Федеральное государственное автономное

образовательное учреждение высшего образования

Университет ИТМО

Отчет

кпрактическомузаданию№1

подисциплине“Системноепрограммноеобеспечение”

Выполнил студент группы P4114: Манна Рани

Преподаватель:

Кореньков Юрий

Дмитриевич

г. Санкт-Петербург

2024

Цели

Использовать средство синтаксического анализа по выбору, реализовать модуль для разбора текста в соответствии с языком по варианту. Реализовать построение по исходному файлу с текстом синтаксического дерева с узлами, соответствующими элементам синтаксической модели языка. Вывести полученное дерево в файл в формате, поддерживающем просмотр графического представления.

Задачи

1. Изуч*ить выбранное средство синтаксического анализа*

    a. *Средство должно поддерживать программный интерфейс, совместимый с языком Си*

    b. *Средство должно параметризоваться спецификацией, описывающей синтаксическую структуру разбираемого языка*

    c. *Средство может функционировать посредством кодогенерации и/или подключения необходимых для его работы дополнительных библиотек*

    d. *Средство может быть реализовано с нуля, в этом случае оно должно использовать обобщённый алгоритм, управляемый спецификацией*

2. *Изучить синтаксис разбираемого по варианту языка и записать спецификацию для средства синтаксического анализа, включающую следующие конструкции:*

    a. *Подпрограммы со списком аргументов и возвращаемым значением*

    b. *Операции контроля потока управления – простые ветвления* if-else *и циклы или аналоги*

    c. *В зависимости от варианта – определения переменных*

    d. *Целочисленные, строковые и односимвольные литералы*

    e. *Выражения численной, битовой и логической арифметики*

    f. *Выражения над одномерными массивами*

g. *Выражения вызова функции*

3. *Реализовать модуль, использующий средство синтаксического анализа для разбора языка по варианту*

    a. *Программный интерфейс модуля должен принимать строку с текстом и возвращать структуру, описывающую соответствующее дерево разбора и коллекцию сообщений ошибке*

    b. *Результат работы модуля – дерево разбора – должно содержать иерархическое представление для всех синтаксических конструкций, включая выражения, логически представляющие собой иерархически организованные данные, даже если на уровне средства синтаксического анализа для их разбора было использовано линейное представление*

4. *Реализовать тестовую программу для демонстрации работоспособности созданного модуля*

    a. *Через аргументы командной строки программа должна принимать имя входного файла для чтения и анализа, имя выходного файла записи для дерева, описывающего синтаксическую структуру разобранного текста*

    b. *Сообщения об ошибке должны выводиться тестовой программной (не модулем, отвечающим за анализ!) в стандартный поток вывода ошибок*

5. *Результаты тестирования представить в виде отчета, в который включить:*

    a. *В части 3 привести описание структур данных, представляющих результат разбора текста (3а)*

    b. *В части 4 описать, какая дополнительная обработка потребовалась для результата разбора, предоставляемого средством синтаксического анализа, чтобы сформировать результат работы созданного модуля*

    c. *В части 5 привести примеры исходных анализируемых текстов для всех синтаксических конструкций разбираемого языка и соответствующие результаты разбора*

Описание работы

Работа выполнена с использованием **bison** и **flex**. Включает в себя лексический парсер, синтаксический анализатор и метод для обхода синтаксического дерева (DFS).

Реализована грамматика по заданию:

```
source: sourceItem*;

typeRef: {
      |builtin: 'bool'|'byte'|'int'|'uint'|'long'|'ulong'|'char'|'string';
      |custom: identifier;
      |array: 'array' '[' (',')* ']' 'of' typeRef;
};

funcSignature: identifier '(' list<argDef> ')' (':' typeRef)? {
      argDef: identifier (':' typeRef)?;
};

sourceItem: {
      |funcDef: 'method' funcSignature (body|';') {
            body: ('var' (list<identifier> (':' typeRef)? ';')*)? statement.block;
      };

statement: {
      |if: 'if' expr 'then' statement ('else' statement)?;
      |block: 'begin' statement* 'end' ';';
      |while: 'while' expr 'do' statement;
      |do: 'repeat' statement ('while'|'until') expr ';';
      |break: 'break' ';';
      |expression: expr ';';
};

expr: { // присваивание через ':='
      |binary: expr binOp expr; // где binOp - символ бинарного оператора
      |unary: unOp expr; // где unOp - символ унарного оператора
      |braces: '(' expr ')';
      |call: expr '(' list<expr> ')';
      |indexer: expr '[' list<expr> ']';
      |place: identifier;
      |literal: bool|str|char|hex|bits|dec;
};
```

На вход подается файл с программным кодом, соответствующий грамматике, например:

```
method main(args) : int
var a : int, b : array [1, 2, 3] of int , c : str;
begin
  if a >= b then c := a; else c := 0;
  while (true) do
  begin
    repeat a := b - c;
    while a >= b;
    break;
  end
  call(expr1, expr2);
  indexer [e1, e2];
end
```

В результате работы программа выводит в консоль AST-дерево:
Аспектыреализации

Структура для хранения узла AST-дерева:
Функция обхода дерева (DFS):

Src/

Ast.c :

```
#include <stdbool.h>

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

// #include "ast.h"

#include "dfs.h"

extern int is_verbose;

static int nr_spaces = 0;

const char *ot_symbol[] = {

    [OT_BIN_PLUS]  = "+",  [OT_MINUS] = "-",

    [OT_BIN_MUL]  = "*",   [OT_BIN_DIV]  = "/",

    [OT_BIN_LESS]  = "=<", [OT_BIN_GREATER]  = ">=", [OT_BIN_EQUALS]  = "==",

    [OT_ASSIGN]  = ":=",

    [OT_UN_NOT]  = "not"};

static struct ast_node *make_node(enum ast_node_type type) {

  struct ast_node *node = (struct ast_node *)malloc(sizeof(struct ast_node));

  if (node == NULL) {

    fprintf(stdout, "%s: MEMORY ALLOCATION ERR\n", __func__);

    exit(1);

  }

  node->type = type;

  node->is_visited  = false;

  is_verbose  &&fprintf(stdout, "%s: ", ant_names[type]);

  return node;
```

```
}

struct ast_node *make_typeref_array(struct ast_node *node_arr) {
  struct ast_node *node = make_node(T_TYPE_REF);
  node->as_typeref.type = TR_ARR;
  node->as_typeref.sub_field = node_arr;
  return node;
}
struct ast_node *make_typeref_ident(struct ast_node *node_ident) {
  struct ast_node *node = make_node(T_TYPE_REF);
  node->as_typeref.type = TR_IDENT;
  node->as_typeref.sub_field = node_ident;
  return node;
}
struct ast_node *make_typeref(enum type_ref type) {
  struct ast_node *node = make_node(T_TYPE_REF);
  node->as_typeref.type = type;
  return node;
}
struct ast_node *make_typeref_lt(enum literal_type type) {
  struct ast_node *node = make_node(T_TYPE_REF_LT);
  node->as_typeref_lt.type = type;
  return node;
}
struct ast_node *make_ident(const char *name, struct ast_node *typeref_node) {
  struct ast_node *ident = make_node(T_IDENT);
  strncpy(ident->as_ident.name, name, MAXIMUM_IDENTIFIER_LENGTH);
  ident->as_ident.type = typeref_node;
  is_verbose &&fprintf(stdout, "[name: %s]\n", name);
  return ident;
}
```

```
struct ast_node *make_func_sign(struct ast_node *name,
                                struct ast_node *arg_list) {
  struct ast_node *fs = make_node(T_FUNC_SIGN);
  fs->as_func_sign.ident = name;
  fs->as_func_sign.arg_list = arg_list;
  return fs;
}
struct ast_node *make_func(struct ast_node *sign, struct ast_node *body) {
  struct ast_node *func = make_node(T_FUNC);
  func->as_func.func_sign = sign;
  func->as_func.body = body;
  return func;
}
struct ast_node *make_body(struct ast_node *var_list,
                           struct ast_node *statement) {
  struct ast_node *body = make_node(T_BODY);
  body->as_body.var_list = var_list;
  body->as_body.statement = statement;
  return body;
}

struct ast_node *set_type_array(struct ast_node *node, struct ast_node *type) {
  // struct ast_node *body = make_node(T_BODY);
  struct ast_node *iter = node->as_list.current;

  return node;
}

struct ast_node *make_literal(const char *value, struct ast_node *lt) {
  struct ast_node *constant = make_node(T_LITERAL);
```

```
  constant->as_literal.type = lt;
  constant->as_literal.value = value;
  is_verbose &&fprintf(stdout, "[op: %s]\n", value);
  return constant;
}
struct ast_node *make_expr(struct ast_node *node) {
  struct ast_node *expr = make_node(T_EXPR);
  expr->as_expr.some_node = node;
  return expr;
}

struct ast_node *make_unexpr(enum operation_type op, struct ast_node *arg) {
  struct ast_node *unexpr = make_node(T_UN_EXPR);

  unexpr->as_unexpr.op = op;
  unexpr->as_unexpr.argument = arg;
  is_verbose &&fprintf(stdout, "[op: %s]\n", ot_symbol[op]);
  return unexpr;
}
struct ast_node *make_binexpr(enum operation_type op, struct ast_node *arg1,
                              struct ast_node *arg2) {
  struct ast_node *binexpr = make_node(T_BIN_EXPR);
  binexpr->as_binexpr.op = op;
  binexpr->as_binexpr.arg1 = arg1;
  binexpr->as_binexpr.arg2 = arg2;
  return binexpr;
}

struct ast_node *make_expr_call(struct ast_node *expr,
                                struct ast_node *expr_list) {
  struct ast_node *node = make_node(T_CALL_EXPR);
```

```
  node->as_call_expr.expr  = expr;
  node->as_call_expr.expr_list  = expr_list;
  return expr;
}

struct ast_node *make_break() {
  struct ast_node *node = make_node(T_BREAK);
  return node;
}
// struct ast_node *make_expr_indexer(struct  ast_node *expr,
//                                    struct ast_node *expr_list) {
//   return expr;
// }
struct ast_node *make_branch(struct  ast_node *test, struct ast_node *consequent,
                             struct ast_node *alternate) {
  struct ast_node *br = make_node(T_BRANCH);
  br->as_branch.test  = test;
  br->as_branch.consequent  = consequent;
  br->as_branch.alternate  = alternate;
  return br;
}

struct ast_node *make_while(struct  ast_node *test, struct ast_node *body) {
  struct ast_node *wh = make_node(T_WHILE);
  wh->as_repeat.test  = test;
  wh->as_repeat.body  = body;
  return wh;
}

struct ast_node *make_repeat(struct  ast_node *test, struct ast_node *body) {
  struct ast_node *repeat  = make_node(T_REPEAT);
```

```
  repeat->as_repeat.test  = test;
  repeat->as_repeat.body  = body;
  // is_verbose  && fprintf(stdout,  ANT_REPEAT_FMT  "\n", (long unsigned int)
  // test, (long unsigned int) body);
  return repeat;
}
struct ast_node *make_expr_indexer(struct  ast_node *expr,
                                   struct ast_node *expr_list) {
  struct ast_node *indx = make_node(T_INDEXER);
  indx->as_indexer.expr  = expr;
  indx->as_indexer.expr_list  = expr_list;
  return indx;
}
#define MAKE_LIST(type,  head, next)                                    \
  struct ast_node *list = make_node(type);                              \
  list->as_list.current = head;                                         \
  list->as_list.next = next;                                            \
  return list;
#define INSERT_LIST(head_p,  first, func)                               \
  if (*head_p)  {                                                       \
    struct ast_node  *const oldstart =                                  \
        func((*head_p)->as_list.current,  (*head_p)->as_list.next);     \
    free(*head_p);                                                      \
    (*head_p)  = func(first, oldstart);                                 \
  } else                                                                \
    *head_p = func(first, NULL);                                        \
  return *head_p;

struct ast_node  *insert_stat_list(struct ast_node **head_p,  // maybe void?
                                   struct ast_node *first) {
  INSERT_LIST(head_p,  first, make_stat_list);
```

```
}

struct ast_node *make_stat_list(struct ast_node *head, struct ast_node *next) {
  MAKE_LIST(T_STMTS_LIST, head, next);
}

struct ast_node *make_statement(struct ast_node *node) {
  struct ast_node *stat = make_node(T_STMT);
  stat->as_statement.some_node = node;
  return stat;
}
struct ast_node *make_argdef_list(struct ast_node *head,
                                  struct ast_node *next) {
  MAKE_LIST(T_ARGDEF_LIST, head, next);
}
struct ast_node *insert_argdef_list(struct ast_node **head_p,
                                    struct ast_node *first) {
  INSERT_LIST(head_p, first, make_argdef_list);
}
struct ast_node *make_array(struct ast_node *head, struct ast_node *next) {
  MAKE_LIST(T_ARRAY, head, next);
}
struct ast_node *insert_array(struct ast_node **head_p,
                              struct ast_node *first) {
  INSERT_LIST(head_p, first, make_array);
}
struct ast_node *insert_expr_list(struct ast_node **head_p,
                                  struct ast_node *first) {
  INSERT_LIST(head_p, first, make_expr_list);
}
struct ast_node *make_expr_list(struct ast_node *head, struct ast_node *next) {
```

```
  MAKE_LIST(T_EXPR_LIST,  head, next);
}

struct ast_node *make_literal_list(struct ast_node *head,
                                   struct ast_node *next) {
  MAKE_LIST(T_LIT_LIST,  head, next);
}
struct ast_node *insert_literal_list(struct ast_node **head_p,
                                    struct ast_node *first) {
  INSERT_LIST(head_p,  first, make_literal_list);
}
struct ast_node *make_program(struct ast_node *child) {
  struct ast_node *program = make_node(T_PROGRAM);
  program->as_program.child = child;
  return program;
}
void free_ast(struct ast_node *);
static void postprint_ast() { nr_spaces -= 2; }
static void preprint_ast(struct ast_node *node) {
  nr_spaces += 2;
  if (nr_spaces >= 0) {
    for (size_t i = 0; i < nr_spaces; ++i) {
      fprintf(stdout,  " ");
    }
  }
  fprintf(stdout,  "%s ", ant_names[node->type]);
  switch (node->type) {
  case T_PROGRAM: {
    fprintf(stdout,  "\n");
    break;
  }
```

```
case T_EXPR_LIST:
case T_ARGDEF_LIST:
case T_ARRAY:
case T_LIT_LIST:
case T_STMTS_LIST:  {
  fprintf(stdout,  "\n");
  break;
}
case T_REPEAT:  {
  fprintf(stdout,  "\n");
  break;
}
case T_LITERAL:  {
  fprintf(stdout,  "[val: %s, type: %s]\n", node->as_literal.value,
          literal_names[node->as_literal.type->as_typeref.type]);
  break;
}
case T_IDENT:  {
  fprintf(stdout,  "[name: %s]\n", node->as_ident.name);
  // fprintf(stdout,  "[type_ref: %s]\n", typeref_names[node->as_ident.type]);

  break;
}
case T_TYPE_REF:  {
  fprintf(stdout,  "[type: %s]\n", typeref_names[node->as_typeref.type]);

  break;
}
case T_INDEXER:
case T_CALL_EXPR:
case T_BREAK:
```

```
  case T_EXPR:
  case T_WHILE:
  case T_BRANCH:
  case T_STMT:
  case T_BODY:
  case T_FUNC:
  case T_FUNC_SIGN:  {
    fprintf(stdout,  "\n");
    break;
  }
  case T_UN_EXPR:  {
    fprintf(stdout,  "[type: %s]\n", ot_symbol[node->as_unexpr.op]);
    break;
  }
  case T_BIN_EXPR:  {
    fprintf(stdout,  "[type: %s]\n", ot_symbol[node->as_binexpr.op]);
    break;
  }
  default:  {
    fprintf(stdout,  "<unknown-node>\n");
    break;
  }
  }
}


int print_ast(struct ast_node *root) {
  dfs_bypass(root,  preprint_ast,  postprint_ast);
  return 1;
}
```

dfc.c:

```c
#include "dfs.h"
#include <stddef.h>
#include <stdio.h>

void dfs_bypass(struct ast_node *node, process_cb preproccess_cb,
                process_cb postprocess_cb) {
  if (node == NULL)
    return;

  node->is_visited = true;
  preproccess_cb(node);
  switch (node->type) {
  case T_PROGRAM: {
    dfs_bypass(node->as_program.child, preproccess_cb, postprocess_cb);
    break;
  }
  case T_EXPR_LIST:
  case T_ARGDEF_LIST:
  case T_ARRAY:
  case T_LIT_LIST:
  case T_STMTS_LIST: {
    struct ast_node *iter = node;
    while (iter != NULL) {
      dfs_bypass(iter->as_list.current, preproccess_cb, postprocess_cb);
      struct ast_node *temp = iter;
      iter = iter->as_list.next;
      // postprocess_cb(temp);
    }
    return;
    // break;
  }
  case T_WHILE:
  case T_REPEAT: {
    dfs_bypass(node->as_repeat.test, preproccess_cb, postprocess_cb);
    dfs_bypass(node->as_repeat.body, preproccess_cb, postprocess_cb);
    break;
  }
  case T_BRANCH: {
    dfs_bypass(node->as_branch.test, preproccess_cb,
               postprocess_cb);
    dfs_bypass(node->as_branch.consequent, preproccess_cb, postprocess_cb);
    dfs_bypass(node->as_branch.alternate, preproccess_cb, postprocess_cb);
    break;
  }
```

```
  case T_BIN_EXPR: {
    dfs_bypass(node->as_binexpr.arg1, preproccess_cb, postprocess_cb);
    dfs_bypass(node->as_binexpr.arg2, preproccess_cb, postprocess_cb);

    break;
  }
  case T_UN_EXPR: {
    dfs_bypass(node->as_unexpr.argument, preproccess_cb, postprocess_cb);
    break;
  }
  case T_INDEXER: {
    dfs_bypass(node->as_indexer.expr, preproccess_cb, postprocess_cb);
    dfs_bypass(node->as_indexer.expr_list, preproccess_cb, postprocess_cb);
    break;
  }
  case T_CALL_EXPR: {
    dfs_bypass(node->as_call_expr.expr, preproccess_cb, postprocess_cb);
    dfs_bypass(node->as_call_expr.expr_list, preproccess_cb, postprocess_cb);
    break;
  }
  case T_FUNC_SIGN: {
    dfs_bypass(node->as_func_sign.ident, preproccess_cb, postprocess_cb);
    dfs_bypass(node->as_func_sign.arg_list, preproccess_cb, postprocess_cb);
    break;
  }
  case T_FUNC: {
    dfs_bypass(node->as_func.func_sign, preproccess_cb, postprocess_cb);
    dfs_bypass(node->as_func.body, preproccess_cb, postprocess_cb);
    break;
  }
  case T_IDENT: {
    dfs_bypass(node->as_ident.type, preproccess_cb, postprocess_cb);
    break;
  }
  case T_BODY: {
    dfs_bypass(node->as_body.var_list, preproccess_cb,
               postprocess_cb);
    dfs_bypass(node->as_body.statement, preproccess_cb, postprocess_cb);
    break;
  }

  case T_STMT: {
    dfs_bypass(node->as_statement.some_node, preproccess_cb, postprocess_cb);
    break;
  }
  case T_EXPR: {
    dfs_bypass(node->as_expr.some_node, preproccess_cb, postprocess_cb);
    break;
```

```
  }
  case T_TYPE_REF: {
    if (node->as_typeref.type == TR_IDENT || node->as_typeref.type == TR_ARR)
{
      dfs_bypass(node->as_typeref.sub_field, preproccess_cb, postprocess_cb);
    }
    break;
  }
  default: {
    break;
  }
  }
  postprocess_cb(node);
  return;
}
```

bison.tab.c :

```
int main( int argc, char** argv )
{
  int optidx =
0;
  int is_input =
0;
  struct option options[] =
{
    { "file",    required_argument, NULL, 'f'
},
    { "help",    no_argument,       NULL, 'h'
},
    { 0, 0, 0, 0
}
  };



  char
brief_option;

  while ( -1 != (brief_option = getopt_long( argc, argv, "vhf:o:", options,
&optidx )) )
    switch ( brief_option )
{
      case
'h':
```

```
        fprintf(
stdout,
          "SYNOPSYS"

          "\n\tspc [-v] [-f <input
file>]"
          "\nDESCRIPTION"

          "\n\t-h, --
help"
          "\n\t\tshows this help message and
exits"
          "\n\t-f, --
file"
          "\n\t\tspecifies the input file path, default:
stdin"
        );

        return
0;
      case
'f':
        yyin = fopen( optarg, "r" ); is_input = 1;
break;
      default: {
      printf("can't open file");
      exit(-1);
        }
    }



  bool parse_result = yyparse(
);
  if ( is_input
)
    fclose( yyin
);


  // traverse AST for generating
TAC
  if ( root == NULL ) printf("invalid syntax\n");

   //is_verbose && fprintf( stdout, "can't find root\n"
);
  else {
```

```
   print_ast( root
);
  //  print_tac( root
);
  }

  //

  //// free the
ast
  //free_ast( root
);
  return
parse_result;

}
```

Makefile :

```
Bison = C:\ProgramData\chocolatey\bin\win_bison.exe
Flex = C:\ProgramData\chocolatey\bin\win_flex.exe

all: bison.tab.o lex.yy.o ast.o dfs.o
    gcc lex.yy.o bison.tab.o ast.o dfs.o -o lab1

bison.tab.o: bison.tab.c
    gcc -c bison.tab.c -o bison.tab.o

lex.yy.o: lex.yy.c
    gcc -c lex.yy.c -o lex.yy.o

bison.tab.c:
    $(Bison) -Wcounterexamples -d  bison.y

lex.yy.c:
    $(Flex) flex.l

ast.o:
    gcc -c ./src/ast.c -o ast.o

dfs.o:
    gcc -c ./src/dfs.c -o dfs.o

clear:
    del /f bison.tab.h bison.tab.c lex.yy.c bison.tab.o lex.yy.o ast.o dfs.o
bison.gv out.svg lab1
```

Результаты

Результатом выполненной работы является программа, которая способна создавать и выводить в консоль абстрактное синтаксическое дерево (AST) для кода, соответствующего заданной грамматике.

Output.txt:

```
program
 func
  func-sign
   identifier [name: main]
    type-ref [type: int]
   arg-def-list
    identifier [name: args]
     type-ref [type: none]
   body
    arg-def-list
     identifier [name: a]
      type-ref [type: int]
     identifier [name: b]
      type-ref [type: array]
       array
        literal [val: 1, type: dec]
        literal [val: 2, type: dec]
        literal [val: 3, type: dec]
      identifier [name: c]
       type-ref [type: ident]
        identifier [name: str]
         type-ref [type: none]
      statements-list
       statement
        repeat-until
```

```
    statement
      repeat-until
        statement
          binary-expression  [type: -]
            binary-expression  [type: :=]
              expression
                identifier  [name: a]
                  type-ref [type: none]
              expression
                identifier  [name: a]
                  type-ref [type: none]
            expression
              literal [val: 1, type: dec]
        binary-expression  [type: >=]
          expression
            identifier  [name: a]
              type-ref [type: none]
          expression
            literal [val: 0, type: dec]
    unary-expression  [type: not]
      binary-expression  [type: ==]
        expression
          identifier  [name: c]
            type-ref [type: none]
        expression
          literal [val: "hi", type: str]
statement
  branch
    binary-expression  [type: >=]
      expression
        identifier  [name: x]
```

```
      type-ref [type: none]
    expression
      identifier [name: y]
      type-ref [type: none]
  statement
    statements-list
      statement
        binary-expression [type: :=]
          expression
            identifier [name: x]
              type-ref [type: none]
          expression
            literal [val: 2, type: dec]
      statement
        binary-expression [type: *]
          binary-expression [type: :=]
            expression
              identifier [name: y]
                type-ref [type: none]
            expression
              literal [val: "123", type: str]
          expression
            identifier [name: x]
              type-ref [type: none]
      statement
        expression
          identifier [name: passLab]
            type-ref [type: none]
      statement
        break
    statement
```

```
branch
 binary-expression [type: =<]
  expression
   identifier [name: x]
    type-ref [type: none]
  expression
   identifier [name: y]
    type-ref [type: none]
 statement
  binary-expression [type: +]
   binary-expression [type: :=]
    expression
     identifier [name: a]
      type-ref [type: none]
    expression
     identifier [name: x]
      type-ref [type: none]
   expression
    identifier [name: y]
     type-ref [type: none]
```

Выводы

В процессе выполнения первой практической работы я освоил методы построения AST-дерева кода и создал программу, используя bison и flex. Это было непросто для меня, поскольку изначально у меня не было достаточных навыков для выполнения задания.