Теория по дисциплине "Языки программирования" за II семестр.

Большинство ответов составлены по методичке Лукиновой О. В.

1. КАКОЙ ЯЗЫК ЯВЛЯЕТСЯ ВЕДУЩИМ В ОБЛАСТИ: НАУЧНЫХ ВЫЧИСЛЕНИЙ; КОММЕРЧЕСКИХ ПРИЛОЖЕНИЙ; ИСКУССТВЕННОГО ИНТЕЛЛЕКТА; СИСТЕМНЫХ РАЗРАБОТОК? ПОЧЕМУ?

а) В ***научных вычислениях*** нужны языки типа ***Fortran***, для которых обязательным требованием является наличие:

- арифметики с плавающей точкой, т.е. реальных чисел;
- массивов, для реализации операций с векторами и матрицами;
- циклических конструкций, для осуществления итерационных вычислений.

б) В ***коммерческом направлении*** – язык ***Cobol***, электронные таблицы типа Excel). Здесь обеспечиваются:

- выполнение операций с фиксированной точкой;
- генерацию различных отчетов, а, следовательно, должна быть мощная обработка строковых данных;
- возможность обрабатывать большие объемы данных.

в) В ***системах искусственного интеллекта*** нужны языки ***Lisp, Prolog***, ориентированые на задачи обработки текстовых символов, вывод решений в нестандартных ситуациях, общение с пользователем на естественном языке.

г) В ***системном программировании*** необходимы средства низкого уровня для написания программ (драйверов) связи с аппаратурой. Наиболее подходящим языком здесь является язык ***C***, т.к. в своем составе он имеет набор битовых операций.

2. ПРИВЕСТИ ПРИМЕР ОРТОГОНАЛЬНОСТИ В ЯП С И ОБЪЯСНИТЬ ПОЧЕМУ.

Ортогональность – возможность конструирования каких-либо конструкций языка из элементарных (например, возможность объявить новый пользовательский тип данных для описания предметной области реализуемой задачи).

В языке СИ имеется стандартный набор типов данных, используя который можно сконструировать новые (уникальные) типы данных.

```
union code
{
int digit;
char letter;
};
```


3. КАКАЯ КОНСТРУКЦИЯ ЯП ПОДДЕРЖИВАЕТ АБСТРАКЦИЮ ПРОЦЕССА И В ЧЕМ СОСТОИТ ПОНЯТИЕ АБСТРАКЦИИ ДАННЫХ?

Абстракция – возможность определять сложные объекты, игнорируя детали.

Абстракция состоит из:

- ***Абстракции процесса.*** Под абстракцией процесса в ЯПВУ понимается подпрограмма (процедура или функция), поскольку она определяет способ, с помощью которого программа может выполнить некоторый процесс, без уточнения деталей того, как именно это следует сделать (по крайней мере, в вызывающей программе).
- ***Абстракции данных.*** Абстракцию данных следует рассматривать как понятие переменной, которая представляет собой абстракцию ячейки ОП. При этом одной переменной определенного типа ставится в соответствие одна ячейка соответствующего типа. (Дополнение из интернета: абстракция данных связывает тип данных с определенным на нем набором операций.)

4. КАКИЕ ЯП НАЗЫВАЮТСЯ ИМПЕРАТИВНЫМИ?

Императивным языком называется язык программирования, который основан на принципах фон Неймана (см. вопрос 14). К данной категории относятся классические языки, такие как Fortran, Algol, C, C++, Pascal и пр. Как правило, императивный язык включает в себя следующие языковые структуры и объекты:

- понятие «переменной», являющейся базовым объектом языка и абстрактным аналогом ячейки оперативной памяти;
- операторы ввода/вывода данных в/из ячейки оперативной памяти;
- операторы присваивания, отражающих пересылки данных между ячейками;
- команды управления (условного и безусловного перехода, многовариантного ветвления, циклических конструкций).

5. ЧТО ТАКОЕ СОВМЕЩЕНИЕ ИМЕН. ПРОБЛЕМЫ АЛЬТЕРНАТИВНЫХ ИМЕН.

Совмещение имен предполагает использование одной и той же ячейки памяти под разными именами. Этот механизм является источником ненадежности в программе, хотя и оправдывал себя в условиях дефицита оперативной памяти. Примерами совмещений имен могут служить оператор EQUIVALENCE в языке FORTRAN или тип данных объединение Union в языке C.

6. ТРИ МЕТОДА РЕАЛИЗАЦИИ ЯП.

На сегодняшний день существуют три системы реализации ЯПВУ: *компиляция, чистая интерпретация и смешанная реализация*.

Компиляция

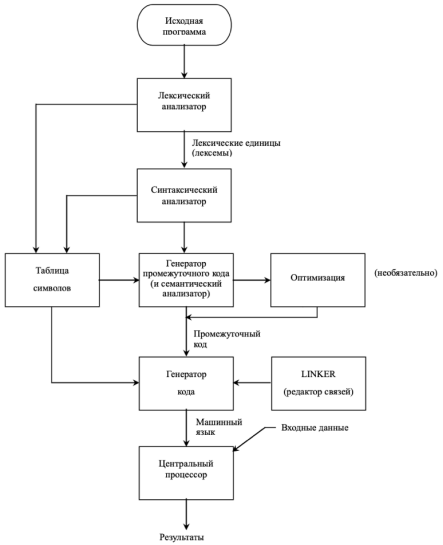
1. Здесь лексический анализатор объединяет символы программы в лексические единицы (лексемы) – идентификаторы, ключевые слова, знаки операций и т.п.

2. На основе лексем синтаксический анализатор строит деревья синтаксического анализа, таким образом, определяя синтаксическую структуру программы и сразу идентифицируя ошибки.

3. Производится оптимизация программы путем уменьшения ее размера и возможностей ускорения выполнения.

4. Генератор кода переводит оптимизированную программу в объектный код на машинном языке, а семантический анализатор выявляет ошибки статической семантики.

5. После компиляции объектный код программы подвергается обработке редактором связей, который разрешает все существующие в коде ссылки (объектный код программы дополняется пользовательскими и библиотечными процедурами и функциями), и преобразуется в загрузочный модуль на машинном языке, который поступает на выполнение в процессор.



Преимущества:

- компилируется сразу целиком весь модуль;
- после компиляции образуется выполняемый модуль, т.е. обеспечивается мобильность программы.

Чистая интерпретация

Программа-интерпретатор расшифровывает каждую команду программы и сразу ее выполняет в процессоре. К недостаткам можно отнести следующее: скорость чистой интерпретации от 10 до 100 раз медленнее, чем компиляции.

Смешанные системы реализации

Некоторые системы реализации ЯП представляют компромисс: они транслируют программу на промежуточный язык, разработанный для обеспечения более легкой интерпретации. Примером может служить язык Java, реализованный на основе смешанной реализации, программа преобразуется в промежуточный язык – байтовый код.

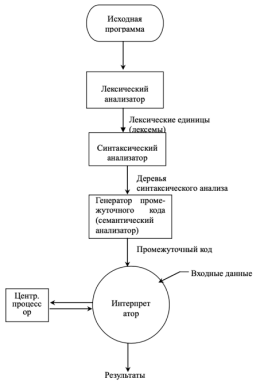


Рисунок 3. Система смешанной реализации

7. КАКИЕ АРГУМЕНТЫ МОЖНО ПРИВЕСТИ В ПОЛЬЗУ СОЗДАНИЯ ЕДИНОГО ЯП? КАКИЕ - ПРОТИВ?

Аргументы за создание единого ЯП:

- Удобство в обучении языку для новых специалистов в области программирования.
- Упрощенная система обновления структуры данного языка.

Аргументы против единого ЯП:

- Как правильно, ЯПы создавались под определенный круг решаемых задач, с учетом этого оформлялась их архитектура и функциональность. Создание единого ЯП может нарушить эту концепцию и получившийся язык должен будет включать в себя процедуры для слишком большого количества задач.

- Помимо прочего, постоянно меняются решаемые задачи — началось все с вопросов обороны (расчеты таблиц для артиллерии, взлома шифров ENIGMA, расчета атомных бомб и траекторий ракет), потом подтянулась экономика, потом компьютер заменил телефон, потом телефонокомпьютер (смартфон) заменил фото и видео камеру, стал распознавать образы, добавилась дополненная реальность и т.д.

- Данная идея невостребованная в наше время, поскольку уже имеется огромное количество готовых языков, с их производными, а также существует огромное множество обученных специалистов в конкретных языках, которым придется переучиваться.

- Большинство языков предполагают своего рода компромисс между медленным, высокоуровневым, но простым в использовании и быстрым, но слишком усложненным исполнением. Создание одного универсального языка, несомненно, заставит выбирать с точки зрения производительности, что затем ограничит его полезность в различных ситуациях.

Характеристики	Критерии оценки		
	Читабель-ность	Легкость создания	Надежность
1. Ортогональность/простота	×	×	×
2. Управляющие структуры	×	×	×
3. Типы и структуры данных	×	×	×
4. Синтаксическая структура	×	×	×
5. Поддержка абстракции		×	×
6. Проверка типов			×
7. Обработка исключительных ситуаций			×
8. Ограниченное совмещение имен переменных			×

8. КРИТЕРИИ И ХАРАКТЕРИСТИКИ ЯП. В ЧЕМ ОНИ ЗАКЛЮЧАЮТСЯ?

Простота языка предполагает:

- разумное количество элементарных конструкций;
- разумное количество описаний одних и тех же средств в реализации языка
- разумная перегрузка операторов.

Ортогональность – возможность конструирования каких-либо конструкций языка из элементарных (например, возможность объявить новый пользовательский тип данных для описания предметной области реализуемой задачи).

Управляющие структуры. Язык должен обладать достаточным многообразием управляющих структур.

Разнообразие типов и структур данных должно быть таким, чтобы давать возможность адекватного описания предметной области задач.

Продуманная ***синтаксическая структура программы*** предполагает:

- длину имен объектов в программе такую, чтобы иметь возможность отображать смысл объектов задачи;
- наличие зарезервированных специальных слов, отображающих смысл конструкций языка;
- наличие семантически оправданных символов составных операторов (например, слова begin...end вместо {....});
- отсутствие множественного смысла одних и тех же зарезервированных слов и операторов и т.п.

Поддержка абстракции. Абстракция – возможность определять сложные объекты, игнорируя детали. Абстракция в ЯПВУ представляется двумя понятиями: абстракция данных и абстракция процесса. (см. вопрос 3).

Проверка типов является одним из основных механизмов надежности языка, которая предполагает наличие средств обнаружения в программе всевозможных ошибок. При этом эти средства могут функционировать как при компиляции, так и при выполнении программы и осуществляют:

- проверку типов (статическую, динамическую);
- проверку диапазона изменения индексов массивов;
- обработку исключительных ситуаций.

Средства обработки исключительных ситуаций позволяют перехватывать ошибки и другие нештатные ситуации во время выполнения программы, принимать меры и затем продолжать работу. Этот механизм значительно повышает надежность программ.

Совмещение имен предполагает использование одной и той же ячейки памяти под разными именами.

Однако язык программирования можно оценивать и с точки зрения других классификаторов:

Мощность языка. Определяется тем разнообразием задач, которые программируются на данном языке. Наиболее мощным языком является машинный язык.

Уровень языка. Языки программирования делятся на:

- языки низкого уровня (машинные языки);
- ассемблеры, включающие мнемонические обозначения команд;
- языки высокого уровня.

Мобильность языка. Определяется независимостью от аппаратных средств.

Эффективность языка. Обеспечивает эффективную реализацию языка (включая эффективную реализацию компилятора и эффективные программы, генерируемые компилятором).

Основные виды ЯП:

1. ***Императивные языки (см. вопрос 4).***
2. ***Языки объектно-ориентированного программирования (ООП).*** Все используемые на сегодняшний день языки, такие как Visual C++, Delphi, Fortran90, ADA95 и пр., содержат объектную модель данных и обладают, наряду, с императивными, также свойствами ООП, т.е., по сути, они являются императивными языками, но с объектной надстройкой. Только язык Smalltalk80 остается единственным в чистом виде языком ООП. Наиболее близким к парадигме ООП на сегодняшний день является также и язык Java.
3. ***Функциональные языки.*** Осуществляют вычисления с помощью математических функций над двумя структурами данных: атомами (символами языка) и списками (последовательностью атомов). Понятия переменной, как модели ячейки оперативной памяти, оператора, выполняющего действия над содержимым ячеек в таких языках отсутствуют. Поэтому для реализации функциональных языков требуется не неймановская архитектура, однако таковой на сегодняшний день не существует. Первым языком (и наиболее распространенным), поддерживающим парадигму функционального программирования, является язык LISP. Затем возникли его диалекты, такие как Scheme, ML, COMMON LISP, Haskell, которые приобрели некоторые императивные свойства: переменные, операторы присваивания и цикла.
4. ***Логические языки.*** Основаны на символьной исчисления высказываний и логике предикатов. Здесь не существует порядка выполнения команд, задаваемом операторами действий императивных языков. Система реализации ЭВМ сама выбирает порядок выполнения команд в соответствии с правилами логического вывода, который приведет к нужному результату. Характерным представителем логического программирования является язык Prolog и его диалекты. Эффективность реализации логических языков в архитектуре фон Неймана также оставляет желать лучшего.

9. ОБЪЯСНИТЬ, КАКИЕ ФАКТОРЫ ОКАЗАЛИ НАИБОЛЬШЕЕ ВЛИЯНИЕ НА РАЗВИТИЕ ЯП.

На развитие языков программирования высокого уровня оказывают влияние несколько факторов.

Во-первых, ***архитектура вычислительной машины***, во-вторых, ***методологии программирования***. При этом под методологией (от греч. «учение о методах») будем понимать систему принципов и способов организации и построения теоретической и практической деятельности, а также учение об этой системе.

В этой связи следует упомянуть о трех методологических системах, которые являются базовыми факторами, повлиявшими и на состав языковых средств, и на технологию программирования:

1. Структурное (процедурное) программирование,
2. Информационно-ориентированное программирование,
3. Объектно-ориентированное программирование.

Третьим фактором является ***сложность и масштабность задач***, необходимость реализации которых возникает в практической жизни.

10. ОЦЕНИТЬ И ОБОСНОВАТЬ СВОИ ОЦЕНКИ ЯП С.

По слитым билетам это не требуется, да и вряд ли она станет такое давать :)

11. УКАЗАТЬ В ТАБЛИЦЕ В ХРОНОЛОГИЧЕСКОМ ПОРЯДКЕ РАССМОТРЕННЫЕ ЯП, А ТАКЖЕ УСОВЕРШЕНСТВОВАНИЯ, КОТОРЫЕ ПОЯВЛЯЛИСЬ В ЯП.

НАЗВАНИЕ ЯП		ЧЕМ ХАРАКТЕРИЗУЕТСЯ
Группа языков FORTRAN		
FORTRAN	**1995г.**	• отсутствие операторов описания типов данных: типы данных присваивались по умолчанию (переменные, имена которых начинаются с букв i, j, k, l, m, n – всегда целого типа, все остальные – вещественного); • наличие только арифметического оператора условного перехода; • длина идентификаторов менее 6 символов; • отсутствие раздельной компиляции подпрограмм; • система типов включала только 4 базовых типа данных; • массивы только статические, их размерность не превышала 3; • цикл только типа пересчета (FOR...); • практически отсутствовала проверка типов данных.
FORTRAN-II	–	• независимая компиляция подпрограмм; • операторы описания типов; • логический оператор IF...THEN; • оператор описания общих областей COMMON;
FORTRAN-IV		• оператор EQUIVALENCE, который определял единую память для нескольких переменных; • довольно большая встроенная библиотека математических функций.

FORTRAN-77	–	• символьные строки; • логический оператор условного перехода IF...THEN...ELSE; • циклы с условием (WHILE...);
FORTRAN-90	–	• динамические массивы; • указатели; • структурные типы данных; • операторы CASE, CYCLE (переход на «голову цикла» без выхода из него); • поддержка объектно-ориентированного программирования; • реализована обширная встроенная математическая библиотека.
Группа алголоподобных языков		
ALGOL	**1958г.**	• операторами описания типов данных; • составными операторами; • вложенными операторами условного перехода логического типа if...then....else; • массивами любой размерности.
ALGOL-60	–	• передача параметров в подпрограммы, как по значению, так и по ссылке; • блочная структура программы: • рекурсивные подпрограммы; • автоматические массивы.
ALGOL-68	–	• динамическими массивами; • указателями; • некоторыми типами, определяемыми пользователем; • оператором многовариантного ветвления switch.
Наследниками семейства языков ALGOL являются: язык ***Pascal*** – самый лучший язык программирования, придуманный человечеством, обладающий такими качествами как простота, выразительность, надежность, *скромность, красота, благочестие, девственность*; его объектно-ориентированный вариант ***Delphi***; машинно-независимый язык ***C*** и его потомки ***C++, Java, C#,*** поддерживающие абстрактные типы данных и ООП.		
Группа языков Ada		
Ada	**1983г.**	• пакеты как средство поддержки абстрактных типов данных; • настраиваемые подпрограммs; • наличие обширных средств обработки исключительных ситуаций; • возможность параллельного выполнения процессов, оформленных в виде специальных блоков (заданий), на базе механизма рандеву.

Ada-95	–	• средствами разработки графического интерфейса пользователя; • поддержкой ООП; • наличием обширных библиотек; • усовершенствованием механизмов параллельной обработки данных посредством инкапсулированного задания.

12. КАКИЕ ВОПРОСЫ РАЗРАБОТКИ ЯП СВЯЗАНЫ С ИМЕНАМИ В ЯЗЫКЕ?

См. вопрос 5 и 18. Других идей, что тут должно быть у меня нет.

13. ДАТЬ ОПРЕДЕЛЕНИЕ СВЯЗЫВАНИЮ И ВРЕМЕНИ СВЯЗЫВАНИЯ. ПРИМЕРЫ.

Связывание – это процесс установления связи между объектом и его атрибутом (свойством). Эффективность столь сложного процесса, как обработка программного кода в ЭВМ, которая состоит их 3-х основных этапов – компиляция кода, его редактирование и выполнение – напрямую зависит от того, как организован процесс связывания переменной и ее характеристик.

Связывание может происходить:

– на этапе компиляции – ***статическое связывание***, при этом оно более надежное, т.к. не изменяется в ходе обработки и выполнения программы, занимает меньше времени, не требует дополнительных ресурсов;

– на этапе выполнения программы – ***динамическое связывание***, является более затратным по времени и по ресурсам, менее надежно с точки зрения организации процесса обработки.

14. В ЧЕМ СОСТОЯТ ПРИНЦИПЫ ФОН НЕЙМАНА?

В основе же любой современной архитектуры ЭВМ лежит так называемая машина фон Неймана, архитектурно-функциональные принципы построения которой были сформулированы венгерским математиком фон Нейманом в 1946 г.

Принципы фон Неймана:

1. Программное управление работой ЭВМ. Программа, реализующая алгоритм задачи, состоит из команд, каждая команда осуществляет единичный акт преобразования информации. Все разновидности команд данной ЭВМ составляют систему команд этой ЭВМ.

2. Принцип хранимой программы, т.е. в машине всегда должно быть запоминающее устройство (ЗУ), которое хранит команды и данные, при этом данные отправляются в арифметико-логическое устройство (АЛУ), а команды - в

устройство управления (УУ). Способ выборки команд и данных из памяти и время выборки одинаковы.

3. Наличие команд условного перехода, т.е. переход на любой участок программы в зависимости от условий. Этот принцип позволяет ввести также циклические конструкции, осуществляющие итерационные вычисления.

4. Иерархичность ЗУ. Несоответствие между быстродействием АЛУ и ЗУ частично преодолевается введением в архитектуру различной по быстродействию памяти:

- быстрая регистровая память для хранения управляющей информации в процессе счета;
- оперативная память (ОП), в которой хранятся данные и сама программа во время счета;
- долговременная память (жесткие диски, дискеты, CD и пр.).

5. Двоичная система счисления для кодирования информации, которая обрабатывается ЭВМ. Это позволяет:

- иметь достаточно простую элементную базу;
- использовать минимальную единицу измерения информации в один бит (код 1 или 0);
- реализовать операцию умножения с помощью сложения и последующего сдвига разрядов.

Принципы фон Неймана напрямую определяют не только архитектурный состав и структуру ЭВМ, но влияют также на состав языковых средств. Таким образом, эти принципы определяют метафункциональность любого языка программирования высокого уровня данного класса.

15. СТАТИЧЕСКОЕ И ДИНАМИЧЕСКОЕ СВЯЗЫВАНИЯ . ТИПЫ СВЯЗЫВАНИЯ.

Существует три способа связывания типа с переменной:

1. ***Статическое связывание.*** Реализуется с помощью:

   - операторов объявления типов;
   - правил объявления типа, принятых при разработке языка.

   Статическое связывание обычно реализуется на стадии компиляции, поэтому оно поддерживает надежность языка программирования (и в этом основное преимущество статического связывания).

2. ***Динамическое связывание.*** При динамическом связывании при объявлении переменной ее тип не указывается. Определение типа и его связывание с ней происходит при присвоении переменной значения.

Преимущества динамического способа заключаются в том, что обеспечивается значительная гибкость программирования: в одни и те же переменные в рамках одной и той же программы могут загружаться и обрабатываться данные различных типов. Однако такой способ имеет существенные недостатки:

- в силу отсутствия статического контроля типов, может случиться ситуация, когда неверные типы в правой части оператора присваивания не могут быть обнаружены ни при компиляции, ни при выполнении.
- сложная внутренняя реализация. В силу того, что проверка типов, распределение памяти делается только на этапе выполнения, каждая переменная должна иметь дескриптор для запоминания текущего типа.
- обычно реализуется через интерпретатор, т.к. сложность динамического связывания требует значительных затрат времени, его использование целесообразно только при таком способе реализации языка.

3. Логический вывод типа. Реализован в языке ML, который поддерживает и функциональное, и императивное программирование. Он использует механизм логического вывода типа. Например, в операторе `c = 3.14*r*r` константа `3.14` определяет тип переменных r и c, если константы нет, то тип не определен.

16. ПЕРЕМЕННАЯ. ТИПЫ ПЕРЕМЕННЫХ С ТОЧКИ ЗРЕНИЯ СВЯЗЫВАНИЯ.

Базовым объектом процесса программирования является ячейка оперативной памяти (ОП).

Характеристики (атрибуты) ячейки (переменной):

– символьное имя, т.е. идентификатор, под которым ячейка известна в программе, при этом при разработке нового ЯП необходимо решить такие проблемы, как:

- какова должна быть длина имен,
- разрешить ли использование верхнего и нижнего регистров,
- какие символы допустимо использовать при формировании идентификатора и т.п.

– адрес или ссылка – определяет номер ячейки ОП, который, с одной стороны, связан с символьным именем, с другой – обеспечивает доступ к ячейке со стороны операционной системы. Существует три варианта связи имени с адресом:

- одна ячейка – одно имя (оператор объявления переменной);
- одна ячейка – два имени (тип данных «объединение» (Union)); ○ одно имя – две ячейки (локальные и глобальные переменные).

– содержимое переменной (ячейки) – то значение, которое можно присвоить переменной;

– тип переменной – определяет множество значений и множество операций над этими значениями

– время жизни переменной, т.е. время, в течение которого переменная связана с ячейкой;

– область видимости – определяется фрагментом программы, т.е. теми операторами, в которых к переменной можно обратиться.

Переменная – это абстракция ячейки памяти, при этом содержимое ее может меняться множество раз в процессе выполнения программы.

Константа – это абстракция ячейки памяти, при этом ее содержимое не может быть изменено в ходе выполнения программы.

Также см. вопрос 19.

17. ОПРЕДЕЛЕНИЕ ТИПА ДАННЫХ. КОНТРОЛЬ ТИПОВ. СТРОГАЯ ТИПИЗАЦИЯ.

Тип переменной (данных) определяет (специфицирует): множество значений, которые может принимать переменная данного типа; множество операций, определенных над объектами данного типа; внутреннее представление данных и способ доступа к ним.

Следствие 1. Если дан новый тип, то можно описать и инициировать, т.е. определить значение объекта этого типа.

Следствие 2. Если даны две переменные некоторого типа, то их можно сравнить, по крайней мере, на равенство или неравенство.

Следствие 3. Если два типа отличаются с точки зрения указанных свойств, то они считаются различными.

Это определение говорит о том, что основа надежности ЯП заключается в возможности проверить, что любая операция, выполняемая над объектом данных соответствует специфицируемому типу этого объекта.

Проверка, обеспечивающая анализ совместимости типов операндов оператора, называется ***контролем типов***. Она заключается в определении типов выражений и их согласованности с правилами типизации языка программирования.

Согласованность типов операндов может быть обеспечена:

- вручную (программист следит сам, чтобы типы были согласованы);
- автоматически (с помощью функций преобразования типов или операций приведения типа), операции приведения типа могут быть:

– суживающие – более широкий тип преобразуется к более узкому (float к типу integer);
– расширяющие. Расширяющее преобразование почти всегда безопасно.

Контроль типов может выполняться:

- только при компиляции и, если все ошибки типов выявляются при компиляции, то такой язык называется с полным статическим контролем типов;
- только при выполнении – если все ошибки выявлены при выполнении, то такой язык называется с полным динамическим контролем типов;
- и при компиляции, и при выполнении (язык со смешанным контролем типов).

Правила типизации языка предполагают:

- наличие объявления типов данных;
- реализация связывания типов с переменной;
- определение типов смешанных арифметических выражений;
- наличие правил преобразования типов.

Если для языка определены правила типизации, то язык считается ***типизированным***, если эти правила для языка не определены – язык ***нетипизирован***.

При этом языки имеют **несколько уровней типизации**:

1. Слабая типизация – язык, в котором информация о типе используется только для обеспечения корректности представления данных в ячейке.

2. Сильная типизация – язык, в котором осуществляется полный контроль типов (статический, динамический или смешанный), в таком языке ошибки типов выявляются все и всегда.

3. Защитно-типизированный язык – язык, в котором операторы с возможными ошибками недопустимы.

Сильная типизация обеспечивает высокую надежность.

a) ***Pascal*** – сильнотипизированный (за исключением вариантных записей) с элементами защитной типизации;
b) ***Fortran*** – слаботипизированный, т.к. в языке нет проверки соответствия типов формальных и фактических параметров подпрограмм, а также переменных операторов EQUIVALENCE.
c) ***Ada*** – почти сильнотипизированный, но позволяет программистам отключать проверку типов;
d) ***C*** – слаботипизированный, имеются некоторые функции, параметры которых не подвергаются проверке, кроме того, для типа данных Union проверка не осуществляется, по крайней мере, в статике.

18. ОПРЕДЕЛЕНИЕ ТИПА ДАННЫХ. ЭКВИВАЛЕНТНОСТЬ ТИПОВ.

Тип переменной (данных) определяет (специфицирует): множество значений, которые может принимать переменная данного типа; множество операций, определенных над объектами данного типа; внутреннее представление данных и способ доступа к ним.

Если реализуется оператор `x := <выражение>`, то он реализуется только в том случае, когда типы операторов справа и слева эквивалентны, т.е. необходимо определить правила вычисления эквивалентности типов данных.

Существуют ***два способа определения эквивалентности типов***, которые принято использовать для структурных типов данных:

1. Эквивалентность (совместимость) по структуре или структурная эквивалентность – две переменные имеют совместимые типы в том случае, если у их типов одинаковые структуры.

2. Эквивалентность типов по имени или именная эквивалентность – две переменные имеют совместимые типы в том случае, если имена их типов одинаковы (в независимости от того, какова их структура и множество значений).

С точки зрения удовлетворения правилам эквивалентности, в языках вводится понятие производных типов и подтипов.

- ***Производный тип*** строится на основе базового типа путем определения именного, включающего логически связанные объекты, задачи. Производный тип наследует значения и операции базового типа.
- ***Подтипы*** позволяют ограничивать множество значений и множество операций базового типа.

19. ВРЕМЯ ЖИЗНИ ПЕРЕМЕННОЙ, ОБЛАСТЬ ВИДИМОСТИ СТАТИЧЕСКИЙ И ДИНАМИЧЕСКИЙ ОБЗОРЫ ДАННЫХ.

Время жизни переменной – это время, в течение которого переменная связана с ячейкой оперативной памяти.

Размещение переменной в памяти предполагает:

- сопоставление реального адреса с именем переменной;
- занесение в эту ячейку конкретного значения.

Удаление из памяти предполагает открепление адреса ячейки ОП от имени переменной.

С точки зрения времени жизни, переменные могут быть:

1. Статические, связываются с ячейкой на стадии компиляции и остаются связанными с той же ячейкой до конца выполнения программы (глобальные);

Достоинства: эффективная прямая адресация; при выполнении программы не затрачивается время на размещение и удаление из памяти.

Недостатки: уменьшается гибкость программирования; не поддерживаются рекурсии; невозможность совместного использования одной и той же ячейки.

В языках FORTRAN-1, -2, -4 все переменные были статическими. В языках C, C++, Java реализовано гибкое управление памятью посредством специальных модификаторов. Например, для объявления статических переменных существует модификатор static.

2. Динамические. Это безымянные ячейки из, так называемой, «кучи», размещаемые и удаляемые с помощью явных команд периода выполнения, которые определяются программистом. При этом связывание с памятью – динамическое, связывание с типом – статическое. Обращаться к таким переменным возможно только с помощью указателей и специальных функций. Например, в языке C, такими функциями являются malloc() для распределения памяти и free() для удаления.

Явные динамические переменные часто используются в таких динамических структурах как связные списки и деревья, которым необходимо расти и\или сокращаться во время выполнения программы.

Недостатки: корректное использование указателей и ссылок требует от программиста высокого профессионализма.

3. Автоматические – «живут» внутри блока или подпрограммы, размещаются в особом разделе оперативной памяти – стеке – при обращении к подпрограмме и удаляются из памяти при завершении работы. Связывание автоматической переменной с типом происходит статически при компиляции на основе оператора объявления. Связывание с памятью – при выполнении программы. Позволяют использовать рекурсии, т.к. рекурсивным подпрограммам требуется некоторая локальная память, чтобы каждая активная копия рекурсивной подпрограммы имела свою версию локальных параметров.

Недостатки: существенные затраты времени на размещение и удаление переменных на стадии выполнения программы.

4. Неявные динамические – это переменные, размещение которых происходит при загрузке в них конкретных значений.

Преимущества: гибкость при использовании оперативной памяти, т.к. в разные моменты времени можно использовать одну и ту же ячейку для разных типов данных.

Недостатки: атрибуты таких переменных, в том числе, типы, диапазоны индексов массивов и т.п., определяются в динамическом режиме, поэтому необходимо формировать и поддерживать дескриптор, что требует существенных временных затрат. Кроме того, использование неявных динамических переменных влечет высокую ненадежность программы.

20. СРЕДА ССЫЛОК. ОБЪЯСНИТЬ, ЕСТЬ ЛИ РАЗНИЦА МЕЖДУ ВРЕМЕНЕМ ЖИЗНИ И ОБЛАСТЬЮ ВИДИМОСТИ ПЕРЕМЕННОЙ.

Средой ссылок оператора называется совокупность всех имен, видимых в операторе.

СОД: Среда ссылок при статическом обзоре состоит из:

- области видимости его локальных переменных,
- совокупности областей видимости статических предков.

ДОД: Среда ссылок в языке с динамическим обзором состоит из:

- локально объявленных переменных,
- переменных всех других активных на данный момент подпрограмм.

При этом некоторые переменные активных процедур могут быть скрыты от среды. Новые активации процедур могут скрывать переменные в предыдущих активациях.

Область видимости – фрагмент программы, в котором переменная видима, т.е. определены ее атрибуты и к ней можно обратиться.

Существует два способа реализации области видимости:

- в виде ***статического обзора данных (СОД)***,
- в виде ***динамического обзора данных (ДОД)***.

Статический обзор данных – это связывание глобальных (нелокальных) переменных с атрибутами (т.е. определение области видимости переменных) на стадии компиляции. Он создается определениями главной программы, подпрограмм и блоков и определяет механизм вложенности подпрограмм или блоков друг в друга.

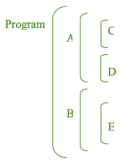


Рисунок 5. Схема областей видимости главной программы Program, подпрограмм A,B,C,D,E.

- Когда в языке со статическим обзором данных компилятор обнаруживает переменную, ее атрибуты определяются путем поиска объявившего ее оператора.
- Локализованные в блоке имена могут совпадать с ранее объявленными глобальными переменными. В этом случае считается, что локальное имя «закрывает» глобальное и делает его недоступным.
- Основной недостаток СОД заключается в том, что все переменные, объявленные в главной программе, видимы во всех подпрограммах иерархии и избежать этого нельзя (только если закрыть локальной).

Динамический обзор данных. При динамическом обзоре данных видимость переменных определяется последовательностью вызовов подпрограмм, а не их структурной вложенностью.

- Определяющим фактором для реализации механизмов видимости здесь является иерархия активаций подпрограмм. Это означает, что все объявления переменных инициируются не при компиляции, а в динамике. Поэтому динамический обзор всегда реализуется только при выполнении программы.
- Тип переменной *x* здесь определяется последовательностью активаций процедур. При ссылке на переменную *x* во время выполнения программы начинается поиск среди локальных переменных процедуры. Если такого объявления не находится, то для определения типа переменной *x* обращение идет не к структурной иерархии, породившей процедуру, а к иерархии активаций процедур, вызванных ранее (динамическим родителям), и используются их объявления, начиная с нижнего уровня.

Понятия времени жизни переменной и области видимости тесно связаны, но всё же различаются: первое – это время, в течение которого переменная существует как таковая, а второе – часть программы, в которой можно обратиться к данной переменной без ошибок.

21. ЧТО ТАКОЕ ДЕСКРИПТОР. ПРИМЕРЫ ДЕСКРИПТОРОВ.

Дескриптор – это специальная область в памяти устройства для хранения служебной информации о программах и подпрограммах.

Дескриптор одномерного массива:

Имя
тип элемента
тип индекса
начальный индекс
конечный индекс
адрес 1-ого элемента

Дескриптор записи имеет следующий вид

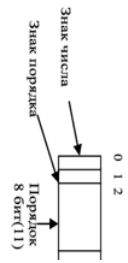
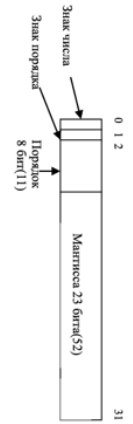
Имя записи
Имя поля 1
Тип поля 1
смещение поля 1
⋮
Имя поля k
Тип поля k
Смещение поля k
Адрес 1-го поля

Дескриптор такой строки имеет следующий вид:

<имя>
Общая длина
Текущая длина
Адрес 1-го байта

Выполнение некоторых механизмов подпрограмм обеспечиваются с помощью специального дескриптора, который называется *запись активации подпрограммы.*

22. ПРЕИМУЩЕСТВА И НЕДОСТАТКИ ЭЛЕМЕНТАРНЫХ ТИПОВ ДАННЫХ.

Название типа / *Название критерия*	**Целый (integer)**	**Плавающий (real)**	**Логический (boolean)**	**Символьный (character)**
1. Формат объявления	**C**: <тип> <имя> **Pascal**: var <имя> : <тип>	**C:** <тип> <имя> **Pascal**: var <имя> : <тип>	**C**: <тип> <имя> **Pascal**: var <имя> : <тип>	**C**: <тип> <имя> **Pascal**: var <имя> : <тип>
2. Поддержка на аппаратном уровне	Целочисленные типы реализуются непосредственно на аппаратном уровне.	Поддерживается на аппаратном уровне.	–	–
3. Представление во ВП			Булев тип данных может быть реализован и храниться в памяти с использованием только одного бита, но обычно используется минимальная адресуемая ячейка памяти (обычно байт или машинное слово), как более эффективная с точки зрения быстродействия единица хранения при работе с регистрами процессора и оперативной памятью.	Символьный тип данных представляет собой один символ, внутреннее представление которого в компьютере реализуется одной из следующих систем кодирования: ASCII – код в диапазоне 0 – 127 занимает 7 бит, – код в диапазоне 0 – 255 занимает 8 бит (байт) Unicode – код символа реализуется двумя байтами, занимает 16 бит.
4. Диапазон (значений видимо?)	В Паскале: диапазон: ±32768.	Большинство ЯП содержат два типа: float (real) – стандартный размер, обычно в 4 байта памяти; и double – мантисса занимает в 2 раза больше бит памяти. Примерно от 10^{-39} до 10^{38}.	Диапазон данного типа состоит из двух значений: true и false. Значения представляются в машине минимальной ячейкой памяти (байт).	В Паскале: все символы 8-разрядной кодировки для char.
5. Операции	Сложение (+), вычитание (-), умножение (*), деление (/), взятие остатка от деления (mod), возведение в степень (**), сравнение на равенство (=), сравнение на неравенство (/=), сравнение на меньше (<), сравнение на меньше или равно (<=), сравнение на больше (>), сравнение на больше или равно (>=), унарный плюс (+), унарный минус (-), абсолютное значение (abs).	Основные операции над данными вещественного типа: сложение (+), вычитание (-), умножение (•), деление (/), возведение в степень (**), сравнение на равенство (=), сравнение на неравенство (/=), сравнение на меньше (<), сравнение на меньше или равно, (<=), сравнение на больше (>), сравнение на больше или равно (>=), унарный плюс (+), унарный минус (-), абсолютное значение (abs).	Три стандартные логические операции применяются к операндам логического типа, давая в результате логическое значение: логическое дополнение (not), логическое «И» (and), логическое «ИЛИ» (or). Основное назначение логического типа состоит в реализации условий для условного оператора и оператора цикла.	Числовая и символьная интерпретация типа данных **char** позволяются использовать обычные операции для работы с целыми числами для обработки символов текста. Тип данных **char** не имеет никаких ограничений на выполнение операций, допустимых для целых переменных: от операций сравнения и присваивания до арифметических операций и операций с отдельными разрядами.

6. Проблемы	Целочисленное переполнение	Проблема представления произвольных множеств символов.	–	1. Длина имен 2. Чувствительность к регистру 3. Возможный алфавит

23. МНОЖЕСТВА. ОПЕРАЦИИ НАД МНОЖЕСТВАМИ.

Данные множественного типа представляют собой неупорядоченную совокупность отдельных величин некоторого порядкового или перечислимого типа:

```
type <имя типа> = set of <значения базового перечислимого порядкового типа>;
```

Таким образом, ***переменная типа множество*** является объектом, содержащим неупорядоченный набор значений, которые принадлежат некоторому базовому типу.

Реализация множеств осуществляется битами машинного слова, которое может включать различное количество байт в разных архитектурах ЭВМ. Поэтому при переносе пользовательской программы, включающей множества, на другой компьютер возникает проблема совместимости. Мощность множества определяется величиной машинного слова.

Операции над переменными множественного типа по смыслу идентичны операциям над математическими множествами:

1. *Умножение переменных* есть пересечение двух множеств;

2. *Сложение переменных* представляет собой объединение множеств;

3. *Разность переменных* – это элементы 1-ого множества, которые не принадлежат 2-ому;

4. *Эквивалентность*. Две переменные множественного типа эквивалентны, если все их элементы одинаковы, причем порядок следования безразличен (true, если множества эквивалентны).

5. *Неэквивалентность* <, >. Результат операции есть true, если множества не эквивалентны.

6. *Включение* <= дает true, если первое множество включено во 2-ое;

7. *Принадлежность элемента in*. Результат операции равен true, если элемент или выражение принадлежит множеству, иначе – false.

Реализован множественный тип только в семействе вовеки веков лучшего языка на этой планете – Pascal!

24. ТИП СТРОКА. ОПЕРАЦИИ, РЕАЛИЗАЦИЯ, ВАРИАНТЫ СТРОК.

Символьные строки – это последовательности символов. При этом реализация в разных языках программирования различна. Так, в языках C, C++ строки вводятся как символьные массивы типа cahr. Набор операций над такими массивами вызывается из стандартной библиотеки string.h. Строки символов здесь завершаются специальным символом, который называется «нуль-байт» - /0. Этот нуль-байт заносится автоматически при создании строки символов. В языках Pascal (с версии 5.0), Delphi , Ada, Fortran77, 90, Бэйсик, Java тип string является встроенным.

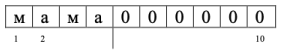
Операции над строками делятся на два класса:

1. Встроенные: конкатенация (слияние строк), операции отношений и присваивания;

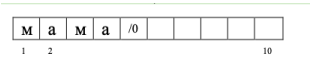
2. Реализованные в виде функций: выделение подстроки, определение длины строки, сравнение строк, определение индекса элемента и др.

Реализация строк:

1. Статическая реализация. Длина может быть статической и задаваться в объявлении: `var str : string[10].`



2. Строки переменной длины с ограничением. Текущая длина при этом ограничивается специальным символом, в C и C++ ставится нулевой байт.



3. Строки с переменной неограниченной длиной (SNOBOL4, Perl). Такая реализация требует больших временных затрат на размещение и удаление из оперативной памяти, но обеспечивает высокую гибкость.

25. ХАРАКТЕРИСТИКА ПОРЯДКОВОГО (ПЕРЕЧИСЛИМОГО) ТИПА.

Перечислимый тип данных – это такой тип, который имеет следующие характеристики:

- задается набором элементов,
- в качестве элементов используются строковые константы,
- количество элементов не должно быть больше 256,
- внутреннее представление каждого элемента совпадает с типом byte.

Формат описания:

```
Type
     <имя типа> = (<стр.константа1>,< стр.константа 2>,...,<стр.константа k>)
```

Операции над данным типом также могут быть как встроенные (сравнения на « = », « ≠»), так и реализованные в виде функций: определение номера по значению; определение значения по номеру; определение последующего (предыдущего) значения; определение min/max значений. Однако, если операции « = », « ≠» вводятся всегда, т.к. они семантически оправданы, то другие операции отношения могут не иметь смысла. Поэтому вопрос о том, какие операции будут представлены в ЯП, для разработчиков данного языка представляет определенную сложность.

***Проблемы*:** при использовании данного типа возникают некоторые трудности, требующие нестандартных субъективных подходов, например, может ли одна и та же символьная константа появиться в разных типах. Одним из таких решений может быть следующее правило: чтобы разрешить использовать в разных типах одну и ту же символьную константу, необходимо указывать имя типа перед значением константы по формату <имя типа>.<стр.константа>.

26. ОПРЕДЕЛЕНИЕ МАССИВА, ВАРИАНТЫ МАССИВОВ. ПОНЯТИЕ АССОЦИАТИВНОГО МАССИВА. ОПЕРАЦИИ, РЕАЛИЗАЦИЯ МАССИВОВ.

Массив – это поименованная область оперативной памяти (ОП), ячейки в области перенумерованы и являются переменными одного типа. Элемент массива представляет собой индексированную переменную:

```
var
     <имя> : array [нач.зн. . . кон.зн.] of <тип>;
```

Тип массива включает в себя тип элемента и тип индекса.

Обращение к каждому элементу массива осуществляется по общему имени и индексу элемента: A[i].

Категории массивов. Массивы делятся на 4 категории, которые указывают когда и где выделяется память:

1. Статический массив.

Это массив, в котором диапазоны значений индексов связываются статически, размещение в памяти также статическое и осуществляется в период компиляции.

2. Фиксированный автоматический массив.

Диапазон значений связывается в статике, а размещение в памяти происходит динамически, во время выполнения.

3. Автоматический массив.

Размещение в памяти происходит динамически. Но после связывания индексов и размещения в памяти и диапазоны, и адрес памяти массива не изменяется в течение всей жизни переменной.

4. Динамический массив.

Это массив, в котором связывание индексов и размещение в динамической памяти происходит только при выполнении программы. Для этого используются специальные функции, а также механизмы на основе указателей.

Операции над массивами. Операцией над массивом называется действие, при выполнении которого массив считается единым целым.

В языке Fortran-90 операции линейной алгебры над массивами реализованы как встроенные функции (операции с матрицами, векторами).

В языке Pascal возможны операции присваивания.

В языке Ada реализованы:

- операции отношения;
- присваивание;
- конкатенация (сцепление) массивов.

Операция ***инициализации массива*** – это заполнение его начальными значениями. В языке Pascal инициализация не допускается.

Если инициализаторов меньше, то оставшиеся элементы массива не определяются. В этих языках есть особенность – компилятор сам может устанавливать длину массива на основании инициализатора, тогда длина массива устанавливается в четыре элемента.

В некоторых языках есть операция сечения, т.е. выделение из структуры массива некоторой подструктуры. Если M – матрица, то сечение может быть строка или столбец. Сечение не является новым типом, это просто способ обращения к части массива.

Ассоциативный массив – это массив с неупорядоченным множеством элементов, индексированных таким же количеством величин, называемых ключами.

Ключи должны содержаться в структуре массива, т.е. каждый элемент здесь является парой: ключом и величиной. Такие массивы используются в таких языках, как Perl, Java. В языке Perl ассоциативные массивы называются хэшами, создаются и удаляются с помощью функций хэширования. Размер хэша всегда динамический.

27. ТИП ЗАПИСЬ, РЕАЛИЗАЦИЯ, ОПЕРАЦИИ НАД ЗАПИСЬЮ.

Запись – это совокупность данных различных типов, в которой отдельные элементы идентифицируются символьными именами. Каждый элемент называется полем записи.

В силу того, что запись позволяет моделировать таблицы со столбцами различных по типу данных, впервые они были введены в язык Cobol, ориентированный на программирование бухгалтерских задач. В языке C записи называются структурами.

Формат записи:

```
type <имя> = record;
     <имя поля 1>.<тип>;
     . . .
     <имя поля n>.<тип>;
end;
```

Обращение к элементу осуществляется по формату:

```
<имя переменной>.<имя поля>;
```

Над записями осуществляются следующие виды ***операций***:

1. Сравнение на «=» и «≠».
2. Копирование полей: из записи источника – в запись результат.
3. Над полями записей определены все операции, присущие объявленному типу.

28. ОБЪЕДИНЕНИЯ, ТИПЫ ОБЪЕДИНЕНИЙ, ОПЕРАЦИИ

Объединение – представляет собой ячейку оперативной памяти, которая в разные моменты времени выполнения программы, может хранить данные различных типов.

Реализуются в Fortran в виде оператора EQUIVALENCE:

```
INTEGER <имя1>
REAL <имя2>
EQUIVALENCE (имя1, имя2)
```

В языке C введен специальный тип – объединение union:

```
union u {
     int i;
     char c;
     float r;
}
```

Под переменную типа объединение всегда выделяется ячейка памяти в соответствии с элементом максимальной длины.

Для переменной типа union определены операции сравнения на равенство и неравенство.

Это довольно ненадежный тип данных, т.к. проверку типа в объединении можно осуществить только на стадии выполнения программы, что требует к тому же больших временных затрат.

В системе данных языка Pascal тип объединение как таковой отсутствует. Однако, начиная с 5-ой версии, в нем декларированы вариантные записи, которые реализованы как размеченные объединения.

29. УКАЗАТЕЛИ, ОПЕРАЦИИ, РЕАЛИЗАЦИЯ В ЯЗЫКАХ С И PASCAL, ПРОБЛЕМЫ УКАЗАТЕЛЕЙ.

Указатель – переменная, значением которой может быть адрес ячейки памяти или особый символ 0, т.е. нулевой адрес, который используется как пометка того, что указатель свободен.

Указатели используются в следующих случаях:

1. Организация косвенной адресации для нединамических объектов. В частности, в языке С указатели широко используются при обращении к массивам, строкам, структурам. И это является одним из факторов ненадежности данного языка.

2. Для организации обращения к ячейкам динамической памяти (кучи). Переменные, размещенные в куче, не имеют имен, обращаться к ним можно только с помощью указателей и ссылок.

Как любая переменная, указатель должен иметь тип. Тип указателя определяет объем динамической памяти в байтах, выделяемой из кучи. Однако в языках существует два вида указателей:

1. Типизированные, когда тип указателя определяется типом переменной, на которую он будет указывать. В нижеприведенном примере все объявленные указатели являются типизированными.

Язык Pascal:

```
var
     i, j : ^integer;
     r : ^real;
     ii : int;
```

Язык С:

```
char *ch;
int *i, *j;
int ii;
```


2. Нетипизированные, тогда объем выделяемой памяти определяется с помощью специальных функций. Нетипизированные указатели существуют в языке Pascal, для их объявления задействован специальный тип pointer.

<u>Операции над указателями:</u>

1. ***Инициализация указателя*** – присваивание указателю адреса. Может происходить в следующих случаях:

- присваивание динамического адреса. По запросу пользовательской программы операционная система отводит блок ячеек динамической памяти и помещает адрес начальной ячейки в указатель, тем самым устанавливается связь между указателем и адресом. Для этого в языках используются специальные функции, например, Pascal – `new(i)`; С – `malloc(i);`
- обеспечение косвенной адресации для нединамического объекта, например, массива: `int*(A+индекс)=*A+индекс*длина;`
- посредством операции взятия адреса (операция «амперсант»): `i=&ii;`

2. ***Разыменование указателя*** – взятие (присваивание) значения из/в ячейки, на которую указывает указатель.

3. ***Освобождение динамической памяти***. Освободить память означает, что необходимо возвратить динамическую переменную в кучу, т.е. отдать ее адрес в распоряжение операционной системы. Для этого существуют специальные функции: `Dispose (i)` (в Pascal), `Free (i)` (в С). Следует заметить, что, вообще говоря, эти функции не освобождают указатель, они только возвращают динамическую переменную в кучу.

4. ***Освобождение указателя*** – помещение в него нулевого адреса (на примере языка Pascal). Только после того, как в указатель занесли 0, его можно использовать для другого адреса.

```
const c : pointer = NIL //нулевой адрес в Pascal
     ...
     dispose(i);
     i = c;
```

5. Присваивание. Операцию присваивания можно реализовывать только для однотипных указателей.

6. Операции отношений: >, <, =, <=, >=.

7. Арифметические операции над указателями (осуществляются в адресном пространстве):

```
int *p1; // длина int равна 4-м байтам
p1 = 2000;
p1 = p1+3 //оператор поместит в p1 адрес 2012
```

8. Операция идентификация поля записи.

При использовании указателей возникают ***две основные проблемы***:

1. Висячий указатель – указатель, содержащий адрес динамической переменной, которая уже удалена из доступной памяти и возвращена в кучу. Они создаются в следующей ситуации:

```
шаг 1: пусть p1 указывает на некоторую ячейку;
шаг 2: p2 = p1; //присваиваем p2 значение p1
шаг 3: возвращаем ячейку в кучу и освобождаем p1, но p2 при этом
продолжает содержать адрес ячейки.
```

Такая ситуация может привести к следующим последствиям:

- в эту ячейку могут загрузиться новые данные или данные другой пользовательской программы, а, поскольку p2 имеет доступ к данной ячейке, то есть большая вероятность их испортить;
- более фатальная ситуация может возникнуть, если эту ячейку (с доступом через p2) захватит коварная операционная система.

2. Потерянные переменные – это ячейки, размещенные в динамической памяти и переставшие быть доступными для пользовательской программы. Такие переменные называются мусором. Они создаются в случае, если:

```
шаг 1: целью указателя p1 устанавливается некоторая динамическая
переменная,
шаг 2: целью того же указателя p1 становится другая динамическая
переменная.
```

Тогда первая ячейка становится недоступной для программы, т.к. связь с ней теряется. Но она недоступна также и для операционной системы, т.к. тоже выделена программе. Эта проблема называется утечкой памяти и затрагивает ЯПВУ, в которых требуется явное удаление из памяти переменных.

30. ОПРЕДЕЛЕНИЕ КС-ГРАММАТИКИ.

КС-грамматика G представляет собой кортеж четырех объектов: $G = \{T, V, S, P\}$, где

1. ***Множество T*** – это конечное множество символов, из которых состоят цепочки языка. Множество T называется ***множеством терминалов*** и представляет собой алфавит языка.

2. ***Множество V*** – множество переменных грамматики, при этом любая переменная представляет собой класс цепочек языка, обладающих единым метасмыслом. Множество V называется еще ***множеством нетерминальных символов***.

3. $\mathbf{S \subset V}$ – это ***стартовый (начальный) символ***, т.е. переменная, которая представляет определяемый язык. Другие переменные позволяют доопределить описываемый язык, задаваемый стартовым символом.

4. ***Множество P*** – это ***множество продукций или правил вывода***, которые представляют собой правила образования цепочек языка. Таким образом, предложения языка создаются с помощью последовательности продукций, начиная со стартового символа – этот процесс называется порождением или выводом и обозначается символами $\rightarrow$ или $\Rightarrow$.

31. ОПРЕДЕЛЕНИЕ ЯЗЫКА, ЗАДАННОГО КС-ГРАММАТИКОЙ.

Если кортеж *G* есть КС-грамматика, то язык *L(G)*, описываемый этой грамматикой, представляет собой множество терминальных цепочек *w*, порожденных из стартового символа *S*: $L(G) = \{w \in T: S \Rightarrow w\}$.

32. БНФ

Основные понятия формы Бекуса-Наура (БНФ).

Язык программирования – это множество последовательностей символов некоторого алфавита, удовлетворяющих правилам синтаксиса и задающих последовательность вычислений в соответствии с правилами семантики.

Алфавит ЯП – набор символов, включающий в себя буквы, цифры, специальные символы и лексемы.

Лексема – синтаксическая единица нижнего уровня, имеющая смысл. Для языка программирования лексемами являются идентификаторы, ключевые слова, знаки операций и т.п.

Ключевые слова – множество зарезервированных в языке слов, определяют семантические понятия языковых конструкций или их частей.

Идентификаторы – символические имена, которыми именуются объекты программы.

Помимо синтаксической лексемы грамматика БНФ оперирует термином «*грамматическая лексема*», которая определяет смысл лексемы синтаксической, например, для оператора `i := i + 1` грамматическими лексемами будут являться следующие:

```
i – идентификатор;
:= – знак оператора присваивания;
+ – знак операции;
1 – текстовая константа.
```

Списки идентификаторов или объектов переменной длины описываются в БНФ следующим образом:

1) Рекурсивными правилами, т.е. правилами, в которых левая часть входит в правую:

```
<список идентификаторов> → <идентификатор> | <идентификатор>,
                                        <список идентификаторов>;
```

2) Фигурными скобками, которые указывают на то, что заключенная в них часть, может повторяться неограниченное количество раз или отсутствовать:

```
<список идентификаторов> → <идентификатор> {идентификатор};
```

Квадратными скобками обозначается необязательная часть правой части цепочки символов (оператора языка):

```
if <логическое выражение> then <оператор> [else <оператор>];
```

Круглыми скобками обозначают выбор из группы одного элемента:

```
for <переменная>:=<выражение>(to | downto)<выражение> do <оператор>;
```

33. ПОРОЖДЕНИЯ ВЫРАЖЕНИЙ И ОПЕРАТОРОВ ПРИСВАИВАНИЯ

Так же как и в КС-грамматике, в БНФ используются нетерминалы, которые определяют типы тех или иных операторов ЯПВУ. Например, оператор присваивания в терминах множества переменных V в БНФ представляется переменными `<присвоить>` и `<выражение>` следующим образом:

```
<присвоить>:=<выражение>;
```

Пример 1. Пусть требуется описать грамматику для представления языка программирования, на котором можно писать программы с тремя переменными A, B, C, операторами присваивания " := " , арифметическими выражениями со знаками операций "+", "-" и ключевыми словами begin, end.

Грамматика представляется следующим образом:

T={A, B, C, :=, +, - , ; , begin, end}

V={<программа>, <оператор>, <список операторов>, <переменная>, <выражение>}

S={<программа>}

P : <программа>→begin→<список операторов>end

< список_операторов > → < оператор > |< оператор >;

< список_операторов >

< оператор > → < переменная > := < выражение >

< переменная > → A | B | C

< выражение > → < переменная > + < переменная >

|< переменная > - < переменная >

| < переменная >

Пример порождения в описанной грамматике:

< программа > => begin < список_операторов > end

=> begin |< оператор >; < список_операторов > end

=> begin < переменная >:= < выражение >;< список_операторов > end

=> begin A := < выражение > ; < список_операторов > end

=> begin A := < переменная > + < переменная > ;

< список_операторов > end

=> begin A := B + < переменная >; < список_операторов > end

=> begin A := B + C ; < список_операторов > end

=> begin A := B + C ; < оператор > end

=> begin A := B + C ; < переменная > := < выражение >end

=> begin A := B + C ; B := < выражение >end

=> begin A := B + C ; B := < переменная >end

=> begin A := B + C ; B := C end

Каждая строка вывода называется ***сентенциальной формой***. Здесь реализован левосторонний порядок вывода (левое порождение), т.е. подстановка формул осуществлялась слева (может быть и правосторонний вывод).

Сентенциальная форма, в которой присутствуют только лексемы и терминалы называется ***порожденным предложением***.

34. ОПРЕДЕЛЕНИЕ СИНТАКСИЧЕСКОГО ДЕРЕВА, КРОНЫ ДЕРЕВА. ПРИМЕРЫ.

Порождения можно представить в виде синтаксического дерева или дерева разбора, которое графически показывает, как группируются символы (терминалы) в цепочки языка.

Синтаксическое дерево – это графическое представление порождения в виде иерархического списка, в котором:

1. ***Корень дерева*** – это стартовый символ грамматики - S.

2. ***Узлы (вершины) дерева*** могут быть либо внутренними, т.е. являться переменными, принадлежащими множеству V, либо конечными – терминалами, которые принадлежат множеству T. При этом внутренние узлы всегда имеют наследников, конечные называются ***листьями дерева*** и потомков не имеют.

3. Если внутренний узел помечен переменой $A \in V$, а ее наследники отмечены листьями $x_1, x_2, ..., x_n$, то A является ***продукцией***, порождающей цепочку $x_1,x_2,...,x_n$ ($A \rightarrow x_1,x_2,...,x_n$), причем $x_1,x_2,...,x_n$ выписаны слева направо по дереву.

Цепочка, выведенная из корня и выписанная слева направо из отметок листьев, называется ***кроной дерева.***

Терминальная цепочка принадлежит языку грамматики тогда и только тогда, когда она является кроной, по крайней мере, хотя бы одного дерева разбора.

Пример 2. Пусть дана грамматика:

```
<присвоить> → < идентификатор > := <выражение>
< идентификатор > → A | B | C
<выражение> → < идентификатор > + <выражение>
      | < идентификатор > * <выражение>
      | (< выражение > )
      | < идентификатор >
```

Здесь, например, оператор A := B * (A + C) порождается следующим выводом

```
<присвоить> => < идентификатор > := <выражение>
=> A := <выражение>
=> A := < идентификатор > * <выражение>
=> A := B * <выражение>
=> A := B * ( <выражение> )
=> A := B * (< идентификатор > + <выражение> )
=> A := B * ( A + <выражение> )
=> A := B * ( A + < идентификатор > )
=> A := B * ( A + C )
```

И его синтаксическое дерево выглядит следующим образом:

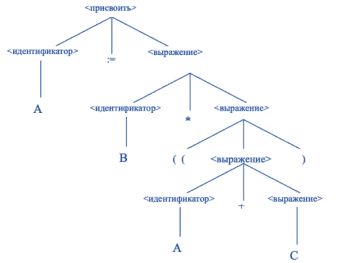


Рисунок 6.1. Пример синтаксического дерева

35. ПРОБЛЕМЫ НЕОДНОЗНАЧНОСТИ ГРАММАТИК ВЫРАЖЕНИЙ.

Для некоторых грамматик можно найти терминальные цепочки с несколькими деревьями разбора или с несколькими левыми или правыми порождениями. Такие грамматики являются ***неоднозначными***.

Пусть задана грамматика оператора присваивания:

```
<присвоить> → < идентификатор > := <выражение>
< идентификатор > → A | B | C
<выражение> → < выражение > + <выражение>
      | < выражение > * <выражение>
      | (< выражение > )
      | < идентификатор >
```

Эта грамматика неоднозначна, т.к. вывод цепочки A:= B+C*A можно представить двумя деревьями (рис.6.2):

Причины неоднозначности грамматики выражений заключаются в следующем:

1. В грамматике не учтены приоритеты арифметических операций.

2. Не определен порядок выбора правил из множества P при выводе выражения.

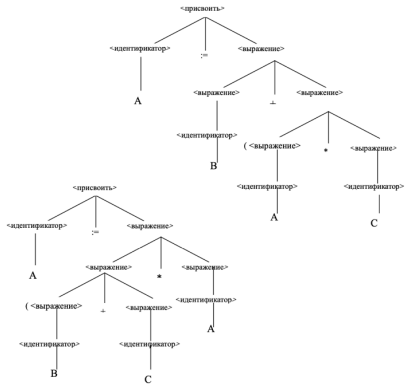


Рисунок 6.2. Синтаксические деревья к выводу цепочки A:= B+C*A в неоднозначной грамматике

Устранение неоднозначности состоит во введении в грамматику дополнительных переменных:

1. Множителей (сомножителей или факторов),
2. Слагаемых (термов).

Множитель – это выражение, которое не может быть разделено на части никакой примыкающей операцией, ни “*”, ни “+”. Множителями для нашей грамматики могут быть идентификаторы или выражения в скобках.

Терм – это выражение, которое не может быть разорвано операцией “+”. Для нас терм – это произведение нескольких факторов F*F*.... Тогда выражение – это любое возможное выражение, которое представляет сумму одного или нескольких термов.

36. КОНЦЕПЦИЯ ИНКАПСУЛЯЦИИ В ЯП. ИНКАПСУЛЯЦИЯ В ООП.

Там вопрос про развитие концепции, так что вставил этот кусок для ознакомления, самое важное – пониже!

Концепция объектно-ориентированного программирования (ООП) уходит корнями в язык SIMULA67, который был предназначен исключительно для моделирования систем. Основным слабым местом языков того времени при использовании их для моделирования были подпрограммы. Для моделирования требовались подпрограммы, позволяющие перезапускать их с того места, на котором их выполнение было ранее прервано. Подпрограммы с таким типом управления называются сопрограммами, реализация их была сделана в языке АЛГОЛ-60.

Для поддержки сопрограмм в языке SIMULA67 была разработана конструкция класса. Это усовершенствование положило начало нашим понятиям об абстрактных типах данных и ООП. Класс в языке SIMULA67 поддерживал инкапсуляцию (объединение данных и кода их обработки), обладал некоторыми свойствами наследования, определял абстрактный тип данных, но не ограничивал доступ клиента к сущностям класса, т.е. не обеспечивал надежного сокрытия информации в классе.

Идеи ООП, возникшие при разработке SIMULA67, нашли свое развитие и воплощение в языке Smalltalk80, который явился первым чисто объектным языком. На уровне объектов здесь реализовано все: от целочисленных констант до больших сложных систем программного обеспечения.

Концепции ООП как методология программирования в настоящее время получила широкое распространение. Поэтому основные императивные языки (C, Pascal, Fortran, Ada) имеют свои объектно-ориентированные диалекты (C++, Delphi, Fortran90, Ada95 и пр.).

(!) Итак, ***методология объектно-ориентированного программирования*** базируется на следующих основных понятиях:

1. Инкапсуляция,
2. Абстрактный тип данных (АТД),
3. Объект как основной элемент абстракции в отличие от переменной основного элемента абстракции структурного программирования,
4. Наследование,
5. Полиморфизм.

Инкапсуляция – это способ объединения в единое целое данных и кода, который эти данные обрабатывает, ее общая структура представлена на рисунке:

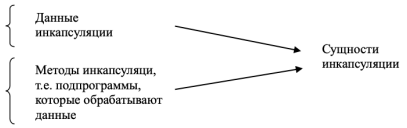


Рисунок 5.1. Структура инкапсуляции

При организации инкапсуляции возникает ***задача обеспечения безопасности сущностей инкапсуляции***. Для этого необходимо:

1. Осуществлять проверку типов интерфейсов при обращении к инкапсуляциям,
2. Разграничить доступ к данным и методам инкапсуляции со стороны других синтаксических единиц.

Чтобы обеспечить безопасность инкапсуляции существует ***система доступа к сущностям инкапсуляции.*** Она заключается в том, что сущности с точки зрения их доступности могут быть представлены следующими типами:

– private – видимы и доступны только внутри данной инкапсуляции,
– public – видимы и доступны для других программных единиц, осуществляют внешний интерфейс,
– protected – доступны только потомкам по определенным правилам.

37. АБСТРАКТНЫЙ ТИП ДАННЫХ В ООП

Развитие абстракции данных в языках программирования связано с представлением на уровне программного кода объектов оперативной памяти. При этом выделяются следующие этапы:

1. Ячейка – переменная одного типа. Абстракция данных в языке началась с понятия переменной, которая представляет собой абстракцию ячейки ОП. При этом одной переменной определенного типа ставится в соответствие одна (условно) ячейка соответствующего типа.

2. Ячейка – разнотипные переменные. Оператор EQUIVALENCE в FORTRAN или Union в C, C++ позволяют в разные моменты времени в течение работы пограммы загружать в одну и ту же ячейку переменные различных типов.

3. Область ячеек – объединение однотипных переменных – массив, который представляет область ячеек памяти одного типа.

4. Область ячеек – объединение разнотипных переменных. Возможность описания области ячеек памяти одной структурой разнотипных объектов (например, таблица данных разного типа), так в ЯП появился тип запись или структура.

В языках объектно-ориентированного программирования произошел синтез абстракции данных и абстракции процесса на принципах инкапсуляции. Возникла идея определить новый тип, инкапсулируя какой-либо тип данных и методы их обработки.

Таким образом, ***абстрактный тип данных (АТД)*** – это инкапсуляция данных одного типа и операций (методов) для их обработки. При этом следует иметь в виду, что в инкапсуляции, конечно, описывается несколько видов данных (переменные, массивы, структуры), но среди них есть один, обработка которого с помощью операций инкапсуляции является целью создания АТД с точки зрения семантики задачи.

Абстрактный тип данных (АТД) — это тип данных, который удовлетворяет следующим условиям:

- определение типа и операции над объектами данного типа содержатся в одной синтаксической единице; переменные же данного типа можно создавать и в других модулях;
- внутренняя структура объектов данного типа скрыта от программных модулей, использующих этот тип, так что над такими объектами можно производить лишь те операции, которые прямо предусмотрены в определении типа.

Встроенные операции, которые можно выполнять над объектами АТД:

- конструктор и деструктор объектов. ***Конструкторы*** используются для инициализации вновь создаваемых объектов. ***Деструкторы*** используются для освобождения областей динамической памяти, которые могут быть заняты объектами абстрактного типа.
- присваивание,
- проверка на равенство и неравенство.

В ООП по аналогии с языком SIMULA67 абстрактный тип данных принято называть ***классом***. Экземпляр класса (переменная типа класса) является ***объектом***. Класс – шаблон для создания объекта. Формат описания класса следующий (нотация языка C++):

```
Class < имя класса > {
     Private :
          < приватные сущности класса >
     Protected :
          < защищенные сущности класса >
     Public :
          < общие сущности класса >
} [ список объектов класса ];
```

Здесь важно понять преемственность классического понятия типа данных в императивном и объектном смыслах:

1. Отношения между классом и объектом такие же, как и между переменной и типом в императивном языке. Поэтому синтаксис оператора описания объекта (экземпляра класса) такой же, как и синтаксис оператора описания переменной

2. Методы класса являются операциями, сконструированными пользователями и, фактически, дополняющие встроенный в язык набор операций классических императивных типов данных.

Метод – это действие, которое можно выполнить над объектом. Вызвать метод (обратиться к нему) означает послать сообщение объекту. Весь набор методов объекта называется протоколом (интерфейсом) сообщения. Синтаксис сообщения следующий:

```
<имя объекта> . < имя метода>.
```

Программные модули, которые используют некоторый АТД, называются ***клиентами этого типа***. Все вычисления в среде ООП выполняются с помощью передачи сообщения от клиента к объекту для вызова одного из его методов.

Атрибуты объекта (свойства объекта) – это общие данные (public). Чтобы изменить характеристики объекта, надо изменить его свойства:

```
< имя объекта> . < свойство > = < значение >
```

Все экземпляры класса совместно используют единый набор методов, но каждый экземпляр получает свой собственный набор данных класса (в отличие от подпрограмм) (рис. 5.2).

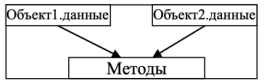


Рисунок 5.2. Схема представления АТД в оперативной памяти

38. КОНЦЕПЦИЯ НАСЛЕДОВАНИЯ В ООП

Класс, который определяется через наследование от другого класса, называется ***производным классом, или подклассом, или потомком***.

Класс, от которого производится новый класс, называется ***родительским классом, или суперклассом, или предком***.

При наследовании существует три возможности:

1. Потомок наследует все сущности (переменные и методы) родительского класса. Это наследование можно усложнить, введя управление доступом к сущностям родительского класса в соответствии с рисунком 5.3.

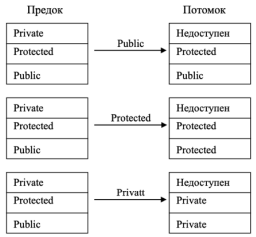


Рисунок 5.3. Схема наследования системы безопасности

2. Потомок модифицирует некоторые методы предка. Модифицированный метод имеет то же самое имя и часто тот же самый протокол, что и метод, модификацией которого он является:

```
     < имя метода > < протокол > = < имя старого метода >
                                        < протокол старого метода >;
```

Наиболее общее предназначение замещающего метода – выполнение операции, специфической для объектов производного класса, но не свойственной для объектов родительского класса.

3. Потомок добавляет новые методы. Если потомок имеет один родительский класс, то этот процесс называется одиночным наследованием и представляется в виде дерева наследования.

Если класс имеет несколько предков, то такой процесс называется ***множественным наследованием (multiple inheritance)***. Взаимоотношения классов при множественном наследовании можно изобразить с помощью графа наследования. Синтаксис при этом следующий

```
Class < имя потомка > : < список предков > {
     ......
     } [ список объектов] ;
```

Здесь `< список предков > = < режим доступа > < имя предка1 >, < режим доступа > < имя предка2 > . . . < режим доступа > < имя предка n >.`

Разработка программы для объектно-ориентированной системы начинается с определения иерархии классов, описывающей отношения между объектами, которые войдут в программу, реализующую поставленную задачу. Чем лучше эта иерархия классов соответствует проблемной части, тем более естественным будет полное решение.

39. ВИДЫ ПОЛИМОРФИЗМА В ЯП

Понятие ***полиморфизма*** означает, что одно и то же имя может использоваться для логически связанных, но разных целей, т.е. имя определяет набор действий, которые в зависимости от типа данных могут существенно отличаться.

Полиморфизм широко используется в императивных языках на уровне знаков операций. Например, привычные нам символы арифметических операций «+», «-», «*», «/» и др. являются перегруженными (их реализация осуществляется в идеологии полиморфизма) в любом ЯПВУ, т.е. один и тот же символ олицетворяет и действие для целого типа, и для вещественного. А в языках С, С++ реализована перегрузка операций <<, >>, которые имеют двоякий смысл – поразрядный сдвиг или вывод на консоль. Когда перегружается знак операции, компилятор анализирует тип операндов и в зависимости от этого делает выбор.

Помимо операций в императивном подмножестве языка программирования полиморфизм реализуется также и для определенных видов подпрограмм. Это – перегруженные и настраиваемые подпрограммы.

1) Перегруженная подпрограмма представляет собой разновидность специального полиморфизма, реализуемого также в статике.

Перегруженная подпрограмма – это подпрограмма, имя которой совпадает с именем другой подпрограммы, но при этом каждая версия перегруженной подпрограммы должна иметь свой уникальный протокол, т.е. она должна отличаться от других версий количеством, порядком, типами своих параметров или типом возвращаемого значения, если она является функцией.

Ярким примером перегруженных подпрограмм являются подпрограммы (процедуры в Pascal или функции в С) ввода/вывода данных, которые имеют такую значимость в языках программирования, что воспринимаются наряду с остальными операторами языка как обычные команды.

2) Настраиваемые подпрограммы представляет собой динамическую разновидность параметрического полиморфизма.

Настраиваемая подпрограмма – это подпрограмма, при каждом вызове которой загружаются фактические значения разных типов, т.е. параметрами здесь являются типы формальных переменных. Изначально типы формальных параметров не определяются, они связываются в динамике с типами фактических значений.

Такие подпрограммы позволяют использовать один и тот же алгоритм для данных различных типов.

40. КОНЦЕПЦИЯ СВЯЗЫВАНИЯ В ООП

В ООП под ***связыванием*** понимается, прежде всего, связывание сообщения с определением метода (обращение к методу). При этом возможны два варианта

– статическое связывание
– динамическое связывание, которое реализуется посредством:
  а) механизмов настраиваемых подпрограмм
  б) полиморфных переменных.

Полиморфная переменная – это переменная типа суперкласса (базового класса), которая используется в потомке (подклассе) для обращения к замещаемым или виртуальным методам.

Замещаемый метод – это модифицированный в потомке метод суперкласса, который имеет то же самое имя и часто тот же самый протокол, что и метод, модификацией которого он является.

```
< имя метода > < протокол > = < имя старого метода >
                                          < протокол старого метода >;
```

Виртуальный метод – это метод, прототип которого (часто без определения) включен в суперкласс (в предка), а определение или переопределение, т.е. описание тела метода, содержатся в потомках. При этом прототипы методов в разных потомках одинаковы. Если прототипы методов различны, то механизм виртуальности не включается. Формат описания виртуального метода следующий:

```
Virtual < тип > < имя метода> [ ; = 0 ] [ { . . . } ]; .
```

Такие замещаемые или виртуальные методы вызываются через полиморфную переменную и этот вызов динамически связывается с определением метода в соответствующем потомке. Полиморфная переменная определяется через указатель на базовый класс и ссылку.

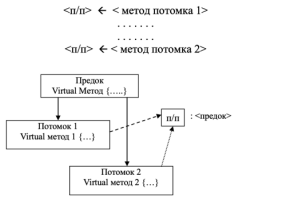


Рисунок 5.4. Схема обращения к виртуальному методу

Таким образом, полиморфная переменная – это своеобразный параметр, через который в разные моменты времени можно обращаться к методам разных потомков.

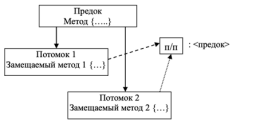


Рисунок 5.5. Схема обращения к замещаемому методу

41. ПОНЯТИЕ ОБЪЕКТА, МЕТОДА, СВОЙСТВА, СОБЫТИЯ В ООП.

См. вопрос 37, чего нет – не требуется. Слова «событие» даже в методичке нет ☺

Материал из интернета:

Событие в объектно-ориентированном программировании — это сообщение, которое возникает в различных точках исполняемого кода при выполнении определённых условий. События предназначены для того, чтобы иметь возможность предусмотреть реакцию программного обеспечения.

Для решения поставленной задачи создаются обработчики событий: как только программа попадает в заданное состояние, происходит событие, посылается сообщение, а обработчик перехватывает это сообщение.

Свойство — это способ доступа к внутреннему состоянию (атрибутам) объекта. Обращение к свойству объекта выглядит так же, как и обращение к полю, но, в действительности, реализовано через вызов функции.

42. ПОНЯТИЕ КОНСТРУКТОРА И ДЕСТРУКТОРА ОБЪЕКТА.

См. вопрос 37.

43. ПРИНЦИПЫ СОКРЫТИЯ ИНФОРМАЦИИ В ООП.

Материал из интернета:

Сокрытие данных – пользователь класса может работать только с его интерфейсной частью и не имеет доступа к реализации функциональности класса.

44. ПРОЦЕДУРА, ФУНКЦИЯ, БЛОК. ИХ СХОДСТВО, РАЗЛИЧИЕ И РЕАЛИЗАЦИЯ.

Подпрограмма – это некоторая последовательность операторов языка, оформленная особым образом, к которой можно обратиться по имени, т.е. она представляет собой поименованный блок.

По типу подпрограммы делятся на:

- ***процедуры*** – подпрограммы, которые могут иметь сколько угодно параметров или не иметь их совсем, при этом активизация процедуры происходит посредством самостоятельного оператора;

- ***функции*** – подпрограммы, имеющие сколько угодно входных параметров, но только один выходной, который является обязательным и содержится в ячейке, идентифицируемой именем функции

Заголовок процедуры имеет вид:

```
Procedure <имя процедуры> [ ( <список формальных параметров> ) ];
```

В процедуре `<список формальных параметров>` необязателен и может отсутствовать. Если же он есть, то в нем должны быть перечислены имена формальных параметров и их тип.

Выходное (нет, блин, буднее) значение для функции должно быть помещено в ячейку `<имя функции>`. Поэтому в заголовке функции присутствует описатель `<тип>`, который определяет тип переменной `<имя функции>`, а также тип возвращаемого функцией результата. Заголовок функции следующий (синтаксис Pascal):

```
Function <имя функции> [(<список входных формальных параметров>)] : <тип>;
```

Как видно из примеров, параметры в списке отделяются друг от друга точкой с запятой. Несколько следующих подряд однотипных параметров можно объединять в подсписки, например, заголовок функции можно написать проще:

```
Function F (a, b : real) : real;
```

Различия были расписаны выше, сходства: оба являются структурными единицами программы, имеют в себе один или более параметр, в обоих совершенно на равных участвуют как формальные параметры, так и глобальные и локальные переменные.

45. МЕХАНИЗМ РАБОТЫ СТЕКА ПРИ РЕАЛИЗАЦИИ ПОДПРОГРАММ.

Обмен параметрами при реализации описанных режимов и моделей происходит через стек выполняемой программы.

1. Стек выполняемой программы инициализируется и поддерживается системой поддержки выполнения программ.
2. Параметры, передаваемые по значению, копируются в ячейки стека. Эти ячейки затем служат хранилищем для соответствующих формальных параметров.
3. Передача параметров по результату реализуется как противоположность передаче параметров по значению.
4. Значения, присвоенные фактическим параметрам, передаваемым по результату, помещаются в стек, откуда они могут быть извлечены вызывающим программным модулем после завершения работы вызванной подпрограммы.
5. Передача параметров по значению и результату может быть реализована в соответствии со своей семантикой как комбинация передачи по значению и передачи по результату.
6. Ячейка стека инициализируется вызовом и затем пользуется как локальная переменная в вызываемой подпрограмме.
7. Передача параметров по ссылке наиболее проста для реализации. Независимо от типа фактического параметра в стек должен помещаться лишь его адрес. Если параметр является выражением, то компилятор должен построить код для вычисления выражения непосредственно перед передачей управления в вызываемую подпрограмму.
8. Адрес ячейки памяти, в которую код помещает результат своих вычислений, затем записывается в стек.
9. При передаче по ссылке может возникнуть следующая проблема. Допустим, что подпрограмма завершилась аварийно (возникла исключительная ситуация). В этом случае фактический параметр, передаваемый по значению и результату, не изменится, в то время как при передаче параметров по ссылке соответствующий фактический параметр может измениться до появления ошибки.

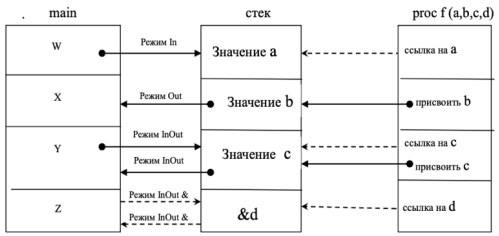


Рисунок 4.4 . Механизм работы стека.

46. СПОСОБЫ ПЕРЕДАЧИ ИНФОРМАЦИИ В ПОДПРОГРАММЫ.

Итак, если мы оформили часть алгоритма в виде подпрограммы, то основная задача – передать информацию внутрь блока и “выбросить” результат работы в вызывающую программу для дальнейшего использования.

Существует два способа информационного обмена между подпрограммой и вызывающей программой:

1. Прямой доступ к глобальным переменным.

2. Передача через параметры подпрограммы. Этот способ является более гибким, чем глобальные переменные. Подпрограмма может вычисляться много раз для любых новых фактических данных, а глобальные переменные надо переопределять перед каждым вызовом. Кроме того, глобальные переменные будут видны и в других подпрограммах.

47. РЕЖИМЫ ПЕРЕДАЧИ ДАННЫХ В ПОДПРОГРАММЫ

Передача данных между фактическим и формальным параметрами может происходить в 3-х режимах:

1. Режим ввода (In) – формальный параметр только получает данные от фактического (только входные);

2. Режим вывода (Out) – формальный параметр только передает данные фактическому (только выходные);

3. Режим ввода-вывода (In-Out) – формальный параметр может и получать, и передавать данные фактическому (быть и входным и выходным).

48. МОДЕЛИ РЕАЛИЗАЦИИ ПЕРЕДАЧИ ПАРАМЕТРОВ В ПОДПРОГРАММЫ

Существует несколько моделей реализаций указанных режимов передачи параметров.

1. Модель передачи по значению. Реализует режим ввода и означает:

а) реальное копирование значения фактического параметра в ячейку формального, который становится реальной локальной переменной. Надежный способ, но копирование может быть дорогим и по времени, и по памяти (требуется хранить 2 копии), если передаем большой массив.

б) передача пути доступа к значению фактической переменной в вызывающей программе, и при этом фактическая переменная блокируется на запись (например, C++): можно только читать из нее. Защита от записи может оказаться сложным делом, особенно, если передавать параметр по иерархии подпрограмм.

2. Модель передачи по результату. Реализует режим вывода. Реализуется аналогично п.1.: копированием формальной переменной в фактическую или передачей пути доступа. При этом возможна следующая проблема.

> *Пусть подпрограмма имеет выходной параметр list[i]. Возникает вопрос, когда вычислять адрес фактической переменной: во время вызова подпрограммы или при возвращении из нее. Если переменная i изменяется во время выполнения процедуры, то и адрес list[i] на входе в подпрограмму и на выходе из нее будет разный.*

3. Модель передачи по значению и по результату. Режим ввода-вывода. Комбинация передач по значению и по результату. Фактически при этом формальные параметры должны храниться в локальной области процедуры, и дважды делается копирование: в них и из них.

4. Модель передачи по ссылке. Реализация режима ввода-вывода, заключается в передачи пути доступа (адреса) фактической переменной в подпрограмму. Эффективна по времени и по объему памяти, поэтому находит реализацию во многих языках. Недостаток заключается в возможности совмещения имен (альтернативные имена). Ниже приведены примеры таких ситуаций.

5. Модель передачи по имени. Реализация режима ввода-вывода, но с особенностями. Особенности заключаются в следующем. Вид фактического параметра диктует выбор модели реализации. Если фактический параметр – скалярная величина, то передача равносильна передаче по ссылке. Если – константное выражение, то равносильна передаче по значению. Если фактический параметр выражение, содержащее переменную, то передача особая. Она характеризуется поздним связыванием, т.е. связывание формального параметра с фактическим произойдет не в момент вызова процедуры, а в момент присваивания формальному параметру конкретного значения или ссылки на него. Преимущества передачи по имени – гибкость. Недостаток заключается в медленном выполнении этого механизма.

49. РАЗДЕЛЬНАЯ И НЕЗАВИСИМАЯ КОМПИЛЯЦИЯ

Возможность компилировать части программы без компиляции всей программы существенна при создании больших систем программного обеспечения. Следовательно, языки, разрабатываемые для таких приложений, должны допускать такой вид компиляции. Существуют два разных подхода к компиляции частей программы:

- раздельная компиляция,
- независимая компиляция.

Части программ, которые могут компилироваться отдельно, называются единицами компиляции.

Раздельная компиляция означает, что:

– единицы компиляции могут компилироваться в разное время,

– между единицами компиляции существует связь по данным,

– осуществляется проверка типов данных и протоколов модулей при компиляции.

Примером раздельной компиляции может служить возможность организации модулей Unit в языке Pascal.

В некоторых языках, среди которых выделяются ранние версии языков C и Fortran, допускалась ***независимая компиляция***. При независимой компиляции:

– программные модули компилируются без связи с другими программными единицами,

– проверка типов и интерфейсов модулей не осуществляется.

Интерфейс подпрограммы на языке Fortran-77 представляет собой список параметров. Когда подпрограмма компилируется отдельно, типы ее параметров не хранятся вместе с компилируемым кодом или в библиотеке. Следовательно, при компиляции другой программы, вызывающей данную подпрограмму, типы фактических параметров в вызове не могут проверяться на совместимость с типами формальных параметров подпрограммы, даже если доступен машинный код вызываемой подпрограммы.

СОДЕРЖАНИЕ: