**Министерство науки и высшего образования Российской Федерации**

Федеральное государственное бюджетное образовательное учреждение высшего образования

«Владимирский государственный университет

имени Александра Григорьевича и Николая Григорьевича Столетовых» (ВлГУ)

**РАЗРАБОТКА КОМПИЛЯТОРА**

Пояснительная записка

На 26 листах

|  |  |  |
| --- | --- | --- |
| Руководитель |  | к.т.н. доцент кафедры ИЗИ Ю.М. Монахов |
| Исполнитель |  | студент гр. ИСБ-118 Д.В. Волков |

**Владимир 2021**

**Оглавление**

[**Аннотация** 3](#_Toc72715524)

[**1** **ПРОЕКТИРОВАНИЕ КОМПИЛЯТОРА** 4](#_Toc72715525)

[1.1 Основные требования 4](#_Toc72715526)

[1.2 Лексический анализатор 5](#_Toc72715527)

[1.3 Синтаксический анализатор 6](#_Toc72715528)

[1.4 Построение генератора объектного кода 10](#_Toc72715529)

[1.5 Построение транслятора в целевой код 12](#_Toc72715530)

[**2** **ПРОВЕРКА НА СООТВЕТСТВИЕ ОСНОВНЫМ ТРЕБОВАНИЯМ** 13](#_Toc72715531)

[**Приложение** 25](#_Toc72715532)

[Приложение 1 (реквизиты) 25](#_Toc72715533)

[Приложение 2 (КС-грамматика языка) 25](#_Toc72715534)

# **Аннотация**

В данном программном документе приведён текст компилятора подмножества процедурно-ориентированного языка. Компилятор реализован на языке Python с использованием библиотек rply и llvmlite. Основная функция компилятора – проверка принадлежности исходной цепочки входному языку и генерация выходной цепочки символов виде llvm-кода для дальнейшей трансляции в целевой код при помощи утилит llc и gcc.

Разработка компилятора подмножества процедурного языка в ассемблер состоит из следующих стадий:

* построение лексического анализатора;
* построение синтаксического анализатора;
* построение генератора llvm-кода
* генерация целевого кода

# **ПРОЕКТИРОВАНИЕ КОМПИЛЯТОРА**

## Основные требования

Разработка будет производиться в соответствии со следующими требованиями:

* Требования к входному языку:

1. Должны присутствовать операторные скобки;
2. Должна игнорироваться индентация программы;
3. Должны поддерживаться комментарии любой длины;
4. Входная программа должна представлять собой единый модуль, но поддерживать вызов функций.

* Требования к операторам:

1. Оператор присваивания;
2. Арифметические операторы;
3. Логические операторы (И, ИЛИ, НЕ);
4. Условный оператор (ЕСЛИ);
5. Оператор цикла (while);
6. Базовый вывод (строковой литерал, переменная);
7. Типы (целочисленный, вещественный).

## Лексический анализатор

Лексический анализатор является первой фазой компилятора. Он преобразует входной поток символов в поток токенов.

Грамматика языка реализована с использованием библиотеки rply.lexer. Грамматика языка включает в себя элементы из популярных языков, таких как Python, C и др.

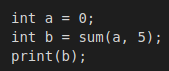
Список зарезервированных слов:

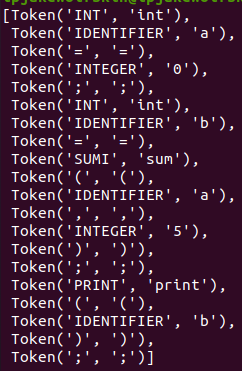
* int
* float
* def
* return
* print
* while
* break
* continue
* if
* else
* and
* or
* not

А также прочие символы на подобии *“{}”, “=”, “>”* и тд.

В качестве результата лексический анализатор выдает поток токенов.

Пример:





## Синтаксический анализатор

Второй стадией компилятора является синтаксический анализ. На вход синтаксическому анализатору подаётся набор токенов из лексического анализатора. На основе грамматики языка строится дерево разбора грамматики.

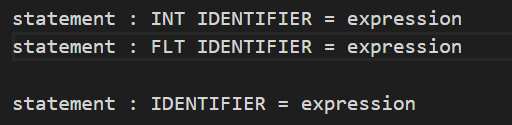
Все взаимодействия во время построения дерева разбора обрабатываются через класс Parser, который возвращает классы модуля AST при совпадении правила из КС-грамматики языка.

КС-грамматика языка представлена ниже.

Привила использования грамматики представлена в Приложении.

**Присвоение**

Блок присвоения представляется следующим правилом КС-грамматики:

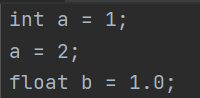


Если переменная не была определена в программа (то есть ее нет в таблице символов), то указывается тип переменной (поддерживаются 2 типа: *int* – целочисленное и *float* – с плавающей точкой), имя и выражение, которое следует присвоить переменной.

Если переменная уже определена в программе (есть в таблице символов в текущей функции), то тип повторно не указывается, необходимо указать имя переменной и выражение, которое ей присваивается.

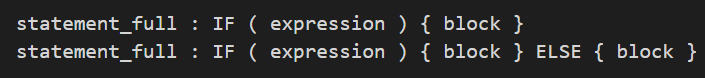
Переменные в разных функциях могут иметь одинаковые имена.

Пример:



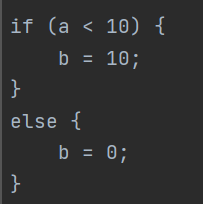
**Ветвление**

Блок ветвления определяется следующим правилом КС-грамматики:



Блок обязательно содержит ключевое слово *if*, условие ветвления в скобках и тело в фигурных скобках. Опционально блок может содержать ветку *else*, которая выполняется в случае, если условие ветвления ложно.

Пример:



**Циклы**

Блок цикла определяется следующим правилом КС-грамматики:

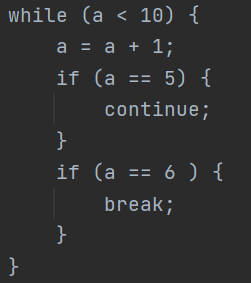


Аналогично блоку ветвления, блок цикла содержит ключевое слово (*while*), условие цикла в скобках и тело цикла в фигурных скобках.

Тело цикла будет выполняться до тех пор, пока условие цикла истинно.

Кроме того, для данного блока имеются 2 специфичных оператора: *break* и *continue*. Оператор *break* позволяет досрочно выйти из тела цикла и продолжить выполнение команд, идущих за ним, а оператор *continue* позволяет прервать текущую итерацию и начать выполнение тела цикла заново.

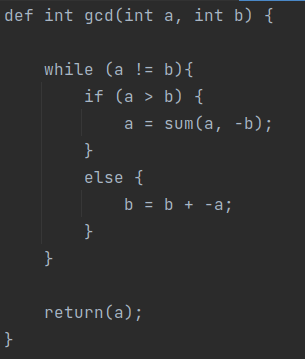
Пример:



**Определение функций**

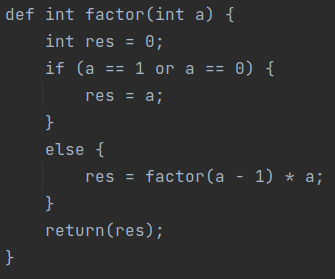
Пользователь может определять свои функции для использования их в коде. Функция обязательно должна иметь тип и возвращать выражение указанного типа. Опционально функция может содержать аргументы, которые так же должны иметь тип.

Пример (алгоритм Евклида):



Так же в функции возможен вызов самой себя (рекурсия).

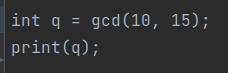
Пример (подсчет факториала):



**Вызов функций**

Функции, определенные пользователем, можно вызывать.

Пример:



Помимо определяемых пользователем функций, в языке есть встроенные функции, такие как *sum(a, b)*, *sumf(a, b), sub(a, b)* и *subf(a, b)*.

Пример:

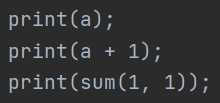


В результате выполнения данной команды в переменной *a* будет храниться значение 4 типа *int*.

**Стандартный вывод**

Встроенная функция *print(a)* позволяет выводить на стандартный вывод выражение *a.*

Пример:



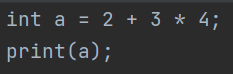
## Построение генератора объектного кода

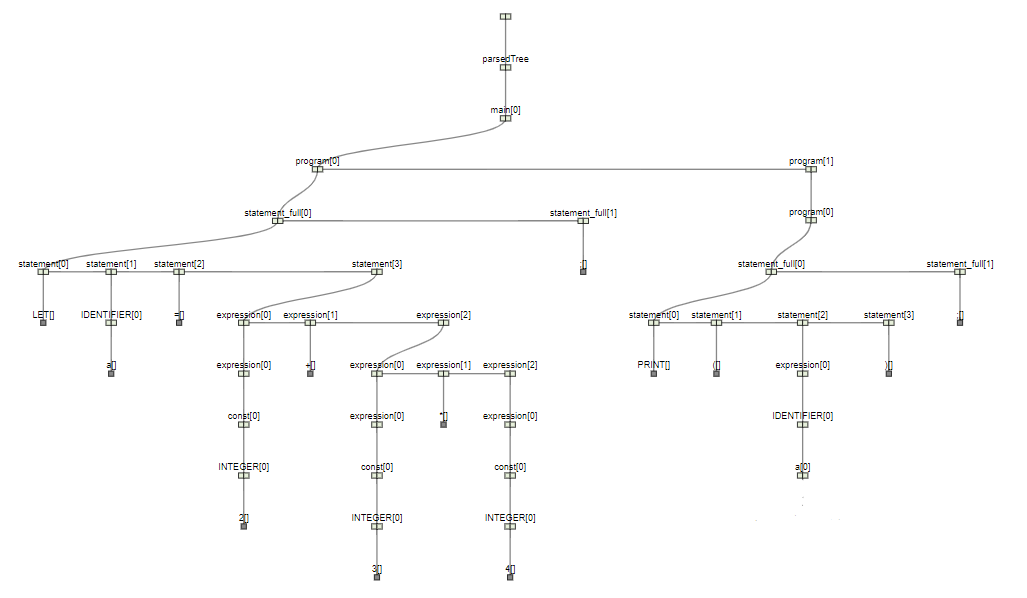
Генерация объектного кода выполняется объектами модуля AST, каждый из которых имеет метод генерации llvm-кода (*eval*).

Независимо от исходного кода в целевой код на языке llvm добавляются встроенные функции языка, в том числе определяется функция стандартного вывода.

В модуле AST определены классы вершин синтаксического дерева. При синтаксическом анализе эти вершины добавляются к корневой вершине *main*.

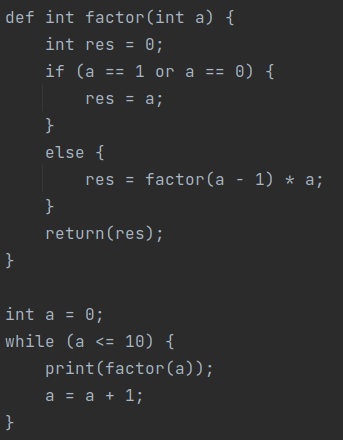
Пример дерева разбора для программы:

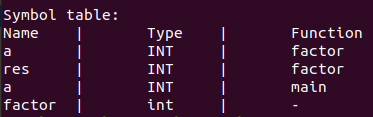




Кроме того, генератор объектного кода создает таблицу символов для программы. Запись в таблице содержит 3 поля: сам символ, тип данных и функция, в которой определен символ (или прочерк, если символ – функция).

Пример таблицы символов:

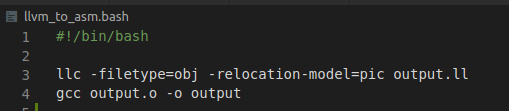




## Построение транслятора в целевой код

Трансляция в целевой код из кода llvm производится с помощью bash-скрипта. Сначала идет вызов утилиты llc, которая генерирует объектный код, далее вызывается утилита gcc, генерирующая целевой код.

Содержимое файла llvm\_to\_asm.bash:



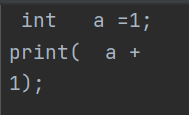
В результате генерируются 2 файла: *output.o*, содержащий объектный код и *output*, содержащий целевой код и являющийся исполняемым.

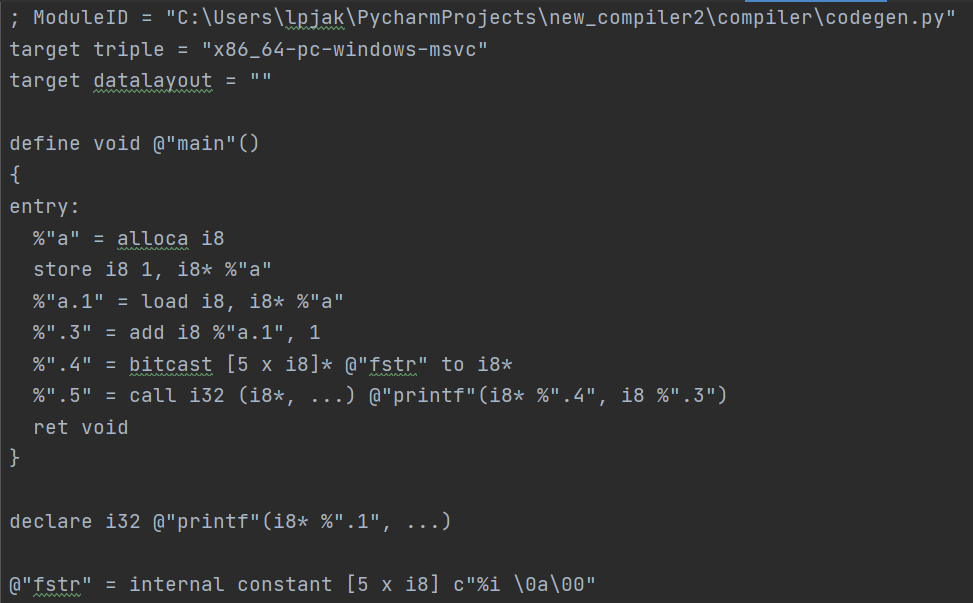
# **ПРОВЕРКА НА СООТВЕТСТВИЕ ОСНОВНЫМ ТРЕБОВАНИЯМ**

При проектировании компилятора к основному языку были установлены следующие минимальные требования: наличие операторных скобок, игнорирование пробелов и идентации программы, поддержка многострочных комментариев и вызова функций. Наличие операторов присваивания, условных, цикла, арифметических, логических. Должны присутствовать два типа данных – целочисленный и вещественный.

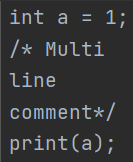
Далее приведено тестирование компилятора.

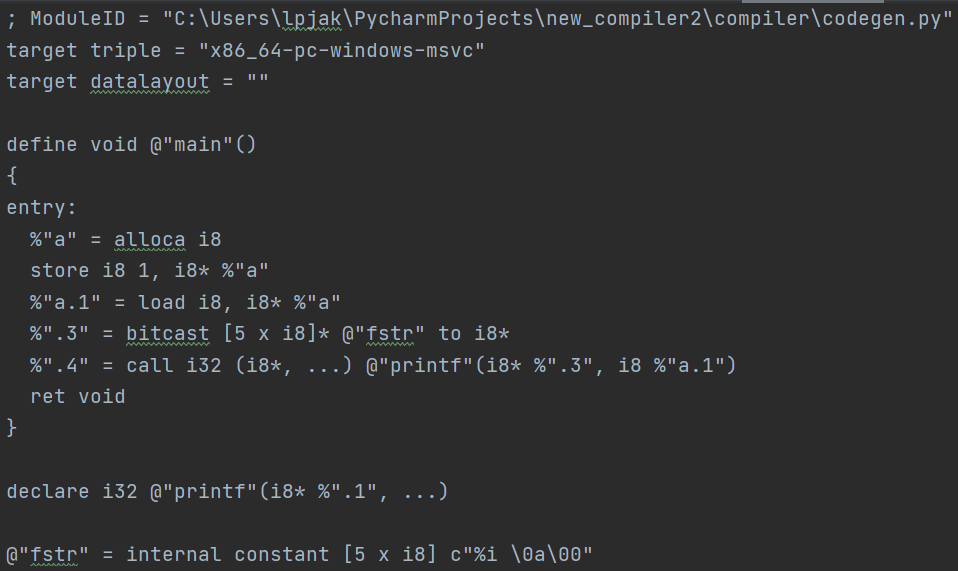
Проверка на игнорирование пробелов и индентацию:



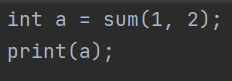


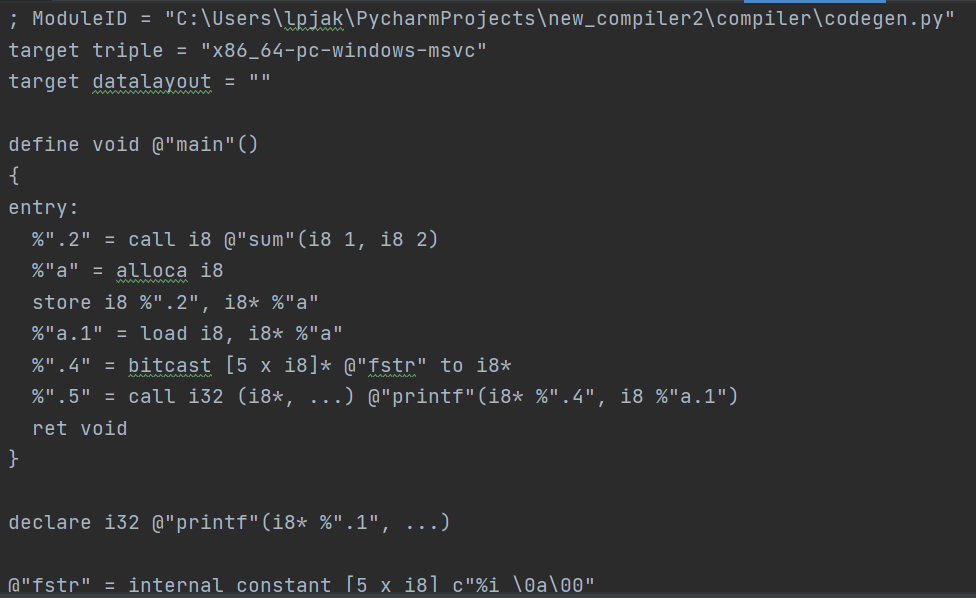
Проверка многострочных комментариев

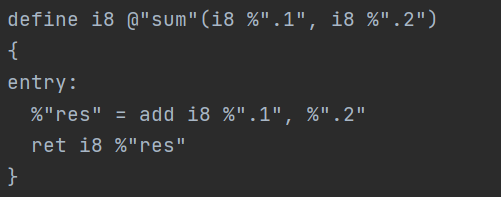




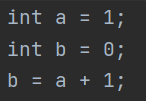
Проверка вызова функций

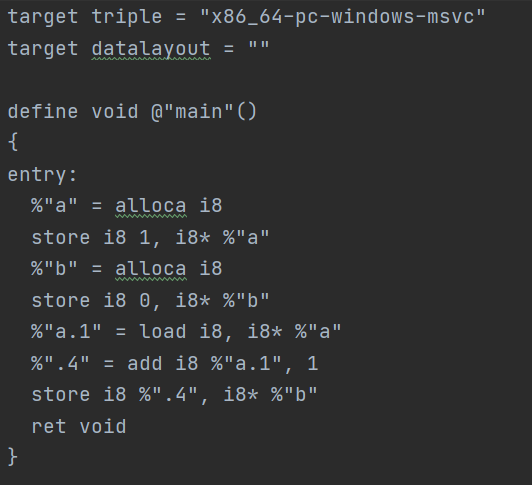




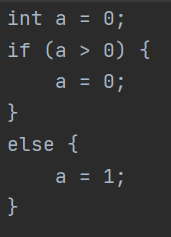


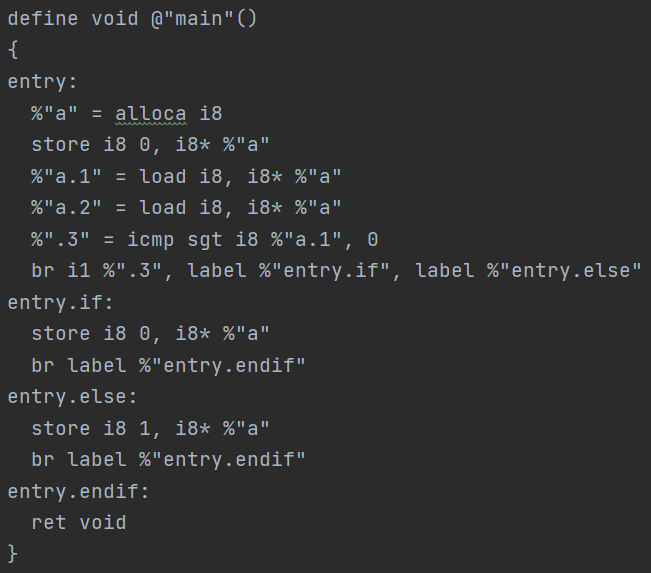
Проверка оператора присваивания:



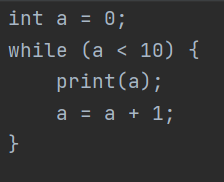


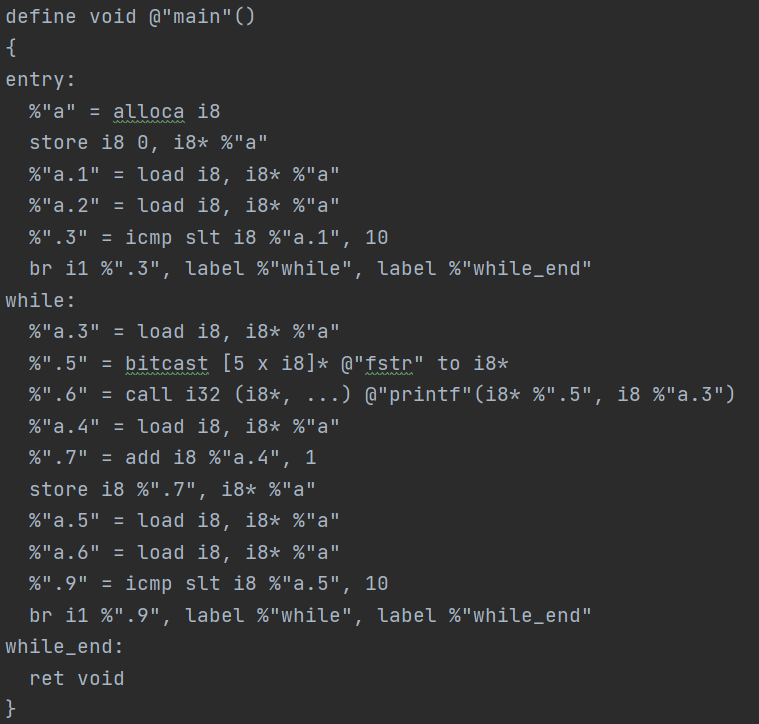
Проверка условного оператора:

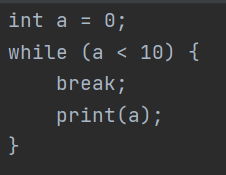


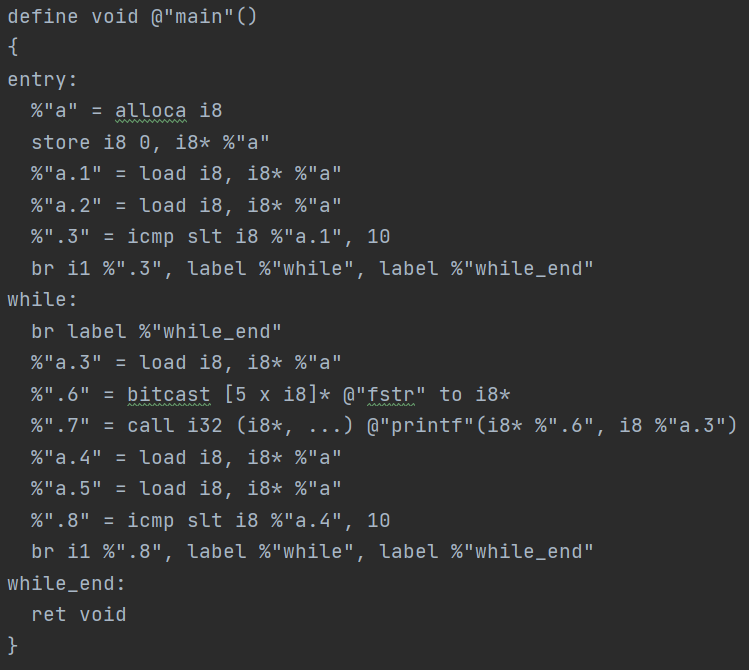


Проверка оператора цикла:



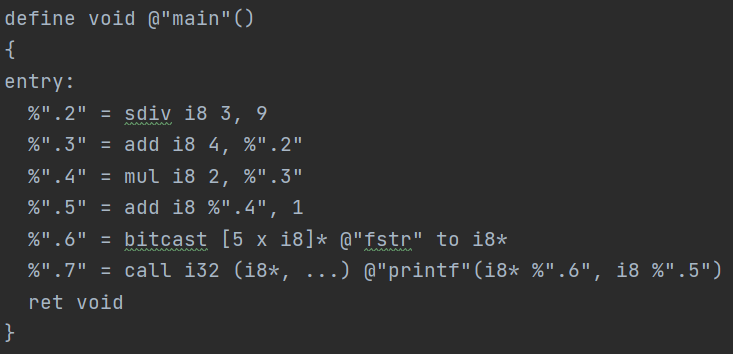




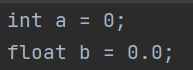


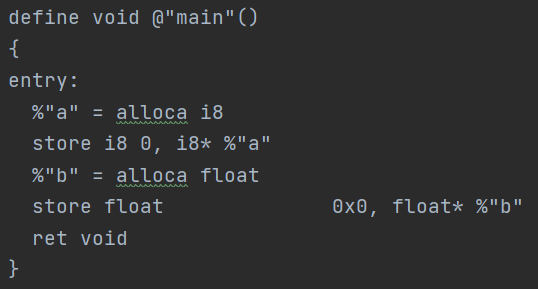
Проверка арифметики:





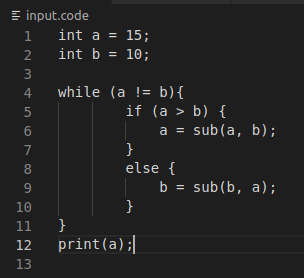
Проверка типов данных:



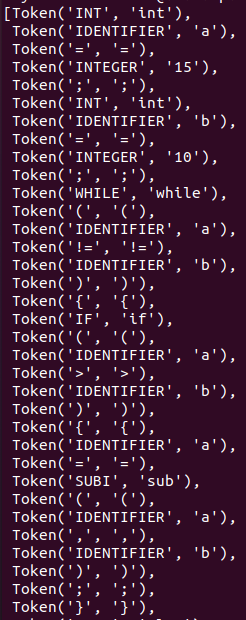
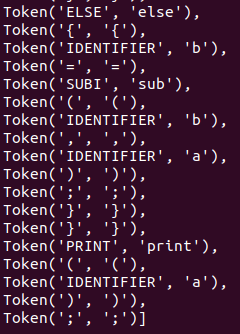


Проверка языка на прикладной задаче (алгоритм Евклида):

1. Исходный код



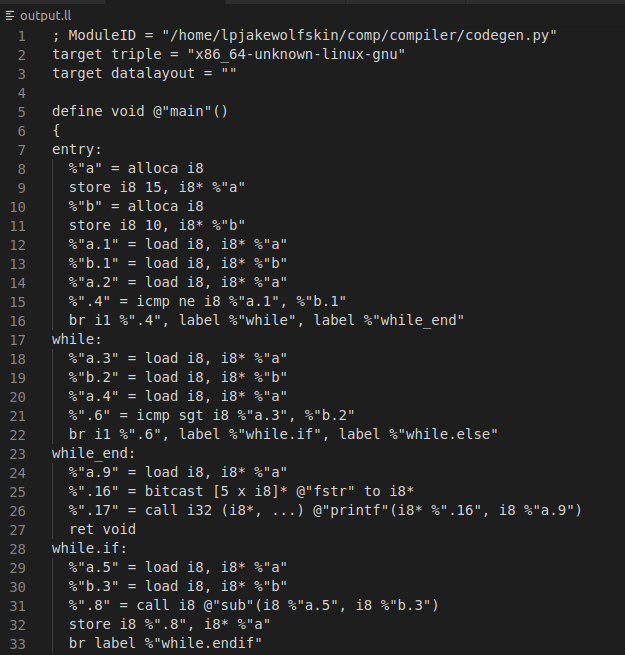
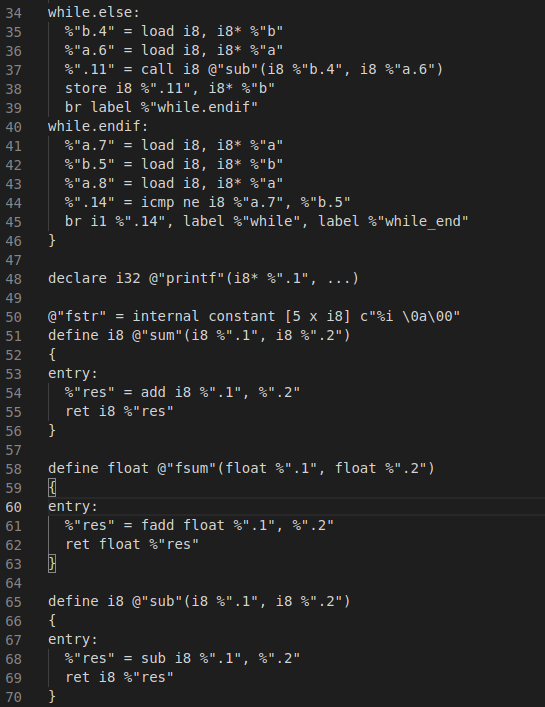
1. Поток токенов

1. Таблица символов



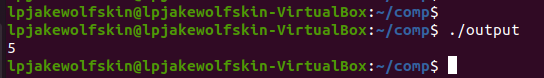
1. llvm-код

1. Целевой код



1. Результат выполнения



Программа верно выполнила исполнила прикладной алгоритм (НОД(15, 10) = 5).

В данном алгоритме был показан следующий функционал языка:

* Типы данных (int)
* Оператор присваивания
* Условный оператор
* Оператор цикла (while)
* Базовый вывод
* Вызов встроенных функций (sub)

# **Приложение**

## Приложение 1 (реквизиты)

Github-репозиторий:

<https://github.com/LPJWS/Compiler/tree/main>

## Приложение 2 (КС-грамматика языка)

main : program

program : statement\_full

program : statement\_full program

statement\_full : IF ( expression ) { block }

statement\_full : IF ( expression ) { block } ELSE { block }

statement\_full : WHILE ( expression ) { block }

statement\_full : statement ;

block : statement\_full

block : statement\_full block

statement : expression

statement : BREAK

statement : CONTINUE

statement : PRINT ( expression )

statement : INT IDENTIFIER = expression

statement : FLT IDENTIFIER = expression

statement : IDENTIFIER = expression

expression : expression SUM expression

expression : expression SUB expression

expression : expression MUL expression

expression : expression DIV expression

expression : SUB expression

expression : expression != expression

expression : expression == expression

expression : expression >= expression

expression : expression <= expression

expression : expression > expression

expression : expression < expression

expression : expression AND expression

expression : expression OR expression

expression : NOT expression

expression : SUMI ( expression , expression )

expression : SUMF ( expression , expression )

expression : SUBI ( expression , expression )

expression : SUBF ( expression , expression )

expression : IDENTIFIER

expression : const

expression : ( expression )

const : FLOAT

const : INTEGER

const : STRING

statement\_full : FUNC INT IDENTIFIER ( args ) { block }

statement\_full : FUNC FLT IDENTIFIER ( args ) { block }

arg : INT IDENTIFIER

arg : FLT IDENTIFIER

args : arg

args : arg , args

expression : IDENTIFIER ( )

expression : IDENTIFIER ( args\_call )

args\_call : expression

args\_call : expression , args\_call