Антон Е.Москаль,
Олег А.Плисс,
Андрей А.Терехов,
Андрей Н.Терехов,
Владимир В.Уфнаровский

Конспект лекций "Обзор и сравнительный анализ языков программирования"

Санкт-Петербург
2000

Оглавление

Введение

Общая задача курса – дать понятие о современных языках программирования, научить выбирать правильный язык программирования, обучить созданию ЯП и т.п.

Общая схема рассмотрения языка:

1. Название, кем, когда и зачем был создан
2. Общая характеристика:
   - декларативный или императивный;
   - язык-оболочка или язык-ядро;
   - хороший или плохой (красивый? схожесть операторов, т.е. если что-то изменить, то насколько велика вероятность неявной ошибки? Эффективность – по скорости исполнения; по скорости компиляция; по времени отладки, т.е. перекомпиляции; отзывчивость (responsiveness), например, можно даже продолжать исполнение программы).
3. Синтаксис
4. Данные (типы, представление)
5. Управление последовательностью действий
6. Управление данными (передача параметров, подготовка к исполнению операций)
7. Управление памятью
8. Модульность (по-хорошему надо бы записать после 3)
9. Среда ЯП, работа с файлами, устройствами

Как описывается синтаксис ЯП?

- неформально
- в форме Бэкуса-Неймана (БНФ):

  метасимволы в угловых скобочках
  ::= есть «обозначает»
  | есть «один из вариантов»

Пример БНФ:

<число> ::= <знак><последовательность цифр>
<знак> ::= <пусто> | –
<последовательность цифр> ::= <цифра><последовательность цифр>
<цифра> ::= 0|1|2|3|4|5|6|7|8|9

Расширенная БНФ (для записи регулярных выражений)
вводятся дополнительные символы *, +, { }, [ ]

Пример РБНФ:

<число> ::= { пробел | –}{0|1|2|3|4|5|6|7|8|9}+
или [–] {0|1|2|3|4|5|6|7|8|9}+

Как описывается семантика ЯП?

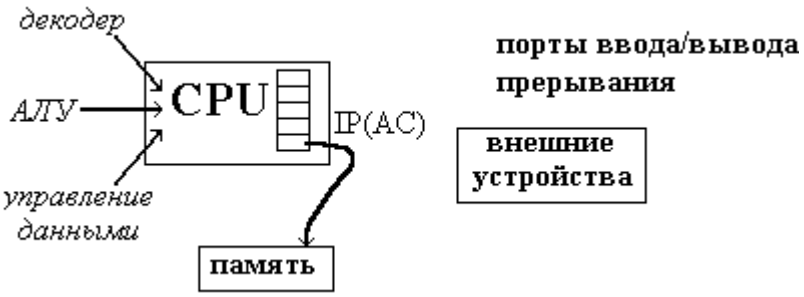
В терминах **абстрактного вычислителя**; считается, что программа транслируется в терминах вычислителя (**виртуальной машины**). Кроме того, бывает *последовательность виртуальных машин*.

Интерпретация

Программно-аппаратная реализация – в микропрограммах. Что такое микропрограмма? Современные процедуры сделаны либо совсем аппаратно, либо с помощью микропрограмм, с помощью которых можно добраться до процессора напрямую.

Архитектура ЭВМ

Раньше делались процессоры, ориентированные на язык (ФОРТРАН-машины). Дело в том, что выгоднее потратить транзисторы на очень простые операции, но сделать их очень быстрыми.



Память состоит из бестиповых ячеек, у которых есть только целый адрес, туда можно записать что угодно.

У процессора есть instruction pointer (он же program counter). Есть декодер и АЛУ.

Работает все это так: декодер вынимает следующую команду, управление данными вынимает операнды, АЛУ все это исполняет, IP инкрементируется.

В современных ЭВМ все это сопровождается наворотами (несколько параллельно исполняемых команд, выборка вперед и т.п.). Бывает иерархия памяти (например, **кэш** – промежуточный уровень между быстрой и медленной памятью).

Когда пишешь что-то в порты I/O сразу записывается что-то на внешнее устройство. Интересно, что input и output регистры не совпадают.

Это не очень удобно. Прерывания – различные уровни (256 уровней). Прервал программу – передай управление на обработчик.

Память – данные по 8, 16, 32 бита (выравнивание). Бывают binary/decimal, packed/unpacked, floating.

Операции над данными – арифметические, сравнения (результат – в регистре состояния), проверка битов состояния ($\Rightarrow$ ветвление), переходы бывают безусловными и условными, ввод/вывод в порты прерывания; иногда управление иерархией памяти; бывают аппаратно поддерживаемые стеки. В таких случаях можно поддерживать подпрограммы (адрес кладется на стек, по окончании снимается). Это поддержка не везде и если не поддержано, то надо делать руками.

Многопроцессорные системы – несколько декодеров + память общая. Основания проблема – управление памятью, особенно кэшем.

Данные в ЯП:

- встроенные типы данных (есть такие-то и других не может быть)
- конструируемые типы (т.е. существует конечный набор элементарных типов, есть набор конструкторов типов), например:
  элементарные (первичные) = (целые, плавающие, вещественные, литеры)
  конструкторы = (записи, массивы, объединения)
- абстрактные типы данных (тип можно не только сконструировать, но и сделать неотличимым от первичного)

Теперь подробнее о типах.

Элементарные типы:

целые числа (двоичные и десятичные представления)
числа с плавающей точкой (знак, мантисса, порядок)
логические значения (true/false либо логические шкалы)
литеры (литеры внешнего вида, т.е. 7 или 8 бит)

Целые числа отличаются от реальных тем, что они обрезаны слева и справа. Вещественные числа отличаются ограниченным диапазоном и ограниченной точностью, т.е. a+b–b может быть ≠ a (например, из-за переполнения).

Операции: `+, -, *, /, mod, abs, <, >, ==, !=`, преобразование целых к вещественным; у вещественных – `log, sin/cos`, возведение в степень.

Логические: `&(and), | (or), !(not), ^(xor)`

Если логические значения представлены шкалами, то над каждым битом операции проводятся побитно. Кроме того, бывают сдвиги влево/вправо (простые/циклические).

В большинстве современных ЯП нет прямой адресации битов (⇒ нет типа bit). Иногда бывают комплексные числа (пара плавающих чисел).

Набор элементарных типов в разных языках ЯП отличаются: могут быть быть строки фиксированной или переменной длины, ограниченной длины.

Строки обычно представляются просто как последовательность байт. У строк фиксированной длины должен быть еще счетчик, либо конец строки должен явно обозначаться. Счетчик может быть в самой строке, либо хранится в дескрипторе.

Выборка литеры из строки, конкатенация строк, выборка подстроки и т.п. операции над строками реализуются по-разному (например, копирование куска в фикс. или заведение другого дескриптора при выборке строки).

Обычно все это транслируется программными средствами (то есть обычно нет аппаратной поддержки для выборки подстроки).

Конструкторы типов данных:

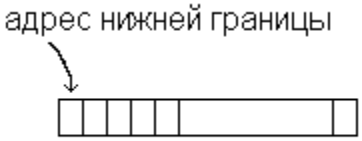
Массивы:

- одномерные и многомерные
- однородные и неоднородные
- фиксированной и изменяемой длины (обычно целым числом, иначе это называется коллекциями)
- плотные и неплотные

1. Одномерные массивы (однородные плотные фиксированного размера):
   - нижняя граница
   - верхняя границы

   Конструктор: тип элемента массива

   В памяти все это представляется просто:



   Операции: создать, узнать upb/lwb, вырезка и запись по номеру в массив.
2. Многомерные массивы

   Можно смоделировать с помощью одномерного массива:

$$n = 1 : base, itemsize, low, i \Rightarrow addr(i) = base + itemsize * (i - low)$$
$$n > 1 : addr(i_1, i_2) = base + size * (i_1 - low_1) + size * (i_2 - low_2) = c + size_1 * i_1 + size_2 * i_2$$

Формальная корректность – гарантия того, сто программа написанная человеком может быть понята машиной.

Отсутствие разночтений – не должно быть возможности истолковать программу по-разному.

Пример 1 (Pascal):

```
if  <условие1>
then
    if  <условие2>
    then  …
else
```

else на самом деле принадлежит второму условному предложению. Если бы в Паскале существовал endif, то этой проблемы не было бы.

Пример 2 (Fortran, PL/I):

A(i) – может быть вырезкой из массива или вызовом функции. Если массив забыли описать, то ошибка обнаружиться только при сборке.

Если вырезка обозначается A[i], то проблема не возникает.

Удобство чтения и **удобство записи** – противоречащие требования.

Простота трансляции – зависит от регулярности языка (хороший язык – Pascal).

Синтаксис

Синтаксические элементы:

1. входной алфавит
2. формат исходного текста
3. идентификаторы
4. пробелы
5. знаки операций
6. ключевые слова (зарезервированные)
7. комментарии и шумовые слова
8. ограничительные скобки
9. выражения
10. операторы
11. программные единицы
12. литералы

Входной алфавит – набор литер, с помощью которых можно писать программы (например, ASCII – тогда возникают проблемы с национальными символами).

Формат исходного текста:
– свободный – корректный не зависящий от позиции слов программы
– фиксированный (точнее с фиксированными полями) – в разных полях можно писать разные элементы программы
– с частично зафиксированными полями (часть строки зарезервирована – в Фортране первые 5 символов для метки)
– с выделенными позициями ('*' в седьмой колонке в Коболе, '\' в C/C++ в последней колонке - перфокарты)

VLIW-ассемблер – разным полям соответствуют разные устройства, поэтому программа может выглядеть так:

Метка	OR x1, x2	SHL …	Метка перехода	Комментарий

Идентификаторы – обычно последовательности вида:

<идентификатор> ::= <буква> {<буква>|<цифра>}*

Иногда у множеству букв или символов добавляют специальные символы, например '_'. Из множества возможных идентификаторов исключаются ключевые слова. Длина идентификаторов иногда ограничивается ($\Rightarrow$ 2 реакции на слишком длинный идентификатор: ошибка либо отсечение хвоста). Возможно различение больших и маленьких букв ($abc \neq ABC$ в C, но = в Pascal-e).

Пробелы (под этим подразумевается white space, т.е. еще табуляции, переводы строк и т.п.) Иногда пробелы игнорируются (например, в Фортране или в Алголе 68: MyVar≡My Var, 123≡1 2 3).

Знаки операций – обычно специальные символы. Иногда их объединяют в более длинные операции (+=). В некоторых языках – специальные слова (Plus), в Фортране – сравнение записывается как .EQ.

<шестнадцатиричная цифра> ::= <цифра><шестнадцатиричная цифра>* {h|H} – в начале цифра, иначе идентификатор!

Литералы – изображения числе, строк и т.п.

Например <десятичное беззнаковое число> ::= {<цифра не ноль><цифра>* | 0 }

<десятичное знаковое число> ::=[-|+] <десятичное беззнаковое число>

0 выделен, так как во многих языках с 0 начинаются восьмеричные числа: 0123≠123!

Ключевые слова – имеют особый смысл для образования конструкций, могут быть зарезервированными (т.е. их нельзя использовать как идентификаторы), это сильно облегчает трансляцию.

Комментарии и шумовые слова – комментарии бывают от скобки до скобки и до конца строки. Шумовые слова – для повышения читабельности (GO [TO])

Ограничители и скобки – " есть ограничитель, (–), begin-end – это скобки.

Литералы (константы):

<целое со знаком> [.<целое без знака>] [{e|E}<целое со знаком>]

Литеры: могут быть в апострофах 'abc', либо в кавычках. Если нужен апостроф, то либо дублируют "", либо вводят специальный символ '\"; иногда бывает специальный символ спереди: $a.

Строки: как литеры, только символов может быть несколько.

Булевские значения: могут быть идентификаторы true/false (зарезервированные/не зарезервированные), могут быть специальные символы (.T. или .R.)

Выражения – составные конструкции языка. Состоят из литералов, идентификаторов, знаков операций и разделителей (скобок).

Выражения могут быть в следующих формах:

1. Префиксная (польская запись) – знак операции, затем операнд.
2. Постфиксная (обратная польская) – в начале операнды, затем знак операции.
3. Инфиксная – знак между операндами.
4. Смешанная.

Пример 1. (PLUS (MINUS A B) C) ~ (A–B)+C

либо PLUS (MINUS A B) C – то же самое

Скобки можно вообще убрать, если задана арность операции:

PLUS MINUS A B C

Пример 2. То же самое, но наоборот

((A B MINUS) C PLUS)

A B MINUS C PLUS

Пример 3. Для бинарных операций:

A+B.

1) Приоритет – разным операциям приписывается приоритет ∈ [0, MAX_PRIO].

Если нет приоритетов, то A+B*C=(A+B)*C

Если они есть, то первым выполняется более приоритетный оператор.

2) Присоединение.

Если присоединение слева направо, то A+B+C=(A+B)+C, если присоединение справа налево, то A+B+C=A+(B+C)

Ясно, что приоритеты и присоединения нужны для уменьшения числа скобок в выражениях.

Пример 4. Используется практически во всех языках программирования (для разных операций – разная форма записи)

`f(x)` – префиксная

`a+-sin(x)`

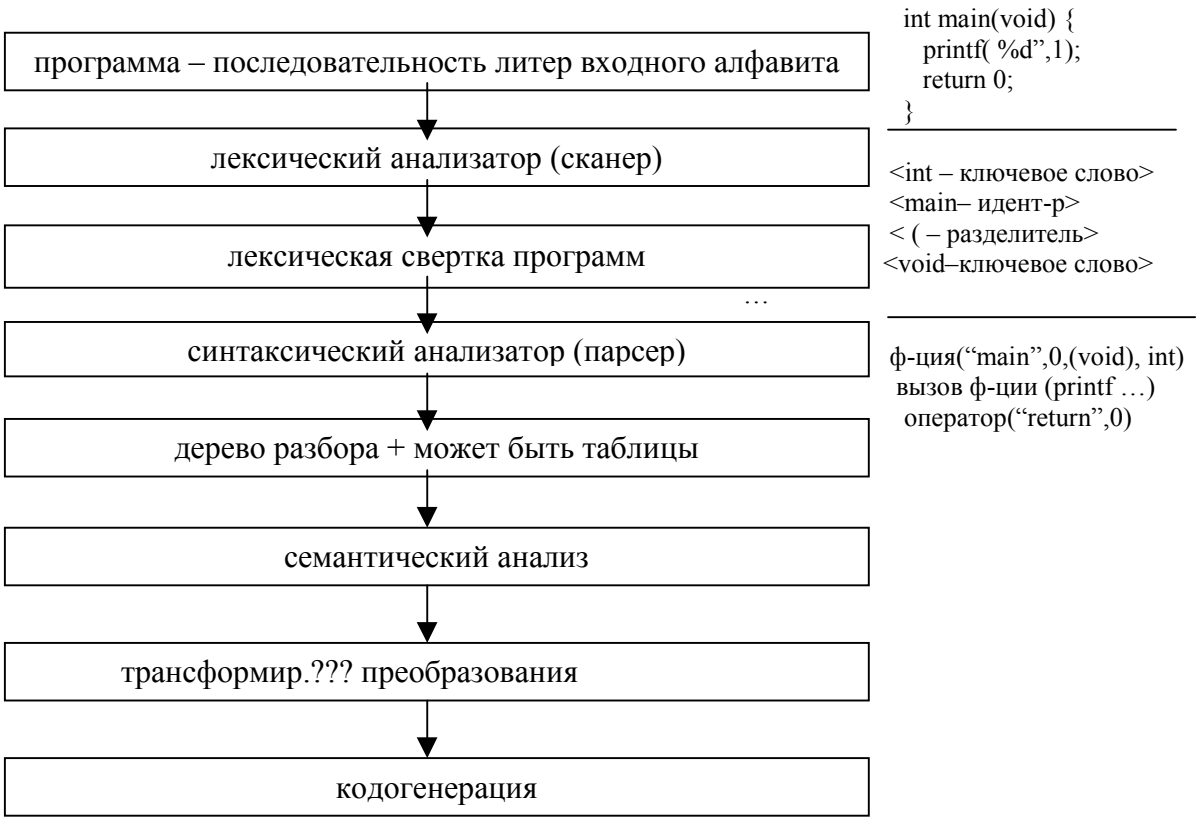
Операторы – основные конструкции языка, из которых состоят программные единицы

Программные единицы:

- функции
- процедуры
- подпрограммы
- модули
- файлы

Трансляция программ

Трансляция программ выполняется программой (времена ящиков никогда не вернутся)



Объектный код можно линковать, загружать и исполнять при исполнении относительные адреса заменяются на абсолютные.

Программы про трансляции:

– препроцессор; ⇒ время препроцессора
– транслятор; ⇒ время трансляции
– линковщик; ⇒ время загрузки
– загрузчик. ⇒ время исполнения

(Массив фиксированный ⇒ фиксированный в какое-то время из этих четырех)

Компилятор – берет исходный текст, производит объектный модуль.

Интерпретатор – берет исходный текст и сразу его исполняет.

Есть промежуточный вариант – остановиться на некотором Промежуточном Языке (ПЯ) (виртуальная машина) и интерпретировать ее.

Пример

```
# include <...>

int a [MAX_SIZE];     //a - фиксированный массив времени препроцессора
int b [sizeof(a)];    //b - фиксированный массив времени трансляции
int c [b[0]];         //c - фиксированный массив времени исполнения
```

Система типов – та информация в рамках языка, которая относится к значениям переменных во время исполнения.

Языки:
– бестиповые (Smalltalk, Forth, Basic)
– со статической типизацией
– с динамической типизацией
– со смешанным типовым контролем (часть во время компиляции, часть во время исполнения)

$$addr(i_0,\ldots,i_n) = base + \sum_{k=0}^{n} c_k i_k$$

Полный дескриптор массива:

type – тип элемента массива
dim – количество индексов
base – базовый указатель массива
lwbound
upbound } dim штук граничных индексов
width – для удобства, ширина массива

Что известно во время трансляции?

Обычно тип и количество индексов можно убить, т.к. известно во время трансляции
Про неплотные массивы рассказано не будет, так как они почти нигде не встречаются в виде базовой конструкции

Неоднородные одномерные массивы (структуры/записи)

```
struct S {
   int a;
   float b;
   char c;
};
```

Как структуры представляются? Выделяется память, отводится место для записей (элементов структур). Обычно в памяти это идет в несколько другом порядке, в связи с выравниванием.

Компилятор может вычислить смещение полей $\Rightarrow$ выборка из структуры = адрес + смещение

Неоднородные многомерные массивы

Структуры с вложенными структурами. Дальше – все точно так же.

Объединения размеченные

```
union S {
   int a;
  float b;
  char c;
}
```

В размеченных объединениях есть скрытое поле тэга, по которому можно определить какое поле используется в действительности; в неразмеченном объединении это узнавать, исходя из каких-то других соображений.

Исполняемые единицы (процедуры/программы)

В простейшем случае – процедура есть кусок кода, а представляется она адресом ее кода (Фортран, С).

Если есть блочные среды ⇒ представляется парой (адрес процедуры, адрес блочной среды)

Управление последовательностью действий

I. Неявное управление последовательностью действий.

1) последовательное выполнение (в порядке записи)

2) совместное исполнение (компилятор может определить удобный ему порядок исполнения, например:

```
x= 5;
y= x*x++;
```

может получится либо 25, либо 30, в зависимости от компилятора)

3) параллельное исполнение (по возможности исполняется параллельно)

II. Явное управление последовательностью действий

1) условное выражение:

```
if условие then  ... [else ...] fi
```

2) циклические предложения

а)
```
forever
      тело  цикла
repeat
```

Выйти, конечно, можно – возвратом процедуры программы, break-ом, переходом на метку.

б) цикл с предусловием:
```
while (условие)
    тело цикла
end
```

в) цикл с постусловием:
```
begin
    тело цикла
repeat until (условие)
```

г) цикл со счетчиком:
```
for int i from … to … [by …]
```

д) цикл со списком:

```
for i in (1,2,5,10)
```

Был язык в котором все эти циклы присутствовали одновременно в одной универсальной циклической конструкции (Алгол 60):

3) выбор по метке:

```
case (<выражение>
   <константное выражение> of …;
   default
end-case
```

Вычисляется выражение, которое затем сравнивается с константными выражениями.

Есть вариации в которых на месте константного выражения могло стоять любое булевское выражение

4) оператор безусловного перехода на метку:

```
goto … ;
```

«Трудно выйти в С из нескольких вложенных циклов». Не оператор плохой, а плохи люди, которые не умеют его использовать.

а)
```
goto <метка>
   …
<метка>:
```

б)
```
goto выражение (A,B,L,M)
   …
A:
B:
   …
M:
```

в)
```
goto x; // где x – некоторое выражение
```

Например,

```
read (x);
goto x
```

Если строки занумерованы. Трудно сделать что-нибудь приличное, так как любой строке потенциально соответствует метка.

5) вызов процедуры:

```
call <имя процедуры> (параметр 1, … ,параметр n)
```

Такая запись называется **позиционным следованием**: где-то есть описание процедуры:

```
proc f (int a, float b=1, char c) // формальные параметры
     f(1,2,'?')                   // фактические параметры
     f(1,,'?')                    // один из параметров по умолчанию
```

Если параметры по умолчанию допускаются только в конце, то от таких странных запятых можно избавиться.

Передача параметров с ключевыми словами

```
f(a=1, b=2, c='?');
```

Процедура не является математической функцией!!!

Различия между процедурой и математической функцией

1. Процедура записывается алгоритмически.
2. Обычно у математических функций есть область значений и область определения, например:

f: **Z**x**R**xChar

Но процедуры определены лишь частично на своих областях определения (нет деления на 0)

3. Процедуры могут использовать неявные параметры (глобальные переменные).
4. Неявные результаты
5. Процедура может не возвращаться в точку вызова (исключительные ситуации, переходы)
6. Значение процедуры может быть недетерминированным.

Замечание Терехова про время в процедурализме и логицизме.

Какие бывают процедуры?

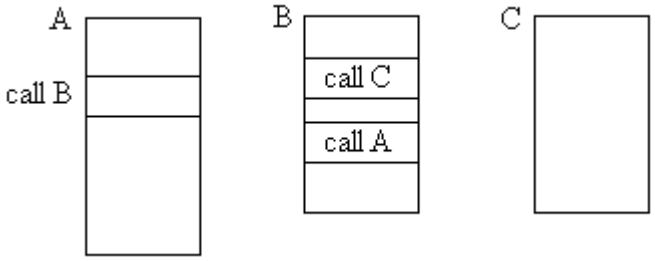
1. Простые: a→b→c→b→a
2. Рекурсивные: a→a

[прямая и косвенная – в том числе может быть через параметры: `f (proc g)`]

Вернулись к простым:

в Фортране только это и есть: никакого стека не нужно. Однако при таком подходе никакой рекурсии не получится.

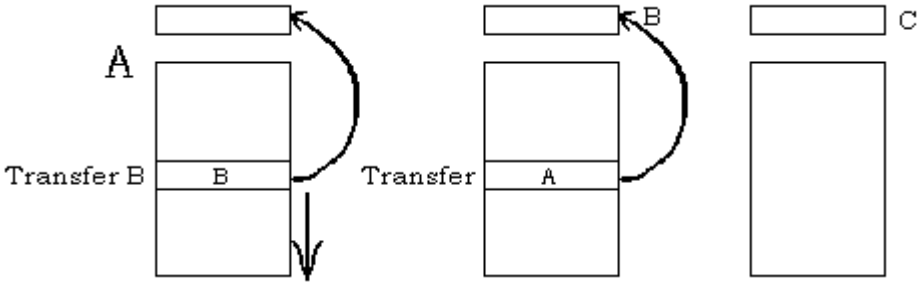
Если есть рекурсивные вызовы, то нужен стек, так как надо уметь идентифицировать различные экземпляры процедур.



3. Сопрограммы

call переходит без возврата, но с запоминанием адреса.

Исполнение продолжается с той точки где оно было прервано.
Как это моделировать в языках, где этого нет?



В C это можно сделать с помощью longjmp.

4. Процесс – после вызова может выполняться в прикрепленном состоянии или отсоединиться.

```
fork A   (...)
detach
```

5. Исключения или прерывания

Можно сделать процедуру, которая будет выполняться после выполнения какого-либо условия (заранее заданного или собственного):

```
ON OVERFLOW f
```

Если самостоятельно определено условие, то его надо и самостоятельно анализировать:

```
on <условие> g
  …
signal <условие>
```

6. Планируемые процедуры – вместо какого-то события (как в прерываниях) ставится просто булевское условие. Это может быть связано со временем между запусками и т.п. (Симула 67)

Управление данными (управление средами)

Связывание (binding); обычно идентификаторы связывают со значениями:

- времени препроцессирования `#define A 10`
- времени компиляции
```
const int b=2;
const int a=b+10;
```
- времени линковки
```
extern int b=2; // надо разрешить все внешние ссылки
                // во время линковки
```
- времени исполнения

  а) время вызова процедуры (формальные/фактические параметры) `A(1,2)`

  б) время входа в блок
```
{
  int a=1;
}
```
  в) время исполнения данного оператора
```
p→f();
```
Если f – виртуальная функция, то до самого последнего момента неясно, что мы вызовем.

Ассоциация идентификатора со значением:

1. Создать
2. Уничтожить
3. Временно деактивировать
4. Активировать заново
5. Поменять

Промежуток между созданием и уничтожением – **время жизни**. Области, где идентификатор активирован – **области видимости**.

```
void f (void) {
    static int a=0;
}
```

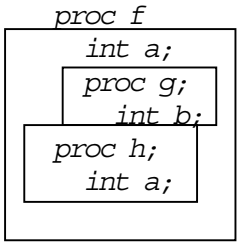
Создание переменной происходит при старте программы, но активизирована она будет только в момент вызова f.

Среда – это доступный в данный момент набор идентификаторов.

Бывают локальные и нелокальные (если внутри данной процедуры могут быть видимы и процедуры из других процедур).

Среды могут быть простыми и вложенными (могут быть статическими и динамическими).

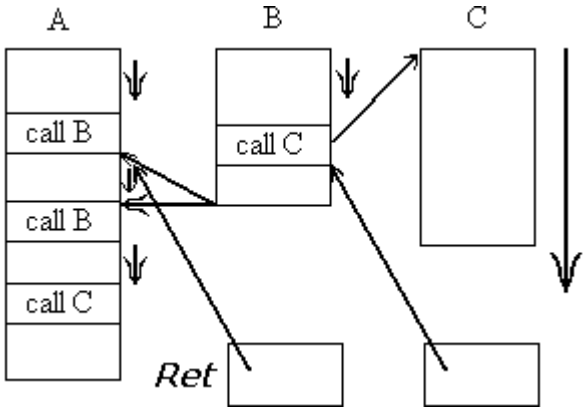
Отличаются ли статические вложенные среды от динамически вложенных? Да. В статических видимость определяется последовательностью описаний в тексте, а в динамической – последовательностью вызова.




```
f
f   g

f h g   – если статическая вложенность (посл-ть описаний)
a a b   – если динамическая вложенность (посл-ть вызовов)
–   –
  – –
```

Три процедуры A,B,C:



Если есть рекурсия, то необходимо отличать экземпляры ⇒ у каждого свой адрес возврата. Можно завести для каждой процедуры свой стек, а можно завести общий.

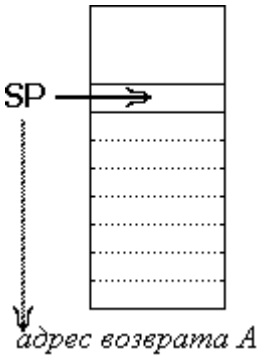
Среда

Модульные среды – нелокальные статические.

Локальные динамические среды.

Процедура: адрес возврата
секция активации

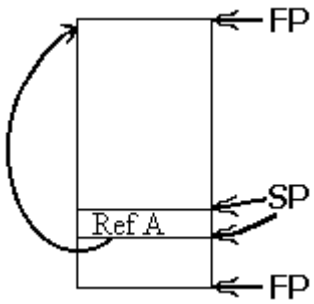
Устройство секции активации:
Существует два способа записи стековой рамки: один или два указателя



а) если один указатель ⇒ не можем динамически выделять память во время исполнения процедур.

FP – рамка стека

Необходимо разместить локальные данные в стек, который у подпрограммы уже есть.



б) два указателя: на рамку активной процедуры и на вершину стека.

```
push FP    //  Адресуем локальные данные относительно FP
mov FP,SP  //  с отрицательным смещением
sub SP,N_B // (не относительно рамки стека).
```

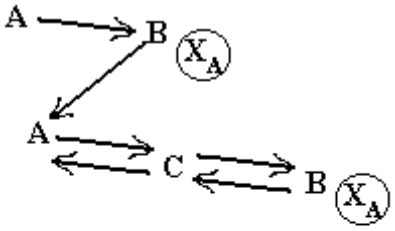
Обратно аналогично:

```
mov SP,FP
pop FP
```

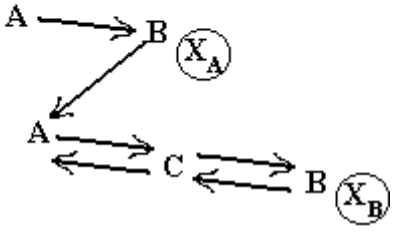
Нелокальные среды статической вложенности:

```
Proc A {
   int x;
   proc B {
      real y;
      x++; // x из A
   }

   proc C {
      char x;
      B();
   } B(); C();   };
```



при статической вложенности



при динамической вложенности (по динамической цепочке)

Реализация:

1) Статическая цепочка: указатель на секцию активации, статическая объемлющая процедура. Знаем уровень вложения статически.

   Для адресации: количество шагов по статической цепочке (уровень)+смещение в секции активации

Проблема в том, что необходимо все время ходить по статической цепочке. Можно звенья записывать и вместо хождения по цепочке смотрим в регистры.

2) Динамические вложенные среды: адресация переменной:
   а) самое простое – просмотр динамической цепочки
   б) hash-таблица

Передача параметров

1) по значению – при входе копируем в локальную переменную (теряем при выходе)
2) по результату – при выходе из процедуры копируем локальную переменную (на входе – ничего)
3) по значению-результату – при входе копируем в локальную переменную, при выходе – обратно
4) по ссылке
5) по имени – если в качестве параметра передаем выражение

Пример

```
int y,x;
proc f(int x) {
      x+=1; y+=1;
      print(x,y);
      x-=1; y-=1;
}
  y=0; x=1;
  f(y);
  print(y);
```

1) по значению
   x y
   1 1
     0

2) по результату
   x y
   2??? 1
     0

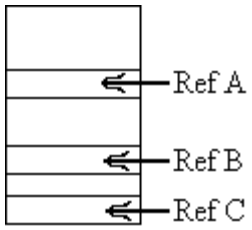
3) по значению-результату
   x y
   1 1
     0

4) по ссылке
   x y
   2 2
     0

5) по имени – то же самое

Параметры могут передаваться на стеке или на регистрах.

а) на стеке



б) на регистрах:

Группы регистров:

- для архивации вызова и передачи параметров
- регистры, которые хранят процедуры (сохраняют вызывающую процедуру)
- сохраняются вызванной процедурой
- рабочие (ни те, ни другие не хранятся)

Управление памятью

1. Статическое (работа только во время компиляции и линковки)
2. Стек (статический или динамический)
3. Куча (управление ручное или автоматическое)
   Зависит от запросов (бывают фиксированного и нефиксированного размера)

Фрагментация

Самолокаторы

Исторический обзор

ФОРТРАН был самым первым языком высокого уровня (1956 г.) и породил целый класс интуитивно неясных ошибок:

```
x = 1/3 // выдает 0, а не 0.3333
```

Другая ошибка:

```
x = 9.3 // x – число двойной точности
```

Порождение было такое:

```
LE 0, = E "9.3" // младшие 32-бита не зачищались
STD 0, x
```

Как это можно было обойти? Либо написать 9.3000000 (⇒ длинное), либо 9.3D (double).

Наконец, классическая ошибка (самая дорогая ошибка программирования).

```
DO 3 I=1.4 // (надо было не ".", а ",") Получилось присваивание.
```

Пример:

```
F(1)
I=1
WRITE ()I // печатается 0

SUBR F(a)
   A=0
```

Кроме того, ФОРТРАН не переносим, так как у него не было формального описания.

Спорное утверждение про то, что ЯП лучше создавать группами. Рассказывает про IFIP.

Working group 2.1 по языкам программирования
Working group 2.3 по методологии программирования

1958, 1960 гг. Алгол 60

а) формальная грамматика. БНФ;
Пример программы на Алголе 60:

```
begin
integer i,j;
real b,c
b:=c+1; #обругал C++ с его "=="
if then else # (fi – еще не было)
for i:=A step B until C do #обругал предвычисление B и C; двойное вычисление B
                                                    # из-за знака
procedure p(a,b);
  value a; real a,b
begin
   …
end
```

Подстановка по значению p(3+0.5, x) или подстановка по наименованию (с изменением значения и перевычислением). Например, p(3+0.5, x[i]) – круто, да?

б) переносимость программ;
в) динамические массивы;
г) рекурсивные процедуры

Про оптимизацию заведомо плохих алгоритмов – спорный момент.

Разделение на PL/I и Алгол 68.

Язык-оболочка; громоздкий

Пересмотренное сообщение Алгол60 – 1964 (от начала до содержательного языка прошло 6 лет)

«Белая книга Алгола 60» – целая пачка статей, определивших развитие программирования.

Обсудили регулярные выражения и их эквивалентность БНФ. Далее про конечные автоматы (и они тоже эквивалентны).

Бесконтекстные и контекстные грамматики.

```
ret MODE assignment:
        ret MODE variable, ':=',
                MODE expression
```

То, что написано большими буквами – определяется в другой грамматике, более высокого уровня. Единственное ограничение – что всюду надо подставлять одно и тоже ($\Rightarrow$ все виды в присваивании одинаковые).

Приведений в Алголе 68 всего 6 (ср. с PL/I, где их более 200)

1964 г. придумано структурное представление данных (Хоар, struct).

Декабрь 1968 г. Алгол 68

Реализовать А68 оказалось кошмарно сложным занятием, ибо язык был существенно сложным.

```
mode цепь = struct (int зв, ret цепь след.)
mode цепь = struct (int зв, ret цепь след)
```

Эти два вида эквивалентны? В структурной эквивалентности – да, а в именной эквивалентности – нет. (В Алголе 68 структурная эквивалентность).

```
union (m1, m2, m3)
```

Ограничения вводили сразу же – нельзя использовать *int* и *ref int*
case и in (m1): A1, (m2): A2, …

Если под *union* есть рекурсия, то деревья бесконечные и нельзя определить предшествование видов (можно создать пример, в котором типы неэквивалентны и оба типа предшествуют друг другу). Если еще и раздельно транслировать, то совсем плохо.

Вместо модели видов использовали конечный автомат и оказалось возможным доказать эквивалентность видов за *nlogn* шагов. Кроме того, придумали еще и канонический порядок (в том числе и трансляции модулей).

Тем не менее, у этого алгоритма была очень большая константная составляющая и на СМ1420 стандартное вступление транслировалось 1.5 минуты. В результате вернули старый экспоненциальный алгоритм.

Короче, много тем для диссертаций дал А68 (А. Рухлин защитил диссертацию с тем же названием, что и П. Сёки в 1976 г.).

Что же такое А68?

```
int, real, bool, char;      можно сказать ref m;
s struct (m1 a1, m2 a2) ⇒ a of s
(flex)? [l:u] m M;      ('l:' можно опускать, тогда от 1)
```

Триммерные вырезки (на примере умножения матриц)

```
for k to N do
  s+:=a[i,k]*b[k,j]
od
```

Цикл выполняется n^3 раз, при этом i,j не меняются. Поэтому можно написать так:

```
ref [] real ai = a[i,];
ref [] real bj = b[,j];
for k to N do
  s+:=ai[k]*bj[k]
od
```

При этом копирования не происходит (заводится лишний паспорт).

```
   proc (m) n ;
flex [3] int a;
  a:=(); a:=(10,20,30,40);   … – меняются границы

  a; b; c – последнее исполнение
  (a; b; c) – совместное исполнение (порядок не определен)
par (a; b; c) – параллельное исполнение (с семафорами)
```

А68 оказался слишком тяжелым в реализации (с А68 тоже ушло 6 лет на доделку – пересмотренное сообщение в 1974).

Мнение меньшинства (Minority Report).

Через несколько лет Вирт опубликовал Паскаль, но в нем было слишком много дырок и неточностей. «Двухкопеечный язык».

Похожесть языков программирования: «Золотой фонд системного программирования». Эта часть системного программирования устоялась.

Ассемблер

Зачем нужен ассемблер?

1. Исторически – чтобы не считать смещения и адреса в машинном коде, что для своего времени было большим продвижением.
2. Чтобы оптимизировать узкие места и использовать расширения команд, которые транслятор напрямую сгенерировать не может (например, Pentium MMX и выше)

Развитие ассемблера продолжается, в направлениях NGWS (Next Generation Windows System) и Typed Assembler (ассемблер с гарантированной надежностью типов).

Ассемблер имеет смысл рассказывать для конкретной архитектуры (в данной лекции – для PDP-11).

Команды бывают с двумя операторами: mov src, dst или с одним (инвертирование знака).

В ASM принципиально нет выражений ⇒ a:=b-c-1 расписывается как

```
mov b,a      ; a:=b
sub c,a      ; a-:=c
dec a        ; a-:=1
```

В PDP-11 было 6 регистров общего назначения. R0-R5, кроме того SP, PC.

Работа с адресами:

0 Rx
1 (Rx)+
2 –(Rx)
3 x(Rx)
4 @Rx – содержимое регистра используется как адрес
5 @(Rx)+
6 @–(Rx)
7 @x(Rx)

} используются реже

Команды *push* нет, вместо этого пишут *mov r0, –(sp)*, вместо *pop* пишут *mov (sp)+, r0*

Программа копирования строчки, заканчивающейся нулем:

```
; r0 – src
; r1 – dst

L: mov b (r0)+, (r1)+
   Bne L
```

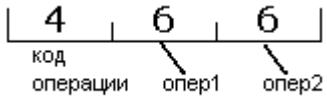
Руками так писать просто, но транслировать в это непросто, так как в исходном языке было написано что-то вроде:

```
i:=1;
repeat
   a[i]:=b[i];
   i:=i+1;
until a[i-1]=chr(0)
```

В частности, одной из причин популярности C может быть то, что в нем эти конструкции схожи с ASM'ом: `while (*a++=*b++);`

Команды с двумя операндами:

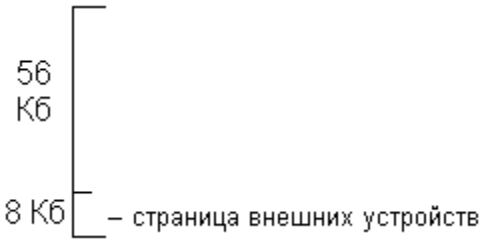
mov
movb
add
sub
cmp
cmpb



По первому биту можно определить, 4 бита используется для кода операции или 10(а то и 16 – безоперандовые команды)

Коды условий – хранят результат последнего сравнения 0, >0, <0 (например, `bne` – «переход по ≠0»). Большинству команд установлены коды условий (иначе пришлось бы вставлять дополнительные проверки).

Адресация:



Прямая адресация в 64Кб (16-битный адрес). Как достучаться до слова в памяти?

```
mov (PC)+, r0
.word 10            (~mov #10, r0)
-> PC указывает сюда
```

Одна из проблем PDP-11 – в невозможности конвейеризации команд. Если же хочется достучаться до элемента по адресу 10, то надо писать

```
mov @#10, r0
```

Были команды `.byte n`, `.word n`, `.ascii "abc"(+'0')` и `.asciz` – автоматически заполнит конечным нулем (все команды заносят операнд в память); например `i: .word [1]` – можно сразу проинициализировать (все метки, используемые как операнды, означают адрес в памяти : `inc@#i`)

Структуры управления – практически сводятся к goto.

position-independent code (PIC) – можно переместить в другое место без заморочек.

`inc @#i – не перемещаемый код (позиционно-зависимый, если только не от PC, у которого адрес абсолютен)`

`inc i – не реентерабельный, но позиционно-независимый`

Реентерабельный процесс – весь контекст зависит от регистров (все данные – в отдельных областях памяти).

Зачем ASM? Например, для переписывания узких мест.
Макропроцессор в ASM тоже есть (а кроме него, есть только в PL/I и в C).

```
.macro name args        ; определяется псевдокоманда с именем name
   …                    ; текстуальная замена
.endmacro (.endm)
```

Пример:

```
.macro push arg
   mov arg, -(sp)        ⇒    push r1   ~   mov r1, -(sp)
.endm
```

Пример: (сохранение регистров)

```
.macro push args
   .irp arg args
      mov arg, -(sp)
   .endr
.endm
```

```
Push <r1,r2,r3>
```

(угловые скобки делают из набора элементов один аргумент). Сами скобки никуда не передаются

Есть значения по умолчанию (,sptr=sp).

В ASM есть раздельная трансляция (модульности, конечно, нет). Те переменные, которые должны быть видны из других модулей, должны быть помечены как `.globl f` (аналог `extern`).

```
...
f:                      (аналог static, если нет globl)
```

Чему соответствует подпрограмма?

```
jsr rx, addr    ;    jump to subroutive
```

```
-(sp) ← rx
  rx  ← pc
  pc  ← addr
```

Есть парная команда rts (return):

```
rts rx
```

```
pc ← rx
rx ← (sp)+
```

Вопрос: Что будет, если написать `jsr pc, x` ? В таких случаях, не заводятся промежуточные регистры; все кладется на стек (~ `call x`)

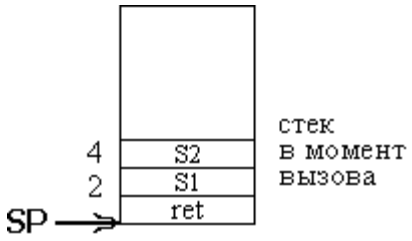
Пример: strcpy(s1,s2). Параметры запихиваются на стек справа налево, запускается функция, а затем вызывавшая процедура чистит стек:

```
mov #a, -(sp)
mov #b, -(sp)
call strcpy
add #4,sp
```

~ *strcpy(b,a)*

- можно было написать *cmp (sp)+, (sp)+* (более компактный код)

```
strcpy:
     mov 2(sp), r0; s1
     mov 4(sp), r1; s2
L:
     movb (r1)+,(r0)+
     bne L
     ret
```



Зачем нужны промежуточные регистры при вызове?

Пример: На Фортране была функция *strcat(b,s1, …)*

```
     jsr r5, strcat
     bne L
     .word
      :  s1
      :  s2
      :
L:
```

⇒

При входе в *strcat:*

```
; r5 → число параметров
movb (r5), … ; сохранили куда-то
```

Пример: ***memmov*** (число копируемых байт – константно)

```
     mov #a, -(sp)
     mov #b, -(sp)
     jsr r5, cpy
(r5) → .word n
(r5)+→ …
cpy:
     mov (r5)+, r0
       …
     rts r5
```

Т.о. за счет соглашения о связях можно вот так сэкономить

Была условная компиляция: *ife, ifne, ifeven*

```
.if …
                ; .iff ≡ else (условие ложно)
                ; .ift – если выполнено условие
                ; .iftr – выполнялось безусловно
.endc
```

Это может быть удобно, если что-то надо делать в начале и в конце предложения

BASIC

BASIC – Beginners All-purpose Symbolic Instruction Code

Создан в 1961 г. для студентов гуманитарных вузов.

Что в нем есть?

Переменные: A … Z, A1… (после буквы можно поставить цифру).
Массивы: A(5), B(5,10)
A(α), где α-выражение – это вырезка. Если $\alpha \notin \mathbf{Z}$, то отбрасывается дробная часть.
BASIC – интерпретирующий язык ⇒ очень медленный

«Удобная среда для создания и редактирования программ». Заключалось в том, что каждой строчке присваивается номер:

```
5 DIM A(10)
10 FOR I=1 TO 10
15 PRINT A(I)
20 NEXT I
```

Рекомендовали писать в каждой строчке несколько операторов, разделяя их специальными символами **:** или \
На самом деле, это очень неудобно – никогда не знаешь, какой длины программа из-за возможности различной нумерации.
Позже появилось: RENUM, RESEQ, которые перенумеровывают строки (соответственно изменяя адреса переходов в операторах GOTO).

Команды BASIC:

1) *PRINT* – печать, возможно использование нескольких параметров
   Если через ‘,’ то вывод через табуляцию
   Если через ‘;’, то через пробел
   Если просто так несколько параметров, то просто так и печаталось
   Можно было написать *PRINT USING <формат>*
   Можно было указать устройство вывода
2) *INPUT* – ввод с клавиатуры
   INPUT “Введите числа” A, B, C
3) *IF B THEN n ~ IF B GOTO n*
   IF B THEN <последовательность операторов для выполнения в той же строчке>
4) Приоритет операций
   0) скобки, функции
   1) ^,¬ (возведение в степень)
   2) – (унарный минус)
   3) *, /
   4) +, –
   5) <, >, =

Потом появились OR, AND, NOT, XOR, IMP-импликация, EQU ~ ⇔. У этих операторов приоритета либо нет, либо OR, AND, NOT, а затем все остальные.

5) Описание переменных

COM – одномерный массив

DIM – многомерный массив (для массивов описание обязательно!)

Были и такие версии Basic'а в которых описание массива также было необязательно. Такие массивы могут быть произвольной длины и реализовывались как «пачка переменных»

6) LET – необязательное слово. A=5 ~ LET A=5

7) GOTO n – безусловный переход; почти все программы на Basic имели оператор GOTO.

GOSUB n – переход с возвратом (вызов подпрограммы).

При это важно то, что у Basic'а все данные глобальны (⇒ нет процедур или функций).

Можно GOTO/GOSUB <переменная> – переход по значению переменной.

8) Циклы: FOR <переменная> = <значение> TO <значение> [STEP <значение>]

Если STEP неуказан, то шаг=1

9) REM (от слова REMARK) – комментарии, печатались на листинге

Модификации Basic'а:

```
CALL addr
  rts, ref
DEF FN y=sinx/x
  FN y(1)
```

Обозначения типов (с помощью добавления суффиксов):

! – целое
% – плавающая точка
# – двойная точность
$ – строка

A1 $NAME
A$(10,10,25) – двумерный массив строк меньших 25 символов.
Есть функции преобразования типов.

Основной язык для игровых приставок – графика, звук и т.д. Пишут интерфейс на Basic'е и вставки на ASM'е.

SCREEN – магическая функция, в зависимости от параметров меняет цвет экрана, звук и т.д.

10) `DATA 5,10,8,7 …`

```
…
READ A,B,C,D
RESTORE
```

– перевернуть стек

11) `ON ERROR` (`GOTO` | `GOSUB`) – обработка ошибок

`ON` α (`GOTO` | `GOSUB`) `5,10,135` – в зависимости от целой части α

если 0 ⇒ на 5

если 1 ⇒ на 10 и т.д.

Есть Quick Basic, Turbo Basic – более продвинутые.

Visual Basic – совсем другой язык, на нем писать удобно.

Pascal/Modula 2

Pascal (Н. Вирт, около 1972 г.), был придуман как язык для обучения, но со временем перерос изначальный замысел и даже собственные возможности.

В первоначальном виде язык был очень простым и минимизировал ошибки с типами (в отличие от С).

Система типов данных (достаточно удачная и позже растиражированная во многих ЯП).

Скалярные типы (представлены одним значением):

Встроенные типы: *integer, char, boolean, real*

Перечислимые типы: *type color = (red, green, blue)*, причем этот тип не совпадает напрямую с *integer*.

Диапазон значений: *type natural = 1 … maxint;* эти типы совместимы с *integer*, но во время работы производится проверка на нахождение в диапозоне.

Отметим, что тип boolean может быть определен как *type boolean = (false, true)*

Составные типы:

1) Массив:

```
type vec10 = array [1..10] of char;
```

здесь может быть диапазон или любой другой тип: chartab = *array [char] of char* и это юудет нормальный массив с индексами типа char.

Многомерные массивы – как массив массивов.

```
type mat = array [1..10] of
           array[0..20] of integer;
```

∼ (полная эквивалентность)

```
array [1..10,0..20] of
        integer;
```

2) Запись

```
type complex = record
      re, im: real;
 end
```

Есть еще указатели: *type ptrint =↑integer;* (ныне записывается как ^integer).

Записи с вариантами:

```
type ptree = ↑tree;
type tag = (node, leaf);
type tree = record
case tag: tag of
     leaf : (info : integer)
     node : (l,r : ptree)
 end;
```

В Pascal’е описание типа должно текстуально предшествовать использованию за исключением описания рекурсивных типов. Кроме того, в каноническом Pascal’е

необходимо все типы описать явно (то есть нельзя описать переменную a: ↑tree, хотя большинство компиляторов это не учитывает).

Типы считаются одинаковыми, если их имена совпадают, либо если они описаны как тождественные некоторому третьему. То есть,

```
var a = array [char] of char;
var b = array [char] of char;
```

Неэквивалентные типы, т.е. нельзя a:=b. Для этого необходимо описать общий тип

```
type  t = array [char] of char;
```

Структура программы на Pascal'е:

```
[program имя;]
[label … ]
[const … ]
[type  … ]
[var … ]
[proc & func ]
begin
end
```

Все блоки должны следовать именно в таком порядке. Любой блок может быть опущен. Современные трансляторы понимают программы, в которых эти блоки встречаются несколько раз, но в каноническом Pascal'е этого нет (равно как и константных выражений).

Операторы

1) a:=expr;

+, –, *, /, div, mod

m[i,j] ~ m[i][j]. Можно присвоить строку m[i]:=m[j], но нельзя вырезать строку или столбец как Алголе 68.

Разыменование указателей: a↑ (это лучше, чем в C++, так как a↑. выглядит нормально, но (*a).b выглядит так плохо, что даже ввели специальный опертаор a→b).

2)
```
if <bool expr>
 then <stmt>
[else <stmt>]
```

Причем <stmt> должно быть именно одним оператором, иначе необходимо окружить операторными скобками *begin … end*

3) Циклы трех видов:

```
while <bool> do <stmt>
repeat <s1;s2;…> until <bool>   – выполняется пока условие ложно
for i:=<b> to/downto <expr> do <s>
```

Начальное и конечное выражения вычисляются только при входе в цикл ⇒ присваивание переменной цикла приводит к непредсказуемым последствиям. Выражение переменной цикла может быть любого перечислимого типа.

4) `with <expr> do <s>`

Здесь выражение должно выдавать переменную типа запись и тогда его внутренние поля доступны как простые переменные

5) Выделение памяти:

`new (p[,tag, …])`

В классическом Pascal'е это единственный способ доступиться к памяти (адрес переменной получить вообще-то нельзя).

Пример

```
var one,two,t: ptree;
begin
      new(one,leaf);
      one↑.tag:=leaf;
      one↑.info:=1;
      new(t,node);
      with t↑ do
         tag:=node; l:=one; r:=two;
      end;
end.
```

Процедуры в Pascal'е

`procedure p (…);`

`function f (…): integer;` – возвращаемое значение может быть только скалярного типа (⇒ не запись)

Параметры могут быть:

а) a: t – по значению

б) var a:t – по ссылке (причем переменная должна быть точно совпадающей по типу)

в) параметры процедурного типа

Процедуры без параметров пишутся без скобок, что порою приводит к двусмысленности прочтения.

Процедуры могут быть **вложенные**, что в более поздних языках не используется. По утверждению Москаля, это не так просто как кажется…

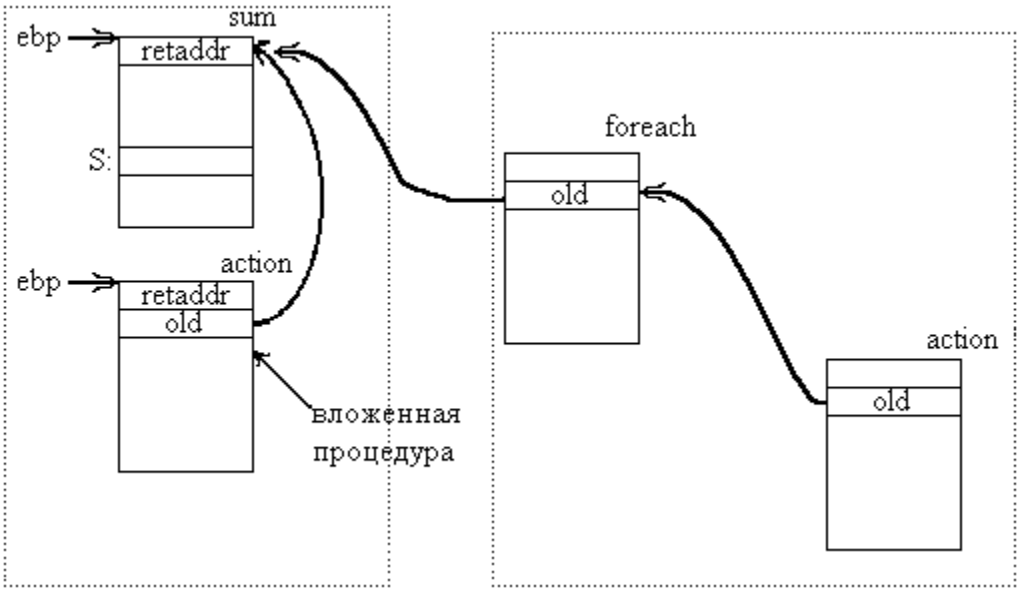
```
procedure foreach (t: ptree; procedure action (var t: integer));
begin
with t↑ do
   case tag of
   leaf: action(info);
   node: begin
         foreach(l, action);
         foreach(r, action);
         end
   end
end;
```

Наблюдаются существенные различия между Turbo Pascal'ем и прочими реализациями (в Turbo Pascal'е необходимо было, чтобы передаваемая параметром процедура была глобальной).

Кроме того, в TP запрещен выход из процедуры по метке.

```
function sum (t: ptree): integer;
var s: integer;
```

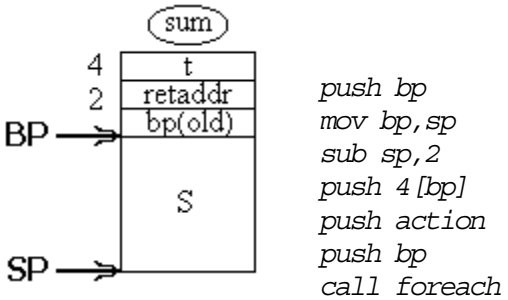
```
procedure action (var i: integer);
begin s:=s+i end;
begin
    s:=0;
    foreach (t,action);
    sum:=s;
end;
```



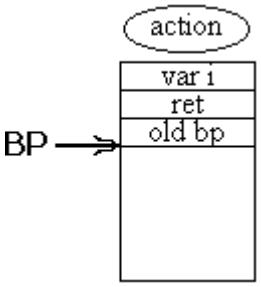
– так ничего работать не будет, ибо достучаться до переменных не удается (поэтому в TP передаваемая процедура должна быть глобальной)

просто вызов влож. процедуры
(до S можно добраться, поднявшись вверх по стеку)

процедура, передаваемая параметром




```
push bp
mov bp,sp
sub sp,2
push 4[bp]
push action
push bp
call foreach
```




```
push bp
mov bp,sp
mov bx, [bp]
mov ax, 4[bp]
add -2[bx],ax
mov sp,bp
pop bp
ret 2
```

Как в нем вызывается *action(i)*?

```
push <i>
mov si,bp(action)
mov di,action
xchg bp,si
call [di]
xchg bp,si
```

Модула 2 – улучшенный вариант Pascal'я.

1) Запретили передавать вложенные процедуры параметром.

2) Поддержка раздельной трансляции (напоминает Pascal'евские unit)

```
DEFINITION MODULE M;
…
END

IMPLEMENTATION MODULE M
…
END
```

В Модуле 2 различаются большие и маленькие буквы.

Все что описано в definition module, должно быть реализовано в implementation module. После этого можно писать

```
IMPORT M;
M.P;          – вызов P из модуля M
END
```

Чем это лучше C-шной системы с include? При трансляции definition module создается табличка, которая может быть подкачана при встрече оператора import ($\Rightarrow$ уменьшаем время трансляции). Кроме того, модули не могут мешать друг другу (в отличие от include'ов, которые могут описывать одно и то же, вызывая ошибку).

В C++ системные заголовки – порядка 10 Мб и при трансляции каждого файла все это заново транслируется.

C/C++

Как возник язык С? Все началось с языка CPL (середина 60-х гг., Кристофер Стрейч). Этот язык так и не был доведен даже до формального описания, но была придумана масса полезных идей.

Упрощенная версия CPL – Basic CPL или BCPL. Был реализован методом раскрутки (bootstrapping), для этого выкинули из языка всю динамику и т.д.

MULTICS – первая попытка написать ОС на языке высокого уровня. Провалилась из-за чрезмерной раздутости и монстрячности. Затем попытались сделать что-то попроще, сделали язык B и ОС UNIX.

Специфика языка была связана со спецификой машины PDP7, для которой он создавался. Адресация на PDP7 была условная, объем доступной памяти – 16 килослов (никаких байтов!) ⇒ сделали автокод бестиповым (все данные – целые).

*(a+i) – это адрес i-го элемента массива. Массив – это указатель на начало, так как ничего другого нет (структур нет).

f(a,b,c) { … } – никаких типов для параметров было не нужно.

Все имена всех полей структур должны были быть различными (эта особенность дожила даже до стандарта Кернигана и Ричи).

Язык С (Керниган, Ричи, 1978 г.) – язык-ядро, компилирующий язык. Основные задачи:

- эффективность;
- низкоуровневость;
- вставки на ASM.

Что появилось нового?

```
1)  union S {
        int a;
        float b;
      }
```

Под S будет отведено памяти ровно столько, сколько нужно под максимальный описанный в нем тип. Удобно в синтаксических анализаторах.

2) Макропроцессор (до того был в PL/I и частично в ASM)

Пример:

file1.c

```
struct S {int a; char b}
    S*  T1;
```

file2.c

```
struct S {int a; chat b}
    S*  T1;
```

Вынесем в один заголовочный файл: **structdef.h** struct s {…}
И заменим на #include.

Существует проблема избыточных описаний – не всегда получается все иерархически организовать и одни и те же определения могут быть включены несколько раз. Против этого применяется следующий трюк:

```
#ifndef STRUCT_DEFS_H
#define STRUCT_DEFS_H

   struct S …
#endif
```

И если данный файл уже был использован, то переопределения не происходит.

С помощью макросов из C++ можно сделать «почти Алгол 68»!

Можно #define int16 int и при переносе не иметь проблем с заменой типов (по делу, конечно, надо использовать typedef).

typedef – синоним типа (⇒ типы int16 и int были бы приводимы).

struct – всегда создает новый тип. Например S1 и S2 – разных типов:

```
struct S1 { int a;}
struct S2 { int b;}
```

Итак, что хорошо в С?

- можно писать эффективно, практически без контроля;
- мощная поддержка и переносимость (хотя размер типов данных отличается, стандартные библиотеки разные и т.д.)
- работа с памятью

Что плохо в С?

- можно описать функцию без параметра, а вызвать с параметром:

```
void f ();
int main ()
{
  f(2);          И это не будет отловлено
}
```

- функции с неопределенным числом параметров (printf – определяет по формату: если формат и переменные не совпадают ⇒ облом);

```
printf("My name is %s%s\n",2);
```

- значения по умолчанию;
- индексация всегда с нуля ⇒ `int a[15]; a[15]:= …;` – это ошибка, но отловлено опять-таки не будет
- интуитивно неясные операторы: `if (a=b) // надо "=="`
- отсутствие break в case

Язык С++ (Страуструп, 80-е)

Что в С++ существенного нового?

- классы (classes);
- шаблоны(templates);
- пространства имен (namespaces);
- перегрузка (overload);
- потоки (streams);
- исключения (exception)

Замечание. Практически нигде С++ в полном объеме не используется, что бы там ни писал Страуструп. Обычно выдергивают какие-то части, а остальное берут из чистого С.

Наиболее часто из этого набора используются, естественно, классы (обычно без множественного наследования). Вместо пространства имен значительно чаще используются

средства макропроцессора. Перегрузка используется достаточно часто (ибо действительно удобно), но иногда ведет к трудно находимым ошибкам или ошибкам времени компиляции:

```
void f1(char);      void f1(long);
void f2(char*);     void f2(int*);

void foo (int i)
{
     f1(i); // неоднозначность
    f2(0);  // также неоднозначность: char* или int
}
```

Потоки удобны тем, что они безопасны с точки зрения типов параметров и универсальны как для строенных типов, так и для типов, определяемых пользователем.

Исключения

Впервые появились в COBOL и PL/I (но там пользователь не мог определить новых исключений). Замечателен оптимизм названия – «исключительные» ситуации возникают сплошь и рядом.

Без исключений приходится делать безусловный переход наружу и там обрабатывать ситуацию. Еще вариант – setjmp/longjmp (глобальный переход). Проблема в том, что не вызываются деструкторы для созданных объектов (подразумеваются в конце блока). Иначе надо проверять коды возврата для каждой функции, проверять успешность каждой операции и т.д. (defensive programming).

Как же выглядит обработка исключений в C++?

```
{
   try {
         … foo(); …
   } catch( … ) {
       // обработка исключений, возникших в блоке try
   }
}
void foo (void)
{
   … if (!something_ready)
            throw 0; …
}
```

Есть проблемы, прежде всего с освобождением памяти. Теперь свертка стека произойдет, НО не будет производиться освобождение памяти для всех ресурсов, управляемых программистом (new/delete).

Для решения этой проблемы предлагается создавать «классы-обвязки» (wrapper classes) с освобождением всех ресурсов в деструкторе класса – довольно неуклюже выглядит на практике.

В C++ исключения могут создаваться пользователем; используется механизм наследования try {throw 1;} catch (long) {} не работает (нельзя приводить базовые типы), но try {throw B();} catch(A) {} будет работать, если B – производный от A/

В C++ НЕТ finally (считается повышающим вероятность ошибок в коде). Если хочется, можно написать опять-таки класс и поместить эти действия в его деструктор.

PROLOG

Язык Prolog (PROgramming in LOGic), около 1970 г. Является языком логического программирования.

Логическое программирование (язык описаний) являются противоположностью к подходу процедурного программирования (языка предписаний).

Описывается модель мира, затем формулируются вопросы, т.е. не говорится «как делать», а спрашивается «верно/неверно»?

Формулируются:

1. Факты
2. Правила
3. Вопросы

1) **Факты**

```
нравится(джон, деньги)
нравится (мэри, рыба)
король(иван, франция)
нравится(джон, мэри)
женщина(мэри)
женщина(лиза)
красивый(лиза)
```

Все должно быть с маленькой буквы! Правдивость в реальной жизни не имеет значения. Важен порядок – сверху вниз.

2) **Вопросы**

```
? – нравится(джон, деньги): да
? – ценный(золото): нет
```

Согласуется ли данный факт с базой данных? СМотрим сверху вниз, ищем совпадающий предикат, затем – совпадающие параметры.

Переменные:

```
? – нравится(джон, X)
```

X=деньги ???

X=мэри;

???

Переменные – с большой буквы. Могут принимать любые значения. Переменные бывают анонимными:

```
? – отец(_,джон)
```
конкретизируется самостоятельно

3) **Правила, конъюнкции, дизъюнкции**

```
женщ_нравится(джон,X) :- женщина(X), красивый(X).
```

С помощью таких правил можно порождать новые факты. Как это работает? Вначале X не кокретизирован, при конкретизации X подставляется во всём правиле.

Цели согласуются по порядку: вначале находим женщину, затем пытаемся узнать, является ли она красивой. Если вторая цель не согласовывается, то производится возврат (backtracking) к первой цели и т.д.

Backtracking ужасно неэффективен.

Для записи дизъюнкции правило записывается второй раз с другой правой частью:

```
женщ_нравится(джон,X) :- женщина(X), нравится(джон,X).
```

В этом случае Мэри тоже подойдет.

Упражнение. Пусть есть предикаты *родитель(X,Y)*, *ребенок(X,Y)*, *женщина(X)*, *мужчина(X)*. Написать предикаты для брата, сестры, дочери, тети и т.п.

4) **Рекурсия**

```
человек(адам).
человек(X) :- человек(Y), отец(Y,X).
```

Таким образом – это либо адам, либо тот, чьим прапра…прадедом был адам.

Очень важен порядок – если поменять факт и правило местами, то программа зациклится на правиле (никогда не завершится).

PROLOG допускает только правостороннюю рекурсию. Надо также избегать кольцевой рекурсии:

```
ребенок(X,Y):-родитель(Y,X).
родитель(Y,X):- ребенок(X,Y).
```

Какие еще бывают неприятности с рекурсией?

Рассмотрим списки: структуры с головой и хвостом, '|' – объединение двух списков. [a,b,c, [a,b]] – список, где [a] – голова, [b,c,[a,b]] – хвост и т.д.

Теперь:

```
список([A|B]) :- список(B).
список([]).
```

В эти правила замечательно укладывается список, приведенный выше, а также другие конкретные списки. Но если задать вопрос:

? – список(X)

произойдет зацикливание. Надо поменять местами факт и правило, а также заменить работу с хвостом на работу с головой списка.

Наконец, рекурсия может быть использована для порождения бесконечного числа элементов, как в следующем примере:

```
целое(0).
целое(X) :- целое(Y), X is Y+1.
```

Этот пример породит 0,1,2,3…

5) **Отсечения**

Предназначены для повышения эффективности программ путем уменьшения числа возвратов. Используются в тех случаях, когда программист знает, что встретив данное решение, дальше искать не надо.

Пример: Библиотечная система, в которой обычным читателям доступны все услуги, а читателям не возвратившим в срок книги (хотя бы одну), доступны только основные услуги:

```
услуги(Читатель, Вид_услуг) :- книга_не_возвр(Читатель,Книга),
                                !,
                                основные_услуги(Вид_услуг).
услуги(Читатель,Вид_Услуг) :- все_услуги(Вид_услуг).
осн_услуги(польз_книгами).
осн_услуги(сдача_книг).
доп_услуги(бан).
доп_услуги(межбибл_абонемент).
все_услуги(X) :- осн_услуги(X).
все_услуги(X) :- осн_услуги(X).
   …
книга_не_возвр('А.Терехов','Феденко').
книга_не_возвр('А.Терехов','Брукс').

читатель('А.Терехов').
читатель('В.Уфнаровский').
```

```
? – читатель(X), услуги(X,Y).
```

Данный вопрос выводит список всех читателей и доступных им услуг. При этом если хотя бы одна книга не возвращена, то происходит отсечение и все сделанные до этого выборы (в том числе, конкретизация переменных) фиксируются, т.е. влево от '!' мы уже не пойдем.

Пример: `foo(X) :- a,b,c,!,d,e,f.`

Здесь после того, как согласованы a,b и c, они больше уже никогда не изменятся, и если мы не согласуем d,e и f при данных условиях, то все правило выдает **нет**. Т.о. '!' – заборчик, через который можно перепрыгнуть только в одну сторону.

С точки зрения нахождения решения отсечение эквивалентно отрицанию.

```
A:- B,!,C.                    A:-B,C.
A:-D.            ~            A:-not(B),D.
```

Но во втором случае перебор существенно больше.

В некоторых реализациях Пролога not выражается через отсечение в явном виде.

```
not(P) :- call(P),!, fail.
not(P).
```

Когда бывают отсечения?

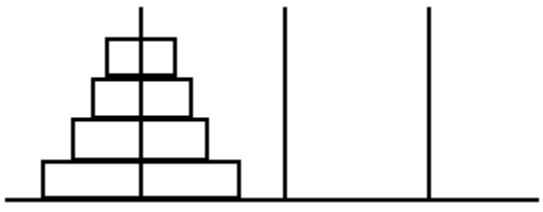
1) нашли условие, из которого ясно, что других согласований не нужно
2) если что-то не так, то сразу закончить перебор обломом (см. определение not через отсечение)
3) в качестве успешного останова (например, в качестве граничного условия, как в следующем примере).

Пример: считаем сумму чисел от 1 до N:

```
сумма(1,1) :- !.
сумма(N,результат):- N1 is N-1, сумма(N1, Результат), Результат is Результат+N.
```

Пример(заключительный)**:**
Ханойские башни

1) перемещать можно только верхнее кольцо.
2) класть можно только на кольцо большего диаметра.



```
ханой(N) :- переместить(N,лев,средн,прав).
переместить(0,_,_,_) :- !.
переместить(N,A,B,C) :-
            M is N-1,
            переместить (M,A,B,C),
            сообщить(A,B),
            переместить (M,C,B,A).
сообщить(X,Y) :-
     write([переместили, диск, со, штыря, X, на, Y]), nl.
```

Occam

Язык Оккам – зык параллельного программирования. Давно было известно, что увеличение числа процессоров от 1 до n не означает ускорения вычислений в n раз. Однако постоянно ведутся работы над созданием эффективной модели для параллельных вычислений. Одной из первых таких моделей стала CSP, предложенная Тони Хоаром (C.A.R. Hoare). Основная идея – применение относительно независимо выполняющихся процессов, обменивающихся информацией с помощью посылки и приема сообщений по каналам (CSP = Communication Sequential Processes, взаимодействующие последовательные процессы).

Недавно возникла еще одна популярная модель параллелизма – BSP (Bulk-Synchronous Parallelism) – 1990 г., Оксфорд, McColl.

Программа на Оккаме – это процесс, описывающий некоторые действия, подлежащие выполнению (т.е. алгоритм).

Какие бывают процессы?

1) Примитивные процессы

а) SKIP – не выполняет никаких действий и завершается (так же полезен в Оккаме, как 0 в десятичной системе);

б) STOP – тоже не выполняет никаких действий, но **не завершается** (аварийная ситуация, например, тупик, запрещенная команда и т.п.)

2) Последовательные процессы

Последовательное выполнение заключается в присваивании переменным определенных значений, на основании которых в дальнейшем принимаются некоторые решения. Присваивание выглядит так:

переменная := выражение

И выражение и переменные не имеют типов и представляют собой просто комбинации битов размером в слово данного компьютера. В некоторых операциях эти последовательности битов используются как числа, логические значения или символы, но ни язык, ни компиляторы Оккама не поддерживают распознование типов (до 2-й версии языка – в ней были введены различные типы данных, что «позволяет отнести эти версии языка к АЯВУ»).

Последовательный процесс (SEQ-процесс) конструируется посредством записи последовательности его компонент, располагаемых одна под другой с небольшим отступом вправо от ключевого слова SEQ.

Пример:

```
SEQ
  x:=3
  y:=x=7
  x:=x+6
  x:=(y+z)/2
```

Последовательный процесс выполняет компоненты в порядке записи (сверху вниз).

Замечание. Система с отступами используется во всем языке для обозначения структуры программы. Про грамматику Оккама по этой причине иногда говорят, что она двумерная. Таким образом, программисту навязывается хороший стиль программирования. Некоторые современные языки (например, Huskill) переняли эту форму записи программ.

```
IF
  n<0
     sign:=-1
  n=0
     sign:=0
  n>0
     sign:=1
```

Просмотр производится сверху вниз, пока не встретим истинное условие – тогда начинает выполняться соответствующий процесс (причем выполняется **только** один процесс). Если истинное условие не встретилось, то возникает ошибочная ситуация и IF-процесс останавливается (т.е. ведет себя как STOP).

Интересно, что в качестве условий можно было бы перечислить n=0, n<=0 и TRUE, и это тоже допустимо по стандарту, но является дурным стилем программирования.

3) Параллельные процессы

Выполнение параллельной композиции (PAR-процесса) состоит в выполнении каждой его компоненты до полного завершения, причем порядок исполнения не определен.

```
PAR
   x:=y-1
   z:=y+1
```

Этот процесс будет работать, но его выполнение будет происходить в неизвестном порядке. Поэтому для PAR установлено правило, гласящее, что ни в одной из ее компонент значение переменной, используемой в любой другой ветви, изменяться **не может**.

Т.о. следующий PAR-процесс – это ошибка:

```
PAR
   x:=y-1
   z:=x+2
```

Как взаимодействуют параллельные процессы?

Посредством ввода/вывода по каналам.

Процесс вывода: *канал ! выражение*
передает значение выражения по каналу
Процесс ввода: *канал ? переменная*
принимает значение из канала и записывает в переменную

Замечание 1. Запись

```
канал ? пер1; пер2; … ; перn
```

эквивалентна

```
SEQ
   канал ? пер1
   канал ? пер2
      …
   канал ? перn
```

Замечание 2. Ввод/вывод эквивалентен присваиванию (если не считать того, что значения и переменная находятся в разных процессах)

```
PAR
   канал ? выражение     ≈   переменная:=выражение
  канал ? переменная
```

Замечание 3. Каждый конец канала может быть использован в одной и только одной компоненте конструкции PAR (подобно правилам использования переменных в параллельных процессах).

Примечание. В переменных принято отделять слова точкой, которая является единственным легальным символом для переменной кроме букв и цифр.

Выбор процесса в зависимости от готовности других процессов

```
ALT
  выражение & (процесс ввода | SKIP) // охраняющий процесс, «сторож»
```

Пример:

```
ALT
  red.selected & red ? x
       cut ! x
  green.selected & gretn ? x
       cut ! x
  NOT (red.selected & green.selected) & SKIP
       cut ! default.value
```

Охраняющий процесс ввода готов, когда выводится соответсвующее значение в соответствующий канал и верно предшествующее выражение. SKIP готов всегда.

ALT выполняет только одну ветвь; если ни один «сторож» не готов, то процесс ведет себя как STOP.

4) Описания данных

Каждое имя должно быть описано до использования (таким образом именуются константы, переменные и каналы)

1) Константы

```
DEF имя = константное выражение
```

Константные выражения отделяются запятыми, а в конце ставится ‘:’

2) Переменные – это битовые комбинации определенного размера. Должны быть описаны текстуально выше процесса, в котором будут использованы. Разделяются запятыми, в конце ‘:’. Изначально их значение не определено (значение появляется в результате присваивания или ввода).

Описания переменных обычно размещаются перед SEQ-процессом.

3) Каналы – описываются так же, как и переменные, но с ключевым словом CHAN. В области действия описания любого канала должно быть 2 **параллельных** процесса: один передающий, другой принимающий. Обычно описания каналов размещают перед PAR-процессами.

4) Массивы – бывают только одномерные (двумерные появились во второй версии). Единственный способ структуризации в Оккаме.

5) Описание процесса

Можно давать имена процессам; процесс сам себя вызывать не может; в конце описания процесса ставится ‘:’

У процесса бывают параметры, следующих типов:

VALUE – значение
VAR – переменная
CHAN – канал

Никаких возвращаемых значений!

```
PROC assign.character ( VAR x, VALUE s[], i) =
    x := s[BYTE i) :
VAR ch:
SEQ
   assign.character(ch, "This is a string",13)
```

6) Циклы

1) Неограниченные циклы WHILE – обязательно последовательные (в сущности, обычный while, как в Pascal или C++

```
SEQ
   x:=0
   v[n]:=key
   WHILE v[x]<>key
     x:=x+1
```

2) Ограниченные индексируемые циклы FOR:

Могут быть представлены как массив процессов; форма записи:

имя = [база FOR количество]

Таким образом SEQ-FOR эквивалентны циклам for в Паскале или Алголе:

```
SEQ year = [1280 FOR 62]                SEQ
   Celebrate.Christmas(year)   эквив.     Celebrate.Christmas(1280)
                                            …
                                          Celebrate.Christmas(1341)
```

Замечание. Нельзя использовать индекс цикла слева в присвоении и не надо описывать его за пределами цикла.

IF–FOR обеспечивает ограниченный поиск.

Пример:

```
DEF otherwise=TRUE,
    not found=n:
IF
  IF i=[0 FOR n]
   v[i]=key
       x:=j
  otherwise
      x:=not.found
```

Переменная x будет равна индексу элемента с совпадающим значеним или not.found, если значение key не входит в массив.

В принципе, в данном примере использован нехороший стиль программирования (TRUE в условии IF).

Перейдем к практическому применению параллельных программ на практике.

Программа параллельного умножения матрицы на вектор

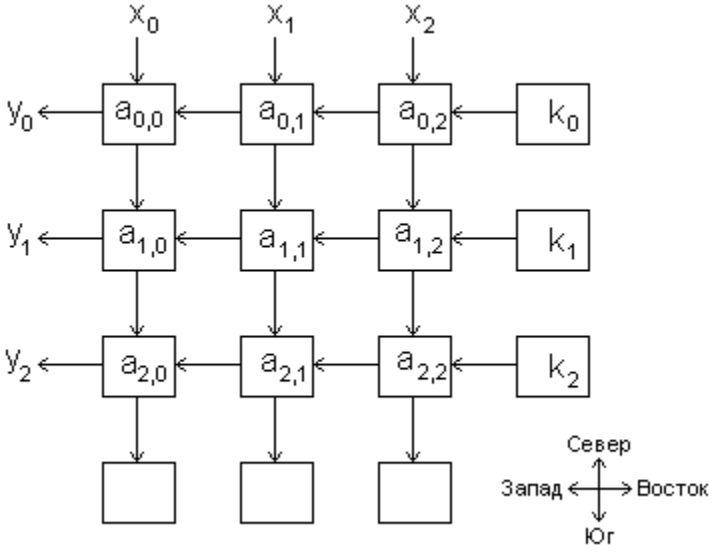
Рассмотрим геометрическую задачу – поворот или перенос объектов. Для каждой точки (x_0, x_1, x_2) необходимо вычислить соответствующие преобразованные координаты (y_0, y_1, y_2) по формуле

$$y_i = \sum_{j=0}^{2} a_{ij} x_j + k_i$$

Вычисления по этой формуле требует 9 умножений и 9 сложений для каждой точки в представлении объекта. Если изображение выводится на экран, и пользователь может интерактивно влиять на процесс, то скорость вычисления критична: если используется 1000 точек, то требуется 100000 умножений матриц на вектор в секунду.

Наиболее существенные временные затраты в этой формуле – это умножение $\Rightarrow$ преимущество будет иметь структура, обеспечивающая выполнение максимального числа умножений.

Естественной конфигурацией для решения этой задачи является квадратная матрица процессоров, копирующая матрицу *a*, каждому элементу которой соответствует процессор, выполняющий умножение на этот элемент.



Последовательные значения координат *x* вводятся в массив с Севера на Юг, а последовательные значения преобразованных координат *y* выводятся с Востока на Запад с левой стороны матрицы. Каждый процессор помечен параметром за который он ответственен.

Будем считать, что преобразование постоянно (иначе нужен дополнительный массив каналов справа).

Каждая ячейка умножителя в процессе умножения на матрицу выполняет 3 задачи:

- получение следующей координаты x_j от северного соседа и передача южному;
- умножение;
- получение частичной суммы от восточного соседа, сложение с собственным результатом и передача суммы на Запад

Эти задачи могут быть *последовательно* выполнены следующим образом:

```
VAR xj, aij.times.xj,yj:
  WHILE TRUE
    SEQ
      SEQ
         north ? xj
         south ! xj
      aij.times.xj := aij*xj
      SEQ
         east ? yi
         west ! yi+aij.times.xj
```

Этот процесс никогда не завершается и выполняется в строго фиксированном порядке.

Попробуем оптимизировать программу – реорганизовать ее так, чтобы независящие друг от друга части выполнялись в произвольном порядке:

```
PROC multiplier (VALUE aij, CHAN north, south, west,east)=
  VAR xj, aij.times.xj,yj:
  SEQ
    north ? xj
    WHILE TRUE
       SEQ
          PAR
             south ! xj
             aij.times.xj := aij*xj
             east ? yi
          PAR
             west ! yi+aij.times.xj
             north ? xj        :
```

Интересно, что multiplier не зависит от i, j $\Rightarrow$ при аппаратной реализации можно использовать 9 одинаковых схем.

Вдоль восточного края матрицы разместим источники значений смещений k_i:

```
PROC offset (VALUE ki, CHAN west)=
   WHILE TRUE
       west ! ki    :
```

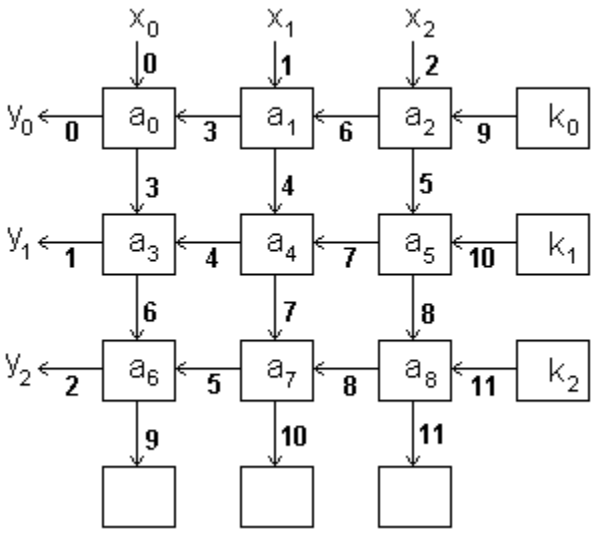
Кроме того, на южном края должен быть предусмотрен сток для избыточных значений x_j (необходимо для того, чтобы в multiplier всегда можно было бы делать вывод).

```
PROC sink(CHAN north)=
   WHILE TRUE
       north ? ANY    :
```

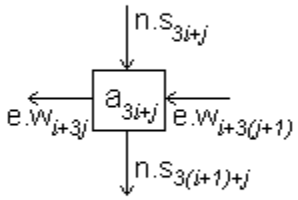
Объединение этих компонент в матричном умножителе состоит в выборе подходящего способа перечисления каналов и использовании массивов CHAN с подходящей индексацией, например:

```
DEF n=3:
VAR a[n*n], k[n] :
SEQ
    … инициализация a,k
CHAN north.south [(n+1)*n], east.west[n*(n+1)]:
PAR
  PAR j=[0 FOR n]          - - генератор координат x[j]
    produce.xj(j,north.south[j])
  PAR                      - - умножитель матрицы на вектор
    PAR i=[0 FOR n]
      offset(k[i],east.west[(n*n)+i])
    PAR i=[0 FOR n]
      PAR j=[0 FOR n]
        multiplier ( a[(n*i)+j],
                     north.south[(n*i)+j],
                     north.south[(n*(i+1))+j],
                     east.west[i+(n*j)],
                     east.west[i+(n*(j+1))] )
    PAR j=[0 FOR n]
      sink(north.south[(n*n)+j])
  PAR i=[0 FOR n]          - - потребитель координат y[i]
    consume.yi(i,east.west[i])
```

Для данной программы мы все перенумеровали в одномерную запись, как того требует Оккам:



Каналы:



Если данную программу выполнять на однопроцессорном компьютере, то будет медленнее, из-за накладных расходов на организацию процессов.

Однако на специализированном оборудовании все это будет существенно быстрее. Наиболее длинный путь для данных от ввода до вывода – от x_2. До использования в вычислении y_2 (6 передач, 3 сложения, 1 умножение). Все эти действия выполняются последовательно.

Программа спроектирована в предположении, что время затрачиваемое на одно умножение больше времени, необходимого для выполнения всех других действий. Если это верно, то параллельная программа почти в 9 раз быстрее последовательной программы.

Функциональные языки программирования

Обоснование функционального программирования. В традиционных программах ключевую роль играет присваивание, но оно вводит потенциальную зависимость по данным между любыми двумя операторами. В функциональном программировании не надо думать о «состоянии программы». Кроме присваивания, мы отказываемся еще и от косвенных эффектов. Взамен мы можем получить сборку мусора.

На сегодняшний день функциональное программирование надо воспринимать не как новый подход, а как попытку обобщения процедурных языков программирования.

30-е гг.: λ-исчисление (Черч):

здесь x – связанная переменная; все остальные – свободные переменные

λx. <λ-term> – аналог текста процедуры в Алголе 68
<λ-term><λ-term><id> – аналог вызова процедуры с параметром

Неоднозначность разрешается с помощью скобок; некоторые скобки подразумеваются:
(λx.xx) ~ (λx.(xx))
xyz ~ (xy)z – левоассоциативность

Оказывается, в такой формализации алгоритма достаточно удобно программировать (в отличие, скажем, от машины Тьюринга).

Пример. $(\lambda x.x*x)2 \overset{\beta-редукция}{\Rightarrow} 2*2$
Как определяются базовые алгоритмы конструирования в λ-исчислении?

1) *let x=y m e ~ (int x=y; e)* ~ (λx.e)y
Есть несколько преобразований:
1) α-конверсия (позволяет заменить связанные переменные в пределах терма и все свободные)
2) β-редукция (замена формального параметра на фактический):
(λx.T)T′ ⇒ T[T′/x]

2) *true: λx.λy.x* – функция, выдающая первый параметр
false: λx.λy.y
if c then a else b: c a b – если сюда вместо c подставить *true*, то получим первый параметр, иначе второй параметр

<x,y>: λf.f x y
fst <x,y>: <x,y> true

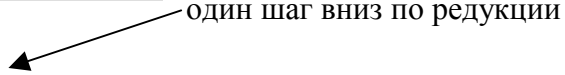
Проверим: *(λf.f a b) (λx.λy.x)* $\overset{подставим}{\Rightarrow}$ *(λx.λy.x) a b* $\overset{true(a,b)}{\Rightarrow}$ *a*
Числа также выражаются через *true*, *false* и упорядоченную пару

3) Рекурсия

f(Yf) = Yf – операция фиксированной точки (переводит значение в самого себя)

```
fact n=if n=0 then 1 else n*fact(n-1)
fact=Y λf.λn.if n=0 then 1 else n*fact(n-1)
```

Обозначим `λf.λn.if n=0 then 1 else n*fact(n-1)` за *F*




```
fact 2 = (YF)2 ⇒ F(YF)2 ⇒
if 2=0 then 1 else 2*(YF)(2-1) ⇒ 2*(YF)=1 ⇒ …
```

К вопросу о существовании такого Y

```
Y = λf.(λx.f(xx))( λx.f(xx))
```

λ-вычисление порядок вычислений (энергичный и ленивый) не определяет; если мы будем всегда подставлять значение в самый внешний терм, то мы придем к нормальной форме (если таковое вообще возможно).

В принципе, если ленивый и энергичный оба доходят до конца, то мы всегда получим одинаковый результат.

LISP (LISt Processing, McCarthy, 1958). От него был порожден диалект с убранными наслоениями и упорядоченным синтаксисом. **Scheme** (~ 1978 г.)

Все конструкции имеют следующий вид:

- идентификатор или константа;
- expr: id | const | (<expr>+)

Лексика: кроме скобок, практически все символы могут встречаться в идентификаторах. Практически всегда первое слово – функция, все последующие ее аргументы:

```
(+ 1 2)                          – легальная запись на Scheme
(define <id> <expr>)             – определяет конструкцию
(lambda (x y) (+ x y))           – текст функции без имени; если хотим чтобы ее можно
                                   было вызвать, то необходимо ее обозначить:
( define add
   ( lambda (x y) (+ x y))       – определили функцию
)
(add 1 2)                        – вызвали с двумя параметрами
                                   [ можно было написать (define add +) ]
```

Так как определять функции приходится часто, то эту часть можно переписать так:

```
( define (add x y) (+ x y))
```

Замечание В define можно использовать рекурсию определения

В Scheme есть целые числа, рациональные числа и числа с неограниченной точностью; есть булевские значения #t, #f; есть пустое значение ().

Scheme – есть язык с динамической типизацией (ломается только во время исполнения).

Пара: $(e_1.e_2)$

Для пар имеются две функции – car, выбирающая первый элемент, и cdr, выбирающая второй элемент.

Существует функция (cons a b), образующая пару из a,b ⇒ (a b)

Специальный вид пар это список :

либо (a.()), либо список, т.е. список всегда выглядит как (a.(b.(c.(… (x.()) …)

Списки можно записывать как (a b c … x)

Правила вычисления выражения для списка: первый элемент должен быть функцией, все остальные должны быть аргументами (их придется вычислять заранее, они должны быть легальными выражениями).

Т.о. *(cons a (b c))* – это ошибка, т.к. b – не функция.

Для того, чтобы вычислять значение, есть функция (quote b):

(cons 'a (b c)) – будет работать нормально!

Пример:

```
(define (foldl f acc l)
    (if  (null? l)
         acc
      (foldl f (f (car l) acc)
                  (cdr l))
  ))
```

null? – предикат (принято заканчивать на вопросик). Здесь реализуется хвостовая рекурсия (fail-recursion), которая на практике реализуется как цикл, причем заведомо не переполняющий стек.

```
(foldl + 0 'a (1 2 3)) ⇒ (3+(2+(1+0)))
```

бинарная операция — счетчик — параметр

(foldl cons '() '(1 2 3)) ⇒ 3.(2.(1.())) – т.е. развернули список задом наперед

Пример:

```
(define (unfoldl f arg)
     (let ((p (f arg)))
        (if  (null? p)
             '()
        (cons (car p)
            (unfoldl f (cdr p)))
)))
(define (num → dig n)
     (if (= n 0) '()
         (cons (remainder n 10)
               (quotient n 10)
)))
```

«Теперь мы все это радостно используем», но не сразу…

```
(define (dig → num n m)
        (+ n (* m 10)))
```

Следующие две строчки выдадут результат 5691 (путем превращения числа 1965 в список (5 6 9 1), а затем собирания его в число):

```
(foldl dig → num 0
```

```
(unfoldl num → dig 1965))
```

Функциональное программирование: общая идея в том, что оператор присваивания считается вредным и без него стараются обойтись.

Язык ML (1978 г., разработан для системы автоматического доказательства LCF)

1. ML – язык со статической типизацией
2. Система типов Hindley & Milner
3. ML – компилирующий язык (хотя чаще генерят байткод)

1987 г. – пересмотрен и назван Standard ML. Сейчас же мы будем рассматривать язык Objective Caml (O'Caml), который разрабатывался во Франции.

Типы данных

Стандартные – int, char, float, string, bool, unit (~ void в C++, Algol 68). Есть стандартная константа типа unit: ().

ML можно считать логичным и последовательным обобщением Алгол 68 (© Москаль)

Производные типы:

1) функции: int → int
с несколькими возвращаемыми значениями: int → int → int (для сложения)

2) λ-выражения

```
fun x → <выражение>
```

x – идентификатор, в общем случае образец, например:

```
fun x → x+1
```

Интересно, что типы можно нигде не писать, а компилятор сам определит тип (например, в примере: 1 – целое, '+' определен над целыми ⇒ все целое).

Вопрос: как будет типизировано выражение `fun x → x` ? Результатом будет **полиморфный тип:** `'a → 'a` (вместо a можно написать что угодно. Типы однобуквенные согласовываются одинаково). (Плавающая сложение записывается '+.')

```
let id
   fun x → x
id 1
```

происходит типизация функции как int

Почему все это не приводит к противоречиям – по причине хорошего проектирования языка (см. ???)

Если знак заключен в скобки, то он используется как идентификатор (аналог слова operator в C++). Несколько сокращений (можно не писать `=fun fun x → fun y` для 2-х аргументов и т.д.)

```
let (>>) x f = f x
(сокращение от let (>>) = fun x → fun f → f x)
```

Теперь можно писать `x >> sin >> cos` вместо `cos(sinx)`

Заметим, что в функциональных языках аргумент функции пишется сразу после функции без скобок (⇒ в примере cos sin x скобки ставятся как (cos sin) x).

Из описания >> видно, что f имеет вид *'a → 'b,* выдаваемый результат равен выдаваемому значению *f='b*; итого имеем для >> (что уже неплохая информация о типах):

`'a → ('a → 'b) → 'b` что уже неплохая информация о типах

В C++ для записи такой функции пришлось бы использовать конкретный тип, либо template (хотя работает плохо на приведенном примере – ни x, ни sin классами не являются).

Пример (классический, композиция функций):

```
let compose f g =
          fun  x → f (g x)
('a → 'b ) → ('c → 'a) → ('c → 'b)
```

тип f тип g тип compose

Как реализуется полиморфизм? Любой тип укладывается в слово, либо ставится указатель на адрес со значением (плохо для плавающих чисел). Полиморфные функции.

В ML существуют переменные и присваивания. Существуют конструкторы с параметром (параметр записывается перед конструктором). Пространство имен и пространство типов не пересекаются. Имена конструкторов – с большой буквы, имена переменных – с маленькой. Case sensitive.

```
ref:  'a → 'a ref          (аналог new, по значению создает переменную)
(!):  'a ref → 'a          (аналог '*', выдает значение переменной)
(:=): 'a ref → 'a → unit   (":=" выдает пустое значение)
```

Пример.

```
(( let n = ref 0 in
 (( for i=1 to 10 do
      n:=(!n)+i
  done);
   !n))
```

; – бинарная операция; считает первое значение, его затем убивает и выводит второе значение.

Вместо '(','))' можно писать *begin* , *end* в качестве структурных скобок.

В этом примере все переменные целые, т.к. используется *ref 0*, т.е. целые.

Алгебраические типы данных (на самом деле, скорее тэгированные объединения), обычно это рекурсивные типы – либо пустые, либо набор (голова, хвост), возможно с рекурсией.

```
type 'a list =
     Nil
   | Cons of 'a*'a list
```

'*' образовывает упорядоченную пару из двух элементов. Используется рекурсия. Когда в такой список запихают переменную какого-либо конкретного типа, список будет принимать переменные **только** такого типа (встроенные списки реализуются именно так).

Пример использования списка: `cons (1, cons(2, Nil))`. Т.к. списки встроены, то это можно записать как [1;2]

```
let rec map f l =
     match l with
      Nil → Nil
    | Cons (hd, tl) →
         Cons (f hd, map f tl)
let div a b =
    (a/b, a mod b)
 int → int → int*int
let (q,r) = div n 10 in …
```

Сравнение с образцом после левой части образца можно также вставить `when <expr>` Словечко `rec` ставится для того, чтобы можно было использовать в правой части переменную из левой части `let`.

Здесь тоже упорядоченная пара, '*' используется только в типах (см. типы функций). И дальше используется пара q,r…

Пример (символические вычисления. Будем упрощать выражения):

```
type expr
   Con of int
 | Var of string
 | Bin of (expr*string*expr)

let rec simplify e =
       match e with
        Bin (a,op,b) →
          let rec s' e = match e with
              (con a, "+", con b) → con(a+b)
            |(con a, "*", con b) → con(a*b)
            |(con n, ("+"|"*") as op,b) → s' (b,op con n)  ???
            |(a, "+", con 0) → a
            |(a, "*", con 0) → con 0
            |(a, "*", con 1) → a
            | _ → Bin e                    т.е. не упростилось
     in

       s' (simplify a,op, simplify b)
  | _ → e
```

Замечание Апостроф не в первой позиции – обычная буква

ML – язык с самой мощной модульностью. Модули могут сколь угодно много раз вкладываться, бывают параметризованные модули (причем это не приводит к генерации кода).

Haskell

Язык чисто функциональный (нет присваивания, последовательного выполнения и т.п.) язык с ленивыми вычислениями, строго типизированный (~ такой же как в ML, но более навороченный, например, параметром типа может быть его конструктор, решена проблема перегрузки операторов):

Пример (бесконечный список):

```
from n = n: from (n+1)
```

‘:’ – аналог cons $\Rightarrow$ бесконечный список. Однако, напечатать такой список нельзя. Можно воспользоваться функцией *take*, печатающей первые k чисел списка: `take 10 from 0`

Пример (бесконечный список простых чисел. «Ленивое решето Эратосфена»):

```
primes = from [2..] where
   from (n: tail) = n:
       from [m|m ← tail, m `mod`n /=0]
main = putStr (show(
          take 20 primes ))
```

Smalltalk

Smalltalk - разработка фирмы Xerox, около 1970 г., сугубо ОО-язык. Smalltalk – первый чистый ОО-язык; Smalltalk представляет из себя систему, которая абстрагирует пользователя от реального мира с помощью виртуальной машины (но даже ее пользователь может переписать). Объекты обладают модульностью ($\Rightarrow$ раздельная трансляция). ST позволяет управлять сложностью системы.

Из чего состоит система Smalltalk?

1. **Объекты** – простейший компонент системы. Содержат в себе данные и методы (операции над данными). Наружу из объекта торчат интерфейсы.
2. **Сообщения** – все управление программой производится с помощью сообщений. Сообщение состоит из:
   - объекта-получателя;
   - имени сообщения;
   - [аргументы].

   Интерфейс – это множество всех сообщений, на которое объект может ответить.
3. **Класс** – описывает тип объекта и его структуру. Бывают данные класса и методы класса (которые разделяются между всеми **экземплярами** класса).
4. **Переменные** – это объект, принадлежащий памяти какого-то объекта и ссылающаяся на какой-то другой объект (может ссылаться на специальный объект Nil). Существуют правила видимости, контексты.

   Имена классов, данных классов должны начинаться с большой буквы. Переменные бывают разные:

   а) доступные только одному объекту
   - переменные экземпляра;
   - временные переменные (например, переменная, объявленная в блоке);

   б) общие переменные, доступные либо экземплярам данного класса, либо всем объектам, либо выборочно:
   - переменные класса;
   - глобальные переменные;
   - переменные пула.

Программирование на ST состоит из создания иерархии классов, описания внутренности этих классов и правил взаимодействия между ними. Существует достаточно много предопределенных классов, что сильно облегчает программирование.

Иерархия классов

Создается с помощью **наследования.** Наследующий класс называется **подклассом**, от которого наследуется – называется **суперклассом**. Все, классы от кого-то наследуются (наиболее часто от Object'а – некоторого класса, у которого нет суперкласса). Множественного наследования как такового нет.

Отрывок из иерархии классов

```
Object
    Magnitude
        Character
        Date
        Time
        Number
            Float
            Fraction
            Integer
                LargeNegativeInteger
                LargePositiveInteger
              SmallInteger
    Collection
        SequenceableCollection
            LinkedList
                 Semaphore
            ArrayedCollection
                 Array
                 Word Array
                     Display Bitmap
                 Run Array
                 String …
```

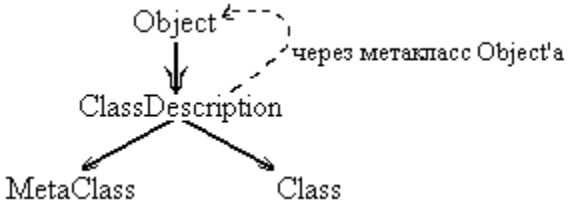
Почему надо знать иерархию объектов? Потому, что сообщение, которое не может быть обработано, передается на обработку его суперклассу, затем его суперклассу и т.д. Если дошли до Object'а и не смогли обработать, то высылается ответное сообщение doesNotUnderstand: (':' – т.к. системное сообщение).

О гармонии мироздания:

Объекты – являются экземплярами класса.

Классы – если все является объектами ⇒ у них тоже должны быть свои прототипы ⇒

Метаклассы – невидимы для пользователя; классы являются экземплярами метакласса. Все метаклассы являются экземплярами MetaClass.



Синтаксис языка:

Выражения:

1. Литералы
2. Имена переменных (только эти выражения зависят от контекста)
3. Выражения посылки сообщений (значение определяется классом-получателем)
4. Выражения-блоки

Массив:

```
# ('1' '2' '3')
# ((1 2) 3)
```

Внутри массива могут быть другие массивы (но они решеткой не обозначаются). Массивы могут быть неоднородными.

Присваивания:
Обозначаются “←” (в более поздних версиях “:=” тоже разрешено)

Сообщения:
Имя получателя
Имя сообщения
Сообщения могут быть:

- унарные alpha sin (могут выдать результат или ошибку)
- бинарные Number+1 (результат будет выдан наружу)
- ключевые ages at: 'Вася' put 3 (эквивалентно at: put: 'Вася' 3, т.е. сообщение с 2 переменными)

Правила разбора сообщений:

1. Унарные и бинарные – слева направо
2. Бинарные выше по приоритету, чем ключевые
3. Унарные выше бинарных
4. Выражения в скобках выше унарных

Каскадирование сообщений:

```
Collection new add:1; add:2; add:3
```

что эквивалентно

```
|temp|                          Описание переменных заключается в |…|.
 temp ← Collection new.         Операторы разделяются “.”
 temp ← add:1.
 temp ← add:2.
 temp ← add:3.
```

Выражения-блоки:

```
[index ← index+1]
```
– вычисляется только по получении сообщения value

Т.о. записывается, например, цикл for:

```
4 Times Repeat: [index ← index+1]
```

Если послать value блоку [], то получим Nil

Условные выражения:

```
(number \\ 2) = 0
    if True:  [parity ← 0]
    if False: [parity ← 1]
```

```
[: array | total ← total+array size]
```
– параметром будет передан массив и добавится его размер

Пример:

```
Class name     History
superclass     Object
instance variable names allIncomes incomes
class variable names RateTax
class methods
    initialize
         initialize
              RateTax ← 0.12
   instance creation
         initial Balance : amount
             ^ super new setInitialBalance : amount
         new
             ^ super new setInitialBalance : 0
instance methods
    transaction recording
        recieve: amount  from: source
             incomes at: source
             put: (self totalRecievedFrom : source)+amount
   inquires
       totalRecievedFrom: source
         (incomes includekey: source)
             if True:  [^incomes at: source]
             if False: [^0]...
```

Заключительная лекция

Subtyping – подтипы (строятся наследованием, но не только им).

Определение. Тип A является подтипом B, если в любом контексте, где можно использовать B, допустимо использование A. Обозначается A<:B. В частности, для простейших типов множество значений A является подмножеством множества значений B.

Пример (Модула 3): 1..10 <: Integer (диапазон является подтипом)

Мы будем рассматривать простейшую систему типов, в которой имеются:
void, int, AxB, A→B, refin B, refout B, refio B

Свойства:

A:<A

A<:B & B<:C ⇒ A<:C

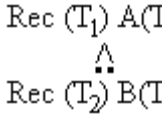
A<:void (т.е. там, где нужен void, там допустим любой тип. Аналогично, наследование сужает множество значений, а не наоборот).

A<:B, C<:D ⇒ AxB<CxD (это верно для ML или Java, но не для C++, для которог должно быть A=B)

(т.е. для того, чтобы тип «функция» был подтипом другого типа «функция», необходимо, чтобы множество входных значений было **меньше**)

refin A <: refin B ⇔ A<:B
refout A <: refout B ⇔ B<:A
refio A <: refio B ⇔ A<:B, B<:A т.е. они практически одинаковы (но не равны!)

Ситуация становится менее тривиальной, если появляются рекурсивные типы: rec(T)=…(T)…



T1<:T2 ⇒ A(T1) <: B(T2)
Для этого и нужна вся эта теория, т.к. для рекурсивных типов множество их значений совсем неочевидно.

1. Динамическую типизация можно представить как статическую типизацию всего с одним типом – объект. Т.о. в LISP'е можно реализовать все типы, начиная с объединения и одного типа.
2. На что похожи объекты? На запись активации процедур.
   Simula 67 – в ней объект описывается точно так же, как процедура, конструктор – вызов процедуры (единственная разница в том, что после вызова такая процедура не умирает).

3. Обработка исключительных ситуаций – как это делалось в Алголе 68:

```
mode exchand = proc (excarg) void;
proc (excarg) void error :=
   (excarg a) void : (print(("exception")); stop)
```

Теперь мы хотим сделать культурную обработку ошибки (аналог try-catch):

```
exchand old = error;
 error := (excarg a) void :
    if a = overflow then
         goto l
    else
 error := old; old(a)
        fi;
...
l:
   error:=old
```

T.o. exception похож на goto с процедурными значениями.

Аналог:

```
try { … }
catch (overflow) { обработка}
```

4. Аналогия полиморфных типов в ML и template'ов в C++:

```
template <class A, class B>
   A apply (A (*f)(B), B arg)
    { return f(arg);}
```

Затем можно будет написать apply(sin, 3.141).

И теперь как это в ML:

```
let apply f arg = f arg
```

Ее тип: `apply ('b → 'a) → 'b → 'a`

Указатель литературы

1. **М. Бен-Ари** *"Языки программирования. Практический сравнительный анализ"*, М.: Мир, 2000, 366 с.
2. **Т. Пратт** *"Языки программирования: разработка и реализация"*, М.: Мир, 1979
3. **Ф. П. Фишер, Д. Ф. Суиндл** *"Системы программирования"*, М.: Статистика, 1971
4. **Г. Катцан** *"Язык Фортран 77"*, М.: Мир, 1982, 208 с.
5. *"Алгоритмический язык Алгол 60. Модифицированное сообщение",* М.: Мир, 1982, 72 с.
6. **Г. Ф. Дейкало, Б. А. Новиков, А. П. Рухлин, А. Н. Терехов** *"Новые средства программирования для ЕС ЭВМ: Транслятор с языка Алгол 68 и диалоговая система JEC"*, М.: Финансы и статистика, 1984, 207 с. (часть I: "Введение в Алгол 68")
7. **Г. В. Вигдорчик и др.** *"Основы программирования на ассемблере для СМ ЭВМ"*, М.: Финансы и статистика, 1987, 240 с.
8. **Брусенцов Н.П.** *"Миникомпьютеры"*, М., "Наука", 1979 (глава 9: "Многорегистровые миникомпьютеры DEC PDP-11")
9. **М. Кергаль** *"Методы программирования на Бейсике (с упражнениями)"*, М.: Мир, 1991, 288 с.
10. **Н. Вирт** *"Язык программирования Паскаль"*, Л.: Изд-во ЛГУ, 1974, 49 с.
11. **К. Йенсен, Н. Вирт** *"Паскаль. Руководство для пользователя"*, М.: Финансы и статистика, 1989, 255 с.
12. **Б. Керниган, Д. Ритчи** *"Язык программирования Си"* (2-е изд.), М.: Финансы и статистика, 1992, 272 с.
13. **Б. Страуструп** *"Язык программирования C++"* (3-е изд.), СПб, М.: "Невский диалект", "Бином", 1991, 991 с.
14. **А. Сергеев** *"Исключительные меры для чрезвычайных ситуаций"*, BYTE/Россия, №7–8, 1999, с. 62–69.
15. **У. Клоксин, К. Меллиш** *"Программирование на языке Пролог"*, М.: Мир, 1987, 336 с.
16. **Д.Ж. Стобо** *"Язык программирования Пролог"*, М.: Радио и связь, 1993, 368 с.
17. **Г.С. Цейтин** *"От логицизма к процедурализму. На автобиографическом материале" //* В сб.: Алгоритмы в современной математике и ее приложениях. ВЦ СОАН, Новосибирск, 1982, ч.2, с. 181–193.
18. **Г. Джоунз** *"Программирование на языке Оккам"*, М.: Мир, 1989, 208 с.
19. **Ч. Хоар** *"Взаимодействующие последовательные процессы"*, М.: "Мир", 1989, 264 с.
20. *"Справочная книга по математический логике"*, книга IV, глава 7: **Х. Барендрегт** *"Бестиповое лямбда-исчисление"*, М.: "Наука", 1983
21. **А. Филд, П. Харрисон** *"Функциональное программирование"*, М., "Мир", 1993
22. **С. С. Лавров, Г. С. Силагадзе** *"Автоматическая обработка данных. Язык ЛИСП и его реализация"*, М.: "Наука", 1978
23. *"Revised[5] Report on the Algorithmic Langauge Scheme"*, available at http://www.swiss.ai.mit.edu/projects/scheme/index.html
24. **M. Mauny** *"Functional Programmig using Caml Light"*, January 1995, available at http://cristal.inria.fr/tutorial/index.html
25. *"Report on the Programming Language Haskell 98"*, available at http://www.haskell.org/definition/
26. **Д. Вебер** *"Технология Java™ в подлиннике"*, СПб: BHV – Санкт-Петербург, 1997, 1104 с.