Мартыненко Борис Константинович

УЧЕБНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ ПРОЕКТ РЕАЛИЗАЦИИ АЛГОРИТМИЧЕСКИХ ЯЗЫКОВ

Аннотация

Предлагается тема проекта для студентов 3–5 курсов университетов, специализирующихся в области информационных технологий, связанная с исследованием схемы реализации алгоритмических языков на базе объектно-ориентированного описания семантики языка и метода синтаксически управляемого перевода входной программы в промежуточный объектно-ориентированный код. Семантика программы реализуется за счёт использования одной полиморфной функции, вызываемой рекурсивно для исполнения конструкций программы в динамической последовательности, зависящей от значений данных. Этот полиморфизм учитывает как бесконтекстную синтаксическую структуру программы, так и её контекст, связанный видами или типами конструкций, её составляющих. Такая объектно-ориентированная спецификация семантики языка программирования приводит к функционально-ориентированной структуре выходной программы, которая может реализовываться на базе функциональных систем программирования.

Цель работы: изучить метод описания алгоритмических языков по А. ван Вейнгаардену и исследовать предлагаемую схему их реализации с использованием современных средств синтаксического анализа и систем объектно-ориентированного программирования.

Ключевые слова: алгоритмические языки, анализаторы, грамматики, языки объектно-ориентированного программирования.

1. ВВЕДЕНИЕ

В свое время (в конце 1960-х г.) публикация сообщения об алгоритмическом языке Алгол 68 [1] была воспринята сообществом программистов как существенный шаг в развитии методов формализованного описания языков программирования. Метод основан на двухуровневой формальной грамматике А. ван Вейнгаардена и описании операционной семантики на естественном языке, но с использованием тщательно подобранных терминов и точно определённых понятий.

В свете этой публикации интересно посмотреть, в какой мере эта модель описания алгоритмических языков соответствует модели их реализации на базе объектно-синтаксической парадигмы программирования с использованием современных синтаксических анализаторов, в частности, испытать схему сборки программ с использованием промежуточного кода в виде дерева экземпляров объектов-конструкций, составляющих программу. Это дерево представляет семантический образ входной программы, созданный по её синтаксической структуре благодаря методам (в смысле ООП), описывающим исполнение программных конструкций.

Семантическое дерево конструкций, узлы которого представлены объектами, реализующими исполнение конструкций, составляющих исходный текст, строится на стадии синтаксического анализа и имеет структуру, в первом приближении подобную дереву вывода в правилах КС-грамматики (без учёта видов конструкций). Учёт видов конструкций не влияет на его структуру. От видов зависят лишь методы, реализующие исполнение конструкций, составляющих это дерево.

При таком подходе семантика программы реализуется за счёт использования одной полиморфной функции, вызываемой рекурсивно для исполнения конструкций программы в динамической последовательности, зависящей от значений данных. Этот полиморфизм учитывает как бесконтекстную синтаксическую структуру программы, так и её контекст, связанный видами или типами данных, а управляющая структура преобразованной программы становится функционально ориентированной.

Действительно, программа, представленная в виде семантического дерева конструкций, интерпретируется множеством одноимённых функций, рекурсивно вызываемых друг другом, начиная со стартового вызова вида program^.Ran. Таким образом, макроструктура этих вызовов подобна суперпозиции функций, то есть является функционально ориентированной.

Несколько слов, поясняющих название данной статьи.

• Проект «учебный», потому что в процессе его выполнения придётся изучить обширный материал по методам описания алгоритмических языков и языков и систем ООП и овладеть соответствующими навыками программирования.

• Проект «исследовательский», потому что предлагается тема для исследования, а не законченный проект для исполнения. Его цель – исследовать именно способ синтаксически-управляемой сборки программ из объектов и оценить его в разных аспектах, а не реализацию конкретного языка с помощью конкретных языка и системы ООП.

• Проект реализации «алгоритмических языков», а не «языков программирования», потому что язык становится языком программирования только после его реализации.

И, наконец, то, что касается терминологии, используемой в статье. Поскольку используется источник [1] как образец описания алгоритмических языков, то используются понятия и термины этого источника. Применение Паскаля в качестве промежуточного инструментального языка ООП для иллюстрации подхода вынуждает использовать соответствующую (иногда альтернативную) терминологию.

2. ОБЩАЯ СХЕМА СБОРКИ И ИСПОЛНЕНИЯ ПРОГРАММ

Роль синтаксического анализатора состоит в том, чтобы по спецификации трансляции скомпилировать языковый процессор – *конвертер*, который любую конкретную программу на реализуемом алгоритмическом языке отображает в текст программы построения промежуточного кода (дерево конструкций программы) на инструментальном ООП-языке *I*.

Промежуточная программа на инструментальном языке определяет процесс построения дерева объектов-конструкций, составляющих данную конкретную программу пользователя.

Заметим, что в описываемой здесь схеме реализации языков любая данная программа имеет несколько семантически эквивалентных представлений:

1) программа на языке пользователя *L* на входе системы сборки (см. рис. 1);

2) программа на промежуточном инструментальном ООП-языке *I* на выходе конвертера;

3) программа на выходе компилятора промежуточного языка – exe-код той платформы, на которой она будет исполняться, заканчивающаяся вызовом метода, реализующего исполнение заглавного объекта-конструкции «программа». Этот вызов воспроизводит рекурсивный обход дерева конструкций программы, при котором вызываются методы, реализующие исполнение тех подконструкций, которые участвуют в исполнении данной программы при заданных исходных данных.

Компилятор инструментального языка служит для получения исполнимого кода, семантически адекватного данной входной программе на языке пользователя.

Такая объектно-синтаксическая архитектура программы в виде семантического дерева конструкций, её составляющих, имеет ряд преимуществ:

1. Представление программы в виде семантического дерева не зависит от платформы, на которой оно интерпретируется. Требуется лишь наличие реализации объектно-синтаксической модели семантики входного языка в виде скомпилированного модуля родовых конструкций реализуемого языка.

2. В отличие от исходного текста программы семантическое дерево уже готово к интерпретации за счёт методов объектов-конструкций.

3. Семантическое дерево синтаксически непосредственно связано с исходным текстом программы. Это даёт возможность диагностики динамических ошибок в терминах входной программы, а благодаря интерпретации, – и в терминах данных во время её исполнения.

Подобная схема сборки программ может пригодиться для тестирования полноты описания алгоритмического языка или помочь избежать неудачных решений на стадии его проектирования.

Исследование подобной схемы реализации алгоритмических языков на базе объектно-ориентированного описания семантики

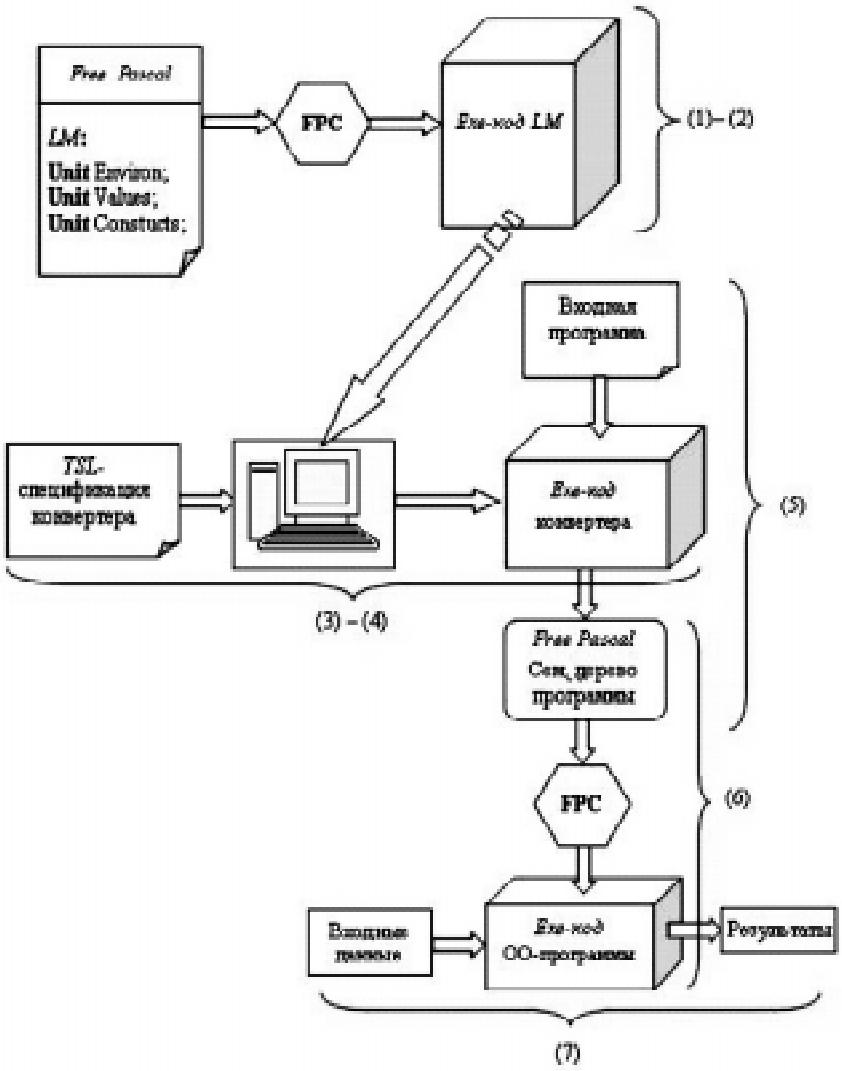

Рис. 1. Схема синтаксически управляемой ОО-сборки программ

языка и метода синтаксически управляемого перевода входной программы в промежуточный объектно-ориентированный код могло бы составить тему учебного исследовательского проекта для студентов 3–5 курсов университетов, специализирующихся в области информационных технологий.

3. ОБЪЕКТНО-СИНТАКСИЧЕСКАЯ СБОРКА ПРОГРАММ

Проиллюстрируем этот подход на примере использования технологического комплекса SYNTAX [2] для получения конвертера, осуществляющего перевод программы из входного языка пользователя *L* типа Алгол 68 в ООП-язык FP (Free Pascal) [3] в качестве промежуточного кода, и компилятора FPC для получения исполнимой программы, то есть .exe-кода, семантически эквивалентного исходной программе.

TK SYNTAX был разработан на опыте решения задачи синтаксического анализа для Алгола 68. Он включает язык для спецификации трансляций. Спецификация трансляции состоит из КС-грамматики и описания операционной среды. Правые части правил КС-грамматики представлены регулярными выражениями относительно нетерминалов и терминалов, а также контекстных символов – семантик и резольверов. Семантики ассоциируются с процедурами без параметров, изменяющими состояние операционной среды, а резольверы – с булевскими функциями (тоже без параметров), тестирующими эти состояния. Эти процедуры и функции вместе с необходимым для них окружением и составляют описание операционной среды. Именно они обеспечивают контекстную чувствительность механизма трансляции, результат которой фиксируется как финальное состояние операционной среды.

По заданной спецификации строится управляющая таблица, которая используется штатным процессором TK SYNTAX для осуществления трансляции.

Построение конвертера ставится как задача реализации трансляции из текста на входном языке пользователя в программу построения дерева конструкций на инструментальном языке ООП.

В 2006 г. А.С. Лукичёв защитил диссертицию на тему «Реализация атрибутных грамматик в технологии «SYNTAX»» [4], в которой он применил механизм аффиксов Костера (C.H.A. Koster) для передачи контекста в трансляционных грамматиках системы SYNTAX. В результате появилась возможность параметризации контекстных процедур и булевских функций, что придало системе большую надежность, а пользователю предоставило больше удобств и дополнительной информации при проектировании средств синтаксической обработки данных.

Для того чтобы реализовать сборку исполнимой программы из её текста на входном языке *L*, необходимо:

1) разработать и описать на промежуточном языке *I* (FP) семантическую модель языка *L*, включающую модули, представляющие:

– значения любых видов входного языка в форме объектов-значений, инкапсулирующих все их свойства,

– конструкции входного языка в форме объектов-конструкций, методы которых реализуют их исполнение,

– организацию окружения (пространства значений) периода исполнения программ;

2) откомпилировать это описание семантических модулей с помощью компилятора (FPC) промежуточного языка *L* (FP);

3) написать TSL-спецификацию[1] трансляции любой входной программы на языке в дерево объектов-конструций на промежуточном языке *I* (FP);

4) с помощью TK SYNTAX получить действующий конвертер $L \rightarrow I$ для преобразования любой программы на входном языке *L* в программу построения семантического дерева на промежуточном ООП-языке *I* (FP).

Затем с помощью полученного конвертера можно:

5) текст любой входной программы *L* на языке преобразовать в программу построения дерева объектов-конструкций на про-

[1] TSL – входной язык TK SYNTAX для совместного описания синтаксиса и семантики языка.

межуточном языке *I* (FP), семантически эквивалентного этой входной программе;

6) наконец, с помощью компилятора FPC промежуточного языка *I* получить семантическое дерево в формате .exe-кода, который можно

7) многократно исполнять с различными входными данными.

В этом процессе (см. рис. 1) этапы 1 и 3 – ручная работа, выполняемая однократно. Остальные этапы выполняются автоматически: этапы 2 и 4 – компилятором FPC и TK SYNTAX соответственно, и тоже один раз; этап 5 выполняется конвертером $L \to I$ один раз для каждой входной программы; этап 6 – один раз для каждой входной программы; этап 7 – один раз для каждого набора входных данных.

Далее описывается «инфраструктура» программы, обеспечивающая навигацию между конструкциями входного языка по управлению и данным.

4. ОБЪЕКТНО-ОРИЕНТИРОВАННАЯ МОДЕЛЬ ПРОГРАММЫ

Согласно [1], *программа* – заглавная конструкция языка, состоящая из других конструкций (её *подконструкций*).

Состав конструкций любой конкретной программы определяется в процессе синтаксического анализа её текста. С учётом конкретного конструкционного состава данной программы по общему описанию семантики языка определяется, из каких действий состоит её исполнение.

Эти действия выстраиваются, следуя синтаксической структуре текста программы, а исполняются в порядке, определяемом семантикой конструкций. Исполнение конструкции даёт результат – *значение*.

Поскольку структура текста программы в общем случае рекурсивна, то и структура действий по её исполнению имеет рекурсивный характер.

Правило семантики, описывающее исполнение конструкции, в общем случае ссылается на правила, описывающие исполнение (*предысполнение*) других конструкций (*подконструкций*), из которых она состоит, и действий, относящихся к исполнению непосредственно данной конструкции. При этом исполнение каждой подконструкции даёт значение (*подзначение*), и эти подзначения используются для получения результата (*значения*) самой конструкции.

Существуют конструкции *элементарные*, результат которых определяется непосредственно, без ссылки на правила исполнения других конструкций. Таким образом, все действия, определяемые семантикой языка, по исполнению данной программы выстраиваются по её синтаксической структуре, определяемой грамматикой входного языка.

Синтаксис языка определяет конструкции не бесконтекстно. Каждая конструкция имеет *вид* в зависимости от контекста, в котором она находится. Ёе результат является значением соответствующего вида. Поэтому исполнение конструкции происходит по-разному, в зависимости от её вида. Естественно реализовать функциональность конструкций разных видов за счёт полиморфизма методов и инкапсуляции свойств значений в зависимости от вида.

Если в данном месте программы находится конструкция не того вида, который требуется контекстом, то над её непосредственным результатом – значением *априорного* вида – выполняются *приведения*, то есть действия, дающие другое значение *апостериорого* вида, требуемое контекстом. Приведения планируются (компилируются) во время трансляции, поскольку они определены *статически*, то есть текстуально. Собственно, исполнение приведений результатов конструкций к требуемому виду относится ко времени счёта.

Образно говоря, предназначение семантической модели языка состоит в том, чтобы воспроизвести историю возникновения, существования, преобразования и «гибели» значений при исполнении любой программы. Соответственно, придётся разработать схему их размещения в памяти компьютера и способ доступа к ним конструкций программы.

Для реализации этой модели, как было упомянуто выше, используется язык ООП, на котором описываются классы объектов для представления конструкций реализуемого языка, а другие – для представления

значений и организации механизма доступа ним через *окружения*.

Понятие текущего окружения зависит от блочной структуры программы и определяет множество значений, доступных программе в текущий момент её исполнения.

5. ОБЪЕКТНО-ОРИЕНТИРОВАННАЯ МОДЕЛЬ ПРОСТРАНСТВА ЗНАЧЕНИЙ

Окружения образуются во время исполнения блоков – конструкций, в составе которых есть описания *индикаторов* вида и операций, а также идентификаторов любого вида (именованных констант, переменных, процедур и т. д.). Эти индикаторы и идентификаторы далее обозначаются общим термином ИНДИКАТОРЫ.

В предлагаемой модели собственно значения представляются экземплярами объектов в куче, а индикаторы суть ссылки (указатели) на них. Индикаторы, описанные в одном блоке, образуют *участок* доступа к значениям, которыми они обладают в момент исполнения этого блока.

Текстуальная вложенность блоков отображается в иерархию *окружений*, определяемую отношением *«младше»*, *«старше»* или *«такая как»* над областями действия окружений. Именно, окружение $\mathtt{E}_1$, созданное данным блоком, младше окружения `E`, образованного блоком, непосредственно объемлющим данный, или, если угодно, при той же ситуации окружение `E` старше окружения $\mathtt{E}_1$.

Иначе говоря, стек, разбитый на участки, соответствующие блочной структуре программы, – синоним пространства окружений, а куча (*heap*[1]) – синоним пространства значений.

Любое значение образуется в результате исполнения конструкции `C` в соответствующем окружении `E`, то есть идентификационной среде. Пара `S = (C, E)` называется *сценой*.

Говорят, что окружение `E` необходимо для сцены `S`. Некоторые сцены тоже могут использоваться как значения (см., например, реализацию переходов и вызовов).

Для метки, например, необходимое окружение образуется блоком, в котором находится её определяющее вхождение (перед основой). Необходимое окружение для процедуры образуется минимальным блоком, содержащим определяющие вхождения индикаторов, используемых в её теле, но не описанных в нём.

Доступ к сценам внутри участков осуществляется через использующие вхождение метки или индикатора-вида при исполнении фактического описателя вида в составе генератора.

Не следует смешивать область действия окружения (с точки зрения модели – идентификационной среды) с областями действия значений, доступных внутри его участка. Область действия окружения предпочтительно используется при определении области действия сцен, для которых оно (окружение) необходимо, или области действия выдачи генераторов, для которых оно является «локализующим». Область действия окружения определяется относительно области действия некоторого другого окружения, так что создаются иерархии областей действия, в конечном счёте, зависящие от области.

В отличие от реальной схемы размещения значений, использующей стек как основной, но не единственный способ оптимальной организации использования оперативной памяти компьютера, каждое значение в разрабатываемой модели представляется экземпляром объекта соответствующего типа и располагается в куче.

В дальнейшем термин *стек* используется как моделируемое понятие[2], содержательно обозначающее вместилище индикаторов – информационных структур фиксированного

[1] Проблема «сборки мусора», то есть освобождения памяти из-под объектов-значений, с которыми программа потеряла связь, здесь не рассматривается, так как является общей для систем, использующих такую организацию памяти, как «куча».

[2] В описываемой реализации стек – коллекция участков (наследник TCollection), связанных между собой в список по отношениям *«младше»* и *«старше»*, упомянутым выше. Эта связь используется для обновления таблицы Display при смене окружения во время исполнении конструкции вызова.

формата, используемых для доступа к значениям в куче, а не собственно значений, над которыми выполняются действия.

С одной стороны, это согласуется со свойствами инструментального объектно-ориентированного языка моделирования, в частности, с механизмом реализации полиморфизма значений разных видов через виртуальные методы, а с другой – диктуется особенностью семантики реализуемого языка, связанной с понятиями *сцен* и *областями действия значений и окружений*.

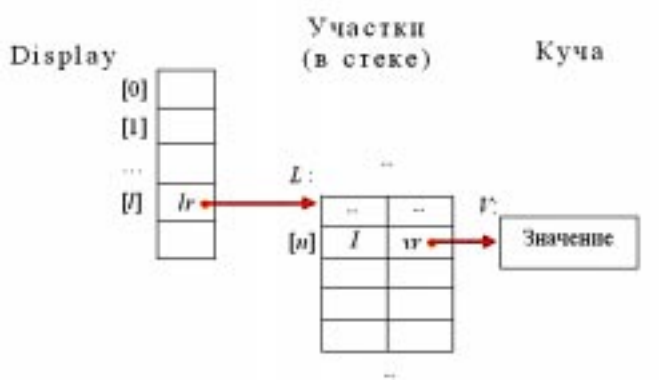

Рис. 2. Схема доступа программы к значениям

Заметим, что, помимо явно поименованных ИНДИКАТОРАМИ величин входной программы, при её исполнении возникают *анонимные* значения, как, например, при вычислении операндов выражений, которые тоже должны размещаться в текущем окружении и быть доступными операциям над ними. Естественно размещать такие анонимные значения на участке текущего блока вслед за именованными значениями.

Таким образом, участок каждого блока состоит в общем случае из двух частей: *статической части*, в которой статически размещаются именованные значения, и *динамической части*, в которой размещаются анонимные величины, динамически сменяющие друг друга значения изменяющейся структуры. Ссылки на индикаторы анонимных величин используются тоже анонимно, то есть относительно текущей вершины стека данных. При этом, как только использование анонимного значения завершается, его ИНДИКАТОР сразу же удаляется из динамической части участка вместе со значением, которым он обладает.

Структура же статической части участка остаётся неизменной до выхода из текущего блока. Несмотря на эти различия, способ доступа конструкций к именованным и анонимным значениям одинаков – через *статический адрес* ИНДИКАТОРА в стеке данных, то есть (l, n), где l – уровень блока, создавшего участок, а n – номер ИНДИКАТОРА на этом участке.

Для того, чтобы отслеживать, какие участки составляют текущее окружение, используется таблица Display (см. рис. 2) в описываемой здесь реализации, наследник TCollection. Её элементами являются указатели на участки в стеке данных, к значениям которых программа имеет доступ в текущий момент исполнения. На рис. 2 *lr* и *vr* – типизированные указатели типа PLocale и PValue соответственно.

Максимальное число элементов этой таблицы выясняется на этапе синтаксического анализа текста конкретной входной программы, выполняемого конвертером. Оно равно максимальной глубине вложенности блоков программы.

Таблица Display создаётся промежуточной программой перед развёртыванием семантического дерева конструкций и относится к административной системе вместе со стеком и списком программных точек (см. далее п. 7).

Общая схема доступа конструкции «программа» к её данным, то есть значениям в первом приближении такова.

Пусть некоторый экземпляр объекта-значения V в куче доступен некоторой конструкции программы через ИНДИКАТОР i, расположенный на участке L, образованном блоком уровня l.

Методы объектов-конструкций ссылаются на значения в стеке данных через пару номеров (l, n), где l – уровень блока, а n – номер ИНДИКАТОРА в данном блоке. Например, указатель на объект-значени V в схематическом изображении мо-

жет быть получен таким образом: Display^.At(*l*)^.At(*n*)^.GetValue.

Эта пара номеров (*l*, *n*) называется *статическим адресом* ИНДИКАТОРА и представляется объектом типа TAddr (см. листинг 1).

Фактически объект типа TAddr имеет методы для доступа к индикаторам в целом (GetCouple), к каждой из двух компонент в отдельности: GetIndic – внешнее представление индикатора, GetValue – указатель на значение, которым он обладает, а также метод PutValue (v : PValue) для замены значения, которым обладает индикатор, на другое значение. Этот метод используется для выполнения собственно присваиваний.

6. РЕАЛИЗАЦИЯ ВЫЗОВОВ

Не будь процедур, участки доступа резервировались и удалялись бы из памяти компьютера в магазинном режиме, следуя вложенности блоков исполняемой программы, и текущее окружение представлялось бы всеми участками, находящимися в памяти в данный момент. Однако вызовы процедур временно могут изменить текущее окружение на период исполнения тел вызываемых процедур.

Действительно, в месте вызова процедуры программа имеет доступ к данным текущего блока и всех блоков, его объемлющих. В этом окружении исполняются фактические параметры вызова. Затем исполняется тело процедуры с уже вычисленными значениями параметров. Тело вызываемой процедуры имеет доступ к своим локальным данным и данным блоков, текстуально объемлющих его. Так что в общем случае окружение тела процедуры может отличаться от окружения в месте вызова: значения не всех участков, имеющихся в стеке, будут доступны программе во время исполнения тела процедуры. После завершения исполнения тела вызванной процедуры текущее окружение становится таким же, как перед вызовом. Для восстановления окружения, актуального в месте вызова, служит полное динамического описание точки возврата, представленной как значение сцены в стеке данных.

Листинг 1

```
TAddr = object (TObject)
RLevelNmb, { Блочный уровень блока, к которому относится данный адрес. }
ItemNmb { Номер ИНДИКАТОРА на участке уровня RLevelNmb }: integer;
constructor Init (l, n : integer);
{ Инициализирует:
(a) номер блочного уровеня RLevelNmb по параметру l и
(b) номер ИНДИКАТОРА на участке по параметру n. }
destructor Done; virtual;
function Show : PChar; virtual;
{ Готовит строку, представляющую данный статический адрес. }
function GetCouple : PCouple;
{ Выдаёт указатель на ИНДИКАТОР в стеке данных.}
function GetIndic : string;
{ Выдаёт внешнее представление ИНДИКАТОРА.}
function GetValue: PValue; virtual;
{ Выдаёт указатель на значение, которым обладает ИНДИКАТОР.}
procedure PutValue (v : PValue); virtual;
{ Указатель v вставляется в стек данных на место, задаваемое данным
 статическим адресом, не меняя внешего представления индикатора назначения.
function GetScope : integer; virtual;
{ Даёт область действия значения, размещённого по данному адресу. }
end;
```

Как известно, процедурные значения образуются в результате исполнения текстов процедур. Они бывают с параметрами и без параметров. Используются процедурные значения в вызовах и формулах. В последнем случае процедура имеет один или два параметра. Семантика формулы аналогична семантике вызова. Поэтому достаточно рассмотреть реализацию вызова.

Общая схема ситуации во входной программе может быть представлена следующим образом (см. листинг 2).

Результат текста процедуры – процедурное значение (сцена), представленное парой указателей **(C, E)**, где **C** – адрес точки входа в машинный код, реализующий действия тела процедуры, а **E** – ссылка на стек, представляющая окружение, необходимое для тела процедуры, – адрес секции данных минимального блока, объемлющего текст процедуры, о котором шла речь ранее. Это процедурное значение доступно конструкции вызова через индикатор процедуры на участке блока, содержащего текст процедуры.

Предполагается, что P обозначает конструкцию, доставляющую упомянутое выше процедурное значение, а Y_1, ..., Y_n – конструкции, играющие роль фактических параметров рассматриваемого вызова.

Фактические параметры исполняются в окружении, образуемом вызывающим блоком, а их значения используются телом процедуры в его собственном окружении (взамен формальных параметров x_1, ..., x_n). Очевидно, процедура **(C, E)** может вызываться лишь в таком окружении, которое не младше **E**.

Выражаясь в терминах реализации, секции данных, образующие окружение **E**, должны быть в одной статической цепочке с секцией данных вызывающего блока.

В кратком изложении семантика вызова такова:

1) исполнить конструкцию, доставляющую процедуру **(C, E)**, которая вызывается;

2) исполнить конструкции, используемые в качестве фактических параметров вызова;

3) выполнить тело процедуры на месте вызова, используя значения фактических параметров взамен формальных.

7. СМЕНА ОКРУЖЕНИЯ

Заметим, что состояние таблицы Display в текущий момент исполнения программы представляет текущее окружение.

В начальный момент исполнения программы текущее окружение E = ∅, то есть число элементов в таблице Display равно нулю.

Текущее окружение изменяется в следующих случаях.

I. *Вход в блок.* Пусть текущее окружение E перед входом в блок характеризуется состоянием Display = (lr_0, lr_1, ..., lr_n), где lr_i – указатели на участки L_i (i = 0, 1, 2, ..., n) в стеке данных. Здесь n – номер текущего участка окружения. Блоком создается новый участок L, который присоединяется к текущему окружению E, образуя новое ок-

Листинг 2

```
(... ( { начало минимального блока, объемлющего текст процедуры, содержащий определяющее вхождение индикатора, используемого в теле процедуры, но не описанного в нём }
...{ текст процедуры }
(M1 x1, ..., Mn xn)M :
  ({ тело процедуры, использующее формальные параметры x1, ..., xn}); ...
({ начало вызывающего блока } ...
P(Y1, ..., Yn) {вызыв, использующий фактические параметры Y1, ..., Yn }; ...
){ конец вызывающего блока } ...
    ) { конец минимального блока } ...

)
```

ружение $E' = E \cup L$, характеризуемое состоянием Display = (lr_0, lr_1, ..., lr_n, lr), где lr – указатель на участок L. Ясно, что в этот момент L – верхний участок стека.

II. *Выход из блока.* Пусть текущее окружение E в этот момент характеризуется состоянием Display = (lr_0, lr_1, ..., lr_{n-1}, lr_n), где lr_i – указатели на участки L_i ($i = 0, 1, 2, ..., n$) в стеке данных. Из текущего окружения E удаляется верхний участок. Новое окружение $E' = E \setminus L$ характеризуется состоянием Display = Display = (lr_0, lr_1, ..., lr_{n-1}).

III. *Переход по метке.* Любая метка l представляется сценой, то есть объектом **(C, E)**, где **C** – указатель на объект-конструкцию (основу), помеченную этой меткой, а **E** – указатель на необходимое окружение этой сцены. Эта сцена как значение доступна конструкциям перехода с помощью ИНДИКАТОРА, представляющего идентификатор метки, расположенной на участке, образованном блоком, непосредственно содержащим помеченную конструкцию. Конструкция перехода исполняется по следующим шагам:

1. Если метка выводит из текущего блока, то участки блоков промежуточных уровней исключаются из текущего окружения вместе с доступными значениями. Соответственно корректируется счётчик элементов таблицы Display на разницу в уровнях.

2. Затем запускается исполнение помеченной конструкции **C**.

IV. *Вызов процедуры.* Процедурное значение (routine) образуется в результате исполнения текста процедуры и представляется в форме значения-сцены (C_p, E_p), где C_p – указатель на тело процедуры, а E_p – указатель на необходимое окружение для него или, другими словами, указатель на начало статической цепочки участков, образующих необходимое окружение для тела процедуры (см. рис. 3). Конструкция вызова получает доступ к этому значению сцены, как обычно, через ИНДИКАТОР, расположенный на участке вызывающего блока L.

Конструкция вызова как объект имеет три поля данных, которые используются в указанном порядке.

1. Значение сцены (C_r, E_r), представляющее точку возврата после исполнения тела процедуры, где C_r – конструкция, следующая за вызовом, а E_r – текущее окружение, представленное указателем на участок L текущего блока в момент перед вызовом.

• Сцена точки возврата анонимно выкладывается на вершину стека.

2. Коллекция конструкций фактических параметров.

• Каждая конструкция-фактический параметр исполняется, и полученное значение оставляется на текущей вершине стека. Таким образом, все значения фактических параметров вызова оказываются в верхней области стека в порядке их исполнения.

3. Значение сцены (C_p, E_p), представляющей вызываемую процедуру.

• Устанавливается окружение E_p, и управление передаётся телу вызываемой процедуры, то есть исполняется основа (блок) C_p.

При исполнении тела вызываемой процедуры образуется участок L'_p над окружением вызывающего блока. Он прицепляется к необходимому окружению тела вызываемой процедуры: $E'_p = L'_p \cup E'_p$. Так образованное окружение E'_p становится текущим.

Заметим, что вновь образованный участок тела процедуры L'_p, в отличие от участка обычного блока, включает так называемые формальные ИНДИКАТОРЫ, предназначенные для доступа к соответствующим значениям фактических параметров вызова. Прежде чем исполняются конструкции тела процедуры, происходит перенос значений параметров из динамической части участка **L** вызывающего блока в формальные ячейки участка тела вызываемой процедуры, расположенные на участке L'_p. Причём, значения фактических параметров достаются анонимно через индикаторы в конце участка вызывающего блока **L**, а выкладываются на место формальных индикаторов именованным образом через номера элементов коллекции, представляющей участок тела процедуры.

V. *Возврат из тела процедуры по завершении вызова.* Во время исполнения тела процедуры доступны данные через ИНДИКАТОРЫ на его участке уровня $i + 1$, вклю-

чая формальные ячейки, и на участках всех блоков, образующих необходимое окружение для тела процедуры до уровня 1. Эти участки, связанные между собой посредством поля Prev в статическую цепочку (см. рис. 3), и образуют текущее окружение в этот момент. Тело процедуры, как всякая основа, даёт результат, размещаемый в универсальной переменной UValue типа PValue административной системы.

Исполнение тела процедуры завершается вызовом метода Return, который использует значение сцены с вершины стека для восстановления окружения в точке вызова и продолжения исполнения конструкций, следующих за вызовом.

После возврата из тела процедуры, когда на вершине стека находится участок вызывающего блока с уже удалёнными фактическими параметрами и точкой возврата, результат вызова из UV помещается анонимно на новую вершину стека (см. рис. 4).

8. ПЕРЕДАЧА УПРАВЛЕНИЯ МЕЖДУ КОНСТРУКЦИЯМИ ПРОГРАММЫ

Как указано во введении, промежуточная программа на инструментальном языке

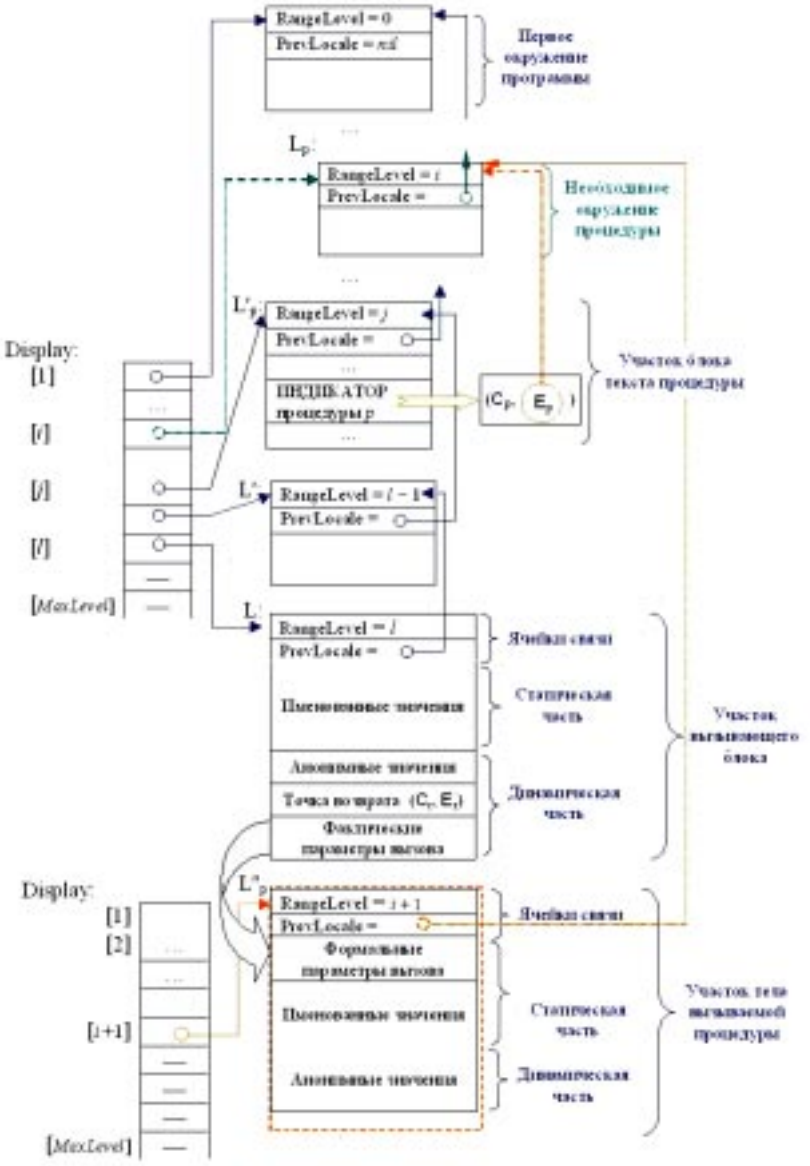

Рис. 3. Состояние стека данных после входа в тело процедуры

определяет процесс построения дерева объектов-конструкций, составляющих данную конкретную программу пользователя. Семантический образ входной программы представляется с помощью коллекций, элементами которых являются указатели на объекты-конструкции, составляющие данную программу.

При исполнении переходов или вызовов используются значения сцен, связанные с передачей управления между конструкциями.

Для образования значений сцен при входе в блок, непосредственно содержащем помеченные основы или тексты процедур, используется список программных точек, входов в эти конструкции. Элементами этого списка являются указатели на конструкции, о которых идёт речь.

Список программных точек создаётся при развёртывании семантического дерева опять же в виде коллекции и входит в состав административной системы, наряду с таблицей Display и стеком. Доступ к элементам списка программных точек при создании значений соответствующих сцен производится по уникальному номеру программной точки, определяемому во время синтаксического анализа входной программы конвертером.

При входе в блок на участке, образующемся в этот момент, помещаются указатели на конструкции, о которых идёт речь, в паре с собственным указателем на этот уча-

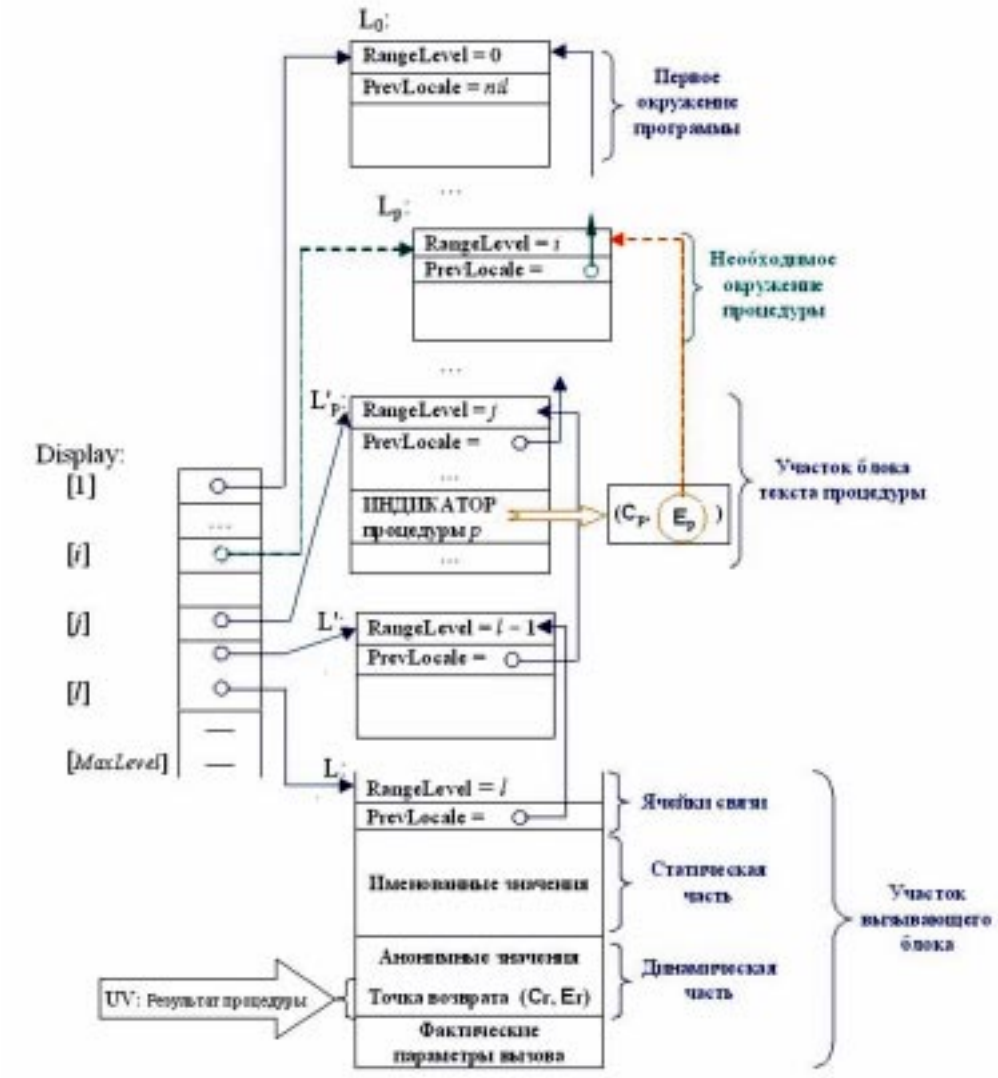

Рис. 4. Состояние стека данных после завершения тела процедуры

сток. Так образованная пара из двух указателей, один из которых позволяет передать управление конструкции, а с помощью другого установить необходимое для неё окружение, и есть конкретное представление значения сцены в описываемой здесь модели реализации семантики языка.

9. ПРИМЕР

Далее следует программа построения семантического дерева программы на Алголе 68

```
(int m := (int i := 123, j := 4, k ; i := j ;
loc int := j))
```

и её самоинтерпретации (см. листинг 3).

Протокол исполнения входной программы представен в листинге 4.

10. ОСНОВНЫЕ ЗАДАЧИ ПРОЕКТА И ПОРЯДОК ИХ ВЫПОЛНЕНИЯ

1. Описать модельный алгоритмический язык для испытания описанной выше схемы его реализации.

• Описать семантику этого языка в виде описания конструкций, значений и организации окружений на выбранном языке ООП.

2. Написать грамматику модельного языка с учётом метода его анализа, совмещённого с процессом построения семантического дерева.

• Провести автономное тестирование конвертера.

3. Провести тестирование всей схемы реализации модельного языка на входных текстах.

11. ЗАКЛЮЧЕНИЕ

Подведём краткие итоги изложенному.

1. Метод описания языков, предложенный А. ван Вейнгаарденом, может рассматриваться в качестве модели их реализации в современных системах объектно-ориентированного программирования.

2. Семантика программы представляется как дерево конструкций в форме объектов, инкапсулирующих все их свойства с учётом видов и методов их исполнения. При этом синтаксис языка может быть задан отдельно от семантики, лишь бы грамматика, его определяющая, и метод анализа давал возможность построить семантическое дерево требуемой структуры. Другими словами, при фиксированной семантике языка его синтаксис может альтернативно варьироваться.

3. Программа является своим собственным интерпретатором, а также может выполнять функции автоаннотирования и трассировки.

4. Управляющая структура программы на промежуточном ОО-языке оказывается функционально ориентированной, благодаря использованию одной полиморфной функции, рекурсивно вызываемой при обходе семантического дерева программы, с учётом текущих данных. Объектно-ориентированная сборка программы даёт исполнимый код в стиле функционального программирования.

5. Множество конструкций стандартных видов поддерживается, с одной стороны, неизменным модулем конструкций стандартных видов, а с другой – дополнительным модулем конструкций видов, описанных в частной программе пользователя.

6. Пространство значений, возникающих при исполнении программы, поддерживается, с одной стороны, неизменным модулем значений стандартных видов, а с другой – дополнительным модулем значений видов, описанных в частной программе пользователя.

7. Организация самого пространства значений составляет отдельный модуль, независимый от видов или типов конструкций реализуемого языка.

8. Предполагается, что предлагаемый метод может быть использован при проектировании языков программирования на этапе их разработки и тестирования.

9. Что касается учебных целей, то выполнение предлагаемого проекта даст студентам существенный опыт применения систем объектно-ориентированного программирования, методов совместного описания синтаксиса и семантики алгоритми-

Листинг 3

```
program test;
uses Objects, Strings, CONSTRUCTS, ENVIRON, VALUES;
{ Objects, Strings - штатные модули системы Free Pascal;
CONSTRUCTS, ENVIRON, VALUES - модули системы сборки }
var ig0, ig1 : PIntegralGenerator;
ivd0, ivd1 : PVariableDeclaration;
tl0, tl1 : PTagList;
Ci, Cj : PIntegralDenotation; Ck : PIntegralGenerator;
tag i, tag j, tag k, tag m: PTag;
Assignment0, Assignment1, Assignment2 : PAssignment;
Vm, Vi, Vj : PVariable;
Range0, Range1 : PRange;
Cl0, Cl1 : PConstructList;
begin
{ Начало сборки семантического дерева входной программы }
Stack := New (PStack, Init (3));
Display := New (PDisplay, Init (2));
Ci := New (PIntegralDenotation, Init (123));
tag i := New (PTag, Init (false, 'i', Ci));
Cj := New (PIntegralDenotation, Init (4));
tag j := New (PTag, Init (false, 'j', Cj));
tag k := New (PTag, Init (false, 'k', nil));
tl1 := New (PTagList, Init (3, 0));
tl1^.Insert (tag i);
tl1^.Insert (tag j);
tl1^.Insert (tag k);
ig1 := New (PIntegralGenerator, Init);
ivd1 := New (PVariableDeclaration, Init (ig1, tl1));
Vi := New (PVariable, Init (1, 0));
Vj := New (PVariable, Init (1, 1));
Assignment1 := New (PAssignment, Init (Vi, Vj));
Assignment2 := New (PAssignment, Init (ig1, Vj));
Cl1 := New (PConstructList, Init (3, 0));
Cl1^.Insert (ivd1);
Cl1^.Insert (Assignment1);
Cl1^.Insert (Assignment2);
Range1 := New (PRange, Init (1, 10, Cl1));
tag m := New (PTag, Init (false, 'm', nil));
tl0 := New (PTagList, Init (1, 0));
tl0^.Insert (tag m);
ig0 := New (PIntegralGenerator, Init);
ivd0 := New (PVariableDeclaration, Init (ig0, tl0));
Vm := New (PVariable, Init (0, 0));
Assignment0 := New (PAssignment, Init (Vm, Range1));
Cl0:= New (PConstructList, Init (2, 0));
Cl0^.Insert (ivd0);
Cl0^.Insert (Assignment0);
Range0 := New (PRange, Init (0, 10, Cl0));
{ Конец  сборки семантического дерева входной программы }
Range0^.Run; { Запуск самоинтерпретации семантического дерева }
end.
```

Листинг 4

```
СЕМАНТИЧЕСКОЕ ДЕРЕВО ВХОДНОЙ ПРОГРАММЫ СОЗДАНО
BEGIN {0}
[0] .int m
[1] <0, 0> :=
    BEGIN {1}
    [0] .int i := 123, j:= 4, k
    [1] <1, 0> := <1, 1>
    [3] .int := <1, 1>
    END {1}
END {0}
ИСПОЛНЕНИЕ ВХОДНОЙ ПРОГРАММ НАЧИНАЕТСЯ
Текущее окружение после исполнения  int m
Display [0] ::
[0] m => nil
Assnment0.Run началась ...
Адрес Destination: <0, 0>
Destination.Run выполнена
Результат Destination: m => nil
Source:
    BEGIN {1}
    [0] .int m
    [1] <0, 0> :=
    END {1}
Range1.Run началась ...
Текущее окружение после исполнения  int i := 123, j := 4, k;
Display [0] ::
[0] m => nil
Display [1] ::
[0] i => 123
[1] j => 4
[2] k => nil
Assignment1.Run началась ...
Destination.Run
Результат Destination: i => 123
Source: <1, 1>
Assignment1.Run закончилась
Текущее окружение после исполнения i := j
Display [0] ::
[0] m => nil
Display [1] ::
[0] i => 4
[1] j => 4
[2] k => nil
Assignment2.Run началась ...
Destination.Run выполнено
Результат Destination .int => 0
Source: <1, 1>
Результат Source: j => 4
Assignment2.Run закончилась
Текущее окружение до выхода из текущего блока
Display [0] ::
[0] m => nil
Display [1] ::
[0] i => 4
[1] j => 4
```

```
[2] k => nil
[3] .int => 4
Текущее окружение после выхода из блока уровня 1
Display [0] ::
[0] m => 4
Range1.Run закончилась
Результат Source: j => 4
Assignment0.Run закончилась
Display [0]::
[0] m => 4
Текущее окружение после выхода из блока уровня 0 ПУСТО.

% Нет смысла комментировать подробнее программу сборки семантического дерева
% входной программы и протокола её исполнения ввиду очевидности обозначений.
```

ческих языков и соответствующих средств синтаксического анализа.

Представление значений и конструкций в объектно-ориентированной модели реализации алгоритмических языков, а также синтаксические проблемы генерации дерева конструкций – темы для последующих публикаций.

Литература

1. Под ред. *А. ван Вейнгаарден, Б.Майу, Дж. Пек, К. Костер и др.* Пересмотренное сообщение об Алголе 68. М., 1979. 533 с. (Edited by *A. van Wijngaarden, B.J. Mailloux, J.E. L. Peck, C.H.A. Koster at al*. Revised report on the algorithmic language Algol 68. 1975.)

2. *Мартыненко Б.К.* Синтаксически управляемая обработка данных. 2-е изд. СПб., 2004. 315 с.

3. *Michaël Van Canneyt.* Reference guide for Free Pascal. 2002. 188 p.

4. *Лукичёв А.С.* Использование атрибутов в технологии SYNTAX // Вест. С.-Петерб. ун-та. Сер. 1. Вып. 2. СПб., 2005. С. 65–73.

Abstract

The scheme for implementation of algorithmic languages is proposed, based on object-oriented specification of semantics and syntax-directed translation technique. The resulting code is generated in object-oriented style. The project is suggested as a research topic for 3rd, 4th, or 5th year IT students.

Program semantics are implemented by means of a polymorphic function being recursively called for the purpose of executing the program constructs in a dynamic sequence, depending on actual data values. The polymorphism takes into account context-free structure of the program, as well as context dependence on the modes or types of constructions. The object-oriented specification of the programming language semantics results in functional program structure that may be implemented by a functional programming system.

The main goals are to study the algorithmic language description method by A. van Wijngaarden and to investigate the scheme for language implementation by means of the present-day parsing technique and object-oriented programming systems.

Мартыненко Борис Константинович, доктор физико-математических наук, профессор кафедры информатики математико-механического факультета СПбГУ,

mbk@ctinet.ru