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

ФАКУЛЬТЕТ **Информатики и систем управления**
КАФЕДРА **Теоретической информатики и компьютерных технологий**

КУРСОВАЯ РАБОТА
НА ТЕМУ:

Пошаговый отладчик для компилятора Простого Рефала

Студент **ИУ9-72** _____ _____
(Группа) (Подпись, дата) (И.О.Фамилия)

Руководитель курсового проекта _____ _____
(Подпись, дата) (И.О.Фамилия)

Москва
2016 г.

Оглавление

Введение

Простой Рефал – один из диалектов РЕФАЛа, языка функционального программирования, ориентированный на символьные вычисления, обработку и преобразование текстов. Простой Рефал ориентирован на компиляцию в исходный текст на C++ и обладает следующими особенностями:

- поддержка только подмножества Базисного Рефала (предложения имеют вид образец = результат), отсутствие более продвинутых возможностей (условия, откаты, действия);
- поддержка вложенных функций;
- простая схема кодогенерации;
- самоприменимость компилятора;
- в основе лежит классическая списковая реализация.

Простой Рефал создавался как исследовательский проект компиляции кода на РЕФАЛе в имеративный код (язык C++). Для чего были поставлена цель написания минимального алгоритмически полного компилятора диалекта Базисного РЕФАЛа. При этом простота транслятора была важнее удобства программирования на языке. Однако в процессе развития Простого Рефала компилятор отошел от своей первоначальной простоты: в связи с новыми приложениями языка — использование в качестве back-end`а для Модульного Рефала, исследование реализации вложенных функций — язык расширял свою функциональность — были добавлены такие возможности, как идентификаторы, абстрактные типы данных и статические ящики, вложенные функции.[1] При этом целостность и согласованность языка не пострадали. Однако каким бы продуманным ни был язык программирования, невозможно избавить пользователей языка от совершения ошибок при написании программ.

Однако, существующие средства компилятора Простого Рефала для помощи программисту при отладке очень скудны. Во-первых, это посмертный отладчик — выдача аварийного дампа при аварийном останове выполнения программы. Во-вторых, ключами компиляции можно указать выдачу дампа поля зрения на каждом шаге ре-

фал-машины начиная с указанного шага выполнения, также ввести ограничение по объему используемой памяти и предельному числу шагов, при превышении которых программа останавливается с выдачей дампа.[1] И хотя компилятор Простого Рефала порождает код на C++, пользоваться интерактивными отладчиками C++ для отладки кода на Рефале крайне неудобно, поскольку они работают на более низком уровне абстракции. Однако, интерактивные отладчики C++ неплохи для отладки библиотеки времени выполнения и функций, тело которых пишется на C++.

В настоящей работе проводится разработка и реализация пошагового отладчика для компилятора Простого Рефала. Интерфейс разрабатываемого отладчика включает в себя весьма распространенный набор команд, свойственный для любого отладчика, с теми особенностями, которые вносит специфика входного языка.

Основными требованиями к реализации пошагового отладчика — переносимость и отключаемость, позволяющая запустить программу без отладочных алгоритмов, увеличивающих затраты на выполнение.

1. Архитектура компилятора Простого Рефала

1.1. Внутренние абстракции программы

1.1.1. Абстракции выполнения программ на Рефале

Рефал-машина — это некоторая абстракция, позволяющая описывать процесс выполнения любой программы, написанной на Рефале. Рефал-машина работает в пошаговом режиме, при этом единицей обработки является *определенное выражение*, то есть выражение, содержащее скобки конкретизации, но не содержащее переменные. При этом обрабатываемое определенное выражение называется *полем зрением* рефал-машины.

За один шаг рефал-машина находит в поле зрения *первичное активное подвыражение* — самую левую пару скобок конкретизации, не содержащую внутри себя других скобок конкретизации, — вызывает замыкание, следующее за открывающей скобкой, с выражением между этим замыканием и закрывающей скобкой в качестве аргумента. Затем ведущая пара скобок заменяется на определённое выражение, являющееся результатом выполнения замыкания, и рефал-машина переходит к следующему шагу. Выполнение рефал-машины продолжается до тех пор, пока поле зрения будет содержать скобки конкретизации.

Особенность выполнения предложения на Простом Рефале

Для удобства отладки программ на Простом Рефале выполнение функции на абстрактной рефал-машине разбивается на три стадии:

1. **Сопоставление с образцом.** На данном этапе содержимое терма активации, которое состоит из угловых скобок, имени функции и ее аргумента, не изменяется, чтобы в случае неудачи сопоставления следующее предложение получило аргумент в том же виде, а если предложение последнее, то чтобы по дампу поля зрения можно было понять, в каком случае процесс выполнения завершился с ошибкой.

2. **Распределение памяти для новых узлов.** На данном этапе начало списка свободных блоков инициализируется новыми значениями — копии переменных, новые узлы-литералы: атомы, скобки; содержимое же терма активации так же, как и на стадии сопоставления с образцом, не изменяется из соображений отладки.
3. **Построение результата.** Поскольку построение результат осуществляется только путём изменения указателей двусвязного списка, эта стадия не может завершиться неуспешно; части терма активации, которые не понадобились в результате, переносятся в список свободных блоков; данный этап всегда завершается инструкцией

```
return refalrts::cSuccess;
```

1.1.2. Структуры данных для абстракций выполнения

Поле зрения внутри программы представляется в виде двусвязного списка. Узлы этого списка представляют собой структуры, содержащие тег типа (tag) и поле данных (info). В зависимости от типа такой узел может представлять собой атом, одну из структурных скобок или одну из скобок конкретизации:

- Узлы-атомы — это числа, идентификаторы, символы, замыкания без контекста; в качестве данных узлы-атомы содержат само значение атома.
- Узлы, представляющие собой структурную скобку, в поле info содержат ссылку на соответствующие парные скобки (см. Рисунок 1); такой прием обеспечивает распознавание скобок в образце за константное время:

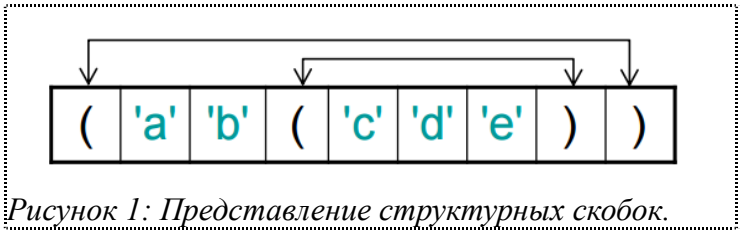

Рисунок 1: Представление структурных скобок.

- Узлы открывающих угловых скобок содержат ссылки на узлы соответствующих закрывающих скобок. Узлы же с закрывающими угловыми скобками указывают на узлы с открывающими угловыми скобками, которые станут лидирующими после выполнения текущей пары скобок конкретизации. Таким образом, угловые скобки образуют стек вызовов функций (см. Рисунок 2).

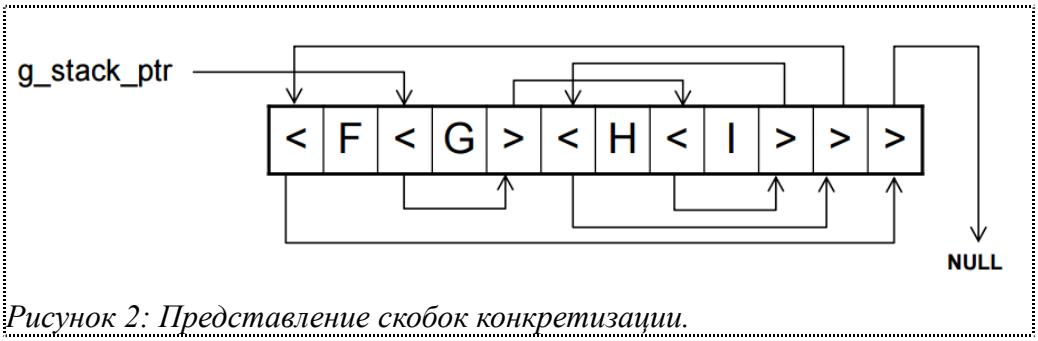

Рисунок 2: Представление скобок конкретизации.

Для ускорения операций создания новых узлов, а также для предотвращения утечек памяти, используется список свободных узлов.

1.2. Интерфейсы компилятора

Как говорилось во **Введении**, простота транслятора выходной язык была одной из базовых целью при создании и развитии Простого Рефала. На момент выполнения работы компиляция программы на Простом Рефале в программу на C++ выполняется в восемь проходов ([2] - описание проходов):

1. **Загрузка исходного текста программы.** Происходит открытие и считывание исходного текста программы в виде потока char'ов; всегда завершается символом перевода строки, даже если в исходном коде его не было.
2. **Лексический анализ.** Преобразования потока char'ов в последовательность токенов; каждый токен содержит тип лексемы, номер строки, на которой в исходном коде расположена лексема, и атрибутов лексемы, зависящих от ее типа.
3. **Синтаксический анализ.** Построение абстрактного синтаксического дерева программы с генерацией списка контекстно-независимых ошибок.
4. **Проверка контекстных зависимостей.** Пополняется список ошибок контекстно-зависимыми ошибками; дальнейший разбор программы выполняется при отсутствии ошибок на выходе этого шага.
5. **Редуктор до подмножества.** Построение редуцированного абстрактного синтаксического дерева путем избавления от синтаксического сахара.
6. **Генерация высокоуровнего RASL`а.** Перестроение синтаксического дерева, путем замены в функциях предложений на последовательности команд

обработки данных — резервирование стека для локальных переменных, сопоставление с образцом, циклы по открытым переменным.

7. **Генерация низкоуровнего RASL`a.** Генерация на основе преобразованного синтаксического дерева последовательности элементарных команд, каждая из которых в дальнейшем отображается в код на C++.
8. **Генерация целевого кода на C++.** Преобразование каждой из команд промежуточного кода во фрагмент кода на C++, получение выходного кода в виде последовательности строк целевого файла.

1.3. Режимы компиляции программы: прямая кодогенерация и интерпретация

В коде на C++ могут быть как явно прописаны элементарные операции последних двух стадий в виде вызовов соответствующих функций — режим прямой кодогенерации, — так и последовательность интерпретируемых команд в виде константного статического массива, который затем передаётся специальной функции- интерпретатору — режим интерпретации.

По умолчанию, производится компиляция в обоих режимах, а в процессе выполнения выбор режима осуществляется директивами условной компиляции препроцессора C++.

Рассмотрим подробнее режим интерпретации, поскольку именно в этом режиме предполагается работа разрабатываемого отладчика.

Как уже было сказано выше, в режиме интерпретации каждая функция представляет собой массив интерпретируемых команд — структур с тегом команды и тремя однобайтовыми аргументами, смысл которых зависит от команды. При этом возможны случаи команд без аргументов (или с количеством меньшим, чем три), тогда некоторые из трёх последних байтов не несут никакой полезной информации. Разбор команд производится в бесконечном цикле выполнения программы.

2. Подход к реализации отладчика

Вопрос выбора подхода к реализации пошагового отладчика можно разделить на реализацию интерфейса отладчика и реализацию логику отладчика.

Интерфейс может быть, как консольным, так и графическим.

- **Консольный интерфейс** (в стиле GDB). Используется реализациями Рефал-2, Рефал-5 и Рефал-6.
- **Графический интерфейс**. Насколько известно, таких отладчиков для Рефала пока не существует.

При выборе реализации логики можно рассмотреть следующие варианты:

- Компоновка со специальным рантаймом (как в Рефале-2).
- Компиляция в интерпретируемый код и написание интерпретатора, выполняющего этот код по шагам, как отладка в Рефал-5 и в Рефал-6, поскольку они компилируются в интерпретируемый код, затем выполняются интерпретатором.
- Компиляция в интерпретируемый код, и представляет собой отличную базу для развития отладочных средств.
- Компиляция в специальную динамическую библиотеку (.dll, .so), загружаемую интерактивным отладчиком. Можно добавлять в исполнимые (.exe) файлы экспортируемые функции и загружать эти файлы как dll. В рамках этого варианта стоит подробнее рассмотреть вопрос переносимости такого отладчика.

В данном проекте реализовывался консольный интерфейс и расширение библиотеки поддержки времени выполнение, доступное пользователю через определение глобальной переменной ENABLE_DEBUGGER.

3. Интерфейс отладчика

Минимальный набор команд, который должен выполнять любой отладчик — это установка и сброс точек прерывания, печать отладочной информации на данной точке и продолжение выполнения, при этом выполнение может приостановиться на следующем шаге, по выходу из следующей функции или на следующей точке прерывания. При этом также стоит включить в минимальный набор команду аварийного завершения программы, чтобы можно было моментально выйти из процесса отладки.

Этот набор может иметь различные способы реализации в зависимости от входного языка и внутренних структур компилятора. В качестве примеров разрабатываемого интерфейса можно рассмотреть упомянутые ранее отладчики языков Рефал-2, Рефал-5, Рефал-6. Их интерфейсы кратко описаны ниже. А описание разработанного и реализованного интерфейса отладчика Простого Рефала можно прочитать в гл. 6.1.

3.1. Примеры интерфейсов отладчиков

3.1.1. Отладчик Рефала-2

Отладчик Рефала-2 имеет довольно непривычный для пользователя, но зато очень простой в реализации интерфейс. В качестве способа выбора команды — выбирается одно из одиннадцати приглашений, выдающиеся пользователю программой отладки. Ввод же текстовых данных ограничивается вводом аргументов этих команд — управляющей информацией в ответ на приглашение. Аргумент ограничивает двумя форматами — число (номер шага), либо список имен функций с пробельными разделителями. При этом управляющая информация, задаваемая в ответ на приглашения, делится на две группы — управление остановом и управление прокруткой. Под прокруткой подразумевается выдача протокола о ходе выполнения Рефал-программы по отдельным шагам или функциям. Информация об отдельном шаге состоит из:

- номера шага;
- ведущего функционального терма;
- результата выполнения шага.

Информация об обращениях к заданным функциям состоит из:

- номера шага, на котором произошло обращение;
- ведущего терма;
- номера шага, на котором завершилось полное вычисление обращения к функции;
- окончательный результат замены, т.е. то выражение, которое получается, когда в выражении, возникшем из обращения к функции, не осталось ни одного функционального терма.

Задание на прокрутку может быть двух видов:

- задание диапазона прокрутки (для прокрутки по отдельным шагам);
- задание условий прокрутки (для прокрутки по вызовам некоторых функций). [3]

Помимо неудобства интерфейса управления отладчиком, возникает проблема большого объема отладочной информации, которой после запуска невозможно управлять — остановить вывод на середине диапазона прокрутки, убрать лишние поля. Зато такой узкоспециализированный интерфейс прост в реализации и не требует сложной внутренней структуры отладчика.

3.1.2. Отладчик Рефала-5

Отладчик Рефала-5 имеет интерфейс схожий с интерфейсом терминалов (например, cmd в системах семейства Windows или bash в дистрибутивах Linux). Отладчик в начале своей работы и перед исполнением каждой команд он выводит приглашение к вводу:

```
TRACE>
```

Ввод представляет собой запись:

```
имя_команды аргумент
```

При этом, часть аргументов могут быть необязательными или параметром из списка достустимых, а имя каждой команда имеет набор синонимов, в т.ч. одно-, двухбуквенных, сокращающих запись.

Отладчик помимо минимального набора команд, установленного выше, предоставляет также дополнительный набор для управления выводом отладочной информации характеристик процесса исполнения.[4]

Так, отладчик Рефала-5 имеет привычный пользователю и широко функциональный интерфейс, который однако требует развитой внутренней структуры отладчика, в т.ч. средств разбора вводимой пользователем информации.

3.1.3. Отладчик Рефала-6

Отладчик Рефала-6 так же, как и отладчик Рефала-2, имеет упрощенный интерфейс. Вместо имени команды и ее аргумента предлагается использовать простые однобуквенные команды, но при этом интерфейс в плане функциональность не уступает отладчику Рефала-5 насколько это возможно, минимизировав разбор аргументов — аргументы остались только для расстановки точек прерывания и задания выполнения некоторого числа шагов.[5]

Данный интерфейс является некоторым промежуточным решением между крайне упрощённым интерфейсом отладчика и большой функциональностью. Однако именно по причине поддержки большого числа команд (а именно, 15[5]) упрощенный интерфейс становится недостатком с точки зрения пользователя — не всегда очевидно соответствие между именем однобуквенной командой и ее работы. Это заставляет пользователя внимательно работать с описанием команд либо учиться на своих ошибках. В данной же работе предполагается разработка т. н. «дружелюбного» отладчика, команды для работы с которым будут интуитивно понятны пользователю.

4. Модификация алгоритмов кодогенерации для получения отладочной информации

Для реализации алгоритмов отладчика нужно помимо создания отладочной библиотеки времени выполнения несколько изменить работу алгоритмов компиляции программы. Так нужно добавить генерацию таблицы переменных и дополнить информацию о функции во входной программе именем модуля, в котором эта функция была определена.

4.1. Изменение дескриптора рефал-функции

Поскольку в практики возможны случаи компиляции программы из разных модулей, при чем таких, что хотя бы в паре модулей встречаются определения одноименных функций, и компилятор вполне разрешает такие конфликты, то для отладки таких программ может потребоваться узнать какое именно определение было использовано для вызова функции. Для этих целей нужно изменить дескриптор функции, который генерирует компилятор для описания скомпилированной для режима интерпретации рефал-функции. В качестве имени модуля функции нужно брать имя файла, в котором эта функция определена.

Для решения проблемы стоит учесть, что дескриптор функции определен только для режима интерпретации, как и отладочная библиотека. Поэтому вносить изменения нужно только в код режима интерпретации. Иначе в прямой кодогенерации появятся бесполезные определения и участки кода, которые могут вызывают ошибки при использовании некоторых поддерживаемых компиляторов.

Однако возникает проблема: определение режима компиляции и определение целевого файла для записи сгенерированного кода на C++ выполняются на разных проходах компилятора. Определение режима компиляции в целевом коде происходит при генерации последовательности элементарных команд, описывающих скомпилированную рефал-функцию. Так в режиме смешанной компиляции участки режима прямой кодогенерации и режима интерпретации окружаются директивами условной компиляции препроцессора. При этом для каждой функции, а их в модуле может быть

множество, генерируется свое пространство имен внутри директив, которые также определены отдельно для последовательности кода каждой функции. То есть, если просто вставить внутри каждого блока определенного только для режима интерпретации вставить глобальную переменную, содержащую имя модуля, то, без дополнительной обработки, появится множественное определение переменной. В связи с этим возможны два решения — генерировать определение глобальной переменной внутри пространства имен функции, либо, выполнив первое определение, убирать все последующие. В первом случае, появляются избыточные переменные, которые однако, могут убираться оптимизационными средствами компиляторов C++. Во втором же может пострадать читабельность сгенерированного кода — при такой генерации определения сложно управлять положением определения глобальной переменной, и его придется искать внутри кода.

4.2. Генерация отладочной таблицы переменных

Предлагается для отладки программ на каждом шаге для активной функции составлять отладочную таблицу переменных, в которую будут входить имена переменных из сопоставленного образца и соответствующих им данных — ссылку на распознанную часть аргумента. Для точного описания, в имя переменной помимо ее типа (e-, s-, t-переменные), ее имени в программе добавляется индекс глубины, показывающий вложенность функции, в которой она была впервые определена.

Предлагается, составление такой таблицы начать там же, где происходит распознавание переменных и констант и генерируются команды для заполнения вспомогательных таблиц компилятора. Таким образом, формирование отладочной таблицы переменных начинается на этапе генерации высокоуровнего RASL'a (см. Интерфейсы компилятора, шестой проход компилятора). Поскольку на это этапе нельзя определить в каком режиме происходит компиляция, то все, что можно сделать — добавить новы команды на заполнение таблиц. Далее, когда происходит деление процесса компиляции на режимы интерпретации и прямой кодогенерации, в коде для интерпретатора появляются команды на добавление в таблицу переменных с необходимой информацией по ним, а код для выполнения, содержит только комментарии с той же информацией, которую при желании можно распарсить и получить ту же отладочную таблицу.

5. Библиотека поддержки времени выполнения

Библиотека поддержки времени выполнения содержит следующие компоненты:

- Поддержка самого языка (файлы refalrts.h и refalrts.cpp) обеспечивает имитацию абстрактной рефал-машины и включает в себя такие функции:
  - определение структур данных узлов поля зрения и интерпретируемых команд;
  - реализации элементарных операций сопоставления с образцом, распределения памяти и сборки результата выполнения функции;
  - средства распределения памяти для новых узлов, поддержка списка свободных блоков;
  - вывод дампа поля зрения в отладочных целях;
  - основной цикл программы;
  - интерпретатор (для режима интерпретации).
- Библиотека функций, написанных на C++ (Library) включает в себя функции, которые невозможно написать на Рефале: средства ввода-вывода, арифметические операции, операции преобразования атомов и т.д.

5.1. Отладочная библиотека

Отладочная библиотека состоит из класса отладчика, который содержит открытые поля — классы вспомогательных структур, таких как отладочная таблица переменных, множество точек прерывания, таблица трассируемых функций — и отрытые методы для управления работой отладчика — выполнение отладочного цикла, поверки условий прерывания, выполнение команд, вспомогательные функции.

Вся библиотека окружена директивами условной компиляции препроцессора. Таким образом, отладочная библиотека будет включена в библиотеку поддержки времени выполнения при определении ENABLE_DEBUGGER в качестве переменной окружения или флагом компилятора C++.

Подробнее о командах отладчиках, реализованных в отладочной библиотеке можно прочесть в разделе 6.Руководство пользователя.

6. Руководство пользователя

Для отладки программы на Простом Рефале скомпилировать в режиме интерпретации, установив макрос ENABLE_DEBUGGER компилятора C++. Опции компилятору C++ при компиляции исходного кода можно передать через переменные окружения (см. [sref-guide]).

6.1. Общее описание интерфейса

Интерфейс отладчика для Простого Рефала создавался по примеру интерфейса отладчика Рефала-5 (см. гл. 3.1.2). После запуска программы с отладчиком происходит прерывание на нулевом шаге — до вызова функции Go, с которой в Простом Рефале начинается выполнение любой программы. Во время прерывания для ввода команд выдается приглашение пользователю:

```
debug>
```

Также на каждом шаге прерывания на консоль выводится номер текущего шага и имя активной функции.

Каждая команда имеет как полное имя так и набор синонимов вплоть до однобуквенных сокращений. У команд есть обязательные и необязательные аргументы команд, также есть команды с набором определенных параметров.

Для комплексного описания общий набор команд можно разделить на следующие логические подмножества:

- **команды получения вспомогательной информации об отладчике и текущем состоянии процесса отладки**; этот набор команд выводит справку по командам отладчика, выполняет вывод установленных точек и условий прерывания;
- **команды установки и сброса точек прерывания и условий прерывания**; это команды для установки/сброса точек прерывания по имени активной функции или номеру текущего шага, активации условия прерывания при пре-

вышении количества шагов после последней остановки или при превышении заданного числа узлов памяти;

- **команды вывода отладочной информации**; в этом наборе присутствуют команды вывода отладочной таблицы переменных, значения отдельной переменной по ее имени, вывода активного выражения, активной функции, ее аргумента, результата предыдущего шага или цепочки пропущенных предыдущих шагов (после команды `next` или `run`), добавление или удаление из списка трассируемых функций;
- **команды возврата к выполнению программы**; в это подмножество входят команды на выполнение шага, на выполнение следующей активной функции до пассивного результата, на выполнение до следующей точки прерывания, также есть команда для прерывания процесса отладки и выхода из программы.

Имена некоторых команд составлены из нескольких ключевых слов — имен других команд. Соответственно, каждую такую часть составного имени можно заменить на синоним простой команды и получить команду синонимичную составной.

Вывод отладочной информации и команд print break, print tace по умолчанию производится в стандартный поток вывода, однако, его можно явно перенаправить в файл. Пример для команды выводы значения переменной Листинг 1.

Листинг 1: Пример перенаправления вывода в файл. Вывод значения переменной по ее имени

```
print e.VarName > filename.txt
print e.VarName >> filename.txt
```

В обоих случаях содержимое переменной e.VarName не будет выводиться на экран. В первом случае, значение будет записано в файл. Во втором, значение будет дописано к содержимому файла.

6.2. Команды получения вспомогательной информации об отладчике

6.2.1. help, h

Команда для вывода справки по всем командам отладчика. Без аргумента. Вывод только в стандартный поток вывода. В справке через запятую перечисляются в строке синонимичные имена команды, как и в заголовках команды этого Руководства, и описание функции. О параметрах можно узнать из описания работы. Это может быть имя функции или переменной, номер шага, количество узлов и пр. Стоит также заметить

6.2.2. print break

Команда выводит множество установленных точек прерывания. Без аргумента. Возможен вывод в файл. Общее множество точек прерывания делится на подмножества точек по типу — прерывание по имени функции или прерывание по номеру текущего шага.

6.2.3. print trace

Команда выводит таблицу трассируемых функций. В таблице имена функций, которым в соответствие ставится указатель на файл. Поскольку эти указатели не несут для пользователя осмысленную информацию, а получение по ним имени файла возможно только с помощью системных функций, что противоречит требованию к переносимости отладчика, то при выводе таблицы указывается только то, будет ли вывод в файл или в стандартный поток вывода, если имя файла не установлено.

6.3. Команды для установки и сброса точек прерывания

6.3.1. breakpoint, break, b

Команда устанавливает точку прерывания. В качестве аргумента можно указать имя функции — имя чувствительно к регистру, — по вызову которой произойдет пре-

рывание, либо номер шага, на котором будет выполнена остановка. Номер шага — целое число, слитно перед которым пишется знак решетки (‘#’). Вывода нет.

6.3.2. clear, cl, remove, rm

Команда, обратная предыдущей (6.3.1); снимает точку прерывания. Аргумент тот же. Вывода нет.

6.3.3. steplimit

Команда устанавливает точку прерывания относительно номера текущего шага. Аргумент — целое число, однако, имеет смысл только для натурального аргумента. Вывода нет. Точка прерывания добавляется во множество всех установленных точек прерывания.

6.3.4. memorylimit

Команда устанавливает точку прерывания по условию превышения установленного числа узлов памяти. Аргумент — неотрицательное целое число, характеризующее предел количества узлов памяти. При превышении лимита, число сбрасывается.

6.4. Команды вывода отладочной информации

6.4.1. vars

Команда выводит отладочную таблицу переменных. Аргумента нет. Вывод можно перенаправить в файл. В таблице указаны имена, распознанных для текущего шага переменных, и значения этих переменных. Суффикс в имени вида ‘*#число*’, где *число* — это индекс глубины определения переменной, или вложенности вызова функции, внутри которой переменная была впервые разобрана.

6.4.2. print ИмяПеременной, ИмяПеременной

Команда выводит значение переменной по ее имени. Аргумент — имя переменной (указано), который может включать в себя суффикс с глубиной определения переменной. Если индекса глубины нет, то может быть выведено несколько значений — по одному для каждой определенной глубины. Вывод можно перенаправить в файл.

6.4.3. print call

Команда выводит текущее активное выражение. Аргумента нет. Вывод можно перенаправить в файл.

6.4.4. print callee

Вывод имени текущей активной функции. Аргумента нет. Вывод можно перенаправить в файл. В качестве имени активной функции выбирается первый терм после открывающей скобки конкретизации.

6.4.5. print arg

Команда выводит аргумент текущей активной функции. Аргумента нет. Вывод можно перенаправить в файл. В качестве аргумента активной функции выбирается все выражение после терма ее имени и до закрывающей скобки конкретизации.

6.4.6. print res

Команда выводит результат выполнения предыдущего шага. Аргумента нет. Вывод можно перенаправить в файл. Если было пропущено несколько шагов, то выводится результат последнего.

6.4.7. trace, tr

Команда добавляет функцию в таблицу трассируемых. Аргумент — имя функции, чувствительно к регистру. Возможен вывод данных трассировки в файл. После это, при каждом вызове функции будет выполняться вывод с именем окружающей функции — в аргументе которой был встречен вызов трассируемой функции — и вызовом трассируемой функции.

6.4.8. notrace, notr

Команда обратная предыдущей (6.4.7); удаляет из таблицы трассируемых. Аргумент — имя функции, чувствительно к регистру. Вывода нет.

6.5. Команды возврата к выполнению программы

6.5.1. step, s

Команда выполнения следующего шага и последующим прерыванием. Аргумента нет. Вывода нет. Множество точек прерываний и прочие условия не меняются.

6.5.2. next, n

Команда выполнения активной функции до пассивного результата или до удовлетворению условиям прерывания и последующей за этим остановкой. Аргумента нет. Вывода нет. Множество точек прерываний не меняется. Добавляется условие остановки после выполнения активной в момент вызова команды функции.

6.5.3. run, r

Команда продолжения выполнения программы до следующей точки прерывания или до удовлетворению условиям прерывания. Аргумента нет. Вывода нет. Множество точек прерываний и прочие условия не меняются.

6.5.4. . (точка)

Команда, повторяющая последнюю команду выполнения. Аргумента нет. Вывода нет. При выполнении первой командой выполнения соответствует команде `step`.

6.5.5. quit, q

Завершение выполнения процесса отладки и процесса выполнения программы и выход из нее. Аргумента нет. Вывода нет.

Заключение

По результатам получен пошаговый отладчик Простого Рефала, работающего в консольном режиме для программ, скомпилированных в режиме интерпретации, и располагающий основными командами любого отладчика.

Для реализации отладчика был незначительно модифицирован код компилятора, написанный на входном языке — Простом Рефале, и изменена библиотека поддержки времени выполнения, написанная на выходном языке — C++. При этом изменения затрагивают только режим интерпретации и сохраняется возможность запуска программы без отладчика.

В планах ближайшего развития можно добавить поддержку скриптов при распознавании команд отладки, безымянная команда, повторяющая последнюю команду, добавление параметра повторяющего команду или последовательность команд заданное число раз, ведение протокола отладки, переход к графическому пользовательскому интерфейсу.

Библиография

[1]: Коновалов А.В, Простой Рефал. Руководство пользовател, 2016

[2]: Коновалов А.В. и др, Интерфейсы между отдельными проходами компиляции, 2016

[3]: Белоус Л.Ф., Романенко Н.Н, Рефал-2. Компиляция и исполнение Рефал-програм, 1987

[4]: Турчин В., Фрейм: РЕФАЛ-5. Руководство и справочник. Получено 10 января 2017 г., из Содружество «РЕФАЛ/Суперкомпиляция»: http://refal.ru/rf5_frm.htm, 1989

[5]: Климов Арк.В, Рефал-6 для Windows-95/98/NT (а также для DOS), 2001

sref-guid: Коновалов А.В, Простой Рефал. Руководство пользовател, 2016