**БИЛЕТ 19**

**1.Системні управляючі програми**

##### Системная программа - программа общего пользования, выполняемая вместе с прикладными программами и служащая для управления ресурсами компьютера: центральным процессором, памятью, вводом-выводом.

##### Системная программа - согласно ГОСТ 197881-90 - программа, предназначенная:

##### - для поддержания работоспособности системы обработки информации; или

##### - для повышения эффективности ее использования.

##### Различают системные управляющие и системные обслуживающие программы.

##### Системні управляючі програми призначені для виконання та управління в ОС. Ці програми були призначені для введення завантаження програм в комп’ютер; введення/виведення інформації на зовнішні присторої; управління подіями, що виникають в обчислювальних пристроях; розподілу ресурсами; управління доступом і захистом інформації в комп’ютері.

##### *Управляющие системные программы* организуют корректное функционирование всех устройств системы. Системные управляющие программы составили основу операционных систем и в таком виде и в таком виде используются и в настоящее время. Основные системные функции управляющих программ:

##### управление вычислительными процессами и вычислительными комплексами и

* работа с внутренними данными ОС.

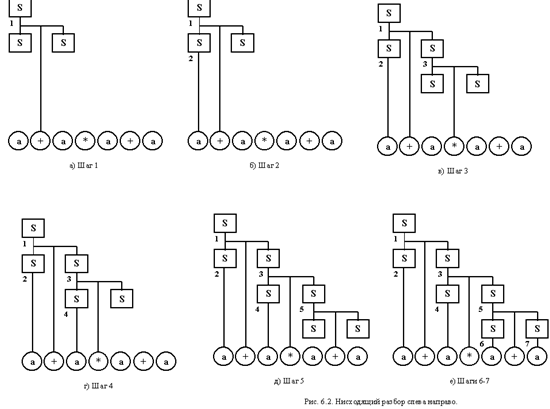
До системних управляючих програм відносять всі програми ОС:

* Програми початкового завантаження
* Програми попереднього програмування та контролю обладнання
* Програми управляння файловою системою
* Програми переключення задач

Как правило, они находятся в основной памяти. Это резидентные программы, составляющие ядро ОС. Управляющие программы, которые загружаются в память непосредственно перед выполнением, называю транзитными (transitive). В настоящее время системные управляющие программы поставляются фирмами-разработчиками и фирмами-дистрибьюторами в виде инсталляционных пакетов операционных систем и драйверов специальных устройств.

**2. Алгоритм Низхідного синтаксичного 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.

**(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.

**3. Способи переключення задач**

**Організація переключення задач**

В процесорі Пентіум передбачено спеціальні механізми переключення задач. В реальному режимі або режимі ДОС при створенні нової задачі або процесу (потоку) для нього створюється окремий стек в якому зберігаються зміст регістрів переривань задачі, які часто називають надтекстом задачі, однак в реальному режимі при обробці переривань. програми можуть використовувати для своєї роботи фрагменти стеку перерваної задачі, що призводить до проблем захисту цілісності кодів виконуваних задач. Щоб усунути проблеми захисту і узагальнити процеси переключення задач важливо використовувати той самий сегмент стану TSS. Крім регістрів в сегменті TSS записується інформація про стан об’єктів введення-виведення у спеціальному розширенні сегмента. Якщо в задачі використовується регістр з плаваючою точкою, то їх збереження покладається на відповідальність програміста, залежно від того як використовуватиме регістр з плаваючою точкою задача що переривається і задача що надходить на виконання.

Для того щоб перейти до кодів, які пов’язані з сегментом перерваної задачі необхідно спочатку підготувати в спеціальному регістрі задач номер відповідної задачі, а потім необхідно виконати команду переходу jmp, call на відповідний сегмент задачі. При цьому в старому сегменті задачі будуть збережені сегментні регістри та регістри загального користування, а з нового сегменту TSS будуть відновлені регістри нової задачі на яку відбувається переключення.

Для організації переключення задач за такою схемою для кожної задачі (в тому числі і для системних) потрібно побудувати принаймі один сегмент статусу задач TSS. Якщо для виконання якоїсь задачі передбачено виконувати підзадачі, які називаються в більшості ОС потоками (thread), які на відміну від процесів (process) виконуються в тому ж адресному просторі, що і головні процеси задач. Таким чином для потоків треба також створювати сегмент TSS, але в ни буде зберігатися інформація про однакові таблиці сегментів і сегментні регістрів, але різна інформація про уточнюючі значення регістрів.

В захищеному режимі selector включає 13 бітовий номер дескриптора в локальну або глобальну таблицю дескрипторів в молодшому біті ознака локальної(1) або глобальної (0) таблиці дескрипторів і ще 2 біта необхідні при встановленні доступу. Сам дескрипторний сегмент задачі включає початкову адресу сегментам (32), довжину сегмента (20) а всі інші біти для управління інформацією. Крім того у дескрипторі зберігається біт зайтяності сегмента