МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«КУРСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»

Факультет физики, математики, информатики
Кафедра алгебры, геометрии и теории обучения математике

КУРСОВАЯ РАБОТА
по дисциплине
Теория формальных языков и трансляций
на тему: РАЗРАБОТКА КОМПИЛЯТОРА МОДЕЛЬНОГО ЯЗЫКА ПРОГРАММИРОВАНИЯ
Вариант 1

Обучающегося 3 курса
очной формы обучения
направления подготовки
02.03.03 Математическое обеспечение
и администрирование
информационных систем
Направленность (профиль)
Проектирование информационных
систем и баз данных
Андреенко Александра Денисовича
(фамилия, имя, отчество)

Руководитель: к.п.н., доцент
Селиванова Ирина Васильевна
(ученая степень, должность, фамилия,
имя, отчество)

Допустить к защите:
_____ / _____
«_____» _____ 20_____г.

Курск, 2022

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Компилятор – это программа, которая преобразует программный код, написанный на каком-либо языке программирования, в код машинный. Преобразование программного кода в машинный называется компиляцией.

В силу особенностей структуры процессоров, машинный код не универсален, а, следовательно, для компиляции одного и того же кода под разные процессоры требуются разные компиляторы (или универсальный компилятор, способный компилировать код под различные процессоры). То же самое касается и взаимодействия компиляторов с операционными системами: одна и та же программа может работать на одной ОС, но не запустится на других.

На данный момент самыми известными являются такие языки, как C++, Java, C# и Python. Все они используют один и тот же компилятор — GCC (gcc), за исключением языка C++, который, в свою очередь, использует CLAN (Common Language Infrastructure for Objective-C).

Для решения большинства задач встроенных средств уже существующих языков программирования будет достаточно, но решение более специфичных проблем может потребовать разработки собственного языка программирования, и, как следствие, компилятора для него. Кроме того, опыт разработки собственного компилятора позволит на более глубоком уровне понять устройство уже существующих компиляторов. И, наконец, если у разработчиков есть собственное представление о том, какой компилятор им необходим, то процесс создания такого компилятора будет упрощен. Этим определяется актуальность темы иссследования.

Объект исследования – языки программирования.

Предмет исследования – алгоритмы построения компиляторов языков программирования.

Целью данной курсовой работы является изучение различных аспектов функционирования компиляторов и особенностей их разработки.

Из цели исследования вытекают следующие задачи:

1) изучить необходимую литературу по теории создания компилятора;
2) разработать грамматику модельного языка программирования по варианту задания;
3) описать алгоритм лексического анализа и реализовать его программно;
4) описать алгоритм синтаксического анализа реализовать его программно;
5) описать алгоритм семантического анализа реализовать его программно;
6) описать алгоритм трансляции в ассемблер.

Работа выполняется по варианту модельного языка №1.

<программа> ::= program var <описание> begin <оператор>; {<оператор>;} end.

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

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

<тип> ::= int | float | bool

<составной оператор> ::= ”{“ {<оператор>} “}”

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

<условный оператор> ::= if <выражение> then <оператор> [else <оператор>]

<оператор фиксированного цикла> ::= for <присваивание> to <выражение> do <оператор>

<оператор условного цикла> ::= while <выражение> do <оператор

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

<оператор вывода> ::= write (<выражение>{, <выражение>})

<признак начала комментария> ::= /*

<признак конца комментария> ::= */

1 Разработка грамматики модельного языка программирования

1.1 Форма Бэкуса-Наура

Форма Бэкуса-Наура (БНФ) представляет собой формальную систему описания синтаксиса, в которой одни синтаксические категории последовательно определяются через другие.

БНФ, разработанная на основании варианта задания имеет следующий вид:

<программа> ::= program [<секция переменных>] <секция кода>

<секция переменных> ::= {var <описание>}

<секция кода> ::= begin <оператор>; {<оператор>;} end

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

<идентификатор> ::= <буква>[{<цифра> | <буква>}]

<тип> ::= int | bool

<оператор> ::= <составной оператор> | <присваивание> ; | <условный <оператор> | <оператор фиксированного цикла> | <оператор условного цикла> | <оператор ввода> ; | <оператор вывода> ;

<составной оператор> ::= ”{“ {<оператор>} “}”

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

<условный оператор> ::= if <выражение> then <оператор> [else <оператор>]

<оператор фиксированного цикла> ::= for <присваивание> to <выражение> do <оператор>

<оператор условного цикла> ::= while <булево выражение> do <оператор>

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

<оператор вывода> ::= write(<выражение>{, <выражение>})

<выражение> ::= <выражение> + <выражение>

| <выражение> - <выражение>

| <выражение> * <выражение>

| <выражение> / <выражение>

| <выражение> ^ <выражение>

| <выражение> or <выражение>

| <выражение> and <выражение>

| <выражение> > <выражение>

| <выражение> < <выражение>

| <выражение> >= <выражение>

| <выражение> <= <выражение>

| <выражение> != <выражение>

| “(“ <выражение> “)”

| not “(“ <выражение> “)”

| <булева константа>

| <число>

| <идентификатор>

<признак начала комментария> ::= //

<булева константа> ::= true | false

<число> ::= 0 | [-] <цифра без нуля> {<цифра>}

<цифра без нуля> ::= 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<цифра> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<буква> ::= 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 | 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

1.2 Формальная грамматика

Формальная грамматика G для данного варианта задания имеет вид: G = <T, N, S, P>, где T – алфавит терминальных символов, N – алфавит нетерминальных символов, S – начальный символ, P – набор правил вывода.

Число в квадратных скобках рядом с некоторыми нетерминальными символами в множестве P обозначает степень приоритета данного правила.

Чем выше это число, тем выше приоритет соответствующего правила относительно остальных.

T = {program, var, begin, end, int, bool, if, then, else, for, to, do, while, read, write, not, or, and , +, -, *, /, ^, <, >, ==, >=, <=, ;, {, }, :, a-z, A-Z, 0-9}

N = {

start = <программа>,

var_section = <секция переменных>,

code_section = <секция кода>,

declaration = <описание>,

ID = <идентификатор>,

type = <тип>,

operator = <оператор>,

block_op = <составной оператор>,

assign_op = <присваивание>,

if_op = <условный оператор>,

for_op = <оператор фиксированного цикла>,

while_op = <оператор условного цикла>,

read_op = <оператор ввода>,

write_op = <оператор вывода>,

expression = <выражение>

}

S = start

P = {

start -> PROGRAM program_repetition code_section

var_section -> VAR var_section_repetition_plus

code_section -> BEGIN code_section_repetition_plus END

declaration -> id_list : type ;

type -> INT

type -> BOOL

operator -> assign_op

operator -> if_op

operator -> for_op

operator -> while_op

operator -> read_op

operator -> write_op

operator -> block_op

assign_op -> ID = expression ;

if_op -> IF (expression) THEN operator

if_op -> IF (expression) THEN operator ELSE operator

for_op -> FOR assign_op TO expression DO operator

while_op -> WHILE expression DO operator

read_op -> READ (id_list) ;

write_op -> WRITE (expression_list) ;

block_op -> { block_op_repetition }

expression -> expression + expression

expression -> expression - expression

expression -> expression * expression

expression -> expression / expression

expression -> expression ^ expression

expression -> expression OR expression

expression -> expression AND expression

expression -> expression > expression

expression -> expression < expression

expression -> expression == expression

expression -> expression >= expression

expression -> expression <= expression

expression -> expression != expression

expression -> (expression)

expression -> NOT (expression)

expression -> BOOL_CONST

expression -> INT_CONST

expression -> ID

id_list -> id_list , ID

id_list -> ID

expression_list -> expression_list , expression

expression_list -> expression

program_repetition -> λ

program_repetition -> program_repetition var_section

var_section_repetition_plus -> declaration

var_section_repetition_plus -> var_section_repetition_plus declaration

code_section_repetition_plus -> operator

code_section_repetition_plus -> code_section_repetition_plus operator

block_op_repetition -> λ

block_op_repetition -> block_op_repetition operator

}

2 Проектирование и разработка компилятора

2.1 Лексический анализатор

Первым этапом разработки компилятора является лексический анализ.

Лексический анализ — процесс аналитического разбора входной последовательности символов на распознанные группы (лексемы) с целью получения на выходе идентифицированных последовательностей, называемых «токенами».

Для реализации лексического анализа был использован генератор лексических анализаторов, идентичный популярному генератору Lex.

Алгоритм работы анализатора, сгенерированного подобной программой заключается в последовательном рассмотрении символов из входной последовательности и их сопоставлением с заранее определенным набором регулярных выражений. Если анализатор нашел совпадение, то он возвращает токен соответствующего типа. В противном случае – возвращает токен, обозначающий, что данная последовательность символов не подходит ни под одно правило и продолжает анализ.

Анализатор, созданный в ходе выполнения данной работы, выделяет целые числа, идентификаторы, операции, булевы константы, а также все ключевые слова в соответствии с вариантом модельного языка.

Лексический анализ осуществляется следующим кодом:

```
const lexer = this.parser.lexer;
lexer.setInput(code);
while (!lexer.done) {
    let token = lexer.lex();
    if (token in parser.terminals_) {
      token = parser.terminals_[token];
    }

    let lexeme = `${token}`
    for (let i = 0; i < 13 - token.length; i++) {
      lexeme += ' ';
    }
    lexeme += `"${lexer.yytext}"\n`;
    this.lexemes += lexeme;}
```

На вход поступает исходный текст программы. На выходе получаем последовательность лексем.

Регулярные выражения, согласно которым происходит разделение на токены представлены в таблице 1.

Таблица 1 – Регулярные выражения и соответствующие им токены

Регулярное выражение	Возвращаемый токен
\s+	
“//”.*	
“program”	PROGRAM
“var”	VAR
“begin”	BEGIN
“end”	END
“bool”	BOOL
“if”	IF
“then”	THEN
“else”	ELSE
“for”	FOR
“to”	TO
“do”	DO
“while”	WHILE
“read”	READ
“write”	WRITE
“not”	NOT
“or”	OR
“and”	AND
“true” | “false”	BOOL_CONST
([-]?[1-9]\d*|0)\b	INT_CONST
[a-zA-Z][0-9a-zA-Z]*\b	ID
“==”	==

Таблица 1 – Продолжение

“>=”	>=
“<=”	<=
“!=”	!=
“=”	=
“*”	*
“/”	/
“-”	-
“+”	+
“^”	^
“>”	>
“<”	<
“(”	(
“)”	)
“{”	{
“}”	}
“,”	,
“;”	;
“:”	:
<<EOF>>	EOF
.	INVALID

Результат работы лексического анализатора – последовательность токенов – используется для дальнейшей обработки синтаксическим анализатором.

2.1 Синтаксический анализатор

Синтаксический анализ — процесс сопоставления линейной последовательности лексем естественного или формального языка с его формальной грамматикой.

Синтаксический анализатор (программу, выполняющую анализ) называют парсером.

Алгоритмы синтаксического анализа делятся на две группы – нисходящие и восходящие. Нисходящий парсер раскрывает продукции грамматики, начиная со стартового символа, до получения требуемой последовательности токенов. Восходящий парсер восстанавливает продукции из правых частей, начиная с токенов и заканчивая стартовым символом.

В данной работе используется вид восходящего парсера LALR – упрощенная версия LR-парсера.

LR-анализатор — синтаксический анализатор, который читает входной поток слева направо и производит наиболее правую продукцию контекстно-свободной грамматики.

Синтаксис многих языков программирования может быть определён грамматикой, которая является LR или близкой к этому, и по этой причине LR-анализаторы часто используются компиляторами для выполнения синтаксического анализа исходных кодов.

LR-парсер основан на алгоритме, приводимом в действие таблицей анализа – структурой данных, которая содержит синтаксис анализируемого языка. Таким образом, термин LR-анализатор на самом деле относится к классу анализаторов, которые могут разобрать почти любой язык программирования, для которого предоставлена таблица анализа. Таблица анализа создаётся генератором синтаксических анализаторов.

LR-анализаторы, в большинстве случаев, непрактично создавать вручную, ввиду высокой сложности ручного построения таблицы разбора. Автоматически сгенерированный программой код всегда может быть дополнен рукописным кодом для увеличения мощности результирующего синтаксического анализатора.

В зависимости от того, как была создана таблица анализа, эти анализаторы могут быть названы простыми LR-анализаторами (SLR), LR-анализаторами с предпросмотром (LALR) или каноническими LR-

анализаторами. LALR-анализатор имеют значительно большую распознавательную способность, чем SLR-анализаторы.

Все разновидности LR-анализатора основаны на одном и том же алгоритме – «сдвиг-свертка».

1. Сдвиг. Перенос очередного входного символа на вершину стека.
2. Свертка. Правая часть сворачиваемой строки должна располагаться на вершине стека. Определяется левый конец строки в стеке и принимается решение о том, каким нетерминалом будет заменена строка.
3. Принятие. Объявление об успешном завершении синтаксического анализа.
4. Ошибка. Обнаружение синтаксической ошибки и вызов подпрограммы восстановления после ошибки.

LR-анализатор принимает решение о выборе «сдвиг/свертка», поддерживая состояния, которые отслеживают, где именно в процессе синтаксического анализа мы находимся.

Результатом работы синтактического анализатора является абстрактное синтаксическое дерево, которое используется при дальнейшей обработке.

Модель LR-анализатора представлена на рисунке 1.

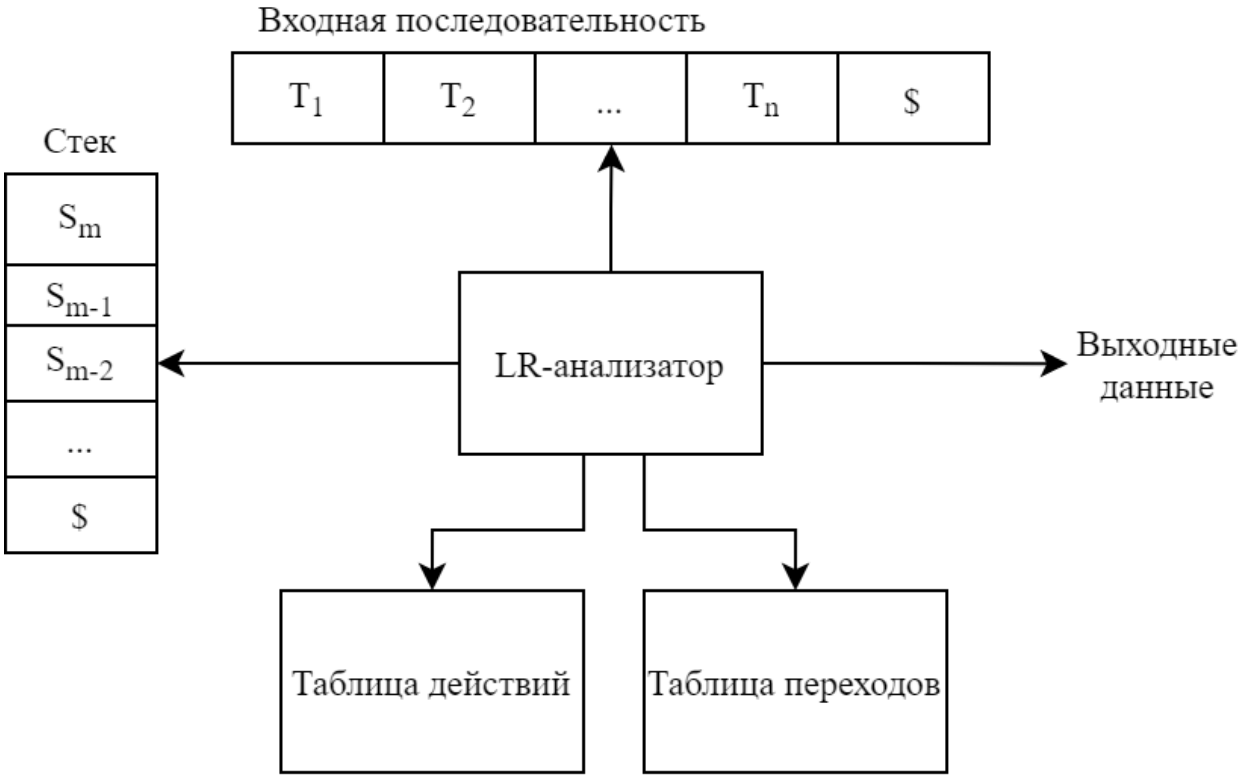

Рисунок 1 – Модель LR-анализатора

Для реализации синтаксического анализа используется функция function parse(input).

На вход поступает последовательность токенов, на выходе получаем абстрактное синтаксическое дерево.

Приоритеты операций, используемые при синтаксическом анализе представлены в таблице 2.

Таблица 2 – Приоритеты операций

Операция	Ассоциативность	Приоритет
=	Правая	1
OR	Левая	2
AND	Левая	3
==, !=	Левая	4
<, >, <=, >=	Левая	5
+, -	Левая	6
*, /	Левая	7
^	Левая	8
NOT	Правая	9
(,)	Левая	10
THEN, ELSE	Правая	11

Пример абстрактного синтаксического дерева для оператора присваивания представлен ниже.

```
[
  {
    "type": "assign",
    "value": [
      "a",
      "5"
    ]
  }
]
```

У каждого оператора есть тип – type и значение – value. В массиве value элементами могут быть как константные значения, так и другие операторы.

2.2 Семантический анализатор

На этапе семантического анализа компилятор проверяет исходный код на наличие ошибочных по смыслу, но правильных синтаксически, конструкций языка.

Настоящий компилятор выявляет три типа семантических ошибок:

1. Повторное объявление переменных;
2. Присваивание необъявленным переменным;
3. Присваивание переменным значения несоответствующего типа.

Алгоритм синтаксического анализа в данном компиляторе реализован таким образом, чтобы обнаруживать семантические ошибки во время парсинга.

Обнаружив объявление переменных парсер вызывает соответствующую функцию и передает ей в качестве аргумента список переменных и их тип. Данная функция проверяет наличие каждой переменной в таблице идентификаторов. Если такой переменной нет в таблице, то функция добавляет ее вместе с соответствующим ей типом. В противном случае –

выводится ошибка, сообщающая о повторном объявлении переменной. Функция, описанная выше имеет следующий вид:

```
function initId(idList, type, lineNo)
```

На вход поступает список переменных, их тип и номер строки, на которой произошло объявление. Возвращаемое значение отсутствует.

Обнаружив присваивание, парсер вызывает функцию, обрабатывающую присваивания, передавая ей в качестве аргумента переменную и значение, которое нужно ей присвоить. Функция проверяет таблицу идентификаторов на наличие переменной. Если в таблице ее нет, то возвращается ошибка присваивания необъявленной переменной. В противном случае – проверяется тип данной переменной и тип значения, которое нужно присвоить. Если они не совпадают, то функция возвращает ошибку, указывающую на несоответствие типов и завершает свою работу. Функция, описанная выше имеет следующий вид:

function setId(id, value, lineNo)

На вход поступает имя переменной, значение присваивания и номер строки, на которой произошло присваивание. Возвращаемое значение отсутствует.

Любая ошибка на этапе компиляции приводит к завершению работы компилятора. На экран выводится сообщение об ошибке и ее причине.

2.3 Перевод в ассемблер

Язык ассемблера — машинно-ориентированный язык программирования низкого уровня. Представляет собой систему обозначений, используемую для представления в удобно читаемой форме программ, записанных в машинном коде. Его команды прямо соответствуют отдельным командам машины или их последовательностям.

На данном этапе компилятор использует полученное на этапе синтаксического анализа абстрактное синтаксическое дерево для перевода каждого оператора в набор инструкций на языке ассемблера.

В настоящем компиляторе алгоритм перевода в ассемблер реализован путем вызова функции генерации кода для каждого оператора исходной программы. Данная функция, в свою очередь, в зависимости от текущего оператора, вызывает соответствующую ему функцию, которая генерирует код только для него и помещает каждую команду в массив команд. Путем взаимных рекурсивных вызовов этих функций данный алгоритм способен перевести любую программу, соответствующую грамматике в набор эквивалентных инструкций на ассемблере.

Функция генерации кода имеет следующий вид:

```
generateCode(tree, regNum) {
  let t = tree.type;
  if (t === 'block') {
    for (let op of tree.value) {
      this.generateCode(op, regNum);
    }
  } else {

this[`generate${t[0].toUpperCase()}${t.slice(1)}`](tree.valu
e, regNum);
    }
}
```

На вход поступает синтаксическое дерево и текущий номер регистра (изначально равен нулю). На выходе получаем последовательность команд на языке ассемблера.

Пример сгенерированного кода ассемблера для каждого оператора представлен в таблице 3.

Таблица 3 – Код ассемблера для каждого оператора

Оператор	Сгенерированный код
a = 5	MOV a, 5

Таблица 3 – Продолжение

if (a > 3) then {} else {}	MOV R0, a
	MOV R1, 3
	CMP R0, R1
	JG T0
	JMP F0
	T0:
	F0:

for (a = 0) to (a < 5) do {}	MOV a, 0
	L0:
	MOV R0, a
	MOV R1, 5
	CMP R0, R1
	JL T0
	JMP F0
	T0:
	JMP L0
	F0:

while (a < 5) do {}	L0:
	MOV R0, a
	MOV R1, 5
	CMP R0, R1
	JL T0
	JMP F0
	T0:
	JMP L0
	F0:

Таблица 3 – Продолжение

read(a, b, c);	PUSH c PUSH b PUSH a CALL READ

write(3 + 4, true, a);	PUSH a
	MOV R0, -1
	PUSH R0
	MOV R0, 3
	MOV R1, 4
	ADD R0, R1
	PUSH R0
	CALL WRITE

2.4 Разработка интерфейса приложения

Для решения поставленных задач было создано приложение, интерфейс которого представлен на рисунке 2.

Рисунок 2 – Интерфейс приложения

Цифрами на рисунке обозначены:

1. Поле ввода кода программы;
2. Таблица токенов;
3. Поле вывода абстрактного синтаксического дерева;
4. Таблица идентификаторов;
5. Поле вывода ассемблированного кода;
6. Поле вывода ошибок.

3 Тестирование приложения

Результаты тестирования приложения представлены на рисунках 3 – 11.

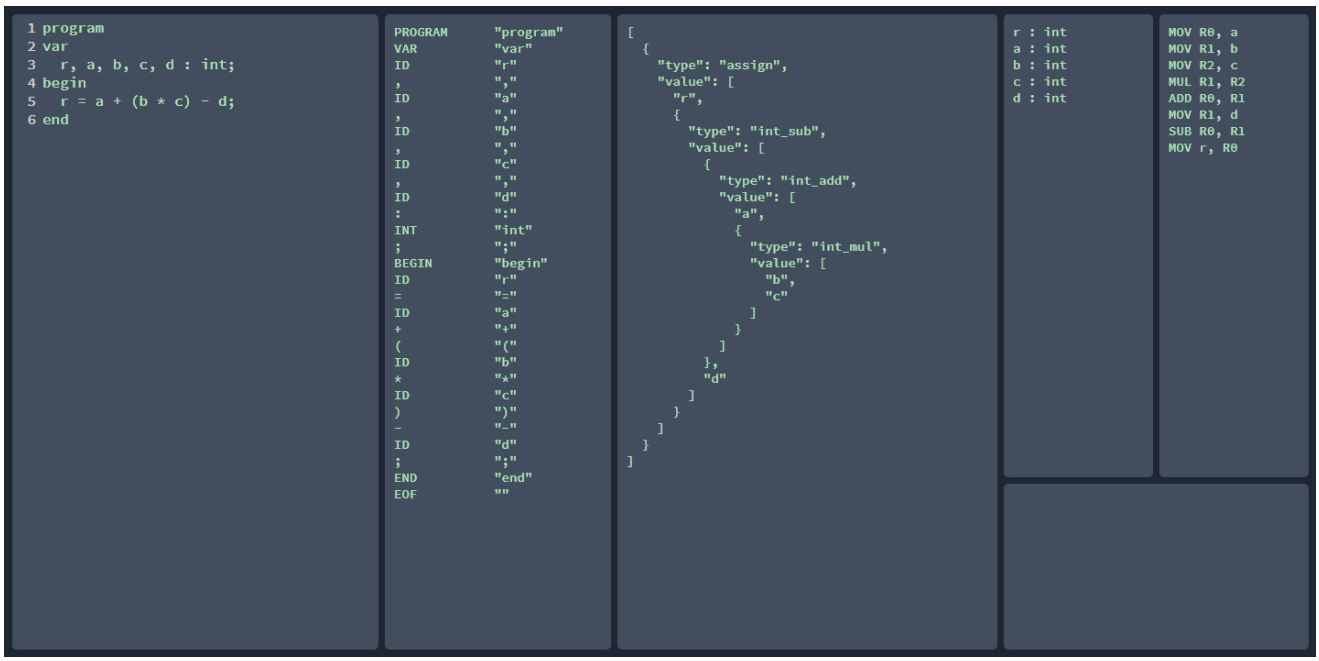

Рисунок 1 – Тестирование оператора присваивания

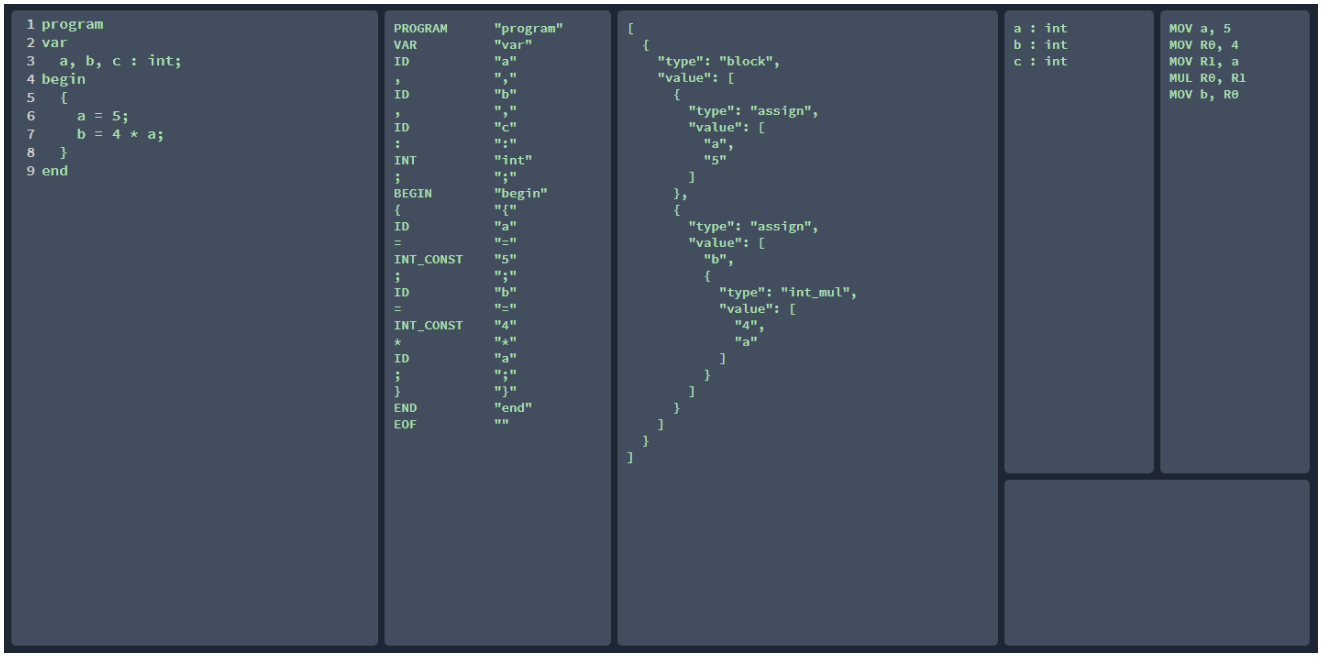

Рисунок 2 – Тестирование составного оператора

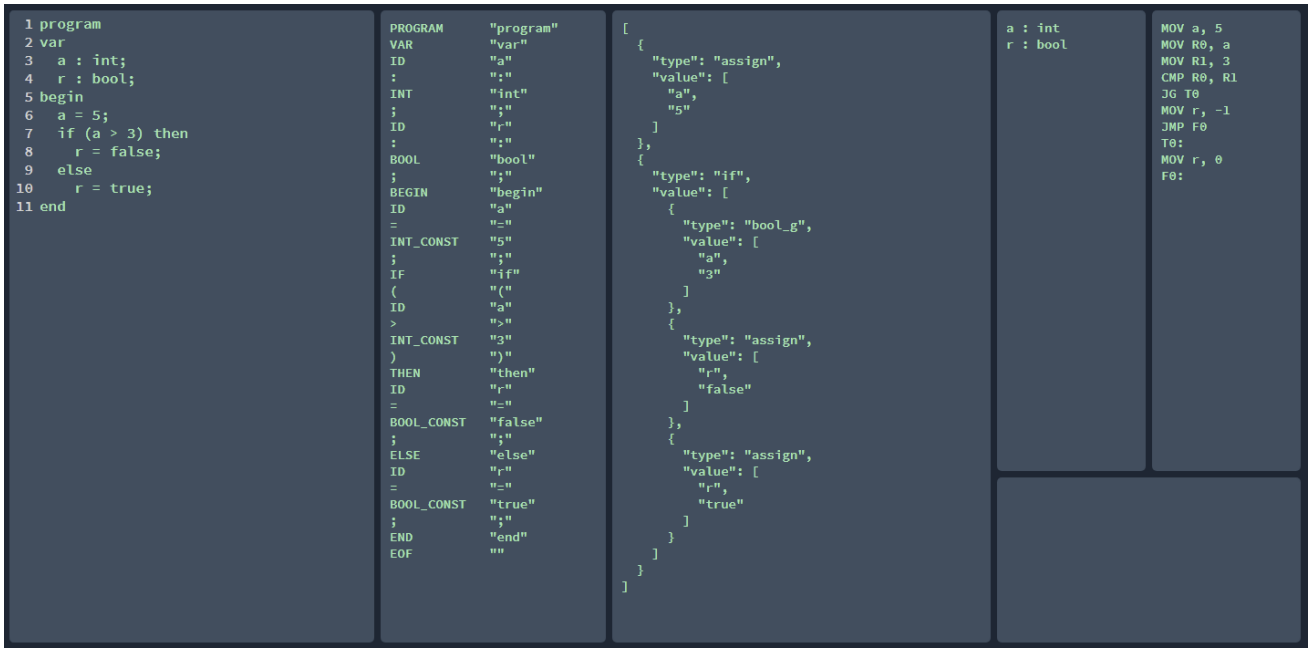

Рисунок 3 – Тестирование условного оператора

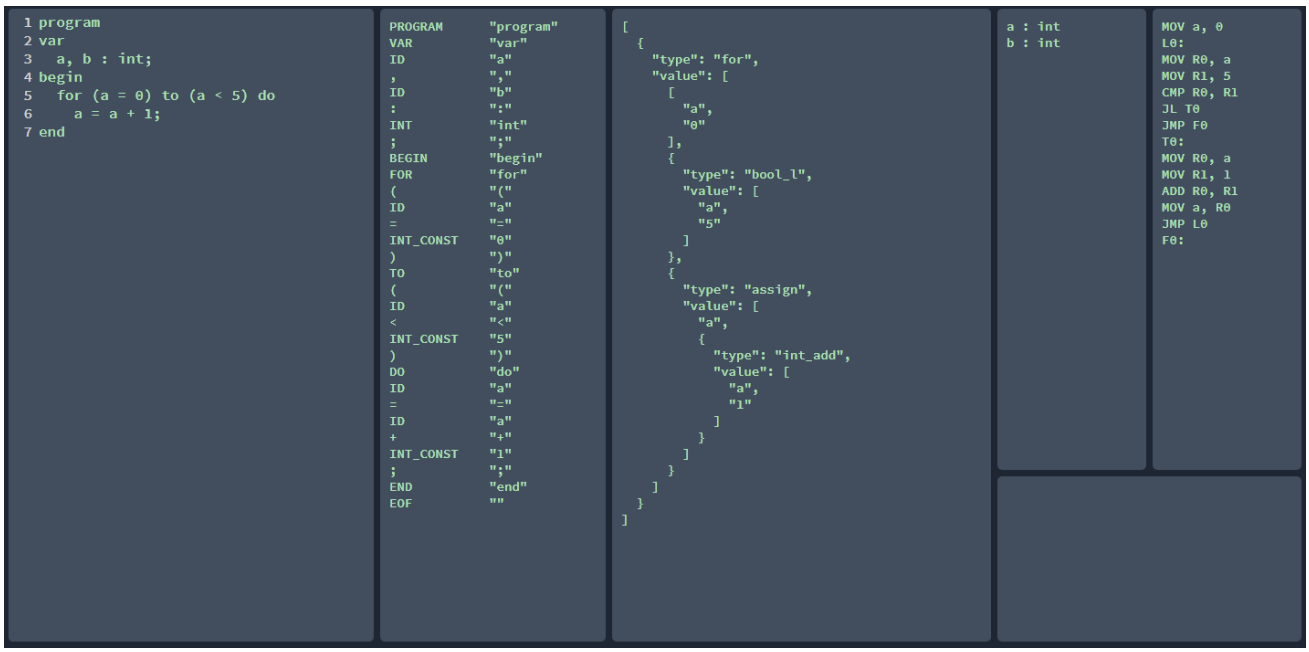

Рисунок 4 – Тестирование оператора фиксированного цикла

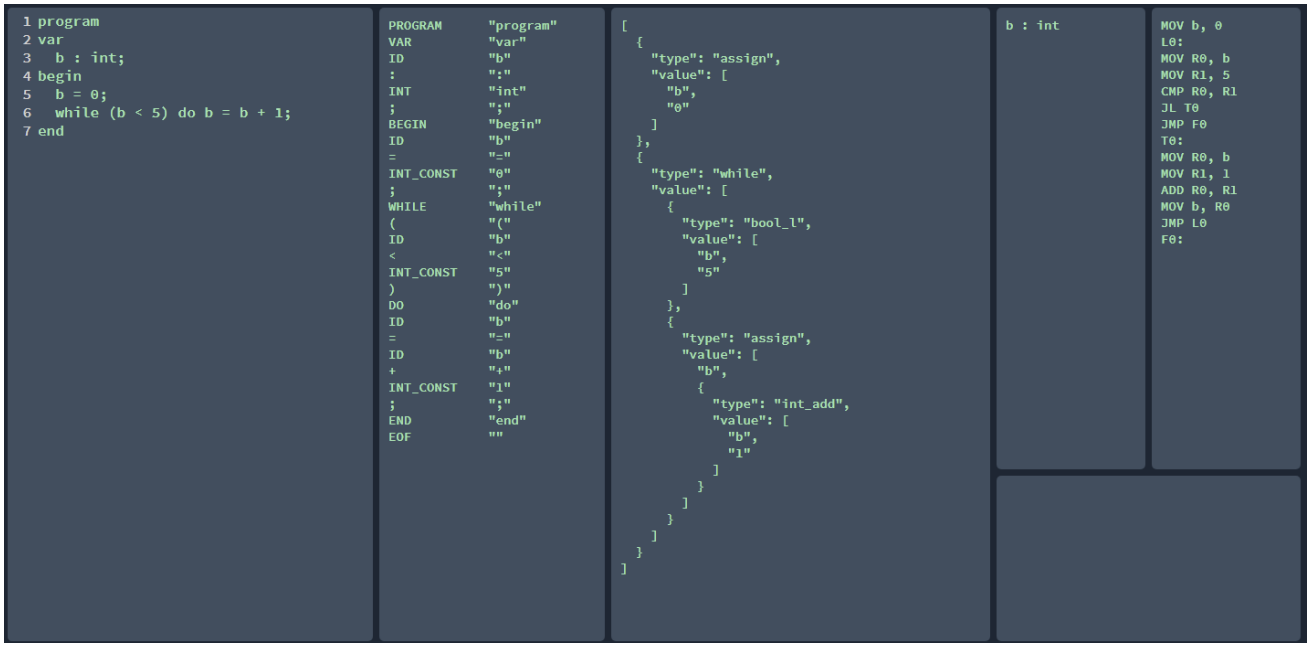

Рисунок 5 – Тестирование оператора условного цикла

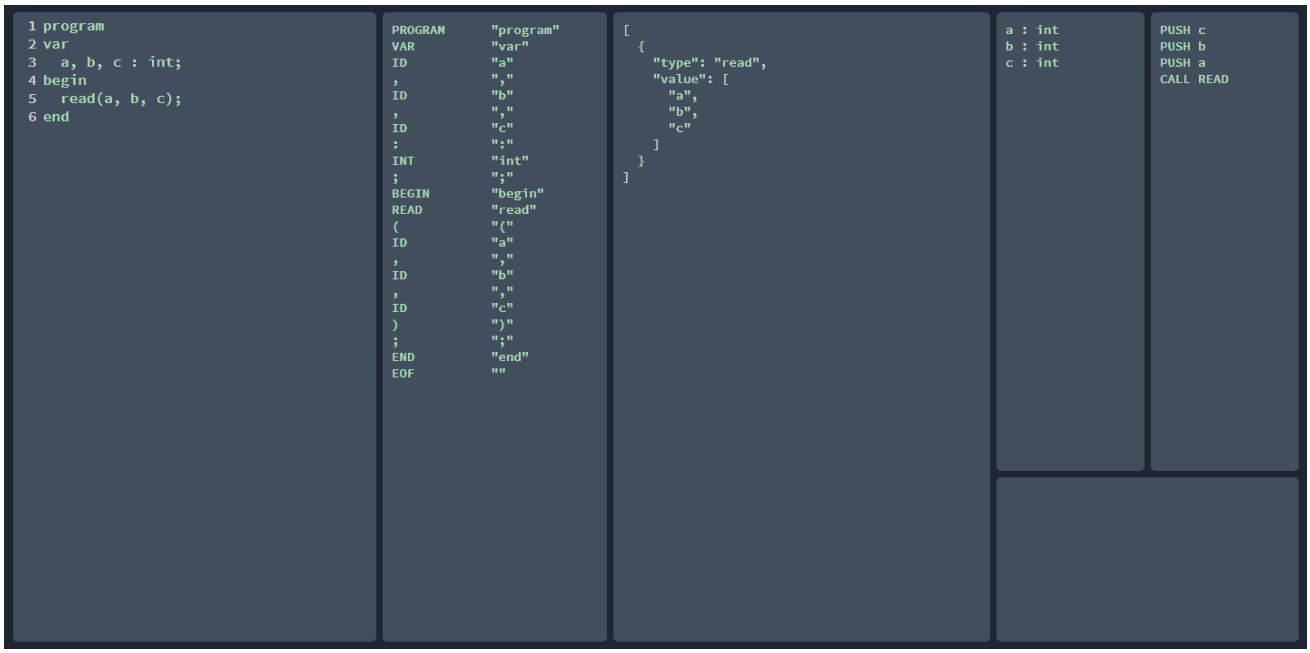

Рисунок 6 – Тестирование оператора ввода

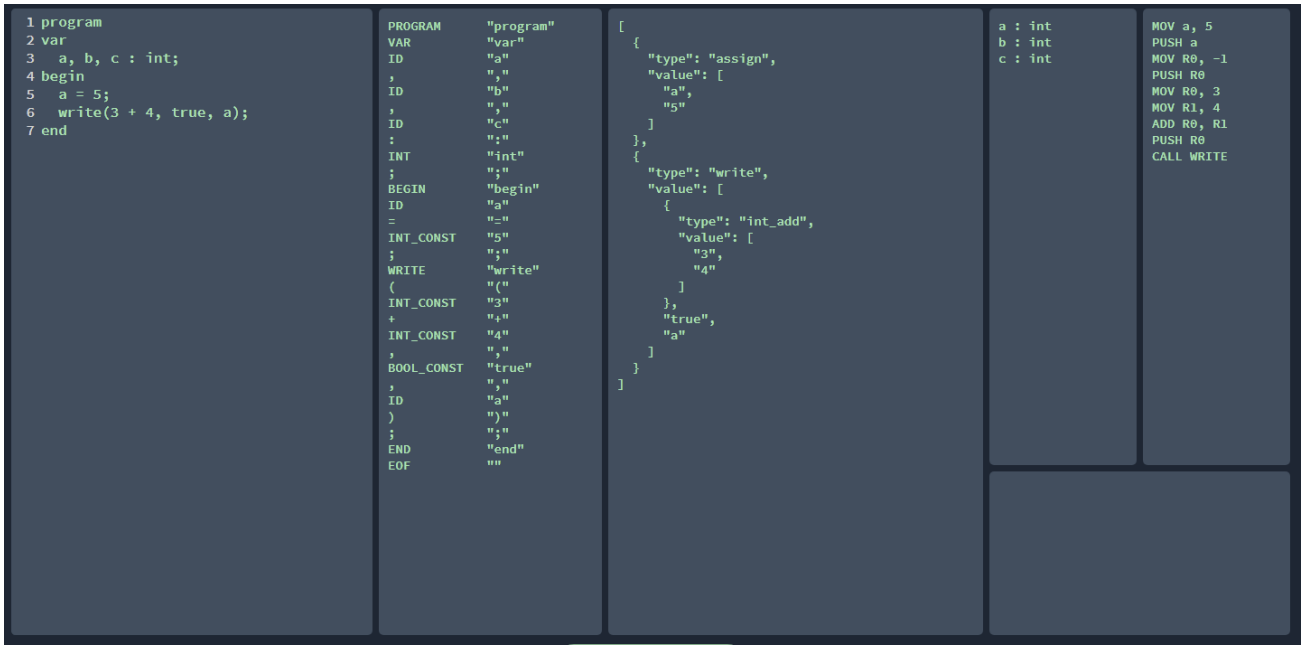

Рисунок 7 – Тестирование оператора вывода

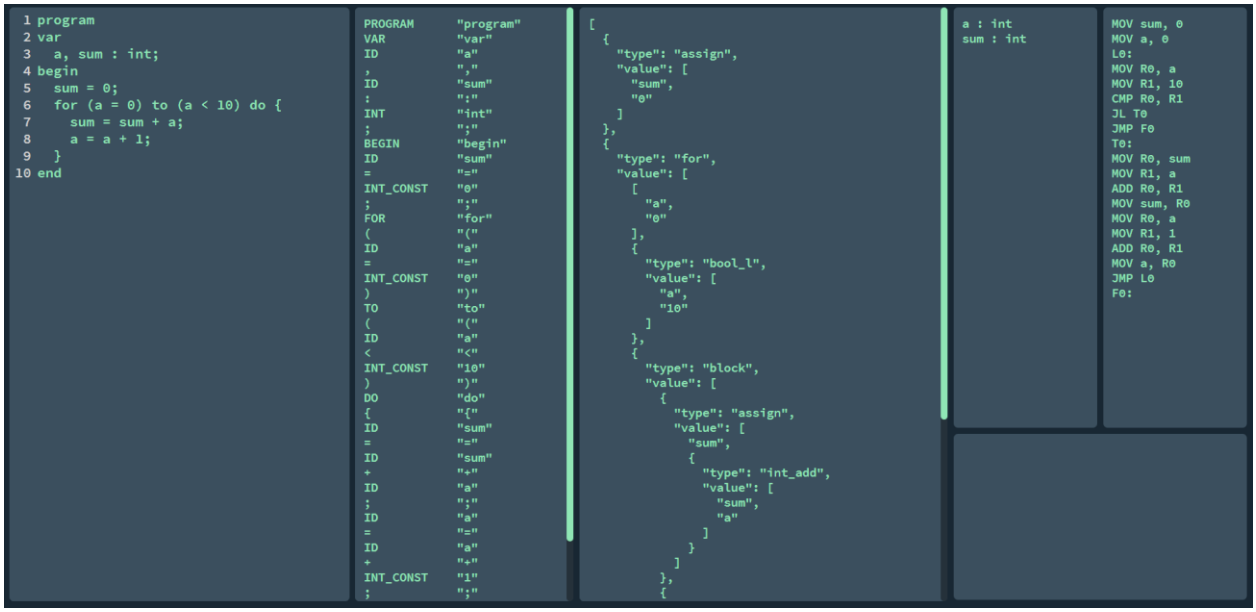

Рисунок 8 – Тестирование программы, вычисляющей сумму цифр от 1 до 10

Рисунок 9 – Тестирования распознавания ошибки повторного объявления переменной

Рисунок 10 – Тестирования распознавания ошибки присваивания необъявленной переменной

Рисунок 11 – Тестирование распознавания ошибки присваивания переменной значения несоответствующего типа

Рисунок 12 – Тестирование распознавания синтаксической ошибки

ЗАКЛЮЧЕНИЕ

В ходе выполнения курсовой работы был разработан компилятор для перевода полученного исходного кода на модульном языке в код ассемблера. При этом были выполнены поставленные задачи:

1) изучение необходимой литературы по теории создания компилятора;
2) разработка грамматики модельного языка программирования по варианту задания;
3) разработка лексического, синтаксического и семантического анализаторов;
4) реализация перевода в ассемблер;
5) программная реализация разработанных алгоритмов.

Входными данными программы является исходный код на модельном языке программирования.

Результатом работы программы является код ассемблера.

Можно выделить следующие направления совершенствования настоящего программного продукта:

- усовершенствование алгоритма перевода в ассемблер;
- реализация дополнительных алгоритмов обнаружения семантических ошибок.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Хомский Н., Миллер Дж. Введение в формальный анализ естественных языков // Кибернетический сборник / Под ред. А.А.Ляпунова и О.Б.Лупанова. — М.: Мир, 1965.
2. Пратт *T. M.* Языки программирования: разработка и реализация – Питер 2002. – 688 с.
3. Ишаков Е. Н. Разработка компиляторов – Оренбург: Изд-во ОГУ. – 2005.

ПРИЛОЖЕНИЕ А
ЛИСТИНГ ПРОГРАММЫ

grammar.jison:

```
/* lexical grammar */
%lex
%%

\s+                   /* skip whitespace */
"//".*                /* skip comments */

"program"             return 'PROGRAM';
"var"                 return 'VAR';
"begin"               return 'BEGIN';
"end"                 return 'END';
"int"                 return 'INT';
"bool"                return 'BOOL';
"if"                  return 'IF';
"then"                return 'THEN';
"else"                return 'ELSE';
"for"                 return 'FOR';
"to"                  return 'TO';
"do"                  return 'DO';
"while"               return 'WHILE';
"read"                return 'READ';
"write"               return 'WRITE';
"not"                 return 'NOT';
"or"                  return 'OR';
"and"                 return 'AND';
"true"|"false"        return 'BOOL_CONST';


([-]?[1-9]\d*|0)\b    return 'INT_CONST';
[a-zA-Z][0-9a-zA-Z]*\b return 'ID';

"=="                  return '==';
">="                  return '>=';
```

```
"<="                    return '<=';
"!="                    return '!=';
"="                     return '=';
"*"                     return '*';
"/"                     return '/';
"-"                     return '-';
"+"                     return '+';
"^"                     return '^';
">"                     return '>';
"<"                     return '<';

"("                     return '(';
")"                     return ')';
"{"                     return '{';
"}"                     return '}';
","                     return ',';
";"                     return ';';
":"                     return ':';

<<EOF>>                 return 'EOF';
.                       return 'INVALID';

/lex
/* operator precedence and assosiations */

%right '='
%left  'OR'
%left  'AND'
%left  '==' '!='
%left  '<' '>' '<=' '>='
%left  '+' '-'
%left  '*' '/'
%left  '^'
%right 'NOT'
%left  '(' ')'
%right 'THEN' 'ELSE'
```

```
%token INVALID

%start start

/* language grammar */

%ebnf
%%
start
    : program EOF {return $1}
    ;

program
    : PROGRAM var_section* code_section    {$$ = {type:"program", value:[$2, $3]}}
    ;

var_section
    : VAR declaration+                   {$$ = {type:"variables section", value: $2}}
    ;

code_section
    : BEGIN operator+ END                  {$$ = {type:"code section", value:$2}}
    ;

declaration
    : id_list ':' type ';'
{initId($1, $3, yylineno); $$ = {type:"declaration", value:$1}}
    ;

type
    : INT
    | BOOL
    ;

operator
    : assign_op ';'{$$ = {type:"assign", value: $1}}
    | if_op        {$$ = {type:"if",     value: $1}}
```

```
    | for_op       {$$ = {type:"for",    value: $1}}
    | while_op     {$$ = {type:"while",  value: $1}}
    | read_op ';'  {$$ = {type:"read",   value: $1}}
    | write_op ';' {$$ = {type:"write",  value: $1}}
    | block_op     {$$ = {type:"block",  value: $1}}
    ;

assign_op
    : ID '=' expression              {$$ = [$1, $3]; setId($1, $3, yylineno)}
    ;

if_op
    : IF '(' expression ')' THEN operator                {$$ = [$3, $6]}
    | IF '(' expression ')' THEN operator ELSE operator  {$$ = [$3, $6, $8]}
    ;

for_op
    : FOR '(' assign_op ')' TO '(' expression ')' DO operator   {$$ = [$3, $7, $10]}
    ;

while_op
    : WHILE '(' expression ')' DO operator                       {$$ = [$3, $6]}
    ;

read_op
    : READ '(' id_list ')'                               {$$ = $3}
    ;

write_op
    : WRITE '(' expression_list ')'                      {$$ = $3}
    ;

block_op
    : '{' operator* '}' {$$ = $2}
    ;

expression
```

```
    : expression '+'  expression          {$$ = {type: 'int_add',  value:[$1, $3]}}
    | expression '-'  expression          {$$ = {type: 'int_sub',  value:[$1, $3]}}
    | expression '*'  expression          {$$ = {type: 'int_mul',  value:[$1, $3]}}
    | expression '/'  expression          {$$ = {type: 'int_div',  value:[$1, $3]}}
    | expression '^'  expression          {$$ = {type: 'int_pow',  value:[$1, $3]}}
    | expression OR   expression          {$$ = {type: 'bool_or',  value: [$1, $3]}}
    | expression AND  expression          {$$ = {type: 'bool_and', value: [$1, $3]}}
    | expression '>'  expression          {$$ = {type: 'bool_g',   value: [$1, $3]}}
    | expression '<'  expression          {$$ = {type: 'bool_l',   value: [$1, $3]}}
    | expression '==' expression          {$$ = {type: 'bool_e',   value: [$1, $3]}}
    | expression '>=' expression          {$$ = {type: 'bool_ge',  value: [$1, $3]}}
    | expression '<=' expression          {$$ = {type: 'bool_le',  value: [$1, $3]}}
    | expression '!=' expression          {$$ = {type: 'bool_ne',  value: [$1, $3]}}
    | NOT expression                      {$$ = {type: 'bool_not', value: $2}}
    | '(' expression ')'                  {$$ = $2}
    | BOOL_CONST                          {$$ = $1}
    | INT_CONST                           {$$ = $1}
    | ID                                  {$$ = $1}
    ;

id_list
    : id_list ',' ID                      {$$ = concatList($1, $3)}
    | ID                                  {$$ = $1}
    ;

expression_list
    : expression_list ',' expression      {$$ = concatList($1, $3)}
    | expression                          {$$ = $1}
    ;
```

compiler.js:

```
class Compiler {
  constructor() {
    this.parser = parser;
    this.error = false;
  }

  compile(code) {
    this.tree = '';
    this.ids = new Map();
    this.tCount = 0;
    this.fCount = 0;
    this.lCount = 0;
    this.error = false;
    this.assembly = [];
    this.lexemes = '';

    const lexer = this.parser.lexer;
    lexer.setInput(code);
    while (!lexer.done) {
        let token = lexer.lex();
        if (token in parser.terminals_) {
          token = parser.terminals_[token];
        }

        let lexeme = `${token}`
        for (let i = 0; i < 13 - token.length; i++) {
          lexeme += ' ';
        }
        lexeme += `"${lexer.yytext}"\n`;
        this.lexemes += lexeme;
    }
    const parsedObj = this.parser.parse(code);
    if (this.error) return;
    this.tree = JSON.stringify(parsedObj.value[1].value, null, 2);
    for (let operator of parsedObj.value[1].value) {
      this.generateCode(operator, 0);
    }
  }

  generateCode(tree, regNum) {
    let t = tree.type;
    if (t === 'block') {
      for (let op of tree.value) {
        this.generateCode(op, regNum);
      }
    } else {
      this[`generate${t[0].toUpperCase()}${t.slice(1)}`](tree.value, regNum);
    }
  }

  generateAssign(value, regNum, destR) {
    let dest = destR ? destR : value[0];
    if (value[1] === Object(value[1])) {
      switch(value[1].type.split('_')[0]) {
        case 'int':
          this.generateIntOp(value[1], regNum);
          break;
        case 'bool':
          this.generateBoolOp(value[1], regNum);
          break;
      }
```

```
      this.assembly.push(`MOV ${dest}, R${regNum}`);
    } else {
      let res;
      if (Number.isInteger(parseInt(value[1]))) {
        res = value[1];
      } else {
        res = +JSON.parse(value[1]) ? -1 : 0;;
      }
      this.assembly.push(`MOV ${dest}, ${res}`);
    }
  }

  generateIntOp(tree, reg1, reg2 = reg1 + 1) {
    if (tree === Object(tree)) {
      let left = tree.value[0];
      let right = tree.value[1];
      this.generateIntOp(left,  reg1);
      this.generateIntOp(right, reg1 + 1);
      let opCode = tree.type.split('_')[1].toUpperCase();
      this.assembly.push(`${opCode} R${reg1}, R${reg2}`);
    } else {
      this.assembly.push(`MOV R${reg1}, ${tree}`);
    }
  }

  generateBoolOp(tree, reg1, reg2 = reg1 + 1, then, els, loop) {
    if (tree === Object(tree)) {
      if (tree.type.split('_')[0] === 'int') {
        this.generateIntOp(tree, reg1);
        return;
      }
      let left = tree.value[0];
      let right = tree.value[1];
      let opCode = tree.type.split('_')[1].toUpperCase();
      this.generateBoolOp(left, reg1); //...args
      this.generateBoolOp(right, reg1 + 1);
      if (opCode === 'OR' || opCode === 'AND') {
        this.assembly.push(`${opCode} R${reg1}, R${reg2}`);
      } else {
        this.assembly.push(`CMP R${reg1}, R${reg2}`);
        this.assembly.push(`J${opCode} T${this.tCount}`);

        then ?
          els ? this.generateCode(els, reg1) : null // else section
        : this.assembly.push(`MOV R${reg1}, 0`);

        this.assembly.push(`JMP F${this.fCount}`);
        this.assembly.push(`T${this.tCount++}:`);

        then ? this.generateCode(then, reg1) : this.assembly.push(`MOV R${reg1}, -
1`); // then section

        if (loop) this.assembly.push(`JMP L${this.lCount++}`);
        this.assembly.push(`F${this.fCount++}:`);
      }
    } else {
      let res;
      if (tree === 'true' || tree === 'false') {
        res = JSON.parse(tree) ? -1 : 0; // -1 to set every bit in R to 1
      } else {
        res = tree;;
      }
```

```
      this.assembly.push(`MOV R${reg1}, ${res}`);
    }
  }

  generateIf(value, regNum, loop, destR) {
    if (value.length === 3) {
      this.generateBoolOp(value[0], regNum, regNum + 1, value[1], value[2]);
    } else {
      this.generateBoolOp(value[0], regNum, regNum + 1, value[1], null, loop, destR);
    }
  }

  generateFor(value, regNum) {
    this.generateAssign(value[0], regNum);
    // this.generateIntOp(value[0][1], regNum);
    this.assembly.push(`L${this.lCount}:`);
    this.generateIf([value[1], value[2]], regNum, true);
  }

  generateWhile(value, regNum) {
    this.assembly.push(`L${this.lCount}:`);
    this.generateIf([value[0], value[1]], regNum, true);
  }

  generateRead(value, regNum) {
    for (let i = value.length - 1; i >= 0; i--) {
      this.assembly.push(`PUSH ${value[i]}`);
    }
    this.assembly.push(`CALL READ`);
  }

  generateWrite(value, regNum) {
    let t;
    console.log(value);
    for (let i = value.length - 1; i >= 0; i--) {
      if (value[i] === Object(value[i])) {
        t = value[i].type.split('_')[0][0].toUpperCase() +
value[i].type.split('_')[0].slice(1);
        this[`generate${t}Op`](value[i], regNum);
        this.assembly.push(`PUSH R${regNum}`);
      } else if (value[i] === 'true' || value[i] === 'false') {
        this.generateBoolOp(value[i], regNum);
        this.assembly.push(`PUSH R${regNum}`);
      } else {
        this.assembly.push(`PUSH ${value[i]}`);
      }
    }
    this.assembly.push('CALL WRITE');
  }
}
```

main.js:

```
const compileButton = document.getElementById("compile-button");
const codeInput = document.getElementById("code-input");
const lexemes = document.getElementById("lexemes");
const identifiers = document.getElementById("identifiers");
const tree = document.getElementById("tree");
const assembly = document.getElementById("assembly");
const errors = document.getElementById("errors");

const compiler = new Compiler();
const editor = CodeMirror.fromTextArea(codeInput, {
  lineNumbers: true,
  theme: "material-darker",
  tabSize: 2
});
editor.save();

compileButton.onclick = () => {
  const code = editor.getValue();
  lexemes.value = '';
  identifiers.value = '';
  tree.value = '';
  assembly.value = '';
  errors.value = '';

  compiler.compile(code);
  if (compiler.error) return;

  for (let l of compiler.lexemes) {
    lexemes.value += l;
  }

  compiler.ids.forEach((value, key) => {
    let id_str = `${key} : ${value}\n`
    identifiers.value += id_str;
  });
```

```
  tree.value += compiler.tree;

  for (line of compiler.assembly) {
    assembly.value += line + '\n';
  }
}

function initId(idList, type, lineNo) {
  const ids = compiler.ids;
  init = (id) => {
    if (!ids.has(id)) {
      ids.set(id, type);
    } else {
      compiler.error = true;
      let error = `Semantic error at line ${lineNo + 1}:\nVariable with name "${id}"
has already been declared.`
      errors.value += error + '\n';
      console.error(error);
    }
  }
  if (Array.isArray(idList)) {
    for (id of idList) {
      init(id);
    }
  } else {
    init(idList);
  }
}

function setId(id, value, lineNo) {
  const ids = compiler.ids;
  const idType = ids.get(id);

  typingCollision = () => {
    compiler.error = true;
    let error = `Semantic error at line ${lineNo + 1}:\n`
    +`Type collision while assinging to ${idType}-typed variable.`;
    errors.value += error + '\n';
```

```
    console.error(error);
  }

  if (ids.has(id)) {
    if (value === Object(value)) { // if value is object
      if ((idType === 'int' && value.type.split('_')[0] === 'bool')
      || (idType === 'bool' && value.type.split('_')[0] === 'int')) {
         // type mismatch
        typingCollision();
      } else {
        // assign the variable here
      }
    } else { // value is not an object
      if ((idType === 'bool' && value != 'true' && value != 'false')
      || (idType === 'int' && !Number.isInteger(parseInt(value)))) {
        // type mismatch
        typingCollision();
      } else {
        // assign the variable here
      }
    }
  } else { // no such id
    compiler.error = true;
    let error = `Semantic error at line ${lineNo + 1}:\n`
    +`Assigning to an undeclared variable "${id}".`
    errors.value += error + '\n';
    console.error(error);
  }
}

concatList = (list, add) => Array.isArray(list) ? [...list, add] : [list, add];
```