Министерство образования и науки Российской Федерации
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
“САНКТ-ПЕТЕРБУРГСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ, МЕХАНИКИ И ОПТИКИ”

Факультет Программной инженерии и компьютерной техники

Образовательная программа Системное и прикладное программное обеспечение

Направление подготовки (специальность) 09.03.04. – Программная инженерия

О Т Ч Е Т

об учебной практике, по получению первичных профессиональных умений и навыков

Тема задания: **Разработка компилятора из императивного языка в байт-код виртуальной машины DWARF-VM**

Обучающийся **Черноусов Е.А., группа Р3210**

Руководитель практики от университета: **Маркина Т.А., Университет ИТМО, факультет ПИиКТ, старший преподаватель**

Практика пройдена с оценкой _____

Подписи членов комиссии

_____(_____)
(подпись)

_____(_____)
(подпись)

_____(_____)
(подпись)

Дата _____

Санкт-Петербург
2019

Содержание

1. Определение поставленной задачи.

Во время практики передо мной была поставлена задача реализовать простой компилятор из императивного языка в байт-код виртуальной машины. Согласно заданию, предоставленному руководителем, компилятор должен обрабатывать простейшие конструкции языка (арифметика, функции, условия и цикл while).

Для качественного выполнения задачи я должен был применить знания из курса «Языки системного программирования», который прослушал в 3ем семестре, лекционные материалы по разработке компиляторов, рассказанные руководителем практики и открытые источники для решения возникающих по ходу работы теоретических вопросов.

Я не был ограничен в выборе стека технологии, поэтому после ликбеза, для разработки компилятора мной, по рекомендации преподавателя, был выбран язык программирования Haskell, предоставляющий набор готового функционала для написания части программы. Следующей основной целью стало изучение теоретического материала.

2. Ознакомление с основами разработки компиляторов

Для начального ознакомления с процессом разработки компиляторов была посещена online лекция Игоря Олеговича Жиркова[1] на которой были разобраны следующие критерии:

- Что такое язык программирования, что включало основную информацию необходимую для общего понимания структуры языков, что необходимо для разбора компилятора. Например, были изучены понятие деревьев кода и связанные с ним абстрактный и конкретный синтаксисы.
- Языки можно разделить по синтаксису (грамматике):
    - Регулярные языки
    - Контекстно-свободные грамматики
    - Контекстно-зависимые грамматики
    - Неограниченные (машины Тьюринга)

  Для выполнения работы необходимо было выбрать язык с контекстно-свободной грамматикой, т. к. он наиболее легок в лексическом разборе.
- После на примере были разобраны составляющие грамматики (терминалы, нетерминалы, стартовый нетерминал, правила), и способ объединения их в структуры (деревом разбора и AST - абстрактным синтаксическим деревом), необходимые для дальнейшего построения байт-кода с учетом особенностей синтаксиса языка. Для преобразования текста программы в древовидную структуру основных синтаксических единиц,

которая преобразуется в лаконичное AST, удовлетворяющее синтаксису, необходимо использовать алгоритм рекурсивный спуск.

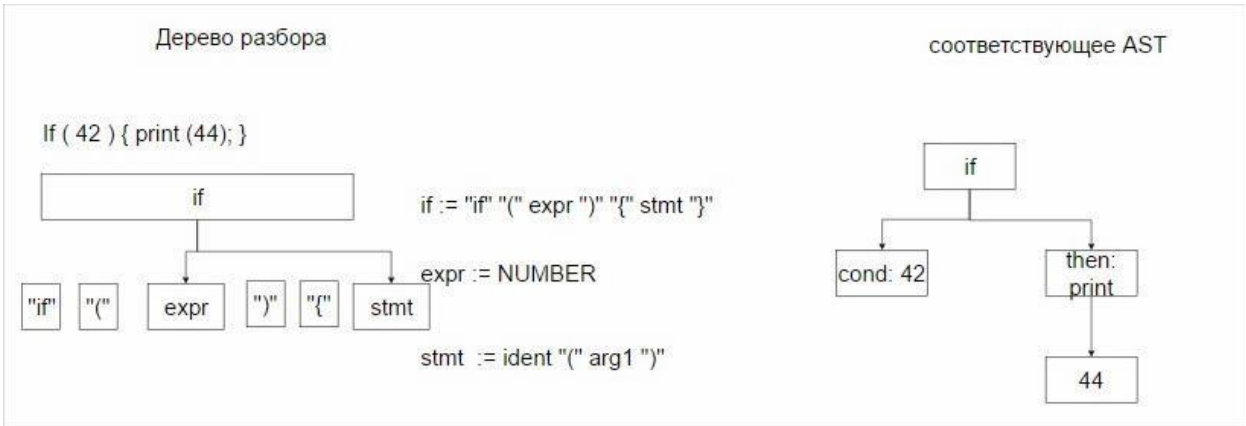

Рисунок 1 Визуальный пример создания дерева разбора и AST

- Для более глубокого изучения этого алгоритма был прочитана лекция «Синтаксические анализаторы. Нисходящие анализаторы», взятая в интернете [2] в ней были описаны основная логика работы алгоритма, условия использования и несколько примеров. Изучение данного материала способствовало разработке в будущем.
- Познакомился с основами устройства компилятора:

1. Лексический анализ
   a. Вводится текст программы
   b. Выход: список “лексем”, “токены”

   В языке есть числа, идентификаторы, пробелы, ключевые слова
   Исключаются пробелы, берутся лексемы, повторяем пока программа не кончилась.
   Вход:
   “If (x == 42) {
   Print(44);
   }”
   Выход: последовательность лексем
   “If” “(“ { x } “==” {num:42} “)” “{“ { print } “(“ {num:44} “)” “;” “}”
2. Синтаксический анализ
   Вход: поток лексем
   Выход: Дерево разбора => AST
   (if (==x 42) (call “print” 44)) - описание дерева в виде текста “quoted tree”
3. Семантический анализ
   Вход: AST
   Выход: размеченное типами, ссылками AST

Дополнительные промежуточные представления IR (intermediate representation)

Эта стадия реализуется с помощью рекурсивного спуска.

4. Генерация кода

В результате должен получиться байт-код, поддерживаемый выбранной виртуальной машиной.

3. Исследование технологии и их применение (IMP – императивный язык, dwarf-vm – основные инструкции и синтаксис, Parsec)

- Для разработки компилятора мною был выбран простой императивный C-like язык, информацию о синтаксисе и грамматике которого я подчерпнул из книги И.О. Жиркова «Low-Level Programming»[3]. Данный язык обладает наиболее простой грамматикой и полностью подходит под мое техническое задание.

```
<statements> ::= <statement> | <statement> ";" <statements>
<statement> ::= "{" <statements> "}" | <assignment> | <if> | <while> | <print>
<print> ::= "print" "(" <expr> ")"
<assignment>  ::=  IDENT  "="  <expr>
<if> ::= "<if>" "(" <expr> ")" <statement> "<else>" <statement>
<while> ::= "<while>" "(" <expr> ")" <statement>

<expr> ::= <expr0> "<" <expr> | <expr0> "<=" <expr>
| <expr0> "==" <expr> | <expr0> ">" <expr> | <expr0> ">=" <expr> | <expr0>
<expr0> = <expr1> "+" <expr> | <expr1> "-" <expr> | <expr1>
<expr1> ::= <atom> "*" <expr1> | <atom> "/" <expr1> | <atom>
<atom> ::= "(" <expr> ")" | NUMBER
```

Рисунок 2 Грамматика языка

- Компиляция производится в байт-код dwarf-vm — это простая стековая виртуальная машина, которая транслирует байт-код «игрушечных» языков в машинный код.

  Для получения более подробной информации обратился к документации, сгенерированной с помощью Doxygen. Этот файл находится в репозитории проекта на GitHub[4]. Для способности генерации кода для виртуальной машины полученная документация была изучена, после чего оттуда был выбран «Джентельменский набор инструкций»:
  - Арифметика
  - Прыжки с условием и без (как минимум 2)
  - Ввод-вывод
  - Работа с памятью / локальными переменными

- В процессе разработки, на этапе написания лексического анализатора (лексера) – модуля, преобразующего исходный текст в набор лексем, мной было найдено готовое решение в виде Parsec. Он оказался удобным и быстрым инструментом, который позволяет строить свои собственные парсеры комбинируя готовые примитивные парсеры при помощи парсерных комбинаторов. Подход, используемый Parsec, вполне естественен для создания рекурсивных парсеров в рамках функционального программирования и базируется на следующих принципах:

    1) Каждый парсер представляется в виде некоторой функции.

    2) Эта функция принимает на вход некоторую строку, которую надо обработать. Назовём тип результата просто *a*. Получим следующий тип: type Parser a = String -> a, где *a* - параметр типа Parse*r*, показывающий, что именно мы хотим от него получить.

    3) Определяются функции высшего порядка (комбинаторы), которые определённым образом сочетают парсеры, позволяя производить такие операции определения грамматик, как последовательное распознавание двух символов грамматики (sequencing), параллельное - то есть выбор из двух альтернатив (choice) или же более сложные конструкции расширенной нотации Бэкуса-Наура вроде повторов (repeating). Для всех описанных конструкций существуют аналоги в Parsec. Последовательное выполнение парсеров записывается с помощью операции связывания >>, выбор с помощью комбинатора <|>, а повтор - с помощью комбинатора *many*. Таким образом парсер индентификаторов в Parsec будет выглядеть так: identifier = letter >> many (letter <|> digit). Затем на основе элементарных комбинаторов строятся более сложные комбинаторы и так далее.

    4) Комбинаторы и парсеры формируют некую абстрактую структуру, называемую *"монадой"*, пришедшую в Haskell из теории категорий. Монаду можно представить как некий контейнерный тип, в котором элементы контейнера могут быть связаны друг с другом некоторой стратегией вычислений. В данном случае, каждой составляющей контейнера является конкретный парсер, а под их связыванием подразумевается последовательное выполнение двух парсеров с промежуточной передачей результата разбора первым - второму. "Упаковка" парсеров в монаду помогает сокрыть такие вещи, как явную передачу промежуточного состояния парсинга, а также позволяет записывать парсеры в более компактной и удобочитаемой форме.

Таким образом, можно отметить, что использование комбинаторных монадических парсеров является типичным функциональным подходом к разбору, позволяет избежать специфических языков, изучение которых необходимо для написания грамматик (как при работе с yacc, ANTLR), кроме того, в случае Parsec - парсеры являются first-class citizens языка, что позволяет значительно ускорить процесс их создания.

Основная информация о работе Parsec была найдена в интернете в статье «Haskell: комбинаторные парсеры Parsec» [5].

4. Процесс разработки компилятора

Как было сказано выше было принято решение вести разработку на языке Haskell. В соответствии с полученными теоретическими знаниями процесс разработки был разбит на этапы на каждом из которых создавалось необходимое представление исходного кода. На последнем этапе был получен байт-код виртуальной машины.

- Лексический анализ
  Для преобразования текста исходного кода в набор лексем было использовано готовое решение – Parsec. Для его интеграции в проект надо было подключить необходимые модули в соответствии с инструкцией в статье «Создаём парсер на Haskell»[6]. Теперь нужно запустить парсер. Для этого есть функция parse, которой нужно передать главный нетерминал, имя текста (используется для вывода ошибок) и сам текст. Функция parse возвращает либо описание ошибки, либо полученные данные. Если всё распарсилось, то в полученных данных мы ищем запись по имени секции и имени параметра. Поиск может вернуть либо найденное значение (Just), тогда мы его выводим, либо ничего (Nothing) — тогда выводим сообщение об ошибке.

- Синтаксический анализ
  Лексемы, полученные на предыдущем этапе, сопоставляются с формальной грамматикой языка, после чего строится дерево разбора, распределяя лексемы в структуре данных, учитывая их тип.
  Для хранения использовалась таблица символов - это структура данных, используемая транслятором (компилятором или интерпретатором), в которой каждый идентификатор переменной или функции из исходного кода ассоциируется с информацией, связанной с его объявлением или появлением в коде: типом данных, областью

видимости и в некоторых случаях местом в памяти (смещением). Обще используемой реализацией является хеш-таблица. Компилятор может использовать единую таблицу для символов.
Основную информацию по созданию такой структуры данных подчерпнул из статьи на wikipedia.org «Таблица символов» [7] и лекции по конструированию компиляторов [8].

- Семантический анализ
  После построения синтаксического дерева и таблицы символов на их основе было построено AST (абстрактное синтаксическое дерево), но с учетом данных, полученных на прошлом шаге была осуществлена проверка типов, когда компилятор проверяет, имеет ли каждый оператор операнды соответствующего типа, неправильное объявление циклов, использование конструкций, указывающих на начало или конец блока кода и т. д. (то есть рассматриваются особенности спецификации языка).
  Для этого по полученному дереву разбора строится AST - ориентированное дерево, в котором внутренние вершины сопоставлены (помечены) с операторами языка программирования, а листья — с соответствующими операндами. В нём отсутствуют узлы и рёбра для тех синтаксических правил, которые не влияют на семантику программы.
  Параллельно с этим проводится семантический анализ и при условии ошибки она выводится на экран.
  Для обхода дерева использовался рекурсивный спуск, т. е. пока в синтаксическом дереве были узлы мы совершали обход в глубину, добавляя элементы в AST.
  Пример создания дерева разбора и AST можно увидеть на Рисунок 1 Визуальный пример создания дерева разбора и AST.

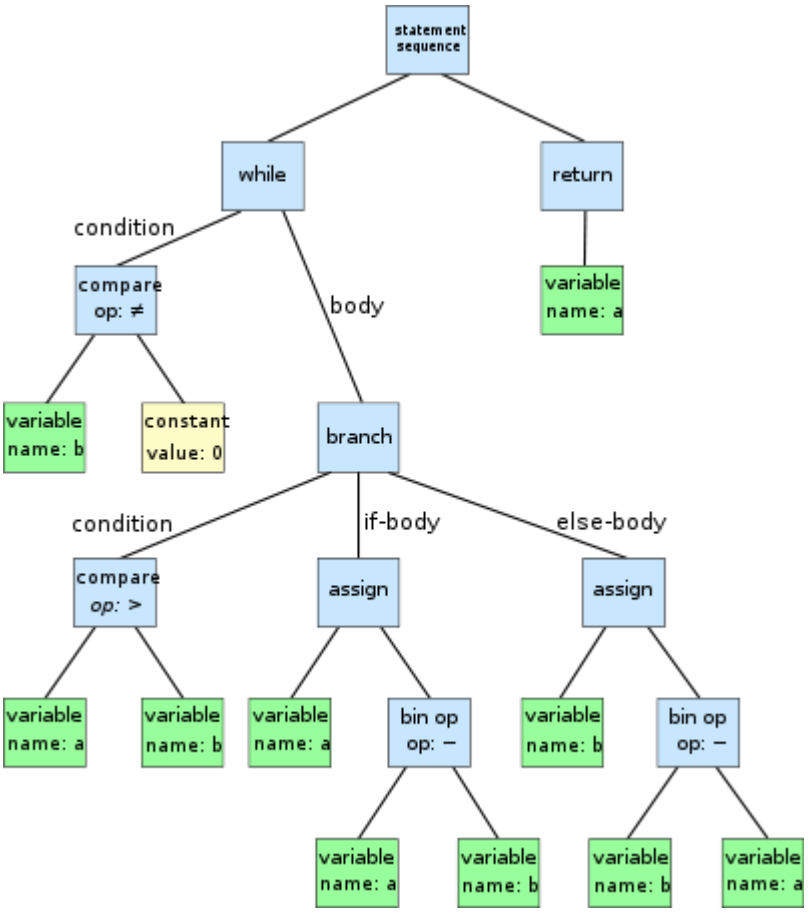

Рисунок 3 Визуальный пример AST (алгоритм Евклида)

- Генерация кода
  В полученном AST узлы были разбиты на подгруппы функций, простых инструкций, примитивов и т. д. Далее происходит генерация строк кода.
  В результате для полученного AST и таблицы символов определяются «токены» (узлы AST) и в зависимости от условий создаются соответствующие объекты (подгруппы функций, простых инструкций, примитивов и т. д.)

5. Процесс тестирования

Для тестирования программы необходимо было собрать проект с dwarf-vm, с помощью утилиты make.

Краткая инструкция по установке:

Setup VM:

git clone https://github.com/sayon/dwarf-vm

cd dwarf-vm

make

Для сборки проекта и получения исполняемого байт-кода можно воспользоваться следующей инструкцией:

cabal install parsec

cabal install data-binary-ieee754

cd compiler-for-dwarf-vm

cabal build
cabal run <src_file.cdv>
vm <built_file.dv>

Результат работы компилятора (проверка корректности компиляции арифметических операций) и результат скомпилированной программы.

Исходный код:

```
int fib (int n) {
    int r1 = 0
    int r2 = 1
    int i = 1
    while (i <= n) {
        int s = r1 + r2
        r1 = r2
        r2 = s
        i = i + 1
    }
    return r1
}

void main() {
        printi(fib(10))
}
```

В результате работы конечный результат: 0 1 1 2 3 5 8 13 21

6. Составление документации

Для конечного пользователя был написан текстовый файл – инструкция (readme), в котором находится краткая инструкция по установке необходимых компонентов для работоспособности компилятора, а также по его использованию.

7. Выводы:

В результате прохождения практики мной был разработан компилятор из простого императивного C-like языка в байт код dwarf-vm. По ходу выполнения я получил множество теоретических знаний и практических навыков в сфере разработки компиляторов. Были освоены основы структуры языков программирования, а также набор машинных инструкций и устройство байт-кода. Структура байт-кода dwarf-vm была опробована в практической части задания.

В результате выполнения работы выяснилось, что в общем случае компилятор состоит из:

1. Лексического анализатора
2. Синтаксического анализатора
3. Семантического анализатора
4. Генератора кода

В каждом из этих этапов исходный код представляется в виде новой структуры, учитывающей синтаксис и семантику языка, а в итоге получим байт-код, соответствующий выбранной машине.

Но способов реализация большое кол-во и каждый из них может привезти к улучшению алгоритма, так, например мной была использована таблица символов, которая позволила ускорить и упростить процесс семантического анализа.

Готовая программа была написана на языке Haskell, что дает оптимальную скорость работы и кроссплатформенность. Также байт-код генерируется для виртуальной машины dwarf-vm, которая генерирует оптимизированный машинный код.

Основным результатом проделанной работы является рабочий компилятор, позволяющий на практически любой системе получать исполняемый байт-код для dwarf-vm, а, следовательно, и машинный код.

Список использованной литературы.

[1] Online лекция Жиркова Игоря Олеговича [Электронный ресурс] // URL: https://docs.google.com/document/d/1wjYwYzXkc_F6H8hdDRbA0Ci0bsCfQN2aeZwd_vlfjSI/edit#

[2] Лекция «Синтаксические анализаторы. Нисходящие анализаторы» [Электронный ресурс] // URL: http://ict.edu.ru/ft/005128/ch6.pdf

[3] Low-Level Programming: C, Assembly, and Program Execution on Intel 64 Architecture by Igor Zhirkov [Электронный ресурс] // URL: https://www.amazon.com/Low-Level-Programming-Assembly-Execution-Architecture/dp/1484224027

[4] Репозиторий проекта dwarf-vm [Электронный ресурс] // URL: https://github.com/sayon/dwarf-vm

[5] Статья «Haskell: комбинаторные парсеры Parsec» от avsmal — habr [Электронный ресурс] // URL: https://dying-sphynx.livejournal.com/66854.html

[6] Статья «Создаём парсер на Haskell» [Электронный ресурс] // URL: https://habr.com/ru/post/50337/

[7] Таблица символов — Wikipedia [Электронный ресурс] // URL: https://bit.ly/2VB5xTz

[8] Лекции по конструированию компиляторов - Глава 7. Организация таблиц символов компилятора [Электронный ресурс] // URL: http://www.codenet.ru/progr/alg/cons/008.php