7 構文解析の演習

STAGE 4 関数, ポインタ, 配列

この STAGE では, 関数呼び出し, ポインタ, 配列などの構文の解析処理ができるようにし, , mini-C のコンパイラを完成させる.

課題 4.1	関数
課題 4.2	return 文
課題 4.3	ポインタ
課題 4.4	配列

課題 4.1 関数

一般の関数が扱えるようにする

1. parse_call

   putchar, putint, getchar, getint に加えて, プログラムに定義された任意の関数が扱えるようにする.

   - 関数の呼び出しの一般形は

     ```
     関数呼出し ::= 関数名 "(" 引数リスト ")"
     引数リスト ::= ε | 式 ( "," 式 )*
     ```

     であり, 「関数名」と "(" に続いてコンマで区切られた「式」が 0 個以上並び, 最後に ")" が来るというものである.

   - putchar, putint, getchar, getint 以外の関数では, 次のコードを生成する.

     ```
     ISP (STACK_FRAME_RESERVE)
     (第 1 引数の 式 のコード)
     (第 2 引数の 式 のコード)
     …
     (第 n 引数の 式 のコード)
     ISP -(STACK_FRAME_RESERVE + n)
     CALL (関数の開始番地)
     ```

     すなわち, 引数をスタックに積み, 関数の先頭番地に分岐するコードである. 関数の開始番地は global 記号表を検索して求める.

2. 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 test41.mc (再帰呼出しを行って, 5, 4, 3, 2, 1 と表示する) をコンパイルせよ.

```
int sub(int i)
{
  if (i>0) {
    putint(i);
    putchar('\n');
    sub(i-1);
  }
}

int main()
{
  sub(5);
}
```

- 実行して

```
5
4
3
2
1
```

が出力されることを確認せよ．フレームの構築がきちんとできていないと, 無限ループに陥ることがある.

課題 4.2 **return 文**

「return 文」を処理できるようにする.

1. parse_return

   「return 文」の構文は

   ```
   return 文 ::= "return" 式 ";"
   ```

   であり, 対応する VSM コードは

   ```
   LA 1 0 …(1)
   (式のコード)
   SI
   RET
   ```

   である. (1) はフレーム中の返り値 RV のアドレスである. parse_return を完成せよ.

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 fact.mc (再帰呼出しによって階乗計算を行う) 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.
   - プログラム gcd.mc (2 数の最大公約数を求める) 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.

課題 4.3 **ポインタ**

ポインタを用いた変数の読み書きができるようにする.

1. アドレス計算 (&)

   変数参照に "&" を付けられるようにする

   ```
   変数参照 ::= ( ε | "&" ) 変数名
   ```

   VSM コードはこれまで

   ```
   LV x y
   ```

   という形であったが, これを,

   "&" が無いときにはそのまま.

   "&" があるときには

   ```
   LA x y
   ```

   となるようにすればよい.

   「式５」の処理の追加

   「式５」から「変数参照」を呼ぶのは, 先頭記号が変数名 (識別子) の 場合だけとなっているが, "&" で始まる場合にも「変数参照」を呼び出す必要がある. この処理を追加する.

2. 「式」中のポインタ修飾

   「式」に現れる "*" 演算子を処理できるようにする.

   「式４」の文法を

   ```
   式４ ::= "*"* 式５
   ```

   と拡張する. VSM コードは, 「式５」のコードの後に "*" と同じ数だけ

   ```
   LI
   ```

   を追加すればよい.

3. 「左辺式」中のポインタ修飾

   文法は,

   ```
   左辺式 ::= "*"* 変数名
   ```

   となり, VSM コードは, 「式」中の "*" 演算子と同様, 「変数名」のコードの後に "*" と同じ数だけ

   ```
   LI
   ```

   を追加すればよい.

4. 確認

   - コンパイルし, エラーが出ないことを確認
   - 次のようなポインタの計算を含むプログラム pointer.mc

```
 1:  int a;
 2:  int c;
 3:
 4:  int sub ()
 5:  {
 6:     int b;
 7:     int c;
 8:     int *p;
 9:     a=1234;
10:     b=9876;
11:     p=&a;
12:     b=(*p)+2;
13:     p=&c;
14:     *p=2000;
15:     putint(a);
16:     putchar(' ');
17:     putint(b);
18:     putchar(' ');
19:     putint(c);
20:     putchar('\n');
21:  }
22:
23:  int main ()
24:  {
25:     sub();
26:  }
```

をコンパイルして実行し,

```
1234 1236 2000
```

が出力されることを確認せよ.

- プログラム swap.mc (2 つの変数に格納されているデータを交換する関数 swap を含む) 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.

課題 4.4 **配列**

配列の読み書きができるようにする.

1. parse_array_index

   式中の配列の添字 ([10][7] など) の解析を行なう.

   この部分の文法は

   ```
   ( "[" 式 "]" )*
   ```

   VSM コードは, n 次元の配列で

   ```
   [式 1][式 2] … [式 n]
   ```

   を解析したときには

```
(式 1 のコード)
LC (1 次元目の elementsize)
MUL
ADD
(式 2 のコード)
LC (2 次元目の elementsize)
MUL
ADD
…
(式 n のコード)
LC (n 次元目の elementsize)
MUL
ADD
```

を生成すればよい. すなわち, d 次元目の式を解析してコードを出力した直後に

```
LC (d 次元目の elementsize)
MUL
ADD
```

を生成すればよい.

- 「d 次元目の elementsize」は, parse_array_index に引数として渡される, t (この配列変数が登録されている記号表へのポインタ) と i (その表中での番号) を用いて, t->atab[t->itab[i].aref+d-1].elementsize で求められる.
- ここで解析した添字の次元数が, 配列の宣言された次元数と異なっている場合には, 文法エラーとする.

☆ 配列へのアクセスには実はかなりの計算が必要であることが分かる. 一般のコンパイラでは elementsize が 2 の冪乗の場合には, 乗算をシフトで置き換えて高速化を図るが, そうでない場合には乗算が必要となり, 配列のアクセスはかなり遅くなってしまう.

2. parse_lhs_expession の完成

「左辺式」の中で配列が扱えるようにする

文法は

```
左辺式 ::= "*"* 変数名 ( "[" 式 "]" )*
```

「変数名」を解析し, それに対する LA または LV 命令を生成した直後に parse_array_index(c, x, gt, lt, t, i) を呼び出して ("[" 式 "]")* の解析を行ない, 添字のコードを生成させる.

ポインタ修飾に対する LI 命令の生成は, parse_array_index の呼び出しの後になる.

3. parse_variabl_reference の完成

「変数参照」の中で配列が扱えるようにする

文法は

```
変数参照 ::= ( ε | "&" ) 変数名 ( "[" 式 "]" )*
```

「変数名」を解析し, それに対する LA または LV 命令を生成した直後に, parse_array_index(c, x, gt, lt, t, i) を呼び出して, ("[" 式 "]")* の解析を行ない, 添字のコードを生成させる.

コード生成を, 配列以外の変数参照も含めて整理すると, 次のようになる. ただし, address は変数のフレーム内での番地, b は local 変数のとき 1, global 変数のとき 0 である (配列以外の変数参照で, これまでに計算したもの).

配列以外の変数で "&" がある場合

```
1:  LA b address
```

配列以外の変数で "&" がない場合

```
1:  LV b address
```

配列変数で "&" がある場合

```
1:  LA b address
2:  (添字のコード)
```

配列変数で "&" がない場合

```
1:  LA b address
2:  (添字のコード)
3:  LI
```

なお, 2 番目の「配列以外で "&" がない場合」を

```
1:  LA b address
2:  LI
```

と処理しても構わない. コンパイラ自身のプログラムは若干きれいになるが, 生成する命令数は増える.

4. 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 matrix.mc をコンパイルして実行し, 行列の乗算が正しく行なわれていることを確認せよ.
- プログラム ssort.mc をコンパイルして実行し, ソーティングが正しく行なわれていることを確認せよ.

☆ これで完成です. 大変お疲れ様でした. でも, これでコンパイラの仕組みがずいぶん分かったと思います.

Nagisa ISHIURA