警告は文法があいまいであることに起因する。

```
calc1.y: warning: 16 shift/reduce conflicts [-Wconflicts-sr]
```

警告なので、とりあえず C プログラムは出力され、実行ファイルも作成できるが、例えば $12/6-2$ を計算させると答えが 3 になってしまう。

定義部に次の 2 行を加えてみよう。今度は警告が出されず、思ったように式が計算されるだろう。

リスト 3　あいまい性の解決

```
1  %left '+' '-'
2  %left '*' '/'
```

これらは、和差積商の演算が左結合的であること、和と差よりも積と商が優先されることを指定している。べき乗のような右結合的な演算 ($a^{b^c} = a^{(b^c)}$) は `%right` に続けて記述する。各行には優先度が等しい演算を並べ、優先度が低い順に記す。

問題 4　通常の記法で書かれた式を後置記法に変換するプログラムを作ろう。bison の入力ファイルはリスト 4 のように作る。ただし、ルール部の空欄は適宜埋めること。10〜13 行目と 18, 19 行目の記述は、非終端記号の意味値の型が 2 種類以上ある場合の対応である。今回の場合は、NUM が int 型、E が NodePtr 型である。リスト 5 の node.h、リスト 6 の node.c を入力して、bison の出力ファイルと node.c を一緒に gcc でコンパイルすれば目的のプログラムができる。

リスト 4　構文木の作成 tree.y

```
1  %{
2  #include <stdio.h>
3  #include "node.h"
```

```
4  int yylex();
5  int yyerror( char * );
6  NodePtr root;
7  NodePtr newBinopNode( char, NodePtr, NodePtr );
8  %}
9
10 %union{
11     int ival;
12     NodePtr node;
13 }
14
15 %left '+' '-'
16 %left '*' '/'
17
18 %token<ival> NUM
19 %type<node> E
20
21 %%
22 line: E              { root = $1; }
23     ;
24 E   : E '+' E        { $$ = newBinopNode( '+', $1, $3 ); }
25     | E '-' E        {                                   }
26     | E '*' E        {                                   }
27     | E '/' E        {                                   }
28     | '(' E ')'      {                                   }
29     | NUM            { $$ = newNode( $1 ); }
30     ;
31 %%
32 NodePtr newBinopNode( char op, NodePtr l, NodePtr r ){
33 NodePtr n;
34     n = newNode( op );
35     n->left = l;
36     n->right = r;
37     return n;
38 }
39
40 int ntoken;
41 char **tokens;
42 int yylex(){
43     static int i=0;
44     if( ++i >= ntoken ) return 0;
45     if( sscanf(tokens[i], "%d", &yylval)==1 ) return NUM;
46     return tokens[i][0];
47 }
48
49 int yyerror( char *msg ){
50     fprintf( stderr , "yyerror(): \"%s\".\n", msg );
51     return 0;
52 }
53
54 int main( int argc, char **argv ){
55     ntoken = argc; tokens = argv;
56     yyparse();
57     showPostfix( root );
58     return 0;
59 }
```

リスト 5　node.h

```
1  /* 木を表す構造体の定義 */
2  typedef struct Node * NodePtr;
3  typedef struct Node {
4      int val;
5      NodePtr left, right;
```

```
6  } Node;
7
8  /* プロトタイプ宣言 */
9  NodePtr newNode( int );
10 void push( NodePtr );
11 NodePtr pop();
12 void showStack();
13 void showInfix( NodePtr );
14 void showPostfix( NodePtr );
```

リスト 6　node.c

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include "node.h"
4  /* スタック */
5  #define stacksize 100
6  NodePtr stack[stacksize];
7  int ptr = stacksize;
8
9  /* 新規ノードを作成 */
10 NodePtr newNode( int val ){
11 NodePtr r;
12     r = malloc( sizeof( Node ) );
13     r->val = val;
14     r->left = r->right = NULL;
15     return r;
16 }
17
18 /* スタック操作 */
19 void push( NodePtr val ){
20     ptr--;
21     stack[ptr] = val;
22 }
23
24 NodePtr pop(){
25 NodePtr r;
26     r = stack[ptr];
27     ptr++;
28     return r;
29 }
30
31 /* スタックの内容を表示 */
32 void showStack(){
33 int i;
34     for( i=ptr; i<stacksize; i++ ){
35         printf( "%d:", i );
36         showInfix( stack[i] );
37         puts( "" );
38     }
39     puts( "----------------" );
40 }
41
42 /* 中置表現に変換 */
43 void showInfix( NodePtr t )
44 {
45     /* 葉の場合 */
46     if( t->left == NULL ){
47         printf( "%d", t->val );   /* 数値 */
48         return;
49     }
50     /* 内点の場合、以下を順に表示 */
51     putchar( '(' );          /* 開きカッコ */
52     showInfix( t->left );    /* 左部分木の式 */
```

```
53      putchar( t->val );        /* 演算子 */
54      showInfix( t->right );    /* 右部分木の式 */
55      putchar( ')' );           /* 閉じカッコ */
56      return;
57  }
58
59  /* 後置記法に変換 */
60  void showPostfix( NodePtr t )
61  {
62      /* 葉の場合 */
63      if( t->left == NULL ){
64          printf( "%d", t->val ); /* 数値 */
65          return;
66      }
67      showPostfix( t->left );      /* 左部分木の式 */
68      putchar( ' ' );
69      showPostfix( t->right );     /* 右部分木の式 */
70      putchar( ' ' );
71      putchar( t->val );           /* 演算子 */
72      return;
73  }
```

あいまい性の利用

あいまい性をあえて残しておくことも考えられる。次の文法を考える。

$\langle S \rangle$ ::= **if cond** $\langle S \rangle$ | **if cond** $\langle S \rangle$ **else** $\langle S \rangle$ | **stmt**

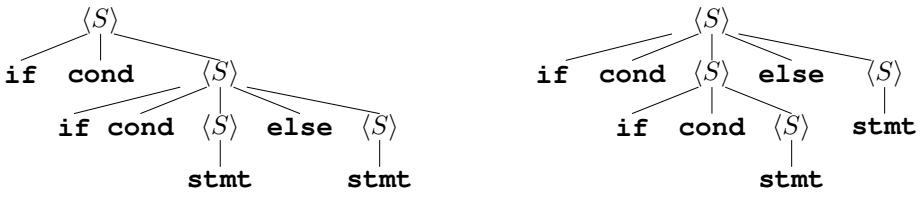
$\langle S \rangle$ はプログラムの「文」を表す非終端記号である。**if** と **else** はそれぞれ予約語 if と else、**cond** は (条件式)、**stmt** は if 文以外の文を表す終端記号とする。この文法はあいまいであり、C や Java の文

```
if( x==0 ) if( y==0 ) z=0; else z=1;
```

に相当する

if cond if cond stmt else stmt

に対して、2 通りの導出木が存在する。



(a) (b)

図 7　if-else-if の 2 つの構文木

(a) は **if cond** { **if cond stmt else stmt** } (b) は **if cond** { **if cond stmt** } **else stmt** という解釈であり、C 言語を始めとする多くのプログラミング言語では (a) の解釈が正しい。bison は (a) の解釈となるような構文解析器を作ってくれるが、警告が出される。

どうしても警告を出したくないという場合は、次のような文法を考えることもできる。$\langle S_a \rangle$ は対応する else がある if 文

または if 文以外の文を表す。$\langle S_b \rangle$ は対応する else がない if 文を表す。

$$\langle S \rangle ::= \langle S_a \rangle \mid \langle S_b \rangle$$
$$\langle S_a \rangle ::= \texttt{if cond} \langle S_a \rangle \texttt{ else} \langle S_a \rangle \mid \texttt{stmt}$$
$$\langle S_b \rangle ::= \texttt{if cond} \langle S \rangle \mid \texttt{if cond} \langle S_a \rangle \texttt{ else} \langle S_b \rangle$$

この文法での **if cond if cond stmt else stmt** に対する導出木を図 8 に示す。

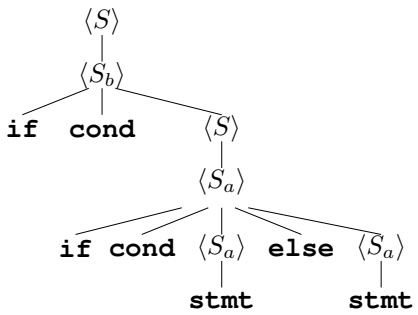


図 8　あいまい性のない文法

bison が出力する構文解析器がやっていること

bison が出力する構文解析器は次のようなことを実行する。詳しくは言語処理系論の講義にて。構文解析器はオートマトンと 2 本のスタックからなる。スタックのうち、1 本はオートマトンの状態を、もう 1 本は記号とその意味値の組を記憶する。構文解析器は入力された語に対して、その記号を一つずつ読んで、シフト動作または還元動作を行う。

シフト動作は、sn と表される。まず、オートマトンは状態 n に遷移する。それとともに第 1 のスタックに n を push し、第 2 のスタックに入力記号とその意味値の組を push する。

還元動作は、rk と表される。k は文法規則の番号を表す。k 番目の文法規則の左辺が A であり、右辺が m 個の記号からなるものとする。まず、第 1 のスタックから m 個の番号を pop して捨てる。また、第 2 のスタックから m 個の記号と意味値の組を pop する。記号の列は文法規則の右辺と等しいはずである。次に pop した意味値の列から A の意味値を計算し、第 2 のスタックに A とその意味値の組を push する。最後に、オートマトンをスタックのトップにある状態と A とによって決まる状態へ遷移させ、第 1 のスタックにその状態の番号を push する。

これらの動作は次のような構文解析表で表される。

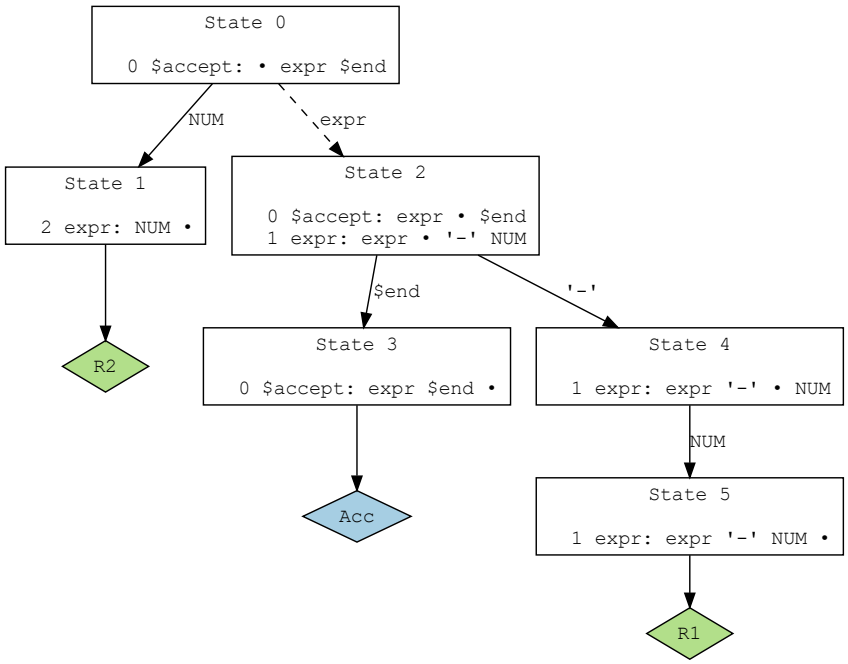
表 1　構文解析表の例

状態	動作			行先
	NUM	−	$	expr
0	s1			2
1	r2	r2	r2	
2		s4	s3	
3	acc	acc	acc	
4	s5			
5	r1	r1	r1	

これは、次の文法規則から bison が生成したものである。

```
expr := expr − NUM
expr := NUM
```

例えば、状態 0 で NUM が入力されたとき、シフト動作 s1 を、状態 1 では任意の入力に対して、還元動作 r2 を実行する。還元動作では左辺の値が expr であり、その結果、スタックのトップが 0 の時に状態 2 に遷移する。acc は受理して解析を終了すること、空欄はエラーで解析を中止することを表す。



これらの動作は図のように表される。ここで、長方形の State n と表されるのがオートマトンの状態である。またひし形の Rk で還元動作を表す。実線の矢印は、シフト動作の行先または還元動作を表す。矢印にそえて入力記号が書かれている。一方、破線の矢印は還元動作の後に遷移する状態を表す。この矢印には文法規則の左辺が書かれている。