Министерство науки и высшего образования Российской Федерации
Федеральное государственное автономное образовательное учреждение высшего образования
«Московский государственный технический университет имени Н.Э. Баумана (национальный исследовательский университет)»
(МГТУ им. Н.Э. Баумана)

ФАКУЛЬТЕТ «Информатика и системы управления»

КАФЕДРА «Программное обеспечение ЭВМ и информационные технологии»

РАСЧЕТНО-ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

К КУРСОВОМУ ПРОЕКТУ

НА ТЕМУ:

«Компилятор языка Oberon»

Студент ИУ7И-22М _____ Динь Вьет Ань
(Группа) (Подпись, дата) (И.О.Фамилия)

Руководитель _____ А. А, Ступников
(Подпись, дата) (И.О.Фамилия)

2025 г.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Компилятор — это специализированное программное обеспечение, предназначенное для преобразования исходного кода программы, написанного на определенном языке программирования, в машинный код, который может быть исполнен на целевой платформе. Процесс компиляции включает в себя фазы анализа, оптимизации и генерации кода, обеспечивая эффективную трансляцию программного кода в исполняемый формат [1].

Основной целью данного курсового проекта является разработка прототипа компилятора на основе скорректированной грамматики, использующий библиотеку ANTLR4 для синтаксического анализа входного потока данных и построения AST-дерева.

Для достижения поставленной цели требуется решить следующие задачи.

— Провести анализ грамматики языка Oberon, что позволит полноценно понять его структуру и особенности.

— Изучить существующие инструменты для анализа исходного кода программ, а также системы для генерации низкоуровневого кода, который может быть запущен на широком спектре платформ и операционных систем.

— Разработать прототип компилятора, воплощающий в себе изученные методики и принципы компиляции, а также способный преобразовывать код на языке Oberon в исполняемый формат.

1 Аналитическая часть

1.1. Компоненты компилятора

Компилятор является сложной программной системой, состоящей из нескольких взаимосвязанных компонентов. Типичная архитектура компилятора включает в себя следующие основные составляющие:

1. Frontend (передняя часть) - этот компонент отвечает за преобразование исходного кода программы на высокоуровневом языке в промежуточное представление. Он включает в себя следующие этапы:
   - Препроцессор - выполняет предварительную обработку исходного текста, такую как раскрытие макросов, обработка директив препроцессора и т.д.
   - Лексический анализатор - выполняет разбиение входного текста на лексемы (токены) в соответствии с правилами грамматики.
   - Синтаксический анализатор - строит синтаксическое дерево, основываясь на потоке лексем, полученных на предыдущем этапе.
   - Семантический анализатор - проверяет семантическую корректность программы, определяет типы, области видимости, связывает объявления и использования идентификаторов.
   - Генератор промежуточного представления - на основе синтаксического дерева и результатов семантического анализа генерирует промежуточное представление программы, которое будет использоваться на следующих этапах компиляции.
2. Middle-end (средняя часть) - этот компонент отвечает за оптимизацию промежуточного представления программы. Он выполняет различные анализы и преобразования, направленные на улучшение характеристик генерируемого кода, таких как время выполнения, размер, энергопотребление и т.д.
3. Backend (задняя часть) - этот компонент отвечает за генерацию машинного кода (или ассемблерного) для целевой аппаратной платформы. Он

использует оптимизированное промежуточное представление и выполняет следующие основные этапы:

— Регистровый распределитель - распределяет переменные программы по доступным регистрам процессора.

— Генератор кода - генерирует машинные инструкции, соответствующие промежуточному представлению.

— Оптимизатор кода - выполняет дополнительные оптимизации на уровне машинных инструкций.

Рисунок 1.1 иллюстрирует основные фазы работы компилятора, описанные выше.

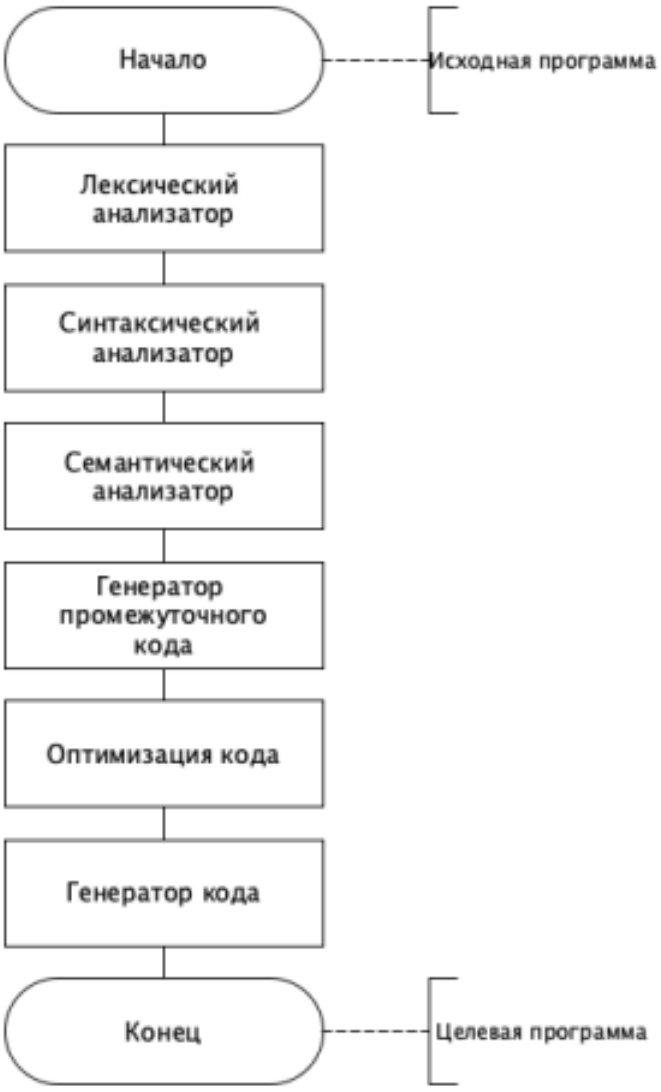

Рисунок 1.1 – Фазы компилятора.

Взаимодействие этих компонентов обеспечивает эффективную трансляцию исходного кода программы в машинно -зависимый двоичный код, готовый к выполнению на целевой аппаратной платформе

1.1.1. Препроцессор

Препроцессор является первым звеном в цепочке компиляции, выполняя ряд важных функций по подготовке входных данных для последующих этапов:

1. Обработка директив препроцессора:
   - Раскрытие макросов с параметрами и без.
   - Включение внешних файлов.
   - Реализация условной компиляции.
   - Управление символами препроцессора.
2. Преобразование исходного текста:
   - Удаление комментариев.
   - Нормализация белых пробелов.
   - Обработка директив языковых расширений.
3. Формирование потока лексем для последующего лексического анализа.

Листинг 1.1 – До препроцессинга

```
#include <stdio.h>

#define MAX_SIZE 100

int main() {
#ifdef DEBUG
    printf("Debugging mode enabled.\n");
#endif
    int arr[MAX_SIZE];
    // Заполнение массива
    for (int i = 0; i < MAX_SIZE; i++) {
        arr[i] = i;
    }
    return 0;
}
```

Рассмотрим пример работы препроцессора (см. Листинг 1) на программе на языке C, которая делает следующее:

— Создает массив arr размером 100 элементов.

— Заполняет этот массив числами от 0 до 99.

— Если определен макрос DEBUG, выводит сообщение ”Debugging mode enabled”.

Листинг 1.2 – После препроцессинга

```
1      int main() {
2      printf("Debugging mode enabled.\n");
3      int arr[100];
4      // Заполнение массива
5      for (int i = 0; i < 100; i++) {
6          arr[i] = i;
7      }
8      return 0;
9  }
```

В результате работы препроцессора (см. Листинг 1.2):

1. Директива #include <stdio.h> была заменена на подключение соответствующей библиотеки.

2. Макрос #define MAX_SIZE 100 был раскрыт, заменив все вхождения MAX_SIZE на значение 100.

3. Директива #ifdef DEBUG и соответствующий блок вывода были сохранены, так как условие DEBUG было определено.

4. Комментарии остались без изменений.

Таким образом, препроцессор выполнил обработку директив, раскрытие макросов и подготовил финальный исходный код для последующих этапов компиляции.

Следовательно, препроцессор играет ключевую роль в подготовке и трансформации исходного кода программы, обеспечивая необходимую входную информацию для следующих этапов компиляции.

1.1.2. Лексический анализатор

Лексический анализ – это критическая процедура в разработке компилятора, отвечающая за преобразование входного потока символов в последовательность токенов. Этот процесс, также известный как ”токенизация группирует

определенные терминальные символы в лексемы в соответствии с заданными правилами, обычно выраженными через регулярные выражения или конечные автоматы.

Основные функции лексического анализатора включают:

– Удаление пробелов и комментариев из входного потока. Например, строка x = 10. 10 будет преобразована в последовательность токенов без комментария.

– Идентификация и формирование числовых констант из последовательностей цифр. Например, 42 или 3.14 будут распознаны как числовые литералы.

– Распознавание идентификаторов и ключевых слов языка. Например, var, if, return будут определены как ключевые слова языка, а myVariable как идентификатор.

Рисунок 1.2 – Лексический анализатор.

Лексический анализатор находится между входным потоком и синтаксическим анализатором, как показано на рисунке 1.2. Он считывает символы из входного потока, группирует их в лексемы и передает последующим стадиям 9компиляции токены, образованные из этих лексем.

Например, последовательность символов var x = 42; будет разбита лексическим анализатором на следующие токены:

— var (ключевое слово)

— x (идентификатор)

— = (оператор присваивания)

— 42 (числовая константа)

— ; (разделитель)

Лексический анализ может рассматриваться как один из этапов синтаксического анализа, но это также самостоятельная задача, ответственная за обнаружение и устранение лексических ошибок, таких как недопустимые символы, ошибки в идентификаторах или числовых константах. Эффективный лексический анализатор является фундаментом для последующих стадий компиляции, обеспечивая надежную и точную обработку входного языка.

1.1.3. Синтаксический анализатор

Синтаксический анализ (или разбор) — 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.

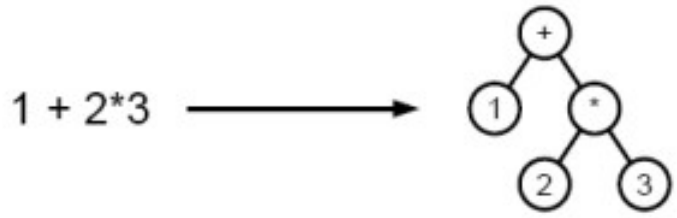

Рисунок 1.3 – Пример разбора выражения с преобразованием его структуры из линейной в древовидную.

В ходе синтаксического анализа исходный текст преобразуется в структуру данных, обычно — в дерево, которое отражает синтаксическую структуру входной последовательности и хорошо подходит для дальнейшей обработки. На Рисунке 1.3 представлен пример разбора выражения с преобразованием его структуры из линейной в древовидную.

Как правило, результатом синтаксического анализа является синтаксическое строение предложения, представленное либо в виде дерева зависимостей, либо в виде дерева разбора (см. Рисунок 1.4), либо в виде некоторого сочетания

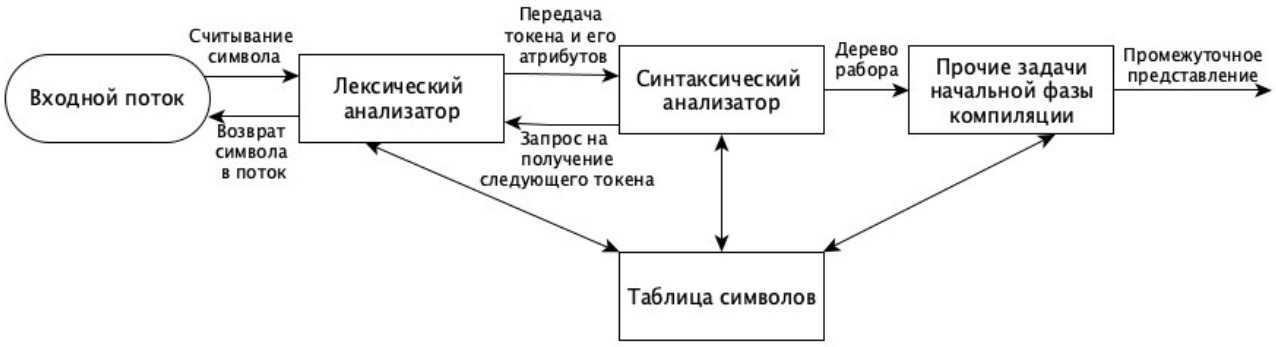

Рисунок 1.4 – Синтаксический анализатор.

первого и второго способов представления, где каждый внутренний узел является оператором, а дочерние – его аргументами. Среди них можно выделить несколько групп связанных объектов:

— элементы арифметических выражений: каждый узел представляет собой операцию и содержит её аргументы;

— элементы системы типов: базовые типы (числовые, строковые, структуры и т.п.), указатели, массивы и функции;

— выражения пяти типов: арифметические, блочные и управляющие выражения, условные конструкции, циклы.

Синтаксический анализатор использует следующие типы алгоритмов, выбор которого зависит от сложности и особенностей используемой грамматики:

— Нисходящий парсер — продукции грамматики раскрываются, начиная со стартового символа, до получения требуемой последовательности токенов.

  + Метод рекурсивного спуска
  + LL-анализатор

— Восходящий парсер — продукции восстанавливаются из правых частей, начиная с токенов и заканчивая стартовым символом.

  + LR-анализатор
  + GLR-парсер

Перед обработчиком ошибок синтаксического анализатора стоят следующие основные задачи, сочетающие выявление ошибок и корректную обработку данных:

— Четко и точно сообщать о наличии ошибок в анализируемой входной последовательности.

— Обеспечивать быстрое восстановление после обнаружения ошибки, чтобы можно было продолжить поиск других ошибок.

— Не замедлять существенно обработку корректной входной последовательности.

Таким образом, синтаксический анализ закладывает фундамент для всего процесса компиляции, формируя структурное представление исходной программы. Эффективная реализация синтаксического анализатора является ключевой для производительности и корректности работы компилятора в целом.

1.1.4. Семантический анализатор

Этап семантической валидации в компиляции выполняет проверку корректности смысловых связей в коде программы и собирает данные о типизации для последующего этапа компиляции — генерации исполняемого кода. Для этого используются структуры данных, сформированные на этапе синтаксического анализа, которые позволяют определить связи между операторами и операндами в выражениях и командах.

Модуль семантической валидации обычно состоит из нескольких подмодулей, каждый из которых специализируется на проверке определённого вида конструкций. Каждый подмодуль активируется синтаксическим анализатором, когда тот обнаруживает конструкцию, требующую семантической проверки.

Подмодули семантической валидации обмениваются информацией через специализированные структуры данных, такие как таблица символов, что обеспечивает их взаимодействие и координацию.

Приведем несколько общих примеров использования семантического анализатора:

— **Проверка объявления переменных**

Семантический анализатор проверяет, что каждая переменная, использу-

емая в программе, была корректно объявлена. Он отслеживает информацию о типах, областях видимости и других атрибутах переменных, хранящуюся в таблице символов.

Например, при обнаружении обращения к неописанной переменной, семантический анализатор сгенерирует сообщение об ошибке.

— **Проверка согласованности операций**

Семантический анализатор также проверяет, что операции, выполняемые над операндами, корректны с точки зрения их типов.

Так, при попытке сложить число и строку, анализатор выявит несоответствие типов и сообщит об ошибке.

— **Проверка областей видимости**

Анализатор контролирует, чтобы переменные использовались только в тех областях, где они были объявлены. Он отслеживает вложенность блоков кода и правила управления областями видимости.

Например, при попытке обратиться к локальной переменной за пределами ее блока, анализатор сгенерирует сообщение об ошибке ”использование недоступной переменной”.

Теперь рассмотрим конкретный сложный пример с фрагментом кода, в котором используются шаблоны проектирования и обобщённое программирование (см. Листинг 1.3):

Листинг 1.3 – Фрагмент кода с шаблонами проектирования и обобщенным программированием

```
#include <stdio.h>
template<typename T>
T max(T a, T b) {
    return a > b ? a : b;
}

int main() {
    int i = max(3, 7);
    double d = max(6.34, 3.12);
    std::string s = max(std::string("apple"), std::string("orange"));
}
```

В данном примере на этапе семантической валидации анализатор должен убедиться, что функция max может быть применена к различным типам данных. Для этого он проверяет, что оператор сравнения > поддерживается для

каждого из типов T, передаваемых в функцию. Кроме того, анализатор должен проверить, что для каждого вызова функции max существуют соответствующие типы аргументов, которые могут быть сравнены, и что результат сравнения может быть возвращён из функции.

Таким образом, семантический анализатор играет ключевую роль в обеспечении смысловой корректности анализируемой программы, выявляя широкий спектр логических ошибок на ранних этапах трансляции.

1.1.5. Генерация кода

Последний этап – генерация кода. Начинается тогда, когда во все системные таблицы занесена необходимая информация. В этом случае, компилятор переходит к построению соответствующей программы в машинном коде. Код генерируется при обходе дерева разбора, построенного на предыдущих этапах.

Для получения машинного кода требуется два отдельных прохода:

— генерация промежуточного кода;

— генерация собственно машинного кода.

Для каждого узла дерева генерируется соответствующий операции узла код на целевой платформы. В процессе анализа кода программы данные связываются с именами переменных. При выполнении генерации кода предполагается, что вход генератора не содержит ошибок. Результат – код, пригодный для исполнения на целевой платформе.

В качестве примера рассмотрим трансляцию следующего выражения $a = b * -c + b * -c$ в промежуточный и машинный код. Предположим, что переменные a, b, и c уже определены и расположены в памяти.

Листинг 1.4 – Шаг 1. Генерация промежуточного кода. Выражение преобразуется в последовательность трехадресных инструкций

```
1 t1 = -c
2 t2 = b * t1
3 t3 = -c
4 t4 = b * t3
5 a = t2 + t4
```

Листинг 1.5 – Шаг 2. Оптимизация промежуточного кода. Устраняем повторяющиеся вычисления

```
t1 = -c
t2 = b * t1
a = t2 + t2
```

Листинг 1.6 – Шаг 3. Генерация машинного кода. Промежуточный код транслируется в инструкции для конкретной целевой машины

```
LOAD R1, c
NEG R1
MUL R2, b, R1
ADD R3, R2, R2
STORE a, R3
```

Этот пример иллюстрирует ключевые этапы трансляции: от промежуточного представления выражений до их физического выполнения на аппаратном уровне.

1.2. Методы реализации лексического и синтаксического анализаторов

В современной разработке компиляторов применяются разнообразные методы для создания лексических и синтаксических анализаторов. Эти методы можно классифицировать как:

— Традиционные алгоритмы, основанные на проверенных временем техниках;

— Современные инструменты, автоматизирующие процесс генерации.

1.2.1. Генераторы лексического анализатора

На рынке представлено множество инструментов для генерации лексических анализаторов, среди которых выделяются Lex, Flex, ANTLR4 и другие. [2]

Lex: Этот инструмент является классическим решением в среде Unix и часто используется в паре с Yacc для создания синтаксических анализаторов. Lex обрабатывает входные данные и генерирует исходный код на C, структурированный в три секции: определения, правила и код на C.

Flex: Как усовершенствованная версия Lex, Flex используется в GNUсистемах и предлагает схожую функциональность.

ANTLR4: Этот инструмент поддерживает широкий спектр языков программирования и обеспечивает генерацию классов для рекурсивного нисходящего синтаксического анализа. ANTLR4 позволяет создавать анализаторы, основанные на РБНФ грамматике, и предоставляет возможности для построения и обхода деревьев анализа.

Сравнение инструментов для лексического анализа выделяет ANTLR4 как наиболее предпочтительный инструмент по следующим причинам:

— Универсальность: ANTLR4 поддерживает широкий спектр целевых языков программирования, что делает его идеальным выбором для проектов, требующих кросс-платформенной совместимости. [3]

— Продвинутые возможности: В отличие от Lex и Flex, ANTLR4 предлагает более продвинутые возможности для работы с грамматикой, включая поддержку РБНФ, что позволяет создавать более сложные и мощные анализаторы.

— Простота использования: ANTLR4 упрощает процесс разработки анализаторов благодаря интуитивно понятным абстракциям и шаблонам, что сокращает время разработки и упрощает поддержку кода.

— Эффективность: Генерируемые ANTLR4 анализаторы характеризуются высокой производительностью и оптимизацией, что критически важно для компиляторов и других инструментов анализа кода.

В то время как Lex и Flex остаются надежными инструментами, особенно в Unix-подобных системах, их функциональность ограничена по сравнению с ANTLR4. ANTLR4 предоставляет более широкие возможности и гибкость, что делает его предпочтительным выбором для современных проектов по созданию лексических и синтаксических анализаторов.

1.2.2. Генераторы синтаксического анализатора

В сфере разработки парсеров ключевым аспектом является выбор инструментария, который определяет эффективность и гибкость создаваемого решения. В этом контексте выделяются следующие инструменты:

Yacc/Bison: Эти генераторы парсеров трансформируют грамматические определения в исполняемый код на C, обеспечивая интеграцию с системами

Unix (Yacc) и GNU (Bison). [4]

Coco/R: Платформа для генерации парсеров, поддерживающая множество языков программирования, включая C++ и Java, использует концепции конечных автоматов для лексического анализа и рекурсивного спуска для синтаксического.

Применение конечных автоматов позволяет лексическим анализаторам эффективно распознавать лексемы, в то время как рекурсивный спуск обеспечивает точность синтаксического анализа, необходимую для обработки вложенных структур и сложных грамматик. ANTLR, интегрируя эти методы, предоставляет разработчикам мощный инструментарий для создания высокопроизводительных парсеров, способных работать с разнообразными грамматическими конструкциями.

1.3. LLVM

LLVM (Low Level Virtual Machine) – проект программной инфраструктуры для создания компиляторов и сопутствующих им утилит. В его основе лежит платформонезависимая система кодирования машинных инструкций – байткод LLVM IR (Intermediate Representation). LLVM может создавать байткод для множества платформ, включая ARM, x86, x86-64, GPU от AMD и Nvidia и другие. В проекте есть генераторы кода для множества языков, а для компиляции LLVM IR в код платформы используется clang. В состав LLVM входит также интерпретатор LLVM IR, способный исполнять код без компиляции в код платформы. [5]

Некоторые проекты имеют собственные LLVM-компиляторы, например, LLVM-версия GCC.

LLVM поддерживает целые числа произвольной разрядности, числа с плавающей точкой, массивы, структуры и функции. Большинство инструкций в LLVM принимает два аргумента (операнда) и возвращает одно значение (трёхадресный код).

Значения в LLVM определяются текстовым идентификатором. Локальные значения обозначаются префиксом %, а глобальные – @. Тип операндов всегда указывается явно и однозначно определяет тип результата. Операнды арифметических инструкций должны иметь одинаковый тип, но сами инструкции «перегружены» для любых числовых типов и векторов.

LLVM поддерживает полный набор арифметических операций, побито-

вых логических операций и операций сдвига. LLVM IR строго типизирован, 18поэтому существуют операции приведения типов, которые явно кодируются специальными инструкциями. Кроме того, существуют инструкции преобразования между целыми числами и указателями, а также универсальная инструкция для приведения типов bitcast.

Помимо значений регистров в LLVM есть работа с памятью. Значения в памяти адресуются типизированными указателями. Обратиться к ней можно с помощью двух инструкций: load и store. Инструкция alloca выделяет память на стеке. Она автоматически освобождается при выходе из функции при помощи инструкций ret или unwind.

Для вычисления адресов элементов массивов и структур с правильной типизацией используется инструкция getelementptr. Она только вычисляет адрес без обращения к памяти, принимает произвольное количество индексов и может разыменовывать структуры любой вложенности.

Выводы

В данном разделе приведён обзор основных фаз компиляции, описана каждая из них. Также был выбран генератор лексического и синтаксического анализаторов – ANTLR и LLVM в качестве генератора машинного кода.

2 Конструкторская часть

2.1. IDEF0

Концептуальная модель разрабатываемого компилятора в нотации IDEF0 представлена на рисунках 2.1.

Рисунок 2.1 – Концептуальная модель в нотации IDEF0 (A1-A6).

2.2. Язык Oberon

Oberon – язык программирования высокого уровня, предназначенный для исполнения программ на одноимённой операционной системе и основанный на таких языках, как Modula-2, Pascal.

Основные свойства и особенности языка Oberon:

— **Простота и лаконичность**: Oberon известен своей минималистичной и лаконичной синтаксической структурой. Язык был разработан так, чтобы быть легким для восприятия и использования, что делает его подходящим как для обучения, так и для написания эффективного системного программного обеспечения.

— **Модульная структура**: Как и его предшественники, Oberon поддерживает модульную структуру программ. Модули являются основной единицей компиляции и разделения кода, что способствует разработке более структурированных и легко поддерживаемых программ.

— **Статическая типизация**: Oberon использует статическую типизацию, предоставляя компилятору полную информацию о типах данных во время компиляции. Это помогает обнаруживать ошибки типов на ранних этапах разработки и повышает надежность программ.

— **Поддержка системы типов и строгие проверки**: Язык обеспечивает строгую проверку типов, включая проверки на совместимость и преобразования типов. Это уменьшает количество ошибок времени выполнения и улучшает безопасность кода.

— **Ассоциация с операционной системой Oberon**: Язык тесно интегрирован с операционной системой Oberon, которая была разработана для демонстрации эффективного использования языка и его возможностей. Операционная система Oberon предоставляет среду, в которой программы на данном языке могут работать наиболее эффективно.

— **Гарвардская архитектура и интеграция с конкретным оборудованием**: Oberon был разработан с учетом особенностей гарвардской архитектуры, где программы и данные хранятся раздельно. Это помогает оптимизировать выполнение программ и использование памяти. Краткий пример кода на языке Oberon приведен в Листинге 10, где:

— MODULE HelloWorld; — объявляет новый модуль с именем HelloWorld.

— IMPORT Out; — импортирует модуль Out, который содержит процедуры для вывода текста.

— PROCEDURE Main*; — объявляет основную процедуру Main, помеченную звездочкой, что означает, что эта процедура экспортируется и может быть вызвана извне.

— BEGIN ... END; — определяет тело процедуры, в котором вызывается процедура Out.String для печати строки на экран, за которой следует вызов Out.Ln, который переводит

Листинг 2.1 – Краткий пример кода на языке Oberon

```
MODULE HelloWorld;
IMPORT Out;

PROCEDURE Main*;
BEGIN
Out.String("Hello, World!"); Out.Ln;
END Main;

END HelloWorld.
```

Грамматика языка Oberon формально описана в Приложении А и включает в себя правила синтаксиса для всех конструкций языка, таких как объявления модулей, процедур и типов данных. Этот раздел является ключом к пониманию внутренней структуры языка и его компиляции.

2.3. Лексический и синтаксический анализаторы

Лексический и синтаксический анализаторы в данной работе генерируются с помощью ANTLR. На вход поступает грамматика языка в формате ANTLR4 (файл с расширением .mod).

В результате работы создаются файлы, содержащие классы лексера и парсера, а также вспомогательные файлы и классы для их работы. Также генерируются шаблоны классов для обхода дерева разбора, которое получается в результате работы парсера.

На вход лексера подаётся текст программы, преобразованный в поток символов. На выходе получается поток токенов, который затем подаётся на вход парсера. Результатом его работы является дерево разбора.

Ошибки, возникающие в ходе работы лексера и парсера, выводятся в стандартный поток ввода-вывода.

2.4. Семантический анализ

Абстрактное синтаксическое дерево можно обойти двумя способами: применяя паттерн Listener или Visitor.

Listener позволяет обходить дерево в глубину и вызывает обработчики соответствующих событий при входе и выходе из узла дерева.

Visitor предоставляет возможность более гибко обходить построенное дерево и решить, какие узлы и в каком порядке нужно посетить. Таким образом, для каждого узла реализуется метод его посещения. Обход начинается с точки входа в программу (корневого узла).

Выводы

В текущем разделе была представлена концептуальная модель в нотации IDEF0, приведена грамматика языка Oberon, описаны принципы работы лексического и синтаксического анализаторов и идея семантического анализа.

3 Технологическая часть

3.1. Выбор средств программной реализации

В качестве языка программирования была выбрана Python, ввиду нескольких причин.

— Расширенная библиотека стандартных модулей и сторонние библиотеки: Python предоставляет широкий спектр встроенных модулей и сторонних библиотек, которые облегчают разработку компилятора. В частности, библиотеки для парсинга (например, PLY) и работы с абстрактными синтаксическими деревьями (AST) делают разработку компилятора более простой и быстрой.

— Простота и читаемость кода: Python известен своей простотой и читаемостью, что особенно важно при разработке и поддержке сложных проектов, таких как компиляторы. Это позволяет легче понимать и изменять код, что сокращает время разработки и уменьшает количество ошибок.

— Динамическая типизация и быстрые прототипы: Python поддерживает динамическую типизацию, что позволяет быстрее создавать прототипы и тестировать новые идеи. Это особенно полезно на ранних стадиях разработки компилятора.

— Накопленный опыт и существующий код: На момент реализации уже был накоплен существенный опыт в использовании Python, а также существующий код, который можно было использовать и адаптировать для нового проекта. Это существенно сократило бы время и затраты на обучение и разработку.

— Кросс-платформенность: Python работает на различных операционных системах, включая Windows, macOS и Linux, что делает его универсальным инструментом для разработки кросс-платформенных приложений.

3.2. Сгенерированные классы анализаторов

В результате работы ANTLR генерируются следующие файлы.

1. Oberon.interp и OberonLexer.interp содержат данные (таблицы предсказания, множества следования, информация о правилах грамматики и т.д.) для интерпретатора ANTLR, используются для ускорения работы сгенерированного парсера для принятия решений о разборе входного потока.

2. Oberon.tokens и OberonLexer.tokens перечислены символические имена токенов, каждому из которых сопоставлено числовое значение типа токена. ANTLR4 использует их для создания отображения между символическими именами токенов и их числовыми значениями.

3. Основные модули для компиляции и исполнения в папках compiler и global_ops, где папка compile отвечает за инициализацию, декодирование и запуск модулей, а папка global_ops содержит константы, классы и функции для работы с объектами, типами и инструкциями.

4. Папка complier содержит файл с функциями:

   — Compile() – основной процесс компиляции включает в себя инициализацию исходного модуля и запуск модуля. Используется библиотека OSS для инициализации модуля и библиотека OSP для запуска процесса компиляции.

   — Decode() – функция для декодирования инструкций. Используется библиотека OSG.

   — Load() – функция для загрузки модуля. Проверяет наличие ошибок перед загрузкой, если ошибок нет и модуль ранее не был загружен, загружает модуль используя библиотеку OSG, а также обновляет состояние переменной loaded.

   — Exec(S) – функция для выполнения декодированных инструкций. Использует библиотеку OSG.

5. Папка global_ops содержит файл с классами и функциями:

   — **Item** – класс для описания элементов с различными полями, такими как режим, уровень, тип данных, адрес, и другие характеристики.

— **ObjDesc** – класс для описания объектов с различными свойствами, такими как класс объекта, уровень вложенности, следующий объект в цепочке, тип объекта, имя и значение.

— **TypeDesc** – класс для описания типов данных, которые включают форму данных, поля, базовый тип, размер и длину.

— Глобальные переменные, такие как intType, boolType, curlev, pc, и массивы для хранения разметки и инструкций:

  + intType, boolType – указатели на структуры описания типов для целых чисел и булевых значений.
  + curlev, pc, relx, cno – переменные для отслеживания текущего уровня вложенности, счётчика программ, указателя на текущую команду, и счётчика команд.
  + regs – множество используемых регистров.
  + code – массив для хранения кодов инструкций.
  + rel – массив для хранения информации о относительных адресах.
  + comname, comadr – массивы для хранения имён и адресов команд.

— Функции для работы с регистрами, генерации инструкций и обработки операций:

  + IncLevel(n) – увеличивает текущий уровень вложенности на заданное значение.
  + MakeConstItem(x, Type, val) – создает элемент-константу с заданным типом и значением.
  + MakeItem(x, y) – создает элемент на основе описания объекта.
  + Field(x, y) – обновляет элемент на основе поля объекта.
  + Index(x, y) – обновляет элемент на основе индексации массива.
  + Open() – начальная инициализация глобальных переменных, таких как уровень вложенности и счётчик программ.
  + Close(S, globals) – завершение процедуры, включающее финальные инструкции (например, возврат из функции).
  + GetReg(r) – получение свободного регистра.

+ Put(op, a, b, c) – генерация инструкции с заданными операцией и аргументами.
+ TestRange(x) – проверка значения на допустимый диапазон.
+ Header(size) – создание заголовка в коде из указанных размеров.
+ Enter(size) – функция для входа в новую процедуру с указанием размера области.
+ EnterCmd(name) – сохранение команды с заданным именем.

6. Папка constants содержит все константы.
7. Папка processor содержит все опепаторы.
8. Папка file_io содержит функции работы с файлами.
9. Папка keywords содержит все ключевые слова языка Oberon.
10. Папка output_executable содержит результат программы.

Таким образом, представленные модулю предоставляют полный набор средств для компиляции и выполнения кода на языке Oberon, включая инициализацию, декодирование, загрузку, выполнение инструкций, а также управление параметрами компиляции и выполнения программ.

3.3. Тестирование

Для проверки корректной работы программы был написан класс TestMethod в файле test.py, который наследуется от unittest.TestCase. В этом классе определены методы для тестирования различных функций компилятора. Тесты используют файл с исходным кодом программы на языке Oberon, находящийся по адресу tests/data/source.mod, и проверяют корректность работы компилятора и выполнения скомпилированного кода.

Каждый метод теста начинает с компиляции, декодирования и загрузки исходного файла. Затем вызывается метод Exec() для выполнения конкретной тестовой команды, и результат сравнивается с ожидаемым результатом при помощи метода assertEqual().

Листинг 3.1 – Пример части класса тестирования

```
1 c
2 lass TestMethods(unittest.TestCase):
```

```
3
4  def test_multiply_procedure(self):
5          compiler.Compile(filename="tests/data/source.mod")
6          compiler.Decode()
7          compiler.Load()
8          self.assertEqual(compiler.Exec("Multiply 5 5"), " 0 40 25")
9          self.assertEqual(compiler.Exec("Multiply 0 0"), " 0 0 0")
10         self.assertEqual(compiler.Exec("Multiply -1 -1"), " -1 -1 0"
       )
11         self.assertEqual(compiler.Exec("Multiply 0 -1"), " 0 -1 0")
12         self.assertEqual(compiler.Exec("Multiply 1000 9999"), " 0
       102389769999000")
```

Рисунок 3.1 иллюстрирует вывод результатов тестирования.

Test #	Test Name	Status
1	Binary Search Test	Passed
2	Bubble Sort Test	Passed
3	Division Test	Passed
4	Factorial Test	Passed
5	Fibonacci Sequence Test	Passed
6	Multiplication Test	Passed
7	Power of Two Test	Passed
8	Array Reversal Test	Passed
9	Series Sum Test	Passed
10	Element Swap Test	Passed

Рисунок 3.1 – Вывод результатов тестирования.

ЗАКЛЮЧЕНИЕ

Таким образом, в рамках текущей курсовой работы рассмотрены основные части компилятора, алгоритмы и способы их реализации. Также были рассмотрены инструменты генерации лексических и синтаксических анализаторов.

Был разработан прототип компилятора языка Oberon, использующий ANTLR для синтаксического анализа входного потока данных и построения AST-дерева, и LLVM для последующих преобразований, переводящих абстрактное дерево в IR.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. *Ахо А.* Компиляторы: принципы, технологии и инструменты. — М.: Вильямс, 2008.
2. *Lesk M. E.*, *Schmidt E. L.* A lexical analyzer generator. — Murray Hill, NJ : Bell Laboratories, 1975.
3. What is ANTLR? [Электронный ресурс]. – Режим доступа: https://www.antlr.org/ (Дата обращения: 25.04.2025).
4. *C. D.* Bison: The Yacc-compatible Parser Generator. — Samurai Media Limited, 2015.
5. The LLVM Compiler Infrastructure Project [Электронный ресурс]. – Режим доступа: https://llvm.org/ (Дата обращения: 27.04.2025).

Листинг 3.2 – Грамматика языка Oberon

```
grammar oberon;

ident: IDENT;
qualident: ident;
identdef: ident '*'?;

integer: (DIGIT+);
real: DIGIT+ '.' DIGIT* ;
number: integer | real;

constDeclaration: identdef '=' constExpression;
constExpression: expression;

typeDeclaration: identdef '=' type_;
type_ : qualident | arrayType;
arrayType: ARRAY length OF type_;

length: constExpression;

identList: identdef (',' identdef)*;
variableDeclaration: identList ':' type_;

expression: simpleExpression (relation simpleExpression)?;
relation: '=' | '#'| '<'| '<='| '>'| '>=';
simpleExpression: ('+' | '-')? term (addOperator term)*;
addOperator: '+'| '-'| OR;
term: factor (mulOperator factor)*;
mulOperator: '*'| '/'| DIV| MOD| '&';

factor: number| STRING| designator (actualParameters)?| '('
   expression ')'| '~' factor;
designator: qualident selector*;
selector: '[' expList ']';
expList: expression (',' expression)*;
actualParameters: '(' expList? ')';
statement: (assignment| ifStatement| whileStatement| forStatement
   )?;
assignment: designator ':=' expression;
statementSequence: statement (';' statement)*;
ifStatement: IF expression THEN statementSequence (ELSIF
   expression THEN statementSequence)* (ELSE statementSequence)?
   END;
whileStatement: WHILE expression DO statementSequence (ELSIF
   expression DO statementSequence)* END;
forStatement: FOR ident ':=' expression TO expression (BY
   constExpression)? DO statementSequence END;
declarationSequence: (CONST (constDeclaration ';')*)? (TYPE (
   typeDeclaration ';')*)? (VAR (variableDeclaration ';')*)?;

module: MODULE ident ';' declarationSequence (BEGIN
   statementSequence)? RETURN factor ';' END ident '.' EOF;

ARRAY: 'ARRAY';
OF: 'OF';
END: 'END';
TO: 'TO';
OR: 'OR';
DIV: 'DIV';
MOD: 'MOD';
```

```
52 IF: 'IF';
53 THEN: 'THEN';
54 ELSIF: 'ELSIF';
55 ELSE: 'ELSE';
56 WHILE: 'WHILE';
57 DO: 'DO';
58 FOR: 'FOR';
59 BY: 'BY';
60 BEGIN: 'BEGIN';
61 RETURN: 'RETURN';
62 TYPE: 'TYPE';
63 VAR: 'VAR';
64 MODULE: 'MODULE';
65 STRING: ('"' .*? '"') | (DIGIT HEXDIGIT* 'X');
66 IDENT: LETTER (LETTER | DIGIT)*;
67 LETTER: [a-zA-Z];
68 DIGIT: [0-9];
69 COMMENT: '(*' .*? '*)' -> skip;
70 WS: [ \t\r\n] -> skip;
```