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 1

工学部情報学科 3 回生 1029255242
勝見久央

作成日: 平成 27 年 6 月 13 日

1　課題 1

この課題では Small C の文法に従ってソーティングプログラムを作成した. なお、Small C では BNF の記述から C 言語と異なって、関数内部で変数宣言を全て行ってから変数への値の代入を行わないとエラーが発生することに留意する.

1.1　課題 1 の回答

コードは次のようになった.

リスト 1: Small C で記述したバブルソードプログラム

```
int comp_num(int a, int b);
int sort_array[8];
int main(){
    int i;
    int j;
    int h;

    sort_array[0] = 6;
    sort_array[1] = 4;
    sort_array[2] = 2;
    sort_array[3] = 5;
    sort_array[4] = 7;
    sort_array[5] = 8;
    sort_array[6] = 1;
    sort_array[7] = 3;


    for(j = 8; j > 1; j = (j-1)){
        for(i = 0; i < (j-1); i = (i+1)){
            if(comp_num(sort_array[i+1],sort_array[i])){
                h = sort_array[i+1];
                sort_array[i+1] = sort_array[i];
                sort_array[i] = h;
            }

        }
    }
    i = 0;
    while(8 > i){
        if(i != 7){
        print(sort_array[i]);
        }
        else(print(sort_array[7]));
        i = (i+1);
    }

}

int comp_num(int a, int b){
    if(a > b) return 0;
    if(a < b) return 1;
    if(a == b) return 2;
}
```

2　課題 5

この課題では Small C の構文解析器を作成した. この構文解析器はプログラムを受け取って抽象構文木を主な終端記号の位置情報を含んだ形で返す.

2.1　課題 5 の回答

コードは次のようになった.

リスト 2: 構文解析器のプログラム

```
#lang racket
(require parser-tools/lex
         (prefix-in : parser-tools/lex-sre)
         parser-tools/yacc)
(provide (all-defined-out))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;データ型定義;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(struct declaration_st (type-spec declarator-list)#:transparent)
;例) (int (id1 id2 *id3))

(struct func_declarator_st (name para-list)#:transparent)
(struct func_declarator_null_st (name)#:transparent)
(struct func_declarator_ast_st (name para-list)#:transparent)
(struct func_declarator_ast_null_st (name)#:transparent)
;例)nameは関数名、para-listは引数のリスト.

(struct func_proto_st (type-spec func-declarator-st)#:transparent)
(struct func_def_st (type-spec func-declarator-st compound-state-list)#:transparent)
;例) (int funcname1 (id1 id2 *id3) compound_list)
;compound_listの中身は構造体declaration_listとstatement_listからなる.

(struct declarator_st (var)#:transparent)
(struct declarator_ast_st (var)#:transparent)
;declaratorの定義

;declaration_listは構造体declarationがリストになったもの.以下で定義.
;(struct declaration_st2 (type-spec para-list)#:transparent);;;;;;
;例) (int (id1 id2 *id3 id4[5]))

(struct para_declaration_st (type-spec para)#:transparent)
;例) (int id1)

(struct exp_st (exp)#:transparent)
;式を格納する構造体.

(struct assign_exp_st (dest src pos)#:transparent)
;例)代入を表す構造体. x = 3なら(x 3)

(struct logic_exp_st (log-ope op1 op2 pos)#:transparent)
;例) (or a b)もしくは(and a b)

(struct rel_exp_st (rel-ope op1 op2 pos)#:transparent)
;例) rel_opeは'equal 'not 'less 'and_less 'more 'and_moreで (less a b)など

(struct alge_exp_st (alge-ope op1 op2 pos)#:transparent)
;例) alge_opeは'add 'sub 'mul 'divで('add a b)など

(struct id_st (name pos)#:transparent)
(struct id_ast_st (name pos)#:transparent)
```

```
;終端記号 identifierを表す構造体.

(struct array_st (name num pos)#:transparent);宣言時.posはnameの位置.
(struct array_var_st (name num pos)#:transparent);式の中で用いる.posはnameの位置.
;終端記号となりうる配列を表す構造体.

(struct spec_st (type pos)#:transparent)
;データ型を表す構造体

(struct unary_exp_st (mark op pos)#:transparent)
;postfix_expを表す構造体.markは'minus、'ast、'amp

(struct constant_st (cons pos)#:transparent)
;定数を表す構造体.

(struct null_statement_st (null)#:transparent)
;セミコロンのみからなるstatementを表す構造体

(struct exp_with_semi_st (exp)#:transparent)
;expressionとセミコロンからなる式を表す構造体.

(struct exp_in_paren_st (exp)#:transparent)
;()で囲まれたexpression

(struct if_st (cond-exp state pos)#:transparent);else無し.posはifの位置.
(struct if_else_st (cond-exp state else-state if-pos else-pos)#:transparent);elseあり
;if文を表す構造体.

(struct while_st (cond-exp statement pos)#:transparent);posはwhileの位置.
;while文を表す構造体.

(struct for_0_st (cond-exp1 cond-exp2 cond-exp3 statement pos)#:transparent)
(struct for_1_st (cond-exp1 cond-exp2 statement pos)#:transparent)
(struct for_2_st (cond-exp1 cond-exp2 statement pos)#:transparent)
(struct for_3_st (cond-exp1 cond-exp2 statement pos)#:transparent)
(struct for_4_st (cond-exp1 statement pos)#:transparent)
(struct for_5_st (cond-exp1 statement pos)#:transparent)
(struct for_6_st (cond-exp1 statement pos)#:transparent)
(struct for_7_st (statement pos)#:transparent)
;for文を表す構造体.0〜7の数字によってnullの位置が構文木作成の時点でわかる.

(struct return_st (exp pos)#:transparent);posはreturnの位置.
(struct return_null_st (exp pos)#:transparent)
;return文を表す構造体.

(struct compound_st (declaration-list statement-list)#:transparent)
(struct compound_dec_st (declaration-list)#:transparent)
(struct compound_sta_st (statement-list)#:transparent)
(struct compound_null_st (null)#:transparent)
;compound_statementを表す構造体.

(struct func_st (name para)#:transparent)
(struct func_nopara_st (name)#:transparent)
;関数呼び出しを表す構造体
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(define-empty-tokens tokens-without-value
  (+ * & - / =
     l_small_paren r_small_paren ;()
     l_mid_paren r_mid_paren ;[]
     l_big_paren r_big_paren ;{}
     int void
     if while for else
     or and equal not;||、&&、==、!=
     less and_less;<、<=
     more and_more;>、>=
     return
     semicolon comma
     EOF))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(define-tokens tokens-with-value
  (NUM VAR))

(define-lex-trans uinteger
  (syntax-rules () ((_ d) (:+ d))))

(define-lex-abbrevs
  (digit   (char-range "0" "9"))
  (number  (uinteger digit))
  (identifier-char (:or (char-range "a" "z")
                        (char-range "A" "Z")
                        "_"))
  (identifier (:: identifier-char
                  (:* (:or identifier-char
                           digit)))))

(define sub-program-lexer
  (lexer-src-pos
   ("(" (token-l_small_paren))
   (")" (token-r_small_paren))
   ("[" (token-l_mid_paren))
   ("]" (token-r_mid_paren))
   ("{" (token-l_big_paren))
   ("}" (token-r_big_paren))
   ("int" (token-int))
   ("void" (token-void))
   ("if" (token-if))
   ("while" (token-while))
   ("for" (token-for))
   ("else" (token-else))
   ("||" (token-or))
   ("&&" (token-and))
   ("==" (token-equal))
   ("!=" (token-not))
   ("<" (token-less))
   ("<=" (token-and_less))
   (">" (token-more))
   (">=" (token-and_more))
   ("return" (token-return))
   (";" (token-semicolon))
   ("," (token-comma))
   ("+" (token-+))
   ("*" (token-*))
   ("&" (token-&))
   ("-" (token--))
   ("/" (token-/))
   ("=" (token-=))
   (number (token-NUM (string->number lexeme)))
   (identifier (token-VAR (string->symbol lexeme)))
   (whitespace (return-without-pos (sub-program-lexer input-port)))
   ((eof) (token-EOF))))
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define program-parser
  (parser
   (start program);開始記号に当たる非終端記号
   (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
   (src-pos);位置情報を含むオブジェクトを返す
   (debug "siple-parser.tbl")
   (error (lambda (tok-ok? tok-name tok-value start-pos end-pos)
             (error "parse error:" tok-name tok-value)))
   (tokens tokens-with-value tokens-without-value)

   ;;;;;;;;;;;;;;;;;;;;;;;;;;BNFの指定;;;;;;;;;;;;;;;;;;;;;;;;;;
   (grammar
    (program ((external_declaration) $1)
             ((program external_declaration) (cons $1 $2)))
    (external_declaration ((declaration) $1)
                          ((function_prototype) $1)
                          ((function_definition) $1))
    (declaration ((type_specifier declarator_list semicolon) (declaration_st  $1 $2)))
    (declarator_list ((declarator) $1)
                      ((declarator_list comma declarator)(cons $1 $3)))
    (declarator ((direct_declarator) (declarator_st $1))
                ((* direct_declarator) (declarator_ast_st $2)))

    (direct_declarator ((VAR) (id_st $1 $1-start-pos));構造体id_stを作成.
                       ((VAR l_mid_paren NUM r_mid_paren)(array_st $1 $3 $1-start-pos)))

    (function_prototype ((type_specifier function_declarator semicolon)
                         (func_proto_st $1 $2)));構造体func_proto_stを作成.
    (function_declarator ((VAR l_small_paren parameter_type_list r_small_paren)
                          (func_declarator_st $1 $3));構造体func_declarator_stを作成.
                         ((VAR l_small_paren r_small_paren)(func_declarator_null_st $1))
                         ((* VAR l_small_paren parameter_type_list r_small_paren)
                          (func_declarator_ast_st (id_ast_st $2 $2-start-pos) $4))
                         ((* VAR l_small_paren r_small_paren)
                          (func_declarator_ast_null_st (id_ast_st $2 $2-start-pos))))

    (function_definition ((type_specifier function_declarator compound_statement)
                          (func_def_st $1 $2 $3)));構造体func_def_stを作成.
    (parameter_type_list ((parameter_declaration) $1)
                         ((parameter_type_list comma parameter_declaration)(cons $1 $3)))
    (parameter_declaration ((type_specifier parameter_declarator)(para_declaration_st $1 $2)))
    (parameter_declarator ((VAR) (id_st $1 $1-start-pos));構造体id_stを作成.
                          ((* VAR)(id_ast_st $2 $2-start-pos)));構造体id_ast_stを作成.
    (type_specifier ((int) (spec_st 'int $1-start-pos));構造体spec_stを作成.
                    ((void) (spec_st 'void $1-start-pos)));構造体spec_stを作成.
    (statement ((semicolon) (null_statement_st 'null))
               ((expression semicolon)(exp_with_semi_st $1));構造体exp-stを作成.
               ((compound_statement) $1)
               ((if l_small_paren expression r_small_paren statement)
                (if_st $3 $5 $1-start-pos));構造体if-stを作成.
               ((if l_small_paren expression r_small_paren statement else statement)
                (if_else_st $3 $5 $7 $1-start-pos $6-start-pos));構造体if_else_stを作成.
               ((while l_small_paren expression r_small_paren statement)
                (while_st $3 $5 $1-start-pos));構造体while-stを作成.
               ((for l_small_paren
                  expression semicolon expression semicolon expression
                  r_small_paren statement)
                (for_0_st $3 $5 $7 $9 $1-start-pos));構造体for_0_stを作成.
               ((for l_small_paren
                  semicolon expression semicolon expression
                  r_small_paren statement)
                (for_1_st $4 $6 $8 $1-start-pos));構造体for_1_stを作成.
               ((for l_small_paren
                  expression semicolon semicolon expression
                  r_small_paren statement)
                (for_2_st $3 $6 $8 $1-start-pos));構造体for_2_stを作成.
               ((for l_small_paren
                  expression semicolon expression semicolon
                  r_small_paren statement)
                (for_3_st $3 $5 $8 $1-start-pos));構造体for_3_stを作成.
               ((for l_small_paren expression
                  semicolon semicolon
                  r_small_paren statement)
                (for_4_st $3 $7 $1-start-pos));構造体for_4_stを作成.
               ((for l_small_paren
                  semicolon expression semicolon
                  r_small_paren statement)
                (for_5_st $4 $7 $1-start-pos));構造体for_5_stを作成.
               ((for l_small_paren
                  semicolon semicolon expression
                  r_small_paren statement)
                (for_6_st $5 $7 $1-start-pos));構造体for_6_stを作成.
               ((for l_small_paren
                  semicolon semicolon
                  r_small_paren statement)
                (for_7_st $6 $1-start-pos));構造体for_7_stを作成.
               ((return expression semicolon)(return_st $2 $1-start-pos));構造体return_stを作成.
               ((return semicolon)(return_null_st 'null $1-start-pos)));構造体return-null-stを作成.
    (compound_statement ((l_big_paren declaration_list statement_list r_big_paren)
                         (compound_st $2 $3));構造体compound_stを作成.
                        ((l_big_paren declaration_list r_big_paren)
                         (compound_dec_st $2));構造体compound_dec_stを作成.
                        ((l_big_paren statement_list r_big_paren)
                         (compound_sta_st $2));構造体copound_sta_stを作成.
                        ((l_big_paren r_big_paren)
                         (compound_null_st 'null)));構造体compound_null_stを作成.
    (declaration_list ((declaration) $1)
                      ((declaration_list declaration)(cons $1 $2)))
    (statement_list ((statement) $1)
                    ((statement_list statement)(cons $1 $2)))
    (expression ((assign_expr) $1)
                ((expression comma assign_expr)(cons $1 $3)))
    (assign_expr ((logical_OR_expr) $1)
                 ((logical_OR_expr = assign_expr)
                  (assign_exp_st $1 $3 $2-start-pos)));構造体assign_exp_stを作成.
    (logical_OR_expr ((logical_AND_expr) $1)
                     ((logical_OR_expr or logical_AND_expr)
                      (logic_exp_st 'or $1 $3 $2-start-pos)));構造体logic_exp_stを作成.
    (logical_AND_expr ((equality_expr) $1)
                      ((logical_AND_expr and equality_expr)
                       (logic_exp_st 'and $1 $3 $2-start-pos)));構造体logic_exp_stを作成.
    (equality_expr ((relational_expr) $1)
                   ((equality_expr equal relational_expr)
                    (rel_exp_st 'equal $1 $3 $2-start-pos));構造体rel_exp_stを作成.
                   ((equality_expr not relational_expr)
                    (rel_exp_st 'not $1 $3 $2-start-pos)));構造体rel_exp_stを作成.

    (relational_expr ((add_expr) $1)
                     ((relational_expr less add_expr)
                      (rel_exp_st 'less $1 $3 $2-start-pos));構造体rel_exp_stを作成.
                     ((relational_expr more add_expr)
                      (rel_exp_st 'more $1 $3 $2-start-pos));構造体rel_exp_stを作成.
                     ((relational_expr and_less add_expr)
                      (rel_exp_st 'and_less $1 $3 $2-start-pos));構造体rel_exp_stを作成.
                     ((relational_expr and_more add_expr)
                      (rel_exp_st 'adn_more $1 $3 $2-start-pos)));構造体rel_exp_stを作成.

    (add_expr ((mult_expr) $1)
              ((add_expr + mult_expr)
               (alge_exp_st 'add $1 $3 $2-start-pos));構造体alge_exp_stを作成.
              ((add_expr - mult_expr)
               (alge_exp_st 'sub $1 $3 $2-start-pos)));構造体alge_exp_stを作成.
    (mult_expr ((unary_expr) $1)
               ((mult_expr * unary_expr)
                (alge_exp_st 'mul $1 $3 $2-start-pos));構造体alge_exp_stを作成.
               ((mult_expr / unary_expr)
                (alge_exp_st 'div $1 $3 $2-start-pos)));構造体alge_exp_stを作成.
    (unary_expr ((postfix_expr) $1)
                ((- unary_expr)
                 (unary_exp_st 'minus $2 $1-start-pos));構造体unary-exp-stを作成.
                ((& unary_expr)
                 (unary_exp_st 'amp $2 $1-start-pos));構造体unary-exp-stを作成.
                ((* unary_expr)
                 (unary_exp_st 'ast $2 $1-start-pos)));構造体unary-exp-stを作成.

    (postfix_expr ((primary_expr) $1)
                  ((postfix_expr l_mid_paren expression r_mid_paren)
                   (array_var_st $1 $3  $1-start-pos));構造体array_var_stを作成.
                  ((VAR l_small_paren argument_expression_list r_small_paren)
                   (func_st $1 $3));構造体func_stを作成.
                  ((VAR l_small_paren r_small_paren)
                   (func_nopara_st $1)));構造体func_nopara_stを作成.
    (primary_expr ((VAR)(id_st $1 $1-start-pos));構造体id_stを作成.
                  ((NUM) (constant_st $1 $1-start-pos))
                  ((l_small_paren expression r_small_paren)
                   (exp_in_paren_st $2)));構造体exp_stを作成.
    (argument_expression_list ((assign_expr) $1)
                              ((argument_expression_list comma assign_expr)(cons $1 $3)))
    )
   )
  )
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(define (parse-string s)
  (let ((p (open-input-string s)))
  (program-parser (lambda () (sub-program-lexer p)))))

(define (parse-port p)
  (program-parser (lambda () (sub-program-lexer p))))

;実行例
;(define p (open-input-file "kadai01.sc"))
;(port-count-lines! p)
;(parse-port p)
```

2.2 設計方針

この構文解析器は構文解析の際にプログラム（文字列）-> トークン列 -> 抽象構文木という 2 段階の流れをとっている. すなわち、構文解析器はプログラムをトークンに置き換える部分とトークンの列から抽象構文着を作成する部分からなる.

2.3　各部の説明

前節で述べた「プログラム（文字列）-> トークン列」の変換を行うのが関数 sub-program-lexer である。この関数内部では関数 lexer-src-pos が使われており、これは指定した文字列もしくは指定した文字列に対応するトークンを文字列中に発見したときに返すトークンを個別に指定できる関数である. また、lexer-src-pos の返り値には位置情報が含まれる. なお、特定の文字列、正規表現とトークンの対応付けは define-lex-abbrevs で定義し、その他のトークンは define-empty-tokens で定義される. 次に、「トークン列 -> 抽象構文木」の変換においては、Small C の BNF の非終端記号の終端記号への置換えを parser 関数の grammer で指定している. ここでは特定の非終端記号もしくは終端記号を解析したときに起こすアクションをそれぞれ個別に指定できる. このプログラムの場合は抽象構文木のノードとなる構造体を返す. また、それらの構造体はプログラムの冒頭部分で #:transparent をつけて定義している. なお、プログラムの実行は、関数 parse-string に文字列としてプログラムを直接渡すか、ファイルからポートオブジェクトを生成して関数 parse-port に渡すことで行う.

3　課題 6

この課題では、課題 5 のプログラムを使って実際に構文解析を行う.

3.1　課題 6 の実行結果

test.sc に次のような内容を記述する.

リスト 3: 解析対象の例

```
int func1(int a, int b);
int func2(int a, int b);
int array[2];

void main(){
    array[0] = 100;
    array[1] = 9999;
    func2(999, func1(1, 9));
}

int func1(int a, int b){
    b = (a + b);
    if(b > 8){
        b = (b * 100);
    }
    else{
        b = 0;
    }
    return b;
}

int func2(int a, int b){
    int d;
    while(a > b){
        b = ((b + 1) * 2);
    }
    array[0] = b;
    array[1] = a;
    d = (array[0]+array[1]);
    return d;
}
```

これをポートオブジェクトに変換して構文解析を行った.

リスト 4: 構文解析器プログラムの実行例

```
>(define p (open-input-file "test.sc"))
>(port-count-lines! p)
>(parse-port p)

(cons
 (cons
  (cons
   (cons
    (cons
     (func_proto_st
      (spec_st 'int (position 1 1 0))
      (func_declarator_st
       'func1
       (cons
        (para_declaration_st (spec_st 'int (position 11 1 10)) (id_st 'a (position 15 1 14)))
        (para_declaration_st (spec_st 'int (position 18 1 17)) (id_st 'b (position 22 1 21))))))
     (func_proto_st
      (spec_st 'int (position 26 2 0))
      (func_declarator_st
       'func2
       (cons
        (para_declaration_st (spec_st 'int (position 36 2 10)) (id_st 'a (position 40 2 14)))
        (para_declaration_st (spec_st 'int (position 43 2 17)) (id_st 'b (position 47 2 21)))))))
    (declaration_st (spec_st 'int (position 51 3 0))
                    (declarator_st (array_st 'array 2 (position 55 3 4)))))
   (func_def_st
    (spec_st 'void (position 66 5 0))
    (func_declarator_null_st 'main)
    (compound_sta_st
     (cons
      (cons
       (exp_with_semi_st
        (assign_exp_st
         (array_var_st (id_st 'array (position 83 6 4))
                       (constant_st 0 (position 89 6 10))
                       (position 83 6 4))
         (constant_st 100 (position 94 6 15))
         (position 92 6 13)))
       (exp_with_semi_st
        (assign_exp_st
         (array_var_st (id_st 'array (position 103 7 4))
                       (constant_st 1 (position 109 7 10))
                       (position 103 7 4))
         (constant_st 9999 (position 114 7 15))
         (position 112 7 13))))
      (exp_with_semi_st
       (func_st
        'func2
        (cons
         (constant_st 999 (position 131 8 10))
         (func_st 'func1 (cons (constant_st 1 (position 142 8 21))
                               (constant_st 9 (position 145 8 24))))))))))))
  (func_def_st
   (spec_st 'int (position 153 11 0))
   (func_declarator_st
    'func1
    (cons
     (para_declaration_st (spec_st 'int (position 163 11 10))
                          (id_st 'a (position 167 11 14)))
     (para_declaration_st (spec_st 'int (position 170 11 17))
                          (id_st 'b (position 174 11 21)))))
   (compound_sta_st
    (cons
     (cons
      (exp_with_semi_st
       (assign_exp_st
        (id_st 'b (position 182 12 4))
        (exp_in_paren_st (alge_exp_st 'add (id_st 'a (position 187 12 9))
                                      (id_st 'b (position 191 12 13))
                                      (position 189 12 11)))
        (position 184 12 6)))
      (if_else_st
       (rel_exp_st 'more (id_st 'b (position 202 13 7))
                   (constant_st 8 (position 206 13 11)) (position 204 13 9))
       (compound_sta_st
        (exp_with_semi_st
         (assign_exp_st
          (id_st 'b (position 218 14 8))
          (exp_in_paren_st (alge_exp_st 'mul (id_st 'b (position 223 14 13))
                                        (constant_st 100 (position 227 14 17))
                                        (position 225 14 15)))
          (position 220 14 10))))
       (compound_sta_st
        (exp_with_semi_st (assign_exp_st (id_st 'b (position 257 17 8))
                                         (constant_st 0 (position 261 17 12))
                                         (position 259 17 10))))
       (position 199 13 4)
       (position 243 16 4)))
     (return_st (id_st 'b (position 281 19 11)) (position 274 19 4))))))
 (func_def_st
  (spec_st 'int (position 287 22 0))
  (func_declarator_st
   'func2
   (cons
    (para_declaration_st (spec_st 'int (position 297 22 10))
                         (id_st 'a (position 301 22 14)))
    (para_declaration_st (spec_st 'int (position 304 22 17))
                         (id_st 'b (position 308 22 21)))))
  (compound_st
   (declaration_st (spec_st 'int (position 316 23 4))
                   (declarator_st (id_st 'd (position 320 23 8))))
   (cons
    (cons
     (cons
      (cons
       (while_st
        (rel_exp_st 'more (id_st 'a (position 333 24 10))
                    (id_st 'b (position 337 24 14)) (position 335 24 12))
        (compound_sta_st
         (exp_with_semi_st
          (assign_exp_st
           (id_st 'b (position 349 25 8))
           (exp_in_paren_st
            (alge_exp_st
             'mul
             (exp_in_paren_st (alge_exp_st 'add (id_st 'b (position 355 25 14))
                                           (constant_st 1 (position 359 25 18))
                                           (position 357 25 16)))
             (constant_st 200000 (position 364 25 23))
             (position 362 25 21)))
           (position 351 25 10))))
        (position 327 24 4))
       (exp_with_semi_st
        (assign_exp_st
         (array_var_st (id_st 'array (position 383 27 4))
                       (constant_st 0 (position 389 27 10))
                       (position 383 27 4))
         (id_st 'b (position 394 27 15))
         (position 392 27 13))))
      (exp_with_semi_st
       (assign_exp_st
        (array_var_st (id_st 'array (position 401 28 4))
                      (constant_st 1 (position 407 28 10))
                      (position 401 28 4))
        (id_st 'a (position 412 28 15))
        (position 410 28 13))))
     (exp_with_semi_st
      (assign_exp_st
       (id_st 'd (position 419 29 4))
       (exp_in_paren_st
        (alge_exp_st
         'add
         (array_var_st (id_st 'array (position 424 29 9))
                       (constant_st 0 (position 430 29 15))
                       (position 424 29 9))
         (array_var_st (id_st 'array (position 433 29 18))
                       (constant_st 1 (position 439 29 24))
                       (position 433 29 18))
         (position 432 29 17)))
       (position 421 29 6))))
    (return_st (id_st 'd (position 455 30 11)) (position 448 30 4))))))
```

さらに、例えば test.sc の内容を

```
int func1(int a, int b);
int func2(int a, int b);
int array[2];
```

```

void main(){
    array[0] = 100;
    array[1] = 9999;
    func2(999, func1(1, 9));
    unvalid = + unvalid
}
```

と書き換えて同様に構文解析を行うと、構文解析器は停止し、何も出力しない. また処理系の Dr.Racket 側ではエラーが

```
parse error: + #f
```

と出力され、文字列を右側から構文解析し、

```
= +
```

という文字列を読み取った途端このプログラムが定義していた BNF から外れたことを発見し、エラーを出力している事がわかる.

4 課題 7

この課題では、課題 5 の内容とは逆に、抽象構文木を受け取って、プログラムとして文字列を返すような関数を作成する.

4.1 課題 7 の回答

コードは次のようになった.

リスト 5: 抽象構文器をプログラムに変換する関数

```
#lang racket
(require parser-tools/lex
         (prefix-in : parser-tools/lex-sre))
(require parser-tools/yacc)
(provide (all-defined-out))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;データ型定義;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;programはexternal-declarationのリスト.
;external_declarationはdeclaartion、func-proto、func-defからなるリスト
(struct declaration_st (type-spec declarator-list)#:transparent)
;例) (int (id1 id2 *id3))

(struct func_declarator_st (name para-list)#:transparent)
(struct func_declarator_null_st (name)#:transparent)
(struct func_declarator_ast_st (name para-list)#:transparent)
(struct func_declarator_ast_null_st (name)#:transparent)
;例)nameは関数名、para-listは引数のリスト.

(struct func_proto_st (type-spec func-declarator-st)#:transparent)
(struct func_def_st (type-spec func-declarator-st compound-state-list)#:transparent)
;例) (int funcname1 (id1 id2 *id3) compound_list)
;compound_listの中身は構造体declaration_listとstatement_listからなる.

(struct declarator_st (var)#:transparent)
(struct declarator_ast_st (var)#:transparent)
;declaratorの定義

;declaration_listは構造体declarationがリストになったもの.以下で定義.
;(struct declaration_st2 (type-spec para-list)#:transparent);;;;;;
;例) (int (id1 id2 *id3 id4[5]))

```

```
(struct para_declaration_st (type-spec para)#:transparent)
;例) (int id1)

(struct exp_st (exp)#:transparent)
;式を格納する構造体.

(struct assign_exp_st (dest src pos)#:transparent)
;例)代入を表す構造体. x = 3なら(x 3)

(struct logic_exp_st (log-ope op1 op2 pos)#:transparent)
;例) (or a b)もしくは(and a b)

(struct rel_exp_st (rel-ope op1 op2 pos)#:transparent)
;例) rel_opeは'equal 'not 'less 'and_less 'more 'and_moreで (less a b)など

(struct alge_exp_st (alge-ope op1 op2 pos)#:transparent)
;例) alge_opeは'add 'sub 'mul 'divで('add a b)など

(struct id_st (name pos)#:transparent)
(struct id_ast_st (name pos)#:transparent)
;終端記号 identifierを表す構造体.

(struct array_st (name num pos)#:transparent);宣言時.posはnameの位置.
(struct array_var_st (name num pos)#:transparent);式の中で用いる.posはnameの位置.
;終端記号となりうる配列を表す構造体.

(struct spec_st (type pos)#:transparent)
;データ型を表す構造体

(struct unary_exp_st (mark op pos)#:transparent)
;postfix_expを表す構造体.markは'minus、'ast、'amp

(struct constant_st (cons pos)#:transparent)
;定数を表す構造体.

(struct null_statement_st (null)#:transparent)
;セミコロンのみからなるstatementを表す構造体

(struct exp_with_semi_st (exp)#:transparent)
;expressionとセミコロンからなる式を表す構造体.

(struct exp_in_paren_st (exp)#:transparent)
;()で囲まれたexpression

(struct if_st (cond-exp state pos)#:transparent);else無し.posはifの位置.
(struct if_else_st (cond-exp state else-state if-pos else-pos)#:transparent);elseあり
;if文を表す構造体.

(struct while_st (cond-exp statement pos)#:transparent);posはwhileの位置.
;while文を表す構造体.

(struct for_0_st (cond-exp1 cond-exp2 cond-exp3 statement pos)#:transparent)
(struct for_1_st (cond-exp1 cond-exp2 statement pos)#:transparent)
(struct for_2_st (cond-exp1 cond-exp2 statement pos)#:transparent)
(struct for_3_st (cond-exp1 cond-exp2 statement pos)#:transparent)
(struct for_4_st (cond-exp1 statement pos)#:transparent)
(struct for_5_st (cond-exp1 statement pos)#:transparent)
(struct for_6_st (cond-exp1 statement pos)#:transparent)
(struct for_7_st (statement pos)#:transparent)
;for文を表す構造体.0〜7の数字によってnullの位置が構文木作成の時点でわかる.

(struct return_st (exp pos)#:transparent);posはreturnの位置.
(struct return_null_st (exp pos)#:transparent)
;return文を表す構造体.

(struct compound_st (declaration-list statement-list)#:transparent)
(struct compound_dec_st (declaration-list)#:transparent)
(struct compound_sta_st (statement-list)#:transparent)
(struct compound_null_st (null)#:transparent)
;compound_statementを表す構造体.

(struct func_st (name para)#:transparent)
(struct func_nopara_st (name)#:transparent)
;関数呼び出しを表す構造体
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;



;syn-to-codeは構文木として構造体を文字列に変換する.
```

```

;dec-list-to-codeは構造体(declaration_st-declarator-list x)を文字列に変換する際の補助関数.
;para-list-to-codeは構造体(func-declarator_st-para-list x)を文字列に変換する際のの補助関数.
;arg-list-to-codeは構造体(func_st-para x)を文字列に変換する際の補助関数.
;上記の関数はカンマの数などがわかっていないときの処理に使用する.

(define (syn-to-code x)
  (cond ((struct? x) (cond ((declaration_st? x)
                            (string-append (syn-to-code (declaration_st-type-spec x))
                                           " "
                                           (dec-list-to-code (declaration_st-declarator-list x))
                                           ";"))

                           ((func_declarator_st? x)
                            (string-append (symbol->string (func_declarator_st-name x))
                                           "("
                                           (para-list-to-code (func_declarator_st-para-list x))
                                           ")"))
                           ;(struct func_declarator_st (name para-list)#:transparent)

                           ((func_declarator_null_st? x)
                            (string-append (symbol->string (func_declarator_null_st-name x))
                                           "()"))
                           ;(struct func_declarator_null_st (name)#:transparent)

                           ((func_declarator_ast_st? x)
                            (string-append "*"
                                           (syn-to-code (func_declarator_ast_st-name x))
                                           (syn-to-code (func_declarator_ast_st-para-list x))))
                           ;(struct func_declarator_ast_st (name para-list)#:transparent)

                           ((func_declarator_ast_null_st? x)
                            (string-append "*"
                                           (syn-to-code (func_declarator_ast_null_st-name x))))
                           ;(struct func_declarator_ast_null_st (name)#:transparent)

                           ((func_proto_st? x)
                            (string-append (syn-to-code (func_proto_st-type-spec x))
                                           " "
                                           (syn-to-code (func_proto_st-func-declarator-st x))
                                           ";"))
                           ;(struct func_proto_st (type-spec func-declarator-st)#:transparent)

                           ((func_def_st? x)
                            (string-append (syn-to-code (func_def_st-type-spec x))
                                           " "
                                           (syn-to-code (func_def_st-func-declarator-st x))
                                           (syn-to-code (func_def_st-compound-state-list x))))
                           ;(struct func_def_st
                           ;(type-spec func-declarator-st
                           ;compound-state-list)#:transparent)

                           ((declarator_st? x) (syn-to-code (declarator_st-var x)))
                           ;(struct declarator_st (var)#:transparent)

                           ((declarator_ast_st? x)
                            (string-append "*"
                                           (syn-to-code (declarator_ast_st-var x))))
                           ;(struct declarator_ast_st (var)#:transparent)

                           ((para_declaration_st? x)
                            (string-append (syn-to-code (para_declaration_st-type-spec x))
                                           " "
                                           (syn-to-code (para_declaration_st-para x))))
                           ;(struct para_declaration_st (type-spec para)#:transparent)

                           ((exp_st? x) (syn-to-code (exp_st-exp x)))
                           ;(struct exp_st (exp)#:transparent)

                           ((assign_exp_st? x) (string-append (syn-to-code (assign_exp_st-dest x))
                                                              " = "
                                                              (syn-to-code (assign_exp_st-src x))))
                           ;(struct assign_exp_st (dest src pos)#:transparent)

                           ((logic_exp_st? x)
                            (string-append (syn-to-code (logic_exp_st-op1 x))
                                           (cond ((eq? (logic_exp_st-log-ope x) 'or)
                                                  " || ")
                                                 ((eq? (logic_exp_st-log-ope x) 'and)
                                                  " && "))
```

```
                (syn-to-code (logic_exp_st-op2 x))))
;(struct logic_exp_st (log-ope op1 op2 pos)#:transparent)

((rel_exp_st? x) (string-append
                  (syn-to-code (rel_exp_st-op1 x))
                  (cond ((eq? (rel_exp_st-rel-ope x) 'equal)
                         " == ")
                        ((eq? (rel_exp_st-rel-ope x) 'not)
                         " != ")
                        ((eq? (rel_exp_st-rel-ope x) 'less)
                         " < ")
                        ((eq? (rel_exp_st-rel-ope x) 'and_less)
                         " <= ")
                        ((eq? (rel_exp_st-rel-ope x) 'more)
                         " > ")
                        ((eq? (rel_exp_st-rel-ope x) 'and_more)
                         " >= "))
                  (syn-to-code (rel_exp_st-op2 x))))
;(struct rel_exp_st (rel-ope op1 op2 pos)#:transparent)

((alge_exp_st? x) (string-append
                   (syn-to-code (alge_exp_st-op1 x))
                   (cond ((eq? (alge_exp_st-alge-ope x) 'add)
                          " + ")
                         ((eq? (alge_exp_st-alge-ope x) 'sub)
                          " - ")
                         ((eq? (alge_exp_st-alge-ope x) 'mul)
                          " * ")
                         ((eq? (alge_exp_st-alge-ope x) 'div)
                          " / "))
                   (syn-to-code (alge_exp_st-op2 x))))
;(struct alge_exp_st (alge-ope op1 op2 pos)#:transparent)

((id_st? x) (symbol->string (id_st-name x)))
;(struct id_st (name pos)#:transparent)
((id_ast_st? x)
 (string-append "*"
                (symbol->string (id_ast_st-name x))))
;(struct id_ast_st (name pos)#:transparent)

((array_st? x) (string-append (symbol->string (array_st-name x))
                              "["
                              (number->string (array_st-num x)) "]"))
;(struct array_st (name num pos)#:transparent);宣言時.posはnameの位置.
((array_var_st? x)
 (string-append (syn-to-code (array_var_st-name x))
                "["
                (syn-to-code (array_var_st-num x)) "]"))
;(struct array_var_st (name num pos)#:transparent);式の中で用いる.
;posはnameの位置.

((spec_st? x) (symbol->string (spec_st-type x)))
;(struct spec_st (type pos)#:transparent)

((unary_exp_st? x)
 (string-append (cond ((eq? (unary_exp_st-mark x) 'minus) "-")
                      ((eq? (unary_exp_st-mark x) 'ast) "*")
                      ((eq? (unary_exp_st-mark x) 'amp) "&"))
                (unary_exp_st-op x)))
;(struct unary_exp_st (mark op pos)#:transparent)

((constant_st? x) (number->string (constant_st-cons x)))
;(struct constant_st (cons pos)#:transparent)

((null_statement_st? x) ";")
;(struct null_statement_st (null))

((exp_with_semi_st? x)
 (string-append (syn-to-code (exp_with_semi_st-exp x)) ";"))
;(struct exp_with_semi_st (exp)#:transparent)

((exp_in_paren_st? x)
 (string-append "(" (syn-to-code (exp_in_paren_st-exp x)) ")"))
;(struct exp_in_paren_st (exp)#:transparent)

((if_st? x) (string-append "if ("
                           (syn-to-code (if_st-cond-exp x))
                           ")"
                           (syn-to-code (if_st-state x))))
;(struct if_st (cond-exp state pos)#:transparent);else無し.
```

```
;posはifの位置.
((if_else_st? x) (string-append "if("
                                (syn-to-code (if_else_st-cond-exp x))
                                ")"
                                (syn-to-code (if_else_st-state x))
                                "else"
                                (syn-to-code (if_else_st-else-state x))))
;(struct if_else_st
;(cond-exp state else-state if-pos else-pos)#:transparent)
;elseあり

((while_st? x) (string-append "while("
                              (syn-to-code (while_st-cond-exp x))
                              ")"
                              (syn-to-code (while_st-statement x))))
;(struct while_st (cond-exp statement pos)#:transparent)
;posはwhileの位置.

((for_0_st? x) (string-append "for("
                              (syn-to-code (for_0_st-cond-exp1 x))
                              ";"
                              (syn-to-code (for_0_st-cond-exp2 x))
                              ";"
                              (syn-to-code (for_0_st-cond-exp3 x))
                              ")"
                              (syn-to-code (for_0_st-statement x))))
;(struct for_0_st
        ;(cond-exp1 cond-exp2 cond-exp3 statement pos)#:transparent)

((for_1_st? x) (string-append "for(;"
                              (syn-to-code (for_1_st-cond-exp1 x))
                              ";"
                              (syn-to-code (for_1_st-cond-exp2 x))
                              ")"
                              (syn-to-code (for_1_st-statement x))))
;(struct for_1_st (cond-exp1 cond-exp2 statement pos)#:transparent)

((for_2_st? x) (string-append "for("
                              (syn-to-code (for_2_st-cond-exp1 x))
                              ";;"
                              (syn-to-code (for_2_st-cond-exp2 x))
                              ")"
                              (syn-to-code (for_2_st-statement x))))
;(struct for_2_st (cond-exp1 cond-exp2 statement pos)#:transparent)

((for_3_st? x) (string-append "for("
                              (syn-to-code (for_3_st-cond-exp1 x))
                              ";"
                              (syn-to-code (for_3_st-cond-exp2 x))
                              ";)"
                              (syn-to-code (for_3_st-statement x))))
;(struct for_3_st (cond-exp1 cond-exp2 statement pos)#:transparent)

((for_4_st? x) (string-append "for("
                              (syn-to-code (for_4_st-cond-exp1 x))
                              ";;)"
                              (syn-to-code (for_4_st-statement x))))
;(struct for_4_st (cond-exp1 statement pos)#:transparent)

((for_5_st? x) (string-append "for(;"
                              (syn-to-code (for_5_st-cond-exp1 x))
                              ";)"
                              (syn-to-code (for_5_st-statement x))))
;(struct for_5_st (cond-exp1 statement pos)#:transparent)

((for_6_st? x) (string-append "for(;;"
                              (syn-to-code (for_6_st-cond-exp1 x))
                              ")"
                              (syn-to-code (for_6_st-statement x))))
;(struct for_6_st (cond-exp1 statement pos)#:transparent)

((for_7_st? x) (string-append "for(;;)"
                              (syn-to-code (for_7_st-statement x))))
;(struct for_7_st (statement pos)#:transparent)

((return_st? x) (string-append "return"
                               (syn-to-code (return_st-exp x))
                               ";"))
;(struct return_st (exp pos)#:transparent);posはreturnの位置.

```

```
                           ((return_null_st? x)
                            (string-append "return"
                                           (syn-to-code (return_null_st-exp x))
                                           ";"))
                           ;(struct return_null_st (exp pos)#:transparent)

                           ((compound_st? x)
                            (string-append "{"
                                           (syn-to-code (compound_st-declaration-list x))
                                           (syn-to-code (compound_st-statement-list x))
                                           "}"))
                           ;(struct compound_st (declaration-list statement-list)#:transparent)

                           ((compound_dec_st? x)
                            (string-append "{"
                                           (syn-to-code (compound_dec_st-declaration-list x))
                                           "}"))
                           ;(struct compound_dec_st (declaration-list)#:transparent)

                           ((compound_sta_st? x)
                            (string-append
                             "{"
                             (syn-to-code (compound_sta_st-statement-list x)) ;;;;;;
                             "}"))
                           ;(struct compound_sta_st (statement-list)#:transparent)

                           ((compound_null_st? x) "{}")
                           ;(struct compound_null_st (null)#:transparent)

                           ((func_st? x) (string-append (symbol->string (func_st-name x))
                                                        "("
                                                        (arg-list-to-code (func_st-para x))
                                                        ")"))
                           ;(struct func_st (name para)#:transparent)

                           ((func_nopara_st? x)
                            (string-append (symbol->string (func_nopara_st-name x))
                                           "("
                                           ")"))
                           ;(struct func_nopara_st (name)#:transparent)




                           ;(#t "error: unknown syntax")
                           ))


        (else (string-append (syn-to-code (car x))
                             " "
                             (syn-to-code (cdr x))))))

(define (dec-list-to-code x)
  (cond ((struct? x) (syn-to-code x))
        (else (string-append (dec-list-to-code (car x))
                             ", "
                             (syn-to-code (cdr x))))))

(define (para-list-to-code x)
  (cond ((struct? x) (syn-to-code x))
        (else (string-append (para-list-to-code (car x))
                             ", "
                             (syn-to-code (cdr x))))))

(define (arg-list-to-code x)
  (cond ((struct? x) (syn-to-code x))
        (else (string-append (arg-list-to-code (car x))
                             ", "
                             (syn-to-code (cdr x))))))
```

実行結果は以下のようになった.

リスト 6: プログラムへの変換例

```
#lang racket
> (syn-to-code
   (cons
    (cons
(cons
 (cons
  (cons
   (func_proto_st
    (spec_st 'int (position 1 1 0))
    (func_declarator_st
     'func1
     (cons
      (para_declaration_st (spec_st 'int (position 11 1 10))
                           (id_st 'a (position 15 1 14)))
      (para_declaration_st (spec_st 'int (position 18 1 17))
                           (id_st 'b (position 22 1 21))))))
   (func_proto_st
    (spec_st 'int (position 26 2 0))
    (func_declarator_st
     'func2
     (cons
      (para_declaration_st (spec_st 'int (position 36 2 10))
                           (id_st 'a (position 40 2 14)))
      (para_declaration_st (spec_st 'int (position 43 2 17))
                           (id_st 'b (position 47 2 21)))))))
  (declaration_st (spec_st 'int (position 51 3 0))
                  (declarator_st (array_st 'array 2 (position 55 3 4)))))
 (func_def_st
  (spec_st 'void (position 66 5 0))
  (func_declarator_null_st 'main)
  (compound_sta_st
   (cons
    (cons
     (exp_with_semi_st
      (assign_exp_st
       (array_var_st (id_st 'array (position 83 6 4))
                     (constant_st 0 (position 89 6 10)) (position 83 6 4))
       (constant_st 100 (position 94 6 15))
       (position 92 6 13)))
     (exp_with_semi_st
      (assign_exp_st
       (array_var_st (id_st 'array (position 103 7 4))
                     (constant_st 1 (position 109 7 10)) (position 103 7 4))
       (constant_st 9999 (position 114 7 15))
       (position 112 7 13))))
    (exp_with_semi_st
     (func_st
      'func2
      (cons
       (constant_st 999 (position 131 8 10))
       (func_st 'func1 (cons (constant_st 1 (position 142 8 21))
                             (constant_st 9 (position 145 8 24))))))))))))
(func_def_st
 (spec_st 'int (position 153 11 0))
 (func_declarator_st
  'func1
  (cons
   (para_declaration_st (spec_st 'int (position 163 11 10))
                        (id_st 'a (position 167 11 14)))
   (para_declaration_st (spec_st 'int (position 170 11 17))
                        (id_st 'b (position 174 11 21)))))
 (compound_sta_st
  (cons
   (cons
    (exp_with_semi_st
     (assign_exp_st
      (id_st 'b (position 182 12 4))
      (exp_in_paren_st (alge_exp_st 'add (id_st 'a (position 187 12 9))
                                    (id_st 'b (position 191 12 13))
                                    (position 189 12 11)))
      (position 184 12 6)))
    (if_else_st
     (rel_exp_st 'more (id_st 'b (position 202 13 7))
                 (constant_st 8 (position 206 13 11))
                 (position 204 13 9))
     (compound_sta_st
      (exp_with_semi_st
       (assign_exp_st
        (id_st 'b (position 218 14 8))
        (exp_in_paren_st
         (alge_exp_st 'mul (id_st 'b (position 223 14 13))
                      (constant_st 100 (position 227 14 17))
                      (position 225 14 15)))
        (position 220 14 10))))
```

```
          (compound_sta_st
           (exp_with_semi_st
            (assign_exp_st (id_st 'b (position 257 17 8))
                           (constant_st 0 (position 261 17 12))
                           (position 259 17 10))))
          (position 199 13 4)
          (position 243 16 4)))
        (return_st (id_st 'b (position 281 19 11)) (position 274 19 4))))))
    (func_def_st
     (spec_st 'int (position 287 22 0))
     (func_declarator_st
      'func2
      (cons
       (para_declaration_st (spec_st 'int (position 297 22 10))
                            (id_st 'a (position 301 22 14)))
       (para_declaration_st (spec_st 'int (position 304 22 17))
                            (id_st 'b (position 308 22 21)))))
     (compound_st
      (declaration_st (spec_st 'int (position 316 23 4))
                      (declarator_st (id_st 'd (position 320 23 8))))
      (cons
       (cons
        (cons
         (cons
          (while_st
           (rel_exp_st 'more (id_st 'a (position 333 24 10))
                       (id_st 'b (position 337 24 14))
                       (position 335 24 12))
           (compound_sta_st
            (exp_with_semi_st
             (assign_exp_st
              (id_st 'b (position 349 25 8))
              (exp_in_paren_st
               (alge_exp_st
                'mul
                (exp_in_paren_st
                 (alge_exp_st 'add (id_st 'b (position 355 25 14))
                              (constant_st 1 (position 359 25 18))
                              (position 357 25 16)))
                (constant_st 200000 (position 364 25 23))
                (position 362 25 21)))
              (position 351 25 10))))
           (position 327 24 4))
          (exp_with_semi_st
           (assign_exp_st
            (array_var_st (id_st 'array (position 383 27 4))
                          (constant_st 0 (position 389 27 10))
                          (position 383 27 4))
            (id_st 'b (position 394 27 15))
            (position 392 27 13))))
         (exp_with_semi_st
          (assign_exp_st
           (array_var_st (id_st 'array (position 401 28 4))
                         (constant_st 1 (position 407 28 10))
                         (position 401 28 4))
           (id_st 'a (position 412 28 15))
           (position 410 28 13))))
        (exp_with_semi_st
         (assign_exp_st
          (id_st 'd (position 419 29 4))
          (exp_in_paren_st
           (alge_exp_st
            'add
            (array_var_st (id_st 'array (position 424 29 9))
                          (constant_st 0 (position 430 29 15))
                          (position 424 29 9))
            (array_var_st (id_st 'array (position 433 29 18))
                          (constant_st 1 (position 439 29 24))
                          (position 433 29 18))
            (position 432 29 17)))
          (position 421 29 6))))
       (return_st (id_st 'd (position 455 30 11)) (position 448 30 4))))))
   )
"int func1(int a, int b); int func2(int a, int b); int array[2]; void main(){array[0] = 100; array[1] = 9999; func2(999,
```

4.2　設計方針

このプログラムは抽象構文木として受け取ったプログラムを文字列に変換する. 具体的には抽象構文木の構造体の種類によって行う処理を１つずつ cond 文で定める. また、受け取る抽象構文木は構造体を属性値として持つ構造体であるから、この動作を再帰的に繰り返すプログラムを作成する.

4.3　各部の説明

このプログラムは、構造体を受け取って受け取った構造体の種類に応じて様々な処理を行う関数 syn-to-code を再帰的に定義することで作成されている. 基本的にはある構造体を syn-to-code が受け取ったら、その中の属性である構造体を再び syn-to-code に渡し、その結果を string-append でつなぐという設計をとっている. ただし、属性の数が任意の個数の構造体もしくはシンボルになり得るような構造体、すなわち、関数定義の際の引数のリスト等を処理する際は、他のものと同じような再帰的処理の方法ではカンマの数が不適当になるため、別の関数を用意する必要がある. それらは関数 dec-list-to-code、para-list-to-code、arg-list-to-code として定義されている.

5　感想

今回の実験では前年度のコンパイラの授業を受けてコンパイラの作成をするということで、その初期段階として構文解析器を作成しましたが、コンパイラの授業がなかなか理解できず苦戦した僕にとっては非常に苦労した課題でした. しかし実際に作ってみると去年なかなか理解できなかったものが意外とすんなりと理解でき、作りながら理解するという実験の恩恵をストレートに感じることが出来ました.