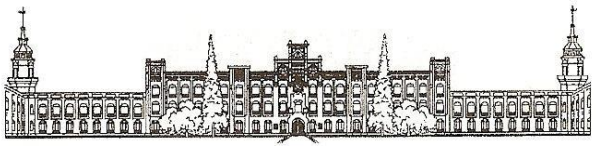

Міністерство освіти і науки України
Національний технічний університет України
«Київський політехнічний інститут»

Розрахунково-графічна робота

з дисципліни «ОСНОВИ ПРОЕКТУВАННЯ ТРАНСЛЯТОРІВ»

«РОЗРОБКА СИНТАКСИЧНОГО АНАЛІЗАТОРА»

Виконав студент групи: КВ-11

ПІБ: Терентьєв Іван Дмитрович

Перевірив: _____

Київ 2024

Постановка задачі

1. Розробити програму синтаксичного аналізатора (СА) для підмножини мови програмування SIGNAL згідно граматики за варіантом.

2. Програма має забезпечувати наступне:

• читання рядка лексем та таблиць, згенерованих лексичним

аналізатором, який було розроблено в лабораторній роботі «Розробка лексичного аналізатора»;

• синтаксичний аналіз (розбір) програми, поданої рядком лексем (алгоритм синтаксичного аналізатора вибирається за варіантом);

• побудову дерева розбору;

• формування таблиць ідентифікаторів та різних констант з повною інформацією, необхідною для генерування коду;

• формування лістингу вхідної програми з повідомленнями про

лексичні та синтаксичні 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
```

Таблиця переходів машини Кнута

```
1.    /*   ADDR ADDR_TO TERM AT AF_ADDR*/
2.    /*<signal-program> --> <program> */
3.    rule(0, SIGNAL_PROGRAM, NULL, false, ERROR);
4.    rule(1, PROGRAM, NULL, false, ERROR);
5.    rule(2, SIGNAL_PROGRAM_FINISH, NULL, true, ERROR);
6.    /*<program> --> PROGRAM <procedure-identifier> ; <block> .*/
7.    rule(3, 0, "PROGRAM", false, ERROR);
8.    rule(4, PROCEDURE_IDENTIFIER, NULL, false, ERROR);
9.    rule(5, 0, ";", false, ERROR);
10.         rule(6, BLOCK, NULL, false, ERROR);
11.         rule(7, 0, ".", true, ERROR);
12.         /*<block> --> <declarations> BEGIN <statements-list> END*/
13.         rule(8, DECLARATIONS, NULL, false, ERROR);
14.         rule(9, 0, "BEGIN", false, ERROR);
15.         rule(10, STATEMENTS_LIST, NULL, false, ERROR);
16.         rule(11, 0, "END", true, ERROR);
17.         /*<declarations> --> <constant-declarations>*/
18.         rule(12, CONSTANT_DECLARATIONS, NULL, true, ERROR);
19.         /*<constant-declarations> --> CONST <constant-declarations-
    list> | <empty>*/
20.         rule(13, 0, "CONST", false, ERROR);
21.         rule(14, CONSTANT_DECLARATIONS_LIST, NULL, true, 15);
22.         rule(15, EMPTY, NULL, true, ERROR);
23.         /*<constant-declarations-list> --> <constantdeclaration>
24.          * <constant-declarations-list> | <empty>*/
25.         rule(16, CONSTANT_DECLARATION, NULL, false, ERROR);
26.         rule(17, CONSTANT_DECLARATIONS_LIST, NULL, true, 18);
27.         rule(18, EMPTY, NULL, true, ERROR);
28.         /*<constant-declaration> --> <constant-identifier> =
    <constant>;*/
29.         rule(19, CONSTANT_IDENTIFIER, NULL, false, ERROR);
30.         rule(20, 0, "=", false, ERROR);
31.         rule(21, CONSTANT, NULL, false, ERROR);
32.         rule(22, 0, ";", true, ERROR);
33.         /*<statements-list> --> <statement> <statement-list> |
    <empty>*/
34.         rule(23, STATEMENT, NULL, false, ERROR);
35.         rule(24, STATEMENTS_LIST, NULL, true, 25);
36.         rule(25, EMPTY, NULL, true, ERROR);
37.         /*<statement> --> CASE <expression> OF <alternativeslist>
    ENDCASE ;|
38.         <variable-identifier> := <expression> ;*/
39.         rule(26, 0, "CASE", false, 32);
40.         rule(27, EXPRESSION, NULL, false, ERROR);
41.         rule(28, 0, "OF", false, ERROR);
42.         rule(29, ALTERNATIVES_LIST, NULL, false, ERROR);
43.         rule(30, 0, "ENDCASE", false, ERROR);
44.         rule(31, 0, ";", true, ERROR);
45.         rule(32, VARIABLE_IDENTIFIER, NULL, false, ERROR);
46.         rule(33, 0, ":=", false, ERROR);
47.         rule(34, EXPRESSION, NULL, false, ERROR);
48.         rule(35, 0, ";", true, ERROR);
49.         /*<alternatives-list> --> <alternative> <alternativeslist> |
    <empty>*/
50.         rule(36, ALTERNATIVE, NULL, false, ERROR);
51.         rule(37, ALTERNATIVES_LIST, NULL, true, 38);
52.         rule(38, EMPTY, NULL, true, ERROR);
```

```
53.         /*<alternative> --> <expression> : /<statements-list>\*/
54.         rule(39, EXPRESSION, NULL, false, ERROR);
55.         rule(40, 0, ":", false, ERROR);
56.         rule(41, 0, "/", false, ERROR);
57.         rule(42, STATEMENTS_LIST, NULL, false, ERROR);
58.         rule(43, 0, "\\", true, ERROR);
59.         /*<expression> --> <summand> <summands-list> | - <summand>
    <summands-list>*/
60.         rule(44, SUMMAND, NULL, false, 46);
61.         rule(45, SUMMANDS_LIST, NULL, true, ERROR);
62.         rule(46, 0, "-", false, ERROR);
63.         rule(47, SUMMAND, NULL, false, ERROR);
64.         rule(48, SUMMANDS_LIST, NULL, true, ERROR);
65.         /*<summands-list> --> <add-instruction> <summand> |
    <summands-list> |
66.          * <empty>*/
67.         rule(49, ADD_INSTRUCTION, NULL, false, ERROR);
68.         rule(50, SUMMAND, NULL, true, 51);
69.         rule(51, SUMMANDS_LIST, NULL, true, 52);
70.         rule(52, EMPTY, NULL, true, ERROR);
71.         /*<add-instruction> --> + | -*/
72.         rule(53, 0, "+", true, 54);
73.         rule(54, 0, "-", true, ERROR);
74.         /*<summand> --> <variable-identifier> | <unsigned-integer>*/
75.         rule(55, VARIABLE_IDENTIFIER, NULL, true, 56);
76.         rule(56, UNSIGNED_INTEGER, NULL, true, ERROR);
77.         /*<constant> --> <unsigned-integer>*/
78.         rule(57, UNSIGNED_INTEGER, NULL, true, ERROR);
79.         /*<variable-identifier> --> <identifier>*/
80.       rule(58, IDENTIFIER, NULL, true, ERROR);
81.         /*<constant-identifier> --> <identifier>*/
82.       rule(59, IDENTIFIER, NULL, true, ERROR);
83.         /*<procedure-identifier> --> <identifier>*/
84.       rule(60, IDENTIFIER, NULL, true, ERROR);
85.
86.         rule(UNSIGNED_INTEGER, 0, "", true, ERROR);
87.         rule(IDENTIFIER, 0, "", true, ERROR);
88.         rule(STRING, 0, "", true, ERROR);
89.         rule(EMPTY, 0, "", true, ERROR);
```

Тестування

Тест 1

```
1. (**)
2. PROGRAM TEST02;
3. CONST
4.     ELEM3 = 45;
5.     BEGUN
6.     ELEM4 := 30 + 45;
7.     CASE ELEM3 - ELEM4
8.     OF
9.     ELEM2 + ELEM1 :
10.          /ELEM5 := 20 - 10;\
11.          ENDCASE;
12.          END. (*(*End of file*)
```

```
1. SYNTAX:
2. <signal-program>
3.
4. ERRORS:
5. #1|Error(Syntax)| Line->3, Column->2 |: 'BEGIN' expected, but
   'BEGUN' found.
```

Tecm 2

```
1. (**)
2. PROGRAM TEST02;
3. CONST
4.     ELEM3 = 45
5.     BEGIN
6.     ELEM4 := 30 + 45;
7.     CASE ELEM3 - ELEM4
8.     OF
9.     ELEM2 + ELEM1 :
10.          /ELEM5 := 20 - 10;\
11.          ENDCASE;
12.          END. (*(*End of file*)
```

```
1. SYNTAX:
2. <signal-program>
3.
4. ERRORS:
5. #1|Error(Syntax)| Line->3, Column->2 |: 'BEGIN' expected, but
   'ELEM3' found.
```

Tecm 3

```
1. (**)
2. PROGRAM TEST02;
3. CONST
4.     ELEM3 = 45;
5.     BEGIN
6.     ELEM4 := 30 + 45;
7.     CASE ELEM3 - ELEM4
8.     OF
9.     ELEM2 + ELEM1 :
10.          /ELEM5 := 20 - 10;\
11.          ENDCASE;
12.          END. (*(*End of file*)
```

```
1. SYNTAX:
2. <signal-program>
3. |<program>
4. ||PROGRAM
5. ||<procedure-identifier>
6. |||<identifier>
7. ||||TEST02
8. ||;
9. ||<block>
10.      |||<declarations>
11.      ||||<constant-declarations>
12.      |||||CONST
13.      |||||<constant-declarations-list>
14.      ||||||<constant-declaration>
15.      |||||||<constant-identifier>
16.      ||||||||<identifier>
17.      |||||||||ELEM3
18.      |||||||=
19.      |||||||<constant>
20.      ||||||||<unsigned-integer>
21.      |||||||||45
22.      |||||||;
23.      ||||||<empty>
24.      |||||||<empty>
25.      |||BEGIN
26.      |||<statements-list>
27.      ||||<statement>
28.      |||||<variable-identifier>
29.      ||||||<identifier>
30.      |||||||ELEM4
31.      |||||:=
32.      |||||<expression>
33.      ||||||<summand>
34.      |||||||<unsigned-integer>
35.      ||||||||30
36.      ||||||<summands-list>
37.      |||||||<add-instruction>
38.      ||||||||+
39.      |||||||<summand>
40.      ||||||||<unsigned-integer>
41.      |||||||||45
42.      |||||;
43.      ||||<statements-list>
44.      |||||<statement>
```

```
45.     ||||||CASE
46.     ||||||<expression>
47.     |||||||<summand>
48.     ||||||||<variable-identifier>
49.     |||||||||<identifier>
50.     |||||||||ELEM3
51.     |||||||<summands-list>
52.     ||||||||<add-instruction>
53.     |||||||||-
54.     ||||||||<summand>
55.     |||||||||<variable-identifier>
56.     ||||||||||<identifier>
57.     ||||||||||ELEM4
58.     ||||||OF
59.     ||||||<alternatives-list>
60.     |||||||<alternative>
61.     ||||||||<expression>
62.     |||||||||<summand>
63.     ||||||||||<variable-identifier>
64.     |||||||||||<identifier>
65.     |||||||||||ELEM2
66.     |||||||||<summands-list>
67.     ||||||||||<add-instruction>
68.     ||||||||||+
69.     ||||||||||<summand>
70.     ||||||||||<variable-identifier>
71.     |||||||||||<identifier>
72.     |||||||||||ELEM1
73.     ||||||||:
74.     ||||||||/
75.     ||||||||<statements-list>
76.     |||||||||<statement>
77.     ||||||||||<variable-identifier>
78.     |||||||||||<identifier>
79.     |||||||||||ELEM5
80.     ||||||||||:=
81.     ||||||||||<expression>
82.     |||||||||||<summand>
83.     ||||||||||||<unsigned-integer>
84.     ||||||||||||20
85.     ||||||||||<summands-list>
86.     ||||||||||||<add-instruction>
87.     ||||||||||||-
88.     ||||||||||||<summand>
89.     ||||||||||||<unsigned-integer>
90.     ||||||||||||10
91.     ||||||||||;
92.     |||||||||<empty>
93.     ||||||||||<empty>
94.     ||||||||\
95.     |||||||<empty>
96.     ||||||||<empty>
97.     ||||||ENDCASE
98.     ||||||;
99.     |||||<empty>
100.    ||||||<empty>
101.    |||END
102.    ||.
```

Tecm 4

```
1. PROGRAM TEST02;
2. CONST
3.     ELEM3 = 23;
4.     BEGIN
5.     ELEM4 := 5 - ELEM3;
6.     END.
```

```
1. SYNTAX:
2. <signal-program>
3. |<program>
4. ||PROGRAM
5. ||<procedure-identifier>
6. |||<identifier>
7. ||||TEST02
8. ||;
9. ||<block>
10.     |||<declarations>
11.     ||||<constant-declarations>
12.     |||||CONST
13.     |||||<constant-declarations-list>
14.     ||||||<constant-declaration>
15.     |||||||<constant-identifier>
16.     ||||||||<identifier>
17.     |||||||||ELEM3
18.     |||||||=
19.     |||||||<constant>
20.     ||||||||<unsigned-integer>
21.     |||||||||23
22.     |||||||;
23.     ||||||<empty>
24.     |||||||<empty>
25.     |||BEGIN
26.     |||<statements-list>
27.     ||||<statement>
28.     |||||<variable-identifier>
29.     ||||||<identifier>
30.     |||||||ELEM4
31.     |||||:=
32.     |||||<expression>
33.     ||||||<summand>
34.     |||||||<unsigned-integer>
35.     ||||||||5
36.     ||||||<summands-list>
37.     |||||||<add-instruction>
38.     ||||||||-
39.     |||||||<summand>
40.     ||||||||<variable-identifier>
41.     |||||||||<identifier>
42.     ||||||||||ELEM3
43.     |||||;
44.     ||||<empty>
45.     |||||<empty>
46.     |||END
47.     ||.
```

Tecm 5

```
1. PROGRAM TEST02;
2. CONST
3.     SOME4 = 4;
4.     BEGIN
5.     SOME4 := 5+SOME3;
6.     END.
```

```
1. SYNTAX:
2. <signal-program>
3. |<program>
4. ||PROGRAM
5. ||<procedure-identifier>
6. |||<identifier>
7. ||||TEST02
8. ||;
9. ||<block>
10.      |||<declarations>
11.      ||||<constant-declarations>
12.      |||||CONST
13.      |||||<constant-declarations-list>
14.      ||||||<constant-declaration>
15.      |||||||<constant-identifier>
16.      ||||||||<identifier>
17.      |||||||||SOME4
18.      |||||||=
19.      |||||||<constant>
20.      ||||||||<unsigned-integer>
21.      |||||||||4
22.      |||||||;
23.      ||||||<empty>
24.      |||||||<empty>
25.      |||BEGIN
26.      |||<statements-list>
27.      ||||<statement>
28.      |||||<variable-identifier>
29.      ||||||<identifier>
30.      |||||||SOME4
31.      |||||:=
32.      |||||<expression>
33.      ||||||<summand>
34.      |||||||<unsigned-integer>
35.      ||||||||5
36.      ||||||<summands-list>
37.      |||||||<add-instruction>
38.      ||||||||+
39.      |||||||<summand>
40.      ||||||||<variable-identifier>
41.      |||||||||<identifier>
42.      ||||||||||SOME3
43.      |||||;
44.      ||||<empty>
45.      |||||<empty>
46.      |||END
47.      ||.
```

Tecm 6

```
1. PROGRAM TEST02;
2. CONST
3.     SOME4 = 4;
4.     BEGIN
5.     SOME4 := 5 SOME3;
6.     END.
```

```
1. SYNTAX:
2. <signal-program>
3.
4. ERRORS:
5. #1|Error(Syntax)| Line->5, Column->17 |: '-' expected, but 'SOME3'
   found.
```

Код програми

```
==> cli.c <==
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

#include "cli.h"
#include "error.h"

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

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 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) {
    _fp = fopen(_file, "w");

    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 + 1];
        i++;
      } else if (strcmp(argv[i], "-o") == 0 && i + 1 < argc) {
        params._output_file = argv[i + 1];
        i++;
      } else if (strcmp(argv[i], "-v") == 0)
        params.verbose = 1;
    }
  }

  if (params._input_file == NULL) {
    add_to_errors(create_error_without_linecolumn(
        FILE_ACCESS, "Input filename is empty.", true));
  } else {
    check_file_access(params._input_file, true);
    check_file_access(params._output_file, false);
```

```
    check_file_missing(params._output_file);
  }
}

==> 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;
}
==> error.c <==
#include <bits/types.h>
#include <stdbool.h>
#include <stddef.h>
#include <stdio.h>
#include <stdlib.h>

#include "error.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(__uint8_t state, char
*_error_message,
                                      bool critical) {
  Error error = {errorCount + 1, state, _error_message, critical,
false, 0, 0,
                 NULL,           NULL,  false};
  return error;
}

Error create_error_with_linecolumn(__uint8_t 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;
}
==> 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;
}
==> 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"

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, __uint8_t 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;
}
==> knut_tables.c <==
#include "error.h"
#include "knut_tables.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, ERROR);
  rule(50, SUMMAND, NULL, true, 51);
  rule(51, SUMMANDS_LIST, NULL, true, 52);
  rule(52, EMPTY, NULL, true, ERROR);
  /*<add-instruction> --> + | -*/
  rule(53, 0, "+", true, 54);
  rule(54, 0, "-", true, ERROR);
  /*<summand> --> <variable-identifier> | <unsigned-integer>*/
  rule(55, VARIABLE_IDENTIFIER, NULL, true, 56);
  rule(56, UNSIGNED_INTEGER, NULL, true, ERROR);
  /*<constant> --> <unsigned-integer>*/
  rule(57, UNSIGNED_INTEGER, NULL, true, ERROR);
  /*<variable-identifier> --> <identifier>*/
  rule(58, IDENTIFIER, NULL, true, ERROR);
  /*<constant-identifier> --> <identifier>*/
  rule(59, IDENTIFIER, NULL, true, ERROR);
  /*<procedure-identifier> --> <identifier>*/
  rule(60, 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;
}

==> lexer.c <==
#include "bits/types.h"
#include "lexer.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);
  }
}

==> lexer_get.c <==
#include "lexer_get.h"

__uint8_t symbol_type(char symbol) {
  __uint8_t 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;
}

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_structure.c <==
#include <stdlib.h>

#include "error.h"
#include "lexer_structure.h"
void add_buffer_symbol() {
  lexer._buffer =
      (char *)realloc(lexer._buffer, (lexer.bufferSize + 1) *
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;
}
==> main.c <==
#include "lexer.h"
#include "out.h"
#include "syntax.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.verbose) {
    out_file_lexer();
    print_file_out();
  } else
    out_file_lexer();

  if (gotError) {
    print_errors();
    return -1;
  } else {
    just_clean();
    proc_syntax();
  }

  if (params.verbose) {
    out_file_syntax();
    print_file_out();
  } else
    out_file_syntax();

  free_trees();
  free_errors();
  free_tables();
  free_tokens();
  return 0;
}
==> out.c <==
#include <stdio.h>

#include "constant.h"
#include "identifier.h"
#include "lexer.h"
#include "out.h"
#include "strings.h"
#include "syntax.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";
  short int state = error.state;
  if (error.critical) critical = "Error";
  if (state == LEXER_STATE)
    if (error.hasLineColumn)
```

```
      printf("#%ld|%s(Lexer)| Line->%ld, Column->%ld |: %s\n",
error.number,
             critical, error.row, error.col, error._error_message);
    else
      printf("#%ld|%s(Lexer): %s\n", error.number, critical,
             error._error_message);
  else if (state == FILE_ACCESS)
    printf("#%ld|%s(File IO): %s\n", error.number, critical,
           error._error_message);
  else if (state == SYNTAX_STATE)
    printf("#%ld|%s(Syntax): %s\n", error.number, critical,
           error._error_message);
  else if (state == MEMORY_ACCESS)
    printf("#%ld|%s(Memory): %s\n", error.number, critical,
           error._error_message);
  else
    printf("#%ld|%s(Unknown): %s\n", error.number, critical,
           error._error_message);
}

void get_error(Error error, FILE *__output_file) {
  char *critical = "Warning";
  short int state = error.state;
  if (error.critical) critical = "Error";
  if (state == LEXER_STATE)
    if (error.hasLineColumn)
      fprintf(__output_file, "#%ld|%s(Lexer)| Line->%ld, Column->%ld
|: %s\n",
              error.number, critical, error.row, error.col,
              error._error_message);
    else
      fprintf(__output_file, "#%ld|%s(Lexer): %s\n", error.number,
critical,
              error._error_message);
  else if (state == FILE_ACCESS)
    fprintf(__output_file, "#%ld|%s(File IO): %s\n", error.number,
critical,
            error._error_message);
  else if (state == SYNTAX_STATE)
    fprintf(__output_file, "#%ld|%s(Syntax): %s\n", error.number,
critical,
            error._error_message);
  else if (state == MEMORY_ACCESS)
    fprintf(__output_file, "#%ld|%s(Memory): %s\n", error.number,
critical,
            error._error_message);
  else
    fprintf(__output_file, "#%ld|%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";
  if (error._here == NULL)
    error._here = "";
  fprintf(__output_file,
          "#%ld|%s(Syntax)| Line->%ld, Column->%ld |: \'%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: %lu\n", lexer.row);
  printf("Current col: %lu\n", lexer.col);
  printf("Current symbol: %c\n", lexer.symbol);
  printf("Current symbol type: %d\n", lexer.symbolType);
}
void print_token(Token token) {
  printf("[%lu][%lu] %lu: %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, "|%6ld|%6ld|%6ld|%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, "|");

  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 free_errors() { free(_errors); }

void free_tokens() { free(_tokens); }

void free_tables() {
  free(_constants);
  free(_identifiers);
  free(_strings);
}

void free_trees() { free_tree(_tree); }
==> 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;
}
```

```
==> 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;

size_t tokenIterator = 0;
char* _expected;

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

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{
    if(run.result->id >= tokenCount){
          add_to_errors(create_error_syntaxer(
        _tokens[tokenIterator].row, _tokens[tokenIterator].col,
        run.result->_value, ""));
    }else{
    add_to_errors(create_error_syntaxer(
        _tokens[tokenIterator].row, _tokens[tokenIterator].col,
        run.result->_value, _tokens[run.result->id]._data));
    }
  }
}

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(newTree,
create_node(_tokens[tokenIterator]._data, i));
            state = true;
          }
          break;
        case IDENTIFIER:
          if (is_identifier(_tokens[tokenIterator].code)) {
            add_branch(newTree,
create_node(_tokens[tokenIterator]._data, i));
            state = true;
          }
          break;
        case STRING:
          if (is_stringy(_tokens[tokenIterator].code)) {
            add_branch(newTree,
create_node(_tokens[tokenIterator]._data, i));
            state = true;
          }
          break;
        case EMPTY:
          add_branch(newTree, create_node(name_by_id(EMPTY), i));
          state = true;
          break;
        default:
          if (strcmp(rules(i).code._term,
_tokens[tokenIterator]._data) == 0) {
            add_branch(newTree,
create_node(_tokens[tokenIterator]._data, i));
            state = true;
          }
      };
      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);
          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;
}
==> token.c <==
#include "id_generator.h"
#include "token.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;
}

==> token_structure.c <==
#include "id_generator.h"
#include "token_structure.h"
Token create_token(size_t row, size_t col, char *_data, size_t
dataSize,
                   __uint8_t 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;
}
==> tree.c <==
#include "symbol_type.h"
#include "token.h"
#include "tree.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);
  }
}

==> cli.h <==
#ifndef CLI_H
#define CLI_H

#include "error.h"

struct params {
  char *_input_file;
  char *_output_file;
  bool verbose;
};
typedef struct params Params;

extern Params params;

void proc_cli(int argc, char *argv[]);

#endif
==> 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
==> error.h <==
#include <bits/types.h>
#include <stdbool.h>
#include <stddef.h>
#ifndef ERROR_H
#define ERROR_H

struct error {
  size_t number;
  __uint8_t 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

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(__uint8_t state, char*
_error_message,
                                      bool critical);
Error create_error_with_linecolumn(__uint8_t 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
==> 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
==> id_generator.h <==
#include <bits/types.h>
#include <stddef.h>
#ifndef ID_GENERATOR_H
#define ID_GENERATOR_H

size_t get_id(size_t row, size_t col, __uint8_t type);
#endif
==> 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
==> lexer_get.h <==
#include <bits/types.h>
#include <stdio.h>

#include "error.h"
#include "lexer_structure.h"
#include "token.h"
#ifndef LEXER_GET_H
#define LEXER_GET_H

__uint8_t symbol_type(char symbol);
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.h <==
#ifndef LEXER_H
#define LEXER_H
#include "lexer_get.h"

// Main procedure of lexer
void proc_lexer(char *_input_file);

#endif
==> lexer_structure.h <==
#include <bits/types.h>
#include <stdbool.h>
#include <stddef.h>

#include "symbol_type.h"
#ifndef LEXER_STRUCTURE_H
#define LEXER_STRUCTURE_H

struct lexer {
  char *_buffer;
  size_t bufferSize;
  size_t row;
  size_t col;
  char symbol;
  __uint8_t symbolType;
```

```
  bool inComment;
};
typedef struct lexer Lexer;
extern Lexer lexer;

void add_buffer_symbol();
void clean_buffer();
#endif
==> out.h <==
#include "cli.h"
#include "error.h"
#include "token_structure.h"
#include "tree.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 just_clean();
void free_trees();
void free_errors();
void free_tokens();
void free_tables();

#endif
==> 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
==> 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

#endif
==> 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);

#endif
==> 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 53
#define SUMMAND 55
#define CONSTANT 57
#define VARIABLE_IDENTIFIER 58
#define CONSTANT_IDENTIFIER 59
#define PROCEDURE_IDENTIFIER 60
#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
==> 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
==> token_structure.h <==
#include <bits/types.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,
                   __uint8_t type);

Token create_token_with_code(size_t row, size_t col, char *_data,
                             size_t dataSize, size_t code);
#endif
==> 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);

#endif
```