(ver. 1.3 (2018) 笹倉)

yacc プログラムの書き方[1]

1 yacc

yacc (Yet Another Compiler Compiler) は UNIX に標準で用意されているツールで構文解析ルーチンを生成する．字句解析されてトークンの並びとなったものを入力とし，yacc の書式に従って記述された文法に従っているかどうかを判定し，そこに記述された処理をするプログラムを自動生成する．具体的には yacc の書式に従って記述された拡張子が y のファイル*.y（これを yacc プログラムと呼ぶことにする) を yacc に通すことによって*.tab.h と　*.tab.c という名前の C プログラムが生成される．自動生成されたプログラムをさらに C コンパイラに通すことによって実際に動く実行ファイルを得ることができる．

ここではコンパイラ実験で必要と思われる yacc プログラムの書き方を解説する．もっと詳しい yacc の書き方を学びたいものは参考文献を参照されたい．

なお，UNIX に標準で用意されているのは yacc であるが，コンパイラ実験ではそれと同じ書き方で動作する bison を処理系として使用する．

2 yacc プログラムの構造

yacc のプログラムも lex と同じく定義部，規則部，ユーザ定義サブルーチン部からなり，それぞれの間が%%で区切られるところも同じである．

定義部には次のものが書ける．

%token この yacc プログラムの中で用いるトークンの名前を列挙する．この yacc プログラムと一緒に用いる字句解析プログラムではこれらのトークンを切り出すように書かれている必要がある．ここで指定した token 名は*.tab.h ファイルの中に#define 文で定義される．*.tab.c と字句解析プログラムはこの*.tab.h を include することによってここで定義したトークン名を共通のものとして使うことができるようになる．

%token の後に <typename> をつけることでここで定義したトークンの型を typename で示された型にすることができる．詳しくは 8 節を参照のこと．

%type <typename> 非終端記号の型を指定するときに用いる．トークンに型をつけるとそれに関連した非終端記号にも適宜型を指定する必要がある．詳しくは 8 節を参照のこと．

%{ と %} で囲まれた部分 そのまま*.tab.c のファイルにコピーされる．C プログラムの最初の部分になるので，必要なヘッダファイルの include などを書く．

規則部では文法規則を書く．文法規則は一つ一つの規則は

左辺:右辺

で書く．これは BNF での文法規則をそのまま書けばよい．実際に構文解析プログラムとして用いるには右辺のあとにアクションを書く必要がある．規則部の書き方の詳細については 3 節を参照されたい．

[1] この資料で扱っている例題は基本的に参考文献 [1] に掲載されているものである．(一部それを改変した)

ユーザ定義サブルーチンに書いたものはすべてのそのまま*.tab.c にコピーされる．yyparse() 関数を呼び出さないと構文解析ルーチンが呼び出されないので，yacc のユーザ定義ルーチンに main 関数を書く場合にはその中で必ず yyparse() を呼び出さなければならない．字句解析ルーチンを呼び出す yylex() 関数は yyparse() の中で呼び出されているので yyparse() を呼び出した場合は特別な必要がない限り yylex() を明示的に呼び出す必要はない．

例として「I am」という並びだけを受理するプログラムを示す．このプログラムでは文法としては主語＋空白＋述語で構成される文を受理するが，字句解析の方で主語は I だけ述語は am だけと限定している．

例 1-1 – sample.y

```
%{
#include <stdio.h>
#include "sample.tab.h"
extern int yylex();
extern int yyerror();
%}
%token SUBJECT PRED SPACE
%%
statement
: SUBJECT SPACE PRED { printf("OK!\n");}
%%
int main(void){
  if (yyparse()) {
    fprintf(stderr, "Error ! Error ! Error !\n");
    return 1;
  }
}
```

例 1-2 – sample.l

```
%{
#include "sample.tab.h"
/* lex for sample.y */
%}
%%
I {return SUBJECT;}
am {return PRED;}
[\t ]+ {return SPACE;}
\n   return 0;
.    return yytext[0];
%%
```

まず，例 1–1 sample.y を見てみよう．これが yacc ファイルである．定義部には%{ %} で囲まれた C 言語の部分と%token で始まるトークン定義の部分がある．

%{ %} で囲まれた C 言語の部分にはまず stdio.h の include 文がある．これは sample.y の規則

部のアクションで printf を用いているために必要となる include 文である．次に sample.tab.h を include している．これは字句解析ルーチンと連携するためである．字句解析ルーチンとの連携の詳細は 10 節にまとめる．次の yylex(), yyerror() の extern 宣言は，yyparse() を使う場合に必要となる．これがないとリンク時にエラーとなる．yylex(), yyerror() は yyparse() から呼ばれている関数である．

%token で始まるトークン定義部には SUBJECT, PRED, SPACE というトークンを使うということを宣言している．これらは規則部で使われるトークンである．規則部で使われるトークンは全てこのトークン定義部で定義されていなければならない．アクション部では SUBJECT SPACE PRED という文法に合致したら”OK!” を出力するコードを書いている．

ユーザ定義ルーチンでは main() 関数が書かれている．この中で yyparse() を呼び出している．yyparse() は定義した文法に合った場合は真，それ以外は偽を返す．入力が文法に合致しなかった場合は yyparse() 自体が標準エラーに” syntax error” というメッセージを出力するので，このプログラムはそれに加えて”Error！Error！Error!” というメッセージが標準エラーに出ることになる．

次に例 1–2 sample.l を見てみよう．ここでは sample.y 用の字句解析ルーチンとして lex プログラムを書く．ユーザ定義ルーチンはなにもない．定義部では yacc プログラムと連携を取るために sample.tab.h の include 文がある．トークンは SUBJECT, PRED, SPACE の 3 種類なのだが，lex 規則部のアクションとして対応するトークン名を return するという形で書いている．これを行うと例えば”I” が来た時に SUBJECT というトークン番号が yylex() を呼び出したプログラムに return 値として返る．sample.tab.h の中を見てみればわかるがトークン名は#define 文で整数値に変換される．実際のトークンの文字列を返したい場合は”.” のアクションにあるように return yytext[0] とする．lex は yytext という配列にトークンの実際の文字列を入れるようなプログラムを生成する．

このプログラムでは”I” の場合 SUBJECT, “am” の場合 PRED, タブまたは空白が一個以上連続した場合 SPACE, 改行が来た時 0, それ以外の文字が来た時にはその入力文字列の先頭の文字のコードを返す．0 が返ってくると yyparse は終了するので，改行の入力は入力の終わりを意味する．yacc と lex の連携の詳細は 10 節を参照されたい．

3　規則部の書き方

3.1　基本

yacc の規則部は，文法の BNF と基本的に同じことを書く．ただ，それだけだと文法に合っているかどうかだけを判定するだけのプログラムになってしまう．コンパイラにするためには，文法に合っていた場合，実際にどのようなコードを生成するかというプログラムを書く必要がある．それを右辺のアクション部に書く．例 1 のプログラムでは単に printf 文でメッセージを出力するだけであったが，ここにもっと複雑なプログラムを書くことでコンパイラが出来ていく．yacc の規則部の書き方のまとめを表 1 にまとめる．

yacc で生成されるプログラムは，入力トークン（の列）が来たらそれに合致するルールを探し，それが見つかれば書いてあるアクションを実行する．見つからなかったら yyerror() に書いてあるコードを実行する．デフォルトでは”syntax error” というメッセージを出力して終わる．

yacc で正しく構文解析するプログラムを生成するためには，定義する文法に注意すべき点がある．ここではそのうち二つの場合について説明する．

表 1: yacc 規則部まとめ

規則の形は　左辺:右辺;
右辺にはアクション（通常は C 言語のコード）も書ける.
左辺は非終端記号（文脈自由言語になる必要十分条件）
最初のルールの左辺がスタートシンボルとみなされる
左辺と右辺の区切りは:
ルールの区切りは;
lex の場合と同様に | が使える (OR と思って使える）
右辺に書かれたアクションはこのルールが評価された時点で実行される

3.2　二つ以上のトークンを先読みしないといけない文法には対応していない

yacc は一つ先のトークンだけを先読みして動く．例 2 には二つ以上先のトークンを先読みしないといけない文法の例を示す.

例 2–1 二つ以上先のトークンを先読みしないといけない文法の例

```
%token HOARSE AND CART PLOW GOAT OX
%%
  phrase : cart_animal AND CART
         | work_animal AND PLOW ;
  cart_animal : HORSE | GOAT;
  work_animal : HORSE | OX;
%%
```

ここで大文字はトークン，小文字は非終端記号である．この文法は HORSE AND CART という文字列も HORSE AND PLOW という文字列も受理する文法であるが，構文木を作るときその HORSE が cart_animal という非終端記号から導出されたものか work_animal という非終端記号から導出されたものかは HORSE の二つ先のトークンが CART か PLOW かをみないと判断できない.

このような文法は，少々冗長にはなるが，一つ先のトークンだけを先読みすればよい文法に書き換えることができる．以下例 2–2 は例 2–1 と同じものを一つ先のトークンだけを先読みすればよいように規則を書き換えたものである.

例 2–2 例 2–1 を一つのトークンだけを先読みするよう書き換えた例

```
%token HOARSE AND CART PLOW GOAT OX
%%
  phrase : HORSE AND work_doing
         | GOAT AND CART
         | OX AND PLOW ;
  work_doing :  CART | PLOW;
%%
```

yacc を用いる時には一つのトークンだけを先読みするような文法にしておく必要がある．二つ

以上先のトークンを読まなければならない文法の場合 bison により警告が出る．例えば例 2–1 の yacc プログラムを教育計算機の bison に通すと，”warning: rule useless in parser due to conflist: work_animal: HORSE” という警告が出る．

3.3 曖昧性のある文法には対応していない

yacc は曖昧性のある文法は扱えない．曖昧性のある文法とは，一つの入力に対して複数の構文木を構成することが可能な文法のことである．曖昧性のある文法には還元/還元衝突 (reduce/reduce conflict) とシフト/還元衝突 (shift/reduce conflict) の 2 種類がある．

還元 (reduce) とは文法の右辺から左辺への書き換えのことをいい，シフト (shift) とは入力されたトークンが文法の右辺に合っていることを確認して右辺に残っている次のトークンに移動することを言う．還元/還元衝突は，右辺から左辺への書き換えで該当する規則が複数ある場合に発生する．シフト/還元衝突はそのままシフトするかもしくは還元するかのどちらもありうる場合に発生する．

例 3–1 は還元/還元衝突の例である．

例 3–1 還元/還元衝突の例

```
%token IDENT PLUS MINUS
%%
target : IDENT MINUS IDENT | IDENT kigou IDENT;
kigou : PLUS | MINUS
%%
```

これは IDENT MINUS IDENT という入力が来た時，target の一つ目の規則の MINUS も二つ目の規則 kigou の右辺の MINUS もどちらも当てはまってしまう．これは還元/還元衝突である．

“warning: rule useless in parser due to conflist: kigou: MINUS” という警告が教育計算機の bison では出る．

例 3–2 はシフト/還元衝突の例である．

例 3–2 シフト/還元衝突の例

```
%token IDENT NUM PLUS MINUS
%%
arith :  var kigou var;
var : IDENT | NUM | arith ;
kigou : PLUS MINUS;
%%
```

この例で IDENT PLUS IDENT PLUS IDENT という入力が来た時，IDENT PLUS まで入力を読んだ時，この PLUS が var の arith の中の kigou なのか（シフト），　var を IDENT だけと解釈して（還元）最初の arith の kihou のなのか両方の可能性がある．これがシフト/還元衝突である．

シフト/還元衝突の場合は “conflicts: 1 shift/reduce” のようなメッセージが出る．例 3–2 のプログラムを教育計算機の bison に通すと”conflicts: 2 shift/reduce” というメッセージが出る．

4　演習問題 1

授業内で示す基本言語文法を yacc で書け．ただし，アクション部には (まだ) 何も書かず，bison を警告なしに通ることを確認せよ．

5　簡単なアクションの実装

それでは yacc を使って加減算のみを行える計算機の文法を記述し，そこに実際に加減算を計算するアクションを書いてみる．ここで実装する計算機は整数の足し算と引き算のみができるものとする．一つの四季の中に複数演算子を書くことができ，複数の演算子がある場合には左から計算していくことにする．例えば 3 + 4 + 5 - 2 という式があればまず　3 + 4 を行いその結果に 5 を足しその結果から 2 を引く．

加減算のみを行える計算機の文法規則を yacc の記述法に従ってかいたものが例 4–1 である．

例 4-1 加減算のみを行える計算機の文法規則

```
%token NUMBER
%%
statement : expression
;
expression : expression '+' NUMBER
  |          expression '-' NUMBER
  |          NUMBER
;
%%
```

ここで' でかこまれたものは文字リテラルトークンというもので' で囲まれた文字そのものがトークンである，%token で宣言されているものは字句解析により切り出されたトークン，それ以外の小文字で書かれているものは非終端記号である．この段階では文法に合っているかどうかのチェックしかしない．これらのトークンと非終端記号を合わせて「シンボル」と呼ぶ．

ではアクション部に実際に計算するコードを書いてこの計算機に加減算を計算させよう．具体的には expression の右辺の expression '+' NUMBER では expression の値と NUMBER の値を足し，expression '-' NUMBER では expression の値から NUMBER の値を引く．NUMBER だけの場合はその値を結果とする．statement の右辺では expression の結果を printf を用いて出力することにする．

シンボルの値を参照する特別な記号が　$1, $2, . . . である．また$$に値を代入することで規則の左辺のシンボルの値を設定することができる．これを使って加減算のアクションを追加したものが例 4–2 である．

例 4–2 例 4–1 にアクションを加えたもの (p-m.y)

```
%{
#include <stdio.h>
#include "p-m.tab.h"
%}
%token NUMBER
%%
statement : expression { printf("= %d\n", $1);}
;
expression : expression '+' NUMBER {$$ = $1 + $3;}
  |          expression '-' NUMBER {$$ = $1 - $3;}
  |          NUMBER {$$ = $1;}
;
%%
```

printf を使うので，定義部に include 文を付け加えている．また C 言語でトークン NUMBER の宣言が必要なので p–m.tab.h を include している．アクション部のアクションは C 言語で書く．この段階で bison は通るが動くプログラムにはなっていない．

6 動くプログラムにするために–lex との連携の基礎

前節の加減算のみを行えるプログラムを実際に動かすためには

1. 字句解析ルーチン（例えば lex プログラム) を加える
2. C の main 関数を加える

の 2 つを行わなければならない．

6.1 対応する lex プログラム

加減算のみを行えるプログラムのための字句解析ルーチンは以下の仕様を満たすものでなければならない．

- 数字（今回の場合は整数）がきたら NUMBER を返す．
- スペース，タブは読み飛ばす．
- 改行が来たら入力終わりとするので 0 を返す．
- それ以外の文字はそのまま yacc へ渡す（今回の例の場合には'+' や'-' を yacc プログラムがそのまま必要とするので．一文字で一種類のトークンである場合にはこのように直接文字を yacc プログラムに渡してもいいし，あるいは PLUS, MINUS のようなトークンを定義してそれを使ってもよい．どちらも可能.）.

これを満たす lex プログラムが以下の例 5–1 である．

例 5–1 加減算のみを行える計算機用の lex プログラムその 1

```
%%
[0-9]+ {return NUMBER;}
[\t ] ; /* ignore whitespace */
\n     return 0;
.      return yytext[0];
%%
```

しかし，この lex プログラムはこれだけでは lex コンパイルを通らないし，例 4–2 の yacc プログラムともうまく連携できない．その原因は二つある．それは

1. NUMBER の宣言がどこにもない．
2. 入力として数字が来た時，その数字の値を lex プログラムから yacc プログラムに渡していないので yacc プログラムの方で足し算・引き算ができない．

である．

NUMBER の宣言

NUMBER の宣言は，連携する yacc プログラムの中の%token でされている．これは yacc を通すとファイル名.tab.h ができてその中で define 文で宣言されている．従ってこのファイル名.tab.h を lex プログラムの定義部で include すればよい．

yacc プログラムへの数字の引き渡し

yacc プログラムが規則部にかかれた文法に合っているかどうかを見るときは lex プログラムが return で返す値を使う．トークンにはファイル名.tab.h で数字が割り当てられているのでその数字と一致するかどうかを見る．例えば例 4–2 の p-m.y のプログラムの'+' のようにトークンとしての宣言がないものの場合はその文字自体と一致するかどうかを文字コードで判断する．そのため lex プログラムは数字や空白，タブ，改行以外の文字の時に yytext[0] を返している．

yacc プログラムのアクション部で$1, $2 のように参照されるシンボル値を設定するには lex プログラムで yylval 変数に値を入れる．そうすると yacc プログラムの方の$1, $2 でその値を使える．今回の場合は数字だった時に return では NUMBER を返し具体的な数値は yylval に入れておくと yacc に数字を渡すことができる．

例 4–2 の yacc プログラムと連携できる lex プログラムを例 5–2 に示す．

例 5–2 加減算のみを行える計算機用の lex プログラム完成版 (p-m.l)

```
#include "p-m.tab.h"
extern yylval;
%%
[0-9]+ {yylval = atoi(yytext); return NUMBER;}
[\t ] ; /* ignore whitespace */
\n     return 0;
.      return yytext[0];
%%
```

6.2 C のプログラムとしてコンパイル可能にするために

この段階で p-m.y(例 4–2）と p-m.l(例 5–2) はそれぞれ yacc と lex を通るはずである．しかし，C プログラムとして実行できるようにするためには main 関数がなければならない．それを付け加えたのが例 4–3 の p-m.y である．

例 4–3 加減算のみを行える計算機用の yacc プログラム完成版 (p-m.y)

```
%{
#include <stdio.h>
#include "p-m.tab.h"
extern int yylex();
extern int yerror();
%}
%token NUMBER
%%
statement : expression { printf("= %d\n", $1);}
;
expression : expression '+' NUMBER {$$ = $1 + $3;}
  |          expression '-' NUMBER {$$ = $1 - $3;}
  |          NUMBER {$$ = $1;}
;
%%
int main(void)
{
  if(yyparse()) {
    fprintf(stderr, "Error\n");
    return 1;
  }
  return 0;
}
```

main 関数は yyparse() を呼ぶだけである．yyparse はもし入力が文法に合わない場合は戻り値が 1 となり，入力ファイルが終わりになった場合は戻り値が 0 になる．このプログラムでは yyparse の戻り値が 1 の場合には”Error” を標準エラーに出力するようにしている．yyparse 関数の中でも文法に合わなかった場合には”syntax error” を出力するので，文法に合わなかった場合は”syntac error” と出力した後に”Error” と出力されることになる．

定義部にある　extern int yylex(); と extern int yerror(); は yyparse 関数から呼ばれる関数である．よってこの二つの文がないとリンクした時にエラーが出る．

6.3 C のプログラムとしてコンパイルする

加減算のみを行える計算機のプログラムを演習室の環境で C プログラムとしてコンパイルするには以下のコマンドを実行すればよい．ただし p-m.y が yacc プログラム p-m.l が lex プログラムとする．

コンパイルの仕方

```
% bison -d p-m.y
% flex p-m.l
% gcc -o p-m p-m.tab.c lex.yy.c -lfl -ll
```

7 演習問題 2

加減算のみを行える計算機のプログラムに乗除算を付け加えてみよう.

8 型付トークン

演習問題 1 の結果，入力プログラムが文法に合っているかのチェックをするプログラムはできた. 演習 2 では四則演算ができる計算機を作成することができた.

しかし，コンパイラにするためには文法に合った入力の時にそれに対応するコードを出力するようにしなければならない．そのためには yacc のアクション部でまず入力プログラムを抽象構文木に書き換えるプログラムを書く．その詳細は資料「抽象構文木」で解説するが，そのための下準備としてここではトークンに型を指定する方法について解説する.

なぜトークンに型をつける必要があるのか.

特に指定しなければ yacc はすべてのトークンの値（シンボル値）を int 型として扱う．しかし，コンパイラでは最終的には例えば数字は数字として int 型で，変数名は変数名として文字列型で受け取らないと後の処理ができない.

必要に応じてアクション部でシンボル値の型を変換する方法もありうるが，それよりも yacc の型付け機能を使って型を指定した方が簡単である．またそれによって文法のミスなどを発見することも可能となる.

トークンに型を指定する方法

トークンのシンボル値の型は yacc プログラムの定義部で指定する．トークンを定義するときに %token の後に < 型名 > をつけることでそこで定義されるトークンの値の型を指定することができる．指定する型は%union 宣言を使って（適当に名前をつけた型を）自分で宣言する.

yacc プログラムではアクションを特に書かなければデフォルトで$$ = $1 というアクションが実行される．つまり一つ目のトークン（もしくは非終端記号）のシンボル値がその規則の左辺の非終端記号のシンボル値となる．そのためトークンに型をつけたらすべての非終端記号にも型が必要となる.

- 一つのトークンに型を付けたら全てのトークンにつける必要がある.（「デフォルトの型」はない）
- アクションの書き方によっては非終端記号にも型をつけなければならない．非終端記号の型は%type 宣言を使う.

以下，演習問題 2 の四則演算のできる計算機を変数も使えるように拡張する例を使って型付きトークンを使ったプログラムを説明する.

例題：変数も使える実数の計算機

演習問題 2 の四則演算のできる計算機に以下の拡張を施す.

- 式の中で変数を扱えるようにする．
- 変数はアルファベット小文字一文字からなるものとする．すなわち変数の数はたかだか 26 である．
- 変数の数がたかだか 26 なので変数に入っている値は 26 個の要素をもつ配列に格納する．この配列の名前を vbltable として定義することにする．
- これまでのように一行だけで計算が終わるようにすると変数を導入した意味がないので，数式の計算結果を変数に代入できるようにする．また改行で入力を終了するのではなく，改行で区切られたいくつもの式を連続で計算できるようにする．
- 一つの計算が終わるたびにその結果を標準出力に表示する．
- $ が入力されたらそこで入力おわりと判断する．

ここで作成する実行ファイルを arithv という名にし，yacc プログラム，lex プログラムの名前をそれぞれ arithv.y, arithv.l とする．aruthv の実行結果の例を以下に示す．

arithv の実行例

```
% ./arithv
> a = 100
a = 100
> b = 35
b = 35
> a * b
= 3850
> $
%
```

ここではわかりやすいように入力行の頭に”> ” を表示するようにしている．

さて，四則演算のできる計算機から変数も扱える計算機 arithv に拡張するためにしなければならないことは以下である．

- 変数用のトークンの定義
- シンボル値の型定義（数字の場合と変数の場合で変える）
- 変数への代入処理

まずは変数も使える計算機の yacc プログラムのトークンを定義する部分と規則部分のみを以下に示す．ここではアクション部分はまだ書いていない状態である．

例 6–1 変数も使える計算機 yacc プログラムのトークン定義と規則部分（アクションなし）

```
%union{
  int dval;
  int vblno;
 }
%token <vblno> NAME
%token <dval> NUMBER
%%
statement_list : statement '\n'
|   statement_list statement '\n'
;

statement : NAME '=' expression
| expression
;

expression : expression '+' mulexp
|            expression '-' mulexp
|            mulexp
;

mulexp : mulexp '*' primary
|        mulexp '/' primary
|        primary
;

primary : '(' expression ')'
|        NUMBER
|        NAME
;
%%
```

規則部からみてみよう．複数式を計算できるように statement_list という非終端記号を導入している．改行'
n' を式の区切りとしているのが規則からわかる．

statement は代入文に相当するものである．NAME が変数を示すトークンである．結果を変数へ代入する式と代入しない式の両方が書けるようになっている．

expression と mulexp については四則演算が行える計算機と同じである．primary に NAME が追加されている．

定義部のを見てみよう．トークンは NAME と NUMBER の二つである．NAME が変数，NUMBER が数字である．シンボル値としては NAME の場合変数が格納される配列の何番目であるかの数字を返すことにする．NUMBER の場合はその数字である．結果的にどちらも今回は整数になるので型をつけなくても動くプログラムは作成できるが，NAME のシンボル値は配列の添え字で

NUMBER のシンボル値は計算に使う数字であるというように性質の違う整数なのでこの例題では別の型を当てることにする．

その型の定義が%union 文である．ここでは int 型の dval という型と int 型の vblno という型をて意義している．%union 文の中に書かれたことはそのままヘッダファイルの中で YYSTYPE という共用体の宣言として使われる．ここで dval を NUMBER の型，vblno を NAME の型と想定して定義している．

型はトークン宣言の時に使われる．%token <vblno> NAME とすることで NAME という型のトークンのシンボル値の型が vblno であることが宣言される．%token <dval> NUMBER も同様である．

これにアクションを加え，lex との連携のためのコードも追加した完成形を次に示す．

変数も使える計算機 yacc プログラム (arithv.y)

```
%{
#include <stdio.h>
#include "arithv.tab.h"
extern int yylex();
extern int yyerror();
int vbltable[26];
%}
%union{
  int dval;
  int vblno;
 }
%token <vblno> NAME
%token <dval> NUMBER
%type <dval> expression mulexp primary
%%
statement_list : statement '\n'
|   statement_list statement '\n'
;
statement : NAME '=' expression {vbltable[$1] = $3;
                    printf("%c = %d\n> ", $1+'a', $3); }
| expression {printf("= %d\n> ", $1);}
;
expression : expression '+' mulexp {$$ = $1 + $3; }
|            expression '-' mulexp {$$ = $1 - $3; }
|            mulexp {$$ = $1;}
;
mulexp : mulexp '*' primary {$$ = $1 * $3; }
|        mulexp '/' primary
          {if($3 == 0) {yyerror("divide by zero"); return 0;}
          else $$ = $1 / $3; }
|        primary {$$ = $1;}
;
primary : '(' expression ')' {$$ = $2; }
|        NUMBER {$$ = $1;}
|        NAME {$$ = vbltable[$1];}
;
%%
int main(void)
{
  printf ("> ");
  if (yyparse()) {
    fprintf(stderr, "Error\n");
    return 1;
  }
  return 0;
}
```

アクション部を付け加えると非終端記号の中で型を指定しなければエラーメッセージがでてくるものがある．例えば primary の型を定義していないと arithv.y:40.16-17: $$ of ‘primary’ has no declared type のような（教育用計算機の環境で確認のこと！）メッセージが yacc を通した時にでる．それをでないようにするために%type 文で必要な型をて意義している．

変数の値を記録するために vbltable という配列を宣言している．変数のシンボル値はこの配列の添え字とする．すなわち a の時 0，z の時 25 がシンボル値となるようにする．そのシンボル値の決定は lex プログラムで行う．次に lex プログラムを示す．

変数も使える計算機 lex プログラム (arithv.l)

```
%{
#include "arithv.tab.h"
#include <math.h>
extern int vbltable[26];
%}
%%
[0-9]+   {yylval.dval = atoi(yytext) ; return NUMBER; }
[\t ] ; /* ignore whitespace */
[a-z]  {yylval.vblno = yytext[0] - 'a'; return NAME; }
"$"   {return 0; /* end of input */}
\n           |
.            return yytext[0];
%%
```

型をつけると yylval が共用体として宣言されるのでシンボル値を設定する時には yylval. 型名に代入する．ここでは数字の場合は関数 atoi を使って yytext から整数に変換し，変数の場合は yytext[0] から’a’ のコードを引くことで a が 0，z が 25 の添え字となる値を設定している．

9 発展問題

変数も使える計算機のプログラム (arithv.y, arithv.l) を整数ではなく実数の計算をするように変更せよ．

10 yacc と lex の連携まとめ

yacc は字句解析の結果を lex から受け取る．受け取れる値（結果）には以下のものがある．

トークンの種類 lex が切り出したトークンがどの種類のものかを知ることができる．

- yacc ファイルの定義部のところで%token の後にトークンの種類の名前を列挙することでトークンの種類を定義できる．普通は大文字で定義する．
  例：%token NUMBER
- 定義されたトークンは yacc ファイルの規則部の規則の中で使われる．
  例：expression : expression ’+’ NUMBER

- lex ファイルの方では，トークンを切り出したときにアクションで return する値がトークンの種類になる．
  例：[0-9]+ {return NUMBER;}
- 注）基本的に lex のアクション部で return した値が yacc の規則部の規則と比較される．普通はこのトークン種類が使われるが，そうでない場合もある．例えば，例 4-1 の
  expression : expression '+' NUMBER
  の'+' は「+」文字そのものを意味しているのだが,「+」を yacc に渡すために lex での return 値は
  return yytext[0];
  となる．

トークンのシンボル値 lex で yylval に値を設定しておくと yacc でそれをトークンのシンボル値として使える．シンボル値とは，$1, $2, ··· で参照できる値のことである．

- lex で yylval に値を入れる．
  例：[0-9]+ {yylval = atoi(yytext); return NUMBER;}
- そうすると yacc の方でその値をシンボル値として受け取ることができるようになる．
  例：expression : expression '+' NUMBER {$$ = $1 + $3;}
- yylval の型は YYSTYPE でデフォルトでは int だが，#define 文 もしくは %union を用いてで任意の型に変えることができる．#define 文を用いた場合はシンボル値の型がすべて#define 文で指定したものになる．%union を用いて複数の型を扱える共用体を用いるとトークン種類ごとに型を変えることができる．コンパイラを作成するときには少なくとも数値と AST のノードをシンボル値として用いることになると思われるので %union を使う必要があるであろう．

トークン文字列 lex がトークンを切り出したとき，そのトークンの文字列は yytext という変数に文字列として入っている．大域変数なので yacc から参照することも可能だが，yytext には常に一番最新の lex の結果切り出されたトークン文字列が入っているので，yacc でトークン文字列が必要になる場合には，次の lex の呼び出しで yytext が上書きされてもよいように，yytext の値をどこかにコピーして保存しておく必要がある．

参考文献

[1] John R. Levine, Tony Mason, Doug Brown 共著，村上列訳，”lex & yacc プログラミング”，アスキー出版，1994.

[2] John R. Levine, “Flex & Bison”, Oreilly & Associates Inc, 2009. (英語)