

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);
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\*)
13. SYNTAX:
14. <signal-program>
16. ERRORS:
17. **#1|Error(Syntax)| Line->3, Column->2 |: 'BEGIN' expected, but 'BEGUN' found.**

***Тест 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\*)
13. SYNTAX:
14. <signal-program>
16. ERRORS:
17. **#1|Error(Syntax)| Line->3, Column->2 |: 'BEGIN' expected, but 'ELEM3' found.**

***Тест 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\*)
13. SYNTAX:
14. <signal-program>
15. |<program>
16. ||PROGRAM
17. **||<procedure-identifier>**
18. |||<identifier>
19. ||||TEST02
20. ||;
21. ||<block>
22. **|||<declarations>**
23. ||||<constant-declarations>
24. |||||CONST
25. |||||<constant-declarations-list>
26. ||||||<constant-declaration>
27. **|||||||<constant-identifier>**
28. ||||||||<identifier>
29. |||||||||ELEM3
30. |||||||=
31. |||||||<constant>
32. **||||||||<unsigned-integer>**
33. |||||||||45
34. |||||||;
35. ||||||<empty>
36. |||||||<empty>
37. **|||BEGIN**
38. |||<statements-list>
39. ||||<statement>
40. |||||<variable-identifier>
41. ||||||<identifier>
42. **|||||||ELEM4**
43. |||||:=
44. |||||<expression>
45. ||||||<summand>
46. |||||||<unsigned-integer>
47. **||||||||30**
48. ||||||<summands-list>
49. |||||||<add-instruction>
50. ||||||||+
51. |||||||<summand>
52. **||||||||<unsigned-integer>**
53. |||||||||45
54. |||||;
55. ||||<statements-list>
56. |||||<statement>
57. **||||||CASE**
58. ||||||<expression>
59. |||||||<summand>
60. ||||||||<variable-identifier>
61. |||||||||<identifier>
62. **||||||||||ELEM3**
63. |||||||<summands-list>
64. ||||||||<add-instruction>
65. |||||||||-
66. ||||||||<summand>
67. **|||||||||<variable-identifier>**
68. ||||||||||<identifier>
69. |||||||||||ELEM4
70. ||||||OF
71. ||||||<alternatives-list>
72. **|||||||<alternative>**
73. ||||||||<expression>
74. |||||||||<summand>
75. ||||||||||<variable-identifier>
76. |||||||||||<identifier>
77. **||||||||||||ELEM2**
78. |||||||||<summands-list>
79. ||||||||||<add-instruction>
80. |||||||||||+
81. ||||||||||<summand>
82. **|||||||||||<variable-identifier>**
83. ||||||||||||<identifier>
84. |||||||||||||ELEM1
85. ||||||||:
86. ||||||||/
87. **||||||||<statements-list>**
88. |||||||||<statement>
89. ||||||||||<variable-identifier>
90. |||||||||||<identifier>
91. ||||||||||||ELEM5
92. **||||||||||:=**
93. ||||||||||<expression>
94. |||||||||||<summand>
95. ||||||||||||<unsigned-integer>
96. |||||||||||||20
97. **|||||||||||<summands-list>**
98. ||||||||||||<add-instruction>
99. |||||||||||||-
100. ||||||||||||<summand>
101. |||||||||||||<unsigned-integer>
102. **||||||||||||||10**
103. ||||||||||;
104. |||||||||<empty>
105. ||||||||||<empty>
106. ||||||||\
107. **|||||||<empty>**
108. ||||||||<empty>
109. ||||||ENDCASE
110. ||||||;
111. |||||<empty>
112. **||||||<empty>**
113. |||END
114. ||.

***Тест 4***

1. PROGRAM TEST02;
2. CONST
3. ELEM3 = 23;
4. BEGIN
5. **ELEM4 := 5 - ELEM3;**
6. END.
7. SYNTAX:
8. <signal-program>
9. |<program>
10. ||PROGRAM
11. **||<procedure-identifier>**
12. |||<identifier>
13. ||||TEST02
14. ||;
15. ||<block>
16. **|||<declarations>**
17. ||||<constant-declarations>
18. |||||CONST
19. |||||<constant-declarations-list>
20. ||||||<constant-declaration>
21. **|||||||<constant-identifier>**
22. ||||||||<identifier>
23. |||||||||ELEM3
24. |||||||=
25. |||||||<constant>
26. **||||||||<unsigned-integer>**
27. |||||||||23
28. |||||||;
29. ||||||<empty>
30. |||||||<empty>
31. **|||BEGIN**
32. |||<statements-list>
33. ||||<statement>
34. |||||<variable-identifier>
35. ||||||<identifier>
36. **|||||||ELEM4**
37. |||||:=
38. |||||<expression>
39. ||||||<summand>
40. |||||||<unsigned-integer>
41. **||||||||5**
42. ||||||<summands-list>
43. |||||||<add-instruction>
44. ||||||||-
45. |||||||<summand>
46. **||||||||<variable-identifier>**
47. |||||||||<identifier>
48. ||||||||||ELEM3
49. |||||;
50. ||||<empty>
51. **|||||<empty>**
52. |||END
53. ||.

***Тест 5***

1. PROGRAM TEST02;
2. CONST
3. SOME4 = 4;
4. BEGIN
5. **SOME4 := 5+SOME3;**
6. END.
7. SYNTAX:
8. <signal-program>
9. |<program>
10. ||PROGRAM
11. **||<procedure-identifier>**
12. |||<identifier>
13. ||||TEST02
14. ||;
15. ||<block>
16. **|||<declarations>**
17. ||||<constant-declarations>
18. |||||CONST
19. |||||<constant-declarations-list>
20. ||||||<constant-declaration>
21. **|||||||<constant-identifier>**
22. ||||||||<identifier>
23. |||||||||SOME4
24. |||||||=
25. |||||||<constant>
26. **||||||||<unsigned-integer>**
27. |||||||||4
28. |||||||;
29. ||||||<empty>
30. |||||||<empty>
31. **|||BEGIN**
32. |||<statements-list>
33. ||||<statement>
34. |||||<variable-identifier>
35. ||||||<identifier>
36. **|||||||SOME4**
37. |||||:=
38. |||||<expression>
39. ||||||<summand>
40. |||||||<unsigned-integer>
41. **||||||||5**
42. ||||||<summands-list>
43. |||||||<add-instruction>
44. ||||||||+
45. |||||||<summand>
46. **||||||||<variable-identifier>**
47. |||||||||<identifier>
48. ||||||||||SOME3
49. |||||;
50. ||||<empty>
51. **|||||<empty>**
52. |||END
53. ||.

***Тест 6***

1. PROGRAM TEST02;
2. CONST
3. SOME4 = 4;
4. BEGIN
5. **SOME4 := 5 SOME3;**
6. END.
7. SYNTAX:
8. <signal-program>
10. ERRORS:
11. **#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**