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。）

```
%{
  /* C宣言部 – 動作記述の中で用いる関数の定義や宣言 */
#define YYSTYPE double     /* トークンの属性の型を宣言 */

#include <stdio.h>
#include <stdlib.h> /* exit関数を使うため */
#include <ctype.h>  /* isdigit関数を使うため */
void yyerror(char* s) {   /* エラーがあった時に呼ばれる関数を定義
   printf("%s\n", s);
}
int yylex(void);
%}
  /* Bison宣言部 */
  /* 終端記号（トークン）（と非終端記号）の宣言 */
  /* トークンは定数マクロとして定義される */
  /* トークンとして使えるのは文字コードかここで定義したマクロ */
%token NUMBER
  /* 曖昧な文法に対して演算子の優先順位と結合性を宣言できる */
%left '+' '-'
%left '*' '/'
  /* leftは左結合を表す、ちなみに右結合は right、非結合は nonasso
  /* 優先順位が高い演算子ほど下に書く。 */

%%
  /* 文法記述とそれに対する動作（還元時に実行されるプログラム） */
  /* 最初に start symbolを書く。 */
input    :   /* 空 */
         | input line    {}
         ;
  /* 通常の BNFの → の代わりに ： を書く。各BNFは ； で終える。 */
line     : '\n'          { exit(0); }
         | expr '\n'     { printf("\t%g\n", $1); }
         ;
  /* $$は左辺の属性値（意味値）、$nは右辺の n番目の文法要素の属性値
expr     : NUMBER        { $$ = $1; }
         | expr '+' expr { $$ = $1 + $3; }
         | expr '-' expr { $$ = $1 - $3; }
         | expr '*' expr { $$ = $1 * $3; }
         | expr '/' expr { $$ = $1 / $3; }
         | '(' expr ')'  { $$ = $2; }
         ;
%%
  /* 追加の Cプログラム */
  /* yylex（字句解析）関数を flexを使わず定義している。yylexの戻り
int yylex(void) {
  int c;
```

```
  do {
     c = getchar ();
  } while (c == ' ' || c == '\t');

  if (isdigit(c) || c == '.') {
    ungetc(c, stdin);
    scanf("%lf", &yylval);
    /* トークンの属性値は yylvalという 大域変数に代入して返す。*/
    return NUMBER;
    /* NUMBERというトークン（マクロ）を返す。*/
  } else if (c == EOF) {
    return 0; /* 終了を表す。*/
  }
  /* 上のどの条件にも合わなければ、文字コードをそのまま返す。*/
  return c;
}

int main(void) {
  printf("Ctrl-cで終了します。\n");
  yyparse(); /* Bisonが生成した関数 */
  return 0;
}

/* この例では、mainを自前で用意しているが、通常は他のファイルに
   main関数など他の関数を定義する。*/
```

説明

Bison の核心部分は BNF とそれに付随するアクションです。 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。

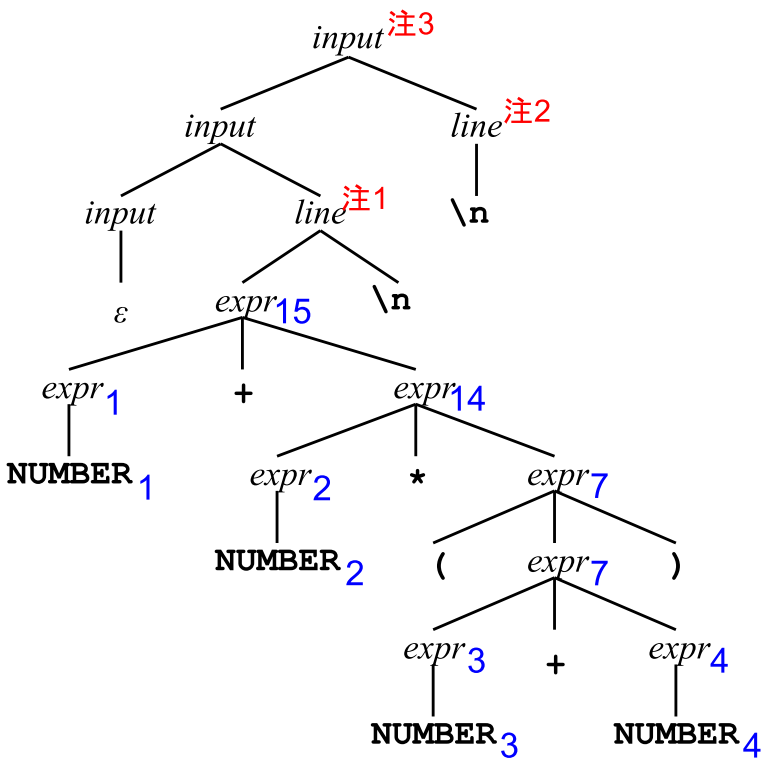
例えば、入力が “`(12+34)*56\n`” という文字列の場合、上の例の `yylex()` の戻り値とそのときの `yylval` の値は次のようになります。

	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。）

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
```

次のコマンドで実行できます。

```
calc
```