ФЕДЕРАЛЬНОЕ АГЕНТСТВО ПО ОБРАЗОВАНИЮ
ГОСУДАРСТВЕННОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО
ПРОФЕССИОНАЛЬНОГО ОБРАЗОВАНИЯ
УФИМСКИЙ ГОСУДАРСТВЕННЫЙ АВИАЦИОННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ

Кафедра ТК

курс лекций по дисциплине

Системное программное обеспечение

Тема: Генерация и оптимизация кода

Преподаватель: к.т.н., доцент Карамзина А.Г.

Тема № 16

Генерация и оптимизация кода

- Общие принципы генерации кода
- Способы внутреннего представления программ
  - Синтаксические деревья
  - Многоадресный код с явно именуемым результатом
  - Многоадресный код с неявно именуемым результатом
  - Ассемблерный код или машинные команды
  - Обратная польская запись операций
- Оптимизация кода

Общие принципы генерации 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 машинном языке (*в машинных кодах*).

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.

Генерация объектного кода выполняется после того, как выполнен синтаксический анализ программы и все необходимые действия по подготовке к генерации кода:

- распределено адресное пространство под функции и переменные;
- проверено соответствие имен и типов переменных, констант и функций в синтаксических конструкциях исходной программы и т. д.

Характер отображения входной программы в последовательность команд, выполняемого генерацией, *зависит* от *входного языка*, *архитектуры вычислительной системы*, на которую ориентирована результирующая программа, а также от *качества* желаемого *объектного кода*.

Общие принципы генерации кода

В идеале компилятор должен выполнить синтаксический анализ всей входной программы, затем провести ее семантический анализ, после чего приступить к подготовке генерации и непосредственно генерации кода. Однако такая схема работы компилятора практически почти никогда не применяется.

Дело в том, что в общем случае ни один семантический анализатор и ни один компилятор не способны проанализировать и оценить смысл всей входной программы в целом.

Формальные методы анализа семантики применимы только к очень незначительной части возможных входных программ.

Поэтому у компилятора нет практической возможности порождать эквивалентную выходную программу на основе всей входной программы.

Общие принципы генерации кода

Как правило, компилятор выполняет генерацию результирующего кода поэтапно на основе законченных синтаксических конструкций входной программы.

Компилятор выделяет законченную синтаксическую конструкцию из текста входной программы, порождает для нее фрагмент результирующего кода и помещает его в текст выходной программы.

Затем он переходит к следующей синтаксической конструкции.

Так продолжается до тех пор, пока не будет разобрана вся входная программа.

В качестве анализируемых законченных синтаксических конструкций выступают операторы, блоки операторов, описания процедур и функций. *Их конкретный состав зависит от входного языка и реализации компилятора*.

Общие принципы генерации кода

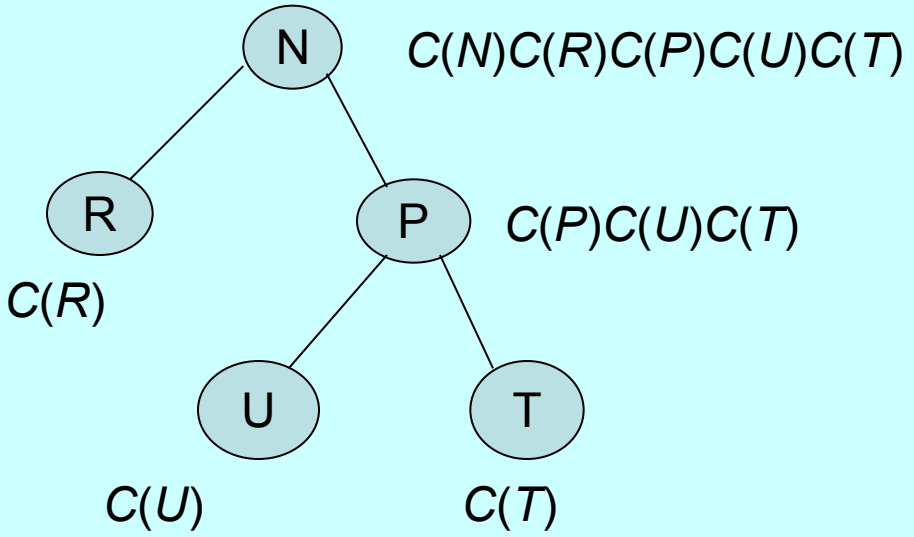
Чтобы компилятор мог построить код результирующей программы для синтаксической конструкции входного языка, часто используется метод, называемый синтаксически управляемым переводом

СУ-перевод – это основной метод порождения кода результирующей программы на основании результатов синтаксического анализа.

Для удобства понимания сути метода можно считать, что результат синтаксического анализа представлен в виде дерева синтаксического разбора (*либо же дерева операций*), хотя в реальных компиляторах это не всегда так.

Общие принципы генерации кода

Суть принципа СУ-перевода заключается в следующем: с каждой вершиной дерева синтаксического разбора *N* связывается цепочка некоторого промежуточного кода *C*(*N*). Код для вершины *N* строится путем сцепления (*конкатенации*) в фиксированном порядке последовательности кода *C*(*N*) и последовательностей кодов, связанных со всеми вершинами, являющимися прямыми потомками *N*. В свою очередь, для построения последовательностей кода прямых потомков вершины *N* потребуется найти последовательности кода для их потомков (*потомков второго уровня вершины N*) и т. д. Процесс перевода идет, таким образом, снизу вверх в строго установленном порядке, определяемом структурой дерева.



N - вершина

C(*N*) - код для вершины *N*

Общие принципы генерации кода

В общем случае схемы СУ-перевода могут предусматривать выполнение следующих действий:

- помещение в выходной поток данных машинных кодов или команд ассемблера, представляющих собой результат работы (*выход*) компилятора;

- выдача пользователю сообщений об обнаруженных ошибках и предупреждениях (*которые должны помещаться в выходной поток, отличный от потока, используемого для команд результирующей программы*);

- порождение и выполнение команд, указывающих, что некоторые действия должны быть произведены самим компилятором (*например, операции, выполняемые над данными, размещенными в таблице идентификаторов*).

Общие принципы генерации кода

Возможна модель компилятора, в которой синтаксический анализ входной программы и генерация кода результирующей программы объединены в одну фазу.

Такую модель можно представить в виде компилятора, у которого операции генерации кода совмещены с операциями выполнения синтаксического анализа.

Для описания компиляторов такого типа часто используется термин «*СУ-компиляция*» – синтаксически управляемая компиляция.

Схему СУ-компиляции можно реализовать не для всякого входного языка программирования.

Принцип СУ-перевода применим ко всем входным КС-языкам.

Способы внутреннего представления программ

Возможны различные формы внутреннего представления синтаксических конструкций исходной программы в компиляторе.

На этапе синтаксического анализа часто используется форма, именуемая деревом вывода.

Но формы представления, используемые на этапах синтаксического анализа, оказываются неудобными в работе при генерации и оптимизации объектного кода.

Поэтому перед оптимизацией и непосредственно перед генерацией объектного кода внутреннее представление программы может преобразовываться в одну из соответствующих форм записи.

Способы внутреннего представления программ

Все внутренние представления программы обычно содержат в себе две принципиально различные вещи – операторы и операнды.

Различия между формами внутреннего представления заключаются лишь в том, как операторы и операнды соединяются между собой.

Также опе
друг от д
порядке.

За различение операндов и операторов отвечает разработчик компилятора, который руководствуется семантикой входного языка.

Способы внутреннего представления программ

Известны следующие формы внутреннего представления программ:

- связочные списочные структуры, представляющие синтаксические деревья;
- многоадресный код с явно именуемым результатом (*тетрады*);
- многоадресный код с неявно именуемым результатом (*триады*);
- ассемблерный код или машинные команды;
- обратная

В каждом конкр
выбранная раз
использованием
программы. На
формы, которы
одна в другую.

Существуют три формы записи выражений:
префиксная – операция записывается перед операндами
+ab;
инфиксная – операция записывается между операндами
a+b;
постфиксная – операция записывается после операндов
ab+ .

Синтаксические деревья

Синтаксические деревья – это структура, представляющая собой результат работы синтаксического анализатора. Она отражает синтаксис конструкций входного языка и явно содержит в себе полную взаимосвязь операций.

Недостаток синтаксических деревьев заключается в том, что они представляют собой *сложные связные структуры*, которые не могут быть *тривиальным образом преобразованы* в линейную форму команд результирующей программы.

Синтаксические деревья – это машинно-независимая форма внутреннего представления программы.

Удобны при работе с внутренним представлением программы на тех этапах, когда нет необходимости непосредственно обращаться к командам результирующей программы.

Синтаксические деревья могут быть преобразованы в другие формы внутреннего представления программы, представляющие собой линейные списки, с учетом семантики входного языка. Эти преобразования выполняются на основе принципов СУ-компиляции.

Многоадресный код с явно именуемым результатом

Тетрады представляют собой запись операций в форме из четырех составляющих: *операция*, *два операнда* и *результат* операции.

<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 тетрад представляет собой унарную операцию*), то он может быть опущен или заменен пустым операндом (*в зависимости от принятой формы записи и ее реализации*).

Результат вычисления тетрады *никогда опущен быть не может*, иначе тетрада полностью теряет смысл.

Многоадресный код с явно именуемым результатом

Порядок вычисления тетрад может быть изменен, но только если допустить наличие тетрад, целенаправленно изменяющих этот порядок (*например, тетрады, вызывающие переход на несколько шагов вперед или назад при каком-то условии*).

Достоинства:

- так как тетрады представляют собой линейную последовательность, то для них несложно написать тривиальный алгоритм, который будет преобразовывать последовательность тетрад в последовательность команд результирующей программы (*либо последовательность команд ассемблера*).

- тетрады *не зависят* от архитектуры вычислительной системы, на которую ориентирована результирующая программа.

Недостатки:

- *требуют больше па*

- *не отражают явно е*

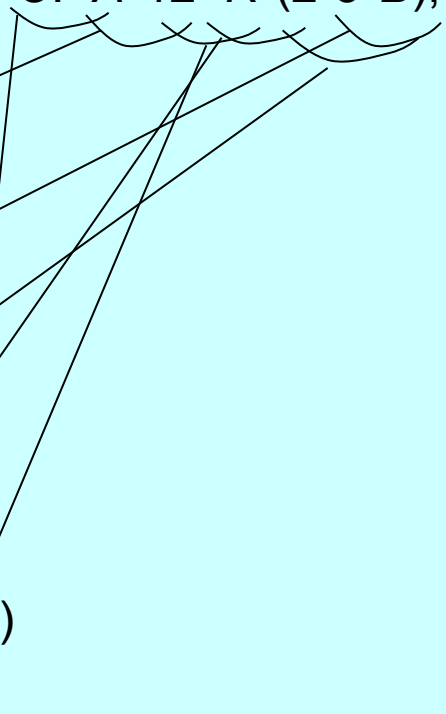
- есть сложности с преобразованием тетрад в машинный код.

они плохо отображаются в команды ассемблера и машинные коды, поскольку в наборах команд большинства современных компьютеров редко встречаются операции с тремя операндами.

Многоадресный код с явно именуемым результатом

Пример:

выражение *C*:=*A**12+*K**(2-3**B*), записанное в виде тетрад, будет иметь вид:



1. * (*A*,12, *D*1)
2. * (3, *B*, *D*2)
3. - (2, *D*2, *D*3)
4. * (*K*, *D*3, *D*4)
5. + (*D*1, *D*4, *D*5)
6. := (*D*5, 0, *C*)

Многоадресный код с неявно именуемым результатом

Триады представляют собой запись операций в форме из трех составляющих: *операция* и *два операнда*.

<операция>(<операнд1>,<операнд2>)

Триады представляют собой линейную последовательность команд

При вычислени
одна за другой

Каждая триада

если в качестве
на другую триад

Если какой-то
представляет
заменен пусты
реализации).

Особенностью триад является то,
что один или оба операнда могут быть
ссылками на другую триаду в том случае,
если в качестве операнда данной триады
выступает результат выполнения другой триады.
Поэтому триады при записи *последовательно нумеруют*
для удобства указания ссылок одних триад на другие
(*в реализации компилятора в качестве ссылок
можно использовать не номера триад,
а непосредственно ссылки в виде указателей
– тогда при изменении нумерации и порядка
следования триад менять ссылки не требуется*).

Результат вы ения триады *нужно сохранять* во временной памяти, так как он может быть затребован последующими триадами.

Многоадресный код с неявно именуемым результатом

Порядок вычисления триад может быть изменен, но только если допустить наличие триад, целенаправленно изменяющих этот порядок (*например, триады, вызывающие переход на несколько шагов вперед или назад при каком-то условии*).

Достоинства:

- триады представляют собой линейную последовательность, а потому для них несложно написать тривиальный алгоритм, который будет преобразовывать последовательность триад в последовательность команд результирующей программы (*либо последовательность команд ассемблера*).
- триады *не зави*
которую ориентиро
- *требуют меньше*
- *явно отражают в*

Необходимость иметь алгоритм, отвечающий за распределение памяти для хранения промежуточных результатов, не является недостатком, так как удобно распределять результаты не только по доступным ячейкам временной памяти, но и по имеющимся регистрам процессора. Это дает определенные преимущества.

Недостатки:

- требуется алгоритм, отвечающий за распределение памяти, необходимой для хранения промежуточных результатов вычисления, так как временные переменные для этой цели не используются.

Многоадресный код с неявно именуемым результатом

Триады ближе к двухадресным машинным командам, чем тетрады, а именно эти команды более всего распространены в наборах команд большинства современных компьютеров.

Пример:

выражение *C*:=*A**12+*K**(2-3**B*), записанное в виде триад, будет иметь вид:

Ассемблерный код или 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.

Использование *команд ассемблера* или *машинных команд* для внутреннего представления программы требует дополнительных структур для отображения взаимосвязи операций.

Ассемблерный код, машинные команды – это *машинно-зависимая форма внутреннего представления программы*.

Машинные команды – это язык, на котором должна быть записана результирующая программа. Поэтому компилятор, так или иначе, должен работать с ними. Кроме того, только обрабатывая машинные команды (*или их представление в форме команд ассемблера*) можно добиться наиболее *эффективной* результирующей программы. Следовательно любой компилятор работает с представлением результирующей программы в форме машинных команд, однако, их обработка происходит, как правило, на завершающих этапах фазы генерации кода.

Обратная польская запись операций

Это постфиксная запись операций, которая предусматривает, что знаки операций записываются после операндов.

По сравнению с обычной (*инфиксной*) записью операций в польской записи операнды следуют в том же порядке, а знаки операций – строго в порядке их выполнения.

Тот факт, что в этой форме записи все операции выполняются в том порядке, в котором они записаны, делает ее чрезвычайно удобной для вычисления выражений на компьютере.

Польская запись *не требует учитывать приоритет операций*, в ней не употребляются скобки, и в этом ее *основное преимущество*.

Она чрезвычайно *эффективна* в тех случаях, когда для вычислений используется стек.

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*).

Обратная польская запись операций

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

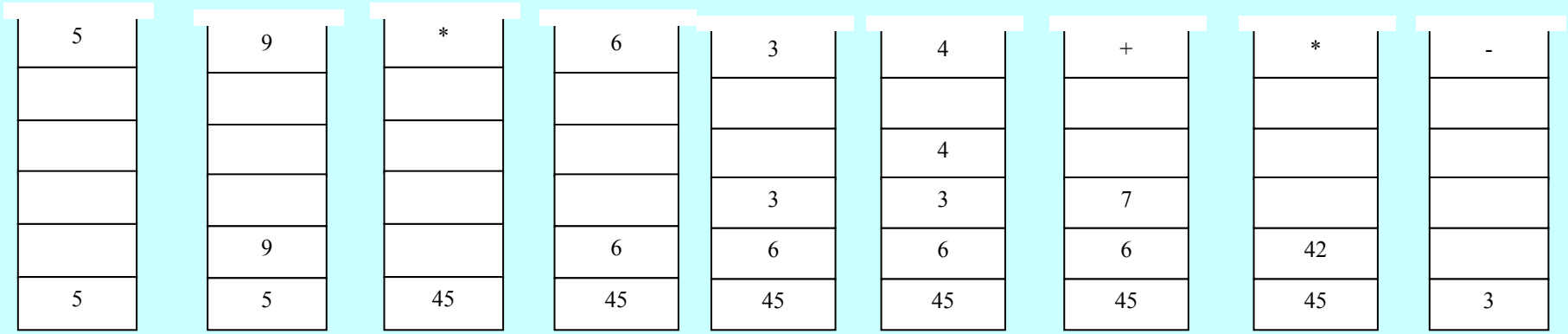
- если встречается операнд, то он помещается в стек (*попадает на верхушку стека*);
- если встречается знак унарной операции (*операции, требующей одного операнда*), то операнд выбирается с верхушки стека, операция выполняется и результат помещается в стек (*попадает на верхушку стека*);
- если встречается знак бинарной операции (*операции, требующей двух операндов*), то два операнда выбираются с верхушки стека, операция выполняется и результат помещается в стек (*попадает на верхушку стека*).

Вычисление выражения заканчивается, когда достигается конец записи выражения.

Результат вычисления при этом всегда находится на верхушке стека.

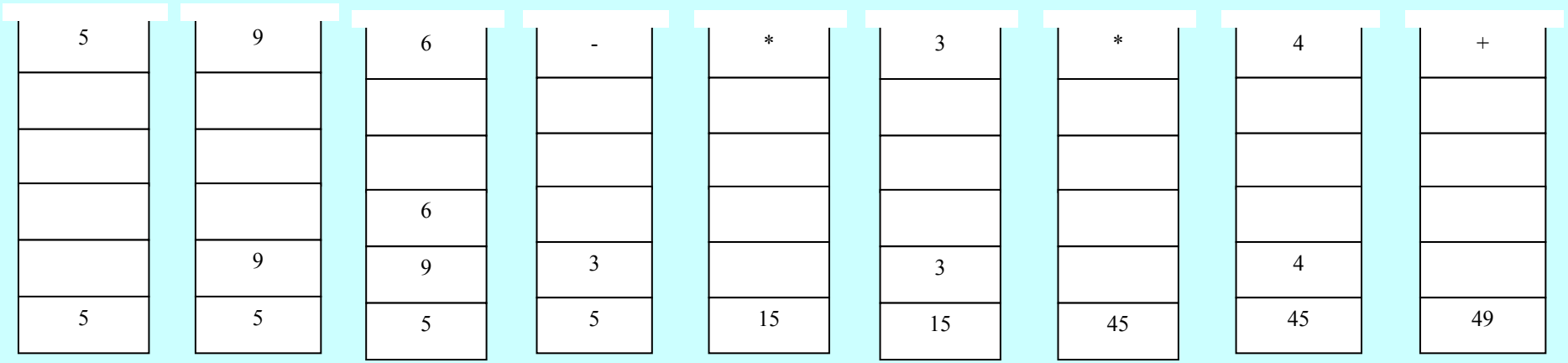
Обратная польская запись операций

Выражение 5*9-6*(3+4)



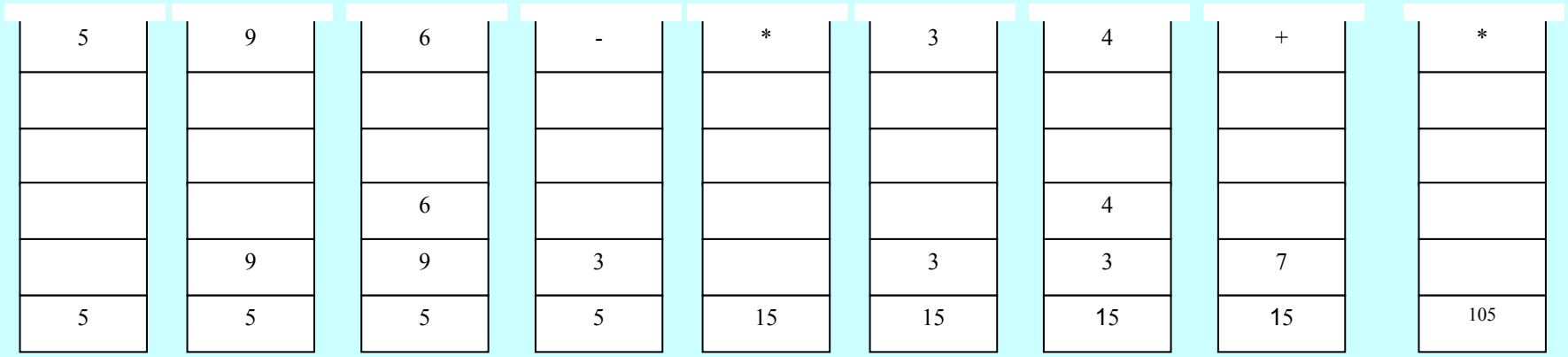
Обратная польская запись операций

Выражение 5*(9-6)*3+4



Обратная польская запись операций

Выражение 5*(9-6)*(3+4)



Оптимизация кода

Большинство современных компиляторов выполняют еще один этап компиляции – оптимизацию результирующей программы, чтобы повысить эффективность насколько это возможно.

Важно зафиксировать два момента:

- *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;

- *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.

Однако практически все компиляторы, так или иначе, выполняют оптимизацию, поскольку их разработчики стремятся завоевать хорошие позиции на рынке средств разработки ПО.

Оптимизация, которая существенно влияет на эффективность результирующей программы, является здесь немаловажным фактором.

Оптимизация кода

Оптимизация программы – это обработка, связанная с переупорядочиванием и изменением операций в компилируемой программе с целью получения более эффективной результирующей объектной программы.

Оптимизация выполняется на этапах 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*);
- *скорость вы*

Далеко не всегда удается выполнить оптимизацию так, чтобы удовлетворить обоим этим критериям. Зачастую сокращение необходимого программе объема данных ведет к уменьшению ее быстродействия, и наоборот. Поэтому для оптимизации обычно выбирается либо один из упомянутых критериев, либо некий комплексный критерий, основанный на них.

Оптимизация кода

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

Если компилятор использует несколько р ных форм внутреннего представления программы, то каждая из может быть подвергнута утреннего представления

Данные преобразования могут зависеть не только от свойств объектного языка, но и от архитектуры вычислительной системы, на которой будет выполняться результирующая программа. При оптимизации может учитываться объем кэш-памяти и методы организации конвейерных операций центрального процессора. В большинстве случаев эти преобразования сильно зависят от реализации компилятора и являются «ноу-хау» производителей компилятора. Именно эти оптимизирующие преобразования позволяют существенно повысить эффективность результирующего кода.

вида оптимизирующи

бразования не зависят
целевой вычислительной
, на которой будет
зультирующая программа.
ованы на выполнении
стных и обоснованных
логических преобразований,
мых над внутренним
лением программы.

Оптимизация кода

Методы преобразования программы зависят от типов синтаксических конструкций исходного языка.

Теоретически разработаны методы оптимизации для многих типовых конструкций языков программирования.

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.

Контрольная работа № 7

Первое выражение записать в виде тетрад и триад.
Второе выражение вычислить в обратной польской записи с использованием стека.

Вариант 1 1. C:=A*(4+K)*3*A 2. 6+8*(10-4)	Вариант 2 1. F:=3*T-Y/3*D 2. 5-6*(3+4)	Вариант 3 1. R:=U-25*Y+(3*A-5) 2. 3*(7-2)*(5+1)
Вариант 4 1. W:=152-R/25+(Y-3*T) 2. (5+6)*(3+4)	Вариант 5 1. E:=9-P*85+Y/7 2. 3*7-2*(5+1)	Вариант 6 1. T:=D*56+T*(5-2*H) 2. (6+8)*10-4
Вариант 7 1. Z:=13*S-(J/45+54) 2. 3*(7-2)*5+1	Вариант 8 1.O:=G-(T-6)/(L*5+F) 2. (6+8)*(10-4)	Вариант 9 1. W:=H-58*Y-69*Y 2. (5-6)*3+4