7 構文解析の演習

演習の概要

mini-c 言語の構文解析/コード生成プログラム mcc.c を作成する.

- 演習 L で作成した字句解析ルーチン lex.c とリンクし, コンパイラ mcc を完成させる.
- テストプログラムのコンパイルと実行を行なう.

注意

- 演習は STAGE 1 ～ STAGE 4 の 4 段階からなり, それぞれに 課題 1.1 , 課題 1.2 , … の演習項目がある.
- 演習は各課題を 1 つづつ順にこなし, 一つの課題で指示されたコーディングを行なう度にコンパイルとテストを行なうこと. 特に, 今回の演習ではプログラムが大きくなるので, プログラムをまとめて打ち込んでから実行しようとすると, 間違えた箇所を発見するのが極めて難しくなり, 膨大な時間がかかることになるので, 注意すること.
- 友人と相談するのは構わないが, **プログラムのコピーは絶対にしないこと**. 発覚した場合, 本講義の単位は不可とする.

プログラム/データのダウンロードとコンパイル

1. ダウンロード
   - 講義のホームページ (http://ist.ksc.kwansei.ac.jp/~ishiura/cpl/) の「プログラム」から, 次のファイルをダウンロードする
     - mcc.c: 構文解析プログラムのテンプレート
     - tab.h: 記号表処理ルーチンのヘッダ
     - tab.c: 記号表処理ルーチンの本体
     - Makefile: make 用ファイル
     - *.mc: mc のテストプログラム
   - lex.c, lex.h および VSM のシミュレータは, これまでの演習で作成したものを用いる.
2. mcc.c のコンパイル
   - make コマンドで mcc を作るのに必要なすべてのコンパイル処理が行なわれる

     ```
     make mcc
     ```

     ※ ここで

     ```
     make:  *** No targets specified and no makefile found.  Stop.
     ```

     というエラーが出たら「Makefile がない」ということ. Makefile をダウンロードして保存した際に, ファイル名が Makefile.txt になっているとこのエラーが出る (ファイル名を修正すれば解決する).

   make が使えない場合は次のコンパイルを行なう

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

3. mcc の実行

- mcc は一応実行することができる (この時点では具体的な処理は何も行われないが).

  コマンドラインで

  ```
  ./mcc
  ```

  を打ち込むと

  ```
  syntax:  mcc [-t TRACE_LEVEL][-o file] PROG.mc
  ```

  と, mcc のコマンドラインの文法を出力する.

- mini-c プログラム declare.mc をコンパイルするには,

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

  と入力すれば良い. VSM コードは declare.vsm というファイルに書き込まれる (この時点では何も出力されないが).

- 完全なコマンド記法は,

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

  である. "-t 1" や "-o declare.vsm" は省略可能なオプションである.

  – -t はトレースオプションで, コンパイルの実行状況を表示する. mcc のデバッグに用いる.

-t 1	字句解析ルーチン (lex.c) が読み込んだ (c_get した) 文字をそのまま 1 文字づつ表示する. どこまでコンパイルが進んだかを知ることができる.
-t 2	構文解析ルーチン (mcc.c) が読み込んだ (lex_get した) トークンと, 呼び出した解析ルーチンの名前を表示する.
-t 3	-t 2 の表示に加え, 記号表の内容を表示する.

  – -o は出力ファイルの指定で, デフォルト以外のファイルに VSM コードを出力したいときに指定する.

付録 7.1 Mini-C 言語の BNF (改良版)

右は, 対応する構文解析の関数名

```
プログラム ::= ( 宣言頭部 ( 関数宣言尾部 | 変数宣言尾部 ";" ) )*          parse_program
宣言頭部 ::= 型 "*"* ID                                                  parse_declaration_head
変数宣言尾部 ::= ( "[" INT "]" )*                                        parse_variable_declaration_tail
関数宣言尾部 ::= "(" (ε | 変数宣言 ( "," 変数宣言 )* ) ")" 関数本体      parse_function_declaration_tail
関数本体 ::  = "{" (変数宣言 ";")*  文* "}"                              parse_function_body
変数宣言 ::= 宣言頭部 変数宣言尾部                                       parse_variable_declaration
型 ::= "int" | "char"
文 ::= ";" | "{" 文* "}" | if 文 | while 文 | return 文 |                 parse_statement
            関数呼出し ";" | 代入文
if 文 ::= "if" "(" 式 ")" 文 ( ε | "else" 文 )                           parse_if
while 文 ::= "while" "(" 式 ")" 文                                       parse_while
return 文 ::= "return" 式 ";"                                            parse_return
代入文 ::= 左辺式 "=" 式  ";"                                            parse_assign
左辺式 ::= "*"* 変数名 ( "[" 式 "]" )*                                   parse_lhs_expression
変数名 ::= ID
式 ::= 式2 ( ( "<" | ">" | "<=" | ">=" | "==" | "!=" ) 式2 )*            parse_expression
式2 ::= ( ε | "+" | "-" ) 式3 ( ( "+" | "-" ) 式3 )*                     parse_expression2
式3 ::= 式4 ( ( "*" | "/" | "%" ) 式4 )*                                 parse_expression3
式4 ::= "*"* 式5                                                         parse_expression4
式5 ::= INT | CHAR | | "(" 式 ")" | 関数呼出し | 変数参照                  parse_expression5
変数参照 ::= ( ε | "&" ) 変数名 ( "[" 式 "]" )*                          parse_variable_reference
関数呼出し ::= 関数名 "(" 引数リスト ")"                                 [parse_call
関数名 ::= ID
引数リスト ::= ε| 式  ( "," 式 )*
```

付録 7.2 mcc.c 内部の型, 関数, 変数

構文解析の関数以外

1. マクロ変数

ARRAY_MAXDIMENSION	配列の次元数の最大値
STACK_FRAME_RESERVE	関数のフレーム中の RA, RF, RV 用のサイズ (3)

2. 型

mcc_trace_t	デバッグ用のトレース情報出力のモードを表す列挙型

mcc_TRACE_NO	何もしない
mcc_TRACE_LOW	-t 1 に対応. 字句解析ルーチンが読み込んだ文字をそのまま出力.
mcc_TRACE_MID	-t 2 に対応. 構文解析ルーチンが読み込んだトークンと, 呼び出した解析ルーチンの名前を表示するトークン単位で出力
mcc_TRACE_HIGH	-t 3 に対応. 記号表の内容も表示する.

3. 変数

int tracemode	現在の mcc のトレースモード.

4. 関数

void arg (int argc, char **argv, char source_f[], char object_f[], mcc_trace_t *trace)	mcc の起動パラメータ argc と argv を解析し, mcc プログラムのファイル名を source_f に, VSM コードを出力するファイル名を object_f に, トレースモードを trace に設定する.
void argerr()	mcc コマンドのシンタックスを表示して終了する.
void at(char *checkpoint)	tracemode が 2 以上の時, 引数で渡される文字列 checkpoint を表示する.
void syntax_error (lex *x, char *msg)	文法エラーのメッセージ msg を現在解析中の行番号とともに出力し, 停止する.
void semantic_error(char *msg)	意味エラーのメッセージ msg を出力し, 停止する.
int id_isfunc (char *id, tab_t *gt, tab_t *lt)	id が関数なら 1 を, 変数なら 0 を返す.

付録 7.3 tab パッケージの仕様

mini-C コンパイラの記号表とその操作のための関数群.

tab.h がヘッダファイルで, tab.c が本体.

記録する情報は, 記号表 itab と配列表 atab より成る.

1. マクロ変数

itab_MAXSIZE	itab の最大エントリ数
atab_MAXSIZE	atab の最大エントリ数
itab_FAIL	記号表の検索や登録が失敗したときの返り値
atab_FAIL	配列表の検索や登録が失敗したときの返り値

2. 型

itab_role_t — 識別子の役割を表す列挙型

itab_role_UNDEF	未定義
itab_role_VAR	変数
itab_role_FUNC	関数

itab_cls_t — 識別子の記憶クラスを表す列挙型

itab_cls_UNDEF	未定義
itab_cls_GLOBAL	グローバル
itab_cls_LOCAL	ローカル
itab_cls_ARG	引数

itab_basetype_t — 識別子の基本型を表す列挙型

itab_basetype_UNDEF	未定義
itab_basetype_INT	整数型
itab_basetype_CHAR	文字型

itab_entry_t — ID 表のエントリ (1 つの識別子に対するデータ) の構造体

char *id	識別子
itab_role_t role	役割 (変数/関数)
itab_cls_t cls	記憶クラス (グローバル/ローカル/引数)
itab_basetype_t basetype	基本型 (int/char)
int ptrlevel	宣言の型につく '*' の個数
int argc	関数の場合にはその引数の数 変数の場合には配列の次元数 (通常変数は 0)
int aref	配列表へのインデックス (配列変数のみ)
int address	変数の場合にはその番地 関数の場合には先頭番地
int size	変数の場合には主記憶に占めるワード数 関数の場合にはコードの長さ

atab_entry_t — 配列表のエントリの構造体

int max	その次元の添字の最大値
int elementsize	その次元の要素サイズ

tab_t — 記号表 (全体) の構造体

int id_maxlen	識別子の長さの上限
int itab_size	ID 表の現在のエントリ数 (登録されている ID の数)
int itab_vsize	ID 表に登録された変数のサイズの合計
int atab_size	配列表の現在のエントリ数
itab_entry_t itab[itab_MAXSIZE]	ID 表
atab_entry_t atab[atab_MAXSIZE]	配列表 (i 番目に登録された変数の配列の d 次元目の情報は t->atab[t->itab[i].aref+d] でアクセスできる.)

3. 関数

tab* tab_new(int id_maxlen)	記号表のためのデータ構造を割り当て, 初期化し, そのポインタを返す. id_maxlen は, 識別子の長さの上限.
void tab_delete(tab_t *t)	t の指す記号表の使用領域を解放する.
void tab_reset(tab_t *t)	t の指す記号表のデータをクリアする.
int tab_itab_new (tab_t *t, char *id)	t の指す記号表に識別子 id を追加し, 表の何番目に登録されたかを返す (0 番から始まる). ただし, 同じ名前の識別子がすでに登録されていた場合には, itab_FAIL を返す.
int tab_itab_find (tab_t *t, char *id)	t の指す記号表中で識別子 id を検索し, 表の何番目に登録されているかを返す. 表中に id が登録されていなかった場合には itab_FAIL を返す.
int tab_atab_append (tab_t *t, int max, int elementsize)	t の指す記号表の配列次元表の末尾にデータを追加する. max はその次元の添字の最大値, elementsize はその次元の要素サイズである.
void tab_dump(tab_t *t)	t の指す記号表の内容を表示する.

付録 7.4 code パッケージの仕様

VSM コードの保持, 入力, 追加, 変更, 出力等を行なう関数群.

code.h がヘッダファイルで, code.c が本体.

1. マクロ変数

code_MAXSIZE	VSM コード中の最大命令数
opcode_BEGIN_	opcode_t 型の最初の命令の番号
opcode_END_	opcode_t 型の最後の命令の番号+1

2. 型

opcode_t	VSM 命令を表す列挙型 (VSM 命令にの先頭に opcode_ を付加)
insn_t	1 つの命令を表す構造体 opcode_t opcode : 命令コード int operand[2] : オペランド (0〜2 個)
code_t	VSM コード全体を表す構造体 int size : コードのサイズ (命令数) insn_t insn[code_MAXSIZE] : 命令

3. 関数

code* code_new()	VSM コードを保持するデータ構造を割り当て, 初期化し, そのポインタを返す.
void code_delete(code_t *c)	c の指す code データ構造の使用領域を解放する.
void code_read(code_t *c, char fn)	c の指す code データ構造に fn という名前のファイルから VSM コードを読み込む.
void code_write(code_t *c, char fn)	c の指す code データ構造が保持する VSM コードを fn という名前のファイルに書き出す.
int code_append (code_t *c, opcode_t op, int p0, int p1)	c の指す VSM コードの命令配列の末尾に VSM 命令を一つ追加し, その命令が挿入された番地を返す. p0 と p1 はそれぞれ第一, 第二オペランド. オペランドがない命令は, その値を 0 にする.
void code_set (code_t *c, int i, opcode_t op, int p0, int p1)	c の指す VSM コードの命令配列の i 番目 (i 番地) の命令を上書きする.

Nagisa ISHIURA