

# 計算機科学実験及び演習 3

## ソフトウェア実験中間レポート 2

工学部情報学科 3 回生 1029255242

勝見久央

作成日: 平成 27 年 7 月 7 日

### 1 課題 8

この課題では

- 単項演算  $-x$
- else 節のない if 文
- for 文 `for(e1, e2, e3) s`
- 配列参照式 `e1[e2]`

のシンタックスシュガーを含めたプログラムの抽象構文木への変換処理を実装した. シンタックスシュガーの実装においてはそれぞれを

- $0-x$
- `if(condition){compound-statement}else{}`
- `e1; while(e2){s e3;}`
- $*(e1+e2)$  (結果として発生する式  $*(e)$  の形は  $e$  に変換)

と変換した. さらに、組み込み関数 `print` のプロトタイプ宣言を抽象構文生成の段階でプログラムに付加するようにした.

#### 1.1 課題 8 の回答

コードは次のようになった.

リスト 1: シンタックスシュガーを含む抽象構文木への変換処理

```
1 #lang racket
2 (require parser-tools/lex
3           (prefix-in : parser-tools/lex-sre)
4           parser-tools/yacc
5           (prefix-in stx: "mysyntax.rkt")
6           (prefix-in k07u: "kadai07upgrade.rkt")
7           )
8 (provide (all-defined-out))
```

```

9
10 (define-empty-tokens tokens-without-value
11   (+ * & - / =
12     l_small_paren r_small_paren ;()
13     l_mid_paren r_mid_paren ;[]
14     l_big_paren r_big_paren ;{})
15   int void
16   if while for else
17   or and equal not;||, &&, ==, !=
18   less and_less;<, <=
19   more and_more;>, >=
20   return
21   semicolon comma
22   EOF))
23
24 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
25 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
26 (define-tokens tokens-with-value
27   (NUM VAR))
28
29 (define-lex-trans uinteger
30   (syntax-rules () ((_ d) (:+ d))))
31
32 (define-lex-abbrevs
33   (digit (char-range "0" "9"))
34   (number (uinteger digit))
35   (identifier-char (:or (char-range "a" "z")
36                          (char-range "A" "Z")
37                          "-"))
38   (identifier (:: identifier-char
39                  (:* (:or identifier-char
40                          digit)))))
41
42 (define sub-program-lexer
43   (lexer-src-pos
44     ("(" (token-l_small_paren))
45     (")" (token-r_small_paren))
46     ("[" (token-l_mid_paren))
47     ("]" (token-r_mid_paren))
48     ("{" (token-l_big_paren))
49     ("}" (token-r_big_paren))
50     ("int" (token-int))
51     ("void" (token-void))
52     ("if" (token-if))
53     ("while" (token-while))
54     ("for" (token-for))
55     ("else" (token-else))
56     ("||" (token-or))
57     ("&&" (token-and))
58     ("==" (token-equal))
59     ("!=" (token-not))
60     ("<" (token-less))
61     ("<=" (token-and_less))
62     (">" (token-more))
63     (">=" (token-and_more))
64     ("return" (token-return))
65     (";" (token-semicolon))
66     (", " (token-comma))
67     ("+" (token-+))
68     ("*" (token-*))
69     ("&" (token-&))
70     ("-" (token--))
71     ("/" (token-/))
72     ("=" (token-=))
73     (number (token-NUM (string->number lexeme)))
74     (identifier (token-VAR (string->symbol lexeme)))
75     (whitespace (return-without-pos (sub-program-lexer input-port)))
76     ((eof) (token-EOF)))
77 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
78 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
79
80 (define program-parser
81   (parser
82     (start program); 開始記号に当たる非終端記号
83     (end EOF); 入力の終端に達した時に字句解析器が返すトークン
84     (src-pos); 位置情報を含むオブジェクトを返す
85     (debug "siple-parser.tbl")
86     (error (lambda (tok-ok? tok-name tok-value start-pos end-pos)
87              (error "parse error:" tok-name tok-value)))
88     (tokens tokens-with-value tokens-without-value)

```

```

89
90 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;BNFの指定;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
91 (grammar
92   (program ((program_with_print)
93     (cons
94       (stx:func_proto_st
95         (stx:spec_st 'void 'print-proto)
96         (stx:func_declarator_st 'print
97           (stx:para_declaration_st
98             (stx:spec_st 'int 'print-proto)
99             (stx:id_st 'v 'print-proto))
100             'print-proto))
101         $1))) ;プログラムの冒頭に組み関数 printのプロトタイプ宣言をつける。
102
103   (program_with_print ((external_declaration) $1)
104     ((program_with_print external_declaration) (cons $1 $2)))
105
106   #; (stx:func_proto_st
107     (stx:spec_st 'int 'print-proto)
108     (stx:func_declarator_st 'print
109       (stx:para_declaration_st (stx:spec_st 'int 'print-proto)
110         (stx:id_st 'i 'print-proto))))
111
112   (external_declaration ((declaration) $1)
113     ((function_prototype) $1)
114     ((function_definition) $1))
115   (declaration ((type_specifier declarator_list semicolon)
116     (stx:declaration_st $1 $2)))
117   (declarator_list ((declarator) $1)
118     ((declarator_list comma declarator)(cons $1 $3)))
119   (declarator ((direct_declarator)
120     (stx:declarator_st $1))
121     ((* direct_declarator)
122     (stx:declarator_ast_st $2)
123     ))
124
125   (direct_declarator ((VAR) (stx:id_st $1 $1-start-pos))
126     ((VAR l_mid_paren NUM r_mid_paren)
127     (stx:array_st $1 $3 $1-start-pos)))
128
129   (function_prototype ((type_specifier function_declarator semicolon)
130     (stx:func_proto_st $1 $2)))
131   (function_declarator ((VAR l_small_paren parameter_type_list r_small_paren)
132     (stx:func_declarator_st $1 $3 $1-start-pos))
133     ((VAR l_small_paren r_small_paren)
134     (stx:func_declarator_null_st $1 $1-start-pos))
135     ((* VAR l_small_paren parameter_type_list r_small_paren)
136     (stx:func_declarator_ast_st $2 $4 $2-start-pos))
137     ((* VAR l_small_paren r_small_paren)
138     (stx:func_declarator_ast_null_st $2 $2-start-pos)))
139
140   (function_definition ((type_specifier function_declarator compound_statement)
141     (stx:func_def_st $1 $2 $3)))
142   (parameter_type_list ((parameter_declaration) $1)
143     ((parameter_type_list comma parameter_declaration)(cons $1 $3)))
144   (parameter_declaration ((type_specifier parameter_declarator)
145     (stx:para_declaration_st $1 $2)))
146   (parameter_declarator ((VAR) (stx:id_st $1 $1-start-pos))
147     ((* VAR)(stx:id_ast_st $2 $2-start-pos)))
148   (type_specifier ((int) (stx:spec_st 'int $1-start-pos))
149     ((void) (stx:spec_st 'void $1-start-pos)))
150   (statement ((semicolon) (stx:null_statement_st 'null))
151     ((expression semicolon)$1)
152     ((compound_statement) $1)
153     ((if l_small_paren expression r_small_paren statement)
154     ; シンタックスシュガー
155     (stx:if_else_st $3
156       $5
157       (stx:null_statement_st 'null)
158       $1-start-pos 'syntax-sygar))
159     ((if l_small_paren expression r_small_paren statement else statement)
160     (stx:if_else_st $3 $5 $7 $1-start-pos $6-start-pos))
161     ((while l_small_paren expression r_small_paren statement)
162     (stx:while_st $3 $5 $1-start-pos))
163     ((for l_small_paren expression
164       semicolon expression
165       semicolon expression
166       r_small_paren statement)
167     ; シンタックスシュガー
168     (stx:compound_sta_st

```

```

169         (cons
170           $3
171           (stx:while_st $5
172             (stx:compound_sta_st (cons $9 $7))
173             'syntax-sugar))))
174   ((for l_small_paren
175     semicolon expression
176     semicolon expression
177     r_small_paren statement)
178    ; シンタックスシュガー
179    (stx:while_st $4
180      (stx:compound_sta_st (cons $8 $6))
181      'syntax-sugar))
182   ((for l_small_paren expression
183     semicolon
184     semicolon expression
185     r_small_paren statement)
186    ; シンタックスシュガー
187    (stx:compound_sta_st
188      (cons
189        $3
190        (stx:while_st (stx:null_statement_st 'null)
191          (stx:compound_sta_st (cons $8 $6))
192          'syntax-sugar))))
193   ((for l_small_paren expression
194     semicolon expression
195     semicolon
196     r_small_paren statement)
197    ; シンタックスシュガー
198    (stx:compound_sta_st
199      (cons
200        $3
201        (stx:while_st $5
202          $8
203          'syntax-sugar))))
204   ((for l_small_paren expression
205     semicolon
206     semicolon
207     r_small_paren statement)
208    ; シンタックスシュガー
209    (stx:compound_sta_st
210      (cons
211        $3
212        (stx:while_st (stx:null_statement_st 'null)
213          $7
214          'syntax-sugar))))
215   ((for l_small_paren
216     semicolon expression
217     semicolon
218     r_small_paren statement)
219    ; シンタックスシュガー
220    (stx:while_st $4
221      $7
222      'syntax-sugar))
223   ((for l_small_paren
224     semicolon
225     semicolon expression
226     r_small_paren statement)
227    ; シンタックスシュガー
228    (stx:while_st (stx:null_statement_st 'null)
229      (stx:compound_sta_st (cons $7 $5))))
230   ((for l_small_paren
231     semicolon
232     semicolon
233     r_small_paren statement)
234    ; シンタックスシュガー
235    (stx:while_st (stx:null_statement_st 'null)
236      $6
237      'syntax-sugar))
238
239   ((return expression semicolon)(stx:return_st $2 $1-start-pos))
240   ((return semicolon)(stx:return_st 'noreturn $1-start-pos)))
241 (compound_statement ((l_big_paren declaration_list statement_list r_big_paren)
242   (stx:compound_st $2 $3))
243   ((l_big_paren declaration_list r_big_paren)
244     (stx:compound_dec_st $2))
245   ((l_big_paren statement_list r_big_paren)
246     (stx:compound_sta_st $2))
247   ((l_big_paren r_big_paren)
248     (stx:compound_null_st 'null)))

```

```

249 (declaration_list ((declaration) $1)
250                   ((declaration_list declaration)(cons $1 $2)))
251 (statement_list ((statement) $1)
252                 ((statement_list statement)(cons $1 $2)))
253 (expression ((assign_expr) $1)
254             ((expression comma assign_expr)(cons $1 $3)))
255 (assign_expr ((logical_OR_expr) $1)
256             ((logical_OR_expr = assign_expr)
257              (stx:assign_exp_st $1 $3 $2-start-pos)))
258 (logical_OR_expr ((logical_AND_expr) $1)
259                 ((logical_OR_expr or logical_AND_expr)
260                  (stx:logic_exp_st 'or $1 $3 $2-start-pos)))
261 (logical_AND_expr ((equality_expr) $1)
262                   ((logical_AND_expr and equality_expr)
263                    (stx:logic_exp_st 'and $1 $3 $2-start-pos)))
264 (equality_expr ((relational_expr) $1)
265                ((equality_expr equal relational_expr)
266                 (stx:rel_exp_st 'equal $1 $3 $2-start-pos)))
267                ((equality_expr not relational_expr)
268                 (stx:rel_exp_st 'not $1 $3 $2-start-pos)))
269
270 (relational_expr ((add_expr) $1)
271                 ((relational_expr less add_expr)
272                  (stx:rel_exp_st 'less $1 $3 $2-start-pos))
273                 ((relational_expr more add_expr)
274                  (stx:rel_exp_st 'more $1 $3 $2-start-pos))
275                 ((relational_expr and_less add_expr)
276                  (stx:rel_exp_st 'and_less $1 $3 $2-start-pos))
277                 ((relational_expr and_more add_expr)
278                  (stx:rel_exp_st 'adn_more $1 $3 $2-start-pos)))
279
280 (add_expr ((mult_expr) $1)
281           ((add_expr + mult_expr)
282            (stx:alge_exp_st 'add $1 $3 $2-start-pos))
283           ((add_expr - mult_expr)
284            (stx:alge_exp_st 'sub $1 $3 $2-start-pos)))
285 (mult_expr ((unary_expr) $1)
286            ((mult_expr * unary_expr)
287             (stx:alge_exp_st 'mul $1 $3 $2-start-pos))
288            ((mult_expr / unary_expr)
289             (stx:alge_exp_st 'div $1 $3 $2-start-pos)))
290 (unary_expr ((postfix_expr) $1)
291             ((- unary_expr)
292              ; シンタックスシュガー
293              (stx:alge_exp_st 'sub
294                           (stx:constant_st 0 'syntax-sugar)
295                           $2
296                           $1-start-pos))
297             (& unary_expr)
298             (if (stx:exp_in_paren_st? $2)
299                 (if (stx:unary_exp_st? (stx:exp_in_paren_st-exp $2))
300                     (if (equal? 'ast
301                                 (stx:unary_exp_st-mark (stx:exp_in_paren_st-exp $2)))
302                         ; 間接参照式のシンタックスシュガー
303                         (stx:unary_exp_st-op (stx:exp_in_paren_st-exp $2))
304                         (stx:unary_exp_st 'amp $2 $1-start-pos))
305                         (stx:unary_exp_st 'amp $2 $1-start-pos))
306                     (if (stx:unary_exp_st? $2)
307                         (if (equal? 'ast (stx:unary_exp_st-mark $2))
308                             (stx:unary_exp_st-op $2)
309                             (stx:unary_exp_st 'amp $2 $1-start-pos))
310                         (stx:unary_exp_st 'amp $2 $1-start-pos))))
311             ((* unary_expr)
312              (stx:unary_exp_st 'ast $2 $1-start-pos)))
313
314 (postfix_expr ((primary_expr) $1)
315               ((postfix_expr l_mid_paren expression r_mid_paren)
316                ; (stx:array_var_st $1 $3 $1-start-pos)
317                ; 配列参照式のシンタックスシュガー
318                (stx:unary_exp_st 'ast
319                                (stx:exp_in_paren_st
320                                 (stx:alge_exp_st 'add $1 $3 'syntax-sugar))
321                                'syntax-sugar))
322               ((VAR l_small_paren argument_expression_list r_small_paren)
323                (stx:func_st $1 $3))
324               ((VAR l_small_paren r_small_paren)
325                (stx:func_st $1 'nopara)))
326 (primary_expr ((VAR)(stx:id_st $1 $1-start-pos))
327               ((NUM)(stx:constant_st $1 $1-start-pos))
328               ((l_small_paren expression r_small_paren))

```

```

329         (stx:exp_in_paren_st $2)))
330     (argument_expression_list ((assign_expr) $1)
331         ((argument_expression_list comma assign_expr)(cons $1 $3))))))
332
333 ::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
334
335 (define (parse-string s)
336   (let ((p (open-input-string s)))
337     (program-parser (lambda () (sub-program-lexer p)))))
338
339 (define (parse-port p)
340   (program-parser (lambda () (sub-program-lexer p))))
341 #;(begin
342   (define p9999 (open-input-file "kadai01.c"))
343   (port-count-lines! p9999)
344   (parse-port p9999))

```

## 1.2 設計方針

この課題は前節の課題5で作成した抽象構文木作成の手続きに上記のシンタックスシュガーの変換処理と print 関数のプロトタイプ宣言を付け加えるだけである。具体的にはシンタックスシュガーに当てはまる文字列を受け取った際に、返す構文木をシンタックスシュガーに対応する構文木に変更し、プログラムを読み込むと同時にその先頭に組み込み関数 print のプロトタイプ宣言を表す構文木を付加するようにしただけである。

## 1.3 各部の説明

単項演算のシンタックスシュガーは次のようになる。

リスト 2: 単項演算のシンタックスシュガー

```

1 ((~ unary_expr)
2  ;シンタックスシュガー
3  (stx:alge_exp_st 'sub
4    (stx:constant_st 0 'syntax-sugar) $2 $1-start-pos))

```

else 節のない if 文のシンタックスシュガーは次のようになる。

リスト 3: if 文のシンタックスシュガー

```

1 ((if l_small_paren expression r_small_paren statement)
2  ;シンタックスシュガー
3  (stx:if_else_st $3
4    $5
5    (stx:null_statement_st 'null)
6    $1-start-pos
7    'syntax-sugar))
8 \end{lstlisting}
9 for文のシンタックスシュガーは次のようになる。
10 なお、for文を表す構造体は条件節の式の有無に応じて8パターン
11 存在するが、ここではその一つを紹介する。
12 \begin{lstlisting}[caption=for文のシンタックスシュガー]
13 ((for l_small_paren expression
14      semicolon expression
15      semicolon expression
16      r_small_paren statement)
17  ;シンタックスシュガー
18  (stx:compound_sta_st
19    (cons
20      $3
21      (stx:while_st $5
22        (stx:compound_sta_st (cons $9 $7))
23        'syntax-sugar))))

```

配列参照式のシンタックスシュガーは次のようになる。

リスト 4: 配列参照式のシンタックスシュガー

```
1 ((postfix_expr l_mid_paren expression r_mid_paren)
2   ;(stx:array_var_st $1 $3 $1-start-pos)
3   ;配列参照式のシンタックスシュガー
4   (stx:unary_exp_st 'ast
5     (stx:alge_exp_st 'add $1 $3 'syntax-sugar)
6     'syntax-sugar))
```

```
1 (unary_expr ((postfix_expr) $1)
2   ((- unary_expr)
3     ;シンタックスシュガー
4     (stx:alge_exp_st 'sub
5       (stx:constant_st 0 'syntax-sugar)
6       $2
7       $1-start-pos))
8   (& unary_expr)
9   (if (stx:exp_in_paren_st? $2)
10     (if (stx:unary_exp_st? (stx:exp_in_paren_st-exp $2))
11       (if (equal? 'ast
12             (stx:unary_exp_st-mark (stx:exp_in_paren_st-exp $2)))
13         ;間接参照式のシンタックスシュガー
14         (stx:unary_exp_st-op (stx:exp_in_paren_st-exp $2))
15         (stx:unary_exp_st 'amp $2 $1-start-pos))
16         (stx:unary_exp_st 'amp $2 $1-start-pos))
17       (if (stx:unary_exp_st? $2)
18         (if (equal? 'ast (stx:unary_exp_st-mark $2))
19             (stx:unary_exp_st-op $2)
20             (stx:unary_exp_st 'amp $2 $1-start-pos))
21         (stx:unary_exp_st 'amp $2 $1-start-pos))))
22   ((* unary_expr)
23     (stx:unary_exp_st 'ast $2 $1-start-pos)))
```

また、`print` 関数のプロトタイプ宣言は次のように実装した。

リスト 5: `print` 関数のプロトタイプ

```
1 (program
2   ((program_with_print)
3     (cons
4       (stx:func_proto_st
5         (stx:spec_st 'void 'print-proto)
6         (stx:func_declarator_st
7           'print
8           (stx:para_declaration_st
9             (stx:spec_st 'int 'print-proto)
10            (stx:id_st 'v 'print-proto))
11            'print-proto))
12       $1)))
```

## 2 課題 10

この課題では、課題 8 で生成された抽象構文木に対して

- 意味解析
- 式の形の検査
- 式の型の検査

行う手続きを作成する。それぞれのソースコードは次のようになった。

## リスト 6: 意味解析について (README.md)

```

1 # compiler
2 ##意味解析部
3 ###analy-declaration_st(チェック部分開発途中)
4 ;(stx:declaration_st...) と
5 ;__分析に使う環境env__と
6 ;__current-lev__
7 ;を受け取って
8 ;(stx:declaration_st type-spec (list (obj...) (obj...)...))
9 ;を返す.
10 ;同時にlistの形で環境に追加.
11 ;同時に環境のチェックも行う.
12
13 ###analy-func_proto_st
14 ;(stx:func_proto_st...)
15 ;を受け取って
16 ;(stx:func_proto_st (stx:spec_st...)
17 ; (stx:func_declarator/_ast/_st 関数名
18 ; (list obj...)))
19 ;を返し
20 ;同時に関数プロトタイプのobject(obj name 0 'proto type)を
21 ;環境に登録.
22 ;パラメータのobject(obj name 1 'parm type)の(list obj...)を
23 ;パラメータ専用の環境をまず初期化してから登録
24
25 ###analy-func_def_st
26 ;stx:func_def_stを
27 ;引数に取り
28 ;(stx:func_def_st stx:spec_st
29 ; (func_declarator_st '関数宣言のオブジェクト'
30 ; 'パラメータのオブジェクトのlist')
31 ; compound-statement)
32 ;(compound-statement部分については関数analy-compound_stに任せる.)
33 ;を返す.
34 ;同時にパラメータのオブジェクトをパラメータ専用の環境に追加、チェック
35 ;同時に関数宣言のオブジェクトを環境に追加、チェック
36
37 ###analy-compound_st
38 ;stx:compound_stか
39 ;stx:compound_dec_stか
40 ;stx:compound_sta_stか
41 ;stx:compound_null_stと
42 ;lev
43 ;を受け取って
44 ;(stx:compound_st declaration-list statement-list)
45 ;を返す.
46 ;ただし
47 ;declaration-listが無いときは'nodecl
48 ;statement-listが無いときは'nostat
49 ;を入れる
50 ;同時に
51 ;意味解析開始時にcurrent-levをひとつ上げる
52 ;終了時に1つ下げる
53
54 ###analy-compdecl
55 ;__analy-declaration_stの派生__
56 ;(list* (stx:declaration_st...)...)と
57 ;lev(解析中のcompound-statementのブロックレベル)
58 ;を引数に取り
59 ;(list* obj)
60 ;を返す関数
61 ;_analy-declarationとは違って外部の大域の環境を更新しない.
62 ;compound-statementの意味解析結果の環境としては(list* obj)を直接使用することとする.
63
64 ###analy-compstate
65 ;levと
66 ;envと
67 ;compound_stなどに入るstatementを
68 ;引数に取り
69 ;それぞれのobjを
70 ;返す関数
71 ;同時にenvをもとにstatement内の定義をチェックする.

```

## リスト 7: 意味解析を行う処理 semantic-analy.rkt

```

1 #lang racket
2 (require "myenv.rkt")

```



```

3 (require (prefix-in stx: "mysyntax.rkt"))
4 (require (prefix-in k08: "kadai08.rkt"))
5 (require "mymap.rkt")
6 (provide (all-defined-out))
7
8 ;(define current-lev 0)
9 ;(define comp-lev 0)
10 (define env '())
11 (define para-env '())
12 (define comp-env '())
13
14
15 (define (analy-declaration_st st lev)
16   ;;;;
17   ;内部定義
18   ;(stx:declarator_st...)と
19   ;levと
20   ;'intもしくは'void
21   ;を引数にとり
22   ;obj
23   ;を返す関数.
24   (define (make-obj-from-decl decl type lev)
25     (let* ((id (cond ((stx:declarator_st? decl)
26                       (stx:declarator_st-var decl)
27                       ((stx:declarator_ast_st? decl)
28                        (stx:declarator_ast_st-var decl))))
29            (name (cond ((stx:id_st? id) (stx:id_st-name id))
30                        ((stx:array_st? id) (stx:array_st-name id))))
31            (flag (cond ((stx:declarator_st? decl) 'nomal)
32                        ((stx:declarator_ast_st? decl) 'pointer)))
33            (kind 'var)
34            (type (cond ((stx:array_st? id) (type_array type (stx:array_st-num id)))
35                        (else (cond ((equal? flag 'nomal) type)
36                                     ((equal? flag 'pointer) (type_pointer 'pointer type))))))
37            (pos (cond ((stx:id_st? id)
38                        (stx:id_st-pos id))
39                       ((stx:id_ast_st? id)
40                        (stx:id_ast_st-pos id))
41                       ((stx:array_st? id)
42                        (stx:array_st-pos id)))))
43       (obj name lev kind type pos)))
44   ;;;;
45   (let* (; typeに入っているのは (stx:spec_st 'intか'void ポジション)
46         (type (stx:declaration_st-type-spec st))
47         (declarator-list (stx:declaration_st-declarator-list st))
48         ;objのlistを作成する.
49         (obj-list (map*
50                    (lambda (x) (make-obj-from-decl x (stx:spec_st-type type) lev))
51                    declarator-list)))
52     ;意味解析上のエラーがないか確認する.
53     (map (lambda (x) (check-decl x env)) obj-list)
54     (map (lambda (x) (check-decl x para-env)) obj-list)
55     ;なければ環境に追加.
56     (set! env (add-list obj-list env))
57     ;構造体を返す.
58     (stx:declaration_st type obj-list)))
59
60
61 (define (analy-func_proto_st st)
62   ;;;;内部定義
63   ;make-obj-from-paralistは
64   ;(list* (stx:para_declaration_st...)...)
65   ;を引数に取り、
66   ;(list obj...)
67   ;を返す.
68   ;'noparaの時は関数の外で処理する.
69   (define (make-obj-from-paralist para-list)
70     (map* (lambda (para-decl)
71             (let* (;typeは'intとか.この時点では確定しない.最終的にはflagと合わせて決定.
72                   (type (stx:spec_st-type (stx:para_declaration_st-type-spec para-decl)))
73                   ;idはstx:id_stかstx:id_ast_st
74                   (id (stx:para_declaration_st-para para-decl))
75                   ;flagはポインタ型なら'pointer、そうでなければ'normal
76                   (flag (cond ((stx:id_st? id) 'normal)
77                               ((stx:id_ast_st? id) 'pointer)))
78                   (name (cond ((equal? flag 'normal) (stx:id_st-name id))
79                               ((equal? flag 'pointer) (stx:id_ast_st-name id))))
80                   (pos (cond ((stx:id_st? id) (stx:id_st-pos id))
81                               ((stx:id_ast_st? id) (stx:id_ast_st-pos id))))
82                   (lev 1)

```

```

83         (kind 'parm)
84         (type (cond ((equal? flag 'normal) type)
85                     ((equal? flag 'pointer)(type_pointer 'pointer type))))
86         (obj name lev kind type pos)))
87     para-list))
88 ;;;; 内部定義ここまで
89 (let* (;このspecがintで返り値が*intの場合あり.
90       ;返り値は最終的にはこの型とflagで決定される.
91       ;specはstx:spec_st
92       (spec (stx:func_proto_st-type-spec st))
93       ;declはstx:func_declarator/_null/_ast/_stの4つの場合がある.
94       (decl (stx:func_proto_st-func-declarator-st st))
95       ;返り値がnormalかpointerか、パラメータの有無がnoremalかnoneか
96       ;para_flagは(struct para_flag (out-type para))で定義される構造体.
97       (flag (cond ((stx:func_declarator_st? decl)(para_flag 'normal 'normal))
98                   ((stx:func_declarator_null_st? decl)(para_flag 'normal 'none))
99                   ((stx:func_declarator_ast_st? decl) (para_flag 'pointer 'normal))
100                  ((stx:func_declarator_ast_null_st? decl) (para_flag 'pointer 'none))))
101       (proto-name (cond ((stx:func_declarator_st? decl)
102                          (stx:func_declarator_st-name decl))
103                         ((stx:func_declarator_null_st? decl)
104                          (stx:func_declarator_null_st-name decl))
105                         ((stx:func_declarator_ast_st? decl)
106                          (stx:func_declarator_ast_st-name decl))
107                         ((stx:func_declarator_ast_null_st? decl)
108                          (stx:func_declarator_ast_null_st-name decl))))
109       ;プロトタイプの位置情報
110       (proto-pos (cond ((stx:func_declarator_st? decl)
111                          (stx:func_declarator_st-pos decl))
112                        ((stx:func_declarator_null_st? decl)
113                          (stx:func_declarator_null_st-pos decl))
114                        ((stx:func_declarator_ast_st? decl)
115                          (stx:func_declarator_ast_st-pos decl))
116                        ((stx:func_declarator_ast_null_st? decl)
117                          (stx:func_declarator_ast_null_st-pos decl))))
118       ;para-listは(list* (stx:para_declaration_st...)...)
119       ;もしくはパラメータが無いときは'noparaが入っている.
120       (para-list (cond ((stx:func_declarator_st? decl)
121                          (stx:func_declarator_st-para-list decl))
122                        ((stx:func_declarator_null_st? decl)
123                          'nopara)
124                        ((stx:func_declarator_ast_st? decl)
125                          (stx:func_declarator_ast_st-para-list decl))
126                        ((stx:func_declarator_ast_null_st? decl)
127                          'nopara)))
128       ;para-obj-listは(list obj...)もしくは'nopara
129       (para-obj-list (cond ((equal? para-list 'nopara) 'nopara)
130                             (else (make-obj-from-paralist para-list))))
131       (proto-type (cond ((equal? 'normal (para_flag-out-type flag))
132                           (type_fun 'fun
133                                     (stx:spec_st-type spec)
134                                     (cond ((equal? 'nopara para-obj-list)
135                                             'nopara)
136                                             (else (map (lambda (x) (obj-type x))
137                                                         para-obj-list)))))
137                         ;(struct type-pointer (pointer type) #:transparent)
138                         ((equal? 'pointer (para_flag-out-type flag))
139                          (type_fun 'fun
140                                    (type_pointer 'pointer (stx:spec_st-type spec))
141                                    (cond ((equal? 'nopara para-obj-list)
142                                            'nopara)
143                                            (else (map (lambda (x) (obj-type x))
144                                                         para-obj-list)))))
144                         (proto-obj (obj proto-name 0 'proto proto-type proto-pos)))
145       ;プロトタイプのオブジェクトのチェック
146       (check-proto proto-obj env)
147       ;プロトタイプのオブジェクトを環境に追加.
148       (set! env (extend-env proto-obj env))
149       ;パラメータ内の二重宣言をチェック.
150       (check-proto-para para-obj-list)
151       ;;;;;;
152       ;返す構造体
153       ;ここで返したいものはlet*で取り出しておく必要がある.
154       (stx:func_proto_st spec (stx:func_declarator_st proto-obj para-obj-list proto-pos)))
155
156 (define (analy-func_def_st st)
157     ;;;; 内部定義
158     ;make-obj-from-paralistは
159     ;(list* (stx:para_declaration_st...)...)
160     ;を引数に取り、

```

```

163 ;(list obj...)
164 ;を返す.
165 ;'noparaの時は関数の外で処理する.
166 (define (make-obj-from-paralist para-list)
167   (map* (lambda (para-decl)
168     (let* (;typeは'intとか.この時点では確定しない.最終的にはflagと合わせて決定.
169           (type (stx:spec_st-type (stx:para_declaration_st-type-spec para-decl)))
170           ;idはstx:id_stかstx:id_ast_st
171           (id (stx:para_declaration_st-para para-decl))
172           ;flagはポインタ型なら'pointer、そうでなければ'normal
173           (flag (cond ((stx:id_st? id) 'normal)
174                       ((stx:id_ast_st? id) 'pointer)))
175           (name (cond ((equal? flag 'normal) (stx:id_st-name id))
176                      ((equal? flag 'pointer) (stx:id_ast_st-name id))))
177           (pos (cond ((stx:id_st? id) (stx:id_st-pos id))
178                     ((stx:id_ast_st? id) (stx:id_ast_st-pos id))))
179           (lev 1)
180           (kind 'parm)
181           (type (cond ((equal? flag 'normal) type)
182                      ((equal? flag 'pointer) (type_pointer 'pointer type))))
183           (obj name lev kind type pos)))
184     para-list))
185 ;;;;内部定義ここまで
186 (let* (;このspecがintで返り値が*intの場合あり.
187       ;返り値は最終的にはこの型とflagで決定される.
188       ;specはstx:spec_st
189       (spec (stx:func_def_st-type-spec st))
190       ;declはstx:func_declarator/_null/_ast/_stの4つの場合がある.
191       (decl (stx:func_def_st-func-declarator-st st))
192       (compo (stx:func_def_st-compound-state-list st))
193       ;返り値がnormalかpointerか、パラメータの有無がnoremalかnoneか
194       ;fundef_flagは(struct fundef_flag (out-type para))で定義される構造体.
195       (flag (cond ((stx:func_declarator_st? decl) (fundef_flag 'normal 'normal))
196                  ((stx:func_declarator_null_st? decl) (fundef_flag 'normal 'none))
197                  ((stx:func_declarator_ast_st? decl) (fundef_flag 'pointer 'normal))
198                  ((stx:func_declarator_ast_null_st? decl) (fundef_flag 'pointer 'none))))
199       (fundef-name (cond ((stx:func_declarator_st? decl)
200                          (stx:func_declarator_st-name decl))
201                          ((stx:func_declarator_null_st? decl)
202                           (stx:func_declarator_null_st-name decl))
203                          ((stx:func_declarator_ast_st? decl)
204                           (stx:func_declarator_ast_st-name decl))
205                          ((stx:func_declarator_ast_null_st? decl)
206                           (stx:func_declarator_ast_null_st-name decl))))
207       ;関数定義の位置情報
208       (fundef-pos (cond ((stx:func_declarator_st? decl)
209                          (stx:func_declarator_st-pos decl))
210                         ((stx:func_declarator_null_st? decl)
211                          (stx:func_declarator_null_st-pos decl))
212                         ((stx:func_declarator_ast_st? decl)
213                          (stx:func_declarator_ast_st-pos decl))
214                         ((stx:func_declarator_ast_null_st? decl)
215                          (stx:func_declarator_ast_null_st-pos decl))))
216       ;para-listは(list* (stx:para_declaration_st...)...)
217       ;もしくはパラメータが無いときは'noparaが入っている.
218       (para-list (cond ((stx:func_declarator_st? decl)
219                        (stx:func_declarator_st-para-list decl))
220                       ((stx:func_declarator_null_st? decl)
221                        'nopara)
222                       ((stx:func_declarator_ast_st? decl)
223                        (stx:func_declarator_ast_st-para-list decl))
224                       ((stx:func_declarator_ast_null_st? decl)
225                        'nopara)))
226       ;para-obj-listは(list obj...)もしくは'nopara
227       (para-obj-list (cond ((equal? para-list 'nopara) 'nopara)
228                            (else (make-obj-from-paralist para-list))))
229       (fundef-type (cond ((equal? 'normal (fundef_flag-out-type flag))
230                          (type_fun 'fun
231                                     (stx:spec_st-type spec)
232                                     (cond ((equal? 'nopara para-obj-list) 'nopara)
233                                             (else (map (lambda (x) (obj-type x))
234                                                         para-obj-list))))))
235                      ((equal? 'pointer (fundef_flag-out-type flag))
236                       (type_fun 'fun
237                                  (type_pointer 'pointer (stx:spec_st-type spec))
238                                  (cond ((equal? 'nopara para-obj-list) 'nopara)
239                                          (else (map (lambda (x) (obj-type x))
240                                                      para-obj-list))))))
240                      (else (error "IN VALID FUNCTION"))))
241       (fundef-obj (obj fundef-name 0 'fun fundef-type fundef-pos)))
242

```

```

243 ;関数定義のオブジェクトのチェック
244 (check-func fundef-obj env)
245 ;関数定義のオブジェクトを環境に追加.
246 (set! env (extend-env fundef-obj env))
247 ;パラメータ内の二重宣言をチェック
248 (check-def-para para-obj-list)
249 ;パラメータの環境を登録
250 (set! para-env para-obj-list)
251 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
252 ;返す構造体
253 ;ここで返したいものはlet*で取り出しておく必要がある.
254 (stx:func_def_st spec
255   (stx:func_declarator_st fundef-obj para-obj-list fundef-pos)
256   (analy-compound_st compo 1 env fundef-obj)
257   ))
258
259 (define (analy-compound_st st lev outer-env func-tag)
260   (let* ((flag (cond ((stx:compound_st? st) (comp_flag 'normal 'normal 1))
261                     ((stx:compound_dec_st? st) (comp_flag 'normal 'nostat 2))
262                     ((stx:compound_sta_st? st) (comp_flag 'nodecl 'normal 3))
263                     ((or (stx:null_statement_st? st)
264                          (stx:compound_null_st? st)) (comp_flag 'nodecl 'nostat 4))
265                     (else (error "UNEXPECTED STRUCTURE! ERROR IN ANALY-COMPOUND." st))))
266         ;decl-listには(list* stx:declaration_st...)
267         (decl-list (cond ((equal? 1 (comp_flag-n flag))
268                          (stx:compound_st-declaration-list st))
269                          ((equal? 2 (comp_flag-n flag))
270                           (stx:compound_dec_st-declaration-list st))
271                          ((or (equal? 3 (comp_flag-n flag))
272                              (equal? 4 (comp_flag-n flag))) 'nodecl)))
273         ;statement-listにはstatementのlist*が入る.
274         ;処理する際はmap*で
275         (stat-list (cond ((equal? 1 (comp_flag-n flag))
276                          (stx:compound_st-statement-list st))
277                       ((equal? 3 (comp_flag-n flag))
278                        (stx:compound_sta_st-statement-list st))
279                       ((or (equal? 2 (comp_flag-n flag)) (equal? 4 (comp_flag-n flag))) 'nostat)))
280         ;意味解析開始時にlevを一つ繰り上げる
281         (this-lev (+ lev 1))
282         ;decl-listに入っているのは(list stx:declaration_st...)が'nodecl
283         (decl-list (cond ((equal? 'normal (comp_flag-decl flag))
284                          ;このときオブジェクトはcomp-envに追加する必要がある.
285                          ;(map* analy-compdecl decl-list
286                           (map* (lambda (x) (analy-compdecl x this-lev)) decl-list))
287                          ((equal? 'nodecl (comp_flag-decl flag))
288                           'nodecl)))
289         ;decl-listからこのcompoun-statement内で新しく生成される環境を格納する.
290         ;comp-env内はこのcompound-statement内で新しく定義されたオブジェクトのlist
291         (comp-env
292          (cond
293           ((equal? 'nodecl decl-list) 'nodecl)
294           (else ;listの各要素がlistであることを保証させるためのmake-list-list
295                (flatten (map (lambda (x) (stx:declaration_st-declarator-list x)) decl-list)))))
296         ;comp-envのチェック
297         ;代入には意味は無い.let*の代入文の段階でチェックを実行しておく必要があるためこのようにした.
298         ;comp-env内のみで二重定義などが無いかどうかをチェックする.
299         (comp-env-check (check-comp-env comp-env))
300         ;大域環境と照らし合わせる
301         (comp-env-check (cond ((equal? 'nodecl comp-env) #t)
302                              (else (map (lambda (x) (check-decl x outer-env)) comp-env))))
303         (comp-env-check (cond ((equal? 'nodecl comp-env) #t)
304                              (else (map (lambda (x) (check-decl x para-env)) comp-env))))
305         ;outer-envは大域環境を含む.
306         (new-comp-env (cond ((equal? 'nodecl comp-env) outer-env)
307                             (else (append comp-env outer-env))))
308         (stat-list (cond ((equal? 'normal (comp_flag-stat flag))
309                          (map*
310                           (lambda (x) (analy-compstate x this-lev new-comp-env func-tag))
311                           stat-list))
312                      ((equal? 'nostat (comp_flag-stat flag))
313                       'nostat))))
314         (stx:compound_st decl-list stat-list)))
315
316 (define (analy-compdecl st lev)
317   ;;;;
318   ;内部定義
319   ;(stx:declarator_st...)と
320   ;levと
321   ;'intもしくは'void
322   ;を引数にとり

```

```

323 ;obj
324 ;を返す関数.
325 (define (make-obj-from-decl decl type lev)
326   (let* ((id (cond ((stx:declarator_st? decl)
327                     (stx:declarator_st-var decl))
328                   ((stx:declarator_ast_st? decl)
329                     (stx:declarator_ast_st-var decl))))
330     (name (cond ((stx:id_st? id) (stx:id_st-name id))
331                 ((stx:id_ast_st? id) (stx:id_ast_st-name id))
332                 ((stx:array_st? id) (stx:array_st-name id))))
333     (pos (cond ((stx:id_st? id) (stx:id_st-pos id))
334                ((stx:id_ast_st? id) (stx:id_ast_st-pos id))
335                ((stx:array_st? id) (stx:array_st-pos id))))
336     (flag (cond ((stx:declarator_st? decl) 'nomal)
337                 ((stx:declarator_ast_st? decl) 'pointer)))
338     (kind 'var)
339     (type (cond ((stx:array_st? id) (type_array type (stx:array_st-num id)))
340                 (else (cond ((equal? flag 'nomal) type)
341                               ((equal? flag 'pointer) (type_pointer 'pointer type)))))))
342   (obj name lev kind type pos)))
343 ;;;; 内部定義ここまで
344 (let* (; typeに入っているのは (stx:spec_st 'intか'void ポジション)
345       (type (stx:declaration_st-type-spec st))
346       (declarator-list (stx:declaration_st-declarator-list st))
347       ;objのlistを作成する.
348       (obj-list (map*
349                  (lambda (x) (make-obj-from-decl x (stx:spec_st-type type) lev))
350                  declarator-list)))
351   ;意味解析上のエラーがないかは外側でチェックするのでここでは実装しなくて良い.
352   (stx:declaration_st type obj-list)))
353
354 ;envはnodeclの場合あり.
355 (define (analy-compstate st lev env func-tag)
356   (cond ((stx:null_statement_st? st)
357         st)
358         ((stx:assign_exp_st? st)
359          (stx:assign_exp_st (analy-compstate (stx:assign_exp_st-dest st) lev env func-tag)
360                              (analy-compstate (stx:assign_exp_st-src st) lev env func-tag)
361                              (stx:assign_exp_st-pos st)))
362         ((stx:logic_exp_st? st)
363          (stx:logic_exp_st (stx:logic_exp_st-log-ope st)
364                              (analy-compstate (stx:logic_exp_st-op1 st) lev env func-tag)
365                              (analy-compstate (stx:logic_exp_st-op2 st) lev env func-tag)
366                              (stx:logic_exp_st-pos st)))
367         ((stx:rel_exp_st? st)
368          (stx:rel_exp_st (stx:rel_exp_st-rel-ope st)
369                          (analy-compstate (stx:rel_exp_st-op1 st) lev env func-tag)
370                          (analy-compstate (stx:rel_exp_st-op2 st) lev env func-tag)
371                          (stx:rel_exp_st-pos st)))
372         ((stx:alge_exp_st? st)
373          (stx:alge_exp_st (stx:alge_exp_st-alge-ope st)
374                          (analy-compstate (stx:alge_exp_st-op1 st) lev env func-tag)
375                          (analy-compstate (stx:alge_exp_st-op2 st) lev env func-tag)
376                          (stx:alge_exp_st-pos st)))
377         ((stx:unary_exp_st? st)
378          (let* ((normal-out (stx:unary_exp_st (stx:unary_exp_st-mark st)
379                                                (analy-compstate
380                                                  (stx:unary_exp_st-op st) lev env func-tag)
381                                                  (stx:unary_exp_st-pos st)))
382                (pos (stx:unary_exp_st-pos st)))
383            ;*で表された配列参照式を判別する.
384            (cond ((equal? 'ast (stx:unary_exp_st-mark st))
385                  (cond ((stx:exp_in_paren_st? (stx:unary_exp_st-op st))
386                        (cond ((stx:alge_exp_st? (stx:exp_in_paren_st-exp
387                                                  (stx:unary_exp_st-op st)))
388                              (cond ((equal? 'add
389                                             (stx:alge_exp_st-alge-ope
390                                              (stx:exp_in_paren_st-exp
391                                               (stx:unary_exp_st-op st))))
392                                ;ここまでで*(x + y)の形までが保証される.
393                                (analy-compstate
394                                  (stx:array_var_st
395                                    (stx:alge_exp_st-op1
396                                      (stx:exp_in_paren_st-exp
397                                       (stx:unary_exp_st-op st)))
398                                    (stx:alge_exp_st-op2
399                                      (stx:exp_in_paren_st-exp
400                                       (stx:unary_exp_st-op st)))
401                                    pos) lev env func-tag))
402                              (else normal-out)))

```

```

403         (else normal-out)))
404     (else normal-out)))
405     (else normal-out))))
406 ((stx:constant_st? st) st)
407 ((stx:exp_with_semi_st? st)
408  (stx:exp_with_semi_st
409   (analy-compstate (stx:exp_with_semi_st-exp st) lev env func-tag)))
410 ((stx:exp_in_paren_st? st)
411  (stx:exp_in_paren_st
412   (analy-compstate (stx:exp_in_paren_st-exp st) lev env func-tag)))
413 ((stx:if_else_st? st)
414  (stx:if_else_st
415   (analy-compstate (stx:if_else_st-cond-exp st) lev env func-tag)
416   (analy-compstate (stx:if_else_st-state st) lev env func-tag)
417   (analy-compstate (stx:if_else_st-else-state st) lev env func-tag)
418   (stx:if_else_st-if-pos st)
419   (stx:if_else_st-else-pos st)))
420 ((stx:while_st? st)
421  (stx:while_st
422   (analy-compstate (stx:while_st-cond-exp st) lev env func-tag)
423   (analy-compound_st (stx:while_st-statement st) lev env func-tag)
424   (stx:while_st-pos st)))
425 ((stx:return_st? st)
426  (stx:sem_return_st
427   (analy-compstate (stx:return_st-exp st) lev env func-tag)
428   (stx:return_st-pos st)
429   func-tag))
430 ((or (stx:compound_st? st)
431      (stx:compound_dec_st? st)
432      (stx:compound_sta_st? st)
433      (stx:compound_null_st? st))
434  (analy-compound_st st lev env func-tag))
435 ; チェック時は環境に 'nodecl' が入ることがあることに注意.
436 ((stx:func_st? st)
437  (stx:func_st (check-func-ref st lev env)
438               (cond ((equal? 'nopara (stx:func_st-para st)) 'nopara)
439                     (else (map (lambda (x)
440                                  (analy-compstate x lev env func-tag))
441                                (flatten (stx:func_st-para st)))))))
442 ((or (stx:id_st? st)
443      (stx:id_ast_st? st)
444      (stx:array_var_st? st))
445  ; stはid_stもしくはid_ast_st
446  ; levはcompound-statementのレベル
447  ; envは大域環境(objのlist)
448  (check-var-ref st lev
449   (append (cond ((equal? 'nopara para-env) '())
450                 (else para-env)) env)))
451 ; デバッグ用. (本来はどんなプログラムを書いてもこの分岐には入らないはず.)
452 (else (error "AN UNEXPECTED STRUCTURE! CONDITION ERROR IN ANALY-COMPSTATE FOR" st))
453 )
454
455 (define (sem-analyze-tree t)
456   (define (sem-analyze-struct st)
457     (cond ((stx:declaration_st? st) (analy-declaration_st st 0))
458           ((stx:func_proto_st? st) (analy-func_proto_st st))
459           ((stx:func_def_st? st) (analy-func_def_st st))
460           (else
461            (error
462             "SYNTAX ERROR! AN EXPECTED ARGUMENT >
463              DECLARATION/FUNCTION PROTOTYPE/FUNCTION DEFINITION."))))
464   (let* ((out-tree (map* sem-analyze-struct t)))
465     (set! env '())
466     out-tree))
467
468
469
470
471 ; テスト
472 #;(begin
473   (define p101 (open-input-file "kadai01.c"))
474   (port-count-lines! p101)
475   (sem-analyze-tree (k08:parse-port p101)))

```

リスト 8: 式の形の検査を行う処理 `analy-form.rkt`

```

1 #lang racket
2 (require (prefix-in k08: "kadai08.rkt"))

```

```

3 (require (prefix-in sem: "semantic-analy.rkt"))
4 (require (prefix-in stx: "mysyntax.rkt"))
5 (require parser-tools/lex
6       (prefix-in : parser-tools/lex-sre)
7       parser-tools/yacc)
8 (require "myenv.rkt")
9 (provide (all-defined-out))
10
11 (define (analy-form t)
12   (begin (map form-check t)
13         (display "OK! THIS PROGRAM IS IN A CORRECT FORM.")))
14
15 (define (form-check st)
16   (cond
17     ((not (list? st)) (list st))
18     ((stx:declaration_st? st) #t)
19     ((stx:func_declarator_st? st) #t)
20     ((stx:func_proto_st? st) #t)
21     ((stx:func_def_st? st)
22      (cond ((eq? 'nostat (stx:compound_st-statement-list
23                    (stx:func_def_st-compound-state-list st)))
24            #t)
25            (else
26             (map form-check (stx:compound_st-statement-list
27                             (stx:func_def_st-compound-state-list st))))))
28     ((stx:assign_exp_st? st)
29      (begin (if (or (stx:unary_exp_st? (stx:assign_exp_st-dest st))
30                    (cond ((obj? (stx:assign_exp_st-dest st))
31                          (and (eq? 'var (obj-kind (stx:assign_exp_st-dest st))
32                                (eq? type_array? (obj-type (stx:assign_exp_st-dest st))))
33                          (else #f)))
34                  #t
35                  (error (format "ERROR! AN INVALID ASSIGN EXPRESSION FORM AT ~a"
36                                (stx:assign_exp_st-pos st))))
37                        (map form-check (stx:assign_exp_st-src st))))
38     ((stx:logic_exp_st? st)
39      (begin (map form-check (stx:logic_exp_st-op1 st))
40            (map form-check (stx:logic_exp_st-op2 st))))
41     ((stx:rel_exp_st? st)
42      (begin (map form-check (stx:logic_exp_st-op1 st))
43            (map form-check (stx:logic_exp_st-op2 st))))
44     ((stx:alge_exp_st? st)
45      (begin (map form-check (stx:logic_exp_st-op1 st))
46            (map form-check (stx:logic_exp_st-op2 st))))
47     ((stx:spec_st? st) #t)
48     ((stx:unary_exp_st? st)
49      (cond ((obj? (stx:unary_exp_st-op st))
50            (cond ((eq? 'var (obj-kind (stx:unary_exp_st-op st)))
51                  #t)
52                  (else (error (format "ERROR! AN INVALID & FORM AT ~a"
53                                      (stx:unary_exp_st st))))))
53            (else (error (format "ERROR! AN INVALID & FORM AT ~a"
54                                (stx:unary_exp_st st))))))
56     ((stx:constant_st? st) #t)
57     ((stx:null_statement_st? st) #t)
58     ((stx:exp_in_paren_st? st)
59      (map form-check (stx:exp_in_paren_st-exp st)))
60     ((stx:if_else_st? st)
61      (begin (map form-check (stx:if_else_st-cond-exp st))
62            (map form-check (stx:if_else_st-else-state st))))
63     ((stx:while_st? st)
64      (begin (map form-check (stx:while_st-cond-exp st))
65            (map form-check (stx:while_st-statement st))))
66     ((stx:sem_return_st? st)
67      (map form-check (stx:return_st-exp st)))
68     ((stx:compound_st? st)
69      (map form-check (stx:compound_st-statement-list st)))
70     ((stx:func_st? st) #t)
71     ((obj? st) #t)
72     ((position? st) #t)
73     ;デバグ用エラー発生 (実際にはどのようなプログラムを読み込んでこの分岐には入らないはず。)
74     (else (error "ERROR! UNEXPECTED STRUCTURES IN AN ARGUMENT OF ANALY-FORM." st))))
75
76
77 ;テスト
78 ;(define p (open-input-file "test01.c"))
79 ;(port-count-lines! p)
80 ;;(sem:sem-analyze-tree (k08:parse-port p))
81 ;(analy-form (sem:sem-analyze-tree (k08:parse-port p)))

```

## リスト 9: 式の型の検査を行う処理 analy-type.rkt

```

1 #lang racket
2 (require (prefix-in k08: "kadai08.rkt"))
3 (require (prefix-in sem: "semantic-analy.rkt"))
4 (require (prefix-in stx: "mysyntax.rkt"))
5 (require parser-tools/lex
6         (prefix-in : parser-tools/lex-sre)
7         parser-tools/yacc)
8 (require "myenv.rkt")
9 (provide (all-defined-out))
10 ;myenv.rkt内で定義
11 ;(struct obj (name lev kind type)#:transparent)
12
13 (define (analy-type t)
14   (begin (map check-type t)
15         (display "OK! THIS PROGRAM IS WELL TYPED.")))
16
17 ;引数は構造体
18 ;戻り値は'well-typed
19 (define (check-type st)
20   (cond
21     ((list? st) (map check-type st))
22     ((stx:declaration_st? st)
23      (let* ((decl-obj-list (stx:declaration_st-declarator-list st))
24             (map (lambda (x)
25                   (cond
26                     ;宣言した変数が配列の時
27                     ;void型、voidポインタ型はエラー
28                     ((type_array? (obj-type x))
29                      (cond ((or (equal? 'void
30                                   (type_array-type (obj-type x)))
31                               (equal? (type_pointer 'pointer 'void)
32                                       (type_array-type (obj-type x))))
33                             (error (format "ERROR NOT WELL TYPED '~a' AT ~a"
34                                             (obj-name x)(obj-pos x))))))
35                     ;宣言した変数が配列でない時
36                     ;void型、voidポインタ型はエラー
37                     (else (cond ((equal? (type_pointer 'pointer 'void) (obj-type x))
38                                   (error (format "ERROR NOT WELL TYPED '~a' AT ~a"
39                                                 (obj-name x)(obj-pos x))))
40                                 ((equal? 'void (obj-type x))
41                                  (error (format "ERROR NOT WELL TYPED '~a' AT ~a"
42                                                 (obj-name x)(obj-pos x))))
43                                 (else 'well-typed))))))
44      decl-obj-list)
45     'well-typed))
46 ((stx:func_proto_st? st)
47  (let* ((func-declarator (stx:func_proto_st-func-declarator st))
48         (func-para-list (stx:func_declarator_st-para-list func-declarator))
49         (func-obj (stx:func_declarator_st-name func-declarator))
50         (func-type (obj-type func-obj))
51         (func-out-type (type_fun-out func-type)))
52    ;関数プロトタイプの
53    ;戻り値がvoidポインタはエラー
54    ;パラメータがvoid型、voidポインタはエラー
55    (cond ;戻り値がvoidのポインタ型であるとき
56          ((equal? (type_pointer 'pointer 'void)
57                   func-out-type)
58           (error (format "ERROR NOT WELL TYPED '~a' AT ~a"
59                           (obj-name func-obj)(obj-pos func-obj))))
60          ;パラメータががvoid型、voidポインタ型であるとき
61          ;check-type-paraはこれらの型の以上がパラメータの中に無いかどうかを判定する。
62          ((equal? 'well-typed (check-type-para func-para-list))
63           'well-typed)
64          (else 'well-typed))))
65 ((stx:func_def_st? st)
66  ;戻り値がvoidポインタはエラー
67  ;パラメータがvoid型、voidポインタはエラー
68  (let* ((func-declarator (stx:func_def_st-func-declarator st))
69         (func-para-list (stx:func_declarator_st-para-list func-declarator))
70         (func-obj (stx:func_declarator_st-name func-declarator))
71         (func-type (obj-type func-obj))
72         (func-out-type (type_fun-out func-type))
73         (func-compound-state (stx:compound_st-statement-list
74                               (stx:func_def_st-compound-state-list st)))
75         (func-compound-decl (stx:compound_st-declaration-list
76                               (stx:func_def_st-compound-state-list st))))
76    (cond ((and (equal? 'well-typed
77                      (cond ((equal? (type_pointer 'pointer 'void)

```



```

79         func-out-type)
80         (error (format "ERROR NOT WELL TYPED '~a' AT ~a"
81                        (obj-name func-obj)(obj-pos func-obj))))
82         ((equal? 'well-typed (check-type-para func-para-list))
83          'well-typed)
84         (else 'well-typed)))
85 ;begin文の前者がうまく実行されれば#tが出力される。
86 ;そうでなければ勝手にエラーで止まる。
87 (begin
88   (cond ((equal? 'nodecl func-compound-decl) 'well-typed)
89         (else (map check-type func-compound-decl)))
90   (cond ((equal? 'nostat func-compound-state) 'well-typed)
91         (else (map check-type func-compound-state))))
92   #t))
93 'well-typed)
94 (else (error (format "ERROR NOT WELL TYPED '~a' AT ~a"
95                      (obj-name func-obj)(obj-pos func-obj))))))
96 ((stx:null_statement_st? st) 'well-typed)
97 ((stx:exp_in_paren_st? st)
98  (check-type (stx:exp_in_paren_st-exp st)))
99 ((stx:sem_return_st? st)
100  (cond ((equal? (stx:sem_return_st-exp st) 'noreturn)
101         'well-typed)
102        ((type-void? (stx:sem_return_st-exp st))
103         (error (format "ERROR NOT WELL TYPED RETURN-STATEMENT AT ~a"
104                        (stx:sem_return_st-pos st))))
105        ((and (type-int? (stx:sem_return_st-exp st))
106              (equal? 'int (type_fun-out (obj-type (stx:sem_return_st-tag st)))))
107         'well-typed)
108        ((and (type-intp? (stx:sem_return_st-exp st))
109              (equal? (type_pointer 'pointer 'int)
110                      (type_fun-out (obj-type (stx:sem_return_st-tag st)))))
111         'well-typed)
112        ((and (type-intp? (stx:sem_return_st-exp st))
113              (equal? (type_pointer 'pointer 'int)
114                      (type_fun-out (obj-type (stx:sem_return_st-tag st)))))
115         'well-typed)
116        (else (error (format "ERROR NOT WELL TYPED RETURN-STATEMENT AT ~a"
117                              (stx:sem_return_st-pos st))))))
118 ((stx:if_else_st? st)
119  (cond ((and (type-int? (stx:if_else_st-cond-exp st))
120              (equal? 'well-typed (check-type (stx:if_else_st-state st)))
121              (equal? 'well-typed (check-type (stx:if_else_st-else-state st))))
122        'well-typed)
123        (else (error (format "ERROR NOT WELL TYPED IF-STATEMENT AT ~a"
124                              (stx:if_else_st-if-pos st))))))
125 ((stx:while_st? st)
126  (cond ((and (type-int? (stx:while_st-cond-exp st))
127              (equal? 'well-typed (check-type (stx:while_st-statement st))))
128        'well-typed)
129        (else (error (format "ERROR NOT WELL TYPED WHILE-STATEMENT AT ~a"
130                              (stx:while_st-pos st))))))
131 ((stx:compound_st? st)
132  (begin
133    (cond ((equal? 'nostat (stx:compound_st-statement-list st)) 'well-typed)
134          (else (map check-type (stx:compound_st-statement-list st))))
135    (cond ((equal? 'nodecl (stx:compound_st-declaration-list st)) 'well-typed)
136          (else (map check-type (stx:compound_st-declaration-list st))))
137    ;各要素はerrorか'well-typedを返すので、mapが実行されれば
138    ;必然的にlistの要素は'well-typedになっている。
139    'well-typed))
140 (else (cond ((or (equal? 'int (type st))
141                 (equal? (type_pointer 'pointer 'int) (type st))
142                 (equal? (type_pointer 'pointer (type_pointer 'pointer 'int)) (type st))
143                 (equal? 'void (type st)))
144            'well-typed)
145         (else (error "ERROR NOT WELL TYPED" st)))))
146
147
148 ;型は'int、(type_pointer 'pointer 'int)、(type_pointer 'pointer (type_pointer 'pointer 'int))
149 (define (sametype? x y)
150   (equal? (type x) (type y)))
151 (define (type-int? x)
152   (equal? 'int (type x)))
153 (define (type-intp? x)
154   (equal? (type_pointer 'pointer 'int) (type x)))
155 (define (type-intpp? x)
156   (equal? (type_pointer 'pointer (type_pointer 'pointer 'int)) (type x)))
157 (define (type-void? x)
158   (equal? 'void (type x)))

```

```

159
160 ;objのlistもしくは(list 'nopara)を受け取って
161 ;エラーすなわち
162 ;パラメーターの中にvoid型、voidポインタ型がなければ
163 ;well-typedを返す.
164 (define (check-type-para para-list)
165   (cond ((equal? 'nopara para-list) 'well-typed)
166         (else (map (lambda (x)
167                     (let ((x-type (obj-type x)))
168                       (begin (cond ((or (equal? 'void x-type)
169                                         (equal? (type_pointer 'pointer 'void) x-type))
170                                (error (format "ERROR NOT WELL TYPED '~a' AT '~a"
171                                               (obj-name x)(obj-pos x))))
172                             (else 'well-typed))
173                         'well-typed)))
174         para-list))))
175
176 (define (type st)
177   (cond
178     ((stx:sem_return_st? st) (type (stx:sem_return_st-exp st)))
179     ((stx:exp_in_paren_st? st) (type (stx:exp_in_paren_st-exp st)))
180     ((stx:assign_exp_st? st)
181      (let* ((type-dest (type (stx:assign_exp_st-dest st)))
182            (type-src (type (stx:assign_exp_st-src st))))
183        (cond ((equal? type-dest type-src)
184              type-dest)
185              (else (error (format "ERROR NOT WELL TYPED '=' AT '~a ~a"
186                                  (stx:assign_exp_st-pos st) st))))))
187     ((stx:logic_exp_st? st)
188      (let* ((type-op1 (type (stx:logic_exp_st-op1 st)))
189            (type-op2 (type (stx:logic_exp_st-op2 st))))
190        (cond ((and (type-int? type-op1)
191                  (type-int? type-op2))
192              'int)
193              (else (error (format "ERROR NOT WELL TYPED '~a' AT '~a"
194                                  (cond ((equal? 'or (stx:logic_exp_st-log-ope st)) '|')
195                                          ((equal? 'or (stx:logic_exp_st-log-ope st)) '&&))
196                                  (stx:logic_exp_st-pos st)))))))
197     ((stx:rel_exp_st? st)
198      (let* ((type-op1 (type (stx:rel_exp_st-op1 st)))
199            (type-op2 (type (stx:rel_exp_st-op2 st))))
200        (cond ((equal? type-op1 type-op2)
201              'int)
202              (else (error (format "ERROR NOT WELL TYPED '~a' AT '~a"
203                                  (cond ((equal? 'equal (stx:rel_exp_st-rel-ope st)) '==)
204                                          ((equal? 'not (stx:rel_exp_st-rel-ope st)) '!=)
205                                          ((equal? 'less (stx:rel_exp_st-rel-ope st)) '<')
206                                          ((equal? 'and_less (stx:rel_exp_st-rel-ope st)) '<=')
207                                          ((equal? 'more (stx:rel_exp_st-rel-ope st)) '>')
208                                          ((equal? 'and_more (stx:rel_exp_st-rel-ope st)) '>='))
209                                  (stx:rel_exp_st-pos st)))))))
210     ((stx:alge_exp_st? st)
211      (let* ((type-op1 (stx:alge_exp_st-op1 st))
212            (type-op2 (stx:alge_exp_st-op2 st))
213            (ope (stx:alge_exp_st-alge-ope st))
214            (pos (stx:alge_exp_st-pos st)))
215        (cond ((equal? 'add ope)
216              (cond ((and (type-int? type-op1)
217                        (type-int? type-op2))
218                    'int)
219                    ((and (type-intp? type-op1)
220                          (type-int? type-op2))
221                     (type_pointer 'pointer 'int))
222                    ((and (type-int? type-op1)
223                          (type-intp? type-op2))
224                     (type_pointer 'pointer 'int))
225                    ((and (type-intpp? type-op1)
226                          (type-int? type-op2))
227                     (type_pointer 'pointer 'int))
228                    ((and (type-int? type-op1)
229                          (type-intpp? type-op2))
230                     (type_pointer 'pointer (type_pointer 'pointer 'int)))
231                    (else (error (format "ERROR NOT WELL TYPED '+' AT '~a" pos))))))
232              ((equal? 'sub ope)
233               (cond ((and (type-intp? type-op1)
234                         (type-int? type-op2))
235                     (type_pointer 'pointer 'int))
236                     ((and (type-intpp? type-op1)
237                           (type-int? type-op2))
238                      (type-int? type-op2))))))

```

```

239         (type_pointer 'pointer (type_pointer 'pointer 'int)))
240         ((and (type-int? type-op1)
241              (type-int? type-op2)) 'int)
242         (else (error (format "ERROR NOT WELL TYPED '~a' AT ~a" pos))))))
243 ((or (equal? 'mul ope)
244      (equal? 'div ope))
245      (cond ((and (type-int? type-op1)
246                  (type-int? type-op2))
247            'int)
248            (else (error (format "ERROR NOT WELL TYPED '~a' AT ~a"
249                                (cond ((equal? 'mul ope) '*')
250                                      ((equal? 'div ope) '/))
251                                pos))))))
252 ;デバッグ用.(本来はどんなプログラムを書いてもこの分岐には入らないはず.)
253 (else (error "ERROR NOT WELL TYPED" st))))))
254 ((stx:unary_exp_st? st)
255  (let* ((mark (stx:unary_exp_st-mark st))
256         (op (stx:unary_exp_st-op st))
257         (type-op (type op))
258         (pos (stx:unary_exp_st-pos st)))
259    (cond ((equal? 'amp mark)
260          (cond ((equal? 'int op) 'int)
261                (else (error (format "ERROR NOT WELL TYPED '&' AT ~a" pos))))))
262          ((equal? 'ast mark)
263           (cond ((equal? (type_pointer 'pointer 'int) type-op) 'int)
264                 ((equal? (type_pointer 'pointer (type_pointer 'pointer 'int)) type-op)
265                  (type_pointer 'pointer 'int))
266                 (else (error (format "ERROR NOT WELL TYPED '*' AT ~a" st))))))
267          ;デバッグ用.(本来はどんなプログラムを書いてもこの分岐には入らないはず.)
268          (else (error "ERROR NOT WELL TYPED" st))))))
269 ((stx:constant_st? st) 'int)
270 ((stx:func_st? st)
271  (let* (;funcを表すobj
272        (func-ref (stx:func_st-name st))
273        ;funcを表すobj中のtype_fun
274        (func-type-fun (obj-type func-ref))
275        ;funcを表すobj中のtyp_fun内のパラメータの型のlist
276        ;もしくは'nopara
277        (func-in-list (type_fun-in func-type-fun))
278        ;type_fun中の関数の戻り値
279        ;(type_pointer 'pointer 'int)か'intか'void
280        (func-out (type_fun-out func-type-fun))
281        ;パラメータのobjのlistもしくは'nopara
282        (func-para (stx:func_st-para st))
283        (pos (obj-pos func-ref))
284        (func-name (obj-name func-ref)))
285    (cond ((equal? (map (lambda (x)
286                        (cond
287                          ((equal? 'nopara x) 'nopara)
288                          (else (type x))))
289                      func-para)
290           func-in-list)
291          (cond
292            ;関数の戻り値として許されるのは
293            ;int、intのポインタ型、void型
294            ((equal? 'int func-out) 'int)
295            ((equal? 'void func-out) 'void)
296            ((equal? (type_pointer 'pointer 'int) func-out) (type_pointer 'pointer 'int))
297            (else (error (format "ERROR NOT WELL TYPED '~a' AT ~a"
298                                func-name pos))))))
299          (else (error (format "ERROR NOT WELL TYPED '~a' AT ~a"
300                                func-name pos))))))
301 ((obj? st)
302  (let* ((type-obj (obj-type st))
303        (cond
304          ;配列型するとき
305          ((type_array? type-obj)
306           ;ポインタ型として許されるのはintのみ
307           (cond ((equal? 'int (type_array-type type-obj)) 'int)
308                 (else (error (format "ERROR NOT WELL TYPED '~a' AT ~a"
309                                     (obj-name st) (obj-pos st))))))
310          ;配列型でないとき
311          ;許されるのはint、intのポインタ型のみ
312          (else
313           (cond ((equal? 'int type-obj) 'int)
314                 ((equal? (type_pointer 'pointer 'int) type-obj) (type_pointer 'pointer 'int))
315                 (else (error (format "ERROR NOT WELL TYPED '~a' AT ~a"
316                                     (obj-name st) (obj-pos st))))))))))
317
318

```

```

319
320
321
322 ;テスト
323 ;(define p100 (open-input-file "kadai01.c"))
324 ;(port-count-lines! p100)
325 ;;(sem:sem-analyze-tree (k08:parse-port p100))
326 ;(analy-type (sem:sem-analyze-tree (k08:parse-port p100)))

```

### 3 実行結果

これらのプログラムを使って次のテストプログラムを解析した.

リスト 10: テストプログラム

```

1  int comp_num(int a, int b);
2  int sort_array[8];
3
4
5  int comp_num(int a, int b){
6      if(a > b) return 0;
7      if(a < b) return 1;
8      if(a == b) return 2;
9  }
10
11
12 int main(){
13     int i;
14     int j;
15     int h;
16
17     sort_array[0] = 6;
18     sort_array[1] = 4;
19     sort_array[2] = 2;
20     sort_array[3] = 5;
21     sort_array[4] = 7;
22     sort_array[5] = 8;
23     sort_array[6] = 1;
24     sort_array[7] = 3;
25
26
27     for(j = 8; j > 1; j = (j-1)){
28         for(i = 0; i < (j-1); i = (i+1)){
29             if(comp_num(sort_array[i+1], sort_array[i])){
30                 h = sort_array[i+1];
31                 sort_array[i+1] = sort_array[i];
32                 sort_array[i] = h;
33             }
34         }
35     }
36     i = 0;
37     while(8 > i){
38         if(i != 7){
39             print(sort_array[i]);
40         }
41         else(print(sort_array[7]));
42         i = (i+1);
43     }
44 }
45
46

```

実行結果は次のようになった.

リスト 11: 課題 8 の実行結果

```

1 #lang racket
2 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;以下が課題8の実行結果です;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
3 (list*
4   (func_proto_st
5    (spec_st 'void 'print-proto)
6    (func_declarator_st

```

```

7      'print
8      (para_declaration_st (spec_st 'int 'print-PROTO) (id_st 'v 'print-PROTO))
9      'print-PROTO))
10     (cons
11       (cons
12         (func_PROTO_st
13           (spec_st 'int (position 1 1 0))
14           (func_declarator_st
15             'comp_num
16             (cons
17               (para_declaration_st (spec_st 'int (position 14 1 13)) (id_st 'a (position 18 1 17)))
18               (para_declaration_st (spec_st 'int (position 21 1 20)) (id_st 'b (position 25 1 24)))
19               (position 5 1 4)))
20         (declaration_st
21           (spec_st 'int (position 29 2 0))
22           (declarator_st (array_st 'sort_array 8 (position 33 2 4))))))
23     (func_def_st
24       (spec_st 'int (position 50 5 0))
25       (func_declarator_st
26         'comp_num
27         (cons
28           (para_declaration_st (spec_st 'int (position 63 5 13)) (id_st 'a (position 67 5 17)))
29           (para_declaration_st (spec_st 'int (position 70 5 20)) (id_st 'b (position 74 5 24)))
30           (position 54 5 4))
31       (compound_stmt
32         (cons
33           (cons
34             (if_else_st
35               (rel_exp_st 'more (id_st 'a (position 85 6 7)) (id_st 'b (position 89 6 11)) (position 87 6 9))
36               (return_st (constant_st 0 (position 99 6 21)) (position 92 6 14))
37               (null_statement_st 'null)
38               (position 82 6 4)
39               'syntax-sygar)
40             (if_else_st
41               (rel_exp_st
42                 'less
43                 (id_st 'a (position 109 7 7))
44                 (id_st 'b (position 113 7 11))
45                 (position 111 7 9))
46               (return_st (constant_st 1 (position 123 7 21)) (position 116 7 14))
47               (null_statement_st 'null)
48               (position 106 7 4)
49               'syntax-sygar))
50             (if_else_st
51               (rel_exp_st
52                 'equal
53                 (id_st 'a (position 133 8 7))
54                 (id_st 'b (position 138 8 12))
55                 (position 135 8 9))
56               (return_st (constant_st 2 (position 148 8 22)) (position 141 8 15))
57               (null_statement_st 'null)
58               (position 130 8 4)
59               'syntax-sygar))))))
60     (func_def_st
61       (spec_st 'int (position 155 12 0))
62       (func_declarator_null_st 'main (position 159 12 4))
63     (compound_stmt
64       (cons
65         (cons
66           (declaration_st
67             (spec_st 'int (position 171 13 4))
68             (declarator_st (id_st 'i (position 175 13 8))))
69           (declaration_st
70             (spec_st 'int (position 182 14 4))
71             (declarator_st (id_st 'j (position 186 14 8))))))
72       (declaration_st
73         (spec_st 'int (position 193 15 4))
74         (declarator_st (id_st 'h (position 197 15 8))))
75     (cons
76       (cons
77         (cons
78           (cons
79             (cons
80               (cons
81                 (cons
82                   (cons
83                     (cons
84                       (assign_exp_st
85                         (unary_exp_st

```

```

87         'ast
88         (exp_in_paren_st
89         (alge_exp_st
90         'add
91         (id_st 'sort_array (position 205 17 4))
92         (constant_st 0 (position 216 17 15))
93         'syntax-sugar))
94         'syntax-sugar)
95         (constant_st 6 (position 221 17 20))
96         (position 219 17 18))
97     (assign_exp_st
98     (unary_exp_st
99     'ast
100     (exp_in_paren_st
101     (alge_exp_st
102     'add
103     (id_st 'sort_array (position 228 18 4))
104     (constant_st 1 (position 239 18 15))
105     'syntax-sugar))
106     'syntax-sugar)
107     (constant_st 4 (position 244 18 20))
108     (position 242 18 18)))
109 (assign_exp_st
110 (unary_exp_st
111 'ast
112 (exp_in_paren_st
113 (alge_exp_st
114 'add
115 (id_st 'sort_array (position 251 19 4))
116 (constant_st 2 (position 262 19 15))
117 'syntax-sugar))
118 'syntax-sugar)
119 (constant_st 2 (position 267 19 20))
120 (position 265 19 18)))
121 (assign_exp_st
122 (unary_exp_st
123 'ast
124 (exp_in_paren_st
125 (alge_exp_st
126 'add
127 (id_st 'sort_array (position 274 20 4))
128 (constant_st 3 (position 285 20 15))
129 'syntax-sugar))
130 'syntax-sugar)
131 (constant_st 5 (position 290 20 20))
132 (position 288 20 18)))
133 (assign_exp_st
134 (unary_exp_st
135 'ast
136 (exp_in_paren_st
137 (alge_exp_st
138 'add
139 (id_st 'sort_array (position 297 21 4))
140 (constant_st 4 (position 308 21 15))
141 'syntax-sugar))
142 'syntax-sugar)
143 (constant_st 7 (position 313 21 20))
144 (position 311 21 18)))
145 (assign_exp_st
146 (unary_exp_st
147 'ast
148 (exp_in_paren_st
149 (alge_exp_st
150 'add
151 (id_st 'sort_array (position 320 22 4))
152 (constant_st 5 (position 331 22 15))
153 'syntax-sugar))
154 'syntax-sugar)
155 (constant_st 8 (position 336 22 20))
156 (position 334 22 18)))
157 (assign_exp_st
158 (unary_exp_st
159 'ast
160 (exp_in_paren_st
161 (alge_exp_st
162 'add
163 (id_st 'sort_array (position 343 23 4))
164 (constant_st 6 (position 354 23 15))
165 'syntax-sugar))
166 'syntax-sugar)

```

```

167         (constant_st 1 (position 359 23 20))
168         (position 357 23 18)))
169 (assign_exp_st
170 (unary_exp_st
171   'ast
172   (exp_in_paren_st
173     (alge_exp_st
174       'add
175       (id_st 'sort_array (position 366 24 4))
176       (constant_st 7 (position 377 24 15))
177       'syntax-sugar))
178   'syntax-sugar)
179 (constant_st 3 (position 382 24 20))
180 (position 380 24 18)))
181 (compound_sta_st
182 (cons
183   (assign_exp_st
184     (id_st 'j (position 399 27 8))
185     (constant_st 8 (position 403 27 12))
186     (position 401 27 10))
187   (while_st
188     (rel_exp_st
189       'more
190       (id_st 'j (position 406 27 15))
191       (constant_st 1 (position 410 27 19))
192       (position 408 27 17))
193     (compound_sta_st
194       (cons
195         (compound_sta_st
196           (compound_sta_st
197             (cons
198               (assign_exp_st
199                 (id_st 'i (position 437 28 12))
200                 (constant_st 0 (position 441 28 16))
201                 (position 439 28 14))
202               (while_st
203                 (rel_exp_st
204                   'less
205                   (id_st 'i (position 444 28 19))
206                   (exp_in_paren_st
207                     (alge_exp_st
208                       'sub
209                       (id_st 'j (position 449 28 24))
210                       (constant_st 1 (position 451 28 26))
211                       (position 450 28 25)))
212                   (position 446 28 21))
213                 (compound_sta_st
214                   (cons
215                     (compound_sta_st
216                       (if_else_st
217                         (func_st
218                           'comp_num
219                           (cons
220                             (unary_exp_st
221                               'ast
222                               (exp_in_paren_st
223                                 (alge_exp_st
224                                   'add
225                                   (id_st 'sort_array (position 491 29 24))
226                                   (alge_exp_st
227                                     'add
228                                     (id_st 'i (position 502 29 35))
229                                     (constant_st 1 (position 504 29 37))
230                                     (position 503 29 36))
231                                   'syntax-sugar))
232                                 'syntax-sugar)
233                               (unary_exp_st
234                                 'ast
235                                 (exp_in_paren_st
236                                   (alge_exp_st
237                                     'add
238                                     (id_st 'sort_array (position 507 29 40))
239                                     (id_st 'i (position 518 29 51))
240                                     'syntax-sugar))
241                                   'syntax-sugar)))
242                               (compound_sta_st
243                                 (cons
244                                   (cons
245                                     (assign_exp_st
246                                       (id_st 'h (position 540 30 16))

```

```

247         (unary_exp_st
248         'ast
249         (exp_in_paren_st
250         (alge_exp_st
251         'add
252         (id_st 'sort_array (position 544 30 20))
253         (alge_exp_st
254         'add
255         (id_st 'i (position 555 30 31))
256         (constant_st 1 (position 557 30 33))
257         (position 556 30 32))
258         'syntax-sugar))
259         'syntax-sugar)
260         (position 542 30 18))
261     (assign_exp_st
262     (unary_exp_st
263     'ast
264     (exp_in_paren_st
265     (alge_exp_st
266     'add
267     (id_st 'sort_array (position 577 31 16))
268     (alge_exp_st
269     'add
270     (id_st 'i (position 588 31 27))
271     (constant_st 1 (position 590 31 29))
272     (position 589 31 28))
273     'syntax-sugar))
274     'syntax-sugar)
275     (unary_exp_st
276     'ast
277     (exp_in_paren_st
278     (alge_exp_st
279     'add
280     (id_st 'sort_array (position 595 31 34))
281     (id_st 'i (position 606 31 45))
282     'syntax-sugar))
283     'syntax-sugar)
284     (position 593 31 32)))
285     (assign_exp_st
286     (unary_exp_st
287     'ast
288     (exp_in_paren_st
289     (alge_exp_st
290     'add
291     (id_st 'sort_array (position 626 32 16))
292     (id_st 'i (position 637 32 27))
293     'syntax-sugar))
294     'syntax-sugar)
295     (id_st 'h (position 642 32 32))
296     (position 640 32 30)))
297     (null_statement_st 'null)
298     (position 479 29 12)
299     'syntax-sugar))
300     (assign_exp_st
301     (id_st 'i (position 455 28 30))
302     (exp_in_paren_st
303     (alge_exp_st
304     'add
305     (id_st 'i (position 460 28 35))
306     (constant_st 1 (position 462 28 37))
307     (position 461 28 36))
308     (position 457 28 32)))
309     'syntax-sugar)))
310     (assign_exp_st
311     (id_st 'j (position 413 27 22))
312     (exp_in_paren_st
313     (alge_exp_st
314     'sub
315     (id_st 'j (position 418 27 27))
316     (constant_st 1 (position 420 27 29))
317     (position 419 27 28))
318     (position 415 27 24)))
319     'syntax-sugar)))
320     (assign_exp_st
321     (id_st 'i (position 692 37 4))
322     (constant_st 0 (position 696 37 8))
323     (position 694 37 6)))
324     (while_st
325     (rel_exp_st
326     'more

```



```

327      (constant_st 8 (position 709 38 10))
328      (id_st 'i (position 713 38 14))
329      (position 711 38 12))
330  (compound_sta_st
331    (cons
332      (if_else_st
333        (rel_exp_st
334          'not
335          (id_st 'i (position 728 39 11))
336          (constant_st 7 (position 733 39 16))
337          (position 730 39 13))
338        (compound_sta_st
339          (func_st
340            'print
341            (unary_exp_st
342              'ast
343              (exp_in_paren_st
344                (alge_exp_st
345                  'add
346                  (id_st 'sort_array (position 751 40 14))
347                  (id_st 'i (position 762 40 25))
348                  'syntax-sugar))
349                'syntax-sugar)))
350            (exp_in_paren_st
351              (func_st
352                'print
353                (unary_exp_st
354                  'ast
355                  (exp_in_paren_st
356                    (alge_exp_st
357                      'add
358                      (id_st 'sort_array (position 796 42 19))
359                      (constant_st 7 (position 807 42 30))
360                      'syntax-sugar))
361                    'syntax-sugar)))
362                (position 725 39 8)
363                (position 785 42 8))
364              (assign_exp_st
365                (id_st 'i (position 821 43 8))
366                (exp_in_paren_st
367                  (alge_exp_st
368                    'add
369                    (id_st 'i (position 826 43 13))
370                    (constant_st 1 (position 828 43 15))
371                    (position 827 43 14))
372                    (position 823 43 10))))
373                (position 703 38 4))))))
374
375
376 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;以下が意味解析の実行結果です;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
377 OK! CRRECT PARAMETERS OF FUNCTION PROTOTYPE
378 OK! CRRECT FUNCTION PROTOTYPE OF 'comp_num'.
379 OK! CRRECT PARAMETERS OF FUNCTION PROTOTYPE
380 OK! CRRECT FUNCTION PROTOTYPE OF 'comp_num'.
381 OK! CRRECT FUNCTION PROTOTYPE OF 'comp_num'.
382 OK! CRRECT FUNCTION PROTOTYPE OF 'comp_num'.
383 OK! CRRECT PARAMETERS OF FUNCTION PROTOTYPE
384 OK! NO DECLARATIONS IN COMONUND STATEMENT
385 OK! CRRECT FUNCTION PROTOTYPE OF 'main'.
386 OK! CRRECT FUNCTION PROTOTYPE OF 'main'.
387 OK! CRRECT FUNCTION PROTOTYPE OF 'main'.
388 OK! CRRECT FUNCTION PROTOTYPE OF 'main'.
389 OK! CRRECT PARAMETERS OF FUNCTION PROTOTYPE.
390 OK! CORRENCT DECLARATIONS IN COMONUND STATEMENT!
391 OK! NO DECLARATIONS IN COMONUND STATEMENT
392 OK! NO DECLARATIONS IN COMONUND STATEMENT
393 OK! NO DECLARATIONS IN COMONUND STATEMENT
394 OK! NO DECLARATIONS IN COMONUND STATEMENT
395 OK! NO DECLARATIONS IN COMONUND STATEMENT
396 OK! NO DECLARATIONS IN COMONUND STATEMENT
397 OK! NO DECLARATIONS IN COMONUND STATEMENT
398 OK! NO DECLARATIONS IN COMONUND STATEMENT
399 OK! NO DECLARATIONS IN COMONUND STATEMENT
400 (list
401   (func_proto_st
402     (spec_st 'void 'print-pto)
403     (func_declarator_st
404       (obj 'print 0 'proto (type_fun 'fun 'void '(int)) 'print-pto)
405       (list (obj 'v 1 'parm 'int 'print-pto))
406       'print-pto))

```

```

407 (func_proto_st
408 (spec_st 'int (position 1 1 0))
409 (func_declarator_st
410 (obj 'comp_num 0 'proto (type_fun 'fun 'int '(int int)) (position 5 1 4))
411 (list (obj 'a 1 'parm 'int (position 18 1 17)) (obj 'b 1 'parm 'int (position 25 1 24)))
412 (position 5 1 4)))
413 (declaration_st
414 (spec_st 'int (position 29 2 0))
415 (list (obj 'sort_array 0 'var (type_array 'int 8) (position 33 2 4))))
416 (func_def_st
417 (spec_st 'int (position 50 5 0))
418 (func_declarator_st
419 (obj 'comp_num 0 'fun (type_fun 'fun 'int '(int int)) (position 54 5 4))
420 (list (obj 'a 1 'parm 'int (position 67 5 17)) (obj 'b 1 'parm 'int (position 74 5 24)))
421 (position 54 5 4))
422 (compound_st
423 'nodecl
424 (list
425 (if_else_st
426 (rel_exp_st
427 'more
428 (obj 'a 1 'parm 'int (position 67 5 17))
429 (obj 'b 1 'parm 'int (position 74 5 24))
430 (position 87 6 9))
431 (sem_return_st
432 (constant_st 0 (position 99 6 21))
433 (position 92 6 14)
434 (obj 'comp_num 0 'fun (type_fun 'fun 'int '(int int)) (position 54 5 4)))
435 (null_statement_st 'null)
436 (position 82 6 4)
437 'syntax-sygar)
438 (if_else_st
439 (rel_exp_st
440 'less
441 (obj 'a 1 'parm 'int (position 67 5 17))
442 (obj 'b 1 'parm 'int (position 74 5 24))
443 (position 111 7 9))
444 (sem_return_st
445 (constant_st 1 (position 123 7 21))
446 (position 116 7 14)
447 (obj 'comp_num 0 'fun (type_fun 'fun 'int '(int int)) (position 54 5 4)))
448 (null_statement_st 'null)
449 (position 106 7 4)
450 'syntax-sygar)
451 (if_else_st
452 (rel_exp_st
453 'equal
454 (obj 'a 1 'parm 'int (position 67 5 17))
455 (obj 'b 1 'parm 'int (position 74 5 24))
456 (position 135 8 9))
457 (sem_return_st
458 (constant_st 2 (position 148 8 22))
459 (position 141 8 15)
460 (obj 'comp_num 0 'fun (type_fun 'fun 'int '(int int)) (position 54 5 4)))
461 (null_statement_st 'null)
462 (position 130 8 4)
463 'syntax-sygar))))))
464 (func_def_st
465 (spec_st 'int (position 155 12 0))
466 (func_declarator_st
467 (obj 'main 0 'fun (type_fun 'fun 'int 'nopara) (position 159 12 4))
468 'nopara
469 (position 159 12 4))
470 (compound_st
471 (list
472 (declaration_st
473 (spec_st 'int (position 171 13 4))
474 (list (obj 'i 2 'var 'int (position 175 13 8))))
475 (declaration_st
476 (spec_st 'int (position 182 14 4))
477 (list (obj 'j 2 'var 'int (position 186 14 8))))
478 (declaration_st
479 (spec_st 'int (position 193 15 4))
480 (list (obj 'h 2 'var 'int (position 197 15 8))))))
481 (list
482 (assign_exp_st
483 (obj 'sort_array 0 'var (type_array 'int (constant_st 0 (position 216 17 15))) (position 33 2 4))
484 (constant_st 6 (position 221 17 20))
485 (position 219 17 18))
486 (assign_exp_st

```

```

487 (obj 'sort_array 0 'var (type_array 'int (constant_st 1 (position 239 18 15))) (position 33 2 4))
488 (constant_st 4 (position 244 18 20))
489 (position 242 18 18))
490 (assign_exp_st
491 (obj 'sort_array 0 'var (type_array 'int (constant_st 2 (position 262 19 15))) (position 33 2 4))
492 (constant_st 2 (position 267 19 20))
493 (position 265 19 18))
494 (assign_exp_st
495 (obj 'sort_array 0 'var (type_array 'int (constant_st 3 (position 285 20 15))) (position 33 2 4))
496 (constant_st 5 (position 290 20 20))
497 (position 288 20 18))
498 (assign_exp_st
499 (obj 'sort_array 0 'var (type_array 'int (constant_st 4 (position 308 21 15))) (position 33 2 4))
500 (constant_st 7 (position 313 21 20))
501 (position 311 21 18))
502 (assign_exp_st
503 (obj 'sort_array 0 'var (type_array 'int (constant_st 5 (position 331 22 15))) (position 33 2 4))
504 (constant_st 8 (position 336 22 20))
505 (position 334 22 18))
506 (assign_exp_st
507 (obj 'sort_array 0 'var (type_array 'int (constant_st 6 (position 354 23 15))) (position 33 2 4))
508 (constant_st 1 (position 359 23 20))
509 (position 357 23 18))
510 (assign_exp_st
511 (obj 'sort_array 0 'var (type_array 'int (constant_st 7 (position 377 24 15))) (position 33 2 4))
512 (constant_st 3 (position 382 24 20))
513 (position 380 24 18))
514 (compound_st
515 'nodecl
516 (list
517 (assign_exp_st
518 (obj 'j 2 'var 'int (position 186 14 8))
519 (constant_st 8 (position 403 27 12))
520 (position 401 27 10))
521 (while_st
522 (rel_exp_st
523 'more
524 (obj 'j 2 'var 'int (position 186 14 8))
525 (constant_st 1 (position 410 27 19))
526 (position 408 27 17))
527 (compound_st
528 'nodecl
529 (list
530 (compound_st
531 'nodecl
532 (list
533 (compound_st
534 'nodecl
535 (list
536 (assign_exp_st
537 (obj 'i 2 'var 'int (position 175 13 8))
538 (constant_st 0 (position 441 28 16))
539 (position 439 28 14))
540 (while_st
541 (rel_exp_st
542 'less
543 (obj 'i 2 'var 'int (position 175 13 8))
544 (exp_in_paren_st
545 (alge_exp_st
546 'sub
547 (obj 'j 2 'var 'int (position 186 14 8))
548 (constant_st 1 (position 451 28 26))
549 (position 450 28 25)))
550 (position 446 28 21))
551 (compound_st
552 'nodecl
553 (list
554 (compound_st
555 'nodecl
556 (list
557 (if_else_st
558 (func_st
559 (obj 'comp_num 0 'fun (type_fun 'fun 'int '(int int)) (position 54 5 4))
560 (list
561 (obj
562 'sort_array
563 0
564 'var
565 (type_array
566 'int

```

```

567         (alge_exp_st
568         'add
569         (id_st 'i (position 502 29 35))
570         (constant_st 1 (position 504 29 37))
571         (position 503 29 36)))
572     (position 33 2 4))
573     (obj
574     'sort_array
575     0
576     'var
577     (type_array 'int (id_st 'i (position 518 29 51)))
578     (position 33 2 4)))
579     (compound_st
580     'nodecl
581     (list
582     (assign_exp_st
583     (obj 'h 2 'var 'int (position 197 15 8))
584     (obj
585     'sort_array
586     0
587     'var
588     (type_array
589     'int
590     (alge_exp_st
591     'add
592     (id_st 'i (position 555 30 31))
593     (constant_st 1 (position 557 30 33))
594     (position 556 30 32)))
595     (position 33 2 4))
596     (position 542 30 18))
597     (assign_exp_st
598     (obj
599     'sort_array
600     0
601     'var
602     (type_array
603     'int
604     (alge_exp_st
605     'add
606     (id_st 'i (position 588 31 27))
607     (constant_st 1 (position 590 31 29))
608     (position 589 31 28)))
609     (position 33 2 4))
610     (obj
611     'sort_array
612     0
613     'var
614     (type_array 'int (id_st 'i (position 606 31 45)))
615     (position 33 2 4))
616     (position 593 31 32))
617     (assign_exp_st
618     (obj
619     'sort_array
620     0
621     'var
622     (type_array 'int (id_st 'i (position 637 32 27)))
623     (position 33 2 4))
624     (obj 'h 2 'var 'int (position 197 15 8))
625     (position 640 32 30))))
626     (null_statement_st 'null)
627     (position 479 29 12)
628     'syntax-sygar)))
629     (assign_exp_st
630     (obj 'i 2 'var 'int (position 175 13 8))
631     (exp_in_paren_st
632     (alge_exp_st
633     'add
634     (obj 'i 2 'var 'int (position 175 13 8))
635     (constant_st 1 (position 462 28 37))
636     (position 461 28 36))
637     (position 457 28 32))))
638     'syntax-sugar))))))
639     (assign_exp_st
640     (obj 'j 2 'var 'int (position 186 14 8))
641     (exp_in_paren_st
642     (alge_exp_st
643     'sub
644     (obj 'j 2 'var 'int (position 186 14 8))
645     (constant_st 1 (position 420 27 29))
646     (position 419 27 28)))

```

```

647         (position 415 27 24))))
648     'syntax-sugar)))
649 (assign_exp_st
650 (obj 'i 2 'var 'int (position 175 13 8))
651 (constant_st 0 (position 696 37 8))
652 (position 694 37 6))
653 (while_st
654 (rel_exp_st
655 'more
656 (constant_st 8 (position 709 38 10))
657 (obj 'i 2 'var 'int (position 175 13 8))
658 (position 711 38 12))
659 (compound_st
660 'nodecl
661 (list
662 (if_else_st
663 (rel_exp_st
664 'not
665 (obj 'i 2 'var 'int (position 175 13 8))
666 (constant_st 7 (position 733 39 16))
667 (position 730 39 13))
668 (compound_st
669 'nodecl
670 (list
671 (func_st
672 (obj 'print 0 'proto (type_fun 'fun 'void '(int)) 'print-proto)
673 (list
674 (obj
675 'sort_array
676 0
677 'var
678 (type_array 'int (id_st 'i (position 762 40 25)))
679 (position 33 2 4))))))
680 (exp_in_paren_st
681 (func_st
682 (obj 'print 0 'proto (type_fun 'fun 'void '(int)) 'print-proto)
683 (list
684 (obj
685 'sort_array
686 0
687 'var
688 (type_array 'int (constant_st 7 (position 807 42 30)))
689 (position 33 2 4))))
690 (position 725 39 8)
691 (position 785 42 8))
692 (assign_exp_st
693 (obj 'i 2 'var 'int (position 175 13 8))
694 (exp_in_paren_st
695 (alge_exp_st
696 'add
697 (obj 'i 2 'var 'int (position 175 13 8))
698 (constant_st 1 (position 828 43 15))
699 (position 827 43 14)))
700 (position 823 43 10))))
701 (position 703 38 4))))))
702
703
704 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;以下が形検査の実行結果です;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
705 OK! CRRECT PARAMETERS OF FUNCTION PROTOTYPE
706 OK! CRRECT FUNCTION PROTOTYPE OF 'comp_num'.
707 OK! CRRECT PARAMETERS OF FUNCTION PROTOTYPE
708 OK! CRRECT FUNCTION PROTOTYPE OF 'comp_num'.
709 OK! CRRECT FUNCTION PROTOTYPE OF 'comp_num'.
710 OK! CRRECT FUNCTION PROTOTYPE OF 'comp_num'.
711 OK! CRRECT PARAMETERS OF FUNCTION PROTOTYPE
712 OK! NO DECLARATIONS IN COMONUND STATEMENT
713 OK! CRRECT FUNCTION PROTOTYPE OF 'main'.
714 OK! CRRECT FUNCTION PROTOTYPE OF 'main'.
715 OK! CRRECT FUNCTION PROTOTYPE OF 'main'.
716 OK! CRRECT FUNCTION PROTOTYPE OF 'main'.
717 OK! CRRECT PARAMETERS OF FUNCTION PROTOTYPE.
718 OK! CORRENCT DECLARATIONS IN COMONUND STATEMENT!
719 OK! NO DECLARATIONS IN COMONUND STATEMENT
720 OK! NO DECLARATIONS IN COMONUND STATEMENT
721 OK! NO DECLARATIONS IN COMONUND STATEMENT
722 OK! NO DECLARATIONS IN COMONUND STATEMENT
723 OK! NO DECLARATIONS IN COMONUND STATEMENT
724 OK! NO DECLARATIONS IN COMONUND STATEMENT
725 OK! NO DECLARATIONS IN COMONUND STATEMENT
726 OK! NO DECLARATIONS IN COMONUND STATEMENT

```

```

727 OK! NO DECLARATIONS IN COMONUND STATEMENT
728 OK! THIS PROGRAM IS IN A CORRECT FORM.
729
730 ;;;;;;;;;;;;;;以下が型検査の実行結果です;;;;;;;;;;;;;
731 OK! CRRECT PARAMETERS OF FUNCTION PROTOTYPE
732 OK! CRRECT FUNCTION PROTOTYPE OF 'comp_num'.
733 OK! CRRECT PARAMETERS OF FUNCTION PROTOTYPE
734 OK! CRRECT FUNCTION PROTOTYPE OF 'comp_num'.
735 OK! CRRECT FUNCTION PROTOTYPE OF 'comp_num'.
736 OK! CRRECT FUNCTION PROTOTYPE OF 'comp_num'.
737 OK! CRRECT PARAMETERS OF FUNCTION PROTOTYPE
738 OK! NO DECLARATIONS IN COMONUND STATEMENT
739 OK! CRRECT FUNCTION PROTOTYPE OF 'main'.
740 OK! CRRECT FUNCTION PROTOTYPE OF 'main'.
741 OK! CRRECT FUNCTION PROTOTYPE OF 'main'.
742 OK! CRRECT FUNCTION PROTOTYPE OF 'main'.
743 OK! CRRECT PARAMETERS OF FUNCTION PROTOTYPE.
744 OK! CORRENCT DECLARATIONS IN COMONUND STATEMENT!
745 OK! NO DECLARATIONS IN COMONUND STATEMENT
746 OK! NO DECLARATIONS IN COMONUND STATEMENT
747 OK! NO DECLARATIONS IN COMONUND STATEMENT
748 OK! NO DECLARATIONS IN COMONUND STATEMENT
749 OK! NO DECLARATIONS IN COMONUND STATEMENT
750 OK! NO DECLARATIONS IN COMONUND STATEMENT
751 OK! NO DECLARATIONS IN COMONUND STATEMENT
752 OK! NO DECLARATIONS IN COMONUND STATEMENT
753 OK! NO DECLARATIONS IN COMONUND STATEMENT
754 OK! THIS PROGRAM IS WELL TYPED.
755 >

```

## 4 感想

今回の実験では前回作成したソースコードを文字列として受け取りそれを抽象構文木に変換して構造体の形で表示するというプログラムを改良し、シンタックスシュガーを識別して抽象構文木を作り直し、それを意味解析、形解析、型解析を適用するという処理を行うプログラムを作成したが、予想以上に苦労した。そもそもプログラム自体は提出期限内に完成したものだと思い込んでいたとバグが見つかり、大幅な改変をせざるを得なくなってしまった。その際に自分の書いたコード中で何を行っている処理なのかをすぐに把握できずかなり苦労したので、これからは let 文で使う値を先に全て取り出してコメントに書いておくなど、工夫の余地も大きいと感じた。