コンパイラ構成論 62105213 勝又圭

概要

本第二回コンパイラ構成論課題において以下の項目を実施した。

- フロントエンド
  - 「//」から行末までをコメントにできるようにしなさい.
  - 構文エラーの際に，行番号と直後の字句を印字するようにしなさい.
  - 生成規則に，エラー回復を考慮してerrorトークンを挿入しなさい.
    - ただしフロントエンド問題3については完璧な実装ではなく、構文エラーの際のエラー行数表示と共存させていないため別実装となる。
- バックエンド
  - 「式1 % 式2」 とすることによって，「式1」 を 「式2」 で 割っ た余りを計算する演算子「%」を付加しなさい.
  - 「int 変数 = 式」とすることによって，変数を宣言するとともに， 「式」の値を初期値として設定できるようにしなさい.
  - 「式1 ^ 式2」 とすることによって，「式1」 を 「式2」 乗する 演算子「^」を付加しなさい.
  - 「変数 ++」 とすることによって，「変数」 の値を 1 増加させ，増加 前の値を返す演算子「++」を付加せよ.
  - 「変数 += 式」 とすることによって，「変数」 の値に 「式」 の 値 を加算し，その結果で「変数」を更新する文を付加しなさい.
  - 「do 文 while (条件)」 とすることによって，「条件」 が満たさ れている間 「文」 を繰り返し実行するような文を付加しなさい（while 文と違い，「文」が少なくとも1回は実行されることに注意）.
  - 「for (変数 = 式1 .. 式2) 文」 の構文で，for文を実現しなさい． ただし，このfor文は次のようなステップで実行するものとする.
  - ここで示したSimpleコンパイラは，関数内の「return e」の「e」 の型と，関数の返戻値の型の一致を調べていない．この型検査を 付加しなさい.

また、これらの項目についてはあくまで自分の確認コードを実行することで確認したものであり、最適/完璧な実装とは限らない。

コード構成

本課題はGithub上で管理を行いながら実施をしたため、github上のcommitを見ていただければその変遷は確認できる。 https://github.com/agate106k/CC8

当初本課題のfrontendとbackendを別々の課題と考えていたため、frontend、client(エラー回復用)、backend(server)ディレクトリに分けて実装を行っている。frontendにてcommentout, エラー行出力の実装、serverはコメントアウトを含めたbackenddのコード、そしてclientにエラー回復単独(あくまでエラー行数は出せていないもの)の実装がされている。

実行結果

backend各種 (return除く)、フロントエンドコメントアウト

実行したsimpleファイルは以下である。他の人にもテストファイルとして共有したため、もしかしたら同一テストを行なっている人がいるかもしれないが、backend項目のreturnを除く全てとコメントアウトを含むコードである。各コメントに出力の想定を記載している。

```
{
    int x, a, b, c;
    // int 変数 = 数値; のテスト
    int rlt = 1;
    a = 5;
    b = 3;
    c = 2;

    // '%' 演算子のテスト
    rlt = a % b;
    iprint(rlt); // 期待される出力: 2
    sprint("\n");
    // '^' 演算子のテスト
    rlt = a ^ c;
    iprint(rlt); // 期待される出力: 25
    sprint("\n");
    // '++' 演算子のテスト
    iprint(a); // 期待される出力: 5
    sprint("\n");
    rlt = a++;
    iprint(rlt); // 期待される出力: 5
    sprint("\n");
    iprint(a);   // 期待される出力: 6
    sprint("\n");
    // '+=' 文のテスト
    rlt += a;
    iprint(rlt); // 期待される出力: 11
    sprint("\n");
    // 'do ... while' 文のテスト
    iprint(a); // 期待される出力: 6
    sprint("\n");
    rlt = 1;
    do {
        rlt += a;
    } while (rlt < 10);
    iprint(rlt); // 期待される出力: 13
    sprint("\n");
    // 'for' 文のテスト
    rlt = 1;
    for (x = 0 .. 5)
        rlt = rlt * 3;
    iprint(rlt); // 期待される出力: 243
    sprint("\n");
}
```

```
[ub422066@remote09 server]$ make
make: 'simc' is up to date.
[ub422066@remote09 server]$ make print_ast
make: Circular print_ast <- print_ast dependency dropped.
make: 'print_ast' is up to date.
[ub422066@remote09 server]$ ./simc back-test.spl
[ub422066@remote09 server]$ ./a.out
2
25
5
5
6
11
6
13
243
[ub422066@remote09 server]$ ./print_ast back-test.spl

Block([VarDec(IntTyp,"x"); VarDec(IntTyp,"a"); VarDec(IntTyp,"b");
VarDec(IntTyp,"c"); InitVarDec(IntTyp,"rlt",IntExp(1))],[Assign(Var
"a",IntExp(5)); Assign(Var "b",IntExp(3)); Assign(Var "c",IntExp(2));
Assign(Var "rlt",CallFunc("%",[VarExp(Var "a"); VarExp(Var "b")]));
CallProc("iprint",[VarExp(Var "rlt")]); CallProc("sprint",[StrExp("\n")]);
Assign(Var "rlt",CallFunc("^",[VarExp(Var "a"); VarExp(Var "c")]));
CallProc("iprint",[VarExp(Var "rlt")]); CallProc("sprint",[StrExp("\n")]);
CallProc("iprint",[VarExp(Var "a")]); CallProc("sprint",[StrExp("\n")]);
Assign(Var "rlt",IncExp(Var "a")); CallProc("iprint",[VarExp(Var "rlt")]);
CallProc("sprint",[StrExp("\n")]); CallProc("iprint",[VarExp(Var "a")]);
CallProc("sprint",[StrExp("\n")]); Assign(Var "rlt",CallFunc("+",
[VarExp(Var "rlt"); VarExp(Var "a")])); CallProc("iprint",[VarExp(Var
"rlt")]); CallProc("sprint",[StrExp("\n")]); CallProc("iprint",[VarExp(Var
"a")]); CallProc("sprint",[StrExp("\n")]); Assign(Var "rlt",IntExp(1));
DoWhile(Block([],[Assign(Var "rlt",CallFunc("+",[VarExp(Var "rlt");
VarExp(Var "a")]))]),CallFunc("<",[VarExp(Var "rlt"); IntExp(10)]));
CallProc("iprint",[VarExp(Var "rlt")]); CallProc("sprint",[StrExp("\n")]);
Assign(Var "rlt",IntExp(1)); For("x",IntExp(0),IntExp(5),Assign(Var
"rlt",CallFunc("*",[VarExp(Var "rlt"); IntExp(3)]))); CallProc("iprint",
[VarExp(Var "rlt")]); CallProc("sprint",[StrExp("\n")])])
```

以上の通り自分の想定解が異なっていなければ本コードにより各種実装が行われていることが確認できる。 また、既存で用意されていたsort.splの実行結果も以下に示す。

```
[ub422066@remote09 server]$ ./print_ast sort.spl

Block([VarDec(ArrayTyp (10,IntTyp),"a"); VarDec(IntTyp,"size");
FuncDec("init",[],VoidTyp,Block([VarDec(IntTyp,"i")],[Assign(Var
"i",IntExp(0)); While(CallFunc("<",[VarExp(Var "i"); VarExp(Var
"size")]),Block([],[Assign(IndexedVar (Var "a",VarExp(Var
"i")),CallFunc("-",[VarExp(Var "size"); VarExp(Var "i")])); Assign(Var
```

```
"i",CallFunc("+",[VarExp(Var "i"); IntExp(1)]))]))])); FuncDec("print",
[],VoidTyp,Block([VarDec(IntTyp,"i")],[Assign(Var "i",IntExp(0));
While(CallFunc("<",[VarExp(Var "i"); VarExp(Var "size")]),Block([],
[CallProc("iprint",[VarExp(IndexedVar (Var "a",VarExp(Var "i")))]);
CallProc("sprint",[StrExp(" ")]); Assign(Var "i",CallFunc("+",[VarExp(Var
"i"); IntExp(1)]))]))); CallProc("sprint",[StrExp("\n")])]));
FuncDec("sort",[(IntTyp,"i")],VoidTyp,Block([FuncDec("min",
[(IntTyp,"j")],VoidTyp,Block([FuncDec("swap",[(IntTyp,"i");
(IntTyp,"j")],VoidTyp,Block([VarDec(IntTyp,"tmp")],[Assign(Var
"tmp",VarExp(IndexedVar (Var "a",VarExp(Var "i")))); Assign(IndexedVar
(Var "a",VarExp(Var "i")),VarExp(IndexedVar (Var "a",VarExp(Var "j"))));
Assign(IndexedVar (Var "a",VarExp(Var "j")),VarExp(Var "tmp"))]))],
[If(CallFunc("<",[VarExp(Var "j"); VarExp(Var "size")]),Block([],
[If(CallFunc("<",[VarExp(IndexedVar (Var "a",VarExp(Var "j")));
VarExp(IndexedVar (Var "a",VarExp(Var "i")))]),CallProc("swap",[VarExp(Var
"i"); VarExp(Var "j")]),None); CallProc("min",[CallFunc("+",[VarExp(Var
"j"); IntExp(1)])])]),None)]))],[If(CallFunc("<",[VarExp(Var "i");
VarExp(Var "size")]),Block([],[CallProc("min",[CallFunc("+",[VarExp(Var
"i"); IntExp(1)])]); CallProc("sort",[CallFunc("+",[VarExp(Var "i");
IntExp(1)])])]),None)]))],[Assign(Var "size",IntExp(10)); CallProc("new",
[VarExp(Var "a")]); CallProc("init",[]); CallProc("sprint",[StrExp("before
sorting\n")]); CallProc("print",[]); CallProc("sort",[IntExp(0)]);
CallProc("sprint",[StrExp("after sorting\n")]); CallProc("print",[])])
[ub422066@remote09 server]$
[ub422066@remote09 server]$ ./simc sort.spl
[ub422066@remote09 server]$ ./a.out
before sorting
10 9 8 7 6 5 4 3 2 1
after sorting
1 2 3 4 5 6 7 8 9 10
```

以上のように正常実行できていることが確認できる。

return 型検査

return型検査に関する実装チェックを以下で行う。本検査にあたり正常実行(void, int)、int異常実行(return抜け)、void異常実行(returnあり)の3パターンを以下に示す。特に関数名に意味はない。

back-return-success.spl

```
{
        int i, k, result, total;
        void sumvoid() {
                total = 300;
        }
        int sum() {
                int total;
                total = 100;
                return total;
        }
        result = 100;
```

```
        result = sum();
        iprint(result); // 期待される出力: 100
        sprint("\n");
        sumvoid();
        iprint(total); // 期待される出力: 300
        sprint("\n");
}
```

実行結果

```
[ub422066@remote09 server]$ ./simc back-return-success.spl
[ub422066@remote09 server]$ ./a.out
100
300
```

back-return-int-fail.spl

```
{
        int i, k, result, total;
        int sum() {
                int total;
                total = 100;
        }
        result = 100;
        result = sum();
        iprint(result);
        sprint("\n");
}
```

実行結果

```
[ub422066@remote09 server]$ ./simc back-return-int-fail.spl
function must have a return statement
```

back-return-void-fail.spl

```
{
        int i, k, result, total;
        void sumvoid() {
                total = 300;
                return total;
        }
        sumvoid();
        iprint(total);
```

```
        sprint("\n");
}
```

実行結果

```
[ub422066@remote09 server]$ ./simc back-return-void-fail.spl
return expression type does not match function return type
```

以上のように関数自体の型と異なるreturn文を含む/含まない場合、エラーが出力されることが確認できた。

構文エラーチェック

構文エラーチェックの行数表示の実行結果を以下に示す。

実行ファイル front-errror-check.spl

```
{
    int a, b, m, n, r;
        sprint ("You must give 2 integers.\n");
        sprint ("First integer: ");
        scan  (a);
        sprint ("Second integer: ");
        scan (b);
        m = a; n = b;
        r = m - (m / n) * n;
        m = n;
        n = r;
        fafa
        testksks
        sprint ("Answer = ");
        iprint (m) ;
        sprint ("\n");
}
```

```
[ub422066@remote09 frontend]$ make
ocamlyacc parser.mly
1 shift/reduce conflict.
ocamllex lexer.mll
76 states, 3428 transitions, table size 14168 bytes
ocamlc -c ast.ml
ocamlc -c parser.mli
ocamlc -c ErrorFlag.ml
ocamlc -c lexer.ml
ocamlc -c parser.ml
ocamlc -c print_ast.ml
ocamlc -o print_ast unix.cma ast.cmo ErrorFlag.cmo lexer.cmo parser.cmo
print_ast.cmo
```

```
[ub422066@remote09 frontend]$ ./print_ast test.spl

Syntax error: Parsing Failed at line 13, column 8: testksks
```

以上のように実行するとsyntax errorとなるfafaの部分の次、testksksがエラー部分として表示される。ただし、エラー回復と同居していないため二つ目のエラーは出ない。

エラー回復

エラー回復を単独で実行したものの実行例を示す。

```
{
    int a, b, m, n, r;

        sprint ("You must give 2 integers.\n");
        sprint ("First integer: ");
        fafa;
        fafa;
        sprint ("Second integer: ");
        scan (b);
        m = a; n = b;
        r = m - (m / n) * n;
        m = n;

        n = r;
    while (r > 0) {
                r = m - (m / n) * n;

                m = n;
                n = r;
    }
        sprint ("Answer = ");
        iprint (m) ;
        testksks
        sprint ("\n");
}
```

```
[ub422066@remote09 client]$ ./print_ast sample.spl
Syntax error
Syntax error
Syntax error
Block([VarDec(IntTyp,"a"); VarDec(IntTyp,"b"); VarDec(IntTyp,"m");
VarDec(IntTyp,"n"); VarDec(IntTyp,"r")],[CallProc("sprint",[StrExp("You
must give 2 integers.\n")]); CallProc("sprint",[StrExp("First integer:
")]); NilStmt; NilStmt; CallProc("sprint",[StrExp("Second integer: ")]);
CallProc("scan",[VarExp(Var "b")]); Assign(Var "m",VarExp(Var "a"));
Assign(Var "n",VarExp(Var "b")); Assign(Var "r",CallFunc("-",[VarExp(Var
"m"); CallFunc("*",[CallFunc("/",[VarExp(Var "m"); VarExp(Var "n")]);
```

```
VarExp(Var "n")])])); Assign(Var "m",VarExp(Var "n")); Assign(Var
"n",VarExp(Var "r")); While(CallFunc(">",[VarExp(Var "r");
IntExp(0)]),Block([],[Assign(Var "r",CallFunc("-",[VarExp(Var "m");
CallFunc("*",[CallFunc("/",[VarExp(Var "m"); VarExp(Var "n")]); VarExp(Var
"n")])])); Assign(Var "m",VarExp(Var "n")); Assign(Var "n",VarExp(Var
"r"))])); CallProc("sprint",[StrExp("Answer = ")]); CallProc("iprint",
[VarExp(Var "m")]); NilStmt])
```

以上のようにsyntaxerrorが実際の無駄なfafa, testksksの分だけ出ていることがわかるほか、実行してみると、ErrorFlagの効果によりエラー判定(いわゆる実行結果がエラーで赤くなる)がエラー回復をしてもなお起こるようになっている。

各種実装

「//」から行末までをコメントにできるようにしなさい

```
| "//" [^ '\n']*          { lexer lexbuf }
```

この行は、`//`に続く任意の文字列（改行文字`\n`を除く）をコメントとして認識し、無視するように指示している。ここで、`[^ '\n']*`は改行文字以外の任意の文字が0回以上続くようにしている。`{ lexer lexbuf }`は、コメントを無視した後に、再び字句解析を続けるような形にしている。

当初は、以下のように`comment`関数を定義して、コメント内の文字を読み飛ばし、改行文字で字句解析に戻るということにしていた。

```
| "//"                    { comment lexbuf }
and comment = parse
| '\n'                    { lexer lexbuf }
| _                       { comment lexbuf }
```

構文エラーの際に，行番号と直後の字句を印字するようにしなさい.

エラーの行数を出力されるようにprint_astとlexerに追加を行った。エラーの種類によりその処理を変えるようにしたが特に本質的な違いはない。lexer.mll内で\nの際にひたすらlineを記録していることにより正確な位置でのエラーがハッシエしている部分を記録できるというのが肝であると言えるだろう。 実際のコードをfrontend内に含まれている。

```
let main () =
  (* The open of a file *)
  let cin = if Array.length Sys.argv > 1 then open_in Sys.argv.(1) else
stdin in
  let lexbuf = Lexing.from_channel cin in
  (* The start of the entire program *)
```

```
  try print_string (ast_stmt (Parser.prog Lexer.lexer lexbuf)) with
  | Parsing.Parse_error ->
      print_string
        (let l = (Lexing.lexeme_start_p lexbuf).pos_lnum in
         let c =
           (Lexing.lexeme_start_p lexbuf).pos_cnum
           - (Lexing.lexeme_start_p lexbuf).pos_bol
         in

         let m = Lexing.lexeme lexbuf in
         sprintf "Syntax error: Parsing Failed at line %d, column %d:
%s\n" l c
           m)
  | Lexer.No_such_symbol ->
      print_string
        (let l = (Lexing.lexeme_start_p lexbuf).pos_lnum in
         let c =
           (Lexing.lexeme_start_p lexbuf).pos_cnum
           - (Lexing.lexeme_start_p lexbuf).pos_bol
         in
         let m = Lexing.lexeme lexbuf in
         sprintf "Syntax error: Unexpected Token at line %d, column %d:
%s\n" l c
           m)
;;
```

```
| ['\n']                 { incr line; Lexing.new_line lexbuf; lexer
lexbuf }
```

生成規則に，エラー回復を考慮してerrorトークンを挿入しなさい.

client/print_ast.ml、client/ErrorFlag.ml、client/lexer.mllを変更した。

エラー回復のために、lexer.mllファイルにおいて、予期しない文字が入力された場合にERRORトークンを生成し、ErrorFlag.set_error ();を呼び出してエラーフラグをセットした。これにより、エラーが発生したことが記録され、単にエラーを飛ばすだけではなく、エラーが発生したことを後々参照できるようにした。

```
| _ as c { Printf.eprintf "Unexpected character %c\n" c;
ErrorFlag.set_error (); ERROR }
```

```
let error_occurred = ref false

let set_error () = error_occurred := true

let check_error () = !error_occurred
```

エラーが発生したかどうかを追跡するためのフラグerror_occurredを定義し、set_error関数でエラーフラグをセットし、check_error関数でエラーの有無を確認できるようにしている。 これに合うようにMakefileも書き換えている。

```
let _ =
        main ();
        if ErrorFlag.check_error () then exit 1
```

実行後に、ErrorFlag.check_error関数を呼び出してエラーフラグを確認し、エラーがあった場合は終了ステータス1でプログラムを終了させる形とした。これにより、エラーが発生した場合に正常実行ではなくエラー終了させるようにしている。

「式1 % 式2」 とすることによって, 「式1」 を 「式2」 で 割っ た余りを計算する演算子「%」を付加しなさい.「式1 ^ 式2」 とすることによって, 「式1」 を 「式2」 乗する 演算子「^」を付加しなさい. 「変数 += 式」 とすることによって, 「変数」 の値に 「式」 の 値 を加算し, その結果で「変数」を更新する文を付加しなさい

面倒なので %と ^と+=は同時に示す。 parser.mlyファイルに新しいトークンPERCENT POW PLUS_ASSIGNをそれぞれ定義し、式の文法規則にexpr PERCENT expr expr POW exprのパターンを追加した。パターンがマッチした場合、CallFunc構造体を使って%関数呼び出しする。 PLUS_ASSIGNについては既存実装であるASSIGN と+を使えばできるためそれを書いた

```
%token PERCENT POW PLUS_ASSIGN

        | expr PERCENT expr { CallFunc ("%", [$1; $3]) }  /* %追加 */
        | expr POW expr { CallFunc ("^", [$1; $3]) }
        | ID PLUS_ASSIGN expr SEMI  { Assign (Var $1, CallFunc ("+",
[VarExp (Var $1); $3])) }
```

```
| "+="                    { PLUS_ASSIGN }
| '%'                     { PERCENT }
| '^'                     { POW }
```

semantではどちらも、両方の引数が整数型であることを確認。

```
          | CallFunc ("%", [left; right]) ->
              (check_int (type_exp left env); check_int(type_exp right
env); INT)
          | CallFunc ("^", [base; exponent]) ->
              (check_int (type_exp base env); check_int(type_exp exponent
env); INT)
```

emitter.mlファイルでは、%では割り算の余りを計算するために、idivq命令を使用し、その結果をスタックに渡してあげ、累乗計算のためのアセンブリコードを生成します。ループを使用して累乗を計算し、その結果をpushする形にした。

```
                  | CallFunc ("%", [left; right]) ->
                                      trans_exp left nest env
                                      ^ trans_exp right nest env
                                      ^ "\tpopq %rbx\n"
                                      ^ "\tpopq %rax\n"
                                      ^ "\tcqto\n"
                                      ^ "\tidivq %rbx\n"
                                      ^ "\tmovq %rdx, %rax\n"
                                      ^ "\tpushq %rax\n"
                | CallFunc ("^", [base; exp]) ->
                                      let base_code = trans_exp base
nest env in
                                      let exp_code = trans_exp exp nest
env in
                                      let loop_label = incLabel() in
                                      let end_label = incLabel() in
                                      base_code
                                      ^ exp_code
                                      ^ "\tpopq %rbx\n"  (* 指数を %rbx に
格納 *)
                                      ^ "\tpopq %rax\n"  (* 基数を %rax に
格納 *)
                                      ^ "\tmovq $1, %rcx\n"  (* 結果の初期
値を 1 に設定 *)
                                      ^ sprintf "L%d:\n" loop_label
                                      ^ "\ttestq %rbx, %rbx\n"  (* 指数が
0 かどうかテスト *)
                                      ^ sprintf "\tje L%d\n" end_label
(* 0 なら終了ラベルへジャンプ *)
                                      ^ "\timulq %rax, %rcx\n"  (* 結果に
基数を掛ける *)
                                      ^ "\tdecq %rbx\n"  (* 指数をデクリメ
ント *)
                                      ^ sprintf "\tjmp L%d\n" loop_label
(* ループラベルへジャンプ *)
                                      ^ sprintf "L%d:\n" end_label
                                      ^ "\tpushq %rcx\n"  (* 結果をスタック
にプッシュ *)
```

「変数 ++」 とすることによって，「変数」 の値を 1 増加させ，増加 前の値を返す演算子「++」を付加せよ

ast.mlでは他の演算子とは異なりある種式の型としてIncExpを実装した。

```
| IncExp of var
```

parserでは変数名の後にINCREMENTトークンが続く場合にIncExp式になるようにした。

```
| ID INCREMENT { IncExp (Var $1) }
```

semant.mlでは、この式の型チェックを行い、IncExp式の変数が整数型であることを確認する。

```
| IncExp v -> (check_int (type_exp (VarExp v) env); INT)
```

各手順はコメントで書いているが、変数の現在の値を一時レジスタに保存し、その値に1を加算して元の位置に戻す。元の値（増加前の値）をスタックにプッシュして。これにより、増加前の値を返すという++演算子の動作をできるようにしている。

```
                  | IncExp v ->
                          trans_var v nest env
                          ^ "\tmovq (%rax), %rbx\n"  (* 値を一時的なレジスタ
に保存 *)
                          ^ "\taddq $1, %rbx\n"  (* 値を増加させる *)
                          ^ "\tmovq %rbx, (%rax)\n"  (* 値を元の位置に保存 *)
                          ^ "\tsubq $1, %rbx\n"  (* 元の値を復元 *)
                          ^ "\tpushq %rbx\n"  (* 元の値をスタックにプッシュ *)
```

「int 変数 = 式」とすることによって，変数を宣言するとともに，「式」の値を初期値として設定できるようにしなさい.

astではint 宣言代入用にInitVarDecを作り登録

```
| InitVarDec of typ * id * exp
```

parserでもいつも通りのように型、変数名、そして初期値を表す式をマッチさせ、InitVarDecノードを生成。

```
  | InitVarDec (t, s, e) ->
      let var_code = trans_var (Var s) nest env in
      let exp_code = trans_exp e nest env in
      exp_code ^ var_code ^ "\tpopq (%rax)\n"
```

新しい変数宣言と初期化の形式に対して、型チェックと再宣言のチェックを行い、変数の型と初期値の式の型が一致することを確認し、変数が既に宣言されていないことを確認している。

```
    | InitVarDec (_,s,_)::rest -> if List.mem s vl then raise (SymErr s)
                                  else check_redecl rest tl (s::vl)
```

```
    | InitVarDec (t, s, e) ->
        let ty = create_ty t tenv in
        let exp_ty = type_exp e env in
        if actual_ty ty != actual_ty exp_ty then
        raise (TypeErr "type mismatch in variable initialization");
        (tenv, update s (VarEntry {ty=ty; offset=addr-8; level=nest})
env, addr-8)
```

期値を表す式を評価し、その結果を変数の位置に格納する。 Blockの処理では、変数宣言とステートメントのリストを処理し、ここが一番難しかったが、スタックフレームを作ってあげるようにした。 emitter.ml

```
  | InitVarDec (t, s, e) ->
      let var_code = trans_var (Var s) nest env in
      let exp_code = trans_exp e nest env in
      exp_code ^ var_code ^ "\tpopq (%rax)\n"

        | Block (dl, sl) ->
                      let (tenv', env', addr') = type_decs dl nest tenv
env in
                      let decs_code = List.fold_left (fun code dec -> code
^ trans_dec dec nest tenv' env') "" dl in
                      let ex_frame = sprintf "\tsubq $%d, %%rsp\n" ((-
addr' + 16) / 16 * 16) in
                      let stmts_code = List.fold_left (fun code stmt ->
code ^ trans_stmt stmt nest tenv' env') "" sl in
                      decs_code ^ ex_frame ^ stmts_code
```

do while, for

こちらも似ているため同時に示す。

DowhileとForの形になるようにastを設定

```
        | DoWhile of stmt * exp
        | While of exp * stmt
        | For of id * exp * exp * stmt
```

lexerにも同様に登録

```
| "for"                  { FOR }
| ".."                   { DOTDOT}
| "do"                   { DO }
```

parserにもDo whileにはDO、FOR用のFORとDOTDOTをそれぞれtokenとして登録し

```
%token FOR DOTDOT DO
     | DO stmt WHILE LP cond RP SEMI { DoWhile ($2, $5) }
     | FOR LP ID ASSIGN expr DOTDOT expr RP stmt { For ($3, $5, $7, $9) }
```

semant.mlでは、whileでもあんまり詳しくチェックされていなかったのでDoWhileとWhileでは条件式がブール型であることを、Forでは初期式、終了式が整数型であることを確認している。

```
          | DoWhile (s, e) ->
               type_cond e env
          | While (e,_) -> type_cond e env
          | For (v, e1, e2, s) ->
          if (type_exp (VarExp (Var v)) env) != INT || (type_exp e1 env)
!= INT || (type_exp e2 env) != INT then
          raise (TypeErr "type error in for statement")
          else type_stmt s env
```

最後に、emitter.mlでは、DoWhileではループの本体を実行した後に条件を評価し、Whileでは条件を評価してからループの本体を実行、Forでは初期式を評価した後、条件を評価してループの本体を実行し、各反復の後に更新式を実行するようにした。

```
                  (* do-while文のコード *)
                  | DoWhile (s,e) -> let (condCode, l1) = trans_cond e
nest env in
                                     let l2 = incLabel() in
                                         sprintf "L%d:\n" l2
                                       ^ trans_stmt s nest tenv env
                                       ^ condCode
                                       ^ sprintf "\tjmp L%d\n" l2
                                       ^ sprintf "L%d:\n" l1
                  (* while文のコード *)
                  | While (e,s) -> let (condCode, l1) = trans_cond e nest
env in
                                    let l2 = incLabel() in
                                        sprintf "L%d:\n" l2
                                      ^ condCode
                                      ^ trans_stmt s nest tenv env
                                      ^ sprintf "\tjmp L%d\n" l2
                                      ^ sprintf "L%d:\n" l1
                  | For (v, e1, e2, s) ->
                                      trans_stmt (Assign(Var v, e1)) nest
tenv env
                                      ^ let (condCode, l1) = trans_cond
(CallFunc ("<", [(VarExp (Var v)); e2])) nest env in
                                      let l2 = incLabel() in sprintf
"L%d:\n" l2  (* コロンを追加 *)
                                      ^ condCode
```

```
                                        ^ trans_stmt s nest tenv env
                                        ^ trans_exp (IncExp(Var v)) nest
env
                                        ^ sprintf "\tjmp L%d\n" l2
                                        ^ sprintf "L%d:\n" l1
```

return 型検査

```
let rec type_dec ast (nest, addr) tenv env =
   match ast with
      (* 関数定義の処理 *)
      FuncDec (s, l, rlt, Block (dl,sl)) ->
      (* 関数名の記号表への登録 *)
      check_redecl ((List.map (fun (t,s) -> VarDec (t,s)) l) @ dl) [] [];
      let formals = List.map (fun (typ,_) -> create_ty typ tenv) l in
      let result = create_ty rlt tenv in
      let env' = update s (FunEntry {formals=formals; result=result;
level=nest+1}) env in
      (* 関数のシグネチャを環境に追加 *)
      enter_function s result formals;
      (* 関数本体のステートメントリストに対する型チェックを追加 *)
      let has_return = List.exists (function CallProc ("return", _) ->
true | _ -> false) sl in
      if actual_ty result != UNIT && not has_return then
          raise (TypeErr "function must have a return statement");
      (tenv, env', addr)

 FuncDec (s, l, rlt, Block (dl,sl)) ->
      (* 関数名の記号表への登録 *)
      check_redecl ((List.map (fun (t,s) -> VarDec (t,s)) l) @ dl) [] [];
      let formals = List.map (fun (typ,_) -> create_ty typ tenv) l in
      let result = create_ty rlt tenv in
      let env' = update s (FunEntry {formals=formals; result=result;
level=nest+1}) env in
      (* 関数のシグネチャを環境に追加 *)
      enter_function s result formals;
      (* 関数本体のステートメントリストに対する型チェックを追加 *)
      let has_return = List.exists (function CallProc ("return", _) ->
true | _ -> false) sl in
      if actual_ty result != UNIT && not has_return then
          raise (TypeErr "function must have a return statement");
      (tenv, env', addr)


          | CallProc ("return", []) ->
                     let current_fun_type = current_function_return_type
() in
                     if current_fun_type != UNIT then
                          raise (TypeErr "non-void function must return a
value")
```

```
            (* return 文の型チェックを行う *)
            | CallProc ("return", args) ->
            let current_fun_type = current_function_return_type () in
            (match args, current_fun_type with
                 | [], UNIT -> ()  (* void関数のreturn文のチェック *)
                 | [arg], _ ->  (* 値を返すreturn文のチェック *)
                 let arg_type = type_exp arg env in
                 if actual_ty arg_type != actual_ty current_fun_type then
                      raise (TypeErr "return expression type does not match
  function return type")
                 | [], _ ->  (* non-void関数が値を返さない場合のチェック *)
                 raise (TypeErr "non-void function must return a value")
                 | _ -> raise (Err "invalid return statement"))
```

正直色々やりすぎてぐちゃぐちゃになっている感は否めないが、関数定義時に指定された返り値の型と、関数内のreturn文で返される値の型が一致するかをチェックするようにsemantに追記した。

FuncDecでは、関数の引数と返り値の型を環境に登録し、関数本体のステートメントリストに対する型チェックを行う。has_return変数を使って、関数内にreturn文が存在するかを確認し、返り値の型がUNIT（返り値がない関数）でない場合、エラーを出す。 加えてreturn文の型チェックを行い、CallProc ("return", args)の処理で、現在の関数の返り値の型を取得し、return文で返される値の型と比較。return文が値を返さない場合（argsが空のリスト）、現在の関数の返り値の型がUNITであることを確認した。一致しない場合、TypeErrを発生させる形の実装とした。

まとめ・感想

以上のように詳細な実装は最適解ではないかもしれないが基本的な課題については全て実行範囲内において正常実行できる形で実行を行うことができた。 正直コンパイラというものをあまり意識してこなかったものの、今回の課題を経てどのように型解析をおこなってコンパイルを実施しているのかということの一部は理解できたような気がした。それと同時にいつもGoやTypescriptなどの言語を中心に使用しているがあの型検査機能等のありがたみを再認識できた。