УДК 681.3

ПОНЯТИЙНЫЙ АНАЛИЗ И КОНТЕКСТНАЯ ТЕХНОЛОГИЯ ПРОГРАММИРОВАНИЯ

В. С. Выхованец[(1)], В. Я. Иосенкин[(2)]

[(1)] *Институт проблем управления им. В. А. Трапезникова, г. Москва;*
[(2)] *Приднестровский государственный университет им. Т. Г. Шевченко, г. Тирасполь*

Рассмотрена контекстная технология программирования, основанная на создании специализированного языка для решения заданного класса прикладных задач путем понятийного анализа предметной области и отражения ее понятийной структуры в понятиях языка. Описаны средства для задания семантики определяемого языка. На примерах программ показана возможность реализации в рамках контекстной технологии некоторых известных методов описания семантики.

ВВЕДЕНИЕ

Современные информационные технологии характеризуются возрастающей сложностью *информационных систем*, создаваемых в различных областях и предназначенных для хранения, обработки, поиска, распространения, передачи и предоставления информации. Накопленный опыт их проектирования показывает сложность и трудоемкость этого процесса, длительного по времени и требующего высокой квалификации участвующих в нем специалистов.

До сих пор проектирование информационных систем выполняется в основном на интуитивном уровне с применением неформализованных методов, основанных на искусстве проектировщиков, их практическом опыте, экспертных оценках и дорогостоящих экспериментальных проверках получаемых результатов. Кроме того, в процессе жизненного цикла информационная система подлежит постоянному изменению в соответствии с изменяющимися потребностями пользователей и развитием их представлений о предметной области, что еще более усложняет разработку и сопровождение таких систем.

Структурная методология предоставила в распоряжение разработчиков методы структурной декомпозиции предметной области. Наглядность и строгость *структурного анализа* позволяет разработчикам и пользователям системы формализовать представления, необходимые для адекватной реализации информационной системы.

В структурном анализе и проектировании применяются различные модели, описывающие функциональную структуру системы (модель SADT) [1], последовательность выполняемых действий (модель IDEF3) [2], передачу данных между функциональными процессами (диаграммы потоков данных DFD) [3], отношения между данными (модель “сущность—связь” или ERM) [4].

Однако широкое применение структурного анализа и следование его рекомендациям связано с трудностями создания детальных формальных спецификаций информационной системы, проверки их на полноту и непротиворечивость. Еще большие трудности ожидают разработчиков при попытке изменения этих спецификаций.

На смену структурному анализу и как естественное его развитие пришла методология *объектно ориентированного анализа* и одноименная технология программирования. Объектно ориентированный анализ необходим для создания объектной модели предметной области. Основные принципы объектно ориентированной технологии (децентрализация, контракты, самодостаточность, классификация, бесшовность) [5] реализуются методами наследования, инкапсуляции и полиморфизма при структурной декомпозиции предметной области [6]. При этом используются такие понятия как объект, класс, атрибут, метод, интерфейс и др.

Формальная спецификация результатов объектно ориентированного анализа осуществляется на языке моделирования UML (Unified Modeling Language) [7], который содержит графические и языковые средства для определения, представления, проектирования и доку-

ментирования результатов анализа предметной области. Язык поддерживает стандартный набор *декомпозиционных схем*, реализуемых в виде всевозможных диаграмм и нотаций. В UML отражена как методология структурного анализа в виде диаграмм классов, компонентов и их размещения, так и объектно ориентированная методология, представленная в виде различных моделей поведения объектов: варианты применения, взаимодействие, состояния, деятельность.

Отличительное свойство UML от других средств моделирования (IDEF3, DFD, ERM) — возможность расширения языка, основанная на таких сущностях, как стереотипы, теги и ограничения. Однако главное достоинство объектно ориентированного подхода состоит в том, что создаваемые объектно ориентированные модели более открыты и легче поддаются внесению изменений, поскольку их конструкция базируется на устойчивых формах отражения результатов декомпозиции. Это дает возможность модели развиваться постепенно и не приводит к полной ее переработке даже в случае существенных изменений представлений о предметной области и требований к проектируемой информационной системе.

Известный недостаток современных технологий программирования заключается в наличии достаточно большого семантического (смыслового) разрыва между содержательными представлениями о предметной области и решаемыми задачами и теми средствами, которые заложены в языке программирования для решения задач. Иными словами, *семантический разрыв* — это явление несоответствия решаемых задач тем средствам, которые используются для их решения.

Одна из особенностей семантического разрыва определяется различием системы понятий языка программирования и системы понятий, применяемых для постановки и решения задач в той или иной предметной области. По своей сути любой универсальный язык программирования навязывает разработчику программы некоторую систему понятий, в то время как высокоуровневая модель предметной области с этими понятиями согласуется плохо или не согласуется вообще.

Высокая стоимость информационных систем, их низкая эффективность и надежность, объективная трудоемкость, интеллектуальная и технологическая сложность процесса программирования в некоторой степени являются последствиями семантического разрыва. Для сокращения семантического разрыва используется повышение уровня абстракций языков программирования [8]. Однако последнее снимает только часть проблем, не затрагивая существенным образом понятийную систему языка программирования.

Настоящая статья посвящена контекстной технологии программирования, которая зародилась как объединение объектно ориентированной и фортоподобной технологий, осуществленное путем контекстной интерпретации слов фортоподобного языка [9] и на основе понятийного анализа предметной области.

С целью сокращения семантического разрыва средствами контекстной технологии создается специализированный язык, отражающий понятийную структуру предметной области и класс решаемых задач. Такой подход основывается на допущении, что уже в процессе изучения предметной области и специфики заданного класса прикладных задач, еще до начала программирования, формируется система понятий, приспособленная для постановки и решения задач, которая и найдет свое отражение в создаваемом языке.

Другой базовый принцип, именем которого названа представляемая технология, состоит в контекстной интерпретации лексем. Контекстная интерпретация как принцип заключается в определении семантического (смыслового) значения лексем специализированного языка в зависимости от окружающего их контекста. В отличие от генераторов компиляторов [10], у которых возможности задания контекстных условий, необходимых для реализации языков программирования высокого уровня, достаточно бедны, в контекстной технологии контекстные условия не только систематически задаются, но и естественным образом используются. Это позволяет не только улучшить выразительные возможности определяемого языка, но и повысить его уровень.

И, наконец, в связи с зарождением контекстной технологии как некоторого развития объектно ориентированной, ей не чужды все те принципы и методы, на которых основана последняя.

1. КОНТЕКСТНАЯ ТЕХНОЛОГИЯ ПРОГРАММИРОВАНИЯ

Данную технологию будем рассматривать как метаязык и систему программирования, которые объединены в рамках единой методологии, определяемой как совокупность методов, применяемых в процессе разработки программ и объединенных одним общим подходом.

1.1. Понятийная модель и прикладные задачи

Под *предметной областью* будем понимать выделенный фрагмент реальной действительности, представляемый в нашем сознании совокупностью некоторых сущностей (предметов, процессов, явлений, задач и т. д.), которые могут быть уникально идентифицированы и описаны.

Наиболее естественным способом описания сущностей предметной области видится соотнесение с ними совокупности понятий, образующих ее понятийную структуру [11]. В процессе программирования будем строить ее *понятийную модель*, для чего предусмотрим специфические языковые средства.

Понятийная модель строится путём выявления и выражения *понятийной структуры* предметной области в объеме, достаточном для решения некоторого класса прикладных задач. В отличие от других технологий, например структурной, основанной на структурном анализе предметной области, и объектно ориентированной, основанной на объектном анализе, контекстную технологию программирования определим на основе понятийного анализа предметной области.

В качестве программы выступает понятийная модель предметной области, дополненная описанием решения прикладной задачи. При этом одна и та же модель может применяться для решения разных или нескольких прикладных задач. В понятийной модели условно выделяются четыре части:

— понятийная структура предметной области;

— синтаксические правила выражения понятий в тексте;

— описание семантики правил выражения понятий;

— описание процесса компиляции.

Понятийная структура и синтаксические правила описываются на декларируемом в контекстной технологии *метаязыке*, а описание семантики и решаемой задачи выполняются на специализированном языке, задаваемом понятийной структурой и синтаксическими правилами. Тогда в результате описания понятийной структуры предметной области и синтаксических правил для выражения понятий определяется некоторый специализированный язык программирования, на котором осуществляется как решение прикладной задачи, так и частичное описание его семантики.

Таким образом, понятийная структура и синтаксические правила задают синтаксис определяемого в модели специализированного языка. *Определяемый язык*, в свою очередь, применяется как для задания семантики синтаксических правил, так и для описания решения прикладной задачи.

1.2. Грамматический разбор и компиляция

Компиляция понятийной структуры и синтаксических правил в предлагаемой технологии осуществляется традиционными методами на основе определения синтаксиса некоторого метаязыка в виде формальной грамматики и содержательного описания его семантики. Компиляция описаний семантики для правил выражения понятий и текста решения прикладной задачи осуществляется создаваемым в процессе программирования специализированным компилятором. Последнее обеспечивается тем, что любое синтаксическое правило может дополняться описанием процесса его компиляции, которое также выражается на определяемом языке.

Синтаксический анализ и грамматический разбор части программы, написанной на метаязыке контекстной технологии, реализуется известными средствами.

Синтаксический анализ и грамматический разбор описаний семантики правил, решаемой задачи и процесса компиляции выполняется специально разработанными для контекстной технологии методами синтаксического анализа и грамматического разбора.

Для повышения эффективности компилятора системы контекстного программирования применяется *разнесенный грамматический разбор*. Последний основан на контекстной интерпретации лексем и заключается в разделении определения применимости синтаксических правил на две части: на контекстное сопоставление, осуществляемое при просмотре назад, и структурное сопоставление, выполняемое при просмотре вперед. Описание компилятора выходит за рамки настоящей статьи, в связи с чем процесс компиляции здесь не рассматривается.

1.3. Семантический разрыв

Как следствие формирования в процессе программирования специализированного языка для описания решаемой задачи, применение контекстной технологии программирования направлено на сокращение семантического разрыва между высокоуровневой моделью предметной областью и языком программирования. Более того, в результате приближения выразительных средств языка программирования к постановке и решению прикладных задач следует ожидать повышения качества и надежности информационных систем, созданных на основе контекстной технологии.

Другой особенностью контекстной технологии, вытекающей из наличия выразительных средств описания процесса компиляции, следует назвать возможность использования системы контекстного программирования совместно с любой из известных целевых платформ, например:

— с микроконтроллерами, не имеющими операционной системы;

— с вычислительными системами, имеющими развитую операционную среду;

— с платформенно-независимыми виртуальными машинами;

— с другими системами программирования.

Очевидно, в случае аппаратной платформы привязка определяемого в модели специализированного языка к целевой платформе осуществляется низкоуровневыми средствами, например, путем прямой генерации кода. Другой крайний случай заключается в привязке определяемого языка к целевой платформе с помощью другой системы программирования, и тогда определяемый язык описывается на языке, поддерживаемом этой системой. Для исследования контекстной технологии авторами реализована система программирования, поддерживающая эти крайние случаи.

Не претендуя на полноту изложения, кратко опишем только те процессы, которые актуальны в рамках настоящей статьи и существенным образом отличают контекстную технологию программирования от других технологий. Для раскрытия содержания контекстной технологии рассмотрим более детально составные части понятийной модели.

2. МЕТАЯЗЫК КОНТЕКСТНОЙ ТЕХНОЛОГИИ

2.1. Базовые понятия

Терм определим как элементарную синтаксическую единицу текста на некотором формальном языке, которая выражается последовательностью *знаков алфавита* этого языка. Сам алфавит и входящие в него знаки назовем *терминальными*.

Текстом будем называть последовательность лексем, предназначенную для выражения некоторого сложного смысла. *Лексему* определим как элементарную смысловую (семантическую) единицу текста, выраженную одним или несколькими термами. Иными словами, лексема — это множество термов вместе с приписанным (сопоставленным) им смыслом. Лексема, выраженная одним терминальным знаком, называется *символом*. Текст, в отличие от лексем, предполагает свое деление на смысловые части, в то время как лексемы такого деления не допускают.

Для выражения синтаксиса текстов на определяемом языке воспользуемся известным формализмом контекстно-свободных грамматик [12]. *Контекстно-свободной грамматикой* называется формальная система <T, N, S, I>, состоящая из терминального алфавита T, нетерминального алфавита N, множества правил вывода, или предложений, S вида $A \to \alpha$ и выделенного не-

терминального знака I, называемого аксиомой, где A — нетерминальный знак, а α — конечная последовательность терминальных и нетерминальных знаков.

Для обозначения термов в правилах грамматики будем пользоваться одинарными кавычками. Например, 'x' означает терм, состоящий из одного знака x. В свою очередь, последовательностями терминальных знаков, не заключенными в одинарные кавычки, обозначим нетерминальные знаки грамматики. Для разделения нетерминальных знаков между собой будем пользоваться пробелами, для чего запретим появление пробела в обозначении нетерминальных знаков.

Как обычно, каждому нетерминальному знаку грамматики сопоставим некоторое понятие, называемое нетерминальным понятием языка, которое для краткости будем именовать просто понятием. Тогда правила вывода или предложения грамматики могут быть интерпретированы как синтаксические правила выражения понятий в тексте.

Под *контекстной интерпретацией* терма будем понимать определение семантического значения терма по его месту в тексте. В общем случае один и тот же терм может быть сопоставлен различным лексемам, т. е. одна и та же последовательность терминальных знаков может служить носителем различных смыслов. Следовательно, при контекстной интерпретации семантическое значение терма в общем случае определяться окружающим его контекстом.

Пример 1. Пусть в грамматике языка имеется два правила с термом 'x': C → A 'x' B и F → 'x' E, где A, B, C, E и F — некоторые понятия, а знак → разделяет определяемое понятие (слева) от понятий и термов (справа), используемых для его выражения в тексте. Таким образом, понятие C определяется первым предложением, а понятие F — вторым. В свою очередь, где-то имеются предложения, определяющие понятия A, B и E.

Анализ приведенных предложений показывает, что если терм 'x' встретится в окружении последовательности терминальных знаков, выражающих понятия A и B, то он интерпретируется как лексема x первого предложения. В свою очередь, A 'x' B (как и порожденная в соответствии с ней последовательность термов) выражает пример понятия C. Если терм 'x' встретится в контексте 'x' E, то он сопоставляется с лексемой x второго предложения, а соответствующая часть текста выражает пример понятия F. Очевидно, если ни один из разрешенных контекстов терма 'x' не распознан, то это говорит об ошибке в тексте. ♦

В демонстрационных целях выберем в качестве языка, определяемого средствами контекстной технологии, язык булевых выражений, описываемый обозримой в рамках настоящей статьи грамматикой.

Пример 2. Язык булевых выражений зададим следующей грамматикой:

Boolean → 'false'
Boolean → 'true'
Boolean → ″[A–Za–z][A–Za–z0–9]*″
Boolean → '(' Boolean ')'
Boolean → 'not' Boolean
Boolean → Boolean 'and' Boolean
Boolean → Boolean 'or' Boolean
Boolean → Boolean 'imp' Boolean,

где нетерминальный алфавит состоит из одного понятия Boolean (аксиома грамматики).

Например, булево выражение '(not x or y) and z' порождается этой грамматикой. Заметим, что третье предложение задает синтаксис выражения переменных, определенный на языке регулярных выражений [13], где регулярное выражение представлено в виде терминального шаблона и для отличия от терма заключено в двойные кавычки. ♦

2.2. Грамматика метаязыка

Метаязык контекстной технологии определим приведенной на рисунке грамматикой, которая описана с точностью до обозначенных курсивом нетерминальных знаков *notion*, *aspect* и *string*. Описываемый метаязык является развитием контекстного языка из работы [14].

Текст программы program состоит из последовательности *деклараций* declaration и *ситуаций* situation (строка *1*). В декларативной части (последовательности деклараций) определяется язык, на котором в ситуационной части описывается решение некоторой прикладной задачи.

Отдельная декларация представляет собой законченный фрагмент определения языка, а ситуация описывает решение прикладной задачи полностью, если в ранее следующих декларациях определяемый язык задан полностью, или частично, когда язык определен в объеме некоторого своего подязыка, уже достаточного для описания части решения.

Декларативная часть состоит из описаний сущностей предметной области essence (строка *2*). Каждой сущности присваивается имя notion нетерминального понятия определяемого языка, а само понятие задается как находящееся в отношении агрегации aggregation и обобщения generalization с системой notions других, ранее определенных понятий (строки *3—5*). Для разделения понятий в списке notions используется один или несколько пробелов space (строка *17*), а при их отсутствии — пустой знак empty, который определен как отсутствие каких-либо знаков или как последовательность пробелов (строка *18*).

Описание любой сущности essence одновременно содержит указание как на обобщение, так и на агрегацию (строка *3*). Более того, любая сущность может быть определена с помощью этих и только этих абстракций (строка *3*, первое правило). Для выражения перекрестных и рекурсивных связей между сущностями используется их предварительное объявление, которое не включает конструкцию aggregation (строка *3*, второе правило). Таким образом, описание *сущности*, в отличие от описания составляющего его понятия, содержит определение существующего для него отношения обобщения.

Каждое описание description содержит предложение sentence (строка *7*), состоящее из последовательности элементов item (строка *8*): имен понятий notion, лексем lexeme и текстов компиляции compilation (строка *9*). Лексема является терминальным понятием определяемого языка. Для выражения лексем могут использоваться как термы term, так и множества термов, задаваемые на языке регулярных выражений [13] в виде терминальных шаблонов pattern (строки *10—12*).

С каждым предложением связывается определяемое *понятие-результат*, именем которого названа описываемая сущность (строка *3*), определяемая предложением полностью, когда больше нет предложений с тем же понятием-результатом, или частично, если такие предложения имеются (строки *6* и *7*).

1	program	→	declaration situation | declaration situation program
2	declaration	→	essence | essence declaration
3	essence	→	generalization *notion* space aggregation | generalization *notion*
4	generalization	→	'(' notions ')' | '(' empty ')'
5	notions	→	notion | notion space notions
6	aggregation	→	description | description aggregation
7	description	→	sentence semantics
8	sentence	→	item | item space sentence
9	item	→	notion | lexeme | compilation
10	lexeme	→	term | pattern
11	term	→	'′' *string* '′'
12	pattern	→	'″' *string* '″'
13	semantics	→	imperative | imperative space semantics
14	imperative	→	'{' text '}' | *aspect* '{' text '}'
15	text	→	phrase | phrase text
16	phrase	→	empty | string | aspect '{' text '}'
17	space	→	' ' | ' ' space
18	empty	→	'' | space
19	compilation	→	'[' text ']'
20	situation	→	'[' text ']'

Порождающая грамматика метаязыка

Пример 3. Представим грамматику из примера 2 на метаязыке контекстной технологии в виде декларации сущности Boolean:

```
() Boolean
        'false' {}
        'true' {}
        ″[A–Za–z][A–Za–z0–9]*″ {}
        '(' Boolean ')' {}
        'not' Boolean {}
        Boolean 'and' Boolean {}
        Boolean 'or' Boolean {}
        Boolean 'imp' Boolean {},
```

где для разделения сущностей, если их несколько, использованы пустые круглые скобки конструкции generalization, а для разделения предложений внутри определения сущности – пустые фигурные скобки конструкции semantics. Для приведенного фрагмента понятийной модели может быть задана ситуация, например, в виде текста '(not x or y) and z', являющегося решением некоторой прикладной задачи, например, вычислением значения булевого выражения. ♦

По своей сути предложение sentence является представлением правил грамматики определяемого языка на метаязыке контекстной технологии с тем отличием, что для задания особенностей компиляции предложения используются конструкции compilation (строка *19*). Их роль — указание компилятору системы контекстного программирования на действия, которые необходимо выполнить в процессе распознавания предложения в тексте, например, проверку более сложных контекстных условий, чем это задается средствами метаязыка (контекстным условиям посвящен п. 2.6). Описание системы контекстного программирования выходит за рамки настоящей статьи, и конструкция compilation более детально не рассматривается.

Рассмотрим теперь семантику описания отношений обобщения и агрегации. Оставшиеся нераскрытыми конструкции метаязыка и их семантика будут описаны позже.

2.3. Понятийный анализ

Абстрагирование понятий представляет собой основной механизм, с помощью которого человек познает окружающий мир. *Абстракция* — это выделение существенных признаков и связей понятий и игнорирование несущественных. Абстрагирование является основным методическим приемом понятийного анализа и используется при разработке понятийной модели. Абстрагирование позволяет декомпозировать предметную область на значимые сущности и построить ее понятийную структуру, адекватную решаемым задачам.

Понятийный анализ определим как методику построения понятийной структуры, основанную на абстрагировании понятий и осуществляемую путем сравнения и сопоставления сущностей предметной области с целью выявления их общих признаков, выделения понятий, объединения понятий на основе их сходства или подобия и т. п.

Пример 4. Детализируем описание языка булевых выражений введением дополнительных понятий:

```
() Constant
        'false' {}
        'true' {}
() Variable
        ″[A–Za–z][A–Za–z0-9]*″ {}
() Logic
        Variable {}
        '(' Boolean ')' {}
```

```
() Negation
        'not' Constant {}
        'not' Logic {}
() Conjunction
        Negation 'and' Negation {}
() Disjunction
        Conjunction 'or' Conjunction {}
() Boolean
        Disjunction 'imp' Disjunction {}
```

В рассматриваемом примере Constant обозначает понятие “логическая константа”, Variable — понятие “пропозиционная переменная”, а Logic — понятие “логическое значение”. Остальные понятия соответствуют более сложным представлениям, правила выражения которых декларируются в примере: Negation обозначает понятие “отрицание”, Conjunction — понятие “конъюнкция”, а Disjunction — понятие “дизъюнкция”.

Очевидно, наиболее сложным понятием является Boolean, обозначающее понятие “булево выражение”. Понятия Constant, Logic, Negation, Conjunction и Disjunction являются частными случаями Boolean. Однако выражение этих частных случаев осуществляется по-разному, что и задается в виде соответствующих предложений. ♦

Из примера видно, что имеются различные взаимосвязи понятий, появляющиеся в результате их абстрагирования. Известны следующие основные типы абстракции [15]: ассоциация, обобщение, типизация и агрегация, причем фундаментальными являются только обобщение и агрегация. Абстракция типизации является частным случаем обобщения, а абстракция ассоциации эквивалентна абстракции агрегации по способу формирования [16]. Рассмотрим средства выражения обобщения и агрегации в понятийной модели, не останавливаясь на особенностях выражения частных типов абстракции.

2.4. Абстракция обобщения

Обобщение понятий — это форма порождения нового понятия на основе одного или нескольких подобных понятий, когда порождаемое понятие сохраняет общие признаки исходных понятий, но игнорирует их более тонкие различия. Для обобщения возможен противоположный процесс, когда исходное понятие делится на несколько более узких. Такой процесс называется *специализацией*, или *ограничением* понятий.

Следовательно, при обобщении подобные видовые понятия соотносятся с родовым, а при специализации, наоборот — родовые понятия делятся на видовые.

Пример 5. Выразим абстракцию обобщения в понятийной модели из Примера 4. В итоге получим:

```
() Constant
        'false' {}
        'true' {}
() Variable
        ″[A–Za–z][A–Za–z0–9]*″ {}
(Variable) Logic
        Variable {}
        '(' Boolean ')' {}
(Constant Logic) Negation
        'not' Logic {}
        'not' Constant {}
(Negation) Conjunction
        Negation 'and' Negation {}
(Conjunction) Disjunction
        Conjunction 'or' Conjunction {}
(Disjunction) Boolean
        Disjunction 'imp' Disjunction {}
```

Видно, что для описания понятия Negation необходимо понятие Logic и Constant, т. е. понятие Negation получено в результате обобщения понятий Logic и Constant. В свою очередь, понятия Logic и Constant являются конкретизацией понятия Negation. Последнее означает, любое выражение в тексте понятий Logic или Constant является и выражением некоторого частного случая обобщающего понятия Negation. ♦

Метаязык понятийной модели позволяет выразить произвольные обобщения (строки *3* и *4*). При этом между понятиями устанавливается связь типа “есть подкласс”. Если понятия видового уровня связаны отношением обобщения только с одним понятием родового уровня, то возникает древовидная структура обобщения. В противном случае имеем некоторую сетевую структуру. Для указания на отсутствие обобщения у некоторого понятия используются пустые круглые скобки (строка *4*). В этом случае понятие считается обобщенным от пустого или универсального понятия empty, обозначаемого как '()'.

Обобщение несколько похоже на наследование, используемое в объектно ориентированных языках, но к последнему не сводится. При выявлении родовых понятий обобщение указывает на его видовые составляющие. Это позволяет при задании родовых понятий не повторять описания его видовых реализаций. В объектно ориентированных языках базовый (родовой) класс реализует общие свойства и методы, которые распространяются на все наследуемые (видовые) классы. Однако базовый класс теряет при этом знание о своих видовых реализациях. Для решения этой проблемы в объектно ориентированных языках был введен механизм виртуальных методов.

2.5. Абстракция агрегации

Агрегация понятий используется в тех случаях, когда вновь порождаемое сложное понятие включает исходные понятия в качестве своих составных частей. Абстракция агрегации позволяет выразить семантику внутренних связей, существующих между отдельными сущностями предметной области. При этом структура сложного понятия раскрывается путем его расчленения на совокупность составляющих его понятий. Процесс, противоположный агрегации, называется *декомпозицией*.

Агрегация в понятийной модели выражается предложениями sentence, задающие связь агрегируемых понятий с понятием-результатом (строки *7* и *8*, см. рисунок). Конструкция aggregation позволяет выразить агрегацию одного понятия или когда в качестве агрегируемых понятий используется одна или несколько лексем. Последнее соответствует агрегации без предварительного объявления агрегируемых понятий путем их прямого (непосредственного) выражения в тексте. Таким образом, лексема может рассматриваться как *терминальное понятие*, имеющее прямое (непосредственное) выражение в тексте программы в виде одной или нескольких терминальных строк.

Пример 6. Продемонстрируем агрегацию понятий на примере описания комплексных чисел:

```
(Real) Complex
        '(' Real ',' Real ')' {}
        Real '+' Real 'i' {}
```

где Real − понятие действительного числа. Приведенные предложения определяют два способа выражения комплексных чисел как состоящих из двух действительных, например

'(1, 2)' и '1 + 2i'. Заметим, что понятие Complex обобщено от понятия Real. Это позволяет комплексные числа выражать действительными. Например, терм '2' в соответствующем контексте может быть интерпретирован как комплексное число.♦

Абстракция агрегации позволяет конструировать новые понятия на основе других, ранее определенных, причем выполнять такое конструирование в каждом предложении по-своему. Последнее отличает агрегацию в контекстной технологии от агрегации в объектно ориентированных языках, где сложный класс конструируется путем простого и единственного перечисления агрегированных в него подклассов.

Таким образом, конструкция essence (строка *3*) позволяет рассматривать абстракции агрегации и обобщения в более широком смысле, чем это определено и использовано в объектно ориентированной технологии программирования.

2.6. Контекстные условия

Известно, что контекстно-свободные грамматики не отражают синтаксиса современных языков программирования. Это обстоятельство вызвано тем, что эти языки не являются контекстно-свободными и имеют более сложную синтаксическую структуру [17]. Синтаксические правила, которые не описываются контекстно-свободными грамматиками, называются *контекстными условиями*.

Проверку контекстных условий связывают с созданием и использованием терминальных понятий языка, которые имеют различную семантику. Задача контекстного анализа заключается в установлении правильности использования таких понятий, а наиболее часто решаемая задача состоит в определении существования терминального понятия и соответствия его использования одному из возможных для него контекстов.

Контекстные условия в метаязыке задаются для имен понятий notion, имен аспектов aspect и лексем lexeme. На рисунке места определения контекстных условий выделены курсивом.

Определение имени понятия *notion* как терминального понятия метаязыка осуществляется при описании сущности (строка *3*). После такого определения имена понятий notion могут появляться при описании обобщения в списке notions (строки *4* и *5*) и в качестве item при описании предложений (строки *8* и *9*).

Имена аспектов *aspect* служат для задания имен конструкций imperative (строка *14*), назначение которых — задание той или иной семантической интерпретации фрагмента текста phrase (строка *16*) при описании императивов (строка *14*), текста компиляции (строка *19*) и ситуаций (строка *20*). Более подробно императивы и аспекты будут рассмотрены в § 3.

Лексемы lexeme в метаязыке задаются в виде термов term и шаблонов pattern (строка *10*). Как терм, так и шаблон используются для задания терминальных понятий *string* определяемых в модели языка (строки *11* и *12*). Причем последние сохраняются не в таблице идентификаторов, как это было для понятий и аспектов, а непосредственно в структуре описываемого предложения sentence (строки *8* и *9*).

В заключение заметим, что имена понятий notion являются терминальными понятиями метаязыка и одновременно нетерминальными понятиями определяемого языка. Имена аспектов aspect, как и имена понятий notion, являются терминальными понятиями для метаязыка, но в определяемом языке аспекты интерпретируются как терминальные понятия. В свою очередь, лексемы lexeme представляются терминальными понятиями *string* определяемого языка.

3. СРЕДСТВА ОПИСАНИЯ СЕМАНТИКИ

Под *семантикой* обычно понимается смысловая или содержательная интерпретация текста, в то время как форма представления или структура текста задаются его синтаксисом [18]. В свою очередь, семантика языков определяется совокупностью правил и соглашений, устанавливаемых описанием языка и предназначенных для выявления смысла текстов на этом языке с целью их интерпретации человеком или автоматом [19].

По характеру семантики можно выделить два наклонения: повелительное или *императивное*, представленное операторами (командами, предписаниями), и изъявительное или *декларативное*, представленное только описаниями. В одних языках, называемых императивными, преобладает описание действий, т. е. процесса, позволяющего получить результат (языки FORTRAN, ALGOL, PASCAL, C, ADA и др.). Другие языки, называемые декларативными, предполагают не построение (вычисление) результата, а описание (декларацию) свойств решения; на основе этих данных система программирования сама строит некоторую последовательность действий, необходимую для получения результата (языки PROLOG, LISP, и др.).

3.1. Императивы

Семантику предложений понятийной модели будем задавать в виде текста, который построен по правилам определяемого языка и является описанием решения некоторой задачи в аспекте различных проблемных ситуаций. Для этого каждое предложение sentence дополним описанием его семантики semantics, которую выразим как совокупность *императивов* imperative (строки *7* и *13*). Императив задает *семантическую интерпретацию* понятия, реализуемую после компиляции предложения в виде некоторой последовательности действий, которую система программирования выполняет всякий раз, когда понятие выражается этим предложением (строка *14*).

Пример 7. Для модели из Примера 5 зададим императивы, ориентированные на вычисление булевых выражений. Для простоты понимания примера императивы определим в виде последовательности ассемблерных команд достаточно распространенной аппаратной платформы на базе процессора Intel [20].

```
() Variable
        ″[A–Za–z][A–Za–z0–9]*″ { ... }
() Constant
        'false' { mov eax, 0; push eax }
        'true' { mov eax, –1; push eax }
(Variable) Logic
        Variable { pop ebx; mov eax, [ebx]; push eax }
        Integer { pop eax; cmp eax, 0; je label; mov eax, –1; label:
        push eax }
        '(' Boolean ')' {}
(Constant Logic) Negation
        'not' Logic { pop eax; not eax; push eax }
```

```
(Negation) Conjunction
        Negation 'and' Negation { pop eax; pop edx; and eax, edx;
        push eax }
(Conjunction) Disjunction
        Conjunction 'or' Conjunction { pop eax; pop edx; or eax, edx;
        push eax }
(Disjunction) Boolean
        Disjunction 'imp' Disjunction {}
```

В примере для доступа к данным и организации их временного хранения использован аппаратный стек, а для хранения переменных – память произвольного доступа с линейной организацией. В предложении ″[A–Za–z][A–Za–z0–9]*″ текст императива (не показан) описывает создание переменной, путем включения имени переменной в таблицу идентификаторов и выделения для ее хранения памяти требуемого объема.

Логические константы 'false' и 'true' реализованы как занесение нуля и минус единицы на вершину стека, для чего использованы команды загрузки в регистр константы и записи регистра в стек.

Переменная определена адресом ячейки памяти, в которой хранится ее текущее значение. Для получения значения логической переменной Variable ее адрес извлекается из стека, содержимое адресуемой ячейки пересылается в регистр, который затем сохраняется в стеке.

Для преобразования целого числа в булево значение использовано предложение () Logic Integer. Если число на вершине стека не равно нулю, то в регистр записывается минус единица, в противном случае там уже имеется требуемое значение. Результат преобразования сохраняется в стеке.

В свою очередь, предложение '(' Boolean ')' не нуждается в императиве, так как его роль заключается в задании правил разбора и приоритета булева выражения, заключенного в круглые скобки. При изменении решаемой задачи возможно появление императивов и у этого предложения. Реализация остальных предложений не вызывает труда и понятна из текста примера.♦

В контекстной технологии императивы могут представляться как в виде последовательности команд целевой платформы, так и программой на некотором целевом языке программирования. Система контекстного программирования выполняет трансляцию текста императива в код целевой платформы непосредственно или организует такую трансляцию путем вызова соответствующих средств целевой системы программирования. Причем в рамках одного императива требуется только один тип вызова: либо вызов ассемблера целевой платформы, который может быть как внешним, так и внутренним по отношению к системе контекстного программирования, либо вызов компилятора целевой системы программирования, который, как правило, является внешним. По своей сути императивы представляют собой *единицы вызова* целевого языка, в то время как предложение — описание синтаксиса таких вызовов.

3.2. Аспекты

Понятно, что понятийная структура предметной области, представленная в декларативном описании, может быть общей для нескольких задач. Однако решение таких задач может преследовать различные цели, в том числе и взаимоисключающие. Для отражения этой ситуации в контекстной технологии предусмотрена возможность задания для одного предложения нескольких императивов imperative (строка *13*) Для отличия одного императива от другого они именуются и задаются в виде aspect {...}, где aspect — имя определяемого императива, который соответствует именованной семантической интерпретации понятия или его *аспекту* (строка *14*).

Пример 8. Рассмотрим интерпретацию вычисления булевых выражений, связанную с другим кодированием булевых значений. В Примере 7 логический ноль кодировался арифметическим нулем, а логическая единица минус единицей. Определим новую семантическую интерпретацию модели или ее аспект с именем bit, когда логическая единица кодируется арифметической единицей.

```
() Variable
        ″[A–Za–z][A–Za–z0–9]*″ { ... }
() Constant
        'false' { ... }
        'true' { ... } bit { mov eax, 1; push eax }
(Variable) Logic
        Variable { ... }
        Integer { ... } bit { pop eax; and eax, 1; push eax }
        '(' Boolean ')' {}
(Constant Logic) Negation
        'not' Constant { ... }
        'not' Logic { ... }
(Negation) Conjunction
        Negation 'and' Negation { ... }
(Conjunction) Disjunction
        Conjunction 'or' Conjunction { ... }
(Disjunction) Boolean
        Disjunction 'imp' Disjunction { ... } ♦
```

При использовании именованных императивов, определяемых содержательной постановкой задачи, ситуационное описание необходимо дополнить указанием на одну из возможных его семантических интерпретаций. В итоге, ситуационная часть модели может стать независимой от содержательной интерпретации конкретной прикладной задачи и задавать некоторое общее решение для целого класса задач.

3.3. Семантическая замкнутость

Для обеспечения полноты и целостности понятийной модели во всех ее частях (ситуация, семантика, компиляция) целесообразно требовать применения декларируемых языковых конструкций не только для описания ситуационной части (text в строке *20*), но и для задания семантики предложений (text в строке *14*), а также для описания процесса компиляции (text в строке *19*). Тем самым будет реализована семантическая замкнутость описания, что позволит использовать для обработки этих описаний одни и те же средства.

Далее будем рассматривать императивы как текст text, состоящий из фраз phrase (строка *15*), построенный по правилам, задаваемым в декларации (строка *16*). Очевидно, для привязки модели к целевой платформе необходимо некоторое множество предложений реализовать низкоуровневыми средствами или на целевом языке.

Пример 9. Анализ Примера 7 показывает, что все предложения, кроме последнего, реализованы низкоуровневыми средствами. Причем этих предложений уже достаточно для определения семантики последнего предложения средствами самой модели. Для операции импликации $\rightarrow$ справедливо тождество $a \rightarrow b = \bar{a} \vee b$. Отсюда получаем:

```
(Disjunction) Boolean
        Disjunction 'imp' Disjunction { not %1 or %2 },
```

где %1 – интерпретируется как операнд, соответствующий первому понятию предложения (первое вхождение Disjunction), %2 – второму.♦

Из рассмотренного примера следует, что семантически замкнутое императивное описание должно быть

представлено текстом, построенным по правилам, определенным в декларативной части. Более того, семантическая интерпретация этого текста должна соответствовать (прямо или косвенно) понятию-результату. Прямое соответствие означает, что текст императивной части выражает понятие-результат непосредственно, а косвенное — что понятие, выраженное текстом императивного описания, может быть преобразовано в понятие-результат на основе связей обобщения или агрегации.

В свою очередь, текст компиляции (строка *19*) и текст ситуационной части (строка *20*) никакого понятия не выражают и, как следствие этого, их результат должен соответствовать отсутствующему понятию empty, обозначаемому пустыми круглыми скобками (строка *4*).

Примеры описания семантики

На данный момент не существует универсального, общепринятого метода описания семантики [10]. Перечислим наиболее распространенные методы. Денотационный метод основывается на функциональных вычислениях, в которых встроенные операции языка отображаются в однозначные математические объекты, которые затем применяются для описания семантики языковых конструкций. Аксиоматический метод базируется на исчислении предикатов, где результат вычисления описывается через взаимосвязь переменных до и после применения конструкций языка. И, наконец, в операционном методе операции языка описываются через команды некоторой абстрактной машины.

Известно несколько подходов для задания семантики языков, описываемых контекстно-свободными грамматиками: W-грамматики, венский метаязык, аксиоматический и денотационный методы, синтаксически управляемые схемы, атрибутные транслирующие грамматики [21].

Так как в рамках контекстной технологии заявлен некоторый универсальный метод для описания семантики языков программирования, покажем на примерах реализацию двух наиболее распространенных подходов: синтаксически управляемых схем и атрибутных грамматик.

Синтаксически управляемые схемы [22] и атрибутные грамматики [19] позволяют задавать соответствия между текстами входного и выходного языков, называемые *переводом*. Такие соответствия отражают структурные или синтаксические свойства входных и выходных текстов.

Пример 10. Рассмотрим пример реализации в рамках контекстной технологии синтаксически управляемого перевода на примере формального дифференцирования выражений, включающих в себя целочисленные константы, переменную x, функции sin и cos, а также алгебраические операции: изменение знака —, сложение +, вычитание — и умножение*.

Свяжем с каждым предложением модели два перевода, формируемые неименованных императивом и именованным императивом dif. Неименованный императив указывает на то, что выражение не дифференцируется, а именованный императив dif — что выражение необходимо продифференцировать. Формальная производная некоторой строки s — это dif{s}, где dif{...} задает имя семантической интерпретации текста в фигурных скобках. Процесс перевода опишем следующей декларацией:

```
(String) Expression
      ″ [0–9]+ ″
            { #1 } dif { '0' }
      'x'
            { 'x' } dif { '1' }
      '(' Expression ')'
            { '(' & %1 & ')' } dif { '(' & dif { %1 } & ')' }
      'sin' '(' Expression ')'
            { 'sin(' & %1 & ')' } dif { 'cos(' & %1 & ')*(' & dif { %1 } & ')' }
      'cos' '(' Expression ')'
            { 'cos(' & %1 & ')' } dif { '–sin(' & %1 & ')*(' & dif { %1 } & ')' }
      '–' Expression
            { '–' & %1 } dif { '–' & dif { %1 } }
      Expression '*' Expression
            { %1 & '*' & %2 } dif { '(' & %1 & '*' & dif { %2 } & '+' & dif {
            %1 } & '*' & %2 & ')' }
      Expression ″+ | –″ Expression
            { %1 & #1 & %2 } dif { '(' & dif { %1 } & #1 & dif { %2 } & ')' },
```

где & — операция конкатенации строк (определена при описании String, в примере не показано), #n — зарезервированный символ синтаксического анализатора, возвращающий строку, сопоставленную терминальному шаблону предложения с номером n, а %m — символ, возвращающий строку, соответствующую сущности, обозначенной понятием с номером m.

Текст ситуационной части 'dif{sin(5*cos(x))–x*x}', включающий в себя выражение, которое необходимо продифференцировать, будет переведен и представлен в виде следующего результата

'(cos(5*cos(x))*(5*–sin(x)*(1)+0*cos(x)*(1))–(x*1+1*x)'.

В случае необходимости, можно определить именованный императив equ, который выполнит очевидные тождественные преобразования выражений, получаемых после дифференцирования, например, путем задания ситуации 'equ{dif{sin(5*cos(x))–x*x}}'.♦

Реализация перевода атрибутными грамматиками сопряжена со значительными трудностями описания контекстно-зависимых условий или смысла конструкций языков программирования. Эти трудности связаны как с самим формализмом, так и с некоторыми технологическими проблемами [23]. К трудностям первого рода можно отнести неупорядоченность процесса вычисления выходных атрибутов при жесткой упорядоченности синтаксического анализа входного текста. Это несоответствие требует искусственных приемов для их сочетания. Технологические трудности определяются низкой эффективностью трансляторов, сгенерированных с помощью атрибутных систем, что связано с большим расходом памяти и наличием искусственных приемов итерационного вычисления атрибутов.

Пример 11. Рассмотрим пример реализации в контекстной технологии трансляции текстового представления числа в формате с фиксированной запятой в его двоичный эквивалент. Семантика такой задачи традиционно описывается атрибутной грамматикой.

Для выражения числа с фиксированной запятой Fixed введем два дополнительных понятия: Integer (целая часть) и Fraction (дробная часть). Понятию Integer припишем атрибут Int, равный значению целой части числа, а понятию Fraction — атрибут Frac, равный дробной части. Атрибуты сопоставим объектам-результатам предложений, выражающим соответствующие понятия:

```
(Arithmetic) Integer
      ″[0–9]″ { #1 }
      ″[0–9]″ Integer { %1 * 10 + #1 }
(Arithmetic) Fraction
      ″[0–9]″ { #1 }
      Fraction ″[0–9]″ { #1 + %1 / 10 }
(Integer) Fixed
      Integer '.' Fraction { %1 + %2 / 10 },
```

где использовано не определенное в примере понятие Arithmetic. При декларации Arithmetic, очевидно, должны быть определены арифметические операции, которые использованы в императивах.

По терминологии атрибутных грамматик атрибут Int является синтетическим, в то время как атрибут Frac — наследуемым. В отличие от атрибутных грамматик, не имеющих средств задания порядка вычисления атрибутов, последовательность вычисления объектов в примере определена выразительными средствами самой модели. Для правильного вычисления синтезируемого атрибута понятие Integer определено как подлежащее грамматическому разбору слева направо. В свою очередь, для вычисления наследуемого атрибута понятие Fraction подвергается разбору справа налево.♦

Заметим, что в понятийной модели с каждым понятием изначально ассоциируется атрибут, соответствующий обозначаемому объекту-результату. При необходимости, эти объекты можно описать как составные, т. е. содержащие требуемое число атрибутов.

ЗАКЛЮЧЕНИЕ

Трудности, связанные с созданием и сопровождением современных информационных систем, в значительной мере объясняются семантическим разрывом, возникающим между содержательными представлениями о предметной области и решаемыми задачами и средствами языков программирования, служащими для решения этих задач. Для сокращения семантического разрыва предлагается применение контекстной технологии программирования, основанной на понятийном анализе предметной области и контекстной интерпретации лексем.

В контекстной технологии понятийный анализ используется для выявления понятийной структуры предметной области, отражаемой в понятиях создаваемого специализированного языка. Контекстная интерпретация, в свою очередь, позволяет естественным образом реализовать контекстные условия для специализированного языка и улучшить его выразительные возможности.

При использовании специализированного языка, создаваемого по контекстной технологии, ожидается получение более простых и надежных программ. Для оценки реальных улучшений и определения предпочтительных областей применения целесообразны экспериментальные исследования.

В заключение укажем на некоторые публикации, где приведены не вошедшие в настоящую статью описания особенностей контекстной технологии программирования. Так, в докладе [24] описан вычислительный механизм контекстной технологии и рассмотрен пример более сложной предметной области. В статье [25] содержится описание принципов контекстного программирования с их привязкой к вычислительной модели. Работа [26] посвящена вопросам создания, хранения и использования знаний, представленных в виде откомпилированных понятийных моделей, а также содержит описание адресных структур вычислительного механизма контекстной технологии.

ЛИТЕРАТУРА

1. *Марка Д. А., Мак-Гоуэн К.* Методология структурного анализа и проектирования. — М.: Мета Технология, 1993.
2. *Черемных С. В., Семенов И. О., Ручкин В. С.* Структурный анализ систем: IDEF-технологии. — М.: Финансы и статистика, 2001.
3. *Калашян А. Н., Калянов Г. Н.* Структурные модели бизнеса: DFD-технологии. — М.: Финансы и статистика, 2003.
4. *Конноли Т., Бегг К.* Базы данных: проектирование, реализация и сопровождение. Теория и практика. — М.: Вильямс, 2003.
5. *Meyer B.* Object Technology: The Conceptual Perspective // Computer. — 1996. — N 1. — P. 86—88.
6. *Буч Г.* Объектно-ориентированное проектирование с примерами применения. — М.: Конкорд, 1992.
7. *Рамбо Дж., Буч Г., Якобсон А.* UML. Специальный справочник. — СПб.: Питер, 2002.
8. *Александреску А.* Современное проектирование на C++: Обобщенное программирование и прикладные шаблоны проектирования. — М.: Вильямс, 2004.
9. *Выхованец В. С.* Контекстная технология программирования // Труды IV Междунар. науч.-техн. конф. по телекоммуникациям (Телеком-99). — Одесса, 1999. — С. 116—119.
10. *Хантер Р.* Основные концепции компиляторов. — М.: Вильямс, 2002.
11. *Горский Д. П.* Вопросы абстракции и образования понятий. — М.: Изд-во АН СССР, 1961.
12. *Рейуорд-Смит В.* Теория формальных языков. Вводный курс. — М.: Радио и связь, 1988.
13. *XML* Schema Part 2: Datatypes. W3C Recommendation (http://www.w3.org).
14. *Выхованец В. С.* Язык контекстного программирования // Тез. докл. 8-й междунар. конференции "Проблемы управления безопасностью сложных систем". — М., 2000. — Т. 2. — С. 89—91.
15. *Макетирование*, проектирование и реализация диалоговых информационных систем / Под ред. Е. И. Ломако. — М.: Финансы и статистика, 1993.
16. *Гаскаров Д. В.* Интеллектуальные информационные системы. — М.: Высш. шк., 2003.
17. *Братчиков И. Л.* Синтаксис языков программирования. — М.: Наука, 1975и.
18. *Першиков В. И., Савинков В. М.* Толковый словарь по информатике. — М.: Финансы и статистика, 1991.
19. *Кнут Д.* Семантика контекстно-свободных языков // Семантика языков программирования. — М.; 1980.
20. *Шагурин И. И., Бердышев Е. М.* Процессоры семейства Intel P6: Архитектура, программирование, интерфейс: Спр. М.: Горячая линия — Телеком, 2000.
21. *Пратт Т., Зелковиц М.* Языки программирования: разработка и реализация. — СПб.: Питер, 2002.
22. *Ахо А., Ульман Д.* Теория синтаксического анализа, перевода и компиляции. — М.: Мир, 1978.
23. *Бездушний А. Н., Лютый В. Г., Серебряков В. А.* Разработка компиляторов в системе СУПЕР. — М.: ВЦ АН СССР, 1991.
24. *Иосенкин В. Я., Выхованец В. С.* Контекстная модель технологического процесса предприятия // Труды II междунар. конф. "Идентификация систем и задачи управления" SICPRO'03. — М., 2003. — С. 859—871.ц
25. *Иосенкин В. Я.* Контекстно-ориентированное программирование // Искусственный интеллект. — 2004. — № 3. — С. 667—677.
26. *Выхованец В. С., Иосенкин В. Я.* Компиляция знаний, представленных на языке ESSE // Тез. докл. 2-й Междунар. конф. по проблемам управления. — М., 2003. — Т. 2. — С. 165.

☎ *(095) 787-88-44*

E-mail: vyk@ipu.ru ❑