J5実験 第3回

第2回で,

- 電卓
- 単純なプログラミング言語のインタプリタ（解釈実行器）

を作り，自分でプログラミング言語に構文を追加し，Lex・Yacc・Cを使ってプログラミング言語処理系を作成する方法が分かってきたと思います． 今回はその知識を利用して,

- 簡単な英文処理プログラム
- 単純なプログラミング言語のコンパイラ

を作成してみましょう.

忘れてしまったところは，過去の授業資料 1.ipynb と 2.ipynb を参照してください.

簡単な英文処理プログラム

Lex や Yacc はもともと「コンパイラを簡単に作るためのツール」として作られたものです． とはいえ，字句のパターンと構文規則を与えれば，テキストを解釈できるプログラムを作ることができるので，色々な応用が考えられます.

そういった応用例の1つとして，かなりいいかげんなものですが，簡単な英文の解析プログラムをここでは作ってみましょう.

与えられた英文テキストに対して，英文法の構成要素を解析し，テキストに情報を付加します．こういった品詞などの解析は， **形態素解析** と呼ばれています． 自然言語処理【プログラミング言語」と区別するために，情報の分野ではそう呼ばれる）では必要な処理ですので，ここまでのプログラミング言語を作るというところに興味がない・他人事だと思っていた人も，**テキストを読み込むプログラムを作る**ということの必要性が分かるのではないでしょうか.

（ここでやるのは，かなりいいかげんな解析ですが）雰囲気を見てみましょう.

まず，Lexを以下のように定義します.

In [ ]:
```
/* lex ex3-1.lex */
alpha   [a-zA-Z]
white   [\t ]
%%
CATS|DOGS|MAN|WOMAN         { strcpy(yylval.name, yytext); return NOUN; }

ROOM|HOUSE|SKY              { strcpy(yylval.name, yytext); return PLACE_NOUN; }

THE|A|AN|THIS|THAT|THOSE|MY { strcpy(yylval.name, yytext); return DETERMINER; }

FIRST|LAST|NEXT|SMALL|BIG   { strcpy(yylval.name, yytext); return ADJECTIVE; }

IS|ARE|RUN|FLY|FIND|BE      { strcpy(yylval.name, yytext); return VERB; }
```

```
CAN|SHOULD|MAY              { strcpy(yylval.name, yytext); return AUXILIARY_VERB

IN|AT|ON|TO|INSIDE          { strcpy(yylval.name, yytext); return PREPOSITIONAL;

[\.,']                      { return yytext[0]; }
{white}                     { ; }
```

今は，考えやすくするため，使える英単語をここに書かれたものに限定します． つまり，

> THE MAN MAY FIND THOSE DOGS AT THAT HOUSE.

のような構文を解析できる解析器ということですね． アクションはこれまで出てきた Lex のプログラムと同等ですが，**Lex で `yylval` に入れておいた値は，Yaccの属性値として利用することができる** という使い方は新たに出てきたものです． こうすることで，Yaccのアクションで直接 `$1` などの値を読み出せるので，定義が簡潔になります．

それでは，Yaccのプログラムを見てみましょう．

In [ ]:
```
/* yacc ex3-1.yacc */
%union {
  char   name[128];
}
%token <name> PREPOSITIONAL VERB AUXILIARY_VERB PLACE_NOUN NOUN DETERMINER ADJEC

%type  <name> np det vp pp obj
%%
ss : np vp '.'  { printf("NP = %s\nVP = %s\n", $1, $2); }
   ;

np : DETERMINER NOUN         { sprintf($$,"%s %s", $1, $2); }
   | ADJECTIVE det NOUN      { sprintf($$, "%s %s %s", $1, $2, $3); }
   | NOUN                    { strcpy($$, $1); }
   ;

det : DETERMINER   { strcpy($$, $1); }
    |              { strcpy($$, ""); }
    ;

vp : VERB             { strcpy($$, $1); strcat($$," "); }
   | AUXILIARY_VERB VERB obj { sprintf($$, "%s %s %s", $1, $2, $3); }
   | VERB det NOUN pp        { sprintf($$, "%s %s %s %s", $1, $2, $3, $4); }
   ;

pp : PREPOSITIONAL det PLACE_NOUN    { sprintf($$, "%s %s %s", $1, $2, $3); }
   |                                 { strcpy($$, ""); }
   ;

obj : det NOUN pp   { sprintf($$, "%s %s %s", $1, $2, $3); }
    | PREPOSITIONAL det PLACE_NOUN    { sprintf($$, "%s %s %s", $1, $2, $3); }
    |               { sprintf($$, ""); }
    ;
```

文は `NP VP` （名詞句と動詞句）です．ここに構文規則を追加していけば，解析できるパターンが増えていくのは，ここまでのYaccの知識で理解できると思います．Lexで `yylval.name` に値を入れておいたことにより，`det : DETERMINER   { strcpy($$, $1); }` のように `token` の値を属性値 `$1` として使うことができます．

Cプログラムは，これまでのプログラムのままです．

In [ ]:
```
/* c ex3-1.c */
#include <stdio.h>
extern char *yytext;

int value;

#include "y.tab.c"
#include "lex.yy.c"

int main() {
  yyparse();
  return 0;
}
```

それでは，上で例に挙げた文を解析してみましょう．

In [ ]:
```
/* a.out */
THE MAN MAY FIND THOSE DOGS AT THAT HOUSE.
```

同様に，別の文でも解析できます．

In [ ]:
```
/* a.out */
CATS CAN FLY TO THE SKY.
```

ここでは，「文」に対する解析結果だけを出力していますが，たとえば `VP` や `PP` がより細かくどういう品詞に分解されるかなども構文定義では分けてありますので，アクションを記述すれば可能だと分かるはずです．さらに，第2回でプログラミング言語に対して作った抽象構文木のような中間構造を用いれば，解析結果を色々な用途で利用できます． 実際には，**世の中にある英語の文にマッチするような構文を用意できるわけではなく**（言語学の分野では統語論などとして知られるが，たぶん一般的な構文ルールを用意するのは無理なんじゃないかなぁ...（個人の感想です）），実用されている形態素解析はこれほど簡単に解析できるものではないです．そのため，構文以外の様々な手法を用いて解析する必要があます（興味がある人は調べてみてください）．ここでは，プログラミング言語の生成以外のLexやYaccの使用例を紹介しました．

以上のように，英文に限らず，コマンドの設定ファイルや，データ形式（たとえば，JSON形式）を読み込みたいというときにも，テキストを読み込んで意味を解釈するプログラムの必要性が理解できたと思います．

単純なプログラミング言語のコンパイラ

第2回までで，中間形式（抽象構文木）の作成と，その解釈実行のプログラム（インタプリタ）を作成できました．

ここではコンパイラを作ってみましょう． コンパイラは，**ソースコードを別のプログラミング言語のプログラムに翻訳する**プログラムです．翻訳先の言語は，

- より低級な言語であったり，
- 実行しやすい（便利な）言語であったり

して，状況に応じて翻訳先を考えます．

すでに，ソースコードの木構造を作るところまでは前回までできていますので，Cプログラム中の `dotree()` で実行する際にコード生成をすればよいわけです．

コード生成 - C言語へのコンパイル

まずは，C言語へのコンパイルを考えてみましょう． C言語のコンパイラは様々実装されていますし，色々な環境で動かせるコードを吐き出せるので，自分が作った言語のコンパイル先としてC言語を考えることはありえます（上の翻訳先の言語の箇条書きでいう「実行しやすい便利な言語」ということです）．また，みなさんもC言語は読めるはずですし，第2回までで作ってきた言語もC言語から大きく離れた構文ではないため，コンパイル先として難しくありません．

ここでは， `dotree()` のかわりに， `emit_c()` を定義して，コンパイルの例を考えてみましょう．

簡単のため，前回の `while` 文が入る前の Lex と Yacc をそのまま使います．

In [ ]:
```
/* lex ex3-2.lex */
alpha   [a-zA-Z]
digit   [0-9]
white   [\n\t ]
%%
read                        { return READ; }
print                       { return PRINT; }
{alpha}({alpha}|{digit})*   { return IDENT; }
{digit}+                    { return NUM; }
[+\-=();{}]                 { return yytext[0]; }
{white}                     { ; }
```

In [ ]:
```
/* yacc ex3-2.yacc */
%union {
  Node* np;
  int i;
}
%type <np> stlist stat expr prim var
%token NUM;
%token IDENT;
%token READ;
%token PRINT;
%%
prog   : IDENT '{' stlist '}'       { emit_c($3); return 0; }
       ;
stlist :                            { $$ = NULL; }
       | stlist stat                { $$ = createNode(T_STLIST, $1, $2); }
       ;
stat   : var '=' expr ';'           { $$ = createNode(T_ASSIGN, $1, $3); }
       | READ var ';'               { $$ = createNode(T_READ, $2, NULL); }
       | PRINT expr ';'             { $$ = createNode(T_PRINT, $2, NULL); }
       ;
expr   : prim              { $$ = $1; }
       | expr '+' prim     { $$ = createNode(T_ADD, $1, $3); }
       | expr '-' prim     { $$ = createNode(T_SUB, $1, $3); }
       ;
prim   : NUM               { $$ = createNode(T_NUM, atoi(yytext), NULL); }
       | var               { $$ = createNode(T_VAR, $1, NULL); }
```

```
       | '(' expr ')'      { $$ = $2; }
       ;
var    : IDENT             { $$ = lookup(yytext); }
       ;
```

Cへコンパイルするには，今作成している言語との差分を考慮しなくてはいけないので，以下のCプログラムでのポイントは，

- Cの `main` 関数をちゃんと書く
- 変数宣言をする
- Cの関数を呼び出すように変換する

という点です．特に今の言語では，`print` は `printf` へ，`read` は `scanf` へ変換できる1対1の対応があるので，コンパイルしやすいですね．

木をたどるプログラムはそのままで，ノードごとの想定している動作にしたがって，Cのプログラムを出力すればよいので，たとえば，`T_ASSIGN` のノードであれば，

```
    case T_ASSIGN:
          printf("\tv%d = ", node->left);
          _emit_c(node->right);
          printf(";\n");
          break;
```

のようにして，`変数 = 式;` というCのプログラムにすればよいですね．（`_emit_c` を再帰的に呼べば，式を出力できるように作ってあるため．）変数のところは，変数表での番号 `n` に置き換えて `vn` と出力します．（`T_ASSIGN` のノードで，左の子にそのまま変数番号が入っていたことを思い出してください）

他の木のノードに対しても記述を追加したCプログラムは以下のとおりです．理解した上で実行してください．

In [ ]:
```
/* c ex3-2.c */
#include <stdio.h>
struct stab {
  int val;
  char name[20];
} stab[100];
int stabuse = 0;

extern char *yytext;

#define T_STLIST 1
#define T_ASSIGN 2
#define T_READ   3
#define T_PRINT  4
#define T_ADD    5
#define T_SUB    6
#define T_NUM    7
#define T_VAR    8

typedef struct Node {
    int node_type;
```

```
    struct Node* left;
    struct Node* right;
} Node;


// 引数にとる文字列が表にあれば見つかった位置を返して，なければ追加する関数
int lookup(char *s) {
  int i;
  // 表の使っているところを順に見ていって
  for (i = 0; i < stabuse; ++i) {
    if (strcmp(stab[i].name, s) == 0) {
      // 見つかった場合はその位置を返す
      return i;
    }
  }
  if (stabuse >= 99) {
    // 変数の上限の個数を超えてしまった
    printf("table overflow,\n");
    exit(1);
  }

  // 表になかったので追加する
  strcpy(stab[stabuse].name, s);
  return stabuse++;  // 追加した位置を返す
}

// 新しいノードを作る関数
Node* createNode(int node_type, Node* left, Node* right) {
    Node* newNode = (Node*)malloc(sizeof(Node));
    newNode->node_type = node_type;
    newNode->left = left;
    newNode->right = right;
    return newNode;
}

void _emit_c(Node* node) {
  if (node == NULL) {
    return;
  }
  switch (node->node_type) {
    case T_STLIST:
      if (node->left != NULL) { _emit_c(node->left); }
      _emit_c(node->right);
      break;
    case T_ASSIGN:
      printf("\tv%d = ", node->left);
      _emit_c(node->right);
      printf(";\n");
      break;
    case T_READ:
      printf("\tscanf(\"%%d\", &v%d);\n", node->left);
      break;
    case T_PRINT:
      printf("\tprintf(\"%%d\",");
      _emit_c(node->left);
      printf(");\n");
      break;
    case T_ADD:
      _emit_c(node->left);
      printf(" + ");
```

```
      _emit_c(node->right);
      break;
    case T_SUB:
      _emit_c(node->left);
      printf(" - ");
      _emit_c(node->right);
      break;
    case T_NUM:
      printf("%d", node->left);
      break;
    case T_VAR:
      printf("v%d", node->left);
      break;
  }
}

void emit_c(Node* node) {
  printf("#include <stdio.h>\n");
  printf("int main() {\n");
  int i;
  for (i = 0; i < stabuse; i++) {
    printf("\tint v%d;\n", i);
  }
  _emit_c(node);
  printf("}\n");
}

#include "y.tab.c"
#include "lex.yy.c"

int main() {
  yyparse();
  return 0;
}
```

それでは，自分達の言語からC言語へコンパイルしてみましょう．

In [ ]:
```
/* a.out */
main {
  read x;
  y = x + 1;
  print y;
}
```

a.outの出力結果は，ちゃんとC言語のプログラムになっているはずです．上のポイントとして挙げた箇条書きに対応した形で書くと，

- `emit_c()` の先頭で `main` 関数などC言語に必要なものを出力する
- 変数表をたどって，変数宣言をする（`v?`）というCの変数名とする
- 必要なところはCの構文に合わせて，関数などを呼ぶようにする

という点ができていることを確認してください．

さらに，このプログラムをCコンパイラで機械語に翻訳すれば，実行できますので，各自でコピーして実行してみてください．

以上のように，中間構造ができているものに対して，コード生成することに難しいことはありません．

コード生成 - アセンブリ言語へのコンパイル

次に，アセンブリ言語へのコンパイルを考えてみましょう． ここでは，**x64アセンブリ言語**への出力を考えます． アセンブリ言語全般については，もしかしたら，計算機通論の**MIPSアセンブリ言語**を習っているかもしれないので，それを思い出してください．

C言語よりも低級な言語であるので，実行形式を考えつつ，コンパイル（要するにコードの出力）をしないといけません． x64アセンブリ言語は**レジスタマシン**なので，先のC言語のように1対1で出力するというわけにはいきません．

この実験では x64 のアセンブリ言語の理解が主眼ではないので，実験に必要なところだけ，以下にまとめました． 以下の「x64アセンブリ言語の説明」を広げて読んでください．

さらなる詳細を知りたい場合には，自分で調べてみてください．

▶ x64アセンブリ言語の説明

それでは，C言語にコンパイルしたときと同様の簡単な言語の例で見てみましょう．

In [ ]:
```
/* lex ex3-3.lex */
alpha   [a-zA-Z]
digit   [0-9]
white   [\n\t ]
%%
read                        { return READ; }
print                       { return PRINT; }
{alpha}({alpha}|{digit})*   { return IDENT; }
{digit}+                    { return NUM; }
[+\-=();{}]                 { return yytext[0]; }
{white}                     { ; }
```

In [ ]:
```
/* yacc ex3-3.yacc */
%union {
  Node* np;
  int i;
}
%type <np> stlist stat expr prim var
%token NUM;
%token IDENT;
%token READ;
%token PRINT;
%%
prog   : IDENT '{' stlist '}'       { emit_asm($3); return 0; }
       ;
stlist :                            { $$ = NULL; }
       | stlist stat                { $$ = createNode(T_STLIST, $1, $2); }
       ;
stat   : var '=' expr ';'           { $$ = createNode(T_ASSIGN, $1, $3); }
       | READ var ';'               { $$ = createNode(T_READ, $2, NULL); }
       | PRINT expr ';'             { $$ = createNode(T_PRINT, $2, NULL); }
       ;
expr   : prim              { $$ = $1; }
       | expr '+' prim     { $$ = createNode(T_ADD, $1, $3); }
       | expr '-' prim     { $$ = createNode(T_SUB, $1, $3); }
       ;
```

```
prim   : NUM               { $$ = createNode(T_NUM, atoi(yytext), NULL); }
       | var               { $$ = createNode(T_VAR, $1, NULL); }
       | '(' expr ')'      { $$ = $2; }
       ;
var    : IDENT             { $$ = lookup(yytext); }
       ;
```

Yaccで，`emit_c`だったところが`emit_asm`になっているだけです．

Cプログラムは以下のようになります．

In [ ]:
```
/* c ex3-3.c */
#include <stdio.h>
struct stab {
  int val;
  char name[20];
} stab[100];
int stabuse = 0;

extern char *yytext;

#define T_STLIST 1
#define T_ASSIGN 2
#define T_READ   3
#define T_PRINT  4
#define T_ADD    5
#define T_SUB    6
#define T_NUM    7
#define T_VAR    8

typedef struct Node {
    int node_type;
    struct Node* left;
    struct Node* right;
} Node;


// 引数にとる文字列が表にあれば見つかった位置を返して，なければ追加する関数
int lookup(char *s) {
  int i;
  // 表の使っているところを順に見ていって
  for (i = 0; i < stabuse; ++i) {
    if (strcmp(stab[i].name, s) == 0) {
      // 見つかった場合はその位置を返す
      return i;
    }
  }
  if (stabuse >= 99) {
    // 変数の上限の個数を超えてしまった
    printf("table overflow,\n");
    exit(1);
  }

  // 表になかったので追加する
  strcpy(stab[stabuse].name, s);
  return stabuse++;  // 追加した位置を返す
}

// 新しいノードを作る関数
```

```
Node* createNode(int node_type, Node* left, Node* right) {
    Node* newNode = (Node*)malloc(sizeof(Node));
    newNode->node_type = node_type;
    newNode->left = left;
    newNode->right = right;
    return newNode;
}

void _emit_asm(Node* node) {
  if (node == NULL) {
    return;
  }
  switch (node->node_type) {
    case T_STLIST:
      if (node->left != NULL) { _emit_asm(node->left); }
      _emit_asm(node->right);
      break;
    case T_ASSIGN:
      _emit_asm(node->right);
      printf("  popq %d(%%rbp)\n", -((int)node->left + 1) * 8);
      break;
    case T_READ:
      printf("  movq $.Lprompt, %%rdi\n");
      printf("  movq $0, %%rax\n");
      printf("  call printf\n");
      printf("  leaq %d(%%rbp), %%rsi\n", -((int)node->left + 1) * 8);
      printf("  movq $.Lread, %%rdi\n");
      printf("  movq $0, %%rax\n");
      printf("  call scanf\n");
      break;
    case T_PRINT:
      _emit_asm(node->left);
      printf("  popq %%rsi\n");
      printf("  movq $.Lprint, %%rdi\n");
      printf("  movq $0, %%rax\n");
      printf("  call printf\n");
      break;
    case T_ADD:
      _emit_asm(node->left);
      _emit_asm(node->right);
      printf("  popq %%rdx\n");
      printf("  popq %%rax\n");
      printf("  addq %%rdx, %%rax\n");
      printf("  pushq %%rax\n");
      break;
    case T_SUB:
      _emit_asm(node->left);
      _emit_asm(node->right);
      printf("  popq %%rdx\n");
      printf("  popq %%rax\n");
      printf("  subq %%rdx, %%rax\n");
      printf("  pushq %%rax\n");
      break;
    case T_NUM:
      printf("  pushq $%d\n", node->left);
      break;
    case T_VAR:
      printf("  pushq %d(%%rbp)\n", -((int)node->left + 1) * 8);
      break;
  }
```

```
}

void emit_asm(Node* node) {
  int stk;
  printf("     .section .rodata\n");
  printf(".Lprompt: .string \"> \"\n");
  printf(".Lread: .string \"%%ld\"\n");
  printf(".Lprint: .string \"%%ld\\n\"\n");
  printf("     .text\n");
  printf(".globl main\n");
  printf("main:\n");
  printf("     pushq %%rbp\n");
  printf("     movq %%rsp, %%rbp\n");
  stk = (8 * stabuse + 15) / 16;
  stk *= 16;
  printf("     subq $%d, %%rsp\n", stk);
  _emit_asm(node);
  printf("     leave\n");
  printf("     ret\n\n");
}

#include "y.tab.c"
#include "lex.yy.c"

int main() {
  yyparse();
  return 0;
}
```

`emit_asm()` では，アセンブリ言語に必要なおきまりのコードを出力し，プログラム本体のコードを生成する `_emit_asm()` を呼び出します． 今の言語には， `read` 文と `print` 文で入出力を扱いますが，これはCの標準ライブラリにある `scanf()` と `printf()` を使います．これらの関数は，引数として浮動小数点数を受けとる可能性があるため，これらの関数を呼び出すときには `rax` に `0` を設定しておく必要があります．

ラベル `.Lprompt` は， `printf()` で， `.Lread` は `scanf()` で， `.Lprint` は `printf()` でそれぞれ使うための制御文字列「 `"> "` 」，「 `"%ld"` 』 `"%ld\n"` 」を入れた場所（アドレス）を参照するためのラベルです（符号付き64ビット長の値を読み書きするので， `%ld` ）． ディレクティブ「 `.string 文字列` 」は，メモリを「文字列」で初期化するためです（Cと同様「文字列」はnull終端される）．

`_emit_asm()` の処理も出力する言語がCからアセンブリ言語になっているだけで， `_emit_c()` と同様です． 出力するときに，実行時の途中結果を渡すかは，勝手に決めればよいので，**式の計算結果はスタックトップに置いてあるもの**としています．

- T_ASSIGN : 右辺を計算するコードを生成し，（結果はスタックトップにあるので）スタックから変数の場所にポップします．
- T_STLIST : 左側に文の並びがあれば，まずそのコードを出力し，続いて右側にある文のコードを出力します．
- T_READ : まずは，プロンプト「 `> ` 」を出力します．それには，書き出すべき文字列のアドレスを（ `printf()` の第1引数である） `%rdi` に入れて `printf()` を呼び出します．次に，変数のアドレス「 `%rbp - (変数番号 + 1) * 8` 」を（ `scanf()` の第2引数である）レジス

タ `%rsi` に，読み取り書式の文字列「 `"%ld"` 」のアドレス（第1引数である）レジスタ `%rdi` に入れて `scanf()` を呼び出します．変数のアドレスは， `%rbp - 8` ， `%rbp - 16` であることに注意．C言語で

```
printf("> ");
scanf("%ld", &変数);
```

と書いたコードに相当します．

- T_PRINT : まず， `_emit_asm()` で式のコードを出力します．（それを実行したらスタックトップに式の結果が置かれているはずなので）スタックトップを（ `printf()` の第2引数である）レジスタ `%rsi` にポップします．出力書式の文字列「 `"%ld\n"` 」のアドレスを（第1引数である）レジスタ `%rdi` に入れて， `printf()` を呼び出します．なお `printf()` には，（ここでは使っていないが）引数として浮動小数点数が渡る可能性があるため，レジスタ `%rax` にゼロを入れておく必要があります．C言語で

  ```
  printf("%ld\n", 式);
  ```

  と書いたコードに相当します．

- T_ADD，T_SUB : 左の項を計算するコードを生成し，ついで右の項を計算するコードを生成します．これでスタックは上から，右の項の結果，左の項の結果となっているので，右の項の値をレジスタ `%rdx` にポップし，ついで左の項の値を `%rax` にポップし，「 `%rax←%rax+%rdx` 」を計算して，その結果をプッシュします．

- T_NUM : 上で決めたことにより，値をスタックにプッシュします．

- T_VAR : 同様に，変数の値をスタックにプッシュします．

それでは，実行してみましょう．C言語へのコンパイラのときには，出力されたCのコードを手動でコンパイルしてみましょう，と説明しましたが，アセンブリ言語を実行するときにはどうすれば良いか説明していませんでした．

アセンブリ言語のコードから，実行形式を作るには， `cc` コマンドを使います．

```
% cc ファイル名.s
```

`cc` コマンドは，

1. サフィックスが `.c` のファイルは，Cソースだと思ってアセンブリ語に翻訳し，翻訳結果をアセンブルする．
2. サフィックスが `.s` のファイルは，直接アセンブルする．
3. いずれの場合も，アセンブル結果の `.o` ファイルを C の標準ライブラリとリンクして `a.out` ファイルを作る．

ということを行います．

今回のこのノートブックでは，直前に作成された `a.out` を自前のコンパイラだと思って（**a.outを作り忘れないように**），自分言語のプログラムを入力として受けとる `uecc` というノートブックのコマンドを作成してあります．ですので，今回は，上で `emit_asm()` を実装したCプログラムを正しく実行できていればコンパイラの `a.out` が生成できているはずで，以下のコードを実行すると， `a.out` が作成できます．

In [ ]:
```
/* uecc ex3-3.uec */
main {
  read x;
  y = x + 1;
  print y;
}
```

確認のためにアセンブリ言語のコードを出力しています．このコードが `.uec.s` ファイルに保存されて，`cc` コマンドで `a.out` が作成されました．

1つ数字を読み込んで，その数に1を足すだけのプログラムですが，以下のように実行できます．

In [ ]:
```
/* a.out */
8
```

以上のように，C言語へコンパイルしていたのと同様に，アセンブリ言語へもコンパイルできました． C言語のときと比べると，

- 実行形式を考える必要がある（スタックの使い方など）
- アセンブリ言語の決まりを意識する必要がある

という点はありますが，コンパイラにおけるコード生成について理解できたものと思います．

練習問題

提出は任意です．

順番にやる必要はなく，できるものから取り組んでください．

提出方法

以下の問題について，この3.ipynbファイルを編集し，完成した自分が編集した .ipynb ファイルをGoogle Classroomの該当課題に提出してください． （もしこのノートブックをファイルに保存したいときには，メニューの「file」→「Download」で .ipynb ファイルをダウンロード出来るので，それを提出してください．）

- 適宜コメントを入れてください．
- 提出する際のファイル名は，「 **`学籍番号.ipynb`** 」としてください．

問題1

C言語へのコンパイラで， `while` 文に対応してください． 前回の練習問題を解いた人は，複合文や `<=` や論理演算などに対応したものでもよいです．

ヒント: LexとYaccは前回のままでいけるはず（ `while` だけではなくて， `<` や `>` や `++` を追加するのも忘れないこと）

In [ ]:
```
/* lex p3-1.lex */
```

```
ここに書く
```

In [ ]:
```
/* yacc p3-1.yacc */
ここに書く
```

In [ ]:
```
/* c p3-1.c */
ここに書く
```

In [ ]:
```
/* a.out */
自分の書いた電卓のプログラムが正しく動作をしていることを確認できる入力を考えて書く
```

問題2

アセンブリ言語へのコンパイラで，`while`文に対応してください．いきなり実行すると難しいと思うので，簡単な例を使って，出てきたアセンブリ言語のコードをちゃんと読んで想定しているようになっているか見てみるとよいと思います．

- ヒント: ループの条件の飛び先を表わす**ラベル**を適宜付けていく つまり，

  ```
  ラベル1:
    条件に対応するコードを生成
    条件が不成立ならばラベル2へ分岐
        条件に合致する場合に実行する命令列（whileの本体）
        ラベル1へ
  ラベル2:
  ```
  というような命令列にすればよいですね．

- ヒント: ラベル番号を管理できるように

  `static int labelno = 1;`
  というような定義を`emit_asm`内に追加しましょう．

- ヒント:`cmp`命令は，`cmp src, dst`の形式で，`src - dst`の結果をコンディションコードレジスタに暗黙的に入れてくれます．「x64アセンブリ言語の説明」をよく読むこと．

- ヒント:`cmp`命令の直後で，`jge ラベル`のような命令でコンディションコードレジスタの結果からジャンプできます．

- ヒント（難しい箇所なので重要）:`while ( cond ) 式`のような構文なので，`cond`に対応するコードは`<`や`>`を表わす`T_LT`や`T_GT`の抽象構文木のノードになっているはずです．なので，`T_LT`や`T_GT`でコードを出力する箇所（`_emit_asm`の`switch`の分岐）では，`jge`や`jle`だけを出力しておいて（ラベルは表示しない），`T_WHILE`の箇所でラベルを出力するようにしましょう．（単独の式として出てくる`x = 1 < 5`のようなものは，今の言語にはないのでこれでよい）

- ヒント:`T_WHILE`の箇所でラベルを出力する際には，適宜`labelno++`して，ラベルがちゃんと被らないようにしましょう．

In [ ]:
```
/* lex p3-2.lex */
```

ここに書く

In [ ]:
```
/* yacc p3-.yacc */
ここに書く
```

In [ ]:
```
/* c p3-.c */
ここに書く
```

In [ ]:
```
/* a.out */
自分の書いた電卓のプログラムが正しく動作をしていることを確認できる入力を考えて書く
```