САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
УНИВЕРСИТЕТ
Математико-механический факультет

Кафедра системного программирования

Иванов Андрей Васильевич

Поддержка встроенных языков в интегрированных средах разработки

Бакалаврская работа

Допущена к защите.
Зав. кафедрой:
д. ф.-м. н., профессор Терехов А. Н.

Научный руководитель:
ст. преп. Григорьев С. В.

Рецензент:
Руководитель группы в ООО “ИнтеллиДжей Лабс” Шкредов С. Д.

Санкт-Петербург
2014

SAINT-PETERSBURG STATE UNIVERSITY
Mathematics & Mechanics Faculty

Chair of Software Engineering

Ivanov Andrei

Support of string-embedded languages in IDE

Graduation Thesis

Admitted for defence.
Head of the chair:
professor Terekhov A. N.

Scientific supervisor:
sen. lect. Grigoriev S. V.

Reviewer:
Team lead at "IntelliJ Labs Co.Ltd" Shkredov S. D.

Saint-Petersburg
2014

Оглавление

Введение

Программы, написанные на современных языках программирования, во время выполнения могут формировать выражения на других языках и выполнять их. Языки, которые используются для написания таких динамически формируемых выражений, называются встроенными.

Несмотря на то что сегодня широко распространено использование таких технологий, как ORM (object-relational mapping), есть области, в которых встроенные языки встречаются достаточно часто. В качестве примера можно привести реинжиниринг программного обеспечения.

Ниже приведен пример использования встроенных языков.

```php
<?php
// Embedded SQL
$query = 'SELECT * FROM my_table';
$result = mysql_query($query);
// HTML markup generation
echo "<table>\n";
while ($line = mysql_fetch_array($result, MYSQL_ASSOC)) {
  echo "\t<tr>\n";
  foreach ($line as $col_value) {
    echo "\t\t<td>$col_value</td>\n";
  }
  echo "\t</tr>\n";
}
echo "</table>\n";
?>
```

Рис. 1: Использование нескольких различных встроенных в PHP языков (MySQL, HTML)

При работе с приложениями, содержащими встроенные языки, необходимо помнить, что порождаемые строки тоже могут являться кодом на некотором языке программирования. Поэтому важно сохранить синтаксическую корректность динамически формируемых выражений.

Одна из основных трудностей в работе со встроенными языками заключается в том, что часто отсутствуют средства разработки, позволяющие определить правильность составленного динамического выражения статическим образом, то есть до запуска основной программы. Это связано с тем, что компилятором такие выражения воспринимаются как простые строки.

С другой стороны, сегодня при создании ПО активно используются интегрированные среды разработки - IDE (Integrated Development Environment), которые предо-

ставляют набор различной функциональности, позволяющей значительно упростить процесс создания программ. Примером могут служить такие функции, как автодополнение, рефакторинг (улучшение написанного ранее кода, не влияющее на его внешнее поведение), дополнительные статические проверки, подсказки.

Такие возможности современных IDE были бы полезны и для встроенных языков, поскольку это позволило бы сократить время на создание, отладку и сопровождение приложений, использующих динамически формируемые выражения. К примеру, подсветка синтаксиса могла бы сигнализировать о правильно набранной синтаксической конструкции или же о допущенной опечатке.

В данной работе представлено механизм поддержки произвольных встроенных языков, описанных грамматикой, для среды разработки Microsoft Visual Studio [7], а также описание реализации этого механизма.

1. Постановка задачи

Целью данной работы является реализация механизма поддержки произвольных встроенных языков в Microsoft Visual Studio, которые описаны грамматикой. Для ее достижения были поставлены следующие задачи.

- Изучить инструмент YaccConstructor.
- Изучить платформу ReSharper SDK.
- Создать алгоритм подсветки встроенных языков.
- Реализовать и апробировать полученный алгоритм.

2. Обзор

2.1. Обзор существующих инструментов

На данный момент существует ряд инструментов, которые могут работать со встроенными языками.

Они кратко описаны ниже.

- PhpStorm [9] — интегрированная среда разработки для PHP, которая осуществляет подсветку и автодополнение встроенного кода на HTML, CSS, JavaScript, SQL. Но эта поддержка осуществляется не для всех строчек. На рисунке 2 правые части присваиваний переменных $hello1 и $hello2 “распознаны” и подсвечены как выражения на языке HTML. Однако про переменную $hello3 такого сказать нельзя.

```php
<?php
    $hello1 = "<html><body>Hello, world!</body></html>";

    $hello2  = "<html>";
    $hello2 .= "<body>";
    $hello2 .= "Hello, world!";
    $hello2 .= "</body>";
    $hello2 .= "</html>";

    $hello3 = "<html>"."<body>"."Hello, world!"."</body>"."</html>";

    echo $hello1;
    echo $hello2;
    echo $hello3;
?>
```

Рис. 2: Пример поддержки встроенных языков в PHPStorm.

- IntelliLang [6] - плагин к средам разработки PHPStorm и IntelliJ IDEA [5], с помощью которого для каждого строкового выражения можно указать, на каком языке оно написано. Например, IntelliLang для PHPStorm поддерживает различные диалекты языков запросов к базам данных. К недостаткам плагина следует отнести то, что для каждого строкового выражения нужно указывать язык явным образом.

- В статье Hyunha Kim, Kyung-Goo Doh and David Schmidt [4] описан инструмент, осуществляющий статическую валидацию HTML-страниц, которые порождаются программами на PHP и JSP. К достоинствам инструмента является то, что валидация осуществляется как синтаксическая, так и семантическая. К примеру, этот инструмент может указывать на такие ошибки, как отсутствие обязательного атрибута, наличие двух разных элементов с одинаковым id и другие.

- Alvor [1] — это плагин к среде разработки Eclipse, который проверяет корректность встроенных SQL-выражений в код на Java. Ищет динамические выражения на SQL и проверяет их на соответствие SQL-грамматике. В случае, когда SQL-запросы содержат синтаксические и семантические ошибки, Alvor подчёркивает соответствующие места в исходном коде и выводит информацию об ошибках до запуска программы.

- Java String Analyzer [2] — это инструмент для анализа формирования строковых выражений на Java. Для каждого такого выражения он строит конечный автомат, представляющий приближённое значение всех значений этого выражения, которые могут быть получены во время исполнения.

- PHP String Analyzer [8] — это инструмент для статического анализа строк, порождаемых PHP. Он аппроксимирует значения таких строк контекстно-свободной грамматикой. Это может быть использовано, например, для валидации генерируемых программами на PHP Web-страниц.

Все приведённые выше инструменты, за исключением PHPStorm, предназначены для валидации динамически формируемых выражений, но не решают задачи подсветки синтаксиса встроенных языков. PHPStorm и IntelliLang (как вместе, так и по отдельности) решают эту задачу, но не для всех языков. Хотелось бы получить инструмент, который может осуществлять подсветку синтаксиса произвольного встроенного языка, описанного грамматикой.

2.2. Обзор используемых инструментов

2.2.1. YaccConstructor

На кафедре системного программирования математико-механического факультета СПбГУ разрабатывается инструмент YaccConstructor [13] [12], позволяющий создавать генераторы синтаксических анализаторов под .NET. Основным языком разработки является мультипарадигмальный язык F# [11]. Этот инструмент имеет модульную структуру, которая позволяет создавать анализаторы с различными алгоритмами разбора. Один из таких модулей создан для работы со встроенными языками. В рамках этого модуля уже реализованы алгоритмы абстрактного лексического и синтаксического анализа. “Абстрактный анализ” [3] здесь означает, что в качестве входа анализатор принимает не линейный поток токенов, а некую структуру.

Рассмотрим работу абстрактного синтаксического анализатора чуть подробнее.

- На вход анализатору подаётся граф токенов (а не линейная последовательность, как в классическом анализе). Все токены образуют алгебраический тип данных Token, который создаётся генератором. Помимо этого каждый токен содержит

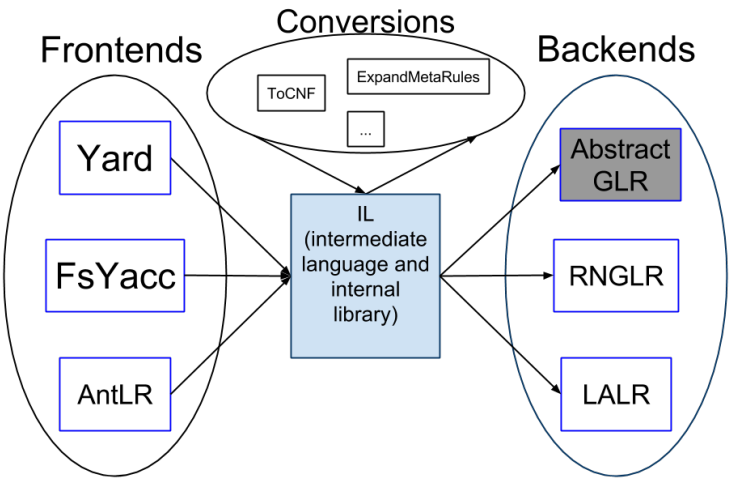


Рис. 3: Архитектура YaccConstructor

в себе некоторую информацию: координаты в исходном файле (что необходимо при подсветке синтаксиса или же при уведомлении об ошибках).

- По входному графу токенов строится SPPF [10] (Shared Packed Parsed Forest) - структура, которая позволяет хранить все деревья разбора в сжатом виде.
- При необходимости можно осуществлять трансляцию деревьев.

2.2.2. ReSharper SDK

Компания JetBrains разработала ReSharper - плагин к Microsoft Visual Studio, который позволяет повысить продуктивность работы, осуществляя дополнительные статические проверки и предоставляя дополнительные средства автодополнения, навигации, поиска и т.п. У этого плагина есть свой SDK (Software Development Kit).

Архитектура состоит из нескольких модулей. Рассмотрим некоторые из них.

- Platform - модуль, осуществляющий интеграцию с MS Visual Studio. Содержит в себе подмодули для работы с файлами solution, с файлами проектов, с файлами с исходным кодом.
- PSI (Project Structure Interface) - модуль, занимающийся структурой программы: лексический и синтаксический анализ языков, которые поддерживается ReSharper'ом. Содержит в себе компоненту, позволяющую добавлять поддержку новых языков. В данной работе используется PSI модуль используется для получения структуры программы на основном языке.
- Daemon. Сборки Daemon являются работающими в фоновом режиме потоками, которые анализируют исходный код и реагируют на различные изменения в нём.

При запуске программы ReSharper запускает свыше трёхсот сборок, которые осуществляют разнообразный анализ. Для того чтобы одна из этих сборок занималась каким-то самостоятельно реализованным анализом, нужно добавить какой-нибудь атрибут к определению класса (например, [DaemonStage] или [ContextAction]) и унаследоваться от какого-нибудь ReSharper'овского интерфейса (например, IDaemonStage). И тогда при запуске программы ReSharper инстанцирует требуемый класс и будет вызывать его методы при соответствующих изменениях.

Тип этих изменений может определяться атрибутом. Например, класс, отмеченный атрибутом [DaemonStage], будет заново инстанцироваться при каждом изменении в файле с исходным кодом (например, напечатан или удалён какой-то символ). А если класс отмечен атрибутом [ContainsContextConsumer], то такая реакция будет происходить на каждое изменение положения курсора.

3. Алгоритм

3.1. SPPF

SPPF [10] — структура, позволяющая хранить все деревья разбора в сжатом виде. Идея в том, что деревья разбора имеют много одинаковых вершин. Хранить каждую из них отдельно для каждого дерева – не самая лучшая идея, поскольку это плохо скажется на памяти. SPPF позволяет куда экономнее её использовать, так как хранит в себе только один фрагмента какого-то дерева, который в свою очередь может разделяться между несколькими деревьями.

В классическом синтаксическом анализе эта структура склеивает узлы, соответствующие одинаковым нетерминалам или терминалам, в один узел, если они принимают одинаковую входную цепочку. Если существует множество различных выводов для одного нетерминала, то множество семейств всех детей разделяет одну родительскую вершину.

Рассмотрим грамматику:

[<Start>]

s : a

a : p r | u v

p : B

u : B

r : C

v : C

Эта грамматика неоднозначна, поскольку при входной цепочке “B C” будет получено два дерева разбора.

На рисунке 4 продемонстрировано, как могут быть сжаты полученные деревья.

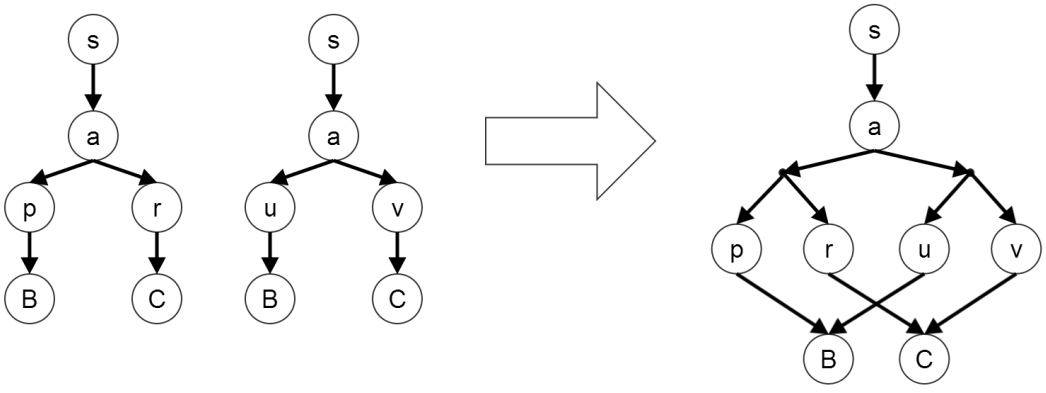


Рис. 4: Пример получения SPPF из нескольких деревьев.

Однако в абстрактном синтаксическом анализе возможна ещё одна ситуация. Предположим, что в грамматике только один стартовый нетерминал (если их несколько, то

можно добавим новое стартовое правило). Если деревья соответствуют разным путям во входном графе, то из их корневых вершин выводятся разные цепочки. Однако некоторая часть дерева, соответствующая одному или нескольким путям от корня дерева, повторяются у всех деревьев. Поэтому такие вершины тоже предлагается сливать в одну, несмотря на то что формально они могут выводить разные цепочки.

Поясним на примере такой грамматики.

[<Start>]

s : a

a : b | d

b : A B

d : A D

Предположим, что на входе у нас граф, изображённый на рисунке 5.

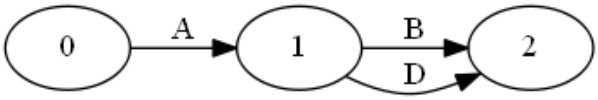


Рис. 5: Пример входного графа.

Построятся два дерева, у которых пути от вершины s до вершины a будут совпадать (см. рис. 6). Несмотря на то что в одном дереве эти вершины выводят строку “A B”, а в другом “A D”, вершины этого пути будут также склеены в одну.

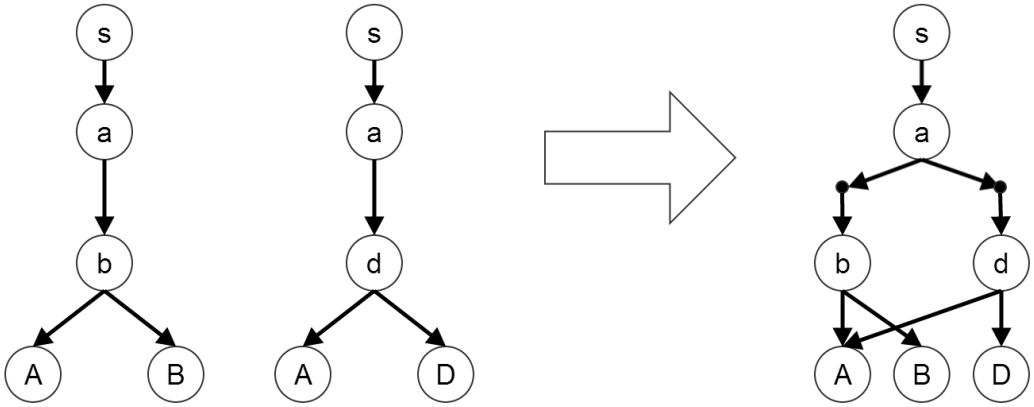
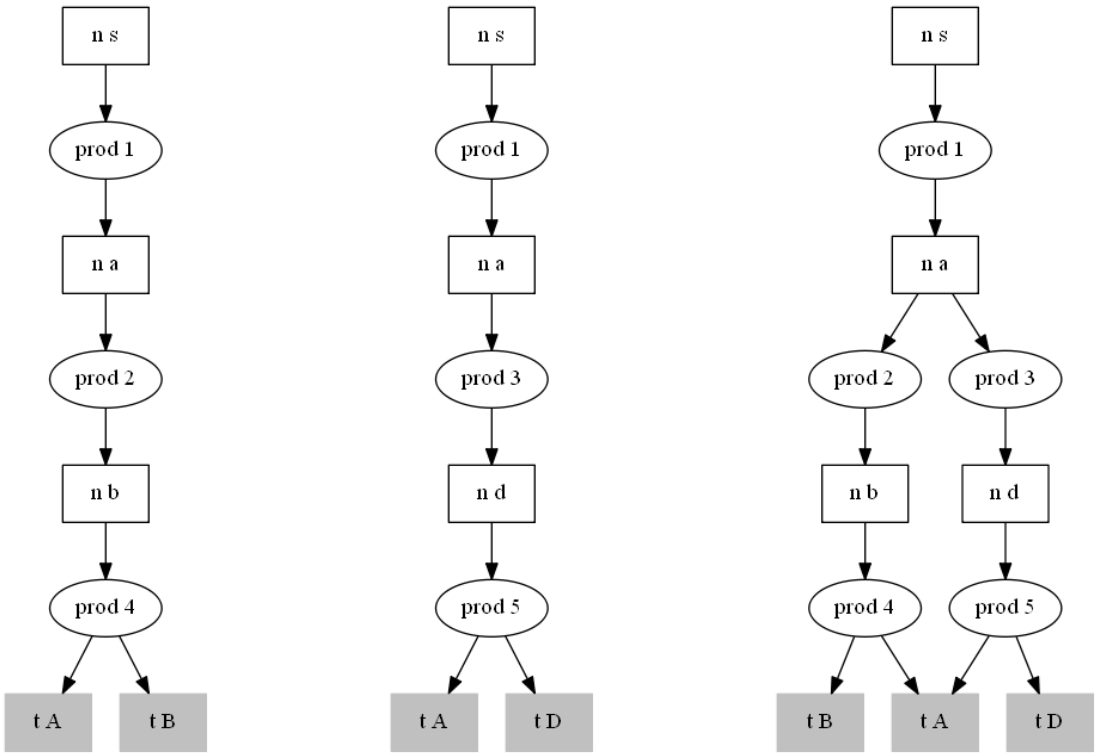


Рис. 6: Пример слияния деревьев в SPPF в абстрактном случае.

На рисунке 7 показано, как будет выглядеть SPPF для предыдущего примера в реализации YaccConstructor. Заметим, что в реализации YaccConstructor у SPPF помимо привычных узлов, соответствующих терминалам и нетерминалам (далее будем называть их “вершинами-символами”), есть ещё промежуточные узлы, которые содержат в себе номера продукции (далее будем называть такие вершины “вершинами-продукциями”).

Выпишем некоторые свойства SPPF, реализованного в YaccConstructor.



Первое дерево Второе дерево Результат слияния

Рис. 7: Слияние деревьев в SPPF в YaccConstructor.

- Корни всех деревьев соответствуют одному и тому же нетерминалу. Поэтому при склеивании деревьев будет одна вершина, в которую не входит ни одна дуга. Далее будем называть такую вершину корнем SPPF, или просто корнем.
- Все токены располагались на листьях деревьев. В SPPF токены располагаются, т.е. на вершинах, из которых не выходит ни одной дуги.
- При обходе графа от корня обнаружить вершину-символ, которую разделяют несколько поддеревьев, достаточно просто: из таких вершин выходит более одной дуги к вершинам-продукциям.

Невозможна ситуация, когда SPPF содержит “лишние деревья”, то есть деревья, для которых нет пути во входном графе. Это свойство вытекает из построения SPPF.

Пусть m различных поддеревьев склеиваются к вершине, соответствующей одинаковому нетерминалу n. Тогда по построению SPPF будет создано m вершин-продукций, к каждой из которых будет вести дуги из n. Если в дальнейшем пути от этих вершин-продукций не пересекаются, то никаких новых деревьев появиться не может.

Таким образом, “лишнее дерево” может случиться только в случае, если какие-то деревья имеют общее поддерево. Рассмотрим это поддерево поподробнее. Если оно не содержит в себе неоднозначностей (т.е. ситуации, когда из вершины-символа выходит более, чем одна дуга, к вершинам-продукциям), то мы получим ровно столько

же деревьев, сколько и разделяет это поддерево. Поэтому “лишнего” дерева в таком случае получиться не может.

Предположим, что это поддерево содержит в себе неоднозначности и при этом была порождено лишнее дерево. Рассмотрим входной граф (рис. 8).

Грамматика выглядит так:

[<Start>]

s: a | b

a : A c

b: A D c

c: B | E F

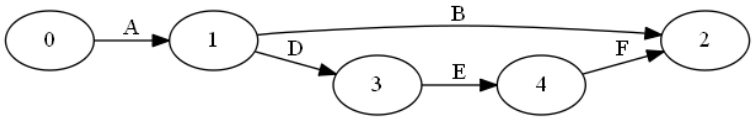
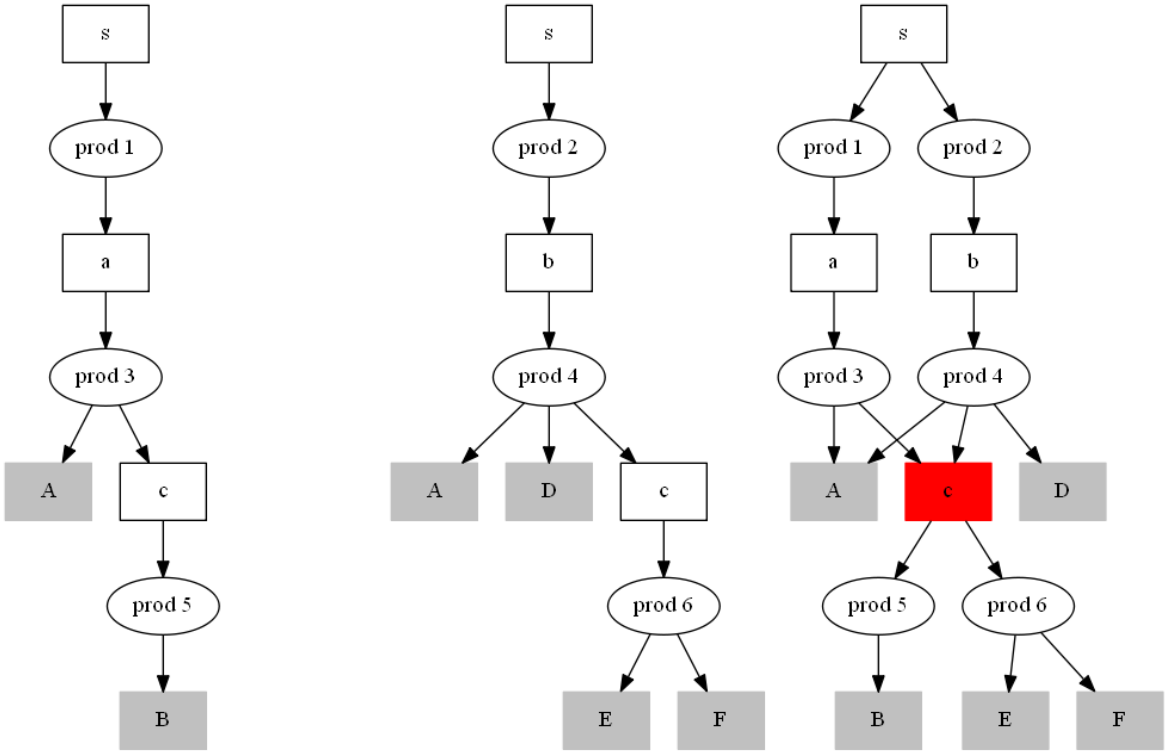


Рис. 8: Входной граф.



Первое дерево Второе дерево Слияние деревьев

Рис. 9: Неправильное слияние деревьев.

В возможном SPPF (рисунок 9) два дерева (одно порождает строку “A B”, а другое - строку “A D E F”) имеют общее поддерево, начинающееся с вершины c. Причём это поддерево содержит в себе два разных поддерева: одно порождает строку

“B”, а другое - строку “E F”. Таким образом, мы можем получить сразу четыре дерева разбора, которые выводят такие строки:

- “A B”,
- “A E F”,
- “A D B”,
- “A D E F”.

Причём только для первого и четвёртого случая существуют пути в графе.

Но граф разбора, изображённый на рисунке 9, получиться не может. Потому что нарушается требование того, что вершины сливаются в одну, если они выводят одинаковую цепочку. В данном случае это требование нарушено: в одном дереве нетерминал c выводит строку “B”, а в другом - строку “E F”. Поэтому сливать вершины, соответствующие вершинам c, в одну, нельзя. Правильный вариант SPPF изображён на рисунке 10.

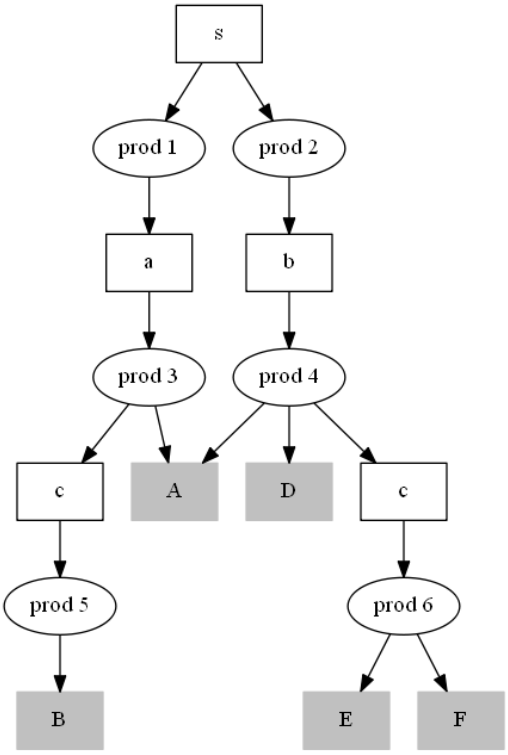


Рис. 10: Правильное слияние деревьев.

Поэтому ситуация, когда общее поддерево, содержащее неоднозначности, приводит к появлению “лишнего дерева”, невозможна.

А значит и общая ситуация, когда SPPF содержит в себе дерево, для которого нет пути во входном графе, невозможна.

3.2. Генерация деревьев из SPPF

3.2.1. Статическая подсветка

При подсветке синтаксиса встроенного языка не обязательно рассматривать все деревья. Достаточно выбрать из леса разбора, полученного синтаксического анализа, подмножество корректных деревьев таким образом, чтобы множество всех токенов (листьев) было покрыто. То есть чтобы для каждого токена существовало дерево из выбранного подмножества, которое содержит этот токен. Действительно, если у двух разных деревьев листья (токены) совпадают, то при подсветке синтаксиса не так уж и важно, какому именно дереву принадлежит тот или иной лист. Важно, что это дерево синтаксически корректно, поскольку над таким деревом можно производить другие действия (например, автодополнение, навигация по коду).

В качестве примера рассмотрим приведённую ниже грамматику и входной граф, изображённый на рисунке 11:

[<Start>]

expr: expr PLUS expr

expr: NUM

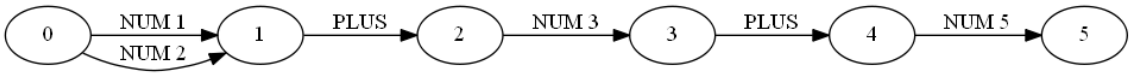


Рис. 11: Входной граф, содержащий Shift/Reduce конфликт.

Для данного графа в результате синтаксического разбора будет построено целых четыре дерева разбора, т.к. для каждого пути будет построено по два дерева (ввиду неоднозначности грамматики). Однако порождённые конфликтом Shift/Reduce деревья соответствуют одному и тому же пути в графе, а значит, множества их листьев совпадают. Поэтому в данном примере при поддержке подсветки синтаксиса для каждого пути в графе достаточно рассмотреть только одно из таких деревьев. То же самое верно и для Reduce/Reduce конфликта.

Наиболее сложной для реализации поддержки языков является, соответствующая ветвлению во входном графе. Сложность заключается в том, что в худшем случае для покрытия всех токенов придётся рассматривать все возможные деревья разбора. На рисунке 12 продемонстрирован пример такого входного графа.

Однако поскольку в вопросе подсветки деревьев скорость является одним из самых критичных, то принято решение осуществлять генерацию деревьев ленивым образом. Это позволит сэкономить время и память.

Для того чтобы описать алгоритм извлечения деревьев из SPPF, введём следующие обозначения.

- OutEdges (v) – количество исходящих дуг из вершины v.

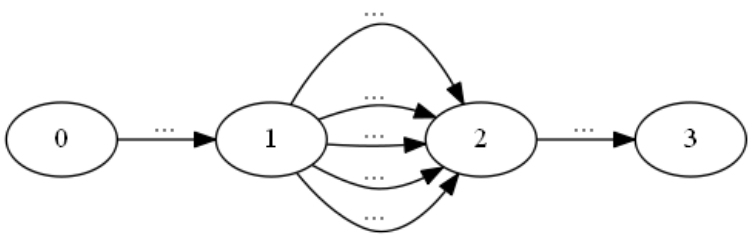


Рис. 12: Входной граф, в котором необходимо рассмотреть все деревья разбора.

- Succ (v) – $\{u \mid$ существует дуга из вершины v в вершину u$\}$.
- Tokens (v) – количество токенов, которые принадлежат графу с корнем v.

На вход алгоритму подаётся граф разбора и множество *unprocessed* - множество токенов, которые нужно посетить. *root* = корень графа разбора. Все вершины окрашены в белый цвет.

Вернуть GetTree (root, unprocessed).

GetTree (v, unprocessed):

- Если unprocessed = $\emptyset$, то вернуть пустое дерево (null).
- Иначе вернуть Handle (v, unprocessed).

Handle (v, unprocessed):

1. Если вершина v - вершина-символ, то
   - Если OutEdges (v) == 0, то это значит, вершина v соответствует какому-то токену t. unprocessed = unprocessed $\setminus\{t\}$;
   - Если OutEdges (v) == 1, то Handle (Succ (v), unprocessed).
   - Если OutEdges (v) $>$ 1, то $s = arg\ max_{u \in Succ(v)}$ | Tokens (u) $\cap$ unprocessed |. Handle (s, unprocessed). // т.е. идём по тому поддереву, которое содержит в себе наибольшее количество токенов из множества unprocessed.

   Перейти к шагу 3.
2. Если текущая вершина - вершина-продукция, то для каждой вершины $u \in$ Succ(v) сделать следующее: Handle (u, unprocessed). Перейти к шагу 3.
3. Покрасить v в чёрный цвет. Если v == root, то перейти к шагу 4.
4. Алгоритм завершает работу. Вернуть дерево, состоящее из “чёрных” вершин, и множество unprocessed.

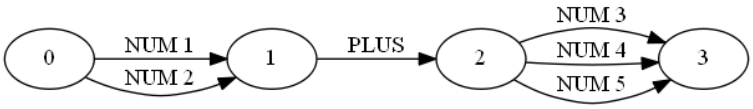


Рис. 13: Пример входного графа.

Рассмотрим работу алгоритма на примере входного графа, изображённого на рисунке 13.

Граф разбора для такого входа будет выглядеть следующим так, как показано на рисунке 14.

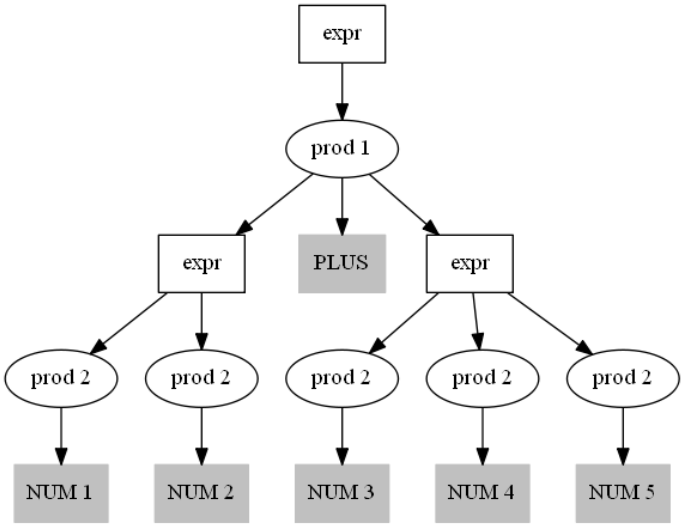


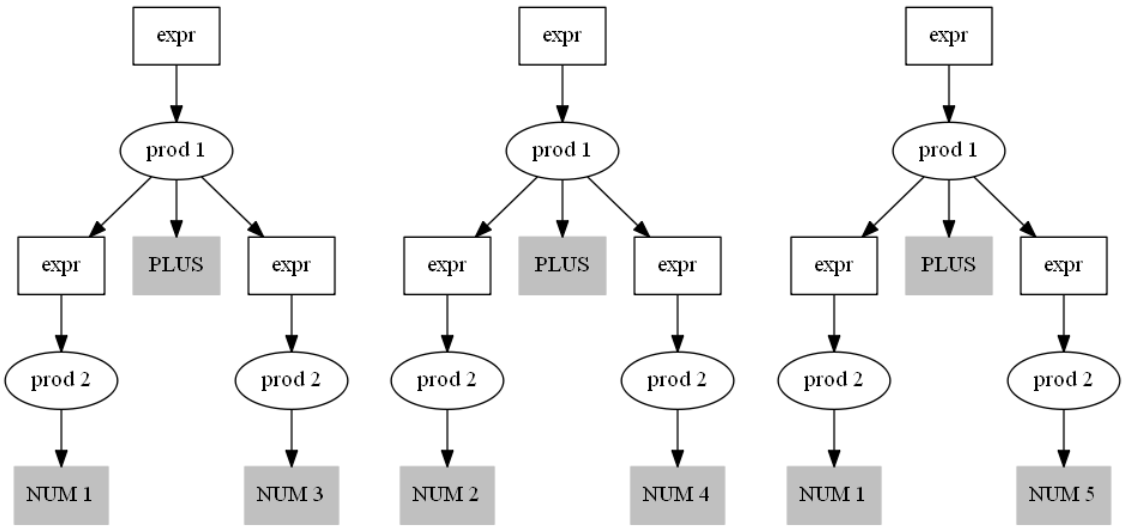
Рис. 14: SPPF для предыдущего примера.

При генерации первого дерева из этой структуры у нас все токены считаются непокрытыми. Поэтому алгоритм на вход принимает множество, состоящее из всех токенов. Поскольку любое из шести возможных деревьев разбора содержит три токена, то алгоритм вернёт любое из них. Предположим, что он вернул дерево, изображённое на рисунке 15a.

Также вместе с этим деревом вернётся множество, состоящее из токенов NUM 2, NUM 4 и NUM 5. Это множество передаётся алгоритму при втором запуске.

На основе этой информации алгоритм снова начинает обход графа с “корня”. На этот раз существует два дерева, которые содержат в себе два “новых” токена (т.е. токена из переданного на вход множества), три дерева, которые содержат в себе один “новый” токен, и одно дерево, которое не содержит в себе “новых” токенов (то, которое было возвращено при предыдущем запуске). Алгоритм выбирает одно из деревьев, которое содержит в себе наибольшее количество новых токенов (см. рисунок 15b). Также возвращается множество, состоящее из NUM 5 - токен, который ещё не покрыли.

При третьем запуске алгоритм на вход получает множество, состоящее из NUM 5,



(a) Результат первого запуска алгоритма.

(b) Результат второго запуска алгоритма.

(c) Результат третьего запуска алгоритма.

Рис. 15: Деревья для графа на рисунке 14, которые покрывают все токены

и снова начинает обход с корня. На этот раз существует четыре дерева, которые не содержат в себе новых токенов, и два дерева, которые содержат в себе один новый токен. И вернёт алгоритм вместе с одним из таких деревьев (например, как на рисунке 15c) ещё и пустое множество.

Если запустить алгоритм, передав ему на вход пустое множество, то он вернёт null, не начиная обхода. Это будет означать, что все возможные “листья” уже покрыты.

3.2.2. Динамическая подсветка

Современные IDE однако могут осуществлять статическим образом, но и на основе некоторой контекстной информации. Например, если каретка курсора располагается непосредственно перед открывающей скобкой, то эта скобка и парная ей закрывающая скобка могут быть особо выделены цветом. В данной работе такая возможность также реализована.

В такой задаче уже не все предположение, сделанные в случае статической подсветки, применимы. Добавим в грамматику арифметических выражений правило

expr : LBRACE expr RBRACE

и рассмотрим вход, изображённый на рисунке 16.

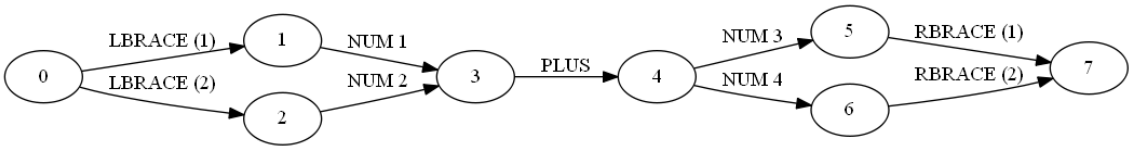


Рис. 16: Пример получения SPPF из нескольких деревьев.

Алгоритм, предложенный в предыдущем разделе, вернёт деревья, изображённые на рисунке 17.

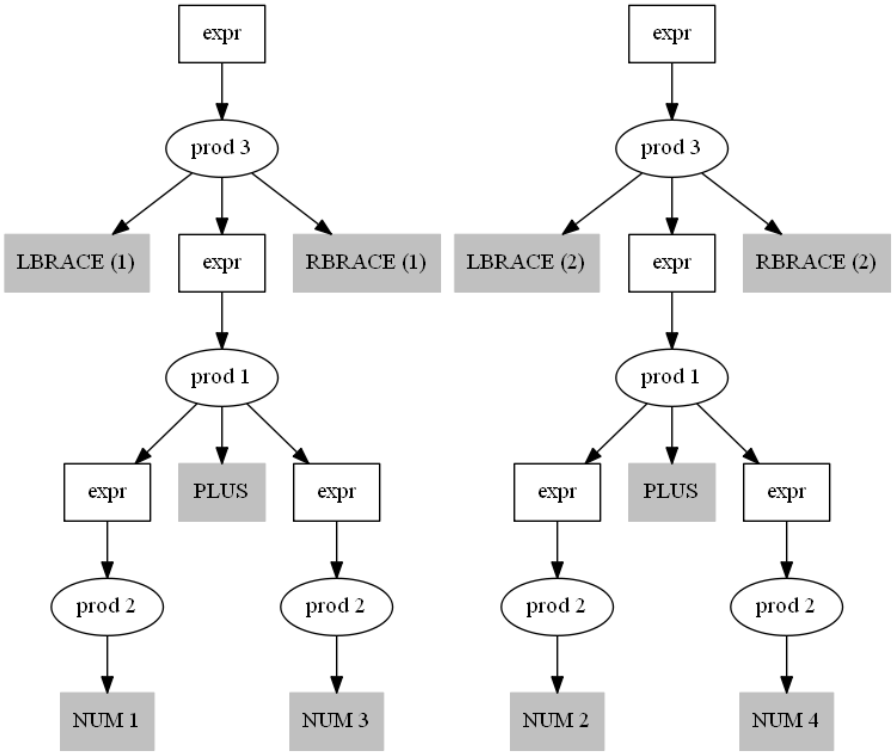


Рис. 17: Результат алгоритма для статической подсветки.

Предположим, что каретка курсора находится непосредственно перед токеном LBRACE (1). Тогда в зависимости от пути во входном графе парными этому токену будут являться RBRACE (1) или RBRACE(2). Поэтому нужно вернуть два дерева разбора. Одно должно содержать токены LBRACE(1) и RBRACE(1), а другое – токены LBRACE(1) и RBRACE(2) (см. рис. 18). Заметим при этом, что наличие других токенов (например, LBRACE 2) нас особо не интересует. Лишь бы получилось корректное синтаксическое дерево, соответствующее некоторому пути во входном графе.

Таким образом, если курсор стоит перед парным токеном, то нам нужно вернуть ровно n деревьев, где n – количество пар данного токена.

Поскольку алгоритмы во многом пересекаются, решено было модифицировать алгоритм для статической подсветки следующим образом.

В качестве входа алгоритм принимает два параметра:

- root – корень графа разбора;
- токен t, пары которому нужно найти.

В ходе своей работы алгоритм сохраняет полученные деревья в множестве trees. Как и в предыдущем алгоритме, все вершины окрашены в белый цвет.

Алгоритм возвращает GetAllTreesWithToken (root, token).

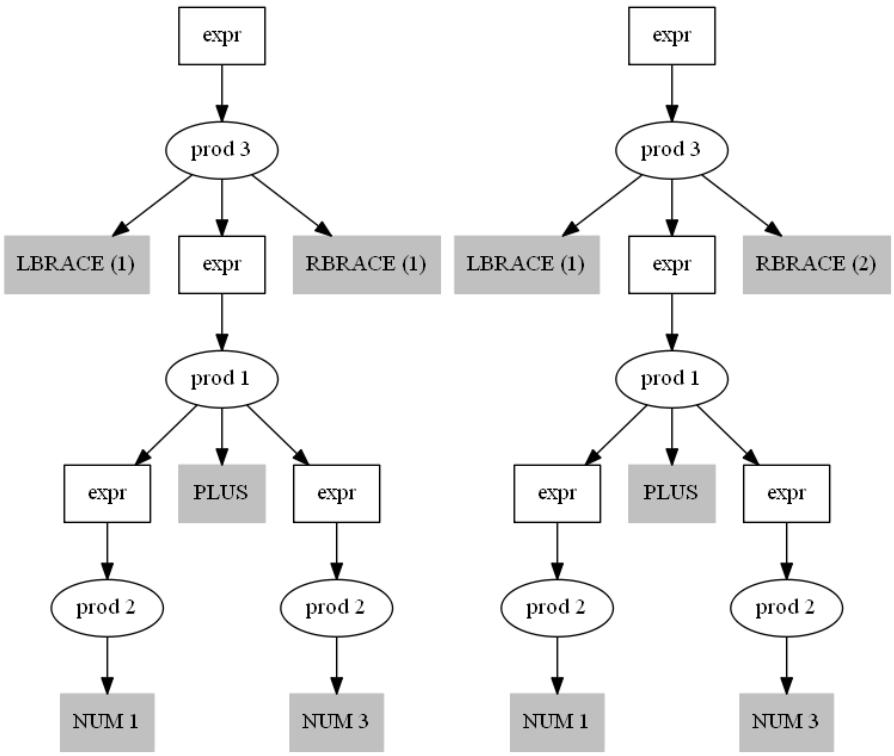


Рис. 18: Результат алгоритма для динамической подсветки.

GetAllTreesWithToken (root, token):

1. trees = $\emptyset$
2. newTree = Handle (root, token, trees)
3. пока newTree $\notin$ trees:
4. trees = trees $\cup$ {newTree}
5. newTree = Handle (root, token, trees)
6. вернуть trees

Handle (v, token, trees):

1. Если вершина v - вершина-символ, то
   - если OutEdges (v) == 1, то Handle (Succ (v), token, trees).
   - если OutEdges (v) > 1, то если среди u $\in$ Succ(v) $\exists$ s такой, что token $\in$ Tokens (s), то Handle (s, token, trees).
   - Иначе идти по такому r $\in$ Succ(v), который содержит наибольшее количество новых токенов (на фоне множества trees).

   Перейти к шагу 3.

2. Если текущая вершина - вершина-продукция, то для каждой вершины u из Succ(v) сделать следующее: Handle (u, token, trees). Перейти к шагу 3.

3. Покрасить v в чёрный цвет. Если v == root, то перейти к шагу 4.

4. Алгоритм завершает работу. Вернуть дерево, состоящее из “чёрных” вершин.

Иллюстрация работы алгоритма продемонстрирован на рисунке 19.

```
static int Calculate(bool cond)
{
    var expr = "(1 + ";

    if (cond)
    {
        expr = expr + " 2)";
    }
    else
    {
        expr = expr + " 3)";
    }
    return Program.Eval(expr);
}
```

Курсор перед открывающей скобкой.

```
static int Calculate(bool cond)
{
    var expr = "(1 + ";

    if (cond)
    {
        expr = expr + " 2)";
    }
    else
    {
        expr = expr + " 3)";
    }
    return Program.Eval(expr);
}
```

Курсор после закрывающей скобки.

Рис. 19: Иллюстрация работы алгоритма с подсветкой скобок.

4. Особенности реализации

4.1. Изменения в генераторе

В генератор добавлены два ключа: -highlighting и -namespace. Значение первого ключа говорит, нужна или нет подсветка. Второй ключ служит больше для технических целей соответствует имени namespace'а в сгенерированных файлах на языке C#.

Если ключ highlighting отсутствует или же имеет значение false, то генератор отработает в обычном режиме и вернёт только парсер этой грамматики.

Если ключ highlighting имеет значение true, то помимо стандартных преобразований надо грамматикой (таких как раскрытие правил EBNF) будут произведены дополнительные. Они направлены на то, чтобы для каждого правила сгенерировать семантику для подсветки. Если же во входной грамматике была определена пользовательская семантика, то она "затирается", а на ее место ставится семантика подсветки.

```
<?xml version="1.0"?>
<SyntaxDefinition name="CalcHighlighting">
    <Tokens color="OVERRIDES_ATTRIBUTE">
        <Token> LBRACE </Token>
        <Token> RBRACE </Token>
    </Tokens>
    <Tokens color="CONSTANT_IDENTIFIER_ATTRIBUTE">
        <Token> NUMBER </Token>
    </Tokens>
    <Tokens color="OPERATOR_IDENTIFIER_ATTRIBUTE">
        <Token> DIV   </Token>
        <Token> MINUS </Token>
        <Token> MULT  </Token>
        <Token> PLUS  </Token>
        <Token> POW   </Token>
    </Tokens>
    <Matched>
        <Pair>
            <Left>  LBRACE </Left>
            <Right> RBRACE </Right>
        </Pair>
    </Matched>
</SyntaxDefinition>
```

Рис. 20: Пример xml файла.

Поскольку в общем случае неизвестно, какие терминалы будут во входной грамматике и соответственно в какой цвет каждый из них подсвечивать, то автоматизировать привязку к каждому токену определенного цвета невозможно. Поэтому решено на этом этапе также возвращать xml-файл (см. рис. 20). Он содержит в себе сопоставление "токен -> цвет". Также в нём поддерживается задание парных символов, что позволяет осуществлять динамическую подсветку (например, для открывающих и закрывающих скобок).

В генераторе используются цвета, принятые в ReSharper SDK. Для удобства пользователя названия всех доступных цветов указываются в комментариях сгенерированного xml файла.

4.2. Архитектура решения

Общая архитектура решения изображена на 21. Серым цветом отмечены модули, которые были созданы либо изменены в ходе данной работы.

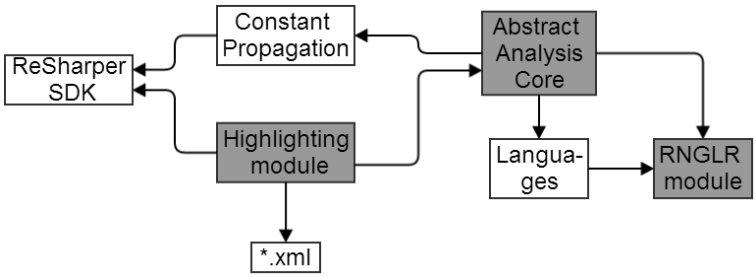


Рис. 21: Архитектура решения.

Рассмотрим эти модули поподробнее.

Модуль RNGLR содержит в себе функции для работы с графом разбора. В частности он включает реализацию описанных в предыдущей главе алгоритмов извлечения деревьев из SPPF.

Модуль Languages содержит в себе информацию, специфичную для каждого языка. Например, у каждого языка свой алфавит, свой синтаксический анализатор, своя функция вычисления семантики. Именно такую информацию Languages и хранит.

Работа модуля AbstractAnalysisCore проходит в два этапа.

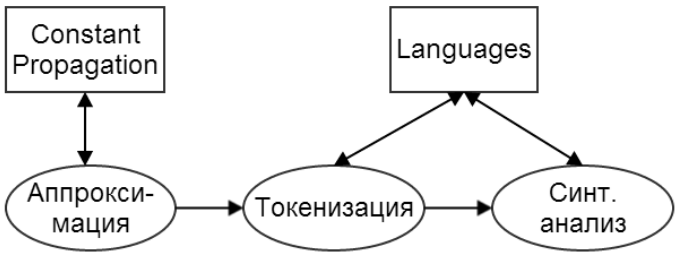


Рис. 22: Анализ файла модулем AbstractAnalysisCore.

На первом этапе происходит анализ файла (рисунок 22). Сперва проходит аппроксимация множества значений строковых выражений, для чего используется другой модуль - ConstantPropagation, целью которого является протягивание констант. Результатом аппроксимации является граф, на рёбрах которого находятся фрагменты строк исходного файла. Также с графом ассоциируется язык, которому соответствуют строки. Далее происходит токенизация полученного графа, в результате которой

получается граф, на рёбрах которого находятся токены, а не строки. После токенизации происходит синтаксический анализ. Результат синтаксического анализа - граф разбора - сохраняется.

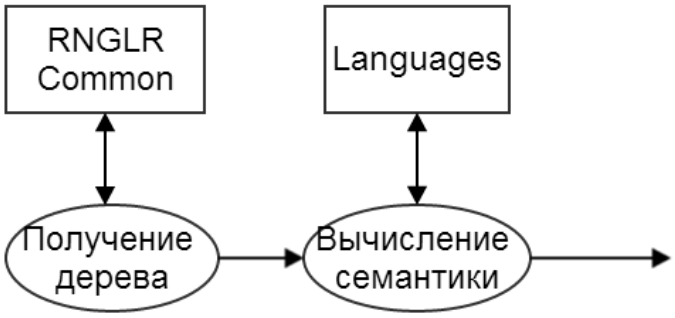


Рис. 23: Получение дерева в AbstractAnalysisCore.

На втором этапе работы модуля AbstractAnalysisCore происходит генерация деревьев. Этот этап проиллюстрирован на рисунке 23. Для получения дерева используется модуль RNGLRCommon, который был описан ранее. После того, как дерево получено, происходит вычисление его семантики, и возвращается полученный результат.

Наконец, модуль Highlighting отвечает за подсветку и интеграцию с ReSharper SDK. Он состоит из трёх подмодулей. Один из них отвечает за статическую подсветку, второй - за динамическую подсветку, а третий (Helper) служит для взаимодействия между ними.

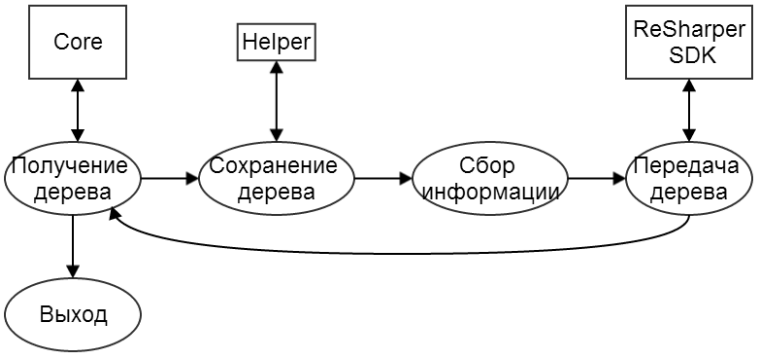


Рис. 24: Схема работы модуля статической подсветки.

Схема работы подмодуля, занимающегося статической подсветкой, изображена на рисунке 24. Этот подмодуль начинает работу при каждом изменении файла с исходным кодом. Такое поведение достигается за счёт установления соответствующего атрибута в одном из классов этого подмодуля. Контекст при этом не учитывается. Модуль создаёт инстанцирует класс AbstractAnalysisCore, который проводит синтаксический анализ кода на встроенном языке. После этого возвращается дерево разбора, полученное согласно алгоритму, описанному в разделе 3.2.1. Полученное дерево со-

храняется в Helper'е. Далее происходит обход этого дерева, в ходе которого происходит сбор информации о том, какой токен на какой позиции в файле находится. Также каждому токену назначается цвет, в который этот цвет красить (здесь всплывает xml файл, который описан в предыдущем разделе). После этого полученная информация передаётся ReSharper SDK, который осуществляет непосредственную подсветку. И начинается запрос нового дерева. Процесс продолжается до тех пор, пока полученное дерево не является пустым.

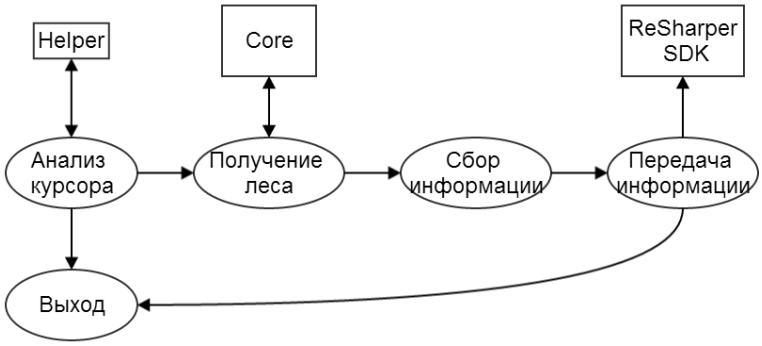


Рис. 25: Схема работы модуля динамической подсветки.

Схема работы подмодуля, занимающегося динамической подсветкой, выглядит несколько иначе (рисунок 25). В отличие от предыдущего модуля он реагирует на каждое изменение положения каретки курсора в файле с исходным кодом. При запуске этого подмодуля сперва происходит анализ положения курсора. Если каретка стоит внутри строкового выражения и перед (после) открывающего (закрывающего) парного символа, то определяется текущий встроенный язык. Это определение происходит за счёт поиска по деревьям разбора, полученным при статической подсветке. Сначала определяется дерево, которое содержит нужный диапазон, а потом по этому дереву определяется, какой именно это встроенный язык. На основе полученной информации у объекта класса AbstractAnalysisCore вызывается функция, которая возвращает все деревья, которые содержат нужный токен. Алгоритм, который используется при этом, описан в разделе 3.2.2. Далее по этим деревьям, так же, как и в работе подмодуля статической подсветки, происходит сбор информации и её передача ReSharper SDK, который подсвечивает парные токены.

Заключение

В рамках данной работы были достигнуты следующие результаты:

- изучены инструмент YaccConstructor и платформа ReSharper SDK.
- создан алгоритм подсветки произвольных встроенных языков, описанных грамматикой;
- реализован и апробирован полученный алгоритм;
- результаты группы, в которую входит автор, приняты на семинар “Наукоёмкое программное обеспечение”.

Весь проект можно найти на сайте https://code.google.com/p/recursive-ascent/, автор принимал участие под учетной записью ivanovandrew2004.

Дальнейшее направление работы

В дальнейшем планируется реализация так называемой семантической подсветки - подсветки, которая основывается на некоторых дополнительных свойствах программы.

На рисунке 26 продемонстрирован один из примеров такой подсветки - подсветка “мёртвого кода”. Как можно заметить, те фрагменты кода, которые не имеют смысла, окрашены серым цветом, вне зависимости от типа языковых конструкций.

```
static int Calc(bool flag)
{
    int res = 0;
    if (flag)
    {
        res = 1;
    }
    else
    {
        res = 2;
    }

    return res;
    return 3;
    return 4;
}
```

Рис. 26: Семантическая подсветка в ReSharper.

Другим направлением является реализация других возможностей современных IDE по отношению ко встроенным языкам. Например, средства навигации по коду (find usage, goto definition), автодополнение, анализ корректности типов.

Список литературы

[1] Alvor. Project site. — https://code.google.com/p/alvor/.

[2] Christensen Aske Simon, Moller Anders, Schwartzbach Michael I. Precise Analysis of String Expressions. // Static Analysis. 10th International Symposium, SAS 2003 San Diego, CA, USA, June 11–13, Proceedings / Ed. by Radhia Cousot. — Springer Berlin Heidelberg, 2003. — P. 1–18.

[3] Grigoriev Semyon, Kirilenko Iakov. GLR-based abstract parsing // CEE-SECR'13 Proceedings of the 9th Central & Eastern European Software Engineering Conference in Russia.

[4] Hyunha Kim, Kyung-Goo Doh, David Schmidt. Static validation of dynamically generated HTML documents based on abstract parsing and semantic processing // Static Analysis. 20th International Symposium, SAS 2013, Seattle, WA, USA, June 20-22, 2013. Proceedings. — Springer Berlin Heidelberg, 2013. — P. 194–214.

[5] IDEA. — http://www.jetbrains.com/idea/.

[6] IntelliLang. — http://www.jetbrains.com/idea/webhelp/intellilang.html.

[7] Microsoft Visual Studio. — http://www.visualstudio.com.

[8] Minamide Y. Static Approximation of Dynamically Generated Web Pages. // WWW'05 Proceedings of the 14th international conference on World Wide Web. — ACM New York, NY, USA, 2005. — 11 May. — P. 432–441.

[9] PHPStorm. — http://www.jetbrains.com/phpstorm/.

[10] Scott Elizabeth, Johnstone Adrian. Right Nulled GLR Parsers // ACM Transactions on Programming Languages and Systems (TOPLAS). — 2006. — P. 577–618.

[11] Syme Don, Granicz Adam, Cisternino Antonio. Expert F# 3.0. — Apress, 2012. — ISBN: 978-1-4302-4650-3. — URL: `http://www.apress.com/9781430246503`.

[12] YaccConstructor. — http://code.google.com/p/recursive-ascent/.

[13] Кириленко Я.А, Григорьев С.В., Авдюхин Д.А. Разработка синтаксических анализаторов в проектах по автоматизированному реинжинирингу информационных систем // Научно-технические ведомости Санкт-Петербургского государственного политехнического университета. Информатика. Телекоммуникации. Управление. — 2013. — С. 94–98.