Олег Доманов[1]

ФОРМАЛИЗАЦИЯ МИНИМАЛИСТСКОГО СИНТАКСИСА В ЯЗЫКЕ AGDA[2]

Аннотация. В статье представлен инструментарий для формализации синтаксических теорий в рамках минималистской программы Н. Хомского (Minimalist Program) в генеративной лингвистике. Он представляет собой набор формальных конструкций в языке Agda, позволяющих определить деревья вывода, синтаксические объекты и основные понятия минималистской теории — операцию соединения (Merge), признаки, копии, цепи, с-командование и др. В то же время, в формализме опущены операция согласования (Agree) и теория фаз.
Ключевые слова: минималистская программа, генеративная лингвистика, Agda.

Oleg Domanov

A FORMALIZATION OF MINIMALIST SYNTAX IN AGDA

Abstract. The paper presents a toolset for the formalization of syntactic theories in the framework of Noam Chomsky's Minimalist Program in generative linguistics. It comprises a set of formal instruments in Agda language, allowing to define derivation trees, syntactic objects and principal notions of minimalist theory—the operation Merge, features, copies, chains, c-command, etc. At the same time, the formalism omits the operation Agree and phase theory.
Keywords: minimalist program, generative linguistics, Agda.

Для цитирования: *Доманов О. А.* Формализация минималистского синтаксиса в языке Agda // Логико-философские штудии. 2022. Т. 20, № 4. С. 371–395. DOI: 10.52119/LPHS.2022.69.13.002.

Минималистская программа (Minimalist Program) была предложена Ноамом Хомским в 90-х годах прошлого века (Chomsky 1995) и в настоящее время служит основным подходом в генеративной лингвистике. Своей задачей эта программа считает максимальное упрощение синтаксической теории путём удаления из разработанной к тому времени генеративной грамматики всего, что не представляется необходимым. В общем виде, синтаксис в ней рассматривается как вычислительная система, позволяющая как строить языковые конструкции, так и понимать их.

[1] *Доманов Олег Анатольевич* — кандидат философских наук, старший научный сотрудник Института философии и права СО РАН.
Oleg A. Domanov, PhD, Seniour Researcher, Institute of Philosophy and Law of the Siberian Branch of the Russian Academy of Sciences.
domanov@philosophy.nsc.ru

[2] Я выражаю свою признательность Ольге Митрениной и Вадиму Лурье за вдохновение и продуктивное обсуждение многих аспектов настоящей статьи.

Она представляет собой абстрактную систему, имеющую два интерфейса — семантический (концептуально-интенциональный) и фонетический (сенсоромоторный). Первый отвечает за интерпретацию и «понимание», а второй — за физическое представление (устное, письменное или какое-то иное). Синтаксические структуры, построенные вычислительной системой, направляются на интерфейсы для последующей интерпретации или физического представления. В настоящее время нет признанной теории минималистского синтаксиса, поэтому минималистская программа — это именно программа, а не теория. Хомский максимально упрощает вычислительную систему, оставляя в ней, по существу, две операции, которые он называет Merge (соединение) и Agree (согласование). Относительно как одной, так и другой продолжаются активные обсуждения (из последних сборников см., например, Smith, Mursell, Hartmann 2020, самим Хомским также предлагаются различные модификации теории, см. Chomsky 2013; 2015). По этой причине от формализации минималистского синтаксиса требуется не столько формализовать некую теорию (которой пока не существует в законченном виде), сколько разработать формальный инструментарий для описания таких теорий. Поэтому задача настоящей статьи состоит в том, чтобы представить такой инструментарий и продемонстрировать некоторые из его возможностей по описанию синтаксических явлений с точки зрения минималистской программы. При этом опущены некоторые части теории, не имеющие достаточно устоявшейся формы, например полностью отсутствует описание операции Agree и теории фаз.

К настоящему времени предложено несколько формализаций минималистского синтаксиса (Collins, Stabler 2016; beim Graben, Gerth 2012; Stabler 2013), в частности, существует определение минималистской грамматики (Stabler 1996: 84). Все эти разработки проводятся в теоретико-множественных терминах. Особенность представленной ниже формализации состоит в том, что она, во-первых, опирается на теоретико-типовой подход и, во-вторых, использует в качестве основного язык программирования Agda. Выбор последнего может показаться неожиданным, однако языки программирования сами по себе являются формальными языками, а такие языки, как Coq, Agda и им подобные, дополнительно опираются на логический язык теории типов, что, в частности, позволяет широко использовать их для формализации математических теорий (для чего они, собственно, и разрабатывались в первую очередь). Предлагаемые ими понятия и инструменты, такие как зависимые типы или записи, часто существенно облегчают представление формальных теорий.

Описанная далее формализация доступна по адресу `https://github.com/odomanov/ttsemantics/tree/ver1/Agda/MinimalistProgram`. Там же можно найти доказательства упомянутых ниже тождеств и детали построения понятий и объектов.

1. Краткий обзор минималистской теории

Более подробно с основными положениями минималистской программы можно ознакомится в специальной литературе по генеративной грамматике, например (Митренина, Слюсарь, Романова 2018; Adger 2003; Boeckx 2006).

Результатом вычисления абстрактной синтаксической системы является синтаксический объект, который затем передаётся на два интерфейса — семантический для понимания и фонологический для внешнего представления. Синтаксический объект строится из лексических единиц (лексем, lexical items), которые выбираются из лексикона языка. Лексемы представляют собой в основном слова, но также функциональные категории (functional categories), ответственные за построение таких объектов как клаузы, глагольные группы и т. д. С помощью операции Merge эти единицы соединяются в синтаксические объекты, которые затем могут соединяться между собой, и далее до бесконечности. Этот процесс называется выводом синтаксического объекта (derivation) и представляется в виде дерева вывода. В минималистской теории принято, что Merge соединяет в точности два объекта, поэтому дерево вывода является бинарным.

Соединение объектов регулируется их признаками (features). Существуют различные понимания того, что они из себя представляют и как функционируют. Для формализации ниже выбран следующий подход. Признаки делятся на обладающие значением (valued), например род, который может быть женским, мужским и средним, и не обладающие значением (unvalued), такие как грамматические категории N, V и пр. Помимо этого, признаки могут быть интерпретируемыми или неинтерпретируемыми. Объекты с первыми могут быть непосредственно переданы на семантический и фонологический интерфейсы, а вторые должны быть предварительно элиминированы. Процедура элиминации называется проверкой (feature checking), и её конкретная форма служит предметом активных обсуждений. В частности, в некоторых вариантах теории её функции может исполнять операция Agree. В любом случае, чтобы проверка произошла, объект с неинтерпретированным признаком должен каким-то образом согласоваться с объектом с соответствующим ему интерпретированным признаком, в результате чего неинтерпретированный признак удаляется. Например, глагол может иметь неинтерпретированный именной признак uN, который проверяется присоединением к нему в качестве дополнения существительного с интерпретированным именным признаком N.

Кроме того, все признаки могут обладать «метасвойствами»: они могут быть сильными (strong) или слабыми (weak), а также обязательными (mandatory) или необязательными (optional).

Конкретный список признаков является открытым; в примерах ниже я ориентируюсь в основном на книгу Д. Эджера (Adger 2003). Набор признаков синтаксического объекта называется его меткой. Каждая лексема в лексиконе обладает своим набором признаков, а признаки других объектов вычисляются при соеди-

нении (Merge). Алгоритм этого вычисление реализован ниже в функции `LMerge`. Эта реализация очень предварительна и ориентирована в основном на использование в примерах. Сам алгоритм должен стать предметом отдельного исследования, в особенности учитывая то, что в вычислении меток также участвует операция Agree, форма которой не ясна до конца.

Перейдём теперь к формализации описанных структур.

2. Формализация

2.1. Синтаксис языка Agda

Формализация проводится в языке Agda (Norell 2009). Это теоретико-типовой язык, являющийся в то же время языком программирования. Agda — типизированный функциональный язык с зависимыми типами, основанный на теории типов П. Мартин-Лёфа (Martin-Löf 1984). Ниже представлен краткий обзор фрагмента языка, необходимого для наших целей. Он очень схематичен и не содержит многих подробностей, которые можно найти в документации по языку Agda по адресу `https://agda.readthedocs.io`.

Все объекты языка имеют тип. Универсум типов обозначается `Set`. Agda содержит бесконечное число иерархически организованных универсумов, но мы для простоты ограничимся лишь одним. Запись `a : A` означает «`a` относится к типу `A`», а запись `A : Set` — «`A` является типом» (относится к универсуму типов `Set`).

Тип функций из типа `A` в тип `B` записывается как `(x : A) → B`. Он означает, что *для каждого* элемента типа `A` имеется способ или процедура получения или вычисления некоторого элемента типа `B`. Запись `f : (x : A) → B` означает, что `f` — это функция из типа `A` в тип `B`. Если `B` не зависит от `x`, его можно опустить и писать просто `f : A → B`. Как и в λ-исчислении, многоместные функции представляются одноместными, значениями которых служат функции. Таким образом, например, `A → (B → C)` — это двуместная функция, в которой первый аргумент относится к типу `A`, второй — к типу `B`, а значение — к типу `C`. При этом скобки в таких случаях можно опускать и писать просто `A → B → C`.

Применение функции $c = f(a, b)$ записывается как `c = f a b`.

Вместо `(x : A) → B` тип функции можно также определять как `{x : A} → B`, что означает, что аргумент `x` является неявным и может не указываться при использовании функции. В этом случае значение аргумента должно вычисляться автоматически на основе дополнительной информации.

В общем виде производные типы в языке Agda определяются с помощью конструкторов, то есть функций, значениями которых являются элементы определяемого типа. Например, определение для типа натуральных чисел записывается таким образом:

```
data ℕ : Set where
```

```
zero : ℕ
suc  : ℕ → ℕ
```

Здесь ℕ — тип натуральных чисел, который мы определяем, `zero` — нульместная функция (константа), конструирующая первое число, а `suc` — одноместная функция ℕ → ℕ, конструирующая для каждого числа следующее число. Таким образом, тип натуральных чисел задаётся двумя конструкторами. Можно показать, что так сконструированные числа удовлетворяют аксиомам Пеано для натуральных чисел.

Тип может зависеть от параметров. Например, тип списка `List A` зависит от типа `A` элементов в этом списке (например, список чисел, список строк и т. д.). Соответственно, список определяется следующим образом:

```
data List (A : Set) : Set where
  []  : List A
  _::_ : A → List A → List A
```

Иначе говоря, список определяется двумя конструкторами: `[]` — нульместная функция (константа), задающая пустой список, и `_::_` — двуместная функция, добавляющая элемент типа `A` к списку типа `List A`. Запись с подчёркиваниями означает, что функцию можно использовать как инфиксный оператор, то есть вместо `_::_ x xs` писать также `x :: xs`. Таким образом, запись `x :: xs` обозначает список, который получается путём присоединения элемента `x` к списку `xs`.

Ещё один пример типа, который нам понадобится, — тип `Maybe`, который используется для опциональных (необязательных) значений:

```
data Maybe (A : Set) : Set where
  nothing : Maybe A
  just    : A → Maybe A
```

Таким образом, тип `Maybe A` имеет два вида элементов, соответствующих двум конструкторам: `nothing`, обозначающий отсутствие опционального значения, и `just x`, где `x` — объект типа `A`.

Конструкторы типа определяют все формы элементов, которые могут относиться к данному типу. Это позволяет проводить индукцию по типу, а также определять функции из этих типов путём использования шаблонов. Например, функция из типа `List A` может быть определена так (здесь, как обычно, сначала декларируется тип функции `f`, а затем определяются её значения на всех возможных аргументах):

```
f : {A : Set} → List A → List A
f [] = []
f (x :: xs) = xs
```

Эта функция вычисляет «хвост» списка, удаляя его первый элемент, если он есть. Она определяется путём указания значений для всех возможных форм аргументов. Согласно этому определению, её значение равно `[]`, если аргумент является пустым списком, и `xs`, если аргумент имеет форму `x :: xs`.

Таковы минимальные сведения о синтаксисе языка Agda, которые нам необходимы, чтобы начать рассмотрение. Дополнительные конструкции будут вводиться по ходу дела. Перейдём теперь к самой формализации.

2.1. Признаки и синтаксические объекты

Материалом для построения синтаксических объектов является лексикон, состоящий из лексических единиц или лексем, Lexical Items. Лексемы обладают признаками (features), регулирующими их сочетание при соединении. Признаки могут иметь или не иметь значение. Признаки делятся на три группы: `PHON` (фонетические), `SEM` (семантические) и `SYN` (синтаксические). Нас интересуют в основном последние. Для описания синтаксических признаков, имеющих значения, мы вводим для них понятия типа признака и значения признака. Например:

```
data SYNType : Set where
  cPerson cNumber cGender : SYNType

data SYNVal : SYNType → Set where
  c1 c2 c3            : SYNVal cPerson
  csing cplural cdual : SYNVal cNumber
  cfem cmasc cneut    : SYNVal cGender
  none : {A : SYNType} → SYNVal A
```

В качестве примера здесь определены типы признаков `cPerson`, `cNumber`, `cGender`, а также значения для них. Значение `none` означает отсутствие значения и может употребляться с любым типом (такое значение должно быть присвоено на каком-то этапе построения объекта прежде, чем он будет передан на интерфейсы). Мы имеем здесь пример зависимого типа — тип `SYNVal` зависит от типа `SYNType` (индексирован типом `SYNType`). Например, значения `csing`, `cplural` и `cdual` являются значениями типа `SYNVal cNumber`, а не `SYNVal cPerson` или какого-то иного.

Признаки, способные обладать значением, будем обозначать как `t ꞉ v`, что означает признак `t`, имеющий значение `v`. В итоге основная часть признака определяется следующим образом:

```
data SYNCore : Set where
  cD cN cV cA cP cv cC : SYNCore
  _꞉_ : (A : SYNType) → SYNVal A → SYNCore
```

Здесь `cD`, `cN` и т. д. — это признаки без значения, а конструктор `_&_` конструирует признаки со значением.

Примеры признаков:

- `cN` — категория существительного,
- `cCase & cNOM` — именительный падеж,
- `cNumber & cplural` — множественное число,
- `cCase & none` — неуказанный падеж (с отсутствующим значением).

Для признаков определены также параметры силы и опциональности:

```
data Strength : Set where
  strong : Strength
  weak   : Strength

data Opt : Set where
  mandatory : Opt
  optional  : Opt
```

В результате полное определение *признака* выглядит следующим образом:

```
data Feature : Set where
  fea : Opt → Strength → SYNCore → Feature
```

Таким образом, признаки конструируются конструктором `fea` на основе параметров опциональности и силы, а также значения `SYNCore`. Например, `fea mandatory weak cV` обозначает обязательный слабый признак `V`.

Agda позволяет вводить сокращения следующего вида, которыми я буду пользоваться:

```
pattern past      = fea mandatory weak (cTense & cpast)
pattern present   = fea mandatory weak (cTense & cpresent)
pattern V         = fea mandatory weak cV
pattern V*        = fea mandatory strong cV
```

Например, вместо `fea mandatory weak cV` можно писать просто `V`.

Метку (label) определим как список признаков, причём разделим их на три группы: фонетические `PHON`, семантические `SEM` и синтаксические `SYN`, причём последние разделим на проверенные (checked) и непроверенные (unchecked).

```
PHONList = List PHON
SEMList  = List SEM
SYNList  = List SYN
```

```
record Label : Set where
  constructor mkLabel
  field
    Phon : PHONList
    Sem  : SEMList
    F    : SYNList
    uF   : SYNList
```

Здесь `PHONList`, `SEMList` и `SYNList` — это списки соответствующих групп признаков. Использованный здесь тип записей `record` является типом с одним конструктором, имеющим поименованные аргументы, определённые в блоке `field`. Поэтому элементы типа `Label` конструируются как `mkLabel p s f uf`, где `p` относится к типу `PHONList`, `s` — к типу `SEMList`, а `f` и `uf` — к типу `SYNList` (соответственно, списки фонетических, семантических, синтаксических проверенных и синтаксических непроверенных признаков). Тип записи можно рассматривать как таблицу со столбцами `Phon`, `Sem`, `F` и `uF`, соответствующими полям. Тогда конструктор записи создаёт строку такой таблицы. Поскольку мы занимаемся синтаксисом, я для простоты буду считать, что `Sem` всегда пуст, а `Phon` содержит только текстовую строку, позволяющую различать лексические единицы. Например, нетранзитивный глагол `run` имеет метку

```
lbl = mkLabel ("run" :: []) [] (V :: []) (N :: []).
```

Как видно, у него `uF` равно `N :: []`, то есть ему требуются комплемент с признаком `N`. Условимся, что первый признак в `uF` относится к комплементу, а второй — к спецификатору (хотя эти термины, строго говоря, не применяются в минималистской теории). Например, для категории `T` поле `uF` может быть равно `v :: N :: []`, а соответствующая метка выглядит так:

```
lbl = mkLabel ("T" :: []) [] (T :: []) (v :: N :: []).
```

Это означает, что `T` имеет глагольную группу в качестве комплемента и именную группу в качестве спецификатора.

Базовые синтаксические объекты, используемые для построения других синтаксических объектов, берутся из заранее составленного перечня, известного как *лексический массив* (Lexical Array). Элементы этого перечня называются *лексемами* (Lexical Items) и определяются следующим образом.

```
record LexItem : Set where
  constructor mkLI
  field
    Phon : PHONList
    Sem  : SEMList
```

```
    F    : SYNList
    uF   : SYNList
```

Как видно, элементы перечня `LexItem` сходны с метками, и это неслучайно — лексемы в минималистской теории определяются списком своих признаков. Лексемы и метки легко преобразуются друг в друга с помощью следующих функций:

```
LI→Label : LexItem → Label
LI→Label (mkLI p s F uF) = mkLabel p s F uF

Label→LI : Label → LexItem
Label→LI (mkLabel p s F uF) = mkLI p s F uF
```

Сам же лексический массив имеет следующий вид:

```
data LexArr : Set where
  [_] : LexItem → LexArr
  _::_ : LexItem → LexArr → LexArr
```

Как можно видеть, он определяется индуктивно и имеет два конструктора. Первый конструирует минимальный перечень, который имеет вид `[ x ]`, где `x` — лексема (таким образом, это перечень из одной лексемы). Второй конструктор является операцией присоединения `LexItem` к уже имеющемуся перечню `LexArr`. Она обозначается как `_::_` (её имя совпадает с именем подобной операции для списков, но Agda в большинстве случаев способна их различить, опираясь на информацию о типах). Таким образом, в отличие от списков `List`, перечни `LexArr` не бывают пустыми и содержат минимум один элемент. Например, следующая запись обозначает перечень элементов `John`, `run`, `v`:

```
v :: John :: run :: [ John ]
```

Это последовательное присоединение элементов `run`, `John` и `v` к одноэлементному перечню `[ John ]`. Как видно, один и тот же элемент может присутствовать в массиве более одного раза, что соответствует использованию, например, одного и того же слова в разных местах текста (если у нас в тексте есть два Джона). Чтобы различить эти разные использования одного и того же слова, можно было бы их нумеровать, но мы вместо этого будем формализовать использование как фрагмент перечня, начинающийся с этого слова (иначе говоря, мы индексируем разные использования «хвостом» перечня, то есть местом в перечне). Тогда, например, мы можем обозначить (индексом $^{\texttt{la}}$ отмечены элементы перечня; как будет видно ниже, они выбираются операцией `Select`)

```
John1^la = [ John ]
run^la   = run :: [ John ]
John2^la = John :: run :: [ John ]
v^la     = v :: John :: run :: [ John ]
```

Понимаемые таким образом, выбираемые элементы перечня сами являются перечнями меньшей длины.

Чтобы описать построение синтаксических объектов, определим сначала понятие дерева вывода (Derivation Tree). *Деревом вывода* `DTree` будем называть дерево, изображающее процесс построения синтаксического объекта. Оно, как мы увидим, похоже по структуре на синтаксический объект, но не содержит меток. Дерево также определяется рекурсивно:

```
data DTree : Set where
  Select : LexArr → DTree
  Merge  : DTree → DTree → DTree
```

Как видно, дерево конструируется двумя операциями — `Select` и `Merge`. Первая операция выбирает некоторый элемент из перечня `LexArr`. Выбранная таким образом лексема образует примитивное дерево вывода («лист» дерева). Вторая операция строит дерево из двух других деревьев и соответствует операции Merge минималистской теории. Из определения видно, что дерево является бинарным.

Определим далее понятие *синтаксического объекта*:

```
data SO : Set where
  ⟦_⟧     : LexArr → SO
  ⟦_-_-_⟧ : Maybe Label → SO → SO → SO
```

Согласно этому определению, синтаксические объекты бывают двух видов. Первый имеет форму `⟦ x ⟧`, где `x` представляет собой выбранный элемент перечня, а второй конструируется из двух синтаксических объектов и метки, являющейся результатом операции над этими объектами, и имеет вид

```
⟦ label - so1 - so2 ⟧.
```

Для удобства чтения синтаксических объектов я буду также записывать последнее как

```
⟦ label
  - so1
  - so2 ⟧.
```

Значение метки может оказаться равным `nothing`, если она не может быть вычислена (в этом случае дерево вывода некорректно и не приводит к правильному синтаксическому объекту). Введём обозначение

```
pattern _•_•_•_ p s x y = just (mkLabel p s x y)
```

Тогда значения меток будут иметь вид либо `nothing`, либо `p • s • x • y`, где `p` — это список фонетических, `s` — семантических, `x` — синтаксических проверенных, а `y` — синтаксических непроверенных признаков.

Из определения видно, что синтаксический объект либо извлекается из перечня, либо конструируется из двух уже имеющихся синтаксических объектов. Таким образом, синтаксические объекты имеют структуру бинарного дерева.

Сформулированные определения позволяют нам проводить некоторые вычисления. Например, легко можно извлечь метку из объекта:

```
lbl : SO → Maybe Label
lbl ⟦ x ⟧ = just (LA→Label x)
lbl ⟦ x - _ - _ ⟧ = x
```

(`LA→Label` — это функция, извлекающая метку из первого элемента `LexArr`.) В этом определении соединяемые синтаксические объекты аргумента не используются в результате, поэтому могут быть изображены подчерком (их значения неважны).

Функция проверки того, является ли синтаксический объект минимальной проекцией, выглядит следующим образом (`Bool` — это тип истинностных значений, имеющий всего два элемента — `true` и `false`):

```
_is-min : SO → Bool
⟦ _ ⟧ is-min = true
_ is-min     = false
```

Она просто проверяет, является ли объект листом дерева, то есть одноэлементным объектом.

Функция проверки максимальной проекции немного сложнее (здесь блок `where` используется для локального — только в рамках данного выражения — определения функции `is-max-Maybe`):

```
_is-max : SO → Bool
_is-max s = is-max-Maybe (lbl s)
  where
  is-max-Maybe : Maybe Label → Bool
  is-max-Maybe (_ • _ • _ • []) = true
  is-max-Maybe _ = false
```

Функция проверяет, равно ли `uF` метки объекта пустому списку (то есть верно ли, что объект не имеет непроверенных признаков).

Легко видеть, что синтаксический объект можно превратить в дерево вывода путём элиминации всех меток:

```
SO→DTree : SO → DTree
SO→DTree ⟦ x ⟧ = Select x
SO→DTree ⟦ _ - so1 - so2 ⟧ = Merge (SO→DTree so1) (SO→DTree so2)
```

(метки здесь не используются в результате, поэтому могут быть изображены подчерком).

Обратная операция требует функции вычисления меток, которую я обозначу `LMerge`. Тогда функция конверсии дерева вывода в синтаксический объект определяется следующим образом:

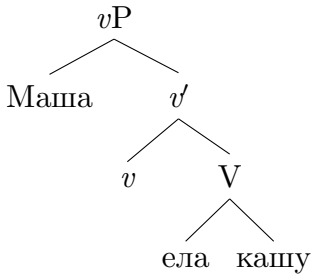
```
DTree→SO : DTree → SO
DTree→SO (Select x)     = ⟦ x ⟧
DTree→SO (Merge s1 s2)  = ⟦ LMerge  (lbl so1) (lbl so2) - so1 - so2 ⟧
  where
  so1 = DTree→SO s1
  so2 = DTree→SO s2
```

(как и выше, блок `where` используется для локального определения величин `so1` и `so2`). Таким образом, листья деревьев преобразуются в одноэлементные объекты ⟦ x ⟧, а к остальным узлам добавляется вычисляемая метка. Построение проводится рекурсивно, начиная с листьев.

Функция `LMerge` вычисляет метку сложного объекта на основе меток его составляющих. Заметим, что у нас нет ограничения на операцию Merge, она может быть применена к любым объектам. Однако в некоторых случаях метка при таком применении оказывается равна `nothing`, что указывает на то, что операция в данном случае синтаксически некорректна. Не всякий вывод приводит к корректному синтаксическому объекту.

`LMerge` содержит в себе алгоритм вычисления меток (называемый в минималистской теории проекцией). Я не буду здесь приводить её определение, достаточно сказать, что в настоящий момент она реализует следующий простой алгоритм. Если один из соединяемых объектов — максимальная проекция, а другой — нет, то функция ищет, имеется ли в первом признак, проверяющий признак для второго. Если этот так, то признак удаляется из метки последнего, и она становится меткой результирующего объекта. В противном случае метка равна `nothing`. Если оба объекта — максимальные проекции, то это интерпретируется как операция присоединения (Adjoin, которая, однако, сама по себе отсутствует в минималистской теории), и метка одного из объектов (по соглашению, второго) становится меткой результата. Наконец, если ни один из объектов не является максимальной проекцией, соединение считается неудавшимся и метка результата оказывается равной `nothing`.

Рассмотрим в качестве примера глагольную группу `vP` *Маша ела кашу*. Она изображается следующим деревом (с небольшим упрощением):



Определим нужные лексемы и затем массив лексем или перечень:

```
Маша^li = mkLI ("Маша" :: []) [] (N :: fem :: [])       []
ела^li  = mkLI ("ела" :: [])  [] (V :: past :: [])      (N :: [])
кашу^li = mkLI ("кашу" :: []) [] (N :: fem :: ACC :: []) []
v^li    = mkLI ("v" :: [])    [] (v^f :: [])            (V :: N :: [])

ела^la  = [ ела^li ]
v^la    = v^li :: [ ела^li ]
Маша^la = Маша^li :: v^li :: [ ела^li ]
кашу^la = кашу^li :: Маша^li :: v^li :: [ ела^li ]
```

(индексы ^{li} и ^{la} обозначают, соответственно, `LexItem` и `LexArr`). Кроме того, для удобства введём обозначения для одноэлементных синтаксических объектов:

```
Маша = ⟦ Маша^la ⟧
ела  = ⟦ ела^la ⟧
кашу = ⟦ кашу^la ⟧
v    = ⟦ v^la ⟧
```

Тогда дерево вывода выглядит следующим образом:

```
dt = Merge (Select Маша^la)
           (Merge (Select v^la)
                  (Merge (Select ела^la)
                         (Select кашу^la)))
```

После перевода его в синтаксический объект получаем:

```
so ≡ ⟦ ("v" :: []) • [] • v^f :: [] • []
       - Маша
       - ⟦ ("v" :: []) • [] • v^f :: [] • N :: []
           - v
           - ⟦ ("ела" :: []) • [] • V :: past :: [] • []
               - ела
               - кашу ⟧ ⟧ ⟧
```

(здесь и далее метки вычисляются с помощью описанной выше функции `LMerge`). В этом выражении ≡ обозначает доказываемое равенство, в отличие от равенства по определению =. Доказательство проводится путём редукции двух выражений к одному и тому же. Таким образом, выражение выше означает, что `so` сводится, редуцируется к выражению справа от знака ≡. Как можно видеть, оно соответствует дереву `dt`, но с добавленными метками.

Для работы с синтаксическими объектами очень полезны функции, позволяющие рекурсивно пробегать по деревьям, выполняя те или иные действия. Рассмотрим две такие функции.

```
foldr : {A : Set} → (SO → A → A → A) → (SO → A) → SO → A
foldr _ g ⟦ x ⟧ = g ⟦ x ⟧
foldr f g ⟦ l - s1 - s2 ⟧ = f ⟦ l - s1 - s2 ⟧ (foldr f g s1)
                              (foldr f g s2)

foldl : {A : Set} → (SO → SO → A → A) → A → SO → A
foldl _ a ⟦ _ ⟧ = a
foldl f a ⟦ _ - s1 - s2 ⟧ = foldl f (foldl f (f s1 s2 a) s1) s2
```

Эти функции вычисляют значение типа `A`, рекурсивно пробегая по всему дереву объекта. Функция `foldr` пробегает по дереву снизу, от листьев, применяя две функции типов

$$\texttt{f : SO → A → A → A},$$
$$\texttt{g : SO → A}.$$

При движении от листьев к корню функция `g` вычисляет значение на листьях, а функция `f` — в узлах, пользуясь объектом узла и уже посчитанными значениями для двух его составляющих.

Функция `foldl`, напротив, пробегает дерево сверху, от корня, применяя функцию типа

$$\texttt{f : SO → SO → A → A}.$$

Помимо обрабатываемого объекта, `foldl` имеет в качестве аргумента эту функцию и значение типа `A`, которое используется при вычислениях в качестве начального. На каждом узле, начиная с корня, `foldl` использует `f`, два объекта-составляющих и начальное значение для получения нового значения. С этим значением в качестве начального функция продолжает вычисления по левой ветке. После того как левая ветка будет обработана, вычисленный результат используется как начальное значение для вычисления правой ветки. При вычислениях на листьях (не имеющих составляющих) значение не изменяется. Таким способом последовательно проходится всё дерево.

Используя эти функции, можно, например, вычислить количество узлов в дереве:

```
nodesNum : SO → ℕ
nodesNum so = foldl f 1 so
  where
  f : SO → SO → ℕ → ℕ
  f _ _ n = (suc (suc n))
```

Можно также вычислить количество рёбер:

```
edgesNum : SO → ℕ
edgesNum so = foldr f g so
  where
  f : SO → ℕ → ℕ → ℕ
  f _ n m = (suc n) + (suc m)

  g : SO → ℕ
  g _ = zero
```

Более интересное применение мы находим в вычислении всех *составляющих* (или частей, или подобъектов) синтаксического объекта ($::^{r}$ обозначает операцию присоединения элемента к списку справа, а не слева, как в случае $::$):

```
Constituents : SO → List SO
Constituents so = foldl f (so :: []) so
  where
  f : SO → SO → List SO → List SO
  f so1 so2 sos = sos ::ʳ so1 ::ʳ so2
```

Например, для примера выше список составляющих выглядит следующим образом:

```
Constituents so ≡ so
                 :: Маша
                 :: ⟦ ("v" :: []) • [] • vᶠ :: [] • N :: []
                      - v
                      - ⟦ "ела" :: []) • [] • V :: past :: [] • []
                          - ела
                          - кашу ⟧ ⟧
               :: v
               :: ⟦ ("ела" :: []) • [] • V :: past :: [] • []
                    - ела
                    - кашу ⟧
               :: ела
               :: кашу
               :: []
```

Отношение «*содержит*» для синтаксических объектов можно было бы определить на основе этого списка составляющих, но мы сделаем это более прямым образом:

```
data _contains_ : SO → SO → Set where
  c0  : {s : SO} → s contains s
  _cl : {s s1 s2 : SO} {l : Maybe Label} →
        (s contains ⟦ l - s1 - s2 ⟧) → s contains s1
  _cr : {s s1 s2 : SO} {l : Maybe Label} →
        (s contains ⟦ l - s1 - s2 ⟧) → s contains s2
```

Здесь определяется отношение `s0 contains s` (которое является также типом или пропозицией), означающее «`s0` содержит (в себе) `s`». Эта пропозиция имеет три вида доказательств (или три конструктора таких доказательств):

- `c0` — всякий объект содержит сам себя;
- `x cl` — если `x` — доказательство того, что `s` содержит `⟦ l - s1 - s2 ⟧`, то `s` содержит `s1`;
- `x cr` — если `x` — доказательство того, что `s` содержит `⟦ l - s1 - s2 ⟧`, то `s` содержит `s2`.

Фактически доказательством пропозиции `s0 contains s` является путь по дереву от `s0` к `s`, выраженный как последовательность `cl` и `cr` (см. примеры ниже). Поэтому мы можем определить *путь* в дереве от одного объекта к другому как

```
path_to_ : SO → SO → Set
path s0 to s = s0 contains s
```

Путь позволяет нам различать вхождения одного и того же объекта в дерево, то есть то, что в минималистской грамматике называется копиями.

Кроме того, отношение `contains` позволяет нам, в частности, определить, является ли соединение объектов внешним (External Merge) или внутренним (Internal Merge). Предикат для соответствующей проверки нетрудно построить.

Назовём *положением* синтаксического объекта в дереве путь к этому объекту от корня дерева. Тип положений объекта определяется тогда следующим образом:

```
data Position (s0 : SO) : Set where
  pos : {s : SO} → s0 contains s → Position s0
```

Элементы `Position s0` являются путями из `s0`. Как видно, такой элемент конструируется для всякого `s`, такого, что `s0 contains s`. Будем считать положения «координатами» подобъектов внутри объекта. На этой основе возможно определить функцию, вычисляющую список всех положений объекта `s0`, то есть список всех его подобъектов (определения которой я здесь не привожу):

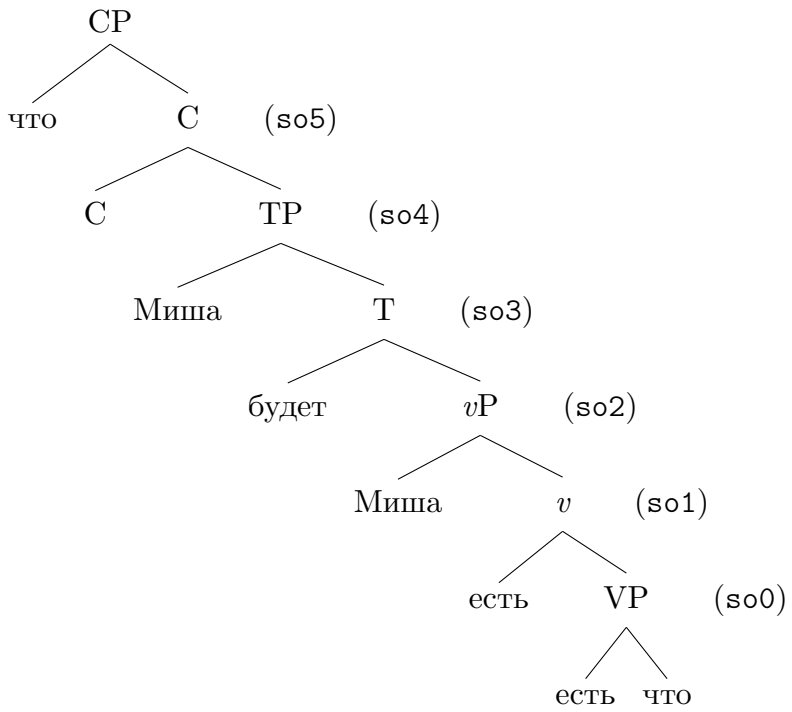


Рис. 1: Дерево для предложения *Что Миша будет есть?*

```
positions : (s0 : SO) → List (Position s0)
```

Рассмотрим, как различаются копии объектов на примере.

Формализуем предложение *Что Миша будет есть?* (Митренина, Слюсарь, Романова 2018: 137). Его дерево представлено на рис. 1 (в скобках для справки приведены имена узлов, используемые ниже). В нём копируются (или, нестрого говоря, передвигаются) три лексемы: *что*, *есть* и *Миша*. Для формализации сначала зададим лексикон и перечень:

```
что^li  = mkLI ("что" :: [])  [] (N :: [])           []
Миша^li = mkLI ("Миша" :: []) [] (N :: masc :: [])   []
есть^li = mkLI ("есть" :: []) [] (V :: [])           (N :: [])
v^li    = mkLI ("v" :: [])    [] (v^f :: [])         (V :: N :: [])
T^li    = mkLI ("T" :: [])    [] (T^f :: future :: []) (v^f :: N :: [])
C^li    = mkLI ("C" :: [])    [] (C^f :: [])         (T^f :: N :: [])

что^la  = [ что^li ]
есть^la = есть^li :: что^la
Миша^la = Миша^li :: v^la
v^la    = v^li :: есть^la
```

```
  Tla     = Tli :: Мишаla
  Cla     = Cli :: Tla
```

Дерево вывода и синтаксический объект выглядят следующим образом:

```
dt = Merge (Select чтоla)
           (Merge (Select Cla)
                  (Merge (Select Мишаla)
                         (Merge (Select Tla)
                                (Merge (Select Мишаla)
                                       (Merge (Select vla)
                                              (Merge (Select естьla)
                                                     (Select чтоla)
                                                     ))))))
```

```
so ≡ ⟦ ("C" :: []) • [] • Cf :: [] • []
       - что
       - ⟦ ("C" :: []) • [] • Cf :: [] • N :: []
           - C
           - ⟦ ("T" :: []) • [] • Tf :: future :: [] • []
               - Миша
               - ⟦ ("T" :: []) • [] • Tf :: future :: [] • N :: []
                   - T
                   - ⟦ ("v" :: []) • [] • vf :: [] • []
                       - Миша
                       - ⟦ ("v" :: []) • [] • vf :: [] • N :: []
                           - v
                           - ⟦ ("есть" :: []) • [] • V :: [] • []
                               - есть
                               - что        ⟧ ⟧ ⟧ ⟧ ⟧ ⟧ ⟧
```

Наконец, список всех положений подобъектов so выглядит следующим образом:

```
positions so ≡ pos c0 ::
               pos (c0 cl) ::
               pos (c0 cr) ::
               pos (c0 cr cl) ::
               pos (c0 cr cr) ::
               pos (c0 cr cr cl) ::
               pos (c0 cr cr cr) ::
               pos (c0 cr cr cr cl) ::
               pos (c0 cr cr cr cr) ::
               pos (c0 cr cr cr cr cl) ::
```

```
pos (c0 cr cr cr cr cr) ::
pos (c0 cr cr cr cr cr cl) ::
pos (c0 cr cr cr cr cr cr) ::
pos (c0 cr cr cr cr cr cr cl) ::
pos (c0 cr cr cr cr cr cr cr) :: []
```

Каждый путь здесь можно рассматривать как последовательность «инструкций» по движению от корня, предписывающих на соответствующем узле повернуть влево или вправо. Эти пути можно представить в более читаемом виде как последовательность узлов, которые мы проходим при таком движении. (Поскольку каждый узел синтаксического объекта сам является синтаксическим объектом, такая последовательность будет состоять опять же из синтаксических объектов.) Для этого сначала определим функцию, преобразующую путь в последовательность объектов:

```
Path→List : {s0 s : SO} → path s0 to s → List SO
Path→List {s0} c0 = s0 :: []
Path→List {_} {s} (p cl) = (Path→List p) ::ʳ s
Path→List {_} {s} (p cr) = (Path→List p) ::ʳ s
```

Затем — функцию, преобразующую позицию в последовательность объектов:

```
Pos→List : {s : SO} → Position s → List SO
Pos→List (pos p) = Path→List p
```

Тогда список всех позиций в виде последовательностей объектов можно определить как

```
positionsSO : SO → List (List SO)
positionsSO so = map Pos→List (positions so)
```

Здесь `map` — стандартная функция языка Agda, имеющая тип (с некоторым упрощением)

```
map : {A : Set} → (A → A) → List A → List A
```

Её применение `map f L` преобразует каждый элемент списка `L` с помощью функции `f`. Таким образом, в определении выше `map` применяет функцию преобразования `Pos→List` ко всем элементам списка `positions so`. Нужно сказать, что эту функцию можно также определить непосредственно, с помощью свёртки `foldr`, но мы здесь не будем этого делать.

Обозначив промежуточные узлы, как указано на рис. 1, получим преобразованный список:

```
positionsSO so ≡
      (so :: [])
    :: (so :: что :: [])
    :: (so :: so5 :: [])
    :: (so :: so5 :: C :: [])
    :: (so :: so5 :: so4 :: [])
    :: (so :: so5 :: so4 :: Миша :: [])
    :: (so :: so5 :: so4 :: so3 :: [])
    :: (so :: so5 :: so4 :: so3 :: T :: [])
    :: (so :: so5 :: so4 :: so3 :: so2 :: [])
    :: (so :: so5 :: so4 :: so3 :: so2 :: Миша :: [])
    :: (so :: so5 :: so4 :: so3 :: so2 :: so1 :: [])
    :: (so :: so5 :: so4 :: so3 :: so2 :: so1 :: v :: [])
    :: (so :: so5 :: so4 :: so3 :: so2 :: so1 :: so0 :: [])
    :: (so :: so5 :: so4 :: so3 :: so2 :: so1 :: so0 :: есть :: [])
    :: (so :: so5 :: so4 :: so3 :: so2 :: so1 :: so0 :: что :: [])
    :: []
```

Как можно видеть, некоторым лексемам соответствует более одного пути, поскольку для них имеются копии в дереве. Например, так выглядят два пути к лексеме Миша:

```
p1 : path so to Миша
p1 = c0 cr cr cl

p2 : path so to Миша
p2 = c0 cr cr cr cr cl
```

Цепью называется последовательность положений одного и того же объекта (то есть его копий). Она вычисляется функцией chainSO, имеющей тип

```
chainSO : SO → SO → List (List SO)
```

и определённой путём фильтрации путей positionsSO s0 по признаку окончания на объекте s. Таким образом, chainSO s s0 имеет значением список путей объекта s внутри объекта s0. Например, мы имеем для объекта Миша:

```
chainSO Миша so ≡ (so :: so5 :: so4 :: Миша :: []) ::
                  (so :: so5 :: so4 :: so3 :: so2 :: Миша :: []) ::
                  []
```

Можно проверить, что этот список содержит в точности два пути к лексеме Миша, приведённых выше (после их преобразования в List SO).

Сестрой узла в дереве называется узел, с которым он имеет общего родителя. Это несложно определить:

```
sister : {s0 : SO} → Position s0 → Maybe (Position s0)
sister (pos c0) = nothing
sister (pos (x cl)) = just (pos (x cr))
sister (pos (x cr)) = just (pos (x cl))
```

Узел не всегда имеет сестру, поэтому тип значения этой функции равен `Maybe (Position s0)`. Аналогично определяется `sisterSO`, так что, например, для сестёр узлов `Миша` имеем:

```
sisterSO (pos p1) ≡ just so3

sisterSO (pos p2) ≡ just so1
```

Наконец, определим ещё одно важное понятие — *с-командование* (c-command). Объект в дереве с-командует своей сестрой и всеми объектами, содержащимися в сестре. Наш формализм позволяет достаточно просто определить список позиций всех подчинённых таким образом объектов:

```
c-commanded : {s0 : SO} → Position s0 → List (Position s0)
c-commanded (pos c0) = []
c-commanded (pos (x cl)) = map (prependPath (x cr))
                               (positions (Pos→SO (pos (x cr))))
c-commanded (pos (x cr)) = map (prependPath (x cl))
                               (positions (Pos→SO (pos (x cl))))
```

В этом определении `positions...` содержит пути, начиная с сестры рассматриваемого объекта (`Pos→SO` вычисляет последний объект пути), а `map prependPath...` добавляет к ним путь до корня, получая список путей до «с-командуемых» объектов от корня `s0`. Аналогичным образом определяется

```
c-commandedSO : {s0 : SO} → Position s0 → List (List SO)
```

Например, для вхождения `p2` лексемы `Миша` мы получаем:

```
c-commandedSO (pos p2) ≡
    (so :: so5 :: so4 :: so3 :: so2 :: so1 :: [])
  :: (so :: so5 :: so4 :: so3 :: so2 :: so1 :: v :: [])
  :: (so :: so5 :: so4 :: so3 :: so2 :: so1 :: so0 :: [])
  :: (so :: so5 :: so4 :: so3 :: so2 :: so1 :: so0 :: есть :: [])
  :: (so :: so5 :: so4 :: so3 :: so2 :: so1 :: so0 :: что :: [])
  :: []
```

В заключение рассмотрим более сложный пример, в котором передвижению подвергается не просто лексема, а целое поддерево. Возьмём предложение *That*

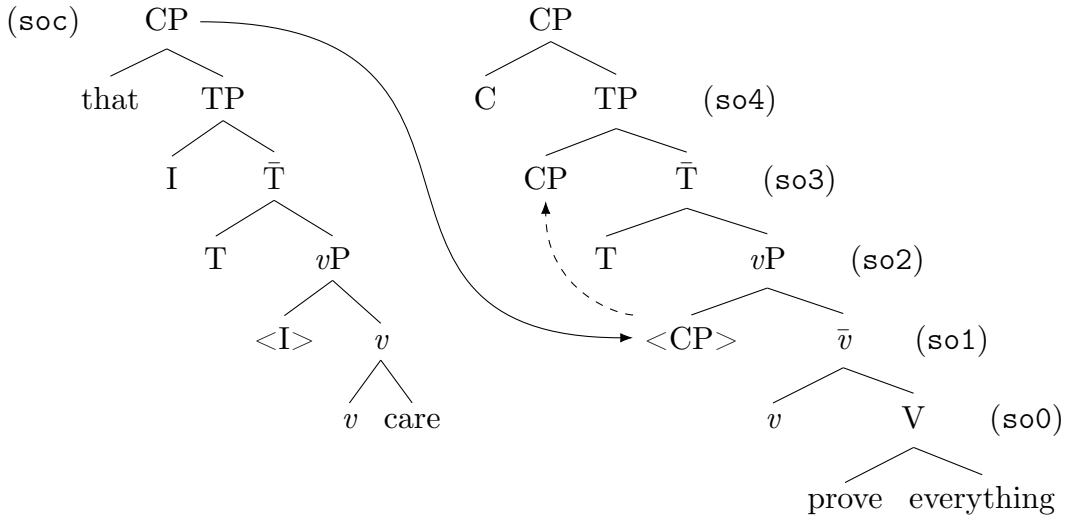


Рис. 2: Предложение *That I care proves everything*

I care proves everything (ср. Adger 2003: 246). Его дерево изображено на рис. 2. Здесь клауза *that I care* присоединяется к дереву на месте спецификатора *v*P и затем копируется (передвигается) на место субъекта в TP. Начнём формализацию с клаузы. Сначала, как и раньше, зададим лексикон:

```
that^li = mkLI ("that" :: []) [] (C^f :: []) (T^f :: [])
I^li    = mkLI ("I" :: [])    [] (N :: [])  []
care^li = mkLI ("care" :: []) [] (V :: [])  []
v^li    = mkLI ("v" :: [])    [] (v^f :: []) (V :: N :: [])
T^li    = mkLI ("T" :: [])    [] (T^f :: []) (v^f :: N :: [])
C^li    = mkLI ("C" :: [])    [] (C^f :: []) (T^f :: [])

that^la = [ that^li ]
I^la    = I^li :: that^la
care^la = care^li :: I^la
v^la    = v^li :: care^la
T^la    = T^li :: v^la

that = ⟦ that^la ⟧
I    = ⟦ I^la ⟧
care = ⟦ care^la ⟧
v    = ⟦ v^la ⟧
T    = ⟦ T^la ⟧
```

Затем дерево вывода и соответствующий объект, который назовём `soc`:

```
dtc = Merge (Select that^la)
            (Merge (Select I^la)
                   (Merge (Select T^la)
                          (Merge (Select I^la)
                                 (Merge (Select v^la)
                                        (Select care^la)))))

soc ≡ ⟦ ("that" :: []) • [] • (C^f :: []) • []
        - that
        - ⟦ "T" :: []) • [] • (T^f :: []) • []
            - I
            - ⟦ ("T" :: []) • [] • (T^f :: []) • (N :: [])
                - T
                - ⟦ ("v" :: []) • [] • (v^f :: []) • []
                    - I
                    - ⟦ ("v" :: []) • [] • (v^f :: []) • (N :: [])
                        - v
                        - care ⟧ ⟧ ⟧ ⟧ ⟧
```

Далее формализуем всё предложение:

```
prove^li      = mkLI ("prove" :: [])      [] (V :: [])  (N :: [])
everything^li = mkLI ("everything" :: []) [] (N :: [])  []
v2^li         = mkLI ("v" :: [])          [] (v^f :: []) (V :: C^f :: [])
T2^li         = mkLI ("T" :: [])          [] (T^f :: []) (v^f :: C^f :: [])

prove^la      = prove^li :: T^la
everything^la = everything^li :: prove^la
v2^la         = v2^li :: everything^la
T2^la         = T2^li :: v2^la
C2^la         = C^li :: T2^la

prove      = ⟦ prove^la ⟧
everything = ⟦ everything^la ⟧
v2         = ⟦ v2^la ⟧
T2         = ⟦ T2^la ⟧
C2         = ⟦ C2^la ⟧

dt = Merge (Select C2^la)
           (Merge dtc
                  (Merge (Select T2^la)
                         (Merge dtc
```

```
                                  (Merge (Select v2ˡᵃ)
                                         (Merge (Select proveˡᵃ)
                                                (Select everythingˡᵃ)
                                                )))))

so ≡ ⟦ ("C" :: []) • [] • (Cᶠ :: []) • []
       - C2
       - ⟦ "T" :: []) • [] • (Tᶠ :: []) • []
           - soc
           - ⟦ ("T" :: []) • [] • (Tᶠ :: []) • (Cᶠ :: [])
               - T2
               - ⟦ ("v" :: []) • [] • (vᶠ :: []) • []
                   - soc
                   - ⟦ ("v" :: []) • [] • (vᶠ :: []) • (Cᶠ :: [])
                       - v2
                       - ⟦ ("prove" :: []) • [] • (V :: []) • []
                           - prove
                           - everything ⟧ ⟧ ⟧ ⟧ ⟧ ⟧
```

Как видно, объект `soc` присутствует здесь в двух местах.

Обратим внимание, что `v2`, `T2` и `C2` полного предложения отличаются от соответствующих объектов клаузы. Действительно, они соответствуют разным элементам массива `LexArr`, то есть разным операциям выбора из лексикона (`Select`). И кроме того, `T` и `T2` отличаются здесь требованиями к субъекту: `T` требует имени (в данном случае, местоимения), а `T2` — клаузы.

В итоге мы можем вычислить цепь для поддерева `soc`:

```
chainSO soc so ≡ (so :: so4 :: soc :: [])
               :: (so :: so4 :: so3 :: so2 :: soc :: [])
               :: []
```

Она имеет два элемента соответственно двум вхождениям этого поддерева.

3. Заключение

Как видим, описанная формализация позволяет определить основные понятия минималистской теории и проводить вычисления с синтаксическими объектами. Она схватывает основные свойства операции Merge. Пути, понятые как «координаты», позволяют различать вхождения подобъектов в дерево и описывать копии или перемещения. Алгоритм вычисления меток или проекции содержится в функции `LMerge` и может варьироваться в различных вариантах минималистских теорий. Тем самым, формализм можно рассматривать как совокупность инструментов для

описания таких теорий. В то же время в формализме отсутствует Agree, другая важная операция минималистской теории, а также тесно связанное с ней понятие фазы. Её формализация затруднена отсутствием общепризнанной теории, однако можно надеяться, что определённое ядро различных пониманий Agree может всё же быть схвачено описанным выше подходом.

Литература

Митренина, Слюсарь, Романова 2018 — *Митренина О. В.*, *Слюсарь Н. А.*, *Романова Е. Е.* Введение в генеративную грамматику. М. : Ленанд, 2018.

Adger 2003 — *Adger D.* Core Syntax : A Minimalist Approach. Oxford University Press, 2003.

beim Graben, Gerth 2012 — *beim Graben P.*, *Gerth S.* Geometric Representations for Minimalist Grammars // Journal of Logic, Language, and Information. 2012. Vol. 21, no. 4. P. 393–432.

Boeckx 2006 — *Boeckx C.* Linguistic Minimalism: Origins, Concepts, Methods, and Aims. Oxford University Press, USA, 2006.

Chomsky 2013 — *Chomsky N.* Problems of projection // Lingua. 2013. Vol. 130. P. 33–49.

Chomsky 2015 — *Chomsky N.* Problems of projection: Extensions // Structures, strategies and beyond: Studies in honor of Adriana Belletti. Amsterdam/New York : John Benjamins, 2015. С. 1—16.

Chomsky 1995 — *Chomsky N.* The Minimalist Program. Cambridge, London : The MIT Preee, 1995.

Collins, Stabler 2016 — *Collins C.*, *Stabler E.* A Formalization of Minimalist Syntax // Syntax. 2016. Mar. Vol. 19, no. 1. P. 43–78.

Martin-Löf 1984 — *Martin-Löf P.* An Intuitionistic Type Theory : Notes by Giovanni Sambin of a series of lectures given in Padua, June 1980. Napoli : Bibliopolis, 1984.

Norell 2009 — *Norell U.* Dependently Typed Programming in Agda // Advanced Functional Programming: 6th International School, AFP 2008, Heijen, The Netherlands, May 2008, Revised Lectures. Berlin, Heidelberg : Springer-Verlag Berlin Heidelberg, 2009. P. 230–266.

Smith, Mursell, Hartmann 2020 — *Smith P. W.*, *Mursell J.*, *Hartmann K.*, eds. Agree to Agree: Agreement in the Minimalist Programme. Berlin : Language Science Press, 2020.

Stabler 1996 — *Stabler E.* Derivational minimalism // LACL 1996. Vol. 1328. Heidelberg : Springer, 1996. P. 68–95.

Stabler 2013 — *Stabler E.* Two Models of Minimalist, Incremental Syntactic Analysis // Topics in cognitive science. 2013. Vol. 5, no. 3. P. 611–633.