Бреслав А. А.

Синтез механизмов композиции в предметно-ориентированных языках

Оглавление

Введение 2
§ 1. Постановка задачи 2
Глава 1. Предварительные сведения 3
§ 1. Объектно-ориентированные модели 3
1.1. Синтаксис моделей 4
1.2. Классы и типы 12
1.3. Модели и мета-модели 21
§ 2. Языки как множества моделей 27
2.1. Нотация для контекстно-свободных грамматик 28
2.2. Метамодели, эквивалентные грамматикам 30
Глава 2. Расширение предметно-ориентированных языков механизмами композиции 33
§ 1. Автоматическое построение языков, поддерживающих типизированные макроопределения 33
1.1. Неформальное описание механизма композиции, основанного на макроопределениях 34
1.2. Шаблоны в языках, порожденных метамоделями 37
Приложение 1. Преобразование объектного представления метамоделей в классы 40
Приложение 2. Нотация для контекстно-свободных грамматик 45
Список литературы 47

Введение

§ 1. Постановка задачи

Целью настоящей работы является **автоматизация разработки механизмов композиции в предметно-ориентированных языках**. Для достижения поставленной цели необходимо решить следующие задачи:

- Разработать автоматизированный метод расширения предметно-ориентированных языков поддержкой типизированных макроопределений.
- Разработать автоматизированный метод расширения предметно-ориентированных языков поддержкой аспектов.
- Формально описать разработанные методы и обосновать их корректность.
- Продемонстрировать применение разработанных методов, применив их к предметно-ориентированному языку, полезному для решения практических задач.

Глава 1.
Предварительные сведения

В этой главе...

§ 1. Объектно-ориентированные модели

Объектно-ориентированные модели (для краткости, ниже мы будем писать просто “модели”) как средство описания предметных областей были предложены в начале 1990-х годов [**?**, **?**] и с течением времени были описаны консорциумом OMG (Object Managenemt Group, [**?**]) в виде ряда промышленных стандартов. Наибольшую известность среди “модельно-ориентированных” подходов получил унифицированный язык моделирования — UML (Unified Modelling Language, [13]); также очень широко используется библиотека EMF (Eclipse Modelling Framework, [14]), основанная на идеях, заложенных в спецификации MOF (Meta-Object Factility, [54]), являющейся частью UML (Infractructure). По сути, объектно-ориентированные модели — это структуры, не привязанные к конкретному языку программирования и представляющие данные (и способы их обработки) в объектно-ориентированном стиле.

Следуя работам [**?**, 83], в последующих разделах мы будем использовать модели как универсальный способ описания предметно-ориентированных языков, однако на этом пути нас ждет небольшое затруднение: промышленные стандарты, упомянутые выше, не предоставляют математической базы для работы с моделями. В литературе описаны различные подходы к формализации этой области [**?**, **?**, **?**], в основном нацеленные на описание полной MOF. Использование этих формализаций затруднительно в первую очередь потому, что MOF —

это довольно объемная система, а следовательно, и формальные теории, ее описывающие, получаются достаточно громоздкими. С другой стороны, на практике, при реализации предметно-ориентированных языков, как правило, используется не вся MOF, а лишь небольшая ее часть, “ядро”, которое называется EMOF (Essential MOF, [**?**]), именно ей соответствует реализация моделей, предоставляемая популярной библиотекой EMF, упомянутой выше. В данном разделе строится формальная теория, описывающая объектно-ориентированные модели, соответствующие EMOF.

1.1. Синтаксис моделей

Чтобы строго определить понятие модели, мы вводим ряд вспомогательных определений, начиная со структурного представления и постепенно вводя ограничения.

Определение 1.1 (Модельный терм)**.** *Модельным термом* (MT) называется формула, построенная по следующим правилам[1]:

$$\begin{aligned}
\mathrm{MT} \;&::=\; \mathbf{object}\ \mathrm{MT}_{id} : \mathrm{MT}_{cl}\ \{\mathrm{P}_1, \ldots, \mathrm{P}_n\}\,, n \geq 0 \\
&\mid\; @\mathrm{MT}_{id} \\
&\mid\; [\mathrm{MT}_1, \ldots, \mathrm{MT}_n]\,, n \geq 0 \\
&\mid\; \{\mathrm{MT}_1, \ldots, \mathrm{MT}_n\}\,, n \geq 0 \\
&\mid\; \mathtt{null} \mid \Sigma^* \mid \mathbb{Z} \mid \mathbb{B}, \\
\mathrm{P} \;&::=\; \mathrm{MT}_p = \mathrm{MT}_v
\end{aligned}$$

где $\mathbb{Z}$ — множество целых чисел, Σ — некоторый конечный алфавит символов, а $\mathbb{B} = \{\mathtt{true}, \mathtt{false}\}$. Вспомогательное правило для P описывает *свойства объектов* (см. ниже).

[1] Для краткости мы пишем $t_1, \ldots, t_n\,,\ n \geq 0$ вместо использования рекурсивных продукций вида

$$tList ::= \varepsilon \mid t \mid t\ ,\ tList$$

Записывая модельные термы, мы, как правило, не будем заключать строки в кавычки, а будем лишь выделять их начертанием, чтобы отличать от *метапеременных*: так, `abc` — это строка из трех символов, а x — метапеременная, которая, в частности может принимать значение `abc`.

Для различных видов модельных термов мы будем использовать специальные названия. Приведем примеры таких термов и их названия:

- $\{\texttt{abc}, 239, \texttt{null}\}$ — *множество*, состоящее из трех констант: строки, числа и специального значения `null`;
- $[1, 2, 3]$ — *список* из трех чисел;
- `@x` — *ссылка* на идентификатор `x`.
- **object** `a` : `@b`$\{\texttt{c} = \texttt{false}\}$ — *объект*, где `a` — *идентификатор*, `@b` — *ссылка на класс*, а `c` $=$ `false` — *свойство* `c` *маркером* `c` и *значением* `false`.

Метапеременные позволяют кратко описать множества термов, обладающих схожей структурой. Если в модельном терме используется метапеременная a, это означает, что вместо нее может быть подставлен любой подтерм, синтаксически допустимый в данном контексте. Так, например, запись $\{a, b\}$ описывает все термы-множества, состоящие из двух элементов, а $@a$ — ссылки на все возможные идентификаторы.

Модельные термы удобны для представления структур абстрактного синтаксиса различных языков [**?**, 83]. В качестве примера рассмотрим λ-исчисление [**?**]. Абстрактный синтаксис λ-исчисления

задается тремя конструкторами:

Обозначение	Пример	Название
$LT ::= \mathbf{Abs}(v, LT)$	$\lambda\, x\, .\, t$	*абстракция*
$\mid \mathbf{App}(LT_1, LT_2)$	$(f\, x)$	*применение*
$\mid \mathbf{Var}(v)$	x	*переменная*

где v пробегает алфавит имен переменных. Например, λ-терм $\lambda\, x\, .\, x\, x$ в абстрактном синтаксисе записывается как $\mathbf{Abs}(x, \mathbf{App}(\mathbf{Var}(x), \mathbf{Var}(x)))$, что соответствует *абстрактному синтаксическому дереву* (Abstract Syntax Tree, AST [**?**]) для данного терма. То же самое дерево можно представить также в виде объектов; см. Рис. 1.1.

```
object t1 : @Abstraction {
  var = object x : @Variable {name = "x"} ,
  body = object t2 : @Application {
    funciton = object t3 : @Usage {var = @x} ,
    argument = object t4 : @Usage {var = @x}
  }
}
```

Рис. 1.1: Представление AST для терма $\lambda\, x\, .\, x\, x$ в виде объектов

Из Рис. 1.1 видно, что объекты модельного терма образуют дерево, соответствующее абстрактному синтаксическому представлению λ-терма, причем роль конструкторов выполняют ссылки на классы. Обратите внимание на использование ссылок в объектах `t3` и `t4`: значение свойства `var` является *ссылкой* на идентификатор объекта, соответствующего определению переменной `x`.

Модельный терм можно рассматривать как граф: объекты являются вершинами, а отношение вложенности и ссылки — ребрами. AST является остовным деревом в этом графе — в нем участвуют только ребра вложенности объектов.

1.1.1. Графическая нотация для модельных термов

Наравне с нотацией, введенной выше, мы будем использовать для изображения объектов графическую нотацию, основанную на диаграммах объектов языка UML [13]. На Рис. 1.2 в виде такой диаграммы изображены объекты, приведенные на Рис. 1.1.

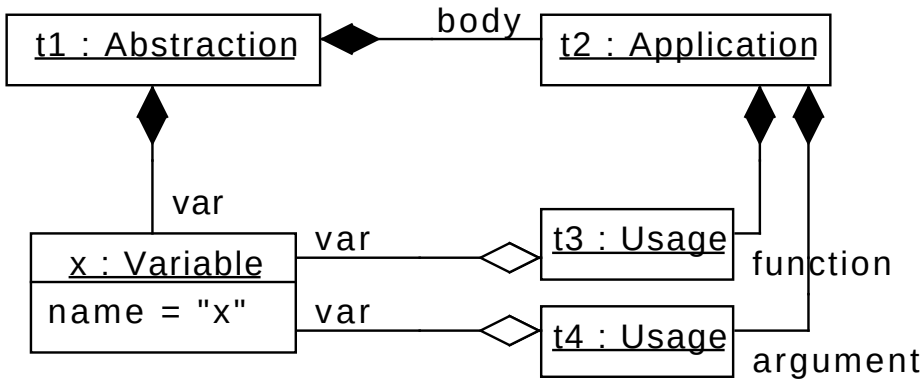

Рис. 1.2: Графическое представление для терма $\lambda x . x x$

Объекты обозначаются прямоугольниками. В верхней части прямоугольника располагается подчеркнутый текст: это идентификатор объекта и *имя класса*. О классах подробно говорится ниже, а пока лишь заметим, что объекты, относящиеся к одному классу, имеют одинаковую структуру. Значения свойств изображаются либо под горизонтальной чертой в самом прямоугольнике (как для объекта `x`), либо в виде ребер графа. Ближе к концу ребра написано имя свойства, а в начале ребра изображается ромб: незакрашенный, если ребро соответствует ссылке[2], и закрашенный, если значением свойства является объект, то есть имеет место *встраивание*[3] — один объект является частью другого. Таким образом, если рассматривать только ребра с закрашенными ромбами, из графа объектов выделяется дерево, соответствующее текстовой нотации, введенной выше (см. Рис. 1.1).

[2] Что соответствует *агрегации* в терминах UML.

[3] *Композиция* — в терминах UML

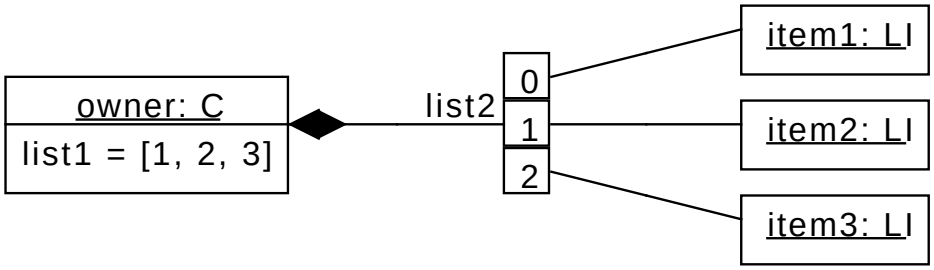

Рис. 1.3: Графическое представление упорядоченных списков

Если значением свойства является множество, оно изображается несколькими ребрами с одинаковой пометкой. В случае списка важен порядок следования элементов, поэтому мы отклоняемся от нотации UML и вводим промежуточный блок с индексами, от ячеек которого отходят ребра к элементам списка. Пример использования данной нотации приведен на Рис. 1.3: список чисел list1 записан непосредственно в теле объекта, а список (встраиваемых) объектов list2 изображен графически.

1.1.2. Конгруэнтность и правильно построенные термы

Определение 1.2 (Конгруэнтность)**.** Отношение *конгруэнтности* на модельных термах, $\cong \subset \mathrm{MT} \times \mathrm{MT}$, есть минимальное отношение эквивалентности, обладающее следующими свойствами:

1. Константы конгруэнтны сами себе:
   $x \cong x$, для любого $x \in \{\,\mathtt{null}\} \cup \mathbb{Z} \cup \mathbb{B} \cup \Sigma^*$;

2. Ссылки на конгруэнтные индентификаторы конгруэнтны:
   $@x \cong @y$, если $x \cong y$;

3. Списки сравниваются поэлементно:
   $[x_1, \ldots, x_n] \cong [y_1, \ldots, y_n]$, если $x_i \cong y_i$ для любого $i \in [1..n]$;

4. Множества сравниваются поэлементно, без учета порядка: $\{x_1, \ldots, x_n\} \cong \left\{x'_{\pi(1)}, \ldots, x'_{\pi(n)}\right\}$, если существует перестановка π размера n такая, что $x_i \cong x'_{\pi(i)}$ при $i \in [1; n]$.

5. Объекты сравниваются без учета порядка свойств:

$$\textbf{object}\ id_1 : c_2 \left\{ \begin{array}{c} p_1 = v_1, \\ \ldots, \\ p_n = v_n \end{array} \right\} \cong \textbf{object}\ id_2 : c_2 \left\{ \begin{array}{c} p'_{\pi(1)} = v'_{\pi(1)}, \\ \ldots, \\ p'_{\pi(n)} = v'_{\pi(n)} \end{array} \right\},$$

если $id_1 \cong id_2$, а также $c_1 \cong c_2$, и существует перестановка π размера n такая, что $p_i \cong p'_{\pi(i)}$ и $v_i \cong v'_{\pi(i)}$.

Приведем несколько примеров:

- $@abc \cong @abc$, поскольку идентификаторы конгруэнтны;
- $[1, 2] \ncong [2, 1]$, при сравнении списков порядок элементов важен;
- а при сравнении множеств — не важен: $\{1, 2\} \cong \{2, 1\}$;
- также не важен порядок свойств при сравнении объектов:

$$\textbf{object}\ abc : @c \left\{ \begin{array}{ccc} a & = & b \\ c & = & d \end{array} \right\} \cong \textbf{object}\ abc : @c \left\{ \begin{array}{ccc} c & = & d \\ a & = & b \end{array} \right\},$$

- но значения свойств важны:

$$\textbf{object}\ abc : @c \left\{ \begin{array}{ccc} a & = & b \\ c & = & d \end{array} \right\} \ncong \textbf{object}\ abc : @c \left\{ \begin{array}{ccc} c & = & b \\ a & = & d \end{array} \right\}.$$

Введем функцию $\mathcal{S}(t)$, вычисляющую множество всех подтермов терма t:

$$\mathcal{S}(t) \stackrel{\text{def}}{=} \{t\} \cup \begin{cases} \mathcal{S}(id) \cup \mathcal{S}(c) \cup \bigcup\limits_i \mathcal{S}(p_i) \cup \bigcup\limits_i \mathcal{S}(v_i), & \\ & t = \textbf{object}\ id : c\,\{p_1 = v_1, \ldots, p_n = v_n\} \\ \mathcal{S}(r), & t = @r \\ \bigcup\limits_i \mathcal{S}(x_i), & t = [x_1, \ldots, x_n] \text{ или } t = \{x_1, \ldots, x_n\} \\ \emptyset, & \text{в остальных случаях} \end{cases} \tag{1.1}$$

Пример: $\mathcal{S}\left(\mathbf{object}\ \mathsf{a} : @\mathsf{b}\,\{1 = [\mathtt{null}, \mathtt{false}]\}\right) = \{$
$\mathsf{a},\quad @\mathsf{b},\quad \mathsf{b},\quad 1,$
$[null, false]\,,\quad \mathtt{null},\quad \mathtt{false}$
$\mathbf{object}\ \mathsf{a} : @\mathsf{b}\,\{1 = [null, false]\}\}\,.$

Как правило, нам будут нужны множества подтермов какого-то определенного вида, например, множество подтермов терма t, являющихся ссылками, мы будем обозначать $\mathcal{S}_{ref}\,(t)$. Пример:

$$\mathcal{S}_{ref}\left(\mathbf{object}\ \mathsf{a} : @\mathsf{b}\,\{1 = [\mathtt{null}, \mathtt{false}]\}\right) = \{@\mathsf{b}\}.$$

Аналогично $\mathcal{S}_{set}\,(t)$ — для подтермов-множеств и $\mathcal{S}_{obj}\,(t)$ — для подтермов-объектов.

Определение 1.3. Модельный терм t называется *правильно построенным*, если выполняются следующие условия:

1. Среди подтермов t не существует двух объектов с конгруэнтными идентификаторами;
2. В $\mathcal{S}_{obj}\,(t)$ не существует объекта, содержащего два свойства с конгруэнтными именами;
3. В $\mathcal{S}_{set}\,(t)$ не существует множества, содержащего два конгруэнтных элемента.

Так, например, терм $[\mathbf{object}\ a : @c\,\{}\,, \mathbf{object}\ a : @c\,\{x = y\}]$ не является правильно построенным (пункт 1). Терм $\mathbf{object}\ \ a\ \ :\ \ @c\,\{a = b, a = c\}$ тоже не является правильно построенным, но по другой причине см. пункт 2. Согласно пункту 3, терм $\{1, 2, 1\}$ также не является правильно построенным.

1.1.3. Ссылочные контексты и пред-модели

Определение 1.4 (Ссылочный контекст)**.** Будем называть *ссылочным контекстом* частичную функцию $\Psi : \mathrm{MT} \to \mathrm{MT}$, обладающую следующими свойствами:

1. Если $\Psi(id)$ определено, то $\Psi(id) = \{\mathbf{object}\ id' : \ldots \{\ldots\}\}$, где $id \cong id'$;

2. $\Psi(id) \cong \Psi(id')$ тогда и только тогда, когда $id \cong id'$.

Другими словами, ссылочный контекст "находит" объект по его идентификатору. Для удобства мы будем писать $\Psi(id) = \bot$ в случае, если $\Psi(id)$ не определено.

Обозначим множество всех подтермов, *упомянутых* в контексте Ψ, то есть являющихся подтермами элементов области определения Ψ и/или области значений Ψ, через $\mathcal{S}(\Psi)$. Аналогично вводятся $\mathcal{S}_{ref}(\Psi)$ и $\mathcal{S}_{obj}(\Psi)$.

Определение 1.5 (Замкнутый контекст)**.** Будем называть ссылочный контекст Ψ *замкнутым*, если

$$o = \mathbf{object}\ id : \ldots \{\ldots\} \in \mathcal{S}_{obj}(\Psi) \Rightarrow \Psi(id) = o$$

и

$$r = @id \in \mathcal{S}_{ref}(\Psi) \Rightarrow \Psi(id) \neq \bot$$

Все объекты, упомянутые в замкнутом контексте, могут быть получены по своим идентификаторам, и каждая ссылка ссылается на существующий объект. Из этого, в частности, следует, что в таком контексте каждому объекту присвоен уникальный идентификатор. Так, например, следующий контекст является замкнутым (здесь функция рассматривается как множество пар вида "аргумент $\mapsto$ результат"):

$$\Psi(id) = \{a \mapsto \mathbf{object}\ a : @a\,\{\}\}$$

Приведем также пример незамкнутого контекста:

$$\Psi(id) = \{a \mapsto \mathbf{object}\ a : @b\,\{c = \mathbf{object}\ b : @a\,\{\}\}\}$$

в последнем примере нарушаются оба требования определения 1.5:

- объект, упомянутый в контексте, не принадлежит области его значений;
- ссылка, упомянутая в контексте, не принадлежит его области определения.

Определение 1.6. Пусть Ψ_1 и Ψ_2 – *согласованные* ссылочные контексты:

$$\forall x \in \mathbf{dom}(\Psi_1), y \in \mathbf{dom}(\Psi_2) : x \cong y \Rightarrow \Psi_1(x) \cong \Psi_2(y).$$

Тогда их *объединение* определяется следующей формулой:

$$(\Psi_1 \cup \Psi_2)(id) \stackrel{\text{def}}{=} \begin{cases} \Psi_1(id), & \Psi_1(id) \neq \bot \\ \Psi_2(id), & \Psi_2(id) \neq \bot \\ \bot, & \text{в противном случае.} \end{cases}$$

По модельному терму t можно построить ссылочный контекст, упоминающий все объекты, входящие в t:

$$\Psi_t \stackrel{\text{def}}{=} \{id \mapsto o \mid o = \mathbf{object}\ id : \ldots \{\ldots\} \in \mathcal{S}_{obj}(t)\}$$

Определение 1.7 (Пред-модель)**.** Будем называть правильно построенный модельный терм t *пред-моделью в контексте* Ψ_0, если контекст $\Psi_0 \cup \Psi_t$ является замкнутым.

Пред-модель обладает важным свойством: любой объект в ней однозначно определяется своим идентификатором, и любая ссылка указывает на известный объект.

1.2. Классы и типы

На практике для удобства навигации по моделям и контроля над их корректностью вводятся дополнительные ограничения (в виде системы типов), регулирующие структуру моделей. В этом разделе мы вводим понятие *класса* для объектов, которое является основой для таких ограничений.

Определение 1.8 (Класс)**.** *Классом* называется кортеж $\langle abs, id, S, P\rangle$, где

1. abs — признак абстрактного класса, принимает значения `true` или `false`; Если $abs = \texttt{false}$, класс называется *конкретным*, иначе — *абстрактным*,
2. id — *идентификатор* класса (имя),
3. S — множество классов, называемых *предками* данного класса (*суперклассов*),
4. P — множество пар вида $p : \tau$, где p — *идентификатор* свойства, а τ — *тип* свойства (множество типов определяется ниже).

Множество P не содержит различных пар, в которых совпадают идентификаторы свойств.

Для удобства, мы будем обозначать конкретный класс $\langle \texttt{false}, id, S, P\rangle$ через $\underline{\textbf{class}}\ id : S\,\{P\}$, а абстрактный класс $\langle \texttt{true}, id, S, P\rangle$ — через ***class*** $id : S\,\{P\}$. Когда значения признака абстрактного класса не важно или однозначно определяется из контекста, мы будем использовать обозначение $\textbf{class}\ id : S\,\{P\}$.

Определение 1.9. *Перечислением* называется кортеж $\langle id, \{L_1, \ldots, L_n\}\rangle$, где id — *идентификатор* перечисления, а L_i — *литералы* перечисления.

Мы будем обозначать перечисления следующим образом: $\textbf{enum}\ id\,\{L_1, \ldots, L_n\}$.

Определение 1.10. Множество типов $\mathfrak{T}$ описывается следующим об-

разом:

$$
\begin{aligned}
\mathfrak{T} \;::=\; & \mathsf{Char} \mid \mathsf{String} \mid \mathsf{Int} \mid \mathsf{Bool} \\
& \mid \mathfrak{T}^{?} \mid \{\mathfrak{T}^{+}\} \mid \{\mathfrak{T}^{*}\} \mid [\mathfrak{T}^{+}] \mid [\mathfrak{T}^{*}] \\
& \mid \mathbf{val}\,(c) \mid \mathbf{ref}\,(c)\,, \text{ где } c \text{ — идентификатор некоторого класса} \\
& \mid \mathbf{enum}\,(e)\,, \text{ где } e \text{ — идентификатор некоторого перечисления} \\
& \mid \top
\end{aligned}
$$

Неформально говоря, приведенное определение описывает следующие разновидности типов:

- Char, String, Int, Bool обозначают примитивные типы: символы, строки, целые числа и булевские значения, соответственно.

- Тип $\tau^{?}$ допускает значение `null` или значение типа τ, например, $\mathsf{Char}^{?}$ соответствует множеству значений $\Sigma \cup \{\texttt{null}\}$.

- $\{\tau^{*}\}$ обозначает тип множеств, содержащих ноль или более элементов типа τ. Например, $\{\mathsf{String}^{*}\}$ — это множества строк, а $\{\{\mathsf{Int}^{*}\}^{*}\}$ — множества множеств целых чисел.

- Аналогично, $[\tau^{*}]$ обозначает списки.

- Если вместо “*” используется “+”, то множество или список должны содержать не менее одного элемента. Например, $\{\tau^{+}\}$ — непустые множества, составленные из элементов τ.

- $\mathbf{val}\,(c)$, где c — некоторый класс (точнее, его идентификатор), соответствует *встраиваемому* объекту класса c. Напомним, *встраиванием* называется ситуация, при которой один объект входит в другой как подтерм, например: $\mathbf{object}\ a : @c\,\{a = \mathbf{object}\ b : @c\,\{\}\}$.

- $\mathbf{ref}\,(c)$ обозначает тип ссылок на объекты класса c, то есть значений вида $@a$, где a — идентификатор объекта, принадлежащего классу c (см. ниже).

- $\mathbf{enum}\,(e)$, где перечисление e определяется как $\mathbf{enum}\ \ e\,\{L_1, \ldots, L_n\}$ обозначает тип, значениями которого являются литералы $L_1, \ldots, L_n$ и только они.
- $\top$ (“Top”) — супертип всех остальных типов (см. ниже).

1.2.1. Графическая нотация для классов

Наравне с текстовой, мы будем использовать для классов графическую нотацию, основанную на диаграммах классов языка UML (см. Рис. 1.4). Класс обозначается прямоугольником, разделенным на две секции: верхняя секция содержит имя класса (*курсивом* в случае абстрактного класса), а нижняя — его свойства.

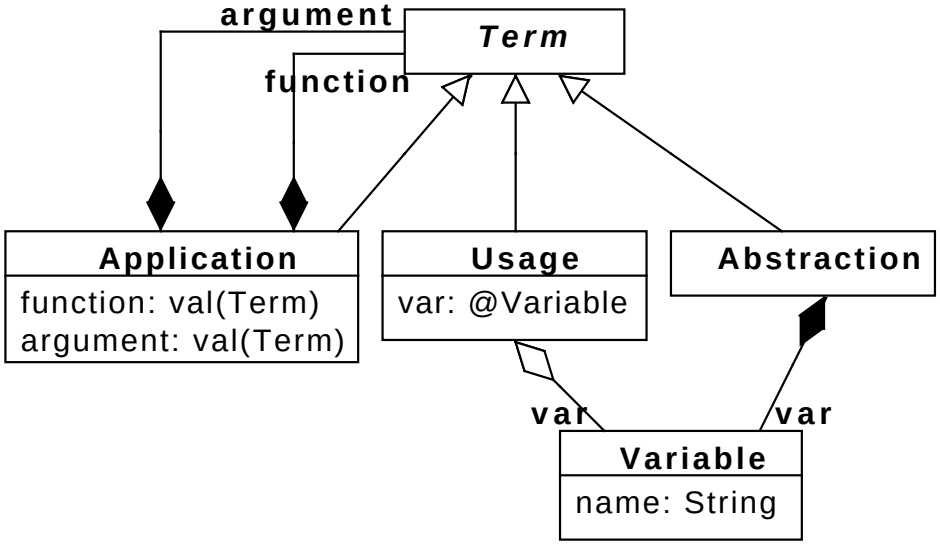

Рис. 1.4: Графическая нотация для классов λ-исчисления

Для обозначения отношения “подкласс-суперкласс” или *наследования* на диаграммах используются ребра с треугольником на конце, обозначающим суперкласс. Заметим, что циклов по наследованию и встраиванию на диаграмме классов быть не должно.

Как и в случае с объектами, некоторые свойства могут обозначаться ребрами (вместо или одновременно с записью в теле класса). Начало ребра обозначается закрашенным или незакрашенным ромбом — для встраивания ($\mathbf{val}\,(\tau)$) и ссылок ($\mathbf{ref}\,(\tau)$) соответственно, а имя свойства пишется ближе к концу ребра, причем для обозначения множеств и списков используется нотация, проиллюстрированная на Рис. 1.5. Для обозначения свойства $\mathsf{bset} : \{\mathbf{val}\,(\mathsf{B})^*\}$ на диаграм-

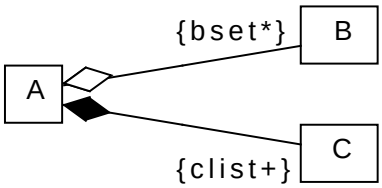

Рис. 1.5: Нотация для множеств и списков на диаграмме классов

ме изображается ребро от класса A, в котором объявлено свойство, к классу B, ближе к концу ребра пишется имя свойства — bset заключенное в фигурные скобки вместе со знаком "*", обозначающие что тип свойства — множество объектов класса B. На Рис. 1.5 начало ребра, изображающего свойство bset отмечено закрашенным ромбом, следовательно элементы множества встраиваются в объекты класса A, что завершает описание типа свойства — $\{\mathbf{val}\,(\mathsf{B})^*\}$. Аналогично, для обозначения списка $\mathsf{clist} : \left[\mathbf{ref}\,(\mathsf{C})^+\right]$ начало ребра отмечено незакрашенным ромбом, а в конце ребра мы пишем `[clist+]`.

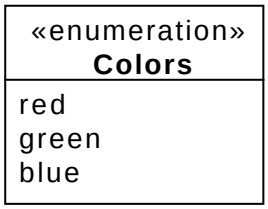

Рис. 1.6: Пример графической нотации для перечислений

Перечисления также отображаются в виде прямоугольников (см. Рис. 1.6), но над идентификатором пишется *стереотип* «`enumeration`», а под чертой перечисляются литералы.

1.2.2. Типы как множества значений

Для типа τ, множество упомянутых в данном типе классов определяется следующим образом:

$$\overline{C}(\tau) \stackrel{\text{def}}{=} \begin{cases} \{c\}, & \tau \in \{\mathbf{val}(c), \mathbf{ref}(c)\} \\ \overline{C}(\sigma), & \tau \in \left\{[\sigma^+], [\sigma^*], \{\sigma^+\}, \{\sigma^*\}, \sigma^?\right\} \\ \emptyset, & \text{в остальных случаях} \end{cases}$$

Пусть дано множество классов

$$C = \{\mathbf{class}\ id_i : S_i \left\{p_j^i : \tau_j^i \mid j \in [1; m_i]\right\} \mid i \in [1; n]\},$$

обозначим $\overline{S}(C) \stackrel{\text{def}}{=} C \cup \overline{S}\left(\bigcup_1^n S_i\right)$ множество всех классов из C, их предков, предков их предков и т.д.

Аналогично определяется множество всех свойств класса $c = \mathbf{class}\ id : S\{p_i : \tau_i\}$. Кроме свойств p_i, объявленных в самом c, рассматриваются также свойства, *унаследованные* из суперклассов: $\overline{P}(c) \stackrel{\text{def}}{=} \bigcup_{i=1}^{n} \{p_j^i : \tau_j^i \mid j \in [1; m_i]\} \cup \overline{P}\left(\overline{S}(c)\right)$. Определим также множество всех классов, встраиваемых в c:

$$\overline{C}_{comp}(c) \stackrel{\text{def}}{=} \{c' \mid \exists (p : \tau) \in \overline{P}(c), \text{ где } \tau \in \left\{\mathbf{val}(c'), \left\{\mathbf{val}(c')^+\right\}, \left[\mathbf{val}(c')^+\right]\right\}\}.$$

Определение 1.11. Конечный набор классов C называется *корректным*, если выполняются следующие свойства:

1. Все предки классов из C содержатся в C: $\overline{S}(C) = C$.
2. Все типы свойств классов из C упоминают только классы из C: пусть $c \in C$ и $\overline{P}(c) = \{p_i : \tau_i\}$, тогда $\bigcup_i \overline{C}(\tau_i) \subseteq C$.

3. Никакой класс из C не является прямо или косвенно предком самого себя: если $c = \mathbf{class}\ id : S\{\ldots\} \in C$, то $c \notin \overline{S}(S)$.

4. Никакой класс из C прямо или косвенно не встраивается сам в себя: $c \notin \overline{C}_{comp}(c)$.

На типах, порожденных корректным набором классов C, вводится отношение "подтип" ($\preceq$) — это наименьшее транзитивное рефлексивное отношение, обладающее следующими свойствами[4]:

$$\frac{\mathbf{class}\ c : S\{\ldots\} \in C \quad s \in S}{\mathbf{val}(c) \preceq \mathbf{val}(s) \wedge \mathbf{ref}(c) \preceq \mathbf{ref}(s)}\ subclass$$

Другими словами, подклассы порождают подтипы. Кроме того, тип $\tau^?$ шире типа τ, поскольку свойства этого типа могут принимать на одно значение — `null` — больше[5]:

$$\frac{}{\tau \preceq \tau^?}\ nullable$$

Аналогично для списков и множеств:

$$\frac{\tau \preceq \sigma \quad \iota, \kappa \in \{+, *\} \quad \iota \leq \kappa}{\{\tau^\iota\} \preceq \{\sigma^\kappa\}}\ set \qquad \frac{\tau \preceq \sigma \quad \iota, \kappa \in \{+, *\} \quad \iota \leq \kappa}{[\tau^\iota] \preceq [\sigma^\kappa]}\ list$$

Здесь отношение порядка на знаках $+$ и $*$ определяется правилом

$$+ < *.$$

Классы и типы используются для описания множеств допустимых модельных термов. Пусть зафиксирован замкнутый ссылочный контекст Ψ. Ниже мы определим функцию $[\![\bullet]\!]^C_\Psi : \mathfrak{T} \to 2^{\mathrm{MT}}$, которая для корректного набора классов C сопоставляет каждому типу множество его значений. Для этого мы сначала введем вспомогательную функцию $\lceil\bullet\rceil_\Psi : \mathfrak{T} \to 2^{\mathrm{MT}}$, которая возвращает все модельные

[4] Здесь и далее, следуя нотации изображения правил вывода в математической логике, мы записываем *посылки* над горизонтальной чертой, а *заключение* — под чертой; справа курсивом указывается имя или номер, на который мы будем ссылаться в тексте.

[5] Правила, не имеющие посылок над горизонтальной чертой, являются *аксиомами*, то есть выполняются для любых значений метапеременных.

термы, упомянутые в контексте Ψ, принадлежащие к данному типу и ни к какому другому, то есть не согласует отношение "подтип" с отношением вложенности множеств: $\tau \preceq \rho \nRightarrow \lceil\tau\rceil_\Psi \subseteq \lceil\rho\rceil_\Psi$. Эта функция задается следующим образом. Типу $\mathbf{val}\,(c)$ ставится в соответствие множество $\lceil\mathbf{val}\,(c)\rceil_\Psi$ всех объектов из $\mathcal{S}_{obj}\,(\Psi)$, в которых ссылкой на класс является терм $@c$, и имеющих все свойства из множества $\overline{P}\,(c)$ и только их, причем значение каждого свойства имеет тип, указанный для данного свойства в классе:

$$\frac{\mathbf{class}\ c : S\,\{\ldots\} \quad \overline{P}\,(c) = \{p_1 : \tau_1, \ldots, p_n : \tau_n\}}{\lceil\mathbf{val}\,(c)\rceil_\Psi = \left\{\mathbf{object}\ id : @c\,\{p_i = v_i\} \in \mathcal{S}_{obj}\,(\Psi) \mid v_i \in [\![\tau_i]\!]^C_\Psi\right\}}\ objects$$

Другие типы характеризуются следующим образом:

$$\frac{\mathbf{class}\ c : S\,\{\ldots\}}{\lceil\mathbf{ref}\,(c)\rceil_\Psi = \{@id \in \mathcal{S}_{ref}\,(\Psi) \mid \Psi(id) \in \lceil\mathbf{val}\,(c)\rceil_\Psi\}}\ refs$$

$$\frac{}{\lceil\{\tau^+\}\rceil_\Psi = \{\{x_1, \ldots, x_n\} \mid x_i \in \lceil\tau\rceil_\Psi\}}\ set$$

$$\frac{}{\lceil[\tau^+]\rceil_\Psi = \{[x_1, \ldots, x_n] \mid x_i \in \lceil\tau\rceil_\Psi\}}\ list$$

$$\frac{}{\lceil\{\tau^*\}\rceil_\Psi = \lceil\{\tau^+\}\rceil_\Psi \cup \{\{\,\}\}}\ eset \qquad \frac{}{\lceil[\tau^*]\rceil_\Psi = \lceil[\tau^+]\rceil_\Psi \cup \{[\,]\}}\ elist$$

$$\frac{}{\lceil\tau^?\rceil_\Psi = \lceil\tau\rceil_\Psi \cup \{\mathtt{null}\}}\ null$$

$$\frac{}{\lceil\mathsf{Int}\rceil = \mathbb{Z}}\ int \qquad \frac{}{\lceil\mathsf{Bool}\rceil = \mathbb{B}}\ bool$$

$$\frac{}{\lceil\mathsf{Char}\rceil = \Sigma}\ char \qquad \frac{}{\lceil\mathsf{String}\rceil = \Sigma^*}\ string$$

$$\frac{\mathbf{enum}\ e\,\{L_1, \ldots, L_n\}}{\lceil\mathbf{enum}\,(e)\rceil = \{L_1, \ldots, L_n\}}\ enum$$

Теперь нам осталось лишь “склеить” значения функции $\lceil\bullet\rceil_\Psi$ согласно отношению $\preceq$, чтобы получить $[\![\bullet]\!]^C_\Psi$:

$$\begin{aligned}[\![\tau]\!]^C_\Psi &= \lceil\tau\rceil_\Psi \cup \bigcup_{\sigma\preceq\tau}\lceil\sigma\rceil_\Psi \\ [\![\top]\!]^C_\Psi &= \bigcup_\tau \lceil\tau\rceil_\Psi\end{aligned}$$

Определение 1.12. Будем называть объекты, входящие в множество $[\![\mathbf{val}\,(c)]\!]^C_\Psi$, *экземплярами* класса c.

Мы будем изображать объекты и классы на одной диаграмме, проводя пунктирные ребра от экземпляров к классам[6]. Чтобы не загромождать рисунки, мы будем проводить лишь некоторые такие ребра. Так, например, на Рис. 1.7 к объектам с Рис. 1.2 добавлены некоторые классы с Рис. 1.4. Один и тот же объект может являться экземпляром сразу нескольких классов: t4 является экземпляром классов Usage и Term одновременно. Заметим, что все экземпляры одного класса имеют общий набор свойств, описанный в этом классе: все экземпляры класса Term разделяют пустой набор свойств, а все экземпляры Usage — набор из одного свойства var. Таким образом, классы

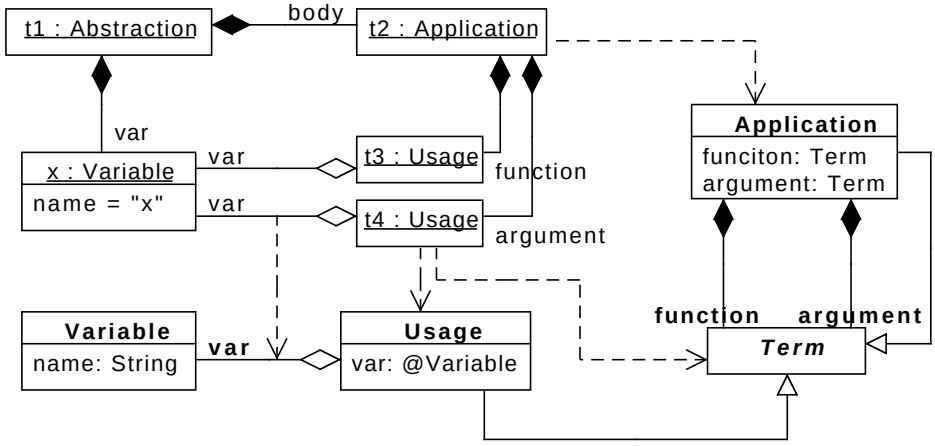

Рис. 1.7: Классы и их экземпляры

[6] В нотации UML на таких ребрах пишется стереотип «instance of», который мы опускаем, чтобы сэкономить место на рисунках.

регламентируют структуру своих экземпляров, гарантируя наличие тех или иных свойств у объектов.

1.3. Модели и мета-модели

Выше мы потребовали, чтобы экземпляры класса $C = \mathbf{class}\ c : S\{P\}$, имели вид $\mathbf{object}\ id : @c\{\ldots\}$, то есть содержали ссылку на идентификатор класса c. Если класс C сам не является объектом, то контекст, в котором упоминаются его экземпляры, не может быть замкнутым. Для того, чтобы устранить это неудобство, мы представим классы в виде объектов и сделаем, таким образом, синтаксис модельных термов замкнутым.

На Рис. 1.8 приведена диаграмма, охватывающая основные аспекты представления классов в виде объектов: приведены представления класса Class, экземпляры которого обозначают классы, класса Type — для типов и Property — для свойств. Пунктирные ребра на этой диаграмме проведены не между классами и объектами, а между объектами, представляющими классы, и объектами, являющимися экземплярами этих классов. Также проведено пунктирное ребро между ребром, изображающим одно из значений свойства @properties объекта Class, и объектом описывающим это свойство. Отметим, что идентификаторы свойств на этой диаграмме имеют вид $@x$, то есть являются ссылками на объекты, описывающие сами свойства. Такое кодирование наиболее удобно, поскольку позволяет, например, узнать тип свойства, просто получив описывающий его объект по идентификатору.

У некоторых объектов на диаграмме Рис. 1.8 не указаны идентификаторы. Это сделано для краткости: эти идентификаторы нигде не используются, соответственно, единственное, что требуется — это чтобы они не совпадали ни с какими другими идентификаторами объектов. Поскольку такие несовпадающие идентификаторы всегда можно выбрать, мы можем позволить себе не изображать их на рисунке.

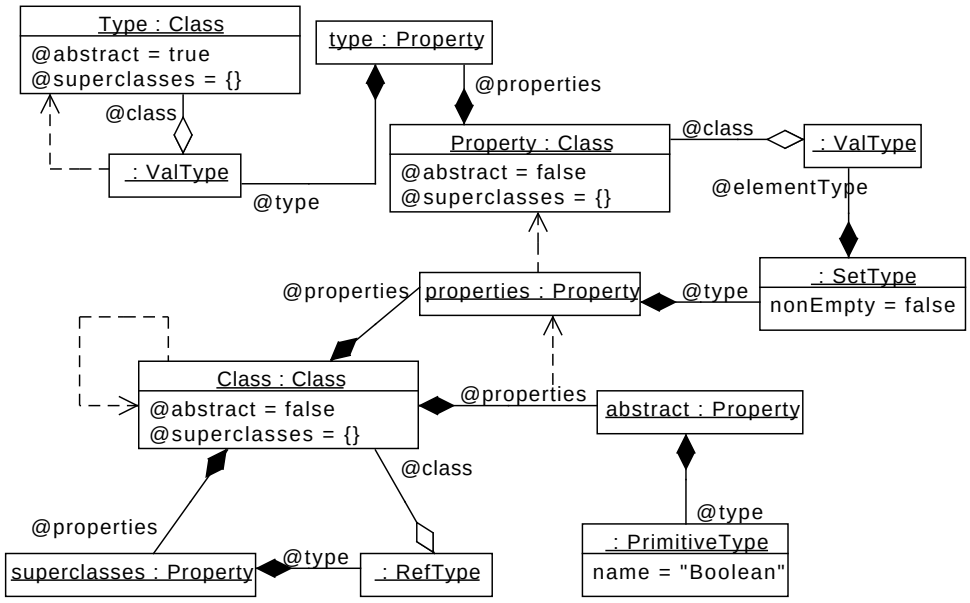

Рис. 1.8: Классы как объекты

Формальное правило для преобразования объектного представления в классы (см. Приложение 1), описанные выше, записывается довольно длинно и довольно тривиально, поэтому мы не приводим его здесь целиком. Основная идея состоит в том, что объекты вида $\mathbf{object}\ c : @\mathsf{Class}\,\{p\}$ преобразуются в классы вида $\mathbf{class}\ c : S\,\{P\}$, где S, P и признак абстрактности вычисляются из значений соответствующих свойств из p.

Представление типов приведено на Рис. 1.9 и Рис. 1.10. Оно также непосредственно трансформируется в типы, определенные выше (см. Приложение 1).

Определение 1.13 (Типовый контекст)**.** Пусть m является предмоделью в некотором ссылочном контексте Ψ_0, и $\Psi = \Psi_0 \cup \Psi_m$. Если по множеству всех экземпляров класса Class, упомянутых в Ψ, строится корректный набор классов C, то функция

$$\mathfrak{T}_m(t) = \begin{cases} \tau, & \text{где } t \in [\![\tau]\!]^C_\Psi, \\ \bot, & \text{в противном случае,} \end{cases}$$

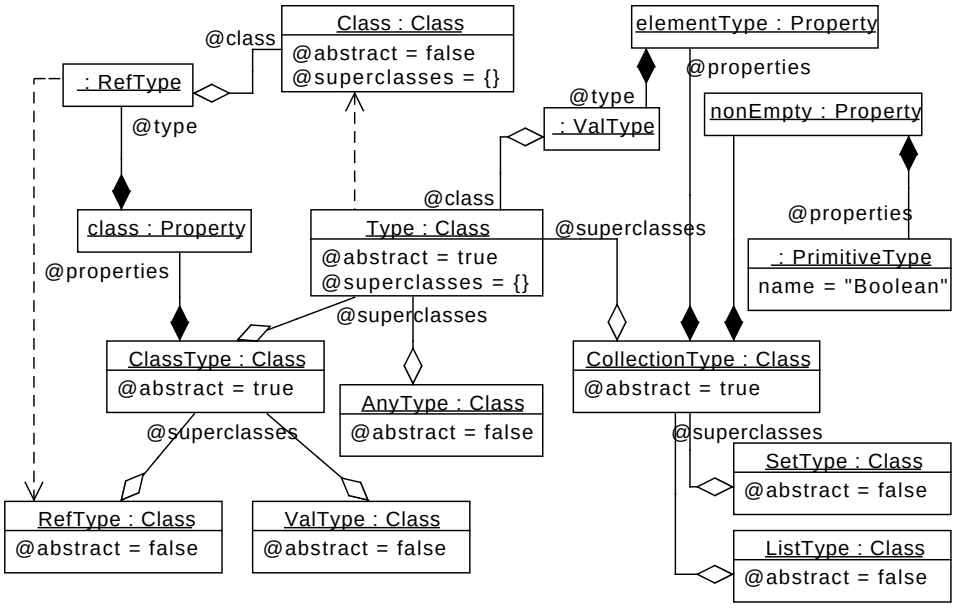

Рис. 1.9: Классы для типов (I)

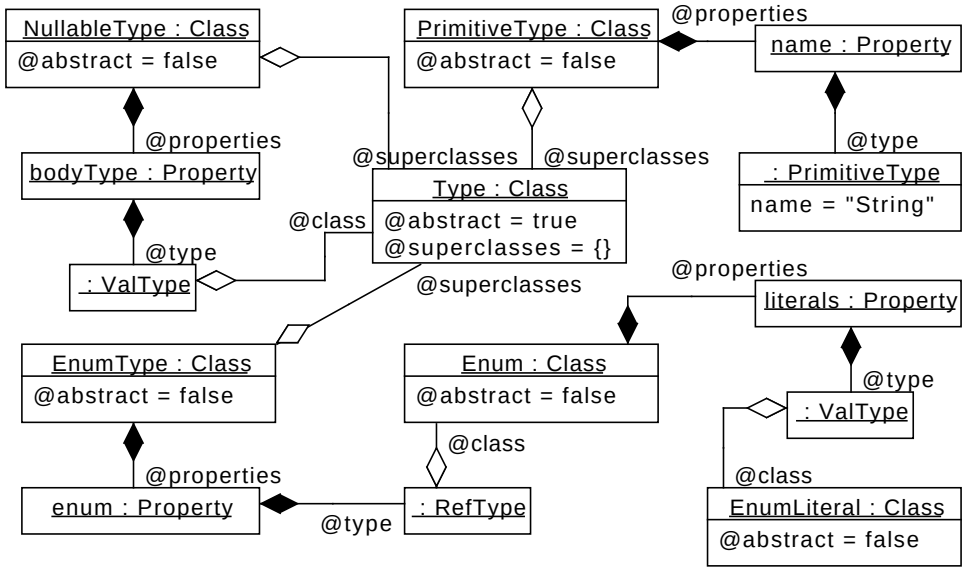

Рис. 1.10: Классы для перечислений типов (II)

называется *типовым контекстом*, порожденным m. Термы t для которых $\mathfrak{T}_m(t) \neq \bot$, называются *типизируемыми* в контексте $\mathfrak{T}_m$.

Заметим, что из типизируемости терма m следует типизируемость всех его подтермов. Заметим также, что m может не содержать описания классов, составляющих C, а лишь ссылаться на них, поскольку сами объекты, описывающие эти классы, могут быть упомянуты в контексте Ψ_0.

Основная идея определения 1.13 в том, что "правильные" термы являются типизируемыми. В дальнейшем мы будем работать только с такими термами:

Определение 1.14 (Модель)**.** Пусть типовый контекст $\mathfrak{T}_m$ построен по пред-модели m (в контексте Ψ_0) согласно определению 1.13. Если $\mathfrak{T}_m(m) \neq \bot$, то m называется *моделью* в контексте Ψ_0.

Отметим следующее важное наблюдение:

Предложение 1.1. *Модельный терм $\mathfrak{M}$, изображенный на Рис. 1.8, Рис. 1.9 и Рис. 1.10 (в совокупности) является моделью в пустом контексте $\Psi_\emptyset$: $\Psi_\emptyset(id) \equiv \bot$.*

Доказательство. Для доказательства этого утверждения необходимо убедиться в том, что

а) $\mathfrak{M}$ является пред-моделью в контексте $\Psi_\emptyset$,

б) из $\mathfrak{M}$ извлекается корректный набор классов,

в) $\mathfrak{T}_\mathfrak{M} \neq \bot$.

Эти утверждения проверяются непосредственно. □

Фактически, ссылочный контекст $\Psi_\mathfrak{M}$ является минимальным контекстом, необходимым для извлечения классов из любого модельного терма. Это связано с тем, что любое описание классов будет содержать ссылки на подтермы $\mathfrak{M}$, такие как Class, Property, ValType

и т.д. Чтобы проиллюстрировать это соображение, рассмотрим представление классов для λ-исчисления, приведенных на Рис. 1.4, в виде объектов: см. Рис. 1.11. Заметим, что все ссылки на подтермы $\mathfrak{M}$ на

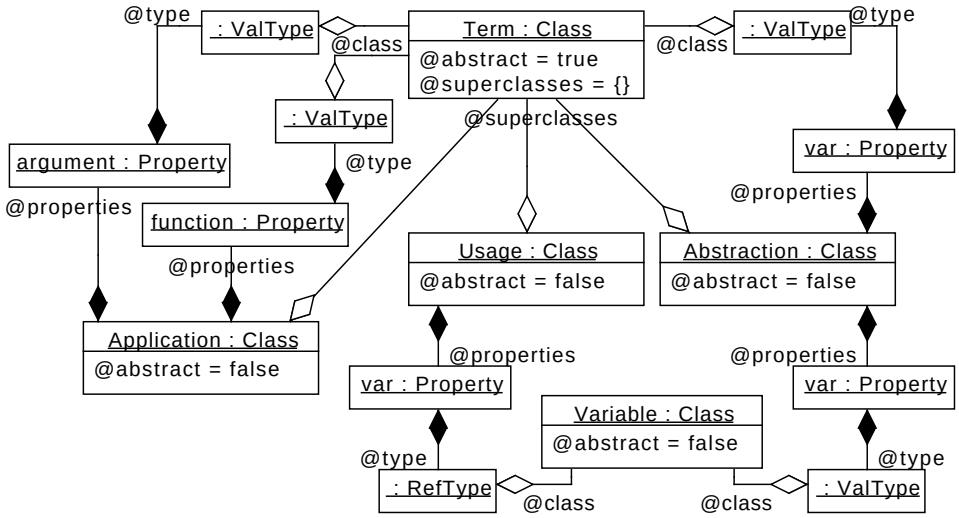

Рис. 1.11: Классы для λ-исчисления (см. Рис. 1.4) в виде объектов

Рис. 1.11 являются не *значениями* свойств объектов, а лишь *ссылками на классы* и *маркерами свойств*, то есть они лишь играют роль "разметки" для данных, задавая их структуру. Эта "разметка" называется *метаинформацией*. Такая ситуация является типичной: хотя объекты могут ссылаться на классы как на равноправные себе сущности, первоочередная роль классов в том, чтобы контролировать структуру термов. Удобно отделять описание классов от данных, структуру которых они регламентируют, поэтому мы вводим понятие *метамодели* как множества всех классов, составляющих метаинформацию для данной модели. Для этого нам понадобится определить несколько вспомогательных понятий.

Пусть m является моделью в контексте Ψ_0, и пусть $\mathcal{M}_0(m)$ — множество всех описаний классов, на которые указывают ссылки на классы из объектов в $\mathcal{S}_{obj}\,(m)$:

$$\mathcal{M}_0(m) \stackrel{\text{def}}{=} \{\Psi_m(cid) \mid \mathbf{object}\ id : @cid\,\{\ldots\} \in \mathcal{S}_{obj}\,(m)\}\,.$$

Фактически, $\mathcal{M}_0(m)$ — это множество тех классов, которые регламентируют структуру модели m.

Пусть $\mathcal{S}^{\overline{\mathcal{M}}}_{obj}(t)$ — множество всех объектов, упомянутых в терме t и термах, на которые t ссылается, но не только в качестве метаинформации (мы считаем, что сам терм t упомянут в замкнутом контексте Ψ_m):

$$\mathcal{S}^{\overline{\mathcal{M}}}_{obj}(t) \stackrel{\text{def}}{=} \begin{cases} \{t\} \cup \mathcal{S}^{\overline{\mathcal{M}}}_{obj}(id) \cup \bigcup\limits_{i=1}^{n} \mathcal{S}^{\overline{\mathcal{M}}}_{obj}(v_i), & \\ & \text{при } t = \mathbf{object}\ id : c\,\{p_1 = v_1, \ldots, p_n = v_n\}, \\ \bigcup\limits_{i=1}^{n} \mathcal{S}^{\overline{\mathcal{M}}}_{obj}(x_i), & \\ & \text{при } t = \{x_1, \ldots, x_n\} \text{ или } t = [x_1, \ldots, x_n], \\ \mathcal{S}^{\overline{\mathcal{M}}}_{obj}(\Psi_m(a)), & \text{при } t = @a, \\ \emptyset, & \text{в остальных случаях} \end{cases}$$

Функция $\mathcal{S}^{\overline{\mathcal{M}}}_{obj}(\bullet)$ естественным образом распространяется на множества термов: $\mathcal{S}^{\overline{\mathcal{M}}}_{obj}(T) = \bigcup\limits_{t \in T} \mathcal{S}^{\overline{\mathcal{M}}}_{obj}(t)$.

Определение 1.15 (Метамодель)**.** Если m — модель, то множество $\mathcal{M}(m) = \mathcal{S}^{\overline{\mathcal{M}}}_{obj}(\mathcal{M}_0(m))$ называется *метамоделью* для m.

$\mathcal{M}(m)$ — это множество всех объектов, непосредственно вовлеченных в описание классов из $\mathcal{M}_0(m)$, эти объекты составляют полную метаинформацию для m. Некоторая громоздкость функции $\mathcal{S}^{\overline{\mathcal{M}}}_{obj}(T)$ объясняется тем, что сами объекты из $\mathcal{M}(m)$ тоже имеют метаинформацию (из $\mathfrak{M}$), которая может быть не задействована непосредственно при описании структуры m. Функция $\mathcal{S}^{\overline{\mathcal{M}}}_{obj}(T)$ призвана исключить эту “метаинформацию для метаинформации”.

Ниже мы называем метамоделью не только множество $\mathcal{M}(m)$, но и терм-множество

$$\{t_1, \ldots, t_n\},$$

где t_i — элементы множества $\mathcal{M}(m)$.

Предложение 1.2. *Метамодель* $\mathcal{M}(m)$ *(как терм) является моделью в контексте* $\Psi_{\mathfrak{M}}$.

Доказательство. В самом деле, $\mathcal{M}_0(m)$ состоит из *описаний классов*, которые удовлетворяют структурным требованиям классов из $\mathfrak{M}$, то есть $\mathcal{S}_{obj}^{\overline{\mathcal{M}}}(\mathcal{M}_0(m)) = \mathcal{M}(m)$ состоит только из объектов, типизируемых в контексте $\mathfrak{T}_{\mathfrak{M}}$. С другой стороны, функция $\mathcal{S}_{obj}^{\overline{\mathcal{M}}}(\bullet)$ гарантирует разрешение всех ссылок, упомянутых прямо или косвенно в $\mathcal{M}_0(m)$, то есть обеспечивает замкнутость контекста $\Psi_{\mathcal{M}(m)} \cup \Psi_{\mathfrak{M}}$. □

Предложение 1.3. *Метамодель* $\mathcal{M}(\mathfrak{M})$ *совпадает с* $\mathfrak{M}$.

Доказательство. Это утверждение проверяется непосредственно. □

Предложение 1.3 показывает, что терм $\mathfrak{M}$ имеет важную особенность: *классы в нем контролируют структуру своего собственного описания*[7]. Такая роль $\mathfrak{M}$ настолько важна, что заслуживает отдельного термина:

Определение 1.16 (Метаметамодель)**.** Модельный терм $\mathfrak{M}$ называется *метаметамоделью*.

§ 2. Языки как множества моделей

Традиционно *языком* называется множество строк, составленных символами некоторого конечного алфавита [2, **?**]. Такой подход вполне оправдан при рассмотрении вопросов теории сложности [**?**], алгебраических свойств операций над языками [**?**, **?**] и задач синтаксического анализа [2]. Однако язык, предназначенный для использования человеком, в первую очередь представляет собой *систему понятий и отношений между ними*, и при рассмотрении таких языков

[7] Но не только: например, они же контролируют структуру описания классов на Рис. 1.11

целесообразно отталкиваться не от текстового *конкретного* синтаксиса, роль которого обычно весьма важна, но не первостепенна, а от какого-то более структурированного описания, не зависящего от визуального представления предложений языка. Такое структурированное представление, как правило, называют *абстрактным синтаксисом*. Сразу заметим, что этот термин сильно перегружен значениями. Так наиболее распространенное понятие *абстрактного синтаксического дерева* (Abstract Syntax Tree, AST [**?**, 2]) не соответствует представлениям об абстрактном синтаксисе языка: абстрактное представление даже весьма простых программ, не будет являться деревом (см. Рис. 1.2).

Мы используем следующее определения языка:

Определение 1.17 (Язык)**.** Пусть M — модель и $\mathcal{M}(M) = \mathfrak{M}$ (элементами M являются описания классов, следовательно M может выступать в качестве метамодели). Множество $\mathcal{L}(M)$, состоящее из всех моделей m, таких что $\mathcal{M}(m) = M$, называется *языком*, порожденным метамоделью M или просто *языком* M.

Возникает вопрос о том, как связано традиционное определение понятия *язык* с тем, которое используем мы. В данном разделе мы покажем, во-первых, что для любого контекстно-свободного языка существует естественное представление в виде языка, порожденного некоторой метамоделью [**?**], а во-вторых, что контекстно-свободная грамматика G, снабженная аннотациями определенного вида [83], может быть механически преобразована в метамодель M_G, такую, что $\mathcal{L}(M_G)$ является в точности множеством абстрактных синтаксических деревьев для языка, порожденного грамматикой G.

2.1. Нотация для контекстно-свободных грамматик

Для описания контекстно-свободных грамматик чаще всего применяется нотация Бэкуса-Наура (Backus-Naur Form, BNF, [**?**]), поз-

воляющая задавать продукции, в правой части которых стоят последовательности терминальных и нетерминальных символов. Поскольку множество контекстно-свободных языков замкнуто относительно регулярных операций объединения и итерации, мы можем рассматривать более удобную для приложений расширенную нотацию Бэкуса-Наура (Extended BNF, EBNF [36]). Существует несколько версий конкретного синтаксиса EBNF, и мы используем вариант, основанный на нотации ANTLR [63][1]. В этой нотации правая и левая части продукции разделяются двоеточием, и в правой части могут использоваться следующие операции (`A` и `B` обозначают грамматические выражения):

- конкатенация: `A B`;
- объединение: `A | B`;
- итерация Клини: `A*` (повторение ноль или более раз);
- повторение один или более раз: `A+`;
- необязательное вхождение: `A?`;
- группировка: `(A)`.

Следуя нотации ANTLR, мы пишем названия терминальных символов заглавными буквами (например `NAME`), а нетерминальных — начиная со строчной буквы, согласно конвенции “CamelCase” (например, `longName`). Ключевые слова и знаки операций, также соответствующие терминальным символам, пишутся в двойных кавычках.

В качестве примера использования данной нотации рассмотрим контекстно-свободную грамматику для упрощенного синтаксиса заголовков методов в языке JAVA [38] (см. Лист. 2.1). Метод может быть либо *абстрактным*, тогда он должен быть помечен ключевым словом `abstract` и после списка параметров следует точка с запятой, либо

[1] Полное описание используемой нами нотации приведено в Приложении 2.

```
methodHeader
  : "abstract" signature ";"
  | signature body
  ;

signature : type NAME "(" parameterList? ")";

parameterList : parameter ("," parameter)*;

type : "void" | NAME;

parameter : type NAME;

body : "{" "}" ;
```

Листинг 2.1: Упрощенный синтаксис описания методов в языке JAVA.

конкретным, тогда после списка параметров следует *тело* метода в фигурных скобках (для простоты в нашем примере тело всегда пустое).

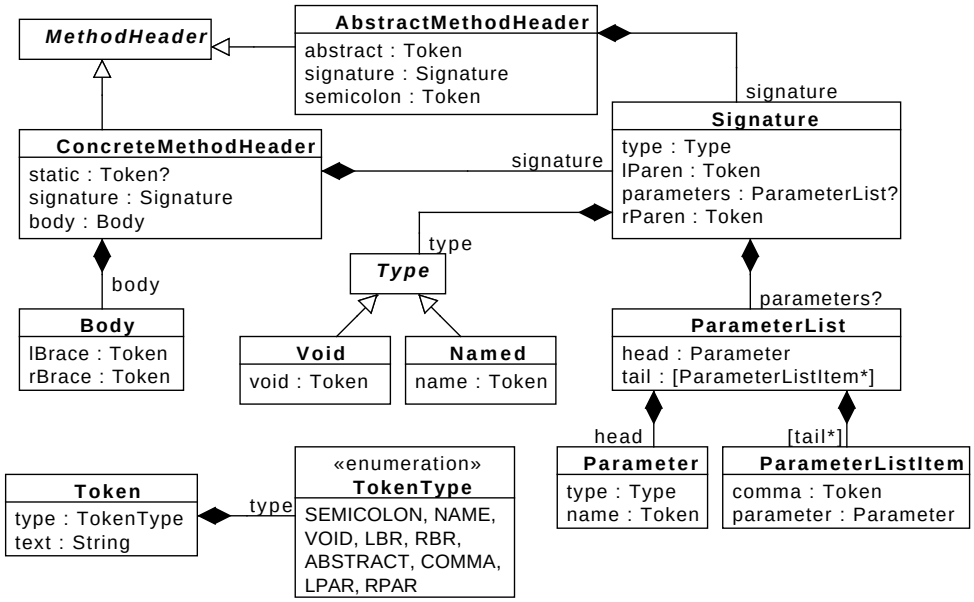

Рис. 2.1: Метамодель, соответствующая Рис. 2.1.

2.2. Метамодели, эквивалентные грамматикам

На Рис. 2.1 приведена диаграмма метамодели, построенной по грамматике из Лист. 2.1. Эта метамодель получена механически, и ни-

же мы приведем процедуру построения таких метамоделей, но сначала рассмотрим этот пример. Большинство классов на Рис. 2.1 соответствуют нетерминалам грамматики и имеют свойства, соответствующие составляющим правых частей соответствующих продукций, так что экземплярами данной метамодели являются в точности деревья разбора, соответствующие предложениям языка, порождаемого грамматикой из Лист. 2.1. Рассмотрим пример такого предложения:

```
void example(String s) {}
```

Соответствующая модель приведена на Рис. 2.2. Из рисунка видно,

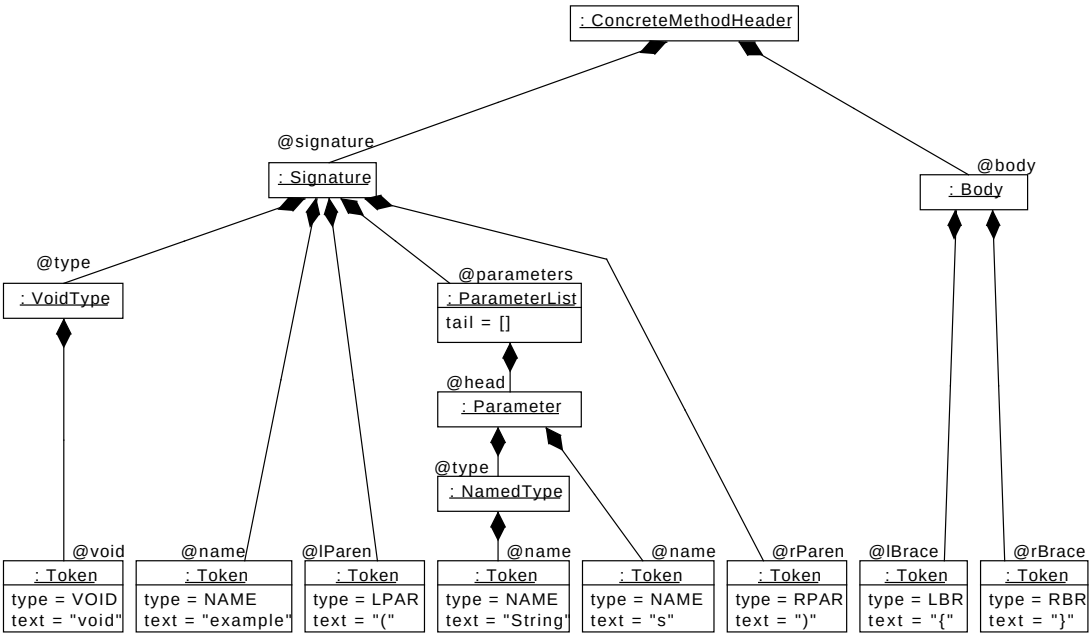

Рис. 2.2: Диаграмма объектов дерева разбора.

что дерево встраивания объектов прямо соответствует дереву разбора. Объекты класса `Token`, занимающие нижний уровень дерева, соответствуют лексемам (каждая лексема хранит текст, из которого она была получена в поле `text` и соответствующий терминальный символ в поле `type`), а на объекты на остальных уровнях дерева соответствуют применениям продукций.

Заметим, что по дереву разбора легко можно восстановить

текст, из которого оно было получено, следовательно можно утверждать, что при преобразовании в модель *сохраняется вся полезная информация*, имевшаяся в исходном тексте[2]. Отсюда следует Предложение 1.4.

Предложение 1.4. *Метамодель на Рис. 2.1 в качестве описания языка эквивалентна грамматике из Лист. 2.1.*

Осталось показать, что по любой контекстно-свободной грамматике можно построить эквивалентную метамодель. Для этого достаточно применить следующую процедуру.

!!!!!!

[2]В нашем примере мы не учитываем пробельные символы, поскольку грамматика их игнорирует, но это легко исправить.

Глава 2.
Расширение предметно-ориентированных языков механизмами композиции

§ 1. Автоматическое построение языков, поддерживающих типизированные макроопределения

В настоящем разделе описан метод, позволяющий по описанию языка построить описание более богатого языка, поддерживающего композицию с помощью типизированных макроопределений. Мы будем называть такой пополненный язык *макро-языком*, построенным на базе исходного языка.

Замечание 2.1 (О терминологии)**.** *В англоязычной литературе используется термин* macro *[24,* **?***, 74]. В качестве русского перевода этого термина употребляется либо слово "макрос", являющееся в сущности сленговым и полученное транслитерацией множественного числа "macros", либо слово "макроопределение", соответствующее более узкому по смыслу термину "macro definition". Близкой по смыслу альтернативой является термин "шаблон" (англ. "template"), используемый в языке* C++ *[77] и ряде других [***?***, 26, 13].*

Мы считаем термин "macro" более подходящим для наших целей, и будем, следуя традиции, использовать более формальный вариант его перевода — макроопределение.

1.1. Неформальное описание механизма композиции, основанного на макроопределениях

Наиболее известными системами, использующими макроопределения, являются язык программирования LISP [75] и препроцессор языка C [44, **?**]. В обоих случаях макроопределения служат для обогащения языка новыми конструкциями, которые преобразуются в базовые конструкции языка во время компиляции[1], этот процесс называется *разворачиванием* макроопределений.

Приведем пример использования макроопределений в языке C: пусть нам требуется реализовать односвязные списки. Для этого в первую очередь необходимо описать *структуру* элемента списка, например, для списка целых чисел эта структура будет выглядеть следующим образом:

```
struct IntList {
    struct IntList* next;
    int data;
};
```

Для списка элементов другого типа, например, строк, структура будет очень похожей, но тип поля `data` будет отличаться:

```
struct StrList {
    struct StrList* next;
    char* data;
};
```

Дублировать описания для каждого нового типа элементов не очень удобно, поэтому мы напишем макроопределение, которое по данному типу элемента генерирует описание соответствующей структуры[2]:

```
#define DEFLIST(name, type) struct name { \
    struct name *next; \
    type data; \
};
```

После директивы `#define` следует *имя* макроопределения, а за ним в скобках — *параметры*. Весь остальной текст — это *тело* макроопределения (знак “\” используется для подавления перевода строки).

[1] Понятие “время компиляции” в данном контексте является собирательным и противопоставляется понятию “время выполнения программы”.

[2] Задача, которую мы решаем в этом примере с помощью макроопределений языка C, более эффективно решается в языке C++ с помощью *шаблонных структур*, которые можно рассматривать как узкоспециализированную разновидность макроопределений.

Для того, чтобы использовать данное макроопределение, достаточно написать его имя и передать в скобках значения параметров (то есть *аргументы*), например:

```
DEFLIST(StrList, char*)
```

В результате *разворачивания* данного определения аргументы будут подставлены в тело вместо соответствующих параметров и мы получим определение структуры `StrList`, приводившееся выше. Аналогично можно получить определения структуры `IntList`, а также структуры элемента списка для любого типа. Заметим, что разворачивание происходит во время компиляции и результатом является исходный текст на языке C, который, в свою очередь, транслируется в машинный код, причем транслятор ничего не знает о том, были использованы макроопределения или нет.

Обобщая сведения, приведенные в данном примере, можно выделить следующие свойства, присущие механизму макроопределений[3]:

- Макроопределение состоит из *списка параметров* и *тела* и имеет уникальное *имя*.
- Тело макроопределения содержит конструкции на целевом языке (в нашем примере — на языке C) и ссылки на параметры. Также тело может содержать обращения к другим макроопределениям.
- При разворачивании ссылки на параметры в теле макроопределения заменяются значениями соответствующих аргументов.
- Разворачивание происходит во время компиляции программы.

Макроопределения представляют собой достаточно гибкий механизм повторного использования. В принципе, этот механизм не зависит от целевого языка, в который разворачиваются макроопределения.

[3]Такое обобщение имеет смысл, поскольку в различных языках и системах макроопределения функционируют схожим образом.

В частности, в языке C поддержка макроопределений обеспечивается препроцессором CPP — независимым программным средством, обрабатывающим исходный код *до* начала работы собственно компилятора языка C. Препроцессор CPP может работать с любым текстом, не только с исходным кодом на языке C, следовательно он (или аналогичный механизм) может применяться для повторного использования и в других языках, в частности в предметно-ориентированных, делая их более пригодными для использования в промышленных проектах.

Однако чисто текстовый препроцессор обладает одним важным недостатком: корректность результата разворачивания макроопределений никак не гарантируется, поскольку препроцессор манипулирует простым текстом и “не знает” о синтаксисе целевого языка.

Вернемся к примеру макроопределения, приведенному выше. Если программист допустит ошибку при использовании макроопределения `DEFLIST`, а именно перепутает порядок аргументов, что случается не так уж редко, препроцессор послушно выполнит свою работу:

```
DEFLIST(char*, StrList)
```

превратится в

```
struct char* {  // error: expected '{' before 'char'
    struct char* *next;
    StrList data;
};
```

Получившийся код синтаксически некорректен, и компилятор, получив этот текст на вход, выдаст сообщение об ошибке:

```
DEFLIST(char*, StrList) // error: expected '{' before 'char'
```

В итоге ошибка программиста обнаружена, но сообщение, выданное компилятором, записано в терминах программы, полученной после разворачивания макроопределений, и совсем не помогает программисту исправить ситуацию. Чтобы разобраться, в чем проблема, придется вручную рассмотреть код, полученный на выходе препроцессора, что является существенным затруднением при разработке больших проектов. Описанная здесь проблема является основной причиной, по

которой профессиональные программисты зачастую избегают широкого использования возможностей макроопределений в программах на C [**?**].

Итак, чисто текстовый препроцессор позволяет легко обеспечить поддержку макроопределений в любом языке, но не обеспечивает своевременного обнаружения ошибок, что затрудняет разработку. В данной главе мы рассмотрим метод реализации макроопределений, который также пригоден для любого языка, но обеспечивает контроль корректности аргументов макроопределений с помощью специальной системы типов, что позволяет избежать проблем, присущих чисто текстовым препроцессорам. Такие макроопределения мы будем называть *шаблонами* (templates).

!!!!!!!!!!!!!!!!!!!!!!!!!!!! Данный подход получил обобщение в виде концепции *многоуровневых языков* (multi-stage languages), реализациями которой являются MacroML [24] и Template Haskell [**?**]. !!!!!!!!!!!!!!!!!!!!!!!!!!!!

1.2. Шаблоны в языках, порожденных метамоделями

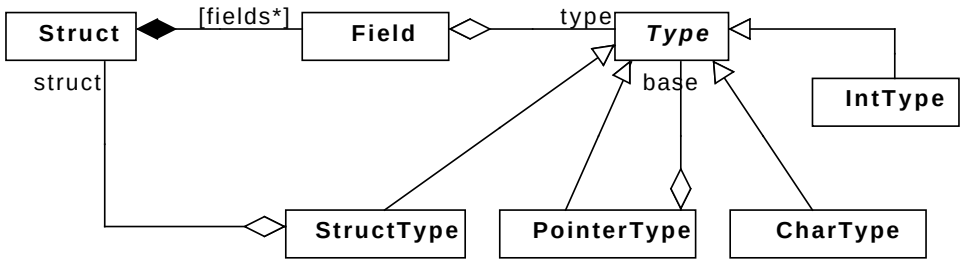

Рис. 1.1: Метамодель абстрактного синтаксиса описаний структур в языке C

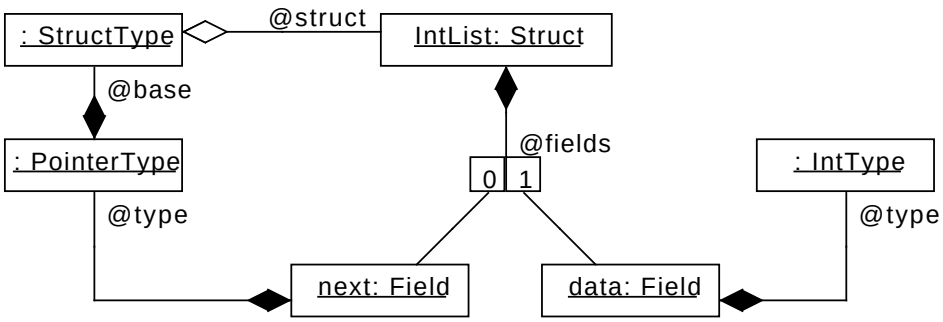

Рис. 1.2: Модель, описывающая структуру `IntList`

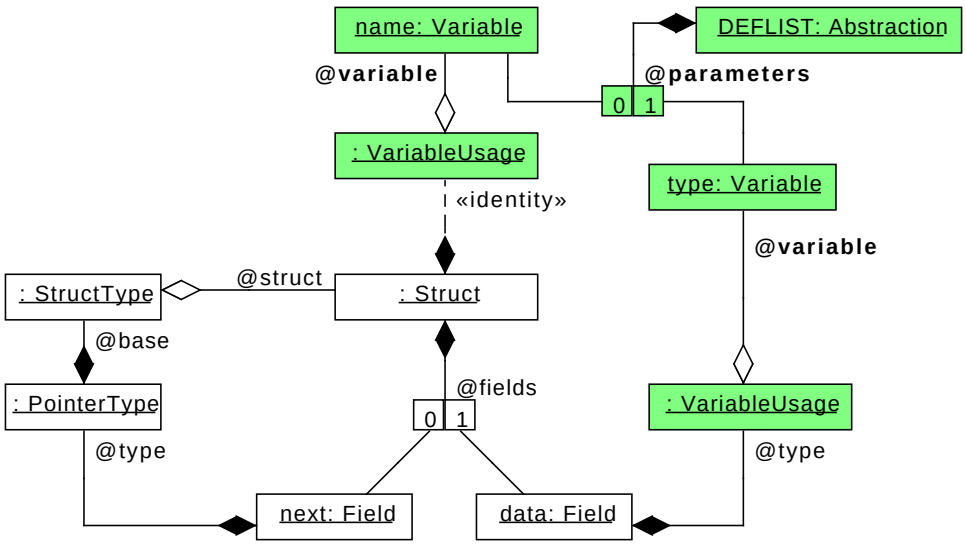

Рис. 1.3: Шаблон описания структуры элемента списка

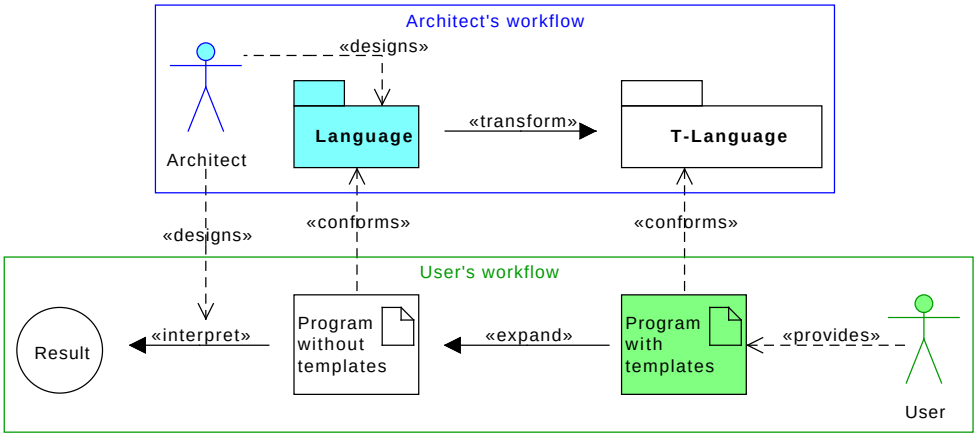

Рис. 1.4: Разработка и использование языков с поддержкой шаблонов

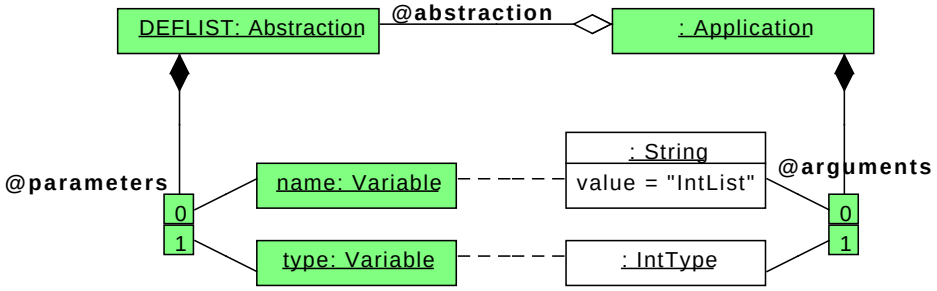

Рис. 1.5: Применение шаблона

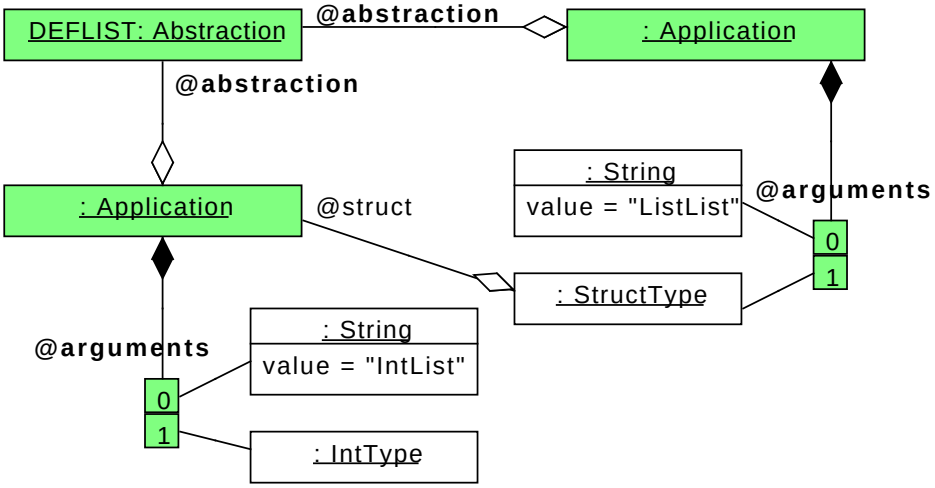

Рис. 1.6: Использование применения шаблона в аргументах

Приложение 1. Преобразование объектного представления метамоделей в классы

Ниже приводится программа на языке HASKELL [**?**], которая преобразует модельный терм (такой, например, как на Рис. 1.8) набор классов (согласно определению 1.8). Вначале определяются типы данных для модельных термов (согласно определению 1.1):

```
import Data.List

data ModelTerm
  = Object {
      id :: ModelTerm,
      classRef :: ModelTerm,
      properties :: [Property]
    }
  | Ref { base :: ModelTerm }
  | List [ModelTerm]
  | Set [ModelTerm]
  | Null
  | Int Int
  | Bool Bool
  | Char Char
  | String String
  deriving (Eq)

data Property = Property {
    name :: ModelTerm,
    value :: ModelTerm
  }
  deriving (Eq)
```

Далее определяется функция $\mathcal{S}(t)$ (см. формулу 1.1).

```
subterms :: ModelTerm -> [ModelTerm]
subterms o = o : case o of
  Object id cr p -> (subterms id) ++ (subterms cr) ++ (concat (map pSubterms p))
  Ref    b       -> subterms b
  List   l       -> concat (map subterms l)
  Set    l       -> concat (map subterms l)
  _              -> []

pSubterms :: Property -> [ModelTerm]
pSubterms (Property n v) = (subterms n) ++ (subterms v)
```

Тип данных и функция для построения ссылочных контекстов (см. определение 1.4):

```
type RContext = [(ModelTerm, ModelTerm)]

rcontext :: ModelTerm -> RContext
rcontext t = map toPair (filter isObject (subterms t))
```

```
  where
    toPair o@(Object id _ _) = (id, o)

    isObject (Object _ _ _) = True
    isObject _              = False
```

Типы данных для классов и типов (см. определения 1.8, 1.9 и 1.10):

```
data Class = Class {
    abstract :: Bool,
    class_id :: String,
    superclasses :: [Class],
    propertyDescriptors :: [PropertyDescriptor]
  }

data PropertyDescriptor = PropertyDescriptor String Type

data Enum = Enum {
    enum_id :: String,
    literals :: [String]
  }

data Type
  = CharT
  | StringT
  | IntT
  | BoolT
  | NullableT Type
  | SetT Type Bool
  | ListT Type Bool
  | ValT Class
  | RefT Class
  | TopT
```

Функция, преобразующая модельный терм в набор классов (см. раздел 1.3):

```
extractClasses :: ModelTerm -> [Class]
extractClasses t@(Set objects) = map (extractClass (rcontext t)) objects

extractClass :: RContext -> ModelTerm -> Class
extractClass context (Object (String id) (Ref (String "Class")) props)
   = Class
       (extractAbstract props)
       id
       (concat (map (superclasses context) props))
       (concat (map (toDescriptors context) props))

extractAbstract :: [Property] -> Bool
extractAbstract l = head (concat (map isAbstract l))

isAbstract :: Property -> [Bool]
isAbstract (Property (Ref (String "Class.abstract")) (Bool v)) = [v]
isAbstract _                                                   = []

superclasses :: RContext -> Property -> [Class]
```

```
superclasses context
    (Property
            (Ref (String "Class.superclasses"))
            (Set refs)) = map (toClass context) refs
superclasses _ _ = []

toClass :: RContext -> ModelTerm -> Class
toClass context (Ref id) =
    case lookup id context of
        Just a -> extractClass context a
toClass _ r = Class False ("E: " ++ (show r)) [] []

toDescriptors :: RContext -> Property -> [PropertyDescriptor]
toDescriptors context
    (Property
        (Ref (String "Class.propertyDescriptors"))
        (Set s)) = map (toDescriptor context) s
toDescriptors _ _ = []

toDescriptor :: RContext -> ModelTerm -> PropertyDescriptor
toDescriptor context
    (Object
        (String id)
        (Ref (String "PropertyDescriptor"))
        [Property
            (Ref (String "PropertyDescriptor.type"))
            propType]) = PropertyDescriptor id (toType context propType)

toType :: RContext -> ModelTerm -> Type
toType context
    (Object
        _
        (Ref (String "PrimitiveType"))
        [(Property
            (Ref (String "PrimitiveType.type"))
            (String "Boolean"))]) = BoolT
toType context
    (Object
        _
        (Ref (String "PrimitiveType"))
        [(Property
            (Ref (String "PrimitiveType.type"))
            (String "String"))]) = StringT
toType context
    (Object
        _
        (Ref (String "PrimitiveType"))
        [(Property
            (Ref (String "PrimitiveType.type"))
            (String "Integer"))]) = IntT
toType context
    (Object
        _
        (Ref (String "PrimitiveType"))
        [(Property
            (Ref (String "PrimitiveType.type"))
            (String "Char"))]) = CharT
```

```
toType context
    (Object
        _
        (Ref (String "NullableType"))
        [(Property
            (Ref (String "NullableType.type"))
            body)]) = NullableT (toType context body)
toType context
    (Object
        _
        (Ref (String "ReferenceType"))
        [(Property
            (Ref (String "ClassType.class"))
            (classRef))]) = RefT (toClass context classRef)
toType context
    (Object
        _
        (Ref (String "ObjectType"))
        [(Property
            (Ref (String "ClassType.class"))
            (classRef))]) = ValT (toClass context classRef)

toType context
    (Object
        _
        (Ref (String "SetType"))
        [
            (Property
                (Ref (String "CollectionType.nonEmpty"))
                (Bool nonEmpty)),
            (Property
                (Ref (String "CollectionType.elementType"))
                elementType)
        ]) = SetT (toType context elementType) nonEmpty
toType context
    (Object
        _
        (Ref (String "ListType"))
        [
            (Property
                (Ref (String "CollectionType.nonEmpty"))
                (Bool nonEmpty)),
            (Property
                (Ref (String "CollectionType.elementType"))
                elementType)
        ]) = ListT (toType context elementType) nonEmpty

toType context
(Object
    _
    (Ref (String "EnumType"))
    [Property
        (Ref (String "EnumType.enum"))
        (Ref enumId)]) = EnumT (toEnum (lookup enumId context))

toType context (Object _ (Ref (String "AnyType")) []) = TopT
```

```
toType context _ = TopT

toEnum :: Maybe ModelTerm -> Main.Enum
toEnum (Just
(Object
    (String id)
    (Ref (String "Enum"))
    [Property
        (Ref (String "Enum.literals"))
        (List literals)])) = Enum id (map toLiteral literals)

toLiteral :: ModelTerm -> String
toLiteral
(Object
    (String id)
    (Ref (String "EnumLiteral"))
    []) = id
```

Приложение 2. Нотация для контекстно-свободных грамматик

```
grammar{[Symbol*]}
  : symbol{>>}*;

symbol{Symbol}
  : NAME{>name} (":" expression{>>expressions})+
  ;

expression{abstract Expression}
  : singleCharacter
  : positiveRange
  : negativeRange
  : stringLiteral
  : option
  : iteration
  : alternative
  : sequence
  : empty
  : symbolReference
  : "(" expression{>} ")"
  ;

positiveRange{Alternative}
  : "[" rangeEntry{>>expressions} "]"
  : "[" rangeEntry{>>expressions} rangeEntry{>>expressions}+ "]"
  ;

negativeRange{NegativeCharacterRange}
  : "[" "^" rangeEntry{>>ranges}+ "]"
  ;

singleCharacter{CharacterRange}
  : CHARACTER {>from >to}
  ;

rangeEntry{CharacterRange}
  : singleCharacter
  : CHARACTER{>from} "-" CHARACTER{>to}
  ;

stringLiteral{StringLiteral}
  : STRING{~>value}
  ;

option{Option}
  : expression{>body} "?"
  ;

iteration{Iteration}
  : expression{>body} iterationKind{>kind}
  ;

iterationKind{enum IterationKind}
```

```
  : "*"{ZERO_OR_MORE}
  : "+"{ONE_OR_MORE}
  ;

alternative{Alternative}
  : expression{>>expressions} ("|" expression{>>expressions})*
  ;

sequence{Sequence}
  : expression{>>expressions} expression{>>expressions}+
  ;

empty{Empty}
  : "#empty"
  ;

symbolReference{SymbolReference}
  : NAME{~~>symbol} // ~~> for late resolve
  ;
```

Список литературы

[1] *Agesen, O.* Adding type parameterization to the java language / O. Agesen, S. N. Freund, J. C. Mitchell // *SIGPLAN Not.* — 1997. — Vol. 32, no. 10. — Pp. 49–65.

[2] *Aho, A. V.* Compilers: principles, techniques, and tools / A. V. Aho, R. Sethi, J. D. Ullman. — Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1986.

[3] *Aksit, M.* Compiler generation based on grammar inheritance. — 1990. http://doc.utwente.nl/19862/.

[4] Apache Derby. — http://db.apache.org/derby/.

[5] The asf+sdf meta-environment: A component-based language development environment / M. G. J. van den Brand, A. v. Deursen, J. Heering et al. // CC '01: Proceedings of the 10th International Conference on Compiler Construction. — London, UK: Springer-Verlag, 2001. — Pp. 365–370.

[6] Aspect-oriented programming / G. Kiczales, J. Lamping, A. Mendhekar et al. // In ECOOP. — SpringerVerlag, 1997.

[7] *A■mann, U.* Invasive Software Composition / U. A■mann. — Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2003.

[8] *Atkinson, C.* Model-driven development: A metamodeling foundation / C. Atkinson, T. Kühne // *IEEE Softw.* — 2003. — Vol. 20, no. 5. — Pp. 36–41.

[9] *Baader, F.* Term rewriting and all that / F. Baader, T. Nipkow. — New York, NY, USA: Cambridge University Press, 1998.

[10] *Bagge, A.* DSAL = library+notation: Program transformation for domain-specific aspect languages / A. Bagge, K. K. T. // First Domain Specific Aspect Language Workshop (DSAL'06). — 2006.

[11] *Berners-Lee, T.* Uniform resource identifiers (uri) generic syntax: Tech. Rep. RFC 2396 / T. Berners-Lee, R. Fielding, L. Masinter: 1998. — http://www.ietf.org/rfc/rfc2396.txt.

[12] *Booch, G.* Object-Oriented Analysis and Design with Applications (3rd Edition) / G. Booch. — Redwood City, CA, USA: Addison Wesley Longman Publishing Co., Inc., 2004.

[13] *Booch, G.* Unified Modeling Language User Guide, The (2nd Edition) (Addison-Wesley Object Technology Series) / G. Booch, J. Rumbaugh, I. Jacobson. — Addison-Wesley Professional, 2005.

[14] *Budinsky, F.* Eclipse Modeling Framework / F. Budinsky, S. A. Brodsky, E. Merks. — Pearson Education, 2003.

[15] *Ciric, M. D.* Parsing in different languages / M. D. Ciric, S. R. Rancic // *FACTA UNIVERSITATIS*. — 2005. — Vol. 18, no. 2. — Pp. 299–307.

[16] The compost, compass, inject/j and recoder tool suite for invasive software composition: Invasive composition with compass aspect-oriented connectors. / D. Heuzeroth, U. A■mann, M. Trifu, V. Kuttruff // GTTSE / Ed. by R. Lämmel, J. Saraiva, J. Visser. — Vol. 4143 of *Lecture Notes in Computer Science*. — Springer, 2006. — Pp. 357–377.

[17] *Dekel, U.* Towards a standard family of languages for matching patterns in source code / U. Dekel, T. Cohen, S. Porat // SWSTE '03: Proceedings of the IEEE International Conference on Software-Science, Technology & Engineering. — Washington, DC, USA: IEEE Computer Society, 2003. — P. 10.

[18] *Denny, J. E.* Ielr(1): practical lr(1) parser tables for non-lr(1) grammars with conflict resolution / J. E. Denny, B. A. Malloy // SAC '08: Proceedings of the 2008 ACM symposium on Applied computing. — New York, NY, USA: ACM, 2008. — Pp. 240–245.

[19] Design Patterns: Elements of Reusable Object-Oriented Software / E. Gamma, R. Helm, R. Johnson, J. Vlissides. — Addison-Wesley, 1995.

[20] *Dinkelaker, T.* Untangling crosscutting concerns in domain-specific languages with domain-specific join points / T. Dinkelaker, M. Monperrus, M. Mezini // DSAL '09: Proceedings of the 4th workshop on Domain-specific aspect languages. — New York, NY, USA: ACM, 2009. — Pp. 1–6.

[21] Eli: a complete, flexible compiler construction system / R. W. Gray, S. P. Levi, V. P. Heuring et al. // *Communications of the ACM*. — 1992. — Vol. 35, no. 2. — Pp. 121–130.

[22] *Fowler, M.* Domain specific languages. — http://martinfowler.com/dslwip/. — 2010.

[23] *Gagnon, E. M.* SableCC, an object-oriented compiler framework / E. M. Gagnon, L. J. Hendren // TOOLS '98: Proceedings of the Technology of Object-Oriented Languages and Systems. — Washington, DC, USA: IEEE Computer Society, 1998. — Pp. 140–154.

[24] *Ganz, S. E.* Macros as multi-stage computations: type-safe, generative, binding macros in macroml / S. E. Ganz, A. Sabry, W. Taha // ICFP '01: Proceedings of the sixth ACM SIGPLAN international conference on Functional programming. — New York, NY, USA: ACM, 2001. — Pp. 74–85.

[25] *Gärtner, F.* The PretzelBook. — http://www.informatik.tu-darmstadt.de/BS/Gaertner/pretzel/dist 1998.

[26] *Gradecki, J. D.* Mastering Apache Velocity / J. D. Gradecki, J. Cole. — Wiley, 2003.

[27] *Grimm, R.* Better extensibility through modular syntax / R. Grimm // *SIGPLAN Not.* — 2006. — Vol. 41, no. 6. — Pp. 38–51.

[28] Haskell 98 Language and Libraries: The Revised Report / Ed. by S. P. Jones. — http://haskell.org/, 2002. — September. — P. 277. http://haskell.org/definition/haskell98-report.pdf.

[29] *Havinga, W.* Prototyping and composing aspect languages / W. Havinga, L. Bergmans, M. Aksit // ECOOP '08: Proceedings of the 22nd European conference on Object-Oriented Programming. — Berlin, Heidelberg: Springer-Verlag, 2008. — Pp. 180–206.

[30] *Hedin, G.* JastAdd: an aspect-oriented compiler construction system / G. Hedin, E. Magnusson // *Science of Computer Programming*. — 2003. — Vol. 47, no. 1. — Pp. 37–58.

[31] *Henry, K.* A crash overview of groovy / K. Henry // *Crossroads*. — 2006. — Vol. 12, no. 3. — Pp. 5–5.

[32] Human-usable textual notation. — http://www.omg.org/spec/hutn/.

[33] Implementation of multiple attribute grammar inheritance in the tool lisa / M. Mernik, V. Žumer, M. Lenič, E. Avdičaušević // *SIGPLAN Not.* — 1999. — Vol. 34, no. 6. — Pp. 68–75.

[34] *Ingalls, D. H. H.* The smalltalk-76 programming system design and implementation / D. H. H. Ingalls // POPL '78: Proceedings of the 5th

ACM SIGACT-SIGPLAN symposium on Principles of programming languages. — New York, NY, USA: ACM, 1978. — Pp. 9–16.

[35] Introduction to Algorithms / T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein. — 2nd edition. — The MIT Press, 2001.

[36] *ISO*. ISO/IEC 14977:1996: Information technology — Syntactic metalanguage — Extended BNF / ISO. — 1996. — P. 12. http://www.iso.ch/cate/d26153.html.

[37] ISO/IEC 9075:1992, Database Language SQL. — http://www.contrib.andrew.cmu.edu/~shadow/sql/sql1992.txt. — 1992.

[38] The Java Language Specification, Third Edition / J. Gosling, B. Joy, G. Steele, G. Bracha. — 3 edition. — Amsterdam: Addison-Wesley Longman, 2005. — June. — P. 688.

[39] *JetBrains*. Meta Programming System (MPS). — http://www.jetbrains.com/mps. — 2009.

[40] *Johnson, S. C.* Yacc: Yet another compiler-compiler: Tech. rep. / S. C. Johnson: Bell Laboratories, 1979.

[41] *Jouault, F.* Km3: A dsl for metamodel specification / F. Jouault, J. Bézivin // FMOODS / Ed. by R. Gorrieri, H. Wehrheim. — Vol. 4037 of *Lecture Notes in Computer Science*. — Springer, 2006. — Pp. 171–185.

[42] *Jouault, F.* Tcs:: a dsl for the specification of textual concrete syntaxes in model engineering / F. Jouault, J. Bézivin, I. Kurtev // GPCE '06: Proceedings of the 5th international conference on Generative programming and component engineering. — New York, NY, USA: ACM, 2006. — Pp. 249–254.

[43] *Kastens, U.* Ordered attributed grammars / U. Kastens // *Acta Informatica*. — 1980. — Vol. 13, no. 3. — Pp. 229–256.

[44] *Kernighan, B. W.* The C programming language / B. W. Kernighan, D. M. Ritchie. — Prentice-Hall, Englewood Cliffs, N.J. :, 1978. — Pp. x, 228 p. ;.

[45] *Khedker, U.* Data Flow Analysis: Theory and Practice / U. Khedker, A. Sanyal, B. Karkare. — Boca Raton, FL, USA: CRC Press, Inc., 2009.

[46] *Klint, P.* Toward an engineering discipline for grammarware / P. Klint, R. Lämmel, C. Verhoef // *ACM Trans. Softw. Eng. Methodol.* — 2005. — Vol. 14, no. 3. — Pp. 331–380.

[47] *Klint, P.* Term rewriting meets aspect-oriented programming. / P. Klint, T. van der Storm, J. J. Vinju // Processes, Terms and Cycles / Ed. by A. Middeldorp, V. van Oostrom, F. van Raamsdonk, R. C. de Vrijer. — Vol. 3838 of *Lecture Notes in Computer Science*. — Springer, 2005. — Pp. 88–105.

[48] *Knuth, D. E.* Semantics of context-free languages / D. E. Knuth // *Theory of Computing Systems*. — 1968. — June. — Vol. 2, no. 2. — Pp. 127–145.

[49] *Kodaganallur, V.* Incorporating language processing into java applications: A JavaCC tutorial / V. Kodaganallur // *IEEE Software*. — 2004. — Vol. 21, no. 4. — Pp. 70–77.

[50] *Lämmel, R.* Recovering Grammar Relationships for the Java Language Specification / R. Lämmel, V. Zaytsev // Ninth IEEE International Working Conference on Source Code Analysis and Manipulation. — IEEE, 2009. — Сентябрь. — Pp. 178–186. — Full version submitted for journal publication.

[51] *Latendresse, M.* Regreg:a lightweight generator of robust parsers for irregular languages / M. Latendresse // *Reverse Engineering, Working Conference on.* — 2003. — Vol. 0. — P. 206.

[52] *Levine, J.* Flex & Bison / J. Levine. — O'Reilly, 2009.

[53] *McPeak, S. G.* Elkhound: A fast, practical glr parser generator: Tech. rep. / S. G. McPeak. — Berkeley, CA, USA: 2003.

[54] Meta-object facility. — http://www.omg.org/mof/.

[55] Model driven architecture. — http://www.omg.org/mda/.

[56] *Moon, D. A.* Programming language for old timers. — http://users.rcn.com/david-moon/PLOT.

[57] *Mossenbock, H.* Coco/r - a generator for fast compiler front ends: Tech. rep. / H. Mossenbock: 1990.

[58] *Nickel, U.* The fujaba environment / U. Nickel, J. Niere, A. Zündorf // ICSE '00: Proceedings of the 22nd international conference on Software engineering. — New York, NY, USA: ACM, 2000. — Pp. 742–745.

[59] *Odersky, M.* Programming in Scala: [a comprehensive step-by-step guide] / M. Odersky, L. Spoon, B. Venners. — artima, 2008. — Vol. 1. ed., version 5. — P. 736.

[60] On language-independent model modularisation / F. Heidenreich, J. Henriksson, J. Johannes, S. Zschaler. — 2009. — Pp. 39–82.

[61] Open fortran parser. — http://fortran-parser.sourceforge.net.

[62] An overview of AspectJ / G. Kiczales, E. Hilsdale, J. Hugunin et al. // ECOOP '01: Proceedings of the 15th European Conference on Object-Oriented Programming. — London, UK: Springer-Verlag, 2001. — Pp. 327–353.

[63] *Parr, T.* The Definitive ANTLR Reference: Building Domain-Specific Languages / T. Parr. Pragmatic Programmers. — First edition. — Pragmatic Bookshelf, 2007. — Май.

[64] *Parr, T. J.* Antlr: A predicated- :::: Ll(k) :::: Parser generator / T. J. Parr, R. W. Quong // *SPE*. — 1995. — Vol. 25, no. 7. — Pp. 789–810.

[65] *Pierce, B. C.* Types and programming languages / B. C. Pierce. — Cambridge, MA, USA: MIT Press, 2002.

[66] PostgreSQL. — http://www.postgresql.org/.

[67] *Rebernak, D.* A tool for compiler construction based on aspect-oriented specifications / D. Rebernak, M. Mernik // COMPSAC '07: Proceedings of the 31st Annual International Computer Software and Applications Conference. — Washington, DC, USA: IEEE Computer Society, 2007. — Pp. 11–16.

[68] Revised report on the algorithmic language algol 68 / A. van Wijngaarden, B. J. Mailloux, J. E. L. Peck et al. // *Acta Inf.* — 1975. — Vol. 5. — Pp. 1–236.

[69] *Safonov, V. O.* Using Aspect-Oriented Programming for Trustworthy Software Development / V. O. Safonov. — New York, NY, USA: Wiley-Interscience, 2008.

[70] Separation of concerns in compiler development using aspect-orientation / X. Wu, B. R. Bryant, J. Gray et al. // SAC '06: Proceedings of the 2006 ACM symposium on Applied computing. — New York, NY, USA: ACM, 2006. — Pp. 1585–1590.

[71] *Sheard, T.* Template meta-programming for haskell / T. Sheard, S. P. Jones // Haskell '02: Proceedings of the 2002 ACM SIGPLAN

workshop on Haskell. — New York, NY, USA: ACM, 2002. — Pp. 1–16.

[72] *Shonle, M.* Xaspects: an extensible system for domain-specific aspect languages / M. Shonle, K. Lieberherr, A. Shah // OOPSLA '03: Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications. — New York, NY, USA: ACM, 2003. — Pp. 28–37.

[73] Silver: an extensible attribute grammar system / E. Van Wyk, D. Bodin, J. Gao, L. Krishnan // *ENTCS*. — 2008. — Vol. 203, no. 2. — Pp. 103–116.

[74] *Skalski, K.* Meta-programming in Nemerle. — http://nemerle.org/metaprogramming.pdf. — 2004.

[75] *Steele Jr., G. L.* Common LISP: the language / G. L. Steele, Jr. — Newton, MA, USA: Digital Press, 1984.

[76] Stratego/xt 0.17. a language and toolset for program transformation / M. Bravenboer, K. T. Kalleberg, R. Vermaas, E. Visser // *Sci. Comput. Program.* — 2008. — Vol. 72, no. 1-2. — Pp. 52–70.

[77] *Stroustrup, B.* The C++ Programming Language / B. Stroustrup. — Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2000.

[78] *Van Wyk, E.* Implementing aspect-oriented programming constructs as modular language extensions / E. Van Wyk // *Sci. Comput. Program.* — 2007. — Vol. 68, no. 1. — Pp. 38–61.

[79] *Varró, D.* Metamodeling mathematics: A precise and visual framework for describing semantics domains of uml models / D. Varró, A. Pataricza // UML '02: Proceedings of the 5th

International Conference on The Unified Modeling Language. — London, UK: Springer-Verlag, 2002. — Pp. 18–33.

[80] Weaving a debugging aspect into domain-specific language grammars / H. Wu, J. Gray, S. Roychoudhury, M. Mernik // SAC '05: Proceedings of the 2005 ACM symposium on Applied computing. — New York, NY, USA: ACM, 2005. — Pp. 1370–1374.

[81] *Wö■, A.* LL(1) conflict resolution in a recursive descent compiler generator / A. Wö■, M. Löberbauer, H. Mössenböck // JMLC / Ed. by L. Böszörményi, P. Schojer. — Vol. 2789 of *Lecture Notes in Computer Science*. — Springer, 2003. — Pp. 192–201.

[82] Xml metadata interchange. — http://www.omg.org/xmi/.

[83] *Behrens, H.* Xtext User Guide. — http://www.eclipse.org/Xtext/documentation/0_7_2/xtext.pdf. — 2009.

[84] *Zdun, U.* A dsl toolkit for deferring architectural decisions in dsl-based software design / U. Zdun // *Information and Software Technology*. — 2010. — Vol. 52. — Pp. 733–748.