8. Yacc による抽象構文木の構築

Mini-C 言語の構文解析系を作成する. C/C++ でプログラミングするのではなく, yacc を用いて構文析系を自動生成する. 構文解析系は, Mini-C 言語のプログラムを読み込んで, 抽象構文木を作り出す. この抽象構文木に対して実行のメソッド `run` を呼び出せばプログラムが実行されるので, これにより, Mini-C 言語のインタープリタが完成する.

☆ 課題の前半は 課題 8.11 の「関数呼び出し式」まで, 後半はそれ以降

8.1 Yacc の概要

Yacc の構文記述の主要部分は,

1. 各構文要素の文法規則の記述と,
2. その文法規則が適用される (その文法規則により還元が行われる) ときに実行されるアクションの記述

により構成される. 例えば,

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

により定義される「while 文」を考える. この構文要素に対する yacc の記述は次のようになる.

```
1:  st_while :  lex_KW_WHILE lex_LPAREN expression lex_RPAREN statement
2:  {
3:      $$ = new St_while($3, $5);
4:  }
```

1 行目が文法規則の記述である. ただし,

`st_while` は, 「while 文」

`lex_KW_WHILE` は, キーワード `while`

`lex_LPAREN` は, 左括弧 (`(`)

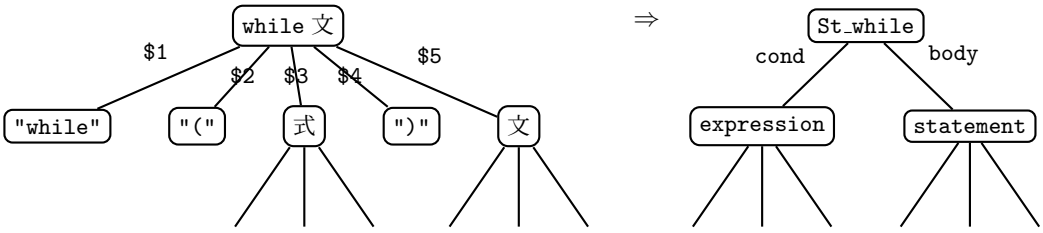
`expression` は, 「式」

`lex_RPAREN` は, 右括弧 (`)`)

`statement` は, 「文」

を表す. 基本的に while 文の文法規則をそのまま書き下せばよい.

2〜4 行目の { } に囲まれた部分がアクションの記述である. 本演習でのアクションとは, 下図のように構文要素に対応する抽象構文木を作ることである.



Yacc による構文解析の結果, while 文を構成する 5 つの各構文要素は何か一つの「値」を返すようになっていて, それが`$1`, `$2`, ···, `$5` で参照できる. 「式」の場合, その値とはその式を表す抽象構文木へのポインタであり, それが `$3` に設定される. 「文」の場合には, その式を表す抽象構文木へのポインタが `$5` に設定される. したがって, while 文の抽象構文木は, 式と文の抽象構文木へのポインタ `$3` と `$5` を用いて作り出すことができるが, それが 3

行目に書かれている．$$ に新たに作った St_while のポインタを代入しているが，これがこの while 文の構文要素の値となる．

全ての構文要素についてこのような記述を作成すれば，開始記号「プログラム」に対する構文木を得ることができる．これに対して run メソッドを呼び出せば，プログラムの実行ができる．

8.2 準備と Yacc ファイルの概要

1. 講義ホームページより次のファイルをダウンロードせよ．
   - Makefile
   - parse.yy … yacc ファイルのテンプレート
   - t822b.mc, factor.mc, gcd.mc … テスト用プログラム
2. parse.yy を見てみる

   yacc のファイルは大まかに 7 つの部分 (ここでは便宜上 Part-1 〜 Part-7 と呼ぶ) からなる．

   Part-1 (先頭の %{ の行から次の %} の行まで)

   ここに書かれたものは，yacc が自動生成する構文解析 C/C++ プログラムの冒頭にそのまま取り込まれる．ヘッダのインクルードやグローバル変数の宣言等をここに書く．

```
#include "ast.h"    … 抽象構文木クラスのヘッダ
#include <cstdio>   … C++ の中で C の stdio.h を使うためのヘッダ

// lex とのインタフェース
extern int linenum;          … 行番号
extern std::FILE* yyin;      … 入力ファイル
int yylex(void);             … 字句を 1 つ取り出す関数
void yyerror(const char*); … エラー表示用関数
```

   Part-2 (%union の宣言)

   構文要素の「値」の型を列挙する．lex の演習で，lex_INT や lex_CHAR の時に int 型の yylval.val を，lex_ID のときに char* 型の string を設定していたが，最初の状態では，この 2 つだけが宣言されている．前節の while 文の例で紹介した「式」や「文」の構文木のポインタも，ここに追加することになる．

   Part-3 (%token の宣言)

   全てのトークンをここで宣言する．下方の

```
%token <val> lex_INT lex_CHAR
%token <string> lex_ID
```

   は，トークンがどのような値を持つか (val や string は Part-2 で宣言されたもの) も併せて宣言している．(逆にいうと，これ以外のトークンは値を持たない.)

   この部分はトークンに関する情報だけなので，以後変更することはない．

   Part-4 (%type の宣言)

   Part-3 で終端記号 (すなわちトークン) の値を宣言したのに対し，Part-4 では非終端記号 (すなわち構文要素) の値の宣言を行う (演習を通じて順次追加する)．

   Part-5 (%start 宣言)

   - 文法の開始記号を宣言する．

```
%start program
```

は program が開始記号であることを宣言している.

Part-6 (文法とアクションの記述)

この部分がメインである. 前節の例のように, 各構文要素に対して文法規則とアクションを記述する.

Part-7 (最後まで)

ここに書かれたものは, 構文解析プログラムの末尾にそのまま取り込まれる. 構文解析に必要な関数の本体をここに書く.

今回は, 構文解析/インタープリタの本体 main もここに記述する. 初期状態の main は, コマンドライン引数 (argv[1]) から Mini-C プログラムの入ったファイル名を受け取り, それを構文解析する. 7 行目の yyparse(); が構文解析の呼び出しである.

```
int main(int argc, char *argv[])
{
  if(std::FILE* fp = std::fopen(argv[1], "r"))
  {
    linenum = 1;  // lex の行番号を 1 に初期設定
    yyin = fp;    // lex のファイルポインタをセット
    yyparse();    // 構文解析関数を呼び出す
  }
  else {
    printf("ファイル '%s' が開けません \n", argv[1]);
  }
}
```

3. yacc ファイルのコンパイル

yacc ファイルのコンパイルは

```
bison --debug -d parse.yy -o parse.cpp
```

で行う. ここでエラーが出たら助けを呼ぶこと.

これにより, 次のようなファイルが生成される.

- parse.cpp … 構文解析プログラム. ざっと中を見てみよ.
- parse.cpp.h (linux)/parse.hpp (Cygwin) …lex に渡すヘッダファイル.

4. lex.ll ファイルの修正 **【忘れないように】**

- lex.ll の A パートに書いていた

```
#include "lex.h"
```

を削除し, 下記のように書き換えよ. lex 用のヘッダファイルは yacc が自動生成する. 先の演習では, yacc を使っていなかったので, 手で作成したヘッダファイル lex.h を与えていたが, 今回は, yacc の生成するヘッダが読み込めるようにする必要がある. また, ast.h もインクルードするようにする.

- Cygwin の場合

```
#include "ast.h"
#include "parse.hpp"
```

- Linux の場合

```
#include "ast.h"
#include "parse.cpp.h"
```

5. 構文解析系のコンパイルとリンク

手動では面倒なので Makefile にまとめてある.

```
make mci
```

エラーが出たら助けを呼ぶこと. mci[1] は次のように, コマンドライン引数に Mini-C のプログラムが入ったファイル名を指定して起動する.

```
./mci factor.mc
```

現時点では, 何も作成していないので, エラーが出て止まる.

8.3 Yacc の演習

☆ 各課題にプログラムを添付する必要はない. 完成したプログラム一式を番最後に添付せよ.

課題 8.1 整数定数

文法規則としては

```
20.  式4 ::= 整数定数 | 文字定数 | 変数 | "(" 式 ")" | 関数呼出し式
```

の部分に該当する. 最初は一度に作ると大変なので, まず,

```
20.  式4 ::= 整数定数
```

の部分だけを作成する.

1. parse.yy の Part-2 に式のポインタを格納するメンバー expression を追加する.

```
%union {
  char* string;
  int val;
  Expression* expression;      ← これを追加
}
```

2. Part-4 に, 構文要素 expression4 の型の記述を追加.

```
// ------------------------------------------------------------------
// [Part-4] 型の宣言
// ------------------------------------------------------------------

%type <expression> expression4
```

3. Part-6 に expression4 の文法規則を加える.

[1] Mini-C Interpreter

```
// ------------------------------------------------------------------
// [Part-6] 文法とアクションの記述
// ------------------------------------------------------------------

program
:

expression4
:  lex_INT
{
  $$ = new Exp_constant(Type_INT, $1);
}
```

- 式４ ::= 整数定数 という文法規則を記述している.
- $1 はその整数定数 lex_INT の整数値である. これを第 2 引数に与えて Exp_constant のコンストラクタを呼ぶことにより, 整数定数の抽象構文木を作っている. そのポインタを $$ に代入することにより, この構文要素の値としている.

4. テストのために, program の文法規則を仮に次のように書き換える.

```
// ------------------------------------------------------------------
// [Part-6] 文法とアクションの記述
// ------------------------------------------------------------------

program
:  expression4
{
  $1->print(std::cout); std::cout<<std::endl;
}

expression4
:  lex_INT
{
  $$ = new Exp_constant(Type_INT, $1);
}
```

- つまり, 全体として

```
 1.  プログラム ::= 式４
20.  式４ ::= 整数定数
```

  という極めて簡単な文法を定義していることになる.
- program のアクションは, $1 の値, すなわち expression4 の値を print メソッドで表示するものである.

5. コンパイル

コンパイルは先程と同様である.

```
make mci
```

6. テスト

- 次のように整数値を一つだけ書いたファイル t801.mc を作成せよ. 整数値は何でも構わない. 前後のスペースや 改行等も意識する必要はない (lex が処理してくれる.)

```
2004
```

- 次により実行せよ.

```
./mci t801.mc
```

きちんと抽象構文木ができていれば, 同じ整数 (この例では 2004) が表示されるはずである.

課題 8.2 文字定数

式４の文法規則を

```
20.  式４ ::= 整数定数 | 文字定数
```

に拡張する.

1. expression4 の文法規則を, これに対応して次のように拡張する.

```
expression4
:  lex_INT
{
  $$ = new Exp_constant(Type_INT, $1);
}
| lex_CHAR
{
  $$ = new Exp_constant(Type_CHAR, $1);
}
```

   – 6 行目の | は「または」を表す.

2. テスト

   - 次のように文字定数を一つだけ書いたファイル t802.mc を作成せよ. 文字は何でも構わない.

```
'x'
```

   次により実行し, 'x' が表示されれば OK.

```
./mci t802.mc
```

課題 8.3 変数

文法規則をさらに

```
20.  式４ ::= 整数定数 | 文字定数 | 変数
```

に拡張する.

1. expression4 の文法規則をさらに拡張.

```
expression4
:  lex_INT
{
  $$ = new Exp_constant(Type_INT, $1);
}
| lex_CHAR
{
  $$ = new Exp_constant(Type_CHAR, $1);
}
| exp_variable
{
  $$ = $1;
}
```

exp_variable は「変数」を表す構文要素である. これは新しい構文要素なので, 文法規則の追加が必要.

```
11.  変数 ::= 識別子
```

なので, Part-6 に次を追加.

```
exp_variable
:  lex_ID
{
  $$ = new Exp_variable($1);
}
```

- $1 は lex_ID の値であり, 識別子の文字列である. これを引数として Exp_variable のオブジェクトを生成している.

2. 新たな構文要素 exp_variable の値に関する宣言を追加する.

- Part-2 の %union に exp_variale の値を格納するメンバー exp_variable を追加.

```
%union {
  char* string;
  int val;
  Expression* expression;
  Exp_variable* exp_variable;   ← 追加
}
```

- Part-4 に構文要素 exp_variale の値が%union の exp_variale メンバーに格納されていることを宣言.

```
// ------------------------------------------------------------------
// [Part-4] 型の宣言
// ------------------------------------------------------------------

%type <expression> expression4
%type <exp_variable> exp_variable   ← 追加
```

3. テスト

- 次のように識別子を一つだけ書いたファイル t803.mc を作成せよ. 前後のスペースや 改行等も意識する必要はない (lex が処理してくれる.)

```
xyz
```

実行して同じ識別子が表示されれば OK.

- この時点で t801.mc, t802.mc についても正しい結果が得られていることを確認せよ.

課題 8.4 乗算

式４はひとまずここで置き, 他の式の構文を先に進める.

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

の

```
19.  式３ ::= 式４ | 式３ "*" 式４
```

の部分を完成させる. この式３からは,

式４,　式４ "*" 式４,　式４ "*" 式４ "*" 式４,　…

のように, 任意回数の乗算を表す式が生成できる.

1. expression3 の文法規則を追加.

```
expression3
:  expression4
{
  $$ = $1;
}
| expression3 lex_STAR expression4
{
  $$ = new Exp_operation2(Operator_MUL, $1, $3);
}
```

2. この部分をテストするために, program の文法規則を変更する.

```
program
:  expression3      ← この行を書き換える
{
  $1->print(std::cout); std::cout<<std::endl;
}
```

3. Part-4 に expression3 の値の宣言を追加.

```
%type <expression> expression4
%type <exp_variable> exp_variable
%type <expression> expression3   ← 追加
```

4. テスト

- 次のように乗算を含む式を書いたファイル t804.mc を作成せよ.

```
  123 *   x
 *2
```

mci を実行して

```
((123 * x) * 2)
```

と表示されれば OK.

- この時点で t801.mc 〜 t803.mc についても正しい結果が得られることを確認せよ.

課題 8.5 除算と剰余算

1. expression3 の文法規則を拡張し, 除算と剰余算も処理できるようにせよ.
   なお, yacc の文法規則記述部に括弧は使えないので, 文法

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

   は,

   ```
   19.  式3 ::= 式4 | 式3 "*" 式4 | 式3 "/" 式4 | 式3 "%" 式4
   ```

   として記述せよ.
2. テスト
   - 次のように乗算, 除算, 剰余算を含む式を書いたファイル t805.mc を作成し, mci を実行せよ.

     ```
     23 * x / 45 % c
     ```

     実行して

     ```
     (((23 * x) / 45) % c)
     ```

     と表示されれば OK.

課題 8.6 単項演算 +

式2に移る. まず,

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

のうちの

```
18.  式2 ::= ( ε | "+" ) 式3
```

の部分を完成させ, 単項演算 (+ と -) が表現できるようにする.

1. これに対応する expression2 の文法規則を追加.

   ```
   expression2
   :  expression3
   {
   $$ = $1;
   }
   | lex_PLUS expression3
   {
   $$ = new Exp_operation1(Operator_PLUS, $2);
   }
   ```

2. この部分をテストするために, program の文法規則を変更する.

   ```
   program
   :  expression2      ← この行を書き換える
   {
     $1->print(std::cout); std::cout<<std::endl;
   }
   ```

3. Part-4 に expression2 の値の宣言を追加.

```
%type <expression> expression4
%type <exp_variable> exp_variable
%type <expression> expression3
%type <expression> expression2   ← 追加
```

4. テスト

   - 次のように単項演算子 + を含む式を書いたファイル t806.mc を作成し, mci を実行せよ.

     ```
     + 81 / a * 23
     ```

     実行して

     ```
     (+((81 / a) * 23))
     ```

     と表示されれば OK.

課題 8.7 単項演算 –

式 2 を

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

まで拡張せよ.

1. テスト

   - 次のようなファイル t807.mc を作成し, mci を実行せよ.

     ```
     -123*x
     ```

     実行して (-(123 * x)) と表示されれば OK.

   - t805.mc, t806.mc に対しても正しい結果が得られていることを確認せよ.

課題 8.8 加減算

式 2 を

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

まで拡張して, この部分の処理を完成させよ.

1. テスト

   - 次のようなファイル t808.mc を作成し, mci を実行せよ.

     ```
     -123 + a*48 - 9/xy
     ```

     実行して

     ```
     (((-123) + (a * 48)) - (9 / xy))
     ```

     と表示されれば OK.

課題 8.9 比較演算

次の文法規則を持つ「式」に対する構文要素 expression を追加し, 比較演算が行えるようにせよ.

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

program の文法規則の変更および Part-4 への追加も 課題 8.6 と同様にして行え.

1. テスト

   - 次のようなファイル t809.mc を作成し, mci を実行せよ.

     ```
     -23*a == b != 2+9 < x > y <= z >= -2
     ```

     実行して

     ```
     (((((((-(23 * a)) == b) != (2 + 9)) < x) > y) <= z) >= (-2))
     ```

     と表示されれば OK.

課題 8.10 括弧付きの式

式４に戻り, 未完成の部分を完成させる. まず, 次の下線部を追加する.

```
20.  式４ ::= 整数定数 | 文字定数 | 変数 | "(" 式 ")"
```

1. 先に作成した expression4 の規則に次を追加せよ. ??? の部分は各自考えよ.

   ```
   | ???  expression ???
   {
     $$ = $2;
   }
   ```

2. テスト

   - 次のようなファイル t810.mc を作成し, mci を実行せよ.

     ```
     (10-(x+y)*8) > a/(-2)
     ```

     実行して ((10 - ((x + y) * 8)) > (a / (-2))) と表示されれば OK.

課題 8.11 関数呼出し式

下線部を追加し, 式４を完成させる. これで「式」はすべて完成する.

```
20.  式４ ::= 整数定数 | 文字定数 | 変数 | "(" 式 ")" | 関数呼出し式
```

1. expression4 の規則に, 「関数呼出し式」を表す

   ```
   | exp_function
   {
     $$ = $1;
   }
   ```

   を追加せよ.

2. exp_function は新しい構文要素である. その文法は,

   ```
   21.  関数呼出し式 ::= 識別子 "(" 式リスト ")"
   ```

   なので, これに対応する記述を追加する. ??? は各自考えよ.

   ```
   exp_function
   :  ???  ???  explist ???
   {
     $$ = new Exp_function($1, *$3);
     delete $3;
   }
   ```

– explist はコンマで区切られた式のリストであり, 値としては expression の list へのポインタである.

– 4 行目の $3 には * が必要である. $3 はポインタであるが, Exp_function のコンストラクタは list のリファレンスを受け取るためである[2]. インスタンスが生成されるので, また, メモリリーク防止のために delete が必要になる.

3. explist も新しい構文要素である. その文法は,

```
16.  式リスト ::= ε | 式 | 式リスト "," 式
```

なので, これに対応する記述を追加する.

```
explist
:
{
  $$ = new std::list<Expression*>;   // 空のリスト
}
| expression
{
  $$ = new std::list<Expression*>;
  $$->push_back($1);
}
| explist lex_COMMA expression
{
  $1->push_back($3);
  $$ = $1;
}
```

– 現時点では少し難しいかも知れないが, 後に類似パターンが出てくるときには, これを参考に記述を作成すること.

4. Part-2 と Part-4 に, exp_function と explist の値と型の宣言を追加する.

- Part-2 に次の 1 行を追加

```
std::list<Expression*>* explist;
```

- Part-4 に次の 2 行を追加

```
%type <expression> exp_function
%type <explist> explist
```

5. テスト

- 次のようなファイル t811.mc を作成し, mci を実行せよ.

```
asum(15,a+3)*9==5
```

実行して

```
((asum(15,(a + 3)) * 9) == 5)
```

と表示されれば OK.

- これで式が完成したので, ここで一度 t801.mc 〜 t810.mc が全てエラー無く処理できるかどうかを確認せよ.

[2]ポインタを受け取れるコンストラクタを用意しておくべきだったかも知れない. C だけで書いている場合には, 全てポインタ渡しなので, あまりこのような問題は起こらない.

☆☆☆ 課題の前半はここまで ☆☆☆

課題 8.12 「文」を処理する枠組

これより「文」の処理に移る. 最初に, 「文」を処理する枠組を作る.

```
9.  文 ::= 代入文 | "{" 文リスト "}" | if 文 | while 文 | return 文 | 関数呼出し文
```

「文」の構文要素は statement であり, 文を表す抽象構文木 Statement へのポインタを構文解析時の値として持つ.

1. Part-6 に次の上記の文法規則に対応する記述を追加する.

```
statement
: st_assign {$$ = $1;}
| lex_LBRACE st_list lex_RBRACE {$$ = $2;}
| st_if {$$ = $1;}
| st_while {$$ = $1;}
| st_return {$$ = $1;}
| st_function {$$ = $1;}
```

st_assign 等については, 取り敢えずコンパイルするために, 空の規則を書いておく.

```
st_assign :  {}
st_list :  {}
st_if :  {}
st_while :  {}
st_return :  {}
st_function :  {}
```

更に, テストのために, 次の文法規則を記述しておく.

```
program
:  statement      ← この行を書き換える
{
  $1->print(std::cout); std::cout<<std::endl;
}
```

2. Part-4 に次の型を追加

```
%type <statement> statement
%type <statement> st_assign
%type <statement> st_if
%type <statement> st_while
%type <statement> st_return
%type <statement> st_function
%type <statement> st_list
```

3. Part-2 に次を追加

```
Statement* statement;
```

4. コンパイル

   make mci でエラーが出ないことを確認.

```
Statement* statement; parse.yy contains 7 useless nonterminals and 26 useless rules
parse.yy contains 1 reduce/reduce conflict.
```

等というメッセージが出るが, 気にしなくてよい.

現時点では実行しても, 各種の文の文法規則を書いていないので, Segmentation fault となる.

課題 8.13 代入文

1. Part-6 に代入文の文法規則

```
10.  代入文 ::= 変数 "=" 式  ";"
```

に対する st_assign の記述を追加する. ??? は各自考えよ.

```
st_assign
:  exp_variable ???  ???  ???
{
  $$ = new St_assign($1, $3);
}
```

2. テスト

- 次のようなファイル t813.mc を作成し, mci を実行せよ.

```
x = (a*b+c)/d ;
```

実行して

```
x = (((a * b) + c) / d);
```

等と表示されれば OK.

課題 8.14 複合文

複合文 (文リスト) に対する記述を追加する.

```
8.  文リスト ::  = ε | 文リスト 文
```

1. Part-6 に複合文に対する st_list と stlist の記述を追加する. (stlist は Part-2, 4 にも追加する.)

   ☆ stlist も st_list も「文のリスト」であるが, stlist は単なる文のリスト (list<Statement*> であるのに対し, st_list は「文のリスト」という構文要素 (St_list) である.

```
st_list
:  stlist
{
  $$ = new St_list(*$1);
  delete $1;
}


stlist
:
{
  $$ = new std::list<Statement*>;
}
| stlist statement
{
  $1->push_back($2);
  $$ = $1;
}
```

2. テスト

- 次のようなファイル t814.mc を作成し, mci を実行せよ.

```
{x = a+2; y = b*c;
{z = x+y;}}
```

結果,

```
x = (a + 2);
y = (b * c);
z = (x + y);
```

等と表示されれば OK.

課題 8.15 if 文

if 文に対する記述を追加する.

```
12.  if 文 ::= "if" "(" 式 ")" 文 | "if" "(" 式 ")" 文 "else" 文
```

1. Part-6 に記述を追加. ????????? の部分は各自考えよ.

```
st_if
:  lex_KW_IF lex_LPAREN expression lex_RPAREN statement
{
  $$ = new St_if($3, $5, NULL);
}
| ?????????
{
  $$ = ?????????;
}
```

2. テスト

- 次のようなファイル t815a.mc を作成し, mci を実行せよ.

```
if (a>0) b = a; else {b = -a;}
```

結果,

```
if ((a > 0)) {
  b = a;
}
else {
  b = (-a);
}
```

等と表示されれば OK.

- 次のようなファイル t815b.mc を作成し, mci を実行せよ.

```
if (a>0) b = a;
```

結果,

```
if ((a > 0)) {
  b = a;
}
else {
}
```

等と表示されれば OK.

課題 8.16 while 文

1. while 文に対する記述を追加せよ.

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

2. テスト

- 次のようなファイル t816.mc を作成し, mci を実行せよ.

```
while (i<n) {s = s * i; i = i + 1;}
```

結果,

```
while ((i < n)) {
  s = (s * i);
  i = (i + 1);
}
```

等と表示されれば OK.

課題 8.17 return 文

1. return 文に対する記述を追加せよ.

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

2. テスト

- 次のようなファイル t817.mc を作成し, mci を実行せよ.

```
return 9;
```

結果同じものが表示されれば OK.

課題 8.18 関数呼出し文

1. 関数呼出し文に対する記述を追加せよ.

```
15.  関数呼出し文 ::= 識別子 "(" 式リスト ")" ";"
```

☆ 課題 8.11 の「関数呼出し式」を参考にせよ. explist は「関数呼出し式」と共通に使えるので, 新たに定義しなおす必要はない.

2. テスト

- 次のようなファイル t818.mc を作成し, mci を実行せよ.

```
gcd(25,b);
```

実行して,

```
gcd(25,b);
```

が表示されれば OK.

課題 8.19 変数宣言

変数宣言の記述を追加する.

```
3.  変数宣言 ::= 型 識別子
4.  型 ::= "int" | "char"
```

1. Part-6 に「変数宣言」を表す variable_dcl と, 「型」を表す type の文法規則を追加せよ.

```
variable_dcl
:  type lex_ID
{
  $$ = new Variable($1, $2);
}

type
:  lex_KW_INT
{
  $$ = Type_INT;
}
| lex_KW_CHAR
{
  $$ = Type_CHAR;
}
```

2. Part-2 と Part-4 に, variable と type の値と型の宣言を追加する.

- Part-2 に次の 2 行を追加

```
Variable* variable;
Type type;
```

- Part-4 に次の 2 行を追加

```
%type <variable> variable_dcl
%type <type> type
```

3. テスト用に Part-6 の program の部分を次のように書き換える.

```
program
:  variable_dcl     ← この行を書き換える
{
  $1->print(std::cout); std::cout<<std::endl;
}
```

4. テスト

- 次のようなファイル t819.mc を作成し, mci を実行せよ.

```
int x
```

実行して同じものが表示されれば OK.

課題 8.20 関数宣言 (大枠)

関数宣言の記述を追加する.

```
5.  関数宣言 ::= 型 識別子 "(" 引数宣言リスト ")" "{" 変数宣言リスト 文リスト "}"
6.  引数宣言リスト ::  = ε | 変数宣言 | 引数宣言リスト "," 変数宣言
7.  変数宣言リスト ::  = ε | 変数宣言リスト 変数宣言 ";"
```

1. まず,

```
5.  関数宣言 ::= 型 識別子 "(" 引数宣言リスト ")" "{" 変数宣言リスト 文リスト "}"
```

の部分の規則を Part-6 に書き加えよ.

– 関数宣言を function_dcl,
引数宣言リストを argument_dcllist,
変数宣言リストを variable_dcllist,
文リストを st_list とする. ??? は各自考えよ.

```
function_dcl
:  ???  ???  ???  argument_dcllist ???  ???  variable_dcllist st_list ???
{
  $$ = new Function($1, $2, *$4, *$7, $8);
  delete $4;
  delete $7;
}
```

2. 次に,

```
6.  引数宣言リスト ::  = ε | 変数宣言 | 引数宣言リスト "," 変数宣言
```

の部分だが, 一度に作ると大変なので, 空の引数のみ扱えるようにする.

```
6.  引数宣言リスト ::  = ε
```

として, 記述を Part-6 に加えよ.

```
argument_dcllist :  {$$ = new std::list<Variable*>;}
```

3. 同様に

```
7.  変数宣言リスト ::  = ε | 変数宣言リスト 変数宣言 ";"
```

も, 一度に作ると大変なので, ひとまず

```
7.  変数宣言リスト ::  = ε
```

として, 記述を Part-6 に加えよ.

```
variable_dcllist :  {$$ = new std::list<Variable*>;}
```

4. Part-2 と Part-4 に, function_dcl, argument_dcllist, variable_dcllist の値と型の宣言を追加する.
   - Part-2 に次を追加

```
Function* function;
std::list<Variable*>* vars;
```

   - Part-4 に次を追加

```
%type <function> function_dcl
%type <vars> argument_dcllist
%type <vars> variable_dcllist
```

5. テスト用に Part-6 の program の部分を次のように書き換える.

```
program
:  function_dcl     ← この行を書き換える
{
  $1->print(std::cout); std::cout<<std::endl;
}
```

6. テスト
   - 次のようなファイル t820.mc を作成し, mci を実行せよ.

```
int kansuu()
{
  a = 10;
  return x + y;
}
```

   実行して

```
int kansuu()
{
  a = 10;
  return (x + y);
}
```

   等と表示されれば OK.

課題 8.21 関数宣言 (引数宣言リストの完成)

1. 引数宣言リストの部分を完成させる. Part-6 の argument_dcllist の記述を

```
6.  引数宣言リスト ::  = ε | 変数宣言 | 引数宣言リスト "," 変数宣言
```

に対応するように書き換えよ. ??? は各自考えよ.

```
argument_dcllist
:
{
  $$ = new std::list<Variable*>;
}
| ???
{
  $$ = new std::list<Variable*>;
  $$->push_back($1);
}
| ???  ???  ???
{
  $1->push_back($3);
  $$ = $1;
}
```

2. テスト

- 次のようなファイル t821a.mc を作成し, mci を実行せよ.

```
int kansuu(int n)
{
}
```

実行して同じものが表示されれば OK.

- 次のようなファイル t821b.mc を作成し, mci を実行せよ.

```
int kansuu(int n, char c, int x)
{
}
```

実行して同じものが表示されれば OK.

課題 8.22 関数宣言 (変数宣言リストの完成)

1. 変数宣言リストの部分を完成させる. Part-6 の variable_dcllist の記述を

```
7.  変数宣言リスト ::  = ε | 変数宣言リスト 変数宣言 ";"
```

に対応するように書き換えよ.

2. テスト

- 次のようなファイル t822a.mc に対して mci を実行せよ.

```
int kansuu(int n, char c, int x)
{
  int i;
  int j;
  int k;
  return x;
}
```

実行して同じものが表示されれば OK.

- 次のようなファイル t822b.mc に対して mci を実行せよ.

```
int asum(int n)
{
  int s;
  int i;
  s = 0;
  i = -n;
  while(i<=n) {
    if (i<0) {
      s = s - i;
    }
    else {
      s = s + i;
    }
    i = i + 1;
  }
  return s;
}
```

実行して同じもの (ただし, 式中の演算には括弧が付く) が表示されれば OK.

課題 8.23 プログラム全体の構文解析

いよいよプログラム全体の記述を追加し, 構文解析系を完成させる.

```
1.  プログラム ::= 宣言リスト
2.  宣言リスト ::= ε | 宣言リスト ( 変数宣言 ";" | 関数宣言 )
```

1. Part-6 の冒頭に書いていたテスト用の program の宣言

```
program
:  function_dcl
{
  $1->print(std::cout); std::cout<<std::endl;
}
```

を下のような最終形に書き換える.

```
program
:  dcllist
{
  ast = build_program($1);
}
```

- 「プログラム」 (program) は「宣言リスト」(dcllist) からなる.
- program は開始記号で, yacc による解析木の根に位置するので, その値 (抽象構文木へのポインタ) はこれまでのように $$ はなく, 別途宣言するグローバル変数 ast に設定する.
- アクション部が複雑になる場合には, 関数を別途定義してこれを呼び出すことができる. ここでは, 「宣言リスト」の抽象構文木データ構造からプログラム全体の抽象構文木を作り出す処理を, build_program という関数にして独立させている (関数の中身は後で書く).

2. Part-6 に「宣言リスト」(dcllist) の文法規則に対する記述を追加する.

```
2.  宣言リスト ::= ε | 宣言リスト ( 変数宣言 ";" | 関数宣言 )
```

「宣言リスト」は, 変数宣言 (variable_dcl) と関数宣言 (function_dcl) が混在したリストである. ここでは, 下図のような Declaration_t という構造体を使って, 変数と関数の情報をそれぞれ変数のリスト (vars) と関数のリスト (funclist) に分けて格納することにする.

Declaration_t

vars	変数への ポインタ	変数への ポインタ	変数への ポインタ
funclist	関数への ポインタ	関数への ポインタ	

追加する記述は次のようになる.

```
dcllist
:
{
  $$ = new Declaration_t;
}
| dcllist variable_dcl lex_SEMICOLON
{
  $1->vars.push_back($2);
  $$ = $1;
}
| dcllist function_dcl
{
  $1->funclist.push_back($2);
  $$ = $1;
}
```

3. Declaration_t の宣言は, ast.h で行う.

- ast.h の Return_t の下あたりに, 次の宣言を追加する.

```
struct Declaration_t
{
  std::list<Variable*> vars;
  std::list<Function*> funclist;
};
```

- この宣言の時点で Varible が何かがわからないとコンパイルできないので, 以前の演習で map のインクルードや Function の前方宣言をした部分に Variable の前方宣言を追加する.

```
#include <map>
class Function;
class Variable;     ← これを追加
```

4. Part-1 にグローバル変数 ast と関数 build_program の宣言を追加.

```
  …
int yylex(void);
void yyerror(char*);

Program* ast;                                          ← 追加
Program* build_program(Declaration_t* dd);    ← 追加
```

5. Part-2 と Part-4 に, dcllist の値と型の宣言を追加する.

   - Part-2 に次を追加

```
Declaration_t* declaration_data;
```

   - Part-4 に次を追加

```
%type <declaration_data> dcllist
```

6. Part-7 に関数 build_program の本体を記述.

   "main" という名前の関数だけ別扱いしているので, その処理をここで行う.

```
Program* build_program(Declaration_t* dd)
{
  std::list<Function*> flist;
  Function* mainf;

  // dd の funclist リスト中の関数のうち, "main" という名前の
  // ものは mainf にセットし, それ以外はリスト flist に入れる.
  std::list<Function*>::iterator f;
  for(f = dd->funclist.begin(); f != dd->funclist.end(); f++) {
    if ((*f)->name() == "main") {mainf = *f;}
    else {flist.push_back(*f);}
  }

  return new Program(dd->vars, flist, mainf);
}
```

7. Part-7 の main に ast の表示処理を追加する.

```
int main(int argc, char *argv[])
{
  if(std::FILE* fp = std::fopen(argv[1], "r"))
  {
    linenum = 1;  // lex の行番号を 1 に初期設定
    yyin = fp;    // lex のファイルポインタをセット
    yyparse();    // 構文解析関数を呼び出す
    ast->print(std::cout);       ← この行を追加
  }
  else {
    printf("ファイル '%s' が開けません \n", argv[1]);
  }
}
```

8. テスト

- ダウンロードした次の mini-C プログラム `factor.mc` に対し `mci` を実行せよ.

```
 1:  char separator;
 2:
 3:
 4:  int factor(int n)
 5:  {
 6:     int d;
 7:     d = 2;
 8:     while (((d * d)  <= n)) {
 9:        if (((n % d) == 0)) {
10:           putint(d);
11:           putchar(separator);
12:           n = (n / d);
13:        }
14:        else {
15:           d = (d + 1);
16:        }
17:     }
18:     putint(n);
19:     putchar('\n');
20:  }
21:
22:
23:  int main()
24:  {
25:     int n;
26:     putchar('n');
27:     putchar('=');
28:     n = getint();
29:     separator = '*';
30:     factor(n);
31:  }
```

実行して同じものが表示されれば OK.

課題 8.24 インタープリタの完成

抽象構文木が構築できれば, インタープリタは完成したも同然である.

1. Part-7 の `main` の

```
ast->print(std::cout);
```

を, 実行メソッド `run` を呼ぶよう

```
ast->run();
```

に書き換えよ.

2. `factor.mc` に対して `mci` を実行し, 下記のように素因数分解が行われることを確認せよ.

```
n=360
2*2*2*3*3*5
```

3. gcd.mc に対して mci を実行し, 下記のように最小公倍数が求められることを確認せよ.

```
n=48
m=36
g=12
n=9
m=8
g=1
n=0
m=0
```

☆ レポートには次のファイルを添付せよ.

- parse.yy
- lex.ll
- ast.h
- ast.cpp

以上で「コンパイラ演習」は終了です. 大変ご苦労様でした.

Nagisa ISHIURA

付録　Mini-C 言語の構文

```
 1.  プログラム ::= 宣言リスト
 2.  宣言リスト ::= ε | 宣言リスト ( 変数宣言 ";" | 関数宣言 )
 3.  変数宣言 ::= 型 識別子
 4.  型 ::= "int" | "char"
 5.  関数宣言 ::= 型 識別子 "(" 引数宣言リスト ")" "{" 変数宣言リスト 文リスト "}"
 6.  引数宣言リスト ::  = ε | 変数宣言 | 引数宣言リスト "," 変数宣言
 7.  変数宣言リスト ::  = ε | 変数宣言リスト 変数宣言 ";"
 8.  文リスト ::  = ε | 文リスト 文
 9.  文 ::= 代入文 | "{" 文リスト "}" | if 文 | while 文 | return 文 | 関数呼出し文
10.  代入文 ::= 変数 "=" 式  ";"
11.  変数 ::= 識別子
12.  if 文 ::= "if" "(" 式 ")" 文 | "if" "(" 式 ")" 文 "else" 文
13.  while 文 ::= "while" "(" 式 ")" 文
14.  return 文 ::= "return" 式 ";"
15.  関数呼出し文 ::= 識別子 "(" 式リスト ")" ";"
16.  式リスト ::= ε| 式 | 式リスト "," 式
17.  式 ::= 式2 | 式 ( "<" | ">" | "<=" | ">=" | "==" | "!=" ) 式2
18.  式2 ::= ( ε | "+" | "-" ) 式3 | 式2 ( "+" | "-" ) 式3
19.  式3 ::= 式4 | 式3 ( "*" | "/" | "%" ) 式4
20.  式4 ::= 整数定数 | 文字定数 | 変数 | "(" 式 ")" | 関数呼出し式
21.  関数呼出し式 ::= 識別子 "(" 式リスト ")"
```