Министерство образования и науки Российской Федерации

Федеральное государственное автономное образовательное учреждение высшего профессионального образования
ДАЛЬНЕВОСТОЧНЫЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ

Школа естественных наук
Кафедра информатики, математического и компьютерного моделирования

Кевролетин Василий Владимирович

Реализация замыканий для языка программирования Pascal

ДИПЛОМНАЯ РАБОТА
по основной образовательной программе подготовки специалистов по специальности 010501.65 – прикладная математика и информатика

Студент ОЗО _____
(подпись)
Руководитель ВКР – ст. преп.
(должность)
_____ А.С. Кленин
(подпись) (и.о.ф)
«____»_____ 2013г.

Защищена в ГАК с оценкой _____
Секретарь ГАК
_____ _____
(подпись) (и.о.фамилия)
«____»_____ 2013 г.

«Допустить к защите»
Заведующий кафедрой – д.ф.-м.н., проф.
(ученое звание, должность)
_____ А. Ю. Чеботарев
(подпись) (и.о.ф.)
«____»_____ 2013 г.

г. Владивосток
2013

Содержание

Аннотация

Цель данной работы - реализовать поддержку замыканий в компиляторе Free Pascal Compiler. Рассмотрены подходы к решению проблемы и осуществлена пробная реализация.

1. Введение

1.1. Глоссарий

Анонимная функция (англ. anonymous function) – функция, не имеющая идентификатора; доступна для вызова либо в момент создания, либо, позднее, через ссылку на функцию, хранящуюся в переменной.

Анонимный метод (англ. anonymous method) – реализация замыканий в языке программирования Delphi[5].

Замыкание (англ. closure) – функция или ссылка на функцию вместе с используемым ею окружением; замыкание сохраняет окружение и позволяет функции обращаться к нему даже если она вызвана за пределами лексического контекста, где была создана.

Захваченная переменная (англ. captured variable) – переменная, используемая в теле замыкания, объявленная во внешнем для замыкания лексическом контексте.

Интерфейс с подсчётом ссылок (англ. reference counted interface) – набор функций, используемый компилятором для автоматического управления памятью объектов. Класс, реализующий такой интерфейс, становятся управляемым типом данных.

Область видимости объявления (англ. scope) – область программы, в которой может использоваться это объявление.

Сборщик мусора (англ. garbage collector) – часть библиотеки времени исполнения, отвечающая за освобождение динамической памяти, занятой недостижимыми данными.

Свободная переменная (англ. free variable) – переменная, используемая в теле функции, но не являющейся параметром или локальной переменной этой функции.

Управляемый тип данных (англ. managed type) – тип данных, для которого компилятор использует специальные техники управления динамической памятью, такие как подсчёт ссылок[6].

Функциональное программирование – парадигма программирования, рассматривающая вычисление как применение математических функций, отрицающая понятие состояния и изменяемых данных.

1.2. Описание предметной области

1.2.1. Компилятор

Компилятор – это программа, которая считывает текст программы, написанной на одном языке – исходном, и транслирует его в эквивалентный текст на другом языке – целевом. В то же время компилятор – огромная программная система с большим количеством внутренних компонентов и алгоритмов, имеющих сложные взаимосвязи. Для того чтобы справиться с этой сложностью разработчики используют хорошо известные и описанные в литературе алгоритмы, структуры данных и шаблоны проектирования, применимые для написания компиляторов.

Как правило, компилятор выполняет только часть работы по созданию исполняемого файла. Широко распространена связка, содержащая [1]:

1. Препроцессор.
2. Компилятор.
3. Ассемблер.
4. Компоновщик.

Любой компилятор высокоуровневого языка программирования выполняет две задачи: анализ исходной программы и генерация целевого кода. На этапе анализа компилятор разбивает программу на составные части, разбирает её синтаксическую структуру и проверяет на корректность. Генерация кода строит требуемую целевую программу на основе промежуточного представления, полученного на этапе анализа. Обычно анализ называют начальной стадией, а генерацию кода конечной.

Анализ и генерация кода представляют собой последовательности фаз, каждая из которых преобразует одно из представлений исходной программы в другое. Типичное разложение начальной стадии на фазы [1]:

1. Лексический анализ.
2. Синтаксический анализ.
3. Семантический анализ.

4. Генерация промежуточного кода.

5. Машинно - независимая оптимизация.

Типичное разложение конечной стадии на фазы:

1. Генерация кода.

2. Машинно - зависимая оптимизация.

1.2.2. Free Pascal Compiler (FPC)

FPC – компилятор языка программирования Pascal с открытым исходным кодом, поддерживающий несколько диалектов Pascal и генерирующий код для большого числа процессорных архитектур и операционных систем [13]. Это открытый проект, разрабатываемый постоянной командой добровольцев.

FPC объединяет в себе препроцессор, компилятор, ассемблер и компоновщик[1]. Кроме того синтаксический анализатор FPC используется интегрированной средой разработки Lazarus [17].

Выдающаяся особенность FPC - поддержка нескольких диалектов Pascal и большого числа целевых процессорных архитектур. Исходный код FPC написан со стремлением минимизировать количество повторяющегося программного кода, работающего в случае разных комбинаций диалекта языка и результирующей платформы. Поэтому для каждой целевой платформы в компиляторе реализован отдельный генератор кода. Все остальные части компилятора общие для разных целевых платформ. В режимах совместимости с различными диалектами Pascal лексический, синтаксический и семантический анализаторы работают немного по-разному. Благодаря этому компилятор может иметь различную реализацию одной концепции языка в разных режимах совместимости.

FPC находится в состоянии активной разработки. Разработчики постоянно улучшают компилятор, добавляя новые возможности, оптимизации, поддержку новых платформ и исправляя ошибки[13]. Одной из новых возможностей, которую разработчики и пользователи FPC хотят увидеть в своём компиляторе, является поддержка замыканий.

[1] Для некоторых целевых платформ используются внешние ассемблер или компоновщик.

1.2.3. Delphi

Еще одна важная особенность FPC – совместимость с языком программирования Delphi. Delphi – коммерческая среда для быстрой разработки приложений, основанная на одноимённом языке программирования. Язык программирования Delphi является диалектом Pascal. Разработчики постоянно развивают этот язык. К примеру, в 2009м году была добавлена поддержка обобщенного программирования.

Разработчики FPC для совместимости добавляют в свой компилятор новые концепции, реализованные в Delphi. FPC никогда не достигал полной совместимости с Delphi, тем не менее, благодаря реализации большинства важных особенности Delphi, компилятор FPC способен корректно компилировать большое множество её программ и библиотек.

Начиная с 2009го года, Delphi поддерживает замыкания[7]. Отсутствие замыканий в FPC снижают его совместимость с Delphi и уменьшают привлекательность компилятора для пользователей.

1.2.4. Замыкания

В современных языках программирования замыкание это практическая реализация идеи лябмда-исчесления о том, что функция может использовать в своём теле свободные переменные(не являющиеся параметрами этой функции). В лямбда-исчислении переменные неизменяемые, поэтому для вычисления значения функции, использующей свободные переменные, достаточно подставить значения свободных переменных в тело функции[24].

В императивных языках программирования переменные изменяемые. Поэтому результат выполнения функции зависит не только от значений параметров функции, но и от значения свободных переменных, используемых в её теле. В качестве свободных переменных могут выступать глобальные переменные и локальные переменные других функций. Глобальные переменные доступны в течении всего времени исполнения программы. Однако срок жизни локальных переменных ограничен временем работы функции, в которой они объявлены. Особенность замыканий состоит в том, они не только могут получать доступ к любым доступным в текущем лексиче-

ском контексте переменным, но, к тому же, могут обращаться к ним, даже если будут вызваны в другом лексическом контексте, где использованные переменные уже недоступны.

Если замыкание ссылается на локальную переменную объемлющей функции, то такая переменная называется "захваченной"[5][25]. Существует два способа захвата переменных: по значению и по ссылке. В случае захвата по значения замыкание в момент своего создания запоминает значение захваченных переменных. Захват по значению – аналог подстановки значений свободных переменных в лябмда-исчислении. В случае захвата по ссылке, в момент своего создания замыкание запоминает ссылку на захваченную переменную. Сложность реализации захвата переменных по ссылке состоит в том, что на одну переменную могут одновременно ссылаться несколько замыканий и несколько выполняющихся функций.

1.2.5. Описание совместной деятельности

Компилятор распространяется под лицензией GNU GPL[16]. Разрабатывается постоянной командой добровольцев и принимает доработки от сторонних разработчиков. Исходный код написан на языке Free Pascal. Разработка ведётся с использованием системы контроля версий SVN[12], системы контроля изменений Mantis[18]. Команда общается при помощи списков рассылки электронных писем.

1.3. Неформальная постановка задачи

Добавить в компилятор Free Pascal поддержку замыканий. В режиме совместимости с Delphi реализация должна вести себя так же, как и анонимные методы Delphi. Дополнительно рассмотреть возможность создания альтернативной улучшенной реализации, не ограниченной требованием совместимости с Delphi. При этом компилятор должен сохранить обратную совместимость, проходить существующие тесты.

1.3.1. Анонимные подпрограммы

В некоторых языках программирования, анонимные функции - единственный способ создать замыкание. Тем не менее эти понятия не тождественны. Анонимная функция - это функция не имеющая идентефикатора. Существуют языки программирования, поддерживающие анонимные функции, но не поддерживающие замыкания. Компилятор FPC должен поддерживать объявление анонимных функций в теле других подпрограмм:

```
function Factory: TProc;
begin
  Result := procedure begin
              Writeln('executed');
            end;
end;
```

1.3.2. Вложенные функции

Вложенная функция может обращаться к переменным объемлющих функций. Некоторые языки программирования позволяют создавать замыкания при помощи вложенных анонимных функций. На данный момент FPC и Delphi поддерживают вложенные именованные функции, но не позволяют создавать с ними замыкания. Без поддержки замыканий ссылка на вложенную функцию не сохраняет окружение. Это значит, что вложенную подпрограмму нельзя вызывать по сохранённой ссылке после того, как работа объемлющей подпрограммы завершилась. Для обратной совместимости семантика вложенных подпрограмм должна остаться неизменной. Вложенные процедуры не должны создавать замыкания. В дальнейшем возможно создание нового синтаксиса для создания замыканий с использованием вложенных именованных функций.

Ниже приведён пример вложенной функции, использующей переменную объемлющей функции. Использование ссылки на вложенную функцию после окончания работы объемлющей подпрограммы, в которой была создана эта ссылка, является ошибкой.

```
procedure outer;
var i: Integer;
```

```
  procedure inner; begin
    i := 10;
  end;

begin
  ...
end;
```

1.3.3. Продление жизни локальных переменных

Замыкание в момент создания сохраняет используемое окружение. Типичной является ситуация, когда замыкание использует локальные переменные объемлющей функции. Без поддержки замыканий время жизни локальных переменных ограничено временем выполнения функции, где она объявлена. Время жизни переменной, захваченной из замыкания заканчивается только когда не осталось использующих её замыканий.

Ниже приведён пример анонимной функции, использующей переменную объемлющей функции. Без замыканий переменная data стала бы недоступной после завершения работы функции Factory. Замыкание продлевают жизнь захваченной переменной data, и может обращаться к ней даже после окончания работы функции Factory. В момент второго вызова процедуры Factory переменная data, захваченная первым замыканием, покинула зону видимости. Теперь она доступна только ссылающемуся на неё замыканию. Следующий вызов Factory создаст другое замыкание, ссылающееся на другую переменную data. Каждое из созданных замыканий хранит ссылку на свой собственный экземпляр захваченной переменной. Вызов f1() напечатает значение 10, а вызов f2() напечатает 20.

```
function Factory(data: Integer): TProc;
begin
  Result := procedure
            begin
              Writeln( data );
            end;
  end;

var f1: TProc;
begin
  f1 := Factory(10);
  f2 := Factory(20);
```

```
  f1();                  { 10 }
  f2();                  { 20 }
end.
```

1.3.4. Захват по ссылке

В языке программирования Delphi замыкания захватывают переменные по ссылке. Необходимо реализовать такой же подход. Для захвата по ссылке типична ситуация, когда захваченную замыканием переменную меняют из другого замыкания, или из обычной подпрограммы. Приведённый ниже код демонстрирует, как сразу два созданных замыкания используют одну и ту же переменную. Кроме того, захваченная переменная используется в теле функции, где она объявлена.

```
type TProcInt: reference to procedure(n: Integer);
     TProc: reference to procedure;
var data: Integer;
    pset: TProcInt;
    pget: TProc;
begin
  pset := procedure(n: Integer) begin
            data := n;
          end;
  pget := procedure begin
            Writeln(data);
          end;
  data := 0;
  pget();                { 0 }
  pset(10);
  pget();                { 10 }
end.
```

1.3.5. Захват по значению

Рассмотреть возможность реализации захвата по значению. Пример ниже демонстрирует возможный синтаксис, позволяющий явно указывать, как захватывать переменные: по ссылке или по значению. За описанием сигнатуры функции следует описание захваченных переменных. После ключевого плова closure в круглых скобках через запятую указываются имена захваченных по значению переменных. Остальные переменных, используемые в теле функции захватываются по ссылке.

```
var i: Integer;
    f: TProc;
begin
  i := 0;
  f := procedure
       closure(i)
       begin
         Writeln(i);
       end;
  i := 10;
  f();                    { 0 }
end.
```

1.3.6. Упрощенный синтаксис

Рассмотреть возможность реализации упрощенного синтаксиса для более лаконичного объявления анонимных функций. В примере ниже в метод map передаётся анонимная функция, принимающая один аргумент типа Integer и возвращающая значение аргумента, увеличенное на 1.

```
type TListVisitor = function(x: Integer): Integer;
var  list: TIntList;
begin
   ...
   list.map(TListVisitor is x + 1)
   ...
end.
```

1.4. Обзор существующих методов решения

1.4.1. Аналогичные решения

Ниже приведена таблица, показывающая, какие из популярных языков программирования поддерживают замыкания. Для языков с замыканиями указано, как именно можно создавать замыкания. При помощи анонимных функций или вложенных именованных функций. Так же приведена информация о том, какой способ захвата использует соответствующий язык.

Некоторые языки имеют особенности, помеченные в таблице знаком +/-. Язык программирования Перл позволяет объявлять вложенные функции, но к ним не применяются обычные правила статической области види-

ЯП	Анонимные функции	Вложенные функции	Захват по значению	Захват по ссылке	Замыкания
Perl	+	+/-		+	+
Python	+	+		+	+
Ruby	+	+		+	+
Scheme	+	+		+	+
Elisp	+	+		+/-	
Scala	+	+		+	+
Java				+	+/-
C					
C++	+		+	+/-	+
Delphi	+	+		+	+
Fpc		+			

Таблица 1: Поддержка замыканий и анонимных функций современными ЯП

мости переменных. Язык программирования Elisp имеет динамическую область видимости переменных. Java позволяет объявлять анонимные классы, содержащие именованные методы, но не позволяет объявлять анонимные функции. C++ предоставляет синтаксис для доступа к переменным по ссылке, но не продлевает жизнь захваченным переменным.

Сегодня большинство популярных языков программирования поддерживают замыкания. В некоторые языки они были добавлены недавно. К примеру, в стандарт c++ замыкания попали в 2011м году. В Java полная поддержка замыканий станет доступной в версии Java 8, запланированной на середину текущего года.

1.4.2. Описание предшествующих работ

Студенты нашей кафедры успешно развивали компилятор Fpc в рамках своих курсовых и дипломных работ:

1. Дипломная работа «Расширение компилятора Free Pascal для поддержки обобщённого программирования». Автор Нелепа А.А. Руководитель Кленин А. С. 2007г.[10]

2. Курсовая работа «Анализ потоков управления для языка программи-

рования Pascal» Автор Баль Н.В. Руководитель Кленин А.С. 2008г.[2]

3. Курсовая работа «Доработка компилятора Free Pascal: case of string» Автор Денисенко М.В. Руководитель Кленин А.С. 2009г.[4]

4. Курсовая работа «Оператор for-in для компилятора Free Pascal» Автор Лукащук М.А. Руководитель Кленин А.С. 2010г.[9]

1.4.3. Вывод

Ценность замыканий подтверждена их популярностью и востребованностью. Сегодня замыкания поддерживают большинство языков программирования, а программисты активно используют их на практике. Поэтому для поддержания компилятора FPC в актуальном состоянии необходимо реализовать в нём поддержку замыканий.

1.5. План работ

1. Согласовать работу с разработчиками FPC.
2. Изучить архитектуру и исходных код компилятора.
3. Изучить реализацию замыканий в похожих языках программирования.
4. Спроектировать реализацию.
5. Поэтапно осуществить реализацию:
   - Добавить возможность объявлять анонимные функции. Запретить захват переменных.
   - Добавить возможность захвата переменных функции, объявляющей замыкание.
   - Добавить возможность захвата любых локальных переменных, доступных в текущем лексическом контексте[2].

[2]За исключением случаев, описанных в разделе "Функциональные требования".

2. Требования к окружению

FPC может компилировать свой собственный исходный код. Поэтому список целевых платформ совпадает со списком платформ, на которых работает FPC.

2.1. Требования к аппаратному обеспечению

Для работы требуется компьютер, пригодный для набора исходного кода программы. Т.е. кроме работающих процессора, оперативной и постоянной памяти требуются клавиатура и монитор.

Поддерживаемые архитектуры процессоров[15]:

- I386
- PowerPC
- Sparc
- AMD64 (x86-64)
- PowerPC64
- ARM
- m68k

2.2. Требования к программному обеспечению

2.2.1. Компилятор

FPC версии 2.6 и выше.

2.2.2. Операционная система

На сайте FPC указано 53 разных поддерживаемых комбинаций архитектуры процессора и операционной системы[15]. Здесь приведём только популярные операционные системы, поддерживаемые компилятором для архитектуры процессора I386:

- Win32 for i386
- Linux for i386
- Target Darwin (Mac OS X) for i386 (2.1.x and later)
- FreeBSD/ELF for i386
- Android for i386

2.3. Требования к пользователям

Программисты, владеющие языком программирования Free Pascal или Delphi.

3. Архитектура системы

Установленный проект FPC содержит следующие каталоги:

Compiler содержит исходный код компилятора и некоторых вспомогательных программ. Компилятор – программа с интерфейсом командной строки. На вход принимает список текстовых или объектных файлов. Результат его работы – файлы с ассемблерным кодом, описанием модулей, отладочной информацией, информацией для межмодульной оптимизации и исполняемый файл. Стоит так же отметить вспомогательную программу ppudump, переводящую файл описания модуля в текстовый наглядный формат. Изменения, сделанные в рамках данной работы затронули эту утилиту.

Ide содержит исходные коды среды разработки с интерфейсом командной строки.

Installer содержит скрипты для установки программы в систему.

Packages содержит исходные коды стандартной библиотеки, не обязательной для работы скомпилированной программы.

Rtl содержит исходные коды библиотеки времени исполнения, необходимой для работы любой скомпилированной программы. К примеру,

менеджер динамической памяти и подпрограммы управления памятью управляемых объектов реализованы здесь.

Tests содержит набор тестов. В ходе выполнения данной работы были созданы новые тесты.

Utils содержит вспомогательные утилиты, не используемые компилятором, предназначенные пользователям.

Несмотря на то, что для конечного пользователя компилятор - это монолитная программа, преобразующая файлы из одного представления в другое, в рамках данной работы необходимо рассмотреть внутреннюю архитектуру самого компилятора. Это необходимо для обоснования принятых решений, последствия которых испытают конечные пользователи. К таким решениям относятся, к примеру, размер указателя на замыкание, либо правила управления памятью, выделенной под замыкания.

3.1. Архитектура компилятора

Весь программный код можно условно разделить на начальную стадию(англ. front-end) и заключительная стадию(англ. back-end)[1].

Начальная стадия считывает входные данные, проверяет их на корректность, строит вспомогательные структуры данных, необходимые для генерации кода и передаёт управление заключительной стадии, которая генерирует код.

Смысл разделения на начальную и заключительную стадию в том, чтобы отделить детали анализа языков высокого уровня от деталей, относящихся к целевой архитектуре. Такой подход позволяет уменьшить сложности одновременной поддержки несколько диалектов Pascal и большого числа целевых архитектур.

3.1.1. Начальная стадия

После запуска FPC обрабатывает параметры и проводит предварительную обработку исходного кода входной программы. Далее он целиком считывает определение и тело одной подпрограммы, проводит необходимые

проверки и генерирует для неё код высокоуровневого ассемблера. Затем считывает следующую подпрограмму и так далее до конца файла.

Обработка одной подпрограммы на начальной стадии включает в себя:

1. Лексический анализ.
2. Синтаксический анализ.
3. Семантический анализ.
4. Машинно-независимую оптимизацию.

Операции, производимые любым компилятором можно реализовывать в виде отдельных проходов по синтаксическому дереву или потоку лексем. К примеру, можно полностью считать лексемы входного файла и получить поток лексем. Затем запустить синтаксический анализ и получить синтаксическое дерево, соответствующее содержимому файла. Затем семантический анализ, который не производит никаких модификаций и лишь проверяет синтаксическое дерево на корректность. И лишь потом производить преобразования синтаксического дерева.

Описанная выше схема уменьшает сложность исходного кода компилятора. Программу, в которой отдельные компоненты совершают чётко определённые действия легко документировать и понимать. К сожалению, многократный обход структур данных, расположенных в памяти непоследовательно, вносит накладные расходы[8], поэтому зачастую несколько отдельных проходов объединяют в один. В компиляторе FPC стадии лексического, синтаксического и семантического анализа реализованы в виде одного прохода.

Кроме того, стадия семантического анализа, кроме вывода типов и проверки синтаксического дерева на корректность, производит вспомогательные модификации синтаксического дерева. FPC на этапе семантического анализа добавляет преобразования типов, заменяет операции с управляемыми данными на вызовы функций из библиотеки времени исполнения и трансформирует некоторые специфичные конструкции. К примеру, для загрузки адреса процедуры вставляется специальный узел синтаксического дерева, отличный от созданного лексическим анализатором.

3.1.2. Синтаксическое дерево

Синтаксическое дерево - иерархическая структура данных представляющая синтаксическую структуру исходной программы[1]. После синтаксического анализа каждый внутренний узел дерева представляет собой оператор, дочерние узлы представляют собой операнды этого оператора. Для представления синтаксического дерева и операций над ним FPC использует приём объектно-ориентированной декомпозиции. В [3] такой подход описан под названием "composite"(англ. составной).

Все узлы синтаксического дерева унаследованы от одного абстрактного класс tnode. tnode объявляет общий интерфейс для объектов синтаксического дерева. Унаследованные от tnode классы, реализуя этот интерфейс, определяют специфичное поведение. Отметим методы, реализующие обходы синтаксического дерева:

- pass_1 – выделение регистров;
- pass_typecheck – семантический анализ;
- pass_generate_code – генерация кода.

На рис. 1 приведён пример того, как использование наследования позволяет отделить общие для нескольких синтаксических конструкций детали от специфичных деталей отдельной конструкции. Так, tunarynode служит базовым классом для всех операций, содержащих 1 операнд. Унаследованный от него класс tloadvmtaddrnode реализует методы pass_1 и pass_typecheck, которые реализованы без использования информации о целевой архитектуре. Метод генерации кода pass_generate_code реализован в классе
tcgloadvmtaddrnode.

Таким образом, для каждой архитектуры процессора можно создать отдельный класс, унаследованный от tloadvmtaddrnode и реализовать генерацию кода, учитывая особенности конкретного процессора. Такой подход к модификации поведения объекта, переопределяя методы родительского класса, описан в [3] под названием "template"(англ. шаблон). Этот подход широко используется в FPC для отделений специфичных деталей генерации кода для узлов синтаксического дерева от остальной логики.

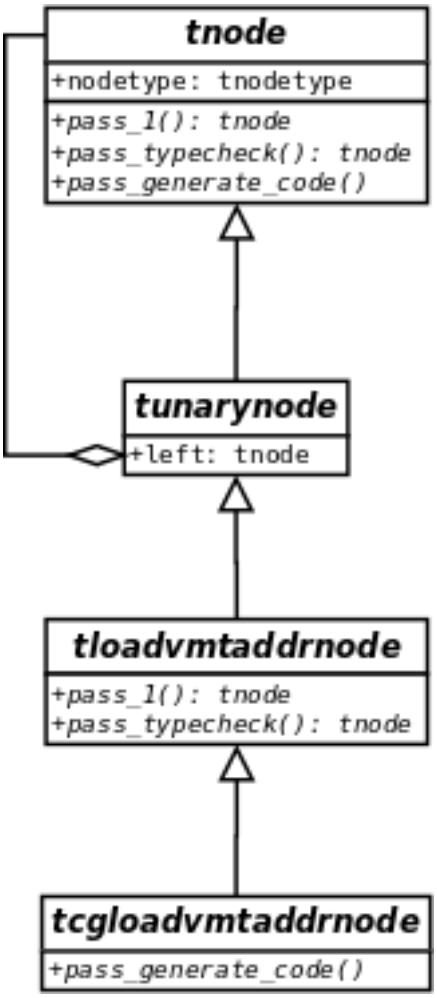

Рис. 1: Диаграмма наследования для класса tcgloadvmtaddrnode

3.1.3. Таблицы символов

Таблица символов это структура данных, которая используется компилятором для хранения информации о именованных объектах программы. Такая информация последовательно накапливается во время разбора синтаксических конструкций. Далее она используется на этапе семантического анализа для проверки соответствия типов и при генерации кода для размещения данных в памяти.

Записи в таблице символов содержат символьные строки, типы, области видимости, местоположение в памяти и прочую связанную с именованными объектами информацию. FPC использует таблицы символов для хранения информации о модулях, константах и переменных, подпрограммах, структуре составных типов данных.

Таблицы символов и синтаксическое дерево тесно связаны: синтакси-

ческое дерево описывает действия с данными. Записи таблицы символов описывают структуру этих данных.

3.1.4. Трансформации синтаксического дерева

Во время компиляции FPC неоднократно трансформирует полученное на этапе синтаксического анализа синтаксическое дерево. Это необходимо для:

- Упрощения логики отдельных узлов синтаксического дерева. К примеру, узлы преобразования типов упрощают логику узлов загрузки значения переменных, арифметических операций и др. выражений.
- Проведения машинно - независимой оптимизации. К примеру удаление недостижимого кода.
- Реализации новых концепций языка через старые. Примерами являются реализация case of string[4] а так же доступ к локальным переменным объемлющей функции для платформ, не поддерживающих арифметику указателей.

Возможность реализации новых концепций языка через старые можно использовать и для реализации замыканий. Заметим, что замыкание отличается от обычной подпрограммы наличием сохранённого состояния свободных переменных. Преобразование замыкания в функцию, не содержащую свободных переменных, называется преобразованием замыканий[11].

Преобразование замыканий давно используется в языках программирования, основанных на лямбда-исчислении. Прямая реализация модели подстановки лямбда-исчисления приводит к многократному вычислению одних и тех же выражений. Преобразованием замыканий позволяет таким языкам избежать бесполезных вычислений и улучшения производительности программ [26]. Функция со свободными переменными заменяется функцией с дополнительными параметром - окружением. Свободные переменные в теле исходной функции заменятся ссылками на окружение. Неизменяемость переменных в лямбда - исчислении позволяет реализовать сохранение переменных в окружение разными способами:

Преобразование замыканий применимо и к императивным языкам программирования. Статически типизированный язык программирования Scala использует преобразования замыканий[19][22]. Scala имеет статическую типизацию и допускает изменение значения переменных. Этом язык очень похожим на Free Pascal. Главное отличие – полностью автоматическое управление динамической памяти компилятором, решающее проблему управления памятью созданных окружений.

3.1.5. Заключительная стадия

В FPC нет промежуточного представления кода, который начальная стадия передавала бы в заключительную стадию. Вместо генерации промежуточного представления начальная стадия во время обхода синтаксического дерева последовательно вызывают методы высокоуровневого генератора кода. Такой подход очень похож на использование промежуточного представления, но избавляет от необходимости генерировать промежуточные структуры данных.

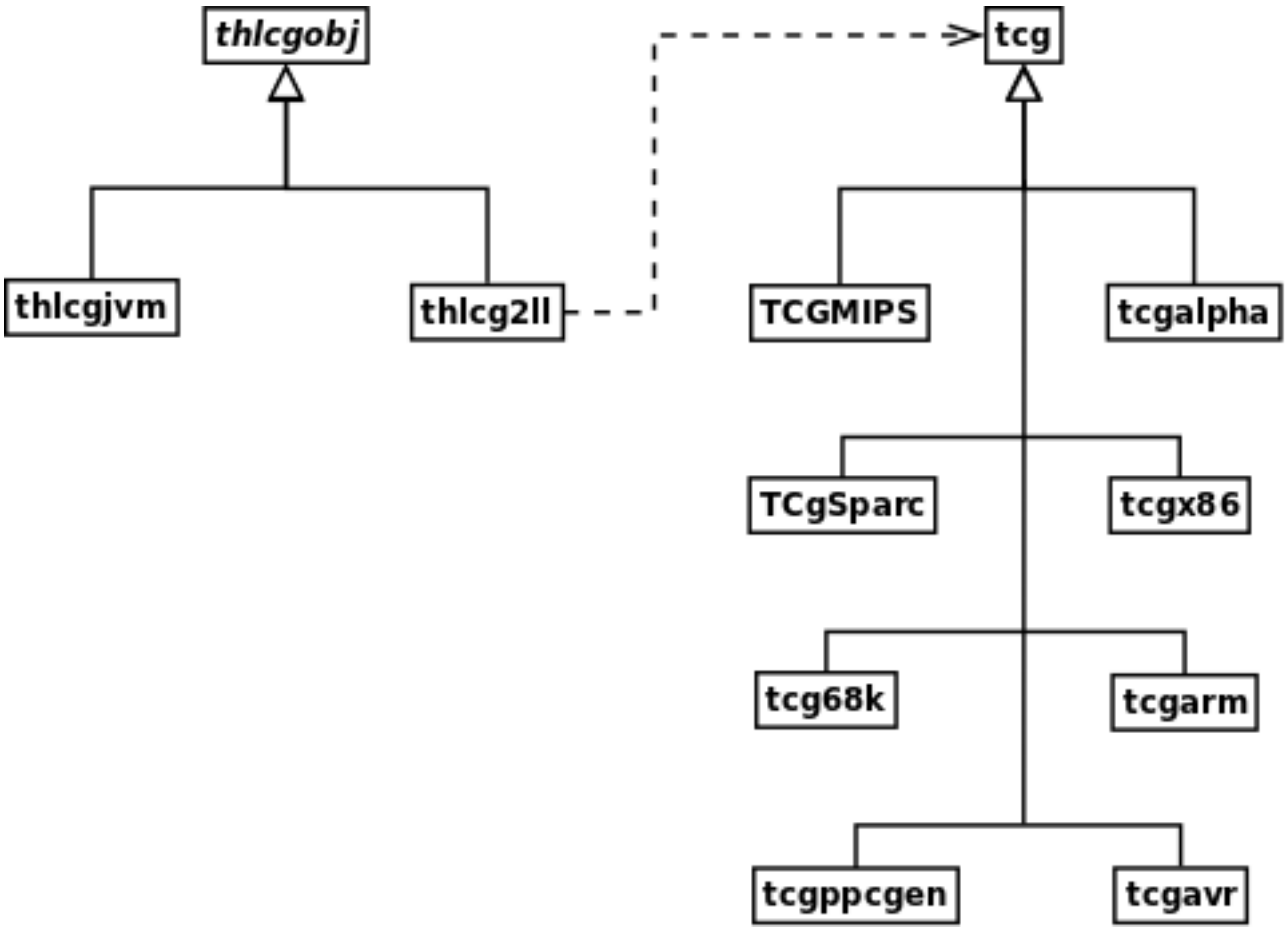

Рис. 2: Диаграмма классов генератора кода

4. Функциональные требования

Компилятор должен:

1. Позволять объявлять анонимные подпрограммы внутри тела других подпрограмм. В том числе с любым уровнем вложенности. Синтаксис объявления анонимной подпрограммы такой же как и для обычной подпрограммы, с двумя исключениями:
   - после ключевых слов procedure и function нет идентификатора, а сразу идет опциональный список формальных аргументов;
   - после списка формальных аргументов (если он опущен, то после ключевых слов procedure или function) не ставится точка с запятой.

   Пример 1:

```
procedure begin
```

```
  Writeln('inside');
end;
```

Пример 2:

```
procedure begin
  Writeln('inside');
end;
```

2. Позволять анонимным функциям возвращать значения, используя ключевое слово Result:

   Пример:

   ```
   function(num: Integer): Integer begin
     Result := num + 10;
   end;
   ```

3. Запрещать анонимным процедурам использовать переменную Result объемлющей функции:

   Пример:

   ```
   function Calculate: Integer;
   begin
     ...
     procedure begin
       Result := 10; // Error, result can't be captured
     end;
   end;
   ```

4. Запрещать анонимным процедурам использовать параметры объемлющей функции, объявленные с модификатором var или out.

5. Запрещать анонимным процедурам использовать переменные, объявленных в обработчике исключений.

6. Позволять анонимным процедурам использовать переменные, объявленные в объемлющей функции:

   Пример:

   ```
   procedure Calculate: Integer;
   var num: Integer;
   begin
     ...
     procedure begin
       num := 10;
     end;
   ```

```
end;
```

7. Позволять анонимным процедурам использовать переменные, объявленные в объемлющей функции, при уровне вложенности больше 1: Пример:

```
procedure Outer;
var num: Integer;
  procedure Calculate;
  begin
    ...
    procedure begin
      num := 10;
    end;
  end;
begin
 ...
```

8. Продлевать время жизни захваченных переменных. Время жизни захваченной переменной определяется не временем работы функции, где она объявлена, а временем жизни всех ссылающихся на неё замыканий.

9. Управлять памятью замыканий автоматически, используя счётчик ссылок. Время жизни замыкания заканчивается только, когда не остаётся указателей соответствующего типа, ссылающихся на это замыкание.

10. Осуществлять захват переменных по ссылке. Замыкания, созданные в одном лексическом контексте должны иметь доступ к одним и тем же переменным.

11. Корректно осуществлять проверку типов в момент присваивания значений переменным-указателям на замыкание. Это значит, что:

    - Переменной можно присваивать определение анонимной функции, если сигнатура анонимной функции совпадает с сигнатурой, указанной в определении типа переменной. Другими словами, в этом случае используется структурная эквивалентность выражений, т.к. определение анонимной функции не содержит имя типа.

- Переменной можно присваивать значение другой переменной, только если имена типов этих переменных совпадают. Т.е. в этом случае используется именная эквивалентность выражений.

12. Корректно осуществлять проверку типов, если замыкание определено в качестве фактического параметра. В этом случае так же используется структурная эквивалентность выражений, т.к. определение анонимной функции не содержит имя типа.

13. Предоставлять для вызова замыкания синтаксис аналогичный синтаксису вызова обычных указателей на функцию:

    - Если замыкание имеет непустой список аргументов, то для вызова в имени переменной-указателя в круглых скобках добавляется список фактических параметров, разделённых запятыми.

    - Если замыкание имеет пустой список параметров, то в выражении, состоящем только из вызова замыкания круглые скобки не содержащие аргументов можно опустить.

14. Корректно осуществлять проверку типов во время вызова замыкания. Это значит, что:

    - Количество фактических параметров должно равняться количеству формальных параметров.

    - Типы фактических параметров должны быть эквивалентны типам формальных параметров.

5. Требования к интерфейсу

Компилятор имеет интерфейс командной строки, который описан в руководстве пользователя FPC[20]. Интерфейс командной строки остаётся неизменным.

Введены новые сообщения об ошибках:

1. Замыкание не может использовать переменную Result объемлющей функции.

2. Замыкание не может использовать параметр <имя параметра>, объясленный с модификатором var.

3. Замыкание не может использовать параметр <имя параметра>, объясленный с модификатором out.

4. Замыкание не может использовать переменную <имя переменной>, объясленную в обработчике исключений.

5. Замыкание не может использовать переменную <имя переменной>, объясленную в обработчике исключений.

Т.к. определение замыкания не содержит имя типа, то существующие сообщения об ошибках вместо имени типа должны выводить его сигнатуру. Перед сигнатурой функции выводятся ключевые слова “reference to”, позволяющие пользователю различить указатель на замыкание от указателя на обычную функцию. К примеру, для кода приведённого ниже, необходимо вывести сообщение "Ожидается выражение типа TProc, получено выражение типа reference to procedure(Integer;String)".

```
type TProc = reference to procedure;
var p: TProc;
begin
  p := procedure(i: Integer; b: String) begin end;
end;
```

6. Проект

6.1. Структуры данных

6.1.1. Базовые структуры данных

Ключевые структуры данных: определения типов, таблицы символов и синтаксическое дерево. Программа на языке программирования Free Pascal имеют рекурсивную структуру, поэтому упомянутые структуры данных имеют сложные циклические зависимости. Пример такой зависимости: узел синтаксического дерева, представляющий обращение к переменной содержит имя переменной. По имени переменной доступен её тип. Типом пере-

менной может быть класс, который содержит метод. Реализация метода может содержать обращение к переменной.

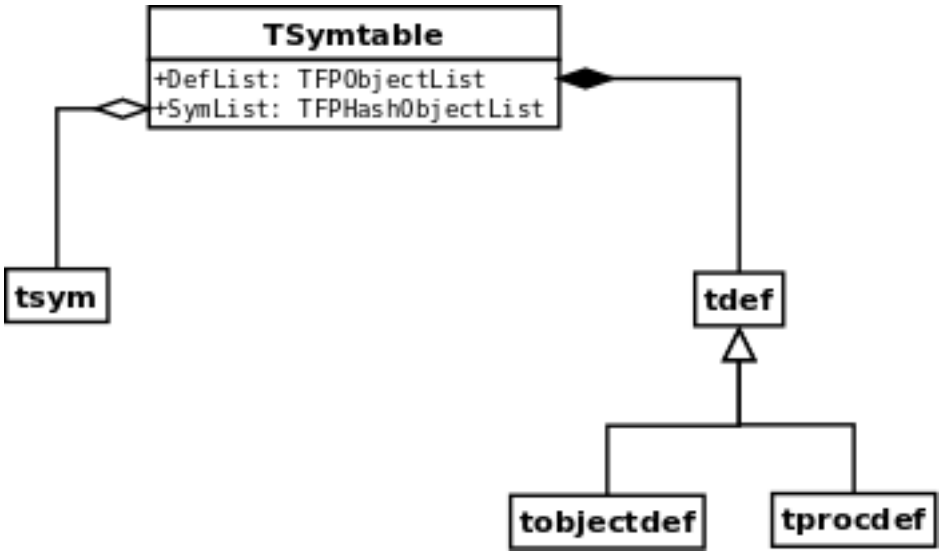

Рис. 3: Диаграмма классов

Таблицы символов отдельно хранят определения типов и символьные имена. Существуют различные таблицы символов: таблица символов модуля, локальная таблица символов, таблица символов с параметрами процедуры и др. Все таблицы символов унаследованы от общего базового класса TSymTable, определяющего общий интерфейс для работы с ними. Любая таблица символов отдельно хранит определения типов и литеральные имена, что отражено на рис. 3. Таким образом, таблицы символов содержит доступные в текущем лексическом контексте именованные объекты и соответствующие им типы.

На рис. 3 показаны 2 конкретных класса, унаследованных от абстрактного класса tdef. Первый из них, tobjectdef, хранит информацию о классе: его родитель, реализованные интерфейсы, список методов, список полей и др. Второй, tprocdef, хранит информацию о типе процедуры: список аргументов, тип возвращаемого значения, локальные переменные. Отметим, что tprocdef и tobjectdef содержат список именованных объектов, каждый из которых, тоже имеет тип. Это отражено на рис. 4, определения типов объекта и процедуры содержат таблицы символов. Причем, они используют разные конкретные реализации.

Литеральные имена, используемые для доступа к типам, называются символами. На рис. 5 изображена диаграмма классов для нескольких

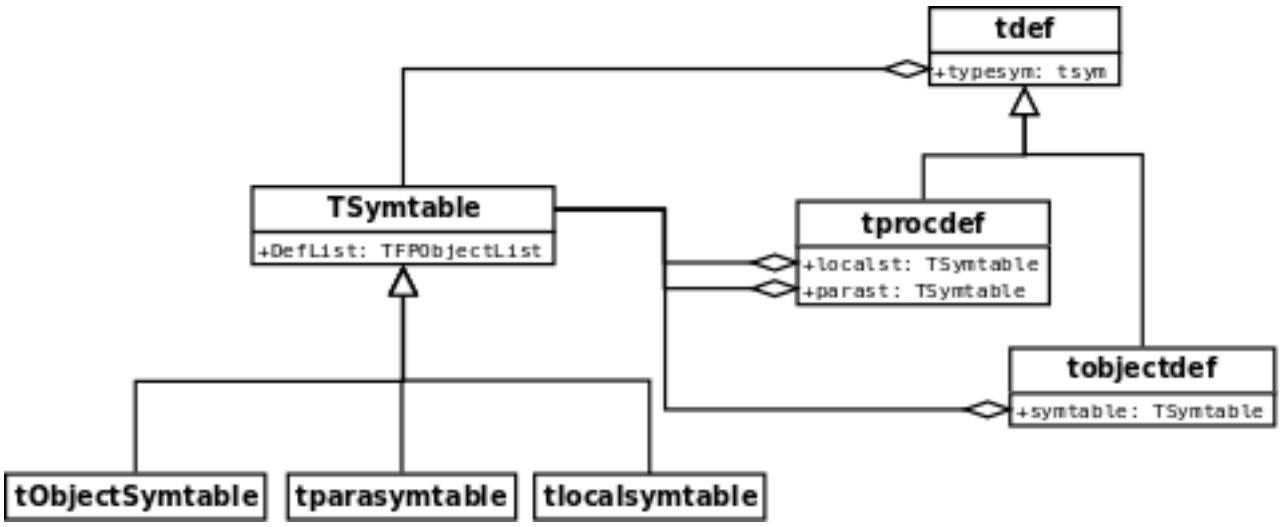

Рис. 4: Диаграмма классов

символов и их связь с определениями типов. Каждый символ унаследован от общего класса tsym. Конкретные реализации рассчитаны для работы с конкретными определениями типов. Из-за того, что Free Pascal поддерживает перегрузку операторов, может существовать несколько подпрограмм с одним именем и разными определениями. Поэтому tprocsym хранит список ссылок на определения типов подпрограмм. В это же время ttypesym содержит только одну ссылку на определение типа.

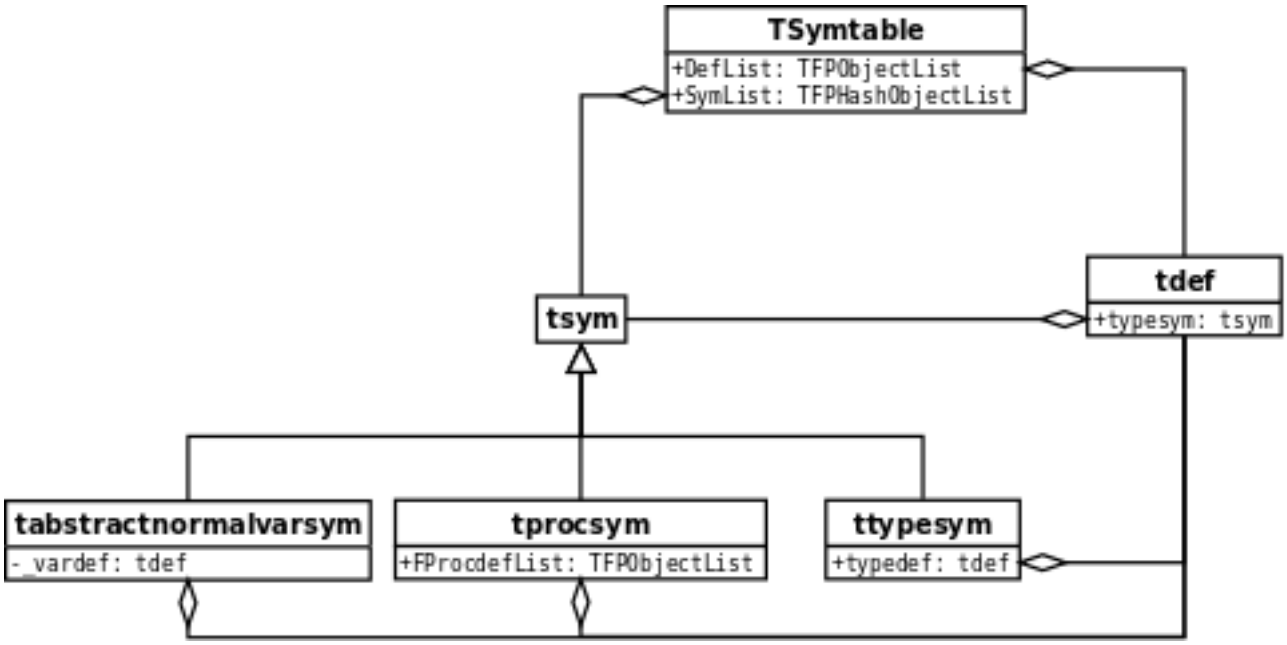

Рис. 5: Диаграмма классов

Теперь рассмотрим связь синтаксического дерева и таблицы символов. Если таблицы символов отражают информацию о именованных данных, то

синтаксическое дерево отражает информацию о действиях. Обращение к переменной является примером действия с именованными данными. Синтаксический узел, отражающий обращение к переменной - tloadnode. На рис. 6 показано, как tloadnode хранит информацию о переменной.

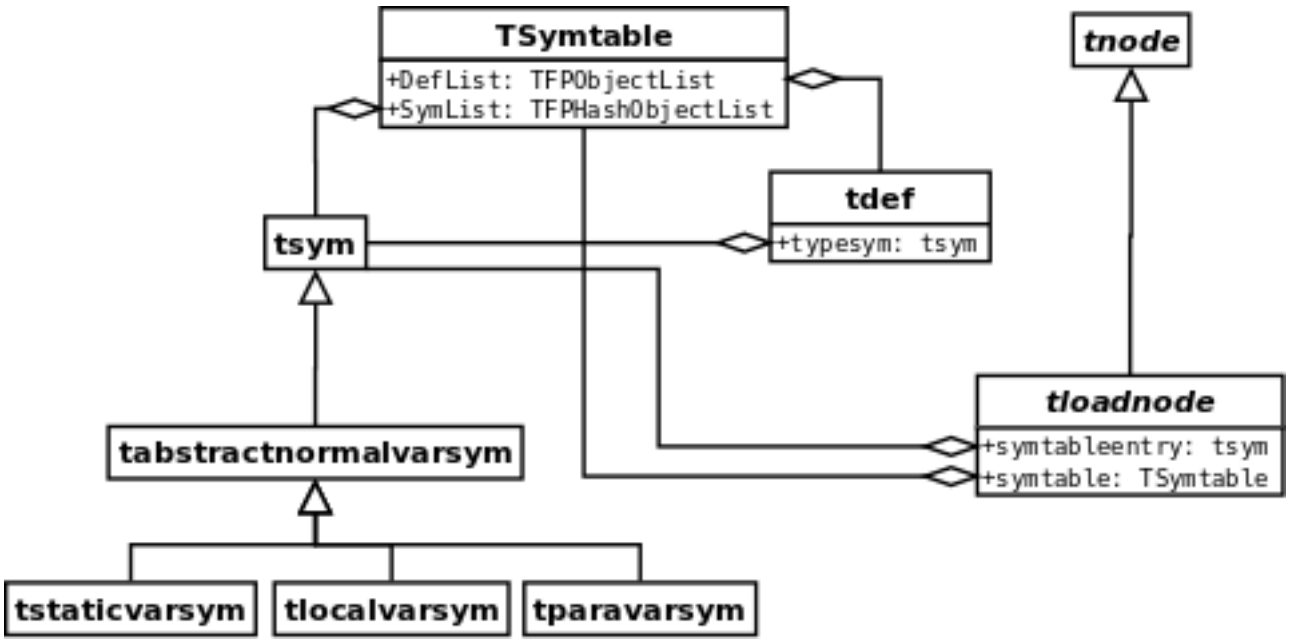

Рис. 6: Диаграмма классов

Кроме ссылок из узлов синтаксического дерева на именованные объекты теоретически возможна обратная ситуация. Подпрограммы имеют реализацию, которую компилятор обрабатывает в виде синтаксического дерева. К счастью, определение типа подпрограммы не содержит ссылок на тело подпрограммы. Связь реализации и информации о типе подпрограммы показана на рис. 7

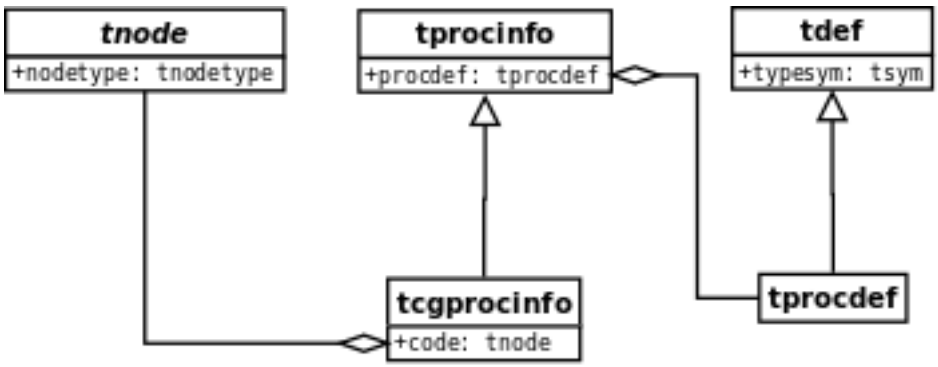

Рис. 7: Диаграмма классов

6.1.2. Стек с таблицами символов

Язык Pascal использует статическую область видимости определений. Это значит, что компилятор способен узнать область видимости во время компиляции.

Программы и модули на языке Pascal состоят из блоков. Блок включает в себя объявления меток, констант, типов, переменных и подпрограмм. Подпрограмма, кроме тела, также может содержать блок. Область видимости объявлений сделанных в блоке начинается с места их объявления до конца блока[21]. Определения классов и модулей также вносят дополнительные правила, определяющие области видимости, тем не менее имеющие общие черты с описанными правилами для блоков.

Переменные, объявленные рядом, как правило, имеют общую область видимости. Это позволяет эффективно реализовать области видимости путём настройки отдельной таблицы символов для каждой области видимости[1]. Кроме того, область видимости объявлений, сделанных во вложенных конструкциях, всегда меньше области видимости объявлений, сделанных во внешних конструкциях. К примеру, область видимости переменной, объявленной во вложенной функции всегда меньше области видимости переменной объемлющей функции. Это позволяет использовать стек для эффективного управления таблицами символов.

FPC использует стек с таблицами символов на этапах синтаксического и семантического анализа, а так же во время проведения трансформаций синтаксического дерева. Таблицы символов, находящиеся на стеке содержат определения, доступные для использования в текущий момент. Стоит отметить, что без поддержки замыканий время жизни локальных переменных и их область видимости – неразделимые понятия. Именно этот факт позволяет организовать хранилища локальных переменных при помощи стека. Замыкания изменяют время жизни захваченных локальных переменных, поэтому стек для их хранения не подходит. Так как замыкания не меняют область видимости локальных переменных, стек таблиц символов по-прежнему остаётся основным инструментом для реализации областей видимости в компиляторе FPC.

6.2. Модули и алгоритмы

6.2.1. Разбор анонимных функций

В язык вводится новый тип данных – указатель на замыкание. Определение указателя на замыкание отличается от определения указателя на обычную функцию ключевыми словами reference to. Для разбора определений нового типа данных были модифицированы лексический и синтаксический анализаторы. Для синтаксического разбора сигнатуры функции замыкания используется уже существующие подпрограммы, разбирающие объявления указателей на обычные подпрограммы. Внутреннее представление типа переменной-указателя на замыкания содержит полученную сигнатуру функции. Структура этого представления будет описана ниже.

В язык вводится новый тип выражений – анонимная подпрограмма. Определение анонимной подпрограммы начинается с ключевого слова procedure или function. Раньше эти ключевые слова использовались только в определении подпрограмм и описании типов. Поэтому в выражениях нельзя было использовать ключевые слова procedure и function. Этот факт упростил модификацию синтаксического анализатора – теперь выражения, начинающиеся с указанных ключевых слов, считаются определением анонимной подпрограммы. Существующая в синтаксическом анализаторе процедура разбора объявлений подпрограмм была модифицирована для разбора анонимных подпрограмм. Анонимная подпрограмма, в отличие от именованной, не содержит идентификатор и после определения её сигнатуры не ставится точка с запятой. Кроме того изменения внесены с учётом того, что анонимная подпрограмма не является вложенной функцией. Дело в том, что вложенная функция для доступа к локальным переменным объемлющих функций использует указатель на стековый фрейм объемлющей функции. Как уже не раз было отмечено, захваченные переменные не располагаются на стеке, поэтому такой же механизм для доступа к захваченным переменным не подходит.

6.2.2. Управление динамической памятью

Самая большая сложность реализации замыканий заключается в управлению памятью. Захваченные переменные не могут располагаться на стеке т.к. срок жизни переменных на стеке ограничен временем работы объемлющей процедуры. Захваченные переменные необходимо располагать в куче. Память, однажды выделенную под захваченные переменные позднее необходимо освобождать. С определением момента, когда можно освобождать память замыкание есть несколько сложностей. Во-первых захват переменных по ссылке делает возможной ситуацию, когда несколько замыканий ссылаются на одни и те же переменные. Поэтому общие данные можно освобождать только когда все замыкания, ссылающиеся на них, становятся недоступными для вызова. Во-вторых, замыкание может быть определено в списке фактических аргументов другой функции. В случае ручного управления памятью пользователям языка программирования необходимо будет устанавливать договорённости о том вызывающая или вызываемая сторона освобождает память. Это сделает использование замыканий источником сложностей и ошибок.

Чтобы избежать ошибок, замыкание необходимо сделать управляемым типом данных, т.е. типом данных с автоматическим управлением динамической памятью. К счастью, FPC уже содержит сборщик мусора, использующий счётчик ссылок, который следит за объектами управляемых типов данных. Управляемыми типами является большинство реализаций строк, динамические массивы, интерфейсы со счётчиком ссылок, Variant и составные типы данных, содержащие или унаследованные от управляемых типов данных.

Вместо того, чтобы реализовывать поддержку нового управляемого типа данных, для реализации замыканий можно воспользоваться существующими интерфейсами со счётчиком ссылок. Такой подход существенно снижает сложность реализации. Поддержка нового управляемого типа данных требует внесения большого количества изменений в генератор кода и библиотеку времени исполнения. Реализация, использующая трансформацию новых синтаксических конструкций в существующие, помогает уменьшить размер генератора кода и библиотеки времени исполнения. Это

упрощает поддержку нескольких целевых платформ, что особенно актуально для FPC.

6.2.3. Интерфейсы с подсчётом ссылок

Интерфейс – это именованный набор методов. В языке Free Pascal интерфейсы это аналог множественного наследования, реализованного, к примеру, в языке c++. Интерфейс может быть унаследован от одного или нескольких других интерфейсов. Класс может реализовывать один или более интерфейсов. Класс, реализующий интерфейс предоставляет все методы, объявленные в определении интерфейса[21].

В языке Free Pascal существует специальный интерфейс IUnknown. Он определяет набор функций, необходимых сборщику мусора для управления памятью объектов, поэтому его называют интерфейсом со счётчиком ссылок. IUnknown и все интерфейсы, унаследованные от него, являются управляемыми типами данных. При помощи интерфейса со счётчиком ссылок пользователь может создавать объекты, памятью которых будет автоматически управлять сборщик мусора. Для этого необходимо объявить класс, реализующий интерфейс с подсчётом ссылок. Затем работать с созданными экземплярами этого класса через переменные, имеющими тип интерфейса с подсчётом ссылок.

6.2.4. Преобразование замыканий

Основная идея реализации, выполненной в рамках данной работы, состоит в том, чтобы преобразовать исходную программу с замыканиями в эквивалентную программу без замыканий.

Замыкание – это подпрограмма вместе с захваченными переменными. Иными словами, это данные и подпрограмма, работающая с этими данными. Такое определение совпадает с определением объекта, содержащего один метод. Действительно, можно представить замыкание в виде объекта содержащего захваченные переменные, либо ссылки на них. Этот объект будет иметь один метод, а так как захваченные переменные, или ссылки на них являются полями этого объекта, метод сможет использовать захваченное окружение.

В языке Pascal нельзя объявлять локальные переменные в теле подпрограммы. Поэтому все замыкания, созданные за время работы подпрограммы, будут ссылаться на одни и те же переменные. Следовательно, можно перенести захваченные локальные переменные этой подпрограммы в один объект, а анонимные подпрограммы, объявленные в её теле сделать методами этого объекта. Основное преимущество такого похода: простота. Все захваченные переменные переносятся в один объект, который создаётся в начале работы функции, а за его освобождение отвечает сборщик мусора. Недостатки проявляются, когда в теле подпрограммы объявлено несколько замыканий, использующих разные переменные. Все замыкания используют один и тот же объект, хранящий все захваченные ими переменные объемлющей подпрограммы. Поэтому захваченные переменные будут находиться в памяти до тех пор, пока доступно хотя бы одно из созданных замыканий.

Альтернативной стратегией является отдельный подсчёт ссылок на каждую переменную. К сожалению, использование интерфейсов со счётчиками ссылок накладывает дополнительные расходы времени исполнения на операцию присваивания. Кроме того, счётчики ссылок для каждой переменной занимают дополнительную память.

Все переменные, захваченные только одним замыканием, заканчивают свою жизнь единовременно. Поэтому предпочтительнее группировать захваченные переменные в составные объекты, используя информацию о том, какие замыкания захватывают какие переменные. Такая реализация потребует проведения дополнительно анализа структуры программы и разработки алгоритма для группировки объектов. Это является возможной оптимизацией, но выходит за рамки данной работы.

Таким образом, выбран и осуществлен следующий алгоритм преобразования замыканий.

1. Каждой подпрограмме, содержащей замыкания, или захваченные переменные ставится в соответствие специальный объект. Назовём его "хранилище".

2. Хранилище создаётся во время активации функции, с которой оно ассоциировано.

3. Хранилище реализует интерфейс с подсчётом ссылок, поэтому его памятью управляет сборщик мусора. Функция, в которой создано хранилище, содержит локальную переменную, ссылающуюся на хранилище. Поэтому оно доступно в течении всего времени работы функции.

4. Захваченные локальные переменные становятся полями объекта - хранилища. Из тела функции, где объявлен этот объект, захваченные переменные доступны через локальную переменную, указывающую на хранилище.

5. Анонимная подпрограмма становятся методами объекта - хранилища объемлющей функции.

6. Захваченные переменные подпрограммы, где объявлена анонимная функции стали полями объекта - хранилища. Теперь они доступны для его методов, которые раньше били анонимными подпрограммами.

7. Возможна ситуация, когда замыкание захватывает локальную переменную объемлющей функций при уровне вложенности больше одного. В этом случае используется специальная ссылка на хранилище объемлющей функции.

6.3. Стандарт кодирования

Стандарт кодирования описан в документации проекта[14]:

- Ключевые слова пишутся маленькими буквами.
- Символы табуляции запрещены.
- Не следует отделять операции, запятые точку с запятой и скобки пробелами.
- Размер отступов: два пробела для каждого уровня отступа.
- Перед блоком begin ... end следует делать отступы.
- Не вложенные подпрограммы разделяют двумя пустыми строчками.

- Вложенные подпрограммы отделяются одним пробелом.
- Однострочные условные операторы запрещены. Условие и действие нужно писать на разных строчках.
- У идущих подряд условных операторов следует писать else и последующее if на одной строке.
- Идентификаторы подпрограмм, состоящие из нескольких слов необходимо писать маленькими буквами, разделяя слова символом подчёркивания.
- Идентификаторы переменных и типов, состоящие из нескольких слов следует писать маленькими буквами слитно без разделительных символов.

7. Реализация и тестирование

Для компилятора FPC создана реализация замыканий, удовлетворяющая всем требованиям, описанным в данной работе. Изменения состоят из 35ти комитов, содержащих 1334 добавлений и 521 удалений строк кода. Так же добавлено 38 тестов общим объёмом 1464 строки. Изменения получили положительные отзывы среди разработчиков, ведётся работа по интеграции в основную ветку кода[27].

Заключение

Таким образом, в процессе выполнения дипломной работы мною были углублены знания о компиляторах современных языков программирования, улучшены навыки поддержки и развития больших программных проектов. Я получил опыт участия в международном проекте с открытым исходным кодом и пообщался с опытными разработчиками. Мною была рассмотрены различные подходы к реализации замыканий и разработано решение для конкретного компилятора FPC.

Итогом работы стала реализация, удовлетворяющая всем требованиям, рассмотренным в данной работе.

Список литературы

[1] Альферд В. Ахо, Моника С. Лам, Рави Сети, Джеффри Д. Ульман. Компиляторы. Принципы, технологии и инструментарий. - 2 изд. - Вильямс, 2008. - 1184 с.

[2] Баль Н.В., Кленин А.С. Курсовая работа «Анализ потоков управления для языка программирования Pascal». - ДВГУ, 2008г.

[3] Гамма Э., Хелм Р., Джонсон Р., Влиссидес Д. Приемы объектно-ориентированного проектирования. Паттерны проектирования. - Спб.:Питер, 2010г.- 386 с.

[4] Денисенко М. В., Кленин А. С. Курсовая работа «Доработка компилятора Free Pascal: case of string». - ДВГУ, 2009г.

[5] Документация Delphi: Anonymous methods http://docwiki.embarcadero.com/RADStudio/XE3/en/Anonymous_Methods_in_Delphi

[6] Документация Delphi: System.Rtti.IsManaged http://docwiki.embarcadero.com/Libraries/XE4/en/System.Rtti.IsManaged

[7] Документация Delphi. Список изменений. http://docwiki.embarcadero.com/RADStudio/XE4/en/What%27s_New

[8] Крис Касперски, Техника оптимизации программ. Эффективное использование памяти. - БХВ-Петербург, 2003г. - 464с.

[9] Лукащук М.А., Кленин А.С. Курсовая работа «Оператор for-in для компилятора Free Pascal». - ДВГУ, 2010г.

[10] Нелепа А.А., Кленин А.С. Дипломная работа «Расширение компилятора Free Pascal для поддержки обобщённого программирования». - ДВГУ, 2007г.

[11] Andrew W. Appel. Modern Compiler Implementation in Java. - 2nd edition - Cambridge University Press, 2004. - 512с.

[12] Apache™ Subversion®. Home page. http://subversion.apache.org/

[13] Free Pascal Compiler http://www.freepascal.org/

[14] Free Pascal Documentation. Coding style http://wiki.freepascal.org/Coding_style

[15] Freepascal Wiki: Platform list http://wiki.freepascal.org/Platform_list

[16] Free Software Foundation, Inc. GNU General Public License. http://www.gnu.org/licenses/gpl.html. - 2007г.

[17] Lazarus http://www.lazarus.freepascal.org/

[18] Mantis bug tracker. Home page. urlhttp://www.mantisbt.org/

[19] Martin Odersky and others. An Overview of the Scala Programming Language. - 2nd edition - Ecole Polytechnique Federale de Lausenne, 2001г. - 20с.

[20] Michaël Van Canneyt, Florian Klämpfl. Free Pascal : User's Guide. 2013г. http://www.freepascal.org/docs-html/user/user.html

[21] Michaël Van Canneyt. Free Pascal : Reference guide. 2013г. http://www.freepascal.org/docs-html/ref/ref.html

[22] Miguel Garcia. Code walkthrough of the UnCurry phase (Scala 2.8). Hamburg University of Technology, 2009г. - 5с.

[23] Keith D. Cooper, Linda Torczon. Engineering a Compiler. - 2nd edition - Morgan Kaufmann Publishers, 2012г. - 801с.

[24] Raul Rojas. A Tutorial Introduction to the Lambda Calculus. - FU Berlin, 1998г. - 9с.

[25] Standard for Programming Language C++ http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2012/n3485.pdf

[26] Yasuhiko M., Greg M., Robert H. Typed Closure Conversion. - Carnegie Mellon University, 1995г. - 40с.

Обсуждения в системе контроля изменений fpc:

[27] freepascal bugtracker. Issue#0024481: Implement closures http://bugs.freepascal.org/view.php?id=24481

Обсуждения в списке рассылок разработчиков fpc (fpc-devel):

[28] Sven's comment - http://lists.freepascal.org/lists/fpc-devel/2013-March/031595.html

[29] Marko's comment - http://lists.freepascal.org/lists/fpc-devel/2013-March/031657.html

Автор работы _____ (Ф.И.О.)
(подпись)

Квалификационная работа допущена к защите

Назначен рецензент

_____
(Фамилия, И.О. рецензента, ученая степень, ученое звание)

Зав. кафедрой информатики,
математического и компьютерного моделирования А. Ю. Чеботарев

Дата «____»_____ 2013 г.