Министерство образования и науки Российской Федерации
НОВОСИБИРСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ

А.А. МАЛЯВКО

СИСТЕМНОЕ ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ

ФОРМАЛЬНЫЕ ЯЗЫКИ И МЕТОДЫ ТРАНСЛЯЦИИ

Часть 3

Утверждено Редакционно-издательским советом университета
в качестве учебного пособия

НОВОСИБИРСК
2012

УДК 004.43(075.8)
М 219

Рецензенты:

А.В. Гунько, канд. техн. наук, доцент,
Е.Л. Романов, канд. техн. наук, доцент

Работа подготовлена на кафедре вычислительной техники
Новосибирского государственного технического университета
для студентов IV курса АВТФ

Малявко А.А.

М 219 Системное программное обеспечение. Формальные языки и методы трансляции: учеб. пособие. В 3 ч. / А.А. Малявко. – Новосибирск: Изд-во НГТУ, 2012. – Ч. 3. – 120 с.

ISBN 978-5-7782-1960-1

В третьей части учебного пособия рассматриваются задачи, решаемые семантическими анализаторами и генераторами объектного кода трансляторов. Основное внимание уделяется принципам, закладываемым в организацию памяти транслируемой программы, и методам доступа к локальным и нелокальным данным процедур. На этой основе рассмотрены применяемые в современных языках подходы к контролю типов данных и функции семантического анализа. Обсуждаются основные задачи генератора кода, такие как формирование последовательности тетрад, управление памятью, выбор инструкций, распределение регистров и порядок вычислений; рассматриваются методы оптимизации кода.

Пособие адресовано студентам старших курсов и аспирантам, а также преподавателям смежных дисциплин. Оно может быть полезно студентам и аспирантам ряда других технических специальностей, связанных с разработкой и использованием программного обеспечения.

УДК 004.43(075.8)

ISBN 978-5-7782-1960-1

1. СЕМАНТИЧЕСКИЙ АНАЛИЗ

Краткое введение

Совокупность исходных данных для семантического анализа формируется предыдущими этапами процесса трансляции.

1. Постфиксная форма записи (ПФЗ) транслируемой программы – промежуточное представление, которое образуется в процессе синтаксического анализа.

2. Набор информационных таблиц, в которых накоплены сведения об используемых программой данных.

Постфиксная форма записи обладает замечательным свойством: в ней порядок следования знаков операций строго совпадает с предусматриваемой алгоритмом решения задачи последовательностью их выполнения (в отличие от исходного текста программы, где этого обычно и не требуется).

Реальное исполнение выявленной транслятором последовательности операций с целью преобразования исходных данных транслируемой программы в ее результаты возможно различными способами, все множество которых принято делить на две основные группы – компиляция и интерпретация.

1. *Компиляция* – продолжение преобразований представления программы на стадии трансляции, завершающееся формированием и сохранением так называемого объектного (целевого) кода, который впоследствии может многократно исполняться процессором компьютера (реальной машиной) уже без какого бы то ни было участия транслятора.

2. *Интерпретация* – исполнение последовательности операций непосредственно транслятором (прямая интерпретация) или специально разработанной для этого программой (так называемой виртуальной машиной – отложенная интерпретация), возможно – с предварительным преобразованием постфиксной записи в более удобную промежуточную или сохраняемую форму.

Четкой границы между компиляцией и интерпретацией не существует. На практике, как правило, реализуются смешанные варианты последовательности действий трансляции/исполнения.

Например, в скомпилированных программах обычно содержатся вызовы функций из библиотек так называемого run-time окружения (поддержки), которые, по существу, интерпретируют операции, являющиеся примитивными в языке программирования, но не реализуемые аппаратурой компьютера на уровне машинных команд. Для обеспечения исполнения подобных операций транслятор вставляет в компилируемый им объектный код команды вызова таких интерпретирующих функций.

В свою очередь, многие системы программирования (такие как Visual Basic), которые первоначально были ориентированы на прямую интерпретацию, впоследствии приобрели ряд признаков компиляции, выполняя перевод текста программы в псевдокод (или байт-код). Псевдокод сохраняется во внешней памяти и может многократно интерпретироваться виртуальной машиной уже без использования таких частей транслятора, как лексический и синтаксический анализаторы.

Для того чтобы исполнять программу в целом (после компиляции или при отложенной интерпретации) или даже по частям (при прямой интерпретации) необходимо предварительно убедиться в том, что каждая исполняемая операция действительно может быть применена к значениям своих операндов, и сформировать требуемый результат. Именно это является основной целью семантического анализа.

Таким образом, в логической последовательности этапов процесса трансляции семантический анализ занимает позицию непосредственно после синтаксического и перед генерацией объектного кода в случае компиляции либо перед исполнением вычислений в случае интерпретации.

Реальная последовательность выполнения этапов при трансляции отнюдь не обязательно совпадает с логической последовательностью. Семантическая проверка некоторого фрагмента программы может предшествовать во времени синтаксическому анализу последующего по тексту фрагмента и/или осуществляться после интерпретации текстуально предыдущего фрагмента (или его преобразования в объектный код). Однако применительно к каждому конкретному синтаксически целостному фрагменту текста (модулю/файлу, подпрограмме/функции, блоку, оператору, …) логическая последовательность этапов трансляции всегда строго соблюдается.

Преобразуются в объектный код при компиляции или исполняются при интерпретации только те фрагменты текста, для которых все этапы анализа (лексический, синтаксический и семантический) завершились успешно.

В том случае, если возможно выполнение всех этапов анализа произвольного фрагмента текста программы без обработки всего ее текста целиком, говорят, что язык допускает однопроходную трансляцию. Ряд языков (Pascal, C и др.) был спроектирован таким образом, чтобы можно было обеспечить возможность быстрой однопроходной трансляции.

Однако существуют и такие языки программирования, в которых выполнение всех функций анализа некоторого фрагмента текста программы невозможно без предварительной обработки последующих фрагментов для извлечения и накопления сведений об используемых объектах и их свойствах (например, Алгол-68 или Ада).

Трансляторы с таких языков вынужденно реализуют более чем один проход по тексту программы (или по одному из ее промежуточных представлений – последовательности лексем, постфиксной записи, …).

Увеличение количества проходов, с одной стороны, приводит к росту затрат времени на трансляцию программ и сложности транслятора, а с другой – хорошо согласуется с потребностями алгоритмов оптимизации программы.

Семантический анализатор транслятора в процессе проверки правильности транслируемой программы осуществляет преобразование ее постфиксной формы записи (или эквивалента ПФЗ – дерева операций), формируемой синтаксическим анализатором, в псевдокод, как показано на рис. 1.1.

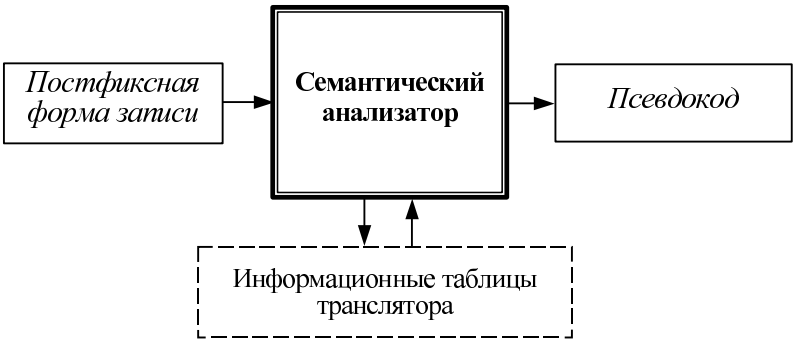


Рис. 1.1. Структура семантического анализатора

Например, пусть транслятор с языка C/C++ обрабатывает функцию вычисления наибольшего общего делителя ее аргументов:

```
int nod( int first, int second ){
        while ( first != second )
                if ( first < second )
                        second –= first;
                else
                        first –= second;
        return first;
}
```

Постфиксная запись этой функции, построенная синтаксическим анализатором, будет выглядеть примерно так:

nod int functionActivate first int getArgument second int getArgument label#0#0: first second != label#0#1 jmpOnFalse first second ≤ label#1#0 jmpOnFalse second first –= label#1#1 jmp label#1#0: first second –= label#1#1: label#0#0 jmp label#0#1: first return

Здесь:

nod int functionActivate first int getArgument second int getArgument – ПФЗ заголовка функции;

label#0#0: first second != label#0#1 jmpOnFalse – ПФЗ заголовка оператора цикла;

first second ≤ label#1#0 jmpOnFalse second first –= label#1#1 jmp label#1#0: first second –= label#1#1: – постфиксная запись условного оператора, составляющего тело цикла;

label#0#0 jmp – ПФЗ завершения оператора цикла;

label#0#1: first return – ПФЗ оператора возврата значения функции.

В этой форме записи для наглядности подчеркнуты все знаки операций. Большинство знаков операций (*functionActivate, getArgument, !=, jmpOnFalse, ≤, –=*) являются бинарными, но есть и унарные знаки (*jmp* и *return*).

Требуемая последовательность выполнения операций программы выявлена и зафиксирована в постфиксной форме записи, что позволит впоследствии построить последовательность машинных команд, которые процессор будет выбирать из рядом расположенных ячеек памяти. Однако с этой формой записи могут быть связаны и другие проблемы. Рассмотрим, например, фрагмент ПФЗ заголовка цикла:

label#0#0: first second != label#0#1 jmpOnFalse

У бинарного знака операции *jmpOnFalse* в этом фрагменте первым операндом является другой бинарный знак операции *!=*. В действительности это означает, что первым операндом операции *jmpOnFalse*

является булево значение, вырабатываемое операцией сравнения. Это значение должно быть сохранено либо в стеке времени выполнения, либо в переменной, формируемой транслятором.

Для того чтобы выявить все такие переменные и выполнить соответствующие семантические проверки, транслятором формируется очередное внутреннее представление текста программы, называемое псевдокодом.

Псевдокод – это последовательность операций в системе команд некоторой виртуальной машины. Эта машина может быть, например, трехадресной, в этом случае каждая ее команда включает в себя 4 поля (и называется тетрадой): код операции, наименования двух операндов и наименование результата. Фрагмент псевдокода тела функции вычисления наибольшего общего делителя для такой виртуальной машины может выглядеть так, как показано в табл. 1.1.

Виртуальная машина, псевдокод которой показан в табл. 1.1, является стековой. Результаты некоторых операций заносятся в стек с помощью указания имени верхушки стека *push* и извлекаются из стека в последующих тетрадах с использованием другого имени верхушки стека *pop*.

Таблица 1.1

Код операции	**Первый операнд**	**Второй операнд**	**Результат**
defineLabel	*label#0#0*		
!=	*second*	*first*	*push*
jmpOnFalse	*label#0#1*	*pop*	
<	*second*	*first*	*push*
jmpOnFalse	*label#1#0*	*pop*	
–=	*first*	*second*	*second*
jmp	*label#1#1*		
defineLabel	*label#1#0*		
–=	*second*	*first*	*first*
defineLabel	*label#1#1*		
jmp	*label#0#0*		
defineLabel	*label#0#1*		
return	*first*		

Недостатком тетрад является то, что для объявления имени (метки) какой-либо операции приходится добавлять специальную тетраду с

кодом операции «Создать метку». Этого недостатка лишены виртуальные машины, у которых каждая операция имеет дополнительное поле метки. Команды такой виртуальной машины называют пентадами (по количеству полей).

В табл. 1.2. показан псевдокод той же функции в виде последовательности пентад. Количество операций уменьшилось за счет удаления операций «Создать метку». В этом варианте виртуальной машины для хранения промежуточных результатов вычислений вместо стека используются временные переменные, создаваемые семантическим анализатором транслятора.

Таблица 1.2

Метка	Код операции	Первый операнд	Второй операнд	Результат
label#0#0	*!=*	*second*	*first*	*tmpVar1*
	jmpOnFalse	*label#0#1*	*tmpVar1*	
	<	*second*	*first*	*tmpVar2*
	jmpOnFalse	*label#1#0*	*tmpVar2*	
	–=	*first*	*Second*	*second*
	jmp	*label#1#1*		
label#1#0	*–=*	*second*	*First*	*first*
label#1#1	*jmp*	*label#0#0*		
label#0#1	*return*	*first*		

Возможны и другие варианты виртуальных машин. Пример псевдокода той же функции для виртуальной машины, операции которой не имеют ни поля метки, ни поля наименования результата, приведен в табл. 1.3.

Таблица 1.3

Код операции	Первый операнд	Второй операнд
!=	*second*	*first*
jmpOnFalse	*+8*	*–1*
<	*second*	*first*
jmpOnFalse	*+4*	*–1*
–=	*first*	*second*

Окончание табл. 1.3

Код операции	Первый операнд	Второй операнд
=	*–1*	*second*
jmp	*+3*	
–=	*second*	*first*
=	*–1*	*first*
jmp	*–9*	
return	*first*	

В этом примере вместо имен операций и промежуточных результатов используется относительная адресация. Числа вида *+8* или *–1* в полях наименований операндов означают указание триады, находящейся на расстоянии 8 вперед или 1 назад по отношению к текущей триаде. Если относительный номер триады используется в операции преобразования данных, то под ним понимается результат, вырабатываемый адресуемой триадой.

Алгоритмы и методы преобразования постфиксной записи в псевдокод рассматриваются в разд. 2.1.

Таким образом может быть построено внутреннее представление транслируемой программы в виде псевдокода. Далее для каждой операции виртуальной машины семантический анализатор должен выполнить:

1) определение типа данных первого операнда;

2) определение типа данных второго операнда;

3) проверку, применим ли код операции к данным этих типов;

4) формирование и сохранение типа данных результата операции.

Все эти задачи можно решать и прямо в процессе формирования операций псевдокода.

Для того чтобы определить, как могут решаться эти задачи, рассмотрим основные понятия, которые объединяются термином «семантика языков программирования». При этом нам понадобится постоянно заглядывать вперед, в тот период времени, когда анализируемая программа будет исполняться либо аппаратурой реального компьютера, либо интерпретатором виртуальной машины.

1.1. Программы и данные

Языки программирования бывают процедурными и непроцедурными. Программы на непроцедурных языках являются совокупностями инструкций, определяющих, что нужно сделать, но не содержащих указаний на то, в какой последовательности эти инструкции должны быть выполнены. К непроцедурным относятся такие языки, как Haskell, Lisp, Prolog.

Здесь будут рассматриваться только процедурные языки, программы на которых представляют собой последовательность инструкций, определяющих не только то, что должно быть сделано, но и то, в каком порядке должны выполняться эти действия. Процедурными являются языки C, Pascal, Java, Fortran, Ada и многие другие.

Процедурными эти языки называются, в частности, потому, что они предоставляют программисту возможность структурировать представление сложных алгоритмов в виде совокупности логически обособленных друг от друга взаимодействующих программных единиц – процедур (подпрограмм).

Текст любой процедуры представляет собой объявление, которое связывает ее наименование (точнее – заголовок, содержащий кроме наименования процедуры еще и описание ее аргументов) с последовательностью инструкций – телом процедуры, и может содержать объявления других процедур и/или обрабатываемых данных. Текст основной программы в процедурном языке также является процедурой.

Под объявлением будем понимать такое описание свойств объекта (процедуры или элемента данных), которое порождает собственно объект и связывает его с заданными в объявлении свойствами. В некоторых языках объявлением считается первое использование наименования (ссылка на объект). Кроме объявлений возможны описания, не приводящие к образованию соответствующего объекта, но требующиеся транслятору для связывания инструкций, в которых используется наименование объекта, со свойствами этого объекта.

Термин «связывание» исключительно важен для понимания существа процессов семантического анализа. Дать исчерпывающее определение понятию связывания не представляется возможным.

Пока просто проиллюстрируем это понятие, рассмотрев показанный на рис. 1.2 фрагмент программы на языке C++ (здесь и далее для удобства ссылок на элементы программ номера строк текста помещаются слева в скобках).

```
...
(k)        unsigned long i;
(k+1)      i = 1;
...
(m)        for ( int i = 0; i<10; i++){
 ...
(n)        }
...
```

Рис. 1.2. Фрагмент программы для иллюстрации понятия связывания

Объявление *unsigned long* в строке (k) связывает наименование переменной *i* с типом значения «беззнаковое длинное целое». Это связывание, очевидно, выполняется в процессе трансляции текста программы, оно необходимо для правильного понимания смысла последующего текста при его преобразовании и/или исполнении.

Оператор присваивания *i=1* в строке (k+1) связывает значение выражения из его левой части (в данном случае – равное единице) с переменной *i*, наименование которой записано в правой части. Это связывание, очевидно, выполняется при исполнении программы. Значение, которое получит переменная *i*, будет использовано позднее.

Строка (m) содержит сразу несколько связываний. Одно из них является объявлением *int*, связывающим наименование *i* с типом значения «целое». Выполнение этого связывания во время трансляции приведет к тому, что объект с именем *i*, объявленный ранее в строке (k), окажется «невидимым» вплоть до обработки транслятором закрывающей фигурной скобки блока в строке с номером (n). Вместо него в заголовке и теле цикла с наименованием *i* теперь ассоциируется другой объект, тип значения которого – тоже «целое». Именно этот объект получит значение 0 в результате выполнения связывания *i=0* в процессе исполнения программы. Значение «закрытого» объекта *i* при этом не изменится.

Таким образом, для того чтобы исполнить любую программу, необходимо выполнить большое количество связываний, причем в разные периоды времени. От того, на какой стадии – компиляции или исполнения программы – выполняются те или иные связывания, зависят и скорость ее работы, и объем памяти, требуемой для хранения инструкций, осуществляющих эти связывания. Более того, от этого зависит изобразительная мощь и гибкость языка. Характер этой зависимости очень прост: чем раньше осуществляется тот иной набор связываний,

тем быстрее будет работать программа, но тем большие ограничения будут наложены на возможность оперировать с соответствующими атрибутами объектов в тексте программы.

Для примера рассмотрим использование массивов в языках Fortran, Pascal и C. Языки Fortran и Pascal требуют объявления как количества размерностей, так и диапазона изменения индексов по каждой размерности в качестве константных значений. Это позволяет выполнить подавляющую часть связываний (в том числе – определение требуемого размера памяти под хранение массива) во время трансляции программы. На период исполнения остается минимально возможное количество связываний (вычисление координат элемента по значениям его индексов, выборка и занесение значения). Однако за достигаемую в результате этого высокую скорость исполнения программы приходится платить невозможностью оперировать с так называемыми динамическими массивами (такими, у которых количество размерностей и/или диапазоны изменения индексов по ним неизвестны до момента исполнения программы).

В языке C массивы могут иметь как константные, так и вычисляемые при исполнении программы размерности и границы. Более того, факт проецирования многомерных массивов на линейно (т. е. одномерно) организованную память вынесен на уровень языковых конструкций (идентичность смысла имени массива и указателя на его первый элемент). Как следствие этого – большая часть связываний при доступе к элементам массива осуществляется при исполнении программы (количество выполняемых машинных инструкций больше, чем при исполнении Pascal- или Fortran-программ).

В качестве инструкций или их составных частей в тексте процедуры могут использоваться наименования других процедур. Использование наименования процедуры в тексте инструкции подразумевает вызов этой процедуры во время исполнения программы в тот момент, когда выполняется эта инструкция. Под вызовом одной процедуры (вызываемой) из другой (вызывающей) понимается временное прекращение исполнения инструкций вызывающей процедуры, исполнение вызываемой процедуры и восстановление исполнения вызывающей процедуры. Если язык допускает возможность вызова процедуры из нее самой, то говорят о рекурсивности процедур. Рекурсивность может быть прямой, если вызов процедуры содержится в ее же тексте, и косвенной.

Кроме процедур часто используются функции – подпрограммы, возвращающие некоторое значение. В некоторых языках (например, C/C++) как процедуры, так и функции считаются функциями (при этом процедура рассматривается просто как функция, возвращающая значение *void*).

С помощью механизма вызова обеспечивается взаимодействие процедур/функций, составляющих программу в целом, по управлению. Кроме того, от языка программирования требуется обеспечить взаимодействие между составными частями программы по данным. Вызов любой процедуры/функции осуществляется не просто для того, чтобы передать управление инструкциям, составляющим ее тело. Процедура/функция вызывается для того, чтобы переработать ее исходные данные в определенные результаты, которые нужны как данные другим составным частям программы. Здесь следует иметь в виду, что в процессе исполнения программы в целом будут использоваться так называемые библиотечные (run-time) процедуры/функции, трансляция которых выполнялась задолго до трансляции любой прикладной программы. Более того, вполне возможно, что эти процедуры/функции были написаны на другом языке программирования.

Таким образом, любая процедура/функция должна уметь оперировать как с данными, объявленными в ее тексте, так и с данными, объявления которых находятся где-то вне ее текста и отнюдь не обязательно доступны транслятору. Объекты, объявленные в тексте процедуры, называются ее локальными объектами. Все прочие объекты, наименования которых используются в инструкциях тела процедуры, называются нелокальными. К нелокальным объектам относятся, в частности, так называемые аргументы (параметры) процедуры. Аргументами называются объекты данных, объявленные где-то вне процедуры, но описанные в ее заголовке. Значения объектов-аргументов либо используются при выполнении процедуры, либо формируются в результате ее исполнения (а могут вначале использоваться, а затем замещаться). Имеется несколько принципиально различных механизмов организации доступа к аргументам процедур, их существо обсуждается ниже.

С помощью аргументов процедур/функций в принципе можно реализовать все необходимые взаимосвязи между ними по данным. Однако в большинстве языков программирования реализуются и другие механизмы доступа процедур к их нелокальным объектам. Для понимания существа таких механизмов необходимо всегда помнить о

том, в каких условиях выполняется трансляция (семантический анализ и генерация кода при компиляции) процедур и что происходит при их исполнении (в частности, при интерпретации).

Транслятору доступен только текст программы, причем, как уже упоминалось, скорее всего, не весь. В процессе трансляции любые объекты представлены только их наименованиями, взятыми из объявлений, описаний или первого использования по тексту. С наименованиями могут быть (но не обязательно должны быть) связаны некоторые характеристики (атрибуты), извлекаемые опять-таки из доступного текста. Каждый объект представлен единственным экземпляром сочетания наименования и атрибутов. Момент возникновения информации об объекте определяется той точкой в тексте транслируемой программы, в которой содержится определение, описание или просто употребление его наименования. Актуальность этой информации, т. е. возможность использования данного объекта в любой другой точке текста программы, не гарантирована автоматически с момента ее возникновения, а определяется совокупностью правил, зависящих, в частности, от того, что происходит с объектами в процессе исполнения программы.

С текстом любой процедуры естественным образом ассоциируется понятие его исполнения, осуществляющееся в результате вызова из некоторой другой процедуры. Исполнение программы в целом начинается с вызова ее основной или главной процедуры. Вызывающей процедурой в этом случае является составная часть операционной системы, называемая загрузчиком и не имеющая никакого иного (кроме переноса образа программы в оперативную память и передачи ей управления) отношения к задачам, решаемым программой.

Вызов любой (в том числе и основной) процедуры в действительности есть сложный процесс, завершающийся передачей управления на самую первую инструкцию ее тела. Этот процесс протекает в промежутке времени между выполнением последней инструкции перед передачей управления в вызывающей процедуре и первой инструкцией тела вызываемой процедуры (здесь имеются в виду инструкции, написанные разработчиками текстов вызывающей и вызываемой процедур). Суть этого процесса рассматривается ниже. Здесь отметим только, что в результате выполнения этого процесса создается так называемая активация вызываемой процедуры.

В момент запуска программы на исполнение гарантированно существует только ее главная процедура (включая ее локальные объекты).

Существование всех прочих объектов программы зависит от свойств языка программирования.

В некоторых языках (таких, например, как Fortran) все объекты программы создаются в момент ее загрузки в память компьютера и существуют в течение всего времени исполнения программы. В других языках все локальные объекты (объявленные в текстах процедур/функций) в момент запуска программы являются (могут считаться) несуществующими, даже если по тексту эти процедуры предшествуют основной процедуре.

Будем считать, что сами такие процедуры (как объекты) уже существуют, поскольку иначе им нельзя было бы передать управление (в действительности, и это не обязательно так: часто процедуры/функции подгружаются в оперативную память компьютера операционной системой по запросу программы в процессе ее исполнения). Образование активации процедуры при использовании таких языков сопровождается созданием ее локальных объектов. В этом случае возврат из вызванной процедуры сопровождается ликвидацией ее активации. Ликвидация активации процедуры сопровождается уничтожением тех ее локальных объектов, которые были созданы в момент вызова.

Для понимания того, каким образом создаются и уничтожаются объекты программы, как осуществляется доступ к их значениям и что должен обеспечивать транслятор, рассмотрим ряд «низкоуровневых» понятий, тесно связанных с семантикой языков программирования.

1.2. Адреса и значения

Исполнение программы состоит в преобразовании набора значений ее исходных данных в значения выходных данных (возможно – с использованием значений промежуточных данных).

Преобразование значений осуществляется процессором компьютера, работающим под управлением последовательности команд, выбираемых из памяти (либо из последовательно расположенных элементов памяти, либо из элементов, расположение которых определяется результатом исполнения команд передачи управления).

Значения данных, так же как и команд, хранятся в памяти компьютера и извлекаются из нее для обработки, а также сохраняются после нее с использованием адресов элементов памяти, формируемых процессором.

Пара «адрес+значение» является фундаментальным понятием применительно к процессам компьютерной обработки данных. Любой

элемент данных полностью и однозначно определяется этой парой в момент исполнения программы. В литературе адрес элемента данных принято называть *l-value*, а значение – *r-value*. Эти названия отражают положение, которое адрес и значение фактически занимают в операторе присваивания. При исполнении реальной машиной оператора присваивания (синтаксис соответствует языку С)

$$x = y;$$

наименование *x*, находящееся слева (*l[eft]*) от знака оператора присваивания, понимается как адрес того элемента памяти, в который должно быть занесено значение, поименованное как *y* и находящееся справа (*r[ight]*) от знака «=».

В языках программирования вместо адресов элементов данных используются их символические наименования. Ясно, что удобные для программиста символические имена рано или поздно, но обязательно до момента исполнения оператора, должны быть заменены численными значениями адресов, единственно приемлемыми для процессора машины при исполнении команд.

Связывание адреса с наименованием при компиляции осуществляет транслятор в процессе генерации объектного кода, распределяя память и формируя численные значения адресов. Реально транслятор даже при компиляции может сформировать только условные (так называемые относительные) адреса объектов, поскольку будущее размещение транслируемой программы в памяти компьютера при ее исполнении ему не может быть известно. Окончательное значение адреса каждого объекта в момент доступа к его значению, т. е. его физический адрес, может быть получено путем сложения относительного адреса с базовым адресом, формируемым операционной системой в момент загрузки процедуры, содержащей объект, в память компьютера.

При интерпретации связывание наименований объектов с адресами элементов памяти, предназначенных для хранения их значений, может осуществляться по-разному в зависимости от используемых методов организации структур данных и исполнения программы. Можно просто считать, что интерпретатор подставляет значения элементов данных вместо их наименований, т. е. использует имена в качестве адресов (другими словами, как имена так и значения хранятся в таблице идентификаторов, для доступа к строкам которой используется поле «номер слова в группе» из лексемы).

Не следует забывать о том, что как значения, так и адреса – это числа. Поэтому вполне возможны вычисление и использование адре-

сов во время исполнения программы. При таких вычислениях адреса, естественно, рассматриваются как значения. Более того, во многих случаях доступ к данным просто невозможен без предварительного вычисления адреса. Например, при работе с массивами перед любой выборкой значения некоторого произвольного элемента массива (или перед присвоением значения) его адрес вычисляется путем сложения адреса самого первого элемента с суммарным размером всех элементов, предшествующих требуемому.

В качестве другого примера приведем локальные (объявленные внутри функции) переменные в языке С. Местоположение в памяти (адрес) любой такой переменной не определено до момента вызова функции. В этот момент перед исполнением самого первого оператора вызванной функции «создаются» все ее локальные переменные. По существу, это означает создание так называемой записи активации функции (понятие записи или фрейма активации подробно обсуждается в п. 1.8.2) и занесение адреса этой записи в специальный регистр процессора. Для разных моментов вызова функции базовые адреса ее фреймов могут быть различными (это легко можно понять, вспомнив, что язык С допускает рекурсивный вызов функций: локальные переменные активации функции, вызванной извне, и активации, вызванной из самой функции, очевидно, должны быть различными). В процессе выполнения функции доступ к значению любой локальной переменной данной активации, естественно, требует вычисления адреса элемента памяти, занимаемого этой переменной.

Языки программирования (за исключением низкоуровневых, типа языка ассемблера), как правило, скрывают за удобными для программиста обозначениями сам факт необходимости вычисления адресов памяти перед доступом к ее содержимому.

Совокупность действий по вычислению адресов памяти принято называть адресной арифметикой. В некоторых языках программирования адресная арифметика полностью скрыта от программиста: *l-value* (адрес) не может быть присвоено в качестве *r-value* (значение) никакому элементу данных программы, и наоборот, никакое значение элемента данных, явно объявленного в тексте программы, не может быть использовано в качестве адреса. Однако существуют и такие языки программирования, которые предоставляют программисту возможность в той или иной степени явно оперировать с адресами элементов данных. Элементы данных, значения которых могут использоваться в качестве адресов других элементов памяти, принято называть указателями.

До сих пор, обсуждая адреса и значения данных, обрабатываемых программой, мы целенаправленно избегали какого бы то ни было упоминания об их смысле, т. е. о том, что они собой представляют и каким образом могут обрабатываться. Тем не менее стоило только упомянуть о том, что адресная арифметика в некоторых языках доступна программисту, но в других скрыта от него, как тут же пришлось описывать, каким образом и для чего могут использоваться указательные значения.

Определение смысла данных в языках программирования производится с использованием понятия их типа. В любом языке программирования существуют так называемые базовые, или примитивные, типы данных и некоторый набор способов конструирования новых – производных – типов на основе уже существующих.

1.3. Базовые типы данных

Перечень базовых типов и их свойства, как правило, полностью определяются стандартом языка программирования, хотя некоторые детали могут определяться аппаратной платформой и конкретной реализацией транслятора. Количество базовых типов данных в любом языке программирования обычно очень ограничено. Так, например, в языке C существует всего три базовых типа данных – символьный (*char*), целый (*int*) и вещественный (*float*). Для целого типа определены три разновидности, не обязательно различающиеся по диапазону значений: короткий (*short*), обычный и длинный (*long*), для вещественного – две разновидности: обычный и двойной точности (*double*). Кроме того, символьные и целые значения могут быть либо знаковыми (*signed*, по умолчанию), либо беззнаковыми (*unsigned*).

В языке Java к аналогичному набору базовых типов данных добавлен булевский тип (*boolean*). В языке Pascal также существует булевский тип данных, но целые и вещественные типы не имеют разновидностей по диапазону значений.

Под типом элемента данных принято понимать:

- с одной стороны, его внутреннее устройство (диапазон возможных значений, размер области памяти в минимально адресуемых единицах, необходимой для хранения значения, формат значения, т. е. назначение каждой двоичной цифры – бита, и т. д.);
- с другой – перечень операций, которые могут применяться к значениям этого типа.

Большинство деталей внутреннего устройства данных (за исключением диапазона значений) обычно скрывается от программистов. Для построения транслятора (и понимания принципов его работы), наоборот, очень важны детали внутреннего строения данных. Именно с ними имеют дело процессы семантического анализа, генерации объектного кода и его оптимизации.

Внутреннее устройство объектов может быть как очень простым, так и чрезвычайно сложным. Например, объект символьного типа в языке С можно считать одним из простейших. Внутреннее представление значения такого объекта занимает минимальный адресуемый участок памяти – один байт. Диапазон значений внутреннего представления – от 0 до 255 (в десятичной системе счисления). В операциях символьное значение рассматривается как целое число без знака.

Другой пример – функция в языке С. Функция есть программная единица, возвращающая значение некоторого типа (на данный момент будем считать, что функция возвращает значение одного из базовых типов). Имя функции может быть использовано в выражении, следовательно, она является объектом программы. Каждая конкретная функция имеет свой собственный уникальный тип, внутреннее устройство которого в самых общих чертах можно описать так.

Функция – это как минимум одна область последовательно расположенных элементов памяти, содержащая исполняемые команды и отдельно расположенные области памяти для хранения адресов связи с другими функциями, значений аргументов, значений локальных переменных, значения результата. Имена функций (с фактическими аргументами) могут быть использованы в выражениях таким образом, что может возникнуть впечатление, будто к ним (функциям) применимы арифметические или иные операции. На самом деле эти операции применяются к значениям, возвращаемым объектом типа «функция». К самим же функциям, как к объектам, применима только операция вызова, т. е. передачи управления.

Информация о внутреннем устройстве объектов программы нужна транслятору для определения того, как именно должны использоваться значения объектов. Пусть, например, имеется выражение $x + y$.

Вычисление значения этого выражения может протекать поразному не только для разных сочетаний типов данных в одном языке программирования, но и в том случае, если типы данных объектов x и y одинаковы, но это выражение появляется в программах на разных языках программирования. Например, если объекты x и y имеют символь-

ный тип, то в программе на языке C/C++ будет выполняться арифметическое сложение численных эквивалентов текущих значений символов с образованием целого значения в качестве результата, а в программе на языке Object Pascal – конкатенация этих символов с образованием значения типа «массив символов» или «строка». Таким образом, результатом вычисления выражения *'0'+'A'* в программе на языке C будет целое беззнаковое число 113 (которое можно рассматривать и как символ *'q'*), а в программе на языке Object Pascal – строка *"0A"* (последовательность из двух символов *'0'* и *'A'*).

Однако знания только внутреннего строения типов данных недостаточно для того, чтобы проверять правильность программы, в которой используются объекты этих типов. Для каждого типа должен быть известен перечень операций, применимых к его значениям. Так, например, к объектам символьного типа в языке C могут применяться операции присваивания, сравнения, арифметические операции, логические (битовые) операции, но не может применяться операция передачи управления на этот символ. В языке Pascal к данным символьного типа не могут применяться битовые и арифметические операции, а могут только операции присваивания и сравнения. К функции, наоборот, может быть применена операция передачи управления (с предварительным сохранением адреса возврата), но не может применяться ни одна арифметическая, логическая или сравнивающая операция (как уже говорилось выше, не следует путать операции с функцией как объектом программы и операции со значением, возвращаемым ею в результате вызова).

Базовые типы данных определены стандартом языка во всех деталях, т. е. разработчику транслятора заранее известно их внутреннее устройство, множество применимых к ним операций, в том числе операций преобразования в другие базовые типы, преобразования из внешнего во внутреннее представление, и наоборот, из внутреннего во внешнее. Базовыми типами данных могут обладать так называемые простые переменные и константы.

Константы бывают именованными и литеральными.

Именованные константы с точки зрения решения задач семантического анализа почти полностью эквивалентны переменным. Единственное отличие состоит в том, что к именованной константе неприменима операция присваивания. Способы объявления именованных констант в разных языках различны.

Литеральными константами (или просто литералами) называются объекты, не имеющие имени, и объявленные просто в виде их значений прямо в тексте инструкции.

С литеральными константами связано несколько проблем, которые могут разными способами решаться разработчиками трансляторов, в том числе:

1) являются ли несколько текстуально одинаковых литеральных констант, встречающихся в разных точках текста программы, одним объектом времени исполнения или каждая такая константа есть самостоятельный объект, занимающий собственный элемент памяти;

2) в какой момент времени (на каком этапе трансляции) должно осуществляться отнесение каждой встреченной в тексте программы константы к тому или иному типу данных и соответственно преобразование литерала (текстового представления константы) во внутреннее представление, т. е. значение.

Единых ответов на эти вопросы не существует. От того, как разработчик языка отвечает на них, существенно зависят свойства языка и соответственно характеристики программ на нем. Проиллюстрируем суть этих проблем на небольшом примере. Пусть в программе на языке С встречается такая последовательность операторов (рис. 1.3):

```
(1)  int val;
(2)  real values[10];
(3)  real * pointer;
(4)  val=1;
(5)  pointer=values;
(6)  *pointer=1;
(7)  pointer+=1;
```

Рис. 1.3. Литеральные константы в языке С

В этой последовательности встречаются три литеральные константы с идентичным текстовым представлением 1. Казалось бы, после обработки текста лексическим анализатором, обнаружившим три вхождения целочисленной константы 1, транслятор должен каждое из этих вхождений связать с одним и тем же объектом. Однако семантические правила языка С таковы, что в операторе присваивания

```
(8)  val=1;
```

должно использоваться целое значение 1, результатом выполнения оператора

(9) **pointer=1;*

в конечном итоге должна быть запись вещественного значения 1.0 (заметим, что внутреннее двоичное представление вещественной единицы может не совпадать с представлением целого числа 1) в самый первый элемент массива *values*, а фактическое значение литеральной константы 1 в строке (7) зависит от реализации транслятора (и от аппаратной платформы) и может оказаться равным 4 или 8.

Этот простой пример показывает, что (по крайней мере в некоторых языках) установление типа данных литеральных констант и формирование внутреннего представления их значений не может выполняться на этапах лексического или синтаксического анализа. Оно должно выполняться семантическим анализатором позднее.

1.4. Производные типы данных

Производные типы конструируются программистом по правилам, определенным стандартом языка. Для разных языков эти правила различаются.

В силу того что возможность и степень удобства конструирования в точности таких типов, которые необходимы для решения каждой конкретной задачи, чрезвычайно важны при программировании, именно средства, предоставляемые языком для этих целей, во многом определяют как потенциальную применимость языка, так и его популярность. В соответствии с тем, что понимается под типом данных, существует несколько принципиально различающихся возможностей для определения производных типов (здесь перечислены только некоторые, наиболее употребляемые способы).

1. *Модификация тех или иных параметров внутреннего устройства базового типа.* Например, в языке C существует возможность определения коротких (*short*) и длинных (*long*), знаковых (*signed*) и беззнаковых (*unsigned*) вариантов обычного целого типа.

2. *Образование однородных* (*содержащих элементы одного и того же типа*) *массивов*, *требуемых размерности и границ по каждому измерению.* Идентификация элементов массива для доступа к их значениям осуществляется с помощью указания индексов (как правило, целочисленных) по каждому измерению. К элементам массивов

обычно оказываются применимы в точности те же самые операции, что и к одиночным элементам данных того же самого типа. К массивам в целом обычно оказываются применимыми только передача в качестве аргумента (параметра) процедуры или функции, возврат в качестве значения функции и (не во всех языках программирования) присваивание.

3. *Определение неоднородных совокупностей из данных разного типа.* Примеры таких совокупностей – записи (*record*) в языке Pascal, структуры (*struct*) в языке C. Идентификация элементов таких совокупностей для доступа к их значениям обычно осуществляется по составному имени, включающему имя совокупности и имя элемента. Перечень операций, применимых к элементам, полностью определяется их типами. К записям (структурам) в целом обычно оказываются применимы такие же операции, что и к массивам (передача в качестве параметров, возврат в качестве результата и присваивание).

4. *Определение процедур/функций*. Это объекты, к которым могут применяться операции вызова с передачей им аргументов требуемых типов и/или операции передачи их в качестве аргументов других процедур/функций. С другой точки зрения процедуры/функции можно рассматривать как новые операции, определяемые программистом и применяемые к значениям их аргументов.

5. *Объектно-ориентированное программирование.* Это развитие способа 4 в сочетании со способом 3 (структуры и записи), которое привело к образованию ряда базовых понятий: классов, их свойств и методов. Свойство экземпляра класса, выглядевшее в тексте программы как составное имя элемента данных, реализуется путем создания двух одноименных процедур, одна из которых предназначена для извлечения значения свойства (аналогичное понятие – *r-value*), а другая – для присваивания ему значения (аналог – *l-value*). В зависимости от того, в каком контексте употребляется имя свойства, транслятор подставляет вместо него вызов либо той, либо другой процедуры. Методы, более похожие на обычные процедуры, стали основой для определения и переопределения операций, применяемых к экземплярам классов (например, операции >> и << классов *cin* и *cout*, определенных в языке C++ для ввода и вывода данных).

6. *Создание указателя на существующий тип данных.* Не в каждом языке допускается явное программирование операций с адресами как со значениями. Как уже упоминалось, в языках, не предназначенных для системного программирования, адресная арифметика обычно пол-

ностью скрывается от программиста. Если же разрешается образовывать указательные типы данных, то опять же, как правило, к ним допускается применять только операции присваивания. И только в языках, предназначенных в основном для написания системных программ (например, в языке C), к указательным значениям оказывается возможным применение некоторых арифметических операций (сложение и вычитание с целым значением) и операций сравнения.

Для каждого из способов конструирования производных типов данных в разных языках предусматриваются различные моменты связываний атрибутов объектов с самими объектами. Выбор момента выполнения связывания, как уже упоминалось ранее, влияет как на мощность изобразительных средств языка, так и на скорость работы программ, разработанных средствами данного языка. Независимо от того, к какому типу – базовому или производному – относятся объекты программы, для каждой операции с некоторым объектом, обнаруженной в ее тексте, семантический анализатор транслятора должен иметь возможность выявления как внутреннего устройства объекта, так и применимости данной операции к его значению. Это делается на основе явных или неявных объявлений типов объектов программы и обычно называется контролем типов данных.

1.5. Контроль типов данных объектов программы

В различных языках программирования реализован широкий спектр подходов к контролю типов данных.

На одном конце спектра находятся такие языки программирования (например, язык Perl), в которых не требуется явно указывать типы используемых в программе объектов (или явное определение типов опционально, как в языке Visual Basic). В этом случае тип объекта может стать известным (определенным) только после первого присваивания ему значения во время исполнения программы. До этого присваивания тип объекта просто не определен. Последующие во времени операции с объектом могут приводить к изменению его типа. Ясно, что в таком случае семантические проверки возможности выполнения операций над значениями объектов могут быть произведены только во время исполнения программы, поскольку типы данных операндов всех или некоторых операций не известны во время трансляции. При интерпретации эти проверки будут выполняться транслятором или виртуальной машиной. Если реализуется компиляция, то семантический

анализатор должен встраивать в программу дополнительные действия для проверки возможности исполнения операций над объектами неизвестного типа, построенных по тексту транслируемой программы.

Такие действия обычно называются динамическими проверками в отличие от статических проверок, которые выполняются во время трансляции программы. Динамические проверки могут значительно увеличивать время исполнения программы независимо от того, интерпретируется она или компилируется. Необходимо отметить, что этот подход, т. е. потенциальная возможность изменения типов объектов программы «на лету», постоянно подвергается жесткой критике вследствие крайней затруднительности разработки безопасных программ (программ, не содержащих трудно обнаруживаемые ошибки). Тем не менее основанные на таком подходе языки программирования существуют и продолжают развиваться.

Другой крайней точкой рассматриваемого спектра подходов к контролю типов данных является так называемая строгая (сильная) типизация. Под строгой типизацией понимается выполнение следующей совокупности требований к программе:

– для каждого используемого в программе типа должно быть известно множество значений (т. е. внутреннее устройство) и множество применимых к ним операций;

– каждый объект программы должен иметь однажды определенный и неизменяемый тип;

– при выполнении любого присваивания значения объекту тип присваиваемого значения должен быть эквивалентен типу объекта;

– использование значения объекта допускается только в качестве операнда операций, допустимых для данного типа.

К строгой типизации стремились разработчики языков Pascal, Ada, Java и ряда других, менее известных.

В случае строгой типизации подавляющая часть полного набора семантических проверок возможности применения операций к объектам может быть выполнена в процессе трансляции программы, т. е. статически. Увы, абсолютно все проверки реализовать во время трансляции нельзя, достаточно вспомнить операцию деления на неизвестное транслятору значение, которым при исполнении программы может оказаться ноль. Первичная информация для проверок извлекается семантическим анализатором из операторов объявления данных и сохраняется в таблице идентификаторов в качестве атрибутов объектов. Как часть общего текста программы эти операторы вначале обрабатываются лексическим

и синтаксическим анализаторами и могут преобразовываться в постфиксную форму записи.

Большое количество языков программирования, в том числе такие популярные (по крайней мере, в свое время), как Algol-60, Fortran, C и многие другие, занимают то или иное промежуточное положение в этом спектре, предоставляя, с одной стороны, полную информацию о типах и возможность (но необязательность в Fortran'e) явного единственного объявления типа каждого объекта в рамках одной программной единицы, но с другой – возможности нарушения двух последних требований строгой типизации. Так, например, в каждом из вышеперечисленных языков допускается возможность присваивания значения одного типа объекту другого типа с подразумеваемым, т. е. неявно выполняемым, преобразованием. В языках Fortran и C существует возможность выполнения операций, не применимых к значениям данного типа за счет неконтролируемой подстановки – присваивания значения объекту другого типа (такие возможности предоставляют объявления *common, equivalence* в Fortran и *union* в C). Любое нарушение правил строгой типизации – потенциальная причина появления в программе ошибок, которые иногда чрезвычайно трудно обнаружить. Именно за это языки, не гарантирующие строгую типизацию, часто подвергаются суровой критике. Вместе с тем требования строгой типизации заметно сужают область применения языка программирования. Системные программы, такие как операционные системы и их утилиты, очень трудно разрабатывать, соблюдая абсолютно все требования строгой типизации.

Явные объявления типов данных, с одной стороны, позволяют обеспечивать более или менее строгий контроль применимости знаков операций к операндам во время трансляции, с другой – порождают ряд проблем, которые так или иначе должны быть разрешены разработчиками языка программирования и трансляторов для него. Потенциальная возможность неоднократного (намеренного или случайного) объявления типа данных (одного и того же или разных) для одного и того же наименования порождает первую проблему: разрешать такую возможность или запрещать ее. Запрет множественных объявлений разных объектов с одним именем в пределах одной транслируемой программы в принципе возможен, но обычно считается слишком жестким ограничением и практически не применяется. В том случае, когда такие объявления разрешаются стандартами языка, ими же должны быть определены правила, позволяющие любое использование наименова-

ния в тексте программы отождествить с единственным объектом (либо обнаружить и диагностировать ошибку). Такие правила можно определить на основе понятия ассоциации (см. разд. 1.7) и использовать их во время трансляции для определения того, к какому именно объекту будет применяться операция во время исполнения программы.

Исполнение программы есть последовательное исполнение операций, преобразующих значения своих операндов в значения, которые будут обрабатываться другими операциями. Для любой операции, не позже чем к моменту ее фактического исполнения, должны быть установлены возможность ее применения к значениям операндов и способ обработки значений. Для примера вспомним, что складывать целые числа, целое и вещественное, целое и указательное и так далее нужно по-разному. Выражение $a+b$ должно быть преобразовано в разные машинные команды для различных сочетаний типов данных a и b. Для выявления смысла каждой операции в общем случае нужно выполнить следующую последовательность действий:

– определить количество операндов (напомним, что одно и то же обозначение в принципе может применяться для разных операций: скажем, знак «–» обычно используется и для обозначения бинарной операции вычитания и для обозначения унарной операции изменения знака числа);

– определить тип каждого операнда;

– проверить, допустим ли этот тип к данной позиции операнда в операции путем сравнения типов;

– на основании результатов всех проверок принять решение, может ли быть выполнена данная операция и если может, то как.

Предполагая, что задачи определения количества операндов и их типов решены (первая может считаться тривиальной, а вторая обычно решается путем выборки атрибутов объекта из таблицы идентификаторов), сосредоточимся на способах решения задачи сравнения типов. Под сравнением будем понимать только установление эквивалентности или неэквивалентности двух типов.

1.6. Эквивалентность типов данных

Для описания типа языковой конструкции будем использовать понятие «выражение типа». Выражение типа, как и любое выражение, имеет вычисляемое значение, которым может являться либо базовый, либо производный тип. В том случае, если тип является производным,

значение выражения типа должно быть построено с помощью применения оператора, называемого конструктором типа, к другим выражениям типа. Множества базовых типов и конструкторов производных типов зависят от проверяемого языка (см. разд. 1.4).

В некоторых языках программирования типам могут даваться имена. Например, во фрагменте программы на языке Pascal, представленном на рис. 1.4, идентификатор *link* объявлен в качестве имени типа ^*cell* (указатель на ячейку).

```
(1)  type link = ^cell; var next : link;
(2)  last : link;
(3)  p : ^cell; q, r :^cell;
```

Рис. 1.4. Имена типов в языке Pascal

Для реализации последующих операций важно знать, идентичны ли типы переменных *next*, *last*, *p*, *q* и *r*? Ответ на этот вопрос зависит от реализации, поскольку в официальном сообщении о Pascal никак не определяется идентичность типов.

В качестве базовых будем использовать типы ***void***, ***boolean***, ***char***, ***integer***, ***real***, ***pointer*** и ***error***. Базовый тип ***void*** означает «отсутствие значения» и может быть использован для проверки операций как объектов. Специальный базовый тип ***error*** будет использоваться для обозначения ошибок, которые могут быть обнаружены в процессе проверки типов.

Правила проверки типов обычно имеют следующий вид.

Если два выражения типов равны, то вернуть тип любого из сравниваемых выражений, иначе вернуть тип ошибки ***error***.

Для реализации правил проверки очень важно иметь точное определение равенства двух типов. Потенциальная неоднозначность возникает с именами выражений типа, которые затем используются в последующих выражениях типа. Ключевой вопрос обычно состоит в том, является ли имя в выражении типа само по себе выражением или сокращением для другого выражения типа.

Для обеспечения высокой эффективности работы транслятора требуется использовать такие представления выражений типа, которые позволяют быстро определять, являются ли сравниваемые типы эквивалентными. Представление типов может быть либо именным, либо структурным, либо кодированным.

1.6.1. Именное представление и сравнение типов

Именная эквивалентность проверяется путем:

– извлечения текстового представления имени типа из программы или формирования его посредством применения конструктора типа;

– приведения этого представления к стандартному виду (замена возможных сокращений, удаление незначащих пробелов, добавление всех возможных скобок и т. д.);

– сравнения двух текстовых представлений как обычных строк.

При применении этого способа к фрагменту программы на рис. 1.4 будут получены имена выражений типов переменных, приведенные в табл. 1.4.

С точки зрения эквивалентности имен переменные *next* и *last* имеют один и тот же тип, поскольку с ними связаны одинаковые выражения типа. Типы переменных *p*, *q* и *r* также эквиваленты, но, например, переменные *p* и *next* разнотипны, поскольку связанные с ними имена выражений типа различны.

Таблица 1.4

next	*link*
last	*link*
p	***pointer****(cell)*
q	***pointer****(cell)*
r	***pointer****(cell)*

При именном представлении каждое уникальное имя представляет собственный тип, отличный от любого типа с другим именем.

1.6.2. Структурное представление и сравнение типов

Концепция структурной эквивалентности основывается на представлении выражений типа в виде графов с листьями для базовых типов и внутренними узлами для конструкторов типов. При этом рекурсивно определенные типы приводят к появлению циклов в таком графе. Сравнивая структурное и именное представления, можно считать, что в графе имена типов заменяются выражениями типа, определяемыми этими именами. Следовательно, два выражения типа структурно эквивалентны, если представить себе, что в их текстовых представлениях все имена

развернуты вплоть до базовых типов. На рис 1.5 показан фрагмент графа структурного представления типов для примера программы, приведенного на рис. 1.2.

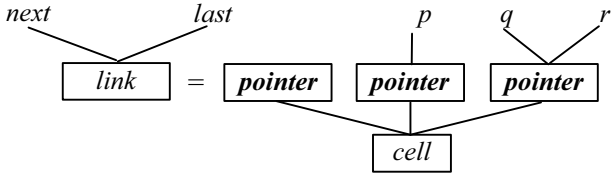


Рис. 1.5. Связь переменных и узлов в графе типов

С точки зрения структурной эквивалентности все пять переменных в рассматриваемом примере имеют один и тот же тип, поскольку *link* представляет собой не что иное, как имя для выражения типа ***pointer****(cell)*.

Типичная реализация функций контроля типов состоит в построении для представления типов соответствующего графа. Всякий раз, когда в объявлении объекта встречается конструктор типа или имя типа, создается новый узел, а при появлении имени базового типа – новый лист. При использовании такого представления два выражения типа считаются эквивалентными, если они представлены одним и тем же узлом в графе типов.

Многие полезные структуры данных, такие как связанные списки и деревья, зачастую определяются в программах рекурсивно, например связанный список либо пуст, либо состоит из ячейки с указателем на связанный список. Такие структуры данных обычно реализуются с использованием записей, содержащих указатели на точно такие же записи. Это создает определенные проблемы для решения задачи проверки эквивалентности типов структурными методами.

Заметим, что именно поэтому при именном представлении типов не делается попыток развернуть все имена вплоть до базовых типов, поскольку применение такого развертывания к рекурсивно определенным типам привело бы к зацикливанию и бесконечному росту текстового представления.

Рассмотрим связанный список ячеек, каждая из которых содержит некоторую целочисленную переменную и указатель на следующую ячейку в списке (пример объявления ячейки списка на языке Pascal приведен на рис. 1.6,*а*, его аналог на языке C – на рис. 1.6,*б*).

```
(1)  type link = ^ce11;
(2)  cell = record
(3)  Info : integer;
(4)  next : link; end;
```

```
(1)  struct cell {
(2)  int info;
(3)  struct cell * next;
(4)  };
```

Рис. 1.6. Рекурсивное определение типов:
а – в языке Pascal; *б* – в языке C

В примере на Pascal имя типа *link* определено с помощью имени *cell*, а имя типа *cell* – с помощью *link*; так что эти два определения рекурсивны.

Рекурсия в определении типов приводит к появлению циклов в графе их структурного представления, как показано на рис. 1.7, *а*.

С точки зрения проверки эквивалентности циклы в графах создают проблемы, аналогичные проблемам при именном представлении. Поэтому от циклов обычно стремятся избавиться. Сделать это можно только искусственным образом, условно (с введением дополнительных атрибутов вершин графа) превращая рекурсивно определенный тип в аналог базового. Направленный ациклический граф, полученный в результате такого преобразования, показан на рис 1.7, *б*.

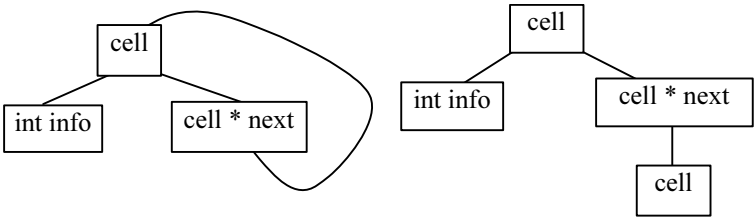


Рис. 1.7. Рекурсивно определенное имя типа *cell*:
а – циклический граф; *б* – ациклический граф

Структурная эквивалентность типов, основанная на направленных ациклических графах, определяется так: два выражения типов либо представляют собой один и тот же базовый тип, либо сформированы путем применения одного и того же конструктора к структурно эквивалентным типам. Например, выражение типа *integer* эквивалентно только *integer*, потому что это один и тот же базовый тип. Аналогично тип *pointer(integer)* эквивалентен только типу *pointer(integer)*, поскольку оба построены посредством применения одного и того же конструктора pointer к эквивалентным типам.

Структурная эквивалентность типов в направленных ациклических графах может быть проверена с использованием рекурсивного алгоритма, параметрами которого являются две заданные вершины *A* и *B*:

– если *A* и *B* есть один и тот же лист (базовый тип), вернуть истину;

– иначе, если и *A* и *B* есть конструкторы, последовательно обойти все пары дуг, исходящие из *A* и *B*, и:

- если найдется хотя бы одна непарная дуга, вернуть ложь;
- или, если хотя бы одна пара вершин, в которые ведут эти дуги, не эквивалентна (для каждой такой пары применяется этот же алгоритм), вернуть ложь;
- иначе вернуть истину.

Свойства языка программирования сильно зависят от того, какие параметры конструкторов типов считаются важными при формировании направленного ациклического графа. Например, если (как в языке Pascal) границы изменения индексов по каждому измерению однородных массивов считаются частью конструктора типа, то тип массива из двух целочисленных элементов не будет эквивалентен типу массива, содержащего три целых числа, и т. д. В результате для программистов практически непреодолимой трудностью была разработка на этом языке процедур/функций для обработки массивов с заранее неизвестным количеством элементов или измерений.

В некоторых реализациях трансляторов для выражений типов стремятся найти значительно более компактную по сравнению с графом запись. В следующем пункте (на основе компилятора с языка C, созданного Д. Ритчи) информация о выражении типа закодирована последовательностью битов, а значит, может интерпретироваться как целое число. Кодирование осуществляется таким образом, что различные числа представляют структурно неэквивалентные выражения типов. Этот подход может использоваться для существенного ускорения проверки структурной эквивалентности и в более сложных языках, в которых полная проверка эквивалентности типов не может быть осуществлена путем кодированного представления. Ускорение может быть достигнуто за счет того, что сначала проверяется неэквивалентность путем сравнения целых (но не полных для некоторых конструкторов) представлений типов, и только в случае их равенства применяется приведенный выше алгоритм структурной проверки.

1.6.3. Кодирование выражений типа

В самой первой реализации языка С его базовые типы кодировались с использованием четырех битов.

Два младших бита обозначают базовые типы: 00 – *char*, 01 и 10 – *integer,* 11 – *real*.

Следующие два бита (которые маскируются при большинстве проверок эквивалентности) используются для хранения значений модификаторов длины (*short* или *long*) и наличия знака (*signed*/*unsigned*).

Для конструкторов типов (в этом примере рассматриваются только указатели, массивы и функции) используются каждые два следующих бита, обозначающие:

00 – отсутствие конструктора;

01 – указатель (*pointer)* на тип *t*, закодированный левее;

10 – массив (*array*) неопределенной длины элементов типа *t*;

11 – функция, возвращающая объект типа *t* (*freturns*).

Размерности и границы изменения индексов массивов, как и количество, и типы аргументов функций, совершенно не учитываются в таком кодировании, их приходится хранить вне кода типа (скорее всего, в таблице идентификаторов, в которой хранится и обсуждаемый код типа).

Таким образом, объекты, структурно эквивалентные с точки зрения целочисленного кодирования, могут быть неэквивалентны с точки зрения полной проверки эквивалентности. Поскольку каждый из этих конструкторов представляет собой унарный оператор, выражения типа, образованные применением этих конструкторов к базовым типам, имеют весьма однородную структуру. Приведем примеры выражений типа в тексте программы (первый столбец), во внутреннем представлении транслятора (второй столбец) и соответствующих им кодов (третий столбец):

int	*int*	000000 0001
int function(...)	*freturns(int)*	000011 0001
**(int function(...))*	*pointer(freturns(int))*	000111 0001
((int function(...)))[...]*	*array(pointer(freturns(int))*	100111 0001

Такое представление чрезвычайно компактно (по сравнению и с именным, и со структурным) и отслеживает последовательность применения конструкторов, появляющихся в любом выражении типа. Две

различные последовательности битов не могут представлять один и тот же тип, поскольку при этом различны либо базовые типы, либо примененная к ним последовательность конструкторов. Тем не менее разные типы могут иметь одну и ту же последовательность битов в силу того, например, что размеры массивов и аргументы функций не представлены в данной системе кодирования.

Кодирование из этого примера в принципе может быть расширено для включения типов записей и объединений (*struct* и *union*). При этом каждая запись при кодировании рассматривается как базовый тип; но отдельная последовательность битов кодирует тип каждого поля записи.

При использовании кодированного представления типов задача выявления эквивалентности резко упрощается, так как вместо сравнения имен типов (с предварительным приведением к каноническому представлению) или рекурсивной обработки вершин графа надо всего лишь сравнивать числа, поставленные в соответствие именам типов.

1.7. Ассоциации наименований объектов

Выше уже упоминалось, что с каждым наименованием в процессе трансляции и исполнения программы может быть ассоциировано несколько объектов. Ассоциацию можно представить в виде пары, состоящей из наименования и соответствующего ему элемента или указателя на этот элемент. В различных реализациях языков могут использоваться разные представления для ассоциаций идентификаторов.

Каждая конкретная ассоциация идентификатора (скажем, *X*) за время своей жизни обычно претерпевают следующие изменения.

1. *Создание.* Существование декларации для *X* внутри программы (процедуры) является причиной создания первоначальной ассоциации для *X* в начале выполнения программы или при входе в процедуру.

2. *Использование.* Употребление *X* в программе приводит к порождению (при трансляции) и последующему исполнению операции обработки ссылки, в которой созданная для *X* ассоциация используется для доступа к поименованному элементу программы или данных.

3. *Деактивирование.* При передаче управления в другую программную единицу (вызове другой или той же самой процедуры) ассоциация для *X* может переходить в неактивное состояние, в котором она продолжает существовать, но ее нельзя использовать. Часто перевод ассоциаций в неактивное состояние происходит при вызовах подпрограмм.

4. *Повторное активирование* ассоциации происходит в результате возврата управления из другой процедуры, после чего эта ассоциация вновь может быть использована для доступа. Деактивирование и активирование любой ассоциации может происходить многократно.

5. *Уничтожение ассоциации.* Ассоциация для *X* уничтожается, идентификатор (именующий именно этот объект) не может быть более использован.

6. *Повторение шагов 1–5*. Для имени *X* может быть создана совершенно новая ассоциация со своим жизненным циклом.

Чтобы ассоциировать идентификатор с элементом данных или элементом программы, необходимо выполнить операцию, называемую именованием. Именование должно предшествовать ссылке, так как операция именования создает ассоциацию для идентификатора, тогда как операция обработки ссылки использует эту ассоциацию для поиска объекта, связанного с конкретным идентификатором. Операцию уничтожения ассоциации для идентификатора можно было бы назвать разыменованием, однако в этот термин в программировании обычно вкладывается совершенно другой смысл: замена имени значением при вычислениях.

Отметим, что именование объекта не совпадает тождественно с его созданием. Обе эти операции являются связываниями и часто выполняются совместно (или, по крайней мере, так представляется). Декларация *real X*, например, указывает и на то, что необходимо создать простую переменную, и на то, что ее именем будет *X* (т. е. с нею следует ассоциировать идентификатор *X*). Аналогично декларация *char S[100]* создает массив для хранения 100 символьных значений и именует его идентификатором *S* (заметим, что моменты времени именования и выделения памяти для массива могут совпадать, но могут и различаться). Однако можно привести пример, когда создание не сопровождается именованием: в результате вызова функции *malloc(100)* создается массив точно такого же размера и возвращается указатель на него. Однако созданный массив не именуется создающей его функцией *malloc*. С другой стороны, именование без создания встречается, например, когда в вызываемой процедуре идентификатор формального параметра ассоциируется с соответствующим объектом данных, представляющим собой фактический параметр.

Точно так же, как создание и именование, можно разделить операции уничтожения структуры данных и разыменования.

Операции активирования и деактивирования ассоциаций идентификаторов часто путают с операциями именования и разыменования. Чтобы ассоциация могла использоваться в ссылках, она должна существовать (как результат операции именования) и быть активной. Неактивные ассоциации – это те, что существуют, но временно не могут использоваться в ссылках.

Типичные примеры реализации описанного механизма можно найти в таких языках, как C++, Algol-60, …

В программах на этих языках в любом блоке могут быть объявлены переменные, значения которых должны существовать только начиная с момента входа в блок и до момента выхода из него. Каждая декларация в блоке выполняет функцию именования, создавая новую ассоциацию между идентификатором и объектом, т. е. простой переменной, массивом, подпрограммой и т. д. Одновременно деактивируется всякая прежде существовавшая ассоциация для декларируемых идентификаторов.

Ссылки внутри блока пользуются этими вновь созданными активными ассоциациями для каждого идентификатора. Выход из блока вызывает разыменование ассоциаций, созданных при входе, и повторное активирование ассоциаций, которые при входе в этот блок были деактивированы.

Итак, на самом деле с такими простыми вещами, как объявление объекта (декларация) и его использование (ссылка по имени), связаны пять основных операций, вытекающих из факта существования ассоциаций идентификаторов.

1. *Именование*: создание ассоциации между идентификатором и объектом программы или данных.

2. *Уничтожение*: ликвидация ассоциации между идентификатором и ранее ассоциированным с ним объектом.

3. *Активирование*: приведение в активное состояние уже существующей ассоциации между идентификатором и объектом программы или данных, делающее именно эту ассоциацию пригодной для использования в ссылках.

4. *Деактивирование*: операция, делающая невозможным использование существующей ассоциации идентификатора с объектом.

5. *Доступ*: любое использование (выборка или присваивание значения для данных, передача управления для программных объектов) объекта, который ассоциирован в текущий момент с данным идентификатором. При доступе может быть использована единственная активная в этот момент ассоциация для данного идентификатора.

Под областью действия объявления объекта принято понимать всю ту совокупность участков текста программы (или нескольких программ), в пределах которой с данным наименованием ассоциируется один и тот же объект. Говорят также, что объект «виден» из любой точки его области действия. Могут существовать такие точки или участки текста программы, из которых объект «не виден» и в которых, следовательно, он не может быть использован. Соответственно для любой точки текста могут считаться определенными два множества объектов – множество «видимых» и множество «невидимых». В разных языках программирования реализуются различные правила определения областей действия и соответственно текстуальной видимости объектов.

Во многих языках допускается существование нескольких одновременно существующих пространств ассоциаций. Например, в одном помеченном операторе присваивания на языке C++:

X:X.X=ClassName::X();

встречаются ссылки на *X* как на метку оператора (первое вхождение), структуру, объединение или класс (второе вхождение), элемент этой структуры (третье вхождение) и метод класса (четвертое вхождение). Если в языке допускаются подобные возможности, то его синтаксис и семантика должны предоставлять транслятору точный способ определения того, к какому именно пространству ассоциаций относится каждое конкретное вхождение наименования *X*.

«Размножение» объектов, т. е. образование новых ассоциаций для одних и тех же наименований локальных данных, может происходить еще и в результате рекурсивного вызова процедур/функций в процессе исполнения программы.

Вся совокупность объектов, значения которых доступны из произвольной точки текста программы, в литературе называется средой ссылок. Различают два понятия – текстуальная среда ссылок (или среда ссылок периода трансляции, иногда ее называют статической, или лексической) и динамическая, или среда ссылок периода исполнения программы. Задача транслятора – спроецировать текстуальную среду ссылок на среду ссылок периода исполнения.

Строение среды ссылок периода исполнения и способ отображения на нее текстуальной среды определяется тем, как разработчики языка отвечают на следующие вопросы.

1. Допускаются ли рекурсивные вызовы процедур?
2. В какой момент создаются локальные объекты процедур?

3. Что происходит с локальными переменными при возвращении управления из процедуры?

4. Может ли процедура обращаться к нелокальным объектам, не являющимся ее фактическими аргументами?

5. Каким образом в процедуру передаются параметры?

6. Может ли процедура быть передана в качестве параметра?

7. Может ли процедура быть возвращена в качестве результата?

8. Может ли память выделяться динамически по явному запросу программы?

9. Должна ли динамически выделенная память освобождаться также по явному запросу?

Возможный спектр ответов на эти вопросы очень широк. В последующих разделах будут рассматриваться в основном типичные решения и вкратце обсуждаться последствия других возможных ответов на данные вопросы.

1.8. Среды ссылок периода исполнения

Среду ссылок периода исполнения будем рассматривать применительно к компиляторам. Интерпретаторы моделируют программно те же самые процессы, которые реализуются аппаратно-программным способом при исполнении скомпилированной программы.

Будем считать, что для исполнения программы операционная системы компьютера выделяет один связный блок элементов памяти. Способ выделения этого блока (будем называть его памятью задачи) и его отображения на физическую память компьютера может быть различен для разных операционных систем и тем более для разных аппаратных платформ. Если программа запускается неоднократно, то вполне вероятно, что при разных запусках место расположения памяти задачи в физической памяти компьютера будет различным. Ясно, что в этих условиях транслятору не может быть известно будущее местоположение программы в памяти. Известными могут быть:

– максимально возможный размер памяти задачи;

– ограничения на использование некоторых фрагментов этой памяти, занятых модулями поддержки периода исполнения;

– аппаратные особенности (направление роста аппаратного стека, количество байт, извлекаемых из памяти за одно обращение и соответственно рекомендации по выравниванию объектов и т. д.).

Некоторые возможные варианты внутреннего устройства памяти задачи для разных программно-аппаратных платформ представлены на рис. 1.8 предполагается, что адреса памяти возрастают по направлению сверху вниз).

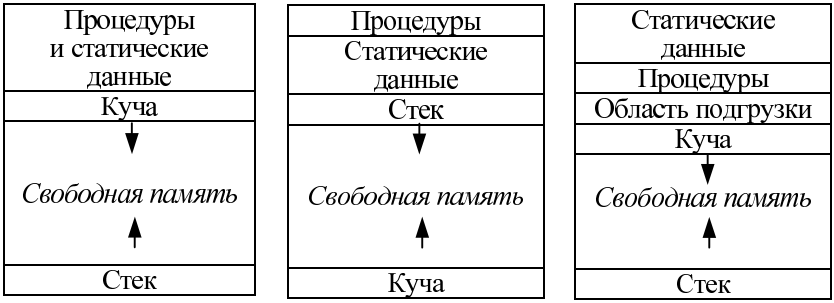


Рис. 1.8. Варианты размещения совокупностей объектов в памяти задачи

Здесь под процедурами понимается совокупность исполняемых инструкций, под областью подгрузки – фрагмент памяти, выделенный для динамически подгружаемых в процессе исполнения программы библиотек (тоже совокупностей процедур), под кучей – специально организованная область памяти, блоки которой выделяются/освобождаются динамически по явным запросам из программы. Статические данные – совокупность объектов, существующих в течение всего времени работы программы (в некоторых языках в явном виде возможность использования статических данных может отсутствовать). Аппаратно реализуемый стек используется многопланово. В частности, именно в стеке, как правило, размещаются локальные данные процедур.

Организация памяти образа задачи, представленная на рис. 1.8 (все варианты), предполагает, что память времени исполнения состоит из единого непрерывного блока, выделенного программе при запуске. При этом суммарный размер стека и кучи, обычно растущих навстречу друг другу в процессе исполнения программы, должен быть достаточно большим, чтобы эти две области никогда не пересекались. Столкновение границ стека и кучи (т. е. пересечение этих областей) обычно приводит к аварийному завершению программы, поскольку для ее выполнения оказалось недостаточно отведенной системой памяти.

Для обнаружения такой ситуации предусматриваются программные и/или аппаратные средства контроля значений границ стека и кучи.

В этих условиях при трансляции логично и естественно считать, что адресация памяти задачи для транслятора начинается с нуля и что процессор компьютера обеспечивает эффективное отображение такой памяти на любой предоставленный операционной системой блок. Именно на такие потребности запуска и исполнения программ и ориентированы базируемые и индексируемые способы адресации памяти современных процессоров, обеспечивающие «на лету» модификацию адресов, сформированных трансляторами, в реальные адреса памяти.

Заметим, что термин «реальные» здесь относится только к адресам памяти внутри образа выполняемой программы. Фактическая система адресации памяти может быть значительно сложнее и предусматривать страничную, сегментную либо странично-сегментную организацию виртуальной памяти, при которой последовательно расположенным областям (страницам или сегментам) образа памяти ставятся в соответствие участки физической памяти компьютера, выделяемые операционной системой по мере необходимости и возможности.

При этом отнюдь не обязательно, чтобы весь образ программы в течение всего времени ее исполнения присутствовал в оперативной памяти. Делается это «прозрачным» для исполняемых программ образом и поэтому на логику работы трансляторов влияния не оказывает.

В том случае, когда язык программирования сконструирован для совместной трансляции, компилятор может распоряжаться всеми известными ему в пределах памяти задачи свободными участками по собственному усмотрению (естественно, с учетом ограничений и особенностей аппаратной платформы) для размещения всей совокупности объектов программы: процедур и обрабатываемых ими данных. Задачи компилятора при раздельной трансляции могут показаться несколько более сложными, в связи с тем что в этом случае ему уже не может быть известно будущее размещение транслируемых частей единой программы внутри памяти задачи. В действительности все сложности по размещению раздельно транслируемых частей программы в одной линейно организованной памяти задачи выпадают на долю других компонентов программного обеспечения: либо редактора связей (другие названия: линковщик, компоновщик, сборщик, …), либо загрузчика операционной системы. Компилятор же, как и в случае совместной трансляции, обеспечивает размещение объектов транслируемых частей программы в линейном образе памяти. Некоторые детали этого процесса будут рассмотрены в главе 2.

Здесь мы отметим только, что размеры сгенерированных при трансляции последовательностей инструкций каждой процедуры становятся известны во время компиляции, так что компилятор может разместить их в образе памяти задачи при генерации объектного кода. Под размещением здесь пока что понимается просто связывание инструкций и адресов элементов памяти задачи, отводимых компилятором для хранения этих инструкций.

Аналогично инструкциям формируются и связываются с памятью и такие объекты данных, которые должны в единственном экземпляре существовать в течение всего периода исполнения программы. Совокупность таких объектов названа статическими данными. Одной из причин стремления к статическому выделению памяти для как можно большего количества объектов данных является то, что адреса этих объектов могут быть использованы прямо при компиляции для формирования использующих эти объекты инструкций. Ниже мы увидим, что для преобразования операторов исходной программы, работающих не со статическими объектами, в инструкции целевой машины приходится формировать дополнительные структуры данных и специальные последовательности машинных команд для их создания и обработки. Это приводит к появлению накладных расходов как памяти, так и затрат процессорного времени при исполнении программы. В некоторых языках, в частности в языке Fortran, память для всех объектов данных может быть выделена статически, т. е. доля таких накладных расходов равна нулю. Однако цена этого «преимущества» – запрет рекурсивных вызовов процедур – может оказаться слишком большой.

В тех языках, которые предполагают возможность рекурсивных вызовов процедур, неизбежно появляются и не статические данные – локальные данные активаций процедур и некоторые другие объекты, необходимые для реализации доступа из процедур к не принадлежащим им данным. Понятие активации процедуры необходимо рассмотреть во всех деталях.

1.8.1. Активация процедуры

Активацией процедуры называется процесс ее выполнения в результате одного конкретного вызова. Этот процесс включает следующие действия (возможно, не все из перечисленных ниже и не обязательно выполняемые в приведенном порядке).

1. Формирование адреса точки входа в вызываемую процедуру (в том случае, если процедура является частью динамически загружае-

мой библиотеки и этой библиотеки еще нет в памяти выполняемой программы, может выполняться предварительное обращение к операционной системе с целью ее подгрузки).

2. Формирование значений, которые должна обработать вызываемая процедура, – фактических параметров. Обеспечение доступности этих значений для инструкций вызываемой процедуры.

3. Формирование и сохранение адреса возврата – адреса той инструкции вызывающей процедуры, которая должна получить управление в момент завершения выполнения вызываемой процедуры.

4. Обеспечение возможности доступа к значениям локальных объектов текущей активации вызывающей процедуры (и, возможно, процедур, находящихся еще раньше в цепочке вызовов) из вызываемой процедуры, если такая возможность предусмотрена языком программирования.

5. Сохранение содержимого регистров процессора, необходимых для возобновления правильного выполнения текущей активации вызывающей процедуры.

6. Собственно вызов процедуры, т. е. передача управления в ее точку входа.

7. Создание локальных объектов (выделение памяти для их размещения).

8. Выполнение процедуры, т. е. обработка значений фактических параметров, протекающая путем:

- формирования и использования значений локальных объектов текущей активации;
- формирования и последующего использования результатов промежуточных вычислений;
- использования/модификации локальных объектов других существующих активаций этой же или других процедур;
- использования/модификации значений статических данных программы.

9. Формирование результата–значения, возвращаемого из процедуры функции. Обеспечение доступности этого значения для инструкций вызывающей процедуры (как правило, только для той инструкции, к которой возвращается управление и которая должна обработать или сохранить результат вызова).

10. Уничтожение объектов, созданных исключительно для выполнения данной активации процедуры.

11. Восстановление содержимого ранее сохраненных регистров.

12. Возврат из процедуры – передача управления в ту точку вызывающей процедуры, адрес которой сформирован на шаге 1.

Заметим, что в тексте исходной программы (имеются в виду, естественно, языки высокого уровня) в явном виде обычно присутствует только вызов процедуры и ее описание (определение типов параметров и типа возвращаемого значения) или объявление – то же самое описание, за которым следует тело. Все вышеперечисленные действия подразумеваются, но явно не указываются.

Задача транслятора – обеспечить выполнение этих действий. От того, каким образом будут реализовываться эти действия в процессе исполнения программы, зависит и то, как должны выполняться семантические проверки во время трансляции.

Некоторые из описанных выше действий (например, 1–3, 6) могут быть выполнены только инструкциями вызывающей процедуры. В свою очередь, действия 8, 9, 12 могут быть выполнены только инструкциями вызываемой процедуры. Остальные действия (4, 5, 7, 10, 11) в принципе могут выполняться инструкциями как вызывающей, так и вызываемой процедур.

В разных языках (и даже в разных реализациях одного и того же языка) могут приниматься различные решения по отнесению этих действий к вызывающей или вызываемой процедуре.

1.8.2. Запись активации процедуры

Данные, необходимые только для однократного исполнения процедуры, обычно размещаются в одном связном блоке памяти и называются записью активации этой процедуры. Запись активации может содержать следующие поля, показанные на рис. 1.9.

Локальные данные
Результаты промежуточных вычислений
Регистры процессора
Связь по доступу (необязательная)
Связь по управлению (необязательная)
Фактические параметры
Возвращаемое значение

Рис. 1.9. Состав записи активации процедуры

Не все поля этой структуры реализуются для любого языка. Например, если функции не могут содержать определения других функций, то поле связи по управлению никогда не понадобится. Такие поля на рисунке отмечены как необязательные.

Для последующего рассмотрения важно знать, в какой момент и каким образом может быть определен размер всей записи активации в целом, а следовательно, размер каждого из ее полей. Поэтому кратко рассмотрим назначение и соответственно способ формирования содержимого записи активации.

1. *Локальные данные.* Здесь в течение времени жизни данной активации процедуры хранятся значения объектов, объявленных в ее теле. Размер этого поля, как правило, может быть вычислен во время трансляции вызываемой процедуры. Существуют языки программирования, в которых возможно исключение из этого правила, допускающие возможность объявления в процедуре локальных массивов, размеры которых зависят от значений фактических параметров. Способы определения размеров полей локальных данных и методы выполнения связываний имен объектов с их адресами внутри этих полей для разных языков детально рассматриваются в разд. 1.9, здесь же отметим, что при возможности их значения желательно размещать в регистрах процессора на время выполнения процедуры.

2. *Результаты промежуточных вычислений.* В том случае, если в инструкциях вызываемой процедуры имеются сложные выражения наподобие *a*b+c*d*, возникает необходимость временного хранения значения произведения *a*b* (или *c*d,* или и того и другого) от момента его вычисления и, по крайней мере, до момента последнего использования. Такие данные во многом эквивалентны локальным данным процедуры, но не имеют никакого имени, присвоенного программистом в тексте программы. В процессе трансляции по мере необходимости образования объектов для хранения промежуточных результатов имена для них могут формироваться и использоваться семантическим анализатором и генератором объектного кода/виртуальной машиной. Как будет показано ниже, перечень и типы данных объектов, необходимых для хранения промежуточных результатов, могут быть определены транслятором в процессе преобразования постфиксной записи в последовательность тетрад/триад. Таким образом, размер этого поля тоже может быть вычислен во время трансляции вызываемой процедуры.

3. *Регистры процессора.* В этом поле сохраняется состояние регистров процессора, каким оно было непосредственно перед вызовом

процедуры и соответственно каким должно быть сразу после возврата из нее. В частности, здесь же обычно хранится адрес возврата, поскольку он является значением одного из регистров, а именно – счетчика команд. Размер памяти, необходимой для сохранения значений регистров, определяется архитектурой компьютера и фиксируется в качестве константы на этапе разработки транслятора.

4. *Связь по доступу.* Используется для обеспечения доступа из вызываемой процедуры к нелокальным данным, хранящимся в другой записи активации. Способы формирования и использования рассматриваются в разд. 1.11, здесь же следует отметить только, что размер этого поля фиксирован и равен размеру любого указательного значения.

5. *Связь по управлению.* Это поле используется для хранения указателя на запись активации вызывающей процедуры. Размер его фиксирован. Используется в реализациях языков, обеспечивающих не текстуальную, а динамическую видимость нелокальных объектов (см. подразд. 1.11.7).

6. *Фактические параметры.* Обычно фактические параметры стремятся передавать через регистры процессора. Тем не менее всегда необходимо учитывать возможность того, что количество фактических параметров превысит доступное количество регистров, поэтому в записи активации процедуры для них должно предусматриваться место. Способы передачи параметров из вызывающей в вызываемую процедуру рассматриваются в разд. 1.12 (и иллюстрируются в разд. 2.4). Здесь следует отметить, что большинство языков программирования предусматривает фиксированное количество и типы значений параметров каждой процедуры. В этом случае размер поля фактических параметров может быть определен во время трансляции вызываемой процедуры. Однако существуют языки (C/C++), допускающие возможность приема вызываемой процедурой заранее неопределенного количества фактических параметров. В этом случае размер поля фактических параметров может быть определен только при трансляции текста вызывающей процедуры.

7. *Поле возвращаемого значения* используется вызываемой процедурой для возврата значения вызывающей процедуре. Для повышения эффективности программы это значение, как правило, возвращается в регистре. Если же для возвращаемого значения резервируется поле в записи активации, то его размер определяется типом значения и может быть легко определен во время трансляции процедуры.

Не во всех языках используется каждое из этих полей (и не все компиляторы с одного и того же языка делают это одинаково). В силу

того что поля записи активации интенсивно используются при выполнении процедуры, для хранения наиболее часто используемых из них разработчики трансляторов стремятся использовать регистры процессора. За формирование полей записи активации обычно отвечают и вызывающая и вызываемая процедуры, но каждая – строго за определенные поля записи.

Запись активации (рис. 1.9) может создаваться (здесь имеется в виду выделение памяти для нее, а не формирование значений полей) следующими способами:

1) статически при трансляции процедуры;

2) динамически в куче;

3) динамически в стеке времени исполнения программы.

Первый способ – создание записи активации в процессе трансляции – преследует цель минимизировать накладные расходы по времени на вызовы процедур при исполнении программы. Типичный пример применения – язык Fortran. С каждой процедурой связана в точности одна запись активации, поскольку статически невозможно создать большее, но заранее неизвестное количество записей активации, а создание ограниченного, но большего единицы количества записей никак не решает проблемы рекурсивности вызовов, но приводит к росту накладных расходов. При статическом создании записей активации не может быть разрешен рекурсивный вызов процедур. Записи активаций процедур размещаются транслятором в области статических данных (см. рис. 1.8).

Динамическое создание записей активации в куче во время исполнения программы предполагает наличие механизма управления кучей (библиотеки процедур, выполняющих учет свободного пространства и обработки запросов на выделение связного блока требуемого размера и, возможно, освобождение ранее выделенного блока). Вполне вероятно, что куча будет использоваться не только для хранения записей активации, поэтому необходимо обеспечивать учет созданных записей (см. рис. 1.9, поле связи по управлению, которое в данном случае становится обязательным).

Следовательно, реализация этого способа приводит к высоким накладным расходам по времени на вызовы, но обеспечивает наибольшую гибкость во взаимодействии вызывающих и вызываемых процедур. В частности, при этом способе время жизни активации вызываемой процедуры в принципе может быть дольше, чем время жизни вызывающей ее активации. Возможна также реализация таких экзотических возможностей, как, например, сопрограммы.

Средняя между предыдущими способами величина накладных расходов в сочетании с наиболее естественной текстуальной видимостью нелокальных данных обеспечивается при динамическом создании записей активации в стеке исполняемой программы. Аппаратная реализация стека минимизирует накладные расходы по времени на выделение памяти для записей активации вызываемых процедур. Рекурсивные вызовы процедур допускаются, при этом количество записей активации одной и той же процедуры может быть сколь угодно большим и ограничиваться только размером области памяти, отведенной под стек программы. В разд. 2.4 рассмотрены два примера результатов семантического анализа и генерации кода с созданием записей активации в стеке времени исполнения.

При создании записей активации процедур в стеке последовательность их уничтожения обратна последовательности образования. Поэтому время жизни активации любой процедуры не может быть больше времени жизни активации вызывающей процедуры.

Другими словами, две произвольные активации любых процедур (возможно, одной процедуры) либо не пересекаются во времени, либо одна из них полностью вложена в другую. На рис. 1.10 показано перемещение потока управления между активациями двух процедур во времени.

Внизу сплошной линией со стрелкой обозначена ось времени. Сплошные вертикальные линии показывают либо вызовы процедур, либо возвраты из них. Жирными горизонтальными линиями показаны процессы исполнения соответствующих активаций процедур. Пунктирными горизонтальными линиями для некоторых активаций выделены интервалы, когда активации существуют, но не исполняются, поскольку в это время процессор занят исполнением другой активации либо той же самой процедуры (интервалы $t_2 - t_3$ и $t_4 - t_5$ для активации 1 процедуры A), либо другой процедуры (интервалы $t_1 - t_2$, $t_3 - t_4$, $t_5 - t_6$ и $t_6 - \ldots$ для активации 1 процедуры A, $t_2 - t_3$ и $t_4 - t_5$ для активации 1 процедуры B).

Согласно дисциплине существования элементов стека после создания, к примеру, активации 2 процедуры A и до ее завершения не может завершиться существование ни активации 1 процедуры B, ни активации 1 процедуры A. Заметим, что и то и другое событие в принципе возможно при динамическом создании записей активаций процедур в куче. Для реализации этих событий нужно просто обеспечивать явное уничтожение записей ненужных активаций и возвраты по адресам, предварительно взятым из них.

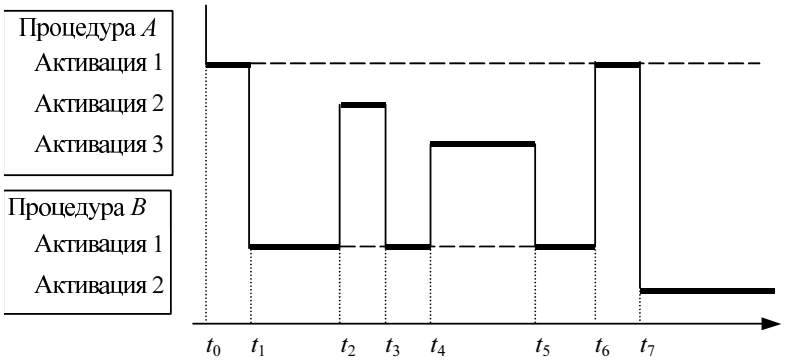


Рис. 1.10. Времена жизни активаций процедур

На рис. 1.11 показана последовательность состояний стека программы, соответствующая моментам времени t_0, … t_7. Записи активаций обозначены буквенным названием процедуры (*A* или *B*) и порядковым номером активации.

Для изображения того, как управление передается активациям и покидает их, может оказаться полезным так называемое дерево активаций. Это графическое представление процесса исполнения совокупности процедур, для которого выполняются следующие условия.

1. Корень дерева представляет активацию основной программы.
2. Каждый узел представляет активацию процедуры.
3. Узел *A* является родительским для узла *B* тогда и только тогда, когда поток управления передается из активации *A* в активацию *B* (узел *B* называется потомком узла *A*).
4. Если два узла *B* и *C* являются потомками одного и того же узла, то узел *B* располагается слева от узла *C* тогда и только тогда, когда время жизни активации *B* начинается раньше времени жизни активации *C*.

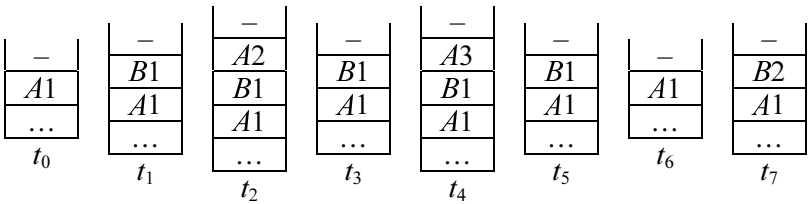


Рис. 1.11. Последовательность состояний стека программы

Поскольку каждый узел представляет единственную активацию, и наоборот, удобно говорить о том, что управление находится в узле, когда оно находится в активации, представленной этим узлом. Если пометить каждый узел дерева именем (обозначением) процедуры и порядковым (во времени) номером ее активации, то все узлы дерева будут иметь уникальные пометки. Фрагмент дерева активаций для случая, изображенного на рис. 1.10, может выглядеть так, как показано на рис. 1.12.

Последовательность событий создания/уничтожения активаций процедур может быть получена путем обхода дерева, начинающегося с корня и заключающегося в последовательном посещении всех поддеревьев каждой вершины строго слева направо. Размещение записей активации процедур в стеке исполняемой программы наиболее часто используется в реализациях языков программирования общего назначения. Независимо от того, где хранятся записи активации в процессе исполнения программы, для любой из них должен быть отведен связный блок памяти такого размера, чтобы в нем размещались все поля записи. Другими словами, к моменту фактического выделения блока памяти под запись активации необходимо точно знать его требуемый размер, который равняется сумме размеров всех полей.

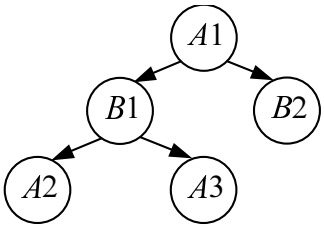


Рис. 1.12. Фрагмент дерева активаций

Рассмотрим, каким образом определяются размеры полей локальных данных. Способы формирования полей промежуточных вычислений и фактических параметров будут рассмотрены позже. Остальные поля записи активации, как уже упоминалось, имеют фиксированные размеры.

1.9. Локальные данные процедур

Для определения размера поля локальных данных транслятор должен разметить (распределить) его образ памяти. Для каждой процедуры этот процесс начинается с фиксации начального адреса образа памяти поля локальных данных, обычно равного сумме размеров всех предшествующих полей записи активации (см. рис. 1.9). Далее последовательно перебираются наименования объектов, ассоциации которых должны быть активными при исполнении инструкций процедуры. Текущее значение адреса первого свободного элемента (обычно байта) памяти заносится в соответствующую объекту запись таблицы ассоциаций (как правило это просто таблица идентификаторов), из этой записи извлекается тип объекта, по нему определяется требуемое количество байт памяти и на это количество увеличивается счетчик занятых байт (т. е. указатель первого свободного байта).

При генерации объектного кода сформированный таким образом адрес объекта будет использован в качестве смещения в любой команде, оперирующей с этим объектом. При исполнении программы в момент вызова процедуры адрес записи активации будет занесен в строго определенный регистр процессора (если записи активации размещаются в стеке, то для доступа к ним может использоваться указатель стека). Для обеспечения этого транслятор формирует и помещает в точку входа в процедуру дополнительные команды. Доступ к значению любого локального объекта осуществляется по адресу, вычисляемому как сумма базового адреса записи активации (из регистра процессора, специально выделяемого для этой цели) со смещением (из команды). Если стек растет в направлении увеличения адресов памяти, то все значения смещений будут отрицательными. Для ситуации, представленной на рис. 1.13, предполагается, что стек растет в сторону уменьшения адресов памяти, поэтому значения смещений положительны.

Кажущаяся примитивность процесса формирования смещений локальных объектов осложняется двумя обстоятельствами. Первое состоит в том, что на компоновку памяти для объектов данных сильное влияние оказывают ограничения системы адресации целевого компьютера и/или требования оптимизации по времени выполнения.

Пусть, например, выборка данных из памяти выполняется четырехбайтными кадрами, адрес первого из которых должен быть кратен четырем. В таком случае для доступа к двухбайтному целому значению может потребоваться два обращения к памяти, если его адрес нечетен. Такой случай возможен, если транслятор экономно использует память и размещает данные слитно. Если же при трансляции преследу-

ется цель построения быстрой программы, то определение размера памяти, отводимой под каждый объект, должно производиться с учетом оптимального выравнивания адреса его первого байта.

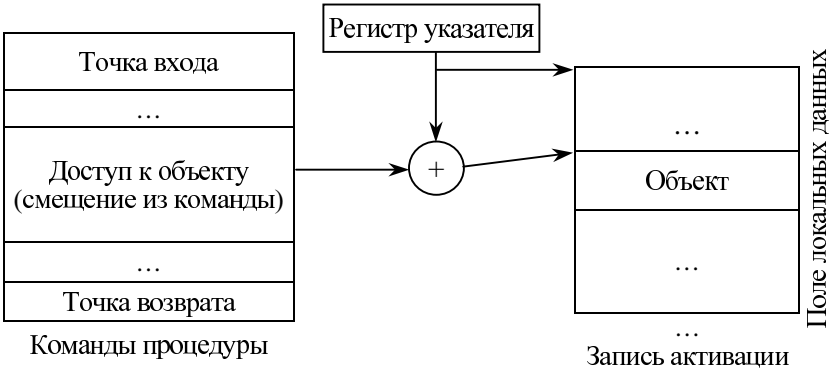


Рис. 1.13. Формирование исполнительного адреса, объект внутри поля локальных данных

Второе усложняющее обстоятельство ранее уже упоминалось: если в процедуре в качестве локального объявлен динамический массив, у которого хотя бы одна граница изменения индекса зависит от значений фактических параметров, то во время трансляции размер этого массива определить невозможно. Следовательно, до момента вызова процедуры невозможно определить размер записи активации. Существует по меньшей мере два способа преодоления этой трудности. Независимо от того, какой способ применяется, в точку входа в процедуру неявно для программиста транслятор добавляет инструкции, вычисляющие для каждого такого массива требуемый размер памяти.

При первом способе (см. рис 1.14) окончательное определение размера поля локальных данных и размера всей записи активации откладывается до момента входа в процедуру.

Ясно, что адреса всех таких массивов, кроме первого из них, не могут быть зафиксированы во время трансляции. Следовательно, машинные команды, оперирующие с элементами второго и всех последующих массивов, не могут быть построены тем же способом, что и команды, работающие с обычными локальными данными.

Вызвано это тем, что для формирования исполнительного адреса элемента массива теперь приходится складывать три величины:

– базовый адрес записи активации;

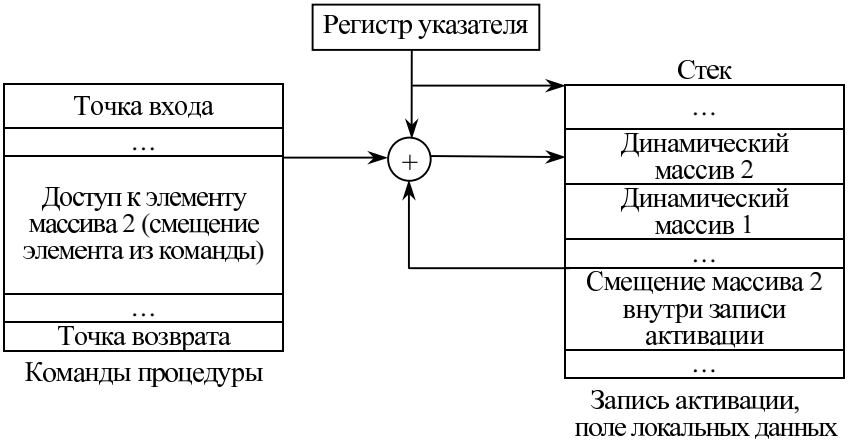


Рис. 1. 14. Формирование исполнительного адреса, элемент динамического массива внутри поля локальных данных

– смещение первого элемента динамического массива внутри записи активации (вычисляется после входа в процедуру и хранится в качестве локального объекта, построенного транслятором);

– смещение элемента массива (берется из команды либо вычисляется до ее выполнения, если транслятору неизвестны индексы этого элемента).

Второй способ, показанный на рис. 1.15, состоит в том, что память для динамических массивов не резервируется в записи активации, а выделяется из кучи после того, как вычислен требуемый размер массива.

В записи активации во время трансляции резервируется фиксированное количество байт для хранения указателя на память, взятую из кучи.

Указатель используется в командах, оперирующих с элементами массива, которые строятся транслятором не так, как команды, работающие с обычными локальными данными (в том числе массивами).

Адрес памяти для доступа к элементу динамического массива при хранении его в куче вычисляется так:

– вначале известное транслятору смещение указателя массива складывается с адресом записи активации и используется для выборки адреса массива и помещения его в регистр процессора;

– затем значение этого регистра складывается со смещением нужного элемента массива и по полученному адресу извлекается операнд.

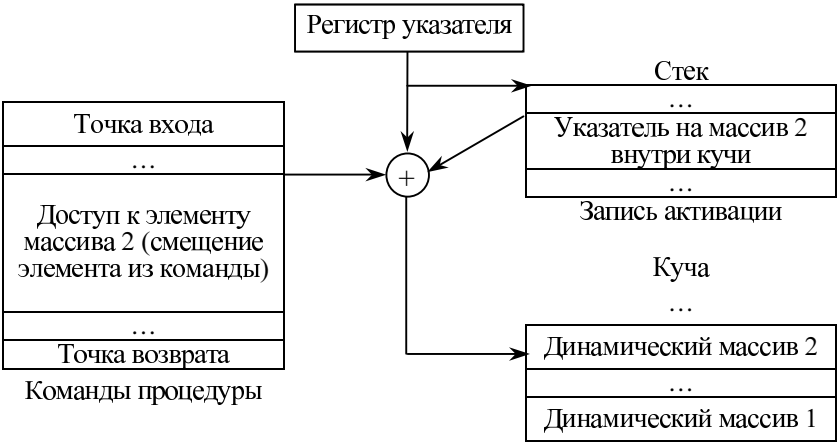


Рис. 1.15. Формирование исполнительного адреса,
память для динамического массива выделяется из кучи

Главным достоинством использования кучи для хранения динамических массивов является вычисление размера записи активации во время трансляции, а не во время исполнения программы.

1.10. Вызывающие последовательности

Вызовы процедур реализуются путем исполнения так называемых вызывающих последовательностей команд. Эти последовательности размещаются частично в вызывающей, частично в вызываемой процедуре. Они обеспечивают передачу управления и данных из вызывающей процедуры в вызываемую (последовательность вызова), и наоборот – из вызываемой в вызывающую (последовательность возврата). Последовательность вызова создает запись активации и заполняет ее поля необходимой информацией. Последовательность возврата восстанавливает состояние машины таким образом, чтобы вызывающая процедура могла продолжать исполнение.

Последовательности вызовов и структура записи активации различаются даже в разных реализациях одного и того же языка. Коды в последовательностях вызова и возврата зачастую разделяются между вызывающей и вызываемой процедурами. Не существует однозначного разграничения задач между вызывающей и вызываемой процедурами – исходный язык, целевая машина и операционная система налагают

свои требования, в силу которых может оказаться предпочтительным то или иное решение.

Поскольку каждый вызов имеет собственные фактические параметры, вызывающая процедура обычно вычисляет их значения и передает их в вызываемую процедуру. Методы передачи параметров обсуждаются в разд. 1.12. В стеке времени исполнения запись активации вызывающей процедуры находится непосредственно под записью активации вызываемой процедуры, как показано на рис. 1.16. Размещение полей параметров и возможного возвращаемого значения вслед за записью активации вызывающей процедуры дает некоторый выигрыш за счет того, что вызывающая процедура может получить доступ к таким полям, используя только смещения от конца собственной записи активации. Для этого не нужна полная информация о компоновке записи активации вызываемой процедуры. В частности, при трансляции вызывающей процедуры, как правило, ничего не известно о локальных переменных вызываемой процедуры.

Несмотря на то что размер поля для временных значений в конечном счете фиксируется во время компиляции, он может быть неизвестен вплоть до окончания этапов оптимизации и первой фазы генерации объектного кода. На этих этапах количество временных переменных для хранения промежуточных значений может быть уменьшено, а следовательно, на этапе семантического анализа размер этого поля может быть неизвестен. Поэтому поле временных переменных лучше размещать после поля локальных данных, поскольку изменения его размера не будут влиять на смещение прочих локальных объектов.

Если вызов процедуры встречается в тексте программы n раз, то часть последовательности вызова в вызывающих процедурах будет построена транслятором n раз, в то время как часть последовательности вызова в вызываемой процедуре генерируется лишь единожды. Следовательно, для экономии памяти желательно разместить как можно большую часть последовательности вызова в коде вызываемой процедуры. Однако не всю работу по организации вызова можно переложить на вызываемую процедуру.

Для выполнения всей работы необходимы два регистра процессора: указатель стека и указатель текущей записи активации.

Указатель стека (УС) в любой момент времени разделяет занятое и свободное пространства в стеке и используется всегда для доступа к единственному объекту, находящемуся на верхушке стека. Он не может быть использован для доступа к любым другим объектам в силу

того, что значение этого указателя постоянно изменяется в результате выполнения различных команд. Поэтому для хранения базового адреса записи активации, как правило, выделяется еще один регистр процессора – указатель записи активации (УЗ). С использованием двух регистров типичная последовательность действий по вызову процедуры выглядит так, как проиллюстрировано на рис. 1.16.

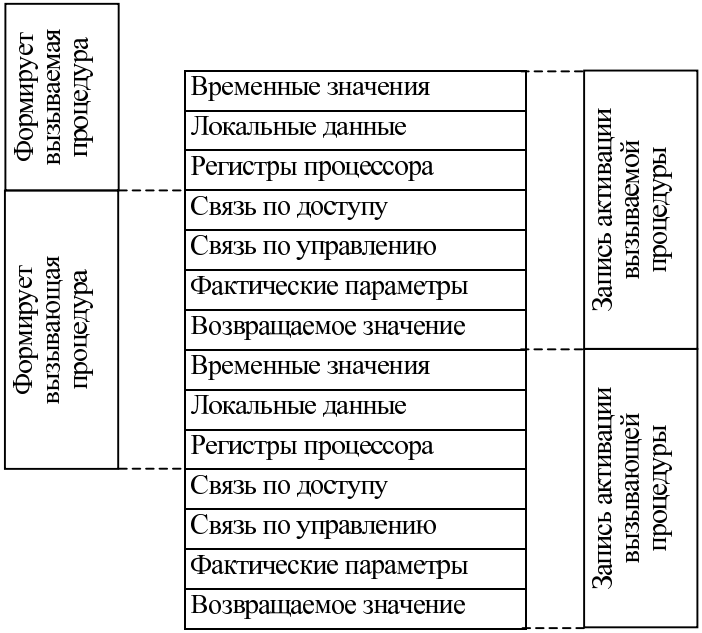


Рис. 1.16. Разделение задач между вызывающей и вызываемой процедурами

Перед началом процесса вызова регистр УЗ указывает на конец поля состояния машины в записи активации.

1. Вызывающая процедура вычисляет фактические параметры и заносит их в стек, модифицируя УС.

2. Вызывающая процедура сохраняет адрес возврата и старое значение УЗ в записи активации вызываемой процедуры, после чего устанавливает значение УЗ равным значению УС, как показано на рис. 1.16. Таким образом, УЗ перемещается за локальные и временные данные

вызывающей процедуры, параметры и поле состояния машины вызываемой процедуры.

3. Управление передается вызываемой процедуре, которая сохраняет в стеке значения регистров и другую информацию о состоянии машины и модифицирует УС таким образом, чтобы зарезервировать место под свои локальные данные и временные значения.

4. Вызываемая процедура инициализирует локальные данные и переходит к исполнению своего тела.

Возможная последовательность возврата из процедуры выглядит следующим образом.

1. Вызываемая процедура размещает возвращаемое значение после записи активации вызывающей программы. Используя информацию в поле состояния машины, вызываемая процедура восстанавливает УЗ и другие регистры и передает управление по адресу возврата в коде вызывающей процедуры.

2. Хотя значение УЗ было уменьшено, вызывающая процедура может скопировать возвращаемое значение в собственную запись активации и использовать его в вычислениях.

Приведенные выше последовательности вызова позволяют реализовывать процедуры с переменным числом аргументов. Транслятор, обрабатывая вызывающую процедуру, может подсчитать количество фактических параметров. Следовательно, при формировании кода вызывающей процедуры ему известен размер поля параметров, который может быть неявно передан тоже в виде параметра. Теперь остается только таким образом формировать код вызываемой процедуры, чтобы он правильно обрабатывал неизвестное при трансляции, но получаемое в момент вызова количество параметров. Другой возможный вариант способа реализации переменного количества параметров – функции типа *printf* в языке С. Первый параметр такой функции содержит перечень и типы остальных параметров в виде текстовой строки. Поэтому как только код тела функции *printf* выбирает первый параметр, он может по нему определить способ доступа к произвольному количеству остальных параметров.

1.11. Доступ к нелокальным объектам

Правила области видимости языка определяют способ работы со ссылками на нелокальные объекты – такие, объявления которых находятся вне текста процедуры, в которой объект используется. Существует общее правило, именуемое правилом текстуальной (или статиче-

ской) области видимости, согласно которому видимость любого объекта определяется исключительно расположением его объявления в тексте программы.

В языках Pascal, C и Ada и многих других используется текстуальная область видимости с добавлением правила «видимо ближайшее вложенное».

В некоторых языках программирования используется другое правило динамической области видимости, при котором ассоциация наименования любого объекта с его объявлением определяется не текстом программы, а совокупностью имеющихся на данный момент активаций процедур. Согласно этому правилу доступ к значению нелокального объекта осуществляется путем поиска его имени в таблицах, ассоциированных с активациями процедур. Просмотр активаций процедур должен осуществляться в заранее обусловленной последовательности. Правило динамической области видимости используется в таких языках, как Lisp, APL и Snobol.

Исторически понятие нелокальных объектов появилось в языке Algol и связано с впервые введенной в этом языке конструкцией блоков. Блок представляет собой последовательность инструкций, содержащую объявления собственных локальных данных. Основная отличительная особенность понятия блока заключается во вложенности структуры. Блоки либо следуют друг за другом, либо один блок является полностью вложенным в другой. Невозможно такое перекрытие блоков B_1 и B_2, при котором первым по тексту начинается блок B_1, после чего начинается блок B_2, затем заканчивается блок B_1, а затем – B_2.

Начало и конец блока оформляются специальными ограничителями (в C для этой цели используются фигурные скобки *{* и *}*; в Algol и Pascal – ключевые слова *begin* и *end*).

1.11.1. Блочные области видимости

Область видимости любого объекта в языке с блочной структурой определяется правилом ближайшего вложенного, суть которого состоит в следующем.

1. Область видимости объявления в блоке B включает весь блок B.

2. Если объект x не объявлен в блоке B, то все появления имени x в B находятся в области видимости объявления x во внешнем окружающем блоке B_o, таком, что:

- B_o содержит объявление объекта с именем x;

• B_o является ближайшим окружающим B блоком среди всех, содержащих объявление x.

Для пояснения того, как действует правило ближайшего вложенного, приведем фрагмент программы на языке С (рис. 1.17).

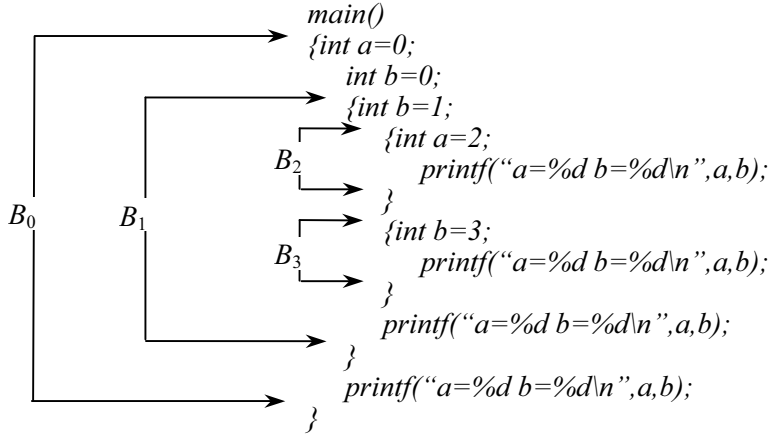


Рис. 1.17. Блоки и видимость переменных в С-программе

В нем каждое объявление инициализирует объявляемый объект числом, равным номеру блока, в котором оно появляется. Обратим внимание на то, что область видимости целой переменной b, объявленной в блоке B_0, не включает блок B_1, поскольку в блоке B_1 объявлена другая переменная с тем же именем b. Такой разрыв называется «дырой» в области видимости объявления.

Всего в этом тексте объявлено пять разных объектов, области видимости которых приведены в табл. 1.5.

Таблица 1.5

Объявление	Область видимости
int a = 0;	B_0, B_2
int b = 0;	B_0, B_1
int b = 1;	B_1, B_3
int a = 2;	B_2
int b = 3;	B_3

Результаты работы этой программы иллюстрируют правило «ближайшего вложенного». При ее исполнении управление передается внутрь блока из точки, находящейся непосредственно перед ним, а затем из блока возвращается в точку, следующую прямо за ним в исходном тексте. Инструкции печати выполняются в последовательности B_2, B_3, B_1 и B_0 (в порядке, в котором завершается

исполнение блоков). Значения переменных a и b в этих блоках показаны в табл. 1.6.

Таблица 1.6

Блок	a	b
B_2	2	1
B_3	0	3
B_1	0	1
B_0	0	0

Блочная структура программы может быть реализована с использованием стекового распределения памяти. Поскольку область видимости объявления не выходит за пределы того блока, в котором оно записано, память для объявленного объекта может быть выделена при входе в блок и освобождена при выходе из него. Это эквивалентно использованию блока как «процедуры без параметров», вызываемой только из одной точки непосредственно перед блоком и возвращающей управление в точку непосредственно за блоком.

Нелокальная среда ссылок для блока может поддерживаться с использованием технологий для процедур, которые будут рассмотрены в этой главе далее. Заметим, что с точки зрения разработчика транслятора реализация блоков несколько проще, чем реализация процедур, поскольку блокам не передаются параметры, а поток управления строго следует статическому тексту программы (не осуществляются передачи управления).

Другая возможная реализация программ с блочной структурой состоит в однократном выделении памяти для всех локальных объектов процедуры в тот момент, когда ей передается управление. При наличии блоков в процедуре размер поля локальных данных ее записи активации рассчитывается транслятором с учетом всех объявлений внутри блоков.

Для переменных программы, приведенной на рис. 1.17, можно выделить память так, как показано на рис. 1.18. Индексы у локальных переменных a и b на этом рисунке соответствуют блоку, в котором они объявлены.

...	a_0	b_0	b_1	a_2, b_3	...

Рис. 1.18. Локальные объекты в записи активации

Заметим, что для объектов a_2 и b_3 может быть отведена одна и та же область памяти, поскольку блоки B_2 и B_3 не пересекаются.

При отсутствии динамических массивов (и других данных переменной длины) максимальное количество памяти, необходимой для выполнения блока, может быть определено в процессе компиляции (с данными переменной длины можно работать, используя указатели, как

это описано в разд. 1.8). Определение этого размера транслятор должен выполнять исходя из предположения, что при исполнении программы будут пройдены все возможные пути управления по тексту процедуры.

1.11.2. Текстуальная область видимости без вложенных процедур

Для изучения того, каким образом реализуется видимость нелокальных объектов в языке, не допускающем текстуально вложенных процедур, рассмотрим программу сортировки элементов массива на языке С (рис. 1.19).

```
(1)  int arr[ ];
(2)  void ReadArray(void) { ... arr ... }
(3)  int PartExch(int iBeg,int iEnd) {
(4)      int i, j, k, mVal;
(5)      i=iBeg; j=iEnd; mVal=arr[i+(j-i)/2];
(6)      while(i<j) {
(7)          for( ;arr[i]<mVal;i+=1);
(8)          for( ;arr[j]>mVal;j-=1);
(9)          if(i<j){k=arr[i];arr[i]=arr[j];arr[j]=k;}
(10)     }   //end while
(11)     return i;
(12) }   //end PartExch
(13) void QuickSort(int iBeg,int iEnd) {
(14)     int iMid;
(15)     if(iBeg<iEnd) {
(16)         iMid=PartExch(iBeg,iEnd);
(17)         if(iBeg<iMid-1) QuickSort(iBeg,iMid-1);
(18)         if(iMid+1<iEnd) QuickSort(iMid+1,iEnd);
(19)     }
(20) }   //end QuickSort
(21)  void main() {
(22)     ReadArray();QuickSort(0,(sizeof(arr)/sizeof(int))-1);
(23) }   //end main
```

Рис. 1.19. С-программа с нелокальным массивом *arr*

Эта программа предназначена для демонстрационных целей, ее не следует (как и приведенную ниже программу на языке Pascal) рассматривать в качестве образца программирования или реализации методов сортировки. Правила текстуальной области видимости в языке С проще, чем в языке Pascal (для языка Pascal они обсуждаются позже) вследствие того, что определение функции не может находиться внутри другой функции. Если в некоторой функции используется не объявленный в ней объект *X*, то предполагается, что его объявление находится вне любой функции. Область видимости любого такого объявления состоит из тел функций, следующих за ним (объявлением) по тексту файла (с дырами, если внутри функций имеются объявления объектов с точно такими же именами, см. подразд. 1.11.1).

При отсутствии вложенных процедур для языка с текстуальными областями видимости можно использовать стратегию стекового распределения памяти для локальных объектов, описанную в разд. 1.9. Память для объектов, объявленных вне процедур, может быть выделена статически. Положение этой памяти известно в процессе компиляции, так что даже если некоторый объект нелокален в теле данной процедуры, транслятор будет использовать статически определенный адрес объекта для построения команд, оперирующих с его значением.

Важное преимущество статического выделения памяти для нелокальных объектов заключается в том, что любые процедуры могут свободно передаваться в качестве параметра другим процедурам и возвращаться как результат вызова (в языке С функция передается как параметр путем передачи указателя на нее).

При использовании текстуальной области видимости и отсутствии вложенных процедур любой нелокальный для одной процедуры объект является нелокальным и для всех остальных процедур, а следовательно, его статический адрес может использоваться всеми процедурами одинаково и независимо от того, каким образом они активируются. Это справедливо для любой процедуры, в том числе и для процедур, которые вызываются по вычисляемым ссылкам.

1.11.3. Текстуальная область видимости из вложенных процедур

Для изучения этого понятия рассмотрим текст программы на языке Pascal, также реализующий сортировку элементов массива, но использующий вложенное определение процедур (рис. 1.20).

```
(1)  program Sort(input, output);
(2)      var arr : array [0..11] of integer;x: integer;mVal,rest:word;
(3)      procedure readarray;
(4)          begin ... end { readarray };
(5)      procedure Exchange(iFirst,iSecond: integer);
(6)          begin
(7)              x:=arr[iFirst]; arr[iFirst]:=arr[iSecond]; arr[iSecond]:=x;
(8)          end { exchange };
(9)      procedure QuickSort(iBeg,iEnd : integer);
(10)         var iMid,k,si,sj: integer;
(11)         function PartExch(iBeg,iEnd: integer): integer;
(12)             var i,j: integer;
(13)             begin
(14)                 k:=mVal; i:=iBeg; j:=iEnd;si:=0;sj:=0;
(15)             while i<j do begin
(16)                 while arr[i]<k do begin si:=si+arr[i]; i:=i+1; end;
(17)                 while arr[j]>k do begin sj:=sj+arr[j]; j:=j-1; end;
(18)                 if (i<j) then Exchange(i,j);
(19)             end;
(20)             Result:=i;
(21)             end { PartExch };
(22)         begin
(23)             if iBeg<iEnd then begin
(24)             iMid:=PartExch(iBeg,iEnd);
(25)             if iBeg<iMid-1 then begin
(26)                 DivMod(si,iMid-iBeg,mVal,rest);
(27)                 QuickSort(iBeg,iMid-1); end;
(28)             if iMid+1<iEnd then begin
(29)                 DivMod(sj,iEnd-iMid,mVal,rest);
(30)                 QuickSort(iMid,iEnd); end;
(31)             end;
(32)         end { QuickSort };
(33)     begin
(34)         readarray; mVal:=arr[5];
(35)         QuickSort(0, 11);
(36)     end; { sort }
```

Рис. 1.20. Pascal-программа с вложенными процедурами

В языке Pascal объект, нелокальный для текста данной процедуры, находится в области видимости ближайшего из его предыдущих по тексту объявлений с учетом правила блочной вложенности, применяемого к процедурам как к блокам.

В этой программе процедуры вложены друг в друга так:

Sort содержит объявления *ReadArray, Exchange, QuickSort.*

QuickSort содержит объявление *PartExch.*

Массив *arr* используется как нелокальный объект в нескольких процедурах, в том числе – в функции *PartExch*, вложенной в процедуру *QuickSort*. Ближайшее вложенное объявление *arr* (и нескольких простых переменных, используемых в разных процедурах) находится в строке (2) процедуры *Sort*. В процедуре *PartExch* кроме *arr* нелокальными являются переменные *k,si,sj*. Их ближайшее предыдущее по тексту объявление находится в строке (10) процедуры *QuickSort*.

Правило ближайшего вложенного применимо и к именам процедур. Так, процедура *Exchange*, вызываемая процедурой *PartExch* в строке (18), является нелокальной по отношению к *PartExch*. При обработке вызова этой процедуры транслятор вначале проверяет, не определена ли процедура *Exchange* в *QuickSort*, и, поскольку это не так, для организации вызова берется ее объявление в программе *Sort*. Для изучения того, как реализуется текстуальная область видимости, предварительно необходимо рассмотреть понятие глубины вложенности процедуры.

1.11.4. Глубина вложенности

Поставим уровень вложенности 1 в соответствие основной процедуре исполняемой программы. Просмотрим текст и при переходе от каждой внешней процедуры к любой вложенной в нее процедуре будем добавлять по единице к текущей глубине вложенности. В тех точках текста, где текущая процедура завершается, глубину вложенности будем уменьшать на единицу.

На рис. 1.20 процедура *QuickSort* в строке (9) имеет глубину вложенности 2, процедура *PartExch* в строке (11) – глубину вложенности 3. Теперь с каждым именем объекта в тексте программы можно связать глубину вложенности той процедуры, в которой этот объект объявлен. Таким образом, имена *arr*, *k* и *i*, встречающиеся, например, в строке 16 процедуры *PartExch*, имеют глубину вложенности 1, 2 и 3 соответственно.

Непосредственная реализация текстуальной области видимости для вложенных процедур может быть получена путем использования указателя, называемого связью доступа, находящегося в каждой записи активации. Если процедура *p* в исходном тексте вложена в процедуру *q* непосредственно, то связь доступа в любой записи активации *p* должна указывать на запись последней по времени активации *q*.

Снимки состояний стека времени исполнения для некоторых моментов выполнения Pascal-программы сортировки элементов массива приведены на рис. 1.21. Для краткости на рисунке приведены только первые буквы имен процедур, а значения связей доступа показаны стрелками.

Связь доступа у активации *Sort* пуста, поскольку у нее нет окружающей ее процедуры (а точнее, мы ее не рассматриваем). Связь доступа в каждой активации *QuickSort* указывает на запись для *Sort*. Следует обратить внимание на то, что на рис. 1.21, *в* связь доступа в записи активации *PartExch(1,3)* указывает на связь доступа в записи последней активации *QuickSort*, а именно на *q(1,3)*.

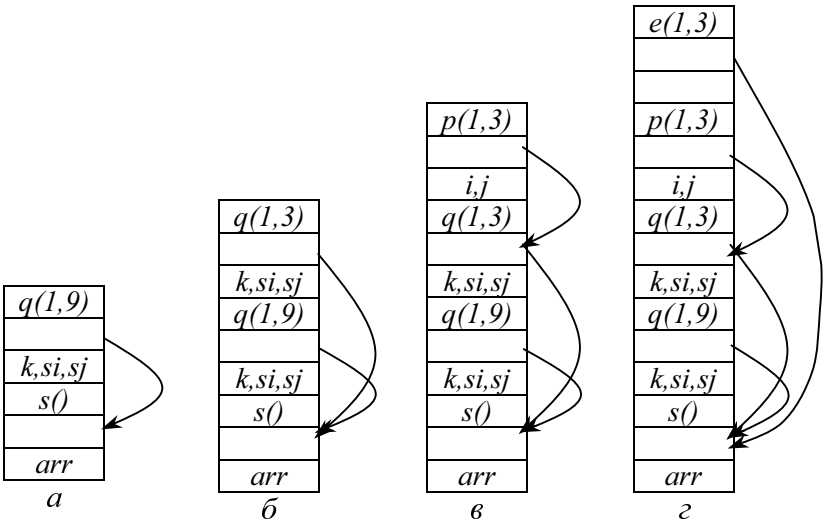


Рис. 1.21. Связи доступа для определения положения нелокальных объектов

Предположим, что процедура *p* с глубиной вложенности n_p обращается к нелокальному объекту *a* с глубиной вложенности n_a (очевид-

но, что $n_a < n_p$). Область памяти, выделенная для объекта *a*, может быть найдена следующим образом.

1. Когда управление находится в *p*, запись активации для *p* располагается на вершине стека. Нужно перейти по ровно $n_p - n_a$ связям доступа от записи, находящейся на вершине стека (заметим, что значение разности $n_p - n_a$ может быть вычислено в процессе трансляции). После прохода по $n_p - n_a$ -связям будет достигнута запись активации той процедуры, по отношению к которой объект *a* локален. Область памяти, выделенная транслятором для хранения его значения, имеет фиксированное положение в записи, т. е. известно ее смещение относительно достигнутой связи доступа.

Следовательно, адрес нелокального объекта *a* в процедуре *p* может быть задан парой значений.

1. Расстояние $n_p - n_a$ по связям доступа от текущей записи активации до той, которая содержит значение нужного объекта.

2. Смещение объекта *a* в содержащей его записи активации.

Оба значения могут быть вычислены в процессе трансляции и использоваться в процессе исполнения.

Например, в строках (16) и (17) процедура *PartExch* с глубиной вложенности 3 (см. рис. 1.20) обращается к нелокальным переменным *arr* и *k* с глубиной вложенности 1 и 2 соответственно. Записи активации, в которых хранятся эти переменные, находятся прохождением соответственно 3–1=2 и 3–2=1 связей доступа от любой записи активации процедуры *PartExch*.

Код формирования связей доступа является частью последовательности вызова и должен строиться транслятором в соответствии с семантикой языка. Предположим, процедура *p* глубины вложенности n_p вызывает процедуру *x* глубины вложенности n_x. Код настройки связи доступа вызываемой процедуры зависит от того, является ли вызываемая процедура вложенной в вызывающую. Возможны два случая.

1. $n_p < n_x$. Поскольку вызываемая процедура *x* вложена по тексту глубже процедуры *p*, она должна быть объявлена в *p* (иначе из *p* она будет просто недоступна). Так происходит, например, при вызове *QuickSort* из *sort* на рис. 1.21,*а* и при вызове *PartExch* из *QuickSort* на рис. 1.21, *в*. В этом случае связь доступа в вызываемой процедуре должна указывать на запись активации вызывающей процедуры, находящейся непосредственно над текущей записью в стеке.

2. $n_p > n_x$. Согласно правилам реализации области видимости процедуры с глубинами вложенности 1, 2, ..., n_{x-1}, окружающие и вызы-

ваемую, и вызывающую процедуры, должны быть одними и теми же как при вызове *QuickSort* самой себя (рис. 1.21, *б*), так и при вызове функцией *PartExch* процедуры *exchange* (рис. 1.21, *г*). Следуя по пути от n_p до n_x по связям доступа от вызывающей процедуры, можно при трансляции определить, какая процедура статически по тексту наиболее тесно охватывает и вызывающую, и вызываемую процедуры. В формируемый код вызываемой процедуры транслятор теперь может заложить инструкции продвижения по (n_p – n_x) связям доступа, последнюю из которых процедура должна занести в собственную запись активации.

Теперь и в том и в другом случае значение связи доступа может быть использовано для вычисления адресов нелокальных объектов процедуры.

1.11.5. Процедуры как параметры

Правила текстуальной области видимости применимы и в том случае, когда вложенная процедура передается в другую процедуру в качестве параметра. В приведенной ниже Pascal-программе (рис. 1.22) функция *d* (строки (6) и (7)) использует нелокальную переменную *m*.

```
(1)   program param(input, output);
(2)       procedure b(function h(n: integer): integer);
(3)           begin writeln(h(2)) end { b };
(4)       procedure c;
(5)           var m: integer;
(6)           function d(n: integer): integer;
(7)               begin Result : =m +n end { d };
(8)           begin m := 0; b(d) end { c } ;
(9)       begin
(10)          c;
(11)      end.
```

Рис. 1.22. Необходимость передачи связи по доступу

Все вхождения переменной *m* для наглядности выделены жирным шрифтом. В строке (8) процедура *c* присваивает переменной *m* нулевое значение, а затем передает процедуре *b* в качестве параметра процедуру *d*. Заметим, что область текстуальной видимости объявления *m* в строке (5) не включает тело процедуры *b* в строках (2)–(3).

В теле процедуры *b* инструкция *writeln (h(2))* активирует *d*, поскольку формальный параметр *h* ссылается на эту функцию. Таким образом, инструкция *writeln* в строке (3) фактически выводит результат вызова функции со значением параметра, равным двум.

Каким образом должна быть настроена связь доступа для данной активации *d*? В соответствии с тем, что говорилось в предыдущем параграфе об определении процедуры, текстуально наиболее тесно охватывающей и вызывающую, и вызываемую процедуры, становится ясно, что вместе с процедурой, передаваемой в качестве параметра, должна передаваться ее связь доступа, как это показано пунктирной линией на рис. 1.23.

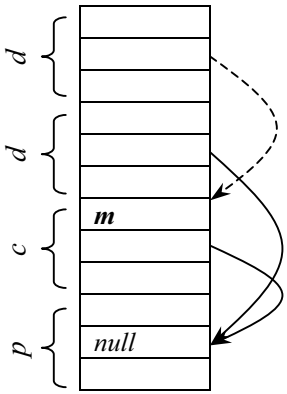


Рис. 1.23. Фактический параметр *d* передается вместе со связью доступа

В тот момент, когда процедура *c* передает в качестве параметра указатель на процедуру *d*, она определяет связь доступа для *d* точно таким же образом, как если бы готовился вызов этой функции из процедуры *c*. Эта связь неявным для программиста образом передается вместе с *d* (фактически передается указатель на *d*) процедуре *b*. Затем при активации *d* из *b* переданная связь используется для настройки связи доступа в записи активации *d*. Эта связь указывает на запись активации процедуры *c*, как и должно быть согласно текстуальной видимости нелокального объекта ***m***.

Следует заметить, что реализация видимости нелокальных объектов путем передачи связей по доступу требует довольно больших временных затрат. Существует более быстрый способ получения того же

самого эффекта, который и используется в большинстве современных реализаций языков с текстуальной вложенностью – доступ через дополнительный стек указателей на записи активации.

1.11.6. Доступ к нелокальным объектам с использованием дисплея

Более быстрый и простой в реализации (по сравнению с использованием связей доступа) способ обращения к нелокальным объектам можно обеспечить с помощью стека *d* указателей на записи активации, который называется дисплеем.

Состояние этого стека в процессе исполнения программы поддерживается таким образом, чтобы память для нелокального объекта *a*, объявленного в процедуре с глубиной вложенности *i*, находилась в той записи активации, на которую указывает элемент дисплея *d*[*i*]. Для обеспечения этого транслятор, естественно, строит вызывающие последовательности процедур должным образом.

Предположим, что управление находится в активации процедуры *p* с глубиной вложенности *j*. Тогда первые *j–1* элементов дисплея указывают на ранее созданные записи активации процедур, текстуально охватывающих процедуру *p*, и элемент дисплея *d*[*j*] указывает на активацию *p*. Значения глубин вложенности всех охватывающих процедур точно известны транслятору, поэтому при необходимости формирования команд обращения к нелокальному объекту генератор кода использует константные значения смещений в дисплее.

Собственно доступ к нелокальному объекту в процессе исполнения программы протекает путем косвенного обращения к памяти через указатель, выбираемый из одной ячейки дисплея. К значению этого указателя добавляется константное (известное транслятору) смещение объекта внутри записи активации.

Заметим, что обращение к собственному локальному объекту протекает путем добавления смещения к указателю, взятому из быстрого регистра указателя записи активации УЗ. Хранению дисплея тоже в регистрах препятствует только то обстоятельство, что размер этого массива указателей в принципе может быть совершенно произвольным (например, еще и потому, что вложенные блоки могут, как уже говорилось, рассматриваться в качестве процедур без параметров). Впрочем, есть и еще одна причина: далеко не во всех языках реализуются вложенные нелокальные среды ссылок. Так, язык С, на котором разра-

батывается большинство операционных систем и системных утилит, не использует этого механизма.

Простая схема поддержки дисплея использует связи доступа в дополнение к нему. Частью и вызывающей, и возвращающей последовательностей является обновление дисплея путем следования по цепочке связей доступа. При следовании по связи к записи активации с глубиной вложенности *n* элемент дисплея *d*[*n*] устанавливается на эту запись активации. По сути, дисплей дублирует информацию цепочки связей доступа (из записи активации выполняющейся в данный момент процедуры).

Эту простую схему можно усовершенствовать. Метод, иллюстрируемый на рис. 1.24, требует меньшей работы при входе в процедуру и выходе из нее при обычном вызове, когда процедура не передается в качестве параметра.

На рис. 1.24 дисплей состоит из отдельного глобального массива (под глобальностью понимается возможность прямого доступа к нему из вызывающей последовательности любой процедуры). Снимки состояний стека времени исполнения и дисплея на рисунке соответствуют тем же моментам выполнения исходного текста, что и на рис. 1.20 (здесь используются только первые символы имен процедур).

На рис. 1.24, *а* показана ситуация непосредственно перед началом активации *q(l, 3)*. Поскольку *QuickSort* имеет глубину вложенности 2, новая активация *q(1, 3)* модифицирует элемент дисплея *d*[2]. Это воздействие показано на рис. 1.24, *б*, где *d*[2] теперь указывает на новую запись активации; старое значение сохранено в новой записи активации. Сохраненное значение потребуется позже для восстановления дисплея в состояние, показанное на рис. 1.24, *а*, когда управление вернется в активацию *q(1, 9)*.

Дисплей изменяется, когда возникает новая активация, и должен быть восстановлен при возврате управления из нее. Правила области видимости языка Pascal и других языков с текстуальной областью видимости позволяют поддерживать дисплей с помощью следующих шагов.

Рассмотрим только простейший случай, когда процедура не передается в качестве параметра. При настройке записи активации для процедуры глубины вложенности *i* выполняются следующие действия.

1. Значение элемента *d*[*i*] сохраняется в новой записи активации.
2. Указатель на новую запись активации заносится в *d*[*i*].
3. Выполняется процедура.
4. Непосредственно перед завершением ее активации восстанавливается ранее сохраненное значение *d*[*i*].

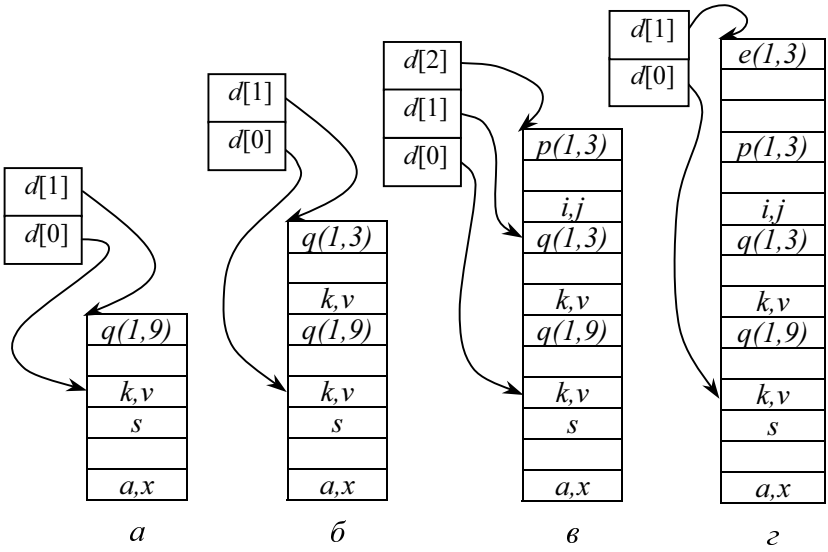


Рис. 1.24. Поддержка дисплея при обычном вызове процедур

Эти шаги обосновываются следующим образом. Предположим, что процедура с глубиной вложенности j вызывает процедуру глубины i. Возможны два случая, зависящие от того, вложена ли вызываемая процедура в исходном тексте в вызывающую или нет (что уже обсуждалось при рассмотрении связей доступа).

1. $j < i$. Тогда $i = j+1$, и вызываемая процедура вложена в вызывающую. Первые j элементов дисплея, таким образом, не требуют изменений, а элемент $d[i]$ устанавливается указывающим на новую запись активации. Этот случай иллюстрируется на рис. 1.24, *а*, где процедура *sort* вызывает *QuickSort*, и на рис. 1.24, *в*, где *QuickSort* вызывает процедуру *PartExch*.

2. $j > i$. В этом случае, как выяснялось ранее при рассмотрении связей доступа, все охватывающие процедуры с глубинами вложенности 1, 2, ..., i–1 и для вызывающей, и для вызываемой процедур должны быть одними и теми же. В этом случае старое значение $d[i]$ сохраняется в новой записи активации, а элемент дисплея $d[i]$ устанавливается указывающим на новую запись активации. Дисплей теперь можно корректно использовать, поскольку все его значимые элементы остаются неизменными (количество таких элементов равно i–1).

Пример второго случая с $i = j = 2$ приведен на рис. 1.24, *б*, где рекурсивно вызывается процедура *QuickSort*. Более интересный пример – когда активация *p (1, 3)* глубины вложенности 3 создает активацию *e(1, 3)* глубины вложенности 2. Охватывающая их процедура s имеет глубину вложенности 1 (см. рис. 1.21, *г*; напомним, что текст программы приведен на рис. 1.20). Обратите внимание на то, что при вызове *e (1,3)* значение *d*[3], принадлежащее *p (1, 3)*, остается в дисплее, хотя и не может быть доступно, пока управление находится в *e*. Если процедура *e* вызовет другую процедуру глубины вложенности 3, она сохранит значение *d*[3] и восстановит его при возврате управления из данной активации *e*.

Таким образом, мы можем показать, что из тела каждой процедуры виден корректный дисплей для всех глубин вложенности вплоть до ее собственной глубины. Заметим, что активация *q (1, 9)* также сохраняет значение элемента *d*[2], хотя может случиться так, что второй элемент дисплея никогда не использовался до этого и его не обязательно восстанавливать. Проще сохранять такие элементы во всех активациях процедур, чем в процессе выполнения программы при входе в любую процедуру принимать решение о нужности/ненужности такого сохранения.

Если в распоряжении транслятора (виртуальной машины) имеется достаточное количество быстрых регистров процессора, то дисплей, рассматриваемый как массив, может храниться в регистрах. Заметим, что транслятор всегда в состоянии определить максимально требуемую длину этого массива для транслируемой программной единицы: она равна максимальной глубине текстуальной вложенности процедур в программе.

Далее, зная количество регистров процессора, транслятор может принять решение – размещать ли дисплей в регистрах. Если использовать регистры по тем или иным причинам невозможно, то дисплей можно размещать в статически выделенной памяти. Тогда все обращения к содержимому записей активации за нелокальными объектами начинаются с использования косвенной адресации посредством соответствующего указателя дисплея.

Этот подход вполне обоснован для тех машин, в которых реализована косвенная адресация памяти, хотя каждое косвенное обращение и стоит цикла обращения к памяти. В принципе есть еще одна возможность (требующая несколько больших затрат времени) – хранить дисплей в стеке времени исполнения и создавать новую его копию при каждом входе в процедуру.

1.11.7. Динамическая область видимости

При использовании динамической области видимости каждая новая активация наследует существующую привязку нелокальных объектов к памяти. Нелокальный объект *a* в вызываемой активации использует ту же область памяти, что и в вызывающей активации. Новые связи устанавливаются только для локальных объектов вызываемой процедуры; они связываются с отведенной памятью в новой записи активации.

Программа, показанная на рис. 1.25, иллюстрирует динамическую область видимости.

Процедура *show* (записанная в строках (3) и (4)) выводит значение нелокальной переменной *r*. При использовании обычной текстуальной области видимости языка Pascal нелокальная переменная *r* располагается в области видимости объявления в строке (2).

Результаты работы программы, очевидно, будут следующими:

0.25 0.25
0.25 0.25

Однако если бы реализовывалась динамическая область видимости, то программа вывела бы совсем другие результаты:

0.25 0.125
0.25 0.125

Когда процедура *show* вызывается в строках (10) и (11) главной программы, ею выводится 0.25, поскольку используется переменная *r*, локальная по отношению к главной программе. Однако при вызове *show* в строке (7) из процедуры *modify* выводится значение 0.125, поскольку теперь используется переменная *r*, локальная по отношению к *modify*.

```
(1)  program dynamic(input, output);
(2)    var r: real;
(3)    procedure show;
(4)        begin write(r) end;
(5)    procedure modify;
(6)        var r: real;
(7)        begin r := 0.125; show end;
(8)    begin
(9)        r := 0.25;
(10)       show; modify; writeln;
(11)       show; modify; writeln
(12)   end.
```

Рис. 1.25. Программа для иллюстрации динамической области видимости

Для реализации динамической области видимости используются следующие два подхода, имеющие некоторое сходство с применением связей доступа и дисплеев при реализации текстуальной области видимости.

1. *Глубокий доступ (deep access)*. Динамическая область видимости образуется в том случае, если для каждой записи активации связь по доступу указывает на ту же запись активации, что и связь по управлению. Простейшая ее реализация состоит в том, чтобы обойтись вообще без связей по доступу и использовать уже имеющиеся связи по управлению для поиска в стеке ближайшей записи активации, содержащей значение требуемого нелокального объекта. Поиск, очевидно, должен производиться по имени объекта, а следовательно, каждая запись активации должна содержать ссылку на таблицу, строками которой являются пары «имя–смещение» для всех локальных объектов данной процедуры.

Термином «глубокий доступ» этот метод назван потому, что местоположение нужного объекта заранее неизвестно и процесс поиска может неопределенно «глубоко» погрузиться в стек. Глубина поиска зависит от входных параметров программы и не может быть определена в процессе компиляции.

2. *Мелкий доступ (shallow access)*. В этом случае абсолютно каждому объекту программы ставится в соответствие его собственная область в статически выделяемой памяти. В этой области хранится текущее значение объекта. При создании любой новой активации процедуры *p* каждый ее локальный объект, объявленный в *p*, ассоциируется со своей статически выделенной памятью, но предварительно сохраняются значения всех локальных объектов предыдущей активации этой процедуры в текущей записи активации. Естественно, в процессе исполнения возвращающей последовательности обеспечивается восстановление всех этих значений из уничтожаемой записи активации в статическую память.

Разница между двумя подходами по их временным характеристикам состоит в том, что при глубоком доступе каждое обращение к нелокальному объекту осуществляется дольше, но при этом нет никаких накладных расходов, связанных с началом и окончанием активации.

Мелкий же доступ обеспечивает быстрое непосредственное обращение к нелокальным переменным, но при этом затрачивается время на копирование их значений в моменты начала и окончания каждой активации вызываемой процедуры. Если допускается передача функций как параметров и возвращение их как результатов, что требует неявной передачи связей по доступу, то глубокий доступ обеспечивает более быструю реализацию динамической видимости.

1.12. Передача параметров

Взаимодействие между вызывающей и вызываемой процедурами по данным обеспечивается путем использования не только нелокальных для вызываемой процедуры объектов, но и фактических параметров, формируемых вызывающей процедурой. Вспомним, например, процедуру *Exchange*, являющуюся частью программы на рис. 1.20. В этой процедуре используются как нелокальные имена, так и параметры (нелокальным именем по отношению к процедуре является массив *arr*, а параметрами – переменные *iFirst* и *iSecond*).

Существует несколько различных методов сопоставления фактических и формальных параметров:

– передача по значению;
– передача по ссылке;
– копирование-восстановление;
– передача по имени;
– макрорасширения.

Большое количество различных методов передачи параметров возникло исторически вследствие возможности различной интерпретации смысла выражений и наименований объектов. В инструкции типа *arr[i]:=arr[j]* выражение *arr[j]* представляет значение (*r-value*), в то время как выражение *arr[i]* определяет адрес памяти, куда должно попасть значение *arr[j]*, т. е. *arr[i]* – это *l-value*.

Решение о том, каким образом трактовать выражение – как значение или адрес памяти, определяется тем, где по отношению к знаку операции присваивания располагается выражение – слева или справа. Различия между способами передачи параметров определяются в первую очередь тем, чем является фактический параметр: *l-value*, *r-value* или попросту текстом, представляющим только самое себя.

1.12.1. Передача по значению

Этот способ, по существу, представляет собой простейший метод передачи параметров. Вызывающая процедура осуществляет вычисление значения фактического параметра, и полученное *r-value* передается вызываемой процедуре. Таким образом происходит передача параметров в языке C, многих реализациях языка Pascal и ряде других языков. Метод передачи по значению реализуется следующим образом.

1. Формальный параметр рассматривается как локальный объект, так что память для него выделяется в записи активации вызываемой процедуры.

2. Вызывающая процедура вычисляет значения фактических параметров и помещает их в память, выделенную для формальных параметров.

Отличительная особенность этого метода заключается в том, что все последующие операции над формальными параметрами не влияют на значения объектов, имена которых фигурируют в вызове в качестве фактических параметров. Если удалить ключевое слово *var* в строке (3) на рис. 1.26, то транслятор с языка Pascal обеспечит передачу параметров *x* и *y* в процедуру *swap* по значению. В этом случае вызов *swap (a, b)* в строке (11) оставит значения *a* и *b* нетронутыми.

```
(1)  program ByRef_ByVal(input, output);
(2)      var a, b: integer;
(3)      procedure swap(var x, y: integer);
(4)          var temp: integer;
(5)          begin
(6)              temp := x;
(7)              x := y;
(8)              y := temp;
(9)          end;
(10)     begin
(11)         a := 1; b := 2;swap (a, b) ;
(12)         writeln('a =', a); writeln('b =', b)
(13)     end.
```

Рис. 1.26. Pascal-программа для иллюстрации передачи параметров по ссылке/значению

При передаче параметров по значению вызов *swap (a, b)* в этой программе эквивалентен следующей последовательности действий:

x : = a; y := b; temp := x; x := y; y := temp;

Здесь *x*, *y* и *temp* являются объектами, локальными по отношению к инструкциям тела процедуры *swap*. Хотя приведенные присвоения и изменяют значения локальных переменных, эти изменения окажутся потерянными при возврате управления из вызова и уничтожении записи активации для процедуры *swap*. Таким образом, этот вызов не повлияет на объекты из записи активации вызывающей программы, значения которых переданы в качестве параметров в вызываемую процедуру.

Процедура с передачей параметров по значению может влиять на объекты вызывающей процедуры либо посредством использования имен нелокальных объектов (см., например, процедуру *Exchange* на рис. 1.20), либо посредством использования указателей с явной пере-

дачей их по значению. В С-программе (рис. 1.27) *x* и *y* объявлены в строке (2) как указатели на целые числа.

```
(1)     swap (int *x, int *y) {
(2)     int temp
(3)         temp = *x; *x = *y; *y = temp;
(4)     }
(5)     main() {
(6)     int i,arr[11];
(7)         for(i=0;i<11;i++)arr[i]=0;
(8)         i=10;swap ( &i, &arr[i]) ;
(9)         printf("i = %d, arr[i] = %d\n",i,arr[i]);
(10)  }
```

Рис. 1.27. С-программа, использующая передачу указателей по значению

Оператор *&* в вызове *swap (&a, &b)* в строке (8) приводит к тому, что в *swap* передаются указатели на *a* и *b*.

Результатом работы этой программы будет: *i* = 2, *arr[i]* = 1.

Следует помнить, что в языке С любой нелокальный объект процедуры/функции может быть только статическим. Это значит, что переменная *i* и массив *arr* (локальные объекты функции *main*) никаким другим способом, кроме доступа через указатели, не могут быть изменены в теле функции *swap.*

1.12.2. Передача по ссылке

При передаче параметров по ссылке (известной также как передача по адресу) вызывающая процедура передает вызываемой указатель на значение фактического параметра, т. е. его *l-value*.

В этом случае при обработке транслятором оператора вызова процедуры возможны две различные ситуации.

1. Если фактический параметр представляет собой имя или выражение, имеющее *l-value*, то вызываемой процедуре передается *l-value*.

2. Если фактический параметр представляет собой выражение (например, *a+b*), не имеющее *l-value*, то результат вычисления выражения размещается в отдельном месте памяти (поле результатов промежуточных вычислений в записи активации, см. подразд. 1.8.2), адрес которого и передается процедуре. Транслятор должен построить машинные инструкции, обеспечивающие выполнение этих действий.

В зависимости от используемого языка программирования могут действовать дополнительные ограничения. Так, например, язык C запрещает инициализацию неконстантной ссылки временным объектом.

Обращения к формальному параметру в вызываемой процедуре превращаются в целевом коде в косвенные обращения через полученный указатель.

Рассмотрим процедуру swap, приведенную на рис. 1.26. Вызов *swap* с фактическими параметрами *i* и *arr[i]* приведет к тому же результату, что и следующая последовательность шагов.

1. Скопировать адреса (*l-value*) *i* и *arr[i]* в запись активации вызываемой процедуры (локальные объекты, выделяемые для хранения этих адресов будем называть далее как *arg1* и *arg2*).

2. Установить значение *temp* равным содержимому ячейки памяти, на которую указывает *arg1* (т. е. установить *temp* равным 10, где 10 – начальное значение *i*). Это соответствует присвоению *temp: =*x* в строке (3) в определении *swap*.

3. Установить содержимое ячейки, на которую указывает *arg1*, равным значению, хранящемуся по адресу, на который указывает *arg2* (т. е. выполнить *i:=arr[10]*). Это аналогично присвоению ***x: = *y*** в строке (3) процедуры *swap*.

4. Установить содержимое ячейки, на которую указывает *arg2*, равным значению *temp*, т. е. выполнить *arr[10]=i*. Это соответствует присваиванию **y: =temp*.

Передача параметров по ссылке используется во многих языках программирования. В языке Pascal по ссылке передаются параметры, описанные с использованием ключевого слова *var* в определении типа формального параметра. В большинстве языков массивы обычно передаются по ссылке.

1.12.3. Копирование-восстановление

Гибридом ранее описанных способов передачи является «копирование-восстановление», известное также как «копирование внутрь и наружу» (*copy-in – copy-out*), и «значение-результат» (*value-result*). В этом случае выполняются следующие действия.

1. Перед передачей управления вызываемой процедуре определяются фактические параметры и *r-value* фактических параметров передаются вызываемой процедуре по значению. Однако в дополнение к этому перед вызовом определяются и сохраняются в записи активации вызывающей процедуры *l-value* тех фактических параметров, которые их имеют.

2. После возвращения управления из вызываемой процедуры в вызывающей процедуре текущие *r-value* формальных параметров заносятся в память, занимаемую фактическими параметрами с использованием их *l-value*, вычисленных перед вызовом. Естественно, копируются только те фактические параметры, для которых существует *l-value* (т. е. объекты, имеющие адрес в памяти; регистровые переменные, например, такого адреса не имеют, но их и не нужно копировать/восстанавливать).

Первый шаг выполняет «копирование внутрь» значений фактических переменных в запись активации вызываемой процедуры (в поле формальных параметров). Второй шаг состоит в «копировании наружу» окончательных значений формальных параметров в запись активации вызывающей процедуры (по *l-value*, вычисленным для фактических переменных перед вызовом).

Заметим, что вызов *swap(i, arr[i])* в последнем примере будет корректно работать и при использовании копирования-восстановления, поскольку адрес элемента массива *arr[i]* вычисляется и сохраняется вызывающей процедурой перед вызовом. Таким образом, окончательное значение формального параметра *y*, которое является исходным значением *i*, копируется в нужную ячейку памяти, хотя положение *arr[i]* и изменилось при вызове в связи с изменением значения *i*.

«Копирование-восстановление» используется некоторыми реализациями Fortran (хотя другие предпочитают передачу параметров по ссылке). Различия между этими двумя методами могут проявиться, если вызываемая процедура имеет несколько вариантов обращения к записи активации вызывающей процедуры. Например, согласно рис. 1.28 активация процедуры *unsafe(a)*, создаваемая при вызове из строки (6), обращается к переменной *a* и как к нелокальной переменной, и как к формальному параметру *x*.

```
(1) program copyout(input, output)
(2)     var a: integer;
(3)     procedure unsafe(var x: integer);
(4)         begin x := 2; a := 0 end;
(5)     begin
(6)         a := 1; unsafe(a); writeln(a)
(7)     end.
```

Рис. 1.28. Иллюстрация различий передачи параметров по ссылке и копированием-восстановлением

Если реализуется передача параметров по ссылке, то присвоение значений как *x*, так и *a* непосредственно изменяет *a*, что дает в резуль-

тате значение *a*, равное 0. При использовании метода «копирование-восстановление» значение *a*, перед вызовом равное 1, копируется в формальный параметр *x*. Значение *x*, равное 2, копируется по *l-value a* непосредственно перед возвратом управления, так что окончательным значение *a* будет 2.

1.12.4. Передача по имени

Передача параметров по имени традиционно определяется следующими правилами языка Algol-60, в котором она впервые была реализована.

1. Процедура рассматривается как макрос, т. е. ее тело в момент вызова так подставляется (будто бы) в текст вызывающей процедуры, что имена формальных параметры побуквенно заменяются именами фактических. Такую буквальную подстановку называют макрорасширением (*macro-expansion*), или подстановкой тела (*in-line expansion*). При этом заметим, что *in-line expansion* при передаче параметров по имени и *inline*-функции языка С представляют собой два разных понятия, которые не следует путать.

2. Локальные объекты вызываемой процедуры содержатся отдельно от объектов вызывающей процедуры. Можно считать, что каждый локальный объект вызываемой процедуры систематически переименовывается и получает уникальное имя перед макрорасширением.

3. Для сохранения целостности все фактические параметры заключаются в скобки (например, фактический параметр вида *a+b* будет передаваться как *(a+b)*).

Вызов *swap(i,arr[i])* теперь, по-существу, будет реализован в виде следующего кода:

```
temp :=(i);
i := (arr[i]);
arr[i] := temp;
```

Таким образом, при передаче параметров по имени *swap*, как и ожидалось, устанавливает *i* равным *arr[i]*, но при этом неожиданно устанавливает равным 10 (где 10 – начальное значение *i*) элемент массива *arr[arr[10]]*, а не *arr[10]*.

Это происходит потому, что значение *x* в присвоении *x: = temp* в процедуре *swap* не вычисляется до тех пор, пока оно не становится необходимым, а к тому времени значение *i* уже изменено. Эту особенность метода передачи по имени (эффект *inline-expansion*) необходимо учитывать при разработке процедур с параметрами.

Хотя передача параметров по имени представляет в первую очередь теоретический интерес, концептуально близкая технология подстановки тела может уменьшить время работы программы. При вызове процедур неизбежны определенные накладные расходы, связанные с выделением памяти для записи активации, сохранения состояния машины, настройки связей и передачи управления.

Для небольших процедур размер кода, связанного с этими расходами, и затраты времени на его выполнение могут превысить размер самого тела процедуры и время его работы. Существенно более эффективным может оказаться использование подстановки тела вызываемой процедуры в код вызывающей процедуры, даже если при этом объем кода программы немного увеличится. В следующем примере такое встраивание кода применяется к процедуре, использующей передачу параметров по значению.

Предположим, функция *f* в присвоении *x:=f(A)+f(B)* использует передачу параметров по значению. Здесь фактические параметры *A* и *B* могут представлять собой выражения. Подстановка выражений *A* и *B* вместо всех появлений формальных параметров в теле *f* приводит к передаче параметров по имени, от чего может возникнуть эффект *inline-expansion*, как в процедуре *swap*.

Для того чтобы избавиться от этого эффекта, можно применить вычисление значений фактических параметров перед выполнением тела процедуры с неявным образованием временных переменных и размещением их в поле промежуточных результатов записи активации. Тогда реализация оператора *x:=f(A)+f(B)* будет выглядеть так:

```
t1 := A; t2 := B;
t3 := f (t1) ; t4 := f (t2);
x := t3 + t4;
```

Теперь при подстановках тела функции *f* вхождения ее формального параметра заменяются на *t1*и *t2* соответственно, что и позволяет избавиться от эффекта *inline-expansion.*

Обычным методом реализации передачи параметров по имени является создание компилятором специальных подпрограмм без параметров (так называемых *thunks*), которые могут вычислить *l-value* и *r-value* фактических параметров в локальной среде ссылок вызывающей процедуры.

В момент вызова вызывающая процедура передает эти подпрограммы вместо параметров вызываемой процедуре. Вызываемая процедура запускает *thunk* каждый раз, когда в ней согласно тексту должен выполняться доступ к фактическому параметру, и в зависимости

от того, должно ли быть выполнено чтение или запись, использует либо *r-value,* либо *l-value*, сформированное соответствующим *thunk*.

Как обычно делается в языках с текстуальной областью видимости при передаче процедур в качестве параметров, запись активации процедуры *thunk* содержит связь доступа, указывающую на текущую запись активации вызывающей процедуры.

1.13. Функции контроля структуры транслируемой программы

Кроме всех рассмотренных в этой главе задач семантический анализатор транслятора может реализовывать ряд дополнительных, иногда довольно экзотических функций, обусловленных спецификой конкретного языка программирования. Назовем только две из них, для того чтобы дать о них самое общее представление.

1.13.1. Проверка взаимных связей между словами

Некоторые языки программирования требуют выполнения проверки правильности взаимных связей между употреблениями некоторых слов, предназначенных для контроля смысла программы. В качестве примера можно привести использование одноименной переменной в заголовке и в завершителе цикла на языке (Visual) Basic:

for i=1 to 10

...

next i

Употребление имени индекса цикла в его последнем операторе способствует улучшению визуального контроля текста программы в случае наличия вложенных циклов.

1.13.2. Контроль количества слов

Иногда транслятор должен осуществлять контроль количества употребления некоторых слов в тексте. Например, если в программе на языке Object Pascal имеется оператор *try*, то должно встретиться хотя бы одно слово *except* или *finally*, но слово *finally* может встретиться не более одного раза для каждого *try*:

try ... except ... except ... finally ... end

В языках C++ и Java есть аналогичная конструкция:

try {...} catch{...} ... finally {...}.

1.14. Семантический анализ: краткое заключение

Итак, основные задачи семантического анализа:

– выявление связей каждой операции с ее операндами и объектом, который будет хранить результат этой операции;

– проверка возможности исполнения операции над значениями операндов в соответствии с их типами;

– определение способа преобразования исходных данных каждой операции в значение результата.

Для решения этих задач приходится обнаруживать объявления данных в тексте программы, формировать и сохранять информацию о типах данных, отслеживать текстуальную или динамическую видимость объектов, обеспечивать доступ к статическим (глобальным), локальным и нелокальным объектам каждой процедуры/функции, выявлять эквивалентность типов и на этой основе принимать решения о допустимости применения операций к их операндам.

В результате всех этих и многих других не упомянутых в данном перечне действий обычно приходится формировать некоторое множество дополнительных операций, не указанных явно в тексте исходной программы. В это множество в зависимости от семантики языка входят все или некоторые из нижеперечисленных операций, предназначенные:

1) для создания записи активации (в том числе локальных данных и не объявленных в исходной программе переменных, предназначенных для хранения промежуточных результатов вычислений) каждой вызываемой процедуры и установления ее связей с записями активации других процедур;

2) обеспечения доступа из процедур к нелокальным для них объектам (управление дисплеем при реализации текстуальной видимости, перемещение значений при мелком и поиск при глубоком доступе в том случае, если реализуется динамическая видимость);

3) преобразования значений операндов к типу, требуемому для выполнения каждой операции;

4) перемещения обрабатываемых значений в ту память, которая непосредственно доступна для выполнения операции (например, в регистры процессора), и результатов вычислений обратно в основную память;

5) уничтожения записи активации (в том числе локальных и временных объектов) процедуры в момент возврата из нее.

Все эти дополнительные операции могут формироваться либо путем модификации постфиксной записи, либо, что обычно бывает более удобным, при создании очередного промежуточного представления программы, которое строится уже генератором кода и называется последовательностью тетрад.

2. ГЕНЕРАЦИЯ И ОПТИМИЗАЦИЯ КОДА

Генератор объектного кода выполняет последнюю стадию преобразования текста транслируемой программы. Обычно к нему предъявляются весьма жесткие требования. Результат его работы должен быть прежде всего корректным, т. е. эквивалентным исходной программе, и оптимальным, т. е. эффективно использующим ресурсы (процессорное время и память) целевой машины. Кроме того, эффективно, т. е. быстро, должен работать и сам генератор кода.

Задача генерации оптимального кода математически неразрешима. Поэтому генераторы кода используют эвристические технологии построения и оптимизации целевой программы, позволяющие получить хороший, но не обязательно оптимальный код.

Исходными данными генератора кода обычно являются:

– промежуточное представление исходной программы, полученное в результате работы предыдущих стадий трансляции;

– таблица идентификаторов (точнее – объектов, объявленных и/или используемых в программе), содержащая накопленные к моменту генерации кода атрибуты объектов.

Будем считать, что перед генерацией кода предыдущими стадиями трансляции было обеспечено выявление типов объектов и формирование дополнительных операций для преобразования типов. Поэтому в дальнейшем предполагается, что значения объектов в промежуточном представлении являются величинами, с которыми процессор целевой машины может оперировать непосредственно (биты, байты, целые и действительные числа, указатели и т.п.). Предполагается также, что были выполнены все необходимые проверки типов, т. е. выявлены все очевидные семантические ошибки (типа использования действительного числа в качестве индекса массива).

Таким образом, стадию генерации кода можно рассматривать в предположении, что ее входной поток не содержит ошибок. Тем не менее, как уже упоминалось, во многих трансляторах семантический анализ выполняется одновременно (или квазиодновременно) с генерацией кода.

Главной задачей генератора объектного кода является формирование образа памяти транслируемой программы. Под образом памяти понимается совокупность начальных значений всех (как исполняемых, так и обрабатываемых, как объявленных в тексте, так и добавленных в процессе трансляции) объектов программы. Обычно образ памяти программы, сформированный во время трансляции, хранится как файл во внешней памяти компьютера и переносится (копируется) в оперативную память загрузчиком операционной системы в момент запуска программы на исполнение. Некоторые возможные варианты внутреннего устройства образа памяти программы вкратце обсуждались в разд. 1.8 (см. рис. 1.7). Здесь следует напомнить, что в формировании значений, образующихся в стеке, куче и области подгрузки, генератор кода явным образом не участвует. Значения объектов, принадлежащих этим областям образа памяти задачи, формируются динамически при исполнении программы. Однако генератор кода отвечает за создание тех последовательностей инструкций (в частности, вызывающих последовательностей), которые сами являются частями статически формируемых областей, но занимаются образованием динамических областей, например записей активации процедур.

В том случае, когда языком программирования допускается раздельная трансляция, генератор кода должен построить образ памяти не всей программы, а только той части, которая транслируется в данный момент (как уже отмечалось, исполняемая программа, как правило, состоит из раздельно транслируемых частей даже в том случае, если язык этого не допускает). В силу этого при дальнейшем рассмотрении будем предполагать, что генератор кода должен формировать только некоторую часть полного образа памяти исполняемой программы, которая в последующем будет объединяться с другими частями, имеющими однотипное внутреннее строение.

Следовательно, общую задачу генерации кода можно рассматривать как совокупность раздельно решаемых задач формирования выполняемых инструкций и локальных данных каждой транслируемой процедуры, а также статических данных программы в целом.

Если отвлечься от назначения формируемых областей, то можно легко обнаружить, что для создания каждой из них нужно выполнять однотипную последовательность действий.

1. Образовать указатель адреса первого незанятого байта.

2. Определить начальное значение указателя (обычно это условный ноль).

3. Для каждого объекта, принадлежащего данной области:
   - текущее значение указателя считать адресом объекта (и записать его в качестве атрибута адреса в таблицу идентификаторов, если формируется объект данных);
   - определить размер объекта в единицах измерения памяти и сформировать полностью или частично его начальное значение (выполняется по-разному для объектов различных областей);
   - занести построенное значение объекта в формируемую область по его адресу;
   - увеличить указатель занятой части области на размер объекта.

Вполне возможна и даже наиболее вероятна ситуация, когда при работе генератора кода в стадии формирования одновременно находятся несколько разных областей. Именно поэтому на втором шаге в качестве начального адреса памяти каждой формируемой области используется условный ноль. Истинное значение начального адреса любой области может быть определено только после того, как будут сформированы все области программы. Только тогда можно будет разместить их в том или ином порядке внутри линейной памяти образа программы.

Наибольшие трудности в вышеописанной последовательности действий возникают при определении размера и формировании начального значения объекта. Дело в том, что значение, а следовательно, и размер многих исполняемых объектов (машинных инструкций), может зависеть от того, в какой области памяти и по какому адресу располагаются их операнды.

В принципе возможны ситуации, когда в момент формирования некоторой инструкции еще неизвестны адреса ее операндов. Наихудший случай – порочный круг так называемых переходов вперед – возникает тогда, когда операнд инструкции перехода располагается в той же самой области памяти, но после этой инструкции. Значение, а следовательно, и размер инструкции перехода, определяются неизвестным (в момент формирования инструкции) адресом операнда, который, в свою очередь, может зависеть от ее же размера.

Для разрешения возникающих трудностей стадию генерации кода обычно реализуют в виде двух или более проходов по тексту программы или одному из ее промежуточных представлений. Это позволяет разделить во времени распределение памяти областей программы (связывание объектов и адресов занимаемой ими памяти) и формирование

значений объектов и хорошо согласуется с потребностями построения оптимального кода (для улучшения степени оптимизации могут быть реализованы три и более прохода).

На первом из проходов генерации кода транслируемая программа обычно из постфиксной записи преобразуется в очередное внутреннее представление – промежуточный код.

2.1. Промежуточный код

Семантические проверки и генерация объектного кода в принципе могут осуществляться путем обработки синтаксического дерева операций или постфиксной формы записи как промежуточного представления транслируемой программы. Однако их более удобно выполнять с использованием еще одной промежуточной формы представления программы – последовательности тетрад (или триад). Под тетрадой можно понимать одну команду условной трехадресной машины, в общем случае включающую наименование кода операции, наименования двух операндов и наименование результата. Удобство использования тетрад для семантического анализа состоит в том, что наименования всех необходимых для выполнения проверок объектов программы локализованы в пределах одной тетрады. В отличие от этого при использовании для семантического анализа как постфиксной записи, так и дерева операций необходимую для выполнения проверок информацию приходится извлекать путем дополнительных просмотров ПФЗ или обходов дерева.

При преобразовании постфиксной записи или дерева операций в последовательность тетрад становится явной задача размещения промежуточных результатов вычислений, которую рано или поздно приходится решать как при компиляции, так и при интерпретации.

2.1.1. Промежуточные результаты вычислений

Рассмотрим существо этой задачи на простейшем примере. Пусть в программе на языке С имеется оператор присваивания

$$a = b * c + d / e ;$$

Постфиксная форма записи этого оператора выглядит так:

$$a\ b\ c\ *\ d\ e\ /\ +\ =$$

Результат выполнения первой операции умножения значения *b* на значение *c* является промежуточным значением, для хранения которого в исходной программе в явном виде не предусмотрено никакой переменной. Тем не менее это значение должно где-то храниться до того момента, когда будет выполняться операция сложения. Необходимость этого можно обнаружить, если переписать постфиксную запись:

$$r\ b\ c\ *\ =$$

$$a\ r\ d\ e\ /\ +\ =$$

Прежде всего, заметим, что такое изменение постфиксной записи эквивалентно с точки зрения получаемых результатов. Действительно, именно такую ПФЗ можно получить из двух операторов присваивания, выполнение которых даст точно такое же значение объекта *a*, что и исходный оператор присваивания (естественно, при одинаковых значениях переменных *b, c, d* и *e*):

$$r = b * c;$$

$$a = r + d / e;$$

Дополнительно к объектам *a, b, c, d* и *e*, использованным в исходном операторе присваивания, в преобразованной ПФЗ появился новый объект (переменная) *r*, предназначенный для хранения промежуточного результата вычислений. Обозначение *r* нельзя понимать как наименование объекта, точно такое же, как и обозначения объектов *a, b, c, d* и *e*. Для объекта *r* должно быть сконструировано наименование, которое нужно добавить в таблицу идентификаторов транслируемой программы, а следовательно, оно не должно совпадать ни с одним из наименований объектов, объявленных в тексте. Это требование не создает заметных трудностей, для его удовлетворения достаточно определить правила образования наименований промежуточных результатов, не совпадающие с правилами образования идентификаторов в данном языке.

Точно так же, как с операцией умножения, придется поступить и с операцией деления, в результате чего исходная ПФЗ будет преобразована к виду

$$r\ b\ c\ *\ =$$

$$p\ d\ e\ /\ =$$

$$a\ r\ p\ +\ =$$

Для хранения промежуточного результата операции деления образовано обозначение еще одного объекта *p*. В данном случае для этой цели не может быть использован ранее образованный объект *r*, поскольку его значение потребуется позже при выполнении операции сложения, однако в принципе повторное использование таких объектов возможно. Оно позволяет уменьшить размер поля результатов промежуточных вычислений записи активации процедуры.

Как правило, задача ликвидации избыточных объектов решается не на стадии их создания, а позже, в процессе оптимизации кода (см. разд. 2.3).

2.1.2. Понятие псевдокода

Последняя построенная постфиксная форма записи имеет вполне систематический вид. Это последовательность трех ПФЗ, каждая из которых состоит из пяти элементов:

1) наименование результата выполнения ПФЗ;
2) наименование первого операнда;
3) наименование второго операнда;
4) наименование операции преобразования значений операндов;
5) наименование операции занесения вычисленного значения в результат.

Перепишем каждую такую ПФЗ в обратной последовательности, т. е. начиная с последнего элемента и заканчивая первым, отбросим при этом присутствующий в каждой ПФЗ знак операции присваивания и представим получаемую запись в табличном виде (табл. 2.1).

Таблица 2.1

Номер	Операция	Операнд1	Операнд2	Результат
1	*	*c*	*b*	*r*
2	/	*e*	*d*	*p*
3	+	*p*	*r*	*a*

Эту запись можно считать промежуточным представлением исходного оператора присваивания в форме последовательности тетрад. Она получена путем разбиения постфиксной записи на фрагменты, соответствующие вычислению значений подвыражений в исходном операторе. Ниже мы увидим, что тетрады, получаемые формальными методами, могут отличаться от построенных неформально. Для рассматриваемого

примера чисто формальные методы выдадут в качестве результата последовательность тетрад, показанную в табл. 2.2.

Таблица 2.2

Номер	Операция	Операнд1	Операнд2	Результат
1	*	*c*	*b*	*r*
2	/	*e*	*d*	*p*
3	+	*p*	*r*	*s*
4	=	*s*	*null*	*a*

Существуют алгоритмы прямого преобразования ПФЗ в последовательность тетрад с использованием стека, не выполняющие предварительного упрощения постфиксной записи, один из таких алгоритмов приведен в подразд. 2.1.3.

Для реализации таких алгоритмов должна быть известна арность каждого знака операции, т. е. количество операндов, необходимых для выработки результата. Для нульарных, унарных и бинарных знаков операций в результате преобразования порождается по одной тетраде. Знаками операций большей арности (3, 4, …) в языках программирования являются наименования процедур и функций.

Выполнение каждой такой операции требует предварительного установления соответствия между фактическими и формальными параметрами процедуры/функции (передачи значения параметров). Соответственно для каждого операнда такого знака операции формируются тетрады, обеспечивающие присвоение значения фактического параметра формальному параметру. Исключение может быть сделано для тех двух параметров, значения которых могут передаваться путем их включения в тетраду вызова процедуры/функции. Заметим, что для разных методов передачи параметров дополнительные тетрады, встраиваемые транслятором в точки вызова, будут существенно различаться (см. разд. 1.12). Реально, как рассматривалось в разд. 1.10, для исполнения *k*-арного знака операции транслятор должен построить две последовательности тетрад, включаемые и в вызывающую, и в вызываемую процедуры.

2.1.3. Преобразование постфиксной записи в псевдокод

Шаг 1. Подготовка. Очистка стека, установка первого слова постфиксной записи в качестве текущего.

Шаг 2. Выявление назначения текущего слова ПФЗ. Если это наименование операнда, то переход к шагу 3, иначе (знак операции) – к шагу 4.

Шаг 3. Текущее слово заносится в стек и удаляется со входа (текущим становится следующее слово ПФЗ). Возврат к шагу 2.

Шаг 4. Знак операции удаляется со входа и заносится в тетраду. Определяется арность этого знака операции, далее формируются наименования операндов тетрады согласно выявленному случаю:

Случай 0 – оба наименования операндов устанавливаются равными *null*.

Случай 1 – с верхушки стека снимается одно наименование и заносится в тетраду в качестве первого операнда, в качестве второго операнда устанавливается *null*.

Случай 2 – с верхушки стека снимаются два наименования и заносятся (в порядке извлечения из стека) в тетраду в качестве операндов.

Случай $k>2$ – с верхушки стека снимаются k слов, для каждого строится вспомогательная тетрада со знаком операции присваивания, снятым со стека наименованием в качестве первого операнда, значением *null* в качестве второго операнда и наименованием формального аргумента процедуры/функции в качестве результата. Каждая вспомогательная тетрада выдается на выход. После этого формируется тетрада с наименованием процедуры/функции в качестве знака операции и пустыми (*null*) наименованиями операндов.

Если для знака операции последней сформированной тетрады подразумевается образование промежуточного результата вычислений, то создается, заносится в таблицу идентификаторов и записывается в тетраду наименование результата, иначе в качестве наименования операнда используется последний снятый со стека операнд. Построенная тетрада выдается на выход.

Шаг 5. Если был достигнут конец входной ПФЗ, то останов процесса преобразования, иначе – переход к шагу 2. Конец алгоритма.

Следует отметить, что этим алгоритмом не предусматривается никаких проверок корректности входной постфиксной записи. ПФЗ формально является корректной, если для любого знака операции на ша-

ге 4 из стека удается извлечь соответствующее его арности количество наименований операндов и если в момент завершения преобразования стек оказывается пустым. Предполагается, что ПФЗ строится в процессе синтаксического анализа и является правильной, если транслируемая программа не содержит синтаксических ошибок. В том случае, если это предположение несправедливо, легко можно дополнить данный алгоритм соответствующими проверками состояния стека.

Кроме того, заметим, что для случая *k*-арного знака операции при $k > 2$ возможна модификация соответствующего случая шага 4 алгоритма, согласно которой строится $k - 2$ вспомогательных тетрады, а 2 последних извлекаемых из стека слова используются в качестве наименований операндов последней формируемой тетрады. В рассматриваемом ниже примере это привело бы к исчезновению тетрад 3 и 4 и замене обозначений *null* на *x* в поле операнд1 и на *y* в поле операнд2 тетрады.

Проиллюстрируем работу этого алгоритма на примере обработки ПФЗ оператора присваивания, в котором выражение содержит функцию *d* с аргументами *x*, *y* и *z*:

$$a = b * c + d (x, y, z);$$

ПФЗ этого оператора будет выглядеть так (слово *d* теперь следует понимать как 3-арный знак операции, а не как наименование операнда; именно поэтому в постфиксной записи оно находится после своих операндов *z, y* и *x*):

$$a\ b\ c\ *\ x\ y\ z\ d\ +\ =$$

В результате ее обработки будет получена последовательность тетрад, приведенная в табл. 2.3.

Таблица 2.3

Номер	Операция	Операнд1	Операнд2	Результат
1	*	*c*	*b*	*tmpValue1*
2	=	*z*	*null*	<имя аргумента 3>
3	=	*y*	*null*	<имя аргумента 2>
4	=	*x*	*null*	<имя аргумента 1>
5	*d*	*null*	*null*	*tmpValue2*
7	+	*tmpValue2*	*tmpValue1*	*tmpValue3*
8	=	*tmpValue3*	*null*	*a*

Вторая, третья и четвертая тетрады этой последовательности содержат в качестве второго операнда условные обозначения вида <имя аргумента *i*>. Тетрады такого типа предназначены для установления соответствия между значениями фактических параметров и наименованиями формальных аргументов процедур/функций. Эта задача обсуждалась в главе 1, здесь мы условно считаем, что параметры передаются по ссылке.

Тетрады, не вырабатывающие промежуточного результата, используемого в других тетрадах, содержат в последнем поле обозначение *null*. В некоторых трансляторах наименование промежуточного результата формируется независимо от того, используется ли он впоследствии. Образованные таким путем ненужные объекты могут удаляться на стадии оптимизации программы или в процессе генерации ее объектного кода.

Последовательность тетрад, получаемая в результате преобразования постфиксной записи, безусловно, не может считаться оптимальным представлением программы ни по объему памяти (пусть даже условному, исчисляемому суммой количества тетрад с количеством строк в таблице идентификаторов), ни по затратам времени (на данный момент их можно оценить как количество шагов выполнения тетрад). Не следует считать, что уменьшение количества тетрад (например, в упомянутом выше случае помещения последних двух операндов в тетраду *k*-арного знака операции) дает лучшее качество промежуточного представления программы. Вопросами оптимизации программы можно заниматься только после того, как будут закончены все проверки ее правильности, поскольку не имеет смысла оптимизировать некорректную программу, код которой никогда не будет выполняться.

2.1.5. Базовые блоки и графы потоков

Линейная структура последовательности тетрад (триад) строго соответствует линейной природе оперативной памяти компьютера, в которой должны размещаться генерируемые инструкции, и поэтому является основой для алгоритмов формирования образа памяти программы. Тем не менее для многих составных элементов этих алгоритмов (распределение регистров, определение размеров записей активации, оптимизация различной степени) очень важна информация о том, до какой (каких в случае разветвляющегося алгоритма транслируемой программы) тетрады включительно является актуальным значение, вырабатываемое при выполнении каждой тетрады.

Такую информацию можно получить в результате анализа возможных путей передачи управления. Для извлечения этой информации последовательность тетрад (триад) преобразуют в так называемый граф потока. Узлами графа потока являются неразветвляющиеся вычисления (базовые блоки), а дугами – передачи управления при разветвлениях.

Базовым блоком называется такая последовательность инструкций, в которую поток управления входит в начале и покидает в конце (внутри последовательности не может быть инструкций возврата из процедуры и/или условных передач управления).

Для разбиения исходной последовательности тетрад на базовые блоки можно использовать следующий алгоритм.

Определяется набор лидеров, т. е. первых тетрад (триад) базовых блоков согласно следующим правилам:

(1) первая тетрада последовательности является лидером;

(2) любая тетрада условного или безусловного перехода (в том числе – вызова процедуры) является лидером;

(3) тетрада, на которую может быть передано управление (в нашем примере – имеющая метку), является лидером;

(4) любая тетрада, непосредственно следующая за условным или безусловным переходом, является лидером.

Теперь для каждой тетрады-лидера можно определить базовый блок, который состоит из этой тетрады и всех тетрад вплоть до следующего лидера (естественно, без него), либо до конца программы.

Рассмотрим в качестве примера процедуру *PartExch* из программы сортировки на языке Pascal (см. рис. 1.20). Постфиксная запись этой процедуры, построенная формальными методами путем расширения действиями грамматики языка, может выглядеть так:

```
k mVal := i iBeg := j iBeg := si 0 := sj 0 :=
Label1: i j < Label10 GoIfFalse
Label11: arr i ^ k < Label12 GoIfFalse
si si arr i ^ + := i i 1 + := Label11 Go
Label12: Label21: arr j ^ k > Label22 GoIfFalse
sj sj arr j ^ + := j j 1 - :=
Label21 Go Label22: i j < Label31 GoIfFalse
i push j push exchange call
Label31:Label1 Go Label10:i return
```

Преобразование этой записи позволит получить последовательность тетрад, показанную в табл. 2.4.

Таблица 2.4

Номер	Метка	Операция	Операнд1	Операнд2	Результат
1		:=	*mVal*	*null*	*k*
2		:=	*iBeg*	*null*	*i*
3		:=	*iEnd*	*null*	*j*
4		:=	*0*	*null*	*si*
5		:=	*0*	*null*	*sj*
6	*Label1:*	<	*j*	*i*	*tmpValue1*
7		*GoIfFalse*	*tmpValue1*	*Label10*	*null*
8	*Label11:*	^	*arr*	*i*	*tmpValue2*
9		<	*tmpValue2*	*k*	*tmpValue3*
10		*GoIfFalse*	*tmpValue3*	*Label12*	*null*
11		^	*arr*	*i*	*tmpValue4*
12		+	*tmpValue4*	*si*	*tmpValue5*
13		:=	*tmpValue5*	*null*	*si*
14		+	*1*	*i*	*tmpValue6*
15		:=	*tmpValue6*	*null*	*i*
16		*Go*	*Label11*	*null*	*null*
17	*Label12:*	*null*	*null*	*null*	*null*
18	*Label21:*	^	*arr*	*j*	*tmpValue7*
19		>	*K*	*tmpValue7*	*tmpValue8*
20		*GoIfFalse*	*tmpValue8*	*Label22*	*null*
21		^	*arr*	*j*	*tmpValue9*
22		+	*tmpValue9*	*sj*	*tmpValue10*
23		:=	*tmpValue10*	*null*	*Sj*
24		-	*1*	*j*	*tmpValue11*
25		:=	*tmpValue11*	*null*	*J*
26		*Go*	*Label21*	*null*	*null*
27	*Label22:*	<	*J*	*i*	*tmpValue12*
28		*GoIfFalse*	*tmpValue12*	*Label31*	*null*
29		*Push*	*I*	*null*	*null*
30		*Push*	*J*	*null*	*null*
31		*Call*	*exchange*	*null*	*null*
32	*Label31:*	*Go*	*Label1*	*null*	*null*
33	*Label10:*	*Return*	*I*	*null*	*null*

Заметим, что в процессе формирования последовательности тетрад образовано довольно много дополнительных объектов с именами вида *tmpValueN*, необходимых для временного хранения промежуточных результатов.

Построение графа потока по этой последовательности позволит, в частности, определить, в каких базовых блоках они будут «живыми», т. е. необходимыми для последующих вычислений.

Применим вышеописанный алгоритм определения лидеров для последовательности, содержащейся в табл. 2.4.

Тетрада с номером 1 является лидером по правилу (1). Согласно правилу (2) лидерами являются тетрады с номерами 7, 10, 16, 20, 26, 28, 31 и 32. Лидерами согласно правилу (3) являются тетрады 6, 8, 17, 18, 27, 32, 33. И, наконец, согласно правилу (4) лидерами являются тетрады с номерами 8, 11, 17, 21, 27, 29 и 33.

Итак, последовательность тетрад следующим образом разбита на базовые блоки (указаны первая и последняя тетрады): 1–5, 6, 7, 8–9, 10, 11–15, 16, 17–19, 20, 21–25, 26, 27, 28, 29–30, 31, 32 и 33.

Базовые блоки являются узлами графа потока. Один из его узлов определяется как начальный – это блок, лидер которого является первой инструкцией программы. От блока B_1 к блоку B_2 ведет направленная дуга, если блок B_2 может непосредственно следовать за блоком B_1 в какой-либо последовательности выполнения блоков программы. Это возможно в следующих случаях.

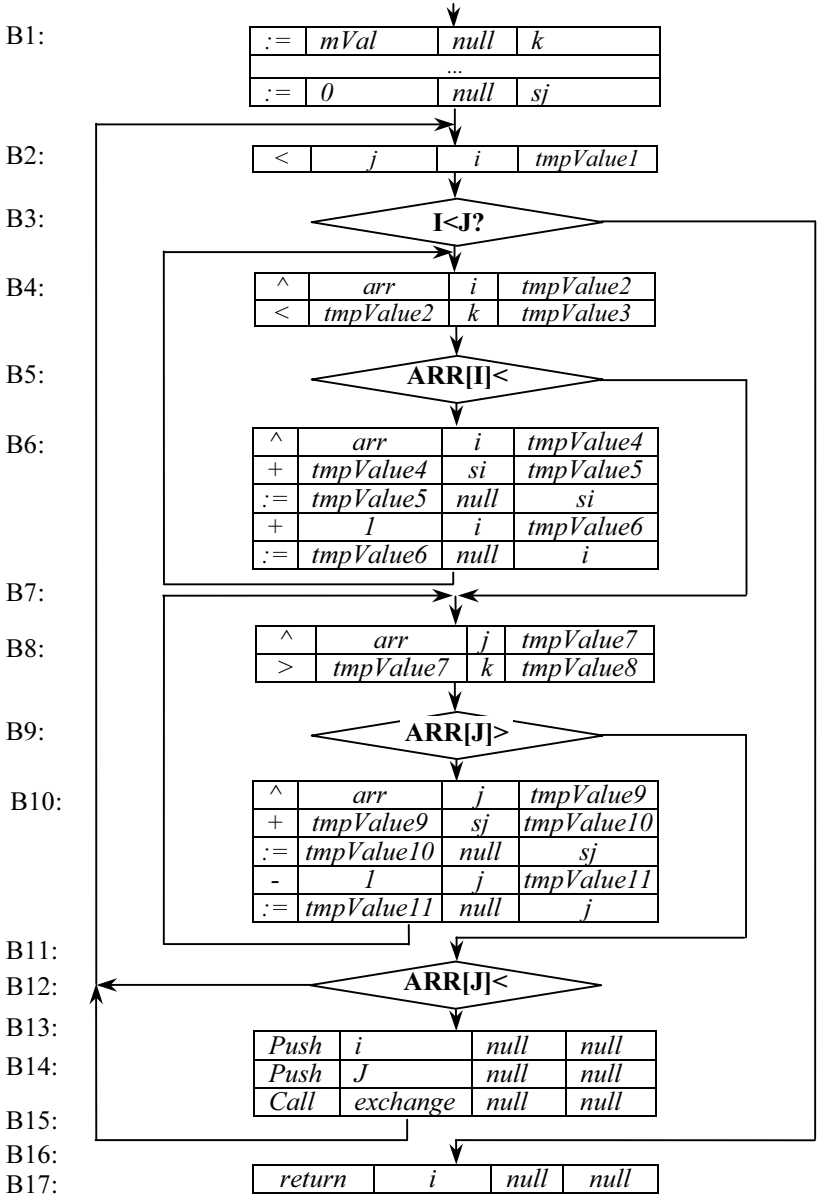
1. Имеется условный или безусловный переход от последней инструкции блока B_1 к первой инструкции B_2.

2. Блок B_2 в последовательности тетрад непосредственно следует за блоком B_1 и при этом блок B_1 не заканчивается инструкцией безусловного перехода.

На рисунке показан граф потока процедуры *PartExch*. В целях экономии места в нем объединены блоки B_{14} и B_{15}.

Как уже говорилось, графы потока используются в первую очередь для определения тех точек программы, начиная с которых значение объекта, вычисленное в каждой данной тетраде, в дальнейшем не используется.

Особенно важно уметь определять интервалы «живости» для тех объектов, значения которых содержатся в наиболее дорогостоящих ресурсах – регистрах процессора. Если значение объекта, для хранения которого отведен регистр процессора, не будет использоваться в последующих операциях, этот регистр может быть назначен для хранения другого объекта. Информация об интервалах «живости» объектов программы нужна также для обнаружения ошибок (типа использования значения объекта без предшествующего присваивания).



Граф потока процедуры PartExch

Использование объекта в тетраде определяется следующим образом. Предположим, что в тетраде с номером i присваивается значение объекту x. Если тетрада j содержит x в качестве операнда и существует путь движения управления от тетрады i к тетраде j, в котором нет ни одной тетрады, присваивающей новое значение объекту x, то тетрада j использует значение x, вычисленное в i.

Говорят, что по отношению к данной точке текста программы объект является «живым», если его значение используется хотя бы одной тетрадой, которая может быть выполнена после данной точки (т. е. может быть достигнута по какому-либо пути в графе потока).

Прежде всего рассмотрим, как определяется «живость» объектов внутри базовых блоков.

Каждый базовый блок сканируется от конца к началу. Предположим, что при таком обратном сканировании обрабатывается тетрада с номером i вида

op	*y*	*z*	*x*

Для нее выполняются следующие действия.

1. Тетрада расширяется текущей (накопленной за время обработки базового блока) информацией о последующем использовании и «живости» объектов x, y и z.

2. Объект x фиксируется как «неживой» (неиспользуемый в последующем).

3. Объекты y и z помечаются как «живые» с последующим использованием в тетраде i.

Заметим, что изменять порядок выполнения шагов (2) и (3) нельзя, поскольку x может представлять собой y или z.

Теперь, предполагая, что каждая процедура имеет единственную точку выхода, начнем с базового блока, соответствующего этой точке, и пройдем по некоторому пути, двигаясь в порядке, обратном передачам управления (пройденные базовые блоки будем помечать для исключения зацикливания), и запоминая во вспомогательном стеке все встретившиеся разветвления. В момент достижения точки входа в процедуру все «живые» объекты для данного пути окажутся помеченными, а в тетрадах пройденного пути будет сохранена информация о точках, в которых используются эти объекты. Повторный, но теперь прямой проход по этому пути с возвратами из точек разветвлений, извлекаемых из стека, позволит определить свойства «живости» всех

объектов процедуры, а точнее – те тетрады, в которых возникает и исчезает это свойство для каждого объекта. Впоследствии эту информацию можно будет эффективно использовать при принятии решения о том, какими машинными инструкциями следует реализовывать вычисления, и для объединения объектов, интервалы «живости» которых не пересекаются.

Графы потока используются также для определения циклически повторяющихся участков текста программы. Это необходимо для решения задач оптимизации (см. разд. 2.3).

Что собой представляют циклы в графе потока и как найти все циклы? Чаще всего на этот вопрос ответить очень просто. Например, на рис. 2.1 показано три цикла, в том числе два внутренних, содержащих блоки B_4–B_7, B_8–B_{11}, и охватывающий их цикл B_2-B_{16}.

Однако в общем случае решение задачи нахождения циклов в графах потока является сложной задачей, методы решения которой здесь не рассматриваются. Цикл представляет собой набор узлов графа потока, удовлетворяющий следующим условиям.

1. Все узлы набора сильно связаны, т. е. имеется путь единичной или большей длины от любого узла в цикле к любому другому, полностью принадлежащий циклу.

2. Набор узлов имеет единственный вход, т. е. узел в цикле такой, что любой путь к узлу в цикле из узла вне цикла проходит через вход.

Способы использования результатов решения задачи выявления циклов будут рассмотрены в подразд. 2.3.13.

2.2. Объектный код

Результатом работы генератора кода является программа, которую способна исполнять целевая машина. Целевая программа может в принципе формироваться либо на машинном языке в абсолютных адресах или в перемещаемом формате, либо на языке ассемблера.

Генерация на машинном языке в абсолютных адресах выполняется обычно при трансляции некоторых системных программ, когда заранее известно, в каком месте памяти будет исполняться целевой код.

При генерации программы в перемещаемом формате будущее место исполнения программы заранее неизвестно, поэтому транслятор не в состоянии сформировать объектный код полностью. Результат генерации кода сопровождается дополнительными структурами данных, содержащими сведения о том, какие элементы кода и каким образом

должны быть модифицированы после определения адреса выделенной памяти. С одной стороны, это позволяет реализовывать раздельную трансляцию частей большой прикладной программы, с другой – возникают дополнительные затраты на связывание и загрузку. Считается, что эти затраты с лихвой компенсируются возможностью вызова других, ранее скомпилированных подпрограмм из объектных модулей.

Получение в качестве выхода генератора кода программы на языке ассемблера несколько облегчает процесс генерации кода. В частности, при этом можно создавать символьные инструкции и использовать возможности макросов ассемблера. Плата за эти удобства – дополнительный проход для обработки ассемблерной программы после генерации кода.

В настоящей главе для большей удобочитаемости текста в качестве целевого языка используется ассемблерный код.

2.2.1. Свойства и характеристики целевой машины

Генератор кода может быть «хорошим» только в том случае, если он использует максимально возможный объем информации о целевой машине, наборе ее инструкций, общих и специальных регистрах процессора, возможных методах адресации памяти. К сожалению, при обсуждении общих вопросов генерации кода невозможно описать все возможные варианты архитектуры целевой машины и их влияние на методы формирования объектного кода.

Рассмотрим в качестве целевого компьютера условную регистровую машину, являющуюся типичным представителем персональных компьютеров. Целевой компьютер представляет собой упрощенный аналог микропроцессора i80x86, т. е. машину с адресацией памяти с точностью до байта, словом из четырех байтов и четырьмя 32-битными регистрами общего назначения *EAX*, *EBX*, *ECX*, *EDX*. Младшие 16 бит и 2 составляющих их байта этих регистров могут использоваться по отдельности, для них применяются обычные обозначения *zX* (младшие 16 бит), *zL* (младший байт регистра *zX*) и *zH* (старший байт регистра *zX*), где *z* – одна из букв *A*, *B*, *C* или *D*.

Кроме регистров общего назначения есть регистры специального назначения: *ESP* (указатель стека), *EBP* (обычно используется в качестве указателя записи активации), *ESI* (регистр индекса), *EIP* (счетчик команд).

Набор команд включает инструкции вида

CodeOp – нульадресные инструкции;

CodeOp Op – одноадресные инструкции;

CodeOp Dst, Src – двухадресные инструкции.

Здесь *CodeOp* – код операции, *Op* (*operand*), *Src* (*source* – источник) и *Dst* (*destination* –получатель) – операнды.

В качестве примеров приведем некоторые коды операций:

Нульадресная команда *RET* – вернуть управление из процедуры.

Одноадресные команды:

PUSH – занести в стек значение указанного операнда (32-битное слово);

POP – снять со стека слово и установить его в качестве значения операнда;

INC (DEC) – увеличить (уменьшить) на единицу значение операнда.

Двухадресные команды:

MOV – скопировать значение источника в память получателя;

ADD – прибавить значение источника к значению получателя;

SUB – вычесть значение источника из значения получателя.

Прочие инструкции будем определять по мере необходимости.

Для указания места хранения значений операндов (источника и приемника в двухадресных командах) имеется некоторый набор режимов адресации. Длина машинной команды (и соответственно затраты времени на ее выборку из памяти) зависит от того, где находятся источник и получатель и каким образом формируются их исполнительные адреса.

1. *Регистровая адресация*: операнд расположен в одном из 8, 16, или 32-разрядных регистров общего назначения.

2. *Непосредственная адресация*: операнд является частью кода команды.

Во всех остальных режимах адресации операнд команды расположен в памяти отдельно от кода команды и его исполнительный адрес вычисляется путем сложения определенного сочетания следующих адресных элементов.

Смещение: 8, 16 или 32 бита, которые являются частью кода команды аналогично непосредственному операнду. Для указания размера смещения, отличающегося от 32 бит, могут использоваться адресные префиксы вида *BYTE PTR* – 8 бит и *WORD PTR* –16 бит.

База: содержимое любого из регистров общего назначения.

Индекс: содержание любого из регистров общего назначения, кроме ESP. Индексные регистры используются для доступа к элементам массивов или цепочки символов.

Масштаб индекса со значениями коэффициента 1, 2, 4, или 8 (умножение значения индексного регистра на заданный коэффициент перед сложением; имеется в процессорах i80x86 и Pentium-X и считается полезным для доступа к массивам структур).

Возможные комбинации этих четырех компонентов составляют 6 дополнительных режимов адресации:

3. *Прямая адресация*: адрес операнда является частью команды в виде 8, 16, или 32-битного адреса. Например:

INC [50032] – увеличить на единицу значение четырехбайтного слова по адресу 50032 (число 50032 есть часть кода команды);

INC WORD PTR [50032] – увеличить на единицу значение двухбайтного слова по тому же адресу.

4. *Косвенная регистровая адресация*: базовый или индексный регистр содержит адрес операнда. Например:

MOV [ECX],EDX – записать значение регистра *EDX* в память по адресу, находящемуся в регистре *ECX*.

5. *Базовая адресация*: содержание базового регистра добавляется к смещению для формирования исполнительного адреса. Например:

MOV [ESP+4],EDX – записать содержимое регистра *EDX* в память по адресу, полученному путем сложения значения регистра *ESP* с непосредственным смещением, равным 4 (и занимающим 4 байта в коде команды, поскольку не указан префикс адреса).

6. *Индексная адресация*: содержание индексного регистра добавляется к смещению. Например:

MOV EAX,TABLE[ESI] – переслать (точнее – скопировать) в регистр *EAX* 4-байтное число из массива слов с именем *TABLE*. Индекс первого байта этого числа относительно начала массива берется из регистра *ESI*. Заметим, что адрес начала массива *TABLE* хранится в коде команды в качестве непосредственного значения.

7. *Базовая индексная адресация*: содержание базового регистра складывается с содержимым индексного регистра и используется как адрес памяти. Например:

ADD EAX,[ESI][EBX] – добавить к значению регистра *EAX* число, адрес которого получен путем суммирования значений регистров *ESI* и *EBX*.

8. *Базовая индексная адресация со смещением*: содержание базового регистра складывается с содержимым индексного регистра и со смещением операнда (непосредственным, т. е. хранящимся в коде команды). Например:

DEC BYTE PTR[ESI][EBP+00FFFFF0H] – вычесть единицу из 8-битного числа, адрес которого есть сумма значений регистров *ESI* и *EBP* со смещением, указанным как шестнадцатеричное значение *00FFFFF0H*.

Для каждого режима адресации (и каждого кода операции, если он занимает больше 1 байта) может быть определена дополнительная «цена» за его использование. Будем считать, что такой ценой является размер памяти в байтах, необходимый для хранения всех компонентов, используемых для вычисления адреса. Не включается в дополнительную цену размер регистров процессора, значения которых используются для вычисления адреса.

Приведем сводную информацию о режимах адресации для 32-битных операндов, т. е. без учета префиксов (табл. 2.5).

Таблица 2.5

Название режима адресации	Обозначение	Адрес значения	Цена
Непосредственный	*число*	*[EIP]*	4
Регистровый	*R*	–	0
Прямой	*[число]*	*[[EIP]]*	4+4
Регистровый косвенный	*[R]*	*[R]*	0+4
Базовый относительный	*[R+c]*	*[R]+[[EIP]]*	4+4
Индексный	*c[R]*	*[R]+[[EIP]]*	4+4
Базовый индексный	*[I][R]*	*[I]+[R]*	0+4
Базовый индексный со смещением	*c[I][R]*	*[I]+[R]+[[EIP]]*	4+4

Стоимость инструкции теперь можно определить как единица (цена байта, содержащего код операции) плюс сумма цен, связанных с режимами адресации всех ее операндов. Эту стоимость можно использовать при генерации кода для оценки возможных вариантов реализации тетрад и выбора самого «дешевого».

Если важно количество используемой памяти, то очевидно, что по возможности должна быть минимизирована длина инструкций. При этом образуется дополнительная выгода за счет того, что время, тре-

бующееся для выборки инструкции из памяти, как правило, превышает время, затрачиваемое на ее исполнение. Следовательно, при минимизации длины инструкций снижается и общее время выполнения программы. Это общее правило, из которого, безусловно, существуют и исключения. Однако даже рассмотрение самых общих критериев, определяемых аппаратурой компьютера, при генерации кода приводит к практически необозримым размерам реальных оценочных таблиц. Поэтому здесь исключения не рассматриваются.

2.2.2. Выбор инструкций

Набор инструкций целевой машины определяет сложность их выбора. Важными факторами этого выбора являются единообразие и полнота множества инструкций. Если целевая машина не поддерживает все типы данных единообразно, то каждое исключение из общего правила потребует специальной обработки.

Если не стоит задача формирования эффективной целевой программы, то выбор инструкций достаточно прост. Например, для каждого знака операции, который может встретиться в тетраде, следует разработать шаблон целевого кода, в который просто преобразуется данная тетрада. Например, для знака операции сложения 32-битных целых чисел может быть построен такой шаблон:

MOV EAX, <operand1> /* загрузить первый операнд в регистр *EAX* */

ADD EAX, <operand2> /* сложить с *EAX* значение второго операнда*/

MOV <result>, EAX /* сохранить значение результата*/

Здесь в угловых скобках используются наименования полей тетрады. Ясно, что такой шаблон может быть применен только к статическим объектам, для которых возможно вычисление адреса в процессе трансляции. Для всех прочих объектов (локальных, нелокальных, временных) связывание имени объекта с адресом памяти происходит уже во время выполнения программы. Следовательно, для каждого возможного сочетания классов памяти (терминология языка С) операндов и результата тетрады придется иметь свой собственный шаблон.

К сожалению, такая генерация кода (раздельная обработка тетрад) часто порождает «плохие» программы. Так, например, последовательность операторов:

$a = b + c; d = a + e;$

будет преобразована в следующую последовательность машинных инструкций:

```
MOV     EAX, B
ADD     EAX, C
MOV     A, EAX
MOV     EAX, A
ADD     EAX, E
MOV     D, EAX
```

Очевидно, что четвертая инструкция, по сути, не нужна, а если значение *a* в дальнейшем не используется (см. подразд. 2.1.5), то не нужна и третья. Впоследствии это может обнаружить оптимизатор кода и удалить ненужные инструкции.

Качество генерируемого кода определяется его размером и скоростью. Целевая машина с богатым набором инструкций может обеспечить несколько путей выполнения одной и той же операции. Поскольку различные реализации существенно различаются по своей эффективности, непосредственная трансляция промежуточного кода может приводить к вполне корректному, но совершенно неприемлемому с точки зрения эффективности работы целевому коду. Например, если целевая машина имеет инструкцию инкремента *INC*, то трехадресный код *a: =a+1* можно реализовать более эффективно с помощью инструкции *INC A* по сравнению с более очевидной последовательностью – занести значение *a* в регистр, просуммировать с единицей и сохранить полученное значение в *a*.

```
MOV     EAX,A
ADD     EAX,1
MOV     A,EAX
```

Некоторые дополнительные сложности генерации кода можно обнаружить, рассмотрев шаблоны с использованием различных возможных методов адресации. Пусть *a*, *b* и *c* – локальные целочисленные переменные. Их локальность приводит к тому, что для доступа к значениям должны использоваться смещения (обозначим их *c_a*, *c_b* и *c_c*) относительно значения регистра *EBP*, хранящего адрес записи активации. Вот несколько примеров.

1. Выполнение сложения через регистр

```
MOV     EAX, C_B[EBP]       // Цена 1+4+4
ADD     EAX,C_C[EBP]        // Цена 1+4+4
MOV     C_A[EBP],EAX        // Цена 1+4+4
```

Суммарная стоимость = (1+4+4) + (1+4+4) + (1+4+4) = 27.

2. Сложение в памяти

MOV C_A[EBP], C_B[EBP] // ЦЕНА 1+4+4+4+4

ADD C_A[EBP], C_C[EBP] // ЦЕНА 1+4+4+4+4

Суммарная стоимость = (1+4+4+4+4) + (1+4+4+4+4) = 34.

Несмотря на то что первая реализация кажется более длинной, ее суммарная стоимость оказывается на 20 % меньше.

Ясно, что для генерации хорошего кода нужно эффективно использовать возможности режимов адресации машины. Знания о скорости выполнения тех или иных инструкций необходимы для создания хорошего (эффективного) целевого кода, однако, к сожалению, точная информация об этом обычно труднодоступна. Кроме того, принятие решения о способе реализации того или иного промежуточного кода зачастую требует учета контекста, в котором появляется та или иная инструкция, а также особенностей работы целевой машины. Код, эффективный при исполнении на процессоре i80286, может оказаться далеко не лучшим при работе на процессоре Pentium, и не только из-за более богатого набора инструкций последнего.

Инструкции, использующие в качестве операндов регистры, обычно короче и быстрее выполняются, чем инструкции, работающие с операндами, расположенными в памяти. Следовательно, эффективное использование регистров – еще одна важная составляющая генерации хорошего целевого кода. Использование регистров часто разделяется на две подзадачи.

1. В процессе распределения регистров выбирается множество переменных, которые будут находиться в регистрах в некоторой точке программы.

2. В последующей фазе назначения регистров выбираются конкретные регистры для размещения в них переменных.

Поиск оптимального назначения регистров переменным представляет собой сложную задачу, с точки зрения математики являющуюся *NP*-полной. Проблема усложняется еще и тем, что аппаратное обеспечение и/или операционная система могут накладывать дополнительные ограничения на использование регистров. Например, операции умножения и деления часто используют пары регистров, но в команде указывается только один.

Таким образом, можно сделать общий вывод: создание хорошего генератора кода – это искусство, в котором эвристические находки могут считаться не менее важными, чем формализованные строгие алгоритмы.

2.3. Оптимизация программы

Методы оптимизации кода теоретически могут применяться на разных уровнях синтаксических конструкций – оператора, блока, цикла, процедуры и до программы в целом. Чем выше уровень конструкции, тем больше потенциальная возможность повышения общей эффективности транслируемой программы. Однако затраты на применение большей степени оптимизации могут значительно увеличить время компиляции.

Оператор – это первичная синтаксическая единица в программе. Большинство компиляторов выполняют некоторую оптимизацию на этом уровне.

Блок – это последовательность операторов, для которой существуют единственная точка входа и единственная точка выхода. Линейный вид блока инструкций позволяет компилятору выполнять оптимизацию, основывающуюся на промежутках жизни различных данных и выражений, используемых внутри блока. Оптимизирующий компилятор разбивает программу на последовательности базовых блоков путем конструирования ориентированного потокового графа (см. подразд. 2.1.5). Большинство компиляторов производят оптимизацию на уровне базового блока.

Цикл содержит выполняемую многократно последовательность операторов. Поскольку многие программы проводят в циклах большую часть времени своего выполнения, оптимизация в этой области может дать существенное улучшение характеристик исполнения программы.

Процедуры – это операторы, которые содержат целые подпрограммы или функции. Оптимизация на этом уровне реализуется редко в силу большой сложности.

Наиболее сложный уровень оптимизации – уровень, на котором учитываются свойства программы в целом, в настоящее время рассматривается чисто теоретически. Пока еще нет реализаций оптимизаторов такого уровня, дающих реальное улучшение качества программ.

Оптимизирующие преобразования могут быть зависящими или не зависящими от архитектуры компьютера. Машинно-независимая оптимизация, такая как выделение общих подвыражений и вынесение инвариантного кода, обычно применяется к промежуточному представлению программы, т. е. к последовательности тетрад/триад. Машинно-зависимая оптимизация применяется к результату стадии генерации кода и использует информацию о наборе команд конкретного

процессора. Эти методы оптимизации в настоящее время реализуются для небольшого «окна» (5 – 10 инструкций машинного уровня) и называются «щелевыми». Типичные примеры щелевой оптимизации включают удаление излишних операций загрузки/сохранения регистров, удаление недостижимого кода, выпрямление передач управления, алгебраические упрощения, снижение мощности (команд) и использование команд, специфических для конкретного процессора.

Существует множество различных машинно-зависимых и машинно-независимых методов оптимизации кода. Рассмотрим некоторые из машинно-независимых методов, считающиеся наиболее употребимыми. О машинно-зависимых методах мы уже упоминали здесь и в подразд. 2.2.2, детально рассматривать их имеет смысл только применительно к характеристикам конкретной аппаратной платформы и, более того, конкретного процессора.

2.3.1. Размножение констант

Это один из простейших методов. При его применении любая ссылка на константное значение замещается самим значением. В следующем примере на языке С повышается эффективность благодаря удалению трех адресных ссылок и замене их константами. Последовательность операторов:

```
x = 2;
if( a < x && b < x) c = x;
```

может быть эквивалентно преобразована в такую:

```
x = 2;
if(a < 2 && b < 2) c = 2;
```

но только при условии, что между этими двумя операторами значение переменной *x* не изменяется. Заметим, что после замены *x* константой *2* первый оператор может быть удален, поскольку переменная *x* перестает быть «живой».

2.3.2. Размножение копий

Этот метод похож на размножение констант, но копируются имена переменных вместо константных значений. Например,

```
x = y;
if(a < x && b < x) c = x;
```

преобразуется в

x = y;

if(a < y && b < y) c = y;

Размножение копий, так же как и размножение констант, позволяет впоследствии удалять объекты, ставшие «неживыми».

2.3.3. Свертка констант

Этот метод сводит выражения, которые содержат константные данные, к по возможности наиболее простой форме. Константные данные обычно используются в программе либо непосредственно (как в случае чисел или цифр), либо косвенно (как в случае объявленных констант).

Для примера приведем фрагмент программы на языке С:

#define TWO 2

...

a = 1 + TWO;

Во время компиляции последний оператор присваивания можно привести к его эквивалентной форме

a = 3;

благодаря чему удаляются ненужные арифметические операции из стадии выполнения программы. Сворачивание констант можно применять к разным их типам, но это требует вычисления и преобразования значений во время трансляции.

2.3.4. Алгебраические упрощения

Это специальный вид свертки констант, предназначенный для удаления арифметических тождеств. Код, сгенерированный для таких операторов, как

x = y + 0;

*x = y * 0;*

x = y / 1.0;

x = y / 0;

не должен содержать команд для выполнения каких-либо арифметических операций. Хороший транслятор в этот момент заодно может обнаружить, что последний оператор ошибочен.

2.3.5. Извлечение общих подвыражений

Преследует цель ликвидации лишних вычислений. Вместо того чтобы генерировать код для вычисления значения каждый раз, когда оно записано в тексте, оптимизатор может попытаться выделить выражение таким образом, чтобы его значение вычислялось только однажды. Там, где это возможно, последующие вхождения этого выражения заменяются на использование ранее вычисленного значения. Например, выражения *y*3* и *a[y*3]* являются общими подвыражениями в следующем тексте:

```
if( a[ y * 3] < 0 || b[ y * 3] > 10) a[ y * 3] = 0;
```

Выделение этих выражений и замена их временными объектами приводит к эквивалентному тексту:

```
t1 = y * 3; a1 = &a[ t1 ]; a2 = &b[ t1 ];
if( *a1 < 0 || *a2 > 10)*a1 = 0;
```

Выделение общих подвыражений обычно выполняется внутри базовых блоков графа потока программы.

2.3.6. Глубокое выделение общих подвыражений

Более сложный метод оптимизации, при котором выполняется поиск общих подвыражений во всех возможных путях движения управления из произвольной точки программы. Выделение того же самого общего подвыражения в операторе

```
if ( a == 0 ) a = y * 3; else b = y * 3;
```

приводит к эквивалентной последовательности операций:

```
t1 = y * 3;if ( a == 0 ) a = t1; else b = t1;
```

Реализация этого метода в общем случае отнюдь не тривиальна и может повлечь возникновение трудноуловимых ошибок.

2.3.7. Понижение мощности операций

Под этим подразумевается замещение долго выполняющихся операций эквивалентными им, но более быстрыми. Оптимизатор может применять снижение мощности несколькими способами. Наиболее показательный пример – замена операций умножения или деления целых чисел на степени двойки операциями сдвига.

2.3.8. Удаление недостижимого кода

Недостижимый код – это такая последовательность инструкций программы, которая не может быть исполнена при любых сочетаниях исходных данных. От значений исходных данных и объявленных в программе констант зависят пути, по которым движется управление. Соответственно недостижимый код может образоваться как следствие предыдущих операций оптимизации, как часть кода, написанного для условной отладки, или как результат частых изменений программы многими программистами. Например, следующие операторы – вариант кода для проверки компилятора на выполнение этого метода оптимизации.

```
#define DEBUG 0
...
if(DEBUG) printf("Debug Function\n");
```

Вызов функции *printf* при данном значении константы *DEBUG* невозможен, код этого вызова недостижим и может не включаться в результат трансляции. Задекларированные константы часто могут скрывать существование недостижимого кода, особенно если такой код определяется внутри включаемого файла-заголовка.

2.3.9. Удаление лишних присваиваний

Этот метод основывается на нахождении промежутка «живости» переменных и преследует цель удаления таких операций присваивания, результаты которых не используются в последующих вычислениях. Как правило, этот метод применяется в целях экономии ограниченных ресурсов, таких как пространство стека или машинные регистры. Например, из последовательности операторов

$$a = 5;\ b = 0;\ a = b;$$

первое присваивание может быть удалено, результаты работы программы от этого не изменятся. Лишние присваивания могут возникать непреднамеренно, когда промежуток жизни переменной велик и между вхождениями переменной имеется более или менее длинный текст.

Лишние присваивания могут быть также результатом работы предыдущих действий по оптимизации. Оптимизатор вынужден заново формировать граф потока программы и определять интервалы времени «живости» ее объектов после каждого своего прохода по ее тексту (или промежуточному представлению).

2.3.10. Распределение переменных по регистрам

Цель этого метода – попытаться обеспечить оптимальное назначение регистров путем сохранения часто используемых переменных в регистрах так долго, как это возможно, для того чтобы исключить более медленный доступ к памяти. Количество регистров, доступных для использования, зависит от архитектуры процессора. Семейство микропроцессоров i80x86, например, имеет много регистров для специального использования и сравнительно мало универсальных регистров.

Некоторые языки программирования (например, язык C) предоставляют спецификатор класса регистровой памяти в помощь распределению переменных по регистрам. Это ключевое слово (*register*) дает программисту возможность указывать, значения каких переменных должны располагаться в регистрах (по его мнению, к которому оптимизатор не обязан прислушиваться согласно стандарту языка).

При назначении переменных регистрам компилятор принимает во внимание не только то, какие переменные нужно выделить, но также и регистры, которым они назначаются. Выбор переменных зависит от частоты их использования, промежутков жизни текущих регистровых переменных (которые определяются при анализе потоков данных) и количества доступных регистров. В зависимости от степени выполняемой компилятором оптимизации промежуток жизни переменной может определяться внутри единственного оператора, внутри базового блока или перекрывать несколько базовых блоков. Переменная сохраняется в регистре только в том случае, если она вскоре будет снова использоваться. Если на переменную в ближайшем будущем (в пределах оператора, базового блока или цепи базовых блоков заданной длины) не имеется ссылок, то ее значение сохраняется в оперативной памяти, а регистр освобождается для другой переменной.

Поскольку оптимизирующий компилятор определяет промежуток «живости» каждой переменной, он не должен генерировать лишние операции сохранения и загрузки (регистров). Лишние операции сохранения значений ликвидируются посредством удаления излишних присваиваний; лишние операции загрузки исключаются с помощью усовершенствованного распределения переменных по регистрам.

2.3.11. Замена вызова функции ее телом

Суть метода ясна из его названия. Применяется он в тех случаях, когда объем тела процедуры без последовательностей команд, обеспечивающих вход/возврат, сопоставим с объемом этих последовательностей. Ясно, что такая подстановка (*inline-substitution*) приводит не только к улучшению временных характеристик программы, но и к увеличению ее потребностей в памяти, поэтому вызов любой процедуры/функции следует заменять ее телом.

Подставляемыми обычно реализуются часто используемые библиотечные функции небольшого размера, такие как *abs*, *min*, *max* … Некоторые языки (или реализации) позволяют управлять такими подстановками прямо в тексте программы.

2.3.12. Сжатие цепочек переходов

При трансляции условных операторов генератор кода компилятора иногда строит инструкции перехода на другие инструкции перехода. Длина таких цепочек переходов зависит от глубины вложенности циклов, условных операторов, переключателей друг в друга в тексте транслируемой программы.

Сжатие цепочек переходов просто превращает связанную последовательность переходов в единственный переход от начала цепочки переходов к конечной цели цепочки.

2.3.13. Оптимизация циклов

Это направление оптимизации следует считать наиболее важным, поскольку время, проводимое в циклах, обычно составляет основную долю от всего времени выполнения программы (во всяком случае такой программы, которую имеет смысл оптимизировать). Наиболее важным в оптимизации циклов является минимизация времени на исполнение одного повторения цикла.

Не так важно, сколько машинных команд будет построено для реализации тела цикла, более существенным является суммарное количество обращений к памяти на одно выполнение тела.

Вынесение инвариантных вычислений – один из основных путей ускорения циклов, заключающийся в переносе операторов за пределы

цикла, если вычисляемые ими значения неизменны для всех повторений цикла. Если инвариантный код выносится из следующего цикла:

```
for (i = 0; i < v; i++)
    x = i * (j+k);
```

то эквивалентом этого цикла будет такая последовательность:

```
t1 = j + k;
for(i = 0; i < v; i++)
    x = i * t1;
```

Дальнейший анализ этого примера показывает, что значение переменной *i* (индекса цикла), формируемое в каждой итерации, не используется в следующих итерациях. Следовательно, если *i* – «живая» переменная вне оптимизируемого цикла, то последнее присваивание ей значения следует вынести за его пределы и вообще удалить из цикла все операции со значением переменной *i*:

```
t1 = j + k;
for(x = 0; x< t1 * v; x += t1) ;
i = v;
```

Если в одном и том же базовом блоке встречаются две такие последовательности операторов (а подобные ситуации встречаются довольно часто):

```
for(i = 0; i < 10; i++) a = b + c [i];
for(i = 0; i < 10; i++) d = e + f [i];
```

то они могут быть объединены в один цикл:

```
for(i = 0; i < 10; i++) {
    a = b + c[i]; d = e + f [i];
}
```

Этот метод оптимизации называется слиянием циклов.

Часто используется такое действие, как разворачивание циклов, цель которого состоит в минимизации затрат путем уменьшения количества или полной ликвидации операций организации повторений тела. Например, цикл инициализации массива:

```
int i;
int a[3];
for(i = 0; i < 3; i++) a[i] = 0;
```

может быть выполнен значительно быстрее путем его замены последовательностью присваиваний с константными индексами элементов массива:

a[0]=0; a[1]=0; a[2]=0;

Вместо лобового размножения тела цикла с подстановкой всех возможных значений индекса цикла часто применяют оптимизацию путем использования специализированных инструкций процессора.

В системах команд многих процессоров имеются быстро выполняемые операции для инициализации и/или перемещения блоков значений в памяти, а также для реализации других часто встречающихся ситуаций массовой модификации данных.

К примеру, строковые инструкции с префиксом повторения *REP* в семействе процессоров i80x86 выполняются значительно быстрее, чем посимвольные команды в цикле.

Оптимизатор может использовать такие команды в тех случаях, когда это возможно. Применение специализированных инструкций процессора к расширенной версии предыдущего примера разворачивания циклов:

```
int a[10000];
int i;
for(i = 0; i < 10000; i++) a[i] = 0;
```

дает такой ассемблерный код для процессора i80x86:

```
MOV     ECX, 10000
MOV     EDI, OFFSET A
XOR     EAX, EAX
PUSH    DS
POP     ES
CLD
REP     STOSD
```

Этот вариант программы будет работать значительно быстрее, чем любая реализация цикла, оперирующая с элементами массива как с отдельными переменными.

2.3.14. Потенциальные проблемы, связанные с оптимизацией

Трансляторы, оптимизирующие объектный код, обычно предоставляют возможность выбора степени (или глубины) оптимизации. Время, требуемое для компиляции программы, обычно зависит от выбранной степени оптимизации.

Для небольших программ затраты на оптимизацию можно не принимать во внимание, но для больших и очень больших оно может иметь существенное значение с точки зрения пользователя. Поэтому малая степень оптимизации обычно используется в течение длительного процесса отладки, а глубокая оптимизация включается в тот момент, когда отлаженная программа передается в эксплуатацию.

Следует помнить, что использование оптимизатора может значительно усложнить отладку программы вследствие генерации такого кода, который трудно непосредственно связать с исходными операторами в программе.

Оптимизация может также неожиданно внести ошибки в код, сгенерированный из вполне правильного текста программы. В частности, вынесение инвариантного кода может быть потенциальным источником ошибок. Например, для такого фрагмента

```
int a[10], x, y;
for(i = 0; i < 10; i++)
        if( y != 0 )
                a[i] = x / y;
```

оптимизатор может решить, что значение выражения *x/y* есть инвариант, и вынесет его за пределы цикла, игнорируя предшествующую проверку *if(y != 0)* и создавая тем самым потенциальную возможность возникновения ситуации деления на ноль.

Обнаружить все подобные ситуации – задача практически неразрешимая, поэтому к коду, построенному оптимизатором, следует относиться с должной осторожностью, проверяя возможность появления ошибок путем сравнения результатов работы оптимизированной и неоптимизированной программ.

ЗАКЛЮЧЕНИЕ

В первой части учебного пособия были изложены задачи лексического анализа, алгоритмы и методы построения лексических анализаторов.

Вторая часть была посвящена рассмотрению методов и алгоритмов синтаксического анализа и построению конечных автоматов для проверки синтаксической правильности текста программы и преобразования его в промежуточное представление – постфиксную запись.

В настоящей части пособия рассмотрены следующие вопросы.

1. Основные понятия семантики современных языков программирования, в том числе: адреса (*l-value*) и значения (*r-value*), базовые и производные типы данных, среды ссылок периода трансляции и периода исполнения, активация процедуры и запись активации, обеспечение доступа из процедур к локальным и нелокальным данным, параметры процедур и способы их передачи.

2. Задачи и функции семантического анализа: установление ассоциаций между объектами и их наименованиями, статические и стековые методы создания записей активации процедур, управление дисплеем как наиболее распространенным средством реализации текстуальной видимости объектов, контроль типов данных для каждой исполняемой операции, способы проверки эквивалентности типов, контроль структуры программы.

3. Генерация промежуточного и объектного кода: понятия тетрад и триад, алгоритмы преобразования постфиксной записи в последовательность тетрад, управление временными переменными для промежуточных результатов, формирование вызывающих последовательностей, формирование адресов используемых объектов, методы и алгоритмы установления связей между базовыми блоками программы и используемыми в них переменными.

4. Оптимизация кода, в том числе наиболее употребительные и эффективные методы выявления общих выражений, сжатия цепочек переходов, понижения мощности операций и оптимизации циклов.

Совместно с первыми двумя частями изложенный материал охватывает теоретические основы и методы проектирования тех трансляторов, которые принято называть компиляторами.

СПИСОК ЛИТЕРАТУРЫ

1. *Ахо А., Сети Р.*, *Ульман Д.* Компиляторы: принципы, технологии и инструменты. – М.: Изд. дом «Вильямс», 2001.

2. *Свердлов С.З.* Языки программирования и методы трансляции: учеб. пособие для вузов. – СПб.: Питер, 2007.

3. *Малявко А.А.* Системное программное обеспечение. Методы трансляции: учеб. пособие.– Новосибирск: Изд-во НГТУ, 2011. – Ч. 1.

4. *Малявко А.А.* Системное программное обеспечение. Методы трансляции: учеб. пособие.– Новосибирск: Изд-во НГТУ, 2011. – Ч. 2.

5. *Карпов Ю.Г.* Теория и технология программирования. Основы построения трансляторов: учеб. пособие. – СПб.: БХВ-Петербург, 2005.

6. *Гавриков М.М.*, *Иванченко А.Н.*, *Гринченков Д.В.* Теоретические основы разработки и реализации языков программирования. – М.: Кнорус, 2010.

7. *Гордеев А.В., Молчанов А.Ю.* Системное программное обеспечение / учебник для вузов. – СПб.: Питер, 2001.

8. *Хопкрофт Д.*, *Мотвани Р.*, *Ульман Д.* Введение в теорию автоматов, языков и вычислений. – СПб.: Вильямс, 2002.

9. *Пратт Т.*, *Зелковиц М.* Языки программирования: реализация и разработка. – СПб.: Питер, 2001.

10. *Рейуорд-Смит В.* Теория формальных языков. Вводный курс. – М.: Радио и связь, 1988.

11. *Хантер Р.* Проектирование и конструирование компиляторов. – М.: Финансы и статистика, 1984.

12. *Ахо А.*, *Ульман Д.* Теория синтаксического анализа, перевода и компиляции. – М.: Мир, 1978.

13. *Льюис Ф.*, *Розенкранц Д.*, *Стирнз Р.* Теоретические основы проектирования компиляторов. – М.: Мир, 1979.

ОГЛАВЛЕНИЕ

Малявко Александр Антонович

СИСТЕМНОЕ ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ

ФОРМАЛЬНЫЕ ЯЗЫКИ И МЕТОДЫ ТРАНСЛЯЦИИ

ЧАСТЬ 3

Учебное пособие

Редактор *Л.Н. Ветчакова*
Выпускающий редактор *И.П. Брованова*
Дизайн обложки *А.В. Ладыжская*
Компьютерная верстка *Л.А. Веселовская*

Подписано в печать 04.05.2012. Формат 60 × 84 1/16. Бумага офсетная. Тираж 100 экз.
Уч.-изд. л. 6,97. Печ. л. 7,5. Изд. № 63. Заказ № Цена договорная

Отпечатано в типографии
Новосибирского государственного технического университета
630092, г. Новосибирск, пр. К. Маркса, 20