情報科学実験II 第3回

コンパイルの4つのステップ

1、字句解析(lexical analysis)
2、構文解析(syntax analysis) ⇐ ここに関係する話の続き
3、意味解析(semantic analysis)
4、コード生成(code generation)

目標
・シンタックス（文法）ベースの考え方にさらに慣れる
・簡単なプログラミング言語のインタプリタを作る

lexとyaccの関係

「yyparseの中でcallするyylexをlexが自動生成する」という関係

⇒ y.tab.cにlex.yy.cを#includeしてコンパイルすればよい(普通は分割コンパイルしてリンクする)

⇒ 字句をどうやって共有するか？ ⇒ yaccが定数（整数）を決める

yaccのファイル

```
%{
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
%}

%token      NUM
%token      IF
%token      ID

%%
        中略

exp: NUM
           | exp '+' exp        { $$ = $1 + $3; }
           ;

%%
```

終端記号（トークン）を定義しておく

yaccがy.tab.cの中で定数を割り当てる

例）
#define NUM 257
#define IF 258
#define ID 259

256まではASCIIで使うので257以降

lexのファイル

```
%{
#include <stdio.h>

int
yywrap(void)
{
  return 1;
}
%}


%%
[\t ]+ {}
[0-9]+ {return NUM}
if {return IF}
[a-z][a-z0-9]* {return ID}

%%
```

対応するトークンの番号を返すように記述すれば、yyparseでトークンを識別して構文解析できる

if(stats版)の例

yaccソースファイル

```
%{
#include <stdio.h>
%}

%token IF
%token LPAR
%token RPAR
%token INC
%token ID
%token SC

%%
stats:  stat
        | stats stat
        ;

stat:   IF LPAR expr RPAR stat
{ printf("if¥n"); }
        | expr SC
        ;

expr:   ID
        | ID INC
        ;
```

トークン（終端記号）

続く・・・

lexソースファイル

トークンの番号をreturnする

```
%%
"if" {return IF;}
[n,v] {return ID;}
"(" {return LPAR;}
")" {return RPAR;}
"++" {return INC;}
";" {return SC;}
[¥t¥n ] {}
. {printf("ignored.¥n");}
%%
```

yaccソースファイル続き

```
%%
#include "lex.yy.c"

int main()
{
        return yyparse();
}

int yyerror(const char *s)
{
        printf("ERROR: %s¥n", s);
        return 0;
}
```

lex.yy.cを#include

以下のようにコンパイル
$ lex xxx.l
$ yacc xxx.y
$ gcc y.tab.c -ll

#includeすればマクロが共有できる。もし#includeしないなら、yaccで-dオプションつきでコンパイルすると、マクロ定義のファイルy.tab.hを出力してくれるので、それをlex.yy.cから#includeすればよい。lexではなくyaccの方で定数を割り当てる点に注意。

似て非なる用語の整理

字句・・・・字句解析用語。文を構成する単なる「文字列」。
終端記号・・・文法用語。構文木の葉に現れる「記号」。
トークン・・・コンパイラ用語。終端記号とその「値」のペア。
例） ”325”という字句は、終端記号NUMでその値が325

yaccが割り当てる定数とは無関係であることに注意

練習1 (lexとyaccの連携)

if(stats版)を入力して実行してみよ。還元に成功すればifと表示されるが、それ以外はなぜそのようなメッセージが出力されるか考察せよ。

例1
if (n) v;↵ （ ↵は改行）

式を入力したらCtrl-D

例2
if(n)v;if (v)n;↵

例3
iff(n)v; ↵

例4
if (n) ; ↵

例5
foo ↵

例6
foo; ↵

練習2 (lexとyaccの連携)

if(stats版)をif elseに拡張せよ。if elseへの還元成功のメッセージは"if else"にする。
以下の入力について、なぜそのタイミングでメッセージが出力されるか考察せよ。

例1
if (n) v; else v; ↵

例2
if (n) v; ↵
続けて
if (n) v; ↵

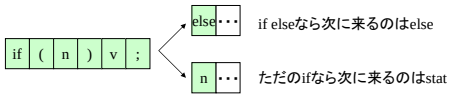
例3
if(n)v; ↵
if(n)v; ↵
n; ↵

例4
if(n)v; ↵
n ↵
; ↵

LALR(1)

(1)はトークンを1つ先読みしている、の意味

基本動作：入力を左端から読んでいって（シフト）、規則の右辺の条件がそろったら左辺の非終端記号に置き換える（還元）。その際、
トークンを常に1つ先読みし、確実に還元できることを確認している。
もしif (n) n;で還元してしまうと、続けてelseが来たとき失敗する。
そこで、if (n) n;の次のトークンを待って、elseありのifかelseなしのifかを判定している。



if elseなら次に来るのはelse

ただのifなら次に来るのはstat

先の例4で、識別子nを入力したところで還元されたのは、nを受け取った所でelseなしであることが確定したから。

練習3（lexとyaccの連携）

括弧つき四則演算の電卓プログラムを、lexを使ってトークンを取り込むように変更せよ。

練習4（代入文）

括弧つき四則演算の電卓プログラムに変数を導入する。そのために、
まず
a=3+5;
といった代入文を解析できるように文法を拡張せよ。

・代入の'='はEQという終端記号に、';'はSCという終端記号にする。
・代入文を表す非終端記号はstatにする。
・stat = ～ { } といった規則により、statのアクション部分で代入された式の値を出力するようにせよ。
・変数名はaからzまでの26種類とし、IDという終端記号として正規表現に追加する。
・非終端記号lineは不要なので関連する規則を削除する。
・式の右辺で変数を使うことはとりあえず考えなくてよい。つまり、a=b+1;のような代入はここではまだ考えなくてよい。

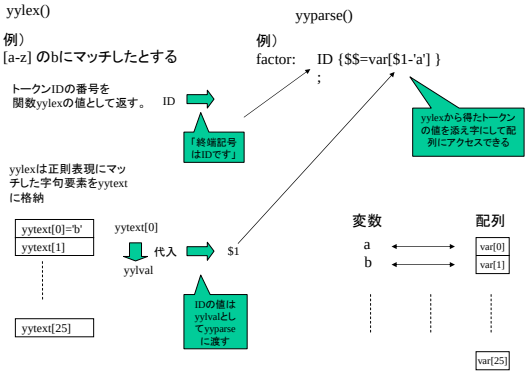
構文解析器が正しいことを確認せよ。

例）
a=3+1;
3+1
4+5;
d=a+1;

練習5（変数）

式の中で実際に変数を使えるようにする。例えば、
a=3+5;
のような代入の後、
b=a*2;
のような代入をしたい。以下の方針で文法を拡張せよ。

（方針）
int型の配列、int var[26]; を宣言し、アクションの中で代入すればよい。
その際、[a-z] {yylval=yytext[0]; return ID;}のように、yytextの先頭のASCIIコードをそのままyylvalに代入すれば、トークンの値として$xで受け取れる点に注意せよ。例えば、$1に引き継がれた変数にアクセスするには、配列の添え字をvar[$1-'a']とすればよいことに注意。



練習6（代入文）

このままでは、1つのstatしか処理できないので、複数の文(複文)をひとまとめにした非終端記号statsを追加して文法を拡張せよ。これにより、以下のような入力を受け付けるようになるようにせよ。

例）
以下続けて入力
a=3+1; ⇨ 4と出力
b=4-2*(5-4); ⇨ 2と出力
c=a+1; ⇨ 6と出力
d=a+b+c; ⇨ 11と出力

課題1（出力）

以下のような、式の値を出力するprintf文()を追加せよ。

```
printf(3+5);
```

```
8
```

この段階で、以下のような記述ができるようになることを確認せよ。

```
a=3;
b=2;
c=a*b;
printf(b+c);
```

課題2（pascal風に）

以下のような、pascal風の記述をするようにせよ。

```
program {
        a=3;
        b=2;
        printf(b+a);
}
```

課題3（比較演算）

if elseを追加したい。そのために、比較演算を導入する。==、!=、<、>、<=、>= が使えるように式を拡張し、真のとき1、偽のとき0がその式の値となるようにせよ。