ИНФОРМАЦИОННЫЕ СИСТЕМЫ И ТЕХНОЛОГИИ

Научная статья

УДК 004.415, 004.434
DOI: 10.18101/2304-5728-2024-1-46-55

МЕТОДИКА АВТОМАТИЗИРОВАННОЙ ОБРАБОТКИ ИНФОРМАЦИИ С ИСПОЛЬЗОВАНИЕМ ЯЗЫКА ПРОГРАММИРОВАНИЯ ВЫСОКОГО УРОВНЯ (ЧАСТЬ 1)

© Федорченко Людмила Николаевна
кандидат технических наук, старший научный сотрудник,
Санкт-Петербургский федеральный исследовательский центр
Российской академии наук
Россия, 199178, г. Санкт-Петербург, 14 Линия В.О., 39;
Санкт-Петербургский государственный университет
Россия, 199034, г. Санкт-Петербург, Университетская наб., 7/9
lnf@iias.spb.su

Аннотация. В работе обсуждается методика автоматизированной обработки информации, в основе которой лежит регулярная модель языка и языковой процессор как основной элемент реализации трансляции. Идеи, положенные в основу данной методики, связаны с возможностью использования регулярных выражений в правых частях правил грамматики и определяют способ их представления в виде ориентированных графов. Первая часть работы посвящена описанию принципиальной схемы автоматизированной обработки информации с использованием языка программирования высокого уровня. Рассмотрена схема процесса компиляции и порядок разработки проекта реализации языка. Кратко изложено проектирование генерирующей части компилятора. Вторая часть работы посвящена разработке анализатора. Сформулированы ограничения на грамматику, представленную в форме системы ориентированных графов (синтаксической граф-схемы), гарантирующие существование детерминированного магазинного анализатора, который далее рассматривается как управляющий механизм для инициирования действий, составляющих процесс трансляции. Специфику методики составляет *алгоритм регуляризации* грамматики, основанный на эквивалентных преобразованиях грамматики входного языка. Регуляризация исходной грамматики является *частью полного цикла реализации языка,* состоящего из цикла *пользователя*, и полуавтоматического цикла *разработчика*.

Ключевые слова: автоматизированная обработка информации, схема процесса компиляции, синтаксическая модель языка, языковой процессор.

Для цитирования
Федорченко Л. Н. Методика автоматизированной обработки информации с использованием языка программирования высокого уровня (Часть 1) // Вестник Бурятского государственного университета. Математика, информатика. 2024. № 1. С. 46–55.

Введение

Каждое программное приложение управляет и взаимодействует с *данными*, которые можно интерпретировать как некие объекты («data» на латинском языке), эти данные являются элементами информации, которыми обмениваются с пользователями или внешними системами. Задача информации — влиять на суждение и поведение получателя. В отличие от данных у информации имеются смысл и назначение. Данные становятся информацией, если их создатель добавляет к ним смысл.

Автоматизированная обработка информации с использованием языка программирования предполагает, что управление процессом обработки данных задается при помощи синтаксической структуры [1–6]. В приложениях, таких как трансляторы языков, управление процессом трансляции определяется синтаксической структурой предложений входного языка [3; 5; 7]. Примером другого применения принципа синтаксического управления является непосредственное задание структуры некоторого вычисления. В этом случае управляющая структура относится не к исходным данным, а к состояниям вычислительного процесса.

1 Принципиальная схема автоматизированной обработки информации

Рассмотрим полный цикл обработки информации с использованием языка программирования высокого уровня. В качестве синтаксических конструкций используется синтаксис языков Паскаль, ADA, Алгол 68 и Си [7].

Всякий процесс обработки информации можно определить как некоторое преобразование *f исходной информации i* в *выходную информацию o* = *f*(*i*).

Этот процесс начинается на понятийном (абстрактном) уровне предметной области, к которой относится задача. Затем выбирается способ представления данных (их описание) и ее воплощение в некотором языке программирования L: $\rho(i) = D_L$ — *входные данные* и $\pi(f) = P_L$ — *программа* на языке высокого уровня. При этом необходимо, чтобы интерпретация ι программы P_L согласно *спецификации* (описанию) σ_L языка L с входными данными D_L давала бы такой результат $R_L = \iota(P_L, D_L, \sigma_L)$, которому на понятийном уровне соответствовала бы выходная информация $o' = \rho^{-1}(R_L)$, ожидаемая на понятийном уровне, т. е. такая, что $o' = o$. Здесь ρ — отображение, реализующее представление информации (*описание данных*), π — отображение преобразования f на языке программирования L (*программирование*), ρ^{-1} — интерпретация результата, представленного на понятийном уровне. Этот процесс назовем *циклом пользователя*, поскольку он включает работу, выполняемую самим пользователем.

Для автоматизированной обработки информации с помощью компьютера необходимо перейти с языкового представления данных и программы на уровень машинного кода целевой машины M_t [1; 4–6]. Разработка

средства этого перехода — компилятора, является задачей *реализатора языка L*. Он должен построить представление в машинном коде компьютера M_t любых видов тех данных, которые определены в языке L, то есть реализацию любой языковой конструкции в терминах машинных команд M_t и определить, а затем и запрограммировать в коде инструментальной машины M_i процесс перевода τ любой программы в языке L в последовательность команд целевой машины M_t. Обычно программа компилятора создается в коде целевой машины, т. е. $M_i = M_t$. Но иногда используют так называемые *кросс-трансляторы* – программы, работающие на инструментальной платформе M_i, и создающие код для другой платформы M_t, например, встроенного процессора, ($M_i \neq M_t$).

Циклы пользователя и реализатора представлены на рисунке 1. Верхняя часть схемы (рис. 1), изображающая этот процесс, — *цикл пользователя*, поскольку он включает работу, выполняемую самим пользователем.

Цикл реализатора представлен на нижней части (рис. 1). Его составляют отображения ω (ввод данных), τ (компиляция программы), ω^{-1} (вывод результатов), которые планируются и конструируются реализатором, а исполняются компьютером M_i.

Полный цикл обработки информации состоит из неавтоматического цикла пользователя и полуавтоматического цикла реализатора. Разработка реализации языка производится человеком, компиляция и машинная обработка данных происходят автоматически.

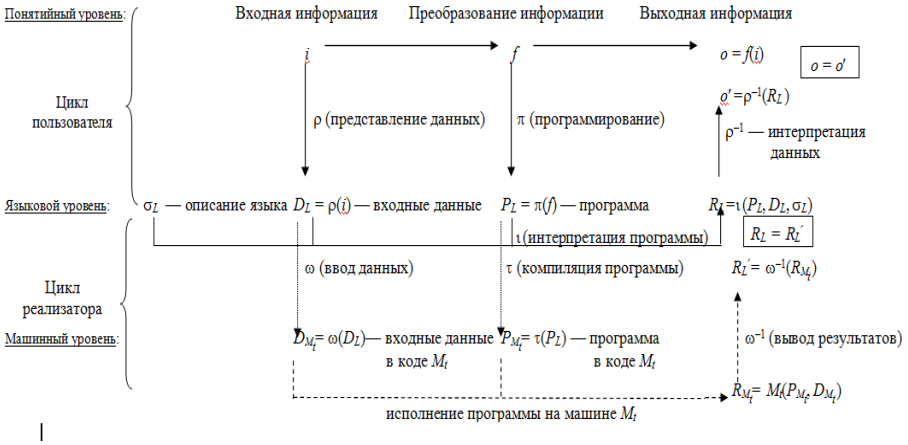

Рис. 1. Схема автоматизированной обработки данных

2 Синтаксическая модель языка

Модель языка — это способ его описания или рассмотрения. С помощью фиксированной модели можно описывать бесконечно много конкретных языков. Синтаксическая модель различает четыре аспекта языка: *лексику, синтаксис, семантику и прагматику* [4–6].

Лексика определяет представление основных символов языка (терминальных символов грамматики) при помощи знаков или кодов входного

устройства компьютера. Обычно лексика задается стандартом языка в виде перечня допустимых комбинаций знаков для каждого терминала. Один терминал может иметь несколько разных вариантов конкретного представления. Из них реализатор отбирает представления, подходящие для используемого оборудования. Иногда терминал представляется одной литерой, например, символ *открыть* может быть представлен литерой ‘(’, или несколькими литерами, как, например, ‘begin’, или символ *присвоить* в виде цепочки литер ‘:=’.

Задача лексического анализа — отображение цепочек литер конкретного представления программы в последовательность терминальных символов (*основных символов языка*).

Синтаксис определяет представление конструкций языка посредством терминальных символов. Обычно он описывается при помощи контекстно-свободной грамматики, правила которой определяют одно или несколько альтернативных порождений для каждого нетерминала грамматики. В последнее время в качестве порождений для нетерминала используют регулярные множества [8–11]. Начальный нетерминал порождает конструкцию языка, называемую *программой*. Синтаксис определяет, из каких других конструкций состоит данная конструкция (в первую очередь программа), если она не элементарна. Эти *конструкции*, составляющие данную, называются ее *подконструкциями*. Не все нетерминалы порождают конструкции, а только те, для которых определена семантика. Никакой терминальный символ или цепочка терминальных символов не имеют смысла сами по себе. Терминальная цепочка имеет смысл только в качестве терминального порождения некоторого понятия (нетерминала), которому смысл приписывается соответствующими правилами *семантики*. Под смыслом терминальной цепочки понимается некоторая последовательность действий по исполнению конструкции, представленной данной цепочкой. Определение конструкционной структуры программы — задача *бесконтекстного синтаксического анализа*.

Синтаксис определяет не только конструкционную структуру программы (контекстно-свободный синтаксис), устанавливающую частичный порядок действий, из которых складывается ее исполнение (подконструкции исполняются раньше, чем сама конструкция), но и *виды конструкций* (контекстный синтаксис), от которых зависит способ реализации этих действий.

Каждая конструкция имеет определенный тип — и такой же тип имеет результат ее исполнения — значение. Различаются априорные и апостериорные типы конструкций и значений.

Априорный вид литеральной константы определяется бесконтекстно по ее изображению. Например, 123 — целого, 3.14 — вещественного, “c” — литерного, **true** — логического вида. Имеются достаточные различительные признаки, чтобы по самой цепочке литер однозначно определить, какого типа литеральную константу она представляет.

Идентификаторы, как известно, выбираются программистом по его усмотрению в качестве обозначений именованных констант, переменных,

процедур и функций, а также типов. Они вводятся в программу соответственно посредством конструкций, называемых описаниями переменных, процедур и т.д. Исключение составляют идентификаторы стандартного и библиотечного окружения, которые считаются описанными вне конкретной программы. В некоторых языках допускается описывать собственные операции в частной программе, и для их обозначения используются *индикаторы* операций. Они определяются в описаниях операций и приоритетов. Иногда индикаторы используются для представления типов, и тогда они вводятся в программу с помощью конструкций, называемых описаниями типов.

Различаются определяющие и использующие вхождения идентификаторов и индикаторов. *Определяющее вхождение идентификатора* — это его вхождение в соответствующее описание (именованной константы, переменной, процедуры или функции), и именно оно определяет априорный тип идентификатора во всех его использующих вхождениях.

Конструкции-описания могут встречаться лишь в составе конструкций, называемых блоками, причем блочная структура программы такова, что любые два блока либо вложены один в другой (находятся на одной и той же ветви дерева вывода программы), либо не имеют между собой ничего общего (находятся в разных ветвях этого дерева).

Каждое использующее вхождение идентификатора или индикатора должно идентифицировать ровно одно определяющее вхождение.

Использующее вхождение идентификатора идентифицирует определяющее вхождение в том же блоке, если оно там имеется, или в одном из объемлющих его блоков. В крайнем случае идентификатор должен определяться библиотечным или стандартным описанием. Наличие нескольких определяющих вхождений для одного идентификатора (в одном блоке) или вовсе отсутствие такового является нарушением контекстных условий идентификации и должно рассматриваться как *контекстная синтаксическая ошибка*.

Задача синтаксического анализа — определить конструкционный состав программы, установить априорные и апостериорные типы конструкций и найти соответствующие приведения — последовательности действий, преобразующих априорное значение конструкции к требуемому апостериорному типу.

Задача реализатора состоит в том, чтобы разработать форму представления всей информации о программе, необходимой для генерации объектного кода, и механизмы ее извлечения из входной программы и выстраивания во время синтаксического анализа.

Семантика определяет исполнение каждой конструкции в соответствии с ее синтаксической структурой. Именно, результат исполнения конструкции — ее *значение* — определяется как некоторая последовательность действий над значениями составляющих ее подконструкций. Правило семантики для данной составной конструкции определяет, какие действия должны быть выполнены над апостериорными значениями подконструкций,

чтобы получить априорное значение данной конструкции. Помимо этих действий в исполнении конструкции задействованы *приведения* – действия по преобразованию априорных значений подконструкций в соответствующие апостериорные значения. В итоге значение составной конструкции выражается через априорные значения элементарных конструкций.

Задача реализатора языка — разработать адекватную интерпретацию действий гипотетического вычислителя по исполнению каждой языковой конструкции на реальной целевой машине. Конкретно он должен подобрать адекватное представление типов данных в объектном коде и выразить действия гипотетического вычислителя по исполнению каждой языковой конструкции в терминах машинных команд над соответствующими машинными форматами данных. При этом, возможно, ему необходимо решить проблему рационального размещения данных в памяти и на регистрах машины во время исполнения объектной программы, то есть разработать схему организации и управление памятью данных в объектном коде.

Прагматика. В приведенной в разделе 1 схеме предполагается, что пользователь, разработчик (реализатор) и, соответственно, транслятор руководствуются одним и тем же определением входного языка. Однако в случае синтаксической неоднозначности грамматики семантика одной и той же программы, выводимая из синтаксической ее структуры, может оказаться также неоднозначной. Кроме того, даже при однозначной грамматике для некоторых конструкций реализатором языка может быть предусмотрено несколько вариантов реализации. Во всех таких случаях проявляется еще один аспект — отношение пользователя к семантике программы, когда она неоднозначна. Пользователю важно иметь средства выбора предпочтительной для него семантики программы, когда существует несколько вариантов ее исполнения. Чаще всего в качестве таких средств используются *прагматические комментарии*, которые в отличие от обычных комментариев системой программирования не игнорируются, а принимаются к исполнению.

3 Схема процесса компиляции

Схема процесса компиляции представлена на рисунке 2. Каждый блок программы транслятора (кроме самого первого) получает результаты работы предыдущего блока через оперативную память или последовательные файлы в виде таблиц и программы в форме, удобной для ее обработки.

Таким образом, каждая пара соседних блоков имеет свой промежуточный язык. Последний блок, завершающий компиляцию, выдает объектный код.

Первый блок получает текст входной программы политерно из входного потока. Его задача отнести каждую литеру к одному из классов-множеств литер — *микролексическому классу* (буква, цифра, открывающая скобка, точка с запятой и т. п.) и передать на свой выход код этой литеры вместе с кодом микролексического класса, которому она принадлежит. Эту пару литера + код класса назовем *микролексемой*. Способ обработки литеры лексическим анализатором выполняется *сканером*. Преобразование литеры в микролексему — задача *транслитератора*.

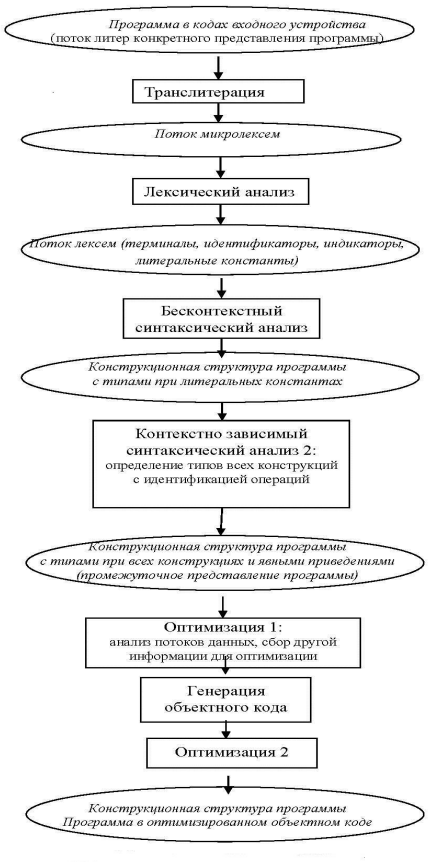

Рис. 2. Общая схема процесса компиляции

4 Порядок разработки проекта реализации языка

Порядок разработки проекта реализации языка противоположен порядку самого процесса трансляции.

4.1 Проектирование генерирующей части компилятора

Прежде чем генерировать код, реализующий действия по исполнению программы, следует определить, какими командами целевой машины будет реализовываться каждая конструкция языка. В общем случае для составных конструкций такое определение будет рекурсивным. Невозможно разрабатывать блок генерации объектного кода, не решив вопрос о представлении в командах элементарных конструкций. Например, для представления значений простых типов (**bool**, **int**, **real** и т. д.), как правило, подбираются подходящие машинные форматы данных. Это гарантирует достижение максимальной эффективности в исполнении действий над

ними посредством соответствующих машинных команд. Действия же, главным образом, определяются теми операциями, которые определены в языке над этим типом данных.

Выбор представления данных в машинном коде означает и определение структуры памяти, в которой такое представление размещается. Для составных значений, а тем более для рекурсивных типов данных, структура памяти будет ассоциативной, т. е. состоять из разрозненных областей, связанных указателями.

После того, как решено, какими командами воспроизводить действия, в терминах которых определяется семантика конструкций языка, можно приступать к разработке представления конструкций в машинном коде. Возникает вопрос о том, как организовать взаимодействие конструкции и составляющих ее подконструкций, ибо исполнение конструкции складывается из исполнения всех или некоторых из ее подконструкций, а ее результат образуется из значений этих подконструкций. В программе имеются именованные величины, представленные идентификаторами, которые определяются соответствующими описаниями. Именно через них пользователь, пишущий программу, получает доступ к данным во время ее исполнения. Каждое описание входит в состав блока, фрагмента текста программы, в пределах которого только и приписывается идентификатору его роль. Говорят, что блок, в котором описан идентификатор, является его областью действия. Каждому блоку программы во время ее исполнения сопоставляется блок данных — некоторая структура в памяти машины, причем к полям этой структуры пользователь получает доступ через идентификаторы, описанные в этом блоке. Блочная структура программы подразумевает размещение блоков доступа к данным в режиме стека. Наличие в языке процедур не препятствует использованию стека, а лишь несколько усложняет поддержание системы адресации.

В некоторых языках программирования [7] предусматриваются также конструкции для освобождения динамической памяти, так что использование динамической памяти полностью подконтрольно пользователю, пишущему программу. В тех языках, где нет средств освобождения динамической памяти, реализаторы языка должны брать эту проблему на себя. Тогда говорят о «сборке мусора», т. е. механизме, позволяющем определить, с какими данными в динамической памяти программа потеряла связь, освободить эти области и реорганизовать размещение в ней актуальных данных. Сборка мусора связана с регистрацией областей динамической памяти, доступных из блоков доступа в стеке, уплотнением данных и переопределением указателей, выводящих на перемещенные данные.

Проектирование семантической части компилятора заканчивается выяснением, какая информация о программе необходима и достаточна для генерации машинного кода, каким образом она должна быть представлена и каким механизмом генерации использована. Две последние проблемы должны решаться совместно. Это решается в определении промежуточного представления программы, которое сгенерируется анализирующей частью компилятора (parser).

Заключение

В статье (часть 1) обсуждается методика автоматизированной обработки информации, в основе которой положена регулярная модель языка и языковой процессор как основной элемент реализации трансляции. Идеи, положенные в основу данной методики, связаны с возможностью использования регулярных выражений в правых частях правил грамматики и определяют способ их представления в виде ориентированных графов [7–11]. Первая часть работы посвящена описанию принципиальной схемы автоматизированной обработки информации с использованием языка программирования высокого уровня.

Вторая часть работы посвящена разработке анализатора. Сформулированы ограничения на грамматику, представленную в форме системы ориентированных графов (синтаксической граф-схемы), гарантирующие существование детерминированного магазинного анализатора, который далее рассматривается как управляющий механизм для инициирования действий, составляющих процесс трансляции. Специфику методики составляет *алгоритм регуляризации* грамматики, основанный на эквивалентных преобразованиях грамматики входного языка. Регуляризация исходной грамматики является *частью полного цикла реализации языка.* Детальное описание каждого этапа компиляции, в том числе этапы предварительной подготовки синтаксических конструкций реализуемого языка программирования [7–11] и представления программы на этом этапе будут даны в части 2.

Литература

1. Ахо А., Ульман Дж. Теория синтаксического анализа, перевода и компиляции. Москва: Мир, 1978. Т. 1. 612 с.

2. Грис Д. Конструирование компиляторов для цифровых вычислительных машин. Москва: Мир, 1975. 544 с.

3. Касьянов В. Н., Поттосин И. В. Методы построения трансляторов. Новосибирск: Наука, 1986. 343 с.

4. Льюис Ф., Розенкранц Д., Стирнз Р. Теоретические основы проектирования компиляторов. Москва: Мир,, 1979. 654 с.

5. Пратт Т. Языки программирования: разработка и реализация. Москва: Мир, 1979. 574 с.

6. Aho A., Sethi R., Ullman J., *Compilers: Principles, Techniques and Tools.* Addison-Wesley Publishing Company, 1986. 796 p.

7. Fraser C., Hanson D. *A retargetable C compiler: Design and implementation.* Addison-Wesley Pub. Company, Menlo Park, California, 1995. 564 p.

8. Fedorchenko L. *Regularization of Context-Free Grammars.* LAP LAMBERT Academic Publishing, Saarbrucken, 2011, 188 p.

9. Федорченко Л. Н. О регуляризации контекстно-свободных грамматик // Изв. вузов. Приборостроение. 2006. Т. 49, № 11, С. 50–54.

10. Федорченко Л. Н. Синтаксически управляемая обработка данных для практических задач // Вестник БГУ. 2013. № 9. С 87–99.

11. Ludmila Fedorchenko and Sergey Baranov Equivalent Transformations and Regularization in Context–Free Grammars. *Bulgarian Academy of Sciences/ Cybernetics and Information Technologies (CIT).* 2015; 14 (4): 11–28.

Статья поступила в редакцию 13.02.2024; одобрена после рецензирования 28.03.2024; принята к публикации 29.03.2024.

METHODOLOGY FOR AUTOMATED INFORMATION PROCESSING USING A HIGH LEVEL PROGRAMMING LANGUAGE (PART 1)

Ludmila N. Fedorchenko
PhD, Senior Researcher
St. Petersburg Federal Research Center of the Russian Academy of Sciences (SPC RAS)
39 14th Line of V.O., St. Petersburg 199178, Russia
lnf@iias.spb.su

Abstract: The paper discusses a technique for automated information processing, which is based on a regular language model and a language processor as the main element of translation implementation. The ideas underlying this technique are related to the possibility of using regular expressions on the right sides of grammar rules and determine the way they are represented in the form of directed graphs. Restrictions on the grammar, presented in the form of a system of directed graphs (syntactic graph-scheme), are formulated to guarantee the existence of a deterministic store analyzer, which is further considered as a control mechanism for initiating actions that make up the translation process. The specificity of the technique is the grammar regularization algorithm, based on equivalent transformations of the grammar of the input language. Regularization of the source grammar is part of a complete language implementation cycle, consisting of a user cycle and a semi-automatic developer cycle. The article presents a schematic diagram of automated information processing using a programming language.

Keywords: automated information processing, compilation process diagram, syntactic language model, language processor.

For citation
Fedorchenko L. N. Methodology for Automated Information Processing Using a High Level Programming Language (Part 1) // Bulletin of Buryat State University. Mathematics, Informatics. 2024. N. 1. P. 46–55.

The article was submitted 13.02.2024; approved after reviewing 28.03.2024; accepted for publication 29.03.2024.