ГЕНЕРАЦИЯ И ОПТИМИЗАЦИЯ КОДА

Семантический анализ и подготовка к генерации кода

Назначение семантического анализа

Полный распознаватель для большинства языков программирования может быть построен в рамках КЗ-языков , поскольку все реальные языки программирования контекстно-зависимы.

Однако известно, что такой распознаватель имеет экспоненциальную зависимость требуемых для выполнения разбора исходной программы вычислительных ресурсов от длины входной цепочки [4 т.1, 5, 15, 58]. Компилятор, построенный на основе такого распознавателя, будет неэффективным с точки зрения скорости работы (либо объема необходимой памяти). Поэтому такие компиляторы практически не используются, а все реально существующие компиляторы выполняют анализ исходной программы в два этапа: первый — синтаксический анализ на основе распознавателя для одного из известных классов КС-языков; второй — семантический анализ.

Для проверки семантической правильности исходной программы необходимо иметь всю информацию о найденных лексических единицах языка.

Примерами таких конструкций являются блоки описаний констант и идентификаторов (если они предусмотрены семантикой языка) или операторы, где тот или иной идентификатор встречается впервые (если семантика языка предусматривает описание идентификатора по факту его первого использования).

Семантический анализ обычно выполняется на двух этапах компиляции

на этапе синтаксического разбора

- всякий раз по завершении анализа определенной синтаксической конструкции входного языка выполняется ее семантическая проверка на основе данных, имеющихся в таблице идентификаторов (такими конструкциями, как правило, являются процедуры, функции и блоки операторов входного языка)

в начале этапа подготовки к генерации кода

- после завершения всей фазы синтаксического анализа выполняется полный семантический анализ программы на основании данных в таблице идентификаторов (сюда попадает, например, поиск неописанных идентификаторов).

Этапы семантического анализа:

- проверка соблюдения в исходной программе семантических соглашений входного языка;

- дополнение внутреннего представления программы в компиляторе операторами и действиями, неявно предусмотренными семантикой входного языка;

- проверка элементарных семантических (смысловых) норм языков программирования, напрямую не связанных с входным языком.

Проверка соблюдения во входной программе семантических соглашений входного языка

Эта проверка заключается в сопоставлении входных цепочек исходной программы с требованиями семантики входного языка программирования. Каждый язык программирования имеет четко специфицированные семантические соглашения, которые не могут быть проверены на этапе синтаксического разбора. Именно их в первую очередь проверяет семантический анализатор.

Примерами таких соглашений являются следующие требования:

- Каждая метка, на которую есть ссылка, должна один раз, присутствовать в программе;
- Каждый идентификатор должен быть описан один раз и ни один идентификатор не может быть описан более одного раза (с учетом блочной структуры описаний);
- Все операнды в выражениях и операциях должны иметь типы, допустимые для данного выражения или операции;
- Типы переменных в выражениях должны быть согласованы между собой;
- При вызове процедур и функций число и типы фактических параметров должны быть согласованы с числом и типами формальных параметров.

Например, если мы возьмем оператор языка Pascal, имеющий вид a := b + c;

то с точки зрения синтаксического разбора это будет абсолютно правильный оператор.

Однако нельзя сказать, является ли этот оператор правильным с точки зрения входного языка (Pascal), пока не будут проверены семантические требования для всех входящих в него лексических элементов. Такими элементами здесь являются идентификаторы a , b и c. Не зная, что они собой представляют, невозможно не только окончательно утверждать правильность приведенного выше оператора, но и понять его смысл. Фактически, необходимо знать описание этих идентификаторов.

Дополнение внутреннего представления программы

Если вернуться к рассмотренному выше элементарному оператору языка Pascal a := b + c;

то можно отметить, что здесь выполняются две операции: одна операция сложения (или конкатенации, в зависимости от типов операндов) и одна операция присвоения результата. Соответствующим образом должен быть порожден и код результирующей программы.

Однако не все так очевидно просто. Допустим, что где-то перед рассмотренным оператором мы имеем описание его операндов в виде

var

a : double; b : integer; c : real;

из этого описания следует, что c — вещественная переменная языка Pascal, b — цело-численная переменная, a — вещественная переменная с двойной точностью.

Существуют правила преобразования типов, принятые для данного языка. Кто должен выполнять эти преобразования?

Это может сделать разработчик программы — но тогда преобразования типов в явном виде должны будут присутствовать в тексте входной программы. Для рассмотренного примера это выглядело бы примерно так:

a := double(real(b) + c).

Однако разработчик исходной программы может не указывать явно используемые преобразования типов. Тогда необходимые преобразования типов выполняет код, порождаемый компилятором , если эти преобразования предусмотрены семантическими соглашениями языка. Для этого в составе библиотек функций, доступных компилятору, должны быть функции преобразования типов (более подробно состав библиотек компилятора описан в главе «Современные системы программирования.

Преобразование типов — это только один вариант операций, неявно добавляемых компилятором в код результирующей программы на основе семантических соглашений. Существуют и другие варианты такого рода операций (преобразование типов — самый распространенный пример).

Таким образом, и в этом случае действия, выполняемые семантическим анализатором, существенным образом влияют на порождаемый компилятором код результирующей программы.

Проверка смысловых норм языков программирования

Проверка элементарных смысловых норм языков программирования, напрямую не связанных с входным языком, — это сервисная функция, которую предоставляет разработчикам большинство современных компиляторов.

Эта функция обеспечивает проверку компилятором соглашений, выполнение которых связано со смыслом как всей исходной программы в целом, так и отдельных ее фрагментов. Эти соглашения применимы к большинству современных языков программирования.

Примерами таких соглашений являются следующие требования:

- каждая переменная или именованная константа должна хотя бы один раз использоваться в программе;
- каждая переменная должна быть определена до ее первого использования при любом ходе выполнения программы (первому использованию переменной должно всегда предшествовать присвоение ей какого-либо значения);
- результат функции должен быть определен при любом ходе ее выполнения;
- каждый оператор в исходной программе должен иметь возможность хотя бы один раз выполниться;
- операторы условия и выбора должны предусматривать возможность хода выполнения программы по каждой из своих ветвей;
-
- операторы цикла должны предусматривать возможность завершения цикла.

Рассмотрим в качестве примера функцию, представляющую собой фрагмент вход-ной программы на языке С:

```
int f_test(int a) { int b,c;
b=0;
c=0;
if(b=1) { return a; } c=a+b;
}
```

Практически любой современный компилятор языка С обнаружит в данном месте входной программы массу «неточностей». Например, переменная с описана, ей присваивается значение, но она нигде не используется. Значение переменной b, присвоенное в операторе b=0;, тоже никак не используется. Наконец, условный оператор лишен смысла , так как всегда предусматривает ход выполнения только по одной своей ветке, а значит , и оператор c=a+b; никогда выполнен не будет. Скорее всего, компилятор выдаст еще одно предупреждение, характерное именно для языка С — в операторе if(b=1) присвоение стоит в условии (это не запрещено ни синтаксисом , ни семантикой языка С, но является очень распространенной семантической ошибкой в языке С). В принципе, смысл (а точнее, бессмысленность) этого фрагмента будет правильно воспринят и обработан компилятором

Однако если взять аналогичный по смыслу, но синтаксически более сложный фрагмент программы, то картина будет несколько иная:

```
int f_test_add(int* a, int* b)
{
*a=1;
*b=0; return *a;
}
int f_test(int a) { int b,c;
b=0;
if (f_test(&b,&c)!=0) { return a; } c=a+b;
}
```

Здесь компилятор уже вряд ли сможет выяснить порядок изменения значений переменных и выполнение условий в данном фрагменте из двух функций (обе они сами по себе независимо вполне осмысленны!). Единственное предупреждение, которое, скорее всего, получит в данном случае разработчик, — это то, что функция f_test не всегда корректно возвращает результат (отсутствует оператор return перед концом функции). И то это предупреждение на самом деле не будет соответствовать истин-ному положению вещей!

Идентификация лексических единиц языков программирования

Идентификация переменных, типов, процедур, функций и других лексических единиц языков программирования — это установление однозначного соответствия между лексическими единицами и их именами в тексте исходной программы. Идентификация лексических единиц языка чаще всего выполняется на этапе семантического анализа.

Примерный перечень действий компиляторов для идентификации переменных, констант, функций, процедур и других лексических единиц языка:

- имена локальных переменных дополняются именами тех блоков (функций, процедур), в которых эти переменные описаны;
- имена внутренних переменных и функций модулей исходной программы дополняются именами самих модулей (это касается только внутренних имен);
- имена процедур и функций, принадлежащих объектам (классам) в объектно-ориентированных языках программирования дополняются наименованиями типов объектов (классов), которым они принадлежат;
- имена процедур и функций модифицируются в зависимости от типов их формальных аргументов.

СОВЕТ

Правила, по которым происходит модификация имен, достаточно просты. Их можно выяснить для конкретной версии компилятора и использовать при разработке. Однако лучше этого не делать, так как нет гарантии, что при переходе к другой версии компилятора эти правила не изменятся — тогда код станет неработоспособным. Правильным средством будет аккуратное использование механизма отключения именования лексических единиц, предоставляемое синтаксисом исходного языка.

Распределение памяти

Распределение памяти — это процесс, который ставит в соответствие лексическим единицам исходной программы адрес, размер и атрибуты области памяти, необходимой для этой лексической единицы.

Область памяти — это блок ячеек памяти, выделяемый для данных, каким-то образом объединенных логически.

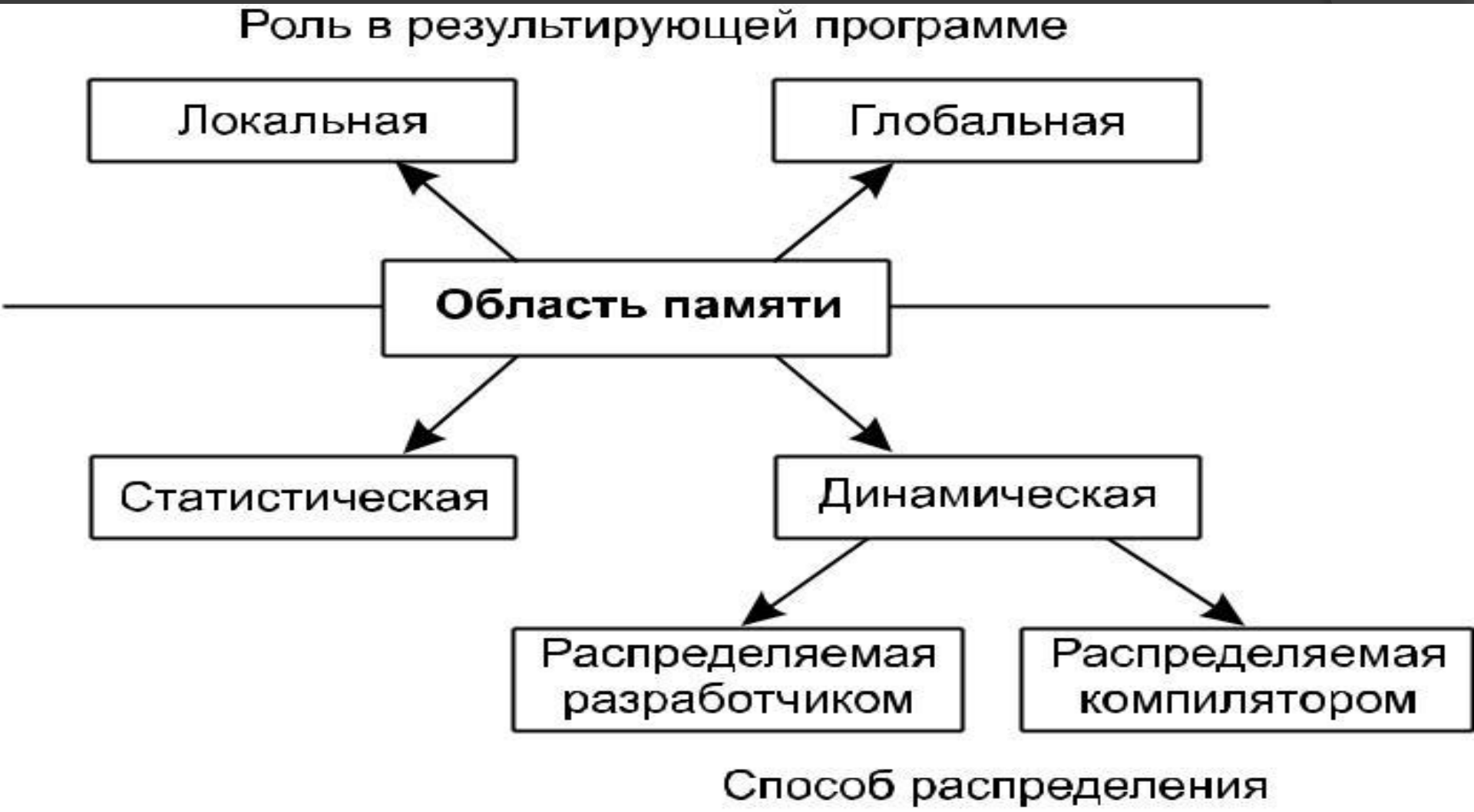
Распределение памяти работает с лексическими единицами языка — переменными, константами, функциями и т. п. — и с информацией об этих единицах, полученной на этапах лексического и синтаксического анализа.

Процесс распределения памяти в современных компиляторах, как правило, работает с относительными, а не с абсолютными (физическими) адресами ячеек памяти.

Распределение памяти выполняется перед генерацией кода результирующей программы, потому что его результаты должны быть использованы в процессе генерации кода.

Классификация областей памяти

Область памяти в зависимости от ее роли и способа распределения



Простые и сложные структуры данных. Выравнивание границ данных

Распределение памяти для переменных скалярных типов

Во всех языках программирования существует понятие так называемых «базовых типов данных», которые также называют основными или *скалярными* типами . Размер области памяти, необходимый для лексемы скалярного типа, считается известным. Он определяется семантикой языка и архитектурой целевой вычислительной системы, на которой должна выполняться результирующая программа.

Идеальным вариантом для разработчиков программ был бы такой компилятор, у которого размер памяти для базовых типов зависел бы только от семантики языка. Но чаще всего зависимость результирующей программы от архитектуры целевой вычислительной системы полностью исключить не удается. Создатели компиляторов и языков программирования предлагают механизмы, позволяющие свести эту зависимость к минимуму.

СОВЕТ

Размер области памяти каждого скалярного типа данных фиксирован и известен для определенной целевой вычислительной системы. Однако не рекомендуется непосредственно использовать его в тексте исходной программы, так как это ограничивает переносимость программы. Вместо этого нужно использовать функции определения размера памяти, предоставляемые входным языком программирования.

Распределение памяти для сложных структур данных

Правила распределения памяти под основные виды структур данных:

1. *для массивов* — произведение числа элементов в массиве на размер памяти для одного элемента (то же правило применимо и для строк, но во многих языках строки содержат еще и дополнительную служебную информацию фиксированного объема);
2. *для структур* (записей с именованными полями) — сумма размеров памяти по всем полям структуры;
3. - *для объединений* (союзов, общих областей, записей с вариантами) — размер максимального поля в объединении;
4. - *для реализации объектов* (классов) — размер памяти для структуры с такими же именованными полями плюс память под служебную информацию объектно-ориентированного языка (как правило, фиксированного объема).

Формулы для вычисления объема памяти :

для массивов: $V_{мас} = \prod_{i=1,n}(m_i)V_{эл}$,

где n — размерность массива, m_i — количество элементов i-й размерности, $V_{эл}$ — объ-ем памяти для одного элемента;

для структур: $V_{стр} = \sum_{i=1,n} V_{поля_i}$,

где n — общее количество полей в структуре, $V_{поля i}$ — объем памяти для i-го поля структуры;

для объединений: $V_{стр} = \max_{i=1,n} V_{поля i}$, где n — общее количество полей в объединении, $V_{поля i}$ — объем памяти для i-го поля объединения.

Выравнивание границ областей памяти

Говоря об объеме памяти, занимаемой различными лексемами языка , следует упомянуть еще один момент, связанный с выравниванием границ областей памяти, отводимых для различных лексических единиц . Архитектура многих современных вычислительных систем предусматривает, что обработка данных выполняется более эффективно, если адрес, по которому выбираются данные, кратен определенному числу байт[1] (как правило, это 2, 4, 8 или 16 байт). Современные компиляторы учитывают особенности целевых вычислительных систем. При распределении данных они могут размещать области памяти для лексем наиболее оптимальным образом. Поскольку не всегда размер памяти, отводимой под лексическую единицу, кратен указанному числу байт, то в общем объеме памяти, отводимой под результирующую программу, могут появляться неиспользуемые области.

Статическое и динамическое связывание. Менеджеры памяти

Глобальная и локальная память

Глобальная область памяти — это область памяти, которая выделяется один раз при инициализации результирующей программы и действует все время выполнения результирующей программы.

Локальная область памяти — это область памяти , которая выделяется в начале выполнения некоторого фрагмента результирующей программы (блока, функции, процедуры или оператора) и может быть освобождена по завершении выполнения данного фрагмента.

Распределение памяти на локальные и глобальные области целиком определяется семантикой входного языка. Только зная смысл синтаксических конструкций входного языка, можно четко сказать, какая из них будет отнесена в глобальную область памяти, а какая — в локальную. Иногда в исходном языке для некоторых конструкций нет четкого разграничения, тогда решение об их отнесении в ту или иную область памяти принимается разработчиками компилятора и может зависеть от и-пользуемой версии компилятора. При этом разработчики исходной программы не должны полагаться на тот факт, что один раз принятое решение будет неизменным во всех версиях компилятора.

Рассмотрим для примера фрагмент текста модуля программы на языке Pascal:

```
...
const
Global_1 = 1;
Global_2 : integer = 2;
var
Global_I : integer;
```

```
...
function Test (Param: integer): pointer; const
Local_1 = 1;
Local_2 : integer = 2;
var
Local_I : integer;
begin
...
end;
```

Статическая и динамическая память

Статическая область памяти — это область памяти, размер которой известен на этапе компиляции.

Поскольку для статической области памяти известен размер, компилятор всегда может выделить эту область памяти и связать ее с соответствующим элементом программы.

Статические области памяти обрабатываются компилятором самым простейшим образом, поскольку напрямую связаны со своим адресом. В этом случае говорят о *статическом связывании* области памяти и лексической единицы входного языка.

Динамическая область памяти — это область памяти , размер которой на этапе компиляции программы не известен.

Размер динамической области памяти будет известен только в процессе выполнения результирующей программы. Поэтому для динамической области памяти компилятор не может определить адрес — для нее он порождает фрагмент кода, который отвечает за распределение памяти (ее выделение и освобождение).

Менеджеры памяти

Многие компиляторы объектно-ориентированных языков программирования используют для работы с динамической памятью специальный *менеджер памяти* , к которому обращаются как при выделении памяти по команде пользователя, так при выделении памяти самим компилятором. Менеджер памяти обеспечивает выполнение функций выделения и освобождения используемой оперативной памяти следит за ее наиболее рациональным использованием.

При создании менеджера памяти разработчики компилятора преследуют две основные цели:

- сокращается количество обращений результирующей программы к системным функциям ОС, обеспечивающим выделение и освобождение оперативной памяти, а поскольку это довольно сложные функции, то в целом увеличивается быстродействие результирующей программы;
- сокращается фрагментация оперативной памяти, характерная именно для объектно-ориентированных языков, поскольку менеджер памяти запрашивает у ОС оперативную память укрупненными фрагментами и освобождает ее также укрупненными фрагментами.

Дисплей памяти процедуры (функции). Стековая организация дисплея памяти

Понятие дисплея памяти процедуры (функции)

Дисплей памяти процедуры (функции) — это область данных, доступных для обработки в этой процедуре (функции).

Как правило, дисплей памяти процедуры включает следующие составляющие:

глобальные данные (переменные и константы) всей программы; формальные аргументы процедуры; локальные данные (переменные и константы) данной процедуры.

Также в дисплей памяти часто включают адрес возврата.

Адрес возврата — это адрес того фрагмента кода результирующей программы, куда должно быть передано управление после того, как завершится выполнение вызванной процедуры или функции.

Стековая организация дисплея памяти процедуры (функции)

Стековая организация дисплея памяти процедуры (функции) основана на том, что для хранения параметров (аргументов) процедур и функций, их локальных переменных, а также адреса возврата в результирующей программе выделяется специальная область памяти, одна на всю программу , организованная в виде стека. Этот стек называют *стеком передачи параметров*, или просто *стеком параметров*.

При стековой организации дисплея памяти процедуры или функции в момент ее вызова все параметры и адрес возврата помещаются в стек параметров. При завершении выполнения процедуры или функции параметры извлекаются («выталкиваются») из стека, а управление передается по адресу возврата.

Внутри вызываемой процедуры или функции на верхушке стека может быть выделено место для хранения всех ее локальных переменных. Объектный код процедуры или функции адресуется к параметрам и своим локальным переменным по смещениям относительно верхушки стека.

Исключительные ситуации и их обработка

Понятие *исключительной ситуации* (exception) появилось в современных объектно-ориентированных языках программирования.

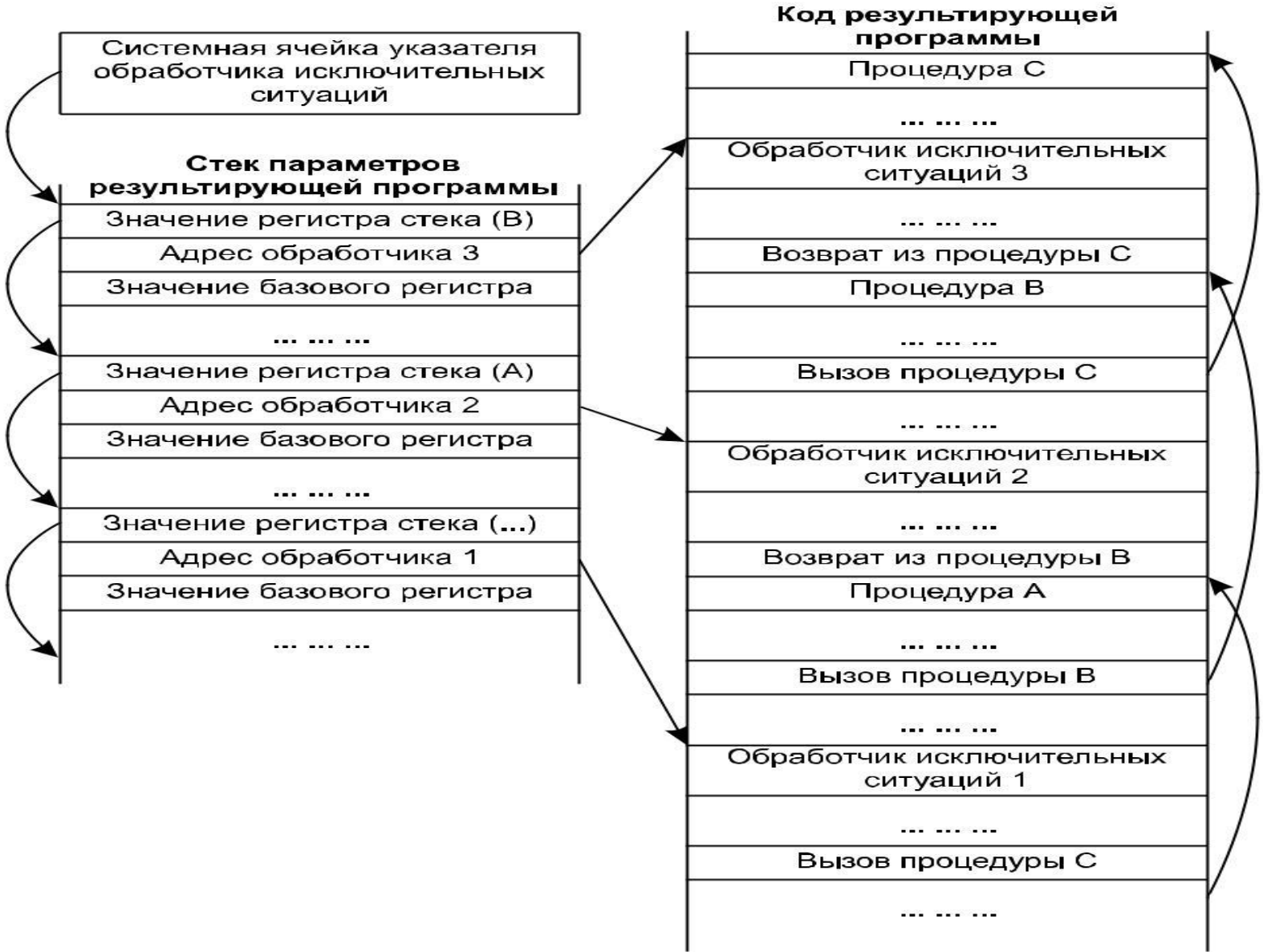
Проблема заключалась в том, что в таких языках есть специального вида функции — конструкторы (constructor) и специального вида процедуры — деструкторы (destructor), которые выполняют действия по созданию объектов (классов) и уничтожению их соответственно. При создании объектов выделяется необходимая для этого область памяти, а при освобождении объектов эта область памяти должна освобождаться. Как и при выполнении любых процедур и функций, при выполнении этих специальных процедур и функций могут произойти нештатные ситуации, вызванные кодом результирующей программы или кодом вызываемых ею библиотек ОС (например, ситуация недостатка оперативной памяти при ее выделении). Но в отличие от всех остальных процедур и функций, которые обычно возвращают в таких ситуациях предусмотренный код ошибки, конструктор и деструктор не могут вернуть код ошибки при возникновении нештатной ситуации, поскольку имеют строго определенный и неизменный смысл.

Обработчики исключительных ситуаций

За обработку исключительных ситуаций отвечают специальные синтаксические конструкции, предусмотренные в объектно-ориентированных языках программирования. Примерами таких конструкций являются блоки типа try … except … и try … finally … в языке программирования Pascal , блоки типа throw … catch …

- языке программирования C++ и др. Считается, что текст исходной программы, помещенный внутрь таких блоков, может вызвать возникновение исключительных ситуаций определенного типа. Текст исходной программы, помещенный в синтаксически выделенную часть таких блоков, считается текстом обработчика исключительных ситуаций, специальные синтаксические конструкции и семантические правила
- каждом языке программирования служат для определения типа обрабатываемых исключительных ситуаций. Компилятор анализирует исходный текст таких блоков и порождает соответствующий объектный код результирующей программы.

Простейший механизм обработки исключительных ситуаций через стек параметров



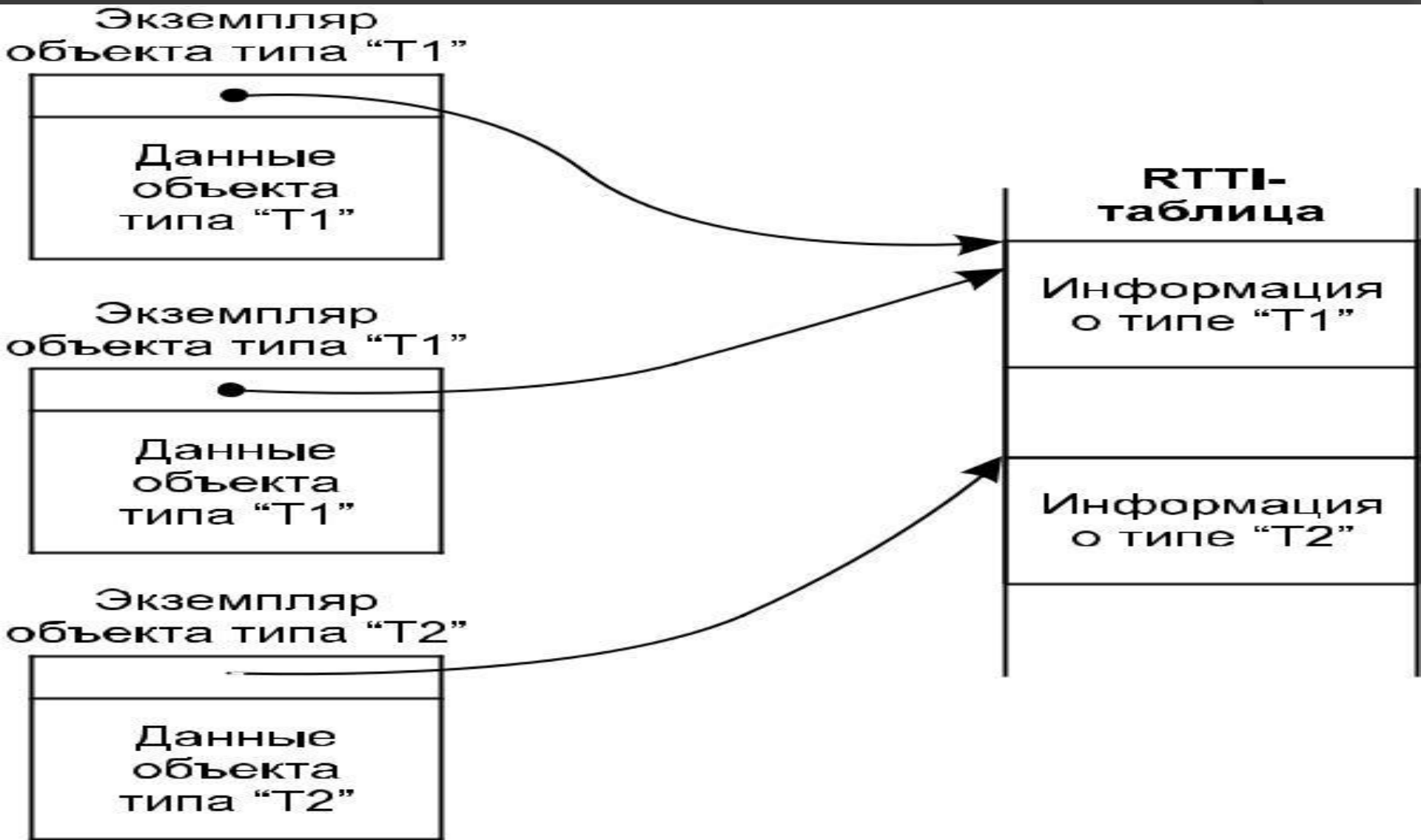
Память для типов данных (RTTI-информация)

объектно-ориентированных языках программирования существует понятие виртуальных (virtual) функций (или процедур) . При обращении к таким функциям и процедурам адрес вызываемой функции или процедуры становится известным только в момент выполнения результирующей программы. Такой вариант вызова носит название «позднее связывание» . Поскольку адрес вызываемой процедуры или функции зависит от того типа данных, для которого она вызывается, в современных компиляторах для объектно-ориентированных языков программирования предусмотрены специальные структуры данных для организации таких вызовов.

Time Type Information — «информация о типах во время выполнения».

Состав RTTI -информации определяется семантикой входного языка и реализацией компилятора . Как правило, для каждого типа данных в объектно-ориентированном языке программирования создается уникальный идентификатор типа данных, который используется для сопоставления типов.

Взаимосвязь RTTI-таблицы с экземплярами объектов в результирующей программе



Генерация кода. Методы генерации кода

Общие принципы генерации кода. Синтаксически управляемый перевод Принципы генерации объектного кода

Генерация объектного кода — это перевод компилятором внутреннего представления исходной программы в цепочку символов выходного языка.

Генерация объектного кода порождает результирующую объектную программу на языке ассемблера или непосредственно на языке машинных кодов. Внутреннее представление программы может иметь любую структуру в зависимости от реализации компилятора, в то время как результирующая программа всегда представляет собой линейную последовательность команд. Поэтому генерация объектного кода (объектной программы) в любом случае должна выполнять действия, связанные с преобразованием сложных синтаксических структур в линейные цепочки.

Синтаксически управляемый перевод

Чтобы компилятор мог построить код результирующей программы для всей синтаксической конструкции исходной программы, часто используется метод, называемый синтаксически управляемым переводом — *СУ-переводом.*

Идея СУ-перевода основана на том, что синтаксис и семантика языка взаимосвязаны. Это значит, что смысл предложения языка зависит от синтаксической структуры этого предложения.

Способы внутреннего представления программ

Виды внутреннего представления программы

Результатом работы синтаксического анализатора на основе КС-грамматики входного языка является последовательность правил грамматики , примененных для построения входной цепочки.

Для полного представления о типе и структуре найденной и разобранной синтаксической конструкции входного языка в принципе достаточно знать последовательность номеров правил грамматики, примененных для ее построения. Однако форма представления этой достаточной информации может быть различной в зависимости как от реализации самого компилятора, так и от фазы компиляции. Эта форма называется *внутренним представлением программы* (иногда используются также тер-мины «промежуточное представление» или «промежуточная программа»).

Синтаксические деревья. Преобразование дерева разбора в дерево операций

Алгоритм преобразования дерева семантического разбора в дерево операций можно представить следующим образом:

Шаг 1. Если в дереве больше не содержится узлов, помеченных нетерминальными символами, то выполнение алгоритма завершено; иначе перейти к шагу 2.

Шаг 2. Выбрать крайний левый узел дерева, помеченный нетерминальным символом грамматики, и сделать его текущим. Перейти к шагу 3.

Шаг 3. Если текущий узел имеет только один нижележащий узел, то текущий узел необходимо удалить из дерева, а связанный с ним узел присоединить к узлу вышележащего уровня (исключить из дерева цепочку) и вернуться к шагу 1; иначе перейти к шагу 4.

Шаг 4. Если текущий узел имеет нижележащий узел (лист дерева), помеченный терминальным символом, который не несет семантической нагрузки, тогда этот лист нужно удалить из дерева и вернуться к шагу 3; иначе перейти к шагу 5.

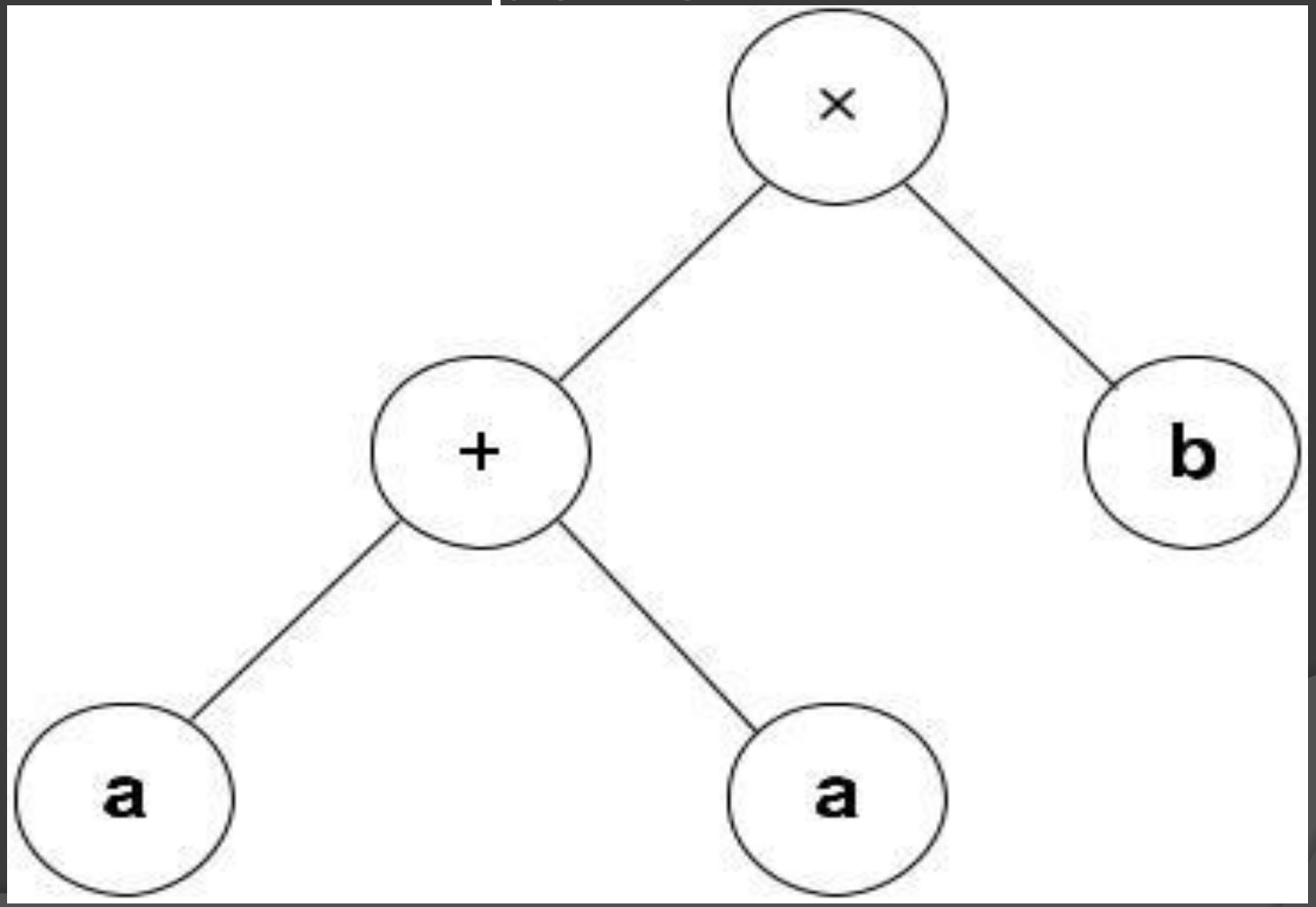
Шаг 5. Если текущий узел имеет один нижележащий узел (лист дерева), помеченный терминальным символом, обозначающим знак операции, а остальные узлы помечены как операнды, то лист, помеченный знаком операции, надо удалить из дерева, текущий узел пометить этим знаком операции и перейти к шагу 1; иначе перейти к шагу 6.

Шаг 6. Если среди нижележащих узлов для текущего узла есть узлы, помеченные нетерминальными символами грамматики, то необходимо выбрать крайний левый среди этих узлов, сделать его текущим узлом и перейти к шагу 3; иначе выполнение алгоритма завершено.

Дерево операций является формой внутреннего представления программы, которой удобно пользоваться на этапах синтаксического и семантического анализа, а также подготовки к генерации кода, когда еще нет необходимости работать непосредствен-но с кодами команд результирующей программы

В результате применения алгоритма преобразования деревьев синтаксического разбора в дерево операций к деревьям, . подучим дерево операций представленное на рис

Пример дерева операций для языка арифметических выражений



Многоадресный код с явно именуемым результатом (тетрады)

Тетрады представляют собой запись операций в форме из четырех составляющих: операции, двух операндов и результата операции. Например, тетрады могут выглядеть так: <операция>(<операнд1>,<операнд2>,<результат>).

Тетрады представляют собой линейную последовательность команд. При вычислении выражения, записанного в форме тетрад, они вычисляются одна за другой последовательно. Каждая тетрада в последовательности вычисляется так: операция, заданная тетрадой, выполняется над операндами и результат ее выполнения помещается в переменную, заданную результатом тетрады. Если какой-то из операндов (или оба операнда) в тетраде отсутствует (например, если тетрада представляет собой унарную операцию), то он может быть опущен или заменен пустым операндом (в зависимости от принятой формы записи).

Необходимость иметь алгоритм, отвечающий за распределение памяти для хранения промежуточных результатов, не является недостатком, так как это дает возможность эффективно распределять результаты не только по доступным ячейкам памяти, но и по имеющимся регистрам процессора. В этом есть определенные преимущества. Кроме того, триады ближе к двухадресным машинным командам, чем тетрады, а именно такие команды более всего распространены в наборах команд большинства современных компьютеров.

Например, выражение A:=B*C+D—B*10, записанное в виде триад, будет иметь вид

-
- (B, C) + (^1, D)
- (B, 10)
-
- — (^2, ^3)
-
- := (A, ^4)

Здесь операции обозначены соответствующим знаком (присвоение также является операцией), а знак ^ означает ссылку операнда одной триады на результат другой.

Постфиксная запись операций

Постфиксная (обратная польская) запись — очень удобная для вычисления выражений форма записи операций и операндов. Эта форма предусматривает, что знаки операций записываются после операндов.

Более подробно постфиксная запись операций рассмотрена далее в разделе «Обратная польская запись операций».

Ассемблерный код и машинные команды

Машинные команды удобны тем, что при их использовании внутреннее представление программы полностью соответствует объектному коду и сложные преобразования не требуются. Команды ассемблера представляют собой лишь форму записи машинных команд, а потому в качестве формы внутреннего представления программы практически ничем не отличаются от них.

Однако использование команд ассемблера или машинных команд для внутреннего представления программы требует дополнительных структур для отображения взаимосвязи операций.

Тем не менее **машинные команды** — это язык, на котором должна быть записана результирующая программа. Поэтому компилятор так или иначе должен работать с ними.

Обратная польская запись операций

Описание и свойства обратной польской записи операций

Обратная польская запись — это постфиксная запись операций. Она была предложена польским математиком Я. Лукашевичем, откуда и происходит ее название.

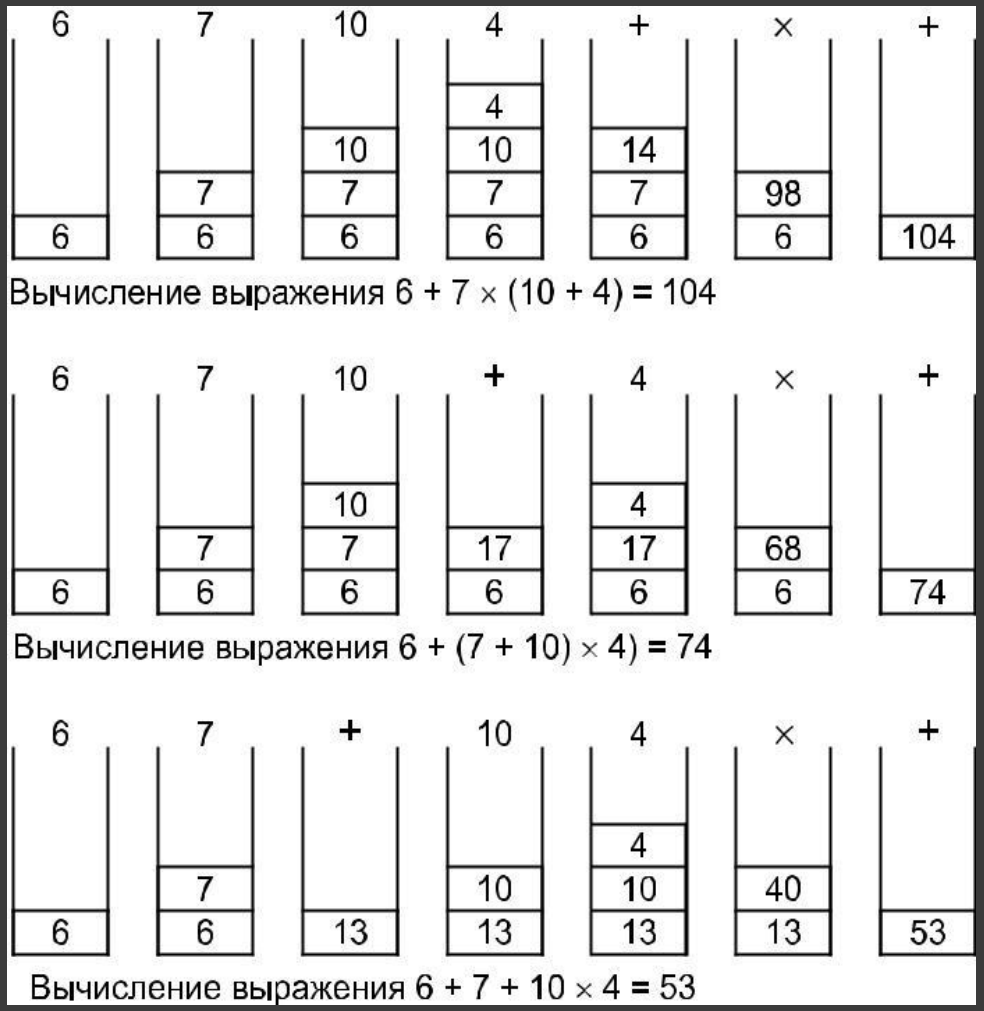
Она чрезвычайно эффективна в тех случаях, когда для вычислений используется стек. Ниже будет рассмотрен алгоритм вычисления выражений в форме обратной польской записи с использованием стека.

Главный недостаток обратной польской записи также проистекает из метода вычисления выражений в ней: поскольку используется стек, то для работы с ним всегда доступна только верхушка стека, а это делает крайне затруднительной оптимизацию выражений в форме обратной польской записи. Практически выражения в форме обратной польской записи почти не поддаются оптимизации.

Вычисление выражений с помощью обратной польской записи

Вычисление выражений в обратной польской записи выполняется элементарно просто с помощью стека. Для этого выражение просматривается в порядке слева на-право, и встречающиеся в нем элементы обрабатываются по следующим правилам:

1. если встречается операнд, то он помещается в стек (на верхушку стека);
2. если встречается знак унарной операции (требующей одного операнда), то операнд выбирается с верхушки стека, операция выполняется и результат помещается в стек (на верхушку стека);
3. если встречается знак бинарной операции (требующей двух операндов), то сна-чала с верхушки стека выбирается второй операнд, затем — первый операнд, операция выполняется и результат помещается в стек (на верхушку стека).



Вычисление выражений в обратной польской записи с использованием стека

Схема СУ-компиляции для перевода выражений в обратную польскую запись

Существует множество алгоритмов , которые позволяют преобразовывать выражения из обычной (инфиксной) формы записи в обратную польскую запись.

Далее рассмотрен алгоритм, построенный на основе схемы СУ-компиляции для языка арифметических выражений с операциями +, —, * и /. В качестве основы алгоритма выбрана грамматика арифметических выражений, которая уже многократно рассматривалась ранее в качестве примера.

Эта грамматика приведена далее:

G({+,—,/,*,a,b},{S,T,E},**P**,S):

P:

S → S+T | S—T | T

T → T*E | T/E | E E → (S) | a | b

Схему СУ-компиляции будем строить с таким расчетом, что имеется выходная цепочка символов R и известно текущее положение указателя в этой цепочке. Распознаватель, выполняя свертку или подбор альтернативы по правилу грамматики, может записывать символы в выходную цепочку и менять текущее положение указателя в ней.

Построенная таким образом схема СУ-компиляции для преобразования арифметических выражений в форму обратной польской записи оказывается исключительно простой [4 т.2, 5]. Она приведена ниже. В ней с каждым правилом грамматики связаны некоторые действия, которые записаны через ; (точку с запятой) сразу за правой частью каждого правила. Если никаких действий выполнять не нужно, в записи следует пустая цепочка (λ).

S → S+T; R(p)="+", p=p+1 S → S-T; R(p)="—", p=p+1 S → T; λ
T → T*E; R(p)="*", p=p+1 T → T/E; R(p)="/", p= p+1 T → E; λ
E → (S); λ

E → a; R(p)="a", p=p+1 E → b; R(p)="b", p=p+1

Оптимизация кода. Основные методы оптимизации

Общие принципы оптимизации кода

Для построения результирующего кода различных синтаксических конструкций входного языка используется метод СУ-перевода.

Построенный таким образом код результирующей программы может содержать лишние команды и данные. Это снижает эффективность выполнения результирующей программы. В принципе, компилятор может завершить на этом генерацию кода, поскольку результирующая программа построена и она является эквивалентной по смыслу (семантике) программе на входном языке. Однако эффективность результирующей программы важна для ее разработчика, поэтому большинство современных компиляторов выполняют еще один этап компиляции — оптимизацию результирующей программы (или просто «оптимизацию»), чтобы повысить ее эффективность, насколько это возможно.

Важно отметить два момента: во-первых, выделение оптимизации в отдельный этап генерации кода — это вынужденный шаг. Компилятор вынужден производить оптимизацию построенного кода, поскольку он не может выполнить семантический анализ всей входной программы в целом, оценить ее смысл и, исходя из него, построить результирующую программу. Оптимизация нужна, поскольку результирующая программа строится не вся сразу, а поэтапно. Во-вторых, оптимизация — это необязательный этап компиляции. Компилятор может вообще не выполнять оптимизацию, и при этом результирующая программа будет правильной, а сам компилятор будет полностью выполнять свои функции.

Оптимизация программы — это обработка, связанная с переупорядочиванием и изменением операций в компилируемой программе с целью получения более эффективной результирующей объектной программы . Оптимизация выполняется на эта-пах подготовки к генерации и непосредственно при генерации объектного кода.

В качестве показателей эффективности результирующей программы можно использовать два критерия: объем памяти, необходимый для выполнения результирующей программы (для хранения всех ее данных и кода), и скорость выполнения (быстро-действие) программы.

Чтобы оценить эффективность результирующей программы, полученной с помощью того или иного компилятора, часто прибегают к сравнению ее с эквивалентной программой (программой, реализующей тот же алгоритм), полученной из исходной программы, написанной на языке ассемблера.

Оптимизацию можно выполнять на любой стадии генерации кода, начиная от завершения синтаксического разбора и вплоть до последнего этапа, когда порождается код результирующей программы.

Таким образом, оптимизация в компиляторе может выполняться несколько раз на этапе генерации кода.

Принципиально различаются два основных вида оптимизирующих преобразований:

-преобразования исходной программы (в форме ее внутреннего представления в компиляторе), не зависящие от результирующего объектного языка;

-преобразования результирующей объектной программы.

Первый вид преобразований не зависит от архитектуры целевой вычислительной системы, на которой будет выполняться результирующая программа. Обычно он основан на выполнении хорошо известных и обоснованных математических и логических преобразований, производимых над внутренним представлением программы (некоторые из них будут рассмотрены ниже).

Второй вид преобразований может зависеть не только от свойств объектного языка (что очевидно), но и от архитектуры вычислительной системы, на которой будет выполняться результирующая программа.

Оптимизация может выполняться для следующих типовых синтаксических конструкций:

-линейных участков программы; логических выражений; циклов;

-вызовов процедур и функций; других конструкций входного языка.

Во всех случаях могут использоваться как машинно-зависимые , так и машинно-независимые методы оптимизации.

Оптимизация линейных участков программы

Принципы оптимизации линейных участков

Линейный участок программы — это выполняемая по порядку последовательность операций, имеющая один вход и один выход. Чаще всего линейный участок содержит последовательность вычислений, состоящих из арифметических операций и операторов присвоения значений переменным.

Для операций, составляющих линейный участок программы, могут применяться следующие виды оптимизирующих преобразований:

удаление бесполезных присваиваний; исключение избыточных вычислений (лишних операций); свертка операций объектного кода; перестановка операций; арифметические преобразования

В общем случае бесполезными могут оказаться не только операции присваивания, но и любые другие операции линейного участка, результат выполнения которых ни-где не используется.

Например, во фрагменте программы:

```
A := B * C;
D := B + C;
A := D * C;
```

Исключение избыточных вычислений (лишних операций) заключается в нахождении и удалении из объектного кода операций, которые повторно обрабатывают одни и те же операнды.

Свертка объектного кода — это выполнение во время компиляции тех операций исходной программы , для которых значения операндов уже известны . Тривиальным примером свертки является вычисление выражений, все операнды которых являются константами.

Перестановка операций заключается в изменении порядка следования операций, которое может повысить эффективность программы, но не будет влиять на конечный результат вычислений.

Например, операции умножения в выражении A:=2*B*3*C; можно переставить без изменения конечного результата и выполнить в порядке A:=(2*3)*(B*C);. Тогда представляется возможным выполнить свертку и сократить количество операций.

Арифметические преобразования представляют собой выполнение изменения характера и порядка следования операций на основании известных алгебраических и логических тождеств.

Например, выражение A:=B*C+B*D; может быть заменено на A:=B*(C+D);. Конечный результат при этом не изменится, но объектный код будет содержать на одну операцию умножения меньше.

Свертка объектного кода

Свертка объектного кода — это выполнение во время компиляции тех операций исходной программы, для которых значения операндов уже известны. Нет необходимости многократно выполнять эти операции в результирующей программе — вполне достаточно один раз выполнить их при компиляции программы.

Алгоритм свертки триад последовательно просматривает триады линейного участка для каждой триады делает следующее:

1. если операнд есть переменная, которая содержится в таблице **T**, то операнд за-меняется соответствующим значением константы;
2. если операнд есть ссылка на особую триаду типа C(K,0), то операнд заменяется значением константы K;
3. если все операнды триады являются константами, то триада может быть свер-нута. Тогда данная триада выполняется, и вместо нее помещается особая триада вида C(K,0), где K — константа, являющаяся результатом выполнения свернутой триады (при генерации кода для особой триады объектный код не порождается, а потому она в дальнейшем может быть просто исключена);
4. если триада является присваиванием типа A := B, тогда

- если B — константа, то A со значением константы заносится в таблицу **T** (если там уже было старое значение для A, то это старое значение исклю-чается);
- если B не константа, то A вообще исключается из таблицы **T**, если оно там есть.

Рассмотрим пример выполнения алгоритма.
Пусть фрагмент исходной программы (записанной на языке типа Pascal) имеет вид:
I := 1 + 1; I := 3;
J := 6*I + I;
Ее внутреннее представление в форме триад будет иметь вид:
+ (1,1).
:= (I,^1).
:= (I,3).
* (6,I).
+ (^4,I).
:= (J,^5).
Процесс выполнения алгоритма свертки показан в таблице

Триада	Шаг 1	Шаг 2	Шаг 3	Шаг 4	Шаг 5	Шаг 6
1	C (2, 0)	C (2, 0)	C (2, 0)	C (2, 0)	C (2, 0)	C (2, 0)
2	:= (I, ^1)	:= (I, 2)	:= (I, 2)	:= (I, 2)	:= (I, 2)	:= (I, 2)
3	:= (I, 3)	:= (I, 3)	:= (I, 3)	:= (I, 3)	:= (I, 3)	:= (I, 3)
4	* (6, I)	* (6, I)	* (6, I)	C (18, 0)	C (18, 0)	C (18, 0)
5	+ (^4, I)	+ (^4, I)	+ (^4, I)	+ (^4, I)	C (21, 0)	C (21, 0)
6	:= (J, ^5)	:= (J, ^5)	:= (J, ^5)	:= (J, ^5)	:= (J, ^5)	:= (J, 21)
T	(,)	(I, 2)	(I, 3)	(I, 3)	(I, 3)	(I, 3) (J, 21)

Исключение лишних операций

Алгоритм исключения лишних операций просматривает операции в порядке их следования. Так же как и алгоритму свертки, алгоритму исключения лишних операций проще всего работать с триадами, потому что они полностью отражают взаимосвязь операций.

Рассмотрим алгоритм исключения лишних операций для триад.

Чтобы следить за внутренней зависимостью переменных и триад, алгоритм присваивает им некоторые значения, называемые числами зависимости , по следующим правилам:

-изначально для каждой переменной ее число зависимости равно 0, так как в на-чале работы программы значение переменной не зависит ни от какой триады;

-после обработки i-й триады, в которой переменной A присваивается некоторое значение, число зависимости A (dep(A)) получает значение i, так как значение A теперь зависит от данной i-й триады;

-при обработке i-й триады ее число зависимости (dep(i)) принимается равным значению 1+(максимальное из чисел зависимости операндов).

Рассмотрим работу алгоритма на примере:

D := D + C*B;

A := D + C*B;

C := D + C*B;

Этому фрагменту программы будет соответствовать следующая последовательность триад:

* (C,B).

+ (D,^1).

:= (D,^2).

* (C,B).

+ (D,^4).

:= (A,^5).

* (C,B).

:= (C,^8).

Видно, что в данном примере некоторые операции вычисляются дважды над одними и теми же операндами, а значит , они являются лишними и могут быть исключены. Работа алгоритма исключения лишних операций отражена в таблице

Обрабатываемая триада i	Числа зависимости переменных A	B	C	D	Числа зависимости триад dep(i)	Триады, полученные после выполнения Алгоритма
1) * (C, B)	0	0	0	0	1	1) * (C, B)
2) + (D, ^1)	0	0	0	0	2	2) + (D, ^1)
3) := (D, ^2)	0	0	0	3	3	3) := (D, ^2)
4) * (C, B)	0	0	0	3	1	4) SAME (1, 0)
5) + (D, ^4)	0	0	0	3	4	5) + (D, ^1)
6) := (A, ^5)	6	0	0	3	5	6) := (A, ^5)
7) * (C, B)	6	0	0	3	1	7) SAME (1, 0)
8) + (D, ^7)	6	0	0	3	4	8) SAME (5, 0)
9) := (C, ^8)	6	0	9	3	5	9) := (C, ^5)

Другие методы оптимизации программ

Оптимизация вычисления логических выражений

Особенность оптимизации логических выражений заключается в том, что не всегда необходимо полностью выполнять вычисление всего выражения для того, чтобы знать его результат. Иногда по результату первой операции или даже по значению одного операнда можно заранее определить результат вычисления всего выражения.

Операция называется предопределенной для некоторого значения операнда, если ее

результат зависит только от этого операнда и остается неизменным (инвариантным) относительно значений других операндов.

Например, выражение A or B or C or D не имеет смысла вычислять, если известно, что значение переменной A есть True («истина»).

Не только логические операции могут иметь предопределенный результат. Некоторые математические операции и функции также обладают этим свойством. Напри-мер, умножение на 0 не имеет смысла выполнять.

Другой пример такого рода преобразований — это исключение вычислений для инвариантных операндов.

Операция называется инвариантной относительно некоторого значения операнда, если ее результат не зависит от этого значения операнда и определяется другими операндами.

Например, логическое сложение (or) инвариантно относительно значения «ложь» (False), логическое умножение (and) — относительно значения «истина»; алгебраическое сложение инвариантно относительно0, а алгебраическое умножение — относительно 1.

Выполнение такого рода операций можно исключить из текста программы, если известен инвариантный для них операнд. Этот метод оптимизации имеет смысл в сочетании с методом свертки операций.

Оптимизация передачи параметров в процедуры и функции

Данный метод прост в реализации и имеет хорошую аппаратную поддержку во многих архитектурах. Однако он является неэффективным в том случае, если процедура или функция выполняет несложные вычисления над небольшим количеством пара-метров. Тогда всякий раз при вызове процедуры или функции компилятор будет создавать объектный код для размещения ее фактических параметров в стеке, а при вы-ходе из нее — код для освобождения ячеек, занятых параметрами.

многократно, то этот код будет заметно снижать эффективность ее выполнения.

Существуют методы , позволяющие снизить затраты кода и времени выполнения на передачу параметров в процедуры и функции и повысить в результате эффективность результирующей программы:

- передача параметров через регистры процессора;
- подстановка кода функции в вызывающий объектный код.

Метод передачи параметров через регистры процессора позволяет разместить все или часть параметров, передаваемых в процедуру или функцию, непосредственно в регистрах процессора, а не в стеке. Это позволяет ускорить обработку параметров функции, поскольку работа с регистрами процессора всегда выполняется быстрее, чем с ячейками оперативной памяти, где располагается стек. Если все параметры удается разместить в регистрах, то сокращается также и объем кода, поскольку исключаются все операции со стеком при размещении в нем параметров.

Этот метод имеет ряд недостатков. Во-первых, очевидно, он зависит от архитектуры целевой вычислительной системы. Во-вторых, процедуры и функции, оптимизированные таким методом, не могут быть использованы в качестве процедур или функций библиотек подпрограмм ни в каком виде. Это вызвано тем, что методы передачи параметров через регистры процессора не стандартизованы (в отличие от методов передачи параметров через стек) и зависят от реализации компилятора. Наконец, этот метод не может быть использован, если где бы то ни было в процедуре или функции требуется выполнить операции с адресами (указателями) на параметры.

Метод подстановки кода функции в вызывающий объектный код (так называемая inline-подстановка) основан на том, что объектный код функции непосредственно включается в вызывающий объектный код всякий раз в месте вызова функции.

Для разработчика исходной программы такая функция (называемая inline-функцией) ничем не отличается от любой другой функции, но для вызова ее в результирующей программе используется принципиально другой механизм. По сути, вызов функции

- результирующем объектном коде вовсе не выполняется — просто все вычисления, производимые функцией, выполняются непосредственно в самом вызывающем коде
- месте ее вызова.

Как правило, этот метод применим к очень простым функциям или процедурам, иначе объем результирующего кода может существенно возрасти. Кроме того, он применим только к функциям, вызываемым непосредственно по адресу, без приме-нения косвенной адресации через таблицы RTTI (см. раздел «Семантический анализ и подготовка к генерации кода»). Некоторые компиляторы допускают его применение только к функциям, предполагающим последовательные вычисления и не содержащим циклов.

Оптимизация циклов

Циклом в программе называется любая последовательность участков программы, которая может выполняться повторно.

Циклы присущи подавляющему большинству программ. Во многих программах есть циклы, выполняемые многократно. Большинство языков программирования имеют синтаксические конструкции, специально ориентированные на организацию циклов. Очевидно, такие циклы легко обнаруживаются на этапе синтаксического разбора.

Однако понятие цикла с точки зрения объектной программы , определенной выше, является более общим. Оно включает в себя не только циклы, явно описанные в синтаксисе языка. Циклы могут быть организованы в исходной программе с помощью любых конструкций, допускающих передачу управления (прежде всего, с помощью условных операторов и операторов безусловного перехода).

Для оптимизации циклов используются следующие методы:

вынесение инвариантных вычислений из циклов;

замена операций с индуктивными переменными;

слияние и развертывание циклов.

Вынесение инвариантных вычислений из циклов заключается в вынесении за пределы циклов тех операций, операнды которых не изменяются в процессе выполнения цикла. Очевидно, что такие операции могут быть выполнены один раз до начала цикла, а полученные результаты потом могут использоваться в теле цикла.

Например, цикл[1] for i:=1 to 10 do begin

A[i] := B * C * A[i];

…

end;

может быть заменен последовательностью операций

D := B * C;

for i:=1 to 10 do begin

A[i] := D * A[i];

…

end;

если значения B и C не изменяются нигде в теле цикла. При этом операция умножения B*C будет выполнена только один раз, в то время как в первом варианте она выполнялась 10 раз над одними и теми же операндами.

Замена операций с индуктивными переменными

Замена операций с индуктивными переменными заключается в изменении сложных операций с индуктивными переменными в теле цикла на более простые операции. Как правило, выполняется замена умножения на сложение.

Переменная называется индуктивной в цикле, если ее значения в процессе выполнения цикла образуют арифметическую прогрессию. Таких переменных в цикле может быть несколько , тогда в теле цикла их иногда можно заменить одной единственной переменной, а реальные значения для каждой переменной будут вычисляться с помощью соответствующих коэффициентов соотношения (всем переменным должны быть за пределами цикла присвоены значения, если, конечно, они используются).

После того как индуктивные переменные выявлены, необходимо проанализировать те операции в теле цикла, где они используются. Часть таких операций может быть упрощена. Как правило, речь идет о замене умножения на сложение [4 т.2, 5, 23, 63].

Например, цикл

```
S := 10;
for i:=1 to N do A[i] := i*S;
```

может быть заменен последовательностью операций

```
S := 10;
T := S; i := 1; while i <= 10 do begin
A[i] := T; T := T + 10; i := i + 1;
end;
```

Слияние и развертывание циклов

Слияние и развертывание циклов предусматривает два различных варианта преобразований: слияния двух вложенных циклов в один и замена цикла на линейную последовательность операций.

Слияние двух циклов можно проиллюстрировать на примере циклов for i:=1 to N do

for j:=1 to M do A[i,j] := 0;

Здесь происходит инициализация двумерного массива. Но в объектном коде двумерный массив — это всего лишь область памяти размером N * M, поэтому (с точки зрения объектного кода, но не входного языка!) эту операцию можно представить так:

K := N*M;

for i:=1 to K do A[i] := 0;

Развертывание циклов можно выполнить для циклов, кратность выполнения которых известна уже на этапе компиляции. Тогда цикл кратностью N можно заменить линейной последовательностью N операций, содержащих тело цикла.

Например, цикл

for i:=1 to 3 do A[i] := i;

можно заменить операциями

A[1] := 1;

A[2] := 2;

A[3] := 3;

Незначительный выигрыш в скорости достигается за счет исключения всех операций с индуктивной переменной, однако объем программы может существенно возрасти.

Машинно-зависимые методы оптимизации

Машинно-зависимые методы оптимизации ориентированы на конкретную архитектуру целевой вычислительной системы, на которой будет выполняться результирующая программа. Как правило , каждый компилятор ориентирован на одну определенную архитектуру целевой вычислительной системы. Иногда можно в настройках компилятора явно указать одну из допустимых целевых архитектур.

В любом случае результирующая программа всегда порождается для четко заданной целевой архитектуры.

Архитектура вычислительной системы есть представление аппаратной и программной составляющих частей системы и взаимосвязи между ними с точки зрения системы как единого целого. Понятие «архитектура» включает в себя особенности и аппаратных, и программных средств целевой вычислительной системы. При выполнении машинно-зависимой оптимизации компилятор может принимать во внимание те или иные ее составляющие. То, какие конкретно особенности архитектуры будут приняты во внимание, зависит от реализации компилятора и определяется его разработчиками.

Количество существующих архитектур вычислительных систем к настоящему времени очень велико. Поэтому не представляется возможным рассмотреть все ориентированные на них методы оптимизации даже в форме краткого обзора. Интересующиеся этим вопросом могут обратиться к специализированной литературе [4 т.2, 5, 33, 58, 59, 63]. Далее будут рассмотрены только два основных аспекта машинно-зависимой оптимизации: распределение регистров процессора и порождение кода для параллельных вычислений.

Распределение регистров процессора

Программно-доступных регистров процессора всегда ограниченное количество. Поэтому встает вопрос об их распределении при выполнении вычислений. Этим занимается алгоритм распределения регистров, который присутствует практически в каждом современном компиляторе в части генерации кода результирующей программы.

При распределении регистров под хранение промежуточных и окончательных результатов вычислений может возникнуть ситуация, когда значение той или иной переменной (связанной с нею ячейки памяти) необходимо загрузить в регистр для дальнейших вычислений, а все имеющиеся доступные регистры уже заняты. Тогда компилятору перед созданием кода по загрузке переменной в регистр необходимо сгенерировать код для выгрузки одного из значений из регистра в ячейку памяти (чтобы освободить регистр). Причем выгружаемое значение затем, возможно, придется снова загружать в регистр. Встает вопрос о выборе того регистра, значение которого нужно выгрузить в память.

При этом возникает необходимость выработки стратегии замещения регистров процессора. Она чем-то сродни стратегиям замещения процессов и страниц в памяти компьютера, которые используются в операционных системах. Однако разница состоит в том, что, в отличие от диспетчера памяти в ОС, компилятор может проанализировать код и выяснить, какое из выгружаемых значений ему понадобится для дальнейших вычислений и когда (следует помнить, что компилятор сам вычислений не производит — он только порождает код для них, иное дело — интерпретатор). Как правило, стратегии замещения регистров процессора предусматривают, что выгружается тот регистр, значение которого будет использовано в последующих операциях позже всех (хотя не всегда эта стратегия является оптимальной).

Оптимизация кода для процессоров, допускающих распараллеливание вычислений

Многие современные процессоры допускают возможность параллельного выполнения нескольких операций. Как правило, речь идет об арифметических операциях.

Тогда компилятор может порождать объектный код таким образом, чтобы в нем содержалось максимально возможное количество соседних операций, все операнды которых не зависят друг от друга. Решение такой задачи в глобальном объеме для всей программы в целом не представляется возможным, но для конкретного оператора оно, как правило, заключается в порядке выполнения операций. В этом случае нахождение оптимального варианта сводится к выполнению перестановки операций (если она возможна, конечно). Причем оптимальный вариант зависит как от характера операции, так и от количества имеющихся в процессоре конвейеров для выполнения параллельных вычислений.

Например, операцию A+B+C+D+E+F на процессоре с одним потоком обработки данных лучше выполнять в порядке ((((A+B)+C)+D)+E)+F. Тогда потребуется меньше ячеек для хранения промежуточных результатов, а скорость выполнения от порядка операций в данном случае не зависит.

Та же операция на процессоре с двумя потоками обработки данных в целях увеличения скорости выполнения может быть обработана в порядке ((A+B)+C)+((D+E)+F). Тогда по крайней мере операции A+B и D+E, а также сложение с их результатами могут быть обработаны в параллельном режиме. Конкретный порядок команд, а также распределение регистров для хранения промежуточных результатов будут зависеть от типа процессора.

На процессоре с тремя потоками обработки данных ту же операцию можно уже разбить на части в виде (A+B)+(C+D)+(E+F). Теперь уже три операции A+B, C+D и E+F могут быть выполнены параллельно. Правда, их результаты уже должны быть обработаны последовательно, но тут следует принять во внимание соседние операции для нахождения наиболее оптимального варианта.