7 構文解析の演習

STAGE 1 宣言部の解析, 出入口コードの生成

この STAGE では, まず mini-C の変数と関数の宣言部の解析を行い, 宣言の内容を記号表に登録する. そして, 関数 main を起動するためのコードを生成する.

課題 1.1	関数 main (mcc.c のメインルーチン) の完成
課題 1.2	関数 preprocess (構文解析の前処理) の完成
課題 1.3	関数 parse_program (「プログラム」の解析) の完成
課題 1.4	関数 parse_declaration_head (「宣言頭部」の解析) の完成
課題 1.5	関数 parse_variable_declaration_tail (「変数宣言尾部」の解析) の完成
課題 1.6	関数 parse_variable_declaration (「変数宣言」の解析) の完成
課題 1.7	関数 parse_function_declaration_tail (「関数宣言尾部」の解析) の完成
課題 1.8	関数 parse_function_body (「関数本体」の解析) の完成
課題 1.9	関数 postprocess (構文解析の後処理) の完成

処理する文法規則

```
 1.  プログラム ::= ( 宣言頭部 ( 関数宣言尾部 | 変数宣言尾部 ";" ) )*
 2.  宣言頭部 ::= 型 "*"* ID
 3.  変数宣言尾部 ::= ( "[" INT "]" )*
 4.  関数宣言尾部 ::= "(" (ε | 変数宣言 ( "," 変数宣言 )* ) ")" 関数本体
 5.  関数本体 ::  = "{" (変数宣言 ";")*  文* "}"
 6.  変数宣言 ::= 宣言頭部 変数宣言尾部
 7.  型 ::= "int" | "char"
22.  関数名 ::= ID
23.  引数リスト ::= ε| 式  ( "," 式 )*
```

☆ STAGE 1 はコンパイラの枠組を作るので基本的に意味が分かりにくく, また他の関数の仕様の説明が十分でないこともあって, 始めは何をやっているのか分からないと思うことがあると思うが, 仕組みの理解よりも STAGE 2 以降のための土台を完成させることが主目的なので, あまり深く理解しようとせずに, 指示に従ってコーディングと確認を行ってとにかく完成させて欲しい.

課題 1.1 関数 main の完成

mcc のメインルーチンである main を完成させる.

- 具体的な作業

  ダウンロードした mcc.c の main は, 次のように, 変数 src, obj を宣言し, 引数解析ルーチン arg の呼び出しを行なっているだけである.

```
1:  int main(int argc, char **argv)
2:  {
3:      char source_f[FILENAME_MAX]; /* mini-C プログラムのファイル名 */
4:      char object_f[FILENAME_MAX]; /* VSM コードのファイル名 */
5:
```

```
 6:     /* (0) 変数 x, c, gt, lex_trace の宣言をこの下に書き込む */
 7:
 8:     /* (1) 引数の解析 */
 9:     arg(argc, argv, source_f, object_f, &mcc_trace);
10:
11:     /* (2)〜(7) の処理をこの下に書き込む */
12:
13:     return 0;
14: }
```

以下の指示に従って (0)〜(7) に必要な処理を書き込むことにより, main を完成させる. この後, mcc.c をコンパイルしてエラーが無いことを確認した後, mcc を実行して, 現時点での正しい結果が得られていることを確認する.

- main の基本構造

  main では以下の処理をこの順に行なう

```
(0) 変数の宣言
(1) 引数の解析
(2) 変数の割り当てと初期化
(3) 前処理ルーチンの呼び出し
(4) プログラム全体の構文解析ルーチンの呼び出し
(5) 後処理ルーチンの呼び出し
(6) VSM コードの生成
(7) 変数の解放と消去
```

- 処理 (プログラミング) の詳細

  (0) 変数の宣言

  変数 x, c, gt, lex_trace を宣言する.

```
lex_t *x;
code_t *c;
tab_t *gt;
lex_trace_t lex_trace;
```

  (1) 引数の解析

  引数を解析する関数 arg を呼び出す (書き込み済み). 関数 arg は, argc, argv に与えられたコマンドライン引数を解析し, source_f に mini-C のプログラムが入ったファイル名を, object_f にコンパイル結果の VSM コードを格納するファイル名を, mcc_trace (グローバル変数) にトレースモードの値を設定する. なお, arg 等の mcc.c の内部関数の仕様の詳細に関しては, [付録 7.2] を参照せよ.

  (2) 変数の割り当てと初期化

  字句解析ルーチン用データ x, VSM コードのデータ c global 変数表 gt , の割り当てと初期化を行う. また, 字句解析のトレースモードを設定する.

```
x = lex_new(source_f);
c = code_new();
gt = tab_new(lex_TOKEN_MAXLEN);
```

```
switch(mcc_trace) {
  case mcc_TRACE_NO: lex_trace = lex_TRACE_NO; break;
  case mcc_TRACE_LOW: lex_trace = lex_TRACE_BY_CHAR; break;
  case mcc_TRACE_MID: lex_trace = lex_TRACE_BY_TOKEN; break;
  case mcc_TRACE_HIGH: lex_trace = lex_TRACE_BY_TOKEN; break;
  default:  assert(0); /* エラー */
}
lex_trace_set(x, lex_trace);
```

ちなみに local 表 `lt` は, mini-C の各々の関数の解析中のみ存在するので, その解析ルーチンの中でその都度割り当て・消去を行う.

(3) 前処理ルーチンの呼び出し

入口コードの生成を行なう前処理ルーチン `preprocess` を呼び出す

```
preprocess(c);
```

(4) プログラム全体の構文解析ルーチンの呼び出し

基本的には「プログラム」の構文解析ルーチン `parse_program` を呼び出すだけだが, その前に, 最初の 1 トークンを読み込む. (構文解析が 1 記号先読みを仮定しているので, 常に次のトークンが読み込まれた状態で解析ルーチンを呼び出す.)

```
lex_get(x);
parse_program(c, x, gt);
```

(5) 後処理ルーチンの呼び出し

構文解析の後処理を行う関数 `postprocess` を呼び出す.

```
postprocess(c, gt);
```

(6) VSM コードの生成

作成されたコードをファイル `object_f` に書き出す. 各解析ルーチンで生成されたコードは, コードのデータ `c` に記憶されていて, `code_write` を呼び出すと, これがファイルに書き出される. (【注】これ以前に解析ルーチンがクラッシュすると, コードは 1 行も書き出されない.)

```
code_write(c, object_f);
```

(7) 変数の解放と消去

`gt`, `x`, `c` を消去する. 各データの解放・消去ルーチンを呼び出せば良い.

```
tab_delete(gt);
code_delete(c);
lex_delete(x);
```

- コンパイルと動作の確認

1. コンパイル

```
make mcc
```

または,

```
gcc -g -c mcc.c
gcc -g -o mcc mcc.o code.o tab.o lex.o
```

でエラーが出ないことを確認

2. テスト実行

ダウンロードした declare.mc をトレースモード 3 でコンパイルする.

```
./mcc -t 3 declare.mc
```

declare.mc の中身は

```
 1:  int a;
 2:  char *b;
 3:  int **c;
 4:  char d[15];
 5:
 6:  int e[4][25];
 7:  char *f[3][7][10];
 8:
 9:  int main()
10:  {
11:  }
12:
13:  int sub1(int p, char *q, int **r, int b)
14:  {
15:      char x; int *y; int z[10][30];
16:  }
```

のように変数と関数の宣言だけから成っている. これに対して,

```
1:  at preprocess
2:  (linenum(1) type(KW_INT) token("int"))
3:  at parse_program
4:  at postprocess
```

と表示されれば OK.

各関数に埋め込んである at 関数 (例えば at("preprocess"); 等) は, トレースモードが 2 以上のときに at preprocess のようにこの地点に到達したことを表示するものである.

従って, 1 行目の at preprocess は, 関数 preprocess が呼び出された (正確には, preprocess 中にある at("preprocess"); が呼び出された) ことを表している.

次の行の (linenum(1) type(KW_INT) token("int")) は, 最初の lex_get が行われ, 1 行目のトークン int を切り出したことを表している.

3〜4 行目は, 続いて parse_program と postprocess が呼び出されたことを表す.

課題 1.2 **関数 preprocess の完成**

構文解析の前処理 (入口コードの仮生成) を行う関数 preprocess を完成させる.

- 具体的な作業

ダウンロードした mcc.c の preprocess は

```
1:  static void preprocess(code_t *c)
2:  {
3:      at("preprocess");
4:  }
```

となっているので, at("preprocess") の下にプログラムを書き込んで行く. 変数の宣言の必要があれば, at("preprocess") の上に書く.

- 処理の詳細

  入口コードとは, VSM プログラムの先頭で, global 変数領域の確保や"main" 起動を行う, 以下のようなコードのことである.

```
0 ISP (global 変数の領域サイズ)
1 LC  (global 変数の領域サイズ)
2 SB 1
3 CALL (関数 main の先頭番地)
4 EXIT
```

  コードの生成は, code の関数 code_append を呼び出すことにより行う. code_append(c, 命令, 第 1 オペランド, 第 2 オペランド) は, コード c の末尾にその命令を追加し, その命令が先頭から数えて何番目 (先頭は 0) であるかを返す. ちなみに, 一旦生成した命令は, その命令が何番目かが分かれば, code_set により書き換えることができる. code パッケージの詳細は [付録 7.4] 参照のこと.

```
1:  code_append(c, opcode_ISP,  0, 0);
2:  code_append(c, opcode_LC,   0, 0);
3:  code_append(c, opcode_SB,   1, 0);
4:  code_append(c, opcode_CALL, 0, 0);
5:  code_append(c, opcode_EXIT, 0, 0);
```

  1 行目と 2 行目: ISP と LC のオペランドは, この時点では global 変数領域のサイズはわからない. そこで, ひとまず 0 とし, プログラム全体の構文解析が終ってから, 後処理 postprocess で正しい値を書き込むことにする.

  4 行目: 同様に, main の先頭番地も決まっていないので, 先頭番地をひとまず 0 としておき, postprocess で正しい値を書き込むことにする.

- コンパイルと動作の確認

  1. コンパイルして

```
./mcc -t 3 declare.mc
```

     を実行すると, 課題 1.1 と同じ結果が表示される.

  2. このとき, コンパイル結果の VSM コードが declare.vsm に書き出されていて, その内容が

```
0 ISP 0
1 LC 0
2 SB 1
3 CALL 0
4 EXIT
```

     となっていれば OK. 取り敢えずの入口コードが生成された.

static 関数

mcc.c の中には, static void preprocess(code_t *c) のように, 頭に static のついた関数が多くある. static あり/なしの意味は次の通り.

- static ありの関数

  有効範囲がそのファイルの中に限定される. 同じファイル中の関数からは呼び出すことができるが, (分割コンパイルしてリンクするときの) 他のファイル中の関数からは呼び出すことができない.

- static なしの関数

  有効範囲はそのファイルの中に限定されない. そのファイル中の関数からも他のファイル中の関数からも呼び出すことができる.

外部に対して公開しない関数は, static 宣言しておくべきである (外部インタフェースは最小限にすべきであるため).

課題 1.3 **関数 parse_program の完成**

mini-C の構文要素「プログラム」の構文解析とコード生成を行う関数 parse_program を完成させる.

- mcc.c に現れる parse_XXXXX という関数は, mini-C の構文要素 XXXXX を構文解析し, 解析した部分に対するコードを生成するものである.
- parse_XXXXX は次の引数をとる.
  - VSM のコードを保持する code_t 型構造体へのポインタ c. これを用いて, 新たなコードを追加する.
  - 字句解析用のデータを保持する lex_t 型構造体へのポインタ x. これを用いて字句解析の関数群を呼び出す.
  - 記号表のデータ tab_t 型構造体へのポインタ. 記号表には global 変数と関数の識別子を保持するグローバル記号表 (ポインタ gt で指される) と, local 変数や引数の識別子を保持するローカル記号表 (ポインタ lt で指される) がある. 構文解析をする場所により, グローバル表だけが必要な場合と両方の表へが必要な場合がある.
  - その他の情報. XXXXX 毎に異なる.

  例えば, parse_program は 3 つの引数 (code_t *c, lex_t* x, tab_t *gt) を取る.

- 具体的な作業

  課題 1.2 と同様, 関数 static void parse_program の at("parse_program") の前に必要な変数の宣言を, 後にプログラムを書き込んで parse_program を完成させる.

- parse_program の基本構造

  mini-C の「プログラム」の文法構造は

  ```
  プログラム ::= ( 宣言頭部 ( 関数宣言尾部 | 変数宣言尾部 ";" ) )*
  ```

  例えば, mini-C プログラム

```
1:  int a[10][20];
2:  char *c;
3:  int main() {*c='c'; sub(10,20);}
4:  int sub(int x, int y) {return(x+y);}
```

において,

宣言頭部	関数宣言尾部	変数宣言尾部
int a	−	[10][20]
char *c	−	(なし)
int main	() {c='c'; sub(10,20);}	−
int sub	(int x, int y) {return(x+y);}	−

である.

「宣言頭部」の後が「関数宣言尾部」か「変数宣言尾部」かの判定は, 「宣言頭部」を解析した後の先頭トークンが

"(" なら「関数宣言尾部」

それ以外なら「変数宣言尾部」

と考えれば良いので, この部分のプログラムは次のように書ける.

```
while (プログラムが終りでない) { …(1)
  「宣言頭部」の構文解析; …(2)
  if (次の記号が "(" ) { …(3)
    「関数宣言尾部」の構文解析; …(4)
  }
  else {
    「変数宣言尾部」の構文解析; …(5)
    if (次の記号が ';') { …(6)
      lex_get(x);
    }
    else {
      文法エラー; …(7)
    }
  }
}
```

- 処理の詳細

(1) while (プログラムが終りでない)

トークンを読み切ると, 次の記号のタイプ lex->type が token_EOF になるので, この部分は

```
while (x->type!=token_EOF) {
```

(2) 「宣言頭部」の構文解析

「宣言頭部」を解析する関数 parse_declaration_head を呼び出す.

```
parse_declaration_head(c, x, &type, &ptrlevel, id);
```

この関数は, 「宣言頭部」構文解析を行なうとともに, 宣言された変数/関数の基底型を type に, 宣言のポインタのレベル数 (* の個数) を ptrlevel に, 宣言された識別子を id に設定する.

type, ptrlevel, id の値は, 例えば次のようになる.

変数宣言 (頭部)	基底型 type	ポインタのレベル数 ptrlevel	識別子 id
int a	int	0	a
char *s	char	1	s
int **xxx	int	2	xxx

type, ptrlevel, id は, ローカル変数として宣言せよ. type は itab_basetype_t という名前の列挙型である.

```
itab_basetype_t type;
int ptrlevel;
char id[lex_TOKEN_MAXLEN+1];
```

(3) if (次の記号が "(")

"(" はタイプが token_LPAREN のトークンなので, この部分は

```
if (x->type==token_LPAREN) {
```

と書ける.

(4) 「関数宣言尾部」の構文解析

「関数宣言尾部」を解析する関数 parse_function_declaration_tail を呼び出す.

```
parse_function_declaration_tail (c, x, gt, type, ptrlevel, id);
```

(2) で得られた, type, ptrlevel, id を引数として渡している.

(5) 「変数宣言尾部」の構文解析

「変数宣言尾部」を解析する関数 parse_variable_declaration_tail を呼び出す.

```
parse_variable_declaration_tail (c, x, gt, itab_cls_GLOBAL, type, ptrlevel, id);
```

4 番目の引数は, 解析中の変数のクラスを表す itab_cls_t という名前の列挙型であり, 次のいずれかである.

itab_cls_GLOBAL	global 変数
itab_cls_LOCAL	local 変数
itab_cls_ARG	引数
itab_cls_UNDEF	不明/未定義

ここでは global 変数の解析をするので, itab_cls_GLOBAL となる.

第 5〜7 引数には, (2) で得られた, type, ptrlevel, id を渡す.

(6) if (次の記号が ";")

(3) と同様.

(7) 文法エラー

文法エラーが発生した場合は syntax_error を呼び出す.

```
syntax_error(x, "';' expected");
```

行番号や最後に読んだトークンを表示させるのに x を渡している. 第二引数の文字列がメッセージとして表示され, コンパイルは強制終了する. ここでのメッセージは「';' を予期していた (がなかった)」の意味.

- コンパイルと動作確認

コンパイルし,

```
./mcc -t 3 declare.mc
```

を実行して,

```
at preprocess
(linenum(1) type(KW_INT) token("int"))
at parse_program
at parse_declaration_head
at parse_variable_declaration_tail
declare.mc:1:  ';' expected (last token 'int')
```

が表示されれば OK. parse_declaratin_head を実行後, 記号が括弧でないので変数宣言と判断して parse_variable_declaration_tail を実行した. ただし, いずれの関数もまだ作っていないので, このようなエラーが出て停止する.

課題 1.4 **関数 parse_declaration_head の完成**

- parse_declaration_head の処理

  引数 (code_t *c, lex_t *x, itab_basetype_t *type, int *ptrlevel, char *id) を受け取って, 「宣言頭部」の解析を行い,

  *type に宣言された変数/関数の基底型を,

  ptrlevel 宣言のポインタのレベル数 (の個数) を,

  *id に宣言された識別子を

  それぞれ設定する.

  「変数/関数の基底型」は itab_basetype_t という名前の列挙型であり, 次のいずれかの値をとる.

itab_basetype_INT	int 型
itab_basetype_CHAR	char 型
itab_basetype_UNDEF	不明/未定義

  例えば, "int **abc" に対して解析を行うと, *type=itab_basetype_INT, *ptrlevel=2, id="abc" となる.

- parse_declaration_head 基本構造

  「宣言頭部」の文法構造は

  ```
  宣言頭部 ::= 型 "*"* ID
  ```

  であるから, プログラムは次のような構造になる

  ```
  (1)「型」を読みとる
  (2) "*" の数を数える
  (3) ID を読みとる
  ```

- 処理の詳細

  (1) 「型」の読みとり

  次の記号がキーワード "int" (token_KW_INT) なら *type に itab_basetype_INT を, キーワード "char" (token_KW_CHAR) なら*type に itab_basetype_CHAR を代入し, それ以外ならエラーとすれば良い.

この後, 次の一記号を読み込むのを忘れないように.

```
if      (???)  { *type = itab_basetype_INT; }
else if (???)  { *type = itab_basetype_CHAR; }
else { syntax_error(x, "type name expected "); }
lex_get(x);
```

(2) "*" の個数

次の記号が "*" (token_STAR) である限り, *ptrlevel を 1 増やし, 次の一記号を読めばよい.

```
*ptrlevel = 0;
while (???)  {
  (???)++;
  lex_get(x);
}
```

(3) 識別子の読み込み

次の記号が識別子 (token_ID) ならば, トークンを id にコピーし, 次の記号を読み込む. そうでなければエラー.

```
if (???)  {
  strcpy(id, x->token);
  lex_get(x);
}
else { syntax_error(x, "identifier expected"); }
```

コンパイルと動作確認

この関数の最後に

```
fprintf(stderr, "type='%c', ptrlevel=%i, id=\"%s\"\n", *type, *ptrlevel, id);
```

を入れてコンパイルし,

```
./mcc declare.mc
```

を実行すると, 4 行目の "[" まで処理され, タイプ, ポインタのレベル, 識別子が次のように表示される.

```
type='I', ptrlevel=0, id="a"
type='C', ptrlevel=1, id="b"
type='I', ptrlevel=2, id="c"
type='C', ptrlevel=0, id="d"
declare.mc:4:  ';' expected (last token '[')
```

☆ 動作が確認できれば, この fprintf 文は消去しておくこと.

itab_basetype_t 型の値の表示

上の実行例で, *type の値を %c で printf すると

itab_basetype_INT は 'I'

itab_basetype_CHAR は 'C'

と表示される. これは, itab_basetype_t 型を次のように宣言してるためである (tab.h の中).

```
typedef enum
{
  itab_basetype_UNDEF = '?',
  itab_basetype_INT = 'I',
  itab_basetype_CHAR = 'C',
} itab_basetype_t;
```

識別子が変数か関数かの区別を表す itab_role_t 型や, 変数のクラスを表す itab_cls_t 型も同様である.

課題 1.5 **関数 parse_variable_declaration_tail の完成**

- parse_variable_declaration_tail の処理

  引数 (code_t *c, lex_t *x, tab_t *t, itab_cls_t cls, itab_basetype_t *type, int *ptrlevel, char *id) を受け取って, 変数宣言の後半「変数宣言尾部」を解析し, 解析した変数宣言の情報を記号表 t に登録する.

- parse_variable_declaration_tail の基本構造

  「変数宣言尾部」の文法構造は,

  ```
  変数宣言尾部 ::= ( "[" INT "]" )*
  ```

  で, 要は配列のサイズの宣言である. 配列でない変数の宣言の場合は, この部分は空列である. この部分を構文解析した後, 変数情報を記号表に書き込む.

  ```
  配列のサイズの宣言があればその解析 … (1)
  この変数のサイズの計算 … (2)
  変数情報の記号表への登録 … (3)
  ```

- 処理の詳細

  (1) 配列サイズ宣言の解析

  次の記号が "[" であれば, この次に整数リテラル, "]" が続くはずである. 2 次元以上の配列の場合にはこの繰り返しとなる.

  配列の次元数 dimension と, 読みとった各次元のサイズを整数配列 max に記録する. max は, at("parse_variable_declaration_tail"); の前に "int max[ARRAY_MAXDIMENSION]" と宣言しておく. ARRAY_MAXDIMENSION は配列の次元の上限を表すもので, mcc.c の先頭で #define されている.

  例えば, サイズの宣言が [3][10][8] であれば,

  dimension=3

  max[0]=3, max[1]=10, max[2]=8

とする.

サイズの宣言がなければ dimension=0 とする.

この部分のプログラムの概要は次の通り.

```
dimension = 0;
while(次の記号が '[') {
  assert(dimension<ARRAY_MAXDIMENSION);
  次の記号を読み込む;
  if (次の記号が整数) {
    max[dimension] = トークンの整数値;
    dimension++;
    次の記号を読み込む;
  }
  else { 文法エラー; }
  if (次の記号が ']' ) { 次の記号を読み込む; }
  else { 文法エラー; }
}
```

(2) 変数のサイズの計算

この変数が占める領域のサイズ size を計算する. この情報は後で, 各変数のアドレスの決定や, 配列の要素にアクセスする場合のアドレス計算に用いる.

配列でない場合 (dimension=0 の時) は size=1

配列の場合は, 同時に第 d 次元目の要素のサイズ elementsize[d] を計算する.

(例) 宣言が int a[3][10][8] の場合には, 次のような値になる.

	size=240	… (4)
max[0]= 3	elementsize[0]=80	… (3)
max[1]=10	elementsize[1]= 8	… (2)
max[2]= 8	elementsize[2]= 1	… (1)

(1) a[i][j][k] の個々の要素は整数なので, サイズは 1 [1]. 従って, elementsize[2]=1 となる

(2) a[i][j] の各要素は, 「整数 8 個の配列」と考えることができる. 従って, elementsize[1]=max[2]×elementsize[2]=8 となる.

(3) 同様に, a[i] の各要素は, 「整数 8 個の配列が 10 個集まったもの」と見ることができるので, elementsize[0]=max[1]×elementsize[1]=80 となる.

(4) 配列全体のサイズは size=max[0]×elementsize[0] で 240 となる.

この部分のプログラムは次のようになる

```
if (dimension==0) {size = 1;}
else {
  elementsize[dimension-1] = 1;
  for (d=dimension-2; 0<=d; d--) {
    elementsize[d] = ???  * ???;
  }
  size = max[0]*elementsize[0];
}
```

(3) 記号表への登録

[1]実際の C 言語では 2 (バイト) や 4 (バイト) だったりするが, mini-C ではすべてサイズ 1 として扱う.

変数 id をポインタ t の指す記号表に登録し, 記号表内での通し番号 i を取得する.

i = tab_itab_new(t, id); で, t の指す記号表に識別子 id が登録できる. 登録された識別子には通し番号が付けられるが, tab_itab_new はその番号を返して来る.

id と同じ識別子がすでに登録されていた場合には, itab_FAIL が返って来る. この場合には「id が 2 度宣言されている」のでエラーとする.

```
i = tab_itab_new(t, id);
if (???)  {
  char errmsg[lex_TOKEN_MAXLEN+256];
  sprintf(errmsg, "multiple definition of %s", id);
  syntax_error(x, errmsg);
}
```

【メモ】3 行目の sprintf は,「文字列に対する出力」. printf や fprintf がファイルに対して出力を行なうのに対し, sprintf は第一引数の文字列 (ここでは errmsg) に対して, printf や fprintf と同じ形式の文字列を書き込む. C 言語では, 文字列の連接や整形などの処理が弱いが, sprintf を使うとで少しそれがカバーできる.

次に識別子表の i 番めの位置に情報を登録する.

```
t->itab[i].role = ???;
t->itab[i].cls = ???;
t->itab[i].basetype = ???;
t->itab[i].ptrlevel = ???;
t->itab[i].argc = ???;
t->itab[i].size = ???;
t->itab[i].address = ???;
t->itab_vsize += size;
```

- role … この識別子が関数のとき itab_role_FUNC, 変数のとき itab_role_VAR をセットする. 変数宣言の構文解析をしているところなので, ここは itab_role_VAR を代入する.
- cls … 引数で渡される cls をそのまま代入.
- basetype … 引数で渡される type をそのまま代入.
- ptrlevel … 引数で渡される ptrlevel をそのまま代入.
- argc … 識別子が関数のときにはその関数が取る引数の数を, 変数のときには配列の次元数をセットする. 今は変数宣言の解析をしているので, 配列の次元数 (dimension) を代入する.
- size … (2) で計算した size を代入
- address … t->itab_vsize を代入

  変数は宣言された順に 0 番地から配置されていく. t->itab_vsize は, 記号表に登録された変数のサイズの合計を表す. 従って, 新しい変数の (先頭) 番地は t->itab_vsize に一致する.

  この後, t->itab_vsize をこの変数のサイズだけ増やしておく必要がある.
- aref … dimension が 0 なら -1. dimension が 1 以上の場合は, 配列表 atab に (2) で計算した max と elementsize を登録し, そこへのインデックスを代入

```
if (dimension==0) {
  t->itab[i].aref = ???;
}
else {
  for (d=0; d<dimension; d++) {
    int a = tab_atab_append(t, max[d], elementsize[d]);
    if (d==0) t->itab[i].aref = a;
  }
}
```

- コンパイルと動作確認

  コンパイルし,

  ```
  ./mcc -t 3 declare.mc
  ```

  を実行すると, main の引数宣言の手前 "(" まで解析が進み, 読み込んだトークンが次々に表示される. トレースモードが 3 の場合には, 次に作成する parse_function_declaration_tail の中で記号表を表示する関数が呼び出されているため, main を解析する時点での global 記号表 (gt) 内容が表示される.

```
  ...
at parse_declaration_head
(linenum(9) type(ID) token("main"))
(linenum(9) type(LPAR) token("("))
at parse_function_declaration_tail
== dump list of gt ==
[itab]          r c b pl ac ar  ad  sz
  0 a           V G I  0  0 -1   0   1
  1 b           V G C  1  0 -1   1   1
  2 c           V G I  2  0 -1   2   1
  3 d           V G C  0  1  0   3  15
  4 e           V G I  0  2  1  18 100
  5 f           V G C  1  3  3 118 210
[atab] mx elsz
  0    15    1
  1     4   25
  2    25    1
  3     3   70
  4     7   10
  5    10    1
at parse_declaration_head
declare.mc:9: type name expected  (last token '(')
```

  [itab] の r, c, b, pl, ac, ar, ad, sz は, それぞれ role, cls, basetype, ptrlevel, argc, aref, address, size を表している.

  r の V は itab_role_VAR, F は itab_role_FUNC を表す.

  c の G は itab_cls_GLOBAL, L は itab_cls_LOCAL, A は itab_cls_ARG を表す.

  b の I は itab_basetype_INT, C は itab_basetype_CHAR を表す.

  [atab] の mx, elsz はそれぞれ max, elementsize を表している.

☆ 記号表の内容, 特に ad, sz があっていることを確認せよ. d, e, f の sz があっていない場合には, 配列の要素サイズの計算のバグが疑われる.

課題 1.6 **関数 parse_variable_declaration の完成**

- 関数 parse_variable_declaration の処理

  「変数宣言」全体の解析を行う. 実際には, これまでに作成した, 「宣言頭部」と「変数宣言尾部」の解析ルーチンを続けて呼び出すだけ. 引数の宣言や local 変数の解析のために起動される.

- parse_variable_declaration の基本構造

  「変数宣言」の構文は

  ```
  変数宣言 ::= 宣言頭部 変数宣言尾部
  ```

  であり, 「宣言頭部」「変数宣言尾部」を解析する関数を続けて呼び出すだけ. 渡す引数を間違えないように注意.

  ```
  parse_declaration_head(c, x, &type, &ptrlevel, id);
  parse_variable_declaration_tail(c, x, t, cls, type, ptrlevel, id);
  ```

  必要な変数宣言を at("parse_variable_declaration"); の前に書き込むこと.

- コンパイルと動作確認

  コンパイルしてエラーがないことを確認

課題 1.7 **関数 parse_function_declaration_tail の完成**

- 関数 parse_function_declaration_tail の処理

  「関数宣言」の後半を解析する. 引数 (code_t *c, lex_t *x, tab_t *gt, int type, int ptrlevel, char *id) を受け取り, 関数, 引数, local 変数の情報を記号表に登録するとともに, 関数の本体に書かれている文のコードを生成する.

- parse_function_declaration_tail の基本構造

  「変数宣言尾部」の文法構造は,

  ```
  関数宣言尾部 ::= "(" ( ε | 変数宣言 ( "," 変数宣言 )* ) ")" 関数本体
  ```

  すなわち, 引数の解析をしたのち, 「関数本体」の解析を行なう. プログラムの構造は

  ```
  (1) local 記号表の割当て
  (2) 引数の解析
  (3) 関数の情報の記号表への登録
  (4) 「関数本体」の処理の呼び出し
  (5) local 記号表の消去
  ```

- 処理の詳細

  (1) local 記号表の宣言と割当て

  local 変数表は, この関数の引数や local 変数を記録するもので, 一つの関数の解析処理の間だけ存在する. 即ち, 関数の処理の初めに割り当て, この処理の最後 (5) で消去する.

  at("…"); の前に変数を宣言する.

  ```
  tab_t *lt;
  ```

  at("…"); の後に割り当てと初期化を行う.

```
lt = tab_new(lex_TOKEN_MAXLEN);
```

(2) 引数の解析

"(" の後が ")" でない限り int a 等の変数宣言がコンマで区切られたリストがあるはずなので, 各々に対して「変数宣言」の解析ルーチンを呼ぶ.

引数の個数を argc に記録する.

この処理のプログラムの書き方にはいろいろなバリエーションが考えられるが, 以下に一例を示す.

☆ この部分は, 記号表 gt のダンプを行っている部分 (if (trace>=mcc_TRACE_HIGH) { の行) より前に記入すること.

```
if (次の記号が "(" ) { 次の記号を読み込む; }
else { 文法エラー; }
argc = 0;
while (次の記号が ")" でない) {
  argc++;
  parse_variable_declaration(c, x, lt, itab_cls_ARG);
  if (次の記号が ")" でない) {
    if (次の記号が "," ) { 次の記号を読み込む; }
    else { 文法エラー; }
  }
}
次の記号を読み込む;
```

(3) 関数情報の記号表への登録

この関数の識別子の情報を記号表 gt に登録する.

まず tab_itab_new を呼び出して, 記号表 gt 内に関数名 id を登録し, 表の中での通し番号を取得する. id と同じ識別子がすでに登録されていた場合には tab_itab_new が itab_FAIL を返すので, 「id は既に登録されている」旨のエラーメッセージを出す.

☆ この部分は, 記号表 gt のダンプを行っている部分 (if (trace>=mcc_TRACE_HIGH) { の行) より前に記入すること.

```
i = tab_itab_new(???);
if (???)  {
  char errmsg[lex_TOKEN_MAXLEN+256];
  sprintf(errmsg, "multiple definition of %s", id);
  syntax_error(x, errmsg);
}
```

次に識別子表の i 番めの位置 (t->itab[i]) に情報を登録する.

```
gt->itab[i].role = ???;
gt->itab[i].cls = ???;
gt->itab[i].basetype = ???;
gt->itab[i].ptrlevel = ???;
gt->itab[i].argc = ???;
gt->itab[i].aref = ???;
gt->itab[i].address = ???;
```

role … 関数なので itab_role_FUNC

cls … 関数は全て global なので itab_cls_GLOBAL

basetype … 引数で渡される type を代入

ptrlevel … 引数で渡される ptrlevel を代入

argc … (2) で計算した argc を代入

aref … 関数の場合は -1

size … 現時点では不明. (「関数本体」の解析後分かる)

address … c->size を代入

c->size はこれまでに生成されたコードのサイズの合計である. 関数のコードは, 宣言された順に配置されていくので, 現在解析中の関数の先頭番地は, c->size に一致する.

(4) 「関数本体」の解析

parse_function_body を呼び出す.

この後, 関数のサイズを計算し, 記号表に書き込む.

☆ この部分は, 記号表 gt のダンプを行っている部分

```
if (trace>=mcc_TRACE_HIGH) {
  fprintf(stderr, "== dump list of gt ==\n");
  tab_dump(gt);
}
```

の後に記入すること.

```
parse_function_body(c, x, gt, lt, argc);
gt->itab[i].size = c->size - gt->itab[i].address;
```

(5) local 記号表の消去

lt に対して tab_delete() を呼び出す.

```
tab_delete(lt);
```

- コンパイルと動作確認

コンパイルし,

```
./mcc -t 3 declare.mc
```

を実行すると, sub1 の本体に入ったところまで処理が進む. 最後に, sub1 の引数の解析が終了した時点での global 記号表 (gt) と local 記号表 (lt) の内容が表示される. (lt の表示は, parse_function_body の中で行われている.)

```
…
at parse_variable_declaration_tail
(linenum(15) type(LBRACE) token("{"))
== dump list of gt ==
[itab]          r c b pl ac ar  ad  sz
  0 a           V G I  0  0 -1   0   1
  1 b           V G C  1  0 -1   1   1
  2 c           V G I  2  0 -1   2   1
  3 d           V G C  0  1  0   3  15
  4 e           V G I  0  2  1  18 100
  5 f           V G C  1  3  3 118 210
  6 main        F G I  0  0 -1   5   0
  7 sub1        F G I  0  4 -1   5   0
```

```
[atab] mx elsz
  0    15    1
  1     4   25
  2    25    1
  3     3   70
  4     7   10
  5    10    1
at parse_function_body
(linenum(15) type(KW_CHAR) token("char"))
== dump list of lt ==
[itab]         r c b pl ac ar  ad  sz
  0 p          V A I  0  0 -1   0   1
  1 q          V A C  1  0 -1   1   1
  2 r          V A I  2  0 -1   2   1
  3 b          V A I  0  0 -1   3   1
[atab] mx elsz
declare.mc:15: '}' expected (last token 'char')
```

gt には sub1 が追加されている. lt には sub1 の引数が登録されている.

課題 1.8 **関数 parse_function_body の完成**

- 関数 parse_function_body の処理

  関数本体を解析し, local 変数の記号表 lt への登録と, 書かれている文の解析およびコード生成を行う.

- parse_function_body の基本構造

  「関数本体」の文法構造は,

  ```
  関数本体 ::  = "{" (変数宣言 ";")*  文* "}"
  ```

  なので, '{' に続いて 0 個以上の変数宣言を解析し, それが終れば '}' が現れるまで文の解析を繰り返し行なえばよい.

  ただし, 文のコードを生成する前に, この関数の入口コードを生成し, 文のコードの後には出口コードを生成しなければならない.

  初期プログラムの構造は

  ```
  "{" があることの確認
  trace>=mcc_TRACE_HIGH なら記号表 lt の表示
  "}" があることの確認
  ```

  となっているが, これを

  ```
  "{" があることの確認
  変数宣言の解析 … (1)
  trace>=mcc_TRACE_HIGH なら記号表の表示
  関数の入口コードの生成 … (2)
  "}" が現れるまで文の解析を繰り返し，コードを生成 … (3)
  "}" の読み飛ばし
  関数の出口コードの生成 … (4)
  ```

  に書き換える.

- 処理の詳細

(1) 変数の宣言

変数宣言の解析は, parse_variable_declaration を呼び出せばよい.

文法上「変数宣言」と「文」は区別がつきにくいが, 先頭がキーワード int か char で始まっていれば「変数宣言」, と考えればよい.

ここで宣言される変数は全て local 変数なので, 記号表 lt を渡し, 変数クラスは itab_cls_LOCAL を指定する.

```
while (次の記号がキーワード 'int' か 'char') {
  parse_variable_declaration(c, x, lt, itab_cls_LOCAL);
  if (次の記号が ';' ) { 次の記号を読み込む; }
  else { 文法エラー; }
}
```

☆ 「キーワード 'int' か 'char'」を表す lex_t の type は token_INT や token_CHAR ではないことに注意!! (これを書くと無限ループに陥る.)

(2) 関数の入口コード

関数の本体の実行に先立ち, RA, RF, RV と local 変数の領域を確保する VSM コード

```
ISP  <RA, RF, RV のサイズ > + <local 変数領域のサイズ >
```

を生成する必要がある.

<RA, RF, RV のサイズ > は STACK_FRAME_RESERVE である. (mcc.c の冒頭の #define 文で 3 と定義している.)

ここで言う「local 変数領域」は引数を含んだものである. そのサイズは, lt->itab_vsize に等しい.

```
code_append(c, opcode_ISP, ???, 0);
```

(3) 文の解析とコード生成

「文」自身の解析とコード生成は parse_statement の呼び出しで行なえるので, '}' が現れるまで繰り返して parse_statement を呼び出せばよい.

```
while (次の記号が '}' でない) {
  parse_statement(c, x, gt, lt);
}
```

(4) 関数の出口コード

RET 命令を生成する. (return 文のない関数用.)

```
code_append(c, ???, 0, 0);
```

- コンパイルと動作確認

コンパイルし,

```
./mcc -t 3 declare.mc
```

を実行すると, declare.mc の構文解析をクリアする. トークンのタイプが type=EOF (次のトークンが無いことを表す) となり, postprocess を呼んで正常終了する.

```
…
(linenum(16) type(SEMICOLON) token(";"))
(linenum(17) type(RBRACE) token("}"))
== dump list of lt ==
[itab]          r c b pl ac ar  ad  sz
  0 p           V A I  0  0 -1   0   1
  1 q           V A C  1  0 -1   1   1
  2 r           V A I  2  0 -1   2   1
  3 b           V A I  0  0 -1   3   1
  4 x           V L C  0  0 -1   4   1
  5 y           V L I  1  0 -1   5   1
  6 z           V L I  0  2  0   6 300
[atab] mx elsz
  0    10   30
  1    30    1
(linenum(17) type(EOF) token(""))
at postprocess
```

記号表 lt のリスティング中に, sub1 の本体内で宣言された local 変数が追加されていることを確認せよ.

課題 1.9 **関数 postprocess の完成**

- 関数 postprocess の処理

  構文解析中には分からなかった変数や関数の番地等の情報がこの時点では確定しているので, 記号表やコードの修正を行う.

- 処理の詳細

  入口コードは

```
0 ISP (global 変数領域サイズ)
1 LC  (global 変数領域サイズ)
2 SB 1
3 CALL (関数 main の先頭番地)
4 EXIT
```

  であるが, preprocess で生成した時点では,

  * 0 番目と 1 番目の「global 変数領域のサイズ」が分からないので, これを仮に 0 としていた,
  * 3 番目の「関数 main の先頭番地」も分からないので, これを仮に 0 としていた,

  ので, この修正を行う.

  – 「global 変数領域のサイズ」は gt->itab_vsize に等しいので, 0 番目と 1 番目のコードの修正は, コードを書き込む (上書きする) 関数 code_set を用いて,

```
code_set(c, 0, opcode_ISP,  ????, 0);
code_set(c, 1, opcode_LC,    ????, 0);
```

  により行なえる (at("postprocess"); の後に書く).

  – 関数 main の番地の修正

関数 main の番地はグローバル記号表を見れば分かるので, 3 番目の命令のアドレスをこれに書き換える.

```
i = tab_itab_find(gt, "main"); … (1)
if (???)  { … (2)
  semantic_error("body of 'main' not defined");
}
else if (gt->itab[i].role == ???)  { … (3)
  semantic_error("'main' is declared as a variable");
}
else if (gt->itab[i].role == ???)  { … (4)
  code_set(c, 3, opcode_CALL, ???, 0);
}
else {
  assert(0); … (5)
}
```

(1) まず記号表 gt 内に "main" を探す. tab_itab_find は記号表中に識別子を探す関数であり, 見つかれば記号表の中での通し番号を返す.

(2) 関数 main がプログラム中に存在しなければ, tab_itab_find は itab_FAIL を返すので, エラーメッセージを出す. この場合は文法エラーではなく, 意味エラーとして処理するので, semantic_error を呼び出す.

(3) 記号表内にこの関数が変数 (itab_role_VAR) として登録されていれば, 「main は変数として使われている」とエラーメッセージを出す. この場合も semantic_error を呼び出す.

(4) 記号表内にこの関数が関数 (itab_role_FUNC) として登録されていれば, 正常. main のアドレスは gt->itab[i].address なので, これを 3 番目の CALL 命令の第 1 オペランドにする.

(5) それ以外の場合はあり得ない. 万一ここに来た場合にはどこかバグっているので, assert で強制終了させる.

- コンパイルと動作確認
  - コンパイルし,

    ```
    ./mcc declare.mc
    ```

    を実行すると, 何も表示されずコンパイルが終了する.

    ```
    ./mcc -t 1 declare.mc
    ```

    を実行すると, declare.mc のプログラムを表示し (lex が読み込んだ文字をそのまま表示する), 最後に EOF がでてコンパイルが終了する.
  - このとき, コンパイル結果の VSM コードが declare.vsm に書き出されていて, その内容が

    ```
    0 ISP 328
    1 LC 328
    2 SB 1
    3 CALL 5
    4 EXIT
    5 ISP 3
    6 RET
    7 ISP 309
    8 RET
    ```

となっていれば OK.

– この VSM コードは実行できる.

```
./vsm declare.vsm
```

を実行すると何も表示せず終了する. (文が全くないプログラムなので, 何も実行は行われない.)

```
./vsm -t 1 declare.vsm
```

で, 実行した命令のトレースを表示するが, これが

```
0 ISP 328 0
1 LC 328 0
2 SB 1 0
3 CALL 5 0
5 ISP 3 0
6 RET 0 0
4 EXIT 0 0
```

となっていれば OK.

これで, 一応プログラムの関数/変数宣言を解析し, 実行可能な VSM コードを生成するコンパイラの骨格が完成した.

Nagisa ISHIURA