

Міністерство освіти і науки України

Національний технічний університет України

«Київський політехнічний інститут»

***Лабораторна робота №2***

***з дисципліни «ОСНОВИ ПРОЕКТУВАННЯ ТРАНСЛЯТОРІВ»***

**«РОЗРОБКА ГЕНЕРАТОРА КОДУ»**

Виконав студент групи: КВ-11

ПІБ: Терентьєв Іван Дмитрович

Перевірив: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Київ 2024**

*Постановка задачі*

1. Розробити програму генератора коду (ГК) для підмножини мови програмування SIGNAL, заданої за варіантом.
2. Програма генератора коду має забезпечувати:

* читання дерева розбору та таблиць, створених синтаксичним аналізатором, що було розроблено в розрахунково-графічній роботі;
* виявлення семантичних помилок;
* генерацію коду та/або побудову внутрішніх таблиць для генерації коду.

1. Скомпонувати повний компілятор, що складається з розроблених раніше лексичного та синтаксичного аналізаторів і генератора коду, який забезпечує наступне:

* генерацію коду та/або побудову внутрішніх таблиць для генерації коду;
* формування лістингу вхідної програми з повідомленнями про лексичні, синтаксичні та семантичні помилки.

*Граматика за варіантом 21*

**<signal-program>** --> <program>

**<program>** --> PROGRAM <procedure-identifier> ; <block>.

**<block>** --> <declarations> BEGIN <statements-list> END

**<declarations>** --> <constant-declarations>

**<constant-declarations>** --> CONST <constantdeclarations-list> | <empty>

**<constant-declarations-list>** --> <constantdeclaration> <constant-declarations-list> | <empty>

**<constant-declaration>** --> <constant-identifier> = <constant>;

**<statements-list>** --> <statement> <statements-list> | <empty>

**<statement>** --> CASE <expression> OF <alternativeslist> ENDCASE ;

**<alternatives-list>** --> <alternative> <alternativeslist> |<empty>

**<alternative>** --> <expression> : /<statements-list>\

**<expression>** --> <summand> <summands-list> | - <summand> <summands-list>

**<summands-list>** --> <add-instruction> <summand> |<summands-list> | <empty>

**<add-instruction>** --> + | -

**<summand>** --> <variable-identifier> | <unsigned-integer>

**<constant>** --> <unsigned-integer>

**<variable-identifier>** --> <identifier>

**<constant-identifier>** --> <identifier>

**<procedure-identifier>** --> <identifier>

**<identifier>** --> <letter><string>

**<string>** --> <letter><string> | <digit><string> |<empty>

**<unsigned-integer>** --> <digit><digits-string>

**<digits-string>** --> <digit><digits-string> | <empty>

**<digit>** --> 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

**<letter>** --> A | B | C | D | ... | Z

*Лістинг програми*

==> main.c <==

#include "lexer.h"

#include "out.h"

#include "semant.h"

#include "verify.h"

int main(int argc, char \*argv[]) {

  proc\_cli(argc, argv);

  if (gotError) {

    print\_errors();

    return -1;

  } else

    proc\_lexer(params.\_input\_file);

  if (params.out\_lexer) {

    if (params.verbose) {

      out\_file\_lexer();

      print\_file\_out();

    } else

      out\_file\_lexer();

  }

  if (gotError) {

    print\_errors();

    return -1;

  } else {

    just\_clean();

    proc\_syntax();

  }

  if (params.out\_syntax) {

    if (params.verbose) {

      out\_file\_syntax();

      print\_file\_out();

    } else

      out\_file\_syntax();

  }

  if (gotError)

  {

    print\_errors();

    return -1;

  } else{

    just\_clean();

    proc\_semant();

  }

  if (params.out\_codegen)

  {

    if(params.verbose)

    {

      out\_file\_codegen();

      print\_file\_out();

    } else

      out\_file\_codegen();

  }

  free\_trees();

  free\_errors();

  free\_tables();

  free\_tokens();

  if (params.\_verify\_file != NULL) {

    verify(params.\_output\_file, params.\_verify\_file);

  }

  return 0;

}

==> lexer\_state/constant.h <==

#include <stdbool.h>

#include <stddef.h>

#include "token\_structure.h"

#ifndef CONSTANT\_H

#define CONSTANT\_H

typedef struct token Constant;

extern Constant \*\_constants;

extern size\_t constantCount;

void add\_to\_constants(Constant constant);

bool is\_constant(size\_t tokenCode);

#endif

==> lexer\_state/id\_generator.h <==

#include <stddef.h>

#include <stdlib.h>

#ifndef ID\_GENERATOR\_H

#define ID\_GENERATOR\_H

size\_t get\_id(size\_t row, size\_t col, unsigned short int type);

#endif

==> lexer\_state/identifier.h <==

#include "error.h"

#include "token\_structure.h"

#ifndef IDENTIFIER\_H

#define IDENTIFIER\_H

typedef struct token Identifier;

extern Identifier \*\_identifiers;

extern size\_t identifierCount;

void add\_to\_identifiers(Identifier identifier);

bool is\_identifier(size\_t tokenCode);

#endif

==> lexer\_state/lexer.h <==

#ifndef LEXER\_H

#define LEXER\_H

#include "lexer\_get.h"

// Main procedure of lexer

void proc\_lexer(char \*\_input\_file);

#endif

==> lexer\_state/lexer\_get.h <==

#include <stdio.h>

#include <stdlib.h>

#include "error.h"

#include "lexer\_structure.h"

#include "token.h"

#ifndef LEXER\_GET\_H

#define LEXER\_GET\_H

void inp(FILE \*\_\_input\_file);

void ws(FILE \*\_\_input\_file);

void dig(FILE \*\_\_input\_file);

void let(FILE \*\_\_input\_file);

void dm1(FILE \*\_\_input\_file);

void dm2(FILE \*\_\_input\_file);

void com\_begin(FILE \*\_\_input\_file);

void com\_confirm(FILE \*\_\_input\_file, size\_t row, size\_t col);

void com\_ending(FILE \*\_\_input\_file, size\_t row, size\_t col);

void s\_error(FILE \*\_\_input\_file);

#endif

==> lexer\_state/lexer\_structure.h <==

#include <stdbool.h>

#include <stddef.h>

#ifndef LEXER\_STRUCTURE\_H

#define LEXER\_STRUCTURE\_H

struct lexer {

  char \*\_buffer;

  size\_t bufferSize;

  size\_t row;

  size\_t col;

  char symbol;

  unsigned short int symbolType;

  bool inComment;

};

typedef struct lexer Lexer;

extern Lexer lexer;

void add\_buffer\_symbol();

void clean\_buffer();

#endif

==> lexer\_state/strings.h <==

#include <stdbool.h>

#include <stddef.h>

#include "token\_structure.h"

#ifndef STRINGS\_H

#define STRINGS\_H

typedef struct token Stringy;

extern Stringy \*\_strings;

extern size\_t stringsCount;

void add\_to\_strings(Stringy str);

bool is\_stringy(size\_t tokenCode);

#endif

==> lexer\_state/symbol\_type.h <==

#ifndef SYMBOL\_TYPE\_H

#define SYMBOL\_TYPE\_H

/\*

@symbolType

\*/

#define SYMBOL\_START 0

#define SYMBOL\_WS 1

#define SYMBOL\_DIG 2

#define SYMBOL\_LET 3

#define SYMBOL\_DM1 4

#define SYMBOL\_DM2 5

#define SYMBOL\_COM\_BEGIN 6

#define SYMBOL\_COM\_CONFIRM 7

#define SYMBOL\_COM\_ENDING 8

#define SYMBOL\_ERROR 10 // 0xA Unknown symbol

#define SYMBOL\_EOF 11   // 0xB End of file symbol

unsigned short int symbol\_type(char symbol);

#endif

==> lexer\_state/token.h <==

#include "error.h"

#include "token\_structure.h"

#ifndef TOKEN\_H

#define TOKEN\_H

extern Token \*\_tokens;

extern size\_t tokenCount;

void add\_to\_tokens(Token token);

#endif

==> lexer\_state/token\_structure.h <==

#include <stddef.h>

#include <stdlib.h>

#ifndef TOKEN\_STRUCTURE\_H

#define TOKEN\_STRUCTURE\_H

struct token {

  size\_t row;

  size\_t col;

  size\_t code;

  char \*\_data;

  size\_t dataSize;

};

typedef struct token Token;

Token create\_token(size\_t row, size\_t col, char \*\_data, size\_t dataSize,

                   unsigned short int type);

Token create\_token\_with\_code(size\_t row, size\_t col, char \*\_data,

                             size\_t dataSize, size\_t code);

#endif

==> semant\_state/semant.h <==

#include <stdlib.h>

#include "syntax.h"

#ifndef SEMANT\_H

#define SEMANT\_H

struct var{

    char\* name;

    char\* value;

};

typedef struct var Var;

struct cnst{

    char\* name;

    char\* value;

};

typedef struct cnst Const;

extern Const \*consts;

extern size\_t constCount;

extern Var \*vars;

extern size\_t varsCount;

extern char \*\*statementsCode;

extern size\_t codeCount;

extern char\* program\_name;

void proc\_semant();

void generate\_final\_output();

bool iAmInConst(char \*v);

bool iAmProgram(char \*v);

bool iAmInVars(char \*v);

void add\_to\_const(Const c);

void add\_to\_vars(Var v);

void add\_to\_statements(char \*value);

char \*\*add\_to\_semant\_final\_program(char \*value);

extern size\_t skip;

Tree \*find\_in\_tree(Tree \*cur\_tree, char \*value);

extern char \*\*semant\_final;

extern size\_t semant\_final\_count;

#endif

==> syntax\_state/knut\_tables.h <==

#ifndef KNUT\_TABLES\_H

#define KNUT\_TABLES\_H

#include <stdbool.h>

#include <stdlib.h>

struct code {

  size\_t addrTo;

  char \*\_term;

  bool isTerm;

};

typedef struct code Code;

struct line {

  size\_t addr;

  Code code;

  bool atAddr;

  size\_t afAddr;

};

typedef struct line Line;

struct table {

  size\_t linesCount;

  Line \*lines;

};

typedef struct table Table;

Table create\_knut\_table();

char \*name\_by\_id(size\_t addr);

#endif

==> syntax\_state/syntax.h <==

#include "knut\_tables.h"

#include "tree.h"

#ifndef SYNTAX\_H

#define SYNTAX\_H

extern Tree \*\_tree;

void proc\_syntax();

struct probably {

  Tree \*result;

  bool status;

};

typedef struct probably ProbablyResults;

ProbablyResults probe(Table table, size\_t i);

Line ruler(Table table, size\_t k);

#define rules(i) ruler(table, i)

#endif

==> syntax\_state/terms.h <==

#ifndef TERMS\_H

#define TERMS\_H

#define SIGNAL\_PROGRAM 0

#define SIGNAL\_PROGRAM\_FINISH 2

#define PROGRAM 3

#define PROGRAM\_ENDING 7

#define BLOCK 8

#define DECLARATIONS 12

#define CONSTANT\_DECLARATIONS 13

#define CONSTANT\_DECLARATIONS\_LIST 16

#define CONSTANT\_DECLARATION 19

#define STATEMENTS\_LIST 23

#define STATEMENT 26

#define ALTERNATIVES\_LIST 36

#define ALTERNATIVE 39

#define EXPRESSION 44

#define SUMMANDS\_LIST 49

#define ADD\_INSTRUCTION 55

#define SUMMAND 57

#define CONSTANT 59

#define VARIABLE\_IDENTIFIER 60

#define CONSTANT\_IDENTIFIER 61

#define PROCEDURE\_IDENTIFIER 62

#define ERROR 666

#define OK 777

/\*

<identifier> --> <letter><string>                                  id>1000

<string> --> <letter><string> | <digit><string> | <empty>          id>750

<unsigned-integer> --> <digit><digits-string>                      id>500

<digits-string> --> <digit><digits-string> | <empty>

<digit> --> 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<letter> --> A | B | C | D | ... | Z

\*/

#define IDENTIFIER 100

#define UNSIGNED\_INTEGER 101

#define STRING 102

#define EMPTY 200

#endif

==> syntax\_state/tree.h <==

#include <stdlib.h>

#ifndef TREE\_BUILDER\_H

#define TREE\_BUILDER\_H

struct tree {

  char \*\_value;

  struct tree \*\*\_branches;

  size\_t branchesCount;

  size\_t id;

};

typedef struct tree Tree;

Tree \*create\_node(char \*\_value, size\_t id);

void add\_branch(Tree \*\_origin, Tree \*\_tree);

void free\_tree(Tree \*\_tree);

/\* add\_branch defines\*/

#define add\_branch\_with\_token(token)                                           \

  do {                                                                         \

    add\_branch(newTree, token);                                                \

    state = true;                                                              \

  } while (0)

#define add\_branch\_def\_token()                                                 \

  add\_branch\_with\_token(create\_node(\_tokens[tokenIterator].\_data, i))

#define add\_branch\_empty() add\_branch\_with\_token(create\_node("<empty>", i))

#endif

==> util/cli.h <==

#ifndef CLI\_H

#define CLI\_H

#include "error.h"

struct params {

  char \*\_input\_file;

  char \*\_output\_file;

  bool verbose;

  bool out\_lexer;

  bool out\_syntax;

  bool out\_codegen;

  char \*\_verify\_file;

};

typedef struct params Params;

extern Params params;

void proc\_cli(int argc, char \*argv[]);

#endif

==> util/error.h <==

#include <stdbool.h>

#include <stddef.h>

#include <stdlib.h>

#ifndef ERROR\_H

#define ERROR\_H

struct error {

  size\_t number;

  unsigned short int state;

  char \*\_error\_message;

  bool critical;

  bool hasLineColumn;

  size\_t row;

  size\_t col;

  char \*\_expected;

  char \*\_here;

  bool syntaxer;

};

typedef struct error Error;

/\*

@state

\*/

#define NOT\_ERROR 0

#define FILE\_ACCESS 1

#define MEMORY\_ACCESS 2

#define LEXER\_STATE 3

#define SYNTAX\_STATE 4

#define SEMANT\_STATE 5

extern Error \*\_errors;

extern size\_t errorCount;

extern bool gotError;

extern bool gotWarning;

Error create\_error\_syntaxer(size\_t row, size\_t col, char \*\_expected,

                            char \*\_here);

Error create\_error\_without\_linecolumn(unsigned short int state,

                                      char \*\_error\_message, bool critical);

Error create\_error\_with\_linecolumn(unsigned short int state,

                                   char \*\_error\_message, bool critical,

                                   size\_t row, size\_t col);

Error create\_error\_def();

void add\_to\_errors(Error error);

bool has\_critical();

void clean\_errors();

#endif

==> util/out.h <==

#include "cli.h"

#include "error.h"

#include "token\_structure.h"

#ifndef OUT\_H

#define OUT\_H

void print\_params();

void print\_error(Error error);

void print\_errors();

void print\_lexer();

void print\_token(Token token);

void print\_tokens();

void out\_file\_lexer();

void print\_file\_out();

void out\_file\_errors();

void out\_file\_syntax();

void out\_file\_codegen();

void just\_clean();

void free\_trees();

void free\_errors();

void free\_tokens();

void free\_tables();

#endif

==> util/verify.h <==

#ifndef VERIFY\_H

#define VERIFY\_H

void verify(char \*\_output, char \*\_verify);

#endif

==> lexer\_state/constant.c <==

#include <stdbool.h>

#include "constant.h"

#include "error.h"

#include "token\_structure.h"

Constant \*\_constants = NULL;

size\_t constantCount = 0;

void add\_to\_constants(Constant constant) {

  constantCount++;

  \_constants = (Token \*)realloc(\_constants, constantCount \* sizeof(Token));

  if (\_constants == NULL)

    add\_to\_errors(create\_error\_with\_linecolumn(

        MEMORY\_ACCESS, "Cannot reallocate \*\_constants", true, constant.row,

        constant.col));

  else

    \_constants[constantCount - 1] = constant;

}

bool is\_constant(size\_t tokenCode) {

  for (size\_t i = 0; i < constantCount; i++)

    if (tokenCode == \_constants[i].code)

      return true;

  return false;

}

==> lexer\_state/id\_generator.c <==

#include <stdbool.h>

#include <string.h>

#include "constant.h"

#include "id\_generator.h"

#include "identifier.h"

#include "lexer\_structure.h"

#include "strings.h"

#include "symbol\_type.h"

size\_t get\_keyword\_id() {

  char \*\_verify[10] = {"PROGRAM", "VAR", "BEGIN",   "END",     "CONST",

                       "CASE",    "OF",  "ENDCASE", "INTEGER", "FLOAT"};

  for (size\_t i = 0; i < 10; i++)

    if (!strcmp(lexer.\_buffer, \_verify[i]))

      return i + 1;

  return 0;

}

size\_t get\_dm1\_id() {

  char \_verify[12] = {'+', '-', ':', '<', '>',  '=',

                      '.', ';', '[', ']', '\\', '/'};

  for (unsigned short i = 0; i < 12; i++)

    if (lexer.\_buffer[0] == \_verify[i])

      return (size\_t)lexer.\_buffer[0];

  return 0;

}

size\_t get\_dm2\_id() {

  char \_verify[3] = {'<', '>', ':'};

  if (strlen(lexer.\_buffer) > 1)

    if (lexer.\_buffer[1] == '=')

      for (size\_t i = 0; i < 3; i++)

        if (lexer.\_buffer[0] == \_verify[i])

          return i + 301;

  return get\_dm1\_id();

}

size\_t get\_id(size\_t row, size\_t col, unsigned short int type) {

  size\_t base = 0;

  switch (type) {

  case SYMBOL\_DIG:

    base = 501;

    for (size\_t i = 0; i < constantCount; i++)

      if (!strcmp(lexer.\_buffer, \_constants[i].\_data))

        return \_constants[i].code;

    base += constantCount;

    add\_to\_constants(create\_token\_with\_code(row, col, lexer.\_buffer,

                                            lexer.bufferSize, base));

    break;

  case SYMBOL\_LET:

    if (get\_keyword\_id()) {

      base = 400;

      base += get\_keyword\_id();

    } else {

      if (lexer.\_buffer[0] > 64 && lexer.\_buffer[0] < 91) {

        base = 1001;

        for (size\_t i = 0; i < identifierCount; i++)

          if (!strcmp(lexer.\_buffer, \_identifiers[i].\_data))

            return \_identifiers[i].code;

        base += identifierCount;

        add\_to\_identifiers(create\_token\_with\_code(row, col, lexer.\_buffer,

                                                  lexer.bufferSize, base));

      } else {

        base = 750;

        for (size\_t i = 0; i < stringsCount; i++)

          if (!strcmp(lexer.\_buffer, \_strings[i].\_data))

            return \_strings[i].code;

        base += stringsCount;

        add\_to\_strings(create\_token\_with\_code(row, col, lexer.\_buffer,

                                              lexer.bufferSize, base));

      }

    }

    break;

  case SYMBOL\_DM1:

    base = get\_dm1\_id();

    break;

  case SYMBOL\_DM2:

    base = get\_dm2\_id();

    break;

  default:

    add\_to\_errors(create\_error\_without\_linecolumn(

        LEXER\_STATE, "Impossible for get\_code()", true));

    return 0;

  };

  return base;

}

==> lexer\_state/identifier.c <==

#include <stdbool.h>

#include "identifier.h"

#include "token\_structure.h"

Identifier \*\_identifiers = NULL;

size\_t identifierCount = 0;

void add\_to\_identifiers(Identifier identifier) {

  identifierCount++;

  \_identifiers =

      (Token \*)realloc(\_identifiers, identifierCount \* sizeof(Identifier));

  if (\_identifiers == NULL)

    add\_to\_errors(create\_error\_with\_linecolumn(

        MEMORY\_ACCESS, "Cannot reallocate \*\_identifiers", true, identifier.row,

        identifier.col));

  else

    \_identifiers[identifierCount - 1] = identifier;

}

bool is\_identifier(size\_t tokenCode) {

  for (size\_t i = 0; i < identifierCount; i++)

    if (tokenCode == \_identifiers[i].code)

      return true;

  return false;

}

==> lexer\_state/lexer.c <==

#include "lexer.h"

#include "stdlib.h"

#include "symbol\_type.h"

Lexer lexer = {NULL, 0, 1, 1, '\0', SYMBOL\_START, false};

void proc\_lexer(char \*\_input\_file) {

  FILE \*\_\_input\_file;

  \_\_input\_file = fopen(\_input\_file, "r");

  if (\_\_input\_file == NULL)

    add\_to\_errors(create\_error\_without\_linecolumn(

        FILE\_ACCESS, "Cannot open input file.", true));

  else {

    inp(\_\_input\_file);

    do {

      switch (lexer.symbolType) {

      case SYMBOL\_WS:

        ws(\_\_input\_file);

        break;

      case SYMBOL\_DIG:

        dig(\_\_input\_file);

        break;

      case SYMBOL\_LET:

        let(\_\_input\_file);

        break;

      case SYMBOL\_DM1:

        dm1(\_\_input\_file);

        break;

      case SYMBOL\_DM2:

        dm2(\_\_input\_file);

        break;

      case SYMBOL\_COM\_BEGIN:

        com\_begin(\_\_input\_file);

        break;

      case SYMBOL\_ERROR:

        s\_error(\_\_input\_file);

        break;

      case SYMBOL\_EOF:

        break;

      default:

        add\_to\_errors(create\_error\_without\_linecolumn(

            LEXER\_STATE, "Impossible if rrly, unknown category", true));

        lexer.symbolType = SYMBOL\_EOF;

        break;

      };

    } while (lexer.symbolType != SYMBOL\_EOF);

  }

  fclose(\_\_input\_file);

}

==> lexer\_state/lexer\_get.c <==

#include "lexer\_get.h"

#include "symbol\_type.h"

void inp(FILE \*\_\_input\_file) {

  lexer.symbol = (char)fgetc(\_\_input\_file);

  if (lexer.symbol == '\n') {

    lexer.row++;

    lexer.col = 1;

  } else {

    if (lexer.symbol == '\t')

      lexer.col += 4;

    else

      lexer.col++;

  }

  lexer.symbolType = symbol\_type(lexer.symbol);

}

void ws(FILE \*\_\_input\_file) {

  do

    inp(\_\_input\_file);

  while (lexer.symbolType == SYMBOL\_WS);

}

void dig(FILE \*\_\_input\_file) {

  size\_t row = lexer.row;

  size\_t col = lexer.col;

  do {

    add\_buffer\_symbol();

    inp(\_\_input\_file);

  } while (lexer.symbolType == SYMBOL\_DIG);

  add\_to\_tokens(

      create\_token(row, col, lexer.\_buffer, lexer.bufferSize, SYMBOL\_DIG));

  clean\_buffer();

}

void let(FILE \*\_\_input\_file) {

  size\_t row = lexer.row;

  size\_t col = lexer.col;

  do {

    add\_buffer\_symbol();

    inp(\_\_input\_file);

  } while (lexer.symbolType == SYMBOL\_DIG || lexer.symbolType == SYMBOL\_LET);

  add\_to\_tokens(

      create\_token(row, col, lexer.\_buffer, lexer.bufferSize, SYMBOL\_LET));

  clean\_buffer();

}

void dm1(FILE \*\_\_input\_file) {

  size\_t row = lexer.row;

  size\_t col = lexer.col;

  add\_buffer\_symbol();

  add\_to\_tokens(

      create\_token(row, col, lexer.\_buffer, lexer.bufferSize, SYMBOL\_DM1));

  clean\_buffer();

  inp(\_\_input\_file);

}

void dm2(FILE \*\_\_input\_file) {

  size\_t row = lexer.row;

  size\_t col = lexer.col;

  add\_buffer\_symbol();

  inp(\_\_input\_file);

  if (lexer.symbolType == SYMBOL\_DM1) {

    add\_buffer\_symbol();

    inp(\_\_input\_file);

  }

  add\_to\_tokens(

      create\_token(row, col, lexer.\_buffer, lexer.bufferSize, SYMBOL\_DM2));

  clean\_buffer();

}

void com\_begin(FILE \*\_\_input\_file) {

  size\_t row = lexer.row;

  size\_t col = lexer.col;

  inp(\_\_input\_file);

  if (lexer.symbol == '\*') {

    lexer.inComment = true;

    com\_confirm(\_\_input\_file, row, col);

  } else {

    add\_to\_errors(create\_error\_with\_linecolumn(LEXER\_STATE, "No \* after (",

                                               true, row, col));

    inp(\_\_input\_file);

  }

}

void com\_confirm(FILE \*\_\_input\_file, size\_t row, size\_t col) {

  inp(\_\_input\_file);

  if (lexer.symbol == '\*') {

    com\_ending(\_\_input\_file, row, col);

  } else {

    if (lexer.symbolType == 7) {

      add\_to\_errors(create\_error\_with\_linecolumn(

          LEXER\_STATE, "Not closed comment", true, row, col));

      inp(\_\_input\_file);

    } else

      com\_confirm(\_\_input\_file, row, col);

  }

}

void com\_ending(FILE \*\_\_input\_file, size\_t row, size\_t col) {

  inp(\_\_input\_file);

  if (lexer.symbol == ')') {

    inp(\_\_input\_file);

    lexer.inComment = false;

  } else {

    if (lexer.symbol == '\*')

      com\_ending(\_\_input\_file, row, col);

    else {

      if (lexer.symbolType == 7) {

        add\_to\_errors(create\_error\_with\_linecolumn(

            LEXER\_STATE, "Not closed comment", true, row, col));

        inp(\_\_input\_file);

      } else

        com\_confirm(\_\_input\_file, row, col);

    }

  }

}

void s\_error(FILE \*\_\_input\_file) {

  if (lexer.symbolType == SYMBOL\_COM\_CONFIRM ||

      lexer.symbolType == SYMBOL\_COM\_ENDING)

    add\_to\_errors(create\_error\_with\_linecolumn(

        LEXER\_STATE, "Comment is not openned or already closed", false,

        lexer.row, lexer.col));

  else

    add\_to\_errors(create\_error\_with\_linecolumn(LEXER\_STATE, "Got error symbol",

                                               true, lexer.row, lexer.col));

  inp(\_\_input\_file);

}

==> lexer\_state/lexer\_structure.c <==

#include <stdlib.h>

#include "error.h"

#include "lexer\_structure.h"

void add\_buffer\_symbol() {

  lexer.\_buffer =

      (char \*)realloc(lexer.\_buffer, (lexer.bufferSize + 2) \* sizeof(char));

  if (lexer.\_buffer == NULL)

    add\_to\_errors(create\_error\_with\_linecolumn(

        LEXER\_STATE, "Cannot resize \*buff", true, lexer.row, lexer.col));

  lexer.\_buffer[lexer.bufferSize] = lexer.symbol;

  lexer.\_buffer[lexer.bufferSize + 1] = '\0';

  lexer.bufferSize++;

}

void clean\_buffer() {

  lexer.\_buffer = NULL;

  lexer.bufferSize = 0;

}

==> lexer\_state/strings.c <==

#include <stdbool.h>

#include "error.h"

#include "strings.h"

#include "token\_structure.h"

Stringy \*\_strings = NULL;

size\_t stringsCount = 0;

void add\_to\_strings(Stringy str) {

  stringsCount++;

  \_strings = (Token \*)realloc(\_strings, stringsCount \* sizeof(Stringy));

  if (\_strings == NULL)

    add\_to\_errors(create\_error\_with\_linecolumn(

        MEMORY\_ACCESS, "Cannot reallocate \*\_strings", true, str.row, str.col));

  else

    \_strings[stringsCount - 1] = str;

}

bool is\_stringy(size\_t tokenCode) {

  for (size\_t i = 0; i < stringsCount; i++)

    if (tokenCode == \_strings[i].code)

      return true;

  return false;

}

==> lexer\_state/symbol\_type.c <==

#include "symbol\_type.h"

#include <stdio.h>

unsigned short int symbol\_type(char symbol) {

  unsigned short int category = 6;

  if ((symbol > 7 && symbol < 14) || symbol == 32)

    category = SYMBOL\_WS;

  else if (symbol > 47 && symbol < 58)

    category = SYMBOL\_DIG;

  else if (symbol > 64 && symbol < 91)

    category = SYMBOL\_LET;

  else if (symbol == '.' || symbol == ';' || symbol == '[' || symbol == ']' ||

           symbol == '=' || symbol == '+' || symbol == '-')

    category = SYMBOL\_DM1;

  else if (symbol == ':' || symbol == '<' || symbol == '>' || symbol == '/' ||

           symbol == '\\')

    category = SYMBOL\_DM2;

  else if (symbol == '(')

    category = SYMBOL\_COM\_BEGIN;

  else if (symbol == EOF)

    category = SYMBOL\_EOF;

  else

    category = SYMBOL\_ERROR;

  return category;

}

==> lexer\_state/token.c <==

#include "token.h"

#include "id\_generator.h"

Token \*\_tokens = NULL;

size\_t tokenCount = 0;

void add\_to\_tokens(Token token) {

  tokenCount++;

  \_tokens = (Token \*)realloc(\_tokens, tokenCount \* sizeof(Token));

  if (\_tokens == NULL)

    add\_to\_errors(create\_error\_with\_linecolumn(MEMORY\_ACCESS,

                                               "Cannot reallocate \*\_tokens",

                                               true, token.row, token.col));

  else

    \_tokens[tokenCount - 1] = token;

}

==> lexer\_state/token\_structure.c <==

#include "token\_structure.h"

#include "id\_generator.h"

Token create\_token(size\_t row, size\_t col, char \*\_data, size\_t dataSize,

                   unsigned short int type) {

  size\_t code = get\_id(row, col, type);

  Token token = {row, col, code, \_data, dataSize};

  return token;

}

Token create\_token\_with\_code(size\_t row, size\_t col, char \*\_data,

                             size\_t dataSize, size\_t code) {

  Token token = {row, col, code, \_data, dataSize};

  return token;

}

==> semant\_state/add\_to.c <==

#include "error.h"

#include "semant.h"

#include <stdio.h>

#include <string.h>

void add\_to\_const(Const c) {

  if (!iAmInConst(c.name) && !iAmProgram(c.name)) {

    constCount++;

    consts = (Const \*)realloc(consts, sizeof(Const) \* constCount);

    if (consts == NULL)

      add\_to\_errors(create\_error\_without\_linecolumn(

          MEMORY\_ACCESS, "Cannot realloc consts", true));

    else {

      consts[constCount - 1] = c;

    }

  } else {

    char val[100];

    snprintf(val, 100, "Cannot create const %s, name used by CONST or PROGRAM",

             c.name);

    add\_to\_errors(create\_error\_without\_linecolumn(SEMANT\_STATE, val, true));

  }

}

void add\_to\_vars(Var v) {

  if (!iAmInConst(v.name) && !iAmProgram(v.name)) {

    varsCount++;

    vars = (Var \*)realloc(vars, sizeof(Var) \* varsCount);

    if (vars == NULL)

      add\_to\_errors(create\_error\_without\_linecolumn(

          MEMORY\_ACCESS, "Cannot realloc vars", true));

    else

      vars[varsCount - 1] = v;

  } else {

    char val[100];

    snprintf(val, 100, "Cannot create var %s, name used by CONST or PROGRAM",

             v.name);

    add\_to\_errors(create\_error\_without\_linecolumn(SEMANT\_STATE, val, true));

  }

}

void add\_to\_statements(char \*value) {

  codeCount++;

  statementsCode = (char \*\*)realloc(statementsCode, sizeof(value) \* codeCount);

  if (statementsCode == NULL)

    add\_to\_errors(create\_error\_without\_linecolumn(

        MEMORY\_ACCESS, "Cannot realloc statementsCode", true));

  else {

    statementsCode[codeCount - 1] = malloc(sizeof(char) \* strlen(value));

    strcpy(statementsCode[codeCount - 1], value);

  }

}

char \*\*add\_to\_semant\_final\_program(char \*value) {

  semant\_final\_count++;

  semant\_final =

      (char \*\*)realloc(semant\_final, sizeof(value) \* semant\_final\_count);

  if (semant\_final == NULL)

    add\_to\_errors(create\_error\_without\_linecolumn(

        MEMORY\_ACCESS, "Cannot realloc semant\_final", true));

  else {

    semant\_final[semant\_final\_count - 1] = malloc(sizeof(char) \* strlen(value));

    strcpy(semant\_final[semant\_final\_count - 1], value);

  }

  return semant\_final;

}

==> semant\_state/generate\_final.c <==

#include "semant.h"

#include <stdio.h>

void generate\_final\_output()

{

  char v[100];

  snprintf(v, 100, ".section .rodata");

  add\_to\_semant\_final\_program(v);

  for (size\_t i = 0; i < constCount; i++) {

    snprintf(v, 100, "\t%s:\t.quad %s", consts[i].name, consts[i].value);

    add\_to\_semant\_final\_program(v);

  }

  snprintf(v, 100, "\n");

  add\_to\_semant\_final\_program(v);

  snprintf(v, 100, ".section .bbs");

  add\_to\_semant\_final\_program(v);

  for (size\_t i = 0; i < varsCount; i++) {

    snprintf(v, 100, "\t%s:\t.space %s", vars[i].name, vars[i].value);

    add\_to\_semant\_final\_program(v);

  }

  snprintf(v, 100, "\n");

  add\_to\_semant\_final\_program(v);

  snprintf(v, 100, ".section .text");

  add\_to\_semant\_final\_program(v);

  snprintf(v, 100, ".globl main");

  add\_to\_semant\_final\_program(v);

  snprintf(v, 100, "main:");

  add\_to\_semant\_final\_program(v);

  snprintf(v, 100, "\tjmp %s", program\_name);

  add\_to\_semant\_final\_program(v);

  snprintf(v, 100, "%s:", program\_name);

  add\_to\_semant\_final\_program(v);

  for (size\_t i = 0; i < codeCount; i++)

    add\_to\_semant\_final\_program(statementsCode[i]);

  snprintf(v, 100, "\tmovq\t$60, %%rax");

  add\_to\_semant\_final\_program(v);

  snprintf(v, 100, "\txor\t%%rdi, %%rdi");

  add\_to\_semant\_final\_program(v);

  snprintf(v, 100, "\tsyscall");

  add\_to\_semant\_final\_program(v);

}

==> semant\_state/iAm.c <==

#include "semant.h"

#include <string.h>

bool iAmInConst(char \*v) {

  for (size\_t i = 0; i < constCount; i++) {

    if (strcmp(consts[i].name, v) == 0)

      return true;

  }

  return false;

}

bool iAmProgram(char \*v)

{

  if(strcmp(program\_name,v) == 0)

    return true;

  return false;

}

bool iAmInVars(char \*v) {

  for (size\_t i = 0; i < varsCount; i++) {

    if (strcmp(vars[i].name, v) == 0)

      return true;

  }

  return false;

}

==> semant\_state/semant.c <==

#include "semant.h"

#include "error.h"

#include <stdio.h>

#include <string.h>

char \*\*semant\_final = NULL;

size\_t semant\_final\_count = 0;

char \*program\_name = NULL;

Const \*consts = NULL;

size\_t constCount = 0;

Var \*vars = NULL;

size\_t varsCount = 0;

char \*\*statementsCode = NULL;

size\_t codeCount = 0;

size\_t skip = 0;

size\_t labelCounter = 0;

size\_t dived = 1;

#define macro\_bbb(val, x, y, z) val->\_branches[x]->\_branches[y]->\_branches[z]

#define macro\_bbbb(val, x, y, z, k)                                            \

  val->\_branches[x]->\_branches[y]->\_branches[z]->\_branches[k]

void process\_summands\_list(Tree \*list, char \*reg) {

  char v[100];

  char err[200];

  if (strcmp(list->\_branches[0]->\_value, "<empty>") == 0) {

  } else {

    if (strcmp(list->\_branches[0]->\_value, "+") == 0) {

      if (strcmp(macro\_bbb(list, 1, 0, 0)->\_value, "<identifier>") == 0) {

        snprintf(v, 100, "\taddq\t%s, %%%s",

                 macro\_bbbb(list, 1, 0, 0, 0)->\_value, reg);

        if (!iAmInVars(macro\_bbbb(list, 1, 0, 0, 0)->\_value)) {

          snprintf(err, 200, "Variable %s used before declaration",

                   macro\_bbbb(list, 1, 0, 0, 0)->\_value);

          add\_to\_errors(

              create\_error\_without\_linecolumn(SEMANT\_STATE, err, true));

        }

      } else

        snprintf(v, 100, "\taddq\t%s, %%%s", macro\_bbb(list, 1, 0, 0)->\_value,

                 reg);

    } else {

      if (strcmp(macro\_bbb(list, 1, 0, 0)->\_value, "<identifier>") == 0) {

        snprintf(v, 100, "\tsubq\t%s, %%%s",

                 macro\_bbbb(list, 1, 0, 0, 0)->\_value, reg);

        if (!iAmInVars(macro\_bbbb(list, 1, 0, 0, 0)->\_value)) {

          snprintf(err, 200, "Variable %s used before declaration",

                   macro\_bbbb(list, 1, 0, 0, 0)->\_value);

          add\_to\_errors(

              create\_error\_without\_linecolumn(SEMANT\_STATE, err, true));

        }

      } else

        snprintf(v, 100, "\tsubq\t%s, %%%s", macro\_bbb(list, 1, 0, 0)->\_value,

                 reg);

    }

    add\_to\_statements(v);

    if (list->branchesCount == 3) {

      process\_summands\_list(list->\_branches[2], reg);

    }

  }

}

void process\_expression(Tree \*expression, char \*reg) {

  char v[100];

  char err[200];

  if (strcmp(expression->\_branches[0]->\_value, "<summand>") == 0) {

    if (strcmp(macro\_bbb(expression, 0, 0, 0)->\_value, "<identifier>") == 0) {

      snprintf(v, 100, "\tmovq\t%s, %%%s",

               macro\_bbbb(expression, 0, 0, 0, 0)->\_value, reg);

      if (!iAmInVars(macro\_bbbb(expression, 0, 0, 0, 0)->\_value)) {

        snprintf(err, 200, "Variable %s used before declaration",

                 macro\_bbbb(expression, 0, 0, 0, 0)->\_value);

        add\_to\_errors(create\_error\_without\_linecolumn(SEMANT\_STATE, err, true));

      }

    } else

      snprintf(v, 100, "\tmovq\t%s, %%%s",

               macro\_bbb(expression, 0, 0, 0)->\_value, reg);

    add\_to\_statements(v);

    process\_summands\_list(expression->\_branches[1], reg);

  } else {

    snprintf(v, 100, "\tmovq\t$0, %%%s", reg);

    add\_to\_statements(v);

    if (strcmp(macro\_bbb(expression, 0, 1, 0)->\_value, "<identifier>") == 0) {

      snprintf(v, 100, "\tmovq\t%s, %%%s",

               macro\_bbbb(expression, 0, 1, 0, 0)->\_value, reg);

      if (!iAmInVars(macro\_bbbb(expression, 0, 1, 0, 0)->\_value)) {

        snprintf(err, 200, "Variable %s used before declaration",

                 macro\_bbbb(expression, 0, 1, 0, 0)->\_value);

        add\_to\_errors(create\_error\_without\_linecolumn(SEMANT\_STATE, err, true));

      }

    } else

      snprintf(v, 100, "\tmovq\t%s, %%%s",

               macro\_bbb(expression, 0, 1, 0)->\_value, reg);

    process\_summands\_list(expression->\_branches[2], reg);

  }

}

void process\_statement(Tree \*stats) {

  Var v;

  v.name = macro\_bbb(stats, 0, 0, 0)->\_value;

  v.value = "8";

  process\_expression(stats->\_branches[2], "rax");

  add\_to\_vars(v);

  char val[100];

  snprintf(val, 100, "\tmovq\t%%rax, %s", v.name);

  add\_to\_statements(val);

}

size\_t labelCounterBackup = 0;

void dive\_alternatives(Tree \*my\_tree, Tree \*parent, char \*val) {

  if (strcmp(my\_tree->\_value, "<expression>") == 0) {

    if (parent != NULL) {

      if (strcmp(parent->\_value, val) == 0 &&

          strcmp("<alternative>", val) == 0) {

        process\_expression(my\_tree, "rbx");

        char v[100];

        snprintf(v, 100, "\tcmpq\t%%rax, %%rbx");

        add\_to\_statements(v);

        snprintf(v, 100, "\tje\t?L%llu", labelCounter++);

        add\_to\_statements(v);

      } else if (strcmp(parent->\_value, val) == 0 &&

                 strcmp("<statement>", val) == 0 && dived == 0) {

        char v[100];

        snprintf(v, 100, "?L%llu: NOP", labelCounter++);

        add\_to\_statements(v);

        process\_statement(parent);

        snprintf(v, 100, "\tjmp\t?L%llu", labelCounterBackup);

        add\_to\_statements(v);

      } else if (strcmp(parent->\_value, val) == 0 &&

                 strcmp("<statement>", val) == 0) {

        dived--;

      }

    }

  } else {

    for (size\_t i = 0; i < my\_tree->branchesCount; i++) {

      dive\_alternatives(my\_tree->\_branches[i], my\_tree, val);

    }

  }

}

void proc\_name() {

  Tree \*name = find\_in\_tree(\_tree, "<procedure-identifier>");

  program\_name = name->\_branches[0]->\_branches[0]->\_value;

}

void proc\_const(Tree \*cur\_tree) {

  Tree \*constDeclars = find\_in\_tree(cur\_tree, "<constant-declarations-list>");

  if (constDeclars != NULL) {

    Const c;

    c.name = macro\_bbbb(constDeclars, 0, 0, 0, 0)->\_value;

    c.value = macro\_bbbb(constDeclars, 0, 2, 0, 0)->\_value;

    add\_to\_const(c);

    proc\_const(constDeclars->\_branches[1]);

  }

}

void proc\_statements(Tree \*cur\_tree) {

  Tree \*statementDeclars = find\_in\_tree(cur\_tree, "<statements-list>");

  if (statementDeclars != NULL) {

    if (strcmp(statementDeclars->\_branches[0]->\_branches[0]->\_value, "CASE") ==

        0) {

      char v[100];

      process\_expression(statementDeclars->\_branches[0]->\_branches[1], "rax");

      dive\_alternatives(statementDeclars->\_branches[0], NULL, "<alternative>");

      labelCounterBackup = labelCounter;

      snprintf(v, 100, "\tjmp\t?L%llu", labelCounter++);

      add\_to\_statements(v);

      labelCounter = 0;

      dive\_alternatives(statementDeclars->\_branches[0], NULL, "<statement>");

      labelCounter = labelCounterBackup;

      snprintf(v, 100, "?L%llu: NOP", labelCounter++);

      add\_to\_statements(v);

    } else if (strcmp(statementDeclars->\_branches[0]->\_branches[0]->\_value,

                      "<variable-identifier>") == 0) {

      process\_statement(statementDeclars->\_branches[0]);

    } else {

      add\_to\_errors(create\_error\_without\_linecolumn(

          SEMANT\_STATE, "Impossible statement", true));

    }

    proc\_statements(statementDeclars->\_branches[1]);

  }

}

void proc\_semant() {

  proc\_name();

  proc\_const(\_tree);

  proc\_statements(\_tree);

  generate\_final\_output();

}

==> semant\_state/tree\_finder.c <==

#include "semant.h"

#include <string.h>

Tree \*find\_in\_tree(Tree \*cur\_tree, char \*value) {

  if (strcmp(cur\_tree->\_value, value) == 0 && skip == 0) {

    return cur\_tree;

  } else {

    if (strcmp(cur\_tree->\_value, value) == 0)

      skip--;

    if (cur\_tree->branchesCount != 0) {

      for (size\_t i = 0; i < cur\_tree->branchesCount; i++) {

        Tree \*temp = find\_in\_tree(cur\_tree->\_branches[i], value);

        if (temp != NULL) {

          return temp;

        }

      }

      return NULL;

    } else

      return NULL;

  }

}

==> syntax\_state/knut\_tables.c <==

#include "knut\_tables.h"

#include "error.h"

#include "terms.h"

Code new\_code(size\_t addrTo, char \*\_term) {

  Code myCode = {addrTo, \_term, false};

  if (\_term != NULL)

    myCode.isTerm = true;

  return myCode;

}

Line new\_line(size\_t addr, Code myCode, bool at, size\_t afAddr) {

  Line myLine = {addr, myCode, at, afAddr};

  return myLine;

}

void insert(Table \*\_table, Line myLine) {

  \_table->linesCount++;

  \_table->lines =

      (Line \*)realloc(\_table->lines, \_table->linesCount \* sizeof(Line));

  if (\_table->lines == NULL)

    add\_to\_errors(create\_error\_without\_linecolumn(

        MEMORY\_ACCESS, "Cannot reallocate \*knut\_lines", true));

  else

    \_table->lines[\_table->linesCount - 1] = myLine;

}

char \*name\_by\_id(size\_t addr) {

  switch (addr) {

  case SIGNAL\_PROGRAM:

    return "<signal-program>";

  case PROGRAM:

    return "<program>";

  case BLOCK:

    return "<block>";

  case DECLARATIONS:

    return "<declarations>";

  case CONSTANT\_DECLARATIONS:

    return "<constant-declarations>";

  case CONSTANT\_DECLARATIONS\_LIST:

    return "<constant-declarations-list>";

  case CONSTANT\_DECLARATION:

    return "<constant-declaration>";

  case STATEMENT:

    return "<statement>";

  case STATEMENTS\_LIST:

    return "<statements-list>";

  case ALTERNATIVES\_LIST:

    return "<alternatives-list>";

  case ALTERNATIVE:

    return "<alternative>";

  case EXPRESSION:

    return "<expression>";

  case SUMMANDS\_LIST:

    return "<summands-list>";

  case ADD\_INSTRUCTION:

    return "<add-instruction>";

  case SUMMAND:

    return "<summand>";

  case CONSTANT:

    return "<constant>";

  case VARIABLE\_IDENTIFIER:

    return "<variable-identifier>";

  case CONSTANT\_IDENTIFIER:

    return "<constant-identifier>";

  case PROCEDURE\_IDENTIFIER:

    return "<procedure-identifier>";

  case UNSIGNED\_INTEGER:

    return "<unsigned-integer>";

  case IDENTIFIER:

    return "<identifier>";

  case STRING:

    return "<string>";

  case EMPTY:

    return "<empty>";

  default:

    return "<error>";

  };

}

/\*

rule(addr,addr\_to,term,at\_addr,af\_addr)

Creates new rule in knut table

\*/

#define rule(addr, addr\_to, term, at\_addr, af\_addr)                            \

  insert(&myTable, new\_line(addr, new\_code(addr\_to, term), at\_addr, af\_addr))

Table create\_knut\_table() {

  Table myTable = {.linesCount = 0, .lines = NULL};

  /\*

  AT - ACTION TRUE

  AF - ACTION FALSE

  \*/

  /\*   ADDR ADDR\_TO TERM AT AF\_ADDR\*/

  /\*<signal-program> --> <program> \*/

  rule(0, SIGNAL\_PROGRAM, NULL, false, ERROR);

  rule(1, PROGRAM, NULL, false, ERROR);

  rule(2, SIGNAL\_PROGRAM\_FINISH, NULL, true, ERROR);

  /\*<program> --> PROGRAM <procedure-identifier> ; <block> .\*/

  rule(3, 0, "PROGRAM", false, ERROR);

  rule(4, PROCEDURE\_IDENTIFIER, NULL, false, ERROR);

  rule(5, 0, ";", false, ERROR);

  rule(6, BLOCK, NULL, false, ERROR);

  rule(7, 0, ".", true, ERROR);

  /\*<block> --> <declarations> BEGIN <statements-list> END\*/

  rule(8, DECLARATIONS, NULL, false, ERROR);

  rule(9, 0, "BEGIN", false, ERROR);

  rule(10, STATEMENTS\_LIST, NULL, false, ERROR);

  rule(11, 0, "END", true, ERROR);

  /\*<declarations> --> <constant-declarations>\*/

  rule(12, CONSTANT\_DECLARATIONS, NULL, true, ERROR);

  /\*<constant-declarations> --> CONST <constant-declarations-list> | <empty>\*/

  rule(13, 0, "CONST", false, ERROR);

  rule(14, CONSTANT\_DECLARATIONS\_LIST, NULL, true, 15);

  rule(15, EMPTY, NULL, true, ERROR);

  /\*<constant-declarations-list> --> <constantdeclaration>

   \* <constant-declarations-list> | <empty>\*/

  rule(16, CONSTANT\_DECLARATION, NULL, false, ERROR);

  rule(17, CONSTANT\_DECLARATIONS\_LIST, NULL, true, 18);

  rule(18, EMPTY, NULL, true, ERROR);

  /\*<constant-declaration> --> <constant-identifier> = <constant>;\*/

  rule(19, CONSTANT\_IDENTIFIER, NULL, false, ERROR);

  rule(20, 0, "=", false, ERROR);

  rule(21, CONSTANT, NULL, false, ERROR);

  rule(22, 0, ";", true, ERROR);

  /\*<statements-list> --> <statement> <statement-list> | <empty>\*/

  rule(23, STATEMENT, NULL, false, ERROR);

  rule(24, STATEMENTS\_LIST, NULL, true, 25);

  rule(25, EMPTY, NULL, true, ERROR);

  /\*<statement> --> CASE <expression> OF <alternativeslist> ENDCASE ;|

  <variable-identifier> := <expression> ;\*/

  rule(26, 0, "CASE", false, 32);

  rule(27, EXPRESSION, NULL, false, ERROR);

  rule(28, 0, "OF", false, ERROR);

  rule(29, ALTERNATIVES\_LIST, NULL, false, ERROR);

  rule(30, 0, "ENDCASE", false, ERROR);

  rule(31, 0, ";", true, ERROR);

  rule(32, VARIABLE\_IDENTIFIER, NULL, false, ERROR);

  rule(33, 0, ":=", false, ERROR);

  rule(34, EXPRESSION, NULL, false, ERROR);

  rule(35, 0, ";", true, ERROR);

  /\*<alternatives-list> --> <alternative> <alternativeslist> | <empty>\*/

  rule(36, ALTERNATIVE, NULL, false, ERROR);

  rule(37, ALTERNATIVES\_LIST, NULL, true, 38);

  rule(38, EMPTY, NULL, true, ERROR);

  /\*<alternative> --> <expression> : /<statements-list>\\*/

  rule(39, EXPRESSION, NULL, false, ERROR);

  rule(40, 0, ":", false, ERROR);

  rule(41, 0, "/", false, ERROR);

  rule(42, STATEMENTS\_LIST, NULL, false, ERROR);

  rule(43, 0, "\\", true, ERROR);

  /\*<expression> --> <summand> <summands-list> | - <summand> <summands-list>\*/

  rule(44, SUMMAND, NULL, false, 46);

  rule(45, SUMMANDS\_LIST, NULL, true, ERROR);

  rule(46, 0, "-", false, ERROR);

  rule(47, SUMMAND, NULL, false, ERROR);

  rule(48, SUMMANDS\_LIST, NULL, true, ERROR);

  /\*<summands-list> --> <add-instruction> <summand> | <summands-list> |

   \* <empty>\*/

  rule(49, ADD\_INSTRUCTION, NULL, false, 54);

  rule(50, SUMMAND, NULL, false, ERROR);

  rule(51, SUMMANDS\_LIST, NULL, true, 52);

  rule(52, ADD\_INSTRUCTION, NULL, false, 54);

  rule(53, SUMMAND, NULL, true, ERROR);

  rule(54, EMPTY, NULL, true, ERROR);

  /\*<add-instruction> --> + | -\*/

  rule(55, 0, "+", true, 56);

  rule(56, 0, "-", true, ERROR);

  /\*<summand> --> <variable-identifier> | <unsigned-integer>\*/

  rule(57, VARIABLE\_IDENTIFIER, NULL, true, 58);

  rule(58, UNSIGNED\_INTEGER, NULL, true, ERROR);

  /\*<constant> --> <unsigned-integer>\*/

  rule(59, UNSIGNED\_INTEGER, NULL, true, ERROR);

  /\*<variable-identifier> --> <identifier>\*/

  rule(60, IDENTIFIER, NULL, true, ERROR);

  /\*<constant-identifier> --> <identifier>\*/

  rule(61, IDENTIFIER, NULL, true, ERROR);

  /\*<procedure-identifier> --> <identifier>\*/

  rule(62, IDENTIFIER, NULL, true, ERROR);

  rule(UNSIGNED\_INTEGER, 0, "", true, ERROR);

  rule(IDENTIFIER, 0, "", true, ERROR);

  rule(STRING, 0, "", true, ERROR);

  rule(EMPTY, 0, "", true, ERROR);

  return myTable;

}

==> syntax\_state/ruler.c <==

#include "syntax.h"

Line ruler(Table table, size\_t k) {

  for (size\_t i = 0; i < table.linesCount; i++)

    if (table.lines[i].addr == k)

      return table.lines[i];

  exit(EXIT\_FAILURE);

}

==> syntax\_state/syntax.c <==

#include <stdbool.h>

#include <stdio.h>

#include <string.h>

#include "constant.h"

#include "error.h"

#include "identifier.h"

#include "knut\_tables.h"

#include "strings.h"

#include "syntax.h"

#include "terms.h"

#include "token.h"

Tree \*\_tree;

Tree \*\_backup;

size\_t tokenIterator = 0;

char \*\_expected;

void proc\_syntax() {

  Table table = create\_knut\_table();

  \_tree = create\_node(name\_by\_id(SIGNAL\_PROGRAM), SIGNAL\_PROGRAM);

  ProbablyResults run = probe(table, PROGRAM);

  if (run.status)

    add\_branch(\_tree, run.result);

  else {

    add\_to\_errors(create\_error\_syntaxer(

        \_tokens[tokenIterator].row, \_tokens[tokenIterator].col,

        run.result->\_value, \_tokens[run.result->id].\_data));

    add\_branch(\_tree, \_backup);

  }

}

ProbablyResults probe(Table table, size\_t i) {

  ProbablyResults ret = {false, NULL};

  bool state = false;

  Tree \*newTree = create\_node(name\_by\_id(i), i);

  size\_t savedTokenPos = tokenIterator;

  bool atNotFinished = true;

  do {

    if (!rules(i).code.isTerm) {

      ProbablyResults inner\_probe = probe(table, rules(i).code.addrTo);

      if (inner\_probe.status == true) {

        if (rules(i).atAddr != true)

          i++;

        else

          atNotFinished = false;

        add\_branch(newTree, inner\_probe.result);

        state = true;

      } else {

        if (rules(i).afAddr != ERROR) {

          i = rules(i).afAddr;

          state = true;

        } else {

          state = false;

          ret.result = inner\_probe.result;

          ret.status = state;

          return ret;

        }

      }

    } else {

      state = false;

      switch (rules(i).addr) {

      case UNSIGNED\_INTEGER:

        if (is\_constant(\_tokens[tokenIterator].code))

          add\_branch\_def\_token();

        break;

      case IDENTIFIER:

        if (is\_identifier(\_tokens[tokenIterator].code))

          add\_branch\_def\_token();

        break;

      case STRING:

        if (is\_stringy(\_tokens[tokenIterator].code))

          add\_branch\_def\_token();

        break;

      case EMPTY:

        add\_branch\_empty();

        break;

      default:

        if (tokenIterator < tokenCount)

          if (strcmp(rules(i).code.\_term, \_tokens[tokenIterator].\_data) == 0)

            add\_branch\_def\_token();

      };

      if (state == false) {

        if (rules(i).afAddr != ERROR) {

          i = rules(i).afAddr;

          state = true;

        } else if (rules(i).addr < 100) {

          ret.status = false;

          ret.result = create\_node(rules(i).code.\_term, tokenIterator);

          \_backup = newTree;

          tokenIterator = savedTokenPos;

          return ret;

        }

      } else {

        if (rules(i).addr != EMPTY)

          tokenIterator++;

        if (rules(i).addr < 100 && rules(i).atAddr != true)

          i++;

        else

          atNotFinished = false;

      }

    }

  } while (atNotFinished && state && errorCount < 1);

  ret.result = newTree;

  ret.status = state;

  return ret;

}

==> syntax\_state/tree.c <==

#include "tree.h"

#include "error.h"

#include "symbol\_type.h"

#include "token.h"

Tree \*create\_node(char \*\_value, size\_t id) {

  Tree \*t;

  t = (Tree \*)malloc(sizeof(Tree));

  t->\_branches = NULL;

  t->branchesCount = 0;

  t->\_value = \_value;

  t->id = id;

  return t;

}

void add\_branch(Tree \*\_origin, Tree \*\_tree) {

  \_origin->branchesCount++;

  \_origin->\_branches = (Tree \*\*)realloc(

      \_origin->\_branches, \_origin->branchesCount \* sizeof(Tree \*));

  if (\_origin->\_branches == NULL)

    add\_to\_errors(create\_error\_without\_linecolumn(

        MEMORY\_ACCESS, "Cannot reallocate \*\_branches", true));

  else

    \_origin->\_branches[\_origin->branchesCount - 1] = \_tree;

}

void free\_tree(Tree \*\_tree) {

  if (\_tree != 0) {

    for (size\_t i = 0; i < \_tree->branchesCount; i++)

      free(\_tree->\_branches[i]);

    if (\_tree->branchesCount != 0)

      free(\_tree->\_branches);

    free(\_tree);

  }

}

==> util/cli.c <==

#include <stdbool.h>

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <unistd.h>

#include "cli.h"

#include "error.h"

#define WIN

Params params = {NULL, "output", false, true, true, true, NULL};

void check\_file\_access(char \*\_file, bool inputFile) {

  if (access(\_file, F\_OK) == -1) {

    if (inputFile)

      add\_to\_errors(create\_error\_without\_linecolumn(

          FILE\_ACCESS, "Missing access to input/verify file", true));

    else

      add\_to\_errors(create\_error\_without\_linecolumn(

          FILE\_ACCESS, "File for output does not exist, creating...", false));

  }

}

void check\_file\_missing(char \*\_file) {

  FILE \*\_fp;

  if (\_file != NULL) {

#ifdef WIN

    \_fp = fopen(\_file, "w+");

#endif

#ifndef WIN

    \_fp = fopen(\_file, "w");

#endif

    if (\_fp == NULL)

      add\_to\_errors(create\_error\_without\_linecolumn(

          FILE\_ACCESS, "Cannot create/open output file", true));

    fclose(\_fp);

  } else

    add\_to\_errors(create\_error\_without\_linecolumn(

        FILE\_ACCESS, "Cannot create/open output file", true));

}

void proc\_cli(int argc, char \*argv[]) {

  if (argc == 2)

    params.\_input\_file = argv[1];

  else {

    for (int i = 1; i < argc; i++) {

      if (strcmp(argv[i], "-f") == 0 && i + 1 < argc)

        params.\_input\_file = argv[++i];

      else if (strcmp(argv[i], "-o") == 0 && i + 1 < argc)

        params.\_output\_file = argv[++i];

      else if (strcmp(argv[i], "-q") == 0)

        params.verbose = 0;

      else if (strcmp(argv[i], "-offsyntax") == 0)

        params.out\_syntax = false;

      else if (strcmp(argv[i], "-offlexer") == 0)

        params.out\_lexer = false;

      else if (strcmp(argv[i], "-offcodegen") == 0)

        params.out\_codegen = false;

      else if (strcmp(argv[i], "-v") == 0 && i + 1 < argc)

        params.\_verify\_file = argv[++i];

    }

  }

  if (params.\_input\_file == NULL) {

    char v[200];

    snprintf(v,200,"Input filename %s is inaccessible.",params.\_input\_file);

    add\_to\_errors(create\_error\_without\_linecolumn(

        FILE\_ACCESS, v, true));

  } else {

    check\_file\_access(params.\_input\_file, true);

    check\_file\_access(params.\_output\_file, false);

    if (params.\_verify\_file != NULL)

      check\_file\_access(params.\_verify\_file, true);

    check\_file\_missing(params.\_output\_file);

  }

}

==> util/error.c <==

#include "error.h"

#include <stdbool.h>

#include <stddef.h>

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

Error \*\_errors = NULL;

size\_t errorCount = 0;

bool gotError = false;

bool gotWarning = false;

bool has\_critical() {

  for (size\_t i = 0; i < errorCount; i++)

    if (\_errors[i].critical)

      return true;

  return false;

}

Error create\_error\_syntaxer(size\_t row, size\_t col, char \*\_expected,

                            char \*\_here) {

  Error error = {.state = SYNTAX\_STATE,

                 .row = row,

                 .col = col,

                 .number = errorCount + 1,

                 .critical = true,

                 .\_expected = \_expected,

                 .\_here = \_here,

                 .syntaxer = true};

  return error;

}

Error create\_error\_without\_linecolumn(unsigned short int state,

                                      char \*\_error\_message, bool critical) {

  Error error = {errorCount + 1, state, NULL, critical, false, 0, 0,

                 NULL,           NULL,  false};

  error.\_error\_message=(char\*)malloc(sizeof(char)\*strlen(\_error\_message));

  error.\_error\_message=strcpy(error.\_error\_message, \_error\_message);

  return error;

}

Error create\_error\_with\_linecolumn(unsigned short int state,

                                   char \*\_error\_message, bool critical,

                                   size\_t row, size\_t col) {

  Error error = {

      errorCount + 1, state, \_error\_message, critical, true, row, col,

      NULL,           NULL,  false};

  return error;

}

Error create\_error\_def() {

  Error error = {0, NOT\_ERROR, "", false, false, 0, 0, NULL, NULL, false};

  return error;

}

void add\_to\_errors(Error error) {

  errorCount++;

  \_errors = (Error \*)realloc(\_errors, (errorCount) \* sizeof(Error));

  if (\_errors == NULL)

    exit(EXIT\_FAILURE);

  else {

    \_errors[errorCount - 1] = error;

    if (error.critical)

      gotError = true;

    else

      gotWarning = true;

  }

}

void clean\_errors() {

  errorCount = 0;

  \_errors = NULL;

}

==> util/out.c <==

#include <stdio.h>

#include "constant.h"

#include "identifier.h"

#include "lexer.h"

#include "out.h"

#include "strings.h"

#include "syntax.h"

#include "semant.h"

/\*This file is not sweet, I know, but I am too lazy\*/

void print\_params() {

  printf("Input file: %s\n", params.\_input\_file);

  printf("Output file: %s\n", params.\_output\_file);

  if (params.verbose)

    printf("Verbose mode enabled\n");

}

void print\_error(Error error) {

  char \*critical = "Warning";

  unsigned short int state = error.state;

  if (error.critical)

    critical = "Error";

  if (state == LEXER\_STATE)

    if (error.hasLineColumn)

      printf("#%lld|%s(Lexer)| Line->%lld, Column->%lld |: %s\n", error.number,

             critical, error.row, error.col, error.\_error\_message);

    else

      printf("#%lld|%s(Lexer): %s\n", error.number, critical,

             error.\_error\_message);

  else if (state == FILE\_ACCESS)

    printf("#%lld|%s(File IO): %s\n", error.number, critical,

           error.\_error\_message);

  else if (state == SYNTAX\_STATE)

    printf("#%lld|%s(Syntax): %s\n", error.number, critical,

           error.\_error\_message);

  else if (state == MEMORY\_ACCESS)

    printf("#%lld|%s(Memory): %s\n", error.number, critical,

           error.\_error\_message);

  else if (state == SEMANT\_STATE)

    printf("#%lld|%s(Semantics): %s\n", error.number, critical,

           error.\_error\_message);

  else

    printf("#%lld|%s(Unknown): %s\n", error.number, critical,

           error.\_error\_message);

}

void get\_error(Error error, FILE \*\_\_output\_file) {

  char \*critical = "Warning";

  unsigned short int state = error.state;

  if (error.critical)

    critical = "Error";

  if (state == LEXER\_STATE)

    if (error.hasLineColumn)

      fprintf(\_\_output\_file,

              "#%lld|%s(Lexer)| Line->%lld, Column->%lld |: %s\n", error.number,

              critical, error.row, error.col, error.\_error\_message);

    else

      fprintf(\_\_output\_file, "#%lld|%s(Lexer): %s\n", error.number, critical,

              error.\_error\_message);

  else if (state == FILE\_ACCESS)

    fprintf(\_\_output\_file, "#%lld|%s(File IO): %s\n", error.number, critical,

            error.\_error\_message);

  else if (state == SYNTAX\_STATE)

    fprintf(\_\_output\_file, "#%lld|%s(Syntax): %s\n", error.number, critical,

            error.\_error\_message);

  else if (state == MEMORY\_ACCESS)

    fprintf(\_\_output\_file, "#%lld|%s(Memory): %s\n", error.number, critical,

            error.\_error\_message);

    else if (state == SEMANT\_STATE)

    fprintf(\_\_output\_file,"#%lld|%s(Semantics): %s\n", error.number, critical,

           error.\_error\_message);

  else

    fprintf(\_\_output\_file, "#%lld|%s(Unknown): %s\n", error.number, critical,

            error.\_error\_message);

}

void get\_syntaxer\_error(Error error, FILE \*\_\_output\_file) {

  char \*critical = "Warning";

  if (error.critical)

    critical = "Error";

  fprintf(\_\_output\_file,

          "#%lld|%s(Syntax)| Line->%lld, Column->%lld |: \'%s\' expected, but "

          "\'%s\' found.\n",

          error.number, critical, error.row, error.col, error.\_expected,

          error.\_here);

}

void print\_errors() {

  for (size\_t i = 0; i < errorCount; i++) {

    print\_error(\_errors[i]);

  }

}

void print\_lexer() {

  printf("Current buffer: %s\n", lexer.\_buffer);

  printf("Current row: %lld\n", lexer.row);

  printf("Current col: %lld\n", lexer.col);

  printf("Current symbol: %c\n", lexer.symbol);

  printf("Current symbol type: %d\n", lexer.symbolType);

}

void print\_token(Token token) {

  printf("[%lld][%lld] %lld: %s\n", token.row, token.col, token.code,

         token.\_data);

}

void print\_tokens() {

  for (unsigned long int i = 0; i < tokenCount; i++) {

    print\_token(\_tokens[i]);

  }

}

void out\_file\_lexer() {

  FILE \*\_\_output\_file;

  \_\_output\_file = fopen(params.\_output\_file, "w");

  if (\_\_output\_file == NULL) {

    add\_to\_errors(create\_error\_without\_linecolumn(

        FILE\_ACCESS, "Cannot write to output file", true));

  } else {

    fprintf(\_\_output\_file,

            "|Line  |Column|Code  |Data  \n+------+------+------+------\n");

    for (size\_t i = 0; i < tokenCount; i++) {

      fprintf(\_\_output\_file, "|%6lld|%6lld|%6lld|%s\n", \_tokens[i].row,

              \_tokens[i].col, \_tokens[i].code, \_tokens[i].\_data);

    }

  }

  out\_file\_errors(\_\_output\_file);

  fclose(\_\_output\_file);

}

void print\_file\_out() {

  FILE \*\_\_output\_file;

  \_\_output\_file = fopen(params.\_output\_file, "r");

  if (\_\_output\_file == NULL) {

    add\_to\_errors(create\_error\_without\_linecolumn(

        FILE\_ACCESS, "Cannot open output file for reading", true));

  } else {

    for (char c = (char)getc(\_\_output\_file); c != EOF;

         c = (char)getc(\_\_output\_file))

      printf("%c", c);

  }

}

void out\_file\_errors(FILE \*\_\_output\_file) {

  if (errorCount > 0) {

    fprintf(\_\_output\_file, "ERRORS:\n");

  }

  for (size\_t i = 0; i < errorCount; i++) {

    if (\_errors[i].syntaxer)

      get\_syntaxer\_error(\_errors[i], \_\_output\_file);

    else

      get\_error(\_errors[i], \_\_output\_file);

  }

}

void just\_clean() { clean\_errors(); }

void out\_node(Tree \*\_my\_tree, FILE \*\_\_output\_file, size\_t level) {

  for (size\_t k = 0; k < level; k++)

    fprintf(\_\_output\_file, "|");

  if(\_my\_tree != NULL){

    fprintf(\_\_output\_file, "%s\n", \_my\_tree->\_value);

  for (size\_t i = 0; i < \_my\_tree->branchesCount; i++) {

    out\_node(\_my\_tree->\_branches[i], \_\_output\_file, level + 1);

  }

  }

}

void out\_file\_syntax() {

  FILE \*\_\_output\_file;

  \_\_output\_file = fopen(params.\_output\_file, "a");

  if (\_\_output\_file == NULL) {

    add\_to\_errors(create\_error\_without\_linecolumn(

        FILE\_ACCESS, "Cannot write to output file", true));

  } else {

    fprintf(\_\_output\_file, "SYNTAX:\n");

    out\_node(\_tree, \_\_output\_file, 0);

  }

  fprintf(\_\_output\_file, "\n");

  out\_file\_errors(\_\_output\_file);

  fclose(\_\_output\_file);

}

void out\_file\_codegen()

 {

  FILE \*\_\_output\_file;

  \_\_output\_file = fopen(params.\_output\_file,"a");

  if(\_\_output\_file == NULL)

  {

    add\_to\_errors(create\_error\_without\_linecolumn(

        FILE\_ACCESS, "Cannot write to output file", true));

  }

  else

  {

    fprintf(\_\_output\_file, "CODEGEN:\n");

    for(size\_t i = 0; i < semant\_final\_count; i++)

      fprintf(\_\_output\_file,"%s\n", semant\_final[i]);

    out\_file\_errors(\_\_output\_file);

    fclose(\_\_output\_file);

  }

 }

void free\_errors() { free(\_errors); }

void free\_tokens() { free(\_tokens); }

void free\_tables() {

  free(\_constants);

  free(\_identifiers);

  free(\_strings);

}

void free\_trees() { free\_tree(\_tree); }

==> util/verify.c <==

#include "verify.h"

#include "symbol\_type.h"

#include <stdbool.h>

#include <stdio.h>

#include <stdlib.h>

char c, d;

size\_t o\_row = 0, o\_col = 0;

size\_t v\_row = 0, v\_col = 0;

#define step\_macro(x, row, col)                                                \

  do {                                                                         \

    if (x == '\n') {                                                           \

      row++;                                                                   \

      col = 0;                                                                 \

    } else if (x == '\t') {                                                    \

      col += 4;                                                                \

    } else {                                                                   \

      col++;                                                                   \

    }                                                                          \

  } while (0)

#define step\_c() step\_macro(c, o\_row, o\_col)

#define step\_d() step\_macro(d, v\_row, v\_col)

#define if\_step(x)                                                             \

  do {                                                                         \

    if (x)                                                                     \

      step\_c();                                                                \

    else                                                                       \

      step\_d();                                                                \

  } while (0)

#define skip\_ws\_char(x, flag)                                                  \

  do {                                                                         \

    if (symbol\_type(x) == SYMBOL\_WS) {                                         \

      x = (char)getc(\_out);                                                    \

      if\_step(flag);                                                           \

      skip\_ws(\_out, \_ver, flag);                                               \

    }                                                                          \

  } while (0)

#define open\_read\_file(pname, filename)                                        \

  FILE \*pname = fopen(filename, "r");                                          \

  if (\_out == NULL) {                                                          \

    printf("Failed to open output file on verify stage\n");                    \

    exit(EXIT\_FAILURE);                                                        \

  }

void skip\_ws(FILE \*\_out, FILE \*\_ver, bool is\_c) {

  if (is\_c)

    skip\_ws\_char(c, true);

  else

    skip\_ws\_char(d, false);

}

void verify(char \*\_output, char \*\_verify) {

  open\_read\_file(\_out, \_output);

  open\_read\_file(\_ver, \_verify);

  do {

    c = (char)getc(\_out);

    d = (char)getc(\_ver);

    step\_c();

    step\_d();

    skip\_ws(\_out, \_ver, true);

    skip\_ws(\_out, \_ver, false);

    if (c != d)

      printf("Output(%llu:%llu:%c) != Verify(%llu:%llu:%c)\n", o\_row + 1, o\_col,

             c, v\_row + 1, v\_col, d);

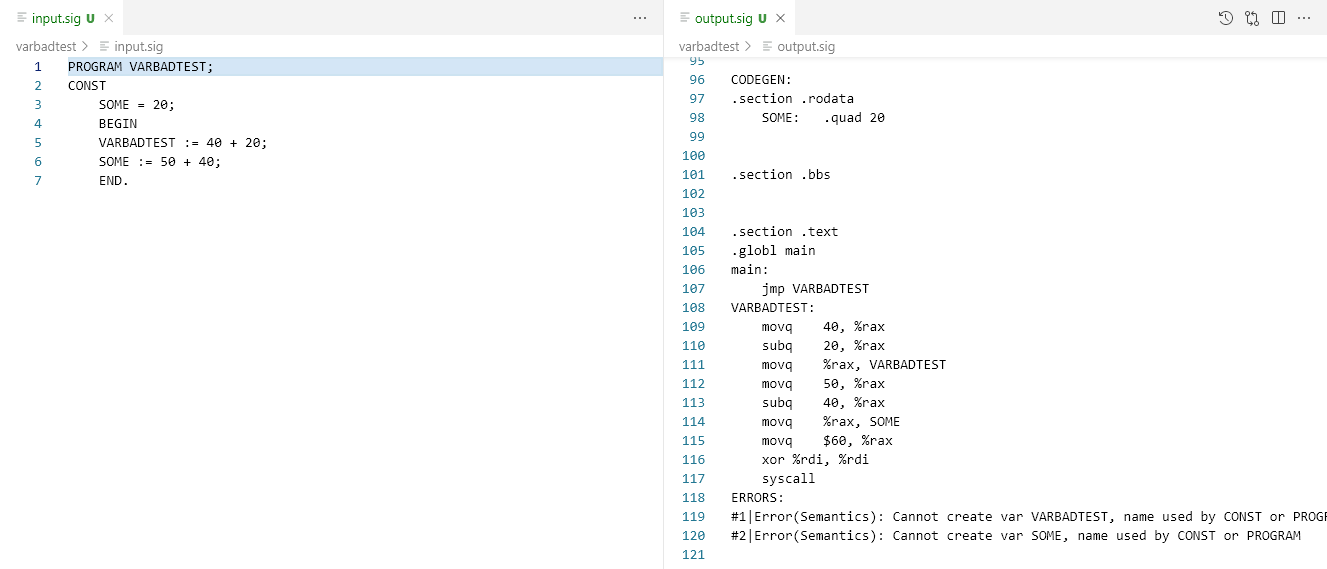
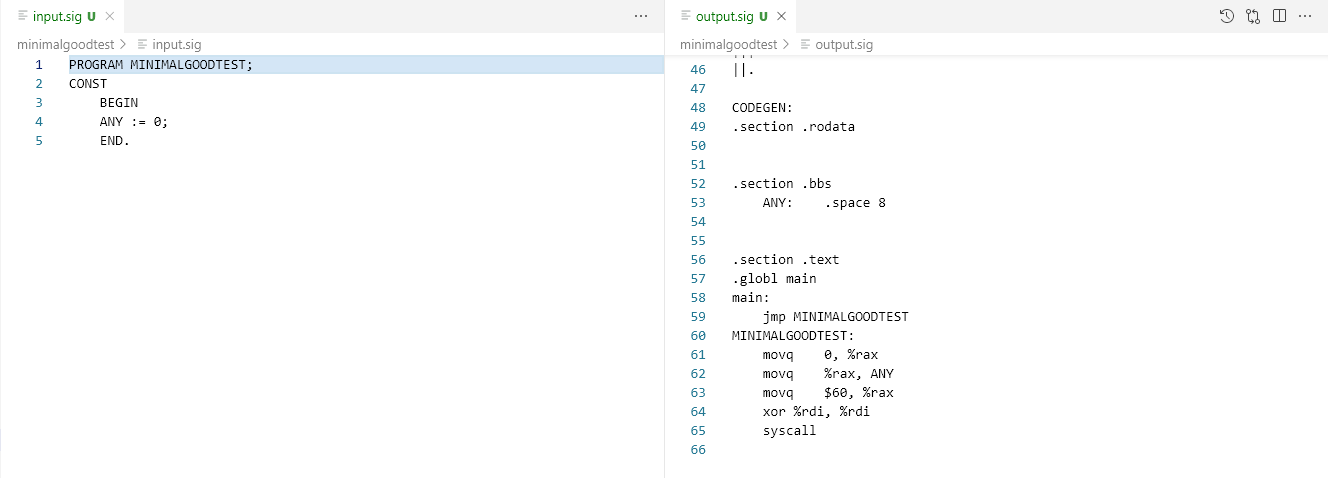
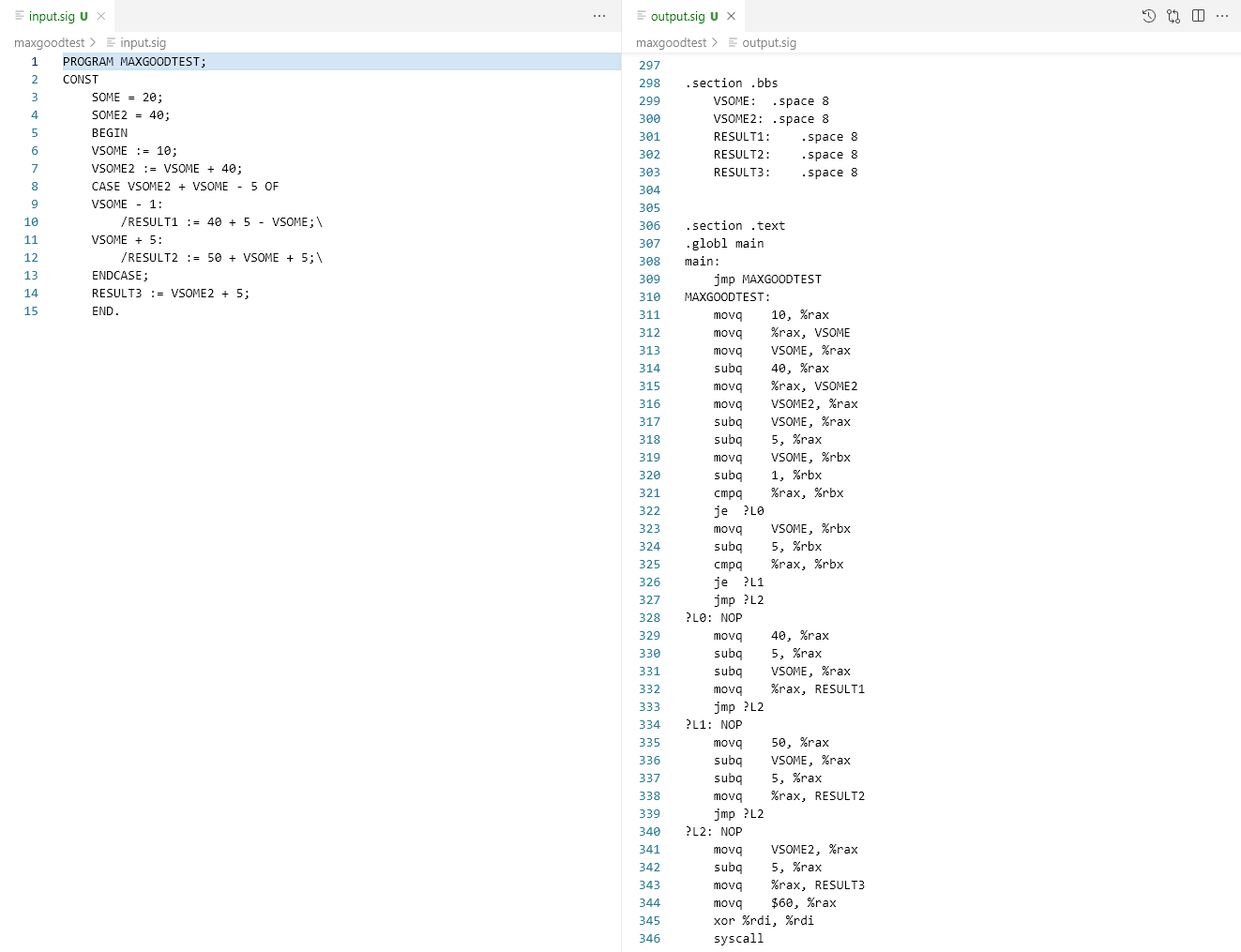
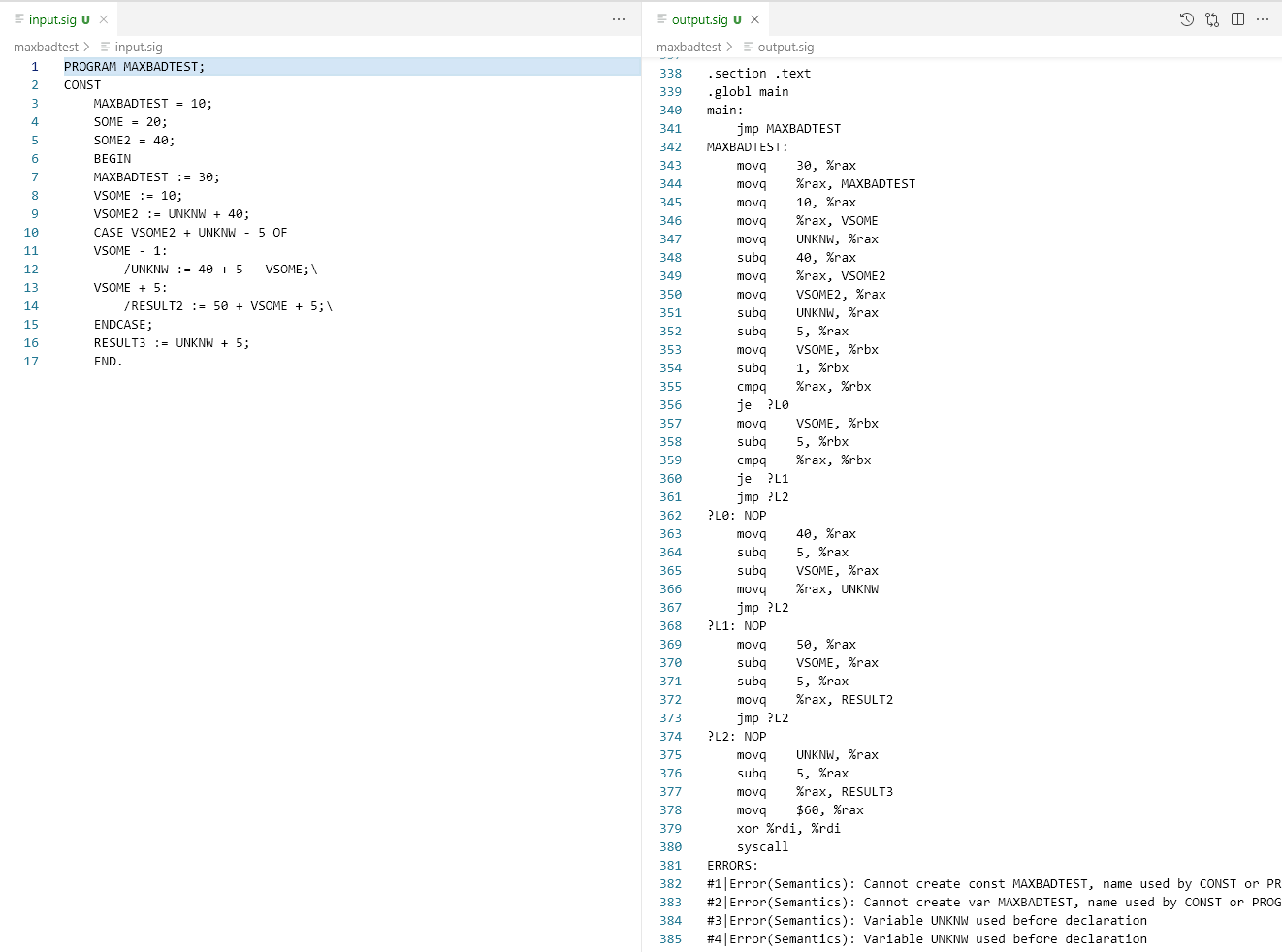
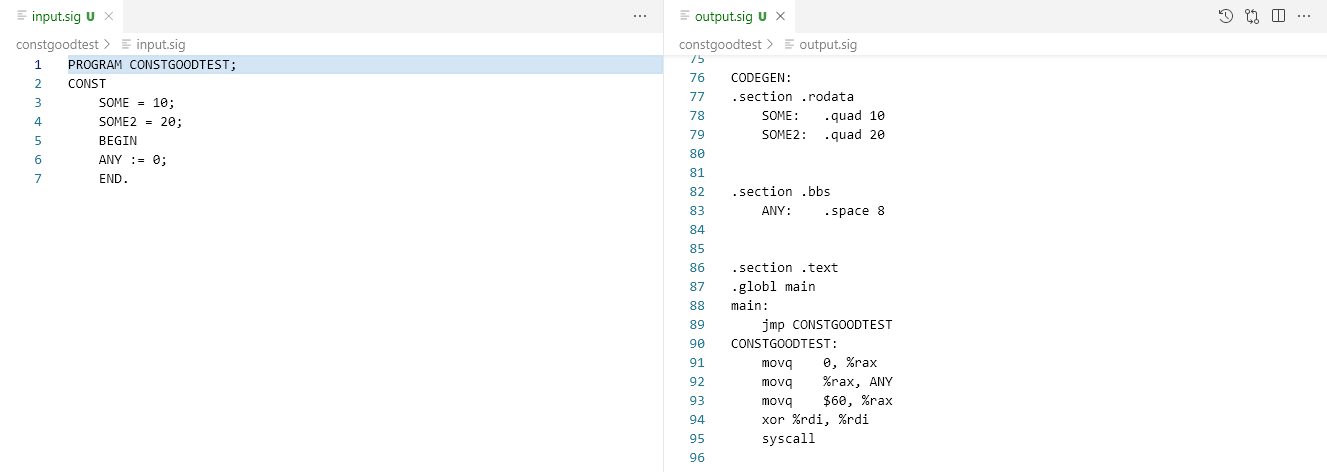
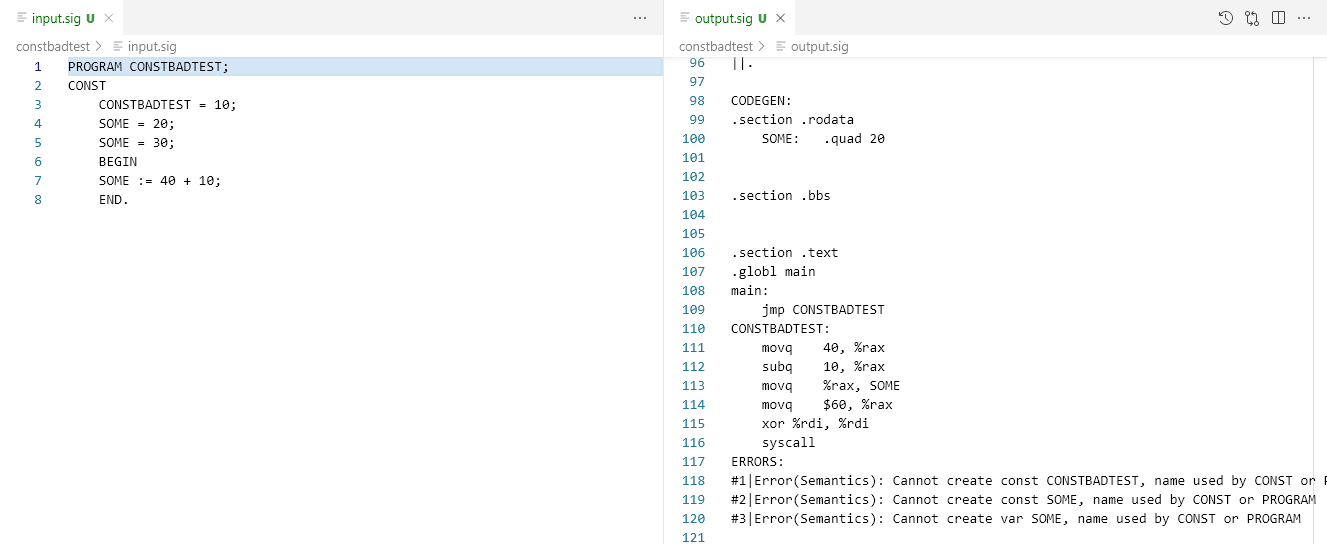
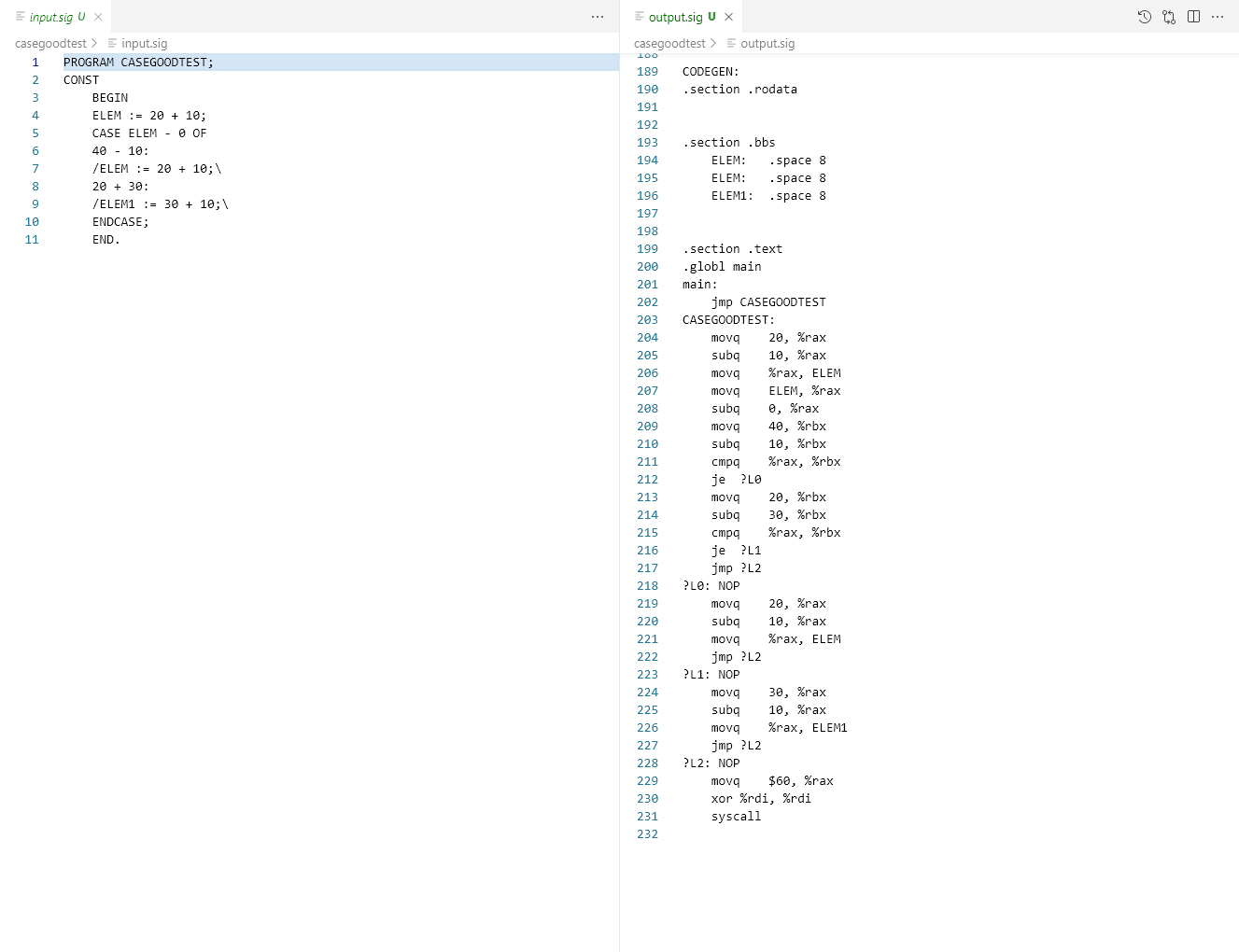
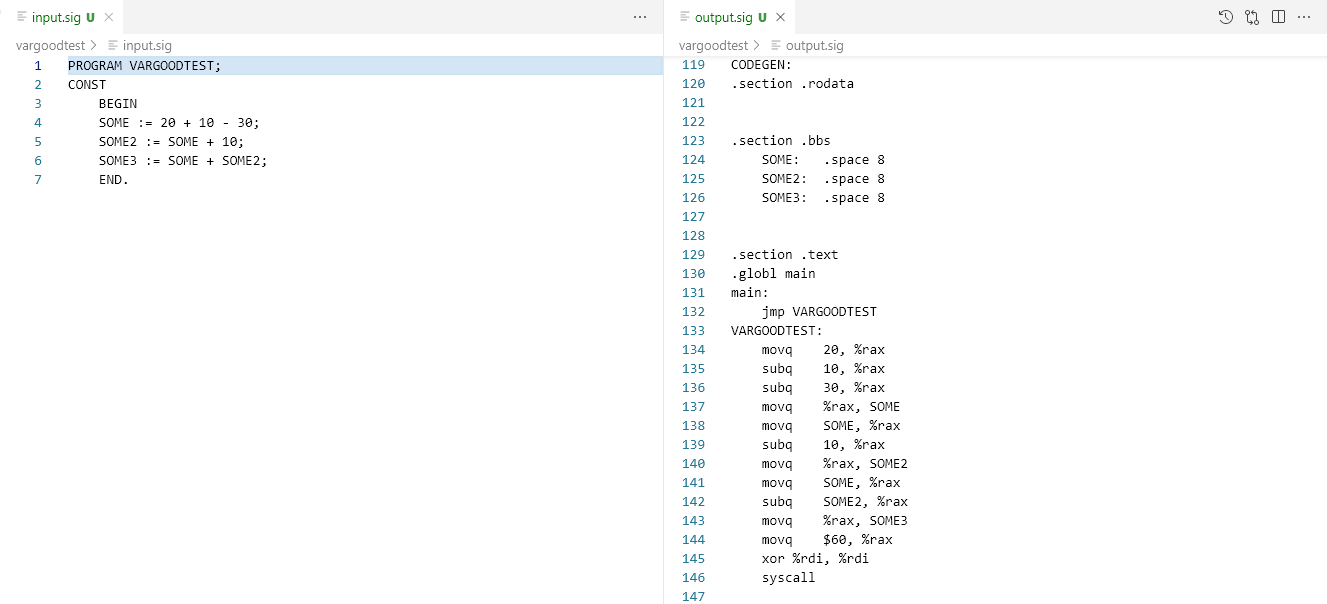
  } while (!(c == EOF || d == EOF));

  fclose(\_out);

  fclose(\_ver);

}

*Тестування*

**

*Всі тести можна запустити з make test\_semant*