Федеральное государственное автономное
образовательное учреждение
высшего образования
«СИБИРСКИЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»

Институт космических и информационных технологий
институт

Вычислительная техника
кафедра

УТВЕРЖДАЮ
Заведующий кафедрой
_____ О.В. Непомнящий
подпись инициалы, фамилия
«____» _____ 20___ г.

БАКАЛАВРСКАЯ РАБОТА

09.03.01 Информатика и вычислительная техника

Базовая часть транслятора экспериментального языка программирования на основе Oberon-07

Руководитель	_____	доцент, канд.техн.наук	Д.А. Швец
	подпись, дата	должность, ученая степень	инициалы, фамилия
Выпускник	_____		П.С. Коваленко
	подпись, дата		инициалы, фамилия
Нормконтролер	_____	доцент, канд.техн.наук	Д.А. Швец
	подпись, дата	должность, ученая степень	инициалы, фамилия

Красноярск 2021

Федеральное государственное автономное
образовательное учреждение
высшего образования
«СИБИРСКИЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»

Институт космических и информационных технологий
институт

Вычислительная техника
кафедра

УТВЕРЖДАЮ
Заведующий кафедрой
_____ О.В. Непомнящий
подпись инициалы, фамилия
«____ » _____ 20___ г.

ЗАДАНИЕ
НА ВЫПУСКНУЮ КВАЛИФИКАЦИОННУЮ РАБОТУ
В форме бакалаврской работы

Студенту Коваленко Павлу Сергеевичу
фамилия,имя,отчество

Группа КИ17-07Б Направление (специальность) 09.03.01
номер код

Информатика и вычислительная техника
наименование

Тема выпускной квалификационной работы Базовая часть транслятора экспериментального языка программирования на основе Oberon-07

Утверждена приказом по университету №_____ от _____

Руководитель ВКР Д.А.Швец, руководитель НУЛ ТП, доцент кафедры ВТ ИКИТ СФУ, канд. техн. наук
инициалы, фамилия, должность, ученое звание и место работы

Исходные данные для ВКР: Реализовать базовую часть транслятора экспериментального языка программирования на основе Oberon-07 с процедурно-параметрическими расширениями.

Перечень разделов ВКР: Описание предметной области, выбор методов реализации, описание разработанного приложения.

Перечень графического материала: презентация в формате PDF, раздаточный материал.

Руководитель ВКР _____ Д.А. Швец
подпись, дата инициалы, фамилия

Задание принял к исполнению _____ П.С. Коваленко
подпись, дата инициалы, фамилия

«____» _____ 20___ г.

РЕФЕРАТ

Выпускная квалификационная работа по теме «Базовая часть транслятора экспериментального языка программирования на основе Oberon-07» содержит 26 страниц текстового документа, 4 иллюстрации, 11 использованных источников, 2 приложения.

ТРАНСЛЯТОР, СИНТАКСИЧЕСКИЙ АНАЛИЗАТОР, СЕМАНТИЧЕСКИЙ АНАЛИЗАТОР, ПАРСЕР-КОМБИНАТОР, МУЛЬТИМЕТОД, МНОЖЕСТВЕННАЯ ДИСПЕТЧЕРИЗАЦИЯ, ПРОМЕЖУТОЧНОЕ ПРЕДСТАВЛЕНИЕ, ТРЕХАДРЕСНЫЙ КОД

Цель работы: создание базовой части транслятора для экспериментального императивного языка программирования семейства Oberon-07 с процедурно-параметрическими расширениями.

Задачи работы:

– Реализация лексического анализатора;

– Реализация синтаксического анализатора;

– Реализация генератора промежуточного представления.

Во введение раскрывается актуальность работы, ставятся цели и задачи.

В первой главе приведен обзор предметной области.

Во второй главе проведен выбор инструментов для реализации компонентов разрабатываемого приложения.

В третьей главе содержится описание разработанного приложения.

В результате работы создана реализация базовой части транслятора языка программирования на основе Oberon-07.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

В современном мире программная инженерия и разработка прикладного программного обеспечения, в частности, является одной из самых бурно развивающихся областей. При этом с каждым годом растут требования к надежности и расширяемости создаваемых программных систем. Это, в свою очередь, подталкивает развитие языков программирования, так как именно язык программирования выбранный для разработки того или иного прикладного ПО задает базис надежности, расширяемости и других характеристик ПО.

Существует множество экспериментальных языков программирования. В них происходит проверка новых возможностей которые теоретически могут повысить те или иные характеристики ПО. При этом некоторые из этих возможностей удачно прошедших проверку временем в экспериментальных языках активно заимствуются языками общего пользования. Например, анонимные функции (лямбда-функции), некогда присутствовавшие только в функциональных языках программирования, сейчас доступны в Java, C#, C++ и во множестве других популярных языков.

Язык программирования Oberon это императивный язык программирования, родственный языку Pascal, первая версия которого была разработана 1986 шведским ученым Никлаусом Виртом. При разработке языка Oberon его создатель следовал принципу: «Сделай все настолько простым, насколько возможно, но не проще» (А. Эйнштейн). Поэтому язык содержит небольшое количество встроенных конструкций и типов, что делает его сравнительно более простым по сравнению с другими современными языками программирования. Oberon-07 — это развитие классического языка Oberon, из него были убраны некоторые конструкции, что еще сильнее упростило язык.

Целью данной выпускной квалификационной работы является создание базовой части транслятора для экспериментального императивного языка программирования семейства Oberon-07 с процедурно-параметрическими расширениями.

Для достижения поставленной цели необходимо решить следующие задачи:

– Реализация лексического анализатора;
– Реализация синтаксического анализатора;
– Реализация генератора промежуточного представления.

1 Описание предметной области

1.1 Язык программирования Oberon-07

Oberon-07 — это универсальный язык программирования созданный Никлаусом Виртом в 2007 году на основе его предшественника языка программирования Oberon-2. Oberon обладает ALGOL-подобным синтаксисом и является попыткой переработать язык Modula-2 с целью уменьшения его сложности. Oberon-07 значительно отличается от языка Oberon-2, в основном добавлением дополнительных ограничений, например: обязательное наличие только одного RETURN в конце функциональной процедуры, строгая типизация и явное приведение типов чисел через встроенные функции. Синтаксис языка в РБНФ [1] представлен в Приложении А. Основными особенностями языка являются:

– Строгая статическая типизация;

– Возможность расширения типов-записей, что аналогично наследованию типов в объектно-ориентированных языках, но без виртуальных методов;

– Система модулей позволяющая скрывать любые символы от других модулей, модуль может хранить состояние, но для других модулей все экспортированные символы доступны только по чтению, аналог инкапсуляции и сокрытия в объектно-ориентированных языках;

– Автоматическая сборка мусора для значений хранимых по указателям.

Синтаксис и встроенные средства языка Oberon-07 сравнительно проще других языков, благодаря этому он хорошо подходит для создания экспериментальных языков на его основе. Полное описание языка умещается на 17 страницах [2].

1.2 Процедурно-параметрическое программирование

Процедурно-параметрическое программирование [3] это парадигма программирования которая основывается на параметрических обобщениях и обобщающих параметрических процедурах. Параметрическое обобщение представляет собой тип данных инкапсулирующий в себе один из нескольких типов объявленных в её определении, при этом каждый из вариантов имеет свое собственное обозначение [4]. Например:

```
T = CASE OF x : Circle | y : Rectangle | z : Triangle END;
```

В данном случае объявляется обобщенный тип `T` который может быть, Circle, Rectangle или Triangle. Таким образом обобщенный тип похож на вариантный тип (например `std::variant` из C++) или на тип-сумму из функциональных языков, хотя не полностью им соответствует, из обобщенного типа нельзя получить значение обычным способом. Единственный способ получить значение хранимое в обобщенном типе это использование обобщающих параметрических процедур [4]. Пример объявления обобщающей параметрической процедуры:

```
PROCEDURE proc {x : T, y : Y} (z : INTEGER) := 0;
```

В данном примере объявляется процедура `proc` с списком обобщенных параметров заключенных в фигурные скобки и списком обычных параметров заключенных в круглые скобки, «`:= 0`» обозначает то что у данной процедуры нет стандартной реализации.

В списке обобщенных параметров обобщающей параметрической процедуры допустимо объявлять несколько параметров, таким образом можно использовать множественную диспетчеризацию. Множественная диспетчеризация это механизм в языках программирования отвечающий за выбор одной из множества функций на основе данных о динамических типах аргументов в точке вызова данной функции. Множественная диспетчеризация

является обобщением одиночной диспетчеризации (виртуальных функций) поддержка которой имеется во многих языках программирования.

Синтаксис используемых в данной работе процедурно-параметрических расширений отличается от синтаксиса представленного в [3], он представлен в Приложении Б.

1.3 Трансляторы

Транслятор — программа которая производит преобразование исходного кода представленного на одном языке программирования в исходный код на другом языке [5].

Процесс работы большинства используемых трансляторов включает в себя следующие этапы:

1) Лексический анализ — исходный код на заданном языке программирования преобразуется в последовательность лексем;

2) Синтаксический анализ — последовательность лексем полученная на этапе лексического анализа преобразуется в дерево разбора;

3) Генерация промежуточного представления — дерево разбора анализируется на корректность и преобразуется в промежуточное представление;

4) После генерации промежуточного представления может происходить один из следующих этапов в зависимости от типа транслятора:

- Оптимизация и генерация низкоуровневого кода (ассемблера);
- Генерация кода на другом языке высокого уровня;
- Интерпретация промежуточного представления без преобразования в код на другом языке.

В зависимости от результатов работы трансляторы разделяются на следующие группы:

1) Компиляторы
   - В исполняемый код;
   - Транспиляторы (в исходный код другого языка);

2) Интерпретаторы — вместо преобразования исходного кода из одной формы в другую интерпретатор производит его выполнение, хотя на промежуточных этапах его работы возможно использование байт-кода.

1.4 Выводы

Oberon — это статически типизированный императивный язык программирования, таким образом для генерации промежуточного представления из данного языка будет необходимо реализовать проверку совместимости типов для тех или иных операций [6]. Процедурно-параметрические расширения дополняют новым типом данных — параметрическим обобщением и расширяют понятие процедуры в языке добавляя обобщенные параметрические процедуры. Так как обобщенные параметрические процедуры являются расширением обычных процедур имеет смысл реализовать общую часть логики процедур в виде отдельного класса. Итогом работы базовой части транслятора является промежуточное представление, но так как данный этап является промежуточным в работе полнофункционального транслятора, то получаемое промежуточное представление должно иметь формат удобный для его дальнейшего использования.

2 Выбор методов реализации

Для построения базовой части транслятора языка программирования необходимо определиться с методами реализации трех основных компонентов программы:

1) Лексический анализатор;

2) Синтаксический анализатор;

3) Генератор промежуточного представления.

В данном списке опущен этап оптимизации, так как для экспериментального языка программирования скорость получаемой программы не является значимым фактором. Также при выборе методов реализации учитывались желательные требования к данным методам:

– Расширяемость без переписывания большей части кода;

– Переносимость между различными операционными системами.

Для всех представленных этапов, существуют специализированные инструменты позволяющие снизить затраты сил и времени на разработку транслятора, но при этом накладывающие определенные ограничения на структуру, быстродействие, переносимость и другие характеристики будущего языка.

2.1 Выбор метода реализации лексического и синтаксического анализа

Желательные требования к лексическому и синтаксическому анализатору:

– Наглядность исходного кода;

– Возможность разбора грамматик любой сложности.

2.1.1 Регулярные выражения

Регулярные выражения — специальный язык предназначенный для поиска и выделения частей текста соответствующих заданному шаблону. Библиотеки с поддержкой регулярных выражений присутствуют в большинстве современных языков программирования высокого уровня. Могут быть использованы для поиска и выделения из текста программы лексем языка вместо генератора лексических анализаторов. Но в тоже время регулярные выражения не обладают наглядностью из-за особенностей синтаксиса и могут

использоваться только при реализации лексического анализа.

2.1.2 Генераторы лексических и синтаксических анализаторов

Генераторы лексических анализаторов — специализированное программное обеспечение предназначенное для генерации лексических анализаторов. В качестве входных данных принимают описание лексем языка в определенном формате, результатом работы генератора, в большинстве случаев, является код на языке высокого уровня реализующий анализатор. Примеры:

– Lex [7] — стандартный генератор лексических анализаторов для операционных систем семейства Unix. Генерирует лексический анализатор на языке программирования;

– JLex [8] — аналог Lex на языке программирования Java. Генерирует лексический анализатор также на языке Java.

Генераторы синтаксических анализаторов — специализированное ПО для генерации синтаксического анализатора. На вход принимают правила составления дерева разбора из заданных лексем, результатом работы является код программы реализующей синтаксический анализатор. Примеры:

– ANTLR [9] — совмещает в себе генераторы лексических и синтаксических анализаторов. Результатом работы является код на языке высокого уровня по выбору;

– Yacc [10] — используется совместно с lex или flex, генерирует код на языке C.

Значительной проблемой генераторов лексических и синтаксических анализаторов является то, что генерируемый ими код очень сложно редактировать и менять, так как он был сгенерирован автоматически.

2.1.3 Комбинаторные синтаксические анализаторы

Комбинаторные синтаксические анализаторы представляют собой объединение лексического и синтаксического анализатора. Комбинаторные син-

таксические анализаторы строятся на основе функций высшего порядка позволяющих объединять несколько простых синтаксических анализаторов в более сложный (комбинаторы). Благодаря этому лексические анализаторы написанные с помощью комбинаторов обладают довольно высокой наглядностью, а высокая расширяемость позволяет реализовать лексический анализатор для языка любой сложности. Так же достоинством комбинаторных синтаксических анализаторов является простота реализации.

2.2 Выбор метода реализации генератора промежуточного представления

Не существует общепринятых генераторов промежуточного представления, так как набор конструкций каждого отдельно взятого языка значительно отличается от других языков. Частично их роль могут выполнять компиляторы компиляторов, но архитектура компилятора компиляторов может накладывать ограничения на структуру генерируемого языка. В некоторых случаях возможна генерация кода на языке высокого уровня, тогда генерация промежуточного представления проводится транслятором этого языка. Существенным минусом такого подхода является невозможность генерации осмысленных ошибок этапа генерации промежуточного представления, так как ошибки нижележащего транслятора могут быть только косвенно связаны с ошибками в коде программы на реализуемом языке.

2.3 Выбор языка реализации

Язык программирования для реализации транслятора желательно должен соответствовать следующим требованиям:

– Стабильность — должен существовать развитый транслятор языка который будет и дальше поддерживаться и развиваться;

– Переносимость между различными операционными системами;

– Компиляция в исполняемый файл;

– Ручное управление ресурсами.

2.3.1 Язык программирования C

Для языка программирования C существует большое количество стабильных компиляторов и стандартных библиотек, также его реализации существуют практически под все возможные аппаратные платформы и операционные системы. Но язык C поддерживает только процедурную парадигму программирования, что может значительно усложнить исходный код реализации транслятора, а так же повысить сложность его расширения.

2.3.2 Язык программирования C++

Язык программирования C++ обладает теми же преимуществами что и язык C, в виде очень высокой переносимости и генерации быстрого исполняемого кода. Кроме того он поддерживает объектно-ориентированную парадигму программирования, что увеличивает выразительность и гибкость получаемого исходного кода. Кроме того, язык C++ поддерживает обобщенное программирование через шаблоны функций и классов.

2.4 Выводы

Для лексического и синтаксического анализа было решено реализовать собственный лексический анализатор и синтаксический анализатор на основе комбинаторов, так как такой синтаксический анализатор полностью соответствует желательным требованиям, а также не ставит транслятор в зависимость от внешней библиотеки для синтаксического анализа.

Генератор промежуточного представления решено реализовать самостоятельно, так как другие варианты не подходят по условиям поставленной задачи.

Окончательным решение в выборе языка реализации транслятора было отдано в пользу языка C++ так как он обладая всеми преимуществами язы-

ка C, так же дает возможность писать более расширяемый и выразительный код, благодаря поддержке объектно-ориентированного и обобщенного программирования.

3 Описание разработанного приложения

В рамках выпускной квалификационной работы был разработан лексический анализатор, синтаксический анализатор и генератор промежуточного представления для языка Oberon-07 с процедурно-параметрическими расширениями. Приложение полностью написано на C++.

Рисунок 1 – Структура разработанного приложения

3.1 Лексический и синтаксический анализаторы

Лексический и синтаксический анализаторы построены из элементарных анализаторов, то есть анализаторов предназначенных для разбора одно-

го символа из входной программы по некоторому заданному предикату. Для разбора более сложных синтаксических структур элементарные анализаторы объединяются с помощью комбинаторов — функций высшего порядка принимающих на вход несколько анализаторов и возвращающих новый анализатор скомбинированный из входных входных аргументов тем или иным образом. В данной реализации комбинаторы являются конструкторами конкретных классов, а сами анализаторы объектами классов. Синтаксический анализатор в общем случае представляет собой направленный граф.

Базовым классом для всех анализаторов является шаблонный класс Parser<T>, где T — тип значения результата разбора. Он него наследуются конкретные анализаторы.

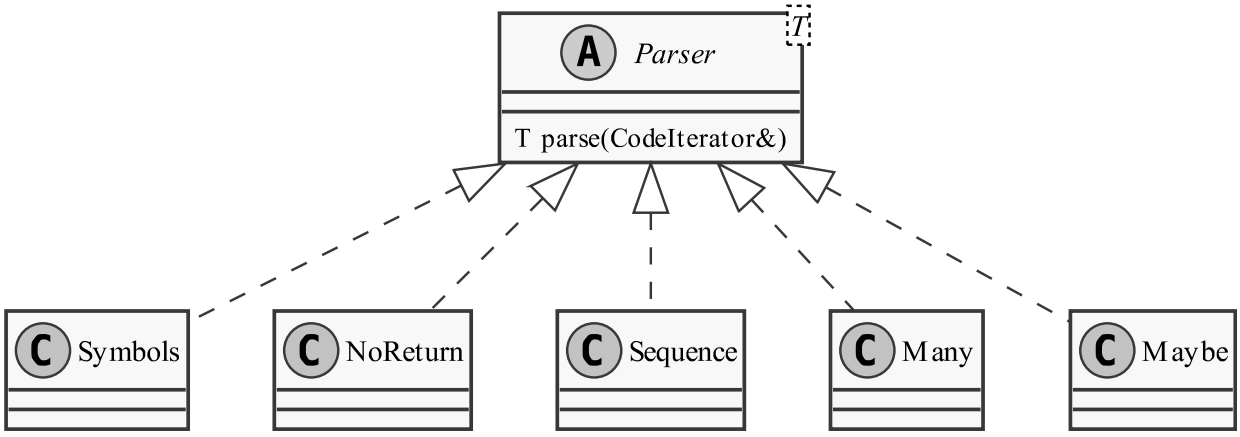

Рисунок 2 – Часть диаграммы классов элементарных анализаторов и комбинаторов

На рисунке 2 изображены основные классы анализаторов, в схему включены не все использующиеся в программе анализаторы. К каждому анализатору прилагается функция имеющая такое же имя как он сам и возвращающая умный указатель.

Листинг 1 – Реализация анализатора идентификатора

```
ParserPtr<char> letter = predicate("letter", [](auto c) {
  return (c >= 'A' && c <= 'Z') || (c >= 'a' && c <= 'z');
});
ParserPtr<char> letter_or_digit = predicate("latter or digit",
                                            is_letter_or_digit);
ParserPtr<Ident> identifier = construct<Ident>(chain(letter,
   many(letter_or_digit)));
```

Листинг 2 – Синтаксис идентификатора (ident) в нотации РБНФ

```
  letter = "A" | "B" | ... | "Z" | "a" | "b" | ... | "z".
  digit = "0" | "1" | "2" | "3" | "4"
        | "5" | "6" | "7" | "8" | "9".
  ident = letter {letter | digit}.
```

В листинге 1 представлен пример записи анализатора идентификатора, то есть анализатора последовательности цифр и букв начинающейся с буквы. В данном листинге в переменную letter записывается анализатор созданный на основе анонимной функции, которая принимает любую букву. В letter_or_digit записывается анализатор на основе предиката is_letter_or_digit. Конечный анализатор identificator составляется из letter и latter_or_digit с помощью комбинаторов chain, many и construct.

Далее представлено описание основных элементарных анализаторов и комбинаторов.

Элементарные анализаторы:

– Symbol — принимает один символ и разбирает только его, возвращает символ;

– Predicate — принимает предикат и разбирает один символ в соответствии с предоставленным предикатом;

– Symbols — принимает строку и разбирает только последовательность

символов в точности соответствующую заданной.

Комбинаторы:

– Many — принимает один анализатор и выполняет разбор текста с его помощью 0 или более раз, возвращает вектор результатов;

– Sequence — принимает любое количество анализаторов и выполняет их поочередно, возвращает кортеж результатов;

– Variant — принимает любое количество анализаторов запускает их поочередно и возвращает результат первого удачного разбора;

– Maybe — принимает анализатор и применяет его 0 или 1 раз, возвращает std::optional;

– Constructor<Type> — шаблон, принимает тип который необходимо сконструировать и анализатор, передает результат анализатор в конструктор заданного типа;

– CountFrom<From> — использует заданный анализатор как минимум From раз;

– Count<From,To> — использует заданный анализатор от From до To раз;

– Except — принимает анализатор и предикат, после разбора проверяет полученные данные с помощью предиката, если предикат возвращает false, то Except возвращает ошибку;

– NoReturn — заставляет анализатор вернуть критическую ошибку, даже если она не была критической до этого;

– Debug — комбинатор предназначенный для вывода отладочной информации, выводит результат работы переданного в него анализатора;

– ErrorDescription — принимает анализатор и описание ошибки, если анализатор возвращает ошибку, то добавляет её описание.

Результатом работы лексического и синтаксического анализаторов является дерево состоящее из заранее заданных типов узлов. Для совместимости

с генератором промежуточного представления каждый узел дерева должен наследоваться от класса Node или его наследников.

3.2 Генератор промежуточного представления

Входными данными для генератора промежуточного представления является дерево разбора полученное на этапе синтаксического анализа. Дерево должно состоять из узлов которые наследуются от абстрактного класса Node или одного из его наследников, каждый из которых представляет свой тип узлов дерева. Каждый тип узлов реализует свою логику работы через виртуальные методы, при этом каждый узел кроме своих данных хранит так же и структуру place в которой указано место в файле (строка и столбец) в котором находился данный узел до синтаксического разбора.

Далее представлено описание основных типов узлов дерева разбора: Expression, Value, Type, Statement и Section.

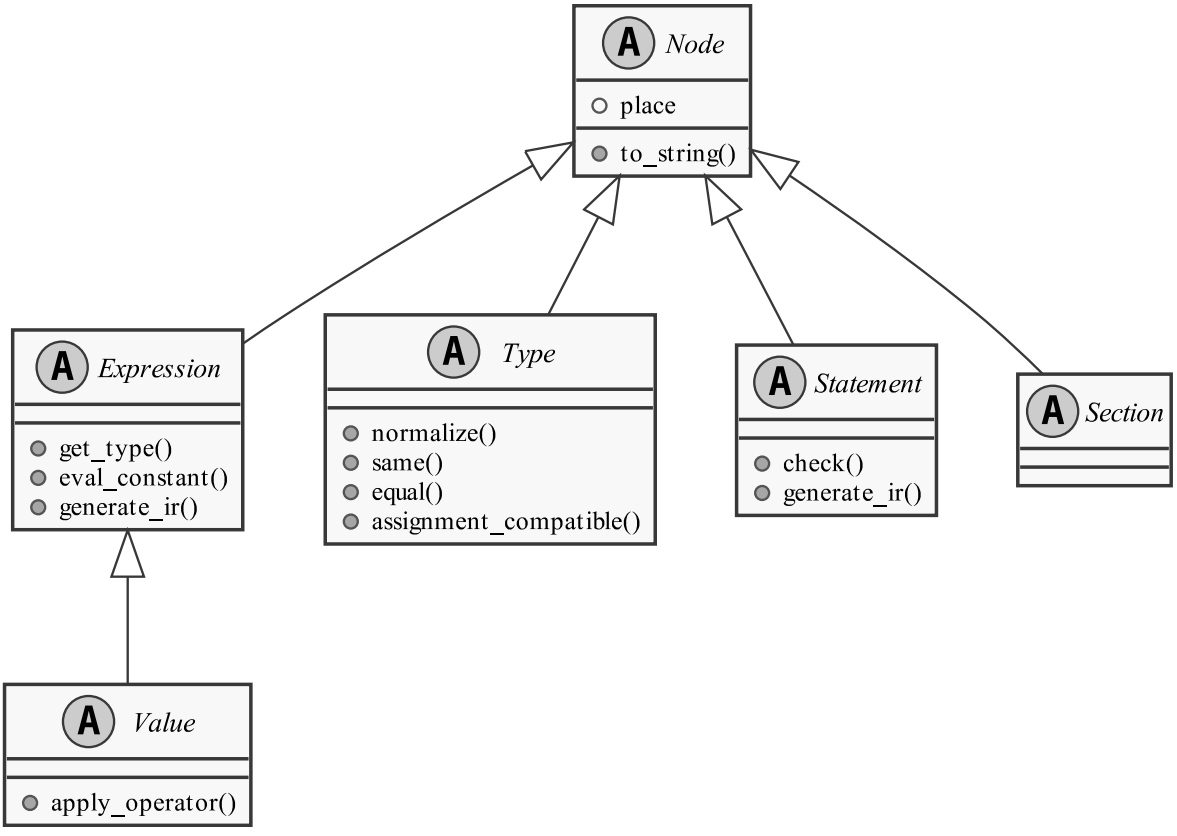

Рисунок 3 – Иерархия типов узлов дерева разбора

3.2.1 Expression

Дочерние классы Expression это выражения, то есть любые синтаксические структуры у которых есть возвращаемое значение, например вызов функции или обращение к элементу массива. Для выражения необходимо реализовать два абстрактных метода:

– get_type — используется для получения возвращаемого типа данного выражения. Возвращает вычисленный тип выражения и его модификатор;

– eval_constant — используется для вычисления выражения если оно является константой. Возвращает результат вычисления или ошибку;

– generate_ir — используется для генерации промежуточного представления выражения.

Примеры классов-потомков Expression:

– Term — представляет собой любые выражения в которых используются алгебраические операторы, например: $x + 2$;

– ProcCall — представляет вызов функций и обращение к полям структуры, например: $value.func(1, 2)$.

3.2.2 Value

Дочерние классы Value это простые значения, то есть строки, числа, множества и т.д. Они должны реализовывать как методы класса Expression так и один метод класса Value:

– apply_operator — используется для вычисления константных выражений в которых используется алгебраические операторы.

Примеры классов-потомков Value:

– IntegerValue — представляет литерал целочисленного значения, например: 42;

– StringValue — представляет литерал строки, например: ”Hello world”;

3.2.3 Type

Дочерние классы Type это простые и сложные типы данных, имена типов и тому подобные структуры. Для узлов данного типа необходимо реализовать четыре виртуальных функции:

– normalize — «нормализация» типа, то есть приведение типа к общему виду, в котором отсутствуют имена типов, не вычисленные константные выражения и т.д. Возвращает новый нормализованный тип;

– same — проверка типов на то что они являются одним и тем же типом. Как и все последующие методы сравнения возвращает булево значение;

– equal — сравнение типов на равенство — структурную эквивалентность;

– assignment_compatible — сравнение типов на совместимость по присваиванию, то есть таких типов что переменной можно присвоить любое значение которое имеет тип совместимый по присваиванию с типом переменной.

Примеры классов-потомков Type:

– BuiltInType — тип в который входят все встроенные типы языка, например: INTEGER;

– RecordType — тип для обозначения структур, например:
RECORD x : INTEGER, y : REAL END.

3.2.4 Statement

Дочерние классы Statement это синтаксические структуры у которых нет возвращаемого типа. То есть это присваивания, циклы, условные переходы и т.д. Для узлов данного типа необходимо реализовать один виртуальный метод:

– check — проверяет корректность декларации. Возвращает булево значение;

– generate_ir — производит генерацию промежуточного представления.

Примеры классов-потомков Statement:

– IfStatement — представляет оператор ветвления, например:

IF x > 0 THEN a ELSE b END;

– Assignment — присваивание, пример: x[1] := y.

3.2.5 Section

Дочерние классы Section это участки программы которые содержат список определений и исполняемого кода, то есть в данном случае это модули, процедуры и мультиметоды. Узлы данного типа не имеют дополнительных виртуальных методов, но в процессе генерации промежуточного представления каждый такой узел преобразуется в таблицу символов (класс-потомок SymbolTable).

3.2.6 Таблицы символов

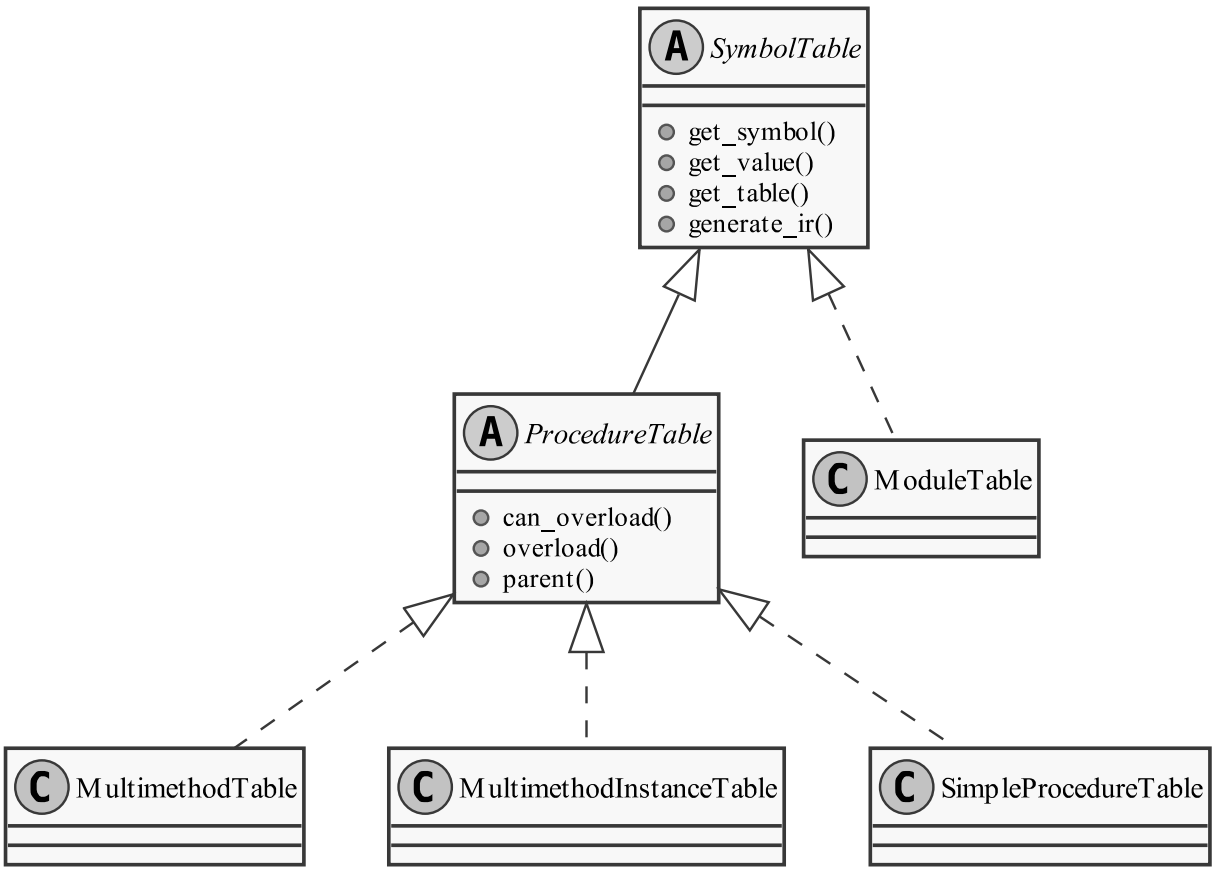

Рисунок 4 – Иерархия таблиц символов

В процессе генерации промежуточного представления каждый узел типа Section в дереве разбора преобразуется в конкретный класс который наследуется от абстрактного класса SymbolTable — таблицу символов. Каждая таблица символов хранит список символов объявленных в соответствующей секции, а также информацию о этих символах, такую как:

– Расположение объявления в исходном файле (строка и столбец);

– Тип символа;

– Значение символа для констант;

– Модификатор символа (переменная, константа или тип).

Всего существует четыре конкретных класса таблиц символов каждый их которых предназначен для своего типа секций и обладает некоторыми отличиями от других классов:

– ModuleTable — предназначен для модулей и кроме списка символов хранит список импортированных модулей и экспортированных символов;

– ProcedureTable — абстрактный класс, оставшиеся три конкретных класса таблиц символов наследуются от него, хранит тип аргументов процедуры и её возвращаемое значение;

– SimpleProcedureTable — предназначен для обычных процедур, наследуется от ProcedureTable, не может быть перегружен, не хранит дополнительных данных;

– MultimethodTable — предназначен для основного объявления мультиметода хранит список реализаций данного мультиметода, может быть перегружен;

– MultimethodInstanceTable — предназначен для конкретных реализаций мультиметода, может быть перегружен, не хранит дополнительных данных.

Таблицы символов в пределах одного модуля образуют древовидную структуру, корнем которой выступает сам модуль. В то же время таблицы

символов модулей образуют ориентированный ациклический граф.

3.3 Формат промежуточного представления

Промежуточное представление в разработанном приложении представляет собой простой трехадресный код [11]. Промежуточное представление состоит из следующих конструкций:

1) Присваивание выражения с бинарным оператором вида $x = y\ op\ z$, где op — бинарный оператор, x — переменная, y и z — переменные или числовые литералы;

2) Присваивание с унарным оператором вида $x = op\ y$;

3) Копирование из одной переменной в другую вида $x = y$;

4) Безусловный переход на метку L вида $goto\ L$;

5) Условный переход вида $if\ y\ goto\ L$, если значение переменной истинно, где L — название метки на которую необходимо перейти;

6) Условный переход вида $ifnot\ y\ goto\ L$, если значение переменной ложно;

7) Условный переход вида $ifrel\ y\ relop\ z\ goto\ L$, если результат выражения это истина, где relop — оператор сравнения;

8) Передача параметра в функцию вида $param\ y$;

9) Вызов функции вида $x = call\ func\ count$, где func — имя функции, count — количество аргументов передаваемых функции, вызов функции может иметь вид $call\ func\ count$, если функция не возвращает результат;

10) Возврат значения из функции вида $return\ y$ или просто $return$, если у функции нет возвращаемого значения;

11) Добавление новой таблицы символов в стек таблиц символов вида $push\ x\ table$, где x — имя переменной с адреса которой начнется новая таблица, table — имя таблицы.

Кроме того любая переменная в промежуточном представлении, может иметь

модификатор разыменования ($*x$) или модификатор взятия адреса переменной ($\&x$).

Таблицы символов в промежуточном представлении представляют собой таблицу из трех столбцов в которых содержится:

1) Имя символа;
2) Является тип указателем или нет (булево значение);
3) Имя типа символа.

И имеют вид

```
symbols name
x_1 bool_1 typename_1
...
x_n bool_n typename_n
```

3.4 Интерфейс приложения

Разработанное приложение имеет текстовый интерфейс, запуск приложения происходит после ввода команды вида:

```
$ oberon Опции Файл
```

Где:

– Опции — одна из возможных опций запуска
  – «-h» — вывод справки;
  – «-p» — вывод дерева разбора в текстовом виде;
  – «-t» — вывод таблиц символов в текстовом виде;
  – «-i» — вывод промежуточного представления;
  – «-d» — указание директорий для поиска модулей.

  По умолчанию используется флаг «-p»;
– Файл — входной файл который необходимо обработать.

3.5 Выводы

В ходе работы были реализованы: лексический анализатор, синтаксический анализатор и генератор промежуточного представления. Все компоненты представляют собой наборы базовых функций и классов, на основе которых была реализована базовая часть транслятора экспериментального языка программирования на основе Oberon-07.

ЗАКЛЮЧЕНИЕ

В результате выполнения выпускной квалификационной работы была разработана базовая часть транслятора экспериментального языка программирования на основе языка Oberon-07 с процедурно-параметрическими расширениями.

В процессе выполнения работы были реализованы наборы классов для реализации лексического анализатора, синтаксического анализатора и генератора промежуточного представления. На их основе была реализована базовая часть транслятора языка программирования Oberon-07 с расширениями. В пояснительной записке приведен анализ методов реализации компонентов транслятора и описание основных частей разработанного приложения.

Разработанное приложение соответствует цели работы. Перспективами данной работы является использование промежуточного представления для генерации машинного кода или интерпретации.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Wirth N. The Programming Language Oberon-07 : тех. отч. / ETH Zurich. — 2016. — С. 16—17.
2. Wirth N. The Programming Language Oberon-07 : тех. отч. / ETH Zurich. — 2016.
3. Легалов А. И. Процедурно-параметрическое программирование. — 2001. — URL: www.softcraft.ru/paradigm/ppp.
4. Легалов А. И., Швец Д. А. Язык программирования O2M. — 2013–2014. — URL: www.softcraft.ru/ppp/o2m/o2mref.
5. Компиляторы: принципы, технологии и инструментарий : Введение в компиляцию : пер. с англ. / А. В. Ахо [и др.] //. — ООО «И.Д. Вильямс», 2018. — С. 29. — ISBN 978-5-8459-1932-8.
6. MIASAP : Type compatibility in Oberon. — 2018. — URL: http://miasap.se/obnc/type-compatibility.html.
7. POSIX Programmer's Manual : LEX. — 2017. — URL: https://man7.org/linux/man-pages/man1/lex.1p.html.
8. JLex: A Lexical Analyzer Generator for Java(TM). — URL: https://www.cs.princeton.edu/~appel/modern/java/JLex/.
9. ANTLR (ANother Tool for Language Recognition). — URL: https://www.antlr.org/.
10. POSIX Programmer's Manual : YACC. — 2017. — URL: https://man7.org/linux/man-pages/man1/yacc.1p.html.
11. Компиляторы: принципы, технологии и инструментарий : Введение в компиляцию : пер. с англ. / А. В. Ахо [и др.] //. — ООО «И.Д. Вильямс», 2018. — С. 450. — ISBN 978-5-8459-1932-8.

ПРИЛОЖЕНИЕ А

Синтаксис языка Oberon-07 в РБНФ

```
letter = "A" | "B" | ... | "Z" | "a" | "b" | ... | "z".
digit = "0" | "1" | "2" | "3" | "4"
      | "5" | "6" | "7" | "8" | "9".
ident = letter {letter | digit}.
number = integer | real.
integer = digit {digit} | digit {hexDigit} "H".
real = digit {digit} "." {digit} [ScaleFactor].
ScaleFactor = "E" ["+" | "-"] digit {digit}.
hexDigit = digit | "A" | "B" | "C" | "D" | "E" | "F".
string = """ {character} """ | digit {hexDigit} "X".
qualident = [ident "."] ident.
identdef = ident ["*"].

ConstDeclaration = identdef "=" ConstExpression.
ConstExpression = expression.

TypeDeclaration = identdef "=" StrucType.
type = qualident | StrucType.
ArrayType = ARRAY length {"," length} OF type.
length = ConstExpression.
RecordType = RECORD ["(" BaseType ")"] [FieldListSequence] END.
BaseType = qualident.
FieldListSequence = FieldList {";" FieldList}.
FieldList = IdentList ":" type.
IdentList = identdef {"," identdef}.
PointerType = POINTER TO type.
ProcedureType = PROCEDURE [FormalParameters].
StrucType = ArrayType | RecordType | PointerType | ProcedureType.

VariableDeclaration = IdentList ":" type.
```

```
expression = SimpleExpression [relation SimpleExpression].
relation = "=" | "#" | "<" | "<=" | ">" | ">=" | IN | IS.
SimpleExpression = ["+" | "-"] term {AddOperator term}.
AddOperator = "+" | "-" | OR.
term = factor {MulOperator factor}.
MulOperator = "*" | "/" | DIV | MOD | "&".
factor = number | string | NIL | TRUE | FALSE |
    set | designator [ActualParameters] | "(" expression ")" |
   "~" factor.
designator = qualident {selector}.
selector = "." ident | "[" ExpList "]" | "^" | "(" qualident ")".
set = "{" [element {"," element}] "}".
element = expression [".." expression].
ExpList = expression {"," expression}.
ActualParameters = "(" [ExpList] ")".

statement = [assignment | ProcedureCall | IfStatement
          | CaseStatement | WhileStatement
          | RepeatStatement | ForStatement].
assignment = designator ":=" expression.
ProcedureCall = designator [ActualParameters].
StatementSequence = statement {";" statement}.
IfStatement = IF expression THEN StatementSequence
    {ELSIF expression THEN StatementSequence}
    [ELSE StatementSequence] END.
CaseStatement = CASE expression OF case {"|" case} END.
case = [CaseLabelList ":" StatementSequence].
CaseLabelList = LabelRange {"," LabelRange}.
LabelRange = label [".." label].
label = integer | string | qualident.
WhileStatement = WHILE expression DO StatementSequence
    {ELSIF expression DO StatementSequence} END.
```

```
RepeatStatement = REPEAT StatementSequence UNTIL expression.
ForStatement = FOR ident ":=" expression TO expression [BY
   ConstExpression]
    DO StatementSequence END.

ProcedureDeclaration = ProcedureHeading ";" ProcedureBody ident.
ProcedureHeading = PROCEDURE identdef [FormalParameters].
ProcedureBody = DeclarationSequence [BEGIN StatementSequence]
    [RETURN expression] END.
DeclarationSequence = [CONST {ConstDeclaration ";"}]
    [TYPE {TypeDeclaration ";"}] [VAR {VariableDeclaration ";"}]
    {ProcedureDeclaration ";"}.
FormalParameters = "(" [FPSection {";" FPSection}] ")" [":"
   qualident].
FPSection = [VAR] ident {"," ident} ":" FormalType.
FormalType = {ARRAY OF} qualident.

module = MODULE ident ";" [ImportList] DeclarationSequence
    [BEGIN StatementSequence] END ident ".".
ImportList = IMPORT import {"," import} ";".
import = ident [":=" ident].
```

ПРИЛОЖЕНИЕ Б

Синтаксис процедурно-параметрических расширений в РБНФ

```
CommonFeature = ident | integer | string.
CommonFPSection = [VAR] ident {"," ident} ":"
     Qualident "<" CommonFeature ">".
CommonPars = "{" [CommonFPSection {";" CommonFPSection}] "}".
CommonProc = PROCEDURE IdentDef CommonPars [FormalPars]
     (":=" "0" | ";" DeclSeq [BEGIN StatementSeq] END ident).
ProcDecl = PROCEDURE IdentDef [CommonPars] [FormalPars] ";"
         DeclSeq [BEGIN StatementSeq] END ident.
CommonFeatureType = INTEGER | BYTE | SET | CHAR | TYPE | LOCAL.
CommonTypeDecl = CASE [CommonFeatureType]
     OF CommonFeature ":"
     Qualident {"|" CommonFeature ":" Qualident}
     [ELSE Type] END.
Type = Qualident
     | CommonTypeDecl
     | ARRAY ConstExpr {"," ConstExpr} OF Type
     | RECORD ["("Qualident")"] FieldList {";" FieldList} END
     | POINTER TO Type
     | PROCEDURE [FormalPars].
ScalarVarDecl = IdentList ":" Qualident "<" CommonFeature ">".
VarDecl = IdentList ":" Type | ScalarVarDecl.

ProcCommonPars = "{" [Qualident {"," Qualident}] "}".
Factor = [ProcCommonPars "."] Designator ["(" [ExprList] ")"]
       | number | character
       | string | NIL | Set
       | "(" Expr ")" | "~" Factor.
```

Федеральное государственное автономное
образовательное учреждение
высшего образования
«СИБИРСКИЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»

Институт космических и информационных технологий
институт

Вычислительная техника
кафедра

УТВЕРЖДАЮ
Заведующий кафедрой
_____ О.В. Непомнящий
подпись инициалы, фамилия
«25» 06 2021 г.

БАКАЛАВРСКАЯ РАБОТА

09.03.01 Информатика и вычислительная техника

Базовая часть транслятора экспериментального языка программирования на основе Oberon-07

Руководитель	18.06.21	доцент, канд.техн.наук	Д.А. Швец
	подпись, дата	должность, ученая степень	инициалы, фамилия
Выпускник	18.06.21		П.С. Коваленко
	подпись, дата		инициалы, фамилия
Нормконтролер	18.06.21	доцент, канд.техн.наук	Д.А. Швец
	подпись, дата	должность, ученая степень	инициалы, фамилия

Красноярск 2021