Федеральное государственное бюджетное образовательное
учреждение высшего профессионального образования Московский
государственный технический университет имени Н.Э. Баумана

Курсовая работа
«Разработка плагина для IntelliJIDEA для языка Рефал-5λ»
по курсу
«Конструирование компиляторов»

Студентка группы ИУ9-71 Терешкина Д.Д.

Руководитель Коновалов А.В.

Москва, 2018

Содержание

1 Введение

Язык Рефал-5λ— точное надмножество Рефала-5λ, основным расширением которого являются функции высшего порядка [1].

IntelliJIDEA является интегрированной средой разработки (IDE) чешской компании JetBrains. Несмотря на то, что в первую очередь Java IDE, JetBrains повторно использовал платформу, на которой он построен для создания IDE для других языков. Существуют отдельные проекты для некоторых языков программирования – Python (PyCharm), Ruby (RubbyMine), HTML / JavaScript (WebStorm), ObjectiveC (AppCode), PHP (PhpShtorm) и C ++ (CLion). Сама платформа, однако, является расширяемой, и основная IDE – IntelliJIDEA – может поддерживать практически любой язык благодаря использованию пользовательских языковых плагинов. Что касается доступности IDE для Рефал— существует подсветка синтаксиса в различных редакторов, таких как текстовый редактор Vim и Far Manager

Например, подсветка синтаксиса в IntelliJ может быть реализованы таким образом, чтобы учитывать, какие идентификаторы относятся к типам, и дает им другой вид, то есть все зависит от написанного плагина.

Целью данной работы является создание плагина для IntelliJIDEA, который бы облегчил написание кода на Рефал-5λ , а именно создать систему интеллектуального автодополнения и подсветки синтаксиса.

2 Описание языка Рефал-5λ

Компилятор Рефала-5λ— оптимизирующий компилятор, поддерживающий возможность как компиляции в промежуточный интерпретируемый код, так и в исходный код на C++. Ключевая особенность компилятора — удобный интерфейс с языком C++.

2.1 Возможности языка

- Синтаксические расширения должны позволять писать более выразительный код не в ущерб эффективности (например, синтаксис присваивания вместо условий в роли присваиваний).
- Эффективная оптимизация на разных уровнях (преобразование синтаксического дерева, промежуточный императивный код, возможность прямой кодогенерации).
- Предсказуемая производительность — классическое списковое представление не прячет временные затраты в стадию сборки мусора.
- Переносимость компилятора — возможность его использовать как с любым компилятором C++, так и без оного.
- Достаточно богатая стандартная библиотека (по сравнению со стандартной библиотекой классического Рефала-5). Библиотека Library предоставляет в дополнение ко всем возможностям Рефала-5 ещё и двоичный ввод/вывод, библиотека LibraryEx — удобные вспомогательные функции и функции высшего порядка: Map, Reduce, «гибрид» MapReduce, которые существенно упрощают написание циклических конструкций.
- Инкапсуляция: поддержка именованных скобок — абстрактных типов данных. Содержимое таких скобок доступно только в той единице трансляции, где они созданы.

- Инкапсуляция: статические ящики. В отличие от глобальной копилки классического Рефала-5, можно объявить статический ящик в локальной области видимости, недоступный извне (кстати, копилка реализована поверх такого статического ящика в библиотеке Library).

- Утилита SRMake, позволяющая отслеживать зависимости между исходниками.

- Целевой файл компиляции — исполнимый файл операционной системы. Для запуска отдельного интерпретатора не нужно. [1]

2.2 Синтаксис языка Рефал-5λ

Язык и реализация предоставляют ряд дополнительных возможностей, которых нет в классическом Рефале-5

- Функции высшего порядка
  Именно они и дали название диалекту — как известно, вложенные безымянные функции на жаргоне называются лямбдами. Множество допустимых символов Рефала-5 было дополнено символом-замыканием, который может представлять собой как ссылку на глобальную именованную функцию, так и объект безымянной функции.

- Блоки
  Блок (и даже последовательность блоков) допустим после любого результатного выражения, в том числе и в условии тоже.

- Присваивание
  Присваивание, в отличие от классического условия, записывается через знак = (вместо ,) и не допускает отката в левую часть или к следующему предложению. Для предложения

```
1 PatA , ResB : PatC = ResD : PatE ,
2   ResF : PatG = ResH;
```

неуспех сопоставления в PatC произведёт откат к PatA, либо к следующему предожению. Неуспех в PatE аварийно остановит программу. Неуспех в PatG откатится либо к PatE, либо (если PatE не допускает другого сопоставления) тоже аварийно остановит программу.
Присваивания — тоже синтаксический сахар, они эквивалентны блокам из одного предложения.
Основное преимущество присваивания перед условием в роли присваивания — эффективность выполнения на списковой реализации. При построении результатной части условия обязательно приходится копировать переменные, поскольку при неуспехе нужно будет тот же аргумент функции передать в следующее предложение. В присваиваниях откат невозможен, а значит компилятор может (и должен) просто переносить значения переменных из аргумента.

- Сокрытие переменных
  При использовании расширенных конструкций (условия, блоки, присваивания) часто в одном из подчинённых образцов строится новая сущность, по смыслу эквивалентная предыдущей. И при этом предыдущая сущность уже становится ненужной. Логично ей дать то же имя переменной, но синтаксис классического Рефала-5 не позволит это сделать — переменная станет повторной и должна будет иметь точно такое же значение.

- Статические ящики
  Статический ящик — это функция, которая при вызове возвращает предыдущий аргумент своего вызова (на первом вызове возвращают пустоту). Иначе говоря, это некоторая глобальная переменная, которая хранит объектное выражение. Его чтение выполняется одновременно с записью — вызов статического ящика как функции возвращает значение, которое в нём хранилось и при этом устанавливает новое.
  Синтаксически это оформляется при помощи директив $SWAP — статический ящик как локальная функция и $ESWAP — как

entry-функция (можно обратиться в других единицах трансляции при помощи $EXTERN).

- Пустые функции

- Абстрактные типы данных
  Они же именованные скобки. Они же квадратные скобки. Они же инкапсулированные скобки. Это в некотором смысле разновидность структурных скобок, только (а) они квадратные, (б) после [обязательно должно располагаться имя функции.
  Если функцию, которая пишется после [, определить как локальную, то содержимое данного скобочного терма будет доступно только в той единице трансляции, где он создан (в других файлах невозможно будет на эту локальную функцию сослаться по имени). В других единицах трансляции на данный терм можно ссылаться только как на t-переменную.
  Для объявления такой функции-тега удобно воспользоваться ключевым словом $ENUM.

- Нативные вставки
  реализация Рефала-5 открыта — можно добавить в язык новые возможности (работа с сетью, с базами данных) не меняя исходную реализацию. Язык допускает т.н. «нативные вставки» (native insertions) — вставки кода на языке C++. Выглядит это так:

```
1 %%
2 #include <stdio.h>
3
4 namespace {
5   int g_next_number = 0;
6 };
7 %%
```

- Включение файлов
  Язык поддерживает ключевое слово $INCLUDE, позволяющее,

по аналогии с C++, включать в текущую единицу трансляции содержимое другого текстового файла (он должен иметь расширение .refi). После ключевого слова должно располагаться имя файла в виде составного символа в кавычках.
$INCLUDE "LibraryEx";

3 IntelliJIDEA плагин

Теперь опишем реализацию самого плагина. Написание плагина любого языка для IntelliJIDEA понятен благодаря их документации [2]. Это обеспечивает достаточно информации о том, как реализовать некоторые из основных функций. Однако, в некоторых случаях, пытаясь достичь чего-то, что не было описано в этом руководстве, необходимо было прочитать код реализации языка Java или иногда просто отлаживать код IntelliJ, чтобы увидеть, как он выполняется и как управлять потоком таким образом, чтобы получить желаемый результат. Первым делом для написания плагина любого языка нужно определить парсер и лексер.

3.1 Синтаксический анализ

3.1.1 BNF грамматика языка Рефал-5λ и PSI дерево

Задача парсера— взять токен и построить абстрактное синтаксическое дерево. Мы используем библиотеку Grammar-Kit [4] для того, чтобы сгенерировать парсер. Grammar-Kit был специально разработан для написания плагинов для пользовательских языков для IntelliJIDEA. Поэтому парсер который генерируется с её помощью отлично интегрирован в архитектуру IntelliJIDEA.

BNF грамматика описана в файле RefalFiveLambda.bnf. Для начала нам понадобиться описать из чего состоит входной файл— он может состоять из нескольких элементов, которые в свою очередь могут быть одним из перечисленного в строках 3-11

```
1 Program ::= ProgramElement*
2 ProgramElement ::=
3     IncludeDec
4   | externalDeclaration
5   | enumDefinition
```

```
6     | swapDefinition
7     | forwardDeclaration
8     | SimpleFunction
9     | NativeIns
10    | KeywordFunction
11    | SEMICOLON
```

После рассмотрим каждое из этих элементов. Стоит обратить особое внимание как разбирается block.

```
1  Block ::= LBRACE (Sentences)? RBRACE
```

Чтобы плагин не ругался при создании пустой функции Sentences могут быть, а могут и отсутствовать.

```
1  Sentence ::= sentenceCorrect | sentenceRecover
2  private sentenceCorrect::=Pattern CondAssignment*
3  ( EQUAL ResultEx | COMMA ResultEx )
4  private sentenceRecover ::= patternRecover
5  patternRecover ::= PatternTerm+
6  Pattern ::= PatternTerm*
```

Разбиение на правильные и неправильные предложения необходимо для того, чтобы было восстановление при ошибке.

```
1  PatternTerm ::=
2       Atom
3       | patternTermParens
4       | patternTermBrackets
5       | RedefinitionVariable
6       | AssignmentCorrect
7       | ConditionCorrect
```

```
1  Var ::= VARIABLE
2  Identifier ::= NAME
3  FuncPtr ::= AMPERSAND FuncName
```

```
FuncName ::= NAME
```

Такое описание нужно для того, чтобы была возможность автодополнения. Мы будем искать FuncPtr и Var.

На следующих двух рисунках показано какое получается дерево.

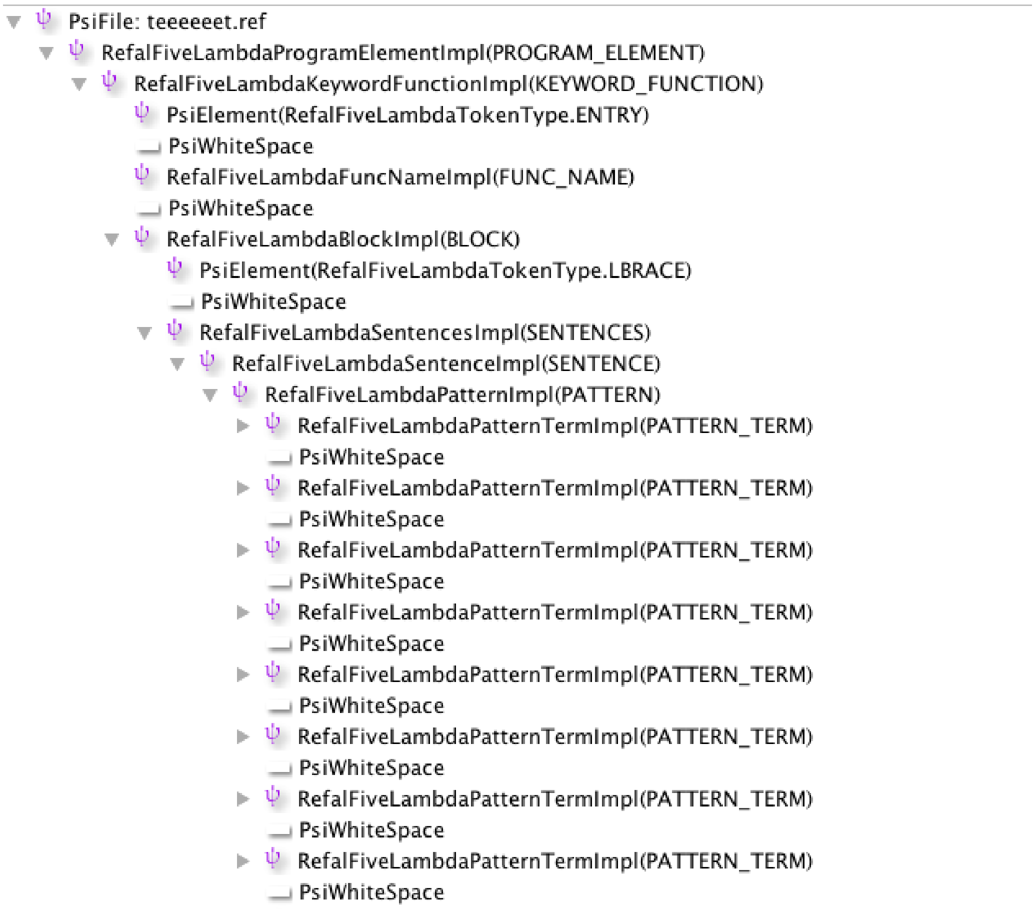

Рис. 1: дерево программы на Рефал-5λ

Рис. 2: дерево программы на Рефал-5λ

3.2 Лексический анализ

Второй немало важный компонент нашего будущего плагина— лексер. Наш плагин используют библиотеку JFlex [3] для генерации лексера из описанной спецификации. Наш лексер описан в файле RefalFiveLambda.flex. Чтобы библиотека поняла, куда нам нужно добавить лексер и с чем он должен быть связан, укажем в начале файла flex следующее:

```
1 import com.intellij.lexer.FlexLexer;
2 import com.intellij.psi.tree.IElementType;
3 import ru.tereshkina.plugin.psi.RefalFiveLambdaTypes
    ;
```

```
4 import com.intellij.psi.TokenType;
```

Задача лексера взять строку кода и выходной список токенов, которые в последствии будут использоваться парсером.

```
1 CRLF= \n|\r|\r\n
2 WHITE_SPACE=[\ \t\f]
3 FIRST_NAME_CHAR=[a-zA-Z]
4 NAME_CHAR=[a-zA-Z_\-0-9]
5 VARIABLE_TYPE = "s"|"t"|"e"
6 DIGIT=[0-9]
7 DECIMAL_INTEGER_LITERAL={DIGIT}+
8 INTEGER_LITERAL={DECIMAL_INTEGER_LITERAL}
```

Здесь описаны регулярные выражения всего, что мы используем в парсере.

```
1  <YYINITIAL> {
2
3      "$ENUM"
4      {return RefalFiveLambdaTypes.ENUM;}
5      "&"
6      {return RefalFiveLambdaTypes.AMPERSAND;}
7
8      {VARIABLE_TYPE}"."{NAME_CHAR}+
9      { return
10     RefalFiveLambdaTypes.VARIABLE; }
11     {FIRST_NAME_CHAR}{NAME_CHAR}*
12     { return RefalFiveLambdaTypes.NAME;}
13     {MULTILINE_COMMENT}
14     { return
15     RefalFiveLambdaTypes.MULTILINE_COMMENT;}
16     {END_OF_LINE_COMMENT}
17     { return
18     RefalFiveLambdaTypes.END_OF_LINE_COMMENT;}
19     {STRING_LITERAL}
```

```
20      { return
21      RefalFiveLambdaTypes.QUOTEDSTRING; }
22      ({CRLF}|{WHITE_SPACE})+
23      { return TokenType.WHITE_SPACE; }
24      {CPP_INLINE}
25      {return RefalFiveLambdaTypes.NATIVE_IN;}
26      .       { yybegin(YYINITIAL);
27        return TokenType.BAD_CHARACTER; }
28  }
```

Также как в строке 3 объявим все ключевые слова, и как в 4 строке— все специальные символы.

Тогда, например, простой символ двоеточие будет конвертироваться в токен SEMICOLON, тогда как последовательность "135"будет конвертироваться в один токен типа
INTEGER_LITERAL. Часть <YYINITIAL> в начале блока означает, что правила внутри блока будут сопоставляться только в том случае, если текущее состояние лексера— YYINITIAL (это состояние по умолчанию для лексера). Состояние может быть изменено по правилам, что означает, что можно иметь лексер с сохранением состояния.

Стоит также отметить, что все, что не попадает под описание в flex-файле будет конвертироваться как BAD_CHARACTER.

Важно отметить, что наш лексер не так строг, как лексер, используемый компилятором RefalFiveLambdareport. Например, следующее— это регулярное выражение, которое мы используем для сопоставления строковых литералов:

```
1  STRING_LITERAL=\'([^\\\'\r\n]|
2    {ESCAPE_SEQUENCE})*\'
```

Это не только делает окончательную кавычку необязательной, но также принимает произвольные escape-последовательности в строке,

что компилятор RefalFiveLambda будет воспринимать как синтаксическую ошибку. Например, строковый литерал «foo d bar» успешно распознается нашим лексером, но отклоняется компилятором с сообщением об ошибке «неверная символьная константа». Эта снисходительность предназначена, поскольку она позволяет нам все еще анализировать файл и собирать информацию о его структуре, даже если в коде есть тривиальные синтаксические ошибки.

3.3 Подсветка синтаксиса

Спецификация лексера обрабатывается JFlex как часть процесса сборки, и создается класс с именем RefalFiveLambdaLexer. Используя этот класс, можно затем превратить строку в поток токенов. Даже не написав синтаксический анализатор, мы уже можем использовать лексер для реализации базовой подсветки синтаксиса. Все, что требуется,— это создать класс, расширяющий SyntaxHighlighterBase, и переопределить два метода— один, который возвращает экземпляр лексера, и другой, который сопоставляет тип токена с набором текстовых атрибутов, которые описывают цвет и стиль шрифта для рендеринга токенов этого типа. , Наша подсветка синтаксиса реализована классом RefalFiveLambdaSyntaxHighlighter, который наследуется от SyntaxHighlighterBase.

```
1  public class RefalFiveLambdaSyntaxHighlighter
2  extends SyntaxHighlighterBase {
3   static final TextAttributesKey RFL_KEYWORD =
4    createTextAttributesKey("REFAL_FIVE_LAMBDA_KEYWORD
       ",
5    DefaultLanguageHighlighterColors.KEYWORD);
6
7   private static Set<IElementType>
8   KEYWORD_ELEMENTS = new HashSet<>();
9   static KEYWORD_ELEMENTS.add(RefalFiveLambdaTypes.
      EXTERN);
10
```

```
11   public TextAttributesKey[]
12   getTokenHighlights(IElementType type) {
13     if (type.equals(RefalFiveLambdaTypes.VARIABLE))
14       return pack(RFL_VARIABLE);
15     if (type.equals(RefalFiveLambdaTypes.SEMICOLON))
16       return pack(RFL_SEMICOLON);
17     if (KEYWORD_ELEMENTS.contains(type))
18       return pack(RFL_KEYWORD);
19     return new TextAttributesKey[0];
20       }
21 }
```

На рисунке 3 изображен файл RefalFiveLambda, открытый в IntelliJ, с выделением синтаксиса.

```
$ENTRY Go {
s.X 1 = <s.X 1  s.X
         {=:,#= }
         (a)
         [Go 1]
&Dg  <Q>
         () [a 1] "jkljlj" 'zxcv' 3 4>   //kjfn

/**dkjfnv*/
  }

$ENTRY Native {
%%
return 0;
%%
}

  $ENTRY GL {
    = <Prout 'Hello'>
      <Prout 1 22 333 4444>
      <Prout "Hello, World!">
      <Prout Hello World>
      <Prout ((() ()))>
      <Prout (Hello 1 'Hello')>
      <Prout >
      <G lerf>
      }
$ENUM a;
```

Рис. 3: Рефал-5λ с подсветкой синтаксиса

3.4 Интеллектуальное автодополнение

IntelliJ поддерживает два способа реализации автодополнения кода— справочное завершение, которое в большинстве случаев является более простым и достаточным, и завершение на основе участия, contributor, которое должно обеспечивать более детальный контроль над тем, как и когда вызывается завершение кода. Мы используем именно его. Для этого необходимо реализовать методы, которые будут определять тип того что нам нужно, и добавлять их в ArrayList. Рассмотрим на примере автодополнения элементов типа VAR. Создадим класс RefalFiveLambdaUtils, который включает в себя все необходимые функции. Для начала нам необходимо находить все элементы типа VAR, для этого реализуем следующее.

```
1  private static String[] getVariablesRec(PsiElement
       top)
2  {
3   PsiElement[] aChildren = top.getChildren();
4   List<String> lChildVariables = new ArrayList<>();
5   for (PsiElement child : aChildren) {
6    if (child.toString()
7      .equals("RefalFiveLambdaVarImpl(VAR)"))
8     lChildVariables.add(child.getText());
9    else if (child.getChildren().length > 0) {
10    String[] result = getVariablesRec(child);
11    Collections.addAll(lChildVariables, result);
12   }
13  }
14  String[] aChildVariables =
15   new String[lChildVariables.size()];
16  return lChildVariables
17   .toArray(aChildVariables);
18 }
```

Функция получает на вход элемент дерева. После находит всех детей этого элемента и для каждого из них проверяет к какому типу PSI

элементов оно относиться. В случае, если это VAR мы добавляем его в список.

```
1  public static String[]
     getPredecessorPatternVariables
2   (PsiElement element, boolean
     includePredecessorPattern) {
3   PsiElement top = element;
4   List<String> lVariables = new ArrayList<>();
5   boolean isMoreThanPredecessor =
     includePredecessorPattern;
6
7   while (top.getParent() != null) {
8    if (isSentence(top.getParent())) {
9     boolean isPattern = top.toString()
10     .equals("RefalFiveLambdaPatternImpl(PATTERN)");
11    boolean isAssignment = top.toString()
12     .equals("RefalFiveLambdaAssignmentCorrectExImpl(
13 ________ASSIGNMENT_CORRECT)");
14    if (!isPattern || isMoreThanPredecessor) {
15     String[] tempVariables =
16      getVariablesRec(top.getParent().getFirstChild()
         );
17     Collections.addAll(lVariables,tempVariables);
18     isMoreThanPredecessor = true;
19    }
20   }
21   top = top.getParent();
22  }
23  String[] aVariables =
24   new String[lVariables.size()];
25  return lVariables.toArray(aVariables);
26 }
```

Выше представленный фрагмент кода необходим для того, чтобы

мы получали нужные нам значения. После этого реализуем класс RefalFiveLambdaCompletionContributor, наследующийся от CompletionContributor. В конструкторе этого класса опишем как нам нужно дополнять необходимые нам значения VAR.

```
1  public RefalFiveLambdaCompletionContributor()
2  { extend(CompletionType.BASIC,
3    PlatformPatterns.psiElement(RefalFiveLambdaTypes.
       VARIABLE).
4    withLanguage(RefalFiveLambdaLanguage.INSTANCE),
5    new CompletionProvider<CompletionParameters>() {
6     public void addCompletions
7     (@NotNull CompletionParameters parameters,
8     ProcessingContext context,
9     @NotNull CompletionResultSet resultSet) {
10     String[] predecessorVariables =
11     RefalFiveLambdaUtils.
         getPredecessorPatternVariables
12     (parameters.getPosition().getParent(), true);
13     String[] siblingPatternVariables =
14      RefalFiveLambdaUtils.getSiblingPatternVariables
15      (parameters.getPosition().getParent());
16     String[] potentials =
17      (String[]) ArrayUtils.addAll
18      (predecessorVariables, siblingPatternVariables)
          ;
19     if (potentials.length > 0) {
20      for (String var : potentials) {
21       resultSet.addElement
22        (LookupElementBuilder.create(var));
23       }
24      }
25     }
26    }
27  );
```

Для функций сделаем по такому же принципу.

3.5 Подсветка ошибок

Перейдем к заключительной части.
Это является одной из самых простых задачей. В выше описанном классе RefalFiveLambdaUtils мы добавим функции проверки типов. Рассмотрим все на том же VAR

```
1  public static boolean isVar
2   (PsiElement element) {
3    return element.toString().equals(
4     "RefalFiveLambdaVarImpl(VAR)");
```

После необходимо создать класс RefalFiveLambdaAnnotator реализующий Annotator, в котором реализуем метод annotate, одним из входнх элементов которого является psiElement
element.

```
1  public void annotate
2  (@NotNull PsiElement psiElement,
3   @NotNull AnnotationHolder annotationHolder) {
4    if (RefalFiveLambdaUtils.isVar(psiElement)) {
5     String[] potentialVariables =
6     RefalFiveLambdaUtils.
          getPredecessorPatternVariables
7      (psiElement, false);
8     boolean isPatternVariable =
9     RefalFiveLambdaUtils.isPatternVariable
10     (psiElement);
11    boolean isInPotentialVariables =
12    Arrays.asList(potentialVariables)
13     .contains(psiElement.getText());
14    if (!isPatternVariable && !isInPotentialVariables
        ) {
```

```
15        annotationHolder
16        .createErrorAnnotation(
17          RefalFiveLambdaUtils.getTextRange(
18            psiElement),
19          "Unresolved variable");
20      }
21    }
```

Здесь в строке два мы как раз проверяем element на соответствие типу. Если он не соответствует типу, которому должен соответствовать по дереву и нет того, чем можем дополнить (т. е. не является префиксом существующих) появляется красная подчеркивание с надписью «Unresolved variable».

4 Тестирование

Проверим работоспособность на фрагменте компилятора нашего языка и простой программе на Рефал-5λ.

```
$ENTRY GenerateGroupWithGCGSubst {
        e.ONE s.FnGenSubst s.FnGenResult
        (e.MarkedPattern) (e.SentencesWithSubst) s.ContextSize s.BaseNum s.Depth
        e.NewGCG-Subst

          = <EnumerateVars-Subst s.BaseNum s.Depth e.NewGCG-Subst>
          : e.TO s.BaseNum e.NewGCG-Subst^

          = <CodeForSubst
              s.FnGenSubst s.ContextSize (e.MarkedPattern) e.NewGCG-Subst s.Depth>
          : e.THREE s.ContextSize^ (e.Vars) (e.MarkedPattern^) e.GCG-Commands

          = <NarrowSentenceSubstitutions (e.NewGCG-Subst) e.SentencesWithSubst>
          : (e.FOUR) e.SentencesWithSubst

          = <FindDivisionAndGenerate
              s.FnGenSubst s.FnGenResult
              (e.MarkedPattern)
              (e.SentencesWithSubst)
              s.ContextSize s.BaseNum s.Depth
            >
          : e.FIUE s.ContextSize^

          = s.ContextSize e.NewGCG-Subst e.THREE e.GCG-Commands;

      }
```

Рис. 4: Входной файл 1

```
$ENTRY Go {
s.X 1 = <s.X 1  s.X
         {=:,#= }
         (a)
         [Go 1]
&Dg  <Q>
         () [a 1] "jkljlj" 'zxcv' 3 4>   //kjfn

/**dkjfnv*/
  }

$ENTRY Native {
%%
return 0;
%%
}

  $ENTRY GL {
    = <Prout 'Hello'>
      <Prout 1 22 333 4444>
      <Prout "Hello, World!">
      <Prout Hello World>
      <Prout ((() ()))>
      <Prout (Hello 1 'Hello')>
      <Prout >
      <G lerf>
      }
$ENUM a;
```

Рис. 5: Входной файл 2

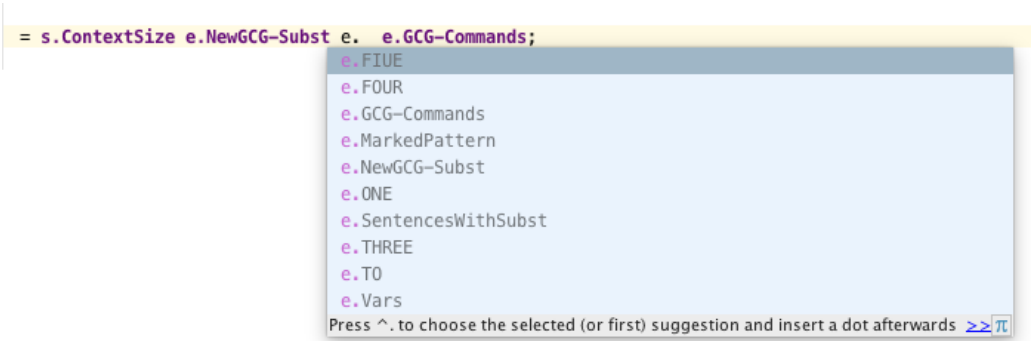

Рис. 6: Автодополнение Variable

Рис. 7: Автодополнение функциями описанными в файле

Рис. 8: Автодополнение встроенными функциями

5 Заключение

В ходе написания курсовой работы был разработан плагин языка Рефал-5λ для IntelliJIDEA, поддерживающий подсветку синтаксиса и «умное» автодополнение. На основе результатов полученных при тестировании можно сделать следующий вывод. Реализованный плагин готов к боевым условиям.

Список литературы

[1] Документация языка Рефал-5λ,
`github.com/bmstu-iu9/refal-5-lambda`

[2] Туториал пользовотельского языка,
`https://www.jetbrains.org/intellij/sdk/docs`

[3] JFlex,
`http://jflex.de/`

[4] JFlex,
`https://github.com/JetBrains/Grammar-Kit`