7 構文解析の演習

STAGE 3 変数参照, 代入文, if 文, while 文

この STAGE では, 変数の読み書き処理を完成させ, if 文と while 文を解析できるようにする.

課題 3.1	id_isfunc の完成
課題 3.2	変数の参照
課題 3.3	代入文
課題 3.4	if 文
課題 3.5	while 文
課題 3.6	文の完成

課題 3.1 id_isfunc の完成

与えられた識別子が変数か関数かを判別する関数 id_isfunc を作成する. id と 2 つの記号表 gt と lt を受取り, 記号表を見てこの判定を行う.

1. id_isfunc の処理
   - 与えられた識別子 id が関数名であれば 1 を, 変数名であれば 0 を返す.
   - 識別子が関数か変数かは, 基本的には記号表に登録されているので, それを見れば分かる. 同じ記号が global 表と local 表の両方に登録されていることがあるが, この場合には, local 表の情報が優先される.

     例) 次のようなプログラムを考える.

```
 1:  int aaa() {
 2:      putint(5);
 3:  }
 4:
 5:  int bbb() {
 6:      char aaa;
 7:      ···
 8:      putchar(aaa);
 9:  }
10:
11:  int main() {
12:      aaa();
13:      bbb();
14:  }
```

     * 8 行目の aaa は, char 型の変数である. (bbb の中で local 変数として宣言されているため, global は aaa は見えなくなる).
     * main の中に現れる aaa は, global に宣言された関数である.
   - ただし, putchar, putint, getchar, getint は組み込み関数として記号表に登録されないので, ID がこれらの関数名に一致した場合は, 例外的に関数として扱う.

- 以上をまとめると, id_isfunc の構成は例えば次のようになる.

```
int isfunc = 0;   // id が関数のとき 1, 変数のとき 0
if (id が local 記号表 lt 中にある) { …(1)
  isfunc = 0;  // 変数である …(2)
}
else if (id が "putchar", "putint", "getchar", "getint" のいずれかに一致) {
  isfunc = 1;  // 関数である
}
else {
  if (id が global 記号表 gt 中にある) {
    if      (関数として登録されている) { isfunc = 1; }
    else if (変数として登録されている) { isfunc = 0; }
    else { 内部エラー }; … (3)
  }
  else { 意味エラー (id は定義されていない);}
}
return isfunc;
```

(1) tab_itab_find(lt,id) の返り値が itab_FAIL 以外であれば id は local 記号表 lt 中に存在し, この返り値が記号表中での番号である. global 記号表の検索も同様.

☆ このときの if 文の書き方として, if (i=tab_itab_find(lt,id)>=0) とはしないように注意!! 代入演算は比較演算よりも結合が弱いので, こう書くと i には lt 記号表の検索結果ではなく, 比較演算の結果 (0 か 1) が代入されてしまう. 書くなら, 括弧を用いて代入演算が優先されるように書くこと. この手のバグは発見が極めて難しい. MISRA-C では, 代入文は独立させる (この例の場合は潔く 2 文に分ける) ことを推奨している.

(2) 関数名が local 記号表に登録されることはないので, local 記号表中に id が見つかればそれは必ず変数名である. 念のため, assert(lt->itab[i].role==itab_role_VAR) などを書いておくとバグが見つけやすい.

(3) gt->itab[i].role が itab_role_FUNC) でなければ itab_role_VAR) のはずだが, バグっているとそうとも限らない (itab_role_UNDEF かも知れない) ので, 念のために 3 つに分けておく方がいいだろう. 内部エラーのメッセージを丁寧に書ければそれにこしたことはないが, 面倒なときはとにかく assert(0); と書くなどして, ガードしておく.

2. 確認

コンパイルし, エラーが出ないことを確認

課題 3.2 **変数の参照**

変数の読み出しができるようにする.

1. 「式」から変数参照ができるようにする.

- 「式５」を

```
式５ ::= CHAR
       | INT
       | "(" 式 ")"
       | 関数呼出し
       | 変数参照
```

に拡張する (これで「式５」はフルスペックが完成する).

- 「関数呼出し」の場合も「変数参照」の場合も, いずれも先頭記号は識別子のため, 区別がつかない. したがって, 先頭記号が識別子だった場合は, id_isfunc を呼んでどちらかを調べ, 関数なら「関数呼出し」を処理する parse_call を, 変数なら「変数参照」を処理する parse_variable_referenc を呼び出すようにする.

```
…
else if (次の記号が識別子) {
  if (識別子が関数) { parse_call(c, x, gt, lt); }
  else              { parse_variable_reference(c, x, gt, lt); }
}
```

2. 「変数参照」の処理 (parse_variable_reference)

- 「変数参照」の解析を行う parse_variable_reference で, ポインタ修飾や配列参照以外の変数参照ができるようにする. この場合の「変数参照」は

```
変数参照 ::= 変数名
```

であり, 文法的には識別子を 1 つ読むだけ. VSM コードは記号表から変数の local/global の区別と address を求め, もし global 変数であれば,

```
LV 0 (address)
```

local 変数であれば,

```
LV 1 (STACK_FRAME_RESERVE + address)
```

を生成すればよい. STACK_FRAME_RESERVE は関数のフレームの RV, RF, RV を格納する領域のサイズで, その値は 3 である (#define で定義されている)

☆ 記号表の参照が少しややこしいかも知れないが, プログラミングの練習と思って, 頑張ってコーディングすること.

3. 確認

- コンパイルし, エラーが出ないことを確認
- 下記のプログラム test32.mc (自分で打ち込む) をコンパイルせよ.

```
int a;
int b;
int main()
{
  int c;
  int d;
  putint(a);
  putint(b);
  putint(c);
  putint(d);
  putchar('\n');
}
```

- test32.vsm 中で, a, b, c, d への変数参照が次のようにコンパイルされていることを確認せよ.

```
 6 LV 0 0  … a
10 LV 0 1  … b
14 LV 1 3  … c
18 LV 1 4  … d
```

- test32.vsm を実行すると, 0000 が出力されるはず (値が入っていないため).

課題 3.3 代入文

変数への代入ができるようにする.

1. 「文」の拡張
   - 「文」は, これまで「関数呼び出し」しかサポートしていなかったが, 「代入文」もサポートできるよう拡張し,

```
文 ::= 関数呼出し ";"
     | 代入文
```

   とする.
   - 「関数呼出し」の場合も「代入文」の場合も, いずれも先頭記号は識別子で区別がつかない. 従って, 先頭記号が識別子だった場合は, id_isfunc を呼んでどちらかを調べ, 関数ならこれまでの処理を行ない, 変数なら「代入文」を処理する parse_assign を呼び出すようにする.

```
if (次の記号が識別子) {
  if (識別子が関数) {
    これまでに作った関数呼び出しの処理;
  }
  else {
    parse_assign(c, x, gt, lt);
  }
}
else 文法エラー;
```

   parse_assign の完成
   - 左辺の変数にポインタ修飾がある場合は除いて, 代入文が処理できるようにする. ここでの代入文の文法は,

```
代入文 ::= 左辺式 "=" 式  ";"
```

   である.

   「左辺式」の解析は parse_lhs_expression(c, x, gt, lt) を呼び出して行なう (lhs は left-hand side (左辺) の略). この後, "=" 記号を確認し, 「式」の解析に parse_expression を呼び出し, 最後に ";" を確認すればよい.
   - 代入文の VSM コードは,

```
(「左辺式」のアドレスを計算するコード) …(1)
(「式」を計算するコード) …(2)
SI
```

   である.

   (1) と (2) はそれぞれ「左辺式」と「式」の解析ルーチンを呼び出すとコードが生成されるので, 代入文のコード生成としては, 最後に SI を追加するだけでよい.

2. 「左辺式」の解析 (parse_lhs_expression)

   ポインタ修飾や配列参照以外の左辺式が扱えるようにする.

   この場合の「左辺式」は

```
左辺式 ::= 変数名
```

であり, 文法的には識別子を 1 つ読み, global 変数であれば,

```
LA 0 (address)
```

local 変数であれば,

```
LA 1 (STACK_FRAME_RESERVE + address)
```

を生成すればよい. 基本的に, 現在の parse_variable_reference と同じで, コード生成の際の LV を LA にするだけでよい.

3. 確認

- コンパイルし, エラーが出ないことを確認
- 下記のプログラム test33.mc をコンパイルせよ.

```
int a;
int b;
int main()
{
  int c;
  int d;
  a = 7; putint(a); putchar('\n');
  c = 4; putint(c); putchar('\n');
  b = a+c; putint(b); putchar('\n');
  d = a*c; putint(d); putchar('\n');
}
```

- 生成されたコード test33.vsm を実行し,

```
7
4
11
28
```

が出力されればよい.

課題 3.4 **if 文**

「文」で「if 文」が扱えるようにする

1. 「文」 (parse_statement) の拡張

「文」を

```
文 ::= 関数呼出し ";"
     | 代入文
     | if 文
```

に拡張する.

先頭の記号が識別子ならこれまでの処理を行ない, そうでなくてキーワード "if" なら「if 文」を解析する parse_if(c, x, gt, lt) を呼び出すよう, parse_statement を拡張する.

2. 「if 文」の解析 (parse_if)

- 「if 文」の文法は,

```
if 文 ::= "if" "(" 式 ")" 文 ( ε | "else" 文 )
```

である. 1 番目の「文」を解析した後, 次の記号がキーワード "else" であれば, "else" 文 の解析を行ない, そうでなければ終了すればよい.

- これに対する VSM コードは "else" がない場合には,

```
    (式のコード)
    BZ L1
    (文のコード)
L1  …
```

であり, "else" がある場合には,

```
    (式のコード)
    BZ L1
    (1 番目の文のコード)
    B L2
L1  (2 番目の文のコード)
L2  …
```

である.

「if 文」のコード生成としては, 「式」や「文」の解析ルーチンを呼び出した後に, 分岐命令を挿入するだけである.

- 分岐命令の挿入に関して困ることは, BZ 命令や B 命令を生成する時点では, 分岐先の番地が確定していないことである. これは, 次の 2 つを使って解決する.
  - c->size がコード c のその時点でのサイズなので, 最後にコードを挿入した番地は c->size - 1 で求めることができる. (code_append) したときの返り値を用いても良い.)
  - code_set(c, address, op, p1, p2) を用いれば, 後から address 番地に命令を上書きできる

すなわち, 分岐番地の分からない命令があれば, 取り敢えず適当な値で命令を生成し, その命令の番地 (c->size - 1) を記憶しておく. 分岐番地などが確定した時点で, 記憶しておいた番地に正しい命令と分岐番地を書き込めばよい.

3. 確認
   - コンパイルし, エラーが出ないことを確認
   - プログラム if.mc をコンパイルせよ.
   - if.vsm を実行し, 2000 未満の入力に対しては 55 が 2 回, 2000 以上の入力に対しては 33 が 2 回出力されることを確認せよ.

課題 3.5 **while 文**

「文」で「while 文」が扱えるようにする

1. 「文」 (parse_statement) の拡張

「文」を

```
文 ::= 関数呼出し ";"
     | 代入文
     | if 文
     | while 文
```

に拡張する.

先頭の記号がキーワード "while" なら「while 文」を解析する parse_while(c, x, gt, lt) を呼び出すよう, parse_statement を拡張する.

2. 「while 文」の解析 (parse_while)

「while 文」 の文法は,

```
while 文 ::= "while" "(" 式 ")" 文
```

であり, これに対する VSM コードは,

```
L1  (式のコード)
    BZ L2
    (文のコード)
    B L1
L2  …
```

である. 「if 文」と同様に構文解析を行なってコードを生成が行なえるよう, parse_while を完成せよ.

3. 確認

- コンパイルし, エラーが出ないことを確認
- 下記のプログラム test35.mc (自分で打ち込む) をコンパイルせよ.

```
int main()
{
  int i;
  i=1;
  while (i<=10) i=i+1;
  putint(i);
  putchar('\n');
}
```

- これを実行し, 11 が出力されることを確認せよ.

課題 3.6 **文の完成**

「文」のその他の処理を完成する

1. 「文」 (parse_statement) の拡張

- 空文, 複文, return 文 すべてが扱えるようにする.

```
文 ::= 関数呼出し ";"
     | 代入文
     | if 文
     | while 文
     | ";"
     | "{" 文* "}"
     | return 文
```

- これまでの処理に以下を追加する.
  - 先頭記号が ";" なら, 読み飛ばして終り (「空文」の処理).
  - 先頭記号が "{" なら, (一記号読んでから) "}" が現れるまで parse_statement を呼び, 「文」を繰り返し処理する (「複合文」の処理).
  - 先頭記号がキーワード "return" なら, 「return 文」の処理をするために parse_return(c, x, gt, lt) を呼ぶ. (「return 文」の解析は後の演習で行なう.)

- さらに, 「代入文」でも左辺の変数がポインタで修飾されている (`*p=10;`) 場合には先頭が `"*"` で始まる「文」となる. これに対応するため, 先頭記号が `"*"` なら `parse_assign(c, x, gt, lt)` を呼び出すようにする. (これも, 実際の処理は後の演習で行なう.)

2. 確認
   - コンパイルし, エラーが出ないことを確認
   - プログラム `sum.mc` (整数 `n` を入力し, `1` から入力した `n` までの和を求めて出力する) をコンパイルして実行し, 正しい結果が得られることを確認せよ (10 を入力して 55 が出力されることを確認せよ).
   - プログラム `factor.mc` (整数 `n` を入力し, これを素因数分解する) をコンパイルして実行し, 正しい結果が得られることを確認せよ (304920 を入力して 2 2 2 3 3 5 7 11 11 が出力されることを確認せよ).

☆ これで「プログラムらしい」ものがコンパイルできるようになった.

Nagisa ISHIURA