Глава 4 Теория перевода

4.1 Трансляторы и компиляторы

4.1.1 Общая схема работы

Сначала вспомним необходимые понятия.

Транслятор – это программа, которая переводит входной текст на исходном языке в эквивалентный ему выходной текст на результирующем (выходном) языке.

Для создания транслятора необходимо прежде всего выбрать входной и выходной языки. Транслятор может, например, переводить художественный текст с английского языка на русский. Или текст программы с одного языка программирования на другой. Результатом работы транслятора будет результирующая программа, но только в том случае, если текст исходной программы является правильным с точки зрения синтаксиса и семантики входного языка.

Компилятор представляет собой транслятор, осуществляющий перевод исходной программы в эквивалентную объектную программу на языке машинных команд или на языке ассемблера.

Всякий компилятор является транслятором, но не наоборот. Компиляторы являются наиболее распространённым, хотя и не единственным, видом трансляторов.

Интерпретатор – это программа, которая воспринимает программу на исходном языке и пошагово выполняет её. В отличие от трансляторов интерпретаторы не порождают результирующей программы, но так же анализируют текст исходной программы; результатом их работы является результат, заданный смыслом исходной программы, если она верна, и сообщение об ошибке в противном случае.

Компиляторы более просты в реализации и более эффективны, чем интерпретаторы. Преимущество интерпретаторов состоит в том, что исходная программа привязана только к семантике языка программирования, а откомпилированный код – к архитектуре вычислительной системы.

Существует огромное множество языков программирования, каждый из которых имеет соответствующий ему компилятор. Но при всём этом многообразии в процессе компиляции можно выделить общие моменты. Независимо от специфики конкретного языка программирования исходная программа представляет собой цепочку знаков, построенную в соответствии с правилами задающей язык грамматики. В процессе преобразования её в цепочку битов – объектный код – можно выделить составные части, общие для всех компиляторов.

В целом процесс компиляции состоит из этапов анализа и синтеза.

На этапе анализа выполняется распознавание текста исходной программы и заполнение таблиц идентификаторов. Результатом его работы является некоторое внутреннее представление программы, понятное компилятору.

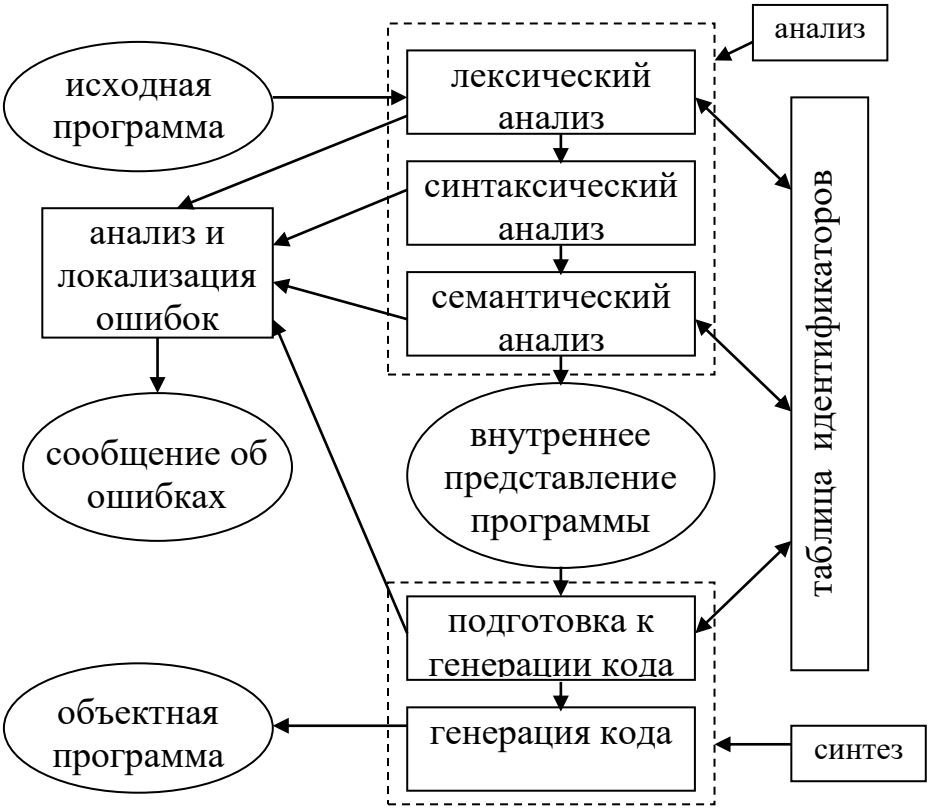
На этапе синтеза по внутреннему представлению программы и информации, содержащейся в таблицах идентификаторов, порождается текст результирующей программы. Результатом этого этапа является объектный код.

Кроме того, в составе компилятора присутствует часть, выполняющая анализ и исправление ошибок, которая при наличии в тексте программы ошибки должна локализовать её и наиболее полно информировать о ней пользователя.

Этапы анализа и синтеза подразделяются на более мелкие, называемые *фазами компиляции*, такие, как:

1) лексический анализ;
2) работа с таблицами;
3) синтаксический анализ, или разбор;
4) генерация кода, или трансляция в промежуточный код;
5) оптимизация кода;
6) генерация объектного кода.

Схема работы компилятора представлена на рисунке.



Состав фаз приведен в общем виде, конкретная их реализация и взаимодействие могут различаться в зависимости от версии компилятора, но так или иначе они присутствуют в любом компиляторе. Главный принцип – никакая входная цепочка не должна нарушать работу компилятора.

С точки зрения теории формальных языков компилятор выполняет две основные функции.

1) Он является *распознавателем* для языка исходной программы, т.к. он должен получить на вход цепочку символов, представляющую исходную программу, проверить её принадлежность языку и определить правила, согласно которым она была получена. Генератором цепочек исходного языка является разработчик программы.

2) Компилятор является *генератором* для языка результирующей программы; он должен на выходе построить цепочку языка по определённым правилам. Распознавателем её является вычислительная система, под которую создается результирующая программа.

Рассмотрим кратко основные функции некоторых фаз компиляции.

Лексический анализ – это часть компилятора, которая читает цепочку символов программы на исходном языке и строит из них слова (лексемы) исходного языка. Выходная информация передается для дальнейшей обработки компилятором на этапе синтаксического разбора. С теоретической точки зрения эта часть не является обязательной, хотя, как правило, присутствует во всех компиляторах.

Синтаксический разбор – основная часть компилятора на этапе анализа. Она выполняет выделение синтаксических конструкций в тексте исходной программы и проверяет его синтаксическую правильность. Синтаксический анализатор выполняет главную роль распознавателя текста входного языка программирования.

Семантический анализ – это часть компилятора, проверяющая правильность текста исходной программы с точки зрения семантики входного языка. Кроме того, семантический анализатор должен выполнять преобразования текста, требуемые семантикой входного языка. В различных реализациях компиляторов эта фаза может частично входить в фазу синтаксического разбора и в фазу подготовки к генерации кода.

Подготовка к генерации кода – на этой фазе компилятором выполняются предварительные действия, непосредственно связанные с синтезом текста результирующей программы, но ещё не ведущие к порождению текста на выходном языке.

Генерация кода – фаза порождения команд, составляющих предложения выходного языка. Это основная фаза на этапе синтеза результирующей программы. Она обычно включает в себя и *оптимизацию кода*, которую иногда выделяют в отдельную фазу компиляции.

Работа с таблицами. Таблицы идентификаторов (таблицы символов) – специальным образом организованные наборы данных, служащие для хранения информации об элементах исходной программы, которые затем используются для порождения текста результирующей программы. Такими элементами исходной программы могут быть переменные, константы, функции и т.п.

Реальные компиляторы выполняют трансляцию текста исходной программы за несколько проходов.

Проход – это процесс последовательного чтения компилятором данных из внешней памяти, их обработка и помещение результатов во внешнюю память. Обычно проход включает в себя одну или несколько фаз компиляции. Результатом промежуточных проходов является внутреннее представление программы, результатом последнего прохода – результирующая объектная программа.

При выполнении каждого прохода компилятору доступна информация, полученная в результате всех предыдущих проходов. В основном используется информация от предыдущего прохода. Информация, полученная при выполнении проходов, хранится в оперативной памяти, которая освобождается при завершении процесса компиляции, либо во временных файлах на диске, которые также удаляются после окончания работы. Эта информация недоступна пользователю, который может даже не знать, сколько проходов выполняет компилятор.

Разработчики стремятся сократить количество проходов, выполняемых компилятором – это позволяет повысить эффективность работы и сократить объёмы занимаемой памяти. Количество необходимых проходов определяется прежде всего грамматикой и семантическими правилами исходного языка. Идеальным является однопроходный компилятор, но он возможен только для очень простых языков. Реальные компиляторы выполняют, как правило, от двух до пяти проходов. Например, первый проход – лексический анализ, второй – синтаксический разбор и семантический анализ, третий – генерация и оптимизация кода.

4.1.2 Контрольные вопросы

1. Чем отличаются трансляторы и компиляторы? Правда ли, что транслятор – это частный случай компилятора?
2. Какие общие фазы можно выделить в процессе компиляции?
3. Какие действия происходят на этапе анализа в процессе компиляции?
4. На каких этапах процесса компиляции выполняется работа с таблицами?
5. На каких этапах процесса компиляции могут быть обнаружены ошибки в тексте программы?
6. Что такое «проход» в процессе компиляции?

4.2 Теория перевода

В общем случае *перевод* (или трансляция) – это некоторое отношение между цепочками, или некоторое множество пар цепочек.

Компилятор определяет перевод, образуемый парами вида (исходная программа, объектная программа). Если в общем рассматривать компилятор как состоящий из трёх фаз – лексический анализ, синтаксический анализ и генерация кода – то каждая из них сама является переводом.

Проблема задания бесконечного перевода конечными средствами аналогична проблеме задания бесконечного языка.

Существует два фундаментальных метода определения перевода. Один из них – схема синтаксически управляемого перевода, которая представляет собой грамматику, снабжённую механизмом, обеспечивающим выход для каждой порождаемой цепочки. В другом методе основную роль играет преобразователь, т.е. распознаватель с выходом, который на каждом такте может выдавать цепочку выходных символов ограниченной длины.

Пусть V_1 – входной алфавит, V_2 – выходной алфавит. *Переводом* с языка $L_1 \subseteq V_1^*$ на язык $L_2 \subseteq V_2^*$ назовём отношение T из V_1^* в V_2^*, для которого L_1 – область определения, а L_2 – множество значений. Если $(x,y) \in T$, то y называется *выходом* для цепочки x.

Замечание. В общем случае в переводе T для данной входной цепочки может быть более одной выходной цепочки. Но перевод, предназначенный для языка программирования, должен являться функцией, т.е. для каждого входа должно быть не более одного выхода.

Устройство, которое по данной входной цепочке x вычисляет такую выходную цепочку y, что $(x,y) \in T$, является транслятором, реализующим перевод T.

Определение перевода должно обладать хорошими свойствами, в частности:

– в нём легко разобраться, т.е. установить, какие пары цепочек принадлежат переводу;
– прямо по определению перевода можно механически построить эффективный транслятор, реализующий этот перевод.

Желательные качества трансляторов:

– эффективность трансляции – линейная зависимость времени обработки входной цепочки от её длины;
– небольшой объём;
– корректность.

4.2.1 Схема синтаксически управляемого перевода

Схема синтаксически управляемого перевода представляет собой грамматику, к каждому правилу которой присоединяется элемент перевода. Когда правило участвует в выводе входной цепочки, с помощью элемента перевода вычисляется часть выходной цепочки, соответствующая части входной цепочки, порождённой этим правилом.

Схемой синтаксически управляемого перевода (трансляции), или СУ-схемой, называется пятёрка $T = (VN, VT_1, VT_2, R, S)$, где

VN – конечное множество нетерминальных символов;

VT_1 – конечный входной алфавит; VT_2 – конечный выходной алфавит;

S – целевой символ;

R – конечное множество правил вида $A \to \alpha,\beta$, где $\alpha \in (VN \cup VT_1)^*$, $\beta \in (VN \cup VT_2)^*$, и вхождения нетерминалов в цепочку β образуют перестановку вхождений нетерминалов в цепочку α.

Если $A \to \alpha,\beta$ – правило, то вхождению каждого нетерминала в цепочку α соответствует некоторое вхождение того же нетерминала в β.

Определим *выводимую пару цепочек схемы T*:

(1) (S,S) – выводимая пара, в которой символы S соответствуют друг другу.

(2) если $(\alpha A\beta, \alpha' A\beta')$ – выводимая пара, в которой два выделенных вхождения нетерминала A соответствуют друг другу, и $A \to \gamma,\gamma'$ – правило из R, то $(\alpha\gamma\beta, \alpha'\gamma'\beta')$ – выводимая пара.

Схема трансляции T определяет некоторый *перевод* $\tau(T)$. Транслятор, реализующий этот перевод, может работать так:

По данной входной цепочке $\boldsymbol{x}$ с помощью правил схемы трансляции T транслятор находит (если это возможно) некоторый вывод цепочки $\boldsymbol{x}$ из S. Пусть этот вывод $S = \alpha_0 \Rightarrow \alpha_1 \Rightarrow \alpha_2 \Rightarrow \ldots \Rightarrow \alpha_n = x$. Затем транслятор строит вывод $(\alpha_0,\beta_0) \Rightarrow (\alpha_1,\beta_1) \Rightarrow (\alpha_2,\beta_2) \Rightarrow \ldots \Rightarrow (\alpha_n,\beta_n)$, состоящий из выводимых пар цепочек, для которого $(\alpha_0,\beta_0) = (S,S)$, $(\alpha_n,\beta_n) = (x,y)$ и каждое β_i получается из β_{i-1} с помощью элемента перевода, соответствующего правилу, применённому при переходе от α_{i-1} к α_i. Цепочка $\boldsymbol{y}$ служит выходом для цепочки $\boldsymbol{x}$.

Переводом $\tau(T)$, *определяемым схемой T*, называется множество пар $\{(x,y) | (S,S) \Rightarrow^* (x,y), x \in VT_1^*, y \in VT_2^*\}$.

Если T – СУ-схема, то $\tau(T)$ называется *синтаксически управляемым переводом* (*СУ-переводом*).

Грамматика $G(VN,VT_1,P,S)$, где $P = \{A \to \alpha \mid (A \to \alpha,\beta) \in R\}$, называется *входной грамматикой* СУ-схемы T. Грамматика $G'(VN,VT_2,P',S)$, где $P' = \{A \to \beta \mid (A \to \alpha,\beta) \in R\}$, называется *выходной грамматикой* СУ-схемы T.

Пример. Рассмотрим схему, определяющую перевод $\{(x,x^R)|x\in\{0,1\}^*\}$:

правило	элемент перевода
(1) $S \to 0S$	S = S0
(2) $S \to 1S$	S = S1
(3) $S \to \lambda$	S = λ

В переводе, определяемом этой схемой, пару вход-выход можно получить, порождая последовательность выводимых пар цепочек (α,β), где α – входная выводимая цепочка, а β – выходная выводимая цепочка.

Рассмотрим перевод входной цепочки 001. $(S,S) \Rightarrow^{(1)} (0S,S0) \Rightarrow^{(1)} (00S,S00) \Rightarrow^{(2)} (001S,S100) \Rightarrow^{(3)} (001,100)$.

СУ-схема T называется *простой*, если для любого правила $(A \to \alpha,\beta)\in R$ соответствующие друг другу вхождения нетерминалов встречаются в α и β в одном и том же порядке.

Перевод, определяемый простой СУ-схемой, называется *простым СУ-переводом* (см. пример выше).

Простые СУ-переводы образуют важный класс, т.к. для них легко можно построить транслятор, представляющий собой МП-преобразователь.

Пример. Простая СУ-схема T= ({S,P,A}, {a,+,*,(,)}, {a,+,*,(,)}, R, S) отображает арифметические выражения в форму, не содержащую излишних скобок.

R: (1) $S \to (S), S$ (4) $P \to (P), P$ (7) $A \to (S+S), (S+S)$
(2) $S \to S+S, S+S$ (5) $P \to A*A, A*A$ (8) $A \to P, P$
(3) $S \to P, P$ (6) $P \to a, a$

Для выражения ((a+(a*a))*a) эта СУ-схема даёт перевод (a+a*a)*a:

$(S,S) \Rightarrow^{(1)} ((S),S) \Rightarrow^{(3)} ((P),P) \Rightarrow^{(5)} ((A*A),A*A) \Rightarrow^{(7)} (((S+S)*A),(S+S)*A) \Rightarrow^{(3)} (((P+S)*A),(P+S)*A) \Rightarrow^{(6)} (((a+S)*A),(a+S)*A) \Rightarrow^{(3)} (((a+P)*A),(a+P)*A) \Rightarrow^{(4)} (((a+(P))*A),(a+P)*A) \Rightarrow^{(5)} (((a+(A*A))*A),(a+A*A)*A) \Rightarrow^{(8)} (((a+(P*A))*A),(a+P*A)*A) \Rightarrow^{(6)} (((a+(a*A))*A),(a+a*A)*A) \Rightarrow^{(8)} (((a+(a*P))*A),(a+a*P)*A) \Rightarrow^{(6)} (((a+(a*a))*A),(a+a*a)*A) \Rightarrow^{(8)} (((a+(a*a))*P),(a+a*a)*P) \Rightarrow^{(6)} (((a+(a*a))*a),(a+a*a)*a)$.

4.2.2 Преобразователи

Простейшим транслятором является конечный преобразователь.

Конечным преобразователем называется шестёрка $M(Q,VT_1,VT_2,\delta,q_0,F)$. Отличие от конечного автомата состоит в том, что добавляется ещё выходной алфавит VT_2, а функция переходов действует в множество конечных подмножеств множества $Q\times VT_2^*$.

Конфигурацией преобразователя M назовем тройку (q,x,y), где кроме текущего состояния q и оставшейся непрочитанной части входной цепочки x появляется ещё y – часть выходной цепочки, выданная вплоть до

настоящего момента.

Бинарное отношение, соответствующее одному такту работы конечного преобразователя M, определим следующим образом: $(q,ax,y) \vdash (q',x,yz)$, если $\delta(q,a)$ содержит (q', z), где $q, q' \in Q$, $a \in VT_1$, $x \in VT_1^*$, $y,z \in VT_2^*$.

Цепочку y назовем *выходом* для цепочки x, если $(q_0, x, \lambda) \vdash^* (q, \lambda, y)$ для некоторого заключительного состояния q и $x \in VT_1^*$, $y \in VT_2^*$.

Переводом, определяемым преобразователем M – ***τ(M)***, называется множество $\tau(M) = \{(x,y) \mid x \in VT_1^*, y \in VT_2^* \text{ и } \exists q \in F \mid (q_0, x, \lambda) \vdash^* (q, \lambda, y)\}$.

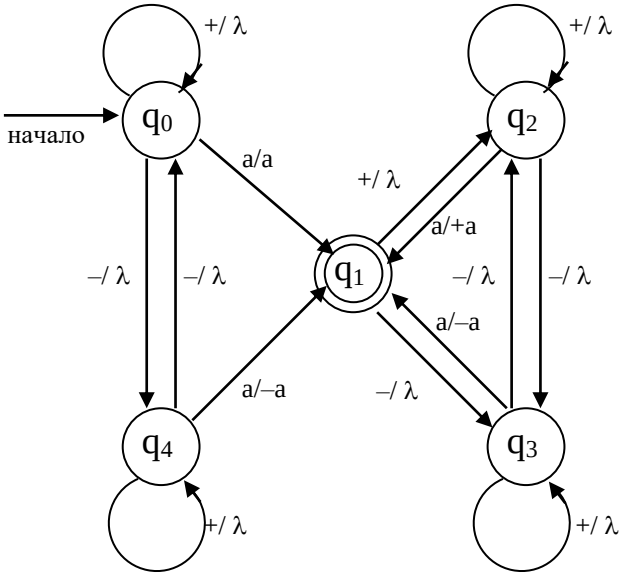
Перевод, определяемый конечным преобразователем, будем называть *конечным преобразованием* или *регулярным переводом*.

Замечание. Для того, чтобы цепочку y можно было считать переводом входной цепочки x, цепочка x должна перевести преобразователь из начального состояния в заключительное.

Пример. Рассмотрим конечный преобразователь, который распознает арифметические выражения, задаваемые правилами:

$S \to a+S \mid a-S \mid +S \mid -S \mid a$ и устраняет из них избыточные унарные операции. Например, выражение ”–a+–a–+–a” он переведет в ”–a–a+a”.

$(\{q_0,q_1,q_2,q_3,q_4\},\{a,+,-\},\{a,+,-\}, \delta, q_0, \{q_1\})$, где δ – граф переходов.



Метка x/y на дуге из q_i в q_j означает, что $\delta(q_i, x)$ содержит (q_j, y).

Состояния q_0 и q_4, q_2 и q_3 считают "–".

Выход из состояния q_1 по символу ′a′ невозможен, поскольку цепочка, в которой два символа ′a′ стоят подряд друг за другом, не выводима в грамматике, задающей исходный язык.

Разбор цепочки происходит поэтапно и выглядит так: $(q_0,–a+–a–+–a, \lambda) \vdash (q_4,a+–a–+–a, \lambda) \vdash (q_1,+–a–+–a, –a) \vdash (q_2,–a–+–a, –a) \vdash (q_3,a–+–a, –a) \vdash (q_1,–+–a, –a–a) \vdash (q_3,+–a, –a–a) \vdash (q_2,–a, –a–a) \vdash (q_1, \lambda, –a–a+a)$.

Преобразователем с МП называется восьмёрка $P = (Q,V,Z,V',\delta,q_0,z_0,F)$, где единственным изменением относительно обычного МПА является добавление выходного алфавита V′, а функция переходов δ действует из

$Q\times(V\cup\{\lambda\})\times Z$ в $Q\times Z^*\times V'^*$. Все остальные символы имеют тот же смысл, что и в определении МП-автомата.

Конфигурацию преобразователя P определим как четвёрку (q,x,α,y), где q,x,α – те же, что у МПА, а y – выходная цепочка, выданная вплоть до настоящего момента.

Если $\delta(q,a,z)$ содержит (r,α,t), то будем писать $(q,ax,z\gamma,y) \vdash (r,x,\alpha\gamma,yt)$ для $\forall x\in V^*$, $\alpha,\gamma\in Z^*$ и $y,t\in V'^*$, $a\in V\cup\{\lambda\}$, $z\in Z$.

Цепочку y назовем *выходом* для x, если $(q_0, x, Z_0, \lambda)\vdash^* (q, \lambda, \alpha, y)$ для некоторых $q\in F$ и $\alpha\in Z^*$.

Переводом, определяемым преобразователем P, назовём множество $\tau(P)=\{(x,y) \mid \exists q\in F \text{ и } \alpha\in Z^* \mid (q_0, x, Z_0, \lambda)\vdash^* (q, \lambda, \alpha, y),\ x\in V^*, y\in V'^*\}$.

Рассмотрим пример одного и того же перевода, задаваемого СУ-схемой и преобразователем с магазинной памятью.

Пример.

Пусть требуется выполнить перевод $\tau= \{(x,y)| x = 0^k1^k, y= a^{2k}b^{2k} | k>0\}$.

Схема синтаксически управляемого перевода $T = (\{S\},\{0,1\},\{a,b\},R,S)$, где R: $S \to 0S1,aaSbb$; $S \to 01,aabb$. Здесь процесс порождения исходной цепочки и результата перевода происходит одновременно. Генерируются ‘0’ в начале исходной цепочки, ‘1’ в её конце и одновременно с этим ‘aa’ в начале и ‘bb’ в конце результирующей цепочки.

Преобразователь с магазинной памятью

$P =(\{q_0,q_1,q_2\},\{0,1\},\{0,z_0\},\{a,b\},\delta,q_0,z_0,\{q_2\})$,

$\delta(q_0,0,z_0)=\{(q_0,0z_0,aa)\}$ $\delta(q_0,1,0)=\{(q_1,\lambda,bb)\}$ $\delta(q_1,\lambda,z_0)=\{(q_2,\lambda,\lambda)\}$

$\delta(q_0,0,0)=\{(q_0,00,aa)\}$ $\delta(q_1,1,0)=\{(q_1,\lambda,bb)\}$

Переведём цепочку ‘0011’. $(q_0,0011,z_0,\lambda) \vdash (q_0,011,0z_0,aa) \vdash (q_0,11,00z_0, aaaa) \vdash (q_1,1,0z_0,aaaabb) \vdash (q_1,\lambda,z_0,aaaabbbb) \vdash (q_0,\lambda,\lambda,aaaabbbb)$. Здесь идёт процесс распознавания исходной цепочки и параллельно с этим порождается итоговая цепочка – на каждый прочитанный ‘0’ порождается ‘aa’, на каждую прочитанную ‘1’ генерируется ‘bb’. При этом все нули сначала копируются в магазин, а затем удаляются оттуда при прочтении единиц (см. пример из 3.1.1). Это обеспечивает равенство их количества, а смена состояний при переходе от считывания ‘0’ к считыванию ‘1’ гарантирует их заданный порядок.

4.2.3 Контрольные вопросы

1. В чём состоит проблема задания перевода?
2. На чём основаны основные методы перевода?
3. Что представляет из себя схема синтаксически управляемого перевода? На чём она основана?
4. Дана СУ-схема T = ({S, A}, {0, 1}, {a, b}, R, S), где R состоит из правил: (1) $S \to 0AS, SAa$; (2) $A \to 0SA, ASa$; (3) $S \to 1, b$;

(4) A → 1, b. Выполнить перевод цепочек: а) x = ’011’, б) y = ’00111’.

5. В чём состоит принципиальное отличие перевода, определяемого схемой трансляции, от перевода, задаваемого преобразователем?
6. Дан МП-преобразователь: P = ({q},{a,+,*},{+,*,E},{a,+,*},δ, q,E,{q}), где δ определяется равенствами:

δ(q, *, E) = {(q, EE*, λ)} δ(q, a, E) = {(q, λ, a)} δ(q, λ, *) = {(q, λ, *)}
δ(q, λ, +) = {(q, λ, +)} δ(q, +, E) = {(q, EE+, λ)}

Выполнить перевод цепочек: а) ’++aaa’; б) ’*+aaa’; в) ’*+aaa’; г) ’+a*aa’.

4.3 Этапы компиляции

4.3.1 Работа с таблицами идентификаторов

Для выполнения семантического анализа и генерации кода необходимо знание характеристик переменных, констант, функций и других элементов, встречающихся в исходном тексте программы. Выделение идентификаторов происходит на фазе лексического анализа. Их характеристики определяются на фазе синтаксического анализа, семантического анализа и подготовки к генерации кода. Конкретный состав этих характеристик зависит от семантики входного языка.

Поскольку компилятор должен иметь возможность работать с идентификаторами и их характеристиками на различных фазах процесса компиляции, информация обо всех идентификаторах сохраняется в таблицах. При этом может быть одна или же несколько различных таблиц – например, своя таблица для каждого программного модуля. Количество полей соответствует количеству различных идентификаторов, а состав информации зависит от семантики входного языка и типа элементов. Например, для переменной это может быть имя и тип данных и связанная с ней область памяти. Для функции – имя, количество и типы формальных аргументов, тип возвращаемого результата и адрес кода функции.

Не вся хранимая в таблицах информация заполняется компилятором сразу. Например, имена переменных могут быть выделены на фазе лексического анализа, типы данных – на фазе синтаксического разбора, а область памяти связывается с переменной только на фазе подготовки к генерации кода.

Независимо от варианта реализации компилятора принцип его работы с таблицей идентификаторов остается неизменным – обращение к ней происходит многократно для поиска информации и записи новых данных. Способы организации таких таблиц могут быть различными, но главной целью является уменьшение времени, необходимого на обработку таблиц. Т.о., таблица должна обеспечивать:

- быстрое добавление новых элементов и сведений о них;
- быстрый поиск элементов.

Простейшим способом организации является неупорядоченная

таблица. Пополнение такой таблицы элементарно и представляет собой добавление элементов в порядке их поступления, но для поиска нужного элемента в такой неупорядоченной таблице размера N компилятору в среднем придется выполнять N/2 сравнений. Поскольку поиск является значительно более часто выполняемой операцией, нежели добавление элемента (как минимум, при каждой попытке добавления необходимо выполнить проверку на предмет наличия такого элемента в таблице, т.е. осуществить поиск), быстродействие этой операции является наиболее важным. Если элементы таблицы упорядочить в лексикографическом порядке по именам идентификаторов, то оказывается возможным ускорить процесс поиска, применив какой-либо из специальных методов, например, бинарный поиск. В таком случае время, требуемое для поиска элемента в таблице, уменьшится до $T=O(\log_2 N)$.

Другие способы организации таблиц – это их построение по методу бинарного дерева, использование хэш-адресации, построение по методу цепочек и т.д., а также различные комбинированные методы. Организация эффективного поиска позволяет существенно ускорить процесс компиляции.

4.3.2 Лексический анализ

Лексический анализ является первым этапом процесса компиляции. На этом этапе входная цепочка символов считывается и превращается в цепочку лексем.

Лексема – это структурная единица языка, которая состоит из терминальных символов языка и не содержит в своём составе других структурных единиц языка. Она рассматривается как единый синтаксический объект. С лексемой связывают пару вида (тип лексемы, некоторые данные). Первой компонентой такой пары является синтаксическая категория, например, «константа», «идентификатор», «ключевое слов», а второй – указатель на адрес ячейки, хранящей информацию об этой конкретной лексеме.

Например, для естественных языков лексемами являются слова, для языков программирования – ключевые слова языка, идентификаторы, знаки операций и т.п.

Лексический анализатор (ЛА) – это транслятор, входом которого является исходный текст программы, а выходом – полученная в результате последовательность лексем, которая затем поступает на синтаксический анализатор для разбора.

Все функции ЛА могут выполняться на этапе синтаксического разбора. Лексический анализ важен для процесса компиляции и его принято выделять в отдельную фазу по следующим причинам:

– на стадии лексического анализа из программы удаляется вся ненужная

информация (например, пробелы, комментарии), поэтому применение ЛА упрощает работу с текстом программы на следующем этапе; кроме того, представление программы становится короче, что также ускоряет последующую работу с ней;

– техника лексического анализа значительно более проста, чем синтаксического разбора;

– при изменении версии языка достаточно перестроить достаточно простой ЛА, не затрагивая сложного по конструкции синтаксического анализатора.

Пример. Рассмотрим лексический анализ на примере оператора присваивания: ***cost:=(price+tax)*0.98.***

На этапе лексического анализа будет обнаружено, что ***cost, price, tax*** – лексемы типа идентификатор, ***0.98*** – лексема типа константа. Знаки операций также являются лексемами. Пусть все константы и идентификаторы требуется отобразить в лексемы типа <ид>. Предполагается, что вторая компонента лексемы представляет собой указатель элемента таблицы, содержащего фактическое имя идентификатора вместе с другими собранными в нем данными. Первая компонента используется синтаксическим анализатором для разбора. Вторая – на этапе генерации кода. В нашем случае в результате выходная последовательность будет иметь вид:

$<ид>_1:=(<ид>_2+<ид>_3)* <ид>_4$

Здесь вторая компонента лексемы показана в виде нижнего индекса. Символы – знаки операций – являются лексемами, тип которых представлен ими самими; они не имеют связанных с ними данных и, следовательно, не имеют указателей.

Заметим, что ЛА должен не только обнаружить лексемы и выдать их, но и проверить, есть ли такие лексемы в таблицах и в случае их отсутствия добавить их туда.

Лексический анализ провести легко, если лексемы, состоящие из нескольких символов, изолированы с помощью знаков, которые сами являются лексемами (как в примере). Но в общем случае лексический анализ может оказаться не таким легким. Например, рассмотрим следующие правильные операторы Фортрана:

(1) DO 10 I=1.15 (2) DO 10 I=1,15

Поскольку пробелы в этом языке игнорируются, то в операторе (1) цепочка DO 10 I представляет собой переменную, а цепочка 1.15 – константу. Во втором операторе DO – ключевое слово, 10 – константа, I – переменная, 1 и 15 – константы. В этом случае для правильного определения типа лексемы лексическому анализатору нужно заглянуть вперёд (до точки или запятой).

Определим два крайних подхода к лексическому анализу, которые

положены в основу большинства известных способов анализа.

Лексический анализатор работает *прямо*, если для данного входного текста и позиции указателя в нём он определяет лексему, расположенную непосредственно справа от указателя, и сдвигает сам указатель вправо от части текста, образующей эту лексему.

Лексический анализатор работает *непрямо*, если для данного входного текста, заданного типа лексемы и позиции указателя в нём он определяет, образуют ли знаки справа от указателя лексему заданного типа, и если да, то сдвигает указатель вправо от части текста, образующей эту лексему. Поскольку в отличие от прямого варианта лексическому анализатору на входе требуется ещё тип ожидаемой лексемы, требуется взаимодействие непрямого ЛА с синтаксическим распознавателем.

Если вернуться к нашему примеру – выражению (2) DO 10 I=1,15, то непрямой анализатор ответит «да» на вопрос о лексеме типа DO или о лексеме типа <идентификатор>. При этом в первом случае указатель передвинется на 2 символа вправо, а во втором – на 5. Прямой ЛА обследует текст вплоть до запятой и, сделав вывод о лексеме типа DO, передвинет указатель на 2 символа, хотя просмотрено было значительно большее количество символов.

Большую часть того, что происходит на этапе лексического анализа, можно моделировать с помощью конечных преобразователей, работающих последовательно или параллельно. ЛА может состоять из ряда последовательно соединенных конечных преобразователей, первый из которых может устранять все несущественные пробелы, второй – ликвидировать комментарии, третий – искать константы и т.п.

Конечный преобразователь считывает входные данные, не производя выхода, пока не обнаружит лексемы заданного типа. Тогда он сигнализирует от появлении этой лексемы и на выходе выдаёт цепочку символов, её образующих. Сигналом обычно служит само заключительное состояние автомата.

При непрямом лексическом анализе можно получить на выходе цепочку лексем, а позднее окажется, что на самом деле такие лексемы отсутствуют; в таком случае алгоритм синтаксического разбора осуществит возврат к цепочке, которая оказалась разобрана неправильно. Такой возврат повторяется до тех пор, пока не будет выполнен правильный разбор. При этом нужно заботиться о том, чтобы не выполнялось ошибочных операций с таблицами. Обычно идентификатор помещают в таблицу, только убедившись, что он действительно встречается. В противном случае потребуется дополнительно обеспечивать механизм удаления из таблиц ненужных элементов.

Задачи лексического анализа имеют широкое распространение и допускают чётко формализованное решение на основе техники

моделирования КА. Итак, проблема непрямого лексического анализа является проблемой построения ДКА по заданному регулярному выражению и его программной реализации.

Моделью прямого ЛА служит множество параллельно работающих КА, или один конечный преобразователь, моделирующий много конечных автоматов и выдающий сигнал о том, который из них распознал цепочку. Для моделирования таких параллельно работающих НКА строится объединённый ДКА, который выдаёт имя лексемы и локализует её вхождение.

Существует несколько подходов к программному моделированию КА и преобразователей. Экономный с точки зрения расходования памяти, но медленный способ состоит в том, что кодируется функция переходов. Поскольку лексический анализ составляет значительную долю процесса трансляции, медленная работа – существенный недостаток, и этот метод является малоприменимым.

Другой подход заключается в том, чтобы для каждого состояния завести свой кусок программы. Тогда в функции программы входит определение очередной буквы, выдача необходимого выхода и переход к месту программы, соответствующему очередному состоянию. Здесь есть ряд возможностей. Например, если из некоторого состояния различные буквы вызывают разные переходы, то удобно использовать таблицы. Но во многих состояниях только небольшое количество символов приводит к каким-то необычным состояниям, а все остальные буквы тогда можно не различать. В таком случае разумным вариантом является осуществление поиска для этих «особенных» букв.

4.3.3 Синтаксический анализ

Синтаксический анализ представляет собой второй этап процесса компиляции. На этом этапе исследуется таблица лексем и устанавливается, удовлетворяет ли она структурным условиям, сформулированным в определении синтаксиса языка. Основу синтаксического анализатора составляет распознаватель текста входной программы на базе грамматики входного языка.

Как правило, синтаксические конструкции языков программирования могут быть описаны с помощью КС-грамматик. Методы анализа цепочек языков, заданных КС-грамматиками, были рассмотрены ранее – это разбор с возвратами, табличные алгоритмы и специальные методы, пригодные для некоторого ограниченного класса грамматик.

В роли распознавателя чаще всего выступает МП-автомат. Но в задачи синтаксического анализатора входит не только выделение синтаксических конструкций во входном тексте программы, установление их типа и проверка правильности, но и представление их в виде, удобном для дальнейшей генерации текста результирующей программы. В таком

варианте синтаксический анализатор является не просто МПА, но преобразователем с магазинной памятью.

Результатом работы такого преобразователя является последовательность правил грамматики, применённых для построения входной цепочки, которую достаточно знать для полного представления о типе и структуре найденной и разобранной синтаксической конструкции языка. По этой последовательности можно построить цепочку или дерево вывода. Форма представления этой информации может быть различной, и эта форма называется *внутренним представлением программы*.

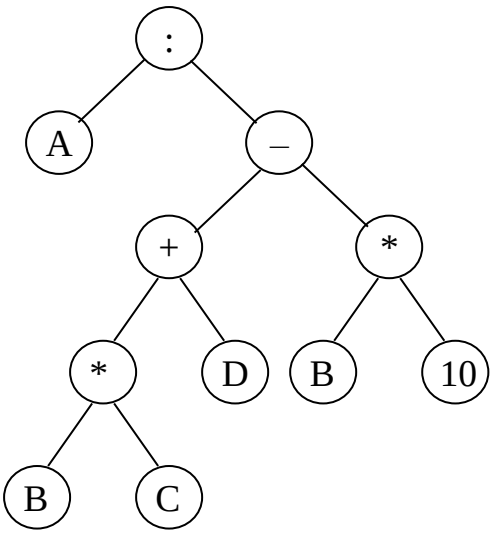
Все внутренние представления программы обычно содержат в себе две принципиально различные вещи – операторы и операнды. Различия между формами внутреннего представления заключены только в том, как операторы и операнды соединяются между собой.

Можно назвать некоторые основные формы внутреннего представления программ:

– синтаксические деревья, имеющие вид списочных структур;
– многоярусный код с явно именуемым результатом (тетрады);
– многоярусный код с неявно именуемым результатом (триады);
– обратная (постфиксная) польская запись операций;
– префиксная польская запись операций;
– ассемблерный код или машинные команды.

В конкретном компиляторе может использоваться одна из этих форм или их некоторое сочетание. На различных фазах компиляции могут использоваться различные формы, которые по мере выполнения проходов компилятора преобразуются одна в другую. Рассмотрим эти формы и проиллюстрируем их на примере выражения **A:=B*C+D–B*10**.

– *Синтаксические деревья* рассматривались ранее. Они отражают полную взаимосвязь операций и являются машинно-независимой формой внутреннего представления программы. Их недостаток заключается в том, что они представляют собой сложные связные структуры, поэтому не могут быть тривиальным образом преобразованы в линейную последовательность команд. Тем не менее они удобны для работы с внутренним представлением программы на тех этапах, когда нет необходимости обращаться к командам результирующей программы. Синтаксические деревья могут быть преобразованы в линейные списки с учетом семантики входного языка.



– *Тетрады* представляют собой запись операций в форме из четырёх

составляющих: операция, два операнда и результат операции. Отсутствующий операнд может быть опущен или заменён пустым. Тетрады являются линейной последовательностью команд, вычисляются одна за другой последовательно. Порядок вычисления тетрад может быть изменён, если только предусмотреть наличие тетрад, целенаправленно изменяющих этот порядок. Последовательность тетрад легко преобразуется в последовательность команд результирующей программы. Тетрады представляют собой машинно-независимую форму внутреннего представления программы. Наше выражение A:=B*C+D–B*10, записанное в форме тетрад, будет иметь вид:

***, B, C, T1 +, T1, D, T2 *, B, 10, T3 –, T2, T3, T4 :=, A, T4, A**

Здесь есть некоторая возможность оптимизации; вместо двух последних тетрад можно записать одну: **–, T2, T3, A**.

– *Триады* представляют собой запись операций в форме из трех составляющих: операция и два операнда. 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. Триады также представляют собой линейную последовательность команд и вычисляются одна за другой. При вычислении триады операция выполняется над операндами, а если в качестве операнда выступает ссылка на другую триаду, то используется результат её вычисления. Результат вычисления триады сохраняется во временной памяти – в этом её отличие от тетрады. Триады представляют собой также машинно-независимую форму внутреннего представления программы. Они требуют меньше памяти для своего представления, чем тетрады, и явно отражают взаимосвязь операций между собой, что более удобно. Наше выражение **A:=B*C+D–B*10** в форме триад будет иметь вид:

1. *, B, C 2. +, ^1, D 3. *, B, 10 4. –, ^2, ^3 5. :=, A, ^4

– Обратная (постфиксная) *польская запись* предполагает, что знаки операций записываются после операндов. В этой форме записи операции следуют одна за другой строго в порядке их выполнения, что весьма удобно для вычисления выражений на компьютере. В ней отсутствует приоритет операций и не используются скобки, что является её несомненным преимуществом. Особенно эффективен этот способ записи при использовании стека. Главный недостаток обратной польской записи – при использовании стека доступна только его верхушка, что затрудняет оптимизацию выражений. Поэтому она широко используется для вычисления выражений в интерпретаторах и командных процессорах, где оптимизация отсутствует или несущественна. Наше выражение в постфиксной записи примет вид: **ABC*D+B10*–:=**.

– В префиксной *польской записи* знаки операций записываются перед

операндами; эта запись также является бесскобочной: **:=A–+*BCD*B10**.

– *Машинные команды* удобны тем, что при их использовании внутреннее представление программы полностью соответствует машинному коду и сложные преобразования не требуются. Хотя в этом случае внутреннее представление программы получается зависимым от архитектуры вычислительной системы, на которую ориентирован результирующий код, но зато при использовании этой формы представления можно добиться наиболее эффективной результирующей программы.

4.3.4 Семантический анализ

Практически все языки программирования не являются, строго говоря, КС-языками, поэтому полный разбор цепочек языка компилятор не может выполнить только с помощью КС-грамматик и МП-автоматов. Но строить компилятор на основе КЗ-грамматик неэффективно, поэтому реальные компиляторы выполняют разбор в два этапа – синтаксический разбор на основе распознавателя КС-языка, затем – семантический анализ.

Входными данными для семантического анализатора служат результат синтаксического разбора и таблица идентификаторов. Обычно семантический анализ частично выполняется на этапе синтаксического разбора и частично – на этапе генерации кода. В первом случае всякий раз по завершении распознавания определенной синтаксической конструкции языка (например, процедуры, функции, и т.п.) выполняется её семантическая проверка на основе содержащихся в таблицах идентификаторов данных. Во втором случае после завершения всей фазы синтаксического разбора проводится полный семантический анализ на основании данных в таблице идентификаторов.

Семантический анализатор выполняет следующие действия:

– проверка соблюдения во входной программе семантических соглашений входного языка;
– дополнение внутреннего представления программы операторами и действиями, неявно предусмотренными семантикой входного языка;
– проверка элементарных смысловых норм языков программирования, не связанных напрямую с входным языком.

Рассмотрим, что может подразумеваться под названными действиями.

– Каждый язык имеет свои соглашения, которые не могут быть проверены на этапе синтаксического разбора. Примеры таких соглашений: каждый идентификатор должен быть описан, и в одном блоке – не более одного раза; типы переменных в выражениях должны быть согласованы; число и типы формальных и фактических параметров подпрограмм должны быть согласованы и т.п.

Пример: Синтаксически правильный оператор Паскаля ***a:=b+c*** может

оказаться лишённым смысла, причем установить, является ли он правильным с точки зрения входного языка, можно только после проверки семантических требований для всех входящих в него лексем. Фактически в этом примере нужно знать, что именно представляют из себя идентификаторы ***a***, ***b***, ***c***. Этот оператор окажется ошибочным, если, например, a – константа, или если какой-то из операндов не описан, или эти идентификаторы обозначают разнородные элементы (один – строка, другой – число или массив, и т.п.). Смысл этого оператора может быть совершенно различным, если, например, под идентификаторами понимать в одном случае – числа, в другом –строки.

– Дополнение внутреннего представления программы операторами и действиями, предусмотренными семантикой входного языка, может быть (наиболее часто) связано с преобразованием типов операндов в выражениях и при передаче параметров в процедуры и функции.

Пример: Если рассмотреть тот же оператор языка Паскаль ***a:=b+c*** и предположить, что используемые в нём переменные имеют разный тип, допустим: ***a*** – Real, ***b*** – Integer, ***c*** – Double, то в таком случае, раз преобразования типов в явном виде в программе не присутствуют, они должны осуществляться в коде. Для этого в составе библиотек функций, доступных компилятору, должны присутствовать функции преобразования типов, и вызовы этих функций как раз и будут встроены в результирующий код. Тогда в нашем примере вместо двух операций окажется четыре: преобразование целочисленной переменной ***b*** в формат вещественных чисел с двойной точностью, сложение двух чисел этого типа, преобразование результата сложения в формат Real и присвоение результата переменной ***a***.

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

Например, это могут быть следующие требования:

- каждая переменная или константа должна хотя бы один раз использоваться в программе;
- каждая переменная должна быть определена до её первого использования при любом ходе выполнения программы;
- результат функции должен быть определен при любом ходе её выполнения;
- каждый оператор программы должен иметь возможность хотя бы один раз выполняться;
- операторы условия и выбора должны предусматривать возможность хода выполнения программы по каждой из своих ветвей;
- операторы цикла должны предусматривать возможность завершения цикла.

Конкретный состав соглашений зависит от семантики языка. То, какие именно соглашения будут выполняться и как, зависит от разработчика. Простейший компилятор может вообще не выполнять этот этап семантического анализа. В любом случае, несоблюдение соглашений такого сорта не может трактоваться как ошибка – как правило, компилятором выдается «предупреждение», и то, реагировать на него или нет, относится к компетенции разработчика программы.

После семантического анализа исходной программы выполняется распределение памяти на основе информации, находящейся в таблице идентификаторов. Результаты распределения памяти используются в процессе генерации кода.

4.3.5 Генерация и оптимизация кода

Генерация объектного кода представляет собой перевод внутреннего представления программы в результирующую объектную программу на языке ассемблера или непосредственно на машинном языке. Внутреннее представление программы может иметь любую структуру, в то время как результирующая программа является линейной последовательностью команд; поэтому компилятор должен выполнять действия, связанные с преобразованием сложных синтаксических структур в линейные цепочки. При генерации кода используются методы перевода, рассмотренные выше.

Как правило, компилятор выполняет генерацию результирующего кода поэтапно, на основе законченных синтаксических конструкций входной программы. Для каждой синтаксической конструкции порождается фрагмент результирующего кода и помещается в текст выходной программы. В качестве исследуемых синтаксических конструкций выступают операторы, блоки операторов, описания процедур и функций и т.п.

Семантика каждой синтаксической конструкции входного языка определяется, исходя из её типа, который определяется синтаксическим анализатором на основании грамматики входного языка. Примеры типов синтаксических конструкций – операторы цикла, условные операторы, операторы выбора и т.п. Одни и те же конструкции характерны для различных языков программирования, при этом они различаются синтаксисом, но имеют схожий смысл.

Для построения кода результирующей программы для синтаксической конструкции входного языка часто используется СУ-перевод, который по дереву разбора позволяет построить линейную последовательность команд. При этом может использоваться рекурсивная процедура обхода дерева. Например, процедура генерации кода по дереву операций прежде всего должна определить тип узла дерева, который соответствует типу операции, символом которой помечен узел дерева. После определения типа узла для него строится код в соответствии с типом операции. Если

для текущего узла все узлы следующего уровня есть листья дерева, то в код включаются операторы, соответствующие этим листьям, и получившийся код становится результатом выполнения процедуры. Иначе процедура должна рекурсивно вызвать сама себя для генерации кода нижележащих узлов дерева и результат выполнения включить в свой порождённый код. Для каждого узла дерева процедура генерации кода должна выполнить конкатенацию цепочек команд, связанных с текущим узлом и с нижележащими узлами. При этом операции, связанные с нижележащими узлами, должны выполняться раньше, чем связанные с текущим узлом.

Как было сказано выше, генерация кода выполняется последовательно для отдельных конструкций программы. При объединении их в общий текст результирующей программы связи между различными фрагментами практически не учитываются. Поэтому построенный таким образом код может содержать лишние команды, что снижает эффективность выполнения результирующей программы.

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

Проблема оптимизации кода является очень сложной; на практике обычно довольствуются улучшением кода. На разных стадиях процесса компиляции применяются различные приемы улучшения кода. Если компилятор использует несколько форм внутреннего представления, то каждя из них может быть подвергнута оптимизации: можно оперировать со структурами, порожденными на стадии синтаксического анализа или в качестве выхода фазы генерации кода. При этом критериями эффективности являются два – скорость работы или объем памяти.

Методы преобразования программы зависят от типов синтаксических конструкций исходного языка. Например, оптимизация может выполняться для следующих типовых синтаксических конструкций:

– линейных участков программ;
– логических выражений;
– циклов;
– вызовов процедур и функций.

Преобразования по улучшению кода можно разделить на машинно-независимые и машинно-зависимые. Примером машинно-независимой оптимизации может служить удаление из программы бесполезных операторов, т.е. таких, которые никаким образом не влияют на выход программы. Поскольку такие операторы вообще не должны появляться в программе, их наличие может означать какую-то ошибку, следовательно, компилятор должен информировать пользователя об обнаружении им подобного оператора. Такие машинно-независимые преобразования полезно иметь во всех компиляторах. Другие примеры – это исключение

избыточных вычислений, перестановка операций (изменение порядка следования операций, которое может повысить эффективность вычислений, но не повлияет на конечный результат).

С помощью машинно-зависимых преобразований можно попытаться привести программу к форме, в которой могут сказаться преимущества, связанные с машинными командами специального вида. Это вопрос достаточно сложный и требует специального рассмотрения: поскольку машинно-зависимые методы оптимизации ориентированы на конкретную архитектуру вычислительной системы, при существующем многообразии архитектур рассмотреть их все вряд ли возможно.

Можно в качестве примеров машинно-зависимой оптимизации назвать такие виды, как распределение регистров процессора (для хранения промежуточных результатов) и порождение кода для параллельных вычислений (как правило, параллельное выполнение операций возможно при вычислении значения арифметических выражений).

4.3.6 Анализ и исправление ошибок

До сих пор предполагалось, что входом компилятора служит правильно построенная программа и что каждую фазу компиляции можно осмысленно довести до конца. На практике же во многих случаях это оказывается не так или не совсем так.

Компилятор имеет возможность обнаруживать ошибки в программе по крайней мере на трех этапах компиляции: во время лексического анализа, синтаксического анализа и при генерации кода.

Часто по неправильной программе компилятору бывает трудно определить, что же имел в виду автор – эта задача порой граничит с приложениями искусственного интеллекта. Хотя в некоторых случаях бывает легко сделать подходящее предположение. Например, если исходный оператор выглядит как A=B+2C, то вполне правдоподобно предположить, что имелось в виду A=B+2*C. В тех случаях, когда процесс разбора доходит до того места во входной цепочке, где он не может дальше правильно продолжаться, некоторые компиляторы стараются произвести минимальное изменение во входной цепочке, с тем, чтобы получить возможность продолжить разбор.

Примеры возможных изменений подобного вида:

1) Замена одного знака. Например, если лексический анализатор выдаёт синтаксическому анализатору идентификатор INTEJER в неподходящем для появления идентификатора месте программы, то компилятор может догадаться, что подразумевается ключевое слово INTEGER.

2) Вставка одной лексемы. Например, компилятор может заменить 2C на 2*C.

3) Устранение одной лексемы. Например, в таком операторе Фортрана, как DO 10 I=1,15 часто неправильно ставят запятую после 10.

4) Простая перестановка лексем.

Если разбор конкретного неправильно построенного оператора становится безнадёжным даже после того, как произведены изменения вроде описанных выше, часто бывает можно просто игнорировать этот неправильный оператор и продолжать разбор так, как будто его не было.

Некоторые алгоритмы синтаксического анализа обладают свойством заканчивать работу программы тогда, когда станет уже ясным, что просмотренную уже часть входной цепочки нельзя продолжить до правильной цепочки.

Как уже было сказано выше, к процедуре анализа и исправления ошибок компилятор может обращаться на этапах лексического анализа, синтаксического анализа и генерации кода. Если исправление произошло успешно, то процесс компиляции продолжается с того места, где произошло обращение к подпрограмме анализа и исправления ошибок. Ошибки разного сорта обнаруживаются на разных стадиях компиляции.

Так, ошибки, при которых в некотором месте входной цепочки не оказалось никакой лексемы, обнаруживаются в ходе лексического анализа. Ошибки, при которых входную программу можно разбить на лексемы, но к этой последовательности лексем не подходит никакое дерево, обнаруживаются в ходе синтаксического анализа. Наконец, ошибки, при которых входная цепочка имеет синтаксическую структуру, но для неё не получается осмысленный код, обнаруживаются в процессе генерации кода.

Рассмотренная модель является только приближением к реальному компилятору. На практике реальные компиляторы проектируются в соответствии с определенными ограничениями – например, так, чтобы они занимали небольшой объем памяти, и тогда получается большее количество фаз компиляции.

4.3.7 Контрольные вопросы

1. Какая информация помещается в таблицы идентификаторов? На каких этапах это происходит?
2. Как должны быть организованы таблицы идентификаторов? Могут ли с ними выполняться ошибочные действия?
3. Что является первым этапом компиляции? Чем обусловлено выделение его в отдельный этап?
4. Чем различаются прямой и непрямой лексический анализ?
5. Какая из стадий процесса компиляции является переводом?
6. Что подаётся на вход синтаксического анализатора?
7. Какие действия выполняются в ходе синтаксического анализа? Какие методы могут быть для этого использованы?

8. Что является результатом этапа синтаксического анализа?
9. Что такое внутреннее представление программы? Какие существуют виды внутреннего представления?
10. Что происходит при семантическом анализе? Возможна ли ситуация, когда синтаксически правильная языковая конструкция оказывается семантически неверной?
11. Почему генерируемый объектный код может оказаться не оптимальным? Какие существуют способы его улучшения?
12. Какого рода ошибки может обнаруживать компилятор? Может ли он каким-то образом пытаться их исправить и к каким результатам это может привести?