Министерство образования и науки Российской Федерации

Федеральное государственное бюджетное образовательное учреждениевысшего образования

«Московский государственный технический университетимени Н.Э. Баумана (национальный исследовательский университет)»(МГТУ им. Н.Э. Баумана)

ФАКУЛЬТЕТ «Информатика и системы управления»

КАФЕДРА «Теоретическая информатика и компьютерные технологии»

РАСЧЕТНО-ПОЯСНИТЕЛЬНАЯ ЗАПИСКА К КУРСОВОЙ РАБОТЕ

ПО КУРСУ: «Компилятор»

НА ТЕМУ:

«Компилятор для языка Python с поддержкой инфраструктуры LLVM**»**

Студент ИУ9-52Б **Нгуен Т.Д** _____ _____

(Ф.И.О) (Подпись, дата)

Руководитель курсового проекта **Синявин А. В.** _____

(Ф.И.О) (Подпись, дата)

2021 г.

Министерство науки и высшего образования Российской Федерации

Федеральное государственное бюджетное образовательное учреждение высшего образования
«Московский государственный технический университет имени Н.Э.

Баумана»(национальный исследовательский университет)

УТВЕРЖДАЮ

Заведующий кафедрой ИУ-9
(Индекс)

_____И.П. Иванов (И.О.Фамилия)

«____»_____2021 г.

З А Д А Н И Е

на выполнение курсовой работы

по дисциплине : «Компилятор»

Тема курсовой работы: Компилятор для языка Python с поддержкой инфраструктуры LLVM

Студент : Нгуен Тхань Дат ИУ9-72б

График выполнения проекта: 25% к 4 нед., 50% к 8 нед., 75% к 11 нед., 100% к 14 нед.

1. Техническое задание

Этап 1: Разбор программы Python на абстрактное синтаксическое дерево

Этап 2: Преобразование дерева синтаксического анализа в язык программирования низкого уровня LLVM

Этап 3: Реализовать метод оптимизации для ускорения процесса компиляции

Этап 4: Выполнение низкоуровневой программы с поддержкой бэкэнд-методов LLVM

2. Оформление курсовой работы

2.1. Расчетно-пояснительная записка на 25 листах формата А4.

2.2. Перечень графического материала (плакаты, схемы, чертежи и т.п.) ____

Дата выдачи задания «___»_____2021 г.

I. Введение

1. Компиляторы и интерпретаторы программы
2. Язык программирования Python
3. LLVM

II. Синтаксический анализатор

1. Грамматика
2. Лексер
3. Синтаксический анализатор

III. Промежуточное представление кода

1. LLVM промежуточное представление – LLVM IR
2. Структура ассемблера LLMV IR
3. Реализация

IV. LLVM JIT и поддержка оптимизатора

1. Оптимизация LLVM
2. Добавление JIT-компилятора

V. Заключение

1. Результат сравнения между JIT-компилятором LLVM и интерпретатором python по умолчанию
2. Будущая работа

VI. Список литературы

I. Введение

1. Компиляторы и интерпретаторы программы

- Это программы, которые помогают преобразовывать язык высокого уровня в машинные коды, понятные компьютерам. Язык высокого уровня - это язык, понятный людям. Однако компьютеры не могут понимать языки высокого уровня, как мы, люди. Они могут понимать только программы, разработанные в двоичных системах, известных как машинный код. Начнем с того, что компьютерная программа обычно пишется на языке высокого уровня, который описывается как исходный код. Эти исходные коды должны быть преобразованы в машинный язык, и здесь появляется роль компиляторов и интерпретаторов.
- Некоторые различия между "Интерпретатором" и "Компилятором"

Интерпретаторы	Компиляторы
Интерпретатор переводит только один оператор программы за раз в машинный код.	Компилятор сканирует всю программу и сразу переводит ее в машинный код.
Интерпретатору требуется гораздо меньше времени для анализа исходного кода. Однако общее время выполнения процесса намного меньше.	Компилятору требуется много времени для анализа исходного кода. Однако общее время, необходимое для выполнения процесса, намного быстрее.
Интерпретатор не генерирует промежуточный код. Следовательно, интерпретатор очень эффективен с точки зрения своей памяти.	Компилятор всегда генерирует промежуточный объектный код. Потребуются дополнительные ссылки. Следовательно, требуется больше памяти.
Продолжает переводить программу непрерывно, пока не будет обнаружена первая ошибка. Если обнаружена какая-либо ошибка, она перестает работать, и, следовательно, отладка становится легкой.	Компилятор генерирует сообщение об ошибке только после того, как он сканирует всю программу, и, следовательно, отладка относительно сложнее при работе с компилятором.

2. Язык программирования Python

- Python — высокоуровневый язык программирования общего назначения с динамической строгой типизацией и автоматическим управлением памятьюориентированный на повышение производительности разработчика, читаемости кода и его качества, а также на обеспечение переносимости написанных на нём программ. Язык является полностью объектно-ориентированным в том плане, что всё является объектами

- Python стал одним из самых быстрорастущих и популярных языков программирования в мире. Он универсален, его легко использовать и развивать.

- Но нет ничего идеального, Python вовсе не исключение. Python - это интерпретируемый язык. Поэтому ограничение скорости выполнения - по сравнению с компилируемыми языками, такими как C ++, Java, ... - является одним из основных недостатков при использовании Python.

- Чтобы справиться с недостатками Python, было создано множество его реализаций. Они определены как интерпретатор и компилятор, поскольку он компилирует код Python в байт-код перед его интерпретацией. Такие как Cython, Jython, PyPy ...

- Этот проект представляет собой реализацию языка программирования Python. Он преобразует язык Python в язык программирования низкого уровня, а затем выполняет его с помощью JIT-компилятора. Все функции, поддерживаемые инфраструктурой LLVM

3. LLVM

- LLVM (ранее Low Level Virtual Machine) — проект программной инфраструктуры для создания компиляторов и сопутствующих им утилит. Состоит из набора компиляторов из языков высокого уровня (так называемых «фронтендов»), системы оптимизации, интерпретации и компиляции в машинный код. В основе инфраструктуры используется RISC-подобная платформонезависимая

система кодирования машинных инструкций (байткод LLVM IR), которая представляет собой высокоуровневый ассемблер, с которым работают различные преобразования.

- В основе LLVM лежит промежуточное представление кода (Intermediate Representation, IR), над которым можно производить трансформации во время компиляции, компоновки и выполнения. Из этого представления генерируется оптимизированный машинный код для целого ряда платформ, как статически, так и динамически (JIT-компиляция). LLVM 9.0.0 поддерживает статическую генерацию кода для x86, x86-64, ARM, PowerPC, SPARC, MIPS, RISC-V, Qualcomm Hexagon, NVPTX, SystemZ, Xcore. JIT-компиляция (генерация машинного кода во время исполнения) поддержана для архитектур x86, x86_64, PowerPC, MIPS, SystemZ, и частично ARM[12].

- LLVM написана на C++ и портирована на большинство Unix-подобных систем и Windows. Система имеет модульную структуру, отдельные её модули могут быть встроены в различные программные комплексы, она может расширяться дополнительными алгоритмами трансформации и кодогенераторами для новых аппаратных платформ. В LLVM включена обёртка API для OCaml.

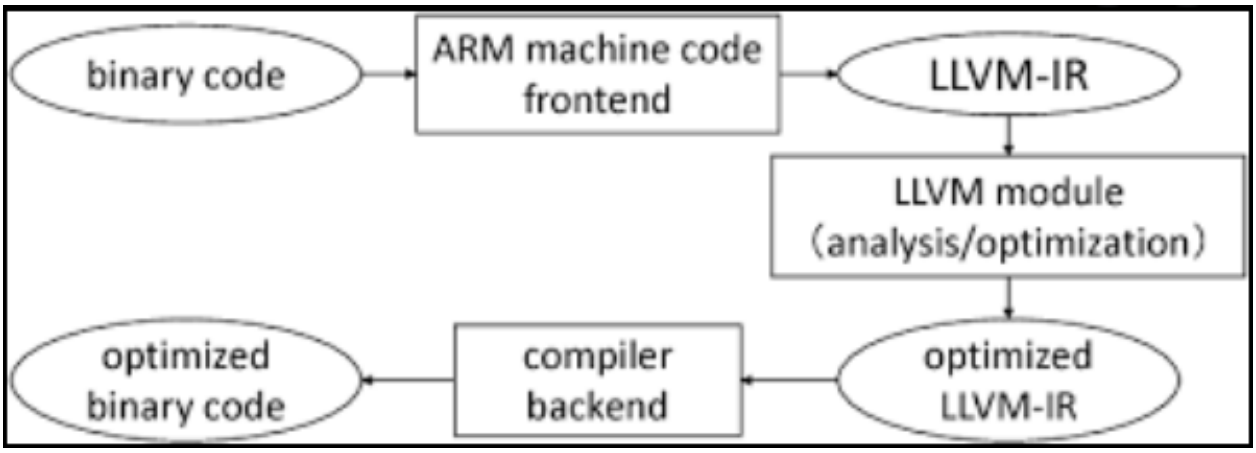

Рисунок 1: Архитектура инфраструктуры LLVM

Инфраструктура LLVM состоит из нескольких основных компонентов.

3.1. Промежуточное представительство – LLVM IR

- Ядро LLVM - это промежуточное представление (IR), язык программирования низкого уровня, похожий на ассемблер. IR - это строго типизированный набор команд вычисления с сокращенным набором команд (RISC), который абстрагирует большинство деталей цели. Например, соглашение о вызовах абстрагируется с помощью инструкций call и ret с явными аргументами. Кроме того, вместо фиксированного набора регистров IR использует бесконечный набор временных файлов вида% 0,% 1 и т. Д.

- LLVM поддерживает три эквивалентных формы IR: удобочитаемый формат сборки, формат в памяти, подходящий для интерфейсы и плотный формат битового кода для сериализации. Пример программы после перевода в IR:

```
@.str = internal constant [14 x i8] c"hello, world\0A\00"

declare i32 @printf(i8*, ...)

define i32 @main(i32 %argc, i8** %argv) nounwind {
entry:
    %tmp1 = getelementptr [14 x i8], [14 x i8]* @.str, i32 0, i32 0
    %tmp2 = call i32 (i8*, ...) @printf( i8* %tmp1 ) nounwind
    ret i32 0
}
```

Рисунок 2: Пример языка ИК-программирования

3.2. График потока управления – CFG

- IR определяет граф потока управления программой. Инструкции IR сгруппированы в помеченные базовые блоки, а метки preds для каждого блока представляют входящие ребра в этот блок. например базовый блок “end:” имеет предшественников “btrue” и “bfalse”:

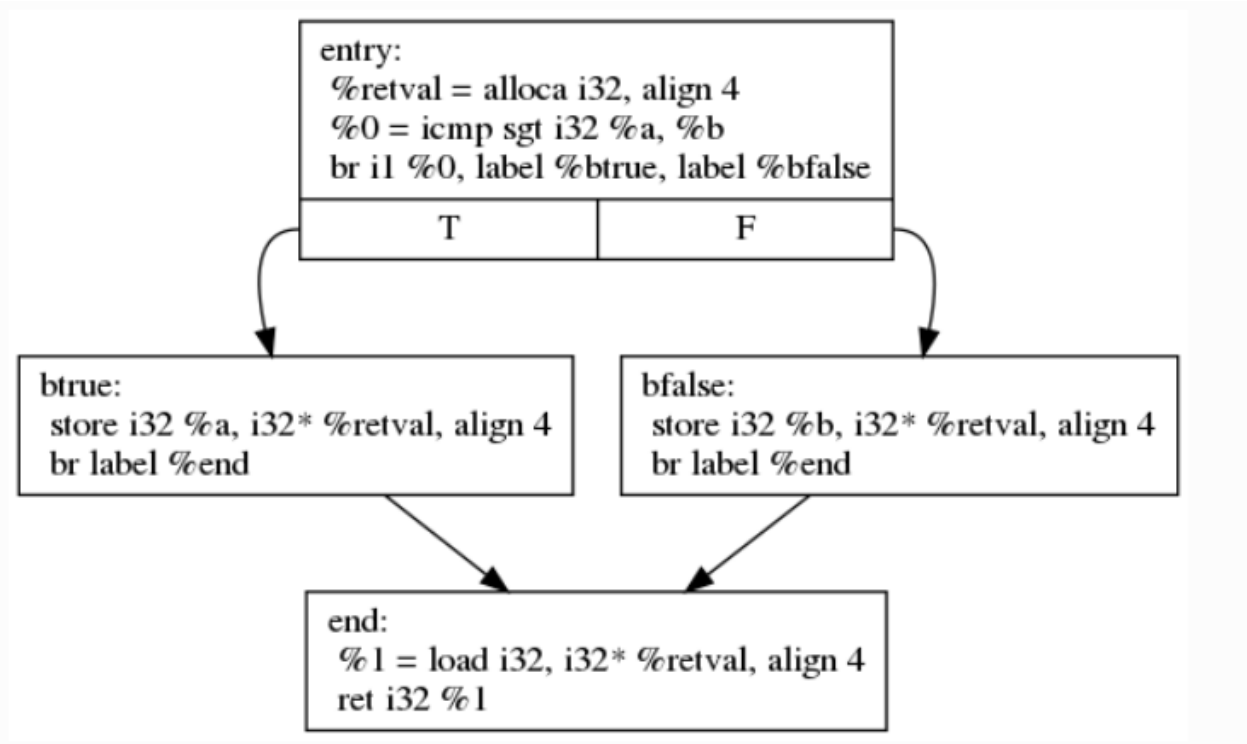

Рисунок 3: Пример графика потока управления

3.3. Фронтенд

- LLVM предоставляет множество функций для создания блоков IR, логики потока управления, управления переменными локатора памяти, ...

```
if op == "#add":
    if left.type == int_type:
        return self.builder.add(left, right)
    else:
        return self.builder.fadd(left,right)
elif op == "#sub":
    if left.type == int_type:
        return self.builder.sub(left, right)
    else:
        return self.builder.fsub(left,right)
elif op == "#mult":
    if left.type == int_type:
        return self.builder.mul(left, right)
    else:
        return self.builder.fmul(left,right)
elif op == "#div":
    if left.type == int_type:
        return self.builder.sdiv(left, right)
    else:
        return self.builder.fdiv(left,right)
elif op == "#mod":
    if left.type == int_type:
        return self.builder.srem(left, right)
```

Рисунок 4: Пример интерфейса LLVM для двоичных операций

3.4. Компилятор LLVM JIT

- К коду, доступному в LLVM IR, можно применять самые разные инструменты. Например, вы можете запустить оптимизацию на нем (как мы сделали выше), вы можете выгрузить его в текстовой или двоичной форме, вы можете скомпилировать код в файл сборки (.s) для некоторой цели или вы можете JIT-скомпилировать его . В представлении LLVM IR приятно то, что это «общая валюта» между многими различными частями компилятора.

3.5. Прохождение оптимизации LLVM

- LLVM предоставляет множество этапов оптимизации, которые выполняют много разных действий и имеют разные компромиссы. В отличие от других систем, LLVM не придерживается ошибочного представления о том, что один набор оптимизаций подходит для всех языков и для всех ситуаций. LLVM позволяет разработчику компилятора принимать полные решения о том, какие оптимизации использовать, в каком порядке и в какой ситуации.

II. Синтаксический анализатор

- На первом этапе процесса компиляции процессор компьютера понял логику программы. Итак, нам нужно разобрать нашу программу.

- Синтакси́ческий ана́лиз в лингвистике и информатике — процесс сопоставления линейной последовательности лексем (слов, токенов) естественного или формального языка с его формальной грамматикой. Результатом обычно является дерево разбора (синтаксическое дерево). Обычно применяется совместно с лексическим анализом.

- Синтаксический анализатор — это программа или часть программы, выполняющая синтаксический анализ. Пример разбора выражения с

преобразованием его структуры из линейной в древовидную.

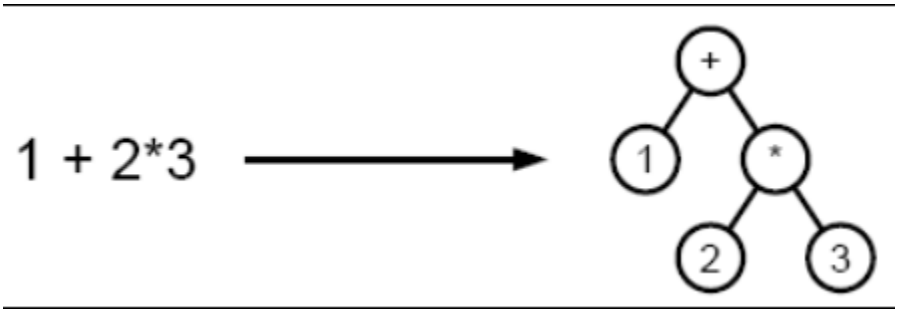

Рисунок 5: Пример дерева синтаксического анализа

- Как правило, результатом синтаксического анализа является синтаксическое строение предложения, представленное либо в виде дерева зависимостей, либо в виде дерева составляющих, либо в виде некоторого сочетания первого и второго способов представления.

- Для анализа любой литературы обязательно определение контекстно-свободной грамматики и линейной последовательности лексем.

1. Грамматика

- Формальная грамматика - это набор правил, синтаксически описывающих язык

- В этом определении есть две важные части: грамматика описывает язык, но это описание относится только к синтаксису языка, а не к семантике. То есть он определяет его структуру, но не его значение. Правильность значения вводимых данных должна быть проверена, при необходимости, каким-либо другим способом.

- В основном при синтаксическом анализе используются два вида грамматик: обычная грамматика и контекстно-свободные грамматики. Обычно какому-то виду грамматики соответствует один и тот же вид языка: обычная грамматика определяет обычный язык и так далее. А контекстно-свободная грамматика больше подходит для описания языка программирования

- Контекстно-свободная грамматика определяется четырьмя кортежами:

$$G = (T, N, S, P)$$

- Где:

  T = набор терминалов (например, токены, возвращенные сканер).

  N = набор нетерминалов (обозначающих структуры внутри язык).

  S = начальный символ (в большинстве случаев наша программа).

  P = набор продуктов (правила, определяющие, как токены организованы в синтаксические единицы).

- Бесконтекстные грамматики хорошо подходят для программирования языков, потому что они ограничивают способ конструкция программирования может быть использована и, таким образом, упростить процесс анализа его использования в программе.

- Абстрактная грамматика для любой типичной программы Python:

```
mod = Module(stmt* body, type_ignore* type_ignores)
    | Interactive(stmt* body)
    | Expression(expr body)
    | FunctionType(expr* argtypes, expr returns)
```

```
expr_context = Load | Store | Del

boolop = And | Or

operator = Add | Sub | Mult | MatMult | Div | Mod | Pow | LShift
             | RShift | BitOr | BitXor | BitAnd | FloorDiv

unaryop = Invert | Not | UAdd | USub

cmpop = Eq | NotEq | Lt | LtE | Gt | GtE | Is | IsNot | In | NotIn
```

```
arguments = (arg* posonlyargs, arg* args, arg? vararg, arg* kwonlyargs,
             expr* kw_defaults, arg? kwarg, expr* defaults)

arg = (identifier arg, expr? annotation, string? type_comment)
       attributes (int lineno, int col_offset, int? end_lineno, int? end_col_offset)
```

```
stmt = FunctionDef(identifier name, arguments args,
                   stmt* body, expr* decorator_list, expr? returns,
                   string? type_comment)
      | AsyncFunctionDef(identifier name, arguments args,
                         stmt* body, expr* decorator_list, expr? returns,
                         string? type_comment)

      | ClassDef(identifier name,
         expr* bases,
         keyword* keywords,
         stmt* body,
         expr* decorator_list)
      | Return(expr? value)

      | Delete(expr* targets)
      | Assign(expr* targets, expr value, string? type_comment)
      | AugAssign(expr target, operator op, expr value)
      -- 'simple' indicates that we annotate simple name without parens
      | AnnAssign(expr target, expr annotation, expr? value, int simple)

      -- use 'orelse' because else is a keyword in target languages
      | For(expr target, expr iter, stmt* body, stmt* orelse, string? type_comment)
      | AsyncFor(expr target, expr iter, stmt* body, stmt* orelse, string? type_comment)
      | While(expr test, stmt* body, stmt* orelse)
      | If(expr test, stmt* body, stmt* orelse)
      | With(withitem* items, stmt* body, string? type_comment)
      | AsyncWith(withitem* items, stmt* body, string? type_comment)

      | Match(expr subject, match_case* cases)

      | Raise(expr? exc, expr? cause)
      | Try(stmt* body, excepthandler* handlers, stmt* orelse, stmt* finalbody)
      | Assert(expr test, expr? msg)

      | Import(alias* names)
      | ImportFrom(identifier? module, alias* names, int? level)

      | Global(identifier* names)
      | Nonlocal(identifier* names)
      | Expr(expr value)
      | Pass | Break | Continue
```

```
expr = BoolOp(boolop op, expr* values)
     | NamedExpr(expr target, expr value)
     | BinOp(expr left, operator op, expr right)
     | UnaryOp(unaryop op, expr operand)
     | Lambda(arguments args, expr body)
     | IfExp(expr test, expr body, expr orelse)
     | Dict(expr* keys, expr* values)
     | Set(expr* elts)
     | ListComp(expr elt, comprehension* generators)
     | SetComp(expr elt, comprehension* generators)
     | DictComp(expr key, expr value, comprehension* generators)
     | GeneratorExp(expr elt, comprehension* generators)
     -- the grammar constrains where yield expressions can occur
     | Await(expr value)
     | Yield(expr? value)
     | YieldFrom(expr value)
     -- need sequences for compare to distinguish between
     -- x < 4 < 3 and (x < 4) < 3
     | Compare(expr left, cmpop* ops, expr* comparators)
     | Call(expr func, expr* args, keyword* keywords)
     | FormattedValue(expr value, int? conversion, expr? format_spec)
     | JoinedStr(expr* values)
     | Constant(constant value, string? kind)

     -- the following expression can appear in assignment context
     | Attribute(expr value, identifier attr, expr_context ctx)
     | Subscript(expr value, expr slice, expr_context ctx)
     | Starred(expr value, expr_context ctx)
     | Name(identifier id, expr_context ctx)
     | List(expr* elts, expr_context ctx)
     | Tuple(expr* elts, expr_context ctx)

     -- can appear only in Subscript
     | Slice(expr? lower, expr? upper, expr? step)
```

Рисунок 6: Грамматика языка Python

2. Лексер

- Лексер преобразует последовательность символов в последовательность токенов

- Лексеры также известны как сканеры или токенизаторы. Лексеры играют важную роль в синтаксическом анализе, поскольку они преобразуют первоначальный ввод в форму, более управляемую соответствующим анализатором, который работает на более позднем этапе.

- Пример токенов, используемых для языка Python :

```
ops = {ast.Add: "#add", ast.Sub: "#sub", ast.Mult: "#mult",
       ast.Div: "#div", ast.Mod: "#mod", ast.Pow: "#pow",
       ast.LShift: "#l_shift", ast.RShift: "#r_shift",
       ast.BitOr: "#bit_or", ast.BitAnd: "#bit_and", ast.BitXor: "#bit_xor",
       ast.And: "#and", ast.Or: "#or", ast.Not: "#not",
       ast.Eq: "==", ast.NotEq: "!=", ast.Lt : "<", ast.LtE: "<=",
       ast.Gt: ">", ast.GtE: ">=", ast.Is : "#is", ast.IsNot: "#isnot",
       ast.In: "#in", ast.NotIn: "#notin"}
```

Рисунок 7: Токены для бинарных операций

```
context = {ast.Store : "#store", ast.Load : "#load", ast.Del : "#del"}

class Var(ast.AST):
    #ctx: Load/Store/Del/Starred
    def __init__(self, id, ctx=None, dtype=None):
        self._id = id
        self._ctx = ctx
        self._dtype = dtype
        self._fields = ["id", "ctx", "dtype"]
```

Рисунок 8: Токены для переменных

3. Синтаксический анализатор

- В контексте синтаксического анализа “анализатор” может относиться как к программному обеспечению, которое выполняет весь процесс, так и просто к соответствующему анализатору, который анализирует токены, созданные лексером. Это просто следствие того факта, что анализатор берет на себя самую важную и сложную часть всего процесса синтаксического анализа.

- Под самым важным мы подразумеваем то, что больше всего волнует пользователя и что он действительно увидит. На самом деле, как мы уже говорили, лексер работает как помощник для облегчения работы синтаксического анализатора.

- В любом выбранном вами смысле вывод синтаксического анализатора представляет собой организованную структуру кода, обычно в виде дерева. Дерево может быть деревом синтаксического анализа или абстрактным синтаксическим деревом. Они оба являются деревьями, но отличаются тем, насколько точно они представляют написанный фактический код и промежуточные элементы, определенные анализатором.

4. Алгоритмы синтаксического анализа и реализация

- Существует две стратегии синтаксического анализа: анализ сверху вниз и анализ снизу вверх. Оба термина определяются по отношению к дереву синтаксического анализа, сгенерированному анализатором. В простых терминах:

- Анализатор сверху вниз сначала пытается определить корень дерева синтаксического анализа, затем он движется вниз по поддеревьям, пока не найдет листья дерева.

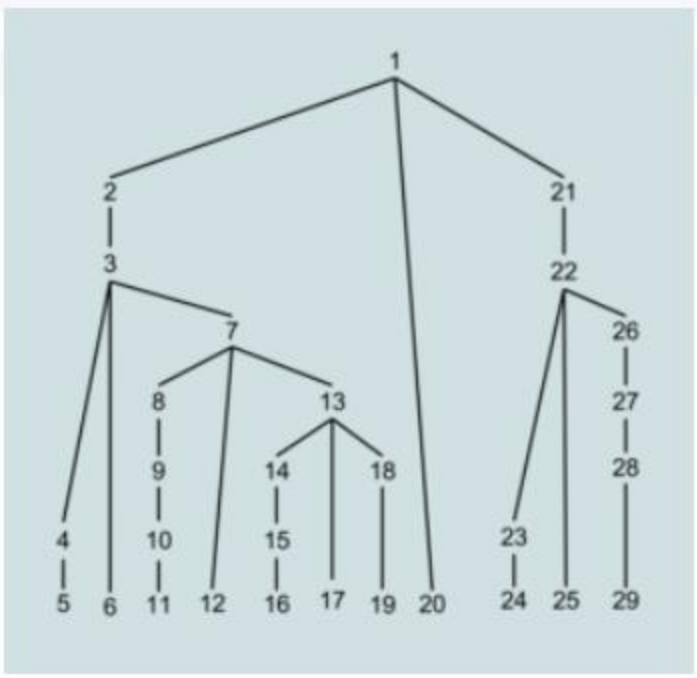

Рисунок 9: Дерево синтаксического анализа сверху вниз

- Вместо этого анализатор снизу вверх начинается с самой нижней части дерева, листьев, и поднимается вверх, пока не определит корень дерева.

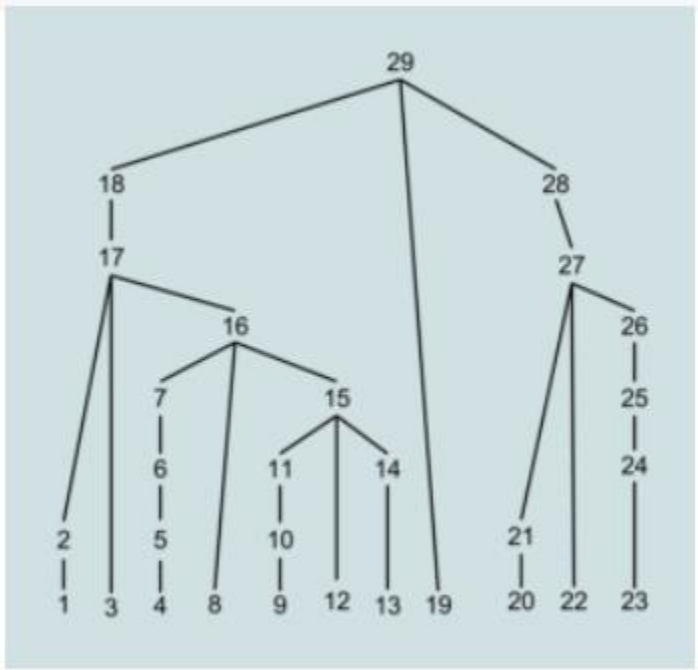

Рисунок 10: Дерево синтаксического анализа снизу вверх

- Метод, который был использован в этом проекте, называется "сверху вниз".

- Класс инициализации и его свойства, такие как исходный код, который необходим для синтаксического анализа, внутренние методы, ...

```
class Parser(Visitor):

    #ast.NodeVisitor.visit(node) --> self.visit_classname()

    def __init__(self, source):
        if isinstance(source, types.ModuleType) or \
           isinstance(source, types.FunctionType) or \
           isinstance(source, types.LambdaType):
            source = dedent(inspect.getsource(source))
        elif isinstance(source, str):
            source = dedent(source)
        else:
            raise NotImplementedError

        self._source = source
        self._ast = ast.parse(source)
        self._syntax_tree = self.visit(self._ast)
```

Рисунок 11: Начальная функция синтаксического анализа

- Основываясь на определенной грамматике, затем выполните метод обхода сверху вниз, чтобы получить дерево синтаксического анализа. Пример при анализе "Function_Def":

```
def visit_Return(self, node):···

def visit_FunctionDef(self, node):
    #name, args, body, decorators_list, returns
    #print("* funcdef *", ast.dump(node))
    name = node.name
    args = self.visit(node.args)

    body = list(map(self.visit, node.body))

    #still doesn't support decorator
    decorator_list = node.decorator_list
    returns = node.returns
    func = FunctionDef(name, args, body, decorator_list, returns)
    return func

def visit_AugAssign(self, node):      ···
```

Рисунок 12: Анализ функции

Исходный код, который необходим для синтаксического анализа.

```
@py_llvm_jit(PARSE=True, INFER=False, LLFUNC=False)
def test(a,b):
    c = a + b
    return c
```

Рисунок 13: Пример исходного кода для синтаксического анализа

Затем это анализируемый результат.

```
=============== PARSE ====================
('FunctionDef',
 {'args': [('Var', {'ctx': None, 'dtype': TVar $a, 'id': "'a'"}),
           ('Var', {'ctx': None, 'dtype': TVar $b, 'id': "'b'"})],
  'body': [('Assign',
            {'targets': ('Var',
                         {'ctx': "'#store'", 'dtype': TVar $b, 'id': "'c'"}),
             'type_comment': None,
             'value': ('BinOp',
                       {'left': ('Var',
                                 {'ctx': "'#load'",
                                  'dtype': TVar $a,
                                  'id': "'a'"}),
                        'op': "'#add'",
                        'right': ('Var',
                                  {'ctx': "'#load'",
                                   'dtype': TVar $b,
                                   'id': "'b'"})})}),
           ('Return',
            {'value': ('Var',
                       {'ctx': "'#load'", 'dtype': TVar $b, 'id': "'c'"})})],
  'decorator_list': None,
  'name': "'test'",
  'returns': None,
  'type_comment': None})
```

Рисунок 14: Результат синтаксического анализа данного исходного кода

III. Промежуточное представление и то, как инфраструктура LLVM поддерживает их создание

1. LLVM промежуточное представление – LLVM IR

- LLVM IR — это низкоуровневое промежуточное представление, используемое фреймворком компиляции LLVM. Вы можете думать об LLVM IR как о платформенно-независимом ассемблере с бесконечным количеством локальных регистров.

- При проектировании компилятора существует огромное преимущество в компиляции исходного языка в промежуточное представление (IR, intermediate representation) вместо компиляции в целевую архитектуру (например, x86).

- Идея использования промежуточного языка в компиляторах широко распространена. GCC использует GIMPLE, Roslyn использует CIL, LLVM использует LLVM IR. Так как много техник оптимизации являются общими (например, удаление неиспользуемого кода, распространение констант), эти проходы оптимизации могут быть выполнены напрямую на уровне IR и использоваться всеми целевыми платформами.

- Использование промежуточного языка (IR), таким образом, уменьшает количество комбинаций, требуемых для n исходных языков и m целевых архитектур (бэкендов) с *N*M* до *N+M*. Таким образом, компиляторы часто состоят из трёх частей: фронтенд, миддленд и бэкенд, каждая из них выполняет свою задачу, принимая на входе и/или отдавая на выходе IR.

    - Фронтенд: компилирует исходный язык в IR
    - Миддленд: оптимизирует IR
    - Бэкенд: компилирует IR в машинный код

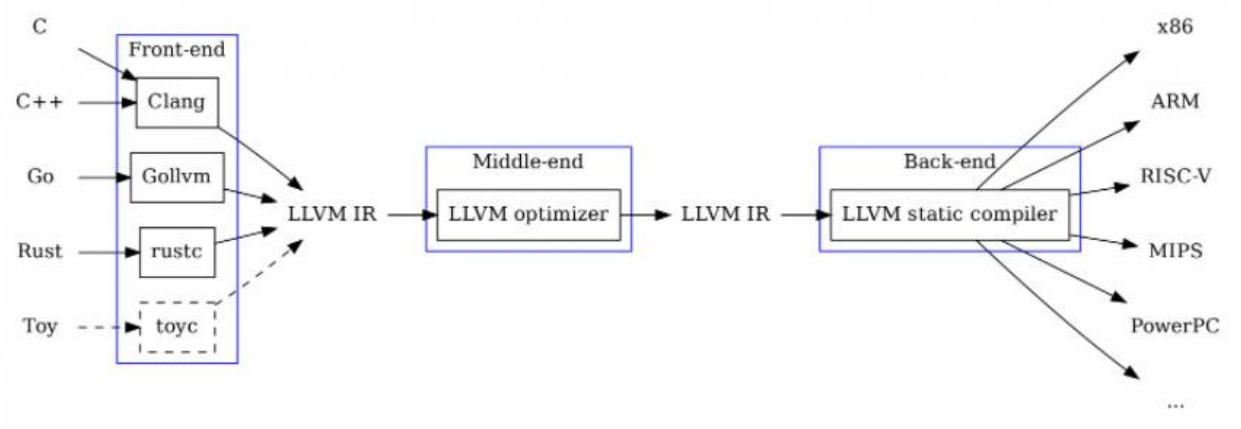

Рисунок 15: Архитектура LLVM

2. Структура ассемблера LLMV IR

- Содержание ассемблерного файла LLVM IR является модулем. Модуль содержит объявления высокого уровня, такие, как глобальные переменные и функции. Декларация функции не содержит базовых блоков, определение функции содержит один или более базовых блоков (т.е. тело функции).

2.1. Основные Типы

- Типы LLVM - это ваши типичные типы машин плюс указатели, структуры, векторы и массивы.

i1 1	логический бит
i32 299792485	целое число
float 3.423e-3	одиночная точность
double 3.2321e-2	двойная точность
{float, i64}	структура
{float, {double, i3}}	вложенная структура
<{float, [2 x i3]}>	упакованная структура
[10 x float]	массив из 10 поплавков
[10 x [20 x i32]]	Массив из 10 массивов по 20 целых чисел.
<8 x float>	Вектор ширины 8 поплавков
float*	Указатель
[25 x float]*	Указатель на массив

2.2. Инструкции

- Все инструкции привязаны к уникальному виртуальному регистру. В SSA (одиночное статическое назначение) регистр никогда не назначается более одного раза.

```
%result = add i32 10, 20
```

- Символы, используемые в модуле LLVM, являются либо глобальными, либо локальными. Глобальные символы начинаются с @, а локальные - с %.

- Числовые инструкции таковы:

add	целочисленное сложение
fadd	плавающей точкой
sub	целочисленное вычитание
fsub	с плавающей точкой вычитание
mul	целочисленное умножение
fmul	умножение с плавающей точкой
udiv	беззнаковое целое частное
sdiv	целое число со знаком деления
fdiv	с плавающей точкой частное
urem	беззнаковый целочисленный остаток
srem	подписанный целочисленный остаток
frem	плавающей запятой в целочисленный остаток

2.3. Память

- LLVM использует традиционную модель загрузки/хранения:
  - load: Загрузка типизированного значения из заданного
  - store: Сохранение типизированного значения в заданном хранилище ссылок:
  - alloca: Выделение указателя на память в виртуальном стеке

```
%ptr = alloca i32
store i32 3, i32* %ptr
%val = load i32* %ptr
```

2.4. Функции

- Функции определяются как набор базовых блоков, тип возвращаемого значения и типы аргументов. Имена функций в модуле должны быть уникальными.

```
define i32 @add(i32 %a, i32 %b) {
  %1 = add i32 %a,%b
  ret i32 %1
}
```

2.5. Основные Блоки

- Функция разделена на базовые блоки, которые содержат последовательности инструкций, и команду-терминатор, которая либо возвращает, либо переходит к другому локальному базовому блоку.

```
define i1 @foo() {
   entry:
      br label %next
   next:
      br label %return
   return:
      ret i1 0
}
```

2.6. Возвращать

- Функция должна иметь терминатор, одной из таких инструкций является набор, который возвращает значение в стек.

```
define i1 @foo() {
   ret i1 0
}
```

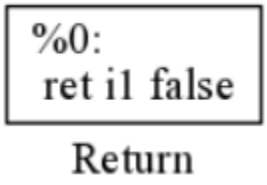

Рисунок 16: IR-программа для возврата

2.7. Безусловная Ветка

- Безусловная ветвь безоговорочно переходит к помеченному базовому блоку.

```
define i1 @foo(){
   start:
      br label %next
   next:
      br label %return
   return:
      ret i1 0
}
```

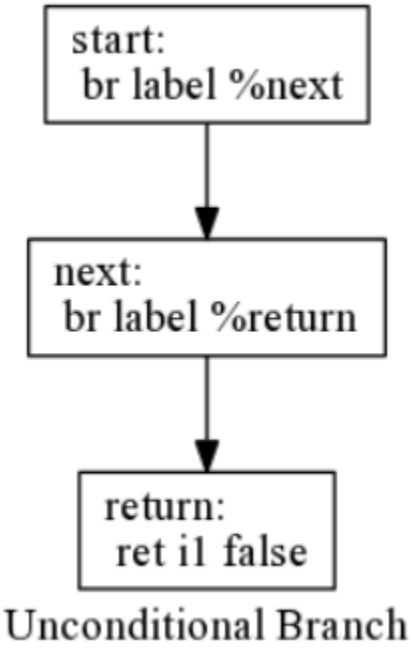

Рисунок 17: IR-программа для безусловного ветвления

2.8. Условная Ветка

```
define i32 @foo(){
   start:
     br i1 true, label %left, label %right
   left:
     ret i32 10
   right:
     ret i32 20
}
```

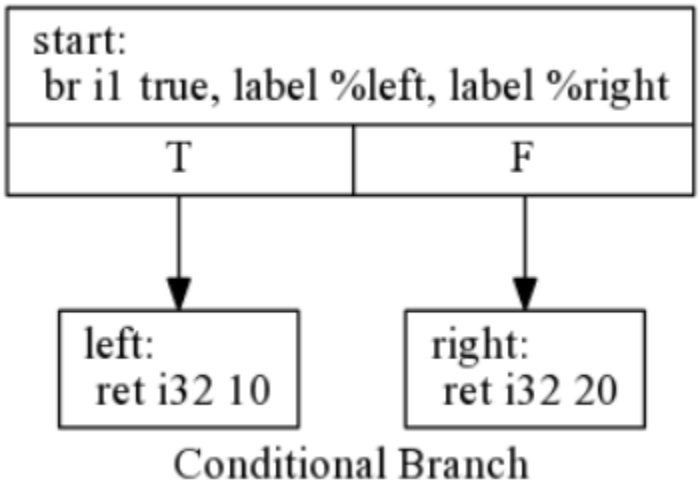

Рисунок 18: IR-программа для условного ответвления

2.9. Phi

- Узлы Phi выдают значение, которое зависит от операнда, соответствующего их предыдущему базовому блоку. Они используются для реализации циклов в SSA.

```
define i32 @foo(){
   start:
     br i1 true, label %left, label %right
   left:
     %plusOne = add i32 0, 1
     br label %merge
   right:
     br label %merge
   merge:
     %join = phi i32 [ %plusOne, %left], [ -1,
%right]
     ret i32 %join
}
```

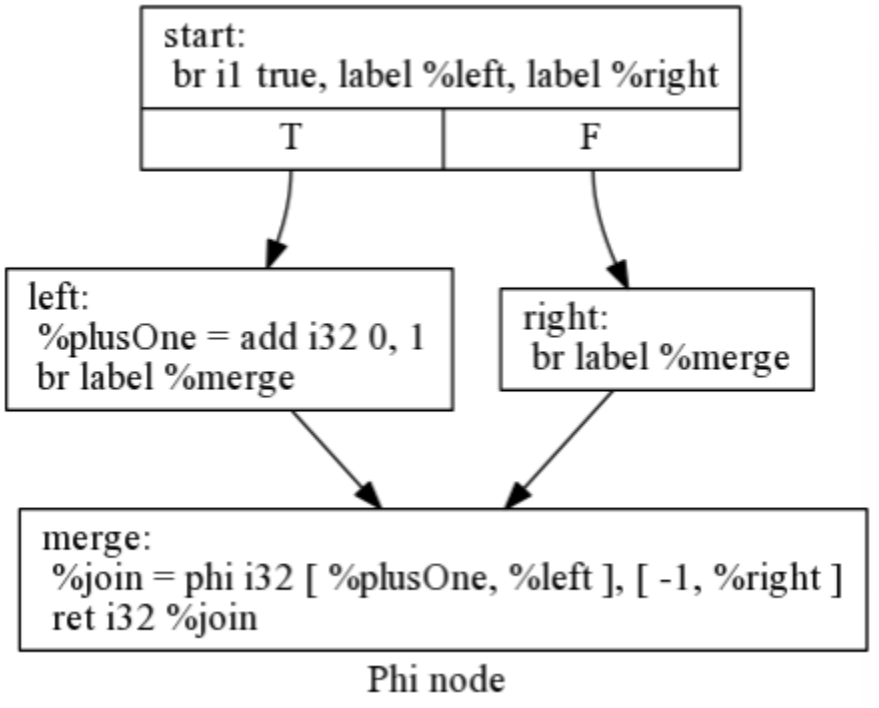

Рисунок 19: IR-программа для узла Phi

2.10. Циклы

- Циклы записываются в терминах условных ветвей и узлов phi.

```
define i32 @count(i32 %n){
   entry:
      br label %loop
   loop:
     %i = phi i32 [ 1, %entry ], [ %nextvar, %loop ]
     %nextvar = add i32 %i, 1
     %cmptmp = icmp ult i32 %i, %n
     %booltmp = zext i1 %cmptmp to i32
     %loopcond = icmp ne i32 %booltmp, 0 br i1
%loopcond, label %loop, label %afterloop
   afterloop:
      ret i32 %i
}
```

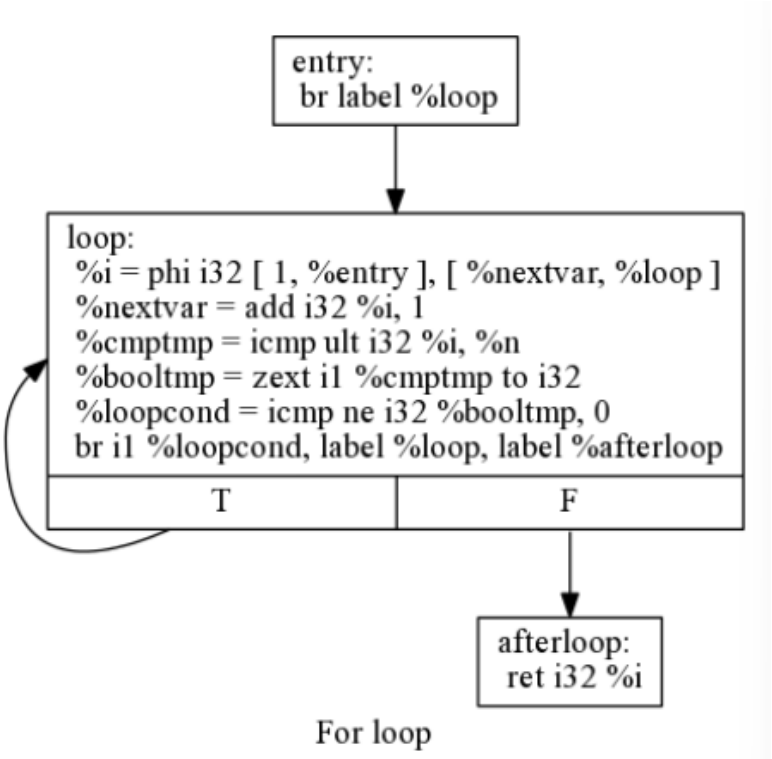

Рисунок 20: IR-программа для цикла for

3. Реализация

- LLVM предоставляет множество API-интерфейсов, соответствующих IR. Чтобы перевести источник в блоки IR, мы можем пройти по дереву синтаксического анализатора, а затем реализовать LLVM API для соответствующей структуры.

- Пример процесса переноса из циклической структуры в IR:

```
def visit_For(self,node):
    #print("------for -- llvm ir--------")
    init_block = self.function.append_basic_block('for.init')
    cond_block = self.function.append_basic_block('for.cond')
    body_block = self.function.append_basic_block('for.body')
    end_block = self.function.append_basic_block('for.end')

    #increament init
    self.branch(init_block)
    self.set_block(init_block)

    args,_ = self.visit(node.iter)
    start,end = args[0], args[1]
    if len(args) == 2:
        inc_size = ir.Constant(int_type, 1)
    else:
        inc_size = args[2]

    inc = self.visit(node.target)
    self._builder.store(start, inc)

    #loop condition
    self.branch(cond_block)
    self.set_block(cond_block)
    cond = self._builder.icmp_signed('<', self._builder.load(inc), end)
    self.cbranch(cond, body_block, end_block)

    # loop body
    self.set_block(body_block)
    _ = list(map(self.visit, node.body))

    # increment counter
    succ = self._builder.add(inc_size, self._builder.load(inc))
    self._builder.store(succ, inc)

    #exit the loop
    self.branch(cond_block)
    self.set_block(end_block)
    #print("------for -- llvm ir--------")
```

Рисунок 21: API-интерфейсы LLVM для создания IR-программы для цикла

4. Результат IR-транскрипции

- Программа Python

```python
@py_llvm_jit(AST=False, PARSE=False, INFER=False, LLFUNC=True)
def test(a,b):
    c = 0
    for i in range(a,b):
        c = c + i
    return c

print(test(2,3))
```

Рисунок 22: Исходный код

- После переведите его в IR-программу

```
=============== LLVM FUNC ====================
define i32 @"test-7069661619265547187"(i32 %"a", i32 %"b")
{
entry:
  %".4" = alloca i32
  store i32 %"a", i32* %".4"
  %".6" = alloca i32
  store i32 %"b", i32* %".6"
  %"retval" = alloca i32
  %"c" = alloca i32
  store i32 0, i32* %"c"
  br label %"for.init"
exit:
  %".28" = load i32, i32* %"retval"
  ret i32 %".28"
for.init:
  %".10" = load i32, i32* %".4"
  %".11" = load i32, i32* %".6"
  %"i" = alloca i32
  store i32 %".10", i32* %"i"
  br label %"for.cond"
for.cond:
  %".14" = load i32, i32* %"i"
  %".15" = icmp slt i32 %".14", %".11"
  br i1 %".15", label %"for.body", label %"for.end"
for.body:
  %".17" = load i32, i32* %"c"
  %".18" = load i32, i32* %"i"
  %".19" = add i32 %".17", %".18"
  store i32 %".19", i32* %"c"
  %".21" = load i32, i32* %"i"
  %".22" = add i32 1, %".21"
  store i32 %".22", i32* %"i"
  br label %"for.cond"
for.end:
  %".25" = load i32, i32* %"c"
  store i32 %".25", i32* %"retval"
  br label %"exit"
}
```

Рисунок 23: IR-программа для исходного кода

IV. LLVM JIT и поддержка оптимизатора

1. Оптимизация LLVM

- LLVM предоставляет множество проходов оптимизации, которые выполняют множество различных задач и имеют разные компромиссы. В отличие от других систем, LLVM не придерживается ошибочного представления о том, что один набор оптимизаций подходит для всех языков и для всех ситуаций. LLVM позволяет разработчику компилятора принимать полные решения о том, какие оптимизации использовать, в каком порядке и в какой ситуации

- LLVM поддерживает оба прохода “всего модуля”, которые просматривают как можно больший объем кода (часто целый файл, но при запуске во время соединения это может быть существенная часть всей программы). Он также поддерживает и включает в себя проходы “для каждой функции”, которые работают только с одной функцией одновременно, не обращая внимания на другие функции.

- LLVM предоставляет широкий спектр оптимизаций, которые могут быть использованы в определенных обстоятельствах. Имеется некоторая документация о различных пропусках, но она не очень полная. Еще один хороший источник идей может быть получен при просмотре проходов, которые Clang выполняет для начала. Инструмент “выбрать” позволяет вам экспериментировать с проходами из командной строки, чтобы вы могли видеть, делают ли они что-нибудь.

- Пример добавления оптимизации LLVM на уровне МОДУЛЯ:

```
def tuning(module):
    #create module

    mod = llvm.parse_assembly(str(module))
    mod.verify()

    pass_manager = llvm.PassManagerBuilder()

    # optimization level
    # 0 - no optimization
    # 2 - enalbe most of optimization
    # 3 - enables optimizations that take longer to perform
    # attempt to make programm run faster
    pass_manager.opt_level = 3

    # vectorizing loop
    pass_manager.loop_vectorize = True

    pass_module = llvm.ModulePassManager()

    #optional optimization
    pass_module.add_constant_merge_pass()
    pass_module.add_dead_code_elimination_pass()
    pass_module.add_global_dce_pass()
    pass_module.add_global_optimizer_pass()
    pass_module.add_cfg_simplification_pass()
    pass_module.add_licm_pass()
    pass_manager.populate(pass_module)

    pass_module.run(mod)

    return mod
```

Рисунок 24: Оптимизация LLVM

2. Добавление JIT-компилятора

- Код, доступный в LLVM IR, может иметь широкий спектр инструментов, применяемых к нему. Например, вы можете выполнить оптимизацию на нем (как мы делали выше), вы можете вывести его в текстовой или двоичной форме, вы можете скомпилировать код в файл сборки (.ы) для некоторой цели, или вы можете скомпилировать его JIT. Самое приятное в представлении LLVM IR заключается в том, что оно является “общей валютой” между многими различными частями компилятора.

- В этом разделе мы добавим поддержку JIT-компилятора в наш интерпретатор. Основная идея, которую мы хотим для Kaleidoscope, состоит в том, чтобы пользователь вводил тела функций, как сейчас, но сразу же оценивал выражения верхнего уровня, которые они вводят. Например, если они наберут “1 + 2;”, мы должны оценить и распечатать 3. Если они определяют функцию, они должны иметь возможность вызывать ее из командной строки.

2.1. Механизм выполнения

- Во-первых, нам нужно создать механизм выполнения. Механизм выполнения - это место, где происходит фактическая генерация и выполнение кода

```
def create_execution_engine():
    # Create an ExecutionEngine suitable for JIT code generation on
    # the host CPU
    target = llvm.Target.from_default_triple()
    target_machine = target.create_target_machine()
    backing_mod = llvm.parse_assembly("")
    engine = llvm.create_mcjit_compiler(backing_mod, target_machine)

    return engine

def code_gen(ast, specialization, spec_args, spec_ret):
    llvm_code = Emitter.LLVMEmitter(specialization, spec_args, spec_ret, MODULE)
    llvm_code.visit(ast)

    mod = llvm_passes.tuning(MODULE)

    engine = create_execution_engine()
    engine.add_module(mod)

    return (llvm_code.function,engine)
```

Рисунок 26: Механизм выполнения

2.2. Функции вызова

- После того, как мы зарегистрировали нашу функцию в исполняемом движке, мы можем вызвать указатель на функцию, которую нам нужно выполнить с помощью метода:

```
func_address = self._engine.get_function_address(self._llfunc.name)
```

Рисунок 27: Получить указатель на функцию

- Чтобы повысить производительность вызовов функций, давайте создадим указатели на вызываемые функции C/C++ из вызываемых функций Python, с помощью библиотеки под названием ctypes. И поскольку мы склонны использовать вызываемую функцию C/C++, нам также нужны аргументы C/C++

```
def jit_func(self):
    llvm_args = self._llfunc.type.pointee.args
    llvm_ret = self._llfunc.type.pointee.return_type

    runnable_ret = self.jit_type(llvm_ret)
    runnable_args = list(map(self.jit_type, llvm_args))

    func_address = self._engine.get_function_address(self._llfunc.name)
    runnable_func = ctypes.CFUNCTYPE(runnable_ret, *runnable_args)(func_address)
    runnable_func.__name__ = self._llfunc.name
    return runnable_func
```

Рисунок 28: Вызываемая функция C из вызываемой функции Python

V. Заключение

1. Результат сравнения между JIT-компилятором LLVM и интерпретатором python по умолчанию

- Давайте посмотрим на простую функцию цикла из (*1- 1.000.000.000*). Первая функция выполняется с помощью JIT-компилятора LLVM из соответствующей программы IR, вторая выполняется интерпретатором Python по умолчанию

```
@py_llvm_jit(AST=False, PARSE=False, INFER=False, LLFUNC=False)
def addup(a,b):
    step = 1
    c = 0
    for i in range(a,b,step):
        if i%2==1:
            c = c + 1
        else:
            c = c - 1
    return c

def addup_1(a,b):
    step = 1
    c = 0
    for i in range(a,b,step):
        if i%2==1:
            c = c + 1
        else:
            c = c - 1
    return c

print("Function with LLVM JIT compiler")
print(addup(1,10000000),"\n")

print("Function without LLVM JIT compiler")
s = time.time()
res = addup_1(1,10000000)
print("=============== TIME EXEC ====================")
print(time.time() - s)
print("=============== RESULT ========================")
print(res)
```

Рисунок 29: Программа тестирования

- Полученный результат

```
D:\My_Code\llvm_py_jit>py code_test_1.py
Function with LLVM JIT compiler
============== TIME EXEC ====================
0.018027067184448242
============== RESULT ======================
1

Function without LLVM JIT compiler
============== TIME EXEC ====================
1.6528706550598145
============== RESULT ======================
1
```

Рисунок 30: Результат программы тестирования

2. Будущая работа

- Перевести весь Python AST
- Используйте subpy для обнаружения функций, прежде чем опускать функцию в LLVM. Обеспечивает лучшее сообщение об ошибках при использовании недопустимой функции высокого уровня вместо сбоя где-то в середине конвейера компилятора.
- Используйте информацию о строках и столбцах в AST Python, чтобы улучшить обработку ошибок.
- Добавьте больше типов и явных приведений или двунаправленный вывод типов.
- Для непереводимых вызовов используйте объектный уровень в C-API, чтобы вернуться к интерпретатору.
- Сопоставьте подмножество вызовов numpy с LLVM.
- Используйте серверную часть LLVM nvptx для настройки логики Python на ядра Nvidia CUDA.
- Используйте pthreads внутри LLVM для логического написания многоядерных программ без обычных ограничений GIL.
- Используйте передачу массива с нулевой копией MPI или Zeromq, чтобы эффективно распределять числовые ядра на нескольких компьютерах.

VI. Список литературы

1. Welcome to Python.org
2. The LLVM Compiler Infrastructure Project
3. llvmlite — llvmlite 0.38.0dev0-93-g00320f6-dirty documentation
4. ctypes — A foreign function library for Python — Python 3.10.1 documentation
5. Context-free grammar - Wikipedia
6. Parsing - Wikipedia