Санкт-Петербургский Государственный Университет
Математико-механический факультет
Кафедра системного программирования

Разработка архитектуры для генератора синтаксических анализаторов

Курсовая работа студента 445 группы
Улитина Константина Андреевича

Научный руководитель . . . . . Я.А. Кириленко
ст. преподаватель / подпись /

Санкт-Петербург
2010

Содержание

Введение 3

Общие требования предметной области 3

Предметно-ориентированный язык 4

Постановка задачи 5

Решение 7

Тестирование 11

Интеграция с Irony 12

Компоненты 12

Результаты работы 13

Приложение 1. Код преобразования внутреннего представления Irony 14

Список литературы 15

Введение

Общие требования предметной области

Задачи автоматического реинжиниринга программ выдвигают особые требования к генераторам синтаксических анализаторов. Во многом это связано с тем, что теория синтаксически управляемой трансляции развивалась одновременно с появлением языков, сейчас называемых устаревшими (legacy languages). Такие языки проектировались, когда еще не были получены основные теоретические результаты, положенные в основу наиболее распространенных современных генераторов. Поэтому устаревшие языки трудны для синтаксического анализа даже с использованием современных инструментов, которые значительно упрощают создание анализаторов.

Среди характеристик, описывающих тот или иной инструмент генерации синтаксических анализаторов, можно выделить следующие:

- Класс принимаемых языков и грамматик.
- Разрешение неоднозначностей в грамматике.
- Скорость работы сгенерированного парсера/транслятора.
- Возможности языка описания грамматики. Поддержка метаправил.
- Близость абстрактного синтаксического дерева к исходному коду анализируемой программы.
- Простота отладки.
- Наличие встроенного или внешнего лексического анализатора.
- Восстановление после ошибок.
- Сопровождаемость, качество документации.
- Целевой язык.
- Возможность модификации инструмента под свои нужны.
- Простота использования.

Многие из этих свойств завязаны на алгоритм генерируемого парсера. Например, при использовании рекурсивного спуска для LL-грамматик отладка существенно проще, чем при табличном методе, как для LL, так и для LR-грамматик. GLR-алгоритм не решает конфликты по ходу разбора входной строки, а строит лес и фильтрует его.

Предметно-ориентированный язык

Отдельной, но близкой к реинжинирингу программ задачей, является работа с небольшими языками, созданными под конкретную задачу в рамках одного проекта. Пользователям таких языков не нужна вся мощь инструмента, описываемый язык может быть очень простым. При этом удобно использовать сторонний инструмент для разработки парсера для такого языка, а не писать его самому. Что дает стандартные преимущества использования сторонних компонент: сохранение времени на разработку основной функциональности и высокое качество компонента, так как он разрабатывался специалистами в данной предметной области и был хорошо протестирован. Однако, есть и обратная сторона. Такой случай требует особые свойства генератора. Полученный парсер является лишь частью разрабатываемого продукта, поэтому необходим способ удобной интеграции его с основной функциональностью. Также может быть, что разработчики не имели дел с генераторами синтаксических анализаторов, и им приходится тратить время на их изучение. Такой случай близок к понятию предметно-ориентированного языка (*Domain Specific Language*) и возникает довольно часто. При такой ситуации удобен способ описания грамматики в объектно-ориентированной форме. Например,

```
command.Rule = Symbol("move") + number + "pixels" + direction + ".";
direction.Rule = Symbol("up") | "down" | "left" | "right";
```

что является эквивалентом следующей записи в форме Бэкуса-Наура:

```
<Command> ::= "move" <number> "pixels" <Direction> "."
<Direction> ::= "up" | "down" | "left" | "right"
```

Как видно, для определения последовательности и альтернатив используется перегрузка операторов. Для определения других конструкций могут использоваться соответствующие классы:

```
// <CommandList> ::= <Command>+
commandList.Rule = MakePlusRule(commandList, null, command);
```

При таком задании грамматики удобно описывать семантику в методах класса, отвечающего за конкретный нетерминал.

Постановка задачи

Как было сказано выше, к генераторам синтаксических анализаторов предъявляется довольно много требований, часто взаимоисключающих. Несколько путей решения задач анализа и трансляции (регулярные выражения, различные грамматики в форме Бэкуса-Наура, parsing expression grammars), помноженные на количество целевых языков трансляции, вызывают появление огромного числа инструментов. К тому же, разные генераторы делают упор на разные требования, не в полной мере удовлетворяя другим. Обзор наиболее известных инструментов можно найти в статье [5].

Виртуальная машина платформы .NET позволяет интегрировать разные программные средства в один проект. При этом, что удобно, они могут быть написаны на различных языках программирования. Однако на данный момент, в силу относительной молодости самой платформы, для нее написано сравнительно мало инструментов для генерации синтаксических анализаторов. В частности, не реализован GLR-алгоритм. Соответственно, не все требования пользователей могут быть удовлетворены.

Исходя из вышеперечисленного, было решено разработать инструмент для генерации синтаксических анализаторов под .NET, использование которого давало бы следующие преимущества:

- Различные способы задания входной грамматики.
- Возможность выбора алгоритма генерации, исходя из требований (см. Общие требования предметной области).
- Расширяемость – возможность написания своих фронтэндов, преобразований и генераторов под текущую задачу.
- Компоновка различных компонент в единую оболочку и выпуск специализированного продукта.

  Это может быть достигнуто такими свойствами инструмента, как:

- Модульность.

  В частности, имеется возможность подключить в качестве парсера входной грамматики или генератора сторонние открытые инструменты, написанные под .NET. При этом необходимо преобразовать их внутреннее представление грамматики в наше для парсера, и наоборот для генератора.
- Универсальное внутреннее представление грамматики. Должно поддерживать как можно больше различных синтаксических конструкций, чтобы быть совместимым с другими инструментами.

Было решено реализовывать инструмент на языке функционального программирования F#, включенного в платформу .NET, в силу удобства предоставляемых им синтаксических конструкций и хорошей интеграцией с самой платформой. За основу был выбран инструмент YARD и порождаемое им внутреннее представление грамматики, вместе с языком описания трансляций. Описание типов правил и синтаксических возможностей, предоставляемых им, можно найти в работе [4].

Решение

Общий процесс автоматического построения синтаксического анализатора можно представить в виде следующей диаграммы активности:

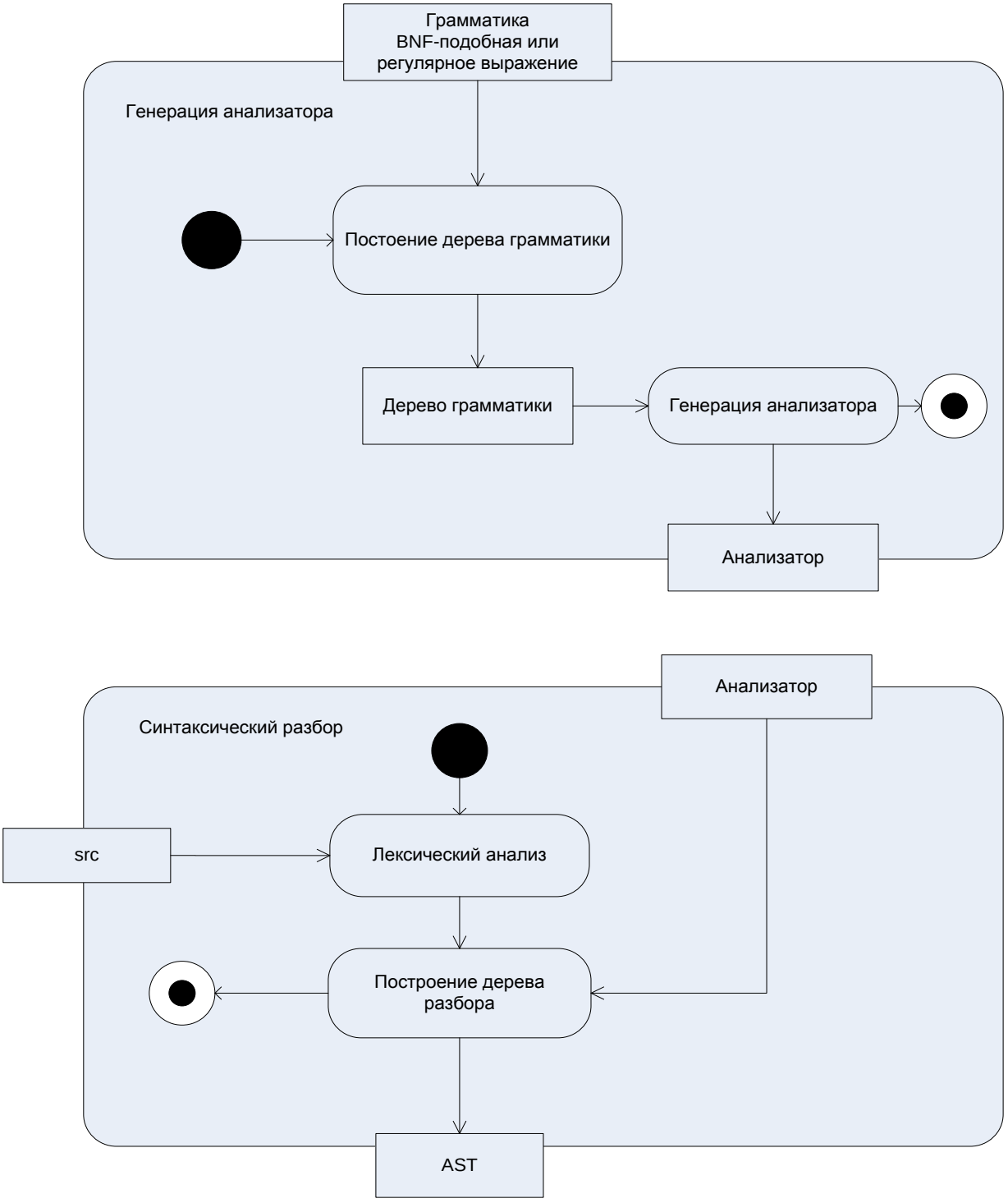

рис. 1

Общий процесс работы с генератором синтаксических анализаторов выделяет 3 последовательно работающих компонента. А именно:

1. Парсер входной грамматики, преобразующий грамматику в некоторое внутреннее представление (дерево граматики). Может включать в себя лексер.
2. Генератор синтаксического анализатора по заданной грамматике. Получает необходимые синтаксическому анализатору данные для разбора именно этой грамматики. Например, для табличного LR анализатора это будут таблицы перехода и action код.
3. Порожденный транслятор. Строит абстрактное синтаксическое дерево по входной строке. Может включать в себя лексер.

Парсер входной грамматики и генератор синтаксического анализатора почти независимые компоненты. Синтаксический анализатор зависит от генератора. Соответственно, было решено попробовать сконструировать среду, в которой бы можно было выбирать разные парсеры грамматики и генераторы для различных целей. Также, такой инсрумент позволял бы выполнять задачи, отлчные от синтаксического разбора входной строки, не нарушая общности архтектуры, а именно:

- Получение информации о грамматике (метрики, типы синтаксических конструкций и др.)
- Свойства грамматики (леворекурсивность, является ли LL(1),… – см. [12]).

Для этого только нужно заменить генератор компонентом, выполняющим соответствующие функции. Для понятности будем все равно далее называть этот компонент генератором.

Чтобы разные парсеры входной грамматики и генераторы смогли работать друг с другом, надо предоставить формат внутреннего представления грамматики, максимально охватывающий возможности разных генераторов. Также могут необходимы преобразования входной грамматики, не изменяющие порождаемый ей язык, но делающие возможность работы с ней выбранным генератором.

В итоге мы пришли к такой модульной схеме разрабатываемого инструмента:

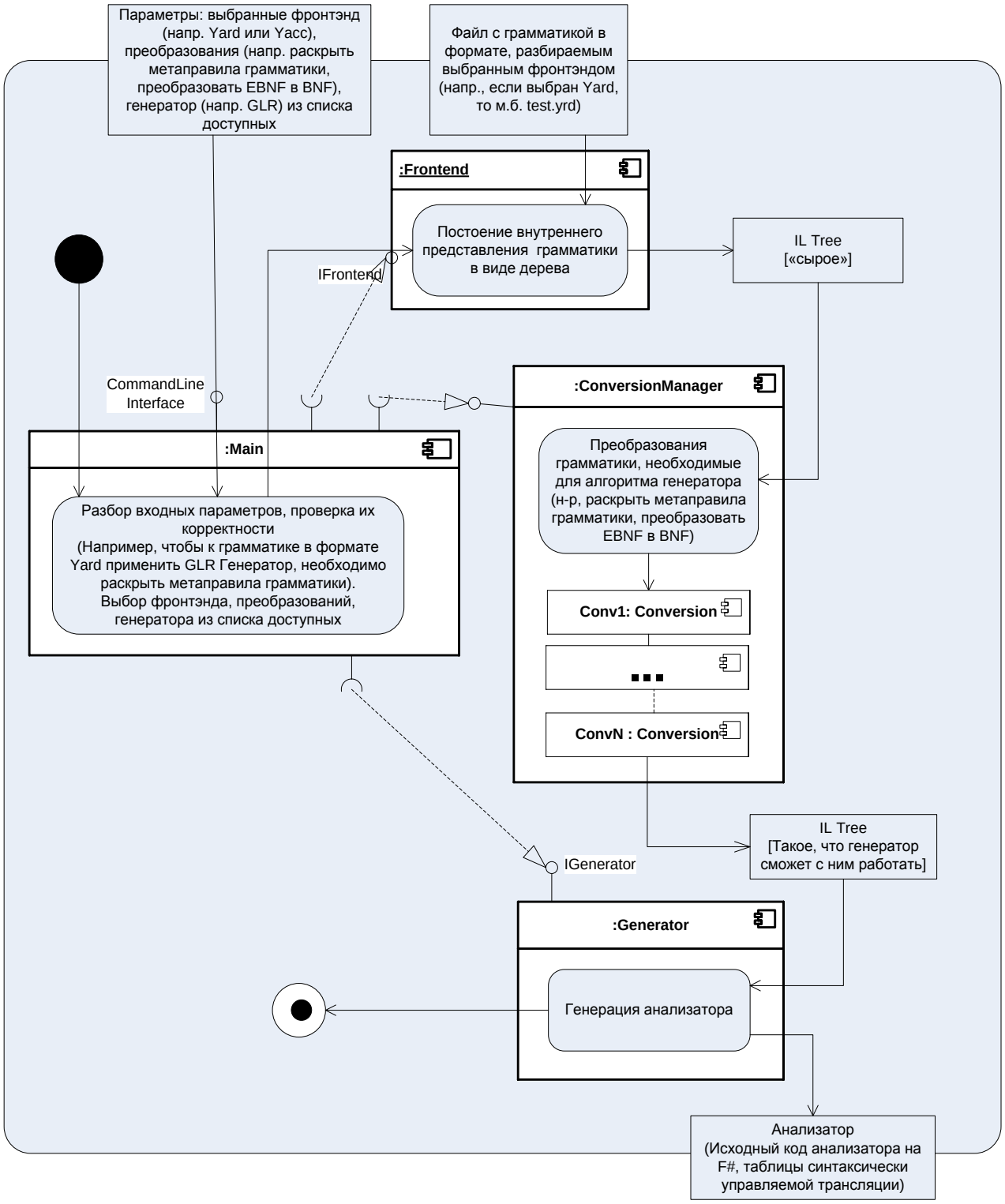

рис. 2

Frontend

Парсер грамматики. Предназначен для преобразования грамматики во внутреннее представление. Архитектура не ограничивает входные данные классической парой файлов с описанием лексем и собственно грамматикой. Frontend сам определяет формат входных данных. Описывает следующий интерфейс:

```
type IFrontend = interface
    abstract Name : string
    abstract ParseGrammar : obj -> Definition.t<Source.t,Source.t>
    abstract ProductionTypes : string list
end
```

IL

Язык внутреннего представления грамматики. Сама грамматика представлена в виде дерева с узлами некоторого алгебраического типа данных(IL tree). За основу был взят инструмент YARD, как поддерживающий многие синтаксические конструкции, в том числе конструкции EBNF, метаправила[4], перестановки.

Conversions

Преобразования дерева внутреннего представления. Представляют собой некоторые функции на множестве деревьев. Служат, как правило, для раскрытия некоторых типов правил, которые не поддерживаются используемым генератором. Например, метаправила[4]. Или конструкции EBNF для некоторых генераторов. Могут применяться последовательно. Описывают интерфейс:

```
type IConvertion = interface
    abstract Name : string
    abstract ConvertList : Rule.t<Source.t, Source.t> list ->
                           Rule.t<Source.t, Source.t> list
    abstract EliminatedProductionTypes : string list
end
```

Generator

Генератор парсера. Принимает дерево внутреннего представления грамматики и делает с ним определенную работу:

- Строит парсер
- Проводит анализ грамматики
- Определяет по грамматике, к какому классу она принадлежит

Генераторы должны описывать интерфейс:

```
type IGenerator = interface
```

```
        abstract Name : string
        abstract Generate : Definition.t<Source.t,Source.t> -> obj
        abstract AcceptableProductionTypes : string list
    end
```

Соответственно, метод `Generate` возвращает объект с требуемыми свойствами, и дальнейшее исполнение программы уже обрабатывает результат, если это необходимо.

Main

Последовательно вызывает разные компоненты инструмента, передавая им необходимые параметры. Разбирает параметры входной строки. Может выдавать список доступных фронтэндов, генераторов и преобразований. Проверяет, что заданная последовательность удовлетворяет некоторым условиям (см. Тестирование).

Тестирование

Описанная выше модульная архитектура и наличие интерфейсов компонент позволяют внедрить в продукт различные виды тестирования. В частности, присутствует следующая проверка на типы правил вывода: каждый узел IL tree является узлом некоторого типа, например:

- **PAlt, PSeq** – альтернатива (A|B), последовательность (AB).
- **PSome, PMany, POpt** – A*, A+, A?
- **PMetaRef** – ссылка на метаправило [4].

Как было сказано выше, разные генераторы могут работать с разными типами правил. Ни один из них не может работать с PMetaRef, не все работают с конструкциями EBNF (PSome, PMany, POpt). В таком случае преобразования грамматики должны раскрывать эти типы правил в другие. Также, разные фронтэнды строят внутреннее представление, содержащее некоторый набор типов правил.

Соответственно, было бы удобно проверять, что заданная последовательность фронтэнда и преобразований будет выдавать генератору только те типы правил, которые он принимает. Эту проверку можно выполнять как при обычной работе программы, так и при автоматическом unit-тестировании. Более подробное описание внедрения тестирования в проект можно найти в работе [10].

Интеграция с Irony

Чтобы показать возможность интеграции стороннего инструмента с нашим продуктом, мной был реализован фронтэнд на основе фронтэнда Irony. В Irony грамматика задается с помощью перегрузки операторов, и сам инструмент направлен на работу с DSL (см. Предметно-ориентированный язык). Для этого было необходимо преобразовать внутреннее представление грамматики Irony в наше. С помощью синтаксических конструкций языка F# это достигается сравнительно просто и наглядно (См. Приложение 1.)

Компоненты

На данный момент в проекте реализованы следующие компоненты:

Frontends

- YARD - язык описания грамматики (Кириленко Я.А.) с поддержкой метаправил (Чемоданов И.).
- IronyFrontend – грамматика задается в виде класса. Разработан путем интеграции open source проекта Irony [11] в наш инструмент.

Convertions

- ExpandMeta – раскрывает метаправила [4].
- ExpandEBNF – раскрывает конструкции расширенной формы Бэкуса-Наура.

Generators

- RecursiveAscent – рекурсивно-восходящий генератор синтаксических анализаторов для работы с неоднозначными грамматиками [9].
- FParsec – библиотека парсер-комбинаторов.

Результаты работы

- Разработана архитектура модульного генератора синтаксических анализаторов, удовлетворяющая заданным требованиям.
- Проведен рефакторинг, необходимый для приведения уже существующих решений к разработанной архитектуре.
- Показана возможность интеграции стороннего компонента в разрабатываемый продукт на примере Irony.

Исходный код и текущее состояние дел проекта можно найти на сайте http://code.google.com/p/recursive-ascent/.

Приложение 1. Код преобразования внутреннего представления Irony

```
        let Convert (ironyGrammar : Irony.Parsing.Grammar) =
            let refTerms = ref [ironyGrammar.Root :> BnfTerm]
            let rec findBnfTerms (start:NonTerminal) =
                ResizeArray.iter
                    (fun termList ->
                        (ResizeArray.iter
                            (fun bnfTerm ->
                                if not (List.exists (fun findedTerm -> findedTerm =
bnfTerm) !refTerms) then
                                    refTerms := bnfTerm :: !refTerms
                                    if bnfTerm :? NonTerminal then
                                        findBnfTerms (bnfTerm :?> NonTerminal) )
                            termList ) )
                    start.Rule.Data

            findBnfTerms ironyGrammar.Root

            let nonTerminals = List.filter (fun (t : BnfTerm) -> (t :? NonTerminal))
!refTerms
            let (grammar : IL.Grammar.t<IL.Source.t, IL.Source.t>) =
                List.map
                    (fun (bnfTerm : BnfTerm) ->
                        let nt = bnfTerm :?> NonTerminal
                        let productionOpt =
                            ResizeArray.fold
                                (fun prOpt bnfTermList ->
                                    let pseq =
                                        PSeq(
                                            ResizeArray.toList (ResizeArray.map
                                                (fun (bnfTerm : BnfTerm)->
                                                    ({omit = false;
                                                    rule = match bnfTerm with
                                                            | :? NonTerminal as term ->
PRef((term.Name, (-1,-1)), None)
                                                            | :? Terminal as term ->
PToken(term.Name, (-1,-1))
                                                            | _ -> failwith "Not
supported BnfTerm type"
                                                            ;
                                                    binding=None;
                                                    checker=None}))
                                                (bnfTermList)),
                                            None)
                                    match prOpt with
                                    | Some(pr)  -> Some(PAlt(pr, pseq))
                                    | None      -> Some(pseq) )
                                None
                                nt.Rule.Data
                        match productionOpt with
                        | Some(pr)  -> {name = nt.Name; args = []; body = pr; metaArgs =
[]; _public = (nt = (ironyGrammar.Root))}
                        | None      -> failwith "minimum 1 alternative is required" )
                    nonTerminals
            grammar
```

Список литературы

1. Ахо А., Сети Р., Ульман Дж. Компиляторы: принципы, технологии, инструменты. М:. Издательский дом «Вильямс» 2003. 768 с.
2. Ахо А., Ульман Дж. Теория синтаксического анализа, перевода и компиляции. Издательство «Мир», Москва, 1978.
3. Мартыненко Б.К. Языки и трансляции. — СПб.: Издательство С.-Петербургского университета, 2004. 229 с.
4. Чемоданов И.С. Генератор синтаксических анализаторов для решения задач автоматизированного реинжиниринга программ, 2007. 37 с.
5. Чемоданов И.С., Дубчук Н.П. Обзор современных средств автоматизации создания синтаксических анализаторов // Системное программирование. - СПб.: Изд-во С.-Петерб. ун-та, 2006. 286-316 с.
6. http://en.wikipedia.org/wiki/Comparison_of_parser_generators
7. Pickering R. Foundations of F#. 2007
8. http://www.research.microsoft.com/fsharp (дистрибутивы и документация по языку F#)
9. Григорьев С.В., Генератор синтаксических анализаторов для неоднозначных контекстно-свободных грамматик, 2010
10. Нишневич А. Внедрение unit-тестирования в проект на F#, 2010
11. http://irony.codeplex.com/
12. Силина О.А. Создание языка для проверки свойств контекстно-свободных грамматик, 2010.