*Государственное образовательное учреждение высшего профессионального образования*

|  |  |
| --- | --- |
| **Gerb-BMSTU_01** | ***«Московский государственный технический университет  имени Н.Э. Баумана»***  ***(МГТУ им. Н.Э. Баумана)*** |

ФАКУЛЬТЕТ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

КАФЕДРА \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**РАСЧЁТНО-ПОЯСНИТЕЛЬНАЯ ЗАПИСКА**

**к курсовой работе на тему:**

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Студент \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ **\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

(Подпись, дата) (И.О.Фамилия)

Руководитель курсовой работы \_\_\_ **\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

(Подпись, дата) (И.О.Фамилия)

Москва, 20\_\_\_

Содержание

[Введение 3](#_Toc482108216)

[1.1. О компиляторах и стадиях компиляции 3](#_Toc482108217)

[1.2. О проектах Numba, Pyston 4](#_Toc482108218)

[Аналитический раздел 5](#_Toc482108219)

[2.1. Общее описание языка Python (версии 2 и 3) 5](#_Toc482108220)

[2.2. Описание грамматики и возможностей MiniPython 8](#_Toc482108221)

[Конструкторский раздел 11](#_Toc482108222)

[3.1. Antlr и генерация лексера и парсера 11](#_Toc482108223)

[3.2. Семантический анализ 12](#_Toc482108224)

[Технический раздел 15](#_Toc482108225)

[4.1. Описание llvm и предпосылки кодогенерации 15](#_Toc482108226)

[4.2. Предложения по разработке бэкенд-части компилятора 15](#_Toc482108227)

[4.3. Пример программы на MiniPython 16](#_Toc482108228)

[Заключение 20](#_Toc482108229)

[Список источников 21](#_Toc482108230)

# Введение

* 1. О компиляторах и стадиях компиляции

Компилятор – это программа, предназначенная для трансляции программного кода на языке программирования в исполняемый машинный код, либо код некоторой виртуальной машины (ВМ). В качестве ВМ может выступать одна из технологий: llvm (low-level virtual machine), jvm (java virtual machine), .net CLR (common language representation) от Microsoft и т. д. В схеме компиляции с виртуальной машиной происходит две трансляции: сначала из исходного кода в код промежуточного представления (код ВМ), а затем в машинный код для целевой платформы (x86, amd64 или другие). В рамках данной работы предполагается ориентироваться на ВМ llvm, как на наиболее открытую и распространенную. К тому же, llvm способна выполнять большое количество оптимизаций уровня кода [8].

Процесс компиляции разделяют на несколько этапов. Сначала выполняется лексический анализ. Его задача – это разбиение входного текста программы на цепочку токенов, в соответствии с лексическими правилами языка. В результате разбиения получается массив структур, каждая из которых относится к определенному типу (идентификатор, литерал, разделитель, ключевое слово, комментарий и т. д.). Внутри структур информация о содержимом токена, месте, где он встретился и его контексте. В английской литературе, модуль лексического анализа называют «лексер» (Lexer) [1].

На втором этапе с полученным массивом токенов работает синтаксический анализатор. Он пытается построить разбор в соответствии с грамматикой исходного языка (обычно это КС-грамматика, описанная в расширенной нотации Бэкуса-Наура). В случае, если не удалось применить ни одно правило, генерируется ошибка вида «в позиции X ожидался токен Y». В случае полностью успешного разбора, строится абстрактное синтаксическое дерево (AST). В вершинах такого дерева находятся нетерминалы, а в листьях токены. В английской литературе модуль синтаксического анализа называют «парсер» (Parser), а этапы анализа в совокупности фронтенд-частью компилятора [1].

Полученное дерево в дальнейшем пропускается через дополнительные утилиты, которые занимаются восстановлением потока исполнения, выведением типов переменных, оптимизацией, и в конечном итоге, генерацией кода виртуальной машины. Все эти этапы выполняются отдельными модулями компилятора и возможно в несколько обходов по AST. В совокупности их называют бэкенд-частью компилятора. Иногда оптимизатор выделяют в отдельную от бэкенда часть.

В данной работе предлагается рассмотреть компиляцию исходного кода из языка MiniPython, описание которого будет предложено ниже, в промежуточный код виртуальной машины llvm под названием IR (intermediate representation).

* 1. О проектах Numba, Pyston

Язык MiniPython образован от языка Python версии 3.6 путем упрощения его грамматики. Он остается полностью совместимым (то есть все коды MiniPython будут корректно запущены на интерпретаторе языка Python) и тьюринг-полным.

Существующие реализации интерпретаторов Python транслируют инструкции языка блок за блоком в собственный байт-код и, по сути, самостоятельно эмулируют виртуальную машину. Их главным недостатком является низкая производительность получаемых программ. Сам язык обладает строгой динамической типизацией, чем собственно и обосновывается подход с интерпретацией [2]. При этом, на практике, в 90% программ достаточно иметь строгую типизацию с выведением типов (type inference). Аналогичный подход к типизации реализован в таких современных языках, как C++ 11 (ключевое слово auto), C# (var) и Haskell. Единственное отличие от динамической типизации в том, что за одним символьным идентификатором закрепляется один тип, который в дальнейшем не может меняться в рамках одной области видимости. В подавляющем большинстве хорошо-написанных программ это условие выполняется. Все это приводит к идее создать компилятор высокопроизводительного кода llvm из грамматики языка Python.

Аналогичный проект разрабатывался в компании Dropbox самим создателем Python Гвидо Ван Россумом и назывался Pyston [6]. Главный упор был сделан на производительности получаемых программ в ущерб удобству использования, стабильности и поддержке инфраструктурных технологий. Разработка совместима только с python2.7 и протестирована на ОС Ubuntu, а в настоящее время развивается слабо. Лучшие решения в дальнейшем перекочевали в интерпретатор python 3. Сейчас же для экспериментов доступен docker образ, на котором и снимали benchmark-и, которые говорят о производительности на уровне статически типизированного CPython.

Отдельная группа энтузиастов занималась задачей ускорения python-кода не путем переработки компилятора, а с помощью генерации оптимизированного кода llvm IR для специально задекорированных функций в коде пользовательской программы. В этом решении также фиксируются типы параметров функции и локальных переменных (однако указывать их нужно вручную). Проект получил название Numba и очень перспективен для использования с массивными математическими вычислениями (numpy и scipy) [9]. Основной недостаток в ограниченной поддержке грамматики python.

Компилятор MiniPython призван стать более производительной версией языка python, сохранив все его преимущества синтаксиса и семантики (простота, выразительность и однозначность).

# Аналитический раздел

2.1. Общее описание языка Python (версии 2 и 3)

Язык программирования Python был создан сотрудником голландского института Гвидо Ван Россумом в 1991 году. По классификации языков, он является преимущественно императивным, но с некоторыми особенностями и конструкциями из функциональных языков. Его развитие подчиняется так называемой философии Python, основные пункты в которой следующие:

* Явное лучше неявного
* Простое лучше сложного
* Особые случаи не должны приводить к нарушению правил
* Должен быть только один очевидный способ, как что-то сделать
* Читаемость кода имеет значение

Благодаря следованию этим пунктам, язык обладает уникальными свойствами. Он лёгок в изучении и создании программ. Программный код разных программистов выглядит одинаковым и легко читается. Листинги содержат в полтора раза меньше строк, чем другие языки. Теперь переходим к собственно описанию языка [2].

Во-первых, в Python всё является объектами. К примеру, литералы – это объекты соответствующих типов (для чисел это int(), float(), для строк str(), unicode(), для булевых это bool()), классы – это объекты метакласса type(), функции, методы классов также объекты, и в добавок модули и пакеты тоже объекты соответствующих классов. Все классы наследуются от глобального класса object.

Во-вторых, система типов языка является строгой (то есть, не допускает неявных приведений типов) и динамической (сами типы определеяются в процессе выполнения). Последнее также означает, что типы переменных, аргументов функций не указываются, а выводятся.

В-третьих, в python имеются специальные составные литералы для коллекций. Это список (используемый в качестве массива list(), причем элементы могут быть различных типов), словарь или ассоциативный массив (ключ может быть только типом, поддерживающим вычисление hash-ей, а значением – любой тип), кортеж – неизменяемый список с возможностью автоматической распаковки/запаковки в переменные (tuple()) и коллекция-множество из python3 (set()). При работе с этими коллекциями можно применять синтаксис списковых включений (к примеру, [I for I in range(10)]), что позволяет создавать заполненные структуры данных за одну строку кода. Литералы коллекций были бы сложны при реализации парсера, если бы не другая особенность языка – выделение блоков кода пробельными отступами.

В-четвертых, в python реализована так называемая «утиная типизация» (duck typing). Это означает отсутствие проверок типа при вычислении операторов “.”, “[]”, “()”. Методы объектов разрешаются на этапе выполнения по правилам, которые могут быть переопределены программистом и, если метод у объекта найден, то программа просто работает.

Доступные в языке грамматические конструкции сведены в таблицу 1.

|  |  |  |  |
| --- | --- | --- | --- |
| **Название** | **Пример** | **Парадигма** | **Комментарий** |
| Присваивание | a, b = 2, 5 | Императивная | Выполняется проверка соответствия типов (lvalue и rvalue) |
| Цикл по коллекции | for I in [1,2,3,4]: | Императивная | Также можно итерироваться по пользовательским объектам, и сопрограммам. |
| Безопасный импорт | import A.B as C from A import B as C | - | Аналог есть в языках D и Haskell |
| Вызов методов у литералов | “/”.join(‘~’, ‘name’)  “1 2 3 4”.split() | ООП | Благодаря тому, что все является объектом |
| Вложенные функции | def main():  def nested(): | Функциональная | Внешний контекст попадает в замыкание функции по ссылкам. |
| Лямбда-функции | (lambda x: x\*x)(5) | Функциональная | Всегда однострочные |
| Декораторы функций | @A.dec  def main(): | Функциональная | Декоратор выполняет добавочные действия при вызове функции |
| Генераторы | def main():  yield 5 | Функциональная | Может использоваться в for. После выхода из функции сохраняется её контекст. |
| with-выражения | with open(‘file’, ‘w’): | ООП | Выполняет добавочные действия при входе в блок и на выходе |
| Ключевые параметры | def main(arg):  pass  main(arg=[2,3]) | Императивная | Связывает конкретный аргумент с фактическим значением |
| Запаковка аргументов | def main(\*args, \*\*kwargs): | Императивная | Возможна как при вызове, так и при определении функции |
| Множественное наследование | class A(B, C): | ООП | Благодаря duck-typing можно применять mixin-ы (примеси). |
| Обработка исключений | try:  except Error as e: | ООП | Любое исключение – это объект-наследник Exception |
| Срез массива | data[3:-1] | Императивный | Взять часть массива как новый массив с помощью :. Допустимы отрицательные индексы. |
| Завершающая запятая | a = {1,2,3,} | - | Во всех перечислениях через запятую |
| Инстанцирование класса | a = Class() | ООП | Без new. Просто круглые скобки |

Таблица 1. Грамматические конструкции языка python

В-пятых, для интеграции пользовательских типов в грамматические конструкции языка используются специальные методы с двумя подчеркивания в начале и в конце названия. Основные это: \_\_init\_\_ для конструктора класса, \_\_call\_\_ для создания вызываемого объекта, \_\_add\_\_ для возможности выполнять сложение объектов оператором “+” и \_\_iter\_\_ для возможности итерироваться по объекту в цикле. Всего таких методов около двух десятков.

В-шестых, в python не реализован ни полиморфизм ни полноценная инкапсуляция. Первое связано с динамической типизацией и философией языка, а второе частично реализовано с помощью соглашений о наименовании методов.

На настоящий момент язык программирования python представлен в двух версиях – 2 и 3. Третий python более новый, исправляет архитектурные ошибки второй версии (к примеру, убраны классы старого типа, сделаны проверки типов в операторах сравнения), все строки сделаны unicode-ными и добавлено много новых и спорных грамматических конструкций (к примеру, ключевые слова async и await, оператор @ для матриц, аннотации типов и другие).

2.2. Описание грамматики и возможностей MiniPython

Язык MiniPython строился как подмножество языка Python версии 3.6, с тем чтобы убрать оттуда редко используемые и спорные конструкции, а существующие упростить с сохранением их основных возможностей. Язык остаётся полностью объектным, с возможностями как процедурного так и функционального программирования. В качестве бэкенд-части компилятора предполагается использование виртуальной машины llvm.

Сохранены функции, классы, декораторы, циклы, условия, литералы для списков, словарей и кортежей, а также списковые включения. Корректно обрабатываются отступы и пространства имен. Убраны генераторы, лямбда и with выражения, обработка исключений, импорты с переименованием как редко используемые и легко заменяемые традиционными процедурными средствами. Упрощена грамматика для аргументов функций, циклов, срезов массивов. В полном составе выкинуты бинарные операции.

Общее количество правил грамматики сократилось с 86 [3] у python3.6 до 45 в MiniPython. Сама грамматика представлена ниже.

// стартовое правило грамматики

file\_input : ( NEWLINE | stmt )\* EOF ;

// применение декораторовdecorator : **'@'** dotted\_name ( **'('** arg\_list **')'** )? NEWLINE ;  
decorators : decorator+ ;  
decorated : decorators ( class\_def | func\_def ) ;

// объявление функций  
func\_def : DEF NAME **'('** var\_arg\_list? **')' ':'** suite ;  
var\_arg\_list  
 : vfpdef ( **','** vfpdef )\* ( **','** vfpdef **'='** test )\* (**',' '\*'** vfpdef )? (**',' '\*\*'** vfpdef )?  
 | vfpdef **'='** test ( **','** vfpdef **'='** test )\* (**',' '\*'** vfpdef )? ( **',' '\*\*'** vfpdef )?  
 | **'\*'** vfpdef ( **',' '\*\*'** vfpdef )?  
 | **'\*\*'** vfpdef ;  
vfpdef : NAME ;  
  
stmt : simple\_stmt | compound\_stmt ;  
simple\_stmt : ( expr\_stmt | flow\_stmt ) NEWLINE ;

// выражения с присваиванием  
expr\_stmt : testlist\_star\_expr ( **'='** test | aug\_assign test\_list | ( **'='** testlist\_star\_expr )\* ) ;  
testlist\_star\_expr : ( test | expr ) ( **','** ( test | expr ) )\* **','**? ;  
aug\_assign : ADD\_ASSIGN | SUB\_ASSIGN | MULT\_ASSIGN | DIV\_ASSIGN | MOD\_ASSIGN ;

// операторы управления потоком исполнения  
flow\_stmt : PASS | BREAK | CONTINUE | RETURN test\_list? ;

// составные выражения (условия, циклы, функции, классы, задекорированные)  
compound\_stmt : if\_stmt | while\_stmt | for\_stmt | func\_def | class\_def | decorated ;  
if\_stmt : IF test **':'** suite ( ELIF test **':'** suite )\* ( ELSE **':'** suite )? ;  
while\_stmt : WHILE test **':'** suite ;  
for\_stmt : FOR expr\_list IN test\_list **':'** suite ;

// Блок кода  
suite : simple\_stmt | NEWLINE INDENT stmt+ DEDENT ;

// логические операторы для предикатов  
test : or\_test ( IF or\_test ELSE test )? ;  
or\_test : and\_test ( OR and\_test )\* ;  
and\_test : not\_test ( AND not\_test )\* ;  
not\_test : NOT not\_test | comparison ;

// операторы сравнения  
comparison : expr ( comp\_op expr )\* ;  
comp\_op : LESS\_THAN | GREATER\_THAN | EQUALS | NOT\_EQ | GT\_EQ | LT\_EQ | IN | NOT IN | IS | IS NOT ;

// математические операции (по приоритетам)  
expr : term ( ( **'+'** | **'-'** ) term )\* ;  
term : factor ( ( **'\*'** | **'/'** | **'%'** ) factor )\* ;  
factor : ( **'+'** | **'-'** ) factor | atom trailer\* ;

// то что может идти после литерала (функциональный вызов, индексирование или обращение по точке)  
trailer : **'('** arg\_list? **')'** | **'['** subscript\_list **']'** | **'.'** NAME ;  
subscript\_list : test | test? **':'** test? ;

// литералы  
atom : NAME | string+ | number | NONE | TRUE | FALSE | **'('** list\_compr? **')'** | **'['** list\_compr? **']'** | **'{'** dict\_compr? **'}'** ;  
list\_compr : test ( comp\_for | (**','** test)\* **','**? ) ;  
dict\_compr : test **':'** test ( comp\_for | **','** test **':'** test )\* **','**? ;  
expr\_list : expr ( **','** expr )\* **','**? ;  
test\_list : test ( **','** test )\* **','**? ;

// объявления классов  
class\_def : CLASS NAME ( **'('** arg\_list **')'** )? **':'** suite ;  
arg\_list : argument ( **','** argument )\* **','**? ;  
argument : test comp\_for? | test **'='** test | **'\*'** test | **'\*\*'** test ;

// списковые включения  
comp\_iter : comp\_for | comp\_if ;  
comp\_for : FOR expr\_list IN or\_test comp\_iter? ;  
comp\_if : IF or\_test comp\_iter? ;  
  
dotted\_name : NAME ( **'.'** NAME )\* ;  
string : STRING ;  
number : INTEGER | FLOAT ;

Данная грамматика в дальнейшем будет использована для генерации парсера в программе antlr [5, 7]. Она записана в расширенной нотации Бэкуса-Наура, с заменой оператора ::= на более простой :.

Для того, чтобы сгенерированные лексер и парсер могли работать с исходным кодом программы на Python и корректно обрабатывать отступы и вложенность блоков кода, пришлось сделать достаточно сложную инъекцию логики в самой грамматике [10]. Был переопределен метод nextToken() лексера так, чтобы отлавливать пробельные символы и обрабатывать их, генерируя специальные виды токенов INDENT и DEDENT. При этом, текущая вложенность накапливается в локальный массив и сбрасывается при выходе из блока. Учитывается также и то, что лексер во время разбора может находиться внутри литерала коллекции и тогда отступы не влияют на генерацию спецтокенов. Затем парсер, работает как обычно, но учитывая токены вложенности, которые также присутствуют в грамматике при описании правила suite.

# Конструкторский раздел

3.1. Antlr и генерация лексера и парсера

Для построения лексера и парсера для фронтенд-части компилятора в этой работе используется инструмент под названием Antlr (версия 4) [7]. Antlr – это генератор нисходящих анализаторов, и он преобразует контекстно-свободную грамматику в программу на С++, Java, C#, Python или Ruby. Antlr является свободным программным обеспечением.

В качестве языка реализации компилятора был также выбран язык программирования Python3. Это не вносит путаницу и ускоряет разработку. К тому же, antlr без проблем работает с ним. Для запуска генерации используется следющая команда:

antlr4 –Dlanguage=Python3 MiniPython.g4 –o ../antlr\_out (1)

, где флаг –o используется для указания папки вывода antlr-а, а MiniPython.g4 это файл с грамматикой из раздела 2.2.

В результате выполнения команды (1), в папке antlr\_out появляются три python модуля. Один для лексера, который будет в соответствии с терминальными правилами делить входные данные на токены, один для парсера, который построит из цепочки токенов лексера абстрактное синтаксическое дерево и один модуль для класса-обходчика этого AST.

Для того чтобы получить из текста программы на языке MiniPython абстрактное синтаксическое дерево, необходимо выполнить следующий код:

// необходимые импорты

**from** antlr4 **import** \*

**from** antlr\_out.MiniPythonLexer **import** MiniPythonLexer  
**from** antlr\_out.MiniPythonParser **import** MiniPythonParser

// метод для создания инстанса парсера

**def** get\_antlr\_parser(input\_stream):  
 lexer = MiniPythonLexer(input\_stream)  
 *# tokens = lexer.getAllTokens()*

*// передаем в парсер выход лексера* stream = CommonTokenStream(lexer)  
 parser = MiniPythonParser(stream)  
 **return** parser

// передаем в качестве входного потока FileStream из antlr4

parser = get\_antlr\_parser(FileStream(file\_name, encoding=**'utf-8'**))

// file\_input – это указание стартового правила грамматики  
tree = parser.file\_input()

В результате выполнения в переменной tree появится построенное AST-дерево. Для того чтобы выполнить по нему обход и подготовить его к генерации кода виртуальной машины, используется класс listener-а. Этот класс также генерируется antlr-ом по грамматике и должен быть унаследован с расширением указанных методов:

*# Enter a parse tree produced by MiniPythonParser#file\_input.***def** enterFile\_input(self, ctx:MiniPythonParser.File\_inputContext):  
 **pass***# Exit a parse tree produced by MiniPythonParser#file\_input.***def** exitFile\_input(self, ctx:MiniPythonParser.File\_inputContext):  
 **pass**

Каждое правило превратилось здесь в пару методов – действие при входе в узел, помеченный левой частью правила, и действие при выходе из узла. Обход происходит в следующем порядке, сначала посещается сам узел и вызывается метод enter, затем все его потомки, а потом вызывается метод exit. Такой подход соответствует паттерну посетитель (visitor) из GoF.

В случае, если входной текст программы не удовлетворяет грамматике MiniPython, парсер, сгенерированный Antlr-ом будет возвращать ошибки вида: «В позиции X ожидался терминал Y».

3.2. Семантический анализ

Для генерации кода llvm недостаточно одного дерева синтаксического разбора. Для начала, в llvm необходимо знать типы переменных и функций, а значит необходимо по дереву определить, какие операции будут выполняться и над какими данными и составить таблицы типов. Также необходимо преобразовать поток исполнения и грамматические конструкции, которых нет в llvm (например, классы, генераторы, вложенные функции) в некоторые программные аналоги. К сожалению, AST не очень для этого приспособлено.

AST – это специальное представление программы. Как пример можно привести дерево, получаемое парсером из выражения a += 1. Отступы обозначают вложенность. Листья всегда терминалы, а остальные вершины – левые части правил грамматики.

<file\_input>  
 <stmt>  
 <simple\_stmt>  
 <expr\_stmt>  
 <testlist\_star\_expr>  
 <test>  
 <or\_test>  
 <and\_test>  
 <not\_test>  
 <comparison>  
 <expr>  
 <term>  
 <factor>  
 <atom>  
 'a'  
 <aug\_assign>  
 '+='  
 <test\_list>  
 <test>  
 <or\_test>  
 <and\_test>  
 <not\_test>  
 <comparison>  
 <expr>  
 <term>  
 <factor>  
 <atom>  
 <number>  
 '1'  
 <newline>  
 '<EOF>'

Видно, что дерево является сильно избыточным, и для анализа программы будет лучше перестроить его в более компактное и удобное семантическое дерево. Для данного выражения оно будет сгенерировано как:

<global>  
 <aug assign: '+='>  
 <var: 'a'>  
 <const: '1'>

В данном примере каждая вершина дерева является вычисляемым выражением. Слева от двоеточия записан тип узла, а справа аргументы для вычисления. Для переменной это её имя, а для константы её представление.

Алгоритм выведения типов обходит дерево несколько раз. Сначала типы присваиваются константам, затем переменным и функциям. После того как известны типы простых выражений, алгоритм вычисляет сколько и какой памяти должно быть выделено под сложные структуры данных: списки, словари и кортежи.

Виды узлов в семантическом дереве сведены в таблицу 2.

|  |  |  |
| --- | --- | --- |
| **Название** | **Пример** | **Окружение** |
| Глобальный контекст | <global> | Всегда корень дерева |
| Объявление функции или метода | <function: name> | Внутри параметры и затем <block> |
| Обычный параметр функции | <input: name> |  |
| Параметр функции с значением по умолчанию | <default: name> |  |
| Условие | <if> | Внутри пары: вычисляемый предикат + <block>. Может встречаться внутри <list>, <dict>. |
| Цикл с условием | <while> | Предикат и затем <block> |
| Цикл по коллекции | <for> | Курсор и коллекция как вычисляемые выражения. Затем <block>. |
| Управление выполнением | <flow: action> | Может быть return, break, pass или continue |
| Класс | <class: name> | Сначала вычисляемые выражения – родители. Затем <block>. |
| Декорирование функции | <decorator: name> | Встречается перед функцией или классом. |
| Логические операции | <and>, <not>, <or>, <comp: op> | Вкладываются друг в друга. Все кроме not бинарные. |
| Математические операции | <additive: op>,  <multiply: op>,  <unary: op> | Обладают различным приоритетом и арностью. |
| Переменная и константа | <var: name>,  <const: literal> | Используются для выведения типов. Среди констант есть True, False и None |
| Составные структуры | <list>, <dict>, <tuple> | Содержат внутри множество вычисляемых элементов. Размер задаётся при выполнении. В dict пары. |
| Функциональный вызов | <call> | Внутри callable выражения. Содержит <param: name> |
| Разрешение имени | <resolve: name> | name – это правая часть после точки |
| Индексирование и срез | <index: op> | Вызов, разрешение имени и индексирование могут идти последовательно, либо вкладываться. |

Таблица 2. Типы узлов семантического дерева.

# Технический раздел

4.1. Описание llvm и предпосылки кодогенерации

LLVM (low-level virtual machine) – это универсальная система анализа, трансформации и оптимизации программ или, как её называют разработчики, «инфраструктура компиляции». LLVM используется как бэкенд-часть компиляторов для многих языков программирования: Си и С++ вместе с фронтендом clang, для Haskell в GHC, Java, Ruby, JavaScript и на нём также основана графическая подсистема OpenGL в Mac OSX. Виртуальная машина написана на С++ и портирована на большинство \*nix систем и windows. Система имеет модульную структуру и может расширяться дополнительными алгоритмами.

В основе работы LLVM лежит понятие промежуточного кода – IR (intermediate representation) [5]. По сути это типизированный трёхадресный код в SSA (static single assignment) форме записи. На практике для хранения кода используется эффективное бинарное представление (bitcode), а генерировать его удобнее всего не в текстовой форме, а с помощью специального API. Для python были разработаны такие библиотеки как llvmpy (реализована поверх API LLVM языка Си и более не поддерживается) и производная от неё llvmlite [4]. Последнюю и предполагается использовать в данной работе.

Llvmlite предоставляет python API для создания в памяти структур данных, которые затем будут использоваться при кодогенерации в llvm IR. Библиотека генерирует IR код напрямую, без вызова дополнительных утилит. Она даёт возможность управлять функциями и их вызовами, созданием констант, переменных и указателей, а также агрегирующих СД по типу Си-структур и массивов. Также среди возможностей библиотеки: условия, работа с памятью, операции над числами и приведение типов, goto и phi-функции. Соответственно, все возможности, которых в ней нет (начиная с циклов и списков, и, заканчивая классами и декораторами) необходимо будет эмулировать.

4.2. Предложения по разработке бэкенд-части компилятора

Для конструкций MiniPython, которых нет в коде промежуточного представления IR, нужно описать их замену на имеющиеся.

Во-первых, это циклы с условием, помечаемые блоком <while> и сводимые к ним циклы для обхода коллекции – <for>. Их предполагается заменять на условие для вычисления предиката и два безусловных перехода goto – для выхода из цикла и следующей итерации. Блоки <continue> и <break> внутри этих циклов заменяются на безусловные переходы. Для того чтобы соответствовать форме SSA должны использоваться phi-функции.

Во-вторых, это вложенные функции (nested-functions), для которых должно формироваться замыкание внешнего контекста. В коде llvm для каждой такой функции выделяется отдельный блок памяти для хранения указателей на переменные контекста. Функции становятся глобальными и обращаются к ней. Для предотвращения конфликтов имен, происходит переименование, зависящее от той области видимости, где функция была определена. Классы, создающие новые пользовательские типы, реализуются похожим образом, блок памяти для атрибутов класса и все методы переименовываются в соответствии с областью видимости. Однако, для объектов, выделяются отдельные блоки памяти (везде, где происходит инстанцирование) для хранения их атрибутов.

В-третьих, специфичная для python конструкция декоратора. Декорирование говорит лишь о том, что вместо указанной функции нужно создать новую, с вызовом тела декоратора до выполнения основного тела.

Отдельно стоит отметить приведение структур данных. Их реализуют отдельно с помощью массивов и указателей в коде llvm. Например, список и кортеж – как массивы указателей на элементы, словарь и множество как сбалансированные деревья поиска (для производительности).

4.3. Пример программы на MiniPython

Как пример приведем программу вычисления факториалов первых десяти целых чисел написанную на языке MiniPython.

// Файл исходных текстов fact.ex

def factorial(n):  
 if n == 0:  
 return 1  
 return n \* factorial(n - 1)  
  
def main():  
 print('Вычисление факториалов')  
 for i in range(10):  
 print('Факториал ', i, ' = ', factorial(i))  
  
main()

В исходном тексте программы представлены две функции: main с циклом по целым числам и вызовом функции factorial, которая внутри себя содержит рекурсивный вызов и условие останова рекурсии. Пропустив этот файл через разработанную программу получим синтаксическое и семантическое деревья.

В синтаксическом дереве программы содержится 306 нетерминальных и терминальных узлов. Из них 11 <stmt>, 5 <suite>, 15 <test>, 19 <atom>. Семантическое дерево в свою очередь содержит лишь 51 узел (в шесть раз меньше) и представлено в следующем листинге.

<global>  
 <function: 'factorial'>  
 <input: 'n'>  
 <block>  
 <if>  
 <comp: '=='>  
 <var: 'n'>  
 <const: '0'>  
 <block>  
 <flow: 'return'>  
 <const: '1'>  
 <flow: 'return'>  
 <multiply: '\*'>  
 <var: 'n'>  
 <var: 'factorial'>  
 <call>  
 <param>  
 <additive: '-'>  
 <var: 'n'>  
 <const: '1'>  
 <function: 'main'>  
 <block>  
 <var: 'print'>  
 <call>  
 <param>  
 <const: ''Вычисление факториалов''>  
 <for>  
 <var: 'i'>  
 <var: 'range'>  
 <call>  
 <param>  
 <const: '10'>  
 <block>  
 <var: 'print'>  
 <call>  
 <param>  
 <const: ''Факториал ''>  
 <param>  
 <var: 'i'>  
 <param>  
 <const: '' = ''>  
 <param>  
 <var: 'factorial'>  
 <call>  
 <param>  
 <var: 'i'>  
 <var: 'main'>  
 <call>

По представленному дереву происходит выведение типов. Сначала типы присваиваются константам (целочисленные, логические, строки, None). Затем, в соответствии с присваиваниями и связыванием параметров в функциональных вызовах, присваиваются типы переменным и параметрам функций. И последним шагом, анализируются узлы <return> внутри функций, чтобы определить их возвращаемое значение. В данной программе: переменная i и параметр n являются целочисленными, а тип функций factorial – int (int), main – None ().

После выполнения анализа типов начинается собственно кодогенерация в IR код LLVM, который для указанной программы примет следующий вид:

// Секция строковых констант

@.str = private unnamed\_addr constant [44 x i8] c"\D0\92\D1\8B\D1\87\D0\B8\D1\81\D0\BB\D0\B5\D0\BD\D0\B8\D0\B5 \D1\84\D0\B0\D0\BA\D1\82\D0\BE\D1\80\D0\B8\D0\B0\D0\BB\D0\BE\D0\B2\00", align 1

@.str.1 = private unnamed\_addr constant [20 x i8] c"\D0\A4\D0\B0\D0\BA\D1\82\D0\BE\D1\80\D0\B8\D0\B0\D0\BB \00", align 1

@.str.2 = private unnamed\_addr constant [4 x i8] c" = \00", align 1

// Функция factorial

; Function Attrs: nounwind

define i32 @factorial(i32 %n) #0 {

%retval = alloca i32, align 4

%n.addr = alloca i32, align 4

store i32 %n, i32\* %n.addr, align 4

%1 = load i32, i32\* %n.addr, align 4

%cmp = icmp eq i32 %1, 0

br i1 %cmp, label %2, label %3

; <label>:2: ; preds = %0

store i32 1, i32\* %retval, align 4

br label %6

; <label>:3: ; preds = %0

%4 = load i32, i32\* %n.addr, align 4

%5 = load i32, i32\* %n.addr, align 4

%sub = sub nsw i32 %5, 1

%call = call i32 @factorial(i32 %sub)

%mul = mul nsw i32 %4, %call

store i32 %mul, i32\* %retval, align 4

br label %6

; <label>:6: ; preds = %3, %2

%7 = load i32, i32\* %retval, align 4

ret i32 %7

}

// Функция main

; Function Attrs: nounwind

define i32 @main() #0 {

%retval = alloca i32, align 4

%i = alloca i32, align 4

store i32 0, i32\* %retval, align 4

call void (i8\*, ...) @print(i8\* getelementptr inbounds ([44 x i8], [44 x i8]\* @.str, i32 0, i32 0))

store i32 0, i32\* %i, align 4

br label %1

; <label>:1: ; preds = %6, %0

%2 = load i32, i32\* %i, align 4

%cmp = icmp slt i32 %2, 10

br i1 %cmp, label %3, label %8

; <label>:3: ; preds = %1

%4 = load i32, i32\* %i, align 4

%5 = load i32, i32\* %i, align 4

%call = call i32 @factorial(i32 %5)

call void (i8\*, ...) @print(i8\* getelementptr inbounds ([20 x i8], [20 x i8]\* @.str.1, i32 0, i32 0), i32 %4, i8\* getelementptr inbounds ([4 x i8], [4 x i8]\* @.str.2, i32 0, i32 0), i32 %call)

br label %6

; <label>:6: ; preds = %3

%7 = load i32, i32\* %i, align 4

%inc = add nsw i32 %7, 1

store i32 %inc, i32\* %i, align 4

br label %1

; <label>:8: ; preds = %1

ret i32 0

}

# Заключение

В рамках проведенной работы была описана грамматика языка программирования MiniPython с 45 правилами, построенными по образу грамматики языка Python версии 3.6. MiniPython является полноценным языком с математическими выражениями, удобными структурами данных Python, ООП (классы, наследование) и функциональными возможностями (декораторы, списковые включения).

С помощью инструмента Antlr 4 были построены модули для лексического и синтаксического анализа исходных текстов программ на языке MiniPython. Их запуск для подготовленных примеров fact.ex (демонстрация математических вычислений), logger.ex (демонстрация функциональных возможностей), web\_view.ex (демонстрация ООП-возможностей) дал синтаксические деревья разбора. В дальнейшем при работе с ними разработанная программа строит семантическое дерево и производит выведение типов. В дальнейшем эту информацию можно будет использовать для кодогенерации в промежуточный код виртуальной машины llvm.

Также в данной работе выработаны предложения по разработке бэкенда-компилятора MiniPython, который будет состоять из преобразований потоков исполнения и структур данных и вызовов API llvmlite. Приведены примеры сгенерированного IR llvm.

# Список источников

1. Альфред В. Ахо, Моника С. Лам, Рави Сети, Джеффри Д. Ульман. Компиляторы: принципы, технологии и инструментарий = Compilers: Principles, Techniques, and Tools. — 2 изд. — М.: [Вильямс](https://ru.wikipedia.org/w/index.php?title=%D0%92%D0%B8%D0%BB%D1%8C%D1%8F%D0%BC%D1%81_(%D0%B8%D0%B7%D0%B4%D0%B0%D1%82%D0%B5%D0%BB%D1%8C%D1%81%D1%82%D0%B2%D0%BE)&action=edit&redlink=1), 2008. — [ISBN 978-5-8459-1349-4](https://ru.wikipedia.org/wiki/%D0%A1%D0%BB%D1%83%D0%B6%D0%B5%D0%B1%D0%BD%D0%B0%D1%8F:%D0%98%D1%81%D1%82%D0%BE%D1%87%D0%BD%D0%B8%D0%BA%D0%B8_%D0%BA%D0%BD%D0%B8%D0%B3/9785845913494).
2. Дэвид М. Бизли. [Python. Подробный справочник, 4-е издание](http://www.symbol.ru/date/784182.html). — Перевод с английского. — СПб.: [Символ-Плюс](https://ru.wikipedia.org/w/index.php?title=%D0%A1%D0%B8%D0%BC%D0%B2%D0%BE%D0%BB-%D0%9F%D0%BB%D1%8E%D1%81&action=edit&redlink=1), 2010. — 864 с — [ISBN 978-5-93286-157-8](https://ru.wikipedia.org/wiki/%D0%A1%D0%BB%D1%83%D0%B6%D0%B5%D0%B1%D0%BD%D0%B0%D1%8F:%D0%98%D1%81%D1%82%D0%BE%D1%87%D0%BD%D0%B8%D0%BA%D0%B8_%D0%BA%D0%BD%D0%B8%D0%B3/9785932861578)
3. Грамматика Python 3. [Электронный ресурс] Режим доступа: URL: <https://docs.python.org/3/reference/grammar.html>.
4. Документация llvmlite. [Электронный ресурс] Режим доступа: URL: <http://llvmlite.pydata.org/en/latest/index.html>.
5. Документация llvm IR. [Электронный ресурс] Режим доступа: URL: <http://releases.llvm.org/3.8.0/docs/LangRef.html>.
6. Документация pyston. [Электронный ресурс] Режим доступа: URL: <https://github.com/dropbox/pyston/tree/master/docs>.
7. Официальный сайт Antlr. [Электронный ресурс] Режим доступа: URL: <http://www.antlr.org/download.html>.
8. Обзор LLVM. [Электронный ресурс] Режим доступа: URL: <https://habrahabr.ru/post/47878/>.
9. Официальный сайт Numba. [Электронный ресурс] Режим доступа: URL: <http://numba.pydata.org/>.
10. Примеры грамматик Antlr. [Электронный ресурс] Режим доступа: URL: <https://github.com/antlr/grammars-v4>.