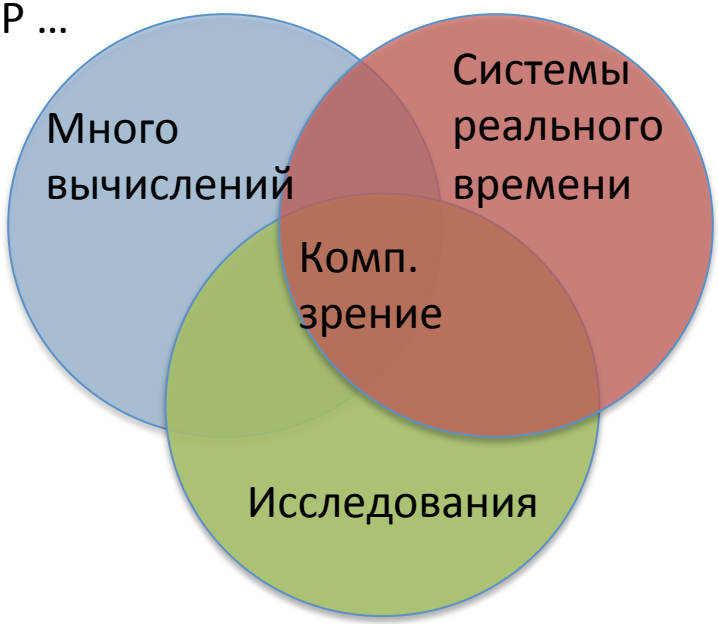
Язык программирования numl

Вадим Писаревский, itseez

Компьютерное зрение – уникальная область

- Большая вычислительная нагрузка:
  - обработка HD/FullHD в реальном времени: 1920x1080x3x30 ~ 200Mb в секунду для однопроходных алгоритмов, гигабайты в случае стерео, детектирования и т.д.
- Разнообразие структур данных и алгоритмов:
  - 1D/2D/3D/nD плотные и разреженные массивы, деревья поиска, геометрические объекты, облака точек, пирамиды и т.д.
- Трудно обрисовать “движок” компьютерного зрения – много новых алгоритмов и вариаций алгоритмов, очень много приходится писать с нуля
- Необходимость работать на различных мобильных/встроенных архитектурах: x86/ ARM/MIPS/GPU/DSP …



Главные свойства ПО

1. Правильность
   - Разумные результаты, надежность
2. Время на разработку, отладку, внедрение
3. Скорость работы
4. Дополнительные фичи, “бантики”

Надежность

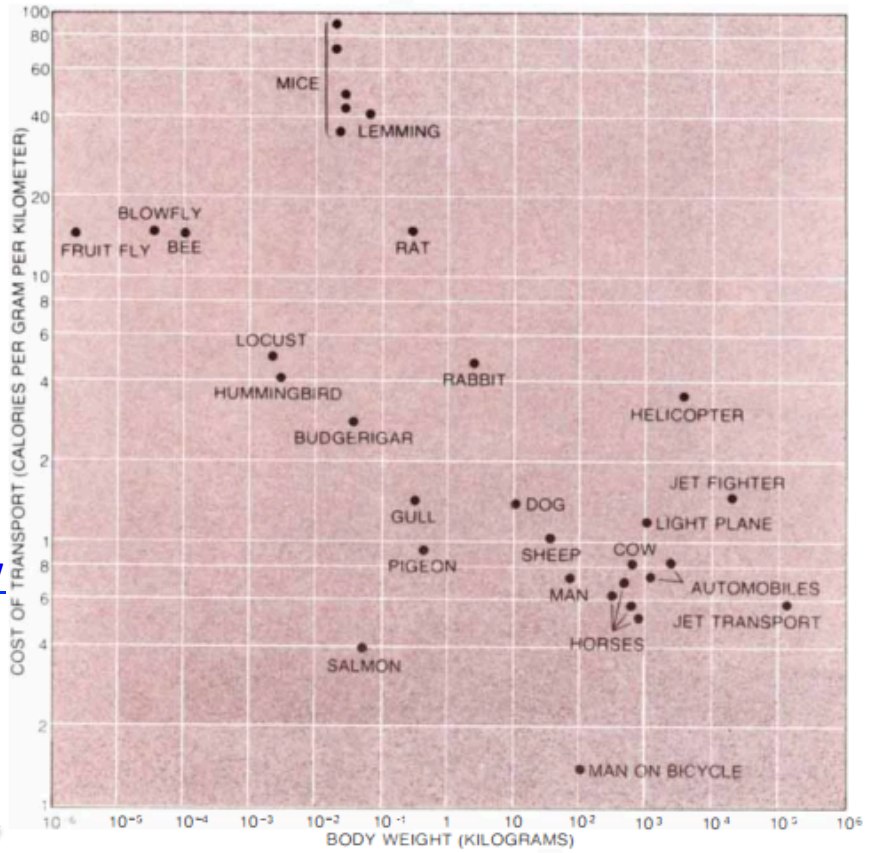
Корректная работа с памятью

- Программа не “падает”
- Отсутствуют утечки памяти и ресурсов
- Отсутствует недетерминизм из-за неинициализированных переменных
- Не зависает (в общем случае невозможно гарантировать)
  - Алгоритмически неразрешимая проблема
  - Бывают псевдо-зависания в случае:
    - алгоритмов с высокой сложностью ($O(N^3)$, $O(e^N)$ …)
    - включения GC …

Два факта из “потерянного интервью” со Стивом Джобсом

https://www.youtube.com/watch?v=lmJzpBH5cRw

1. Программирование – одна из немногих областей индустрии, где разница в продуктивности/качестве может достигать 50 и более раз!
2. Инструменты имеют значение



MAN ON A BICYCLE ranks first in efficiency among traveling animals and machines in terms of energy consumed in moving a certain distance as a function of body weight. The rate of energy consumption for a bicyclist (about .15 calorie per gram per kilometer) is approximately a fifth of that for an unaided walking man (about .75 calorie per gram per kilometer). With the exception of the black point representing the bicyclist (*lower right*), this graph is based on data originally compiled by Vance A. Tucker of Duke University.

Гибридный язык numl как инструмент

- Императивный
- Функциональный
- Объектно-ориентированный
- Модульный
- “Встраиваемый”: спрягаемый с C/C++

Мы не оригинальны в стремлении предложить что-нибудь получше C/C++:

- Julia (http://julialang.org)
- Lua/Torch (http://torch.ch)
- Halide (http://halide-lang.org)
- Data Parallel Haskell (https://wiki.haskell.org/GHC/Data_Parallel_Haskell)

…

Императивные и Функциональные языки

- **Императивные (процедурные) языки:**

  s=s0; for(i=0; i<n; i++) f(&s); // C/C++

  – переменные как именованные ячейки памяти
  – большая роль побочных эффектов
  – явные циклы, явное указание шагов
  – подпрограммы используются для структуризации и для оформления повторно-используемого кода
  – наследники машины Тьюринга

- **Функциональные языки:**

  let rec g(s, i)=if i=0 then s else g(f(s), i-1) in g(s0,n); (* OCaml *)

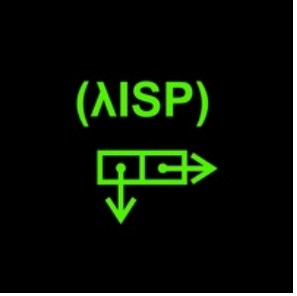
  – значения определяются один раз (как в математике)
  – минимум побочных эффектов, рекурсия вместо циклов (рекуррентные правила)
  – декларативный стиль
  – наследники λ-исчисления

Императивный язык FORTRAN

- FORmula TRANslator, создан в 1957 Джоном Бэкусом (IBM)
- **Встроенные многомерные массивы!**
- Огромное количество численных библиотек, используемых до сих пор
- Допускает очень эффективную компиляцию, на ~20% быстрее C/C++.
- Малоприменим в случае сложных структур данных
- Повлиял на Matlab, который активно используется в исследованиях

```
! Fortran …
subroutine matmul(a, b, c, n)
integer n
real … :: a(:,:), b(:,:), c(:,:)
do j=1,n
 do i=1,n
    tmp = 0.0
    do k=1,n
       tmp = tmp + a(i,k)*b(k,j)
    enddo
    c(i,j) = tmp
 enddo
enddo end subroutine matmul
```

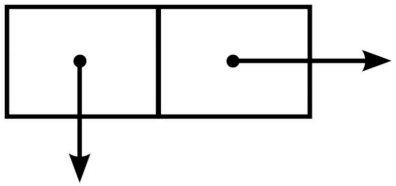
```
// C++
void matmul(std::valarray<double>& a,
…& b, ... &c, int n) {
for(int i=0; i<n; i++)
 for(int j=0; j<n; j++)
 {
    double tmp=0
    for(int k=0; k<n; k++)
   // обращение к valarray
   // медленнее и не проверяет границы
       tmp += a[i*n+k]*b[k*n+j]
    c[i*n+j] = tmp
 }}
```



Функциональный язык LISP

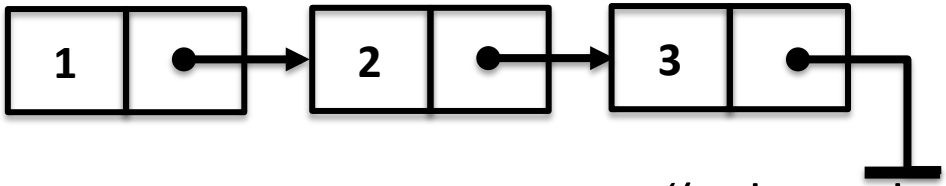
- LISt Processor, создан в 1958 Джоном Маккарти (MIT)
- Первый высокоуровневый язык, описанный на себе самом
- Единообразное описание данных и программ (S-выражения)
- Первый функциональный язык
- Первая реализация сборки мусора
- Первый R-E-P-L язык (read-evaluate-print loop), первый язык с IDE
- Способствовал исследованиям по искусственному интеллекту в MIT
- Многочисленные диалекты, используется до сих пор

```
; Вычисление чисел Фибоначчи
(defun fibonacci (n)
  (if (<= n 1) 1
    ((+ (fibonacci (- n 1)) (fibonacci (- n 2))))))
```

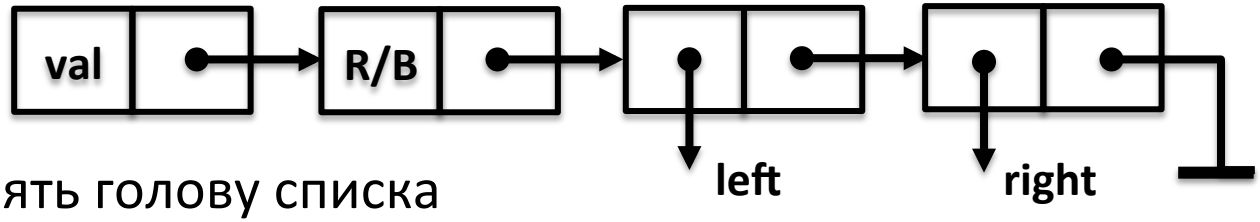


CONS-ячейка

- Пара элементов: **(A . B)** в LISP-нотации
- A/B – либо атомы (числа, символы …), либо указатели на другие CONS-ячейки. NIL – специальный “нулевой” указатель
- ‘(1 2 3) – список == (1 . (2 . (3 . NIL)))



- Сбалансированные деревья: ‘(value color left right)



- (CAR x) – взять голову списка
- (CDR x) – взять хвост списка
- (ATOM x) – является ли значение атомом или CONS-ячейкой
- (CONS x y) – создать пару (x . y): (CONS 1 ‘(2 3)) == ‘(1 2 3)
- (EQ x y) – равны два атома или нет

Классический LISP: +/-

+ Можно представить почти любые структуры данных – кроме структур с произвольным доступом за O(1)
+ Удобно обрабатывать такие структуры с помощью рекурсивных функций
+ Ошибки при работе с памятью невозможны
+ Нет фрагментации. Очень простой двухпроходный алгоритм GC (mark-n-sweep), требует 1 бит на ячейку
+ Поддержка функций лучше чем в C++11 (functions as first-class citizens)
+ **Функциональные структуры данных!**

(insert_rbtree tree element) ; возвращает новое дерево не тронув старого!

- Неэффективен при работе с однородными числовыми данными (массивы были добавлены в LISP со временем)
- Неудобная нотация
- Отсутствие способа описать сложные типы данных и проконтролировать их (частично решено)

Семейство языков ML: кардинальное улучшение LISP

- ML создан в 1978 Робином Милнером (унив. Эдинбурга) => StandardML/SML (стандарты 90, 97); Caml (1985) => OCaml (1996) (INRIA, Париж)
- Инфиксная нотация, меньше скобок
- Статическая типизация с автоматическим выводом типов
- Операция сравнения с образцом
- Полиморфные типы и функции (аналог шаблонов в C++)
- Обработка исключений
- Продвинутая система модулей (очень продвинутая)
- Стандартная библиотека с векторами, вводом-выводом и т.д
- StandardML – первый язык с формально определенной семантикой самого языка и части стандартной библиотеки!

```
(* Вычисление чисел Фибоначчи *)
fun fib(n) = if n<=1 then 1 else fib(n-1)+fib(n-2)
val _= print(Int.toString(fib(10))^"\n")
```

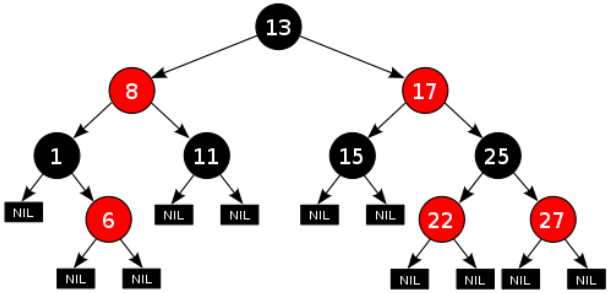
ML: Замена CONS на специализированные структуры (1)

```
(* tuples (кортежи) *)
type complex = real * real
val i = (0.0, 1.0); val re_i = #1(i)
(* записи *)
type rect = { x: int, y: int, width: int, height: int }
val r = {x=1, y=1, width=10, height=20}; val rx = #x(r)
(* списки – внутри те же CONS-ячейки *)
type names = string list
val plangs = [“fortran”, “lisp”, “sml”]
val first_lang = List.hd(plangs)
val plangs1 = “assembler” :: plangs (* :: это аналог операции CONS *)
(* массивы *)
val histogram : int array = [| 0, 0, 0, 0 |]
val _ = update(histogram, 0, sub(histogram, 0) + 1)
```

ML: Замена CONS на специализированные структуры (2)

Как кодировать гетерогенные списки? Как кодировать иерархические структуры?

Ответ: Использовать **типы-суммы** (=суммы типов=алгебраические типы = объединения с тэгом = варианты …)




```
datatype color = Red | Black
datatype rbtree = Empty | RBNode of int*color*rbtree*rbtree
(* создаем дерево из 3 элементов *)
val t3 = RBNode(5, Black, RBNode(1, Red, Empty, Empty),
                          RBNode(10, Red, Empty, Empty))
(* подсчет глубины дерева – используем конструкцию сравнения с образцом *)
fun depth(t) = case t of Empty => 0
                       | RBNode(_, _, left, right) => 1+max(depth(left), depth(right))
```

Типы-суммы настолько же мощны как и S-выражения!

```
datatype atom = INT of int | SYMBOL of string | T
datatype sexpr = NIL | ATOM of atom | CONS of sexpr*sexpr
```

ML: Полиморфизм

Можно ли реализовать структуру данных для хранения экземпляров произвольного типа данных? Легко!

```
datatype color = Red | Black
datatype ‘a rbtree = Empty | RBNode of ‘a*color*rbtree*rbtree
(* создаем дерево из 3 строк, декларация типа значения необязательна *)
val t3 : string rbtree = RBNode(“b”, Black, RBNode(“a”, Red, Empty, Empty), RBNode(“z”, Red, Empty, Empty))
(* подсчет глубины дерева не меняется. декларация типов параметров и результата как и прежде необязательна *)
fun depth(t: ‘a rbtree): int = case t of Empty => 0 | RBNode(_, _, left, right) => 1+max(depth(left), depth(right))
```

ML: +/-

+ **Типобезопасность** при сохранении компактности
+ Потрясающая надежность – защита от почти всех проблем с памятью.
+ Благодаря суммам типов и сравнению с образцом очень удобно описывать и рассматривать различные случаи
+ Компиляторы соперничают с Java, C#, порой C++ по эффективности.
- Неудобно работать с массивами (в OCaml поудобнее), нет многомерных массивов. Операции над массивами императивны
- (это же и **+**) Используют GC
- Нет поддержки параллелизма
- Перегрузка (overloading) отсутствует
- ООП есть в OCaml, нет в SML (частично заменяется модулями)
- Синтаксис мог бы быть получше (больше относится к Ocaml)
- Трудно использовать в существующих проектах на C/C++

numl ≈ SML + MATLAB + …

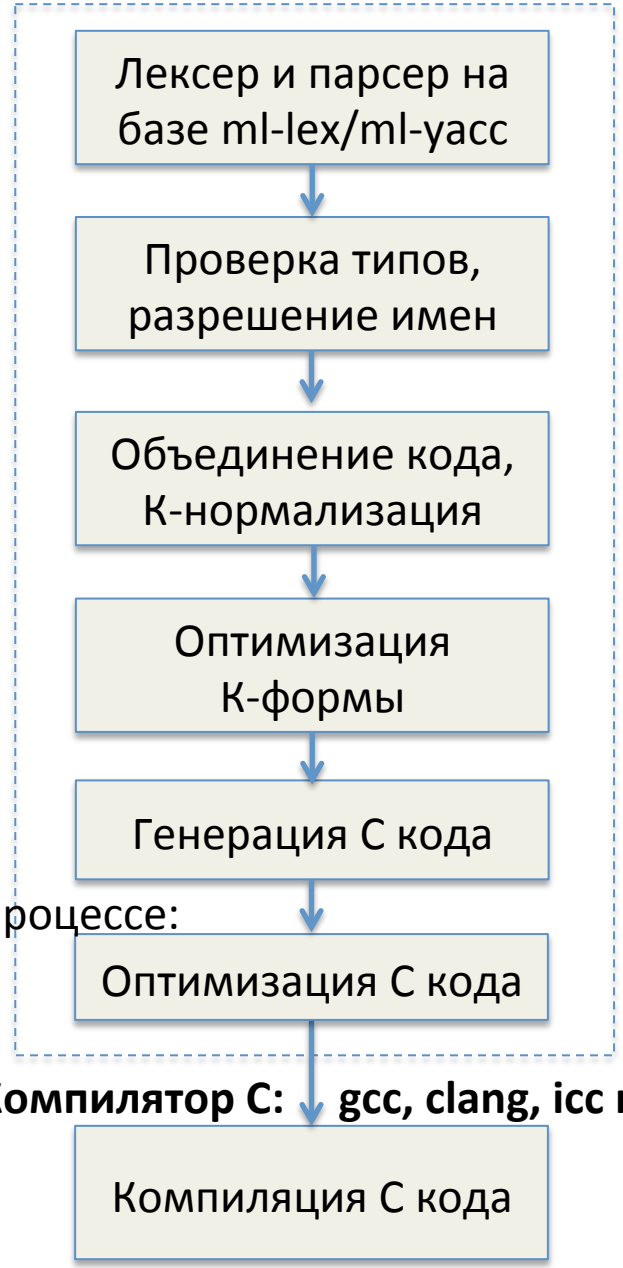
- Активно разрабатывается с конца 2014.
- Лицензия Apache 2 (например как у Android), скоро будет открыт исходный код
- ~30000 строк на StandardML (в процессе переписывания на numl)
- Знает про массивы, умеет с ними работать
- CONS-ячейки, типы-суммы, операция сравнения с образцом, исключения, полиморфные типы/функции и т.д. также есть

```
// Вычисление чисел Фибоначчи
fun fib(n: int): int = if n<=1 then 1 else fib(n-1)+fib(n-2) fi
println("fib(10)=\(fib(10))")

// Умножение матриц – использование "array comprehensions"
fun matmul(a: 't [,], b: 't [,]): 't [,] =
{
   val (ma, na) = size(a)
   val (mb, nb) = size(b)
   val z = a[0,0]-a[0,0]
   // хотите многопоточность? Добавьте parallel перед for!
   [for i in 0:ma for j in 0:nb do
       for k in 0:na update s=z with s+a[i,k]*b[k,j] end
    end]
}
// Вызов функций на C/C++
nothrow fun print(x: int): void = ccode "printf(\"%d\", x);"
```

Структура компилятора

numlc: *.nl => .c, .h



Реализованные и планируемые оптимизации

Optimization	Status
Global optimization	
Alias elimination (beta reduction)	
Dead code elimination	
Flatenning nested expressions	
Constant folding	
Inline expansion	
Loop invariants	
Common subexpressions	
Efficient scalar operations	
Loop fusion	
Efficient iteration over arrays	
Tail recursion	
Reduced reference counting	
Efficient memory allocation	
Array index range check optimization	

Многопроходная оптимизация: пример

```
// 1. Оригинальный код:
// Base.nl
val debug = false // вставляется автоматически; при указании опции –debug получаем “val debug=true”
fun dbg_assert(e: bool): void = if debug && !e then throw AssertionError fi

// Пользовательский код
dbg_assert(x > 0)
```

```
// 2. После beta-редукции (подстановки синонимов)
val debug = false
fun dbg_assert(e: bool): void = if false && !e then throw AssertionError fi

// Пользовательский код
dbg_assert(x > 0)
```

```
// 3. После constant folding
val debug = false
fun dbg_assert(e: bool): void = {}

// Пользовательский код
dbg_assert(x > 0)
```

```
// 4. После устранения мертвого кода, а также кода не имеющего эффекта
```

Производительность

Реализовано 6 бенчмарок отсюда:
http://benchmarksgame.alioth.debian.org

The Computer Language Benchmarks Game

Бенчмарка	numl (сек)	C (сек)	Javascript (V8)	Отношение C/numl (больше=лучше)
nbody	6.5	4.7 (uses SSE intrinsics)	39.8	0.72
btree	15	23	21	1.53
spectralnorm	2.72	2.71	7.8	0.99
mandelbrot (parallel)	2.7	2.1 (uses SSE intrinsics)	165	0.78
pigidits	2.33 (uses GMP)	2.30 (uses GMP)	not implemented	0.98
K-nucleotide	44	17	207	0.38

* Test machine: 4-core Core-i5, Ubuntu 14.04 x64, GCC 4.8, fresh Google Chrome

numl существенно быстрее Python, Lua, Javascript,
как минимум равен Java и приближается по скорости к C!

Бенчмарка Mandelbrot

```
import File

val w = 16000
val h = 16000
val MAX_ITER = 50

val inv = 2.0 / w

type vec8d = (double, double, double, double,
              double, double, double, double)
fun (+) (a: vec8d, b: vec8d): vec8d =
 (a.0+b.0, a.1+b.1, a.2+b.2, a.3+b.3, a.4+b.4, a.5+b.5, a.6+b.6, a.7+b.7)
fun (-) (a: vec8d, b: vec8d): vec8d =
 (a.0-b.0, a.1-b.1, a.2-b.2, a.3-b.3, a.4-b.4, a.5-b.5, a.6-b.6, a.7-b.7)
fun (*) (a: vec8d, b: vec8d): vec8d =
 (a.0*b.0, a.1*b.1, a.2*b.2, a.3*b.3, a.4*b.4, a.5*b.5, a.6*b.6, a.7*b.7)

val x_ = [parallel for x in 0:w do (x :> double) * inv - 1.5 end]
val result: int8 [,] = [
  parallel
  for y in 0:h do for x8 in 0:(w/8) do
    val y_: double = (y :> double) * inv - 1.0
    val x = x8*8
    val cr: vec8d = (x_[x + 0], x_[x + 1], x_[x + 2], x_[x + 3],
                     x_[x + 4], x_[x + 5], x_[x + 6], x_[x + 7])
    val ci: vec8d = (y_, y_, y_, y_, y_, y_, y_, y_)

    var bits = 255
    var zr = cr
    var zi = ci
    var iter = 0
    while iter < MAX_ITER && bits != 0 do
      val rr = zr * zr
      val ii = zi * zi

      val mag = rr + ii
      if mag.0 > 4.0 then bits &= ~128 fi
      if mag.1 > 4.0 then bits &= ~64 fi
      if mag.2 > 4.0 then bits &= ~32 fi
      if mag.3 > 4.0 then bits &= ~16 fi
      if mag.4 > 4.0 then bits &= ~8 fi
      if mag.5 > 4.0 then bits &= ~4 fi
      if mag.6 > 4.0 then bits &= ~2 fi
      if mag.7 > 4.0 then bits &= ~1 fi
      val ir = zr * zi
      zr = (rr - ii) + cr
      zi = (ir + ir) + ci
      iter += 1
    end
    (bits :> int8)
  end
]

fun write_file(): void =
{
  val f = File.open("result.pgm", "wb")
  File.print(f, "P4\n\(w) \(h)\n")
  File.write(f, result) // RAW binary output
  File.close(f)
}
write_file()
0
```

Резюме. Планы

- Языки программирования нового поколения могут сильно помочь в компьютерном зрении и смежных областях. Активно разрабатываются
- Идеальный для нас язык: массивы + параллелизм + функциональное программирование
- Разрабатываемый язык numl удовлетворяет этим требованиям
- **Он безопасен: сборка мусора, доступ к памяти контролируется, все значения инициализируются явно!**
- Стоим на плечах гигантов: Fortran, Matlab, LISP, SML, Python, Java, C/C++.
- Не стесняясь передаем код дальше по цепочке C компиляторам
- Супер-портабельный!

– Пока отсутствует интерактивный режим
– Получающиеся .c файлы большие и долго компилируются. Пока отсутствует раздельная компиляция
– Пока только для CPU
– Библиотека в разработке, компилятор пока не очень удобен, дополнительный инструментарий (отладчик, профилировщик …) отсутствует

– Пока не представлен официально: следите за http://opencv.org !

Литература

- Хювёнен Э., Сеппянен Й. Мир Лиспа. В 2-х т. / Пер. с финск. Можно найти в сети. Образец того какими должны быть книжки по программированию

или

- Абельсон Х., Сассман Д.Д. Структура и интерпретация компьютерных программ. Сокращенно SICP, классический учебник для студентов MIT, использующий диалект Lisp'a Scheme

- Электронный журнал "Практика функционального программирования", более не издается. http://fprog.ru.

  Есть очень интересные статьи, например

  "История разбработки одного компилятора"

  http://fprog.ru/2009/issue2/practice-fp-2-ebook.pdf
- Guy Cousineau, Michel Mauny. The Functional Approach To Programming. Можно найти в сети. Еще один шедевр! После этой книги стало понятно каким должен быть numl
- min-caml. http://esumii.github.io/min-caml/index-e.html. Сказ про то как один японец уместил оптимизирующий компилятор небольшого функционального языка в машинный код Sun Sparc в 2000 строк на OCaml. **min-caml – прародитель numl**. Там есть и слайды, и статья, и исходный код