計算機科学実験及演習 3（ソフトウェア）実験資料

1　実験の目標

この実験では，C 言語のサブセット言語である Tiny C のコンパイラを作成する．

Tiny C 言語は，型は `int` 型のみ，制御構造は `if` と `while` のみ，演算子も基本的なものしか持たない小さなプログラミング言語である．Tiny C 言語の構文については 4 節で示す．意味は原則として C 言語のものに従うとする．例えば，次は Tiny C のプログラムである．

```
% cat test.tc
int f(int x) {
    while(x > 1) {
        x = x - 2;
    }
    return x;
}
```

本実験では，標準入力から Tiny C のプログラムソースを受け取って，標準出力からアセンブリ言語 NASM [3] のコードを返すようなコンパイラ tcc を C 言語によって作成する．上のプログラムをコンパイルした結果は，例えば次のようになる．

```
% tcc < test.tc
        GLOBAL    f
f       push      ebp
        mov       ebp, esp
L1      mov       eax, [ebp+8]
        cmp       eax, 1
        setg      al
        movzx     eax, al
        cmp       eax, 0
        je        L2
        mov       eax, [ebp+8]
        sub       eax, 2
        mov       [ebp+8], eax
        jmp       L1
L2      mov       eax, [ebp+8]
        mov       esp, ebp
        pop       ebp
        ret
```

関数呼出，引数渡しなどの仕様を gcc などの通常の C コンパイラのものに従うことにより，普通の C プログラムと組み合わせて実行することが可能なものになる（はず）である．

コンパイラ tcc は，字句解析部，構文解析部，意味解析部，コード生成部の四部分から成る（ただし，構文解析と意味解析は同時に行なう）．このうち，字句解析部と構文解析部は，それぞれ lex と yacc（正確には，flex と bison）というツールを利用する．

> [注意] 本資料の解説はあくまでも一つの方法を提示しただけなので，Tiny C コンパイラとして正しく動作するものを作成すれば，本資料の方法に従う必要はない．

2 Lex

[教科書 2.6 節も参照すること]

2.1 概要

最初に，lex を用いた字句解析部の作成について解説する．字句解析器は，構文解析器から呼び出される形で利用されるので，構文解析器を作成するためのツールである yacc の概要も同時に説明する．

> [注意] 本実験では，lex を高速化した flex と，yacc の上位互換である bison を利用する．

yacc が生成した構文解析器は `yyparse()` という名前の C の関数として定義されている．構文解析時には，入力されたソースの先頭からトークンを取得するために，`yyparse()` の内部で字句解析を行なう関数 `yylex()` を呼び出す．関数 `yylex()` の働きは，入力ソースの先頭からパターンに適合する字句要素を切り出し，そのパターンに対応する**アクション**を実行する．コンパイラ作成において典型的なアクションは，その字句要素を構文解析に適したデータ（トークン）として構文解析器に返すことである．この `yylex()` を機械的に作成するツールが flex である[*1]．

2.2 実際に使ってみる

例として，次の lex ファイル `calc.l` から字句解析器を作成してみよう．

```
%option noyywrap
%{
int yylval;
%}
digit               [0-9]
%%
{digit}+            {
                        yylval = atoi(yytext);
                        printf("INT = %d\n", yylval);
                    }
"+"|"*"             {
                        printf("OPR = %s\n", yytext);
                    }
[ \t\n]             ; /* スペース，タブ，改行は無視 */
.                   ;
%%
main()
```

[*1] yacc が生成するパーサの C ファイルでは `yylex()` の宣言のみ行なわれ，定義（本体の実装）はされない．

```
{
    yylex();
}
```

この lex ファイルを flex にかけると，yylex() の定義を含む，lex.yy.c というファイルが生成される．これをコンパイルして，サンプルファイルを入力すると，以下のようになる．

```
% flex calc.l
% gcc -o calc lex.yy.c
% cat sample
1*23+456
% ./calc < sample
INT = 1
OPR = *
INT = 23
OPR = +
INT = 456
```

2.3 lex ファイルの説明

lex ファイル calc.l は 3 つのセクションから構成される．セクションは %% で区切られる．

2.3.1 定義セクション

最初のセクションは定義セクションと呼ばれ，インクルードファイルの指定や，マクロ定義を行なうことができる．定義セクションの ‘%{’ と ‘%}’ で囲まれた部分は，生成される lex.yy.c の先頭にそのまま挿入される．マクロ定義のフォームは次の通りである．

name definition

定義したマクロは “{*name*}” という形式で利用する．例えば calc.l に現れる “{digit}” は digit マクロの利用であり，これは “([0-9])” と展開される．

2.3.2 ルールセクション

2 つめのセクションではルールの記述を行なう．ルールのフォームは次の通りである．

pattern action

pattern は表 1 に示すような正規表現を記述する．*action* には，入力文字列から *pattern* に一致する文字列を切り出したときの動作を C 言語の文として記述する．yylex() が呼び出されると，与えられたパターン *pattern* に一致する最長の文字列が見つかるまで，文字が読み込まれる．最長の文字列と一致するパターンが複数個存在する場合は，lex ファイルのより先頭に近いパターンが選択される．一致するパターンが決まると，対応するアクション *action* が実行される．パターンに一致した文字列は，アクション中で yytext という大域変数によって参照できる．ただし，字句要素を切り出す度に yytext の内容は書き変わるので注意が必要である．

action の実行を終えると，引き続き（yylex() の実行を終えずに） *pattern* に一致する次の字句要素を残りの入力文字列から探す．yylex() は以上の動作を入力文字列の終りまで繰り返す．

表 1　pattern の例

pattern	一致する文字列
abc	abc
abc*	ab, abc, abcc, abccc, ...
abc+	abc, abcc, abccc, ...
a(bc)+	abc, abcbc, abcbcbc, ...
a(bc)?	a, abc
[abc]	a, b, c
[a-z]	a, b, c, ..., x, y, z
[a\-z]	a, -, z
[-az]	-, a, z
[A-Za-z0-9]+	アルファベットと数字からなる 1 文字以上の文字列
[\t\n]+	1 文字以上の whitespace
[^ab]	a と b を除く文字
[a^b]	a, ^, b
[a|b]	a, |, b
a|b	a または b
"[a-z]+"	[a-z]+
.	newline を除く任意の文字

2.3.3　C コードセクション

最後のセクションは，そのまま lex.yy.c の最後に挿入される．コンパイラ作成時には，このセクションは利用しない．

2.4　コンパイラにおける字句解析器の働き

calc.l のアクションでは，切り出した字句要素を表示するだけであったが，yacc を用いたコンパイラの作成では，切り出した字句要素を構文解析に適したデータ（**トークン**）に適宜変換して（例えば識別子なら識別子名を表す文字列，数値ならその値等），それを構文解析器に渡す必要がある．トークンの値の他にトークンの種類（識別子，数値，etc.）も渡す必要がある．次節で述べるが，構文解析器はトークンの種類を表す値を終端記号として扱う．

yacc が生成する構文解析器では，トークンの値は yylval という大域変数を通して渡し，トークンの種類は字句解析の関数 yylex() の返り値として渡すことを想定している．トークンの種類は，yacc ファイルの中で定義されており，実際は整数定数である．この定数は ASCII コードを避けた値を割当てられているので，char 型の値（文字）はそのままトークンの種類として使用できる．yacc ファイルで定義されたトークンの種類は，yacc が生成するヘッダファイル中で定義されているので，これを lex ファイルでインクルードすれば，トークンの情報を共有することができる．

このような仕様の yylex() を実装するには次のようにすればよい．まず，定義セクションの中でトークン情報を含むヘッダファイル（yacc が生成したもの）をインクルードする．次に，各ルールのアクション中で yylval にトークンの値を代入する．次に return 文によってトークンの種類を返すコードを記述する．return 文の実行によって yylex() の実行は終了するが，次回 yylex() を呼び出すと次の字句要素を残りの入力文字列から探す動作を再開するようになっている．なお，yylex() の返値の型（トークンの種類を表

す値の型）は int である．

例えば，calc.l の例を考える．yacc ファイル中で，トークンの種類として，整数値を表す Integer が定義されているとし，このトークンの定義は calc.tab.h というヘッダファイル中で定義されているとする．このとき，上で説明したような働きをする yylex() は，次のような lex ファイルによって生成できる．

```
%option noyywrap
%{
#include "calc.tab.h"
%}
digit               [0-9]
%%
{digit}+            {
                        yylval = atoi(yytext);
                        return Integer;
                    }
"+"|"*"             {
                        return *yytext;
                    }
[ \t\n]             ; /* スペース，タブ，改行は無視 */
.                   yyerror("Error:  invalid character");
%%
```

数字の列を切り出したらそれを整数値に変換しトークンの値として yylval に代入する．そしてトークンの種類として Integer を返す．“+”, “*” を切り出したときのアクションは，その文字自身をトークンの種類として返す．yylval へ何も代入していないのは，この場合は種類が分かれば十分だからである．yyerror() は，想定しない字句要素があったときにエラーを発生させるための関数である． yylex() は構文解析器の中からのみ実行されることを想定しているため，C コードセクションには何も書かない．

3 Yacc

[教科書 4.4 節，4.5.3 節，4.6.3 節も参照すること]

3.1 実際に使ってみる

まず，2 節で作成した calc.l を字句定義とする構文解析器を yacc を使って作成してみよう．ここでは，構文解析の結果を，コンパイラのように抽象構文木として出力するのではなく，電卓のように数式を計算した結果を表示するようにしてみる．

受理する構文の定義は，次のようなものである．

program:
　　expression

expression:
　　multiplicative-expr
　　multiplicative-expr + *expression*

multiplicative-expr:
　　constant
　　constant * *multiplicative-expr*

この構文で定義される式（*program*）を受理し，計算結果を返すプログラムは以下の yacc ファイル calc.y で

定義できる[*2].

```
%{
#include <stdio.h>
%}
%token Integer
%%
program:
        expr                    { printf("%d\n", $1); }
        ;
expr:
        mult_expr               { $$ = $1; }
        | mult_expr '+' expr    { $$ = $1 + $3; }
        ;
mult_expr:
        Integer                 { $$ = $1; }
        | Integer '*' mult_expr { $$ = $1 * $3; }
        ;
%%
int yyerror(char *s) {
    fprintf(stderr, "%s\n", s);
    return 0;
}
main() {
    yyparse();
}
```

これを以下のように bison にかけると，トークンの定義などが記述されたヘッダファイル `calc.tab.h` と，構文解析を行なう関数が定義された C のソースファイル `calc.tab.c` が生成される．`calc.tab.h` は，lex ファイル `calc.l` の中でインクルードされているものである．bison の-d オプションは，このヘッダファイルを出力するためのものである．

```
% bison -d calc.y
% flex calc.l
% gcc -o calc lex.yy.c calc.tab.c
% cat sample
1+2*3+4
% ./calc < sample
11
```

[ミニ課題] 上の例で，「`1+2*3+4`」の足し算と掛け算の優先順位が何故正しく反映されているのか，考察せよ．

3.2 yacc ファイルの説明

yacc への入力ファイル（yacc ファイル）も lex ファイル同様，`%%` で区切られた 3 つのセクションからなる．

[*2] yacc ファイル中の非終端記号には `mult-expr` などのようなハイフン '-' は使えない．

3.2.1 定義セクション

最初のセクションでは，トークンの種類や非終端記号の宣言，終端記号の優先順位の宣言等を行なうことができる．lex ファイル同様，‘%{’ と ‘%}’ で囲まれた部分は，生成される構文解析プログラム（C ファイル）の先頭に挿入される．

calc.y 中の “%token Integer” は Integer というトークンの種類名の宣言である．トークンの種類名を表すのに小文字を用いることもできるが，慣例に従い全て大文字で表す．%token によって宣言されたトークンの種類名に対しては，yacc によって自動的に ASCII コード以外の適当な整数値（int 型の値）が割り当てられる．この割り当ては，yacc が生成するヘッダファイル calc.tab.h の中で定義されるので，これを lex ファイルの中でインクルードすれば，Integer などのトークンの種類名を lex ファイルの中でも共有することができる．

> [ミニ課題] 実際に生成されたヘッダファイルの内容を確認し，Integer に整数が割当てられていることを確認せよ．

yacc の構文解析の処理において，yylex() の返値（トークンの種類の値）は終端記号として扱われる（以降トークンの種類のことを単に終端記号と呼ぶことがある）．yacc ファイル上では，終端記号をトークンの種類名あるいは ’*c*’ という形（*c* は任意の文字）で表現する．’*c*’ を終端記号として使う場合は何ら宣言しなくてもよい．%token で宣言されたトークンの種類の値として yacc が ASCII コードを表す整数値を割り当てないのは，この方法による終端記号を利用可能にするためである．2 節の最初の例では，パターン "+"|"*" のアクション “return *yytext;” がこの種の終端記号を返し，yacc ファイル上ではそれぞれ ’+’，’*’ と表現する．

3.2.2 ルールセクション

2 つめのセクションには，生成規則を記述する．生成規則の形式は次の通りである．

nonterminal: *rule*$_1$ *action*$_1$
| *rule*$_2$ *action*$_2$
...
;

これは非終端記号 *nonterminal* の生成規則の定義であり，BNF に似た記法で記述する．空 ε を表現するには *rule*$_i$ を省略すればよい．*rule*$_i$ は，終端記号（トークンの種類名または ’*c*’ で表される）または非終端記号を任意個並べて表現する．なお，このセクションで最初に定義される非終端記号が開始記号となる．

action$_i$ は *nonterminal* へ還元されるときの動作であり，C 言語の複文（‘{’ と ‘}’ で囲んだ文/複文の列）を記述する．実際には，アクションは *rule*$_i$ 中にも記述することができる．例えば，

```
A: B {... $1 ...} C {... $3 ...}
```

がそうであるが，これは

```
A: B X C {... $3 ...}
X: {... $0 ...}
```

と等しい（$*n* については後述）．ここで X の *rule* は空である．

yacc が生成するパーサは LALR(1) 構文解析を行なう．例えば以下の生成規則に対して，入力 ‘1-2+3’ が与えられると次のように構文解析される（字句解析プログラムとして 2 節の最初の例の lex ファイルを用いるものとする）．

```
expr:
```

```
        Integer                 { $$ = $1; }
        | expr '+' Integer      { $$ = $1 + $3; }
        | expr '-' Integer      { $$ = $1 - $3; }
        ;
```

```
·1-2+3
Integer·-2+3          シフト
expr·-2+3             還元
expr-·2+3             シフト
expr-Integer·+3       シフト
expr·+3               還元
expr+·3               シフト
expr+Integer·         シフト
expr·                 還元
```

アクションに現れる $ は，還元時に生成される値の格納場所である．また，n は $rule_i$ の n 番目の記号（終端記号または非終端記号）に対して得られた値を表す．n 番目の記号が終端記号の場合，n の値は終端記号を読み込んだ時点の大域変数 yylval の値である．

■例：

```
expr:  expr '+' Integer         { $$ = $1 + $3; }
      ;
```

この例では，右辺の expr の値 $1 と Integer の値 $3 の和を expr に還元するときの値（還元された expr の値）とする．$3 の値は，Integer のトークンを読み込んだときの yylval の値が使われる．2 節の最初の例のように通常は yylex() が一つの字句要素を切り出したときに yylval に何らかの値を代入するようにする．

3.2.3 C コードセクション

yacc ファイルの最後のセクションは，C 言語のコードを記述することができる．この部分は，yacc が生成する C ファイルにそのまま出力される．上の例で定義されている関数 yyerror() は構文エラーが生じたときにパーサが（自動的に）呼び出す関数である．

■課題 1　calc.l と calc.y を作成し，実際にコンパイル，実行してみよ．

```
% bison -d calc.y
% flex calc.l
% gcc -o calc calc.tab.c lex.yy.c
% cat sample
1+2*3+4
% ./calc < sample
11
```

■課題 2　課題 1 の yacc ファイルの expr に引き算を追加せよ．ここで，

```
expr:
        mult_expr               { $$ = $1; }
        | mult_expr '+' expr    { $$ = $1 + $3; }
        | mult_expr '-' expr    { $$ = $1 - $3; }
```

;

とすると，結果は普通の引き算の意味とは異なるものとなる．例として入力 `1-2-3` を与えて確認せよ．また，その理由を考察せよ．さらに，普通の引き算の意味を反映するように修正せよ．

[ヒント] 普通，引き算は「左結合」であると考えられる．つまり，上の入力例は `(1-2)-3` と解釈されるのが普通である．これに対し，上の生成規則のような「右再帰」で定義される結合子は「右結合」として定義される．

■課題 3 以下の仕様を持つ Tiny C のパーサを作成せよ（アクションは空とする）．`else` は C 言語同様，最も近い if 文に結び付くものとする（これについてはこの課題直後の説明を参照せよ）．*identifier* は英字，数字と下線記号 '_' の列であり，最初の文字は英字とする．大文字と小文字は区別すること．*constant* は数字の列である．終端記号の区切りは任意個のスペース，タブまたは改行とする．なお，この課題では lex ファイルにおいて `yylval` への代入は必要ない．

program:
　external-declaration
　program external-declaration

external-declaration:
　declaration
　function-definition

declaration:
　`int` *declarator-list* **;**

declarator-list:
　declarator
　declarator-list **,** *declarator*

declarator:
　identifier

function-definition:
　`int` *declarator* **(** *parameter-type-list*$_{opt}$ **)** *compound-statement*

parameter-type-list:
　parameter-declaration
　parameter-type-list **,** *parameter-declaration*

parameter-declaration:
　`int` *declarator*

statement:
　;
　expression **;**
　compound-statement
　`if` **(** *expression* **)** *statement*
　`if` **(** *expression* **)** *statement* `else` *statement*
　`while` **(** *expression* **)** *statement*
　`return` *expression* **;**

compound-statement:
　{ *declaration-list*$_{opt}$ *statement-list*$_{opt}$ **}**

declaration-list:
　declaration

declaration-list declaration

statement-list:
statement
statement-list statement

expression:
assign-expr
expression , *assign-expr*

assign-expr:
logical-OR-expr
identifier = *assign-expr*

logical-OR-expr:
logical-AND-expr
logical-OR-expr || *logical-AND-expr*

logical-AND-expr:
equality-expr
logical-AND-expr && *equality-expr*

equality-expr:
relational-expr
equality-expr == *relational-expr*
equality-expr != *relational-expr*

relational-expr:
add-expr
relational-expr < *add-expr*
relational-expr > *add-expr*
relational-expr <= *add-expr*
relational-expr >= *add-expr*

add-expr:
mult-expr
add-expr + *mult-expr*
add-expr − *mult-expr*

mult-expr:
unary-expr
mult-expr * *unary-expr*
mult-expr / *unary-expr*

unary-expr:
postfix-expr
− *unary-expr*

postfix-expr:
primary-expr
identifier (*argument-expression-list*$_{opt}$)

primary-expr:
identifier
constant
(*expression*)

argument-expression-list:
assign-expr
argument-expression-list , *assign-expr*

lex ファイルの雛形:

```
%option noyywrap
%option yylineno
%{
#include "filename.tab.h"
%}
%%
...
%%
```

yacc ファイルの雛形:

```
%{
%}
%error_verbose

%token Integer Identifer ...
...
%%
program:
...
%%
extern int yylineno;

int yyerror(char *s) {
    fprintf(stderr, "%d: %s\n", yylineno, s);
    return 0;
}
main() {
    yyparse();
}
```

文法定義に conflict が存在する場合，yacc は次のようなメッセージを表示する．

```
% bison -d calc.y
calc.y contains 1 shift/reduce conflict.
```

shift/reduce conflict は，あるトークンを読み込んだときにシフトと還元の両方が可能な場合に生じる．reduce/reduce conflict はあるトークンを読み込んだときに 2 通り以上の還元が可能な場合に生じる．yacc は shift/reduce conflict が生じた場合，シフトすることを優先する．例えば課題 3 では，‘else’ を読み込んだときに if-else 文として ‘else’ をシフトするか，else 節を含まない if 文として還元するかの 2 通りのパースが可能である．C 言語の if 文と同じ意味にするのであれば else はシフトすればよいので，この if 文に関する conflict は無視して構わない．一方，reduce/reduce conflict が生じた場合，yacc は適用可能なルールの中で最初に現れるルールを選んで還元するが，これに頼ると可読性を損ない，また誤りを含み易いので reduce/reduce conflict は解消するのが望ましい．具体的にどのように conflict が生じているかを調べるには，yacc (bison) の実行時に -v というオプションを指定し，そのオプションによって生成された *filename*.output というファイルを調べればよい．

■課題 4　課題 3 で作成したパーサを用いていくつかのソースコードのパースを試みよ（構文的に誤りのないソースコードに対しては何も起こらない（表示されない）はずである）．構文的に誤りを持つソースコードを

与えた場合の動作についても何が起きるか確認せよ．

実行例:

```
% cat test.tc
int fact(int x)
{
    int z;

    z = 1;
    while (x >= 1) {
        z = z * x;
        x = x - 1;
    }
    return z;
}
% ./tcc < test.tc
% （構文的に正しいので何も起こらない）
```

■課題 5 [選択課題] yacc の `error` トークンを用いて課題 3 で作成したパーサがエラーリカバリの処理をするように拡張せよ．そして課題 4 で用いた構文的に誤りを持つソースコードに対してエラーリカバリがされるかを確かめよ．`error` トークンを加えた際の shift/reduce conflict や reduce/reduce conflict に注意すること．

4 構文木 (Syntax tree)

[教科書 4.1 節も参照すること]

この節では，前節で作成した構文解析器のアクション部分を実装する．コンパイラにおいて構文解析器はソースプログラムを受け取り，構文的に正しければ，**構文木** (syntax tree) を出力する．この構文木の作成と出力をアクション部で行なう．

4.1 構文木のデータ構造

構文木は，ソースプログラムの内部表現である．入力であるソースプログラムは単にトークンを表す文字と空白文字の羅列に過ぎなかったが，構文木はソースプログラムの構造を反映した木構造を持っているので，構文解析の後に行なわれる処理（意味解析やコード生成，最適化）に適した形をしている．

本実験では，構文木の各ノードは，定数を表す**定数ノード**，識別子を表す**トークンノード**または子要素を持つ **N 組** (N-tuple) とする．このうち，定数ノードとトークンノードが構文木の葉となる．各ノードの型は構造体として定義し，構文木の型はこれらのノードのいずれかの値をもつ共用体として定義する．以下で，これらのデータ構造の定義と，データ生成関数のプロトタイプの例を示す．

> [注意] 以下の一連の型やプロトタイプの定義は，yacc ファイル，lex ファイル両方のコンパイルで必要になるので，別の型定義用のヘッダファイルとして用意し，yacc ファイル，lex ファイルの中で読み込むようにするとよい．

型定義の例:

```
typedef struct c {      /* constant node */
    int op;
    int v;
} *constant;
```

```
typedef struct tk {      /* token node */
    int op;
    char *name;
} *token;
typedef struct tp {      /* n-tuple (n=4) */
    int op;
    union nd *a[4];
} *tuple;
typedef union nd {       /* tree */
    struct {
        int op;
    } n;
    struct tp tp;
    struct tk tk;
    struct c c;
} *tree;
```

データ生成関数のプロトタイプの例:

```
tree make_tuple(int, tree, tree, tree, tree);
tree make_token_node(char *);
tree make_constant_node(int);
```

struct tp において，a は枝を表すポインタとして使用する．op はそのノードの種類を表す int 型の値である．共用体 union nd の全てのメンバは先頭に共通のメンバ op を持っている．C 言語ではこのような場合，op を共用体に属する任意の構造体のメンバとして参照することが許されている．共用体のメンバ n はこの op を参照するためのラベルである．例えば，t を構文木，すなわち tree 型のデータとするとき，その木の根ノードの種類に応じた処理を行なうためには以下のようにすればよい．

```
switch(t->n.op) {
case Integer:
    ...定数ノードに対する処理...
case Identifier:
    ...トークンノードに対する処理...
...
}
```

op の値は，構造体やノードの種類によって異なるようにする必要がある．この値としては，構文解析時の終端記号（Integer や'+'）を使うことができる．終端記号としては使わないが，op の値として使いたい名前（例えば，以下に現れる CONS など）も，yacc ファイルの %token 部で宣言しておけばよい．

[注意] 関数本体の定義を別のソースファイルで行なう場合，各記号への整数値の割当てを行なっているヘッダファイル（*filename*.tab.h）のインクルードが必要である．

N 組を用いた構文木の例として，

```
int f(int x, int y)
{
    int a, b, c;
    statement-list
    statement
}
```

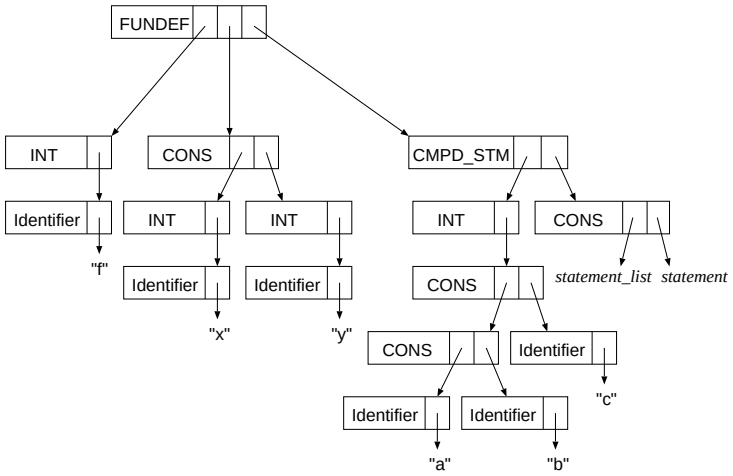

図 1　構文木の例

の構文木を図 1 に示す．図においてそれぞれの箱は N 組を表している．箱の一番左側のフィールドは節の値であり構造体のメンバ `op` に対応する．それ以外のフィールドは構造体のメンバ `a[`n`]` に対応する．なお，この例では様々なサイズの N 組が使われているが，同じサイズ（例えば 4 つ組）で統一して構わない（子のない枝の部分は `NULL` などにしておけばよい）．

節の値 `CONS` は，節の値は特に必要としておらず単に 2 本の枝を持つ節を表現するときに使用している．例えば Tiny C では，*external-declaration* のリストである *program* の構文木を表現するのに `CONS` 節を使用する．この場合の `CONS` 節の 2 本の枝は，*program* の生成規則の右辺である *program* と *external-declaration* のそれぞれに対する構文木を指す．更に，この `CONS` 節が指す *program* の構文木も `CONS` 節を用いて（再帰的に）表現される．*statement* のリストである *statement-list* なども同様に `CONS` 節を用いて表現することができる．

4.2　構文木の生成

パーサで構文木を作成するには，構文木を生成するコードをアクションとして記述すればよい．例えば課題 3 の *add-expr* の部分の構文木の生成は次のように記述できる．

```
add_expr:
   mult_expr
       { $$ = $1; }
   | add_expr '+' mult_expr
       { $$ = make_tuple('+', $1, $3, NULL, NULL); }
   | add_expr '-' mult_expr
       { $$ = make_tuple('-', $1, $3, NULL, NULL); }
   ;
```

`yylval` の値が終端記号の値として用いられることや，yacc ファイルのアクション中では終端記号または非終端記号の値を `$$` や `$`n で参照できることを述べたが，これらの値のデフォルトの型は `int` である．これら記号の値の型はマクロ `YYSTYPE` で定義されており，デフォルトは

```
#define YYSTYPE int
```

である．`YYSTYPE` は，yacc ファイルの最初のセクションの ‘`%{`’ と ‘`%}`’ で囲まれたところでユーザが定義してもよい．

上の `add_expr` の例では，記号の型として `tree` を仮定しているので，次のように記号の型を `tree` とする必要がある．

```
%{
#define YYSTYPE tree
%}
```

この場合，全ての記号の型が tree になるが，もし以下のような生成規則とアクションを考えるならば，Identifier や Integer の値の型（$1 の型）は char * や int となるべきである．

```
primary_expr:
        Identifier
            { $$ = make_token_node($1); }
        | Integer
            { $$ = make_constant_node($1); }
        | '(' expression ')'
            { $$ = $2; }
        ;
```

この生成規則と上の add_expr を共に扱うとすれば，記号の型として複数の型を扱う必要がある．このような場合は YYSTYPE を使用する代りに以下のように %union 宣言すればよい（%union 宣言した場合は YYSTYPE の定義は不要である）．%union 宣言することで yylval や記号の型として複数の型を利用することができる．

```
%union {
    int i;
    char *str;
    tree n;
}
```

%union 宣言した場合は，yacc ファイルの最初のセクションで記号の型を以下のように宣言しておく必要がある．

■yacc ファイルでの非終端記号の型の宣言：

```
/* 非終端記号 add_expr と mult_expr は tree 型 */
%type <n> add_expr mult_expr
```

■yacc ファイルでの終端記号の型の宣言：

```
/* 終端記号 Identifier は char *型 */
%token <str> Identifier
```

また，lex ファイルのアクションにおける yylval への代入時には次のようにして型を指定する必要がある．

```
yylval.i = atoi(yytext); /* yylval を int 型として使用 */
yylval.str = strdup(yytext); /* yylval を char *型として使用 */
```

[注意] 識別子 *identifier* の値（yylval.str の値）として識別子名を表す文字列へのポインタを返す場合，yytext が指す文字列をどこかへコピーし，コピーした文字列へのポインタを yylval.str の値としなければならない．yytext が指す内容はトークンを読み込むたびに更新されるからである．文字列をコピーするのにライブラリ関数 char *strdup(char *)[*3] を利用してもよい．

[*3] ヘッダファイルは string.h

4.3 構文木の表示

本実験では，構文木からコードを生成する前に，構文木が正しく生成できていることを確認するために，構文木を表示するプログラムを作成する．

構文解析が無事終了した場合，生成された構文木は，非終端記号 *program* の値として得られている．従って，構文解析後に表示ルーチンを呼び出すには，次の生成規則をルールセクションの先頭に加えればよい．

```
main:
        program
            { if (yynerrs == 0) print_program($1); }
        ;
```

変数 yynerrs は yacc が宣言する変数であり，これまでに何回構文エラーが生じたかを記憶している．

print_program() は例えば次のように実装すればよい．

```
void print_program(tree p) {
    if (p->n.op != CONS) {
        print_external_declaration(p);
        printf("\n");
    } else {
        print_program(p->tp.a[0]);
        print_external_declaration(p->tp.a[1]);
        printf("\n");
    }
}
```

■課題 6　課題 3（または課題 5）で作成したパーサを拡張して構文木を生成するようにし，生成した構文木を次のように表示するプログラム（基本構造は先順の木の走査を行なうプログラム）を作成せよ．そして構文木が正しく生成されていることを，特に優先順位と結合性について確認せよ．また，課題 3 で示した *expression* の生成規則から，演算子の優先順位と結合性を反映した構文木が作成できる理由を説明せよ．

```
% cat test1.tc
int gcd(int a, int b)
{
    if (a == b) return a;
    else if (a > b) return gcd(a-b, b);
    else return gcd(a, b-a);
}
% ./tcc < test1.tc
((int gcd) ((int a)(int b))
(
  (IF (== a b)
    (RETURN a)
    (IF (> a b)
      (RETURN (FCALL gcd (- a b) b))
      (RETURN (FCALL gcd a (- b a)))
    )
  )
))
```

[注意] この表示例はあくまでも一例であり，このスタイルに従う必要はない．インデントや改行などの工夫も必須ではない．ただし，構文木が正しく生成されていることを正確に確認できるような表示方法にしなければならない．

5　意味解析とコード生成のための情報収集

[教科書 5 章も参照すること]

課題 6 の構文解析器は，構文的なエラーはチェックしているが，意味上のエラーについてはチェックしていない．また，この構文解析器が生成する構文木のままでは，コード生成のための情報が不足している．本節では意味解析によって意味上のエラーをチェックし，コード生成のために必要な情報を収集する．具体的には，

- 名前とオブジェクト（変数と関数）の対応づけ
- オブジェクト情報の収集
  - パラメータ，局所変数の値を格納する位置（ベースポインタからの相対番地）の決定
  - オブジェクトの種類（パラメータか，局所変数か，関数か）
  - 関数の引数の個数 など...
- 関数や局所変数の重複定義のチェック

などを行なう（Tiny C は `int` 型しか持たないので型チェックは扱わない）．これらのほとんどは，構文解析時に並行して行なうので，yacc ファイルのアクション部に記述することになる．ただし，局所変数の相対番地については，コード生成時に必要となる一時変数の扱いも考慮しなければならないので，構文解析終了後（構文木表示時，または，コード生成時）に行なう．

5.1　オブジェクト情報の収集

前節までは，オブジェクト（Tiny C の場合，変数と関数）の情報としてその名前のみを扱った．ここでは，コード生成に必要なその他の情報の取得と管理の方法について述べる．

名前とオブジェクトの対応づけは，教科書 [1] の 5.4 節の方法で行なえばよい．このスタックを実現するためにトークンノード（`struct tk`）の構造を次のように拡張し，スタックをリストとして実装することにする（以降，`token` 型の構造体をオブジェクト構造体と呼ぶことがある）．

```
typedef struct tk {
    int op;
    struct tk *next;
    char *name;
    int lev;
    enum {FRESH, VAR, FUN, PARM, UNDEFFUN} kind;
    int offset;
} *token;
```

5.1.1　`lev`

`lev` はオブジェクトが宣言されたブロックのレベル（深さ）を表す．ブロックのレベルは，

- 大域変数と（大域）関数のレベルは 0
- 関数のパラメータ宣言のレベルは 1
- 関数本体のブロックのレベルは 2

- 関数本体中の入れ子ブロックのレベルは 3 以上（深さに応じる）

とする．現在のブロックレベルを保持する int 型の大域変数 cur_lev を導入しておき，make_token_node() でオブジェクト構造体を生成するときに，cur_lev の値を lev の値として記憶すればよい．cur_lev の値の更新は，パラメータリストの解析を始めるときと複文 (*compound-statement*) の変数宣言の解析を始めるときに 1 増やし，関数定義の解析を終えるときと複文の解析を終えるときに 1 減らせばよい．例えば，

```
compound_statement:
        '{'
                { cur_lev++; }
        declaration_list statement_list '}'
                { ··· cur_lev--; }
        ;
```

のようにすればよい．

> [注意] このように，ルールの中にアクションを混在させる場合，値スタックを参照するときの数字（$\$n$ の n）がずれるので注意すること．例えば上の場合，declaration_list の値は，$3 で参照される．

5.1.2 kind

kind はオブジェクトの種類を表す．それぞれの意味は，

- FRESH: make_token_node() で構造体を生成するときの初期値
- VAR: 変数
- FUN: 関数
- PARM: パラメータ
- UNDEFFUN: 未定義関数

例えば，生成したオブジェクトが変数宣言のところで現れていれば VAR へ更新する．未定義関数の呼出は，関数定義がその呼出より後ろに記述されている（定義される前に関数が使用される）か，ライブラリ関数を使用する場合に生じる．C 言語では，関数のプロトタイプによって未定義関数の情報（返値の型，パラメータの数，各パラメータの型）をコンパイラに知らせることができるが（Tiny C には関数の宣言はない），関数の宣言のない未定義関数であっても呼び出すことができる．Tiny C もこの仕様に従うとする．関数のオブジェクトが定義された場合は，そのオブジェクト構造体の kind の値を FUN に更新する．

5.1.3 offset

offset は二通りの用途で利用する．局所変数及びパラメータの場合は，その値の格納場所（関数フレーム内の相対番地; ベースポインタ ebp を基準とした番地）を保持するのに使用する．一方，オブジェクトが関数の場合は，パラメータの数を記憶するために使用する．なお本実験で作成するコンパイラは，関数フレームの構造として教科書の図 6.5 を採用する．

5.2 スタック操作関数の定義

構造体 token によるスタックの表現を，教科書の図 5.4(c) を例として，図 2 に示す．この図において symtab は token 型の大域変数であり，リストの先頭すなわちスタックのトップを指すのに使用する．

新しく生成されたオブジェクト構造体（token 型の構造体）をスタックに積む操作は，関数 make_token_node() 内で行なうことにする．この実装は図 3 のようにすればよい．

一方，現在のブロックレベルを 1 減らすとき（cur_lev-- を実行するとき），そのブロックで宣言された

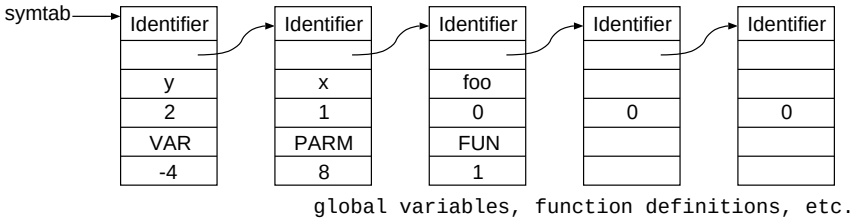

図 2　教科書の図 5.4(c) の 実装

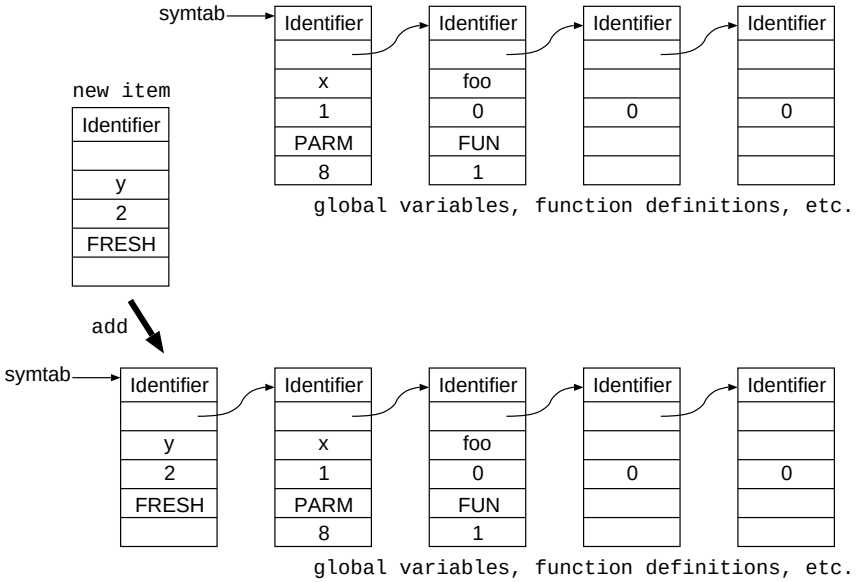

図 3　スタックへ積む操作の実装

オブジェクトはそのスコープを終えるので，それらのオブジェクト構造体をスタックからポップしなければならない．図 2 のスタックの実装の場合，スコープを終えるレベル値を持つ構造体をたどり，たどり終えた構造体の次の構造体を `symtab` で指すようにすればよい．

変数宣言や関数定義によって新しいオブジェクトを作成するとき，同一レベルで同じ名前のオブジェクトがすでに存在していれば二重宣言/二重定義エラーである．この意味的なエラーは構文解析では検出することができないが，名前とオブジェクトの対応付け処理において検出することができる．この準備として，変数宣言や関数定義において直ちに新しいオブジェクト構造体を生成するのではなく，まず同じ名前のオブジェクトがオブジェクト構造体のスタック上に存在するか調べる関数 `tree lookup_sym(char *)` を用意する．この関数は，名前を引数に取り，スタックを上から順に走査して，同名のオブジェクトが見つかればそれを（`tree` 型の値として）返し，なければ `NULL` を返す関数である．具体的には関数 `lookup_sym()` を定義し，以下のように呼び出せばよい．

```
if (!(yylval.n = lookup_sym(yytext)))
    yylval.n = make_token_node(yytext);
return Identifier;
```

ここで，`lookup_sym()` の返り値が `tree` 型であることに注意する．もし，終端記号 `Identifier` の型を `char *` 型にしている場合は変更が必要である．

5.3 意味解析部の実装

二重宣言等の意味的なエラーの検出や名前とオブジェクトの対応づけは（構文解析部に埋め込まれた[*4]）意味解析部で行なう．

具体的には，上で解説したとおり，識別子が現れたときに lookup_sym 関数を呼び出し，既にスタック中に同名のオブジェクトが存在すればそのオブジェクトに，なければ新しく作成したオブジェクトに注目し，意味解析を行なう．プログラム中で識別子が現れるのは，関数定義，関数呼出，パラメータ宣言，変数宣言，変数参照，の五種類であるから，それぞれの場合に応じて解析用の関数を用意する．

5.3.1 変数宣言

変数宣言については以下のようにすればよい．

- もし，その名前が初めて現れたものであれば（kind が FRESH のとき），kind を VAR に変更するだけでよい．
- 既にその名前が関数として定義されているか，未定義関数として使用されている場合，もし変数宣言されているのがレベル 0（すなわち大域変数として宣言されている）ならば，二重宣言であるからエラー．そうでなければ，変数として新たなオブジェクトを作成し，kind を VAR にして，名前をそのオブジェクトに対応させる．
- 既にその名前が変数として宣言されている場合，同じレベルで宣言されている場合は二重宣言であるからエラー．そうでなければ，変数として新たなオブジェクトを作成し，kind を VAR にして，名前をそのオブジェクトに対応させる[*5]．
- 既にその名前がパラメータとして宣言されている場合，新たなオブジェクトを作成し，kind を VAR にして，名前をそのオブジェクトに対応させる．この場合，パラメータが隠されてしまうので，警告を発する．

以上を行なう make_decl() を定義し，構文解析において，変数宣言部に対して次の関数を *declarator-list* へ還元するときのアクションとして実行する．

make_decl() の定義例:

```
tree make_decl(tree n)
{
    switch (n->tk.kind) {
    case VAR: /* すでに変数として名前が宣言されている */
        if (n->tk.lev == cur_lev)
            /* 同一レベルであれば二重宣言である */
            error("redeclaration of `%s'", n->tk.name);
        n = make_token_node(n->tk.name);
        break;
    case FUN: /* すでに関数として名前が定義されている */
    case UNDEFFUN: /* すでに未定義関数として名前が使用されている */
        if (n->tk.lev == cur_lev)
            /* 同一レベルであれば二重宣言である */
            error("`%s' redeclared as different kind of symbol",
                  n->tk.name);
        n = make_token_node(n->tk.name);
        break;
```

[*4] 意味解析部は字句解析部のように独立したモジュールの形態とならない．

[*5] 既に宣言されていた変数は，そのレベル以降では隠される．

```
        case PARM: /* すでにパラメータとして名前が宣言されている */
            warn("declaration of `%s' shadows a parameter", n->tk.name);
            n = make_token_node(n->tk.name);
            break;
        case FRESH:
            break;
        }
        n->tk.kind = VAR;
        return n;
    }
```

make_decl() の使用例:

```
declarator_list:
        declarator { $$ = make_decl($1);
                     /* $1 は前述の yylval.n の値 */ }
        ...
```

ここで，エラーと警告を発生する error() と warn() の定義は次のとおりである．

```
#include <stdio.h>
#include <stdarg.h>
int semnerrs;
extern int yylineno;
void error(char *fmt, ...)
{
    va_list argp;
    va_start(argp, fmt);
    semnerrs++;
    fprintf(stderr, "%d: ", yylineno);
    vfprintf(stderr, fmt, argp);
    fprintf(stderr, "\n");
    va_end(argp);
}
void warn(char *fmt, ...)
{
    va_list argp;
    va_start(argp, fmt);
    fprintf(stderr, "%d: warning: ", yylineno);
    vfprintf(stderr, fmt, argp);
    fprintf(stderr, "\n");
    va_end(argp);
}
```

error()（エラー表示関数）と warn()（警告表示関数）の違いは，変数 semnerrs をカウントアップするかどうかだけである．semnerrs は意味解析におけるエラーをカウントする．この変数の値が 1 以上となった場合は，コード生成を行なうことはできないが， 意味解析レベルのエラーリカバリを行なって構文解析／意味解析を続行することはできる．そのため error() も warn() 同様，コンパイラの実行を終了しないようになっている（warn() による警告を発した場合はコード生成はできる）．意味解析レベルのエラーリカバリとは，例えば make_decl() の場合は，二重宣言のときに make_token_node() によって新たに変数のオブジェクト構造体を生成することである．この後に続くコードでは，二重宣言された新しい変数を使ってプログラムが記述されていると予想されるからである．

5.3.2 パラメータ宣言

パラメータ宣言についても変数宣言と同様である．この関数はブロックレベル 1 のときにしか呼び出されないので，二重宣言が起こるのはパラメータどうしの場合に限られることに注意せよ．

```
tree make_parm_decl(tree n)
{
    switch (n->tk.kind) {
    case VAR:
    case FUN:
    case UNDEFFUN:
        n = make_token_node(n->tk.name);
        break;
    case PARM:
        error("redeclaration of ‘%s’", n->tk.name);
        return n;
    case FRESH:
        break;
    }
    n->tk.kind = PARM;
    return n;
}
```

5.3.3 関数定義

関数定義の場合も同様である．この関数はブロックレベルが 0 のときにしか呼び出されないため，二重宣言が起こるのは関数定義どうし，または大域変数宣言との間だけである．

```
tree make_fun_def(tree n)
{
    switch (n->tk.kind) {
    case VAR:
        error("‘%s’ redeclared as different kind of symbol",
              n->tk.name);
        break;
    case FUN:
        error("redefinition of ‘%s’", n->tk.name);
        break;
    case UNDEFFUN:
    case FRESH:
        n->tk.kind = FUN;
        break;
    }
    return n;
}
```

5.3.4 変数参照

変数やパラメータの参照時には，既に宣言された同名のオブジェクトが `lookup_sym()` によって見つかるはずである．さもなければ，それは未宣言の変数を参照しているのでエラーである．このため，変数参照に対する解析は以下のようになる．

- 既に変数またはパラメータとして宣言されている場合，そのオブジェクトを参照しているものとみなせばよいので何もする必要はない．
- 関数として定義されている，または未定義関数として使用されている場合，関数を変数として参照しているのでエラー．
- まだスタックに同名オブジェクトが存在しないとき（すなわち kind が FRESH のとき），未宣言変数を参照しているのでエラー．この場合，エラーリカバリのため，kind を VAR にしておく．

以上を実現する ref_var() の定義例と使用例は以下のとおりである．

ref_var() の定義例:

```
tree ref_var(tree n)
{
    switch (n->tk.kind) {
    case VAR:
    case PARM:
        break;
    case FUN:
    case UNDEFFUN:
        error("function ‘%s’ is used as variable", n->tk.name);
        break;
    case FRESH:
        error("‘%s’ undeclared variable", n->tk.name);
        n->tk.kind = VAR; /* エラーリカバリ */
        break;
    }
    return n;
}
```

ref_var() の使用例:

```
primary_expr:
        Identifier { $$ = ref_var($1); }
        ...
```

5.3.5 関数呼出

関数呼出の場合も変数参照の場合と同様である．ただし，既に関数として定義されていない名前で関数呼出を行なおうとしていても，それはエラーではなく，未定義関数の呼出であると解釈し，警告を発するようにする．このとき，新たに追加される未定義関数の名前は，スタックの上に積むのではなく，ブロックレベル 0 の位置に積まなければならない．これを実現するために，引数で与えられるオブジェクト構造体を大域関数を表すオブジェクトとして登録し直す関数 globalize_sym() を定義しなければならない．実装は図 4 のようにすればよい．

ref_fun() の定義例:

```
tree ref_fun(tree n)
{
    switch (n->tk.kind) {
    caseVAR:
    case PARM:
        error("variable ‘%s’ is used as function", n->tk.name);
        break;
    case FUN:
```

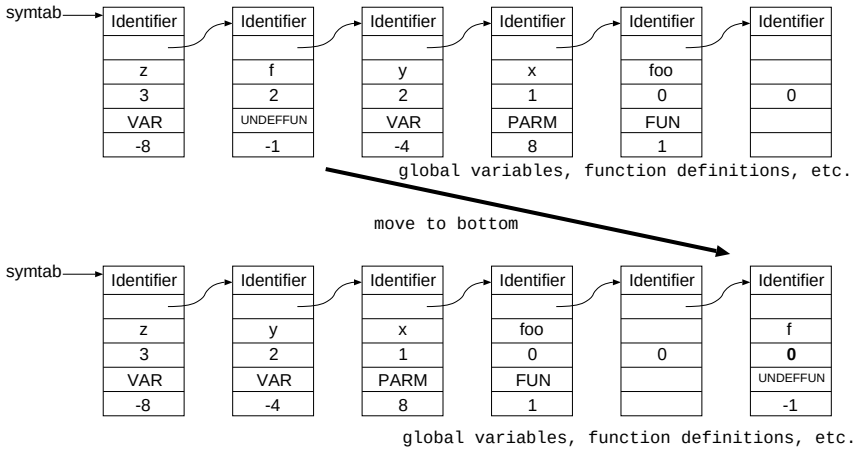

図 4　オブジェクト構造体の大域化の実装

```
            case UNDEFFUN:
                break;
            case FRESH:
                warn("`%s' undeclared function", n->tk.name);
                n->tk.kind = UNDEFFUN;
                if (n->tk.lev > 0) globalize_sym(n);
                break;
            }
            return n;
        }
```

`ref_fun()` の使用例:

```
    postfix_expr
            ...
            | Identifier '(' opt_argument_expression_list ')'
                    { ··· ref_fun($1) ··· }
            ...
```

■課題 7　課題 6 の Tiny C コンパイラ（構文木表示プログラム）の意味解析部を実装せよ.

- 本節で述べた意味解析の他に，関数呼出時の引数の数のチェックも行なうようにせよ.
- パラメータの置かれる相対番地を求めるようにせよ.
- 教科書 6.3 節及び 6.4.1 節の方法を用いて局所変数の置かれる相対番地を求めるようにせよ．相対番地の計算は，構文解析のアクション部ではなく，構文木表示部で行なうべきである.

  [注意] 局所変数の格納領域には，レジスタの値を一時的に退避しておくための一時変数も割り当てられるため，局所変数の格納場所は一般にはコード生成の段階で決まる.（パラメータの相対番地については，構文解析時に求めればよい.）
- 構文木表示プログラムにおいて，大域変数宣言，関数定義，パラメータ宣言，局所変数宣言，変数参照において各識別子を表示する際に，それぞれのブロックレベルを表示するようにせよ.
- さらに，パラメータ宣言と局所変数宣言の表示の際にそれぞれの相対番地を表示するようにせよ.

パース終了時の表示ルーチンを呼び出す部分は，次のように変更すればよい．意味解析におけるエラーが一つ以上見つかった場合も，表示ルーチンは呼び出されない.

```
main:
                { symtab = NULL; cur_lev = 0; }
        program
                { if (yynerrs == 0 && semnerrs == 0)
                      print_program($2); }
        ;
```

実行例:

```
% cat test.c
int x;
int f(int x, int y)
{
    int x;
    {
        int x, y;
        x+y;
        {
            int x, z;
            x+y+z;
        }
    }
    {
        int w;
        x+y+w;
    }
    x+y;
}
int g(int y)
{
    int z;
    f(x, y);
    g(z);
}
% ./tcc < test.c
4: warning: declaration of ‘x’ shadows a parameter
6: warning: declaration of ‘y’ shadows a parameter
(int x:0)
((int f:0) ((int x:1:8)(int y:1:12))
(
  (int x:2:-4)
  (
    (int x:3:-8 y:3:-12)
    (+ x:3 y:3)
    (
      (int x:4:-16 z:4:-20)
      (+ (+ x:4 y:3) z:4)
    )
  )
  (
    (int w:3:-8)
    (+ (+ x:2 y:1) w:3)
  )
  (+ x:2 y:1)
))
```

```
((int g:0) ((int y:1:8))
(
  (int z:2:-4)
  (FCALL f:0 x:0 y:1)
  (FCALL g:0 z:2)
))
```

[注意] 課題 6 と同様，インデントや改行などの工夫は必須ではない．

6 NASM

[教科書 6 章も参照すること]

本実験で作成するコンパイラは Pentium プロセッサを搭載した計算機をターゲットマシンとする．生成するコードは NASM (Netwide Assembler)[3] というアセンブラのアセンブリコードである．

NASM のアセンブリコードの例を以下に示す．これは教科書 [1] p. 160 の関数 `foo` のコードを NASM 用に書き直したものである．

```
        GLOBAL  foo
foo     push    ebp
        mov     ebp, esp
        sub     esp, 4
        mov     eax, [ebp+8]
        imul    eax, [ebp+8]
        mov     [ebp-4], eax
        mov     eax, [ebp-4]
        add     eax, 2
        mov     esp, ebp
        pop     ebp
        ret
```

教科書で用いるアセンブラとの主な相違と NASM の疑似命令，その他注意点について挙げておく．

- `GLOBAL` はアセンブラの疑似命令であり，ラベルを大域的なラベルとして使用することを指示する．大域的なラベルは他のモジュール（オブジェクトファイルやライブラリ）から参照することができる．`GLOBAL` 指示されるラベルは，先に `GLOBAL` 指示をしてからラベルの宣言をしなければならない．
  一方，他のモジュールで宣言されるラベルを参照するには，疑似命令 `EXTERN` を用いてあらかじめ外部宣言しておく必要がある．Tiny C のコンパイラでは，未定義関数の呼出コードを生成する場合に `EXTERN` が必要となる．例えば，ライブラリ関数 `abs` を呼び出す場合は次のように `abs` を外部宣言しておく．
  ```
  EXTERN  abs
  ...
  call    abs
  ...
  ```
  なお，同じラベルに対して 2 回以上 `EXTERN` 指示した場合は，2 回目以降の指示は無視される．
- レジスタ R からの相対番地 n を使ったメモリ番地指定は `[`R`+`n`]` または `[`R`-`n`]` あるいは `[`n`+`R`]` 等の形で記述する．
- 大域変数/関数 `x` のラベルは `x` で表す（アンダースコア (_) は不要）．
- 大域データ領域へメモリを割当てることを指示するには，疑似命令 `COMMON` を用いて次のように行なう．

COMMON *label* n

これは n バイトのメモリを大域データ領域に割当て，そのアドレスを *label* とする指示である．

同じ名前のラベルを COMMON 指示している複数のモジュールをリンクした場合，割当てられるメモリ領域のサイズは n であり，全てのモジュールは同じアドレスを参照する．

Tiny C のコンパイラでは，大域変数の値の格納場所を確保するために COMMON が必要となる．

- 大域データ領域の値の参照は，*label* ではなく［*label*］としなければいけない：

  ```
  imul ebx, [x]
  ```

- ある命令のオペランド列にメモリ番地が使用される場合で，かつレジスタがそのオペランド列中に含まれない場合は，メモリ番地に格納されるデータのサイズを指定しなければならない：

  ```
  push dword loc(v)
  mov dword loc(v), c
  cmp dword loc(v), c
  ```

  dword はメモリ番地に格納されるデータのサイズが 32 ビットであることを指示する．

- ラベル宣言のラベル名の後ろのコロン (:) はなくてもよい．ラベルは他の命令と同じ行である必要はなく，ラベルのみの行があってもよい．
- ; 以降はコメントとみなされる．（ソースプログラムとの対応関係を明確にするために利用するとデバグが楽になる．）

NASM の使用例を以下に示す．

```
% cat a.asm
··· 前述の foo のアセンブリコード ···
% cat main.c
#include <stdio.h>
main()
{
    printf("%d\n", foo(3));
}
% nasm -f elf a.asm
% gcc main.c a.o
% ./a.out
11
```

この例は，関数 foo() （のアセンブリコード）については NASM によって，関数 main() については gcc によってオブジェクトプログラムを生成し，両者をリンクして実行形式プログラム a.out を生成するものである．

[ミニ課題] int 型の引数 n を一つとり，n^2 を返すような関数 foo のコードを NASM で書き，上の main.c とリンクして動作を確認せよ．

7 コード生成

[教科書 6 章も参照すること]

本節では，構文木から NASM のコードを生成する部分の実装を行なう．

コード生成部の基本的な処理の流れは，おおまかに言えば 4 節で作成した構文木表示プログラムと同じである．つまり，構文木表示プログラム同様，構文木を入力とし，（構文木を表示する代りに）コードを表示（生成）するプログラムである．

コード生成の方法は，教科書に詳しく述べられているのでそれに従うこと．ただし，局所変数や一時変数等のレジスタ割当てはしなくてもよいものとする．つまり，汎用レジスタは eax レジスタのみを使用[*6]し，局所変数や一時変数等はすべてスタック上に割当てる．

7.1 コード生成の準備

教科書の一時変数の割り当て方式は，局所変数領域のサイズ *Nlocal* が関数本体のコードを生成した後でないと決定できない（教科書 p. 166）．そこで教科書でも述べられているように，関数全体のコードを一旦メモリに書き出しておき，*Nlocal* の実際の値を後で書き込む等の処理が必要である．これの具体的な手法は，様々であるが，例えば次のようにして行なえばよい．まず，アセンブリコード一命令分（一行分）を保持する構造体を定義する．

```
struct code {
    char *cd;
    struct code *next;
};
```

ここでは，1 命令を単純に 1 つの文字列で表現することにした．生成されるコード列は，この構造体のリストとして表現する．そして 1 命令のコード生成を行なう関数を用意する．

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
struct code *emit(char *inst, char *op1, char *op2)
{
    char buf[80];
    struct code *c = (struct code *)malloc(sizeof(struct code));

    if (inst == NULL)
        buf[0] = '\0';
    else if (op1 == NULL)
        sprintf(buf, "\t%s\n", inst);
    else if (op2 == NULL)
        sprintf(buf, "\t%s\t%s\n", inst, op1);
    else
        sprintf(buf, "\t%s\t%s, %s\n", inst, op1, op2);
    c->cd = strdup(buf);
    c->next = NULL;
    構造体 c をコード列リストの最後尾に追加;
    return c;
}
```

[注意] この emit() 関数ではラベルの挿入ができない．emit() の引数を一つ増やしてラベルも扱えるようにするか，ラベルを挿入するための関数を別途用意するなどしなければならい．

この関数は教科書 p. 186 の emit() に相当する．例えば，

```
emit("mov", "eax", "1")
```

とすると[*7]，文字列 "\tmov\teax, 1\n" が作成され，コード列リストに追加される．ただし，上記の emit() の定義はパラメータの数が固定なので，例えば ret 命令等の生成は

[*6] 例外として，除算のコード生成のところで edx レジスタを使用する．

[*7] ここで示す emit() の実装では，レジスタ eax や整数定数 1 は文字列として emit() に渡す必要がある．

```
emit("ret", NULL, NULL)
```

とする．C 言語の可変引数リストの機能を使えば，任意個の引数を取る `emit()` を実装できるがここでは触れない．

コード生成時には，コード列の先頭を指す `struct code *`型の大域変数を用意し，ここにコード列を追加していく．そして，コード生成終了後にこのコード列を表示すればよい．例えば，以下のようにすればよい．

```
main:
                { symtab = NULL; cur_lev = 0; }
        program
                { if (yynerrs == 0 && semnerrs == 0)
                     { emit_program($2);
                       print_code(); }
                }
        ;
```

また，当然のことながらラベルはコード全体を通して一意的でなければならない．例えば関数のリターン部のラベルを全て `Lret` などとすると，複数の関数定義があるとラベルが重複してしまう．一意的なラベルを生成する方法としては，整数型の大域変数でラベルのカウントを行ない，常に新しい数 n に対して `L`n を生成するような関数 `char *make_label()` などを作ればよい．

> [ミニ課題] このままではコードを見てもラベルの用途がさっぱりわからず，実行時の動作が解りにくい．理解しやすいコードにするために，ラベル名を工夫してみよ．

以下，コード生成部について解説するが，詳細は教科書に載っているので割愛する．

7.2 関数定義のコード生成

関数定義に対しては，以下の雛形に沿ったコードを生成すればよい．

```
        GLOBAL  f
f       push    ebp
        mov     ebp, esp
        sub     esp, Nlocal      ··· (*)
        関数本体のコード
        （ここで top_alloc を計算）
Lret    mov     esp, ebp
        pop     ebp
        ret
```

上で解説した設計によれば，*Nlocal* の書き込みは次のよう行なえる．まず，(*) の行 “`sub esp,` *Nlocal*” の代りに以下のように仮のコードを生成する．

```
struct code *c = emit(NULL, NULL, NULL);
```

次に，関数本体のコード生成終了後に `top_alloc` の値を調べてもし 0 でなければ（*Nlocal* が 0 でなければ）生成した仮のコードの内容を書換える．

```
...
関数本体のコード生成;
if (top_alloc) {
    char buf[80];
    sprintf(buf, "\tsub\tesp, %d\n", -top_alloc);
    c->cd = strdup(buf);
```

```
        emit("mov", "esp", "ebp"); /* 実際はラベルの出力が必要 */
    }
    ...
```

top_alloc の値が 0 の場合は，コード列中に仮のコードが残るが，コード列を出力するときにそのようなコードを無視すれば問題ない．

7.3 文のコード生成

文のコード生成は教科書に詳細が解説されているのでそれを参照すること．
以下は，while 文のコード生成部の一例である．

```
...
case WHILE:
    先頭ラベルと末尾ラベルの作成;
    コード列に先頭ラベルを挿入;
    条件式のコード生成;
    emit("cmp", "eax", "0");
    emit("je", 末尾ラベル, NULL);
    本体のコード生成;
    emit("jmp", 先頭ラベル, NULL);
    末尾ラベルを挿入;
    break;
...
```

7.4 式のコード生成

式に対しては，式を評価し，結果を eax に格納する（代入式については代入操作も行なう）コードを生成するのが基本である．これも，教科書に詳細があるのでそれを参照すること．以下，補足の注意事項を挙げる．

7.4.1 算術演算式のコード生成

二項算術演算式について，最も素朴な方法は全て RSL 型（教科書 6.5.2 節参照）でコードを生成することである．自信がない人はまずは全て RSL 型のコードを作成してみよう．算術演算式の RSL 型のコードは例えば以下のようになる．

```
...
一時変数を用意;
右オペランドのコード生成;
emit("mov", 一時変数, "eax");
左オペランドのコード生成;
emit(命令, "eax", 一時変数);
一時変数の解放;
...
```

レジスタ使用量の計算をする場合も，本実験では汎用レジスタとして eax 一つだけの使用を想定しているので，それに従うならば「レジスタを使用するか否か」だけをチェックすればよい．レジスタを使用しない場合，というのはすなわち変数または定数の場合である．

[ミニ課題] 余力があればレジスタを複数使用したさらなる最適化についても考えてみよ．

■一時変数について　例えば RSL 型のコードを生成するときに右オペランドの値を一時入れておく一時変数は，局所変数と同じ領域に格納され，*Nlocal* の値は一時変数が割当てられる領域も考慮しなければならない．このため，一時変数の格納場所は，局所変数と同様に `allocate_loc` を用いて決定するのがよい．また，一時変数については，それが不要になった段階でメモリ領域を解放しておく（つまり，`release_loc` しておく）べきである．

■除算　除算には，除算命令 `idiv` を使用する．`idiv` は `edx` レジスタと `eax` レジスタで表される 64 ビットの値（上位：`edx`，下位：`eax`）をオペランドのレジスタまたはメモリ番地の値で割った値を `eax` に格納する命令である（割った余りは `edx` に格納される）．e_1 `/` e_2 のコードは次のようにすればよい（除算のコード生成は，例外として `eax` の他に `edx` も用いる）．

■RSL 型コード：

e_2 **の計算（**`eax` **に結果）**
`mov` *temp*`,` `eax`
e_1 **の計算（**`eax` **に結果）**
`cdq`
`idiv dword` *temp*

■L 型コード（e_2 が変数 v の場合）：

e_1 **の計算（**`eax` **に結果）**
`cdq`
`idiv dword` loc(v)

■L 型コード（e_2 が定数 c の場合）：

e_1 **の計算（**`eax` **に結果）**
`cdq`
`mov dword` *temp*`,` c
`idiv dword` *temp*

`cdq` 命令は，`eax` を符合拡張して上位 32 ビットを `edx`，下位 32 ビットを `eax` に格納する命令である．

7.4.2　比較演算

比較演算（==, !=, >, etc.）式の「値」を求めるコードは，`cmp` 命令の後に条件付きジャンプ命令の代りに次のコードを生成すればよい．

`set`c `al`
`movzx eax, al ;` **`al` の値を** `32` **ビット拡張し，**`eax` **に格納する．**

`set`c 命令は，c（`g`, `ge`, `e` 等）で指定した条件と一致する場合に 1 を，一致しない場合に 0 を `al` レジスタに格納する．`al` レジスタは `eax` レジスタの下位 8 ビットにマップされた 8 ビット長のレジスタである．例えば，e_1 `>=` e_2 が RSL 型であれば，

e_2 **の計算（**`eax` **に結果）**
`mov` *temp*`,` `eax`
e_1 **の計算（**`eax` **に結果）**
`cmp eax,` *temp*
`setge al`
`movzx eax, al`

となる．

7.4.3 論理演算式

論理演算（&&, ||）式の「値」を求めるコードは，例えば e_1 && e_2 なら次のようにすればよい．

```
        mov dword temp, 0
        e1 が偽なら L へ
        e2 が偽なら L へ
        mov dword temp, 1
L:
        mov eax, temp
```

7.4.4 関数呼出

関数呼出のコードは，各実引数を push してから，関数を call すればよい．意味解析時のパラメータを格納する相対番地の決め方からも解るとおり，関数呼出 f(e1, e2, ..., en) の引数は en からはじめて右から左の順に push しなければならない．

ソースプログラム中で未定義の関数（つまり kind が UNDEFFUN の関数）を呼び出すときには，関数名のラベルを EXTERN しなければならない．

> [注意] 前節でも述べたとおり，EXTERN の指示は同一ラベルに何度行なってもよい（二度目以降は無視される）．コードのコンパクトさを考えないのであれば，UNDEFFUN の関数を call するたびに EXTERN を指示してもよい．

■課題 8 Tiny C コンパイラのコード生成部を教科書に従い作成せよ．本課題が要求する必須項目は「生成されるコードが正しく動作する」ということだけである．

その他，以下の例に挙げるような拡張や最適化があれば加点対象とする（拡張・最適化した点とその方法はレポートで必ず解説すること）．

- 言語仕様の拡張
  - for, continue, break の追加.
    > [注意] continue, break については，ループ内で使われていることを意味解析でチェックすべきである．
  - ビット演算子，インクリメント/デクリメント演算子（前置/後置）の追加
    > [注意] ビット演算子&，|，^については，それぞれアセンブリ命令 and, or，xor を使えばよい．
  - char 型，ポインタ型の導入
- 無駄なラベルとジャンプ命令の抑制
- 変数のレジスタ割当て
- などなど…（教科書の最適化の項を参照のこと）

> [注意] 生成コードのもう一つの評価基準としてコードの実行速度が挙げられるが，実行速度の速いコードを生成するには（昔の CISC プロセッサと比べるとより多くの）プロセッサのアーキテクチャについての知識が必要である．そのため本実験では，コードのコンパクトさを評価基準とするが，もし実行速度の速いコードについて興味があれば文献 [4] を参考にすること．

8　マニュアル

ライブラリ関数やシステムコール，flex, bison, nasm 等のコマンドの使い方については UNIX コマンドの `man` を利用すること．

Flex と Bison，NASM の詳細なマニュアルは emacs の info を利用するか，URL[2] を参照せよ．info の利用方法については，emacs において M-x info を実行すれば参照することができる．NASM については URL[3] を参照することもできる．

参考文献

[1] 湯淺太一著：コンパイラ，昭晃堂.

[2] 計算機科学実験及演習 3（ソフトウェア）

`http://ecs.kuis.kyoto-u.ac.jp/isle/le3b/index.html`

[3] NASM

`http://nasm.sourceforge.net/doc/nasmdoc0.html`

[4] インテル (R) 64 アーキテクチャーおよび IA-32 アーキテクチャー最適化リファレンス・マニュアル

`http://download.intel.com/jp/developer/jpdoc/248966-017JA.pdf`