ФЕДЕРАЛЬНОЕ АГЕНТСТВО СВЯЗИ
Северо-Кавказский филиал ордена Трудового Красного Знамени
федерального государственного бюджетного образовательного учреждения
высшего образования
«Московский технический университет связи и информатики»

С.А. ШВИДЧЕНКО

Методические указания
для проведения практических занятий
по дисциплине

«СИСТЕМНОЕ ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ»

Кафедра **«Информатика и вычислительная техника»**

Направление подготовки **09.03.01. Информатика и вычислительная техника**

Профили **Программное обеспечение и интеллектуальные системы**
Вычислительные машины, комплексы, системы и сети

Разработала:
Доцент кафедры ИВТ Швидченко С.А.

Ростов-на-Дону
2019

Методические указания
для проведения практических занятий
по дисциплине
«Системное программное обеспечение»

Составитель: Швидченко С.А., доц. каф. «ИВТ»

Рассмотрено и одобрено
на заседании кафедры «ИВТ»
Протокол от «26» августа 2019 г., № 1.

Практическое занятие 1. Управление файловой системой программными средствами.

Для подготовки и выполнения Практическое занятие №1 используется методическая разработка: Чикалов А.Н. «Системное программное обеспечение. Управление файловой системой программными средствами». Методические указания к практическим работам, размещенные на сайте СКФ МТУСИ в разделе «Методические материалы»: http://www.skf-mtusi.ru/?page_id=659

Контрольные вопросы (ПК-11):

1. Приведите определение файловой системы и файла.
2. Перечислите типы файлов и каталогов.
3. Каким образом осуществляется обращение к функциям DOS и BIOS?
4. Поясните назначение вектора прерываний, способ его использования.
5. Каким образом осуществляется обращение из прикладной программы к системным функциям?
6. Какие функции прерывания 21h DOS используются для работы с файлами?
7. Поясните атрибуты файла, где они задаются и какими средствами их можно просмотреть?
8. Перечислите основные операции, допустимые при работе с файлами, каталогами и дисками.
9. Поясните принципы построения "древовидной" структуры каталога. Какими средствами эта структура поддерживается на уровне системных данных?
10. Поясните алгоритм функционирования DOS при создании нового файла, сохранении, переименовании, удалении. Какие данные и в каких системных таблицах при этом изменяются?
11. Поясните процедуру обращения к файлу для чтения данных.
12. Какие данные (коды) в текстовом файле соответствуют клавише ENTER (размещаются в файле при ее нажатии)?
13. Какие типовые операции следует выполнить для записи данных в файл программным способом?
14. Что выполняется при реализации открытия файла?
15. Каким образом добавит фрагмент в текстовый файл программным способом?

Практическое занятие 2. Алгоритмическая реализация лексического анализатора.

Цель работы: изучение основных понятий теории регулярных грамматик, ознакомление с назначением и принципами работы лексических анализаторов (сканеров), получение практических навыков построения сканера на примере заданного простейшего входного языка.

Для выполнения лабораторной работы требуется написать программу, которая выполняет лексический анализ входного текста в соответствии с заданием и порождает таблицу лексем с указанием их типов и значений. Текст на входном языке задается в виде символьного (текстового) файла. Программа должна выдавать сообщения о наличие во входном тексте ошибок, которые могут быть обнаружены на этапе лексического анализа.

Длину идентификаторов и строковых констант считать ограниченной 32 символами. Программа должна допускать наличие комментариев неограниченной длины во входном файле. Форму организации комментариев предлагается выбрать самостоятельно.

Краткие теоретические сведения

Лексический анализатор (или сканер) - это часть компилятора, которая читает литеры программы на исходном языке и строит из них слова (лексемы) исходного языка. На вход лексического анализатора поступает текст исходной программы, а выходная информация передается для дальнейшей обработки компилятором на этапе синтаксического анализа и разбора.

С теоретической точки зрения лексический анализатор не является обязательной, необходимой частью компилятора. Его функции могут выполняться на этапе синтаксического разбора. Однако существует несколько причин, исходя из которых в состав практически всех компиляторов включают лексический анализ. Эти причины заключаются в следующем:

- упрощается работа с текстом исходной программы на этапе синтаксического разбора и сокращается объем обрабатываемой информации, так как лексический анализатор структурирует поступающий на вход исходный текст программы и выкидывает всю незначащую информацию;
- для выделения в тексте и разбора лексем возможно применять простую, эффективную и теоретически хорошо проработанную технику анализа, в то время как на этапе синтаксического анализа конструкций исходного языка используются достаточно сложные алгоритмы разбора;
- сканер отделяет сложный по конструкции синтаксический анализатор от работы непосредственно с текстом исходный программы, структура которого может варьироваться в зависимости от версии входного языка - при такой конструкции компилятора при переходе от одной версии языка к другой достаточно только перестроить относительно простой сканер.

Функции, выполняемые лексическим анализатором, и состав лексем, которые он выделяет в тексте исходной программы, могут меняться в зависимости от версии компилятора. В основном лексические анализаторы выполняют исключение из текста исходной программы комментариев и незначащих пробелов, а также выделение лексем следующих типов: идентификаторов, строковых, символьных и числовых констант, ключевых (служебных) слов входного языка.

В простейшем случае фазы лексического и синтаксического анализа могут выполняться компилятором последовательно. Но для многих языков программирования информации на этапе лексического анализа может быть недостаточно для однозначного определения типа и границ очередной лексемы. Иллюстрацией такого случая может служить пример оператора программы на языке Фортран, когда по части текста *DO 10 I=1...* невозможно определить тип оператора (а соответственно, и границы лексем). В случае *DO 10 I=1.15* - это будет присвоение вещественной переменной *DO10I* значения константы *1.15* (пробелы в Фортране игнорируются), а в случае *DO 10 I=1,15* - это цикл с перечислением от *1* до *15* по целочисленной переменной *I* до метки *10*.

В большинстве компиляторов лексический и синтаксический анализаторы - это взаимосвязанные части. Лексический разбор исходного текста в таком варианте выполняется поэтапно так, что синтаксический анализатор, выполнив разбор очередной конструкции языка, обращается к сканеру за следующей лексемой. При этом он может сообщить информацию о том, какую лексему следует ожидать. В процессе разбора может даже происходить “откат назад”, чтобы выполнить анализ текста на другой основе. В дальнейшем будем исходить из предположения, что все лексемы могут быть однозначно выделены сканером на этапе лексического разбора.

Работу синтаксического и лексического анализаторов можно изобразить в виде схемы на рис. 2.

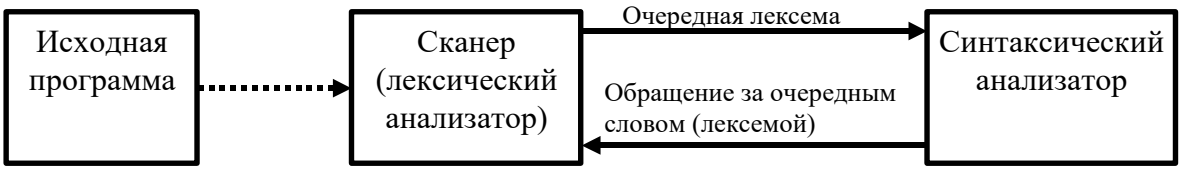

Рис. 2. Взаимодействие синтаксического и лексического анализаторов.

Вот пример фрагмента текста программы на языке Паскаль и соответствующей ему таблицы лексем (табл. 1):

```
...
begin
    for i:=1 to N do
        fg := fg * 0.5
...
```

Таблица 1

Таблица лексем программы

Лексема	Тип лексемы	Значение
begin	Ключевое слово	X_1
for	Ключевое слово	X_2
i	Идентификатор	i : 1
:=	Знак присваивания	S_1
1	Целочисленная константа	1
to	Ключевое слово	X_3
N	Идентификатор	N : 2
do	Ключевое слово	X_4
fg	Идентификатор	fg : 3
:=	Знак присваивания	S_1
fg	Идентификатор	fg : 3
*	Знак арифметической операции	A_1
0.5	Вещественная константа	0.5

Вид представления информации после выполнения лексического анализа целиком зависит конструкции компилятора. Но в общем виде ее можно представить как таблицу лексем, которая в каждой строчке должна содержать информацию о виде лексемы, ее типе и, возможно, значении. Обычно такая таблица имеет два столбца: первый – строка лексемы, второй – указатель на информацию о лексеме, может быть включен и третий столбец – тип лексем.

Лексический анализатор имеет дело с таким объектами, как различного рода константы и идентификаторы (к последним относятся и ключевые слова). Язык констант и идентификаторов в большинстве случаев является регулярным - то есть, может быть описан с помощью регулярных (праволинейных или леволинейных) грамматик. Распознавателями для регулярных языков являются конечные автоматы. Существуют правила, с помощью которых для любой регулярной грамматики может быть построен недетерминированный конечный автомат, распознающий цепочки языка, заданного этой грамматикой.

Недетерминированный конечный автомат задается пятеркой:

$\mathbf{M}=(\mathbf{Q},\mathbf{\Sigma},\delta,q_0,\mathbf{F})$,

где:

Q - конечное множество состояний автомата;

Σ - конечное множество допустимых входных символов;

δ - заданное отображение множества $\mathbf{Q}^*\mathbf{\Sigma}$ во множество подмножеств $\mathbf{P(Q)}$ δ: $\mathbf{Q}^*\mathbf{\Sigma}\rightarrow\mathbf{P(Q)}$ (иногда δ называют функцией переходов автомата);

$q_0\in\mathbf{Q}$ - начальное состояние автомата;

$\mathbf{F}\subseteq\mathbf{Q}$ - множество заключительных состояний автомата.

Работа автомата выполняется по тактам. На каждом очередном такте i автомат, находясь в некотором состоянии $q_i\in\mathbf{Q}$, считывает очередной символ $w\in\mathbf{\Sigma}$ из входной цепочки символов и изменяет свое состояние на $q_{i+1}=\delta(q_i,w)$, после чего указатель в цепочке входных символов передвигается на следующий символ и начинается такт i+1. Так продолжается до тех пор, пока цепочка входных символов не закончится. Конец цепочки символов часто помечают особым символом ⊥. Считается также, что перед тактом 1 автомат находится в начальном состоянии q_0.

Говорят, что автомат допускает цепочку $\alpha\in\mathbf{\Sigma}^*$, если в результате выполнения всех тактов работы над этой цепочкой он окажется в состоянии $q\in\mathbf{F}$. Язык, определяемый автоматом, является множеством всех цепочек, допускаемых автоматом. Для анализа цепочки с помощью конечного автомата достаточно подать ее на вход автомата, выполнить все такты его работы и определить, перешел ли автомат в результате работы в одно из заданных конечных состояний.

Графически автомат отображается нагруженным однонаправленным графом, в котором вершины представляют состояния, дуги отображают переходы из одного состояния в другое, а символы нагрузки (пометки) дуг соответствуют функции перехода. Если функция перехода предусматривает переход из состояния q в q’ по нескольким символам, то между ними строится одна дуга, которая помечается всеми символами, по которым происходит переход из q в q’.

Недетерминированный автомат неудобен для анализа цепочек, так как в нем могут встречаться состояния, допускающие неоднозначность, т.е. такие, из которых выходит две или более дуги, помеченных одним и тем же символом. Очевидно, что программирование работы такого автомата - нетривиальная задача.

Доказано, что любой недетерминированный автомат может быть преобразован в детерминированный так, чтобы их языки совпадали (говорят, что автоматы эквивалентны).

Детерминированный конечный автомат задается пятеркой:

$\mathbf{M'}=(\mathbf{Q'},\mathbf{\Sigma},\delta',q_0',\mathbf{F'})$,

где:

Q' - конечное множество состояний автомата;
Σ - конечное множество допустимых входных символов автомата;
$q_0' \in$ **Q'** - начальное состояние автомата;

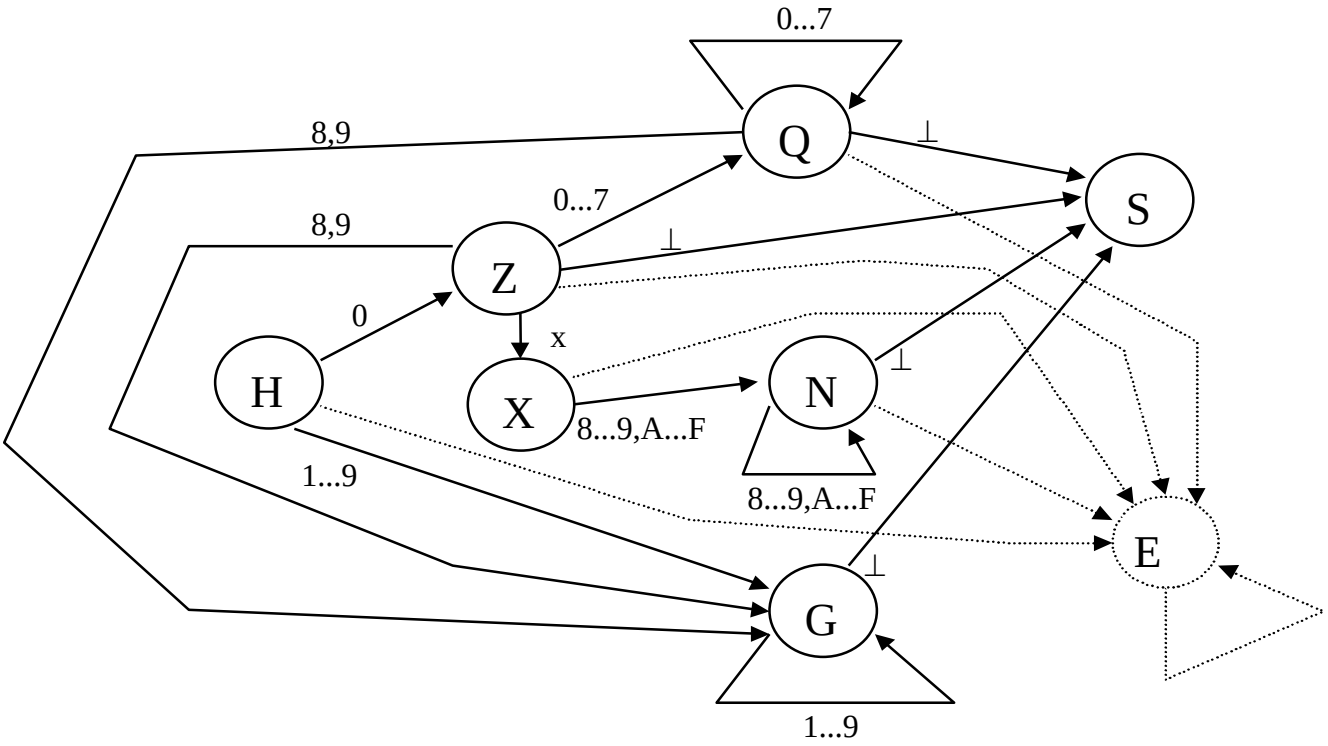

Рис. 3. Граф конечного детерминированного автомата,

распознающего грамматику целых чисел языка Си.

δ' - заданное отображение множества **Q'*Σ** во множество **Q'** δ: **Q'*Σ→Q'**;
F⊆Q' - множество заключительных состояний автомата.

После построения конечный детерминированный автомат может быть минимизирован, т.е. для него может быть построен эквивалентный ему автомат с минимальным числом состояний.

Можно написать функцию, отражающую функционирование любого детерминированного конечного автомата. Чтобы запрограммировать такую функцию, достаточно иметь переменную, которая бы отображала текущее состояние автомата, а переходы автомата из одного состояния в другое на основе символов входной цепочки могут быть построены с помощью операторов выбора. Работа функции должна продолжаться до тех пор, пока не будет достигнут конец входной цепочки. Для вычисления результата функции необходимо по ее завершению проанализировать состояние автомата. Если это одно из конечных состояний, то функция выполнена успешно, и входная цепочка принимается, если нет - то входная цепочка не принадлежит заданному языку.

Рассмотрим пример анализа лексем, представляющих собой целочисленные константы без знака в формате языка Си. В соответствии с требованиями языка, такие константы могут быть десятичными, восьмеричными, либо шестнадцатеричными. Восьмеричной константой считается число, начинающееся с 0 и содержащее цифры от 0 до 7; шестнадцатеричная константа должна начинаться с последовательности символов 0x и может содержать цифры и буквы от A до F (будем рассматривать только прописные буквы). Остальные числа считаются десятичными (правила их записи напоминать, наверное, не стоит). Будем считать, что всякое число завершается символом конца строки ⊥.

Рассмотренные выше правила могут быть записаны в форме Бэкуса-Наура в грамматике целочисленных констант без знака языка Си (для избежания путаницы терминальные символы грамматики выделены жирным шрифтом).

G({**0**...**9**,**A**...**F,** ⊥},{S,G,X,N,Q,Z},P,S)

P: S→ G⊥|Z⊥|N⊥|Q⊥
G→ **1**|**2**|**3**|**4**|**5**|**6**|**7**|**8**|**9**|G**0**|G**1**|G**2**|G**3**|G**4**|G**5**|G**6**|G**7**|G**8**|G**9**|Z**8**|Z**9**|Q**8**|Q**9**
H→ X**0**|X**1**|X**2**|X**3**|X**4**|X**5**|X**6**|X**7**|X**8**|X**9**|X**A**|X**B**|X**C**|X**D**|X**E**|X**F**|
N**0**|N**1**|N**2**|N**3**|N**4**|N**5**|N**6**|N**7**|N**8**|N**9**|N**A**|N**B**|N**C**|N**D**|N**E**|N**F**
X→ Z**x**
Q→ Z**0**|Z**1**|Z**2**|Z**3**|Z**4**|Z**5**|Z**6**|Z**7**|Q**0**|Q**1**|Q**2**|Q**3**|Q**4**|Q**5**|Q**6**|Q**7**

Z→ **0**

Хорошо видно, что данная грамматика является регулярной грамматикой (леволинейной). Конечный детерминированный автомат **M'**({N,Z,X,N,Q,G,S,ER},{0...9,A...F,⊥},δ,H,{S}), который распознает язык, заданный этой грамматикой, изображен на рис. 3. Начальным состоянием автомата является состояние *H*. В автомат дополнительно введено особое состояние *E*, обозначающее ошибку в распознавании цепочки символов. На графе автомата дуги, идущие в это состояние, не нагружены символами. По принятому соглашению они обозначают функцию перехода по любому символу, кроме тех символов, которыми уже помечены другие дуги, выходящие из той же вершины графа. Такое соглашение принято, чтобы не загромождать обозначениями граф автомата (на рис. 3 такие дуги и состояние *E* выделены пунктиром).

Можно написать программу, моделирующую работу указанного автомата. Ниже приводится текст функции на языке Паскаль, его реализующей. Результат функции истинный (*True*), если входная цепочка символов принадлежит входному языку автомата. Границей цепочки считается символ с кодом 0 (*#0*), в функции он искусственно добавляется в конец цепочки.

В программе переменная iState отображает текущее состояние автомата, переменная i является счетчиком символов входной строки. Конечно, рассмотренная программа может быть оптимизирована (например, можно сразу же прекращать разбор по обнаружению ошибки), но в данном примере оптимизация не выполнялась, чтобы можно было четко отследить соответствие между программой и построенным автоматом.

```
type
    TAutoState = ( AUTO_H, AUTO_Z, AUTO_X,
                   AUTO_Q, AUTO_N, AUTO_G,
                   AUTO_ER, AUTO_S );

function RunAuto (sInput: string): Boolean;
var
    iState : TAutoState;
    i : integer;
begin
    sInput := sInput + #0;
    iState := AUTO_H;
    i := 0;
    repeat
        i := i + 1;
        case iState of
        AUTO_H:
            case sInput[i] of
                '0':        iState := AUTO_Z;
                '1'..'9':   iState := AUTO_G;
                else        iState := AUTO_E;
            end;
        AUTO_Z:
            case sInput[i] of
                '0'..'7':   iState := AUTO_Q;
                '8','9':    iState := AUTO_G;
                'x':        iState := AUTO_X;
                #0:         iState := AUTO_S;
                else        iState := AUTO_E;
            end;
        AUTO_X:
            case sInput[i] of
                '0'..'9':   iState := AUTO_N;
                'A'..'F':   iState := AUTO_N;
                else        iState := AUTO_E;
            end;
        AUTO_Q:
            case sInput[i] of
                '0'..'7':   iState := AUTO_Q;
                '8','9':    iState := AUTO_G;
                #0:         iState := AUTO_S;
                else        iState := AUTO_E;
            end;
        AUTO_N:
            case sInput[i] of
                '0'..'9':   iState := AUTO_N;
                'A'..'F':   iState := AUTO_N;
                #0:         iState := AUTO_S;
                else        iState := AUTO_E;
            end;
        AUTO_G:
            case sInput[i] of
                '0'..'9':   iState := AUTO_G;
                #0:         iState := AUTO_S;
                else        iState := AUTO_E;
            end;
        AUTO_E:     iState := AUTO_E;
        end {case};
    until (sInput[i] = #0);
    RunAuto := (iState = AUTO_S);
end; { RunAuto }
```

Однако в общем случае задача сканера несколько шире, чем просто проверка цепочки символов лексемы на соответствие ее входному языку. Сканер должен выполнить те или иные действия по запоминанию распознанной лексемы (занесение ее в таблицу лексем). Набор действий определяется реализацией компилятора. Обычно эти действия выполняются сразу же по обнаружению конца распознаваемой лексемы, поэтому их несложно вставить в соответствующие места рассмотренной выше программы-сканера (в те операторы, где обнаруживается символ *#0*).

Вторая проблема, которая уже обсуждалась выше, это выделение границ лексем. Ведь во входном тексте лексемы не ограничены специальными символами. Если говорить в терминах программы-сканера, то определение границ лексем - это выделение тех строк в общем потоке входных символов, для которых надо выполнять распознавание. В общем случае эта задача может быть сложной, но для простейших входных языков границы лексем распознаются по заданным терминальным символам. Эти символы - пробелы, знаки операций, символы комментариев, а также разделители (запятые, точки с запятой и др.). Набор таких терминальных символов может варьироваться в зависимости от входного языка. Важно отметить, что знаки операций сами также являются лексемами, и необходимо не пропустить их при распознавании текста.

Таким образом, алгоритм работы простейшего сканера можно описать так:

- просматривается входной поток символов программы на исходном языке до обнаружения очередного символа, ограничивающего лексему;
- для выбранной части входного потока выполняется функция распознавания лексемы;
- при успешном распознавании информация о выделенной лексеме заносится в таблицу лексем, и алгоритм возвращается к первому этапу;
- при неуспешном распознавании выдается сообщение об ошибке, а дальнейшие действия зависят от реализации сканера - либо его выполнение прекращается, либо делается попытка распознать следующую лексему (идет возврат к первому этапу алгоритма).

Работа программы-сканера продолжается до тех пор, пока не будут просмотрены все символы программы на исходном языке из входного потока.

Порядок выполнения работы

1. 1. Получить вариант задания у преподавателя.
2. 2. Разработать КС-грамматику входного языка в соответствии с заданием.
3. 3. Подготовить и защитить отчет.
4. 4. Написать и отладить программу на ЭВМ.
5. 5. Сдать работающую программу преподавателю.

Требования к оформлению отчета

Отчет должен содержать следующие разделы:

- Задание по лабораторной работе.
- Описание КС-грамматики входного языка в форме Бэкуса-Наура.
- Описание алгоритма работы сканера или граф конечного автомата для распознавания цепочек (в соответствии с вариантом задания).
- Текст программы (оформляется после выполнения программы на ЭВМ).
- Выводы по проделанной работе.

Основные контрольные вопросы

1. Что такое трансляция, компиляция, транслятор, компилятор?
2. Из каких процессов состоит компиляция? Расскажите об общей структуре компилятора.
3. Какую роль выполняет лексический анализ в процессе компиляции?
4. Как связаны лексический и синтаксический анализ?
5. Дайте определение цепочки, языка. Что такое синтаксис и семантика языка?
6. Какие существуют методы задания языков? Какие дополнительные вопросы необходимо решить при задании языка программирования?
7. Что такое грамматика? Дайте определения грамматики.
8. Как выглядит описание грамматики в форме Бэкуса-Наура.
9. Какие классы грамматик существуют? Что такое регулярные грамматики?
10. Дайте определения контекстно-свободной грамматики, выводимости цепочки, непосредственной выводимости, длины вывода.
11. Что такое лексема? Расскажите, какие типы лексем существуют в языках программирования.
12. Что такое конечный автомат? Дайте определение детерминированного и недетерминированного конечных автоматов.
13. Расскажите о возможности преобразования недетерминированного конечного автомата в детерминированный.

14. Какие проблемы необходимо решить при построении сканера на основе конечного автомата?

Варианты заданий

1. Входной язык содержит арифметические выражения, разделенные символом **;**(точка с запятой). Арифметические выражения состоят из идентификаторов, десятичных чисел с плавающей точкой (в обычной и логарифмической форме), знака присваивания (**:=**), знаков операций +, -, *****, / и круглых скобок.

2. Входной язык содержит логические выражения, разделенные символом **;**(точка с запятой). Логические выражения состоят из идентификаторов, констант **true** и **false**, знака присваивания (**:=**), знаков операций **or**, **xor**, **and**, **not** и круглых скобок.

3. Входной язык содержит операторы условия типа **if** … **then** … **else** и **if** … **then**, разделенные символом **;**(точка с запятой). Операторы условия содержат идентификаторы, знаки сравнения <, >, =, десятичные числа с плавающей точкой (в обычной и логарифмической форме), знак присваивания (**:=**).

4. Входной язык содержит операторы цикла типа **for (**…**;** …**;** …**) do**, разделенные символом **;**(точка с запятой). Операторы цикла содержат идентификаторы, знаки сравнения <, >, =, десятичные числа с плавающей точкой (в обычной и логарифмической форме), знак присваивания (**:=**).

5. Входной язык содержит арифметические выражения, разделенные символом **;**(точка с запятой). Арифметические выражения состоят из идентификаторов, римских чисел, знака присваивания (**:=**), знаков операций +, -, *****, / и круглых скобок.

6. Входной язык содержит логические выражения, разделенные символом **;**(точка с запятой). Логические выражения состоят из идентификаторов, констант **0** и **1**, знака присваивания (**:=**), знаков операций **or**, **xor**, **and**, **not** и круглых скобок.

7. Входной язык содержит операторы условия типа **if** … **then** … **else** и **if** … **then**, разделенные символом **;**(точка с запятой). Операторы условия содержат идентификаторы, знаки сравнения <, >, =, римские числа, знак присваивания (**:=**).

8. Входной язык содержит операторы цикла типа **for (**…**;** …**;** …**) do**, разделенные символом **;**(точка с запятой). Операторы цикла содержат идентификаторы, знаки сравнения <, >, =, римские числа, знак присваивания (**:=**).

9. Входной язык содержит арифметические выражения, разделенные символом **;**(точка с запятой). Арифметические выражения состоят из идентификаторов, шестнадцатеричных чисел, знака присваивания (**:=**), знаков операций +, -, *****, / и круглых скобок.

10. Входной язык содержит логические выражения, разделенные символом **;**(точка с запятой). Логические выражения состоят из идентификаторов, шестнадцатеричных чисел, знака присваивания (**:=**), знаков операций **or**, **xor**, **and**, **not** и круглых скобок.

11. Входной язык содержит операторы условия типа **if** … **then** … **else** и **if** … **then**, разделенные символом **;**(точка с запятой). Операторы условия содержат идентификаторы, знаки сравнения <, >, =, шестнадцатеричные числа, знак присваивания (**:=**).

12. Входной язык содержит операторы цикла типа **for (**…**;** …**;** …**) do**, разделенные символом **;**(точка с запятой). Операторы цикла содержат идентификаторы, знаки сравнения <, >, =, шестнадцатеричные числа, знак присваивания (**:=**).

13. Входной язык содержит арифметические выражения, разделенные символом **;**(точка с запятой). Арифметические выражения состоят из идентификаторов, символьных констант (один символ в одинарных кавычках), знака присваивания (**:=**), знаков операций +, -, *****, / и круглых скобок.

14. Входной язык содержит логические выражения, разделенные символом **;**(точка с запятой). Логические выражения состоят из идентификаторов, символьных констант ‘T’ и ‘F’, знака присваивания (**:=**), знаков операций **or**, **xor**, **and**, **not** и круглых скобок.

15. Входной язык содержит операторы условия типа **if** … **then** … **else** и **if** … **then**, разделенные символом **;**(точка с запятой). Операторы условия содержат идентификаторы, знаки сравнения <, >, =, строковые константы (последовательность символов в двойных кавычках), знак присваивания (**:=**).

16. Входной язык содержит операторы цикла типа **for (**…**;** …**;** …**) do**, разделенные символом **;**(точка с запятой). Операторы цикла содержат идентификаторы, знаки сравнения <, >, =, строковые константы (последовательность символов в двойных кавычках), знак присваивания (**:=**).

Примечание:

- римскими числами считать последовательности больших латинских букв **X**, **V** и **I**;
- шестнадцатеричными числами считать последовательность цифр и символов ‘a’, ‘b’, ‘c’,’d’, ’e’ и ‘f’, начинающуюся с цифры (например: 89, 45ac9, 0abc4);
- задание по лабораторной работе №2 взаимосвязано с заданием по лабораторной работе №3, для уточнения состава входного языка можно посмотреть грамматику, заданную в работе №3 по соответствующему варианту.

Контрольные вопросы (ПК-11):

1. Каковы задачи лексического анализатора?
2. Что такое лексема?
3. Каким образом алгоритмически определить границы лексем?
4. Какие таблицы формируются в процессе работы лексического анализатора?
5. Приведите типы лексем программы.
6. Поясните принцип работы сканера лексического анализатора?
7. Сформулируйте этапы работы алгоритма синтаксического анализатора?
8. Какие конструкции языка программирования используются при построении лексического анализатора?
9. Назовите существующие программы синтаксического анализа.
10. Какие математические модели используются при построении алгоритмов лексических анализаторов?
11. Какие проблемы необходимо решить при построении сканера на основе конечного автомата?
12. Чем отличается функционирование лексического анализатора в составе компилятора от работы обычного КА?
13. Что является выходом лексического анализатора?

Практическое занятие 3. Алгоритмическая реализация синтаксического анализатора.

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 $A \rightarrow X_1\ X_2\ \ldots\ X_n$ – 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 рис.9.

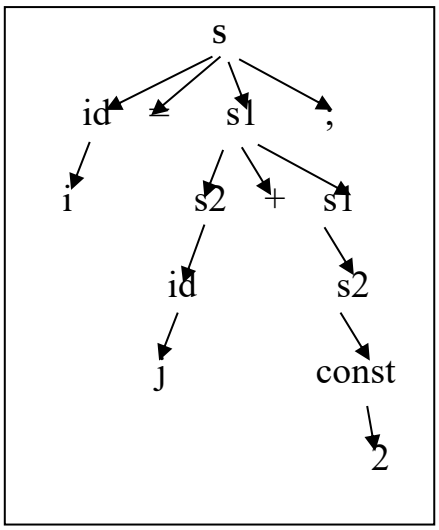

Рисунок 8. Дерево разбора.

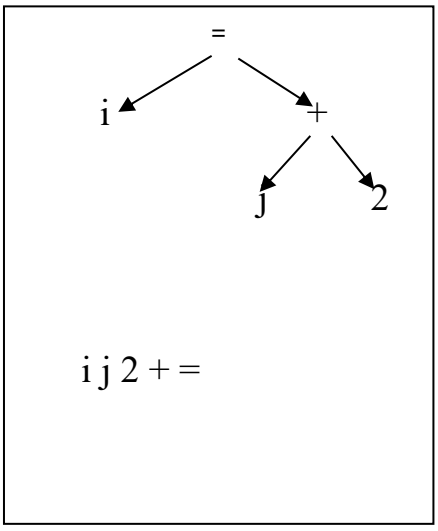

Рисунок 9. Синтаксическое дерево и постфиксная запись.

Семантический анализ имеет целью проверку 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 (ПК-11):

1. Какие задачи выполняют семантический и синтаксический анализ?
2. Что лежит в основе работы синтаксического анализатора?
3. Можно ли обойтись без семантического анализатора? В каких случаях?
4. Как формулируется формальная задача синтаксического анализа?
5. Чем различаются таблица лексем и таблица идентификаторов? В какую из этих таблиц лексический анализатор не должен помещать ключевые слова, разделители и знаки операций?
6. Что является исходными данными для синтаксического анализатора?
7. Какова формальная задача синтаксического разбора (анализа)?
8. Что образует основу синтаксического анализатора?
9. Что является выходными данными синтаксического анализатора?
10. Что такое дерево разбора?
11. Что такое дерево операций?
12. Каким образом осуществляется преобразование деревьев?
13. Какие конструкции языка программирования можно использовать для реализации синтаксического анализатора?

Практическое занятие 4. Разработка учебного компилятора.

ЦЕЛЬ РАБОТЫ: Изучить методы генерации кода с учетом различных промежуточных форм представления программы. Изучить методы управления памятью и особенности из 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 являет собой 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 представление (постфиксная запись), графической 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.

Если, начав с нижнего листа самой левой ветви дерева, обойти все листья и узлы дерева так, чтобы ветви рассматривались с лева 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: $abc \times + dab + / -$.

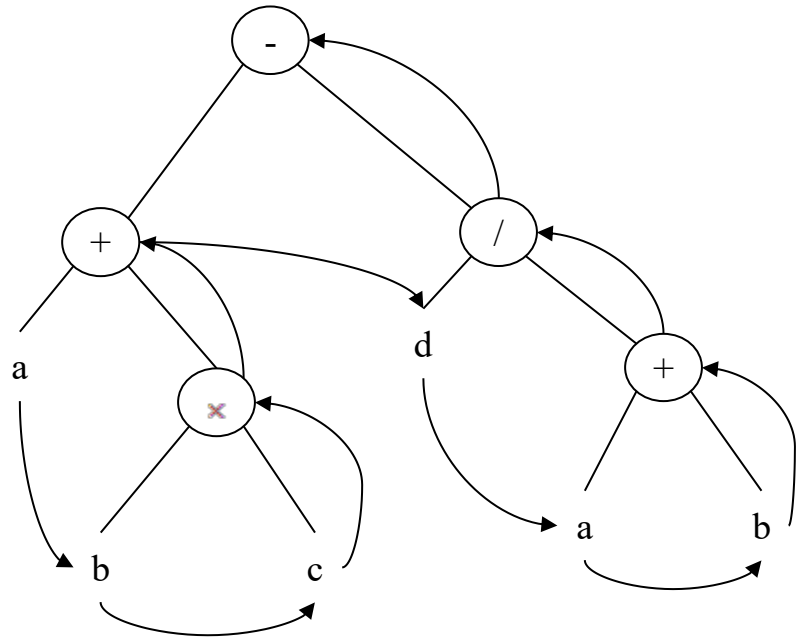

Рисунок 10. Порядок обхода дерева простого арифметического выражения для получения ОПЗ

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 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 таблицы 1. Рассматриваемый на каждом шаге процесса элемент строки отмечен «крышкой». В третьем столбце таблицы 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	2	3	4
1	$\hat{a}\ b\ c \times\ d\ a\ b\ +/-$	посмотреть следующий элемент	------
2	$a\ \hat{b}\ c \times\ +\ d\ a\ b\ +/-$	посмотреть следующий элемент	------
3	$a\ b\ \hat{c} \times\ +d\ a\ b\ +/-$	посмотреть следующий элемент	------
4	$a\ b\ c\ \widehat{\times} + d\ a\ b\ +/-$	$r_1 := b \times c$	$\times\ b\ c\ r_1$
5	$a\ r_1\ \widehat{+}\ d\ a\ b\ +/-$	$r_1 := a + r_1$	$+a\ r_1\ r_1$
6	$r_1\ \hat{d}\ a\ b\ +/-$	посмотреть следующий элемент	------
7	$r_1 d\ \hat{a}\ \ b\ +/-$	посмотреть следующий элемент	------
8	$r_1 d\ a\ \hat{b}\ +/-$	посмотреть следующий элемент	------
9	$r_1 d\ a\ b\ \widehat{+}/-$	$r_2 := a + b$	$+\ a\ b\ r_2$
10	$r_1 d\ r_2\ \widehat{/}-$	$r_2 := d/r_1$	$/\ d\ r_2\ r_2$
11	$r_1\ r_2\ \widehat{=}$	$r_1 := r_1 - r_2$	$-r_1\ r_2\ r_1$

12	r_1	------	------

Алгоритм перевода ОПЗ в машинные команды

Последовательность машинных команд в таблице 4.1 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:
   a. Среди элементов стека СО[*i*–*k*], … ,СО[*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 $l = j$.
   b. 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 $x_1, \ldots, x_k$.
   c. Занести символ r_l в СО [*i*–*k*].
   d. 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 3b 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 под программа, 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 3b 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

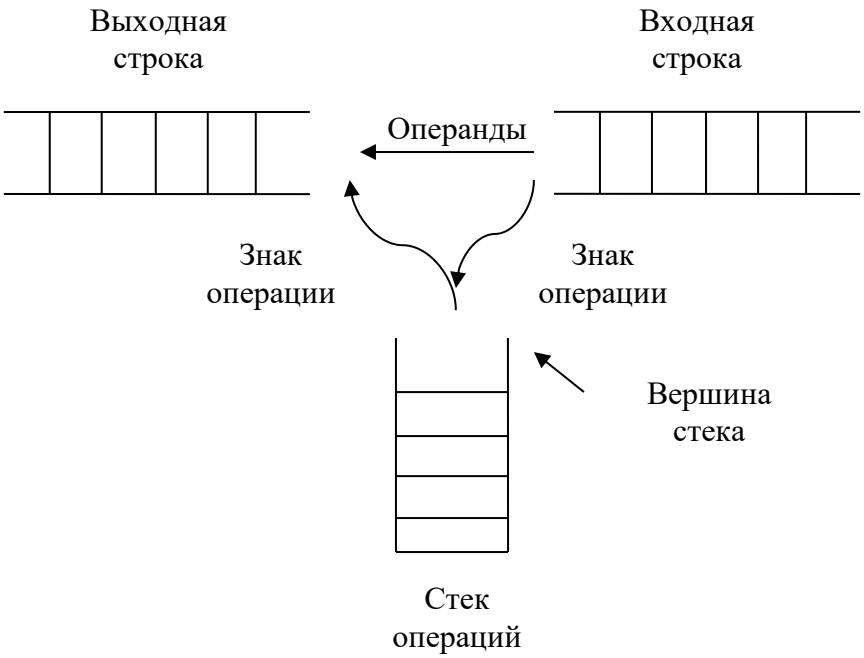

Рисунок 11. Использование стека операций для перевода выражений в ОПЗ

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 или равными приоритету входного знака. 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 дописывания их к выходной строке.

Пример. Перевести в ОПЗ выражение $a + b \times c - d/(a + b)$. Решение показано в таблице 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 $abi1 + jAЭМ + c \times d +$.

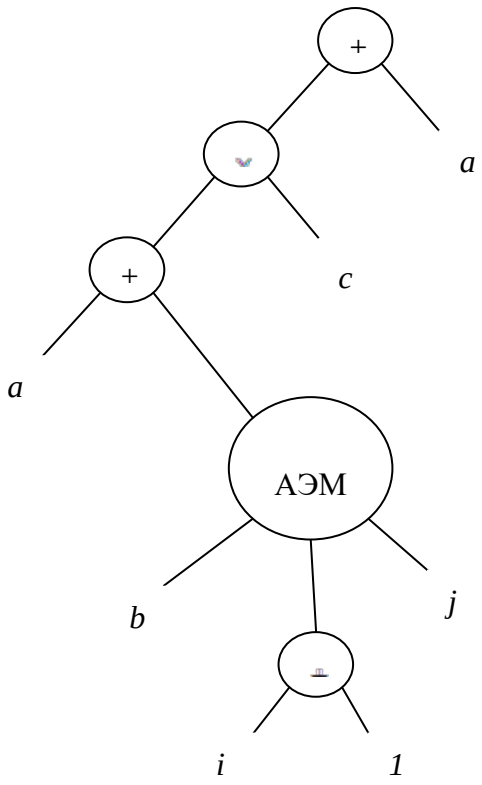

Рисунок 12. Дерево выражения, содержащего переменную с индексами

Как обычно, в ОПЗ левее операции АЭМ расположены операнды, количество которых 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

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

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, изображённым на рис. 13. Обход дерева даёт ОПЗ $abc + =$.

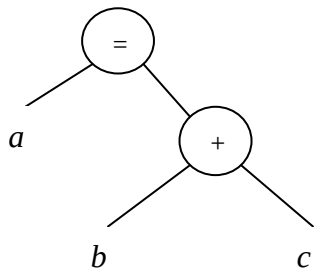

Рисунок 13. Графическое изображение оператора присваивания

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).

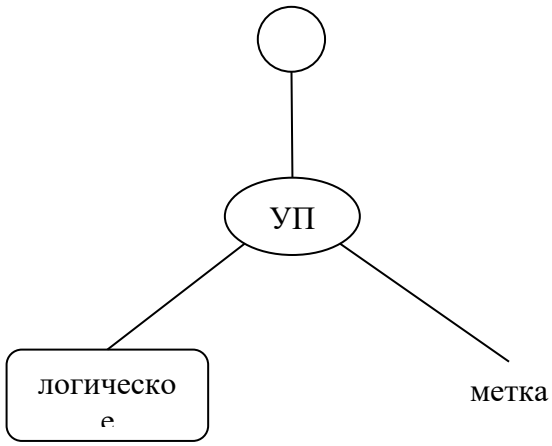

Рисунок 14. Графическое Изображение операции УПЛ

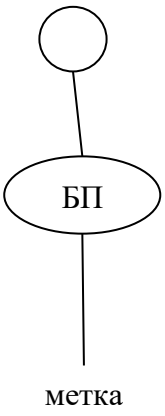

Рисунок 15. Графическое изображение операции БП

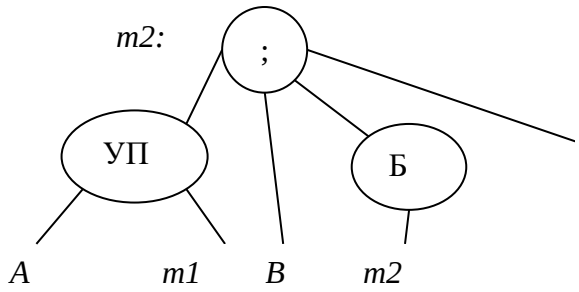

Рисунок 16. Дерево, изображающее условный оператор

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 даёт 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: if(!A) goto m_2; B; goto m_1; m_2: ,

для которой ОПЗ выглядит следующим образом:

m_1: $A\ m_2$ УПЛ $\ B\ m_1$ БП m_2:

Пример. 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: B; if(A) goto m_1; ,

для которой ОПЗ выглядит следующим образом:

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

где ! – унарная операция логического отрицания.

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

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

Оператор цикла с 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:

$A\ \ m_1$: $\ B\ m_2$ УПЛ $\ D\ C\ m_1$ БП m_2:

Пример. for (a=0, x=1; a<n; 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 путем реализации 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.

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 С++ 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 С++ включает:
   - данные типа **int, char**;
   - инструкции описания переменных;
   - операторы присваивания в любой последовательности;
   - операции +, – , <, >, побитовые операции <<, >>, **&**, | .

10) Подмножество языка С++ 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 **+, – , = =, != , <, >** .

Контрольные вопросы (ПК-11):

1. Какова цель семантического анализатора?
2. Почему распределение памяти не может быть выполнено до выполнения семантического анализа?
3. Можно ли построить компилятор без оптимизации кода?
4. В чем заключается процедура генерации кода?
5. Какие способы внутреннего представления программ существуют?
6. Как образуется обратная польская запись операций?
7. Как вычисляются выражения с помощью обратно польской записи?
8. Как представляются в алгоритме операторы и операнды?
9. Какие принципы лежат в основе распределения памяти?
10. Почему в компиляторах используют относительные адреса памяти?

Практические типовые задания:

1. В среде Паскаль с помощью ассемблерных вставок вывести на экран текущие День, месяц и год. (INT 21h, функция 2Ah . Получение системной даты. Позволяет получить значение текущей даты. При вызове: AH=2Ah. При возврате: CX=год (от 1980 до 2099) DH=месяц (от 1 до 12) DL=день (от 1 до 31) (ПК-11).
2. В среде Паскаль с помощью ассемблерных вставок вывести на экран текущее время (INT 21h, функция 2Ch . Получение системного времени. Позволяет получить значение текущего времени. При вызове: AH=2Ch При возврате: CX=часы (от 0 до 23) CL=минуты (от 0 до 59) DH=секунды (от 0 до 59) (ПК-11).
3. В среде Паскаль разработать программу, вычисляющую и выводящую на экран сумму, разность и произведение двух переменных. Операции сложения, вычитания и умножения выполнить в виде ассемблерных вставок (ПК-11).
4. В среде Паскаль создайте программу, обеспечивающую вывод на экран результата сложения, вычитания, умножения и деления двух переменных. Установите точки останова в произвольных местах программы. Выполните трассировку программы. Выполните пошаговую отладку программы. Выполните прогон программы "до курсора". Поясните назначение отладчика (ПК-11).
5. В среде Паскаль создайте программу, обеспечивающую в цикле вывод на экран всех элементов массива (массив из 10 элементов). Установите точки останова в произвольных местах программы. Выполните трассировку программы. Настройте встроенный отладчик таким образом, чтобы в процессе трассировки отображалось значение счетчика цикла. Поясните назначение отладчика (ПК-11).
6. В среде Norton Utilities выполните проверку диска D. Поясните какие ошибки исправляет утилита и каким образом (ПК-11).
7. В среде Norton Utilities выполните дефрагментацию диска D. Поясните причины возникновения фрагментации файловой системы, ее последствия и процесс дефрагментации (ПК-11).
8. Загрузить ПК с системной дискеты, создать файлы Config.sys и Autoexec.bat, позволяющие автоматически загружать программу русификатор, драйвер мыши и VolkovCommander в верхнюю область памяти. Определить размер свободной памяти (ПК-11).
9. Загрузить ПК с системной дискеты, создать файлы Config.sys и Autoexec.bat, позволяющие автоматически загружать программу Volkov Commander . Определить размер свободной памяти. Измените Config.sys и Autoexec.bat таким образом, чтобы программы использовали верхнюю область памяти. Определить размер свободной памяти (ПК-11).
10. Загрузить ПК с системной дискеты, создать файлы Config.sys и Autoexec.bat, позволяющие с помощью меню выбирать варианты загрузки: 1)русификатор + Volkov Commander, 2)мышь+ Volkov Commander (ПК-11).
11. Загрузить ПК с системной дискеты, создать файлы Config.sys и Autoexec.bat, позволяющие с помощью меню выбирать варианты загрузки: 1)русификатор + Volkov Commander, 2)русификатор+мышь+ Volkov Commander (ПК-11).
12. В среде (Сеанс MSDOS) из командной строки на диске D создать каталог 44. Внутри него создать каталог 421. С помощью редактора Edit создать текстовые файлы 421.txt и 422. txt произвольного содержания и сохранить его в каталоге 421. Скопировать файл 421.txt в каталог 44. Переместить файл 422.txt в каталог 44. Переименовать файл 422.txt. Удалить каталог 44 со всем содержимым (ПК-11).
13. В среде (Сеанс MSDOS) из командной строки на диске D создать каталог 44. Внутри него создать каталог 421. С помощью редактора Edit создать текстовые файлы 421.txt и 422. txt произвольного содержания и сохранить его в каталоге 421. Вывести на экран содержимое созданных файлов. Переместить файл 422.txt в каталог 44. Переименовать файл 422.txt. Вывести содержимое каталога 44. Удалить каталог 44 со всем содержимым (ПК-11).
14. В среде (Сеанс MSDOS) из командной строки выбрать диск C, выбрать диск D,

вывести метку тома диска D. Изменить метку тома диска D. Вывести текущее время на экран. Создать на диске D папку kaf44. Перейти в папку kaf44. Удалить папку kaf44 (ПК-11).

15. В среде (Сеанс MSDOS) просмотреть состав сети. Синхронизировать системное время с сервером времени. Определить перечень сетевых ресурсов компьютера SERVER44. Подключите сетевым диском папку 2001 на компьютере SERVER44. Отключите сетевой диск (ПК-11).

16. В среде (Сеанс MSDOS) на диске D создать файл 421.txt. Просмотреть состав сети. Синхронизировать системное время с сервером времени. Определить перечень сетевых ресурсов компьютера SERVER44. Скопируйте созданный файл в папку 2001 компьютера SERVER44 (ПК-11).

17. В среде Volkov Commander выбрать диск C, выбрать диск D. Переключиться между панелями. На диске D создать каталог 44. Внутри него создать каталог 421. Создать текстовые файлы 421.txt и 422. txt произвольного содержания и сохранить его в каталоге 421. Скопировать файл 421.txt в каталог 44. Переместить файл 422.txt в каталог 44. Переименовать файл 422.txt. Удалить каталог 44 со всем содержимым. При выполнении задач командную строку не использовать (ПК-11).

18. В среде Volkov Commander выбрать диск C, выбрать диск D. Переключиться между панелями. На диске D создать каталог 44. Внутри него создать каталог 421. Создать текстовый файл 421.txt произвольного содержания и сохранить его в каталоге 421. Скопировать файл 421.txt в каталог 44. Изменить содержимое файла. Установить для файла атрибут "только чтение". Удалить каталог 44 со всем содержимым. При выполнении задач командную строку не использовать (ПК-11).

19. В среде Windows определите состав ПЭВМ. Выясните, есть ли конфликты между устройствами. Поясните причины возникновения конфликтов и способы их устранения (ПК-11).