

1 Разработка грамматики модельного языка программирования

1.1 Форма Бэкуса-Наура

<программа> ::= { / (<описание> | <оператор>) ; / }

<описание> ::= dim <идентификатор> { , <идентификатор> } <тип>

<идентификатор> ::= <буква> <буква> <непустая последовательность цифр>

(в порядке следования: целый, действительный, логический)

<тип> ::= % | ! | \$

<оператор> ::= <составной> | <присваивания> | <условный> | <фиксированного_цикла> | <условного_цикла> | <ввода> | <вывода>

<составной> ::= « { » <оператор> { ; <оператор> } « } »

<присваивание> ::= <идентификатор> ass <выражение>

<условный> ::= if (<выражение>) « { » <оператор> « } » { elseif (<выражение>) « { » <оператор> « } » } { else « { » <оператор> « } » }

<фиксированного_цикла> ::= for <присваивания> to <выражение> « { » <оператор> “ } ”

<условного_цикла> ::= do while <выражение> « { » <оператор> « } »

<ввода> ::= read (<идентификатор> { , <идентификатор> })

<вывода> ::= output (<выражение> { пробел <выражение> })

<выражение> ::= <число> | <идентификатор> | not (<идентификатор> , <выражение> , <булево значение>) | - (<идентификатор> , <выражение> , <число>) | (<идентификатор> , <число>) <знак> (<идентификатор> , <число>)

<число> ::= <непустая последовательность цифр> | <непустая последовательность цифр> . <непустая последовательность цифр>

<булево значение> ::= true | false

<последовательность цифр> ::= { <цифра> }

<непустая последовательность цифр> ::= { / <цифра> / }

<знак> ::= + | - | * | / | > | < | <= | >= | = | and | or

$\langle \text{буква} \rangle ::= a | b | c | d | e | f | g | h | i | j | k | l | m | n | o | p | q | r | s | t | u | v | w | x | y | z$

$\langle \text{цифра} \rangle ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9$

$\langle \text{начало_комментария} \rangle ::= \langle /* \rangle$

$\langle \text{конец_комментария} \rangle ::= \langle */ \rangle$

1.2 Формальная грамматика

Грамматика представляет собой набор $\{T, N, P, S\}$, где T – множество терминальных символов.

$T = \{ ;, +, -, *, /, >, >=, <, <=, =, !=, , \langle, \rangle, (,), \{, \}, \text{if}, \text{else}, \text{elseif}, \text{for}, \text{to}, \text{do}, \text{while}, \text{ass}, \text{dim}, \text{read}, \text{output}, \text{and}, \text{or}, \text{not}, \text{ID}, \text{TYPE}, \text{BOOL}, \text{NUM} \}$

N – множество нетерминальных символов.

$N = \{ \text{PROG}, \text{CMD}, \text{IDS}, \text{I}, \text{EXPR}, \text{ASSIGN}, \text{FORCOND}, \text{COND1}, \text{COND}, \text{CALC}, \text{ANY} \}$

S – начальный символ из набора нетерминалов.

$S = \text{PROG}$

P – множество правил.

Правила формальной грамматики:

Программа

$\text{PROG} : \text{CMD}$

Команды

$\text{CMD} : \text{EXPR} ; \text{CMD} | \text{EXPR} ; | \text{ASSIGN} ; \text{CMD} | \text{ASSIGN} ;$

Список идентификаторов

$\text{IDS} : \text{IDS} , \text{ID} | \text{I} | \text{I} , \text{ID}$

$\text{I} : \text{ID}$

Выражения

EXPR : dim IDS TYPE | read (IDS) | output (IDS)
| if (COND) { CMD } | if (COND) { CMD } else { CMD
} | if (COND) { CMD } COND1 | do while (COND) { CMD
} | for (FORCOND) { CMD }

Присвоение

ASSIGN : ID ass CALC | ID ass ANY | ID ass I | ID
ass COND

Условие фиксированного цикла

FORCOND : ASSIGN to ANY | ASSIGN to I

Условия конструкций elseif, else

COND1 : elseif (COND) { CMD } else { CMD } |
elseif (COND) { CMD } COND1 | elseif (COND) { CMD }

Условие

COND : ANY = ANY | ANY != ANY | ANY < ANY | ANY <=
ANY | ANY > ANY | ANY >= ANY | ANY and ANY | ANY or ANY
| not ANY | not ID | ANY = ID | ANY != ID | ANY < ID |
ANY <= ID | ANY > ID | ANY >= ID | ANY and ID | ANY or
ID | ID = ANY | ID != ANY | ID < ANY | ID <= ANY | ID >
ANY | ID >= ANY | ID and ANY | ID or ANY | ID = ID | ID
!= ID | ID < ID | ID <= ID | ID > ID | ID >= ID | ID
and ID | ID or ID

Вычисление значения

CALC : ANY + ANY | ANY - ANY | ANY * ANY | ANY / ANY |
ANY + ID | ANY - ID | ANY * ID | ANY / ID | ID + ANY |

ID - ANY | ID * ANY | ID / ANY | ID + ID | ID - ID | ID
* ID | ID / ID

Константа числового или булевого типа

ANY : NUM | BOOL

2 Лексический анализ

2.1 Алгоритмы

Алгоритмы лексического анализа могут основываться на разделении на лексемы с помощью символов разделителей, чтением исходного кода посимвольно или с использованием регулярных выражений. Комментарии могут быть удалены либо же проигнорированы непосредственно при лексическом анализе.

В данном алгоритме используется посимвольное чтение исходного текста и последующая проверка символа или их комбинации на наличие соответствующей лексемы языка. При несоответствии выдаётся ошибка. Во время лексического анализа комментарии игнорируются.

2.2 Ошибки вывода

При лексическом анализе может быть выявлено 2 типа ошибок:

Первый тип – неверный идентификатор. Ошибка выдаётся если встречено слово, не являющееся ключевым словом, описанным в словаре WORDS класса `Lexer` и не соответствующее правилам наименования идентификаторов языка. В этом случае будет выведена ошибка `Lexer error: Unknown identifier "[имя идентификатора]" in line [номер строки]`.

Второй тип – непредвиденный символ. Ошибка выдаётся если встречен символ, не являющийся одним из символов-лексем языка, описанных в словаре SYMBOLS класса `Lexer`. В этом случае будет выведена ошибка `Lexer error: Unexpected symbol "[символ]" in line [номер строки]`.

2.3 Хэш-таблица

Для разбиения исходного кода на лексемы используются встроенные в язык Python хэш-таблицы, именуемые словарями. В множествах и словарях языка Python используется метод открытой адресации. Он заключается в том, что в ячейки таблицы помещаются не указатели на списки, а сами пары ключ-значение. Значение зашифровывается хэш-функцией и при возникновении коллизии пара ключ-значение записываются в следующую пустую ячейку после той, которая получилась в результате работы хэш-функции. При извлечении данных из хэш-таблицы данные дополнительно сверяются с ключом, который так же хранится в хэш-таблице.

Хеш-функцией выступает метод `__hash__` определённый в каждом хешируемом объекте языка Python. Эта функция может быть описана для любого типа данных, а для стандартных описана по стандарту. Для простых типов данных, например, `int` – результатом может быть само число, а для сложных может находиться комбинация хешей для составных частей и генератора случайных чисел, который гарантирует одинаковые значения для одного типа данных в рамках одного запуска программы.

2.4 Реализация лексического анализа

Лексический анализ реализует модуль `lexer` и, в частности, класс `Lexer`, содержащий хэш-таблицы символов и ключевых слов – `SYMBOLS` и `WORDS`. Входными данными для класса является текстовый поток `input_stream` из файла текста программы. Метод `getc()` получает следующий символ из потока `input_stream`. Метод `set_error()` выставляет сообщение об ошибке. Метод `next_token()` считывает следующую лексему посимвольно, определяет тип лексемы, игнорирует комментарии и вызывает метод `set_error()` при возникновении ошибки. В результате в поле `symbol` выводится тип лексемы и в поле `value` значение для целочисленных, дробных и булевых литералов или имя идентификатора.

2.5 Тестирование

Найти значение функции $y = kx + b$

Тест 1. Программа написана без ошибок

Исходный код программы на модульном языке программирования:

```
dim xx1, yy1, kk1, bb1, rr1 !;  
rr1 ass xx1 * kk1;  
rr1 ass rr1 + bb1;  
yy1 ass rr1;
```

Результат работы лексического анализа представлен на рисунке 1.



Рисунок 1 – Тест 1 алгоритма лексического анализа

Тест 1. Программа написана с лексической ошибкой, не существует оператора или идентификатора as.

Исходный код программы на модульном языке программирования:

```
dim xx1, yy1, kk1, bb1, rr1 !;  
rr1 as xx1 * kk1;  
rr1 ass rr1 + bb1;  
yy1 ass rr1;
```

Результат работы лексического анализа представлен на рисунке 2.

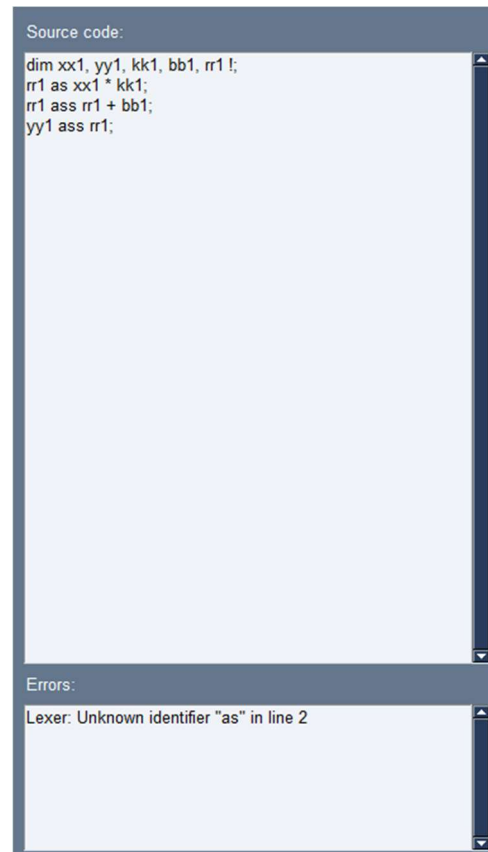


Рисунок 2 – Тест 2 алгоритма лексического анализа

3 Синтаксический анализ

3.1 Алгоритмы

Существует несколько алгоритмов синтаксического анализа:

Алгоритм рекурсивного спуска – алгоритм нисходящего синтаксического анализа, реализуемый путём взаимного вызова процедур парсинга, где каждая процедура соответствует одному из правил контекстно-свободной грамматики или БНФ. Применения правил последовательно, слева-направо поглощают токены, полученные от лексического анализатора. Это один из самых простых алгоритмов парсинга, который является также малоэффективным т.к. может обрабатывать большое количество текста недостаточно быстро и может обрабатывать не все грамматики.

Алгоритм сдвиг-свёртки используется в данном алгоритме. Он используется для грамматики операторного предшествования. Для моделирования его работы необходима входная цепочка символов и стек символов, в котором автомат может обращаться не только к самому верхнему символу, но и к некоторой цепочке символов на вершине стека. Также необходимо построить матрицу операторного предшествования.

Этот алгоритм для заданной КС-грамматики можно описать следующим образом:

1) Поместить в верхушку стека символ «начало строки», считывающую головку МП-автомата поместить в начало входной цепочки. В конец входной цепочки надо дописать символ «конец строки».

2) В стеке ищется самый верхний терминальный символ s_j при этом сам символ s_j остается в стеке. Из входной цепочки берется текущий символ a_i (справа от считывающей головки МП-автомата).

3) Если символ s_j – это символ начала строки, а символ a_i – символ конца строки, то алгоритм завершен, входная цепочка символов разобрана.

4) В матрице предшествования ищется клетка на пересечении строки, помеченной символом s_j , и столбца, помеченного символом a_i .

5) Если клетка, пустая, то значит, входная строка символов не принимается, алгоритм прерывается и выдает сообщение об ошибке.

6) Если клетка, содержит символ “=.” или “<.” то необходимо выполнить перенос. При выполнении переноса текущий входной символ a_i помещается на верхушку стека, считывающая головка сдвигается на одну позицию вправо. После этого надо вернуться к шагу 2.

7) Если клетка, содержит символ “.>”, то необходимо произвести свертку. Для выполнения свертки из стека выбираются все терминальные символы, связанные отношением “=.”, начиная от вершины стека, а также все нетерминальные символы, лежащие в стеке рядом с ними. Эти символы вынимаются из стека и собираются в цепочку.

8) Во всем множестве правил грамматики ищется правило, у которого правая часть совпадает с цепочкой. Если правило найдено, то в стек помещается нетерминальный символ из левой части правила, иначе, если правило не найдено, это значит, что входная строка символов не принимается, алгоритм прерывается и выдает сообщение об ошибке. После выполнения свертки необходимо вернуться к шагу 2.

В данном алгоритме матрица операторного предшествования строится автоматически из правил грамматики перед выполнением алгоритма сдвиг-свёртки.

3.2 Ошибки вывода

При построении матрицы операторного предшествования может быть выведены ошибки:

- 1) при встрече неизвестного символа - `unknown symbol "[символ]"`;
- 2) при обнаружении неверно построенного файла правил вывода - `wrong file formatting`;
- 3) при конфликте в правилах вывода с указанием ошибки;

При выполнении алгоритма сдвиг-свёртки:

- 1) при отсутствии правила в исходных правилах вывода - Unable to locate rule [правило];
- 2) при отсутствии связи в таблице операторного предшествования - unknown construction [символ 1] [символ 2] in [фрагмент кода].

3.3 Реализация синтаксического анализа

Для построения матрицы операторного предшествования в класс `Matrix` передаётся текстовый поток `input_stream` из текстового файла, в котором описаны все терминальные символы, все имена правил и перечислены все правила вывода причём имя и само правила разделяется символом «:». Все терминальные символы должны быть разделены пробелом. Метод `print_matrix()` позволяет вывести результат на экран а Метод `generate()` генерирует матрицу операторного предшествования. Строятся множества крайних левых и крайних правых символов для всех символов и только для терминальных символов. После по этим множествам составляется матрица операторного предшествования.

На рисунке 3 представлена полученная матрица операторного предшествования.

[illegible]

Рисунок 3 – Матрица операторного предшествования

Эта матрица передаётся для выполнения алгоритма сдвиг-свёртки в класс `Parser`. Перед выполнением сдвиг-свёртки все нетерминалы в правилах вывода заменяются на один нетерминал «Е». При вызове метода `check` выполняется проверка лексем на корректность по матрице операторного предшествования. Метод `get_t` возвращает крайний терминал стека, а при передаче аргумента `get_t(n)` может вернуть n-ый символ с конца стека. Для вывода ошибок используется метод `set_error`.

3.4 Тестирование

Проверить, что треугольник со сторонами a, b, c существует.

Тест 1. Программа написана без ошибок

Исходный код программы на модульном языке программирования:

```
dim aa1, bb1, cc1, ss1 !;
dim rr1 $;
rr1 ass true;
read(aa1, bb1, cc1);
ss1 ass aa1 + bb1;
if (ss1 < cc1) {
    rr1 ass false;
};
ss1 ass aa1 + cc1;
if (ss1 < bb1) {
    rr1 ass false;
};
ss1 ass bb1 + cc1;
if (ss1 < aa1) {
    rr1 ass false;
};
output(rr1);
```

Результат работы синтаксического анализа представлен на рисунке 4.

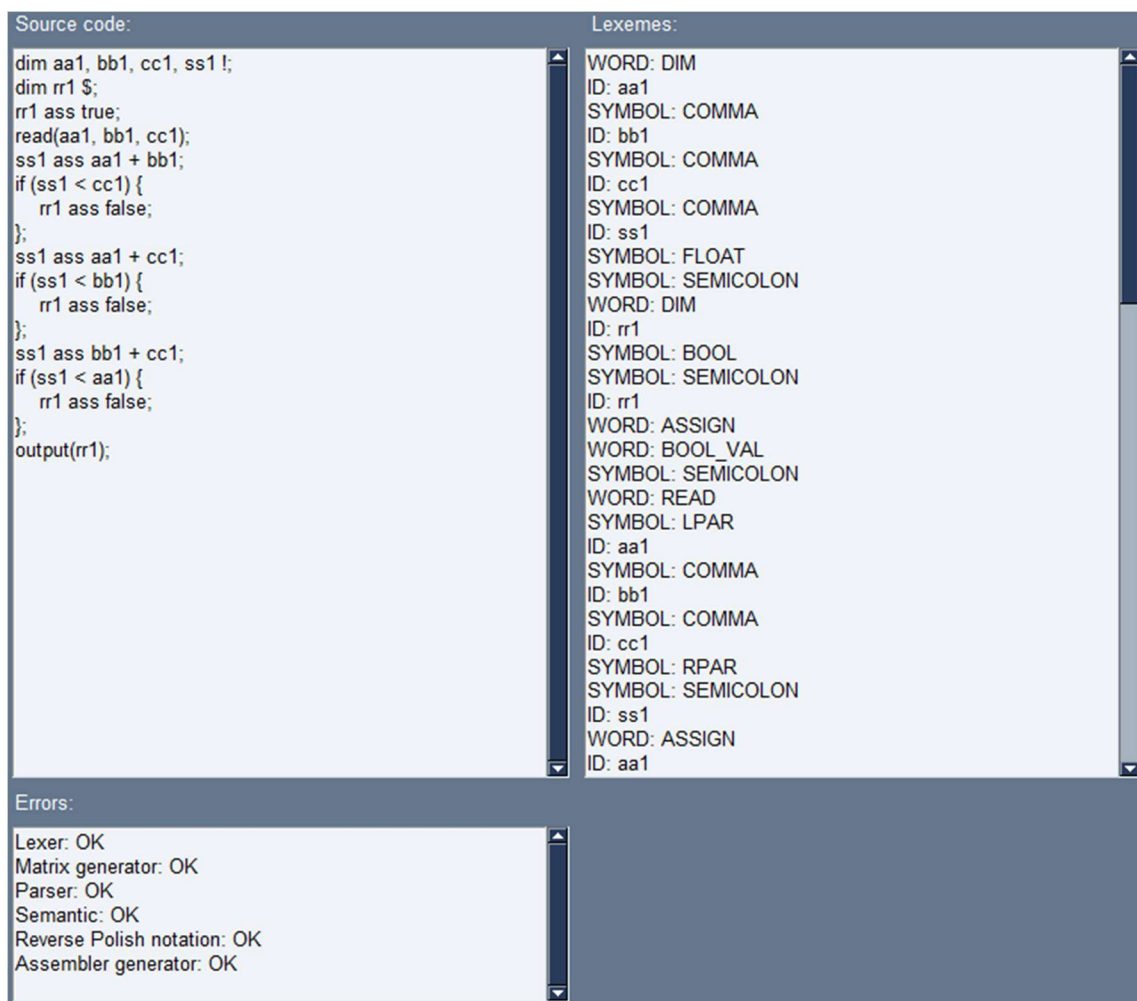


Рисунок 4 – Тест 1 алгоритма синтаксического анализа

Тест 2. Программа написана с синтаксической ошибкой, `elseif` не может идти после «;».

```
dim aa1, bb1, cc1, ss1 !;
dim rr1 $;
rr1 ass true;
ss1 ass aa1 + bb1;
elseif (ss1 < cc1) {
    rr1 ass false;
};
...
```

Результат работы синтаксического анализа представлен на рисунке 5.

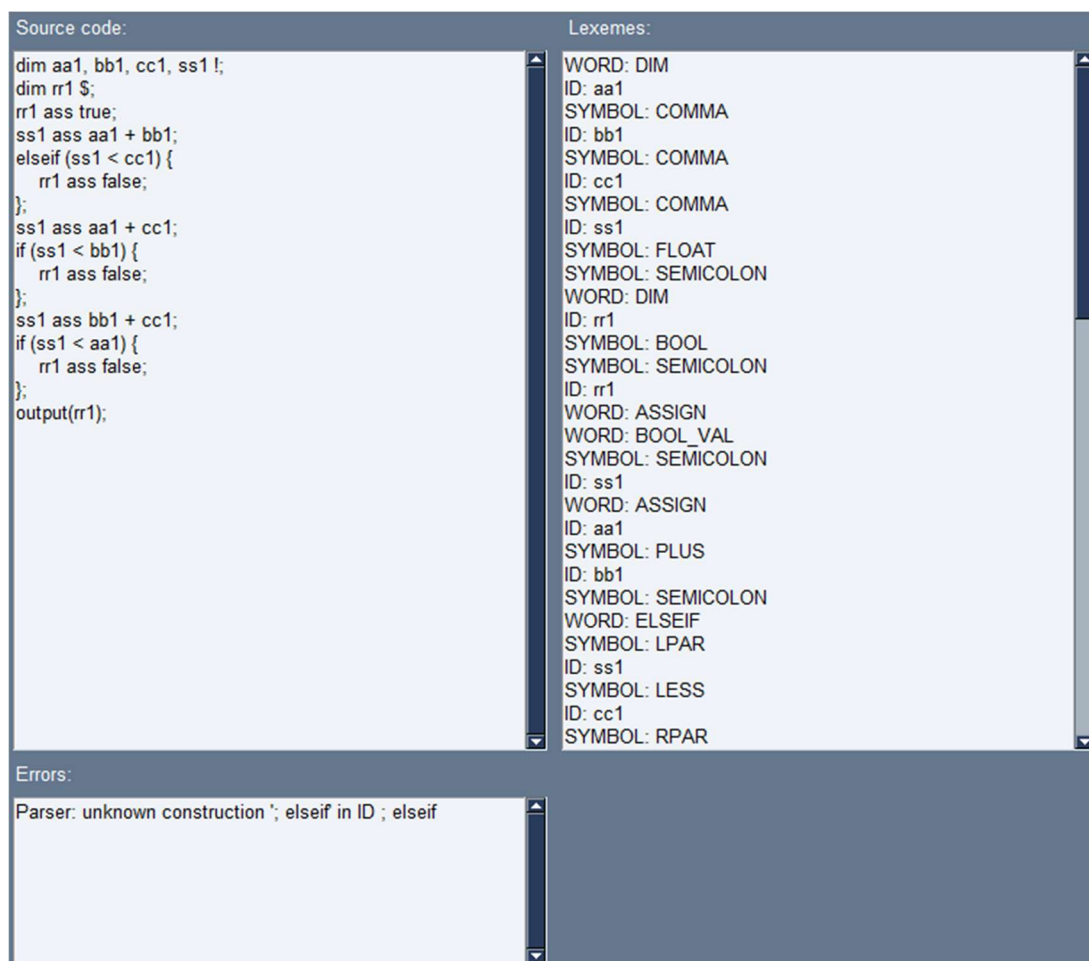


Рисунок 5 – Тест 2 алгоритма синтаксического анализа

Тест 3. Программа написана с синтаксической ошибкой, не существует правила грамматики для фрагмента кода `ss1 < cc1 = 1`

```
dim aa1, bb1, cc1, ss1 !;
dim rr1 $;
rr1 ass true;
ss1 ass aa1 + bb1;
if (ss1 < cc1 = 1) {
    rr1 ass false;
};
...
```

Результат работы синтаксического анализа представлен на рисунке 6.

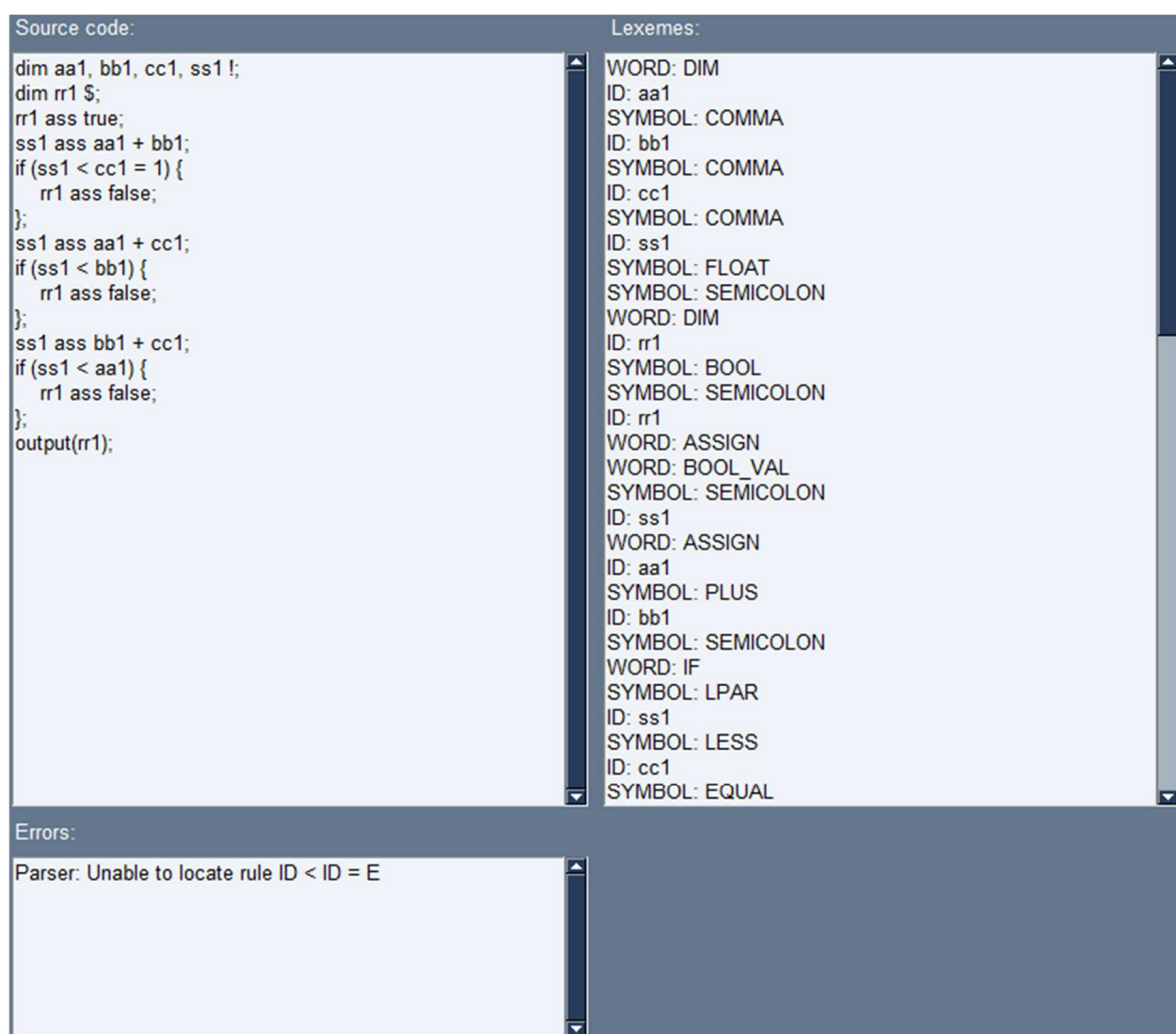


Рисунок 6 – Тест 3 алгоритма синтаксического анализа

4 Семантический анализ

4.1 Алгоритмы

Для семантического анализа используется простой алгоритм поиска в лексемах определённых закономерностей и проверка правильности типов и определения переменных в этих закономерностях.

4.2 Ошибки вывода

При попытке обратиться к переменной, которая не была определена до этого будет выведена ошибка `Undeclared variable` [имя переменной].

При попытке присвоить переменной литерал несоответствующего типа или значение переменной несоответствующего типа будет выведена ошибка `Wrong type: [имя переменной] is [тип переменной] unable to assign` [тип, который пытался присвоить пользователь].

4.3 Реализация семантического анализа

Метод `check` проверяет входную цепочку лексем на соответствие типов и ведёт учёт переменных, определённых в исходном коде. Считывание происходит поэлементно. Алгоритм ищет цепочку вида `dim` [имена переменных] [тип] для занесения переменной в список определённых, цепочки вида `[переменная] ass [переменная]`, `[переменная] ass [литерал]` или `[переменная] ass [переменная или литерал]` [операция] [переменная или литерал] для контроля соответствия типов. Любые другие вхождения переменных будут сопоставляться со списком определённых в программе переменных. В классе определены поля-массивы операций `RETURN_BOOL` и `RETURN_NUM`, которые определяют какой тип возвращают соответствующие операции. Для вывода ошибок используется метод `set_error`.

4.4 Тестирование

Рассчитать сумму чётных чисел от 2 до n.

Тест 1. Программа написана без ошибок

Исходный код программы на модульном языке программирования:

```
dim ii1, ss1, tm1, nn1 !;
read(nn1);
nn1 ass nn1/2;
nn1 ass nn1 + 1;
for (ii1 ass 1 to nn1) {
    tm1 ass ii1 * 2;
    ss1 ass ss1 + tm1;
};
```

Результат работы алгоритма семантического анализа представлен на рисунке 7.

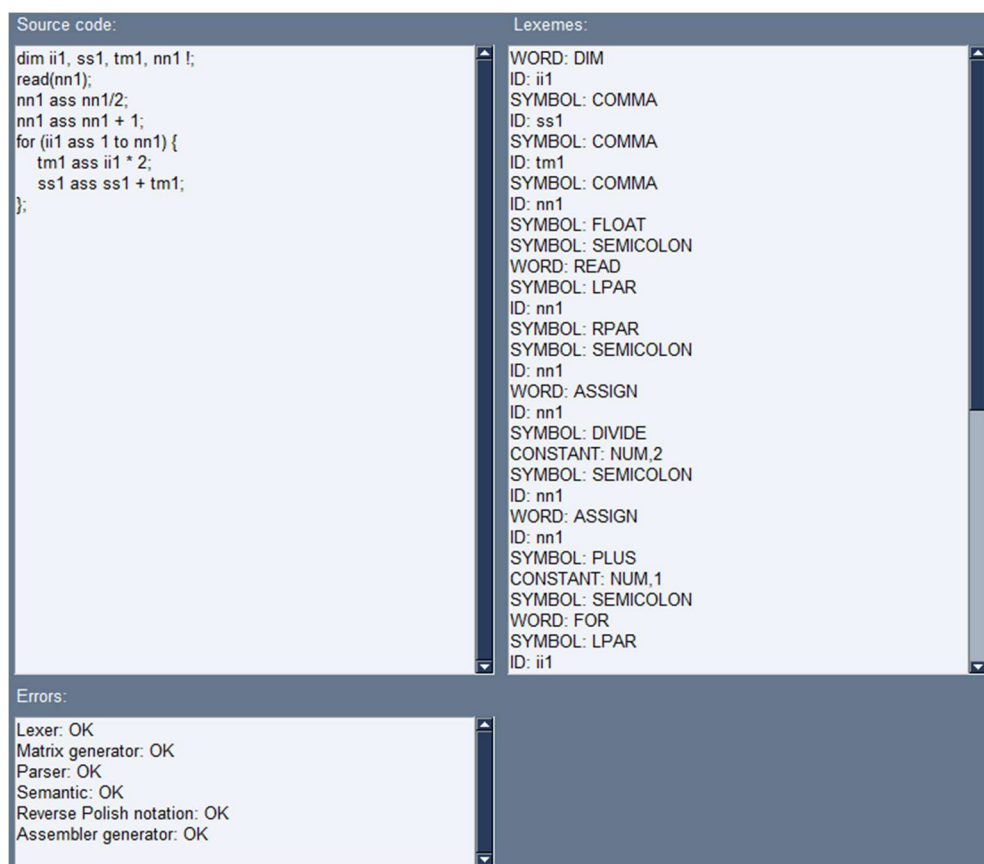


Рисунок 7 – Тест 1 алгоритма семантического анализа

Тест 1. Программа написана с семантической ошибкой, невозможно присвоить переменной типа FLOAT константу типа BOOL

Исходный код программы на модульном языке программирования:

```
dim ii1, ss1, tm1, nn1 !;
read(nn1);
nn1 ass nn1/2;
nn1 ass nn1 + 1;
for (ii1 ass 1 to nn1) {
    tm1 ass true;
    ss1 ass ss1 + tm1;
};
```

Результат работы алгоритма семантического анализа представлен на рисунке 8.

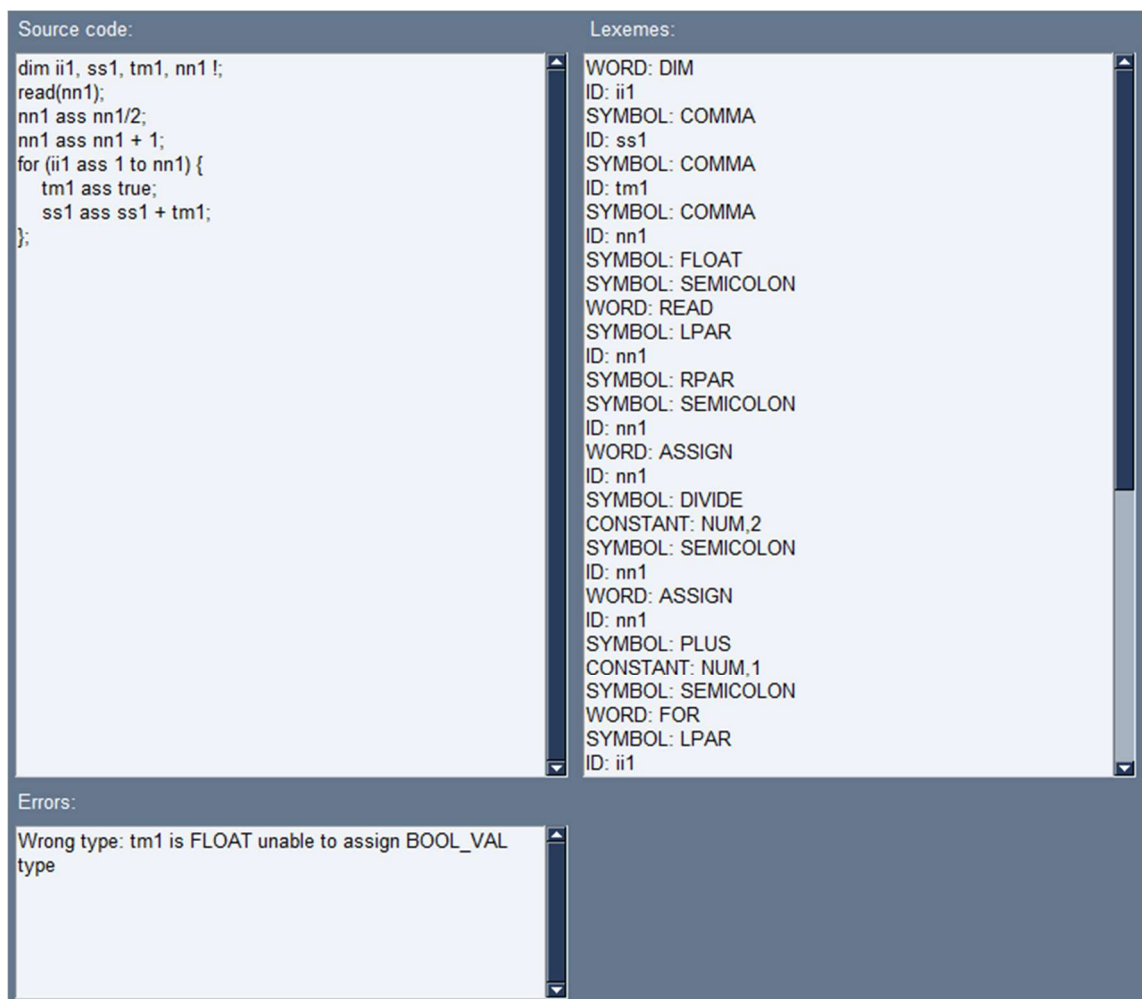


Рисунок 8 – Тест 2 алгоритма семантического анализа

5 Перевод в польскую инверсную запись

5.1 Алгоритм

Для перевода в польскую инверсную запись используется алгоритм Замельсона и Бауэра который заключается в предварительном составлении таблицы приоритетов для каждого оператора языка. В такой таблице хранятся стековый и магазинный приоритеты. Код читается слева направо и каждый элемент добавляется либо в результат, либо в магазин. Идентификаторы и константы переписываются в выходную строку ПОЛИЗа. При обнаружении разделителя его сравнительный приоритет P_C сравнивается с магазинным приоритетом P_M разделителя из вершины магазина операций. Если $P_C > P_M$, то разделитель входной строки помещается в магазин (разделитель из исходной строки поступает в магазин и в том случае, когда магазин пуст). Если $P_C \leq P_M$, то символ извлекается из магазина и записывается в выходную строку ПОЛИЗа.

5.2 Ошибки вывода

Если в результате работы алгоритма магазин не окажется пустым, то будет выведена ошибка `Reverse Polish notation: Stack is not empty.`

5.3 Реализация перевода в ПОЛИЗ

Метод `convert()` класса `RPN` реализует перевод заданной последовательности операторов и операндов в польскую инверсную запись. Метод возвращает два массива с элементами ПОЛИЗа. `declare_rpn` – отдельный ПОЛИЗ для декларирования переменных, `main_rpn` – ПОЛИЗ для основной части программы.

При выполнении перевода используется `if_stack` – стек меток необходимых для перемещения по конструкциям `if-elseif-else`, `end_stack` –

стек меток необходимых для прыжка на конец конструкций, заканчивающихся на закрывающую фигурную скобку, `cycle_stack` – стек позволяющий проще определять какой тип цикла в данный момент преобразуется. Это необходимо т.к. каждый цикл использует свой способ управления метками.

Все приоритеты операторов хранятся в ассоциативной таблице `PRIORITY`. Каждому оператору соответствует свой массив из двух элементов в котором хранится сравнительный и магазинный приоритет.

Для обозначения метки в ПОЛИЗе используется выражение `(имя_метки)`.

Для обозначения перехода по метке используется выражение `[имя_метки]`.

5.4 Тестирование

Рассчитать сумму чётных чисел от 1 до 10 с использованием конструкции `while`.

Тест 1. Программа выводит корректную польскую инверсную запись.

Исходный код программы на модульном языке программирования:

```
dim iil, ssl, tml !;
for (iil ass 1 to 6) {
    tml ass iil * 2;
    ssl ass ssl + tml;
};
```

Результат работы алгоритма преобразования в ПОЛИЗ представлен на рисунке 9.

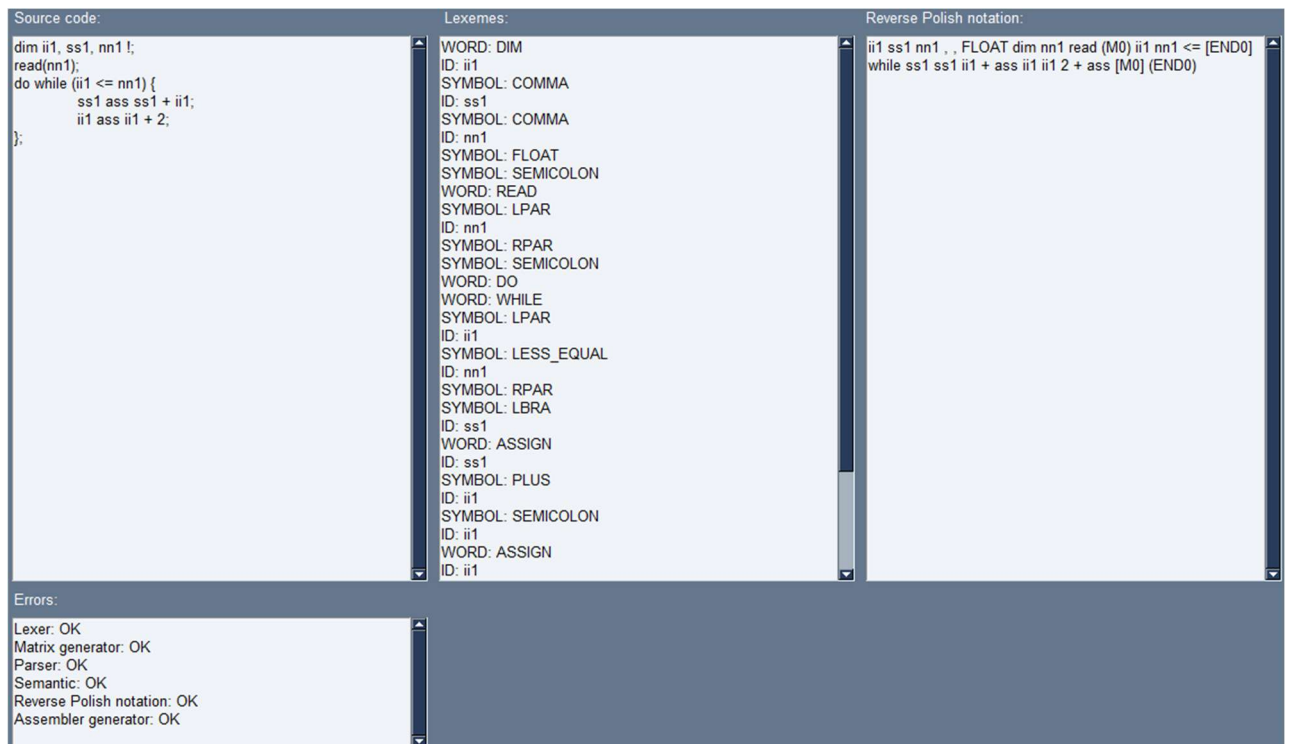


Рисунок 9 – Тест 1 алгоритма преобразования в ПОЛИЗ

6 Генерация ассемблерного кода из ПОЛИЗа

6.1 Алгоритм

Для перевода в ассемблер NASM используется алгоритм, считывающий поэлементно ПОЛИЗ и генерирующий ассемблерный код в соответствии со встретившимся элементом.

6.2 Реализация перевода в ассемблерный код языка NASM

Метод `generate()` класса `ASM` реализует перевод заданной польской инверсной записи в ассемблерный язык NASM x86. Он принимает на вход 2 части ПОЛИЗа – `declare_rpn` и `main_rpn`. Первая часть используется для декларирования переменных в сегменте данных, а вторая для реализации алгоритма программы в сегменте кода. Все числа представлены в формате `single IEEE-754`. Ввод и вывод происходит в том же формате с использованием шестнадцатеричного кодирования. Для конвертации чисел из десятичных чисел модульного языка программирования в нужный формат ассемблерного языка используется библиотека `ieee754` языка `Python`. Ассемблерный код полученный в результате данного алгоритма можно перевести в объектный файл с помощью программы `NASM` и скомпилировать данный файл в исполняемый посредством компилятора `GCC` или же запустить и удостоверится в его работе с помощью программы `SASM`. Создание исполняемого файла возможно нажатием кнопки «Build EXE» графического интерфейса программы.

6.3 Тестирование

Тест 1. Вычислить n -ое число Фибоначчи.

Исходный код программы на модульном языке программирования

```
dim aa1, aa2, ss1, nn1, ii1 !;  
aa1 ass 1;  
aa2 ass 1;
```

```

nn1 ass 10;
ii1 ass 3;
do while (ii1 <= nn1) {
    ss1 ass aa1 + aa2;
    aa1 ass aa2;
    aa2 ass ss1;
    ii1 ass ii1 + 1;
};
output(ss1);

```

Результат работы алгоритма генерации ассемблерного кода представлен на рисунке 10.

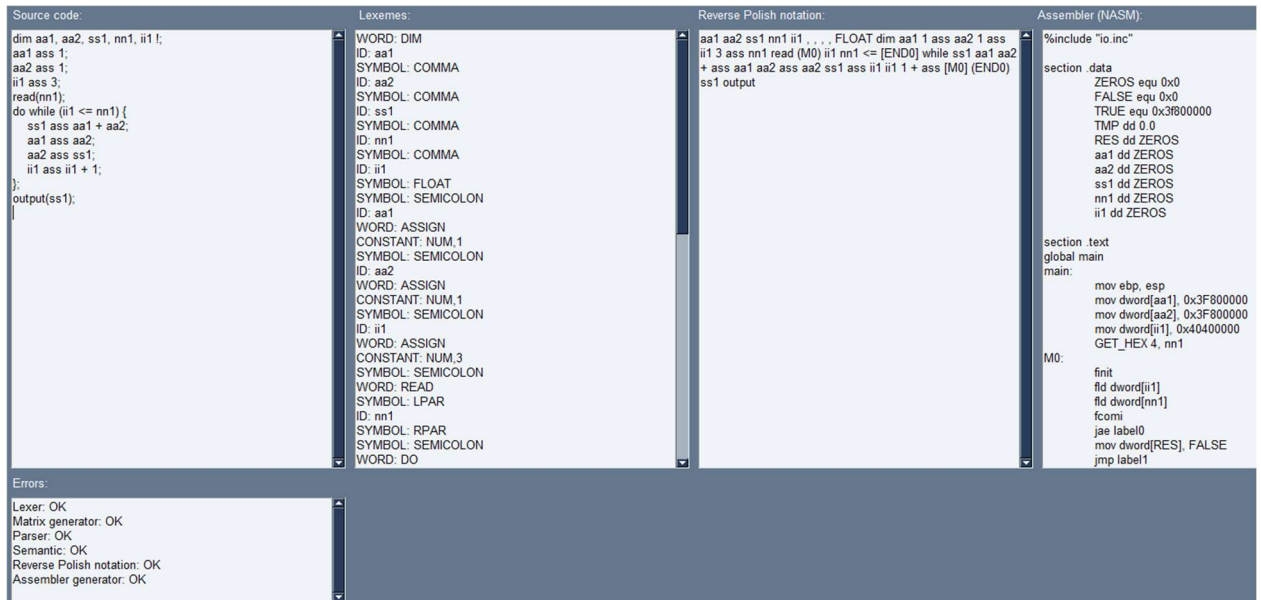


Рисунок 10 – Тест 1 алгоритма генерации ассемблерного кода

Полученный код на языке ассемблера:

```

%include "io.inc"

section .data

    ZEROS equ 0x0
    FALSE equ 0x0
    TRUE equ 0x3f800000

```

```

    TMP dd 0.0
    RES dd ZEROS
    aa1 dd ZEROS
    aa2 dd ZEROS
    ss1 dd ZEROS
    nn1 dd ZEROS
    ii1 dd ZEROS

section .text
global main
main:
    mov ebp, esp
    mov dword[aa1], 0x3F800000
    mov dword[aa2], 0x3F800000
    mov dword[ii1], 0x40400000
    GET_HEX 4, nn1
M0:
    finit
    fld dword[ii1]
    fld dword[nn1]
    fcomi
    jae label0
    mov dword[RES], FALSE
    jmp label1
label0:
    mov dword[RES], TRUE
label1:
    finit
    fld dword[RES]
    mov dword[TMP], 0x00000000

```



```

    fld dword[TMP]
    fcomi
    jz END0
    finit
    fld dword[aa1]
    fld dword[aa2]
    fadd
    fstp dword[RES]
    mov eax, dword[RES]
    mov dword[ss1], eax
    mov eax, dword[aa2]
    mov dword[aa1], eax
    mov eax, dword[ss1]
    mov dword[aa2], eax
    finit
    fld dword[ii1]
    mov dword[TMP], 0x3F800000
    fld dword[TMP]
    fadd
    fstp dword[RES]
    mov eax, dword[RES]
    mov dword[ii1], eax
    jmp M0
END0:
    PRINT_HEX 4, ss1
NEWLINE
    Ret

```

При выполнении ассемблерного кода в среде SASM можно удостовериться, что ассемблирование прошло успешно. В ячейке памяти с

именем `ss1` хранится результат вычисления n -ого числа Фибоначчи. Значение n вводится пользователем а значение n -ого числа Фибоначчи выводится на экран. Входные данные `0x41200000` соответствует числу 10, а `0x425C0000` соответствует 55 в кодировке IEEE-754.

Результат работы ассемблерного кода на языке NASM представлен на рисунке 11.

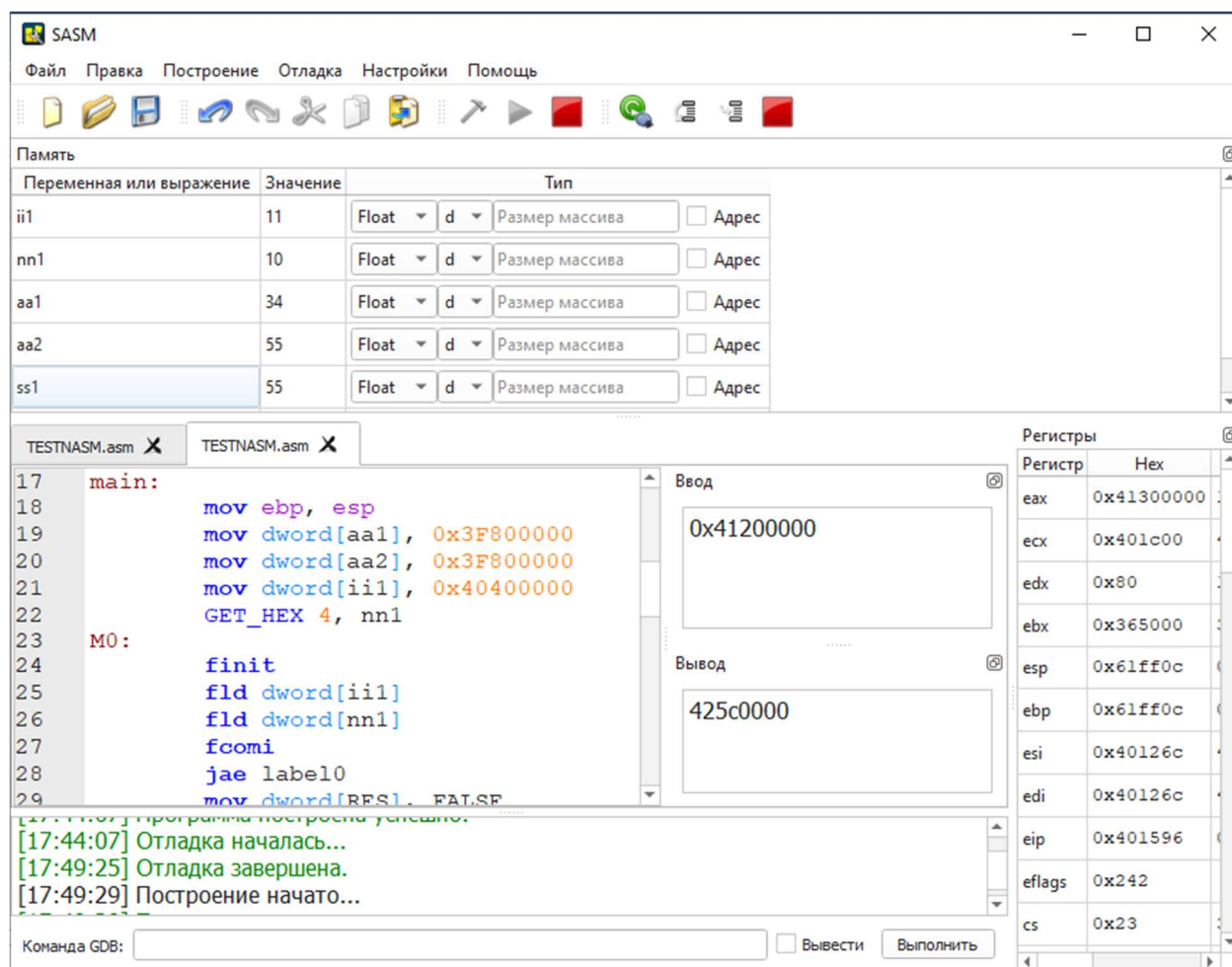


Рисунок 11 – Тест 1 выполнения ассемблерного кода в среде SASM

Тест 2. Выяснить взаимное расположение прямых и найти точку пересечения если она есть.

Исходный код программы на модульном языке программирования

```

dim kk1, kk2, bb1, bb2, xx1, yy1, tm1 !;
dim sm1, pr1, eq1, eq2 $;
read(kk1, bb1, kk2, bb2);
      
```

```

eq1 ass kk1 = kk2;
eq2 ass bb1 = bb2;
if (eq1 and eq2) {
    sm1 ass true;
} elseif (eq1 = true) {
    pr1 ass true;
} else {
    xx1 ass bb2 - bb1;
    tm1 ass kk1 - kk2;
    xx1 ass xx1 / tm1;
    yy1 ass kk1 * xx1;
    yy1 ass yy1 + bb1;
};

output (xx1, yy1);

```

Результат работы алгоритма генерации ассемблерного кода представлен на рисунке 12.

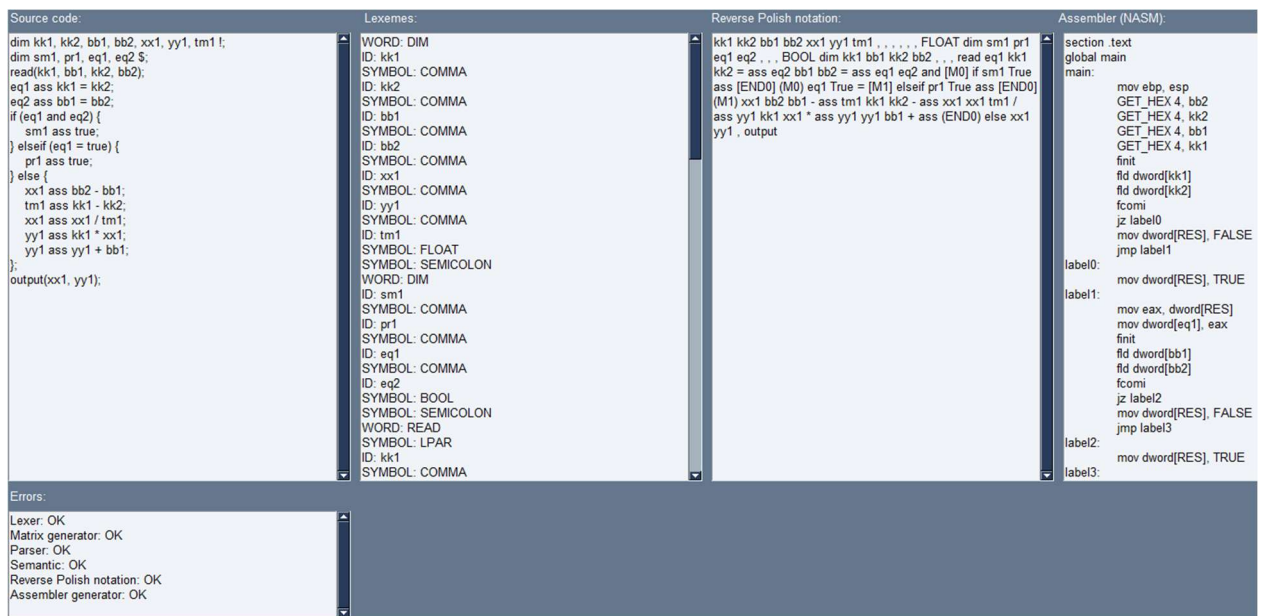


Рисунок 12 – Тест 2 алгоритма генерации ассемблерного кода

Полученный код на языке ассемблера:

```
%include "io.inc"
```

```

section .data
    ZEROS equ 0x0
    FALSE equ 0x0
    TRUE equ 0x3f800000
    TMP dd 0.0
    RES dd ZEROS
    kk1 dd ZEROS
    kk2 dd ZEROS
    bb1 dd ZEROS
    bb2 dd ZEROS
    xx1 dd ZEROS
    yy1 dd ZEROS
    tm1 dd ZEROS
    sm1 dd ZEROS
    pr1 dd ZEROS
    eq1 dd ZEROS
    eq2 dd ZEROS

```

```

section .text
global main
main:
    mov ebp, esp
    GET_HEX 4, bb2
    GET_HEX 4, kk2
    GET_HEX 4, bb1
    GET_HEX 4, kk1
    finit
    fld dword[kk1]
    fld dword[kk2]

```

```

        fcomi
        jz label0
        mov dword[RES], FALSE
        jmp label1
label0:
        mov dword[RES], TRUE
label1:
        mov eax, dword[RES]
        mov dword[eq1], eax
        finit
        fld dword[bb1]
        fld dword[bb2]
        fcomi
        jz label2
        mov dword[RES], FALSE
        jmp label3
label2:
        mov dword[RES], TRUE
label3:
        mov eax, dword[RES]
        mov dword[eq2], eax
        mov eax, dword[eq1]
        and eax, dword[eq2]
        mov dword[RES], eax
        finit
        fld dword[RES]
        mov dword[TMP], 0x00000000
        fld dword[TMP]
        fcomi
        jnz label4

```

```

        jmp M0
label4:
        mov dword[sm1], TRUE
        jmp END0
M0:
        finit
        fld dword[eq1]
        mov dword[TMP], TRUE
        fld dword[TMP]
        fcomi
        jz label5
        mov dword[RES], FALSE
        jmp label6
label5:
        mov dword[RES], TRUE
label6:
        finit
        fld dword[RES]
        mov dword[TMP], 0x00000000
        fld dword[TMP]
        fcomi
        jnz label7
        jmp M1
label7:
        mov dword[pr1], TRUE
        jmp END0
M1:
        finit
        fld dword[bb2]
        fld dword[bb1]

```

```
fsub
fstp dword[RES]
mov eax, dword[RES]
mov dword[xx1], eax
finit
fld dword[kk1]
fld dword[kk2]
fsub
fstp dword[RES]
mov eax, dword[RES]
mov dword[tm1], eax
finit
fld dword[xx1]
fld dword[tm1]
fdiv
fstp dword[RES]
mov eax, dword[RES]
mov dword[xx1], eax
finit
fld dword[kk1]
fld dword[xx1]
fmul
fstp dword[RES]
mov eax, dword[RES]
mov dword[yy1], eax
finit
fld dword[yy1]
fld dword[bb1]
fadd
fstp dword[RES]
```

```

        mov eax, dword[RES]
        mov dword[yy1], eax
END0:

        PRINT_HEX 4, yy1
NEWLINE
        PRINT_HEX 4, xx1
NEWLINE
        ret

```

Результат работы ассемблерного кода на языке NASM для входных данных $k1 = 2$, $b1 = -1$, $k2 = -3$, $b2 = 1$ представлен на рисунках 13-14. $0x3ECCCCCD$ и $0xBE4CCCCC$ в кодировке IEEE-754 соответственно равны $x = 0.4$ и $y = -0.2$.

xx1	0.400000006	Float ▾	d ▾	Размер массива	<input type="checkbox"/> Адрес
yy1	-0.199999988	Float ▾	d ▾	Размер массива	<input type="checkbox"/> Адрес
kk1	2	Float ▾	d ▾	Размер массива	<input type="checkbox"/> Адрес
bb1	-1	Float ▾	d ▾	Размер массива	<input type="checkbox"/> Адрес
kk2	-3	Float ▾	d ▾	Размер массива	<input type="checkbox"/> Адрес
bb2	1	Float ▾	d ▾	Размер массива	<input type="checkbox"/> Адрес

Рисунок 13 – Тест 2 выполнения ассемблерного кода в среде SASM

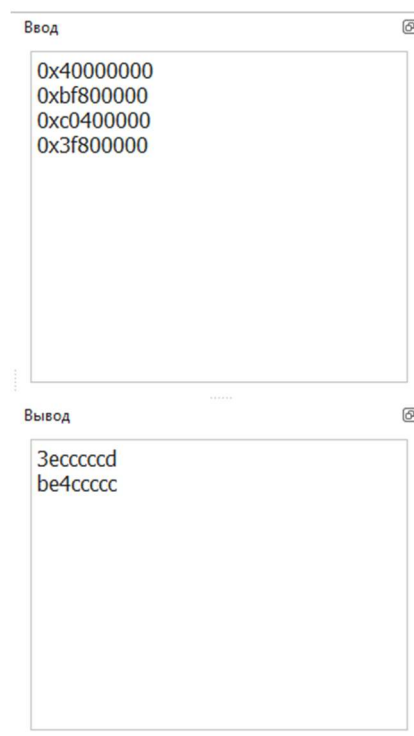


Рисунок 14 – Тест 2 выполнения ассемблерного кода в среде SASM

Тест 3.

Результат работы ассемблерного кода из теста 2 на языке NASM для входных данных $k1 = 2$, $b1 = -1$, $k2 = 2$, $b2 = 1$ представлен на рисунке 15. По результату $pr1 = 1$ видно, что две прямые параллельны.

Память			
Переменная или выражение	Значение	Тип	
xx1	0	Float ▾ d ▾	Размер массива <input type="checkbox"/> Адрес
yy1	0	Float ▾ d ▾	Размер массива <input type="checkbox"/> Адрес
kk1	2	Float ▾ d ▾	Размер массива <input type="checkbox"/> Адрес
bb1	-1	Float ▾ d ▾	Размер массива <input type="checkbox"/> Адрес
kk2	2	Float ▾ d ▾	Размер массива <input type="checkbox"/> Адрес
bb2	1	Float ▾ d ▾	Размер массива <input type="checkbox"/> Адрес
sm1	0	Float ▾ d ▾	Размер массива <input type="checkbox"/> Адрес
pr1	1	Float ▾ d ▾	Размер массива <input type="checkbox"/> Адрес

Рисунок 15 – Тест 3 выполнения ассемблерного кода в среде SASM

Тест 4.

Результат работы ассемблерного кода из теста 2 на языке NASM для входных данных $k1 = 2$, $b1 = 1$, $k2 = 2$, $b2 = 1$ представлен на рисунке 16. По результату $sm1 = 1$ видно, что прямые являются одинаковыми и у них бесконечное количество общих точек.

Переменная или выражение	Значение	Тип			
xx1	0	Float	d	Размер массива	<input type="checkbox"/> Адрес
yy1	0	Float	d	Размер массива	<input type="checkbox"/> Адрес
kk1	2	Float	d	Размер массива	<input type="checkbox"/> Адрес
bb1	1	Float	d	Размер массива	<input type="checkbox"/> Адрес
kk2	2	Float	d	Размер массива	<input type="checkbox"/> Адрес
bb2	1	Float	d	Размер массива	<input type="checkbox"/> Адрес
pr1	0	Float	d	Размер массива	<input type="checkbox"/> Адрес
sm1	1	Float	d	Размер массива	<input type="checkbox"/> Адрес
Добавить...		Smart	d	Размер массива	<input type="checkbox"/> Адрес

Рисунок 16 – Тест 4 выполнения ассемблерного кода в среде SASM

Тест 5.

Ассемблерный код на языке NASM и входные данные взяты из теста 2. Созданные объектный и исполняемый файлы с помощью функции «Build EXE» и демонстрация запуска исполняемого файла представлены на рисунках 17-18.




	asm.asm	24.12.2023 18:42	Assembler source ...	2 КБ
	asm.exe	24.12.2023 18:42	Приложение	52 КБ
	asm.obj	24.12.2023 18:42	3D Object	4 КБ

Рисунок 17 – Тест 5 файлы, полученные при построении исполняемого файла

```
C:\Users\Max\Documents\GitHub\Compiler>asm.exe
0x40000000
0xbf800000
0xc0400000
0x3f800000
3eccccd
be4cccc
```

Рисунок 18 – Тест 5 демонстрация работы исполняемого файла