Білет №23

1. Використання регулярних (автоматичних) граматик для лексичного аналізу

Під регулярною граматикою будемо розуміти якусь ліволінійну граматику. Це така граматика, що її P правила мають вигляд A 🡪 Bt або A 🡪 t, де A є N, B є N, t є T. Для прикладу будемо розуміти якийсь символ «@» кінцем ланцюжка, що аналізується лексичним аналізатором. Для граматик такого типу передбачений свій алгоритм розбору для отримання лексеми:

* Перший символ вихідного ланцюжка замінюємо не терміналом А, для якого в граматиці є правило А 🡪 a
* Далі проводимо ітерації до кінця ланцюжка за наступною схемою: отриманий раніше не термінал А і розташований правіше від нього термінал a вихідного ланцюжка замінюємо не терміналом В, для якого в граматиці є правило В 🡪 Aa

Стан автомату повинен відповідати типу розпізнаної лексеми або типу помилки лексичного аналізу. Вхідними сигналами автомату повинні бути літери вхідної послідовності, а точніше класифікаційні ознаки цієї літери. Таким чином для побудови лексичного аналізатора доцільно мати класифікаційну таблицю літер мови, що обробляє лексичний аналізатор,- двомірну матрицю переходів, що визначає код наступного стану автомату, ондим з яких є поточний стан автомату а другим – класифікатор вхідної літери. Тобто, щоб визначити такий автомат необхідно визначити перенумеровані типи даних для кодів стану і кодів специфікаторів. Власне програма повинна виділяти чергову лексему шляхом циклічного прогряду літер. Цикл закінчується в тому випадку, коли знайдена помилка або лексема закінчується роздільником. Для того, щоб використовувати лексичний аналіз методом теорії автоматів або автоматним методом необхідно використовувати таблицю класифікаторів, складних роздільників та таблицю ключових слів. Результати роботи лексичного аналізу треба рознести по таблицях імен, по таблицях констант і, можливо, по таблицях модулів. Результати лексичного аналізу доцільно розміщувати в спеціалізованих структурах, в яких зберігається інформація про код вхідної лексеми та її внутрішнє подання.

2. Організація роботи з таблицями в системних програмах

**(по конспекту)**

**В системному програмуванні звичайно висувається вимога унікальності (змісту ключових слів). Для цього в табл. ключ. слів всі образи цих слів повинні бути унікальними, що не дозволяє виконувати кожне слово як об’єкт користувача. При створенні табл. імен ключові поля складають образ імені і номер блоку його визначення.**

Таблиці – складні структури даних, завдяки яким можна підвищити ефективність програми. Їх основним призначенням є пошук в них інформації по заданому аргументу. Тому вони мають схожу структуру з базами даних. Таблиці мають задану кількість полів з даними, що є ключовими і функціональними. Таблиці системних програм зберігаються в ОП. Пошук виконується схожим чином з командою вибірки SELECT. Аргумент пошуку визначається значенням, що відповідає умові WHERE. До ключового поля висувається умова однозначності. Для додавання – INSERT. UPDATE і DELETE майже не використовуються в системних програмах.

Основные операции:

- оздание или получение доступа к таблице (конструктор)

- удаление таблицы

- включение нового эл-та

- поиск эл-та (эл-тов)

- упорядочение

- удаление эл-та (эл-тов)

- коррекция эл-та

Приведен пример фрагмента программы поиска по простейшемму линейному алгоритму с последовательным сравнением аргумента поиска с соответствующим полем. Будем считать, что аргумент загружен в аккумулятор AX, начальный адрес таблицы в - ES:DI, а колличество элементов таблицы в - CX

MOV BX, DI ; сохранение начального адреса таблицы

REPNZ SCASW ; сканирование

JNZ NotFnd ; переход если не найден

SUB DI, BX ; определение индекса найденного эл-та

SUB DI, 2 ; компенсация технологического пропуска

SRL DI, 1 ; получения индекса из разницы адресов

В системных программах используются таблицы имен и констант в транслирующих программах, которые предназначены для синтаксического анализа и семантической обработки . Структурно таблицы обычно организуются как массивы и ссылочные структуры. Структуры данных табличного типа широко используются в системных программах, так как обеспечивает простое и быстрое обращение к данным. Таблицы состоят из элементов, каждый из которых представляется несколькими полями. Так при трансляции программы создаются, например, таблицы, содержащие элементы в виде: Ключ (переменная /метка) и характеристики: сегмент(данных / кода), смещение (ХХХХ), тип (байт, слово или двойное слово / близкий или дальний).

3. Основні архітектурні елементи захищеного режиму та їх призначення

**Організація переключення задач у захищенному режимі**

Для перехода в защищенный режим можно воспользоваться средствами того же BIOS и протокола DPHI, предварительно подготовив таблицы и базовую конфигурацию задач защищенного режима. Для организации переключения задач применен метод логических машин управления. Основу его аппаратно-программной реализации в процессорах ix86 составляем команды IMBCALL и IRET, бит NT регистра флагов, а также прерывания.

Для перехода от задачи к задаче при управлении мультизадачностью используются команды межсегментной передачи управления – переходы и вызовы. Задача также может активизироваться прерыванием. При реализации одной из этих форм управления назначение определяется элементом в одной из дескрипторных таблиц.

Тип дескриптора может быть таким, который инициирует выполнение новой задачи после сохранение состояния текущей. Имеется 2 кода типов, определяющих дескрипторы сегментов состояния задачи (TSS) и шлюза задачи. Когда управление передается любому из дескрипторов этих типов, происходит переключение задачи.

Дескрипторы шлюзов хранят только заполненные байты прав доступа и селектор соответствующего обьекта в глобальной таблице дескрипторов, помещенный на место 2-х младших байтов базового адреса. При каждом переключении задачи процессор может перейти с другой локальной дескрипторной таблицы, что позволяет назначить каждой задаче свое отображение логических адресов на физические.

Переключение задачи состоит из действий выполняемых одной из команд JMPPAR, CALLTAR или RET при NT=1:

1. проверка, разрешено ли уходящей задачи переключиться на новую
2. проверка файла, что дескриптор TSS приходящей задачи отмечен как присутствующий и имеет правильный предел (не меньше 67Н)
3. сокращение состояния уходящей задачи
4. загрузка в регистр ТR селектора TSS входящей задачи
5. загрузка состояния входящей задачи из ее сегмента TSS и продолжения выполнения.

При переключении задачи всегда сохраняется состояние уходящей задачи

Перед тем, как переключить процессор в защищённый режим, надо выполнить некоторые подготовительные действия, а именно:

* Подготовить в оперативной памяти глобальную таблицу дескрипторов GDT. В этой таблице должны быть созданы дескрипторы для всех сегментов, которые будут нужны программе сразу после того, как она переключится в защищённый режим. Впоследствии, находясь в защищённом режиме, программа может модифицировать GDT (если, разумеется, она работает в нулевом кольце защиты). Программа может модифицировать имеющиеся дескрипторы или добавить новые, загрузив заново регистр GDTR.
* Для обеспечения возможности возврата из защищённого режима в реальный необходимо записать адрес возврата в реальный режим в область данных BIOS по адресу 0040h:0067h, а также записать в CMOS-память в ячейку 0Fh код 5. Этот код обеспечит после выполнения сброса процессора передачу управления по адресу, подготовленному нами в области данных BIOS по адресу 0040h:0067h.
* Запретить все маскируемые и немаскируемые прерывания.
* Открыть адресную линию A20.
* Запомнить в оперативной памяти содержимое сегментных регистров, которые необходимо сохранить для возврата в реальный режим, в частности, указатель стека реального режима.
* Загрузить регистр GDTR.

Это самый простой этап. Для перевода процессора i80286 из реального режима в защищённый можно использовать специальную команду LMSW, загружающую регистр состояния процессора (Mashine Status Word). Младший бит этого регистра указывает режим работы процессора. Значение, равное 0, соответствует реальному режиму работы, а значение 1 - защищённому.

Если установить младший бит регистра состояния процессора в 1, процессор переключится в защищённый режим:

mov ax, 1

lmsw ax

**Робота програм для обробки преривань в захищенному режимі**

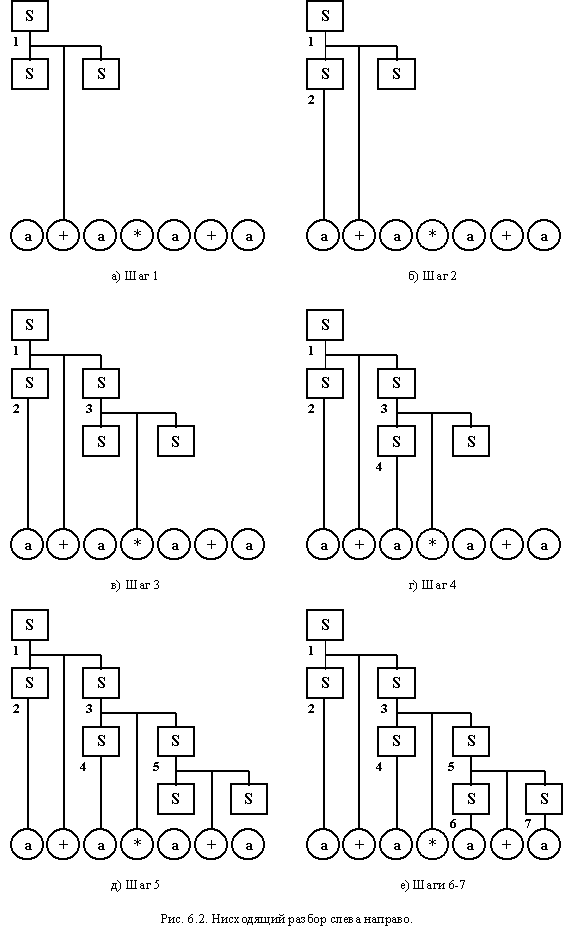
В цьому режимі звичайно в таблицю дискретних преривань заносимо дискретний шлюз преривань, в якому зберігається адреса слова сегмента стану задачі TSS для задачі обробки преривань. В ОС може бути одна чи декілька сегментів задач. При переключені задачі управління передач за новим сегментом стану задачі запамьятовується адреса переваної задачі. В цьому випадку новий сегмент TSS буде повьязан з іншим адресним простором і буде включати в себе новий стек для нової задачі. Регітри переривань програми запамьятовуються в старому TSS і таким чинов в обробчику переривань в захисному режимі нема необхідності зберігати регістри перивань задачі. При виконанні команди IRET наприкінці обробки переривань відбувається перехід до перерваної задачі з відновленого старого TSS.

Білет №24

1. Методи низхідного розбору.

Нисходящий разборзаключается в построении дерева разбора, начиная от корневой вершины. Разбор заключается в заполнении промежутка между начальным нетерминалом и символами входной цепочки правилами, выводимыми из начального нетерминала. Подстановка основывается на том факторе, что корневая вершина является узлом, состоящим из листьев, являющихся цепочкой терминалов и нетерминалов одного из альтернативных правил, порождаемых начальным нетерминалом. Подставляемое правило в общем случае выбирается произвольно. Вместо новых нетерминальных вершин осуществляется подстановка выводимых из них правил. Процесс протекает до тех пор, пока не будут установлены все связи дерева, соединяющие корневую вершину и символы входной цепочки, или пока не будут перебраны все возможные комбинации правил. В последнем случае входная цепочка отвергается. Построение дерева разбора подтверждает принадлежность входной цепочки данному языку. При этом, в общем случае, для одной и той же входной цепочки может быть построено несколько деревьев разбора. Это говорит о том, что грамматика данного языка является недетерминированной. Эти рассуждения иллюстрируются следующим примером. Пусть будет дана грамматика G: **G6 = ({S}, {a, +, \*}, P, S)**, где P определяется как: 1) S ****a; 2) S ****S + S; 3) S ****S \* S

Цепочки, порождаемые данной грамматикой можно интерпретировать как выражения, состоящие из операндов **"a"**, а также операций **"+"** и **"\*"**. Недетерминированность грамматики позволяет порождать одну и ту же терминальную цепочки с использованием различных выводов. Например, выражение **"a+a\*a+a"** можно получить следующими способами:



* S **** S+S **** a+S **** a+S\*S **** a+ a\*S **** a+a\*S+S **** a+a\*a+S **** a+a\*a+a
* S **** S+S **** S+a **** S\*S+a **** S\*a+a **** S+S\*a+a **** S+a\*a+a **** a+a\*a+a (6.1)
* S **** S\*S **** S+S\*S **** S+S\*S+S **** a+ S\*S+S **** a+a\*S+S **** a+a\*S+a **** a+a\*a+a

И так далее. В этом пример число вариантов одной и той же произвольной цепочки вывода настолько велико, что не имеет и смысла говорить о практическом применении данной грамматики. Но в данном случае она позволяет показать, каким образом могут порождаться различные деревья при нисходящем разборе. Пошаговое построение различных деревьев показано на рис. Можно отметить, что процесс построения дерева совпадает с последовательностью шагов вывода входной цепочки.

Методи низхідного розбору:

* Метод рекурсивного спуску
* Метод синтаксичних графів

**(по конспекту)**

**Метод низхідного розбору в якому аналіз конструкції викнується починаючи від найбільш складних конструкції доо термінальних позначень. В цьому випадку за базу можуть бути взяті правила підстановки. Один з найпоширеніших – метод рекурсивного спуску. В цьому методі розбір починається з кінцевого позначення граматики. При виконанні такого розбору аналізатор звертається до підлеглого ресурсу, щоб розібрати спочатку перший а потім наступні позначення правої частини правила підстановки.Рекурсивні правила у формі Бекуса прийнято, так що рекурсивні звертання записуються з правого боку, що утворює ліворекурсивні правила. Однак при такому розборі ми будмо просуватись в глибину рекурсії , не просуваючись вздовж вхідного потоку даних, що фактично призводить до за циклювання аналізатора. Тому для використованого методу рекурсивного спуску необхідно перетворити правила на право рекурсивну форму. Альтернативним методом є метод синтаксичних графів.**

Number>::=[<Sign>]<digit>{<digit>}[<Separator><digit>{<digit>}]

[<Exponent>[<Sign>]<digit>{<digit>}]

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

<Sign>::='+'|'-'

<Separator>::='.'

<Exponent>::='E'|'e'

Теперь на основе этих правил напишем функцию IsNumber, которая в качестве параметра принимает строку и возвращает True, если эта строка удовлетворяет правилам записи числа, и False, если не удовлетворяет.

|  |  |
| --- | --- |
| |  | | --- | | // Проверка символа на соответствие <digit>  function IsDigit(Ch:Char):Boolean;  begin  Result:=Ch in ['0'..'9']  end;  // Проверка символа на соответствие <Sign>  function IsSign(Ch:Char):Boolean;  begin  Result:=(Ch='+') or (Ch='-')  end;  // Проверка символа на соответствие <Separator>  function IsSeparator(Ch:Char):Boolean;  begin  Result:=Ch='.'  end;  // Проверка символа на соответствие <Exponent>  function IsExponent(Ch:Char):Boolean;  begin  Result:=(Ch='E') or (Ch='e')  end;  function IsNumber(const S:string):Boolean;  // Номер символа выражения, который сейчас проверяется  var P:Integer;  begin  Result:=False;  // Проверка, что выражение содержит хотя бы один символ.  // пустая строка не является числом  if Length(S)=0 then  Exit;  // Начинаем проверку с первого символа  P:=1;  // Если первый символ - <Sign>, переходим к следующему  if IsSign(S[P]) then  Inc(P);  // Проверяем, что в данной позиции стоит хотя бы одна цифра  if (P>Length(S)) or not IsDigit(S[P]) then  Exit;  // Переходим к следующей позиции, пока не достигнем  // конца строки или не встретим не цифру  repeat  Inc(P)  until (P>Length(S)) or not IsDigit(S[P]);  // Если достигли конца строки, выражение корректно - число,  // не имеющее дробной части и экспоненты  if P>Length(S) then  begin  Result:=True;  Exit  end;  // Если следующий символ - <Separator>, проверяем,  // что после него стоит хотя бы одна цифра  if IsSeparator(S[P]) then  begin  Inc(P);  if (P>Length(S)) or not IsDigit(S[P]) then  Exit;  repeat  Inc(P)  until (P>Length(S)) or not IsDigit(S[P]);  // Если достигли конца строки, выражение корректно - число  // без экспоненты  if P>Length(S) then  begin  Result:=True;  Exit  end  end;  // Если следующий символ - <Exponent>, проверяем,  // что после него стоит всё то, что требуется правилами  if IsExponent(S[P]) then  begin  Inc(P);  if P>Length(S) then  Exit;  if IsSign(S[P]) then  Inc(P);  if (P>Length(S)) or not IsDigit(S[P]) then  Exit;  repeat  Inc(P)  until (P>Length(S)) or not IsDigit(S[P]);  if P>Length(S) then  begin  Result:=True;  Exit  end  end  // Если выполнение дошло до этого места, значит,  // в выражении остались ещё какие-то символы. Т.к. никакие  // дополнительные символы синтаксисом не предусмотрены,  // такое выражение не считается корректным числом.  end; | |

Для каждого нетерминального символа мы ввели отдельную функцию, разбор начинается с символа самого верхнего уровня - <Number> - и следует правилам, записанным для этого символа. Такой способ синтаксического анализа называется левосторонним рекурсивным нисходящим анализом. Левосторонним потому, что символы в выражении перебираются слева направо, нисходящим - потому, что сначала анализируются символы верхнего уровня, а потом - символы нижнего. Рекурсивность метода на данном примере не видна, т.к. наша грамматика не содержит рекурсивных определений, но мы с этим столкнёмся в последующих примерах.

1. Особливості пересилок і перетворень в розширенні ММХ.

ММХ-расширение предназначено для поддержки приложений, ориентированных на работу с большими массивами данных целого и вещественного типов, над которыми выполняются одинаковые операции. С данными такого типа обычно работают мультимедийные, графические, коммуникационные программы – от этого и название MultiMedia eXtensions. Важное отличие ММХ-команд от обычных команд процессора в том, как они реагируют на ситуации переполнения и заема. Возникают ситуации, когда результат арифм. операции выходит за размер разрядной сетки исходных операндов. В этом случае производится усечение старших бит результата и возвращаются только те биты, которые умещаются в пределах исходного операнда (арифметика с циклическим переносом). Некоторые ММХ-команды в такой ситуации действуют иначе. В случае выхода значения результата за пределы операнда, в нем фиксируется максимальное или минимальное значение (арифметика с насыщением). ММХ\_расширение имеет команды, которые выполняют арифметические операции с использование обоих принципов. При этом среди них есть команды, учитывающие знаки элементов операндов. ПРИМЕР:

*include mmx16.inc*

*.data*

*mem dw 4444h*

*df 111122223333h*

*.code*

*movd rmmx0, mem ; rmmx0=0000 0000 3333 4444*

*movq rmmx0, mem ; rmmx0=1111 2222 3333 4444*

**Команды для пересылки.**

|  |
| --- |
| **MOVD** приемник,источник – пересылка двойных слов |

Команда копирует двойное слово из источника (регистр ММХ, обычный регистр или переменная) в приемник (регистр ММХ, обычный регистр или переменная, но хотя бы один из операндов обязательно должен быть регистром ММХ). Если приемник — регистр ММХ, двойное слово записывается в его младшую половину (биты 31 – 0), а старшая заполняется нулями. Если источник — регистр ММХ, в приемник записывается младшее двойное слово этого регистра.

|  |
| --- |
| **MOVQ** приемник,источник – пересылка учетверенных слов |

Копирует учетверенное слово (64 бита) из источника (регистр ММХ или переменная) в приемник (регистр ММХ или переменная, оба операнда не могут быть переменными).

**Команды преобразования данных.**

|  |
| --- |
| **PACKSSWB** приемник,источник – упаковка байт со знаковым насыщением **PACKSSDW** приемник, – упаковка слов со знаковым насыщением |

Команды упаковывают и насыщает слова со знаком в байты (PACKSSWB) или двойные слова со знаком в слова (PACKSSDW). Команда PACKSSWB копирует четыре слова (со знаком), находящиеся в приемнике (регистр ММХ), в 4 младших байта (со знаком) приемника и копирует четыре слова (со знаком) из источника (регистр ММХ или переменная) в старшие четыре байта (со знаком) приемника. Если значение какого-нибудь слова больше +127 (7Fh) или меньше -128 (80h), в байты помещаются числа +127 и -128 соответственно. Команда PACKSSDW аналогично копирует два двойных слова из приемника в два младших слова приемника и два двойных слова из источника в два старших слова приемника. Если значение какого-нибудь двойного слова больше +32 767 (7FFFh) или меньше -32 768 (8000h), в слова помещаются числа +32 767 и -32 768 соответственно.

|  |
| --- |
| **PACKUSWB** приемник,источник – упаковка с беззнаковым насыщением |

Копирует четыре слова (со знаком), находящиеся в приемнике (регистр ММХ), в 4 младших байта (без знака) приемника и копирует четыре слова (со знаком) из источника (регистр ММХ или переменная) в старшие четыре байта (без знака) приемника. Если значение какого-нибудь слова больше 255 (FFh) или меньше 0 (00h), в байты помещаются числа 255 и 0 соответственно.

|  |
| --- |
| **PUNPCKHBW** приемник,источник – распаковка и объединение старших элементов (байт) **PUNPCKHWD** приемник,источник – распаковка и объединение старших элементов (двойное слово) **PUNPCKHDQ** приемник,источник – распаковка и объединение старших элементов (учетверенное слово) |

Команды распаковывают старшие элементы источника (регистр ММХ или переменная) и приемника (регистр ММХ) и записывают их в приемник через один.

Команда PUNPCKHBW объединяет по 4 старших байта источника и приемника, команда PUNPCKHWD объединяет по 2 старших слова, и команда PUNPCKHDQ копирует в приемник по одному старшему двойному слову из источника и приемника.

Если источник содержит нули, эти команды фактически переводят старшую половину приемника из одного формата данных в другой, дополняя увеличиваемые элементы нулями. PUNPCKHBW переводит упакованные байты в упакованные слова, PUNPCKHWD переводит слова в двойные слова, и PUNPCKHDQ переводит единственное старшее двойное слово приемника в учетверенное.

|  |
| --- |
| **PUNPCKLBW** приемник,источник – распаковка и объединение младших элементов (байтов) **PUNPCKLWD** приемник,источник – распаковка и объединение младших элементов (двойное слов) **PUNPCKLDQ** приемник,источник – распаковка и объединение младших элементов (учетверенное слово) |

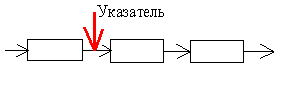
Команды распаковывают младшие элементы источника (регистр ММХ или переменная) и приемника (регистр ММХ) и записывают их в приемник через один аналогично предыдущим командам. Команда PUNPCKLBW объединяет по 4 младших байта источника и приемника, команда PUNPCKLWD объединяет по 2 младших слова, и команда PUNPCKLDQ копирует в приемник по одному младшему двойному слову из источника и приемника. Если источник содержит только нули, эти команды, аналогично PUNPCKH\*, фактически переводят младшую половину приемника из одного формата данных в другой, дополняя увеличиваемые элементы нулями.

1. Побудова таблиць у вигляді списків.

Другая линия развития базовых структур таблиц основана на ссылочных структурах. Использование ссылок и указателей избавляет от необходимости множественных пересылок при сортировки таблиц. Наличие одно- и двунаправленных ссылок вдоль массива элементов приводит к построению одно- и двусвязных списков, в которых практически невозможно получить эффект от упорядочения при поиске.

Классический пример структуры данных последовательного доступа, в которой можно удалять и добавлять элементы в середине структуры, — это линейный список. Различают однонаправленный и двунаправленный списки (иногда говорят односвязный и двусвязный).

Элемены списка как бы выстроены в цепочку друг за другом. У списка есть начало и конец. Имеется также указатель списка, который располагается между элементами. Если мысленно вообразить, что соседние элементы списка связаны между собой веревкой, то указатель — это ленточка, которая вешается на веревку. В любой момент времени в списке доступны лишь два элемента — элементы до указателя и за указателем.

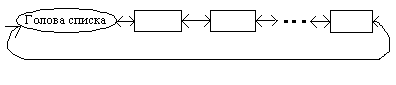


В однонаправленном списке указатель можно передвигать лишь в одном направлении — вперед, в направлении от начала к концу. Кроме того, можно установить указатель в начало списка, перед его первым элементом. В отличие от однонаправленного списка, двунаправленный абсолютно симметричен, указатель в нем можно передвигать вперед и назад, а также устанавливать как перед первым, так и за последним элементами списка.

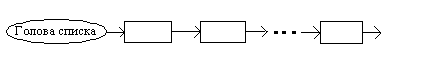
В двунаправленном списке можно добавлять и удалять элементы до и за указателем. В однонаправленном списке добавлять элементы можно также с обеих сторон от указателя, но удалять элементы можно только за указателем.

Удобно считать, что перед первым элементом списка располагается специальный пустой элемент, который называется головой списка. Голова списка присутствует всегда, даже в пустом списке. Благодаря этому можно предполагать, что перед указателем всегда есть какой-то элемент, что упрощает процедуры добавления и удаления элементов.

В двунаправленном списке считают, что вслед за последним элементом списка вновь следует голова списка, т.е. список зациклен в кольцо.



Можно было бы точно так же зациклить и однонаправленной список. Но гораздо чаще считают, что за последним элементом однонаправленного списка ничего не следует. Однонаправленный список, таким образом, представляет собой цепочку, начинающуюся с головы списка, за которой следует первый элемент, затем второй и так далее вплоть до последнего элемента, а заканчивается цепочка ссылкой в никуда.



#### Ссылочная реализация списка

Мы рассмотрели абстрактное понятие списка. Но в программировании зачастую отождествляют понятие списка с его ссылочной реализацией на базе массива или непосредственно на базе оперативной памяти.

Основная идея реализации двунаправленного списка заключается в том, что вместе с каждым элементом хранятся ссылки на следующий и предыдущий элементы. В случае реализации на базе массива ссылки представляют собой индексы ячеек массива. Чаще, однако, элементы списка не располагают в каком-либо массиве, а просто размещают каждый по отдельности в оперативной памяти, выделенной данной задаче. (Обычно элементы списка размещаются в так называемой **динамической памяти**, или **куче** — это область оперативной памяти, в которой можно при необходимости захватывать куски нужного размера, а после использования освобождать, т.е. возвращать обратно в кучу.) В качестве ссылок в этом случае используют адреса элементов в оперативной памяти.

Голова списка хранит ссылки на первый и последний элементы списка. Поскольку список зациклен в кольцо, то следующим за головой списка будет его первый элемент, а предыдущим — последний элемент. Голова списка хранит только ссылки и не хранит никакого элемента. Это как бы пустой ящик, в который нельзя ничего положить и который используется только для того, чтобы написать на нем адреса следующего и предыдущего ящиков, т.е. первого и последнего элементов списка. Когда список пуст, голова списка зациклена сама на себя.

Указатель списка реализуется в виде ссылки на следующий и предыдущий элементы, он просто отмечает некоторое место в цепочке элементов.

В случае однонаправленного списка хранится только ссылка на следующий элемент, таким способом экономится память. Голова однонаправленного списка хранит ссылку на первый элемент списка. Последний элемент списка хранит нулевую ссылку, т.е. ссылку в никуда, т.к. в программах нулевой адрес никогда не используется.

Ценность ссылочной реализации списка состоит в том, что процедуры добавления и удаления элементов не приводят к массовым операциям. Рассмотрим, например, операцию удаления элемента за указателем. Читая ссылку на следующий элемент в удаляемом элементе, мы находим, какой элемент должен будет следовать за указателем после удаления текущего элемента. После этого достаточно связать элемент до указателя с новым элементом за указателем. А именно, обозначим через X адрес элемента до указателя, через Y — адрес нового элемента за указателем. В поле следующий для элемента с адресом X надо записать значение Y, в поле предыдущий для элемента с адресом Y — значение X. Таким образом, при удалении элемента за указателем он исключается из цепочки списка, для этого достаточно лишь поменять ссылки в двух соседних элементах. Аналогично, для добавления элемента достаточно включить его в цепочку, а для этого также нужно всего лишь модифицировать ссылки в двух соседних элементах. Добавляемый элемент может располагаться где угодно, следовательно, нет никаких проблем с захватом и освобождением памяти под элементы.