**Лабораторная работа 1.**

Программу нельзя модифицировать в процессе работы, но можно модифицировать данные. Программа и данные одинаково выглядят и одинаково воспринимаются интерпретатором. В других языках 60 года тоже были данные и программа, но в ЛИСП (язык, работающий со списками)…….

**Рекурсия** – это ссылка на описываемый объект в процессе его описания.

**Рекурсивная функция** – это текст, описание того, что она делает. Мы же описываем функцию и не закончив это описание внутри ссылаемся на это же описание (на эту же функцию).

Рекурсивно можно описать не только функцию, но и СД. (описание списков рекурсивно)

В других программах данные хранятся в одной области памяти, а программы в другой. В декларативных языках для обращения к данным используется ……А мы хотим временами данные выдавать за программу, а программу за данные.

Как отличается двоичный код от двоичного кода? Данные – в одной области памяти, программа – в другой. Если переписывать из одного места в другое – это затраты времени……

Как организована информация в памяти? Где расположить данные, тк время от времени надо выдавать за программу, программу выдавать за данные?

**Список** – это динамическая структура данных. Может быть пустым и непустым. Пустой обозначается Nil или пустыми (). Списки расположены в куче. Непустой – если состоит из …. Список можно модифицировать.

В памяти список представляется бинарным узлом, состоящим из двух указателей: car – указатель на первый элемент, cdr – указатель на оставшуюся часть.

1 список – 1 ячейка

Распределяет память система. Памяти хватает не потому что ее очень много, а потому что система работает эффективно.

Еще есть структуры данных: списки, деревья (это тоже списки), массивы,

**Идентификатор** – это имя объекта.

Статические структуры данных – кол-во эл-тов не меняется (статический массив)

**Динамические структуры данных** – меняется кол-во элементов во время выполнения программы. Доступ к элементу

**Указатель (ссылка)** – адрес ИЛИ ???переменная, которая содержит адрес.

**Переменная** – объект программы, которая имеет имя и идентификатор и хранится в статической области памяти. Переменную мы описываем (т.е. охарактеризована (объявлена), как integer), то в оперативной памяти будет храниться значение переменной. В программе переменная обозначается идентификатором, а в памяти хранится значение. Имя переменной – это аналог (простая форма обозначения той ячейки, где лежит значение).

Описание переменной нужно системе, а не программисту. Описание = указать тип.

**Тип** – от него зависит размер памяти, выделяемой для значения переменной, хранящейся в оперативной памяти.

**Массив** – СД, доступ по индексу (по порядковому номеру элемента): элементы лежат где-то по порядку, и по номеру можно обратиться по порядковому номеру.

К статическим данным обращаются по именам, а на самом деле компьютер найдет эти данные….

**Оперативная память** состоит из ячеек (**ячейка** – минимально адресуемое пространство оперативной памяти). Чтобы найти информацию, компьютер должен знать адрес, поэтому есть шина адреса, адресный регистр и тд. Сначала программировали в виде двоичного кода, человек должен был помнить, что у него лежит, по какому адресу – это было трудно. Поэтому появились переменные, потому что человеку легче запомнить a\_max.

**Адрес** – порядковый номер (целое число) той ячейки, начиная с которой записана информация, которую мы хотим получить.

Если **есть статические данные**, то они в статической памяти и обращение к ним происходит по именам, тк человек не может запомнить, в каком адресе, что лежит. Система создает **таблицу имен:** имя переменной – адрес. Тогда мы говорим программе возьми х, прибавь 2 и положи на старое место, то компьютер через таблицу должен найти в таблице х, его адрес, все сделать, а потом положить по нужному адресу.

Есть область, где лежат данные, где лежит программный код и есть **куча**.

**Как работают с информацией, которая в куче?** **Чтобы можно было один и тот же текст то так, то так воспринимать (то в качестве данных, то в качестве программы),** его нужно разместить **в кучу**.,

ЛИСП работает с указателями. ЛИСП столкнулся с тем, что не хватает памяти.

В ЛИСП появились сборщики мусора.

ЛИСП предложил организовать символьную обработку, это привело к тому, что, началась работа в куче в динамической области памяти началась работа с указателями.

**Базис** – минимальный набор средств, с помощью которого можно решить любую задачу.

В декларативном ?????? языке программирования базис – это операторы (5 команд) + форма организация данных.????

Базис включает в себя структуры – атомы и бинарные узлы. Бинарные узлы позволяют нам указать, это точечная пара или списки + в базис включены функции.

**Для работы с информацией в куче не** используется **таблица имен**, а используется буквально адрес, поэтому, когда мы начинали работать со списками, то мы выделяли и освобождали, выделяли и освобождали, а для статических данных 1 раз описали, 1 раз выделилась память и все время элемент там лежит и с помощью имени определяется адрес и обеспечивается доступ.

**LISP** – обработчик списков., то чтобы выдавать информацию то за данные, то за программу, надо чтобы они находились в одной области данных, поэтому это куча, поэтому списки находятся в куче и работа LISP основана на указателях и она осуществляется очень быстро, т.е. не надо смотреть имя в таблице имен, потом смотреть адрес и полезть в память и найти значение. Тут все сразу напрямую организовано с помощью указателей.

**Атом** – элементарная конструкция языка и с точки зрения синтаксиса и с т. зрения того, как это в памяти с т.зрения синтаксиса атом похож на идентификатор – послед-ть символов без пробелов (1 слово, н-р абсд - атом). Атом – на самом деле космос, тк в паямти он представляется 5 указателями!!!

LISP интерпретирует выражения с учетом того, связанные они или нет. Установили указатель – значит связанные.

Символ в ЛИСП – не литерал, не буковка, а последовательность символов, похожая на идентификатор, но поскольку нет переменных, но используется не термин «переменная», а символ.

**Есть символы** – представл. указателями, а есть символы особенные – Nil – признак пустого списка и t - !!!!! и есть числа. И все это – атомы – элементарные конструкции, поэтому есть атому и есть структуры (точечные пары). Если это пара, то их два и их нужно найти в памяти. Точечная пара представляется в памяти бинарным узлом – бинарная ячейка, которая хранит 2 указателя (на 1 и 2 эл-т)

Все, что не Nil – это true.

**Базис:**

* Atom - Надо отличить атом от структуры, чтобы дальше работать. Поэтому есть базисная ф-я: это атом или нет (atom).
* Cons – конструктор, работает эффективно, создает список (одну списковую ячейку и расставляются указатели) ????
* Eq – сравнение указателей., Предикат, проверяющий идентичность двух символов.
* Car, cdr – селекторы. Car, обеспечивают доступ ????
* Cond – похожа на case,
* Quote
* Lambda –
* Eval –
* Label –

**Базис** - это минимально необходимый набор конструкций с помощью которого можно написать программу.

**Базисные ф-и** - предоставляются языком, минимально необходимые для работы языка.

* селекторы (car cdr)
* конструкторы (cons)
* предикаты – логические ф-ии (null, atom, numberp, ..)
* сравнения (eq, eql, equal, = .)

Сравнение cons и list ???

Самоопределяемые символы – это числа. Символы можно свзяать с 1 значением, а потом с другим, поэтому символы похожи на идентификаторы.

LISP – не счетный язык.

LISP полностью организован на указателях и все находится в куче.

LISP – бестиповый язык, описывать переменные НЕ надо, все представляется указателями, память распределяется автоматически.

По сути в LISP – 2 эл-та: атом и точеченая пара. S-выражение – либо атом, либо пара – просто термин.

LISP основан на LISP.

Точечная пара в памяти представляется 2 указателями.

В ЛИСП есть только либо атомы, либо точечные пары и ВСЕ! БОЛЬШЕ НИЧЕГО НЕТ!

**Список** – частный случай S-выражения. Если он пустой, то бинарный узел не нужен и он представляется с помощью Nil (Nil – это атом). Если непустой, то у него есть, как минимум, голова и хвост (который являетися списком и мб пустой или непустой). 1 список – 1бинарный узел (указ-ль на голову и хвост). У списка второй эл-т – хвост, он мб пустой, мб непустой => хвост - это список. Голова мб любой структурой (атом, точечная пара, список и тд), но хвост – если он атом, то это – точеченая пара. Структура – это пара и синтаксически разделяется точкой. Но если хвост имеет больше эл-тов, чем 1.

Если **список непустой**, то это частный случай точечной пары. У него есть голова – точеченая пара и хвост – пустой или непустой и тд.

**Нотации списка**: Список, если непустой, имеет голову и все остальное, а значит, это – точечная пара. Если он не пустой, то это структура, эта структура – точеченая пара. У хвоста есть своя голова и хвост (мб пустой и непустой) и тд. Каждому списку соответствует своя списковая ячейка.

1 списку соответствует только 1 спсиковая ячейка, а другому списку – другая.

(A B C D) – эл-ты списка через пробел, вместо .(): (A.(B.(C.(D.Nil))) – одно и то же, но пробелы – облегченная форма.

Если это точеченая пара, то явно присутствует точка.

Для точечной пары – это экономия памяти, меньше списковых ячеек, но обрабатывается 1 пара, но времени ………

Любой эл-т списка мб сложно организован, не обязан быть атомом. Есть одноуровневые списки.

**Символьная обработка**. Все решается по умолчанию (где данные, а где программа). Все, что набираем, система **по умолчанию воспринимает как программу** – список. LISP основал на лямбда-исчислении Черча – мат синтаксис записи функции.

Восприятие интерпретатором того, что пишем всегда одно и то же.

ЛИСП – функц язык и на 1 месте всегда – **имя функции** (символьный атом, идентификатор), а потом ее аргументы. Если функция определена, то все отработает, иначе будет ошибка. Данные и программа синтаксически оформляются одинаково, регулярно ??? в виде списков.

Любой список – это программа, надо вычислять. ?????

Если нет описательной части, где в виде операторов написано, что нужно сделать, то остается 1 способ упорядочивания: на одном месте всегда одно. Потом всегда другое, такой порядок в ЛИСП пришел из лямбда-исчислений Черча…

Функции: примитивные (чистые), специальные или функционал

Программа – вызов функции на внешнем уровне.

**По умолчанию это всегда программа, и если мы хотим, что это было воспринять как данные**, то нужно использовать другую функцию, которая являетя базисной – квота.

**Классификация функций:**

* Чистые (математические) – как в математике. У ЛИСП – всюду определенные ф-ии. Фиксировано кол-во аргументов и есть результат.
* Рекурсивные
* Специальные – формы. Существуют функции, которые позволяют передавать разное кол-во аргументов, по-разному обрабатываются аргументы. Функция «+» - форма, принимает произвольное кол-во аргументов.
* Псевдофункции – функции, которые создают эффект на внешних устройствах (на экране)
* Функции высших порядков – функционал, используются для синтаксически управляемых программ. С помощью синтаксиса можно управлять тем, как работает программа. Это абстракция языка. Описание одной функции синтаксически похоже на описание другой функции. Вместо 2 похожих синтаксических текстов можно написать 1 текст и какие-то аргументы. (Н-р хочу посчитать интеграл сегодня от sin, завтра от log)
* Функции с вариантными значениями, из которых выбирается только 1 значение – любая ф-я возвращает только 1 значение в качестве результата

Все ф-ии пишутся, используя базисные ф-ии.

Defun – определение функции, это ф-я, которая является макроопределением, ??не вычисляет свои аргументы, ?? т.е. является формой, эта ф-я. Аргументы через пробелы. Тело ф-ии – s-выражение – определение функции.

В ЛИСП есть функции, которые выполняют одно и то же, но с разной скоростью. Скорость – засчет минимизированной работы (качественная проверка – долго работает, менее качественная - быстрее). Поэтому есть группы функций, которые делают одно и то же, но с разной степенью качества.

Ядро ОС –

Ядро – основные действия, которые наиболее часто используются. В момент загрузки загружается не только базис, но и ядро.

**Чтобы представить инф-ю как данные, надо запретить вычилсять - блокировка вычислений.** ЛИСП – функц язык, если что-то надо сделать, то вызывается ф-я. Блокировка вычислений входит в базис, называется квота. Текст программы интерпретатор должен разобрать, чем длиннее текст, тем дольше разбор, поэтому разрешено вместо функции квота использовать ‘ (**апостроф** - аналог ф-ии квота). Если стоит апостроф, то это данные, но все равно в памяти – в виде списка. Апостроф – быстрее обрабатывается (интерпретируется).

Есть программы, которые можно выполнить – исполнимая часть, а есть описательная часть, но это не команды.

Lisp — функциональный язык программирования, Lisp — безтиповый язык. Основные элементы языка Lisp: атомы, точечные пары, S-выражения, списки. Lisp ориентирован на обработку символьных данных. Атомы могут восприниматься как идентификаторы и как константы в зависимости от контекста. Необходимо соблюдать правила записи идентификаторов (атомов) и правила записи списков (структур). S-выражения — это наиболее общие структуры. Вычислимые S-выражения (списки) могут быть обработаны системой. Списки- рекурсивно описанные структуры — пустые или непустые. Внутреннее представление списков – взаимосвязанная последовательность бинарных ячеек, хранящих указатели.

В вычислимом списке голова является именем функции, а остальные элементы списка — ее аргументами. При наличии символа блокировки вычислений, список воспринимается как структура данных.

Базовыми функциями доступа являются функции: CAR и CDR.

Базовыми функциями создания списка являются функции list и cons.



=============================================

**Куча**— это хранилище памяти, также расположенное в ОЗУ, которое допускает динамическое выделение памяти и не работает по принципу стека: это просто склад для ваших переменных. Когда вы выделяете в куче участок памяти для хранения переменной, к ней можно обратиться не только в потоке, но и во всем приложении. Именно так определяются глобальные переменные. По завершении приложения все выделенные участки памяти освобождаются. Размер кучи задаётся при запуске приложения, но, в отличие от стека, он ограничен лишь физически, и это позволяет создавать динамические переменные.

Вы взаимодействуете с кучей посредством ссылок, обычно называемых указателями — это переменные, чьи значения являются адресами других переменных. Создавая указатель, вы указываете на местоположение памяти в куче, что задаёт начальное значение переменной и говорит программе, где получить доступ к этому значению. Из-за динамической природы кучи ЦП не принимает участия в контроле над ней; в языках без сборщика мусора (C, C++) разработчику нужно вручную освобождать участки памяти, которые больше не нужны. Если этого не делать, могут возникнуть утечки и фрагментация памяти, что существенно замедлит работу кучи.

Лабораторная работа 2.

В Лиспе самыми основными функциями являются car, cdr и cons. Функция cons используется чтобы создать список, а функции car и cdr, используются для того, чтобы список разделить.

Имя функции cons довольно разумно --- оно произошло от сокращения слова `construct' (конструировать). С другой стороны происхождение названий функций car и cdr довольно интересно --- car это сокращение от фразы `Contents of the Address part of the Register' (Содержание Адресной части Регистра); а cdr (произносится как 'куд-ер) --- это сокращение от фразы `Contents of the Decrement part of the Register' (Содержание Декрементной части Регистра).

car от списка --- это первый элемент в списке. То есть car от списка (роза фиалка маргаритка лютик) равен роза. car не удаляет первый элемент из списка --- он только возвращает нам его.

cdr от списка --- это оставшаяся часть списка, то есть, функция cdr возвращает часть списка, которая следует за первым элементом. В то время как car списка (роза фиалка маргаритка лютик) --- это роза, то cdr возвращает оставшуюся часть списка, то есть (фиалка маргаритка лютик).

Как и car, функция cdr не удаляет никаких элементов из списка --- она только сообщает какой второй и последующие элементы.

Кстати в этом примере список цветов цитирован. Если бы это было не так, то интерпретатор Лиспа попробовал бы вычислить список вызвав розу как функцию.

С другой стороны, в списке составленном из списков, первый элемент --- это список. car возвратит этот первый элемент списка. Например, следующий список содержит три вложенных списка --- список плотоядных, список травоядных, и список морских млекопитающихся:

|  |  |
| --- | --- |
|  | (car '((лев тигр гепард)  (газель антилопа зебра)  (кит дельфин морж))) |

В этом случае первым элементом, или car списка является список плотоядных (лев тигр гепард), а cdr списка является список ((газель антилопа зебра) (кит дельфин морж)).

3. **CAAR** - голова головы списка.

    4. **CDAR** - хвост головы списка.

    5. **CADR** - голова хвоста (***второй элемент списка***).

    6. **CDDR** - хвост хвоста списка.

    7. **CADDR** - голова хвоста хвоста (***третий элемент списка***).

    8. **CDDDR** - хвост хвоста хвоста списка.

    9. **CAAAR** - голова головы головы списка.

    10. **CDAAR** - хвост головы головы списка.





