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

ФАКУЛЬТЕТ _____ИНФОРМАТИКА И СИСТЕМЫ УПРАВЛЕНИЯ_____

КАФЕДРА _ТЕОРЕТИЧЕСКАЯ ИНФОРМАТИКА И КОМПЬЮТЕРНЫЕ ТЕХНОЛОГИИ__

РАСЧЕТНО-ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

К КУРСОВОЙ РАБОТЕ

НА ТЕМУ:

«Расширение пошагового отладчика Рефала-5λ»

Студент ___ИУ9-71Б____ _____ **____С.М. Санталов_**
(Группа) (Подпись, дата) (И.О.Фамилия)

Руководитель курсовой работы _____ **____А.В. Коновалов_**
(Подпись, дата) (И.О.Фамилия)

Консультант _____ _____
(Подпись, дата) (И.О.Фамилия)

2021 г.

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ

Для упрощения отладки Рефал-5λ предоставляет разработчику пошаговый отладчик, реализованный как модификация библиотеки поддержки времени выполнения. Взаимодействие с отладчиком происходит посредством командной строки. Однако текущего функционала недостаточно для комфортного анализа происходящего в коде.

Чтобы решить проблему, предлагается дополнить отладчик функциями для анализа контекста вызова функции, расширить синтаксис и улучшить вывод некоторых команд, реализовать настраиваемый вид вывода, улучшить работу с файлами и внести прочие небольшие улучшения.

Таким образом, необходимо получить более полный и удобный в использовании отладчик программ на Рефале-5λ.

1 Рефал

1.1 Синтаксис

Программа на Базисном Рефале состоит из набора функций. *Функция* определяется своим именем и списком предложений. Предложением является выражение вида [1]:

```
Образец = результат;
```

Образец задает множество значений аргумента функции, к которому применимо предложение. *Результат* описывает возвращаемое значение функции, при попадании аргумента функции в описанное множество.

При исполнении функции происходит сопоставление аргумента с образцами предложений сверху вниз, в порядке перечисления предложений. При первом успешном сопоставлении функция завершается, возвращая значение, описанное результатом того предложения, образец которого сработал.

Если предложения отсутствуют или не удалось сопоставить аргумент ни с одним образцом, происходит аварийная остановка программы с ошибкой `recognition impossible.` Выполнение программы начинается с вызова функции `Go` с пустым аргументом.

В ходе выполнение программы происходит преобразование *объектных выражений* – последовательностей символов (символ может быть литерой, числом или функцией) и структурных скобок.

Образец представляет из себя объектное выражение, дополненное переменными. Результат по сравнению с образцом дополняется скобками активации (открывающей «<» и закрывающей «>»), которые означают вызов функции, расположенной сразу после открывающей скобки активации, с аргументом, расположенным между функцией и закрывающей скобкой активации.

Переменные в образце в процессе сопоставления с аргументом принимают свои значения, равные определенным участкам объектного выражения аргумента функции. После присваивания значений переменным, они могут быть использованы в правой части предложения при построении результата.

Пример программы, вычисляющей десятое число Фибоначчи, представлен в листинге 1.

Листинг 1. Программа на Базисном Рефале, вычисляющая десятое число Фибоначчи.

```
$ENTRY Go {
  = <Prout <Fib 10>>
}

Fib {
  0 = 1;
  1 = 1;
  s.N = <Add <Fib <Sub s.N 1>> <Fib <Sub s.N 2>>>;
}
```

1.2 Рассахаривание

Одним из улучшений Рефалф-5λ является расширенный синтаксис предложений. Они состоят из левой и правой части [2].

Левая часть описывает множество аргументов, для которых применимо предложение, и состоит из образца, за которым могут следовать условия.

Условие следует после образца и описывает дополнительные ограничения. Оно состоит из расширенного результатного выражения, за которым через знак «:» следует еще один образец. При проверке условия значение, формируемое расширенным результатным выражением, сопоставляется с образцом условия.

Если сопоставление прошло успешно, то условие считается выполненным, если нет – происходит переход к следующему предложению в функции.

Правая часть дополнена присваиваниями и классическими блоками.

Присваивания служат для вычисления дополнительных значений и присваивания их переменным. Присваивание выглядит как расширенное результатное выражение, за которым следует левая часть. При неуспешном сопоставлении с левой частью в присваивании возникает ошибка.

Расширенное результатное выражение, упоминаемое выше, является результатным выражением, после которого через знак «:» следуют тела функций. Значение результатного выражения передается в первую указанную функций. Значение, вычисленное первой, передается во вторую и так далее. Значение, сформированное последней функцией, является возвращаемым значением расширенного результатного выражения.

Классический блок аналогичен расширенному результатному выражению, состоящему из одного тела функции.

Для удобства программиста дополнительно вводится синтаксис переопределения переменных: переменные со знаком «^» после имени не считаются повторными и связываются со значением заново.

Для работы с пошаговым отладчиком важно понимать, что в Рефале-5λ вышеперечисленные конструкции (кроме условий) являются синтаксическим сахаром. Это значит, что они преобразуются во время компиляции в Базисный Рефал.

Необходимо понимать, как именно происходит рассахаривание, чтобы понимать, как выстроить соответствие преобразованного кода, с которым работает отладчик, исходному коду на Рефале-5λ.

В ходе рассахаривания код преобразуется следующим образом [3]:

- переопределенные переменные, оканчивающиеся на «^», переименовываются;
- присваивания преобразуются в блоки;
- блоки преобразуются в вызовы вложенных функций;
- вложенные функции преобразуются в глобальные.

К переименованным переменным добавляется суффикс `#n`, где `n` – уровень вложенности, на котором переменная создана.

Остальные синтаксические конструкции в конечном счете преобразуются в глобальные функции, имена которых также строятся путем приписывания к имени функции, предложение которой преобразуется, следующих суффиксов:

- `$n` – n-е предложение исходной функции;
- `=n` – n-е присваивание;
- `\n` – n-я вложенная функция;
- `?n` – n-е условие;
- `:n` – n-й блок;

Компилятором в ходе выполнения оптимизация добавляются и другие суффиксы (`*n` и `@n`), однако при работе с пошаговым отладчиком можно считать, что функция с таким суффиксом эквивалентна исходной.

1.3 Поле зрения

Необходимо отметить, что является данными, с которыми работает Рефал-5λ. При выполнении предложения функции, образец сопоставляется с аргументом функции, являющимся объектным выражением.

Сама функция и ее аргумент, заключенные в скобки активации («<» и «>»), являются активным подвыражением, означающим вызов функции. Все активные

подвыражения и результаты предыдущих вызовов активных подвыражений формируют *поле зрения* программы [4]. Пример поля зрения представлен на рисунке 1.

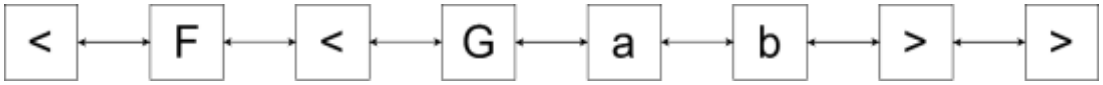

Рис. 1: Пример поля зрения. `F` и `G` – функции, a и b – символы.

Для работы с пошаговым отладчиком необходимо понимать, что представляет из себя шаг исполнения.

На каждом шаге происходит поиск *первичного активного подвыражения* – самого левого активного подвыражения в поле зрения, не содержащего других активных подвыражений. Затем проверяется, что после открывающей скобки активации стоит функция, и эта функция вызывается с аргументом, заключенным в поле зрения между ней и закрывающей скобкой активации. Вид поля зрения из рисунка 1, если вызов функции `G` заменился на пустоту, представлен на рисунке 2.

Рис. 2: Поле зрения из рисунка 1 после замены вызова функции `G` на пустоту.

На первом шаге выполнения поле зрения содержит только функцию `Go` с пустым аргументом. Соответственно поле зрения, изображенное на рисунке 1, может является полем зрения программы, представленной в листинге 2, на втором шаге выполнения.

Листинг 2. Программа, приводящая к полю зрения, изображенному на рисунке 1, на втором шаге выполнения.

```
Go {
    /* пусто */ = <F <G 'ab'>>;
}
```

С точки зрения реализации, в Рефале-5λ поле зрения представлено двусвязным списком, каждый элемент которого может быть символом, структурной скобкой или скобкой активации [5].

Парные структурные скобки содержат указатели друг на друга. Со скобками активации хитрее – они провязаны в стек. Каждая открывающая скобка активации содержит указатель на парную ей закрывающую, однако закрывающая указывает на ту открывающую, вызов активного подвыражения которой будет осуществляться после текущей. Соответственно, самым первым элементом такого стека будет открывающая скобка первичного активного подвыражения. Пример структуры стека представлен на рисунке 3.

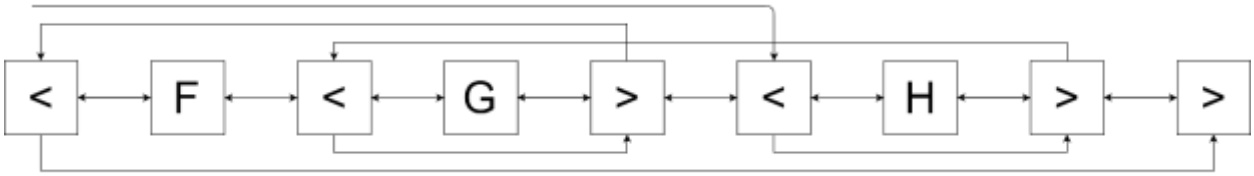

Рис. 3: Структура стека, образуемая скобками активации.

Благодаря такой организации скобок активации, поиск первичного активного подвыражения на каждом шаге осуществляется за константное время, так как для этого нужно взять первые два элемента со стека.

1.4 Пошаговый отладчик

С точки зрения пользователя пошаговый отладчик выглядит как интерфейс командной строки с следующими поддерживаемыми функциями:

- установка точек остановки на имя функции или на номер шага (команды `breakpoint FuncName`, `breakpoint #ddd`);
- выполнение до становления пассивным участка поля зрения на месте первичного активного подвыражения (команда `next`);
- переход к следующему шагу (команда `step`);

- печать текущего вызова и его параметров (команды `print call`, `print callee`, `print arg`);
- печать переменных (команды `vars`, `print e.VarName`, `e.VarName`);
- установку ограничений по памяти и по количеству шагов (команды `memorylimit`, `steplimit`);
- печать значения, возвращенного предыдущей функцией (команда `print res`);
- трассировка функции (команда `trace FuncName`).

Отладчик реализован путем модификации библиотеки поддержки времени выполнения компилятора. На каждом шаге выполнения программы вызывается функция-обработчик отладчика. Она проверяет текущий шаг на наличие точек остановки, превышение ограничения по памяти, количеству шагов или же на необходимость остановки, если на предыдущем шаге пользователь выбрал переход к следующему шагу.

В случае положительного результата проверки, исполнение программы на Рефале-5λ приостанавливается и отладчик запрашивает у пользователя команду для дальнейших действий. В этот момент пользователь может взаимодействовать с отладчиком.

Для остановки программы на указанном пользователем вызове функции отладчик на каждом шаге сравнивает имя вызываемой функции с именем, указанным пользователем.

Для перехода к следующему шагу, отладчик использует стек скобок активации, сохраняя его вершину на текущем шаге, и сравнивая с ней вызываемую функцию на следующем.

Распечатка аргумента функции, переменных, точки остановки на номер шага работают за счет передачи в функцию-обработчик отладчика соответствующей информации из главного цикла библиотеки поддержки времени выполнения.

Для распечатки результата вызова предыдущего первичного активного подвыражения, распечатывается участок, на месте которого оно было, так как активные подвыражения в ходе выполнения заменяются на возвращаемые значения соответствующих функций. Сохраняя на каждом шаге два узла поля зрения, непосредственно предшествующих и следующих после первичного активного подвыражения, запоминается участок, на котором появится результат вызова.

Трассировка осуществляется проверкой, указано ли пользователем имя вызываемой функции для трассировки. Если указано, имя функции и ее аргумент печатаются в указанный пользователем поток вывода.

2 Разработка

Вышеописанных возможностей отладчика бывает недостаточно для комфортной разработки.

Например, при печати контекста текущего вызова, важно знать не только аргумент функции и значения переменных, но и общий контекст функции в поле зрения, что достигается полной печатью последнего: `print viewfield`, `print view`, `print vf`.

Из всего контекста текущего вызова наиболее важной информацией может быть порядок следования следующих вызовов и их аргументы. Эту информацию можно получить из распечатанного поля зрения, однако намного удобнее создать для этого отдельную команду `backtrace`, которая распечатает список вызовов и их аргументы в нужном порядке.

Рассмотрим пример поля зрения парсера арифметических выражений в листинге 3.

Листинг 3. Пример поля зрения парсера арифметических выражений.

```
<Prout#0:0
 <ParseE#1716295250:435960915
  <DoLexer#1716295250:435960915 '9 - \(beta - m\)\) /
\(0 - 11 - 8 * \(1 / 2\)\)'
   ((('\(' )#Bracket 1 )
   .(('2' )#Number 2 )
   .(('-' )#Operator 4 )
   .(('e50' )#Identifier 6 )
   .(('\)' )#Bracket 9 )
   .(('/' )#Operator 11 )
   .(('\(' )#Bracket 13 )
   .(('-' )#Operator 14 ))15
   <DoLexer$5?1#1716295250:435960915 #True >
   <DoLexer$5?2#1716295250:435960915
   .<StringTypeAndLen#1716295250:435960915 '9'
   . <StringTypeAndLen$1?1#1716295250:435960915
   .  <DoIsNumber#1716295250:435960915 '9'
   .   (0 )>>>>>>>
[LAST]
```

Видно, что поле зрения довольно громоздко. Но еще более многословным будет выглядеть его стек вызовов (приводить его здесь излишне), так как его последним элементом будет все поле зрения. Заметим, что в листинге 3 большую часть экранного места занимают маркеры области видимости (суффиксы имен функций вида `#MMM:NNN`, уникальные для каждой единицы трансляции) и аргументы вызовов, и в ситуации, когда важен только порядок вызова, можно их

опускать. Для этого вводится новый режим отображения `skeleton`, пример вывода которого представлен в листинге 4.

Листинг 4. Вид стека вызовов в режиме `skeleton`

```
<DoLexer$5?1 ...>
<DoIsNumber ...>
<StringTypeAndLen$1?1 ...>
<StringTypeAndLen ...>
<DoLexer$5?2 ...>
<DoLexer ...>
<ParseE ...>
<Prout ...>
```

Противоположным режиму `skeleton` является режим `full`. Кроме того, для печати поля зрения и стека вызовов может быть полезно переключаться между выводом в одну строку и в несколько, что осуществляется режимами `oneline/multiline`.

Режимы оформляются в виде одноименных команд, глобально устанавливающих соответствующие настройки, и в виде префиксов, для применения только к текущей команде (листинг 5).

Листинг 5. Примеры префиксов команд.

```
oneline: print view
multiline: backtrace
skeleton: print view
multiline full: backtrace
```

Кроме того, полезно видеть не только порядок вызова, но и какая функция потребит результат вызова текущей функции (они не всегда равны). Для этого вводятся новые обозначения:

- `@N` – функция, расположенная на N-й позиции в стеке вызовов, то есть будет вызвана через N вызовов после текущего, если все промежуточные вызовы заменятся на пассивные объектные выражения;
- `^N` – функция, активное подвыражение которой содержит N других активных подвыражений, каждое из которых содержит первичное активное подвыражение.

Например, для поля зрения, представленного в листинге 6, результат вывода команды `backtrace` продемонстрирован в листинге 7.

Листинг 6. Пример поля зрения.

```
<F <G <K>> <H>>
```

Листинг 7. Вывод команды `backtrace` для поля зрения из листинга 6.

```
@0  ^0  <K ...>
@1  ^1  <G ...>
@2      <H ...>
@3  ^2  <F ...>
```

Естественным продолжением команды backtrace является возможность печатать отдельные элементы стека вызова и ставить точки остановки на них, используя введенные выше обозначения `@N` и `^N` (например, `print ^1` или `breakpoint @3`).

Дополнительно вводится команда `next '@'|'^'N`, эквивалентная паре команд `breakpoint '@'|'^'N` и `next`.

Кроме того, в отладчик вносятся небольшие, но полезные изменения: возможность задания имен файлов в кавычках, переименование команды `print` в команду `list`, вывод имен файлов в списке трассируемых функций, замена

команды «точка», повторяющей последнюю команду, на команду «пустая строка».

Таким образом, для удобства разработки программ на Рефале-5λ предлагается модифицировать пошаговый отладчик следующим образом:

- добавить команду печати поля зрения;
- добавить режимы `oneline/multiline` и `full/skeleton`;
- добавить в синтаксис команд префиксы режимов;
- добавить в синтаксис команд имена файлов в кавычках;
- добавить команду `backtrace`;
- добавить точки остановки на элементы стека, распечатываемые командой `backtrace`;
- изменить команду `print` на команду `list`;
- выводить имя файла в команде `list trace`;
- изменить команду «точка» на команду «пустая строка».

3 Реализация

Выбор языка и технологий для реализации был обусловлен уже используемым набором в компиляторе Рефала-5λ – его библиотека поддержки времени выполнения написана на C++98. Для сборки используется набор скриптов командной строки.

Разработка велась с использованием Git, была сделана отдельная ветка в репозитории https://github.com/bmstu-iu9/refal-5-lambda, куда заливалась история изменений, привязанная к открытой проблеме номер 85 https://github.com/bmstu-iu9/refal-5-lambda/issues/85.

Особенностью реализации является наличие в библиотеке поддержки времени выполнения объекта `refalrts::VM`, позволяющего получить доступ к основным переменным времени выполнения. С учетом этого, реализация первой версии команды `print viewfield` осуществлялась оборачиванием соответствующего метода `refalrts::VM`.

Чтобы сразу добавить и режимы `oneline/multiline`, и возможность вводить имя файла в кавычках, и команду «пустая строка», и улучшить интерактивность интерфейса командной строки, было решено изменить схему обработки вводимых данных.

В старом варианте ввод команды осуществлялся поэтапно: сначала сама команда, затем, если нужно, ее аргумент и затем файл вывода (у некоторых команд). Это блокировало возможность использования «пустой строки», так как команда обязательна. Чтобы опустить файл вывода необходимо было лишний раз нажимать Enter, и было легко запутаться, если ввести команду, но не ввести аргумент, и нажать Enter.

В новой схеме команда считается одной строкой, синтаксис которой представлен в листинге 8. Вышеописанные проблемы легко решаются благодаря новому синтаксису.

Листинг 8. Грамматика команды пошагового отладчика. Терминал `STRING` означает любую ASCII строку, `QUOTED_STRING` – строковый литерал, заключенный в двойные кавычки.

```
Command ::= Prefixes Option Arg File.
Prefixes := Prefix PrefixesEnd | ε.
PrefixesEnd := Prefix PrefixesEnd | ':'.
Prefix := 'oneline' | 'multiline' | 'full' | 'skeleton'.
Option := 'print' | 'breakpoint' | … | ε.
Arg := STRING | ε.
```

```
File := '>' FileName | '>>' FileName.
FileName := STRING | QUOTED_STRING
```

Была написана функция, разбирающая строку команды и возвращающая экземпляр класса `refarts::Debugger::Cmd`, который затем передается в нужную функцию-обработчик. Последняя при необходимости может запросить у пользователя недостающие части команды (например, аргумент).

Настройки режима отображения хранятся в соответствующих полях класса `refalrts::Debugger`, изменяются командами `oneline/multiline`, `full/skeleton` и временно перезаписываются префиксами, при указании их пользователем.

Получение индексов `@N` в команде `backtrace` реализовано путем обхода стека скобок активации, а индексов `^N` с помощью прохода по полю зрения от вершины стека влево. С точки зрения производительности, это приемлемо, так как будет обходиться порядка половины поля зрения, ибо по пути обхода будут встречаться только открывающие скобки активации, а их не больше половины от всех скобок активации, и обход начинается с самого левого активного подвыражения, не содержащего других активных подвыражений.

Точки остановки на элементы стека реализованы через поиск открывающей скобки активного подвыражения с индексом `@N` или `^N` и ее запоминание, после чего она проверяется на равенство с открывающей скобкой первичного активного подвыражения на каждом шаге.

Также был доработан вывод команды `help`, переименованы команды, добавлены имена файлов в `list trace`, и внесены другие небольшие изменения.

4 Тестирование

Тестирование проводилось на парсере арифметических выражений, написанном рекурсивным спуском. Парсер был скомпилирован с флагом `--`

`debug`, и отдельно был создано файл @refal-5-lambda-diagnostics.ini, содержащий параметр enable-debugger = true, включающий пошаговый отладчик. Итоговые примеры работы отладчика приведены ниже, на рисунках 4, 5, 6, 7, 8.

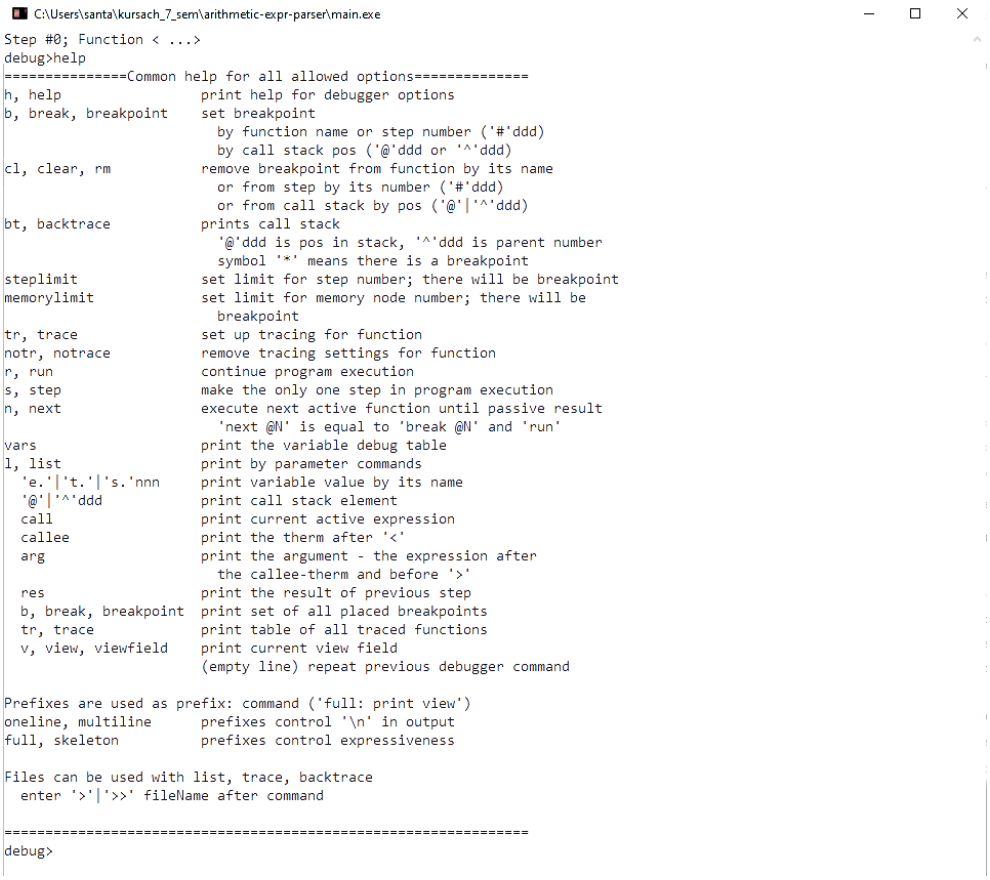

Рис. 4: Вывод команды `help`

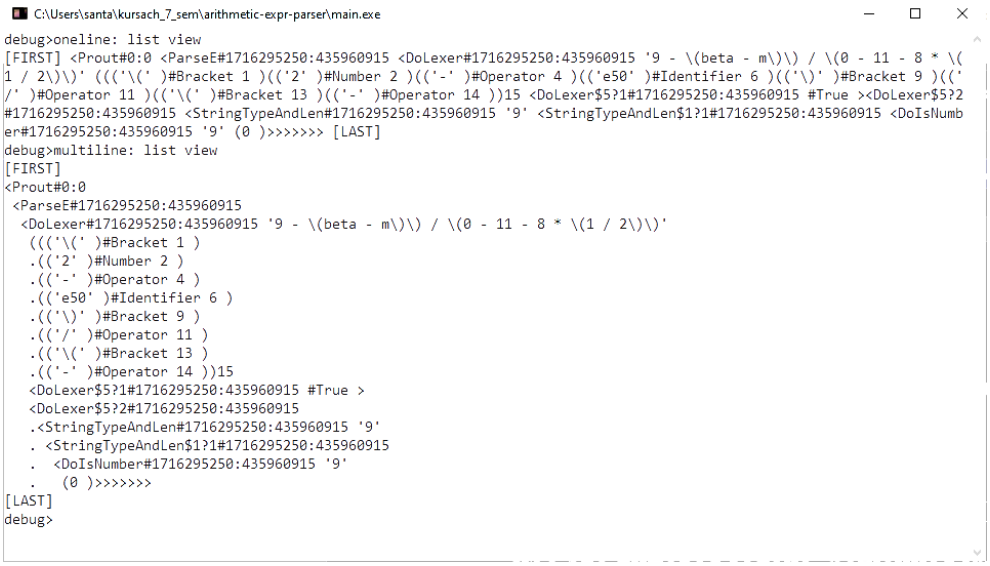

Рис. 5: Вывод команды `list view` в режимах `oneline` и `multiline`

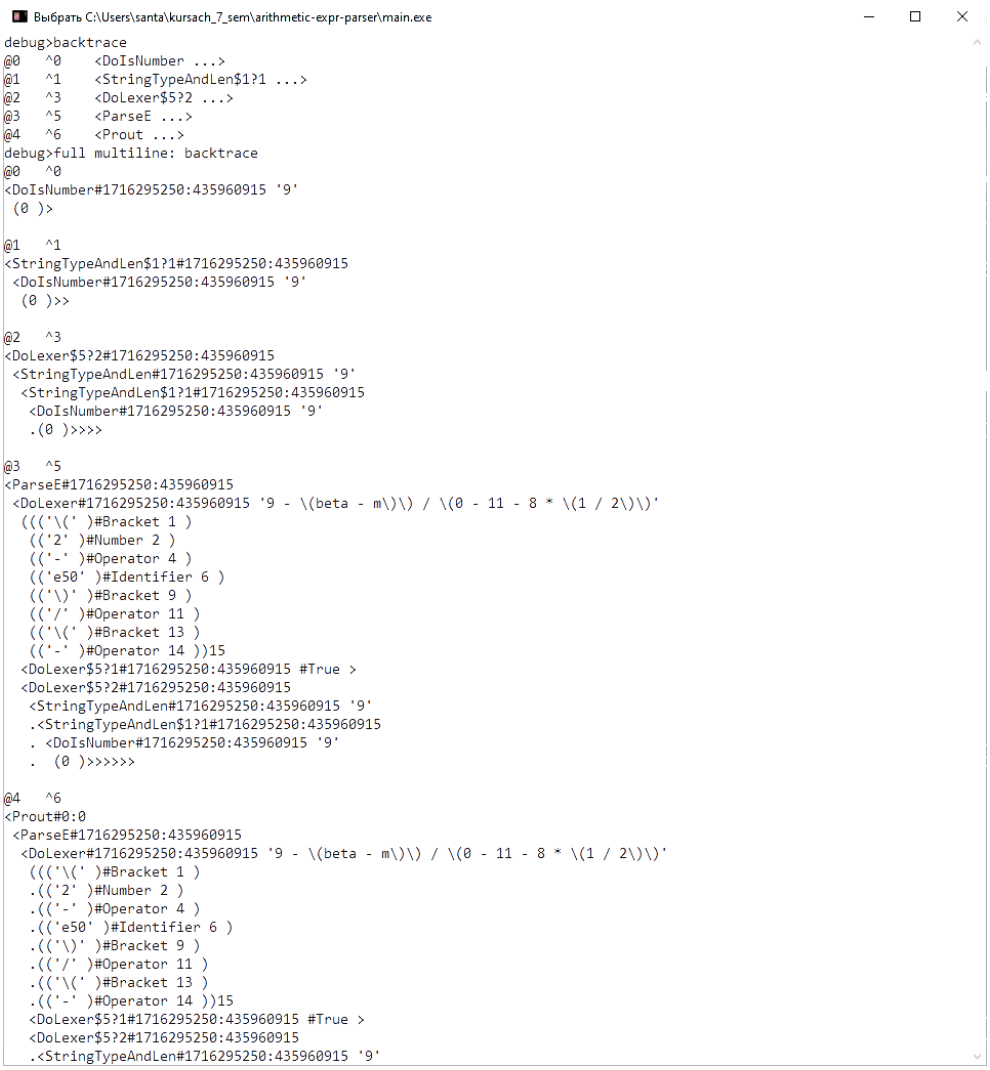

Рис. 6: Вывод команды `backtrace` в режиме `oneline skeleton` и в режиме `full multiline`.

Рис. 7: Точки остановки на элементы стека.

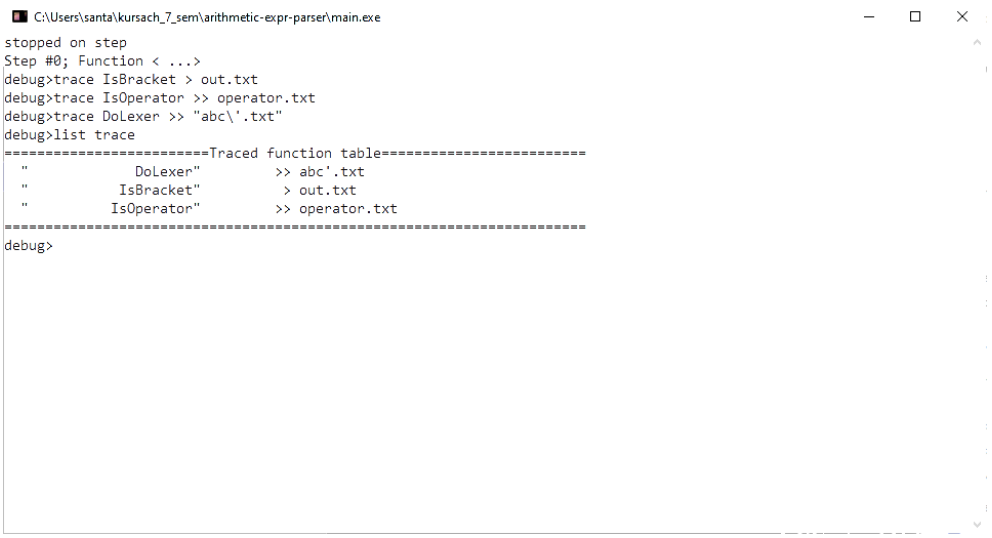

Рис. 8. Вывод имен файлов в `list trace`, и имена файлов в кавычках.

Однако в ходе использования отладчика выяснилось, что точки остановки и команда `next` не всегда корректно срабатывают. Выставив отладочные печати, выводящие указатели на скобки активации, выясняем, что они могут изменяться, хотя в соответствии с абстрактной Рефал-машиной не должны. Например, на рисунке 9 меняются указатели на открывающую скобку активации подвыражений `<StringTypeAndLen …>` и `<DoLexer …>`.

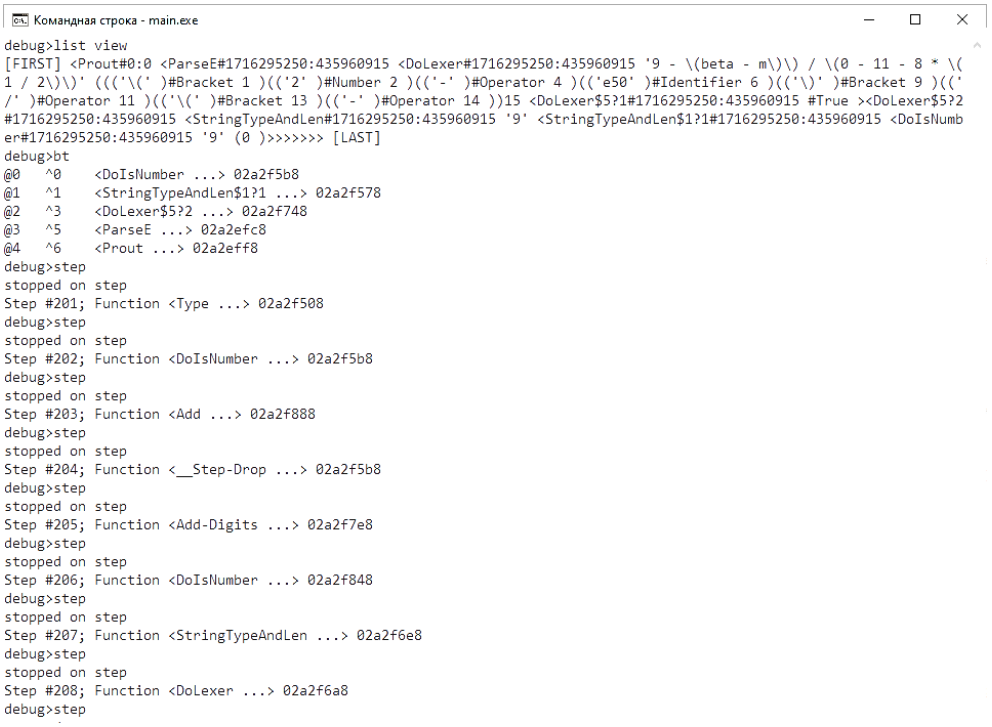

Рис. 9: Изменяющиеся указатели на скобки активации.

Заметим также, что вывод команды `backtrace` на рисунке 6 не соответствует полю зрения на рисунке 5, хотя должен. Это означает, что стек вызовов не содержит всех активных подвыражений, что также может приводить к пропускам точек остановки и некорректной работе команды `next`.

Кроме того, так как отладчик видит поле зрения и стек скобок активации до возвращения значения текущим вызовом, стек может быть пуст (например, когда первичное активное подвыражение является единственным активным подвыражением в поле зрения), что также мешает работоспособности команды `next`.

Решение вышеуказанных проблем в рамках курсового проекта затруднительно, поэтому остановимся на этапе их анализа.

ЗАКЛЮЧЕНИЕ

В итоге получилось переработать обработку команд, сделать их ввод более интерактивным и удобным. Решились проблемы с использованием файлов и команды «пустая строка».

Было реализовано четыре режима отображения, позволяющие выводить данные наиболее оптимальным способом. Новые команды для печати поля зрения и стека скобок активации дают больше возможностей для анализа контекста текущего вызова, что органично дополняется точками остановки на элементы стека.

Однако есть пространство для совершенствования. Особенности работы библиотеки поддержки времени выполнения делают в некоторых случаях точки остановки неработоспособными.

СПИСОК ИСПОЛЬЗУЕМОЙ ЛИТЕРАТУРЫ

1. Рефал-05. Язык Рефал-05, его отличия от Рефала-5 и общее подмножество. / URL: https://mazdaywik.github.io/Refal-05/2-syntax.html (дата обращения: 25.01.2021).
2. Рефал-5λ. О языке и компиляторе. / URL: https://github.com/bmstu-iu9/refal-5-lambda (дата обращения: 25.01.2021).
3. Рефал-5λ. Приложение В. Краткий справочник. / URL: https://bmstu-iu9.github.io/refal-5-lambda/B-reference.html (дата обращения: 25.01.2021).
4. Рефал-05. Реализация: списковое представление и интерфейс с языком Си. / URL: https://mazdaywik.github.io/Refal-05/5-implementation.html (дата обращения: 25.01.2021).
5. Исходный код компилятора Рефала-5*λ*. / URL: https://github.com/bmstu-iu9/refal-5-lambda/blob/master/src/lib/refalrts.h (дата обращения 25.01.2021).