情報処理演習 II

コンパイラの作成

第 4 回 Yacc と Lex によるコンパイラの作成

2005 年度担当: 中井央

概 要

これまでの総合的な演習として、プログラミング言語を念頭においた処理について述べる。

0 宿題の解答

前回の解答を挙げる。

1 の解答

Yacc 記述を図 20 に、Lex 記述を図 1 にそれぞれ示しておく。図 20 では後順で抽象構文木を訪問するように `traverse` を設定している。

```
1  %%
2  [0-9]+  { yylval.val = atoi((char*)yytext);
             return NUM; }
3  "+"     { return '+'; }
4  "*"     { return '*'; }
5  "-"     { return '-'; }
6  "/"     { return '/'; }
7  [ \t]   { /* do nothing */ }
8  "\n"    { return NL; }
9  .       { return yytext[0]; }
```

図 1: 第 3 回テキストの宿題 1 の解答例 (Lex 記述)

2 の解答

`traverse` を前順で訪問するようにすればよい。解答例は省略する。

3 の解答

Yacc 記述の解答例を図 21 に、Lex 記述の解答例を図 2 に示す。

なお、解答例では紙面の都合上、必要最低限の部分しか記述していない。実際には宣言した変数への代入を行えるようにし、「計算」が可能にするほうがよい。また、ここでは二重宣言や未宣言のエラーが発見されても、エラーメッセージを出すだけとした。

たとえば、`a.out` を実行し、次のように入力したとしよう。

```
int a;
int a;
```

この場合には、二重宣言のエラーメッセージが出る。また、`a.out` を実行し、次のように入力したとする。

```
int a;
a+b;
```

今度は未宣言変数の使用のエラーメッセージが出る。

このほか、次のような入力に対しては構文エラーとなる。

```
int a;
a+1;
int b;
```

代入文の追加、計算、についてはこれまでに行ったことをあわせればよいため、各自に任せる。

たったこれだけの、繰り返しも条件分岐もないが、これも 1 つのプログラミング言語である。

```
 1  %%
 2  int                     { return INT; }
 3  ";"                     { return SEMI; }
 4  [_a-zA-Z][_a-zA-Z0-9]*  {
 5            char* tmp2;
 6            tmp2=(char*)malloc(strlen(yytext)+1);
 7            if (tmp2 == NULL){
 8              perror("memory allocation");
 9              exit(EXIT_FAILURE);
10            }
11            strcpy(tmp2, yytext);
12            yylval = tmp2;
13            return IDENTIFIER;
14          }
15
16  [0-9]+          { return NUM; }
17  "+"             { return '+'; }
18  "*"             { return '*'; }
19  "-"             { return '-'; }
20  "/"             { return '/'; }
21  [ \t]           { /* do nothing */ }
22  "\n"            { /* do nothing */ }
23  .               { return yytext[0]; }
```

図 2: 第 3 回テキストの宿題 3 の解答例 (Lex 記述)

1 プログラミング言語の意味解析

ここでいうプログラミング言語の意味解析とは、名前の宣言とその使用について不整合がないか確認することや演算を行う際の型が正しく使われているかをチェックすることである。

以下では、まず、C 言語などの手続き型言語の意味解析を行うにあたり、知っておくべき性質などを述べ、具体的にその処理を行う方法について述べていく。

1.1 手続き型言語の意味解析

C 言語のプログラムを書く際の制約がいくつかある。構文として規定できる項目もあるが、そうでないものもある。これらを列挙してみよう。

- 名前は宣言してからでないと使用できない
- 同一スコープ内 (後述) で複数回同じ名前の宣言をしてはならない
- 演算子あるいは関数には型がある。それぞれ引数 (被演算子) の型と演算子あるいは関数により計算を行った結果の型である。定められた型以外を使用してはならない。
- スコープとは名前の有効範囲である。C 言語では { と } で囲まれた範囲内を 1 つのスコープとする。このとき、スコープを入れ子にすることができる。つまり、{ と } で囲まれた範囲内に{ と } で囲まれた部分が存在してもよい。

C 言語に関してはこれ以外にもたくさんの取り決めがあるが、ここでは省略する。他の手続き型プログラミング言語も大体このような制約をもつ。

コンパイラは入力されたソースプログラムがこれらの制約に違反していないかどうか調べなければならない。たとえば図 3 のようなプログラムを書くことができる。

```
1 hoge(int* a, int* b){
2   if (*a<*b){
3     int c;
4
5     c = *a;
6     *a = *b;
7     *b = c;
8   }
9 }
```

図 3: 入れ子スコープの例 1

ここでは 3 行目で c を宣言している。つまり、hoge の本体を表す 1 行目の { と 9 行目の } の内側にある 2 行目の { と 8 行目の } 内部で宣言を行っている。ここで仮に 8 行目の直後に printf("%d\n", c); と入れると未宣言変数のエラーとなる。つまり、c の有効範囲は 2 行目から 8 行目である。もう 1 つの例を示そう (図 4)。

この例では 8 行目と 12 行目の { と } により、内側にもスコープが存在する。このため、4 行目の a とは独立に 9 行目の a も存在する。しかし、その有効範囲は 12 行目の } までである。このプログラムを実行すると、3, 4, 3 がそれぞれ出力される。

内側のスコープにその外側にある名前と同じ名前が宣言されていた場合、外側の名前は、その内側のスコープが実行の対象である間は隠されることになる。つまり、内側のものが有効となる。

```
 1 #include <stdio.h>
 2
 3 main(){
 4   int a = 3;
 5
 6   printf("out: %d\n", a);
 7
 8   {
 9     int a = 4;
10
11     printf("in:  %d\n", a);
12   }
13
14   printf("out: %d\n", a);
15 }
```

図 4: 入れ子スコープの例 1

1.2 意味解析の具体的な処理

ここでは意味解析の処理をプログラミング言語で実装することについて述べる。

まず、宣言された情報を覚えておかなければならない。また、必要に応じて情報を検索できなければならない。情報の検索は効率よく行えることが望ましいがここでは効率についてはあまり考慮に入れないことにしよう。まずは動くものを作り、同じ動作をするにしても速くするにはどうすればよいかはその後で考え、代替のもので置き換えればよい。なお、置き換えを容易にするためには、それぞれの処理部分をできるだけ独立になるようにプログラムを設計しておく必要がある。

宣言された情報を覚えておくには「記号表」(symbol table) を用いる。意味解析のほとんどは記号表とのやり取りである。以下では、まず、記号表について説明し、次いで記号表を用いた意味解析について述べる。

1.2.1 記号表

情報を蓄え、あるいは引き出すには、前回の宿題の解答のように線形リストを用いるのでも十分である。ただし、上記の入れ子型のスコープに対応できるよう考慮する必要がある。

ここでは基本的には線形リストを使用し、入れ子型スコープの部分についてはそれを扱うためにはどうすればよいかを考えることにしよう。

ソースプログラムの解析は左から右 (ソースプログラムを眺める気分から言えば上から下) へ行われる。

図 4 を例にとろう。図 4 では、まず、全体が最初のスコープである。図 4 では main しか出てこないが、別の関数を定義することもできる。関数名も名前である点に注意する必要がある。プログラマが付けることができるものはみんな「名前」である。

3 行目の { を見つけるとそこで 1 つ内側のスコープに入ったことになる。ここでは main 関数の内部のスコープということになる。そして 4 行目の a の宣言文があり、そのスコープ内には int 型の a という変数があることになった。

8 行目ではさらに { があるため、新たなスコープに入ったことになる。ここでも a が宣言されているが、これは前に宣言されたものとは**別物**である。このため、この宣言情報も蓄えておく必要がある。

12 行目で } を見たとき、最も内側のスコープは終了したことになる。このため、**その内部で得られた情報は、これ以降では参照の対象からは外れる。** 実際にはコンパイラの設計方針などによって変わるが、ここでは、スコープから出たらそのスコープ内の宣言については破棄することにしよう。実際のコンパイラでは後のコード生成などのために情報を保存しておく。

ここまでで気づくことは、このようなデータ構造は**スタック**であるということである。つまり、最後に入れたものは最初に取り除かれる。

一方向リストを使ってスタックを実現したい場合、後の要素が 1 つ前の要素を指すようにするほうが実装としてはよい。

これまでにこのテキストで述べたリストでは先頭要素を指すポインタ h と末尾要素を指すポインタ t を用意していた。ただし、これまでは構造上、先頭から末尾へとリストをたどることはできたが、末尾から先頭へとリストをたどることはできなかった。ここでは末尾から先頭へとリストをたどるようにしなければならない。また、ここではスコープの境目を用意し、あるスコープが終了したら、そこからたどれる一番近い「境目」までの要素を破棄する必要がある[1]。

1.2.2 記号表に関する処理

記号表に対する基本的な操作は検索と登録である。

解析中に宣言が見つかればそれを記号表に追加する。

[1] もっとも、実験用のプログラムではそのままにしておくこともあるが…。使わないメモリ領域はできるだけ解放したほうがよい。

もちろん、2 重宣言であることを考える必要があるので、同一スコープ内に同じ名前が存在しないか確認する必要がある。すなわち、同一スコープ内を検索する必要がある。

また、使用部分で名前が出てきた場合にはそれがすでに宣言されているかどうか確認する必要がある。こちらは同一スコープ内だけではなく、その外側のすべてが対象となる。そして、この場合、一番最初に見つかった名前が対応する名前である。

今回の例では、次のように処理を行う。

新たなブロックに入ったとき 記号表の末尾に「ブロック」であることを意味する要素を付け加える。

宣言部分 同一ブロック内で記号表を検索し、同じ名前が見つかった場合には 2 重宣言のエラーメッセージを表示する。そうでない場合、記号表の末尾にその名前の要素を追加する。

使用部分 記号表全体に対して検索を行う。最初に見つかった名前を対応する名前とする。見つからなかった場合には未宣言のエラーメッセージを表示する。

ブロックから出るとき 末尾からそのブロックを示す要素までを削除する。

1.3 C 言語もどきの作成

それでは C 言語もどきを作成してみよう。

1.3.1 構文

まずは構文を考えよう。ここでは `int xxx;` という宣言文と、`a=...;` もしくは `a*b...;` などの式からなる使用文によって、**ブロック**を構成するとしよう。ブロックは宣言文の並び、そしてそれに続く使用文の並びを`{` と `}` で囲ったものとする。使用文の特殊な例としてブロックがあるとする。これによって入れ子型のブロックを構成できる。

この Yacc 記述 (構文のみ) を図 5 に挙げる。

なお、C 言語もどきでは各算術式ではその値を画面に表示するとする。たとえば図 6 のような入力があったとする。

この入力に対しては次のような出力となる。

```
main{
  int a;
  int b;

  a = 3;
  b = 4;

  {
    int a;
    int b;

    a = 3+4;
    b = 5+6;

    a;
    b;
  }
  a;
  b;
}
```

図 6: C 言語もどきの入力の例

```
3
4
7
11
7
11
3
4
```

すなわち、最初の代入文のそれぞれに対し、3, 4 の出力があり、内側のブロック内の代入文に対し、それぞれ 7, 11 の出力があり、内側の単項の `a`, `b` に対し、それぞれ 7, 11 の出力があり、最後に `main` ブロックの `a`, `b` に対し、それぞれ 3, 4 の出力がある。

演習 1

以下で解説をするが、この時点でこの Yacc 記述に手を加え、各自が C 言語もどきのコンパイラを作成してみよ。

1.3.2 準備 – リスト –

まずは記号表となるリストについて考えよう。これまでにもリストは出てきているのでここでは、その構造体のみ記す。

```
 1 %token NUM
 2 %token INT
 3 %token IDENTIFIER
 4 %token SEMI
 5 %token MAIN
 6 %%
 7
 8 PROGRAM : MAIN BLOCK
 9         ;
10
11 BLOCK   : '{' decl_part use_part '}'
12         ;
13
14 decl_part : decl_list
15         | /* empty */
16         ;
17
18 decl_list : decl_list  decl
19         | decl
20         ;
21
22 decl    : INT IDENTIFIER SEMI
23         ;
24
25 use_part : use_list
26         | /* empty */
27         ;
```

```
28
29 use_list : use_list LIST
30         | LIST
31         ;
32
33 LIST    : E SEMI
34         | A SEMI
35         | BLOCK
36         ;
37
38 A       : IDENTIFIER '=' E
39         ;
40
41 E       : E '+' T
42         | E '-' T
43         | T
44         ;
45
46 T       : T '*' F
47         | T '/' F
48         | F
49         ;
50
51 F       : NUM
52         | IDENTIFIER
53         | '(' E ')'
54         ;
```

図 5: C 言語もどきの構文 (ex10.y)

```
typedef
struct LIST {
  char        *name;
  int          val;
  int          kind;
  struct LIST *prev;
} list;
```

図 7: リストの要素を表す構造体

ここでは typedef を使っている。今回はメンバとしては次のものを考えに入れた。

1. 名前
2. 値
3. 種類 (識別子かブロック)
4. 直前要素へのポインタ

種類 (kind) は、新たなブロックに入ったときに記号表にその境界がわかるように登録する要素を区別するために導入した。kind の値は次の列挙型を用いて宣言する。

```
enum KIND { ID, BLOCK };
```

列挙型は実際には整数に割り振られる。ここではこのように書くことで、ID には 0 が、BLOCK には 1 が割り当てられる。この場合のように種別を決めておきたい場合に便利である。

なお、このプログラムではリストの先頭、末尾を使ってリストを管理する。

```
list *h, *t, *tmp;
```

1.3.3 式中の識別子の生起

何かが現れることを「生起」という。C 言語もどきでは式を書くことができるが、式中には識別子が現れてもよい。

式中に識別子が現れた場合 (図 5 の 52 行目)、記号表を末尾から先頭に向かって順に検索する必要がある。もし見つかればその値を利用するが、そうでなければエラーメッセージを出力する必要がある。

演習 2

記号表 (リスト) を末尾から先頭に向かって検索する関数 search_all を作成せよ。引数としては名前を受け取るとする。また、返り値は、その要素が見つかったらその構造体へのポインタをそうでない場合は NULL とする。また、search_all の結果を使用して値を取得、もしくはエラーメッセージを出力する図 5 の 52 行目に対応するアクションを記述せよ。

1.3.4 代入文の左辺の識別子

図 5 の 38 行目の代入文を表す構文に対して、左辺の識別子については記号表に登録されているかどうか検索する必要がある。もし、宣言されていなければエラーメッセージを出さなければならない。

演習 3

図 5 の 38 行目の代入文に対するアクションを記述せよ。

1.3.5 宣言文の処理

図 5 の 22 行目の宣言文を表す構文では、同一ブロック内に同じ宣言が含まれていてはならない。そうすると記号表内を検索する際、同一ブロック内で宣言された要素だけを検索の対象にしなければならない。このためには、記号表内にどこまでが同一ブロックであるかを示す要素を入れておく必要がある。

ブロックは、{ で始まり、宣言文の並び、使用文の並びが来て、} で終わる。そうすると { を読んで、宣言文の解析をする前にブロックが始まったことを表す要素 (ブロック要素と呼ぶことにする) を記号表に登録しておく必要がある。これにより、以降で宣言文が見つかるたびにブロック要素まで検索すればよい。ブロックは図 5 では 11 行目の構文で表される。しかし、これまで構文の途中でアクションを書く方法については示してこなかった。

Yacc では構文の途中でもアクションを記述できる。また、構文を少し変えて次のようにする方法もある。

```
BLOCK : '{' BLOCK_ACTION decl_part use_part '}'
      ;

BLOCK_ACTION : { /* なにかアクションを書く */ }
      ;
```

つまり、ε 生成規則を挟み、そのアクションとするのである。

実際には Yacc では次のように書くことができ、それは内部で上記のように処理されている。

```
BLOCK : '{' { /* action */ } decl_part use_part '}'
      ;
```

図 5 の 11 行目のアクションとしては、ブロックであることがわかるように要素を追加する。

要素を追加する関数 addlist は 2 つの引数をとる。1 つは登録したい名前であり、もう 1 つは名前の種類である。

このため、ここでのアクションは次のようになる。

```
{ addlist("block", BLOCK); }
```

addlist の最初の引数はなんでもよい。2 番目の引数は enum で宣言した BLOCK である。

そしてブロックの終わりではそのブロックに関する記号表要素をすべて削除する必要がある。すなわち、記号表の末尾からそのブロックを表す要素までのすべてを削除する。ここではこのための関数を delete_block とした。図 5 の 11 行目の末尾ではアクションとしてこの関数を呼び出す。

演習 4

1. 名前を登録する関数 addlist を記述せよ。この関数は引数として名前とその種類 (int 型) を受け取り、何も返さない。
2. 同一ブロック内を検索する関数 search_block を記述せよ。この関数は名前を引数に取り、要素が見つかったらその要素へのポインタを、見つからない場合には NULL を返す。
3. この関数を用いて図 5 の 22 行目のアクションを記述せよ。すなわち、検索しても見つからない場合にはその識別子を記号表の末尾に登録する。検索した結果、同一の名前が見つかった場合はエラーメッセージを出力する。
4. スコープ終了後、そのスコープに関する記号表要素を削除する関数 delete_block を記述せよ。この関数は引数として何も受け取らず、何も返さない。

1.3.6 その他のアクション

その他は基本的にこれまでにやった算術式の部分と同じである。

main 関数は図 8 のようになっている。

```
main(){
  /* initialize the list */
  h = (list*)malloc(sizeof(list));
  if (h == NULL){
    perror("memory allocation");
    exit(EXIT_FAILURE);
  }

  t = h;
  h->prev = NULL;
  h->name = "";

  yyparse();
}
```

図 8: C 言語もどきの Yacc 記述の main 関数

算術式の出力は図 5 の 33 および 34 行目に入れる。

Yacc 記述の冒頭部分は図 9 のようになっている。

1.4 関数

これまで我々の C 言語もどきでは関数あるいは手続きを導入していない。しかし、実際のプログラミング言語には関数あるいは手続きが存在する。手続きと関数の違いについてはコラムを読んでほしい。ここでは俗称として「関数」を使っている。

1.4.1 関数の宣言

関数宣言の文法は次のような感じで作成する。ここでは指針のみ示す。

まず、プログラム全体は複数の関数宣言の集まりとする。1 つの関数宣言は関数頭部とブロックからなるとする。

関数頭部はその関数の返り値型、関数名、引数リストとなる。型については次節で述べる。引数はどのスコープに入るのだろうか。関数名は他から呼ばれる必要がある。引数はその関数内部で参照される。しかし、その外側では参照されない。

```
%{
#include <stdio.h>
#include <stdlib.h>

typedef
struct LIST {
  char        *name;
  int          val;
  int          kind;
  struct LIST *prev;
} list;

list *h, *t;

list* search_block(char*);
list* search_all(char*);
void addlist(char*, int);
void delete_block();

enum KIND { ID, BLOCK };

%}

%union {
  int   val;
  char* name;
}

%type <val> E
%type <val> T
%type <val> F
%type <val> A

%token <val> NUM
%token INT
%token <name> IDENTIFIER
%token SEMI
%token MAIN
%%
```

図 9: C 言語もどきの Yacc 記述の冒頭部分

そうすると関数名と引数との間あたりが境界線のようである。

ここでは関数名を見たらまず、関数名を登録し、ブロック要素を登録する。次に引数リストの解析を行って、その引数を登録してから関数本体であるブロックの登録となる。ここではブロックに入ったことのブロック要素は登録しないようにする。これは引数リストからブロック本体までをここではブロックとみなす必要があるためである。すなわち、この関数の本体まで含めた宣言全体の解析が終了した際、次の関数宣言の処理に移る際は、関数名の直後までを記号表から削除する。

また、関数名の記号表要素にはその関数に関する情報を格納しておく。関数の返り値型や引数の個数など

関数と手続き

関数とは本来は副作用のない、一連の処理 (計算) を行うものを指して言う。手続きとは、違う言い方をすればサブルーチンのことであり、ある一連の処理 (副作用があってもよい) をまとめたもののことを言う。
C 言語ではこの意味での手続きのことを関数と読んでいるのでややこしい。
手続き型言語の場合、この区別は厳密ではない。歴史的な流れから意味を混在させて使っている場合が多い。
会話の中などで「関数」と出てきたら、それは本来の意味での (副作用のない) 関数なのか俗語 (??) としての関数なのか、皆さんには敏感になってほしい。

である。一般的なプログラミング言語の場合、関数の引数はそれぞれが型を持つ。このため、関数名の記号表要素はその引数についての情報も保存しておく。1 つの実現方法として、この情報は別のリストとし、そのリストへのポインタを関数名の要素が持つようにする方法がある。

1.4.2 関数呼び出し

関数呼び出しの部分では引数のチェックをする。引数の個数は正しいか、実引数それぞれは仮引数の型に適合しているかをチェックする。

引数の個数をチェックするには工夫がいる。次のような関数呼び出しの場合は、引数の解析の際に個数を数えればよい。

```
a = f(1,2,3);
```

しかし、実際のコードとしては次のものも考えられる。

```
a = f(g(x, y), h(a, b, c), 4);
```

このときに問題となるのは引数の解析中に関数呼び出しが見つかるため、そこでも引数を数えてしまうと f の呼び出しの引数は 6 になってしまう。

これを扱うには以下のように構文を構成し、アクションを起こす。expression は式を表わす構文であり、これまでに見たようにその何段階か下位の構文に関数呼び出しがある。

```
     fname LPAR params RPAR
...

params : params COMMA expression
         {
           $$.val = $1.val + 1;
           /* 他のアクション */
         }
       | expression
         {
           $$.val = 1;
           /* 他のアクション */
         }
       ;
```

要はコンマで区切られた要素 (expression) の個数を数えている。これにより得られた数と宣言された引数の数が一致するかどうかをチェックすればよい。

1.5 型について

このテキストでは型は整数型しかないものとして話を進めるが、実用的なプログラミング言語には通常型が存在する。ここでは型についての考え方を簡単に述べることにする。

型の種類を分類すると、C 言語での int 型や double 型など、基本となる型がまず存在する。次にこれらを組み合わせるようにして使う配列型や構造体型がある。また、ポインタ型も存在する。

基本型の名前はあらかじめ定められている名前である。これらのものは解析に入る前に記号表に登録するのが 1 つのやり方である。

記号表の 1 つの実現方法として、ある変数宣言があった場合、記号表の 1 つの要素にはその名前を登録し、その名前の種類を登録し、必要に応じてその他の情報を登録する。

型が同じであるかどうかは重要なことである。ここでいう型が同じということについては、2 通りの考えがある。1 つは型の名前が同じものは同じとみなす (名

前同値) というものと型の構造が同じであれば同じとみなす (構造同値) ものである。ここではこの問題には深入りしないが、前者の方が実装が楽である。しかし、この方法では同じ内容の構造体でも型名 (typedef でつけるかあるいは構造体のタグによる) が違えば違う型とみなされる (つまり直接代入などはできない)。

変数名に対する記号表要素は型名の要素へのポインタをもたせると比較が効率よく行える。つまり、型名を文字列として各要素が持っているとメモリを消費するし、比較にはその文字列の長さに比例した時間がかかることになる。型名も要素として登録されているのだから、その要素へのポインタを持ち、比較の際はポインタの比較をすれば効率がよい。

2 仮想計算機

次の節ではコンパイラの最終段階であるコード生成について述べる。そこでは仮想的な計算機を用意し、そのためのコードを出力するとする。現在では、Java がこのような方式をとっている。JVM(Java Virtual Machine) は仮想的に設計された計算機であり、これが実際の計算機上で動く。コンパイラは JVM 用にコードを出力すればよく、その出力されたコードは JVM が動く計算機であれば実行可能である。

2.1 仮想機械のモデル

この節では仮想機械についてその概要を述べる。詳細は参考文献 ([2] など) を調べてほしい。

計算機の仕組みとしては基本的にはスタックを用いる。みなさんは、これまでに後置記法をスタックを用いて評価する方法を学んだかもしれない。1+2*3 の後置記法は１ ２ ３ * +である。これを評価することを考慮してみよう。まず、入力の最初のものに着目する。これは 1 であり、なにかの被演算子であるから、後々に使うのでスタックにプッシュする。次の ２ も同様である。さらに ３ も同様である。この時点でスタックには３ ２ １と積まれている (右がスタックの底とする)。そして、この時点で*を入力上に見る。*は 2 項演算子であるのでスタックから 2 つのオペランドを取り出し演算する。演算結果は後続する入力中に現れる演算子の被演算子となる可能性があるのでスタックにプッシュする。この時点でスタックには６ １ と積まれている。次の入力は+である。これも 2 項演算子であるからスタックから 2 つのオペランドを取り出し演算する。同様に結果をスタックにプッシュする。これにより入力１ ２ ３ * +に対して計算を行い、結果をスタックに残すことができた。

このモデルが計算機で計算を行う基本となっている。

2.2 関数呼び出しの仕組み

ここでコンパイラが出力したコードでは関数呼び出しはどのように行われるのかを簡単に示してみよう。図 10 のようなプログラムを題材に考えてみよう。

```
 1 int fact(int n){
 2   if (n>0){
 3     return n*fact(n - 1);
 4   }
 5   else {
 6     return 1;
 7   }
 8 }
 9
10 main(){
11   int n = 10, res;
12
13   res = fact(n);
14   printf("%d\n", res);
15 }
```

図 10: 関数呼び出しの例題

2.2.1 関数呼び出しとメモリの割り当て

この例ではまず、main から実行が始まる。main 関数も実はどこかから呼ばれているのである。その詳細はここでは省略するが、関数は呼ばれた時点で、関数内で宣言された変数のためのメモリ領域を割り当てられる。

実行のモデルはスタックである。まず main が呼ばれるので main に必要なメモリをスタック上にとる。図 10 では main の中では int 型の変数 n を宣言しているのでその分のメモリをスタック上にとる[2]。

次に 13 行目の関数呼び出しにより、fact 用のメモ

[2]実際には解析をした結果、計算の途中結果を置くなどのための領域が必要になることもある。

リがスタックにプッシュされる。当然、main はまだ終わっていないから main 用に割り当てられたメモリの上の部分にプッシュされるのである。また、関数呼び出しの仕組みから考えて呼び出された側が終了しない限り、呼び出したもとが終了することはない。このことがスタックを用いている理由でもある。すなわち、スタックの底を上側にして描写するとこの時点ではスタック上では次のようにメモリが使われている。

底
main 用のメモリ
fact 用のメモリ

さて、fact はさらに fact を呼ぶかもしれない。もし呼んだなら、さらに今呼び出した分をメモリ上に割り当てる。すなわち、スタックには底の方から、main のためのメモリ領域、最初の fact 用のメモリ領域、2 番目の fact 用のメモリ領域が積まれたことになる。後は同様である。さらに呼び出しが続けばその呼び出し用にさらにメモリがスタック上に取られることになる。

底
main 用のメモリ
fact 用のメモリ (1)
fact 用のメモリ (2)

また、呼び出しが終了すればその関数用のメモリ領域はスタック上から下ろされる (実際には先頭を指すポインタが呼び出しもとのメモリ領域の先頭を指すようになる)。このため、実際にはそれぞれの呼び出しでは戻るときにどこへ戻ればよいかなどの情報もメモリ上に記憶する。

2.2.2 変数 (メモリ) の参照

今度は関数内の変数の参照について考えよう。たとえばある関数 f の中で int a, b, c; などと 3 つの宣言が行われていたとする。この関数への呼び出しがあってスタック上にメモリを確保する際、たとえば、呼び出す直前のスタックのトップから a, b, c の順に必要なメモリをとればよい。

底
...
f のためのメモリ領域
a 用のメモリ領域
b 用のメモリ領域
c 用のメモリ領域
...

このあとは、計算の必要に応じてスタックトップに計算の途中結果が載せられることもある。

各変数はスタックの基準位置 (たとえば、呼び出される直前のスタックトップ) からどのくらい離れているかということをもとにして計算することが可能となる。たとえばもし、c の値を変更したい場合には、直前のスタックトップから xx(int2 つ分) の位置に c 用のメモリ領域があるということは計算できる。

このとき、ある基準 (base) からどれだけ離れているかということをオフセット (offset) という言葉で表す。仮に int が 4 バイトだとすると上述の c は基準の位置からオフセット 8 の位置である。

入れ子型ブロックの場合は、同一ブロック内ではない変数を参照する場合もある。ブロックの中でブロックを宣言した場合、内側のブロックと外側のブロックのレベル差を 1 という。自身のブロックと参照したい変数が宣言されているブロックにレベル差がある場合にはスタック上でそれらの関係を明らかにする必要がある。つまり、上述のように同一名の関数が再帰的に呼ばれると、それぞれの呼び出しに対応してスタック上にメモリが取られる。このようなとき、そのブロックの外側で宣言されている変数はメモリ上のどこに割り当てられているのか探し出さなければならない。このテキストではこの詳細な方法については省略する。これについてはたとえば参考文献 [2] の 6 章などを参考にされたい。

2.3 PL/0 マシン

さて、ここではコンパイラ作成の演習用に開発された小さなプログラミング言語 PL/0 [1] での仮想機械を使うことにする。PL/0 言語は以下のような言語仕様となっている。

- 型は整数型のみ。配列や構造体はない。

- 四則演算、関係演算などが可能である。
- if 文と while 文がある。
- `begin` と `end` で文の列を括る[3]。
- 手続きは再帰呼び出しが可能である。
- 入出力はない (後述)。
- 定数定義はあるが、型定義はない。
- スコープは入れ子型静的スコープ規則に従う。

ここでは入出力はないと述べたが、それではプログラミング言語としては面白くないので、`read(`変数`)`、`write(`式`)`、`writeln` も扱えるとする。`writeln` は改行だけを行う。

2.4 PL/0 コード

ここでは PL/0 機械を動かすためのコードを記す ([2] より)。

(1) LOD, l, a
レベル差 l、オフセット a の変数の値をメモリから読み込みスタックにプッシュする

(2) LIT, $0, a$
定数 (整数)a をメモリから読み込みスタックにプッシュする

(3) STO, l, a
スタックのトップにある値をポップして、レベル差 l、オフセット a のメモリ位置に格納する

(4) OPR, $0, a$
a は演算子の種類を示す。a の種類にしたがって演算を行う。次は a の種類を示している。数字は a が取る値。

0. 手続きから戻る
1. 単項の $-$
2. $+$
3. $-$
4. $*$
5. /
6. スタックトップの値をポップし、奇数なら 1、そうでなければ 0 をプッシュする
7. (未使用)
8. $=$
9. $\neq$
10. $<$
11. $\geq$
12. $>$
13. $\leq$

(5) INT, $0, a$
スタック上に必要なメモリを a だけ確保する (a の分だけなにかをプッシュしたのと同じ)

(6) JMP, $0, a$
a 番地へ飛ぶ (関数の呼び出しなど)

(7) JPC, $0, a$
スタックトップの値をポップし、その値が 0(偽) なら a 番地へ飛ぶ

(8) CAL, l, a
レベル差が l、コードの先頭番地が a の手続きを呼び出す

(9) CSP, $0, a$
これは本来の PL/0 コードにはないが、a で指定した標準手続き[4] を呼び出す。

(10) LAB, $0, a$
a という整数のラベルを立てる。この命令も本来の PL/0 コードにはない

なお、この PL/0 では戻り値なしの手続き呼び出しのみ可能であるが、それでは面白くないので返り値を持つ関数を定義できるようにコードを追加した。

(10) RET, $0, a$ a は引数の個数を示す。a 個の引数をもって呼び出されて呼び出しもとに戻る際に返り値を呼び出しもとのスタックトップにプッシュする。

なお、これらのコードを受け取り、実行を行う仮想機械 `pl0i` を`~nakai/IPP2/pl0i` として置いたので適宜利用されたい。また、この仮想機械のソースプログラムを`~nakai/IPP2/pl0i_src` に置いているので適宜参考にされたい。

[3] `C` 言語の { と } に似ているが厳密には異なる。`C` 言語ではその中で新たに変数の宣言ができるが `PL/0` ではそれはできない。

[4] システムとしてあらかじめ用意している関数など。

3 コードの生成

ここでは対象の計算機を PL/0 機械としてコードの生成法について説明をする。

3.1 意味解析 – レベル、オフセット、記号表 –

以降では、意味解析まで行われていて必要な情報は得られているとする。PL/0 機械では各活性レコードとして機械が制御するために 3 つ分の要素を確保する必要がある。各局所変数については 1 つにつき 1 つ分の要素を確保すればよい。

したがって、意味解析では変数の宣言に対して概ね次のことを行う。

- ブロックに入ったとき、レベルを表す変数を 1 増やし、ブロックから出るときその変数を 1 減らす。
- ブロックに入ったとき、そのブロック内の局所変数用のオフセットを表す変数を 3 に初期化する。
- ブロック内で変数の宣言が見つかると、二重宣言ではないかどうかチェックする。二重宣言でなければ、記号表にレベルとオフセットと一緒にその字面を登録する。そしてオフセットを表す変数を 1 増やす。

以降では、変数の生起 (使用) に対し、記号表を引いた結果としての記号表の要素を `tmp` で表し、そのうちのレベルを `tmp->l` で表し、オフセットを `tmp->o` で表すとする。

ここで用いる PL/0 コードの形式は一行に次のように命令が記述されているとする (`x` や `y` には値が入る)。

```
( OPR,    x,    y )
```

3.2 プログラムの終了コード

プログラムを終了することを表すコードは次である。

```
( OPR,    0,    0 )
```

3.3 変数および定数のコード

一般にプログラミング言語では代入文の左辺には 1 つの変数のみが記述でき、右辺には式 (関数呼び出しを含む) を記述することができる。左辺は値が入れられる場所を示していると考えることができ、右辺は値そのものを表していると考えることができる。左辺に書ける値のことを左辺値 (left value) と呼び、右辺に書ける値のことを右辺値 (right value) と呼ぶ。

ここでは定数と右辺値としての変数について扱う。

2 節で述べたように PL/0 機械はスタックを使用する。演算はその節で述べたように後置記法を扱うのと同様に行われる。このため、式の中で定数もしくは変数の参照があれば、その値をスタックに載せるコードを出力する必要がある。

定数の場合は次のコードになる。

```
( LIT,    0,    a )
```

`a` の部分にはその定数の値が入る。例えば正定数 3 ならば、`( LIT,  0,  3 )` のようになる。

次に変数の場合であるがコードは次のようになる。

```
( LOD, tmp->l, tmp->o )
```

3.4 算術式のコード

スタックマシンの性質から、演算を行う場合の被演算子はスタックに積まれている。すなわち、`1+2` という演算を行う場合、1, 2 が順にスタックに積まれ、`+`の演算が行われる。1, 2 については `LIT` によるコードで、スタックに積まれているとすると、次に`+`のコードがあれば算術演算が行われる。`+`の演算は `( OPR, 0, 2 )` というコードで表される。

では、`1+2` の PL/0 コードを表してみよう。

```
( LIT,    0,    1 )
( LIT,    0,    2 )
( OPR,    0,    2 )
( OPR,    0,    0 )
```

なお、これは PL/0 機械で実行できる形式だが、実際には実行されても何も出力がない。

3.5 代入文のコード

a=3 のような代入文に対するコードは、右辺に対応するコードの並びの後ろにスタックトップの値を指定した活性レコードに保存する命令を置くことになる。a について記号表を引いた結果が tmp に入っているとすると、代入文のコードは次のようになる。

```
( STO, tmp->l, tmp->o )
```

a=1+2 のコードは次のようになる。

```
( LIT,    0,    1 )
( LIT,    0,    2 )
( OPR,    0,    2 )
( STO, tmp->l, tmp->o )
( OPR,    0,    0 )
```

tmp->l および tmp->o にはもちろん、実際の値が入るが、それは意味解析で得られた a についての情報による。

3.6 条件 (if) 文のコード

if 文は多くのプログラミング言語では次のような形をしている。

```
if 条件式 then 部 else 部
```

条件式が成立したら then 部が実行され、そうでなければ else 部が実行される。

このため、条件付のジャンプ命令 (JPC) と無条件のジャンプ命令 (JMP) を使用する。JPC 命令はスタックトップの値 (条件式を評価したもの) が偽だったら指定したアドレスへジャンプする。ジャンプ先の印を作るためには LAB 命令でラベルを作成する。

if 文に対するコードは次のようになる。

```
条件式のコード
( JPC, 0, lab1 )
then 部のコード
( JMP, 0, lab2 )
( LAB, 0, lab1 )
else 部のコード
( LAB, 0, lab2 )
```

3.7 繰り返し (while) 文のコード

while による繰り返しのコードを考えよう。while 文は次のような形をしている。

```
while 条件式 文
```

「文」の部分は複文 (C 言語で言えば { と } で囲まれた部分) の場合もある。while 文のコードもジャンプ文を使用する。while 文のコードは以下のようになる。

```
( LAB, 0, lab1 )
条件式のコード
( JPC, 0, lab2 )
文のコード
( JMP, 0, lab1 )
( LAB, 0, lab2 )
```

3.8 関数呼び出し

PL/0 コードでは、引数なしの手続き呼び出しは想定されているが関数呼び出しは想定されていない。この演習では PL/0 コードを少し拡張し、引数を持つ関数呼び出しが可能になるように考えることにする。

まず、引数なしの手続き呼び出しを考えてみよう。呼び出しをするときのコードは次のようになる。

```
( CAL,    l,    a )
```

ここで l は呼び出しをする手続きのレベルを示し、a はその先頭アドレスを示すのであった。我々がコードを作成する際には、a の部分にはラベルを入れればよい。

また、手続きから戻るには次のコードによる。

```
( OPR,    0,    0 )
```

手続き (関数) の呼び出しでは、ある関数呼び出しがあると制御はその呼ばれた側に移る。呼ばれた側で一連の仕事を終えるともといた場所に戻ってこなければならない。PL/0 の仮想機械では上の 2 つのコードによって、それぞれ呼び出しと戻りを行うことができる。

では次は関数の呼び出しを考えよう。

呼び出し側では引数を相手に渡す必要がある。呼ばれた側は、その関数内で変数の宣言などがあった場合、それ用のメモリを確保する必要がある。現在 (呼び出しの直前) のスタックトップを基準位置にして、スタック

の管理用に 3 のメモリが必要であったため、変数の個数＋ 3 の位置が呼ばれた側でのスタックトップになる。

ここでは呼び出し側は呼び出す直前にスタックに引数を順に積んでおくことにする。そうすると呼ばれた側から見ると、基準のすぐ下からいくつかがその関数への引数の入っている場所ということになる。最初に処理した引数ほど基準位置から遠く、最後に処理した引数は基準位置の直下にある。

すなわち、最後に処理した引数は基準位置から-1 の位置にあり、n 個の引数を取った場合、i 番目に処理された引数は基準位置から $i-n-1$ の位置にある。参照したいときにはこの法則にしたがって、次のようなコードを生成すればよい (i-n-1 には実際の値が入る)。

```
( LOD,    l,   i-n-1 )
```

C 言語などの return 文に相当する文を考えよう。次のような構文になろう。すなわち、キーワード return(文法中では RET) がまずあって、それに式が続く。ここでいう式には関数呼び出しも含む。

```
ret_stmt  : RET E SEMI ;
```

E に対してコード生成が行われたら、その直後に次の命令を置けばよい。

```
( RET,    0,    a )
```

a はこの関数の引数の個数である。この値は関数宣言の引数の解析をする際に数えておき、その時点で関数名を表す記号表の要素に保存しておけばよい。使う際には記号表を探索し、その要素から値を取り出す。

まとめると関数呼び出しのコードは概ね次のようになる。tmp->l はその関数のレベルを表す。

```
引数 1 のコード
引数 2 のコード
...
引数 n のコード
( CAL, tmp->l, lab )
```

呼ばれた関数側のコードは概ね次のようになる。

```
( LAB, 0, lab )
...
return 文の引数 (式) のコード
( RET, 0, a)
```

3.9 組み込み関数

PL/0 機械で今回用意されている組み込み関数はスタックの先頭要素を画面に表示する関数、改行を表示する関数、キーボードからの入力を得てスタックの先頭に載せる関数である。

まず、スタックの先頭要素を画面に表示する関数の呼び出しは次である。

```
( CSP, 0, 1 )
```

次に改行を表示する関数の呼び出しは次である。

```
( CSP, 0, 2 )
```

最後にキーボードからの入力を得てスタックの先頭に載せる関数の呼び出しは次である。

```
( CSP, 0, 0 )
```

最後に階乗計算をする PL/0' プログラム (図 11) とその PL/0 コード (図 12) を記す。

```
function f(n)
begin
  if n < 1 then return 1;
  return n * f(n - 1);
end;

var x;

begin
  read x;
  write f(x);
  writeln;
end.
```

図 11: 階乗計算のサンプルプログラム

```
( JMP,    0,    3 )
( LAB,    0,    1 )
( INT,    0,    3 )
( LOD,    0,   -1 )
( LIT,    0,    1 )
( OPR,    0,   10 )
( JPC,    0,    2 )
( LIT,    0,    1 )
( RET,    0,    1 )
( LAB,    0,    2 )
( LOD,    0,   -1 )
( LOD,    0,   -1 )
( LIT,    0,    1 )
( OPR,    0,    3 )
( CAL,    1,    1 )
( OPR,    0,    4 )
( RET,    0,    1 )
( OPR,    0,    0 )
( LAB,    0,    3 )
( INT,    0,    4 )
( CSP,    0,    0 )
( STO,    0,    3 )
( LOD,    0,    3 )
( CAL,    0,    1 )
( CSP,    0,    1 )
( CSP,    0,    2 )
( OPR,    0,    0 )
```

図 12: 図 11 に対する PL/0 コード

4 演習の解答例

演習 1

略

演習 2

関数 search_all の例を図 13 に示す。また、図 5 の 52 行目に対するアクションを図 14 に示す。

```
list *search_all(char* name){
  list *tmp;

  for(tmp=t;
      tmp != h;
      tmp = tmp->prev){
    if (strcmp(name, tmp->name)==0){
      return tmp;
    }
  }

  return NULL;
}
```

図 13: search_all のコード例

```
IDENTIFIER
{
  list* tmp;

  tmp = search_all($1);
  if (tmp == NULL){
    fprintf(stderr, "this identifier is"
                    " not declared!!\n");
  }
  $$ = tmp->val;
}
```

図 14: 図 5 の 52 行目に対するアクション

演習 3

解答例を図 15 に示す。

演習 4

1. 解答例を図 16 に示す。

```
A   : IDENTIFIER '=' E
{
  list* tmp;

  tmp = search_all($1);
  if (tmp == NULL){
    fprintf(stderr,
            "this identifier has not"
            " been declared!\n");
  }
  else {
    tmp->val = $3;
  }
  $$ = $3;
}
```

図 15: 図 5 の 38 行目に対するアクション

```
void addlist(char* name, int kind){
  list *tmp;

  tmp = (list*)malloc(sizeof(list));
  if (tmp == NULL){
    perror("memory allocation");
    exit(EXIT_FAILURE);
  }

  tmp->name = name;
  tmp->kind = kind;

  tmp->prev = t;
  t = tmp;
}
```

図 16: 関数 addlist の例

2. 解答例を図 17 に示す。
3. 解答例を図 18 に示す。
4. 解答例を図 19 に示す。

参考文献

[1] Niklaus Wirth. *Algorithms + Data Structure = Programs*. Pretice-Hall, 1976.
片山卓也訳:アルゴリズム＋データ構造＝プログラム, 日本コンピュータ協会, 1980.

[2] 佐々政孝. **プログラミング言語処理系**. 岩波書店, 1989.

```
 1 %{
 2 #include <stdio.h>
 3 #include <stdlib.h>
 4
 5 struct NODE {
 6   int nodecode;
 7   int val;
 8   struct NODE *lc, *rc;
 9 };
10
11 struct NODE *root;
12
13 struct NODE *makenode(struct NODE *l,
                struct NODE* r, int no, int val);
14 %}
15
16 %union {
17   int          val;
18   struct NODE* node;
19 }
20
21 %type  <node> E
22 %type  <node> T
23 %type  <node> F
24
25 %token <val> NUM
26 %token NL
27 %%
28 LIST    : E NL { root = $1; }
29         ;
30
31 E   : E '+' T { $$ = makenode($1,$3, '+', 0); }
32     | E '-' T { $$ = makenode($1,$3, '-', 0); }
33     | T { $$ = $1; }
34     ;
35
36 T   : T '*' F { $$ = makenode($1,$3, '*', 0); }
37     | T '/' F { $$ = makenode($1,$3, '/', 0); }
38     | F { $$ = $1; }
39     ;
40
41 F   : NUM { $$ = makenode(NULL, NULL, 0, $1); }
42     | '(' E ')' { $$ = $2; }
43     ;
44
45 %%
```

```
46
47 #include "lex.yy.c"
48
49 void traverse(struct NODE *n);
50
51 main(){
52   yyparse();
53
54   traverse(root);
55   printf("\n");
56 }
57
58 struct NODE *makenode(struct NODE *l,
                struct NODE* r, int no, int val){
59   struct NODE *tmp;
60
61   tmp = (struct NODE*)
           malloc(sizeof(struct NODE));
62   if (tmp == NULL){
63     perror("memory allocation error!");
64     exit(EXIT_FAILURE);
65   }
66
67   tmp->nodecode = no;
68   tmp->val = val;
69   tmp->lc = l;
70   tmp->rc = r;
71
72   return tmp;
73 }
74
75 void traverse(struct NODE *n){
76   if (n->lc != NULL){
77     traverse(n->lc);
78   }
79   if (n->rc != NULL){
80     traverse(n->rc);
81   }
82   if (n->nodecode == 0){
83     printf(" %d ", n->val);
84   }
85   else {
86     printf(" %c ", n->nodecode);
87   }
88 }
89
```

図 20: 第 3 回テキストの宿題 1 の解答例 (Yacc 記述)

```
%{
#include <stdio.h>
#include <stdlib.h>

typedef
struct LIST {
  char        *name;
  struct LIST *next;
} list;

list *h, *t, *tmp;

#define YYSTYPE char*

list* search(char*);
void addlist(char*);

%}

%token NUM
%token INT
%token IDENTIFIER
%token SEMI
%%

PROGRAM : decl_part use_part ;

decl_part : decl_list
        | /* empty */
        ;

decl_list : decl_list  decl
        | decl
        ;

decl    : INT IDENTIFIER SEMI {
            if (search($2) == NULL){
              addlist($2);
            }
            else {
              fprintf(stderr, "this identifier"
              " has been already declared!\n");
            }
          }
        ;

use_part : use_list
        | /* empty */
        ;

use_list : use_list LIST
        | LIST
        ;

LIST    : E SEMI
        ;

E       : E '+' T
        | E '-' T
        | T
        ;

T       : T '*' F
        | T '/' F
        | F
        ;

F       : NUM
        | IDENTIFIER {
            if (search($1) == NULL){
              fprintf(stderr, "this identifier "
                      "is not declared!!\n");
            }
          }
        | '(' E ')'
        ;

%%

#include "lex.yy.c"

main(){
  /* initialize the list */
  h = (list*)malloc(sizeof(list));
  if (h == NULL){
    perror("memory allocation");
    exit(EXIT_FAILURE);
  }

  t = h;
  h->next = NULL;
  h->name = NULL;

  yyparse();
}

void addlist(char* name){
  list *tmp;

  tmp = (list*)malloc(sizeof(list));
  if (tmp == NULL){
    perror("memory allocation");
    exit(EXIT_FAILURE);
  }

  tmp->name = name;

  t->next = tmp;
  tmp->next = NULL;
  t = tmp;
}

list *search(char* name){
  list *tmp;

  for(tmp=h->next;
      tmp != NULL;
      tmp = tmp->next){
    if (strcmp(name, tmp->name)==0){
      break;
    }
  }
  return tmp;
}
```

図 21: 第 3 回テキストの宿題 3 の解答例 (Yacc 記述)

```
list *search_block(char* name){
  list *tmp;

  for(tmp=t;
      tmp != h && tmp->kind != BLOCK;
      tmp = tmp->prev){
    if (strcmp(name, tmp->name)==0){
      return tmp;
    }
  }

  return NULL;
}
```

図 17: 関数 search_block の例

```
decl : INT IDENTIFIER SEMI
{
  if (search_block($2) == NULL){
    addlist($2, ID);
  }
  else {
    fprintf(stderr,
            "this identifier has"
            " been already declared!\n");
  }
}
```

図 18: 図 5 の 22 行目に対するアクション

```
void delete_block(){
  list *tmp;

  for(tmp=t;
      tmp != h && tmp->kind != BLOCK;
      tmp = tmp->prev){
    free(tmp->name);
    free(tmp);
  }
  free(tmp->name);
  free(tmp);
  t = tmp->prev;
}
```

図 19: 関数 delete_block の例