第1章 演算子順位による構文解析 (教科書 p.40)

1.1 構文解析とは（復習）

構文解析とはプログラム（式）の構造を木の形に表すことである。

正規言語の反復補題（ポンプの補題）からわかるように、正規表現 (DFA) では構文解析はできない。

例:

```
gokei = soryo + kosu * tanka
               ↑      ↑      ↑
               Ⓐ      Ⓑ      Ⓒ
```

____

1.2 演算子順位法のアイデア

左から右に読む

Ⓐの地点 … 「=」の後ろに「+」があるのでオペランドの決定を保留する
Ⓑの地点 … 「+」の後ろに「*」があるのでオペランドの決定を保留する
Ⓒの地点 … 「*」の後ろにもう演算子がないのでオペランドを確定する

ここから、以下のアイデアがでてくる

- 演算子の____・____の情報を使う
- データを保留しておく必要がある → スタックを用いる
  左の演算子 < 右の演算子 → ____
  左の演算子 > 右の演算子 → ____

1.3 先人の知恵

- 演算子の関係は左右非対称で良い
  （x < y でも y > x とは限らない） ← 「<」,「>」,「=」の代わりに「__」,「__」,「__」と書く

- 始・終・識別子・かっこなども演算子と同様に ⋖, ⋗, ≐ の関係をつける
(教科書 p.45 表4.2)

左＼右	+	-	*	/	(	)	識別子	終	
⋮					⋱				
(					…				
)	⋗	⋗	⋗	⋗		⋗		⋗	この行を追加
識別子					…				

始・終をともに「＿」と書く

1.4 演算子順位法の例

教科書 p.45 表4.2 の表による演算子順位法で構文解析する。元の BNF は次のようになる。（曖昧な文法である。）

$$E \to E + E \mid E * E \mid \mathbf{id} \mid (E)$$

入力例として　$a+b*(c+d)$ を考える。（ただし、a ～ d は **id** （識別子）に属するトークンである。）

各動作では、スタックの一番上と入力の残りの先頭を比べている。このとき、スタックに非終端記号（この例の場合は E）が入っていても無視する。

スタック	入力の残り	動作	補足
		だから	
		だから	
		だから	
		だから	
		だから	
		だから	
		だから	
		だから	
		だから	
		だから	
		だから	
		だから	
		だから	
		だから	
		だから (※)	
		だから	

		だから	
		だから	

シフト (shift)　　…
還元 (reduce)　　…

補足「≐」は何のためにある？
「⋖」「≐」ともに動作はシフトである。ただし、あとで「⋗」になって還元するときに「≐」を通り越して「⋖」のところまでポップする（２つ以上の終端記号をポップする）。上の※のところ参照。

還元が起こった箇所を下から読むと、

という導出列になっている。

1.5 演算子順位法の特徴

- 最も右側の非終端記号を書き換える（　　　　）
- 解析木の葉から幹へ向かって節を作っていく （　　　　　　）

- （ほとんど）どんなプログラミング言語でもコードを書ける
- 実行時にも構文解析表を更新できる(他の構文解析法の後処理に使える)

1.6 演算子順位法の制限

スタックの終端記号を使うため、右辺に

- εがでてくる、終端記号が出現しない

- あるいは、非終端記号が隣り合う

ような生成規則があると対応できないことが知られている。

1.7 演算子順位行列 (p.45 表 4.2 など) の作り方

1. ⊗ が ⊕ より優先順位が高いとき
   → [⊗　⊕] かつ [⊕　⊗] とする

2. ⊕ と ⊖ が同じ優先順位のとき
   左結合 → [⊕　⊖] かつ [⊖　⊕]、
   右結合 → [⊕　⊖] かつ [⊖　⊕]、
   非結合 → 空欄のまま（エラーを表す）とする

3. すべての演算子 ⊕ について、「**id**」、「(」、「)」、「$」 との関係は、表 4.2 と同じである。

   つまり、[⊕　**id**] かつ [**id**　⊕] かつ [⊕　(] かつ [(　⊕] かつ [)　⊕] かつ [⊕　)] かつ [$　⊕] かつ [⊕　$] とする

4. さらに「**id**」、「(」、「)」、「$」 同士の関係は、表 4.2 と同じである。

   つまり、[(　)] かつ [(　(] かつ [)　)] かつ [$　$] などなど、とする

注： 単項演算子の「−」については、字句解析のときに二項演算子の「−」と区別しておく必要がある

問 1.7.1 教科書 p.45 表 4.2 に累乗演算子「^」と比較演算子「<」を追加せよ。ただし「^」は右結合で「*」や「/」よりも、優先順位が高いものとする。また「<」は非結合（例えば a < b < c は、構文エラー）で、「+」や「−」よりも、優先順位が低いものとする。

左＼右	<	+	−	*	/	^	(	)	識別子	終
始										
<										
+										
−										
*										
/										
^										
(										
)										
識別子										

白地の部分は、教科書 p.45 の表 4.2 から書き写し、黄色地のところを考えよ。

注: 通常はエラーは空欄のままとするが、この問では解答なしと区別するために明示的に「X」を書け。

1.8 演算子順位法の応用

演算子順位法による構文解析では、結局、計算する順番で解析木を生成するので、________を生成するのに利用することができる (p.45)。逆ポーランド記法は ________(RPN)、________(postfix notation) とも呼ばれる。演算子をオペランドの後に書く。そのため（二項演算子しかない場合は）かっこが必要ない。

通常（中置記法）	RPN
12 + 34 * 56	
(12 + 34) * 56	

機械語と順序が同じであるため、式に対する（簡易）________と考えることができる

演算子順位法では、還元が起こったときに、含まれている（かっこ以外の）終端記号を出力すれば良い

例 `$a+b*(c+d)$`（さっきの入力例）

問 1.8.1 下の 2 つの入力例

1. `$a+b*c$`
2. `$(a+b)*c$`

について、教科書 p.45 表4.2 の表による演算子順位法が、さっきの入力例と同様に逆ポーランド記法を出力することを確認せよ

```
 1  /* ****************************************************************
 2   * 演算子順位法による構文解析
 3   *
 4   * 1+2*3 のような式を構文解析して、計算結果(この場合 7)を出力する
 5   *          (構文木のデータ構造はつくらない)
 6   * 表 (prec_table と op_index)と、
 7   * 還元規則 (reduce の補助関数の binary_op)を書き換えて
 8   * 使用してください。
 9   * **************************************************************** */
10
11  /* マクロの定義 --- 終端記号・非終端記号を表す定数を定義する */
12  /* ここからは終端記号、        */
13  #define BGN 256          /* 始 */
14  #define END 257          /* 終 */
15  #define NUM 258          /* 数値 */
16  #define TERM_MAX 258
17  /* ここからは非終端記号の定義 */
18  #define Expr 259
19
20  /* 字句解析部が返す ``属性'' (yylval) の型
21   *  Yacc (Bison)と同じ形式にする。           */
22  typedef double YYSTYPE;  /* 使用するトークンの属性の型に応じて変更する。*/
23  extern YYSTYPE yylval;
24  /* 字句解析部に flex が生成する関数を用いる場合は、ここまでをヘッダーファ
25   * イルとして分離する。                                            */
26
27  #include <stdio.h>
28  #include <stdlib.h>
29  #include <ctype.h>
30
31  YYSTYPE yylval;
32
33  /* **************************************************************** *
34   * スタックの実装
35   * **************************************************************** */
36
37  struct _elem { /* スタックの要素の型 */
38    int token;   /* トークンの種類、259 以上は非終端記号 */
39    YYSTYPE val; /* 属性値 */
40  };
41
42  typedef struct _elem elem;
43
44  elem stack[64]; /* トイプログラムなのでとりあえずスタックの大きさは 64 で十分 */
45
46  elem* sp = stack; /* 大域変数: スタックポインタ */
47
48  void push(int tok, YYSTYPE attr) { /* スタックにプッシュする。 */
49    sp->token = tok;
50    sp->val   = attr;
51    sp++;   /* スタックは下に伸びることに注意 */
52  }
53
54  elem pop(void) { /* スタックをポップする。 */
55    if (sp == stack) {
56      printf("スタックが空です。\n");
57      return *sp;
58    } else {
59      sp--;
```

```
60        return *sp;
61      }
62    }
63
64    void clear_stack(void) {
65      sp = stack;
66    }
67
68    elem* topmost_token_aux(elem* ptr) { /* topmost_token の補助関数 */
69      while (ptr->token > TERM_MAX) { /* ptr は非終端記号を指す */
70        ptr--;
71        if (ptr < stack) {
72          printf("エラー: スタックには終端記号が入っていません。\n");
73          return stack;
74        }
75      }
76      /* ptr->token <= TERM_MAX */
77      return ptr;
78    }
79
80    elem* topmost_token(void) { /* スタックの先頭の終端記号 */
81      return topmost_token_aux(sp - 1);
82    }
83
84    void debug_token(int t, YYSTYPE v) {
85        switch (t) {
86        case BGN:  printf("BGN"); break;
87        case END:  printf("END"); break;
88        case NUM:  printf("NUM_(%.3f)", v); break;
89        case Expr: printf("Expr_(%.3f)", v); break;
90        default:
91          printf("'%c'", t); break;
92        }
93    }
94
95    void debug_stack(void) { /* デバッグ用: スタックの中身を出力する */
96      elem* sp0;
97
98      printf("スタック 底 <<");
99      for (sp0 = stack; sp0 < sp; sp0++) {
100           int t =      sp0->token;
101           YYSTYPE v = sp0->val;
102
103           printf(" | ");
104           debug_token(t, v);
105     }
106     printf(" >> 上\n");
107   }
108
109   /* ************************************************************** *
110    * 字句解析部 (flex が生成する関数に置き換えても良い。)
111    * ************************************************************** */
112
113   int yylex(void) { /* 入力の次のトークンを返す。 */
114     int c;
115
116     do {
117       c = getchar();
118     } while (c == ' ' || c == '\t');  /* 空白を読みとばす */
```

```
119
120     if (isdigit(c) || c == '.') {
121       ungetc(c, stdin);
122       scanf("%lf", &yylval);
123       /* ``値''は yylval という変数に代入して返す。*/
124       return NUM;
125       /* NUMというトークンを返す。*/
126     } else if (c == '\n') {
127       return END; /* 終りの記号 */
128     } else if (c == EOF) {
129       exit(0);   /* プログラムの終了 */
130     }
131     /* 上のどの条件にも合わなければ、文字をそのまま返す。*/
132     return c;
133   }
134
135   /* ************************************************************** *
136    * 構文解析部
137    *
138    *    Expr -> NUM
139    *          | '(' Expr ')'
140    *          | Expr '+' Expr
141    *          | Expr '*' Expr
142    * ************************************************************** */
143
144   /* ************************************************************** *
145    * 演算子順位表の表現      必要に応じて変更する
146    * ************************************************************** */
147   #define LT 0     /* <, Less Than */
148   #define EQ 1     /* =, Equal */
149   #define GT 2     /* >, Greater Than */
150   #define ERR 3    /* エラー, Error */
151
152   int op_index(elem* p) { /* 表を引きやすいように連続した数値に写す。*/
153     switch (p->token) {
154     case BGN:  return 0;
155     case '+':  return 1;
156     case '*':  return 2;
157     case '(':  return 3;
158     case ')':  return 4;
159     case NUM:  return 5;
160     case END:  return 6;
161     default:   printf("op_index: 不正な構文要素 (");
162                debug_token(p->token, p->val);
163                printf(")。\n");
164                exit(1);
165                return 0;
166     }
167   }
168
169   char prec_table[6][6] = { /* 演算子順位表本体 */
170     /* 行に END がないこと、列に BGN がないことに注意。*/
171             /* '+',  '*',  '(',  ')',  NUM,  END */
172     /* BGN */ {LT,   LT,   LT,   ERR,  LT,   EQ   },
173     /* '+' */ {GT,   LT,   LT,   GT,   LT,   GT   },
174     /* '*' */ {GT,   GT,   LT,   GT,   LT,   GT   },
175     /* '(' */ {LT,   LT,   LT,   EQ,   LT,   ERR  },
176     /* ')' */ {GT,   GT,   ERR,  GT,   ERR,  GT   },
177     /* NUM */ {GT,   GT,   ERR,  GT,   ERR,  GT   },
```

```
178  };
179
180
181  /* 演算子順位表を利用する補助関数 */
182  int prec(elem* left, elem* right) {
183    /* left と right の関係を prec_table から引く。 */
184    return prec_table[op_index(left)][op_index(right) - 1];
185  }
186
187  elem* handle_left(void) { /* 還元が起こる記号の列の左端を見つける */
188    elem* next;
189    elem* cur = topmost_token(); /* スタックのトップの終端記号の位置 */
190
191    while (1) {
192      next = topmost_token_aux(cur - 1); /* 次の終端記号の位置 */
193      if (prec(next, cur) == LT) {
194        return next + 1; /* next の手前が求める場所 */
195      } else { /* EQ */
196        cur = next;
197      }
198    }
199  }
200
201  /* ************************************************************** *
202   * 構文規則の表現   必要に応じて変更する
203   * ************************************************************** */
204
205  YYSTYPE binary_op(YYSTYPE left, int op, YYSTYPE right) {
206     printf("   還元: Expr -> Expr "); debug_token(op, 0); printf(" Expr\n");
207     switch (op) {
208     case '+':
209       return left + right;
210     case '*':
211       return left * right;
212     default:
213       printf("binary_op: 処理できない二項演算子: "); debug_token(op, 0); printf("\n");
214       exit(7);
215       return 0;
216     }
217  }
218
219  int reduce(void) {  /* 還元処理 */
220    elem* left = handle_left();    /* 還元する部分の左端を見つける */
221    int num = sp - left;           /* 還元する記号列の長さ */
222    /* printf("reduce:\t"); */
223
224    switch (num) {                 /* どの規則で還元するか? */
225    case 1: {
226      elem data = pop();
227      if (data.token == NUM) {
228        /* Expr -> NUM */
229        printf("   還元: Expr -> NUM_(%.3f)\n", data.val);
230        push(Expr, data.val);         /* ポップしてすぐプッシュ */
231        break;
232      } else {
233        printf("reduce: 不正なオペランド (");
234        debug_token(data.token, data.val);
235        printf(")。\n");
236        exit(2);
```

```
237        }
238      }
239      case 2: {
240        elem data2 = pop(); elem data1 = pop();
241        printf("reduce: 不正な式 (");
242        debug_token(data1.token, data1.val);
243        printf(",");
244        debug_token(data2.token, data2.val);
245        printf(")。\n");
246        exit(3);
247      }
248      case 3: {
249        elem data3 = pop(); elem data2 = pop(); elem data1 = pop();
250        if (data1.token == '(' && data2.token == Expr && data3.token == ')') {
251          /* Expr -> '(' Expr ')' */
252          printf("   還元: Expr -> ( Expr )\n");
253          yylval = data2.val;
254          push(Expr, yylval);
255        } else if (data1.token == Expr && data3.token == Expr) {
256          /* 二項演算子 */
257          yylval = binary_op(data1.val, data2.token, data3.val);
258          push(Expr, yylval);
259        } else {
260          printf("reduce: 不正な式 (");
261          debug_token(data1.token, data1.val);
262          printf(",");
263          debug_token(data2.token, data2.val);
264          printf(",");
265          debug_token(data3.token, data3.val);
266          printf(")。\n");
267          exit(4);
268        }
269        break;
270      }
271      default:
272        printf("reduce: 構文エラー\n");
273        exit(5);
274      }
275      debug_stack();
276      return 0;
277    }
278
279    /* ****************************************************************** *
280     * 構文解析関数本体
281     * ****************************************************************** */
282    int yyparse(void) {
283      elem* top;
284      elem next;
285      char relation; /* 関係 */
286
287      push(BGN, 0 /* 0 はダミー */); /* 始記号をスタックに積んでおく */
288      next.token = yylex(); /* 入力の最初のトークン */
289      next.val   = yylval;
290      debug_stack();
291      while (1) {
292        top = topmost_token();   /* スタックのトップの終端記号 */
293
294        printf("   ");
295        debug_token(top->token, top->val);
```

```
296     printf(" と ");
297     debug_token(next.token, next.val);
298     printf(" を比較: ");
299 
300     if (next.token == END && top->token == BGN) {
301       printf("   シフト\n"); /* デバッグ用 */
302       push(next.token, 0 /* ダミー */);
303       debug_stack();
304       printf("   終了\n"); /* デバッグ用 */
305       clear_stack();
306       return 0; /* 成功で終了 */
307     }
308 
309     relation = prec(top, &next);
310     if (relation == LT || relation == EQ) {      /* シフト */
311       printf("   シフト\n"); /* デバッグ用 */
312       push(next.token, next.val);
313       debug_stack();
314       next.token = yylex();     /* 次のトークンを読み込む */
315       next.val   = yylval;
316       /* printf ("\ntoken=%d\n", next.token); *//* デバッグ用 */
317     } else if (relation == GT) {                /* 還元 */
318       if (reduce()) { /* 0 以外は構文エラー */
319         return 1;
320       }
321     } else { /* 表の空欄部分 --- エラー */
322       printf("yyparse: 不正な先読み (");
323       debug_token(next.token, next.val);
324       printf(")。\n");
325       exit(6);
326     }
327   }
328 }
329 
330 int main(void) {
331   while (1) {
332     if (yyparse() == 0) { /* 0 は正常終了 */
333       printf("   答: %g\n\n", yylval);
334     }
335   }
336 
337   return 0;
338 }
339 
```

第2章 LR 構文解析 (教科書 p.72)

2.1 LR 構文解析の特徴

- パーサーの＿＿＿＿＿＿＿＿＿＿（一方、人手での生成には向かない）
  → Yacc, Bison などの構文解析器生成系
- 取り扱える文法の範囲が広い
- 演算子順位法と同様に、＿＿＿＿・＿＿＿＿＿＿である
  （“LR” は Left-to-Right Rightmost derivation に由来する）
  - スタックを用いるのは同じ
  - シフト／還元の判断法がちがう（Bison のプログラム ～`.y` をデバッグするときに判断法の知識が必要になる）
    ＿＿を用いる （ただしスタックの方に!!）
    この DFA は直観的には BNF の右辺のどこまで処理しているか? を表す

LR 構文解析の例

例 1

$$
\begin{aligned}
S' &\to S\ \$ \\
S &\to (\ L\ ) \\
&\mid\ \mathtt{x} \\
L &\to S \\
&\mid\ L\ ,\ S
\end{aligned}
$$

注: LL 法（後述）は左再帰は扱えないが、LR 法は左再帰の文法のほうが効率が良い。

> **問 2.1.1 （復習）** この BNF の S から導出される終端記号列の例を導出列とともに 3 つ挙げよ
>
> ---
>
> ---

この BNF に対応する DFA は以下のようになる（ただし、DFA の作成法は講義の範囲外である）。

⓪
S’→ . S $
S→ . (L)
S→ . x

①
S→ x .

②
S→ (. L)
L→ . S
L→ . L , S
S→ . (L)
S→ . x

③
S’→ S . $

④
S→ (L .)
L→ L . , S

⑤
L→ S .

⑥
S→ (L) .

⑦
L→ L , . S
S→ . (L)
S→ . x

⑧
L→ L , S .

⑨
S’→ S $.

Transitions: ⓪ –x→ ①, ⓪ –(→ ②, ⓪ –S→ ③, ② –x→ ①, ② –(→ ②, ② –S→ ⑤, ② –L→ ④, ③ –$→ ⑨, ④ –,→ ⑦, ④ –)→ ⑥, ⑦ –x→ ①, ⑦ –(→ ②, ⑦ –S→ ⑧

入力例: ((x , x) , x) $

スタック	入力	状態
		※
		※
		※
		※
		※
		※
		※
		※
		※

※ … スタックをもう一度 DFA にかける
実際にはスタック全体に繰り返し DFA を適用する必要はない。下の表のように
スタックに状態も積んでおくと良い

スタック	入力
⋮	

問 2.1.2 例 1 の BNF に対して、以下の入力例に対する LR 構文解析の過程を書け。

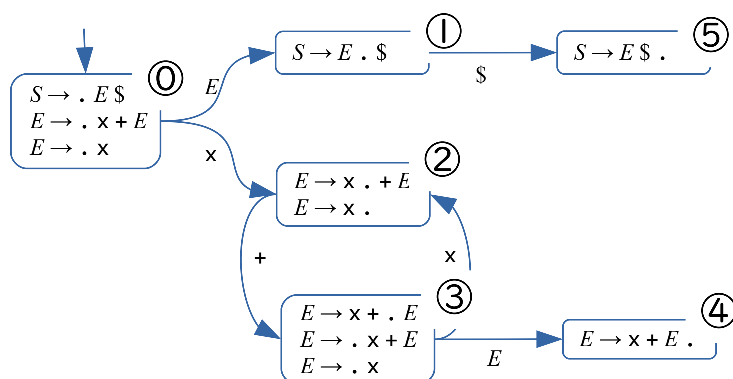
1. ((x), x)$
2. (x, (x))$

長くなるようならば、5 回還元 (reduce) した時点で止めて良い。

例 2

一般にシフト／還元の判定には“先読み”（入力の先頭）も用いる

$$
\begin{array}{rll}
S \to E\,\$ & (1) & \\
E \to \mathbf{x} + E & (2) & \text{右再帰になっている} \\
\mid \ \mathbf{x} & (3) &
\end{array}
$$



状態②では 先読みが + → shift
先読みが + 以外 → reduce

LR 構文解析の階層

先読みの利用の仕方によって、以下のような階層がある

____ > ______ > _____ > ____
例 2 例 1

それぞれ、以下のような特徴がある

LR(1) … ○ 扱える文法の幅が広い
× DFA の状態数が多くなりすぎる

LALR(1) … LookAhead LR
現実的な選択肢である
Yacc, Bison はこれに基づく ← LL(1) よりは（通常）はるかに広い

SLR(1) … Simple LR
× 扱える文法の幅が狭い
○ DFA の状態数は少ない

LR 構文解析表

例 2 に対して LR 構文解析表は次のようになる（表の作り方は講義の範囲外である）

状態＼先読み	x	+	$		*E*
⓪	shift②				goto①
①			shift⑤		
②	reduce(3)	shift③	reduce(3)		
③	shift②				goto④
④		reduce(2)	reduce(2)		
⑤	accept	accept	accept		

説明 shift② … shift して状態②へ
reduce(2) … 生成規則(2) を使って reduce
goto① … 状態①へ

入力例 x + x + x $

スタック	入力	説明
____	____	____
____	____	____
____	____	____
____	____	____
____	____	____
____	____	____, ____
____	____	____, ____
____	____	____, ____
____	____	____
____	____	____

問 2.1.3 次の BNF に対して LR 構文解析表を作成すると、下のようになる。 S', S, T は非終端記号であり、;, w, {, } は終端記号である。

$$
\begin{array}{rll}
S' \to & S\ \$ & (0) \\
S \to & ; & (1) \\
| & \{\ T\ \} & (2) \\
| & \mathtt{w}\ S & (3) \\
T \to & S\ T & (4) \\
| & \varepsilon & (5)
\end{array}
$$

状態＼先読み	;	{	w	}	$		*S*	*T*
⓪	shift①	shift②	shift③				goto④	
①	reduce(1)							
②	shift①	shift②	shift③	reduce(5)			goto⑤	goto⑥
③	shift①	shift②	shift③				goto⑦	
④					shift⑧			
⑤	shift①	shift②	shift③	reduce(5)			goto⑤	goto⑨
⑥				shift⑩				
⑦	reduce(3)							
⑧	accept							
⑨	reduce(4)							
⑩	reduce(2)							

以下の入力例に対する LR 構文解析の過程を書け。

1. {;w;}$
2. {w{;};}$

長くなるようならば、5 回還元 (reduce) した時点で止めてよい。

LR 構文解析と曖昧な文法

曖昧な文法に対して無理に LR 構文解析表を作ると＿＿＿(conflict) が起こる（つまり、表の 1 つの場所に複数の動作が入る）

例 3

$$E \to \mathtt{x} \mid E * E \mid E + E$$

＿＿＿＿＿＿＿＿が起こる。

例　x + x * x

Yacc (Bison) では演算子の＿＿＿＿・＿＿＿を指定して conflict を解消できる

例 4（＿＿＿＿＿＿＿＿, dangling else）

$$
\begin{array}{rl}
S \to & \textbf{if}\ (\ E\ )\ S \\
| & \textbf{if}\ (\ E\ )\ S\ \textbf{else}\ S \\
| & \ldots
\end{array}
$$

例 `if ( E`$_1$` ) if ( E`$_2$` ) S`$_1$` else S`$_2$ ← これは

`if ( E`$_1$` ) {if ( E`$_2$` ) S`$_1$`} else S`$_2$ と解釈するのか？

`if ( E`$_1$` ) {if ( E`$_2$` ) S`$_1$` else S`$_2$`}` と解釈するのか？

__________が起こる。 Yacc (Bison) では ____を採用する

例 5（特殊例の優先）

$$E \rightarrow E \,\hat{}\, E \,_\, E \mid E \,\hat{}\, E \mid E \,_\, E \mid \mathbf{id}$$

（優先順位・結合性を与えても）__________が起こる

例 `x ^ y _ z $`

Yacc (Bison) では先に書かれている生成規則を優先する

これらの例 3 〜 5 はよく知られている形だが、このような形以外の conflict は文法を見直す

Bison について

Bison は BNF とそれに対する動作記述から、 C 言語の構文解析系（パーサー）を自動生成するプログラムです。 Bison のソースファイル（通常 .y という拡張子をつける）は、次のように書きます。（これは通常の四則演算の式の文法です。この例では単項の「-」はサポートしていないので、 -1+2 や 2*(-3) のような式は構文解析できません。単項の「-」を扱う方法は yacc (bison) のマニュアルを調べて下さい。）

ファイル calc.y

```
 1 %{
 2  /* C declarations */
 3 #define YYSTYPE double
 4 #include <stdio.h>
 5 #include <stdlib.h>
 6 #include <ctype.h>
 7 void yyerror(char* s) {
 8    printf("%s\n", s);
 9 }
10 
11 int yylex(void);
12 %}
13  /* Bison declarations */
14 %token NUMBER
15 %left '+' '-'
16 %left '*' '/'
17 
18 %%
19  /* grammar rules */
20 input    :                {}
21          | input line     {}
22          ;
23 line     : '\n'           { exit(0); }
24          | expr '\n'      { printf("\t%g\n", $1); }
25          ;
26 expr     : NUMBER         { $$ = $1; }
27          | expr '+' expr { $$ = $1 + $3; }
28          | expr '-' expr { $$ = $1 - $3; }
29          | expr '*' expr { $$ = $1 * $3; }
30          | expr '/' expr { $$ = $1 / $3; }
31          | '(' expr ')'   { $$ = $2; }
32          ;
33 %%
34  /* additional C code */
35 int yylex(void) {
36   int c;
37 
38   do {
39     c = getchar();
40   } while (c == ' ' || c == '\t');
41 
42   if (isdigit(c) || c == '.') {
43     ungetc(c, stdin);
44     scanf("%lf", &yylval);
45     return NUMBER;
```

```
46      } else if (c == EOF) {
47        return 0;
48      }
49      return c;
50  }
51
52  int main(void) {
53      printf("Exit with Ctrl-c\n");
54      yyparse();
55      return 0;
56  }
```

説明

最初の C 宣言部（C declarations というコメントの部分、「%{」と「%}」の間）には後述の動作記述の中で用いる関数の定義や宣言を書きます。 特に YYSTYPE という定数マクロには属性値（意味値）の型を指定します。ただし、属性値の型が int の場合はこのマクロの定義は必要ありません。

この例では exit 関数や isdigit 関数を使用するため、それぞれ stdlib.h, ctype.h というヘッダーをインクルードしています。また、エラーがあったときに呼ばれる関数として void yyerror(char* s) を定義しておきます。（この例では単に printf を呼び出しています。） また、yylex 関数は下で定義しているため、ここでプロトタイプ宣言してあります。

Bison 宣言部（Bison declarations というコメントの部分、最初の「%%」まで）には終端記号（トークン）（と非終端記号）に関する宣言を書きます。 %token はトークンを宣言します。トークンは出力の C プログラムでは定数マクロとして定義されます。 **下の優先順位の宣言や文法規則部でトークンとして使えるのはここで定義したマクロか文字リテラルです。**

Bison 宣言部には演算子の優先順位と結合性を宣言することもできます。 %left, %right, %nonassoc のいずれかのあとに演算子を表すトークンを空白で区切って並べて書きます。%left は左結合、%right は右結合、%nonassoc は非結合を表します。また、優先順位が高い演算子ほど下に書きます。同じ行に書かれている演算子は優先順位は同じです。上の例では、「+」「-」「*」「/」はすべて左結合で、「*」/」が「+」「-」よりも優先順位が高い、と宣言されています。

文法規則部（grammar rules というコメントの部分）は Bison の核心部分で、最初の「%%」から 2 つめの「%%」までが文法規則部です。 文法規則部には BNF とそれに付随するアクションを書きます。 BNF は教科書などでよく使われる記法の右矢印「→」の代わりにコロン「:」を使っていることに注意してください。また各 BNF の最後にセミコロン「;」を書きます。

特に指定しなければ開始記号 (start symbol) に関する BNF を最初に書きます。

アクションは**還元時**に実行されるプログラムのことです。アクションの中身は通常**属性値**（意味値）の計算です。属性値は解析木の各節（枝分れの部分）に関連付けられる“値”です。

生成規則の中では、終端記号（トークン）は、１文字からなるトークンの場合は通常、**文字リテラル**そのまま、２文字以上からなるトークンの場合は %token で宣言された**マクロ**で表します。 Flex で生成される yylex 関数はトークン（文字リテラルまたはマクロ）を返します。

終端記号（トークン）の属性値は、字句解析器（yylex 関数）から **yylval** という大域変数に代入されて受け渡されます。

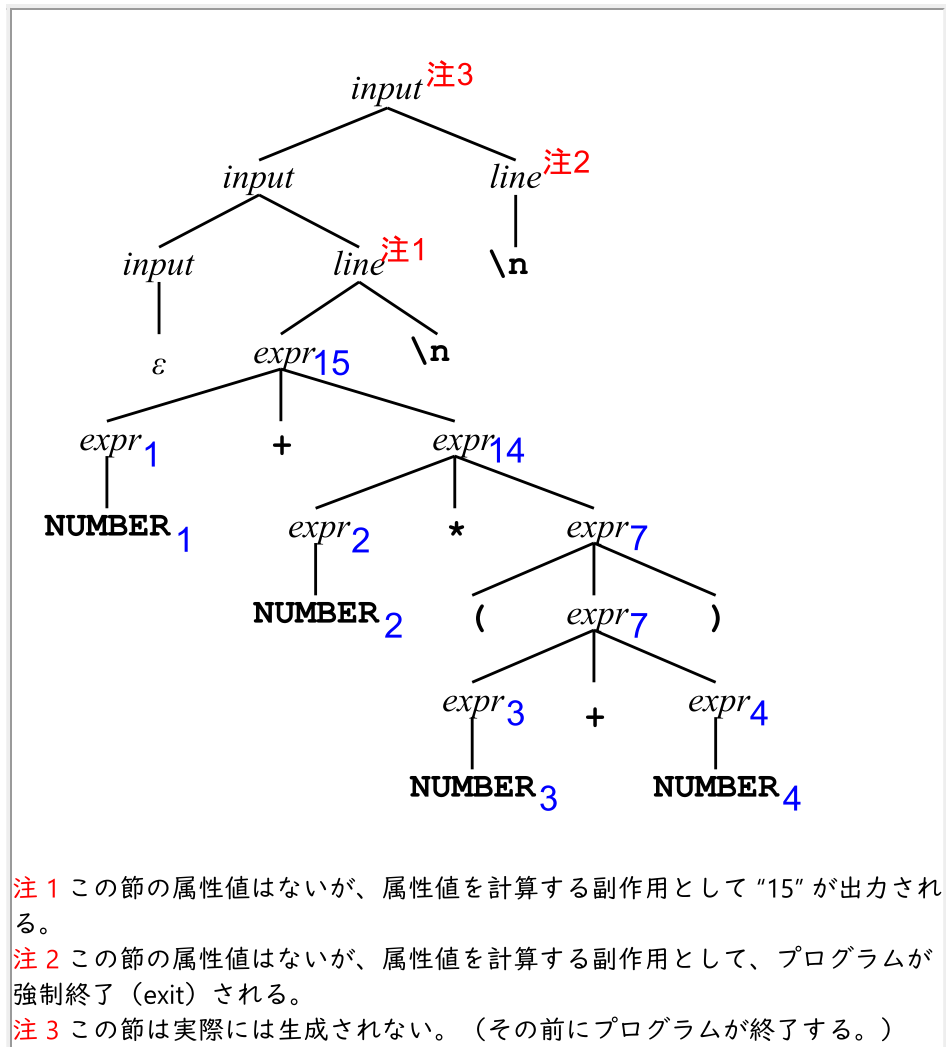
例えば、入力が“(12+34)*56\n”という文字列の場合、上の例の yylex() の戻り値とそのときの yylval の値は次のようになります。

	1回目	2回目	3回目	4回目	5回目	6回目	7回目	8回目
yylex()	'('	NUMBER	'+'	NUMBER	')'	'*'	NUMBER	'\n'
yylval		12.0		34.0			56.0	

還元時（つまり、解析木の節を作るとき）に対応するアクションが実行されます。

非終端記号の属性値は、各部分木の属性値から計算されます。 **$$** が還元される生成式の左辺の属性値、 **$1**, **$2**, ... が右辺の１番目、2 番目、... の文法要素の属性値を表します。例えば、 `$$ = $1 * $3` というアクションでは、１番目と 3 番目の部分木の属性値の積が、節の非終端記号の属性値となります。

例えば、 `1 + 2 * ( 3 + 4 ) \n \n` というトークン列から、上の Bison プログラムは以下のような解析木を生成します。 青字で示されているのが各節の属性値です。



注 1 この節の属性値はないが、属性値を計算する副作用として "15" が出力される。
注 2 この節の属性値はないが、属性値を計算する副作用として、プログラムが強制終了（exit）される。
注 3 この節は実際には生成されない。（その前にプログラムが終了する。）

追加の C プログラム部（addtional C code というコメントの部分）は 2 つめの「`%%`」からファイルの最後までです。ここは文字通り追加の C プログラムを書きます。この部分は C のプログラムの末尾にそのままコピーされます。

この例では `yylex` と `main` 関数を定義しています。普通は `yylex` 関数は Flex などで定義しますが、この例では説明のため Bison プログラム中で定義しています。ここで `yylex` 関数の戻り値はトークンです。すなわち文字コードか、`%token` で定義した定数マクロ（この例では `NUMBER`）です。また、トークンの属性値（意味値）は `yylval` という大域変数に代入して返しています。マクロ `YYSTYPE` は、この `yylval` の型を表します。ファイルの終わりに到達するなど、これ以上トークンがないときは、 `yylex` 関数は 0 を返します。

また、この例では `main` 関数は Bison のプログラム中で用意していますが、通常は別の C のソースファイルに定義します。この例の `main` 関数は、単に `yyparse` を呼び出すだけです。この `yyparse` は、上の文法規則部から Bison が生成する関数です。

生成

このファイル（ファイル名を calc.y とする）から C ソースファイルを生成するには

```
bison calc.y
```

というコマンドを実行します。これで calc.tab.c という名前（.y ファイルの名前の後ろに .tab がつく）の C ソースファイルができます。また、 -o というオプションで、 C のファイル名を指定することができます。例えば、

```
bison -ocalc.c calc.y
```

で calc.c という名前の C ソースファイルができます。

この例の場合は、この C ソースファイルを普通にコンパイルすると、（警告 (Warning) がいくつか出ますが）実行可能ファイルができます。

Microsoft Visual Studio の場合は、

```
cl calc.c
```

GCC の場合は、

```
gcc calc.c
```

次のコマンドで実行できます。

```
.\calc
```

Flex と Bison を同時に使う

Flex と Bison を併用するとき、Flex から生成される関数 (yylex) は、トークン（終端記号） の情報を戻り値とし、それを Bison が利用します。トークンが 1 文字の場合は、通常、戻り値は文字リテラルそのものです。トークンが 2 文字以上の場合は、Bison で %token により宣言されたマクロを使用します。

ここでは 2文字以上の演算子の例として、「*+」という演算子を x *+ y が x * 256 + y を表し、「+」「–」と同じ優先順位を持つ、左結合の演算子として導入します。仮に FOO 演算子と呼ぶことにします.

Flex のソースファイル（mylexer.l）には、C 定義部に次のような #include 文を入れておきます。インクルードされるファイル（myparser.h） は Bison が生成するファイルで、bison を実行するときに与えるオプションで指定した C ソースファイルの名前の .c を .h に変えたものです。ここに %token により宣言されたマクロの定義が書かれています。

ファイル mylexer.l

```
 1 %{
 2     /* C definitions */
 3 void yyerror(char*);
 4 
 5 #define YY_SKIP_YYWRAP
 6 #define YYSTYPE double
 7 int yywrap(void) { return 1; }
 8 #include "myparser.h" /* cf. bison's -o option */
 9 %}
10     /* definitions */
11 %option yylineno
12 %option always-interactive
13 %%
14     /* rules */
15 [ \t]+              { /* do nothing */ }
16 [0-9]+(\.[0-9]+)?(E[+\-]?[0-9]+)?    {
17    sscanf(yytext, "%lf", &yylval); return NUMBER;
18                     }
19 [+\-*/()\n]         { return yytext[0]; }
20 "*+"                { return FOO; }
21 .                   {
22    yyerror("Illegal character."); return '\n';
23 }
24 %%
25     /* additional C code */
```

動作の中に return 文を入れておくと、その式の値が Flex の生成する yylex 関数の戻り値になります。 yylex 関数は呼び出されるたびに、次のトークンを返します。

トークンは、 1 文字からなるトークンの場合は通常、文字コードそのまま、 2 文字以上からなるトークンの場合は %token で宣言されたマクロです。

トークンの“種類”（NUMBER など）を yylex 関数の値として返し、値（“属性”）を yylval という大域変数に代入していることに注意します。これが通常の yylex 関数の書き方です。

この例では [0-9]+(\.[0-9]+)?(E[+\-]?[0-9]+)? という正規表現にマッチする文字列があれば、NUMBER というトークンを yylex の戻り値として返します。そのときの属性値は yylval という大域変数に代入されています。

一般に正規表現にマッチした文字は、 yytext という配列に保持されています。また yyleng という変数にマッチした文字の数が保持されています。だからマッチした文字は一般に yytext[0] ～ yytext[yyleng - 1] ということになります。（通常の C 言語の文字列とは異なり、最後（yytext[yyleng]）にナル文字 '\0' は入っていないので注意が必要です。）

この例では、+, -, *, /, (,), =, \n のいずれかの文字が現れたときは、その文字コードを返します。

例えば、入力が“(12*+34)*56\n”という文字列の場合、 yylex() の戻り値とそのときの yylval の値は次のようになります。

	1回目	2回目	3回目	4回目	5回目	6回目	7回目	8回目
yylex()	'('	NUMBER	FOO	NUMBER	')'	'*'	NUMBER	'\n'
yylval		12.0		34.0			56.0	

Bison のソースファイル（myparser.y）の方は、単独で使う場合とあまり変わりませんが、 yylex 関数は Flex の方で用意するので C 宣言部でプロトタイプ宣言だけしておきます。（この例では Bison のソースファイルに yylex の定義を書いてはいけません。）

ファイル myparser.y

```
 1 %{
 2  /* C declarations */
 3 #define YYSTYPE double
 4 #include <stdio.h>
 5 #include <stdlib.h>
 6
 7 void yyerror(char* s) {
 8    printf("%s\n", s);
 9 }
10
11 int yylex(void); /* prototype declaration */
12 %}
13 /* Bison declarations */
14 %token NUMBER
15 %token FOO
16 /* %left, %right or %nonassoc */
17 /* lower precedence */
18 %left '+' '-' FOO
19 %left '*' '/'
20 /* higher precedence */
21 %%
22 input :    /* empty */
```

```
23          | input line     {}
24          ;
25 line     : '\n'        { exit(0); }   /* an empty line */
26          | expr '\n' { printf("\t%g\n", $1); }
27          ;
28 expr  :  NUMBER          { $$ = $1; }
29          | expr FOO expr { $$ = $1 * 256 + $3; }
30          | expr '+' expr { $$ = $1 + $3; }
31          | expr '-' expr { $$ = $1 - $3; }
32          | expr '*' expr { $$ = $1 * $3; }
33          | expr '/' expr { $$ = $1 / $3; }
34          | '(' expr ')'   { $$ = $2; }
35          ;
36 %%
37 /* additional C code */
38 /* no yylex() function here */
39 int main(void) {
40    printf("Exit with Ctrl-c.\n");
41    yyparse();
42    return 0;
43 }
```

生成規則の中で、トークン（終端記号）として文字リテラル（'+', '-' など）と %token で宣言したマクロ（FOO）を用いることができます。

C ソースファイルはそれぞれ次のコマンドで生成します。

```
bison -omyparser.c -d myparser.y
flex -omylexer.c -I mylexer.l
```

必ず -d オプションをつけて Bison を実行します。このとき -o オプションで、 C ファイル名 （この場合 myparser.c） を指定しておきます。すると、拡張子を除いて同じ名前のヘッダーファイル （この場合 myparser.h） も生成されます。 （-o オプションをつけないと、 myparser.tab.c と myparser.tab.h という名前のファイルが生成されます。）

あとはこの 2 つの C ソースファイルをまとめてコンパイルします。

Microsoft Visual Studio の場合は、

```
cl /Fecalc mylexer.c myparser.c
```

/Fe は実行ファイルの名前を指定するオプションです。

これで calc.exe という名前の実行可能ファイルが生成されます。次のコマンドで実行できます.

```
calc
```

PowerShell の場合は、次のコマンドになる。

```
.\calc
```

第3章 下向き構文解析 (教科書 p.50)

3.1 下向き構文解析の特徴

演算子順位法は … if ~ else などの制御構造の部分には使いにくい
LR 法は … 人手での作成に向かない

アイデア

左端のトークン（if, whileなど）を見て生成規則を選ぶ（________）

例 教科書 p.14 図 1.4 の Statement（改）

$$
\begin{array}{rcl}
Statement & \rightarrow & \texttt{if} \; (\; ConditionExp \;) \; Statement \; ElsePart \\
& | & \{ \; StatementSeq \; \} \\
& | & Id = Expression \; ; \\
& | & Id \; (\; ExpressionList \;) ; \\
& | & \ldots
\end{array}
$$

各非終端記号について次の疑似コードで示すような（再帰的な）関数を定義する

```
someType Statement(void) {
    switch (次のトークン) {
    case IF: {
        IF  を消費;
        '(' を消費;
        c = ConditionExp();
        ')' を消費;
        s = Statement();
        e = ElsePart();
        return c, s, e を使った式
    }
    case '{': {
      '{' を消費;
      s = StatementSeq();
      '}' を消費;
      return s を使った式;
    }

    /* : */

    }
}
```

ここで「消費」とは、（確認して）入力の先頭から取り除くことである。

3.2 再帰下降構文解析 (recursive decent parsing)

- ________の一種 … 幹から葉へ解析木ができていく
- ____があるとまずい

$$expr \to const \\ \quad | \ \underline{expr} * const$$

$$StatementSeq \to \underline{StatementSeq}\ Statement \\ \quad | \ \varepsilon$$

下線部のところが、一番左端の再帰的出現である。これをプログラムにしようとすると、

```
int StatementSeq(void) {
    switch (次のトークン) {
      case IF: case WHILE: … {
        ss = StatementSeq();  /* ← */
        s1 = Statement();
        return …;
      }

       /* ⋮ */

    }
}
```

← のところで入力が変わらないので、止まらなくなる。

必要な準備

- 先頭の共通部分をくくりだす

  $$S \to A\,B \ \mid \ A\,C$$

  は

  $$S \to A\,T \\ T \to B \mid C$$

  に書き換える

- __________する
- BNF の各右辺 α に対して ________ を求める
  （$First(\alpha)$ は α の____________________）
- $A \overset{*}{\Rightarrow} \varepsilon$となりうる非終端記号 A に対して ________ を求める
  （$Follow(A)$ は A の____________________）

再帰下降構文解析のプログラムの作り方

各非終端記号に対して関数を定義する

I. $N \to X_{11}X_{12}\ldots X_{1n_1} \ \mid \ \ldots \ \mid \ X_{m1}X_{m2}\ldots X_{mn_m}$ に対して、次のトークンがどの $First(X_{i1}X_{i2}\ldots X_{in_i})$ に属するかによって分岐する（$X_{i1}X_{i2}\ldots X_{in_i} \overset{*}{\Rightarrow} \varepsilon$ となる場合は $Follow(N)$ も考慮する）

II. 右辺の $X_{i1}X_{i2}\ldots X_{in_i}$ に対して `Xi1(); Xi2 (); … Xini();` のように続けて関数を呼出す（ただし、X_{ij} が終端記号のときは単にトークンを消

費する）
そのあと________を繰り返しに書き換える（効率のため）

3.3 アルゴリズム（左再帰の除去） 《教 p.57》

$$A \to A\alpha_1 \ | \ \ldots \ | \ A\alpha_m \ | \ \beta_1 \ | \ \ldots \ | \ \beta_n$$

とする（α_i, β_j は構文記号列で β_j の先頭の記号は A ではない。 β_j の先頭以外には A は出現しても構わない）
↓
先頭が $\beta_1 \sim \beta_n$ でそのあとに $\alpha_1 \sim \alpha_m$ が 0 回以上繰り返すというかたちになる
↓
次のように書き換えることができる

$$A \to \beta_1 \, A' \ | \ \ldots \ | \ \beta_n \, A'$$
$$A' \to \alpha_1 \, A' \ | \ \ldots \ | \ \alpha_m \, A' \ | \ \varepsilon$$

→

注:

- 最後の ε を忘れない
- 間接的な左再帰（教科書 p.54）があるともう少しややこしくなるが、そのような場合でも除去可能であることが知られている

ただし、左再帰を除去すると上の図のように構文木の形が変わるため、場合によっては後処理が必要になる。

問 3.3.1

$$L \to L \ ; C \ | \ C$$

の左再帰を除去せよ。（L, C は非終端記号、**;** は終端記号である。）

問 3.3.2

$$L \to L \ ; C \ | \ L \ , C \ | \ C$$

の左再帰を除去せよ。（L, C は非終端記号、**;**, **,** は終端記号である。）

問 3.3.3

$$E \to E + F \mid E\,[\,E\,] \mid F$$

の左再帰を除去せよ。（E, F は非終端記号、+, [,] は終端記号である。）

First と Follow の求め方の例 (詳しい説明は教科書 pp.60-61)

例 5.3

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

$First(TE') = First(T) = First(FT') = First(F) = \{(, \mathbf{id}\}$
　　$\leftarrow First(F)$に ε が入っていないので
　　T'や E'は考慮しなくてよい
$First(FT') = First(F) = \{(, \mathbf{id}\}$
$Follow(E') = Follow(E) \cup Follow(E')$
$Follow(E) = \{), \$\} \leftarrow$ 開始記号の$Follow$には \$（入力の終）を追加する
$Follow(T') = Follow(T) \cup Follow(T')$
$Follow(T) = First(E') \setminus \{\varepsilon\} \cup Follow(E) \cup Follow(E')$
$= \{+,), \$\}$
$First(E') = \{+, \varepsilon\} \leftarrow$ 　εになりうる場合、εを加える
（$First(T')$ と $Follow(F)$ は求める必要はない）

3.4 予測型構文解析表 (教科書 表5.1 （p.63）)

- $First$ と $Follow$ の結果をまとめて表にまとめたもの
- 構文解析すべき非終端記号 A と入力の先頭の終端記号 a に対して＿＿＿＿＿＿＿＿＿＿＿を示す

LL(1) 文法

予測型構文解析表のエントリーに重複がない文法のことを＿＿＿＿＿という

- エントリーに重複があると構文解析中に＿＿＿＿が必要となる（通常のプログラミング言語では記述しにくい … Prolog の出番?）
- LL(1) は Left-to-Right Leftmost derivation (1) に由来する。 Leftmost（最左導出）はあとで説明する

表 5.1 教科書 p.63

	id	*	+	(	)	\$
$E \to$	i			i		
$E' \to$			ii		iii	iii
$T \to$	iv			iv		

$T' \rightarrow$		v	vi		vi	vi
$F \rightarrow$	vii			viii		

- i $\because First(TE') = \{(, \mathbf{id}\}$
- ii $\because First(+TE') = \{+\}$
- iii $\because Follow(E') = \{), \$\}$
- iv $\because First(FT') = First(F) = \{(, \mathbf{id}\}$
- v $\because First(*FT') = \{*\}$
- vi $\because Follow(T') = \{+,), \$\}$
- vii $\because First(\mathbf{id}) = \{\mathbf{id}\}$
- viii $\because First((E)) = \{(\}$

入力例 x + y * z ただし、x, y, z は **id** に属するトークンである。

予測	入力		動作

結果

導出列 ____________________

木

- 最左導出 (leftmost derivation) …

- 下向き構文解析 (descent parsing) … 先に根に近い節ができる

```
 1  /*
 2
 3  再帰的下向き構文解析プログラム
 4  (以下の文法に対する構文解析プログラム)
 5     E     -> C
 6           | F ( L )
 7     L     -> E R
 8     R     -> , E
 9           | ε
10
11  -- 以下は終端記号: 字句解析部で処理
12     C     -> 0 | 1 | ... | 9    --  一桁の数のみ
13     F     -> + | - | * | !
14
15
16  -- 予測型構文解析表
17           C       F       (       )       ,       $
18     E     C       F ( L ) ×       ×       ×       ×
19     L     E R     E R     ×       ×       ×       ×
20     R     ×       ×       ×       ε       , E     ×
21
22   */
23
24  #include <stdio.h>
25  #include <stdlib.h>   /* exit() 用 */
26  #include <ctype.h>    /* isdigit() 用 */
27
28  /* 大域変数の宣言 */
29  int token;      /* 入力の先頭のトークンを表す */
30  int yylval;     /* token の属性 */
31
32  int column = 0; /* デバッグ用 */
33
34  /* 終端記号に対応するマクロの定義 */
35  #define C 256
36  #define F 257
37
38  /* 簡易字句解析ルーチン */
39  int yylex(void) {
40      int c;
41
42      if (token == '\n') { /* 前のトークンが改行だったら */
43          column = 0;
44      }
45
46      do {
47          c = getchar();
48          column++;
49      } while (c == ' ' || c == '\t');
50
51      if (isdigit(c)) {
52          yylval = c - '0'; /* 数字から数へ変換 */
53          return C;
54      }
55
56      if (c == '+' || c== '-' || c == '*' ||c =='!') {
57          yylval = c;
58          return F;
59      }
```

```
60
61      if (c == EOF) { /* ファイルの終 */
62          exit(0);
63      }
64      /* 上のどの条件にも合わなければ、文字をそのまま返す。*/
65      return c;    /* '(', ')', ',', '\n' など */
66  }
67
68  /* デバッグ用 */
69  char* tokenName(int t) {
70      switch (t) {
71        case '\n': return "(End of Line)";
72        case 256:  return "C";
73        case 257:  return "F";
74        default:   return "(Unknown)";
75      }
76  }
77
78  /* token(終端記号)を消費して、次の token を読む */
79  void eat(int t) {
80      if (token ==  t) {
81          token = yylex();
82          return;
83      } else {
84          if (isprint(t)) {
85              printf("eat: Character '%c' is expected at column %d ", t, column);
86          } else {
87              printf("eat: Token %s is expected at column %d ",
88                     tokenName(t), column);
89          }
90          if (isprint(token)) {
91              printf("instead of '%c'.\n", token);
92          } else {
93              printf("instead of %s (code %d).\n", tokenName(token), token);
94          }
95          exit(1);
96      }
97  }
98
99  /* エラーメッセージの出力 */
100 void errMessage(char* place) {
101     if (isprint(token)) {
102         printf("%s: Unexpected token: '%c' at column %d.\n", place, token, column);
103     } else {
104         printf("%s: Unexpected token: %s (code %d) at column %d.\n",
105                place, tokenName(token), token, column);
106     }
107 }
108 /* 関数プロトタイプ宣言 */
109 void E(void);
110 void L(void);
111 void R(void);
112
113 /*
114    再帰的構文解析関数群
115    文法の各非終端記号に対応する関数
116  */
117 void E(void) {
118     switch (token) {
```

```
119         case C:
120           eat(C); break;
121         case F:
122           eat(F); eat('('); L(); eat(')'); break;
123         default:
124           errMessage("E");
125           exit(1);
126           break;
127       }
128   }
129
130   void L(void) {
131       if (token == C || token == F ) {
132           E(); R();
133       } else {
134           errMessage("L");
135           exit(1);
136       }
137   }
138
139   void R(void) {
140       switch (token) {
141         case ',':
142           eat(','); E(); break;
143         case ')':
144           /* do nothing */ break;
145         default:
146           errMessage("R");
147           exit(1);
148           break;
149       }
150   }
151
152   /* 各行の処理 */
153   void processLine(void) {
154       E();
155       if (token == '\n') { /* 入力がブロックしないように改行は特別扱い */
156           printf("Correct!\n");   /* eat('\n') の前に出力しておく */
157       }
158       eat('\n');
159   }
160
161   /* main関数 */
162   int main(void) {
163       printf("Ctrl-c で終了します。\n");
164       token = yylex(); /* 最初のトークンを読む */
165       while (1 /* 無限ループ */) {
166           processLine();  /* 各行を処理する */
167       }
168
169       return 0;
170   }
171
```

第4章 記号表と中間語

記号表 (symbol table)

＿＿＿＿はソースプログラム中の識別子 (identifier) に関する情報を集めた表である。

- 記号表のデータ構造がコンパイラーの性能に大きく影響する。
  - 配列、リスト、二分木 — これらは適さない。
  - ＿＿＿＿＿＿(hash table) を用いることが多い。 ⟦教 p.81 6.2.3⟧

    「ハッシュ」は「切り刻む」という意味で、ハッシュドビーフのハッシュと同じ語源らしい。
- 識別子はこの表のエントリーのインデックス（あるいはポインター）として表す。

中間語 (intermediate language) ⟦教 p.84⟧

＿＿＿＿では、ターゲット (CPU) の種類によらない最適化を行なう（中間語を用いるとターゲットの追加が容易になる）。

木（あるいはグラフ）構造、逆ポーランド記法、＿＿＿＿＿、＿＿＿＿＿などがある。 ⟦教 p.85⟧ どれも本質は構文木である。

ただし簡単な 1 パスコンパイラーでは中間語を生成せず直接ターゲットを生成する。

四つ組み (quadruple)

四つ組みとは

＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿

という形である。ただし　オペランドは＿＿＿＿＿＿＿＿＿＿＿＿。

例

ソース	四つ組み
`a + b * c`	＿＿＿＿＿＿ ＿＿＿＿＿＿

- オペランドに複雑な式を書くことができない。
- 途中計算の結果を必ず変数に代入しなければいけない。

第5章 誤り処理

要するに難しい。（＿＿＿＿＿，＿＿＿＿＿、と言われる。）

第6章 実行時環境とレジスター割り付け

できるだけレジスター（高速なメモリー）を効率的に利用できるようにする

第7章 コード生成

7.1 変換の基本パターン

基本パターンにあてはめ、再帰的にコード生成する (教 p.108)。基本的に、式の場合は、その値をスタックにプッシュし、文の場合はスタックは最終的に増えも減りもしない。 以下の生成コード例は Oolong のニーモニックを使用している。

① 代入文:　`v = 式1;`

```
（式1を計算するコード）        ← 再帰的に
＿＿＿＿＿＿                   ; 整数型の場合
```

③ 変数の参照（式）:　v

```
＿＿＿＿＿＿                   ; 整数型の場合
```

定数の参照（式）:　c

```
＿＿＿＿＿＿
```

⑤ 二項演算（式）:　式$_1$ op 式$_2$

```
（式1を計算するコード）
（式2を計算するコード）
（op に対応する命令）      ; 整数の足し算 "+" なら ＿＿＿＿
  ; "-" isub, "*" imul, "/" idiv, "%" irem
```

問 7.1.1 `x`, `y`, `z` がそれぞれ 1, 2, 3 という番号に対応するとき、 `x = y - z / 2;` のコードは、どうなるか?

⑥ if 文:　`if (式1) 文1 else 文2`

```
（式1を計算するコード）
                              ; … 0（偽）ならば L1 にジャンプする
＿＿＿＿＿＿
（文1のコード）
＿＿＿＿＿＿
＿＿＿＿
```

（文$_2$のコード）

ここで L_1, L_2 はフレッシュな（他と異なる）名前である。機械語などで、ラベルでなくオフセット（番地の差）を出力する必要がある場合は、＿＿＿＿＿（後戻りして修正）する

問 7.1.2 else のない if 文: `if (式1) 文1` に対しては、どのようなコードを生成するべきか? ⑥ の囲みのなかにならって書け。ラベルは一つだけ使うこと。

⑦ while 文: `while (式1) 文1`

（式$_1$を計算するコード）　; … 0（偽）ならば L_2 にジャンプする
（文$_1$のコード）

問 7.1.3 do ∼ while 文: `do 文1 while (式1);` に対しては、どのようなコードを生成するべきか? ⑦ の囲みの中にならって書け。
ヒント: Oolong の場合 `ifne` を使うと簡単になる。

⑧ 関数呼出し式: `f(式1, 式2, …)`

（式$_1$を計算するコード）
（式$_2$を計算するコード）
⋮
（f の呼出し命令）　; ＿＿＿＿＿ f、f など

呼出し命令は次のような＿＿＿＿＿(prologue, プロローグ) を行う (教 p.97)

- レジスターや局所変数を＿＿＿＿＿する
- ＿＿＿＿＿をスタックに退避する
- 関数のコードへジャンプする

⑨ return 文: `return 式1;`

（式$_1$を計算するコード）
; 整数型の場合

return 命令は次のような＿＿＿＿＿(epilogue, エピローグ) を行う (教 p.97)

- スタックに退避していた＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿する
- 戻り番地をスタックから取り出して、そこへジャンプする

第A章 Oolong について

A.1 Oolong とは

Oolong は JVM (Java Virtual Machine) のコードをターゲットとするアセンブリー言語である。Oolong については、以下の書籍で紹介されている。

> Joshua Engel: “Programming for the Java Virtual Machine”
> ADDISON-WESLEY, ISBN 0-201-30972-6, 1999

JVM は、仮想 CPU（つまりソフトウェア上でエミュレートされる CPU）の一種である。JVM の命令セットは本質的な部分は Intel x86 や MIPS などの現実の CPU と似ている。しかし、レジスターベースではなく、スタックベースなので、レジスター割り付けの必要がなく、現実の CPU を対象とするよりはコード生成が容易である。

このため、本演習では、作成するコンパイラーのターゲットとして、Intel x86 などの現実の CPU の機械語ではなく、この Oolong（すなわち JVM のアセンブリー言語）を使用する。

JVM はもともと Java というプログラミング言語のコンパイラーのターゲットとして設計された仮想機械である。Java 言語自体は、C 言語によく似た制御構造を持つ”高水準”プログラミング言語であり、Java とアセンブリー言語である Oolong とは、まったくの別物である。（C 言語と Intel x86 のアセンブリー言語が全く異なるのと同じことである。） 実際、JVM をターゲットとする Java 言語以外のプログラミング言語のコンパイラーも、数多く存在する。

A.2 Oolong の実行方法

Oolong の処理系（アセンブラー）自体も JVM 上で実装されているので、Oolong を実行するためには、まず JVM（JRE という Java の実行環境）と oolong.jar というファイルを入手する必要がある。

Oolong ソースファイル の拡張子は `.j` である。*Filename*.j という名前の Oolong ファイルをアセンブルするときのコマンドは次のようになる。（`java` は JVM を起動するためのコマンドである。くどいようだが、 Oolong と Java は別の言語である。）

```
java -jar oolong.jar Filename.j
```

oolong.jar を別のフォルダーに置いている場合は、上記の `oolong.jar` の部分は oolong.jar のフルパスを記述する。このコマンドで、 *Filename*.class というファイルが生成される。このファイルの中身は、仮想機械用のコードなので、直接実行することはできない。実行するには、次のように `java` コマンドを用いる。

```
java Filename
```

この時は、最後に拡張子の .class をつけないことに注意する。

A.3 Oolong のファイル構造

本演習で使用する Oolong ファイルは、すべて次のような雛型に従う。そして、*Statements* の部分を必要に応じて書き換える。Oolong の文法では、改行は意味を持っているので、この例のとおりに改行を入れる必要がある。

```
.super java/lang/Object
.method public static main([Ljava/lang/String;)V

Statements

.end method
```

なお、クラス名はソースファイル名の拡張子（.j）を除いた部分と同じ名前になる。

A.4 Oolong の命令文（*Statements*）

Statements は命令文（*Statement*）の並びである。Oolong では、必ず 1 行に 1 つの命令文を書く。本演習では、次のような命令文を使用する。浮動小数点数関係など、ここで紹介していないその他の Oolong（すなわち JVM）の命令文については、Oracle 社の Web ページ（http://docs.oracle.com/javase/specs/jvms/se10/html/index.html）や Jasmin (Jasmin は Oolong の基になった JVM アセンブラーである。 Jasmin と Oolong の文法はほぼ同一であるが、Jasmin で必要ないくつかの宣言を、Oolong では省略することが可能である。)の Web ページ（http://jasmin.sourceforge.net/guide.html）で知ることができる。

JVM はスタックベースの仮想機械である。つまり、Oolong のほとんどの命令文は（レジスターではなく）スタックに置かれているデータをパラメーターとして動作する。以下では、スタック操作関係、分岐関係、整数演算関係、変数操作関係、その他にわけて Oolong の命令文を紹介する。以下の説明中で、a はスタックの先頭の要素、b はスタックの 2 番目の要素を表す。「増減」はスタック中の要素数の変化を表す。

スタック操作関係

命令	増減	説明
ldc *Int*	1	整数定数をスタックにプッシュする。 (**lo**a**d** **c**onstant)
ldc *String*	1	文字列定数をスタックにプッシュする。
dup	1	スタックの先頭の要素 a を複製する。(**dup**licate)
pop	-1	スタックの先頭の要素 a を取り除く。
swap	0	スタックの先頭の 2 要素 a, b を入れ換える。
nop	0	何もしない。(**no** **op**eration)

分岐関係

JVM は goto などの無条件分岐命令と、条件付きの分岐命令を持っている。JVM のコード中では、オフセットを指定することによりジャンプするが、Oolong のソースコードでは *Label* で分岐先を指定することができる。 *Label* 名

には、アルファベットからはじまり、空白文字を含まない任意の文字列を使用することができる。

命令	増減	説明
Label :	–	goto 文など分岐命令の分岐先ラベルを設定する。
goto *Label*	0	*Label* に無条件に分岐する。
if_icmpeq *Label*	-2	a, b をスタックから取り除き（以下同様）、 b == a ならば、 *Label* に分岐する。 (**if i**nteger **comp**are **eq**ual)
if_icmpne *Label*	-2	b != a ならば、 *Label* に分岐する。 (**if i**nteger **comp**are **not e**qual)
if_icmpge *Label*	-2	b >= a ならば、 *Label* に分岐する。 (**if i**nteger **comp**are **g**reater than or **e**qual)
if_icmpgt *Label*	-2	b > a ならば、 *Label* に分岐する。 (**if i**nteger **comp**are **g**reater **t**han)
if_icmple *Label*	-2	b <= a ならば、 *Label* に分岐する。 (**if i**nteger **comp**are **l**ess than or **e**qual)
if_icmplt *Label*	-2	b < a ならば、 *Label* に分岐する。 (**if i**nteger **comp**are **l**ess **t**han)
ifeq *Label*	-1	a == 0 ならば、 *Label* に分岐する。
ifne *Label*	-1	a != 0 ならば、 *Label* に分岐する。
return	–	メソッドから値を返さずに return する。 **注:** 1.3 節で紹介した雛型でも、メソッドの最後で 必ず return する必要がある。

整数演算関係

ここでは整数に関する算術演算の命令のみを紹介する。

命令	増減	説明
iadd	-1	b + a、加算（スタックから a, b を取り除き、 b + a をスタックに積む。以下同様。） （**i**nteger **add**）
isub	-1	b − a、減算（**i**nteger **sub**tract）
imul	-1	b * a、乗算（**i**nteger **mul**tiply）
idiv	-1	b / a、（整数としての）除算（**i**nteger **div**ide）
irem	-1	b % a、（整数としての）剰余（**i**nteger **rem**ain）

変数操作関係

JVM では、変数は番号で参照され、利用できる番号は 0 〜 65535 までである。 ちなみに、3 節の雛型の場合は、このうち 0 番の変数はメソッドの引数として使用済みである。

命令	増減	説明
iload *Int*	1	*Int* 番目の変数の値をスタックにプッシュする。
istore *Int*	-1	スタックから a をポップし、 その値を *Int* 番目の変数に格納する。

その他

整数や文字列を画面に出力するために必要な命令を紹介する。スタックの先頭（a）に出力したいデータ、スタックの 2 番目（b）に出力ストリームが入っている状態で、出力のための命令を呼び出す。

命令	増減	説明
getstatic java/lang/System/out Ljava/io/PrintStream;		
	1	標準出力ストリームをスタックにプッシュする
invokevirtual java/io/PrintStream/println(Ljava/lang/String;)V		
	-2	出力ストリーム b に a（文字列）を出力する。
invokevirtual java/io/PrintStream/println(I)V		
	-2	出力ストリーム b に a（整数）を出力する。
invokevirtual java/io/PrintStream/println(C)V		
	-2	出力ストリーム b に a（文字）を出力する。

A.5 Oolong のプログラム例

まず、“Hello World!” と出力する Oolong のコードを紹介する。

ファイル名: Hello.j

```
1 .super java/lang/Object
2 .method public static main([Ljava/lang/String;)V
3   getstatic java/lang/System/out Ljava/io/PrintStream;
4   ldc "Hello World!"
5   invokevirtual
  java/io/PrintStream/println(Ljava/lang/String;)V
6   return                          ; 最後にreturnが必要
7 .end method
```

なお、Oolong はセミコロン「;」から行末までが**コメント**になる。

次は、繰返しを用いて “Hello World!” を 10 回出力するプログラムである。

ファイル名: ManyHello.j

```
 1 .super java/lang/Object
 2 .method public static main([Ljava/lang/String;)V
 3   ldc 0
 4   istore 1                  ; 変数1に 0を代入する
 5 
 6   loop:                     ; ここからループ
 7     iload 1
 8     ldc 10
 9     if_icmpge exit          ; 変数1が 10以上なら exitへ
10     getstatic java/lang/System/out
   Ljava/io/PrintStream;
11     ldc "Hello World!"
12     invokevirtual
   java/io/PrintStream/println(Ljava/lang/String;)V
13     iload 1
14     ldc 1
15     iadd
16     istore 1                ; 変数1に 1を足す
17   goto loop                 ; loopへジャンプ
18 
19   exit:
20     return                  ; 最後にreturnが必要
21 .end method
```

ちなみに、これらは、それぞれ次のような Java のプログラムのコンパイル結果に相当する。この中で `System.out.println` は C 言語の `puts` に相当する出

カメソッドである。

ファイル名: Hello.java

```
1 public class Hello {
2   public static void main(String[] args) {
3     System.out.println("Hello World");
4     return;
5   }
6 }
```

ファイル名: ManyHello.java

```
 1 public class ManyHello {
 2   public static void main(String[] args) {
 3     int i = 0
 4     while (true) {
 5       if (i >= 10) break;
 6       System.out.println("Hello World");
 7       i = i + 1;
 8     }
 9     return;
10   }
11 }
```