ПРАВИТЕЛЬСТВО РОССИЙСКОЙ ФЕДЕРАЦИИ
ФГАОУ ВО НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ
«ВЫСШАЯ ШКОЛА ЭКОНОМИКИ»

Факультет компьютерных наук
Образовательная программа «Прикладная математика и информатика»

Отчет о программном проекте на тему:

Разработка компилятора языка программирования Oberon-7

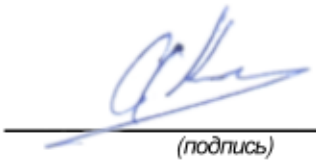
Выполнил:
студент группы БПМИ201
Черников Кирилл
Александрович

(подпись)

16.05.2023
(дата)

Принял руководитель проекта:

Легалов Александр Иванович
Штатный преподаватель, профессор
Департамент анализа данных и искусственного интеллекта



(подпись)

16.05.2023
(дата)

Москва 2023

Содержание

Аннотация

Oberon-7 представляет из себя язык программирования высокого уровня, основой построения программ в котором служат модули, которые поддерживают раздельную компиляцию. Целью данного проекта является разработка компилятора на C++, порождающего на выходе промежуточное представление и генерацию кода в язык C. Для этого написаны построение объектной модели и генератор кода. Результатом данной работы является компилятор из языка Oberon-7 в язык C, генерирующий не только необходимые файлы на языке C, но и Cmake файлы для создания проектов

1 Основные термины и определения

- **Дерево разбора грамматики G** – дерево, которое соответствует некоторой цепочке вывода и удовлетворяет следующим условиям: каждая вершина дерева обозначается символом грамматики, корнем дерева является вершина, обозначенная начальным символом грамматики, листьями дерева являются вершины, обозначенные терминальными символами грамматики или символом пустой строки, если некоторый узел дерева обозначен нетерминальным символом A, а связанные с ним узлы – символами b_1, ..., b_n, то в грамматике существует правило $A \rightarrow b_1$, ..., bn.

- **Лексема** – единые синтаксические объекты, в которые группируются входные символы. Например идентификатор, целое число, строка – это лексемы.

- **Детерминированный конечный автома**т – это конечный автомат, принимающий или отклоняющий заданную строку символов путём прохождения через последовательность состояний, определённых строкой.

- **Автомат с магазинной памятью** – это недетерминированный конечный автомат, имеющий дополнительную потенциально неограниченную ленту памяти (магазин). Доступ к информации в магазине возможен только с одного его конца в соответствии с принципом "последним пришел – первым ушел".

- **Диаграмма Вирта** – графический метаязык, удобный для описания синтаксиса языков.

- **Объектная модель** – представление исходной программы в виде классов и связей между ними. Классы олицетворяют собой смысловые объекты языка. Примером может служить DeclarationSequence – класс, содержащий в себе сведения о декларациях.

2 Введение

2.1 Описание предметной области

Oberon-7 представляет из себя язык программирования высокого уровня. Он является последней версией из семейства Oberon-языков, в рамках которого Никлаус Вирт попытался оставить только самые полезные конструкции из используемых в языках высокого уровня. Последнее описание его датируется 2016 годом [1]. Язык написан в соответствии с принципом "Делай просто, насколько возможно, но не проще этого" (А. Эйнштейн). Основой построения программ на данном языке служат модули, которые поддерживают раздельную компиляцию. В

языке отсутствует возможность освобождения динамически выделенной памяти – этим занимается сборщик мусора.

Целью данного проекта является разработка компилятора на C++, порождающего на выходе промежуточное представление и генерацию кода в язык C. Существует несколько компиляторов языка Oberon-7. Однако требования, предъявляемые к дальнейшему развитию проекта, ведут к необходимости собственной разработки компилятора.

В частности, в дальнейшем предполагаются расширения языка, связанные с включением в него поддержки процедурно-параметрического полиморфизма. Помимо этого планируются исследования возможностей построения семантической модели языка Оберон-7 другого языка программирования с Си-подобным синтаксисом. Для решения этих задач необходим свой компилятор, в котором реализуется промежуточное представление, отличающееся от тех, что используются в уже разработанных компиляторах.

В данной работе предстоит реализовать синтаксический анализ, семантический анализ, генерацию кода, а также рассмотреть возможности дальнейшего развития проекта на сформированных решениях.

2.2 Постановка задачи

В наше время разработано множество различных языков программирования и их компиляторов. Языки развиваются в множестве направлений. Помимо разносторонних задач, на которые нацелены языки, отличаются также и подходы авторов к их созданию. Кто-то пытается наиболее упростить их, чтобы облегчить процесс написания кода и уменьшить количество мест, где программист может допустить ошибку. В таких языках многое возлагается на авторов языка и компилятор (например, контроль за выделенной памятью). Часть авторов, наоборот, делают язык достаточно сложным и гибким, что позволяет лучше контролировать процесс в каждом конкретном случае, что обычно ускоряет программу.

Никлаус Вирт при разработке языков семейства Oberon придерживался первого варианта. Благодаря этому язык получился с относительно простым синтаксисом, который однако позволяет реализовывать такие парадигмы, как объектно-ориентированное программирование, программирование абстрактных типов, модульное программирование и другие.

Последняя версия языка датируется 2016 годом. Поэтому существует достаточно мало компиляторов для него. А существующие решения имеют свои недостатки, которые не позволяют им стать универсальными и удобными для всех. В связи с этим была поставлена задача написать компилятор для самого нового языка семейства Oberon (Oberon-7) на языке C++, порождающего на выходе промежуточное представление и генерацию кода в язык C.

Задача разбивается на разработку следующих компонентов: синтаксический анализатор, семантический анализатор и генератор кода. В дальнейшем можно добавить сборщик мусора. Парсер (распознаватель) нет необходимости разрабатывать, так как он уже написан [4]. Для проверки корректности работы компилятора его необходимо покрыть тестами. Необходимо выбрать структуру семантической модели между построением дерева разбора и построением объектной модели, исходя из особенностей языка. Программа написана на языке C++ и компилируется промежуточное представление и в язык C, так что она должна работать везде, где работает компилятор для языка C++.

2.3 Актуальность и значимость

Язык Oberon имеет как свои преимущества, так и недостатки. Его нацеленность на компонентно-ориентированную разработку решает некоторые проблемы, в частности, проблему хрупкости базового класса. Простота его синтаксиса позволяет достаточно быстро научиться работать с ним, что полезно, например, при обучении программированию.

Данная работа может применятся везде, где будет применятся язык Oberon-7. Кроме генерации в язык C она также создает промежуточное представление, что позволяет в будущем делать кодогенерацию и в другие языки без привязывания к парсеру.

Важно, что код компилятора будет написан на языке C++. Это нужно для проведения последующих различных экспериментов. Например, есть желание в эту же семантическую модель сделать парсер с другого вида синтаксиса в Си-подобном стиле, то есть, подменить фронт-енд и сохранить семантику Oberon.

3 Существующие работы и решения

На данный момент существует несколько компиляторов для языка Oberon-7.

Компилятор Astrobe от CFB Software [5] предназначен для узкоспециализированных целей, а именно, для микроконтроллеров ARM. Этот компилятор удобен тем, что имеет много готовых модулей и бесплатную версию. Он включает в себя полнофункциональный редактор с поддержкой синтаксиса, программный загрузчик, терминал, позволяющий взаимодействовать с приложением, и многое другое. По словам компании, за счет узкой специализации компилятора его удалось сделать компактным, легко поддерживаемым и надежным. Astrobe умеет выявлять ошибки еще до запуска программы. Из минусов – он работает только на операционной системе Windows, не имеет открытых исходников и подходит только для разработки микроконтроллеров.

Компилятор [2] также предназначен для микроконтроллеров ARM. Преимуществами данного компилятора являются открытые исходные коды и набор полезных модулей. Он способен работать не только на Windows, но и на Linux, правда только в установленной среде BlackBox. Из недостатков – компилятор написан на языке Pascal, что не позволяет использовать его для нужных нам целей, а также заточен под микроконтроллеры архитектуры ARMv6-M.

Компилятор Vostok [3] – большой проект, умеющий генерировать код, написанный на Oberon-е в множество промышленных языков программирования, таких как Java, Javascript, C, C++. Он также умеет генерировать машинный код. Компилятор имеет открытые исходные коды, что является значимым преимуществом. Код, генерируемый в C и C++, совместим с gcc и clang. Компилятор осуществляет контроль использования неинициализированных переменных, проверяет переполнения, отслеживает перекрытие имен, а также неиспользуемые элементы в модуле. Работает на Windows, Linux, macOS и Android. Недостатками можно считать то, что в проекте совсем недавно исправлялись разные ошибки, и возможно, хотя и маловероятно, какие-то ошибки еще остались. Также компилятор написан на языке Oberon, что не позволило его использовать в качестве заменителя данного проекта.

Компилятор [6], работающий на Windows, Linux и KolibriOS похож на Vostok, но имеет меньший функционал. Он тоже написан на Oberon-е и его исходные коды также выложены в общий доступ. Его преимуществом является работа с KolibriOS и библиотеки, поставляемые с ним. Компилятор все еще дорабатывается.

Помимо этих есть еще несколько компиляторов, о которых написано слишком мало, чтобы подробно анализировать их. Некоторые из них не имеют открытых исходных кодов и поставляются как папка и исполняемыми файлами [7]. Некоторые сейчас находятся в состоянии реконструкции, значимо меняя структуру компилятора [8].

Ни один из найденных мной существующих компиляторов для Oberon-7 не подходит полностью для заданных целей. Есть несколько хороших и популярных компиляторов, но они написаны на Oberon-е или Pascal-е, а значит не получится даже взять их за основу для проекта.

4 Предлагаемые подходы и методы

Все части компилятора будут написаны на языке C++. Обычно процесс разработки компилятора состоит из следующих частей: лексический анализ, синтаксический анализ, семантический анализ и генерация кода.

На этапе лексического анализа последовательность символов файла программы преобразуется в последовательность лексем. У лексем есть класс и значение. Примерами лексем могут служить идентификатор, целое число, строка и так далее. Лексический анализ бывает прямым и непрямым. Непрямым называется лексический анализ с возвратами. То есть, версии о лексемах проверяются последовательно и в случае не подтверждения происходит откат назад по цепочке символов. Прямой лексический анализ строится на основе одного детерминированного автомата, объединяющего автоматы, распознающие отдельные лексемы. Преимуществами прямого лексического анализа являются высокая производительность и меньшее общее число состояний. Недостатками являются его фактическая неприменимость для некоторых языков и большее время на разработку. В моем случае уже написан общий парсинг с использованием нисходящего разбора и рекурсивного спуска. Это решение за один проход при помощи рекурсивного спуска и откатов назад формирует не только последовательность лексем, но и выполняет часть функций синтаксического анализа, что, на мой взгляд, выглядит более эффективным.

Синтаксический анализ преобразует последовательность лексем в дерево разбора или в объектную модель. Синтаксический разбор является первым этапом синтаксического анализа. Он убеждается, что входная цепочка лексем является программой, а отдельные подцепочки составляют правильные программные объекты. Синтаксический разбор классифицируют по методу разбора (нисходящий, восходящий, комбинированный), по последовательности разбора (слева направо, справа налево, произвольный), по просмотру вперед (на один символ, на два символа и так далее) и по использованию возвратов (есть или нет). Для синтаксического разбора используются такие инструменты как автоматы с магазинной памятью, динамически порождаемые автоматы и диаграммы Вирта, задающие динамически порождаемые конечные автоматы. Я собираюсь использовать последний вариант, так как он дает некоторые преимущества. Диаграммы Вирта позволяют преобразовать исходные синтаксические правила к более простому виду, избавиться от левых или правых рекурсий, освободиться от пустых правил и так далее. Как написано выше, это можно сделать за один проход вместе с формированим лексем.

Семантический анализ формирует семантическую модель из синтаксического представления программы. Он определяет смысловую нагрузку объектов программы. Семантический анализ можно разделить на фазы построения и использования таблицы имен, построения операторной модели, контекстный анализ элементов. Семантическая модель может существовать в отрыве от синтаксиса языка. Среди вариантов построения семантической модели можно выделить построение дерева разбора и построение объектной модели. Преимуществом дерева разбора является простота, а из недостатков можно отметить жесткую привязку к конкретному синтаксису. Кроме того, не всегда возможно построить иерархическое дерево разбора. Объектная модель, хоть и более сложна в разработке, является универсальным отображением семантики без привязки к синтаксису. Также можно отметить, что промежуточная высокоуровневая семантическая модель используется в системе clang [9] при реализации компилятора для семейства языков. Это позволяет формировать не только различные кодогенераторы, но и проводить различные анализы исходного кода. В проекте в дальнейшем также планируются различные манипуляции с промежуточным представлением, поэтому и выбран подобный подход.

Генератор кода преобразует промежуточное представление в код на целевом языке. В моем случае это C, но впоследствии возможно добавить и другие, так как объектная модель позволяет это.

5 Структура и архитектура реализованного компилятора

5.1 Общая структура

Задача данного компилятора состоит в том, чтобы из исходного текста программы на Oberon-7 получить объектную модель и текст программы на языке C. Для этого исходный текст программы преобразуется в последовательность лексем. Последовательность лексем преобразуется в объектную модель, и по объектной модели генерируется код на языке C.

В языке Oberon-7 единицей компиляции является модуль. Первым шагом компиляции является парсер. Он обходит текст исходной программы и, в соответствии с описанием языка [1], проверяет лексемы и заполняет объектную модель. Парсер реализован посредством рекурсивной проверки лексем и объектов. При неудаче происходит откат назад. Проверка каждой лексемы и класса объектной модели реализована в виде автомата. В случае несоответствия программы синтаксису, парсер сообщает об ошибке, описывает, в какой строчке и в каком столбце программы имеется ошибка, и завершает работу.

Помимо синтаксических проверок, парсер также находит семантические ошибки, например, если программа работает с несуществующими переменными или вызывает процедуру с большим числом аргументов, чем в описании. Таким образом, на парсер возложена задача поиска любых ошибок, которые в будущем бы приводили к ошибкам компиляции полученного файла на языке C. Это возможно благодаря тому, что объектная модель строится параллельно с обходом парсера.

Таким образом, за один проход с откатами по исходному тексту программы, парсер либо находит ошибку, либо строит корректную объектную модель.

Каждый модуль может быть импортирован, поэтому, по построенной объектной модели код генерируется в два файла: “module_name.h” и “module_name.c”.

5.2 Архитектура объектной модели

5.2.1 Общая архитектура

Схема общей архитектуры объектной модели показана на рисунке 1. Объектная модель задаваемого модуля содержится в классе Module.

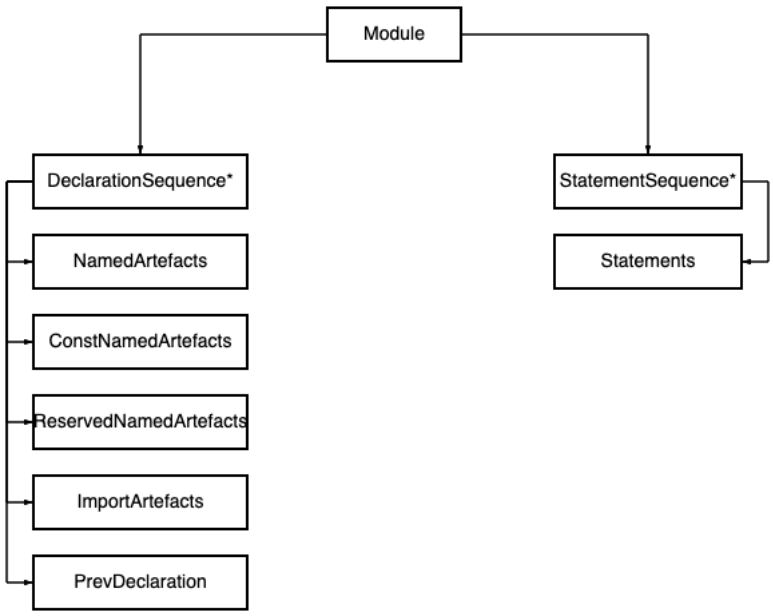


Рисунок 1. Схема общей архитектуры объектной модели

Модуль содержит указатели на DeclarationSequence и StatementSequence. В Oberon-7 можно декларировать типы, процедуры, константы и переменные. Информация о типах,

процедурах и переменных хранится в хэш таблице NamedArtefacts, в которой ключом является имя. Это возможно благодаря наследованию от базового типа Context. Некоторые имена переменных зарезервированы и запрещены для декларации пользователем в программе. К ним относятся базовые типы, имена предопределённых процедур и подобные. Указатель на список таких имён хранится в ReservedNamedArtefacts. Не только модуль, но процедуры и структуры могут иметь свои декларации. При этом, переменные, объявленные в более верхней декларации, доступны в более нижней, хотя и могут быть заменены локальными переменными. Поэтому из DeclarationSequence образуется дерево, каждая вершина которого хранит указатель на предка в переменной PrevDeclaration.

5.2.2 Организация константных переменных

Хранение и реализация константных переменных выделена отдельно в ConstNamedArtefacts. Это связано с тем, что константы должны быть вычислены и доступны сразу после их объявления. Константы могут задаваться через друг друга и через предопределённые константные функции. Запись вида $CONST\ height = 400;\ funcRes = FLT(ABS(-height)) - 1.0$ допускается и должна быть вычислена сразу.

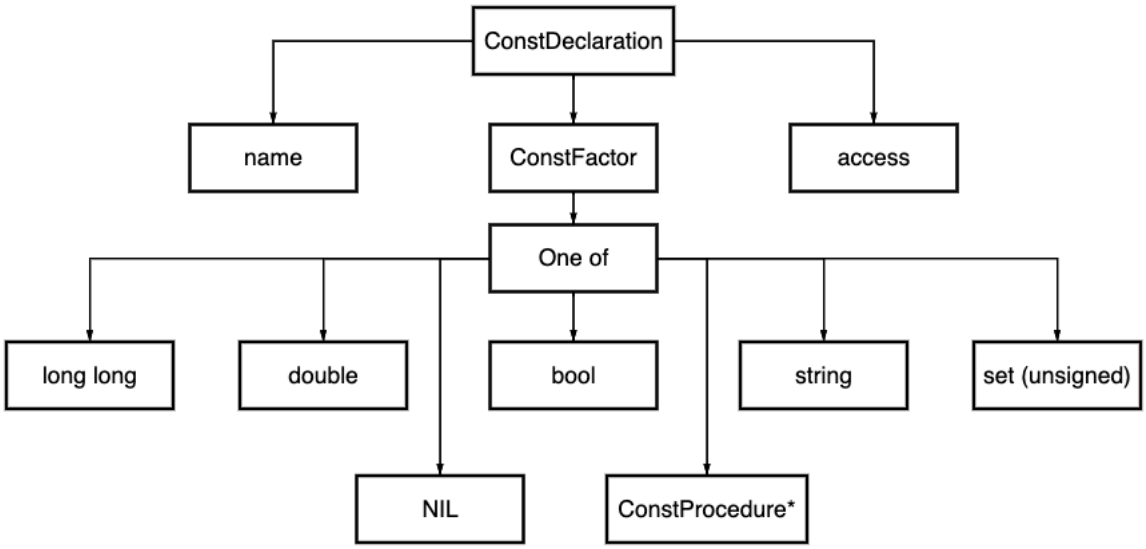


Рисунок 2. Схема организации константных переменных

Схема организации константных переменных представлена на рисунке 2. Тип константной переменной не задаётся при объявлении, задача вычисления его лежит на компиляторе. Константная переменная может быть одним из базовых типов, либо хранить константную процедуру. Это реализуется с помощью класса ConstFactor, который состоит из std::variant и типа хранимой переменной. Тип нужен для того, чтобы константные переменные могли участвовать в выражениях наравне с не константными.

По сути написать правильную обработку и поддержку константных переменных равносильно написанию калькулятора, поддерживающего переменные и разные типы. К константам могут применяться унарные и бинарные операторы, оператор “+” для целых чисел и для сетов – две абсолютно разные операции, компилятор отслеживает, что операторы применяются правильно и сообщает, если это не так (например, если исходная программа прибавляет целое число к сету). Результаты константных выражений тоже являются константными и могут иметь тип, который не встречается ни у одной переменной в выражении. Например, результатом применения оператора “==” будет Boolean.

Правильный порядок применения операторов обеспечивается структурой парсинга, а правильный результат вычислений – функциями, применяющими операторы, проверяющими все

случаи и выдающие сообщение в случае ошибки. ConstProcedure – это базовый класс константной процедуры с виртуальными методами вызова.

5.2.3 Организация не константных переменных

Схема организации не константных переменных представлена на рисунке 3. Не константные переменные хранятся в одной хэш таблице. Это возможно благодаря наследованию от общего типа Context. Oberon-7 позволяет создавать типы из массивов, записей (структур), указателей на структуры, процедур.

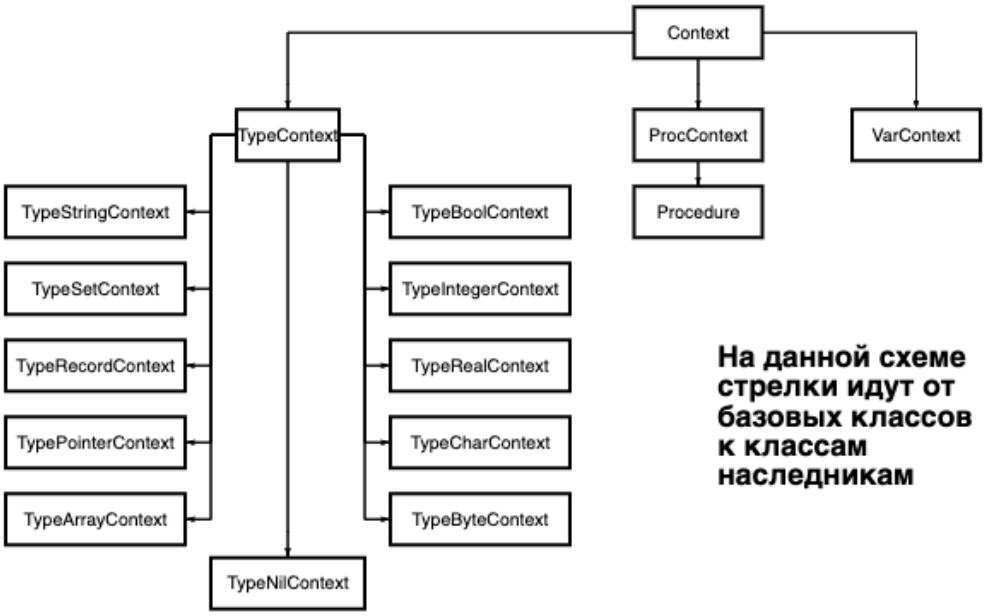


Рисунок 3. Схема организации не константных переменных

В языке Oberon-7 разрешено расширение записей (наследование). Запись хранит в себе DeclarationSequence, которая содержит её поля. Из наследника доступны поля базовой записи, поэтому декларации записей также образуют дерево. Oberon-7 позволяет указателю на базовый тип ссылаться на тип наследника. Отсюда берётся нетривиальный алгоритм проверки типов в выражении. Например, в присваивании $a := b$ проверяется, что тип a является предком типа b в дереве наследования. Для этого TypeRecordContext хранит указатель на тип предка.

Кроме этого, Oberon-7 позволяет создавать указатели на записи, которые будут объявлены позже, при условии, что объявление будет в той же DeclarationSequence. Это сделано для того, чтобы можно было написать, например, вершину односвязного списка. Чтобы это поддержать, запись должна быть добавлена в DeclarationSequence при первом появлении имени её типа, а после подменена на объявленную. Причём, нельзя просто добавить запись как новую, когда она будет задекларирована, потому что тогда указатель, ссылающийся на неё и объявленный ранее, будет ссылаться на пустую временную запись и программа будет работать некорректно. Кроме того, надо отличать этот случай от попытки редекларации, поскольку, надо уметь отсекать случаи декларации двух разных типов с одинаковым именем. Эта задача возложена на DeclarationSequence, которая, дополнительно к этому, ведёт подсчёт предекларацій записей и убеждается, что в итоге, все они будут объявлены.

Oberon-7 позволяет создавать переменные, в которые в дальнейшем будут положены процедуры. Тип такой переменной определяется типом возвращаемого значения и типами аргументов функции. Такой тип хранится в ProcContext, то есть процедуры без тела. Parser также проверяет переменные таких типов в выражениях на сравнимость.

Кроме этого Oberon-7 позволяет определять безымянные типы внутри декларации других типов или даже использовать указатель на тип A внутри определения типа A. В качестве примера можно рассмотреть декларацию типа Node (рисунок 4).

```
Node = POINTER TO RECORD
  rec    : POINTER TO AnyRec;
  next   : Node
END;
```

Рисунок 4. Декларация типа Node

Здесь тип Node используется в процессе своего определения, что затрудняет обработку DeclarationSequence. Это решается изначальным созданием указателя на будущую запись и дальнейшего использования его везде.

5.2.4 Организация дерева деклараций

Возможность мгновенного получения значений, определенных ранее константных переменных и типов не константных переменных, реализовано с помощью дерева из DeclarationSequence. Парсер строит это дерево в процессе обхода и ищет переменные по имени на пути от текущей вершины до корня.

Корень вершины основного дерева деклараций изначально не пуст. Он содержит уже задекларированные стандартные типы, константные функции и заголовки не константных функций из стандартной библиотеки Oberon-7, чьи тела написаны уже на языке C и подключаются на момент генерации кода. Вершины добавляются в дерево при встрече в исходной программе DeclarationSequence, при этом, так как процедуры не обязаны иметь секцию декларации, эти вершины могут быть убраны. Парсер во время обхода хранит указатель на текущую DeclarationSequence.

При этом существуют ещё деревья деклараций для записей. Они не имеют заранее заполненного нулевого уровня, потому что все типы их полей должны быть заполнены при декларации и берутся из текущей вершины основного дерева.

5.2.5 Организация StatementSequence

StatementSequence – это просто список из Statement, которые уже могут быть разные и содержать в себе другие StatementSequence.

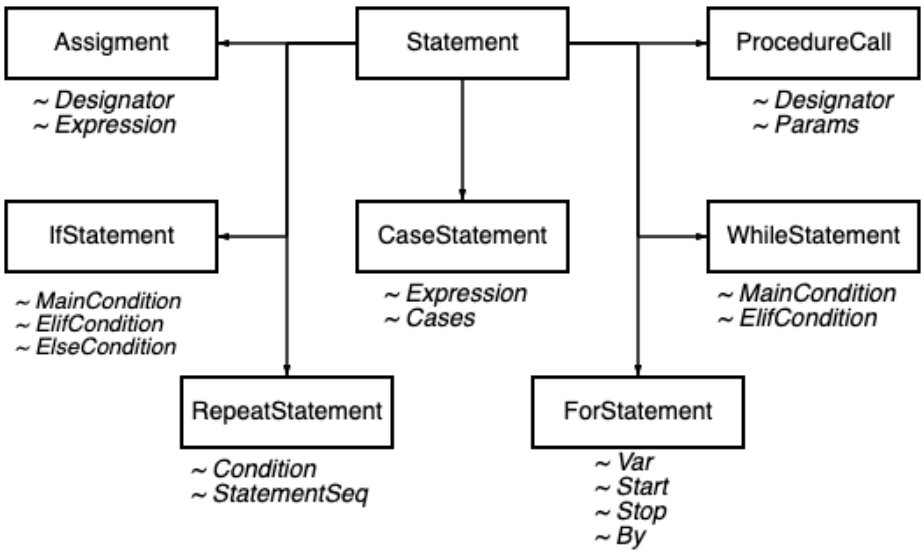


Рисунок 5. Схема организации StatementSequence

На рисунке 5 стрелками обозначается наследование. Все statement'ы, кроме Assigment содержат в себя одну или множество StatementSequence. Из необычного While в Oberon-7 умеет выполнять разные действия, в зависимости от того, какое из условий выполнено.

CaseStatement допускает в условии кейсов как переменные типа Integer и Char, так и типы структур. Во втором случае он проверят, запись какого типа содержит в себе указатель на базовый класс и, в зависимости от этого, выполняет различные операции. Это значит, что компилятор, в зависимости от того какой тип передан в кейс, сравнивает либо типы переменных, либо тип переменной и переданный тип. При этом отсекаются случи, когда, например, в условие передаётся тип Integer, а не переменная типа Integer.

Во всех statement'ах так или иначе фигурируют Designator и Expression. Рассмотрим подробнее, как они реализованы в объектной модели.

5.2.5.1 Организация Designator

Десигнатор состоит из имени переменной или импортируемого модуля и списка селекторов. Схема организации селектора показана на рисунке 6. В качестве примера можно привести $products[i + 1, 2].name$. Здесь $products$ – это идентификатор, $[i + 1,2]$ – IndexSelector, $.name$ –RecordSelector.

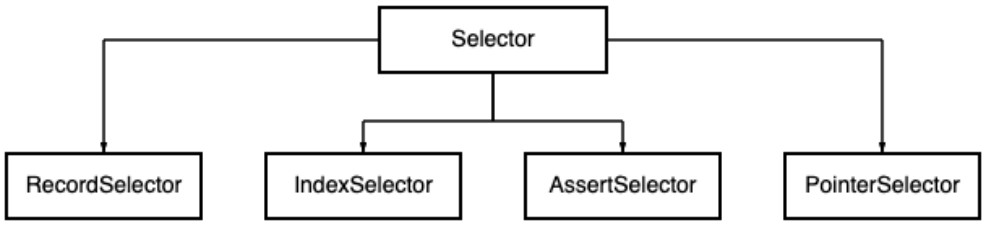


Рисунок 6. Схема организации Selector

Десигнатор поддерживет всю нужную информацию о типе после добавления очередного селектора. Он проверяет, что RecordSelector обращается только к структуре или указателю, что данное поле есть в декларации структуры или у одного из его предков, что число индексов в IndexSelector не превышает размерности массива и другие необходимые проверки. AssertSelector – довольно необычный селектор. Он проверяет во время исполнения, что в указателе на базовый тип лежит запись указанного типа и, в случае если это не так, завершает программу с ошибкой.

5.2.5.2 Организация Expression

Expression – это способ записать выражение, расставив правильно приоритеты операций. Самый верхний уровень – это класс Expression, он обрабатывает операторы отношения ($==, <=$ и подобные). Он хранит в себе до двух SimpleExpression. SimpleExpression обрабатывает оператор "или", а также операторы "сложения" и "вычитания". SimpleExpression хранит список Term. Term обрабатывает операторы умножения и хранит список Factor. Factor хранит либо конкретную переменную, либо константу, либо выражение, записанное в скобках. Конкретная переменная может быть передана в Factor различными путями, например, через Designator или как результат вызова процедуры. Например, в выражении $a + b * (7 + \log(c))$ все выражение является SimpleExpression, термами является $b * (7 + \log(c))$, а факторами $a, b, (7 + \log(c)), 7, \log(c)$.

Выражение также поддерживает, является ли его значение какой-то переменной, у которой можно взять адресс. Например, a и (a + b) – это два выражения, при этом первое является переменной, а второе нет. Необходимость знать, можно ли у результата выражения взять адрес, вызвана возможностью передачи переменных в функцию по ссылке и по значению. Формальные

параметры функции всегда задаются как выражения, но компилятор должен проверять, что по ссылке передаются только переменные, имеющие адрес.

5.3 Архитектура генератора кода

5.3.1 Общая структура

Генератор кода представлен базовым классом Generator, имеющим 39 различных виртуальных функций, генерирующих тот или иной объект объектной модели.

Такой подход позволяет реализовывать генераторы в различные языки из одной объектной модели, достаточно написать генератор на нужный язык, который будет наследником базового генератора. В проекте написан генератор в язык C. В нем, помимо реализации виртуальных функций базового класса, написаны приватные вспомогательные функции.

Принцип взаимодействия с генератором следующий. Компилятор вызывает одну функцию GenerateModule, передавая в неё модуль из объектной модели, имя итогового файла, а также stringstream-ы, куда будет записан код h-файла и c-файла. Эта функция уже вызывает нужные функции у генератора. GeneratorC дружественен классам объектной модели. Поскольку многая информация в объектной модели хранится в указателях на базовый класс, у объектов есть виртуальная функция Generate, которая принимает указатель на генератор и запускает у генератора нужную функцию.

Генератор в язык C записывает в h-файл экспортируемые переменные и описание экспортируемых процедур. В c-файлы записываются неэкспортируемые переменные и тела всех процедур. Также генератор создаёт функцию, которая запускает StatementSequence модуля.

Рассмотрим подробнее реализации некоторых нетривиальных генераций.

5.3.2 Генерация записей

Записи могут быть заданы явно, как тип с именем, и неявно, как структура записи без имени, например, внутри указателя. В коде запись должна быть определена один раз во время декларации типов. После этого используется только имя типа, тело записи заново не декларируется. Если запись задана неявно внутри указателя, то её тело определяется один раз во время декларации типа указателя.

Важно не уйти в бесконечную рекурсию в типах, аналогичных следующему:

```
Node = POINTER TO RECORD
  rec    : POINTER TO AnyRec;
  next   : Node
END;
```

Для этого генератор поддерживает имена типов, генерация которых уже началась, но ещё не закончилась.

Рассматривая подробнее этот пример с Node, можно заметить, что он не переписывается тривиально на C. То есть, код $typedef\ struct\ \{\ AnyRec\ *\ rec, Node\ *\ next\}\ *\ Node;$ не скомпилируется. Для поддержки таких типов генератор даёт свои имена записям без имени и помнит, какие типы ещё не до конца определены. Результатом компиляции в язык C указателя Node будет:

```
typedef struct rec_Node {
    char* __record_type;
    AnyRec* rec;
    struct rec_Node* next;
}* Node;
```

Здесь компилятор выдал безымянной структуре имя rec_Node, запомнил, как оно соотноситься с определяемым типом Node и подменил Node внутри структуры на rec_Node*. Oberon-7 запрещает подчёркивания в именах типов, переменных и функций, так что имена, выданные компилятором, не пересекутся с именами, заданными в тексте исходной программы.

В сгенерированном типе rec_Node можно заметить поле $char * \_\_record\_type$, которого нет в исходном типе. Это дополнительное поле, которое добавляется к каждой сгенерированной записи. В нем лежит имя типа данной процедуры. Это необходимо для того, чтобы быстро определять, какая запись лежит в указателе на базовую запись. Язык Oberon-7 позволяет программисту очень ловко оперировать указателями на записи. Например, AssertSelector и CaseStatement проверяют, какая из записей-расширений (наследников) лежит в указателе на базовую запись.

Введение такого поля позволяет сделать подобные сравнения, просто сравнив значения полей __record_type, но усложняет процесс генерации записей. Дело в том, что это поле должно быть инициализировано у переменной с данным типом до того, как ей начали пользоваться. А, поскольку, в языке С нет конструкторов, обязанность в инициализации данного поля ложится на компилятор. Детали инициализации будут рассмотрены позже.

5.3.3 Генерация массивов

Массивы нельзя генерировать в обычные массивы С, поскольку они могут передаваться в процедуры и в любой момент может быть вызвана функция LEN, возвращающая длину массива. Поэтому массив генерируется в структуру, состоящую из длины и указателя на первый элемент массива. Соответственно, двумерный массив будет содержать указатель на массив одномерных массивов. То есть массив, заданный в Oberon-7 как: $pinMap: ARRAY\ 16\ OF\ ARRAY\ 16\ OF\ ARRAY\ 16\ OF\ INTEGER;$ будет сгенерирован в язык С следующим образом:

```
struct {
    size_t lenght;
    struct {
        size_t lenght;
        struct {
            size_t lenght;
            long long* value;
        }* value;
    }* value;
} pinMap;
```

Соответственно, компилятор также подменяет все обращения к массиву. Такая система генерации позволяет без проблем передавать в процедуру массивы любой длины и легко узнавать длину текущего массива.

Из недостатков, такая система заставляет генератор инициализировать, во-первых, длину, а во-вторых, указатель на массив.

5.3.4 Инициализация записей и массивов

Первое, что надо понять генератору – это место инициализации. Переменные модуля считаются глобальными, поэтому для них инициализация сразу после декларации невозможна и происходит в начале StatementSequence модуля. Для деклараций внутри процедур такой проблемы нет, поэтому там инициализация происходит сразу после декларации. Сложнее с декларациями внутри записи. У записей нет StatementSequence, поэтому поля записи инициализируются там же, где инициализируется запись. Отсюда видно, что инициализация может быть рекурсивной, ведь в записи может лежать массив других записей.

Сама инициализация записи довольно проста, достаточно инициализировать __record_type, а после инициализировать все поля записи, которые являются записями или массивами.

С инициализацией массива все куда сложнее. Во-первых, надо пройтись по всем подмассивам многомерного массива и инициализировать все длины. Во-вторых, надо где-то взять массивы нужных размерности и типа, чтобы инициализировать указатель на первый элемент массива. Для этого, в той же области видимости, создаются вспомогательные массивы, указатели на которые потом и используются. Для обхода подмассивов генерируются циклы for с вспомогательными переменными. Для примера рассмотрим инициализацию массива pinMap. Компилятор создаёт следующие вспомогательные массивы:

```
struct {
    size_t lenght;
    struct {
        size_t lenght;
        long long* value;
    }* value;
} pinMap_base_ref0[16];

struct {
    size_t lenght;
    long long* value;
} pinMap_base_ref1[16][16];

long long pinMap_base_ref2[16][16][16];
```

После этого происходит следующая инициализация:

```
pinMap.lenght = 16;
pinMap.value = (void*) pinMap_base_ref0;
for (size_t i0 = 0; i0 < 16; i0 += 1) {
    pinMap.value[i0].lenght = 16;
    pinMap.value[i0].value = (void*) pinMap_base_ref1[i0];
    for (size_t i1 = 0; i1 < 16; i1 += 1) {
        pinMap.value[i0].value[i1].lenght = 16;
        pinMap.value[i0].value[i1].value = (void*) pinMap_base_ref2[i0][i1];
    }
}
```

5.3.5 Генерация процедур

У процедур DeclarationSequence содержит переменные как объявленные в процедуре, так и переданные. Вторые не нуждаются в генерации в теле процедуры и инициализации. Такие переменные помечаются в DeclarationSequence.

Язык Oberon-7 позволяет передавать переменные в процедуры по ссылке, в то время как в языке C ссылок нет. Поэтому генератор, когда видит передачу в процедуру по ссылке, подменят её на передачу по указателю. Внутри же процедуры такая переменная заменяется на (*__ptr_name).

5.3.6 Другие нетривиальные генерации

Среди statement-ов особое место занимает CaseStatement. Это связано с тем, что в случае успешной проверки типа записи, лежащей в указателе на базовую запись, с указателем можно обращаться как с указателем на конкретное расширение (наследника). Для этого генератор

создаёт временные вспомогательные указатели нужного типа и подставляет их в нужные места StatementSequence.

В языке C нет аналога AssertSelector из Oberon-7, но данный селектор легко представляется в виде конвертирования указателя и assert на __record_type. Проблема заключается в том, что в Oberon-7 AssertSelector может быть посередине десигнатора, а временные переменные на каждый AssertSelector не являются хорошим решением. В итоге было сделано следующее: написана вспомогательная функция в базовой библиотеке, принимающая указатель и тип для сравнения, выполняющая assert и возвращающая указатель нужного типа. Но, для того чтобы её подставить вместо AssertSelector-а, нужно перед десигнатором написать её вызов и первым параметром передать в нее часть десигнатора до AssertSelectora. Поскольку такой селектор в десигнаторе может быть не один, изначально подсчитывается их количество, затем заранее пишутся вызовы всех AssertSelector и правильно расставляются скобки.

Аналогичная проблема возникает в генерации выражений, в которых участвуют переменные типа Set. Какие-то операции с сетами, такие как объединение, пересечение и симметрическая разность могут быть кодированы в одну битовую операцию и не вызывают проблем. Но вычитание одного множества из другого не представляется в виде одной операции, их нужно хотя бы две. Это ломает выражения, в которых такая операция находится где-нибудь посередине. Решение такое же, как и в предыдущей проблеме. В стандартной библиотеке написана функция set_difference, возвращающая результат вычитания одного множества из другого. В выражении заранее подсчитывается количество вычитаний множеств, и вызовы функций расставляются на правильные места.

5.3.7 Пример сгенерированного кода

В качестве примера сгенерированного кода рассмотрим код сортировки слиянием (рисунки 7 и 8)

```
1  PROCEDURE MergeSort* (VAR arr: ARRAY OF INTEGER; l, r : INTEGER);
2  VAR i, a, b, cur, m: INTEGER;
3    newArr : ARRAY r - l OF INTEGER;
4  BEGIN
5    IF r - l > 1 THEN
6      m := (r + l) DIV 2;
7      MergeSort(arr, l, m);
8      MergeSort(arr, m, r);
9      a := l;
10     b := m;
11     cur := 0;
12     FOR i := 0 TO r - l - 1 DO
13       IF a = m THEN
14         newArr[i] := arr[b];
15         INC(b);
16       ELSIF b = r THEN
17         newArr[i] := arr[a];
18         INC(a);
19       ELSIF arr[a] < arr[b] THEN
20         newArr[i] := arr[a];
21         INC(a);
22       ELSE
23         newArr[i] := arr[b];
24         INC(b);
25       END;
26     END;
27
28     FOR i := 0 TO r - l - 1 DO
29       arr[l + i] := newArr[i];
30     END;
31   END;
32 END MergeSort;
```

Рисунок 7. Реализация сортировки слиянием на Oberon-7

```
1  void MergeSort(BASE_ARRAY* __base_ptr_arr, long long l, long long r) {
2      long long i;
3      long long a;
4      long long b;
5      long long cur;
6      long long m;
7      struct {
8          size_t length;
9          long long* value;
10     } newArr;
11     long long newArr_base_ref0[r - l];
12     newArr.length = r - l;
13     newArr.value = (void*) newArr_base_ref0;
14     struct {
15         size_t length;
16         long long* value;
17     }* __ptr_arr = (void *) __base_ptr_arr;
18     if (r - l > 1) {
19         m = (r + l) / 2;
20         MergeSort((BASE_ARRAY*) &(*__ptr_arr), l, m);
21         MergeSort((BASE_ARRAY*) &(*__ptr_arr), m, r);
22         a = l;
23         b = m;
24         cur = 0;
25         // For loop
26         i = 0;
27 _2:
28         if (i <= r - l - 1) {
29             if (a == m) {
30                 newArr.value[i] = (*__ptr_arr).value[b];
31                 INC(&b);
32             } else if (b == r) {
33                 newArr.value[i] = (*__ptr_arr).value[a];
34                 INC(&a);
35             } else if ((*__ptr_arr).value[a] < (*__ptr_arr).value[b]) {
36                 newArr.value[i] = (*__ptr_arr).value[a];
37                 INC(&a);
38             } else {
39                 newArr.value[i] = (*__ptr_arr).value[b];
40                 INC(&b);
41             }
42             i += 1;
43             goto _2;
44         }
45         // For loop
46         i = 0;
47 _3:
48         if (i <= r - l - 1) {
49             (*__ptr_arr).value[l + i] = newArr.value[i];
50             i += 1;
51             goto _3;
52         }
53     }
54 }
```

Рисунок 8. Сгенерированный код сортировки слиянием на C

На данном примере можно наблюдать передачу массива в процедуру (рисунок 8, строка 1), приведение массива к нужному типу (рисунок 8, строки 14-17) и заведение и инициализацию нового массива (рисунок 8, строки 7-13).

6 Тестирование

Для тестирования правильной работы компилятора использовались 30 настоящих программ на Oberon-7, находящиеся в $workspace/o7/big$. Большинство этих тестов использовали библиотеки, код которых отсутствовал, поэтому они скорее проверяли

правильности синтаксического анализа и совсем не проверяли генерацию кода. Для тестирования объектной модели и генератора кода были написаны юнит тесты, делящиеся на две категории: fail и success. Первые проверяют, что компилятор находит неправильные конструкции в исходной программе и возвращает ошибку, вторые же проверяют, что компилятор правильно строит объектную модель и генерирует корректный код. Тесты обширные и включают в себя множество необходимых конструкций, все типы деклараций, а также все statement-ы. Код на языке C, сгенерированный компилятором, компилируется и корректно запускается.

Весь процесс компиляции любого из написанных юнит тестов при замерах занял не более 4 миллисекунд.

7 Интерфейс

У компилятора есть следующие опции, которые можно задать:

- $-h$ $[--help]$ – вывести информацию о доступных опциях
- $-w$ $[--workspace-dir]$ dir – путь до директории $workspace$, в которой в директории $o7p$ находятся модули на Oberon-7. По умолчанию $../o7p/workspace$
- $-f$ $[--file-name]$ $oberon\_file.o7$ – путь к файлу с модулем относительно $workspace/o7$. Расширение ".o7" может быть опущено
- $-d$ $[--debug]$ – вывести debug информацию. Включает в себя построенную объектную модель и сгенерированный код

После запуска компиляции модуля $assigment.o7$ в директории $workspace$ автоматически сгенерируется следующие:

- В директории h будет сгенерирован файл $assigment.h$
- В директории c будет сгенерирован файл $assigment.c$
- В директории $main-c$ будет сгенерирован файл $main-assigment.c$, содержащий функцию main, запускающую инициализацию модуля
- В директории prj будет сгенерирован файл $CMakeLists.txt$, описывающий проект из исполняемого файла $main-assigment.c$ и библиотек $baselib$ и $assigment$

Для того, чтобы собрать проект на C достаточно из директории $prj/assigment$ запустить команду $cmake ./ \&\& make$ и в директории out появится исполняемый файл $assigment$.

8 Заключение

Результатом данной работы является компилятор из языка Oberon-7 в язык C, генерирующий не только необходимые файлы на языке C, но и Cmake файлы для создания проектов. Компилятор успешно проходит написанные тесты и генерирует работающий код. Реализованы все этапы компиляции, построена объектная модель, позволяющая генерировать код на любом языке при наличии генератора.

В дальнейшем возможна как замена генератора, так и замена парсера на парсер с C-подобного синтаксиса с сохранением семантики Oberon-7. Возможны расширения языка, связанные с включением в него поддержки процедурно-параметрического полиморфизма. Также в будущем возможно написание сборщика мусора, который будет отслеживать аллокации, сделанные функцией NEW и, при необходимости, очищать память.

Список источников

[1] The Programming Language Oberon, URL: https://people.inf.ethz.ch/wirth/Oberon/Oberon07.Report.pdf

[2] O7 Compiler, URL: https://github.com/aixp/O7

[3] Project "Vostok", URL: https://github.com/Vostok-space/vostok

[4] O7 Parser, URL: https://github.com/kreofil/o7p

[5] Astrobe, URL: https://www.astrobe.com/

[6] Oberon-07 compiler, URL: https://github.com/AntKrotov/oberon-07-compiler

[7] Exaprog Oberon-07 compiler, URL: http://exaprog.com/rus/index.html

[8] Patchouli Oberon-07 Compiler, URL: https://github.com/congdm/Patchouli-Compiler

[9] Clang, URL: https://clang.llvm.org/