Министерство образования и науки Российской Федерации

НОВОСИБИРСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ

004
Я 411

№ 4056

ЯЗЫКИ ПРОГРАММИРОВАНИЯ И МЕТОДЫ ТРАНСЛЯЦИИ

Методические указания
к лабораторным работам для студентов III курса ФПМИ
направления 010400, 010500

НОВОСИБИРСК
2011

УДК 004.43(076.5)
Я 411

Методические указания содержат: требования по выполнению лабораторных работ и содержанию отчетов, контрольные вопросы к лабораторным работам, варианты заданий, темы расчетно-графических заданий, список рекомендуемой литературы.

Предназначены для студентов ФПМИ, изучающих курс «Языки программирования и методы трансляции».

Составили: канд. техн. наук, доцент *И.Л. Еланцева*
канд. техн. наук, доцент *И.А. Полетаева*

Рецензент д-р техн. наук, профессор *Д.В. Лисицин*

Работа подготовлена на кафедре прикладной математики

ВВЕДЕНИЕ

• *Актуальность курса.* В своей повседневной деятельности специалисты в области математического обеспечения ЭВМ каждодневно соприкасаются с системами программирования, к которым наряду с традиционными компиляторами и интерпретаторами можно отнести и системы управления базами данных и различные информационно-поисковые, интеллектуальные, экспертные и другие программные системы. Система программирования в широком смысле объединяет в себе понятия: язык программирования и виртуальная машина, которая поддерживает исполнение программ на реальной ЭВМ. Теоретическим базисом построения систем программирования являются теория формальных языков и грамматик, теория автоматов, методы трансляции. Полноценная подготовка специалистов в области математического обеспечения невозможна без освоения основ этой теории и получения практических навыков по программированию элементов транслирующих систем.

• Курс входит в число дисциплин, включенных *в учебные планы* подготовки бакалавров в соответствии с государственным образовательным стандартом по направлениям 010400 и 010500.

• Обучающийся данному курсу *должен обладать знаниями* теории множеств, теории формальных языков и грамматик, теории автоматов.

• *Курс адресован* студентам, специализирующимся в областях прикладного и системного программирования, занимающимся решением задач обработки информации на ЭВМ с применением лингвистических моделей.

• *Основная цель* обучающихся: освоить основы синтаксически управляемых методов обработки языков на примере процесса трансляции с алгоритмических языков высокого уровня.

• Курс имеет *практическую часть* (лабораторные занятия – 17 часов и расчетно-графическая работа – 17 часов). Студенты применяют

теоретические положения при программировании элементов трансляторов.

• Для выполнения лабораторных работ используются методические указания, конспект лекций, выпущенный издательством НГТУ.

• *Оценка* знаний и умений проводится по результатам ответов на контрольные вопросы при защите лабораторных работ и РГЗ, а также по ответам на вопросы к теоретическому зачету.

• Содержание лабораторных работ состоит в проектировании транслятора (язык проектирования Си++) с языка Си++ на язык Ассемблер. Этапы проектирования транслятора:

1) проектирование таблиц лексем, используемых в трансляторе;
2) проектирование блока лексического анализа (сканер);
3) проектирование блока синтаксического анализа;
4) проектирование генератора кода.

Транслятор может быть реализован в виде одно-, двух- или трехпроходной схемы. По окончании каждого этапа проектирования (каждой лабораторной работы) следует оформить промежуточный отчет. Для защиты всего проекта необходимо подготовить итоговый отчет, включающий отчет по РГЗ, оформленный по общепринятым требованиям.

ЛАБОРАТОРНАЯ РАБОТА № 1

ПРОЕКТИРОВАНИЕ И РЕАЛИЗАЦИЯ ТАБЛИЦ, ИСПОЛЬЗУЕМЫХ В ТРАНСЛЯТОРЕ

ЦЕЛЬ РАБОТЫ

Получить представление о видах таблиц, используемых при трансляции программ. Изучить множество операций с таблицами и особенности реализации этих операций для таблиц, используемых на этапе лексического анализа. Реализовать классы таблиц, используемых сканером.

МЕТОДИЧЕСКИЕ УКАЗАНИЯ

Структурная схема транслятора представлена на рис. 1. Из перечисленных на рис. 1 фаз компиляции обязательными являются лексический анализ, синтаксический анализ и генерация кода. Остальные задачи компиляции решаются с той или иной мерой полноты в зависимости от целей и назначения компилятора.

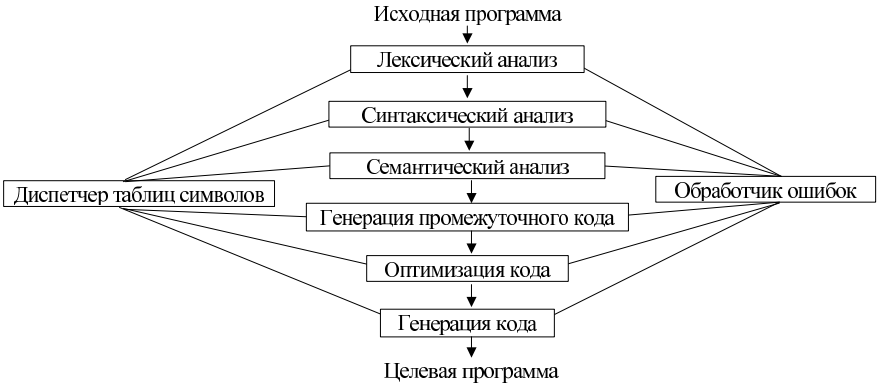

Рис. 1. Фазы компилятора

На всех этапах трансляции используются различные таблицы, которые можно классифицировать на постоянные и переменные. *Постоянные* таблицы создаются одновременно с проектированием транслятора и не изменяются в процессе работы с транслируемой программой. К таким таблицам относятся:

1) таблицы, содержащие алфавит входного языка;
2) таблица зарезервированных слов;
3) таблицы, реализующие какой-либо из методов синтаксического анализа (*LL*-разбор, *LR*-разбор и др.).

К *переменным* таблицам относятся таблицы, создаваемые в процессе трансляции исходной программы:

1) таблица символов (идентификаторов, имен);
2) таблица констант.

Лексема – неделимая единица языка и ее атрибуты.

Единица языка – константа, идентификатор, ключевое слово, разделитель, знак операции.

Множество атрибутов определяется типом единицы языка и может включать имя, значение, тип, параметры (для функций).

Лексемы сохраняются в соответствующих таблицах лексем.

Одной из важнейших функций компилятора является запись используемых в исходной программе идентификаторов и сбор информации о различных атрибутах каждого идентификатора. Эти атрибуты предоставляют сведения об отведенной идентификатору памяти, его типе, области видимости (в какой части программы он может применяться). При использовании имен процедур атрибуты говорят о количестве и типе их аргументов, методе передачи каждого аргумента (например, по ссылке) и типе возвращаемого значения, если таковое имеется. Вся эта информация составляет лексему.

Таблица символов представляет собой структуру данных, содержащую записи о каждом идентификаторе с полями для его атрибутов. Данная структура позволяет найти информацию о любом идентификаторе и внести необходимые изменения.

Когда идентификатор считан из исходной программы, требуется определить, не появлялся ли этот идентификатор ранее. Если лексическим анализатором в исходной программе обнаружен новый идентификатор, он записывается в таблицу символов.

После того как лексема сохранена или найдена в таблице символов, сохраняем токен (тип лексемы и адрес в соответствующей таблице), связанный с этой лексемой, в файле токенов. Однако атрибуты иден-

тификатора обычно не могут быть определены в процессе лексического анализа. В процессе остальных фаз информация об идентификаторе вносится в таблицу символов и используется различными способами. Например, при семантическом анализе и генерации промежуточного кода необходимо знать типы идентификаторов, чтобы гарантировать их корректное использование в исходной программе и сгенерировать правильные операции по работе с ними. Обычно генератор кода вносит в таблицу символов и использует детальную информацию о памяти, назначенной идентификаторам.

Механизм таблицы символов должен обеспечивать эффективный поиск и добавление символов в таблицу. Такими механизмами могут быть линейные списки и хеш-таблицы. Линейный список более прост в реализации, но его производительность невысока. Хеширование обеспечивает более высокую производительность, но за счет большего количества памяти и несколько больших усилий по программированию. При работе с таблицей символов полезна возможность ее динамического роста в процессе компиляции. Для хранения лексем, образующих идентификаторы, не стоит отводить память фиксированного размера. Этой памяти может оказаться недостаточно для очень длинного идентификатора и слишком много для коротких идентификаторов (например, i), что приводит к нерациональному использованию памяти.

Процедуры для работы с таблицей символов ориентированы в основном на хранение и получение лексем. Класс, реализующий таблицу символов, должен содержать следующие основные функции: поиск/добавление идентификатора в таблице; поиск/добавление атрибутов идентификатора в таблицу.

Многие языки используют определенные заранее строки символов, такие как while, main и другие, в качестве символов пунктуации или для указания конкретных конструкций. Такие строки символов, именуемые ключевыми словами, обычно удовлетворяют правилам образования идентификаторов, а потому требуется отличать ключевые слова от прочих идентификаторов. Проблема упрощается, если ключевые слова зарезервированы. В таком случае строка символов является идентификатором, если она не ключевое слово.

Везде, где в выражении встречается отдельное число, на его месте естественно использовать произвольную константу, ее участие в трансляции обозначается созданием лексемы и соответствующего ей токена для такой константы. Таблица констант представляет собой структуру данных, содержащую записи о каждой константе с полями

для ее атрибутов (тип константы и др.). Данная структура позволяет выполнять корректную работу с константами на всех этапах трансляции. Организация таблицы констант и процедуры работы с ней аналогичны механизмам работы с таблицей символов.

ЗАДАНИЕ

В соответствии с выбранным вариантом задания к лабораторным работам с использованием средств объектно-ориентированного программирования:

1) разработать структуру постоянных таблиц для хранения алфавита языка, зарезервированных слов, знаков операций, разделителей и пр.; реализовать для постоянных таблиц алгоритм поиска элемента в упорядоченной таблице;

2) разработать структуру переменных таблиц с вычисляемым входом для хранения идентификаторов и констант (вид хеш-функции и метод рехеширования задает разработчик);

3) реализовать для переменных таблиц алгоритмы поиска/добавления лексемы, поиска/добавления атрибутов лексемы;

4) разработать программу для тестирования и демонстрации работы программ пп. 1–3.

СОДЕРЖАНИЕ ОТЧЕТА

- Цели и задачи проекта.
- Вид, объем и источник исходных данных.
- Структура таблиц.
- Тексты программ.
- Тестовые примеры.

КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Постоянные таблицы, используемые в трансляторах.
2. Переменные таблицы, используемые в трансляторах.
3. Операции с таблицами.
4. Способы организации таблиц.
5. Хеш-функции.
6. Методы рехеширования.
7. Таблицы с вычисляемым входом.

ЛАБОРАТОРНАЯ РАБОТА № 2

РАЗРАБОТКА И РЕАЛИЗАЦИЯ БЛОКА ЛЕКСИЧЕСКОГО АНАЛИЗА (СКАНЕР)

ЦЕЛЬ РАБОТЫ

Изучить методы лексического анализа. Получить представление о методах обработки лексических ошибок. Научиться проектировать сканер на основе детерминированных конечных автоматов.

МЕТОДИЧЕСКИЕ УКАЗАНИЯ

Лексический анализатор конвертирует поток входных символов (исходную транслируемую программу) в поток токенов, который будет являться входным потоком для следующей фазы трансляции – синтаксического анализа. При лексическом анализе символы исходной программы считываются и группируются в поток токенов, в котором каждый токен представляет логически связанную последовательность символов – идентификатор, ключевое слово (if, while и т. п.), символ пунктуации или многосимвольный оператор (++, <=, &&, += и т. п.).

Лексический анализатор ограждает синтаксический анализатор от непосредственной работы с лексемами, представленными токенами. Рассмотрение лексического анализатора начнем с перечисления некоторых выполняемых им функций:

- удаление «пустых» символов и комментариев. Если «пустые» символы (пробелы, знаки табуляции и перехода на новую строку) и комментарии будут удалены лексическим анализатором, синтаксический анализатор никогда не столкнется с ними. Альтернативный способ, состоящий в модификации грамматики для включения «пустых» символов и комментариев в синтаксис, достаточно сложен для реализации;
- распознавание идентификаторов и ключевых слов;
- распознавание констант;
- распознавание разделителей и знаков операций.

Каждый распознаватель представляет собой детерминированный конечный автомат, совокупность которых составляет основу сканера.

На уровне лексического анализатора определяются только некоторые ошибки, поскольку лексический анализатор рассматривает исходный текст программы в ограниченном контексте. Если в программе на языке Си++ строка **fi** впервые встречается в строке **«fi(x>0)»**, то лексический анализатор не сможет определить, является ли **fi** неверно записанным ключевым словом **if** или идентификатором **fi**. Поскольку **fi** является корректным идентификатором, лексический анализатор должен сформировать лексему и токен для идентификатора и предоставить обработку ошибки синтаксическому анализатору, который выдаст сообщение о неверном использовании идентификатора **fi** в данной строке «Неопределенный идентификатор **fi**».

Представим, что лексический анализатор не способен продолжать работу, поскольку ни один из шаблонов не соответствует оставшейся части входного потока (например, в строке «12a34» после начальных цифр встречается буква «a»). Простейшим решением в этой ситуации будет восстановление в «режиме паники». Мы просто пропускаем входные символы до тех пор, пока лексический анализатор не встретит распознаваемый токен; соответствующее сообщение об ошибке выдается пользователю. Иногда это запутывает синтаксический анализатор, но для интерактивной среды данная технология может оказаться вполне адекватной. Существуют другие возможные действия по восстановлению после ошибки:

1) удаление лишних символов;
2) вставка пропущенных символов;
3) замена неверного символа верным;
4) перестановка двух соседних символов.

При восстановлении корректного входного потока могут выполняться различные преобразования. Простейшая стратегия состоит в проверке, не может ли начало оставшейся части входного потока быть заменено корректной лексемой посредством единственного преобразования. Эта стратегия полагает, что большинство лексических ошибок вызвано единственным неверным преобразованием (по сути, опечаткой), и это предположение подтверждается практикой. Пользователю выдается соответствующее сообщение об ошибке и предпринятых действиях по ее исправлению.

Чтобы понять, как происходит анализ входного потока символов, рассмотрим основные понятия теории формальных языков и грамматик.

Формальные языки и грамматики

Алфавит – это непустое конечное множество элементов. Элементы алфавита называются символами.

Цепочка – всякая конечная последовательность символов алфавита.

Если x и y – цепочки, то их *конкатенацией* xy (или *катенацией*) является цепочка, полученная дописыванием символов цепочки y вслед за символами цепочки x.

Хомский впервые в 1956 г. описал формальный язык. *Формальный язык* – это множество цепочек, полученных по некоторым общим правилам.

Например, всевозможные идентификаторы составят формальный язык идентификаторов, в этом случае идентификаторы являются цепочками языка, а литеры – символами языка. Аналогично множество всевозможных программ тоже составит формальный язык, каждая программа может рассматриваться как цепочка символов (ключевых слов, идентификаторов, разделителей, констант).

Правила формирования цепочек языка составляют *грамматику* данного языка.

Правилом подстановки называется упорядоченная пара (y, x), которая обычно записывается $y::=x$, где y, x – непустые конечные цепочки символов; y называется *левой частью*, x – *правой частью* правила. Данное правило обозначает, что последовательность символов, совпадающая с цепочкой y, будет заменена на цепочку x.

Все символы, входящие в алфавит, разделяют на *терминальные* и *нетерминальные. Терминальные символы* – это символы, входящие в цепочки формального языка. *Нетерминальные* символы – это вспомогательные, временные символы, которые используются при описании правил грамматики и отсутствуют в цепочках языка.

Грамматикой $G[z]$ является упорядоченная четверка (V, T, P, z), где V – алфавит; $T \subseteq V$ – алфавит терминальных символов; P – конечный набор правил подстановки; z – начальный символ, с которого начинается вывод любой цепочки языка $z \in (V - T)$; z должен встретиться в левой части, по крайней мере, одного правила. Язык, порожденный грамматикой $G[z]$, – это множество терминальных цепочек, которые можно вывести из z с помощью правил P. Если из контекста ясно, какой символ является символом z, то вместо $G[z]$ будем писать G.

Множество нетерминальных символов обозначают V_N. Как правило, нетерминалы будем заключать в угловые скобки < >.

Множество правил U:: = x, U:: = y, U:: = z с одинаковыми левыми частями будем записывать U:: = x | y | z.

Пример

Грамматика, позволяющая получить десятичное число, G[<число>], записывается следующим образом: V={<число> , <цифра>, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9}, T = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9},

Правила грамматики P:

<число> :: = <число><цифра> | <цифра>

<цифра> :: = 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Эта форма записи называется нормальной формой Бэкуса (НБФ) или формой Бэкуса–Наура.

Мы говорим, что цепочка v непосредственно порождает цепочку w, и обозначаем это $v \Rightarrow w$, если для некоторых цепочек x и y можно написать $v = xUy$, $w = xuy$, где U:: = u – правило грамматики G. Мы также говорим, что w *непосредственно выводима из* v . Цепочки x и y могут быть пустыми.

Говорят, что v *порождает* w или w *приводится* к v, что записывается как $v \Rightarrow +w$, если существует последовательность непосредственных выводов $v = u1 \Rightarrow u2 \Rightarrow ... \Rightarrow un = w$, n>0. Эта последовательность называется *выводом длины* n. Пишут $v \Rightarrow *w$, если $v \Rightarrow + w$ или $v \Rightarrow w$.

Заметим, что, пока в цепочке есть хотя бы один нетерминал, из нее можно вывести новую цепочку.

Пусть $G[z]$ – грамматика. Цепочка x называется *сентенциальной формой*, если x выводится из z, т. е. $z \Rightarrow *x$.

Предложение – это сентенциальная форма, состоящая из терминальных символов.

Язык $L(G[z])$ – это множество предложений.

Хомский определил четыре основных класса языков в терминах грамматик.

Различие четырех типов грамматик заключается в форме правил подстановки, допустимых в P.

Говорят, что G – это (по Хомскому) грамматика типа 0 или грамматика с фразовой структурой, если правила имеют вид

$$u :: = v, \quad \text{где} \quad u \in V^{+} \quad \text{и} \quad v \in V^{*},$$

т. е. левая часть может быть тоже последовательностью символов, а правая часть может быть пустой. Языки этого класса могут служить моделью естественных языков.

Грамматика типа 1 (контекстно-чувствительные, или контекстно-зависимые языки) имеют правила подстановки вида

$$xUy ::= xuy, \quad \text{где} \quad U \in (V - T), \quad u \in V^{+} \quad \text{и} \quad x, y \in V^{*}.$$

«Контекстно-чувствительная» отражает тот факт, что U можно заменить на u лишь в контексте $x...y$.

Грамматика называется контекстно-свободной – типа 2 (КС-грамматика), если все ее правила имеют вид

$$U ::= u, \quad \text{где} \quad U \in (V - T), \quad u \in V^{*}.$$

Грамматика типа 2 является хорошей моделью для языков программирования.

Регулярная грамматика (тип 3 или автоматная грамматика, А-грамматика) – это грамматика, правила которой имеют вид

$$U ::= N \quad \text{или} \quad U ::= WN, \quad \text{где} \quad N \in T, \quad \text{а} \quad W, U \in (V - T).$$

Регулярные грамматики играют основную роль как в теории языков, так и в теории автоматов.

В реальных языках программирования отдельные подмножества можно отнести к третьему классу.

Иерархия языков по Хомскому – включающая, т. е. все грамматики типа 3 являются грамматиками типа 2, все грамматики типа 1 являются грамматиками типа 0 и т. д. Иерархия грамматик соответствует иерархии языков. Например, если язык можно генерировать с помощью грамматики типа 2, то его называют языком типа 2. Эта иерархия опять включающая. Включения справедливы также в том, что существуют языки, которые относятся к типу i, но не к типу $(i + 1)(0 <= i <= 2)$.

Иерархия Хомского важна с точки зрения языков программирования. Чем меньше ограничений в грамматике, тем сложнее ограничения, которые можно наложить на генерируемый язык. Таким образом, оказывается, что чем более универсален язык используемой грамматики, тем больше средств типичных языков программирования можно описать.

Однако чем универсальнее грамматика, тем сложнее должна быть машина (или программа), которая применяется для распознавания строк соответствующего языка.

С грамматикой типа 3 ассоциируется класс распознавателей, известный как конечный автомат, или машина с конечным числом состояний, между которыми происходит передача управления по мере считывания символов строки, причем строка принимается или нет в зависимости от того, какого состояния машина достигает в итоге. Для

языка, генерируемого с помощью КС-грамматики, необходим автомат магазинного типа, т. е. конечный автомат плюс стек, а для контекстно-зависимых языков – линейный автомат с ограничениями, т. е. машина Тьюринга с конечным объемом ленты. Наконец, языку типа 0 требуется машина Тьюринга в качестве распознавателя.

Регулярные выражения и конечные автоматы

Диаграммы состояний

Рассмотрим регулярную грамматику $G[Z]$.

$Z ::= U0 \mid V1$

$U ::= Z1 \mid 1$

$V ::= Z0 \mid 0$

$L(G) = \{ B^n \mid n > 0 \}$, где $B = \{01, 10\}$

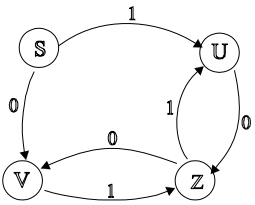

Рис. 2. Диаграмма состояний КА

Легко видеть, что порождаемый ею язык состоит из последовательностей, образуемых парами 01 или 10. Чтобы облегчить распознавание предложений грамматики G, нарисуем диаграмму состояний (рис. 2).

В этой диаграмме каждый нетерминал грамматики G представлен узлом или состоянием; кроме того, есть *начальное состояние* S, предполагается, что G не содержит S.

Каждому правилу $Q ::= T$ соответствует дуга с пометкой T, направленная от начального состояния S к состоянию Q. Каждому правилу $Q ::= RT$ соответствует дуга с пометкой T, направленная от состояния R к состоянию Q.

Чтобы распознать или разобрать цепочку x, используем диаграмму состояний следующим образом.

1. Первым текущим состоянием считаем начальное состояние. Начинаем с самой левой литеры в цепочке x, повторяем шаг 2 до тех пор, пока не будет достигнут правый конец x.

2. Сканируем следующую литеру строки x, продвигаемся по дуге, помеченной этой литерой, переходя к следующему состоянию.

Если при каком-то повторении шага 2 такой дуги не окажется, то цепочка x не является предложением. Если мы достигаем конца x, то x – предложение тогда и только тогда, когда последнее текущее состояние есть Z.

Последовательность действий соответствует алгоритму восходящего разбора. На каждом шаге (кроме первого) основой является имя

текущего состояния, за которым следует входной символ. Символ, к которому приводится основа, будет именем следующего состояния.

Пример

Проведем разбор предложения 101001

Шаг	Текущее состояние	Остаток цепочки *x*
1	*S*	101001
2	*U*	01001
3	*Z*	1001
4	*U*	001
5	*Z*	01
6	*V*	1
7	*Z*	

В данном примере разбор выглядит простым благодаря простому характеру правил. Нетерминалы встречаются лишь как первые символы правой части. На первом шаге первый символ предложения всегда приводит к нетерминалу. На каждом последующем шаге первые два символа *UT* сентенциальной формы *UTt* приводят к нетерминалу *V*, при этом используется правило *V*:: = *UT*. При выполнении этой редукции имя текущего состояния *U*, а имя следующего текущего состояния *V*. Так как каждая правая часть единственна, то единственным оказывается символ, к которому она приводится.

Чтобы избавиться от проверки на каждом шаге, есть ли дуга с соответствующей пометкой, можно добавить еще одно состояние *F* (НЕУДАЧА) и добавлять все необходимые дуги от всех состояний к *F*. Добавляется также дуга, помеченная всеми литерами из *F* в *F*.

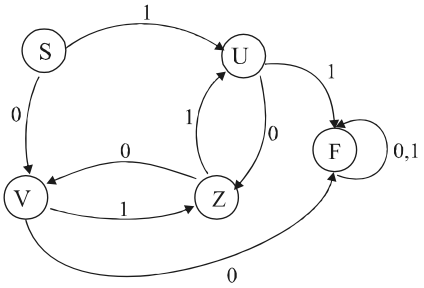

Рис. 3. Диаграмма состояний КА с состоянием *F* (НЕУДАЧА)

Детерминированный конечный автомат

Детерминированный автомат с конечным числом состояний (КА) – это пятерка (K, VT, M, S, Z).

1) K – алфавит элементов, называемых состояниями;

2) VT – входной алфавит (литеры, которые могут встретиться в цепочке или предложении);

3) M – отображение (или функция) множества $K \times VT$ в множестве K (если $M(Q, T) = R$, то это значит, что из состояния Q при входной литере T происходит переключение в состояние R);

4) $S \in K$ – начальное состояние;

5) Z – множество заключительных состояний, $Z \subset K$.

Говорят, что КА допускает цепочку t (цепочка считается допускаемой), если $M(S, t) = P$, где $P \in Z$.

Такие автоматы называются детерминированными, так как на каждом шаге входная литера однозначно определяет следующие текущее состояние.

Пример

Рассмотренной ранее диаграмме состояний соответствует КА
$M(S, 0) = V$, $M(U, 0) = Z$, $M(F, 0) = F$, $M(S, 1) = U$,
$M(V, 0) = F$, $M(Z, 0) = V$, $M(F, 1) = F$, $M(V, 1) = Z$,
$M(Z, 1) = 0$, $M(V, 1) = F$.

Если предложение x принадлежит грамматике G , то оно также допускается КА, соответствующим грамматике G. Для любого КА существует грамматика G, порождающая только те предложения, которые являются цепочками, допускаемыми КА.

Представление в ЭВМ

КА с состояниями $S_1S_2\ldots S_n$ и входными литерами $T_1T_2\ldots T_m$ можно представить матрицей B, состоящей из $n \times m$ элементов. Элемент $B[i, j]$ содержит число k – номер состояния S_k такого, что $M(S_i, T_j) = S_k$. Можно условиться, что состояние S_1 – начальное, а список заключительных состояний представлен вектором. Такая матрица называется матрицей переходов.

Другим способом представления может быть списочная структура. Представление каждого состояния с k дугами, исходящими из него, занимает $2*k + 2$ слов. Первое слово – имя состояния, второе – значение k, каждая последующая пара слов содержит терминальный символ

из входного алфавита и указатель на начало представления состояния, в которое надо перейти по этому символу.

Недетерминированный КА

Если грамматика содержит два правила с одинаковыми правыми частями, то отображение M оказывается неоднозначным. Автомат, построенный по такой диаграмме, называется *недетерминированным* конечным автоматом и определяется следующим образом.

Недетерминированным КА (НКА) называется пятерка (K, V_T, M, S, Z):

1) K – алфавит состояний;
2) V_T – входной алфавит;
3) M – отображение $K \times V_T$ в множество множества K;
4) $S \subseteq K$ – множество начальных состояний;
5) $Z \subseteq K$ – множество заключительных состояний.

Отличия:

1) отображение M дает не единственное, а (возможно пустое) множество состояний;
2) может быть несколько начальных состояний.

Пример

Рассмотрим регулярную грамматику $G[Z]$:

Z :: = $U1 \mid V0 \mid Z0 \mid Z1$
U :: = $Q1 \mid 1$
V :: = $Q0 \mid 0$
Q :: = $Q0 \mid Q1 \mid 0 \mid 1$

Диаграмма состояний и ее НКА имеют вид, показанный на рис. 4.

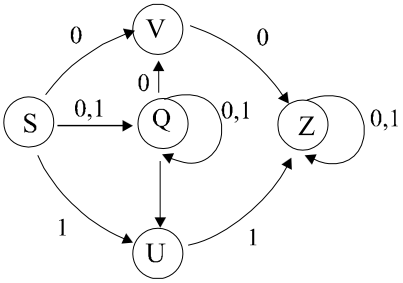

Рис. 4. Диаграмма состояний НКА

НКА $F = (\{S, Q, V, U, Z\}, \{0, 1\}, M, \{S\}, \{Z\})$
$M(S, 0)=\{V, Q\}$, $M(S, 1)=\{U, Q\}$, $M(V, 0) = \{Z\}$, $M(V,1) = \varnothing$, . . .
Состояние НЕУДАЧА представлено подмножеством $\varnothing$.

Построение КА из НКА

Покажем способ построения КА из НКА, при котором параллельно проверяются все возможные пути разбора и отбрасываются тупиковые. Если в НКА имеется, к примеру, выбор из трех состояний *x, y, z*, то в КА будет одно состояние [*xyz*], которое представляет все три.

Пусть НКА $F = (K, V_T, M, S, Z)$ допускает множество цепочек L. Определим КА $F' = (K', V_T, M', S', Z')$ следующим образом.

1. Алфавит состояний K' состоит из всех подмножеств множества. Обозначим элемент множества K' через $[s_1 s_2 \ldots s_i]$, где $s_1, s_2, \ldots s_i \in K$. Положим, что $\{s_1, s_2\}=\{s_2, s_1\} = [s_1 s_2]$.

2. Множества входных литер V_T для F и F' одни и те же.

3. Отображение M' определим так: если $M(\{s_1, s_2, \ldots s_i\}, T) = \{r_1, r_2, \ldots r_j\}$, то $M'([s_1 s_2 \ldots s_i], T)= [r_1 r_2 \ldots r_j]$.

4. Пусть $S = \{s_1, s_2, \ldots s_n\}$, тогда $S' =[s_1 s_2 \ldots s_n]$.

5. Пусть $Z = \{z_1, z_2, \ldots z_l\}$, тогда $Z' =[z_1 z_2 \ldots z_l]$.

Утверждается, что F и F' допускают одно и тоже множество цепочек.

Пример

На рис. 5 показана диаграмма состояний НКА. Начальное состояние *S*. Заключительное состояние *Z*.

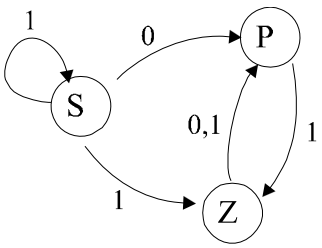

Рис. 5. Диаграмма исходного НКА

Действуя согласно приведенному выше алгоритму, получим КА (рис. 6). Начальное состояние *S* .Заключительные состояния *Z*, *PZ*, *SZ*, *SPZ*.

Состояния *PZ* и *PS* можно исключить, так как нет путей, к ним ведущих. Построенный автомат не является минимальным, можно построить автомат с меньшим числом состояний. Для построения минимального КА из НКА воспользуемся следующим приемом.

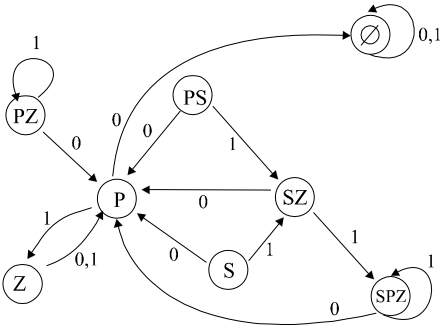

Рис. 6. Полученный по алгоритму КА

Строим матрицу переходов для НКА.

	0	1
S	*P*	*S,Z*
P		*Z*
Z	*P*	*P*

Поскольку из *S* по 1 можно попасть и в *S*, и в *Z*, вводим новое состояние *SZ*.

	0	1
S	*P*	*SZ*
SZ	*P*	*SZ*, *P*
P		*Z*
Z	*P*	*P*

Поскольку из *SZ* по 1 можно попасть и в *SZ*, и в *P*, вводим еще одно новое состояние *SPZ*.

	0	1
S	*P*	*SZ*
SZ	*P*	*SPZ*
SZP, *SPZ*	*P*	*SPZ*
P		*Z*
Z	*P*	*P*

Больше неоднозначных переходов нет, диаграмма состояний КА показана на рис. 7.

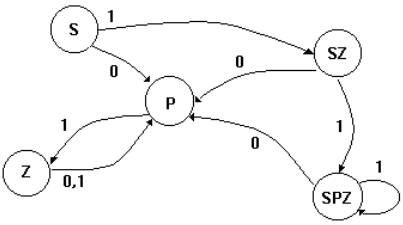

Рис. 7. Минимальный КА

ЗАДАНИЕ

В соответствии с выбранным вариантом задания к лабораторным работам разработать и реализовать лексический анализатор на основе детерминированных конечных автоматов. Исходными данными для сканера является программа на языке C++ и постоянные таблицы, реализованные в лабораторной работе № 1. Результатом работы сканера является создание файла токенов, переменных таблиц (таблицы символов и таблицы констант) и файла сообщений об ошибках.

СОДЕРЖАНИЕ ОТЧЕТА

- Цели и задачи проекта.
- Вид, структура входных и выходных данных.
- Построение детерминированного конечного автомата – основы сканера.
- Алгоритм.
- Тексты программы сканера на языке высокого уровня.
- Тестовые примеры.

КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Интерпретаторы и компиляторы.
2. Основные фазы трансляции.
3. Регулярные выражения и конечные автоматы.

4. Детерминированные конечные автоматы.
5. Методы приведения недетерминированного конечного автомата к детерминированному.
6. Автоматная грамматика.
7. Диаграмма состояний сканера.

ЛАБОРАТОРНАЯ РАБОТА № 3

РАЗРАБОТКА И РЕАЛИЗАЦИЯ БЛОКА СИНТАКСИЧЕСКОГО АНАЛИЗА

ЦЕЛЬ РАБОТЫ

Изучить табличные методы синтаксического анализа. Получить представление о методах диагностики и исправления синтаксических ошибок. Научиться проектировать синтаксический анализатор на основе табличных методов.

МЕТОДИЧЕСКИЕ УКАЗАНИЯ

Разбор или синтаксический анализ включает группирование токенов исходной программы в грамматические фразы, используемые компилятором для синтеза вывода. Обычно грамматические фразы исходной программы представляются в виде дерева. Иерархическая структура программы выражается рекурсивными правилами. Например, если в языке определены только операции ‘+’ и ‘*’, при определении выражения можно придерживаться следующих правил:

1) любой идентификатор есть выражение;
2) любое число есть выражение;
3) если e_1 и e_2 – выражения, то выражениями являются e_1+e_2, e_1*e_2, (e_1).

Точно так же многие языки программирования рекурсивно определяют инструкции языка правилами, аналогичными следующим:

1) если i_1 – идентификатор, e_2 – выражение, то $i_1 = e_2$ есть инструкция;
2) если e_1 – выражение, а st_2 – инструкция, то while (e_1) st_2; if (e_1) st_2; являются инструкциями.

Дерево разбора наглядно показывает, как начальный символ грамматики порождает строку языка. Формально для контекстно-свободной грамматики дерево разбора представляет собой структуру со следующими свойствами.

1. Корень дерева помечен начальным символом.
2. Каждый лист помечен терминальным символом грамматики.
3. Каждый внутренний узел представляет нетерминальный символ.
4. Если A является нетерминальным символом и помечает некоторый внутренний узел, а X_1, X_2, ... X_n – отметки его дочерних узлов, перечисленные слева направо, то $A \rightarrow X_1\ X_2\ ...\ X_n$ – продукция (правило вывода). Здесь X_1, X_2, ... X_n могут представлять собой как терминальные, так и нетерминальные символы.
5. Все листья дерева, прочитанные слева направо, образуют файл токенов.

По результатам разбора для каждого оператора исходной программы можно построить **синтаксическое дерево**, удовлетворяющее следующим требованиям:

1) ключевые слова и знаки операций являются корнями непустых поддеревьев;

2) идентификаторы и константы являются листьями.

Концевой (постфиксный) обход синтаксического дерева позволяет получить постфиксную (обратную польскую) запись.

Пример

Построим дерево разбора, синтаксическое дерево и постфиксную запись для оператора «i=j+2;». Порождающая грамматика содержит правила:

S→id=s1
s1→s2+s1
s1→s2
s2→id
s2→const
id→ идентификатор
const→ константа.

Терминалами являются «идентификатор», «константа», «+», «=» и «;», представленные соответствующими токенами. Начальный символ грамматики – S. Результаты проектирования представлены на рис. 8 и 9.

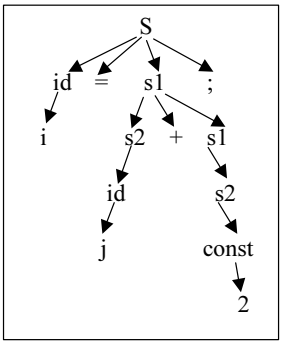

Рис. 8. Дерево разбора

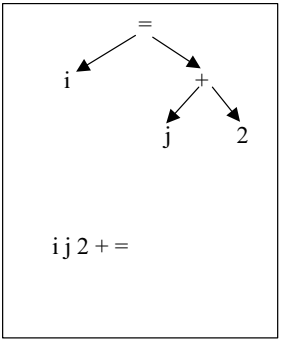

Рис. 9. Синтаксическое дерево и постфиксная запись

Семантический анализ имеет цель: проверка правильности описания типов объектов программы и корректное их использование в инструкциях. При семантическом анализе используются иерархические структуры, полученные во время синтаксического анализа, для идентификации операторов и операндов выражений и конструкций. Важным аспектом семантического анализа является проверка операторов на использование операндов допустимого спецификациями языка типов. Например, определение многих языков программирования требует, чтобы при использовании действительного числа в качестве индекса элемента массива генерировалось сообщение об ошибке. В то же время спецификация языка может позволить определенное насильственное преобразование типов, например, когда бинарный арифметический оператор применяется к операндам целого и действительного типов. В этом случае компилятору может потребоваться преобразование целого числа в действительное. При использовании метода операторного предшествования создаются семантические подпрограммы, учитывающие атрибуты лексем, с целью устранения неоднозначности разбора.

ЗАДАНИЕ

В соответствии с выбранным вариантом заданий к лабораторным работам реализовать синтаксический анализатор **с использованием одного из табличных методов** (*LL*-, *LR*-метод, метод предшествования).

Этапы проектирования синтаксического анализатора:

1) сконструировать КС-грамматику в соответствии с вариантом задания;

2) в случае несоответствия построенной грамматики требованиям выбранного табличного метода разбора следует провести эквивалентные преобразования грамматики либо выбрать другой метод разбора;

3) построить таблицу разбора и запрограммировать драйвер, реализующий работу с этой таблицей.

Исходные данные – файл токенов, таблицы лексем.

Результатом работы синтаксического анализатора является:

- синтаксическое дерево или постфиксная запись;
- файл сообщений об ошибках. В лабораторной работе необходимо реализовать возможности табличного метода по диагностике и исправлению синтаксических ошибок в исходной программе.

СОДЕРЖАНИЕ ОТЧЕТА

- Цели и задачи проекта.
- Вид, структура входных и выходных данных.
- Выбор стратегии разбора.
- Классификация грамматики входного языка, определение необходимости преобразования грамматики к требуемому виду; при необходимости выполнить эти преобразования.
- Схема разбора.
- Таблица разбора.
- Тексты программы анализатора на языке высокого уровня.
- Тестовые примеры.

КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Нисходящая стратегия разбора.
2. Восходящая стратегия разбора.
3. Синтаксические деревья.
4. Классификация языков по Хомскому.
5. *LL*-грамматика.
6. *LL*-таблица разбора.
7. *LR*-грамматика.
8. *LR*-таблица разбора.

ЛАБОРАТОРНАЯ РАБОТА № 4

РАЗРАБОТКА И РЕАЛИЗАЦИЯ БЛОКА ГЕНЕРАЦИИ КОДА

ЦЕЛЬ РАБОТЫ

Изучить методы генерации кода с учетом различных промежуточных форм представления программы. Изучить методы управления памятью и особенности их использования на этапе генерации кода.

Научиться проектировать генератор кода.

МЕТОДИЧЕСКИЕ УКАЗАНИЯ

Последней фазой процесса компиляции является генерация кода. Генератор кода получает на вход промежуточное представление исходной программы и выводит эквивалентную целевую программу, которая должна быть эффективной с точки зрения используемых ресурсов и времени выполнения. Математическая проблема генерации оптимального кода является неразрешимой. На практике вынуждены довольствоваться эвристическими технологиями, дающими хороший, но необязательно оптимальный код. Тщательно разработанный алгоритм генерации кода может давать код, работающий в несколько раз быстрее кода, полученного с помощью недостаточно продуманного алгоритма. Хотя некоторые детали генератора кода зависят от целевой машины и операционной системы, такие вопросы, как управление памятью, выбор инструкций, распределение регистров и порядок вычислений, свойственны практически всем задачам, связанным с генерацией кода.

Входной поток генератора кода – это промежуточное представление исходной программы, полученное на начальных стадиях компиляции, вместе с таблицей символов, которая используется для определения адресов времени исполнения объектов данных, обозначаемых в промежуточном представлении токенами. Имеется несколько видов промежуточного представления исходной программы: линейное пред-

ставление (постфиксная запись), графическое представление (синтаксическое дерево), виртуальное машинное представление (код стековой машины), трехадресное представление («четверка» – конструкция, содержащая первый операнд, второй операнд, результат и код операции).

Сейчас классическим стал метод трансляции выражений, основанный на использовании промежуточной обратной польской записи (ОПЗ). Однако в существующих трансляторах используются и другие методы.

Рассмотрим сущность обратной польской записи на примере. Простое арифметическое выражение с вещественными переменными

$$a + b \times c - d/(a + b)$$

можно графически представить в виде дерева (рис. 10). Узлы дерева соответствуют операции, а ветви – операндам. Левая ветвь, исходящая из узла, отвечает левому операнду, а правая – правому. В каждой ветви операциям, которые выполняются раньше, соответствуют нижележащие узлы. Верхний узел (корень дерева) отвечает операции, которая выполняется последней. С него начинается построение дерева.

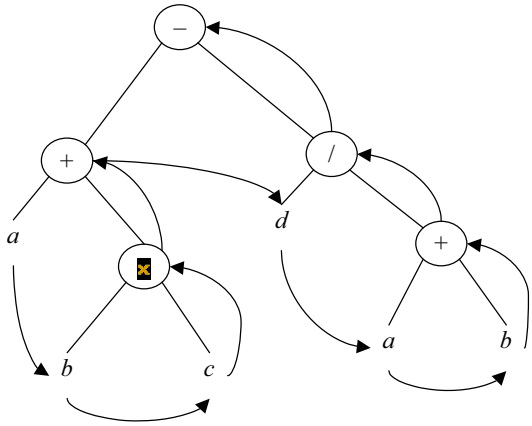

Рис. 10. Порядок обхода дерева простого арифметического выражения для получения ОПЗ

Если, начав с нижнего листа самой левой ветви дерева, обойти все листья и узлы дерева так, чтобы ветви рассматривались слева на право, а узел рассматривался только после обхода всех исходящих из него ветвей, как показано стрелками на рис. 10, то последовательность просмотра листьев и узлов дает ОПЗ этого выражения:

$$abc \times + dab + / -.$$

Эту бесскобочную запись называют также постфиксной записью, потому что знак каждой операции записан после соответствующих операндов. Заметим, что в ОПЗ операнды располагаются в том же порядке, что и в исходном выражении, а знаки операций при просмотре записи слева направо встречаются в том порядке, в котором нужно выполнять соответствующие действия. Отсюда вытекает основное преимущество ОПЗ перед обычной записью выражений со скобками: выражение можно вычислить в процессе однократного просмотра слева направо.

Правило вычисления выражения в ОПЗ состоит в следующем. ОПЗ просматривается слева направо. Если рассматриваемый элемент – операнд, то рассматривается следующий элемент. Если рассматриваемый элемент – знак операции, то выполняется эта операция над операндами, записанными левее знака операции. Результат операции записывается вместо первого (самого левого) операнда, учувствовавшего в операции. Остальные элементы (операнды и знак операции), участвовавшие в операции, вычеркиваются из записи. Просмотр продолжается.

В результате последовательного выполнения этого правила будут выполнены все операции, имеющиеся в выражении, запись сократится до одного элемента – результата вычисления выражения.

Выполнение правила для нашего примера приводит к последовательности строк, записанных во втором столбце табл. 1. Рассматриваемый на каждом шаге процесса элемент строки отмечен «крышкой». В третьем столбце табл. 1 записаны соответствующие действия, а в четвертом столбце – эквивалентные команды трехадресной машины. Результат выполнения операции фиксируется в виде рабочей переменной вида r_j. После очередной операции рабочая переменная r_1 или r_2 вычеркивается, освободившуюся рабочую переменную можно использовать вновь для записи результата операции. Использование каждый раз свободной рабочей переменной с минимальным номером экономит количество занятых рабочих переменных.

Таблица 1

Пример вычисления выражения в ОПЗ

№	Состояние строки	Действие	Машинная команда
1	$\hat{a}bc\times dab+/-$	Посмотреть следующий элемент	------
2	$a\hat{b}c\times+dab+/-$	Посмотреть следующий элемент	------
3	$ab\hat{c}\times+dab+/-$	Посмотреть следующий элемент	------
4	$abc\hat{\times}+dab+/-$	$r_1 := b\times c$	$\times bcr_1$
5	$ar_1\hat{+}dab+/-$	$r_1 := a+r_1$	$+ar_1r_1$
6	$r_1\hat{d}ab+/-$	Посмотреть следующий элемент	------
7	$r_1d\hat{a}b+/-$	Посмотреть следующий элемент	------
8	$r_1da\hat{b}+/-$	Посмотреть следующий элемент	------
9	$r_1dab\hat{+}/-$	$r_2 := a+b$	$+abr_2$
10	$r_1dr_2/-$	$r_2 := d/r_1$	$/dr_2r_2$
11	$r_1r_2\hat{-}$	$r_1 := r_1-r_2$	$-r_1r_2r_1$
12	r_1	------	------

Алгоритм перевода ОПЗ в машинные команды

Последовательность машинных команд в табл. 1 есть, по существу, результат трансляции выражения, записанного в обратной польской записи, в машинные команды. Если для каждого операнда, включая рабочие переменные r_j, известен адрес, то для получения окончательных машинных команд остается лишь заменить знаки операций машинными кодами операции, а операнды – адресами.

Для трансляции выражений из ОПЗ в машинные команды можно использовать стек операндов (СО) с указателем i. В исходном состоянии стек операндов пуст, а указатель $i = 1$. Будем также считать, что в исходном состоянии номер первой свободной рабочей переменной $j = 1$.

Алгоритм трансляции состоит в следующем.

1. Взять очередной символ S из ОПЗ выражения.
2. Если S – операнд, то занести S в СО[i], выполнить $i := i+1$ и перейти к пункту 1, иначе перейти к пункту 3.

3. Если S – не знак операции, то перейти к пункту 4, иначе, если S – знак операции R, выполнить следующее:

а) среди элементов стека CO[i–k], … , CO[i–1], где k – число операндов операции R, найти рабочую переменную с минимальным номером l. Если в рассматриваемых элементах стека нет рабочих переменных, то положить $l = j$;

б) записать машинные команды, реализующие оператор присваивания

$$r_l := R(\text{CO}[i–k], \ldots , \text{Co}[i–1]).$$

Здесь $R(x_1, \ldots , x_k)$ – результат выполнения операции R над операндами $x_1, \ldots , x_k$;

в) занести символ r_1 в CO [$I – k$];

г) выполнить $i := I – k + 1$ и $j := l + 1$.

Перейти к пункту 1.

4. Если запись выражения исчерпана, то трансляция закончена. Стек операндов должен содержать только переменную r_1, а в противном случае нужно записать информацию об ошибке в таблицу ошибок.

Для реализации пункта 3б приведенного алгоритма используются заранее подготовленные «заготовки» групп машинных команд, в которые требуется лишь подставить адреса операндов (или значения самоопределенных операндов), взятые из стека операндов. Эту подстановку выполняет подпрограмма, соответствующая рассматриваемой операции R.

Надо, однако, отметить, что используемая подпрограмма определяется не только знаком операции R, но и типом операндов. Например, одна подпрограмма соответствует операции сложения вещественных чисел, а другая – операции сложения целых. Иногда в пункте 3б приходится выполнять несколько подпрограмм. В частности, если один операнд целый, а другой вещественный, то вначале нужно привести операнды к одному типу, а затем выполнить подпрограмму формирования команды сложения. При несовместимости операндов, например, в операции сложения один операнд вещественный, а другой булевский, или при несоответствии операндов знаку операции выдается информация об ошибке.

Описанный алгоритм пригоден для перевода в машинные команды не только арифметических и логических выражений, но и любых текстов, записанных в ОПЗ с использованием произвольных операций, реализуемых машинными командами.

Поскольку существует относительно несложный алгоритм перевода ОПЗ в машинные команды, для полного решения задачи трансляции выражений достаточно перевести выражения в ОПЗ.

Метод Е.В. Дикстры основан на использовании стека с приоритетами, позволяющего изменить порядок следования знаков операции в выражении так, что получается ОПЗ. Простейший вариант этого метода применим только к простым арифметическим и логическим выражениям, содержащим простые переменные, знаки арифметических и логических операций, знаки операций отношения и круглые скобки. Каждому ограничителю, входящему в выражения, присваивается приоритет (табл. 2). Для знаков операций приоритеты возрастают в порядке, обратном старшинству операций.

Таблица 2

Приоритеты ограничителей

Ограничители	Приоритеты
([	0
=)] , ;	1
∃	2
||	3
&&	4
!	5
> ≥ = ≠ ≤ <	6
+ –	7
× % / *унарный минус*	8

Арифметическое или логическое выражение рассматривается как входная строка символов, которая просматривается слева направо. Операнды переписываются в выходную строку, а знак операций помещаются сначала в стек операций (рис. 11).

Если приоритет входного знака равен нулю или больше приоритета знака, находящегося в вершине стека, то новый знак добавляется к вершине стека. В противном случае из стека «выталкивается» и переписывается в выходную строку знак, находящийся в вершине, а также

следующие за ним знаки с приоритетами, большими приоритета входного знака или равными ему. После этого входной знак добавляется к вершине стека. Особенности имеет лишь обработка скобок. Открывающая круглая скобка, имеющая «особый» приоритет нуль, просто записывается в вершину стека и не выталкивает ни одного знака. Появление закрывающей скобки вызывает выталкивание всех знаков до ближайшей открывающей скобки включительно. В стек закрывающая скобка не записывается. Открывающая и закрывающая скобки как бы взаимно уничтожаются и в выходную строку не переносятся. После просмотра всех символов входной строки происходит выталкивание всех оставшихся в стеке символов и дописывание их к выходной строке.

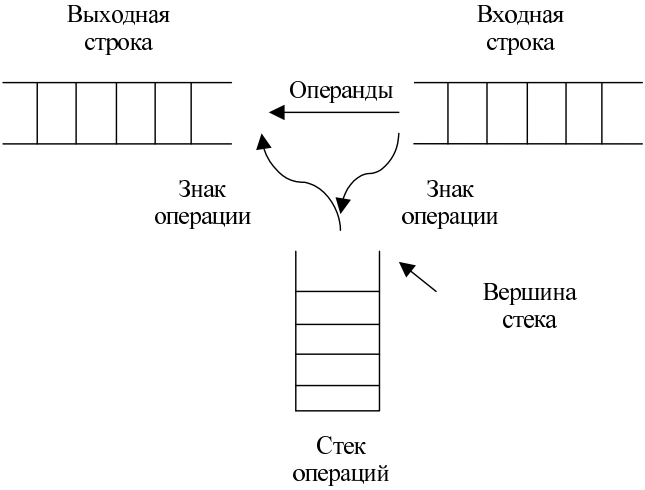

Рис. 11. Использование стека операций для перевода выражений в ОПЗ

Пример

Перевести в ОПЗ выражение $a + b \times c - d/(a + b)$. Решение показано в табл. 3. Окончательная выходная строка совпадает с ОПЗ того же выражения, полученной обходом дерева, изображенного на рис. 10.

Таблица 3

Перевод в ОПЗ арифметического выражения

Выходная строка	a	a	a	a	a	a	a	a	a	a	a	a	a	a
			b	b	b	b	b	b	b	b	b	b	b	b
					c	c	c	c	c	c	c	c	c	c
						×	×	×	×	×	×	×	×	×
						+	+	+	+	+	+	+	+	+
							d	d	d	d	d	d	d	d
										a	a	a	a	a
												b	b	b
													+	+
														/
														–
Стек											+	+		
									(	(	(	(		
				×	×			/	/	/	/	/	/	
		+	+	+	+	–	–	–	–	–	–	–	–	
	a	+	b	×	c	–	d	/	(	a	+	b	)	
	Входная строка													

Переменные с индексами

Пусть требуется вычислить выражение

$$(a + b[i + 1, j]) \times c + d.$$

Для выполнения вычислений на машине необходимо сначала найти адрес переменной с индексами. Введем операцию АДРЕС ЭЛЕМЕНТА МАССИВА (АЭМ), результат выполнения которой – адрес элемента массива (точнее, назначаемый ему адрес) и значение индексных выражений. Тогда рассматриваемое выражение можно представить деревом, показанным на рис. 12. ОПЗ, полученная обходом дерева, имеет вид

$$abi1 + jAЭМ + c \times d + .$$

Как обычно, в ОПЗ левее операции АЭМ расположены операнды, количество которых в операции АЭМ зависит от размерности массива. Это вынуждает вместе со знаком операции АЭМ явным образом задавать количество операндов. Будем обозначать операцию АЭМ парой символов $k]$, где k – целое число, равное количеству операндов, а символ] – символ закрывающей индексной скобки, используемый в качестве знака операции АЭМ. Очевидно, если n – число индексов, то $k = n + 1$. Используя новое обозначение операции АЭМ, ОПЗ можно переписать в виде

$$abi1 + j3] + c \times d + .$$

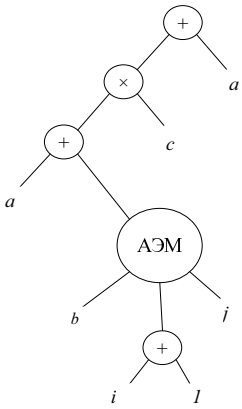

Рис. 12. Дерево выражения, содержащего переменную с индексами

Для построения ОПЗ также можно использовать алгоритм Дикстры.

Оператор присваивания

Требования к величине приоритета знака «=»:

- приоритет знака присваивания должен быть меньше приоритета знака любой арифметической и логической операции, поскольку операция присваивания выполняется после вычисления выражения, записанного в правой части оператора присваивания;
- приоритет знака присваивания должен быть больше приоритета знака конца оператора (точка с запятой), чтобы знак конца оператора очищал стек.

Пример

Оператор $a = b + c$; представляется деревом, изображенным на рис. 13. Обход дерева дает ОПЗ $abc + =$.

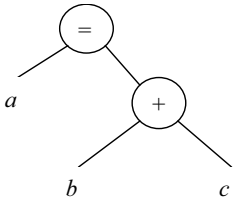

Рис. 13. Графическое изображение оператора присваивания

Условный оператор

Введем две операции:

- условный переход по значению «ложь» (УПЛ);
- безусловный переход (БП).

Операция УПЛ имеет два операнда: логическое выражение и метка. Если логическое выражение истинно, то операция УПЛ пропускается, а если ложно, то происходит переход на метку. Ветвь динамического дерева с узлом УПЛ показана на рис. 14. У операции БП имеется лишь один операнд – метка (рис. 15). Результат операции БП – переход на метку.

Условный оператор вида

if (*A*) *B* **else** *C*,

где *A* – логическое выражение; *B* – оператор; *C* – оператор, можно представить в виде динамического дерева (рис. 16).

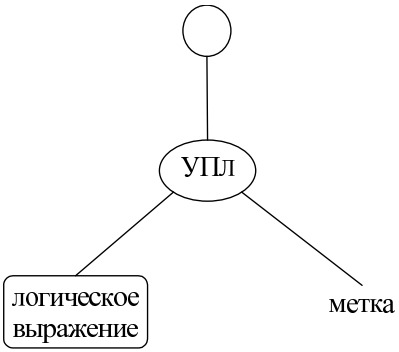

Рис. 14. Графическое изображение

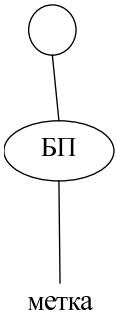

Рис. 15. Графическое изображение операции БП операции УПЛ

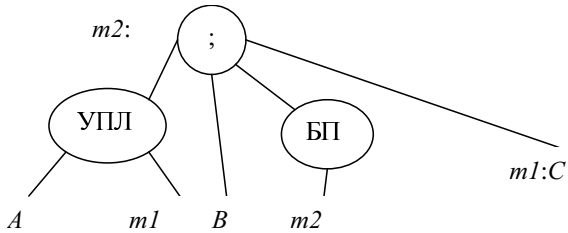

Рис. 16. Дерево, изображающее условный оператор

В изображении дерева знак «;» не является знаком операции. Отвечающий ему узел играет роль пустого узла и служит для объединения отдельных ветвей. В обратную польскую запись знак «;» можно не переносить. Обход дерева дает обратную польскую запись: $A\ m_1$ УПЛ $B\ m_2$ БП m_1: С m_2:.

ОПЗ оператора **if (*A*) *B*;** имеет вид : $A\ m_1$ УПЛ $B\ m_1$:.

Операторы цикла

Оператор цикла может быть заменен на соответствующую последовательность операторов присваивания, условных операторов и операторов безусловного перехода, для которой и строится ОПЗ.

Оператор цикла с предусловием **while (*A*) *B*;**
можно заменить последовательностью операторов

$$m_1\text{: if(!}A\text{) goto } m_2\text{; } B\text{; goto } m_1\text{; } m_2\text{:,}$$

для которой ОПЗ выглядит следующим образом:

$$m_1\text{: } A\ m_2 \text{ УПЛ } B\ m_1 \text{ БП } m_2\text{:}$$

Пример

$$\text{while } (a < 0)\ a = a + 1;$$

ОПЗ имеет вид m_1: $a\ 0 < m_2$ УПЛ $a\ a\ 1 + = m_1$ БП m_2:

Оператор цикла с постусловием: **do *B* while (*A*);**
можно заменить последовательностью операторов

$$m_1\text{: } B\text{; if}(A)\text{ goto } m_1\text{;,}$$

для которой ОПЗ выглядит следующим образом:

$$m_1\text{: } B\ A\ m_2 \text{ УПЛ } m_1 \text{ БП } m_2\text{: или } m_1\text{: } B\ A\ !\ m_1 \text{ УПЛ,}$$

где ! – унарная операция логического отрицания.

Пример

$$\text{do } \{\ x = y * 2;\ y = y - 1;\}\ \text{while } (y > 0);$$

ОПЗ имеет вид: m_1: $x\ y\ 2\ *\ = y\ y\ 1\ -\ = y\ 0 >\ !\ m_1$ УПЛ

Оператор цикла со счетчиком: **for (*A*; *B*; *C*) *D*;**
можно заменить последовательностью операторов

A; while (*B*) { *D*; *C*; } или *A*; m_1: if(!*B*) goto m_2; *D*; *C*; goto m_1; m_2: ,

для которой ОПЗ выглядит следующим образом:

$$A\ m_1{:}\ B\ m_2\ \text{УПЛ}\ D\ C\ m_1\ \text{БП}\ m_2{:}$$

Пример

$$\text{for } (a = 0,\ x = 1;\quad a < n;\quad a = a + 1)\ x = x{*}a;$$

ОПЗ имеет вид $a\ 0 = x\ 1 = m_1{:}\ a\ n <$ УПЛ $x\ x\ a\ {*} = a\ a\ 1 + =$

Выходом генератора кода является целевая программа. Получение в качестве выхода генератора кода программы на языке Ассемблер несколько облегчает процесс генерации кода: можно создавать символьные конструкции и использовать возможности макросов Ассемблера. Плата за эту простоту – дополнительный шаг обработки ассемблерной программы.

Отображение имен исходной программы в адресах объектов данных в памяти во время работы программы выполняется совместно начальными стадиями компилятора и генератором кода. Имя в «четверке» ссылается на соответствующую запись в таблице символов. Записи в таблице символов создавались при рассмотрении объявлений. Тип, используемый в объявлении, определяет размер (количество памяти), необходимый для объявленной переменной. По информации из таблицы символов можно определить относительный адрес имени в области данных процедуры. Имена в промежуточном представлении могут быть преобразованы в адреса в целевом коде реализацией статического и/или стекового распределения областей данных. Генератор кода для каждой лексемы таблицы символов выделяет память в соответствии с типом (необходимым размером памяти) данной переменной.

При генерации машинного кода метку в «четверке» преобразуют в адрес инструкции. Метки представляют собой номера «четверок» в массиве. Поскольку мы сканируем «четверки» по очереди, можно вывести положение первой машинной инструкции, генерируемой данной «четверкой», подсчитывая количество слов, использованных для уже сгенерированного кода.

Набор инструкций целевой машины определяет сложность их выбора. Если целевая машина не поддерживает все типы данных единообразно, то каждое исключение из общего правила потребует специальной

обработки. Для каждого типа «четверок» можно разработать шаблон целевого кода. При этом не всегда выполняется условие оптимальности. Целью семантического анализа на данном этапе является проверка типов операндов и выбор соответствующего шаблона целевого кода.

Инструкции, использующие в качестве операндов регистры, обычно короче и быстрее выполняются, чем инструкции, работающие с операндами, расположенными в памяти. Использование регистров можно разделить на две подзадачи:

1) выбирается множество переменных, которые будут находиться в регистрах в некоторой точке программы;

2) выбираются конкретные регистры для размещения в них переменных.

Поиск оптимального назначения регистров переменным представляет собой трудную задачу, усложняемую тем, что аппаратное обеспечение и/или операционная система могут накладывать дополнительные ограничения по использованию регистров.

Порядок, в котором выполняются вычисления, может существенно влиять на эффективность целевого кода. Изменение порядка вычислений может привести к уменьшению количества необходимых для работы регистров. Выбор оптимального порядка вычислений является NP-полной математической задачей. Можно избежать решения этой задачи, генерируя целевой код для «четверок» в том порядке, в каком они были созданы генератором промежуточного кода (синтаксическим анализатором).

Информация, необходимая в процессе выполнения процедуры, хранится в блоке памяти, именуемом записью активации. Память для локальных имен процедуры также выделяется в ее записи активации. При статическом распределении памяти положение записи активации фиксируется во время компиляции. В случае стекового распределения новая запись активации вносится в стек для каждого выполнения процедуры. Запись снимается со стека по окончании активации.

ЗАДАНИЕ

В соответствии с выбранным вариантом реализовать генератор кода. Исходными данными являются:

- синтаксическое дерево или постфиксная запись, построенные в лабораторной работе № 3;
- таблицы лексем.

Результатом выполнения лабораторной работы является программа на языке Ассемблер, разработанная на основе знаний и практических навыков, полученных при изучении курса «Языки программирования и методы трансляции (часть I)».

В режиме отладки продемонстрировать работоспособность генератора кода и транслятора в целом.

СОДЕРЖАНИЕ ОТЧЕТА

- Цели и задачи проекта.
- Вид, структура входных и выходных данных.
- Выбор промежуточной формы хранения данных (программы).
- Тексты программы генератора.
- Тестовые примеры.

КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Внутренние формы представления программы.
2. Постфиксная запись.
3. Представление генерируемого кода в форме четверок.
4. Организация генератора кода.
5. Среда выполнения.
6. Методы управления памятью.
7. Фазы управления памятью.

ВАРИАНТЫ ЗАДАНИЙ К ЛАБОРАТОРНЫМ РАБОТАМ

1. Подмножество языка C++ включает:
   - данные типа **int**;
   - инструкции описания переменных;
   - операторы присваивания, **if**, **if- else** любой вложенности и в любой последовательности;
   - операции **+, – , *, = =, != , <** .
2. Подмножество языка C++ включает:
   - данные типа **int**;
   - инструкции описания переменных;

• операторы присваивания, **while** любой вложенности и в любой последовательности;
• операции **+, – , < =, >= , < ,** >.

3. Подмножество языка С++ включает:
   • данные типа **int**;
   • инструкции описания переменных;
   • операторы присваивания, **do-while** любой вложенности и в любой последовательности;
   • операции **+=, – = , +, –, ==, !=** .

4. Подмножество языка С++ включает:
   • данные типа **int**;
   • инструкции описания переменных;
   • операторы присваивания, **for** любой вложенности и в любой последовательности;
   • операции **+, – , *, &&,** || .

5. Подмножество языка С++ включает:
   • данные типа **int**;
   • инструкции описания переменных;
   • операторы присваивания, **switch** любой вложенности и в любой последовательности;
   • операции **+, – , *, = =, != , <** .

6. Подмножество языка С++ включает:
   • данные типа **int, float, char**;
   • инструкции описания переменных;
   • операторы присваивания в любой последовательности;
   • операции **+, – , *, = =, != , <, >** .

7. Подмножество языка С++ включает:
   • данные типа **int, float, массивы** из элементов указанных типов;
   • инструкции описания переменных;
   • операторы присваивания в любой последовательности;
   • операции **+, – , *, = =, != , <, >** .

8. Подмножество языка С++ включает:
   • данные типа **int, float, struct** из элементов указанных типов;
   • инструкции описания переменных;
   • операторы присваивания в любой последовательности;
   • операции **+, – , *, = =, != , <, >** .

9. Подмножество языка C++ включает:
   - данные типа **int, char**;
   - инструкции описания переменных;
   - операторы присваивания в любой последовательности;
   - операции **+, – , <, >,** побитовые операции **<<, >>, &,** | .
10. Подмножество языка C++ включает:
   - данные типа **int**;
   - инструкции описания переменных;
   - операторы присваивания в любой последовательности;
   - полный набор арифметических, логических операций и операций сравнения.

11. Подмножество языка C++ включает:
   - данные типа **int**;
   - инструкции описания переменных и функций;
   - несколько функций с параметрами и возвращаемым значением;
   - операторы присваивания в любой последовательности;
   - операции **+, – , = =, != , <, >** .

РАСЧЕТНО-ГРАФИЧЕСКОЕ ЗАДАНИЕ

Расчетно-графическое задание выполняется студентами с четвертой учебной недели. Тему задания студент выбирает с учетом особенностей варианта лабораторных работ и по согласованию с преподавателем. Поскольку изучаемые в ходе выполнения РГЗ методы являются частью процесса трансляции, программа, реализующая эти методы, должна быть внедрена в программу компилятора, которую студент пишет в ходе выполнения лабораторных работ. Рекомендуемые сроки защиты РГЗ: во время защиты соответствующих лабораторных работ.

ТЕМЫ РАСЧЕТНО-ГРАФИЧЕСКОГО ЗАДАНИЯ

1. Прямые методы трансляции. Особенности реализации конструкций языков программирования с использованием прямых методов.
2. Методы оптимизации кода. Оптимизация вычислений с константами.
3. Методы оптимизации кода. Оптимизация выражений.
4. Методы оптимизации кода. Оптимизация циклов.

5. Методы диагностики и исправления ошибок. Лексические ошибки.

6. Методы диагностики и исправления ошибок. Синтаксические ошибки.

7. Методы диагностики и исправления ошибок. Ошибки в употреблении скобок.

8. Методы диагностики и исправления ошибок. Контекстно-зависимые ошибки.

9. Методы диагностики и исправления ошибок. Ошибки, связанные с употреблением различных типов данных.

10. Методы диагностики и исправления ошибок. ошибки выполнения: нахождение индекса массива вне области действия, целочисленное переполнение, попытка чтения за пределами файла.

11. Методы диагностики и исправления ошибок: ошибки, связанные с нарушением ограничений на размер программ, число элементов в таблице символов, размер стека разбора и пр.

12. Семантические действия.

13. Создание надежных компиляторов. Использование формального определения.

14. Создание надежных компиляторов. Модульное проектирование.

15. Создание надежных компиляторов. Проверка компилятора.

16. Анализатор, проверяющий принадлежность грамматики к LL(1)-типу.

17. Программа для построения таблиц синтаксического LL-разбора.

18. Программа для построения таблиц синтаксического LR-разбора.

19. Синтаксический анализатор на основе грамматики простого предшествования.

20. Синтаксический анализатор на основе грамматики расширенного предшествования.

21. Синтаксический анализатор на основе грамматики операторного предшествования.

22. Схемы управления памятью: статическое управление памятью.

23. Схемы управления памятью: стековое управление памятью.

Содержание РГЗ:

• используя конспект лекций и рекомендуемую научную литературу, изучить материал по заданной теме;

- написать реферат;
- программно реализовать исследованные методы, включив их в спроектированный транслятор;
- защитить работу.

Содержание отчета по РГЗ:
- реферат (5…10 м/п листов);
- описание реализованных алгоритмов;
- тексты программ;
- тестовые примеры.

СПИСОК ЛИТЕРАТУРЫ

1. *Карпов Ю.Г.* Основы построения трансляторов: учеб. пособие для вузов. – СПб.: БХВ-Петербург, 2005.
2. *Еланцева И.Л., Полетаева И.А.* Языки программирования и методы трансляции. Раздел «Методы трансляции». Конспект лекций. – Новосибирск: Изд-во НГТУ, 2007.
3. *Полетаева И.А.* Методы трансляции. – Новосибирск: Изд-во НГТУ, 1998.
4. *Ахо А., Сети Р., Ульман Дж.* Компиляторы. Принципы, технологии, инструменты. – М.: Вильямс, 2001. – 768 с.
5. *Хантер Р.* Проектирование и конструирование компиляторов. – М.: Финансы и статистика, 1984.
6. *Льюис Ф., Розенкранц Д., Стирна Р.* Теоретические основы проектирования компиляторов. – М.: Мир, 1979.
7. *Пратт Т.* Языки программирования: разработка и реализация. – М.: Мир, 1979.
8. *Лебедев В.Н.* Введение в системное программирование. – М.: Статистика, 1975.
9. *Маккиман У., Хорнинг Дж., Уортман Д.* Генератор компиляторов. – М.: Статистика, 1980.

ОГЛАВЛЕНИЕ

ЯЗЫКИ ПРОГРАММИРОВАНИЯ
И МЕТОДЫ ТРАНСЛЯЦИИ

Методические указания

Редактор *И.Л. Кескевич*
Выпускающий редактор *И.П. Брованова*
Корректор *Л.Н. Кинщт*
Компьютерная верстка *Н.М. Шуваева*

Подписано в печать 03.1.2011. Формат 60 × 84 1/16. Бумага офсетная
Тираж 150 экз. Уч.-изд. л. 2,55. Печ. л. 2,75. Изд. № 289. Заказ №
Цена договорная

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

№ 4056

004
Я 411

ЯЗЫКИ ПРОГРАММИРОВАНИЯ И МЕТОДЫ ТРАНСЛЯЦИИ

Методические указания

НОВОСИБИРСК
2011