7 構文解析の演習

STAGE 2 入出力と式の計算

この STAGE では, まず入出力部分と「式」の解析/コード生成を完成させ, mini-C で書かれた式の計算ができるようにする.

課題 2.1	putchar 関数の起動とコード生成
課題 2.2	文字リテラル
課題 2.3	putint 関数と整数リテラル
課題 2.4	加減算
課題 2.5	乗除算
課題 2.6	比較演算
課題 2.7	括弧
課題 2.8	getchar, getint 関数

課題 2.1 putchar 関数の起動とコード生成

putchar 関数の呼び出しを完成させ, 課題 2.2 と合わせて文字が出力できるようにする.

1. 文からの関数呼出し

   「文」には, 「代入文」, 「if 文」などいろいろな場合があり, parse_statement は最終的にはすべての場合を処理するものだが, ここではひとまず「関数呼出し」だけが処理できるようにする.

   この場合の「文」の構文は

   ```
   文 ::= 関数呼出し ";"
   ```

   であるから, parse_statement はひとまず次のように構成すればよい.

   ```
   parse_call を呼び出す …(1)
   コード "ISP -1" を出力する …(2)
   セミコロンがあることを確認し，次の記号を読み込む …(3)
   ```

   - (1) で parse_call を呼び出すと, 「関数呼出し」の構文解析が行なわれ, 関数呼出しのための VSM コードが生成される.
   - (2) で ISP -1 を生成するのは次の理由による. 関数に返り値がある場合は, 関数の VSM コードを実行すると返り値がスタックトップに積まれる. putchar([文字]) では, 出力する [文字] のコードが返り値としてスタックに積まれる. 関数が式の中から呼ばれる場合はこの値を用いて計算を続けるが, 「文」として呼ばれた場合はこの返り値を用いることはない. このため, ISP -1 を実行してこの値を pop するのである.
   - (3) では parse_call の呼出し後, 次の記号がセミコロンなら次の一記号を読み込む. そうでなければ文法エラーとする.

2. putchar(c) の処理

   parse_call は, putchar だけでなく一般の関数の呼出しを処理できるが, ここでは取り敢えず putchar だけが処理できるようにする.

putchar 関数の呼び出しの構文は

```
"putchar"  "("  式  ")"
```

で, これに対する VSM コードは次の通り.

```
(式 のコード)
DUP
PUTC
```

式のコードを実行すると, その式の値がスタックトップに残る. DUP は, スタックトップの値を複製し, それをスタックトップに置く命令である (結果, 同じ値が 2 つスタックの上に置かれることになる.) この値を PUTC で出力する.

PUTC で出力を行うと, スタックトップの値は pop されてしまう. putchar 関数は出力した値を返り値としてスタックトップに残さなければならないので, PUTC の前に DUP で複製を作っている.

parse_call のプログラムは, 例えば次のように構成できる.

```
if (次の記号のトークンが "putchar") { …(1)
  次の記号を読み込む;
  if (次の記号が "(") { 次の記号を読み込む; }
  else { 文法エラー; }
  「式」の解析ルーチンを呼び出す; …(2)
  if (次の記号が ")") { 次の記号を読み込む; }
  else { 文法エラー; }
  VSM コード "DUP" を生成;
  VSM コード "PUTC" を生成;
}
else { 意味エラー (関数が定義されていない); }
```

(1) x->token が文字列 "putchar" に等しいかどうかをテスト (strcmp 関数を用いよ).

(2) 式の解析ルーチン」は, parse_expression である. これに, code c と lex のデータ x, および global 記号表 gt, local 記号表 lt を渡して呼び出す.

3. 確認

- コンパイルし, エラーが出ないことを確認
- テスト実行: プログラム test22.mc (自分で打ち込むこと)

```
int main()
{
  putchar('a');
  putchar('\n');
}
```

を mcc でコンパイルすると, putchar の次の "(" まで解析が進み

```
test22.mc:3:  ')'  expected (last token ''a'')
```

というエラーで終了する.

これは, putchar の引数の「式」の解析がまだできていないためである. この部分は 課題 2.2 で完成させる.

どこまで解析が進んだかだけを見たいときは, ./mcc -t 1 や ./mcc -t 2 でコンパイルするとよい.

課題 2.2 文字リテラル

文字リテラル ('a' など) に対する構文解析とコード生成を完成させ, 文字の出力ができるようにする.

1. 「式」の処理ルーチン

一般の「式」は, 次のように 「式２」〜「式５」までの構文要素により定義されている.

```
式 ::= 式２ ( ( "<" | ">" | "<=" | ">=" | "==" | "!=" ) 式２ )*
式２ ::= ( ε | "+" | "-" ) 式３ ( ( "+" | "-" ) 式３ )*
式３ ::= 式４ ( ( "*" | "/" | "%" ) 式４ )*
式４ ::= "*"* 式５
式５ ::= INT
       | CHAR
       | "(" 式 ")"
       | 関数呼出し
       | 変数参照
```

「式」〜「式５」の構文解析ルーチンは, それぞれ次の通り.

式: parse_expression

式２: parse_expression2

式３: parse_expression3

式４: parse_expression4

式５: parse_expression5

文字リテラルは, 「式５」の CHAR の部分である. ひとまず, これだけが解析できるようにするために,

```
式 ::= 式２
式２ ::= 式３
式３ ::= 式４
式４ ::= 式５
式５ ::= CHAR
```

が処理できるようにする.

この場合には, parse_expression は単に parse_expression2 を呼び出すだけで,

```
static void parse_expression(code *c, lex *x, tab *gt, tab *lt)
{
  at("parse_expression");
  parse_expression2(c, x, gt, lt);
}
```

とすればよい. parse_expression2〜parse_expression4 も同様に, それぞれ parse_expression3〜parse_expression5 を呼び出すようにする.

「式５」の構文,

```
式５ ::= CHAR
```

に対するコードは, CHAR の内部表現をスタックに積む

```
LC (CHAR のコード)
```

である. 処理は例えば,

```
if (次の記号が token_CHAR) {
  vsm コード "LC x->val" を生成;
  次の記号を読み込む;
}
else { 文法エラー; }
```

とすればよい.

2. 確認

- コンパイルし, エラーが出ないことを確認
- テスト実行

```
./mcc test22.mc
```

でコンパイルが完了し, VSM コードが test22.vsm に生成される.

- test22.vsm の中身は,

```
 0 ISP 0
 1 LC 0
 2 SB 1
 3 CALL 5
 4 EXIT
 5 ISP 3
 6 LC 97
 7 DUP
 8 PUTC
 9 ISP -1
10 LC 10
11 DUP
12 PUTC
13 ISP -1
14 RET
```

となっており,

```
./vsm test22.vsm
```

で実行すると文字 a が出力され, 改行されるはず.

  - 文字が出力されない場合は VSM コードを点検すること. LC の引数が 0 になっている等の場合は, 字句解析ルーチンで文字コードが x->val に正しく設定されていない可能性があるので, lex.c を点検してみること.
  - 字句解析ルーチンで '\n' の処理ができていないと, コンパイルができないので, この場合は lex.c をデバッグすること.

課題 2.3 **putint 関数と整数リテラル**

整数データが出力できるようにする.

1. putint(i) の処理

parse_call に putint の処理を追加する.

putint 関数の呼び出しの構文は,

```
"putint"  "("  式  ")"
```

で, これに対する VSM コードは次の通り.

```
(式 のコード)
DUP
PUTI
```

putchar の例に習って putint の処理を追加せよ.

putchar と putint の処理内容がほとんど同じなので, 共通の処理は共有するようにプログラムを書くなど, 個人個人の趣味で適当に工夫せよ.

2. 整数リテラル

「式5」の構文を,

```
式5 ::= CHAR | INT
```

に拡張する. 対応する VSM コードは, CHAR, INT いずれの場合もその内部表現をスタックに積む

```
LC (x->val)
```

である. parse_expression5 のプログラムに整数リテラルの処理を追加せよ.

3. 確認

- コンパイルし, エラーが出ないことを確認
- テスト実行: プログラム test23.mc (自分で打ち込むこと)

```
int main()
{
  putint(1997);
  putchar('+');
  putint(5628);
  putchar('\n');
}
```

を mcc でコンパイルすると, test23.vsm にコードが生成され, これを実行すると

```
1997+5628
```

が出力される.

  - 整数値の表示がおかしい場合は, まず VSM コードの LC のオペランドが 1997, 5628 などになっているかどうかを点検してみること.
  - LC のオペランドの値がおかしい場合は, lex.c で整数リテラルを解析する際に x->val への値の設定がうまくいっていない可能性があるので, この部分をデバッグすること.

課題 2.4 **加減算**

加減算ができるように式の解析に処理を追加する.

1. 「式2」の処理 (parse_expression2) の完成

加減算の処理は「式2」の文法規則に従い, parse_expression2 で行なう. 一般形は

```
式2 ::= ( ε | "+" | "-" ) 式3 ( ( "+" | "-" ) 式3 )*
```

である.

- 前半部

```
( ε | "+" | "-" ) 式３
```

のコードは, 先頭の記号が "-" の場合は「式３」の値の符号を反転させるので,

```
(式３のコード)
INV
```

である. それ以外の場合は, 「式３」の値をそのまま残せば良いので,

```
(式３のコード)
```

である. 前半部の解析プログラムは,

```
先頭の記号の型を記憶; …(1)
if (先頭の記号が "+" か "-") { lex_get(x); }
parse_expression3(c, x, gt, lt);
if (記録した記号が "-") { …(2)
  VSM コード "INV" を生成;
}
```

などととすればよい. (1) の「型を記憶」は, 適当な変数に (1) の時点で読んでいるトークンの型を代入し, (2) の時点で参照できるようにしておくという意味である.

- 後半部

```
( ( "+" | "-" ) 式３ )*
```

は, 次の記号が "+" か "-" である限り繰り返し続き,

```
(式３のコード)
"+" なら ADD, "-" なら SUB
```

を生成すればよい. プログラムは例えば次のように構成できる.

```
次の記号の型を記憶;
while (次の記号が "+" か "-") {
  lex_get(x);
  parse_expression3(c, x, gt, lt);
  if      (記録した型が "+") { VSM コード "ADD" を生成; }
  else if (記録した型が "-") { VSM コード "SUB" を生成; }
  else { ; }
  次の記号のタイプを記憶;
}
```

2. 確認

- コンパイルし, エラーが出ないことを確認
- テスト実行: プログラム test24.mc (自分で打ち込むこと)

```
int main()
{
  putint(2+3);
  putchar('\n');
  putint(-4+5-6+7-8+9);
  putchar('\n');
}
```

を mcc でコンパイルすると, test24.vsm にコードが生成される.

- test24.vsm のコードは,

```
 0 ISP 0
 1 LC 0
 2 SB 1
 3 CALL 5
 4 EXIT
 5 ISP 3
 6 LC 2
 7 LC 3
 8 ADD
 9 DUP
10 PUTI
11 ISP -1
12 LC 10
13 DUP
14 PUTC
15 ISP -1
16 LC 4
17 INV
18 LC 5
19 ADD
20 LC 6
21 SUB
22 LC 7
23 ADD
24 LC 8
25 SUB
26 LC 9
27 ADD
28 DUP
29 PUTI
30 ISP -1
31 LC 10
32 DUP
33 PUTC
34 ISP -1
35 RET
```

のようになっていて, これを vsm で実行すると,

```
5
3
```

が出力される.

課題 2.5 **乗除算**

乗算, 除算, 剰余算ができるようにする.

1. 「式３」の処理 (parse_expression3) の完成

   乗算, 除算, 剰余算の処理は「式３」の文法規則に従い, parse_expression3 で行なう. 一般形は次の通り.

   ```
   式３ ::= 式４ ( ( "*" | "/" | "%" ) 式４ )*
   ```

   加減算と同様にして parse_expression3 を完成させよ.

2. 確認

   テスト実行: プログラム test25.mc

   ```
   int main()
   {
     putint(2*3); putchar('\n');
     putint(15/4); putchar('\n');
     putint(10%4); putchar('\n');
     putint(20*5/4%7); putchar('\n');
     putint(2*3+15/4-10%4); putchar('\n');
   }
   ```

   を mcc でコンパイルし,

   ```
   ./vsm test25.vsm
   ```

   の実行結果が

   ```
   6
   3
   2
   4
   7
   ```

   となることを確認せよ.

課題 2.6 **比較演算**

比較演算ができるようにする.

1. 「式」の処理 (parse_expression) の完成

   parse_expression を完成し, 「式」で定義されている比較演算の処理が行えるようにせよ.

   ```
   式 ::= 式２ ( ( "<" | ">" | "<=" | ">=" | "==" | "!=" ) 式２ )*
   ```

2. 確認

   下記のような mini-C プログラム test26.mc

   ```
   int main()
   {
     putint(3<2); putchar('\n');
     putint(3>2); putchar('\n');
     putint(3*2+8==9+5); putchar('\n');
   }
   ```

   をコンパイルして実行し,

```
0
1
1
```

が出力されることを確認せよ.

課題 2.7 **括弧**

括弧を含めた算術式の処理を完成させる

1. 括弧を含む式は, 「式５」の中で「式」を再帰的に用いて定義される.

   「式５」に括弧の処理を追加する

   ```
   式５ ::= CHAR
          | INT
          | "(" 式 ")"
   ```

   先頭の記号が "(" であれば, 次の記号を読み込んだ上で parse_expression を再帰的に呼び出して「式」の解析とコード生成を行ない, 最後に ")" があることを確認 (して次の記号を読み込む)」という処理を追加すればよい. ここで, "(" や ")" を読んだ後に必ず次の記号を読み込む (即ち lex_get を呼ぶ) のを忘れないよう, 注意すること.

2. 確認

   mini-C プログラム test27.mc で動作を確認せよ.

   ```
   int main()
   {
     putint(2*(3+15/(9-3))); putchar('\n');
     putint((2+3>4)+((-5+3)*2!=11)); putchar('\n');
     putint((((((3)))))); putchar('\n');
   }
   ```

   実行結果は次のようになるはず.

   ```
   10
   2
   3
   ```

課題 2.8 **getchar, getint 関数**

文字と整数データの入力ができるようにする

1. getchar(), getint() の処理

   getchar 関数の呼び出しの構文は,

   ```
   "getchar"  "(" ")"
   ```

   これに対する VSM コードは,

   ```
   GETC
   ```

getint 関数の呼び出しの構文は

```
"getint"  "(" ")"
```

これに対する VSM コードは

```
GETI
```

である.

処理を parse_call に追加せよ.

2. 「式」からの関数呼び出し

   getchar, getint 関数は, putchar, putint とは異なり, 通常「式」から呼び出されて, 返り値 (この場合は入力により得た値) が利用される. このため, 「文」からだけでなく「式」からも関数を呼び出せるようにする必要がある.

   このためには「式５」を拡張し, CHAR, INT, (式) 以外に「関数呼出し」も処理できるようにする.

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

   これは parse_expression5 において, 先頭の記号が token_CHAR でも token_INT でも token_LPAREN でもなく token_ID であれば, 「関数呼出し」処理のルーチン parse_call(c, x, gt, lt) を呼び出すようにするだけでよい.

3. 確認

   下記のプログラム test28.mc (自分で打ち込む) で動作を確認せよ.

```
int main()
{
  putchar(getchar()); putchar('\n');
  putint(getint()); putchar('\n');
  putint((getint()+getint()+getint())/3); putchar('\n');
}
```

☆ これで, 値を入力して式の計算を行うような mini-C プログラムがコンパイル・実行できるようになった.

Nagisa ISHIURA