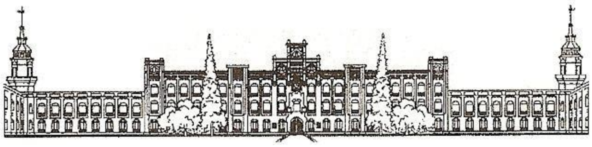


Міністерство освіти і науки України
Національний технічний університет України
«Київський політехнічний інститут»

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.

3. 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;
```

```c
  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;
}
```

```c
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) {
```

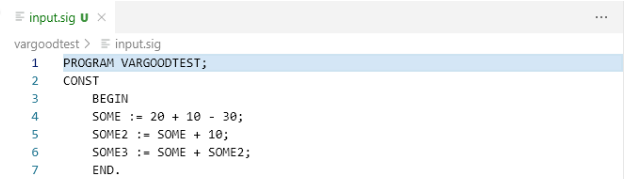
```
#ifdef WIN
    _fp = fopen(_file, "w+");
#endif
#ifndef WIN
    _fp = fopen(_file, "w");
#endif

    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);
  }
```

```c
}

==> 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)
```

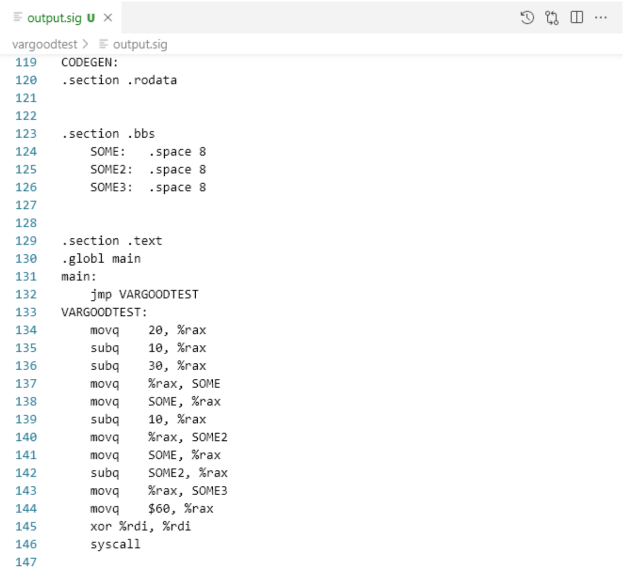
```
    critical = "Error";
  if (state == LEXER_STATE)
    if (error.hasLineColumn)
      printf("#%lld|%s(Lexer)| Line->%lld, Column->%lld |: %s\n", error.number,
             critical, error.row, error.col, error._error_message);
    else
      printf("#%lld|%s(Lexer): %s\n", error.number, critical,
             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);
}
```

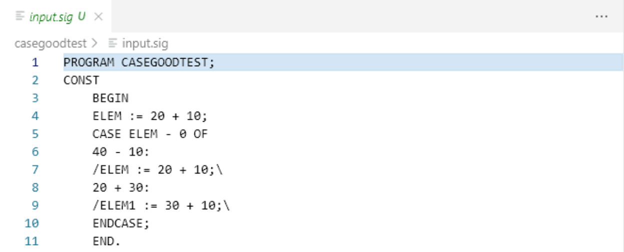
Тестування



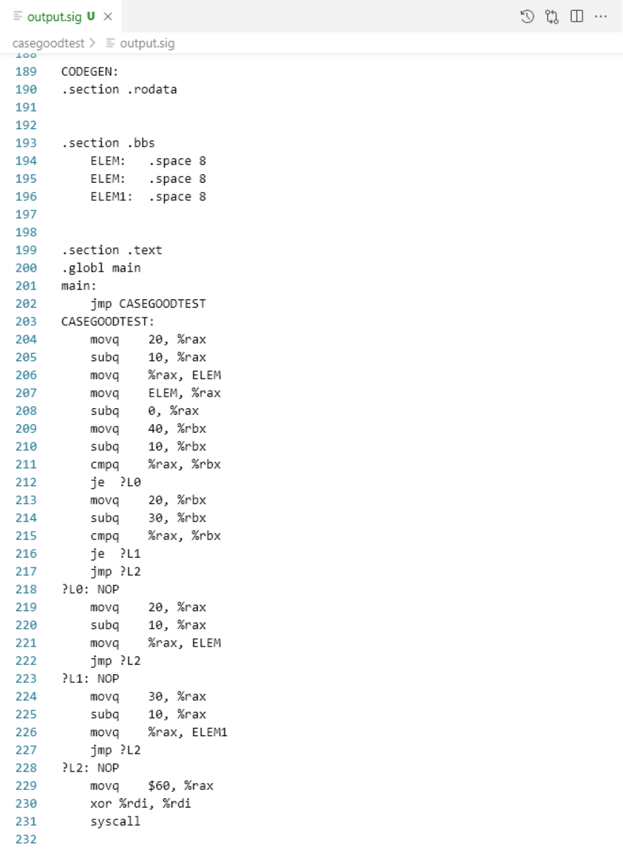
```
PROGRAM VARGOODTEST;
CONST
    BEGIN
    SOME := 20 + 10 - 30;
    SOME2 := SOME + 10;
    SOME3 := SOME + SOME2;
    END.
```



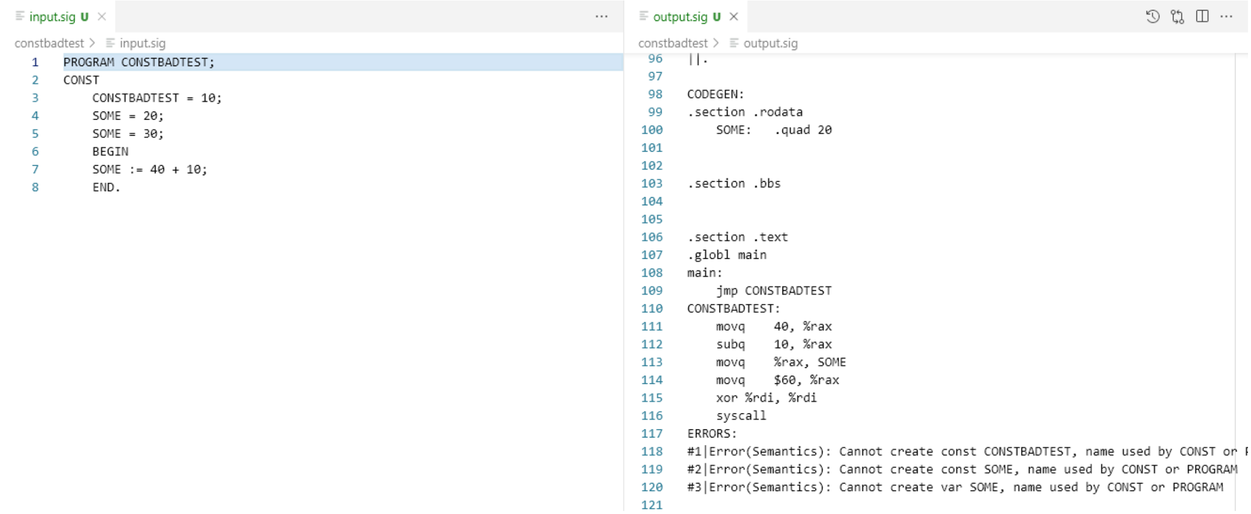
```
CODEGEN:
.section .rodata


.section .bbs
    SOME:   .space 8
    SOME2:  .space 8
    SOME3:  .space 8


.section .text
.globl main
main:
    jmp VARGOODTEST
VARGOODTEST:
    movq    20, %rax
    subq    10, %rax
    subq    30, %rax
    movq    %rax, SOME
    movq    SOME, %rax
    subq    10, %rax
    movq    %rax, SOME2
    movq    SOME, %rax
    subq    SOME2, %rax
    movq    %rax, SOME3
    movq    $60, %rax
    xor %rdi, %rdi
    syscall
```



```
PROGRAM CASEGOODTEST;
CONST
    BEGIN
    ELEM := 20 + 10;
    CASE ELEM - 0 OF
    40 - 10:
    /ELEM := 20 + 10;\
    20 + 30:
    /ELEM1 := 30 + 10;\
    ENDCASE;
    END.
```

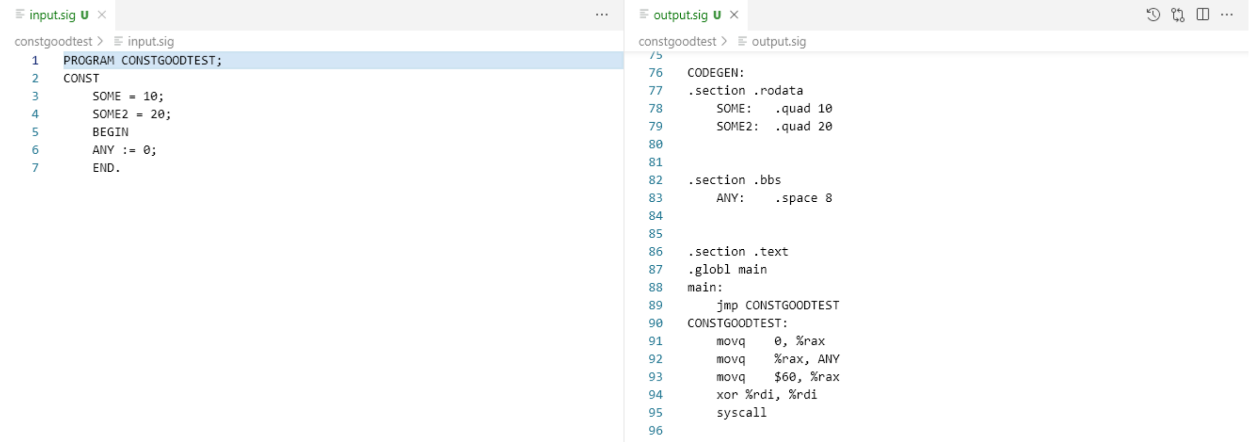


```
CODEGEN:
.section .rodata


.section .bbs
    ELEM:   .space 8
    ELEM:   .space 8
    ELEM1:  .space 8


.section .text
.globl main
main:
    jmp CASEGOODTEST
CASEGOODTEST:
    movq    20, %rax
    subq    10, %rax
    movq    %rax, ELEM
    movq    ELEM, %rax
    subq    0, %rax
    movq    40, %rbx
    subq    10, %rbx
    cmpq    %rax, %rbx
    je  ?L0
    movq    20, %rbx
    subq    30, %rbx
    cmpq    %rax, %rbx
    je  ?L1
    jmp ?L2
?L0: NOP
    movq    20, %rax
    subq    10, %rax
    movq    %rax, ELEM
    jmp ?L2
?L1: NOP
    movq    30, %rax
    subq    10, %rax
    movq    %rax, ELEM1
    jmp ?L2
?L2: NOP
    movq    $60, %rax
    xor %rdi, %rdi
    syscall
```



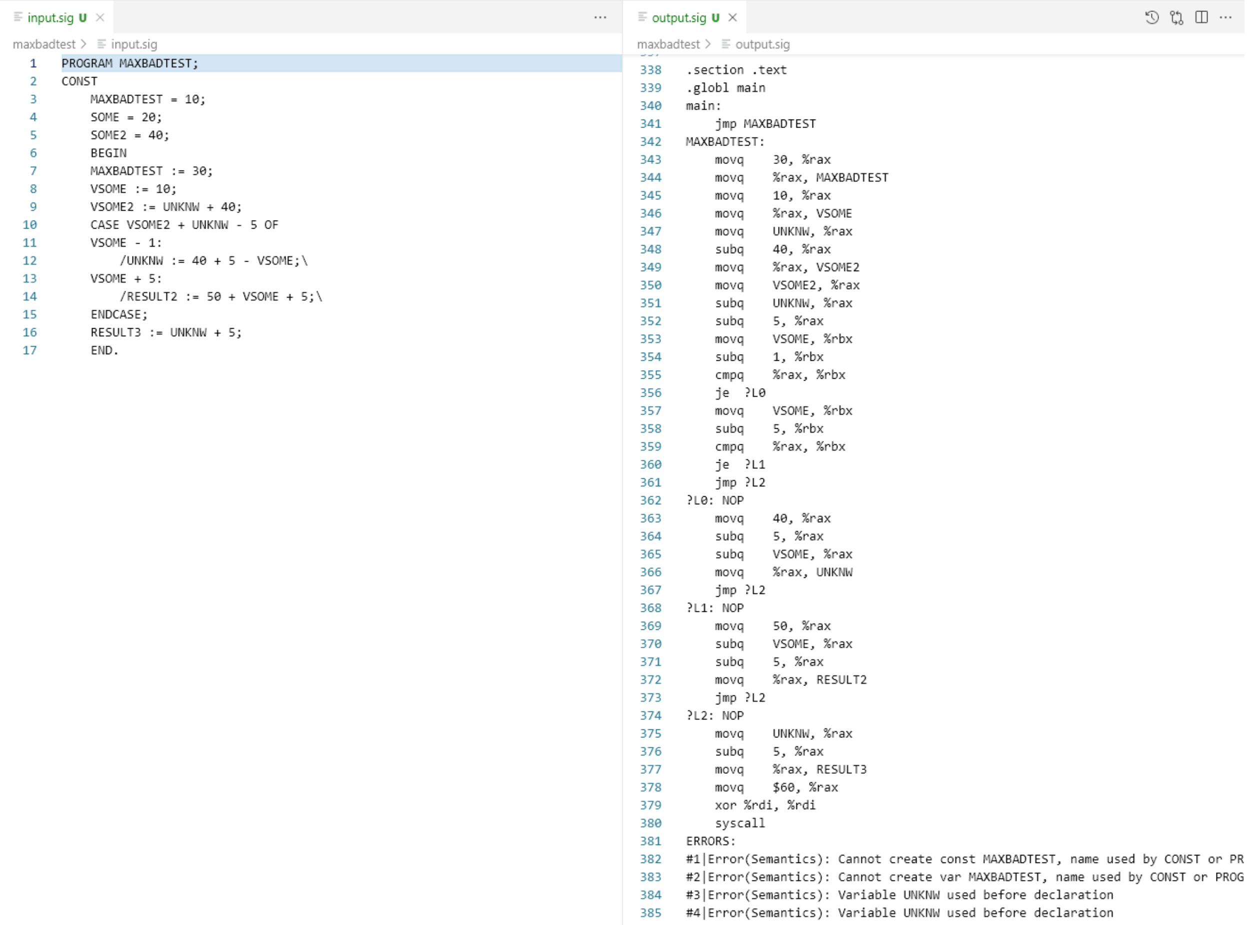
```
PROGRAM CONSTBADTEST;
CONST
    CONSTBADTEST = 10;
    SOME = 20;
    SOME = 30;
    BEGIN
    SOME := 40 + 10;
    END.
```

```
||.

CODEGEN:
.section .rodata
    SOME:   .quad 20


.section .bbs


.section .text
.globl main
main:
    jmp CONSTBADTEST
CONSTBADTEST:
    movq    40, %rax
    subq    10, %rax
    movq    %rax, SOME
    movq    $60, %rax
    xor %rdi, %rdi
    syscall
ERRORS:
#1|Error(Semantics): Cannot create const CONSTBADTEST, name used by CONST or
#2|Error(Semantics): Cannot create const SOME, name used by CONST or PROGRAM
#3|Error(Semantics): Cannot create var SOME, name used by CONST or PROGRAM

```



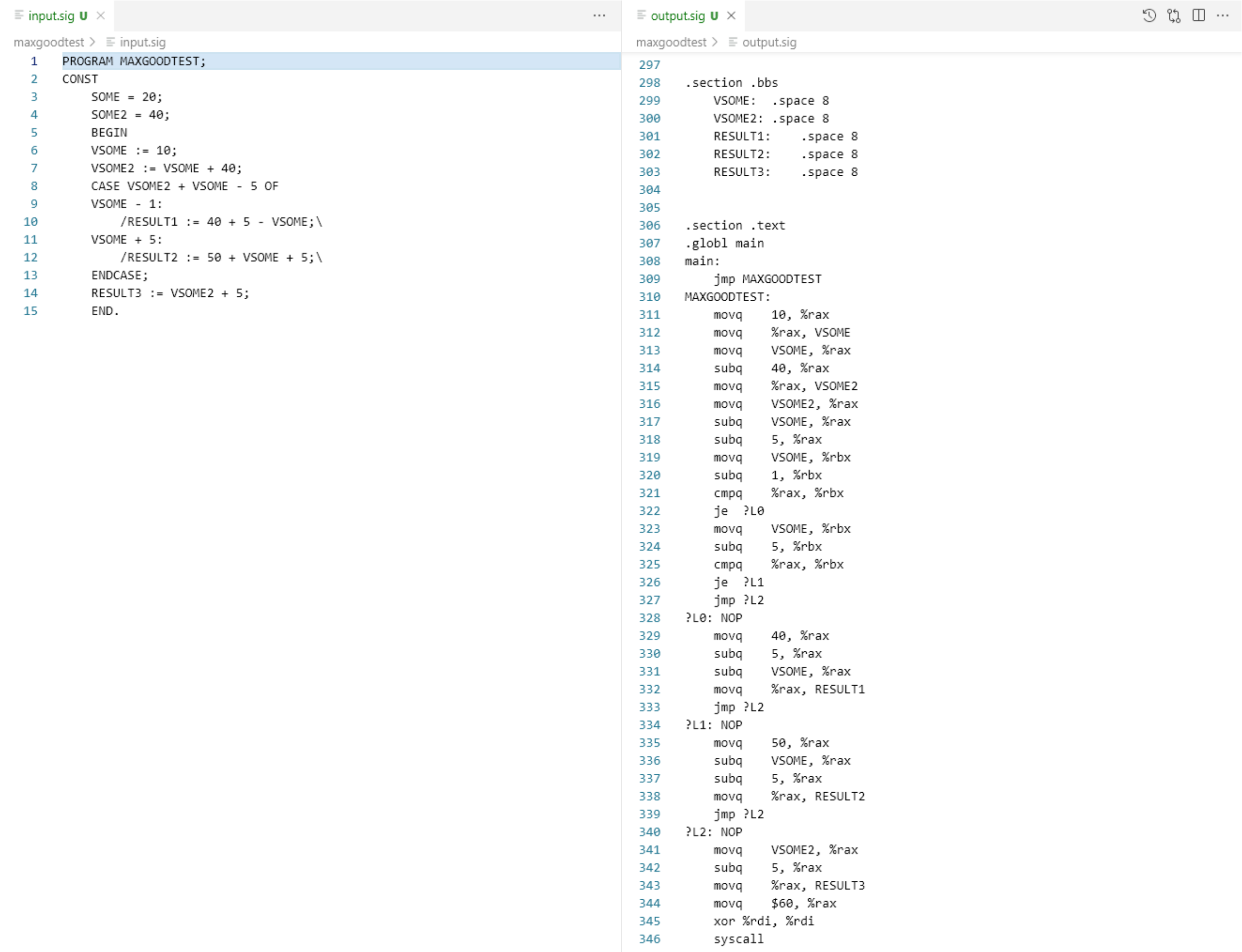
```
PROGRAM CONSTGOODTEST;
CONST
    SOME = 10;
    SOME2 = 20;
    BEGIN
    ANY := 0;
    END.
```

```
CODEGEN:
.section .rodata
    SOME:   .quad 10
    SOME2:  .quad 20


.section .bbs
    ANY:    .space 8


.section .text
.globl main
main:
    jmp CONSTGOODTEST
CONSTGOODTEST:
    movq    0, %rax
    movq    %rax, ANY
    movq    $60, %rax
    xor %rdi, %rdi
    syscall

```



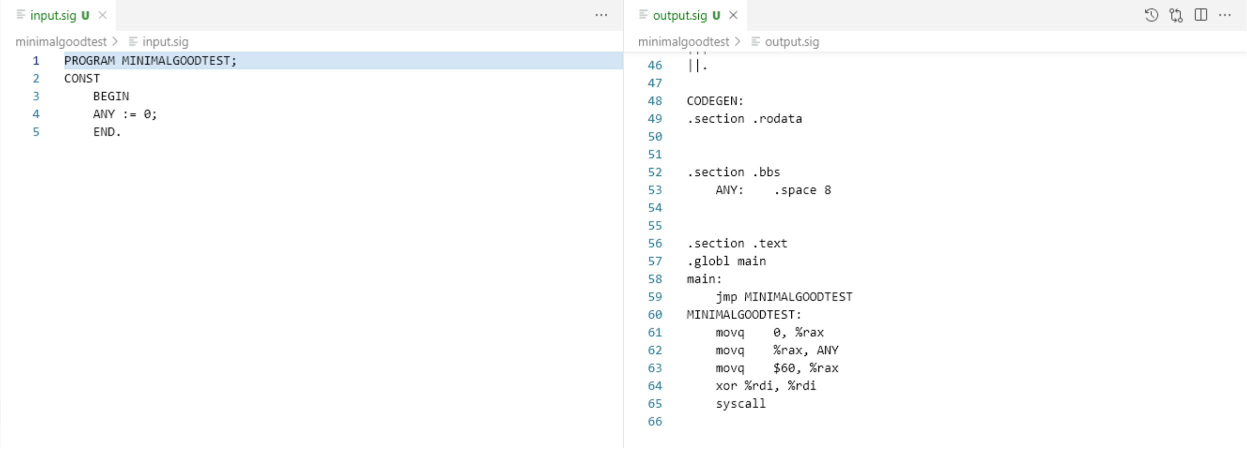
```
PROGRAM MAXBADTEST;
CONST
    MAXBADTEST = 10;
    SOME = 20;
    SOME2 = 40;
    BEGIN
    MAXBADTEST := 30;
    VSOME := 10;
    VSOME2 := UNKNW + 40;
    CASE VSOME2 + UNKNW - 5 OF
    VSOME - 1:
        /UNKNW := 40 + 5 - VSOME;\
    VSOME + 5:
        /RESULT2 := 50 + VSOME + 5;\
    ENDCASE;
    RESULT3 := UNKNW + 5;
    END.
```

```
.section .text
.globl main
main:
    jmp MAXBADTEST
MAXBADTEST:
    movq    30, %rax
    movq    %rax, MAXBADTEST
    movq    10, %rax
    movq    %rax, VSOME
    movq    UNKNW, %rax
    subq    40, %rax
    movq    %rax, VSOME2
    movq    VSOME2, %rax
    subq    UNKNW, %rax
    subq    5, %rax
    movq    VSOME, %rbx
    subq    1, %rbx
    cmpq    %rax, %rbx
    je  ?L0
    movq    VSOME, %rbx
    subq    5, %rbx
    cmpq    %rax, %rbx
    je  ?L1
    jmp ?L2
?L0: NOP
    movq    40, %rax
    subq    5, %rax
    subq    VSOME, %rax
    movq    %rax, UNKNW
    jmp ?L2
?L1: NOP
    movq    50, %rax
    subq    VSOME, %rax
    subq    5, %rax
    movq    %rax, RESULT2
    jmp ?L2
?L2: NOP
    movq    UNKNW, %rax
    subq    5, %rax
    movq    %rax, RESULT3
    movq    $60, %rax
    xor %rdi, %rdi
    syscall
ERRORS:
#1|Error(Semantics): Cannot create const MAXBADTEST, name used by CONST or PR
#2|Error(Semantics): Cannot create var MAXBADTEST, name used by CONST or PROG
#3|Error(Semantics): Variable UNKNW used before declaration
#4|Error(Semantics): Variable UNKNW used before declaration
```



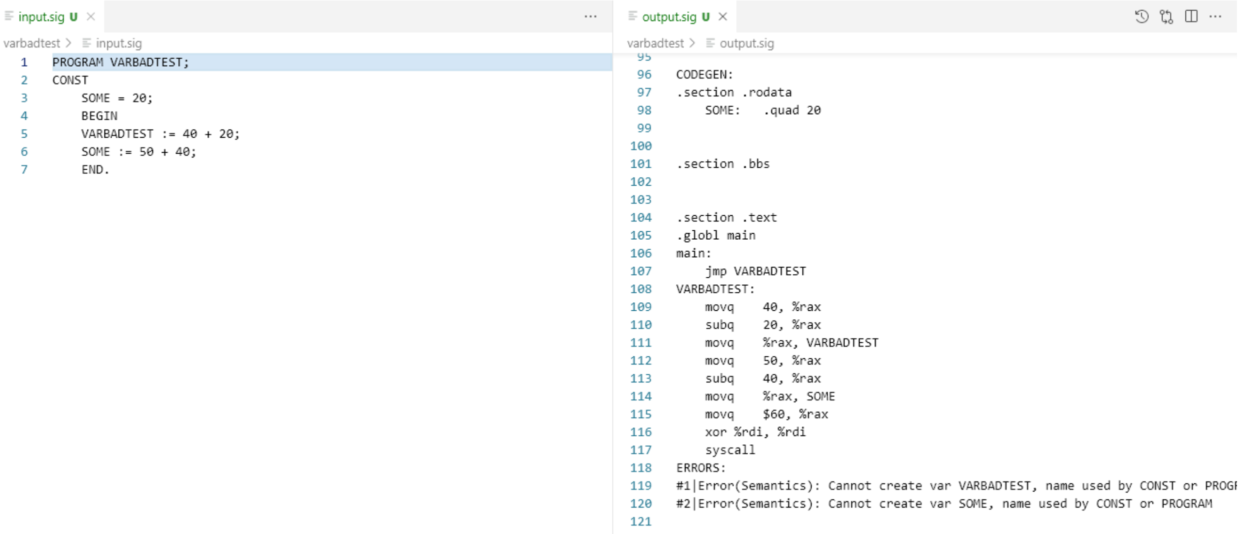
```
PROGRAM MAXGOODTEST;
CONST
    SOME = 20;
    SOME2 = 40;
    BEGIN
    VSOME := 10;
    VSOME2 := VSOME + 40;
    CASE VSOME2 + VSOME - 5 OF
    VSOME - 1:
        /RESULT1 := 40 + 5 - VSOME;\
    VSOME + 5:
        /RESULT2 := 50 + VSOME + 5;\
    ENDCASE;
    RESULT3 := VSOME2 + 5;
    END.
```

```

.section .bbs
    VSOME:  .space 8
    VSOME2: .space 8
    RESULT1:    .space 8
    RESULT2:    .space 8
    RESULT3:    .space 8


.section .text
.globl main
main:
    jmp MAXGOODTEST
MAXGOODTEST:
    movq    10, %rax
    movq    %rax, VSOME
    movq    VSOME, %rax
    subq    40, %rax
    movq    %rax, VSOME2
    movq    VSOME2, %rax
    subq    VSOME, %rax
    subq    5, %rax
    movq    VSOME, %rbx
    subq    1, %rbx
    cmpq    %rax, %rbx
    je  ?L0
    movq    VSOME, %rbx
    subq    5, %rbx
    cmpq    %rax, %rbx
    je  ?L1
    jmp ?L2
?L0: NOP
    movq    40, %rax
    subq    5, %rax
    subq    VSOME, %rax
    movq    %rax, RESULT1
    jmp ?L2
?L1: NOP
    movq    50, %rax
    subq    VSOME, %rax
    subq    5, %rax
    movq    %rax, RESULT2
    jmp ?L2
?L2: NOP
    movq    VSOME2, %rax
    subq    5, %rax
    movq    %rax, RESULT3
    movq    $60, %rax
    xor %rdi, %rdi
    syscall
```



```
PROGRAM MINIMALGOODTEST;
CONST
    BEGIN
    ANY := 0;
    END.
```

```
||.

CODEGEN:
.section .rodata


.section .bbs
    ANY:    .space 8


.section .text
.globl main
main:
    jmp MINIMALGOODTEST
MINIMALGOODTEST:
    movq    0, %rax
    movq    %rax, ANY
    movq    $60, %rax
    xor %rdi, %rdi
    syscall

```



```
PROGRAM VARBADTEST;
CONST
    SOME = 20;
    BEGIN
    VARBADTEST := 40 + 20;
    SOME := 50 + 40;
    END.
```

```
CODEGEN:
.section .rodata
    SOME:   .quad 20


.section .bbs


.section .text
.globl main
main:
    jmp VARBADTEST
VARBADTEST:
    movq    40, %rax
    subq    20, %rax
    movq    %rax, VARBADTEST
    movq    50, %rax
    subq    40, %rax
    movq    %rax, SOME
    movq    $60, %rax
    xor %rdi, %rdi
    syscall
ERRORS:
#1|Error(Semantics): Cannot create var VARBADTEST, name used by CONST or PROG
#2|Error(Semantics): Cannot create var SOME, name used by CONST or PROGRAM
```

Всі тести можна запустити з make test_semant