CS 410/510

Languages & Low-Level Programming

Mark P Jones
Portland State University

Fall 2018

Week 9: Language Design

Copyright Notice

Questions for today

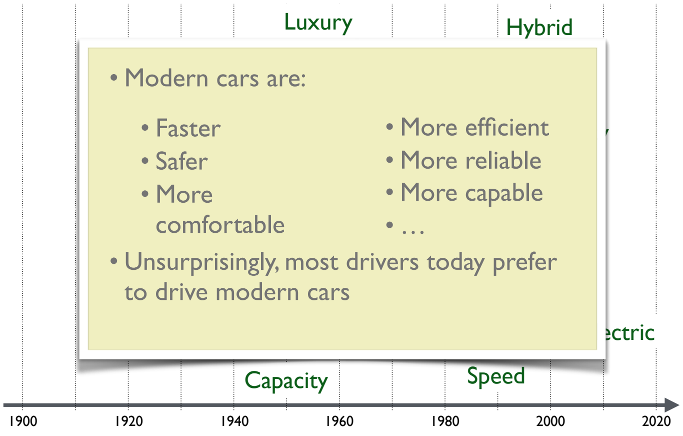
- Why do we have so many languages?
- How can we evaluate or compare language designs?
- What criteria should we use in selecting a language for a task?
- How can we approach the design of new languages?
- How can design languages to suit specific domains?
  - example domain: low-level, bare metal development?

- There are many answers to these questions!

Why so many languages?

- Diversity
  - Different purposes / domains
  - Different paradigms / ways to think about programming
  - Different judgements about language goals and aesthetics
  - Different platforms and environments
- Evolution
  - Improve on existing languages by adding/removing features
  - New languages provide a clean slate
  - Prototype new features and explore their interactions

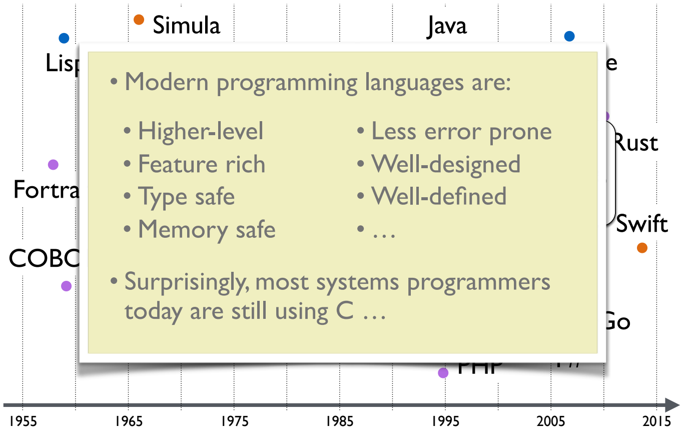
A short history of the automobile



- Modern cars are:
  - Faster
  - Safer
  - More comfortable
  - More efficient
  - More reliable
  - More capable
  - …
- Unsurprisingly, most drivers today prefer to drive modern cars

(Images via Wikipedia, subject to Creative Commons and Public Domain licenses)

A short history of programming languages

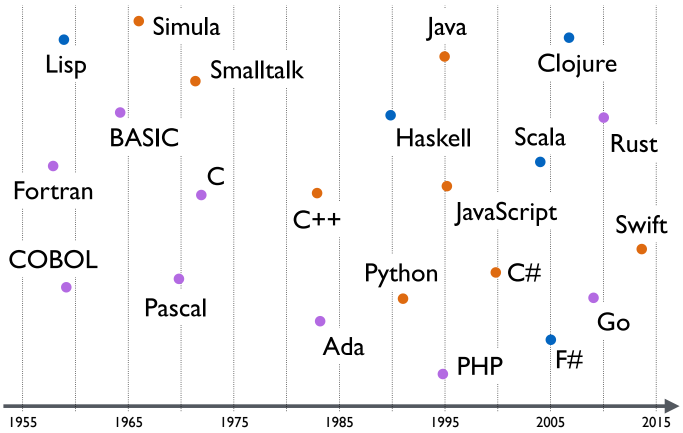


- Modern programming languages are:
  - Higher-level
  - Feature rich
  - Type safe
  - Memory safe
  - Less error prone
  - Well-designed
  - Well-defined
  - …
- Surprisingly, most systems programmers today are still using C …

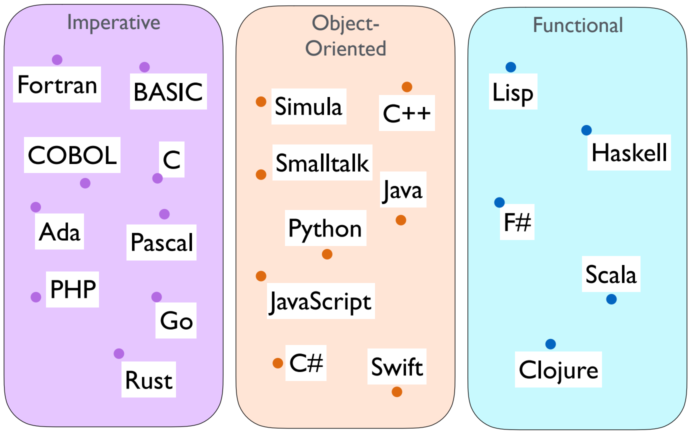
Classifying programming languages

- One way to understand a collection of items is to classify them in ways that exhibit their similarities and their differences
- How might we classify programming languages?
  - By paradigm
  - By expressivity
  - By contrasting domain specific vs general purpose
  - By contrasting high level vs low level
  - …
- In practice, classifications often require subjective judgement …

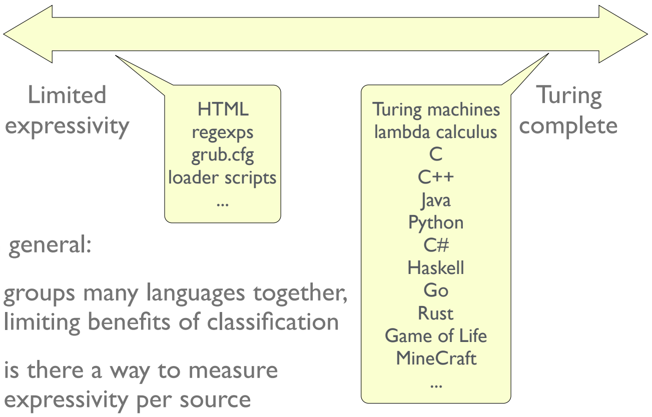
A short history of programming languages



Classification by "Paradigm"



Expressivity



Limited expressivity

HTML
regexps
grub.cfg
loader scripts
…

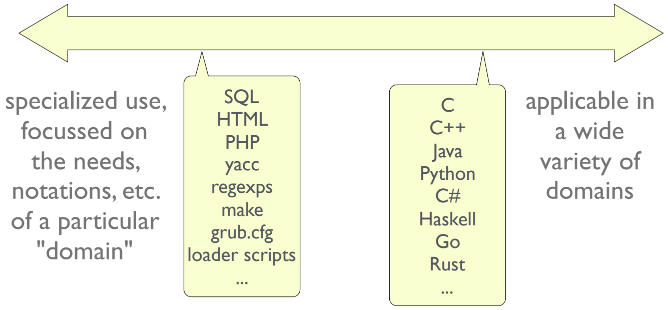
Turing machines
lambda calculus
C
C++
Java
Python
C#
Haskell
Go
Rust
Game of Life
MineCraft
…

Turing complete

In general:

- groups many languages together, limiting benefits of classification
- is there a way to measure expressivity per source character?

Domain Specific vs General Purpose



specialized use, focussed on the needs, notations, etc. of a particular "domain"

SQL
HTML
PHP
yacc
regexps
make
grub.cfg
loader scripts
…

C
C++
Java
Python
C#
Haskell
Go
Rust
…

applicable in a wide variety of domains

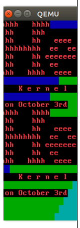
In general:

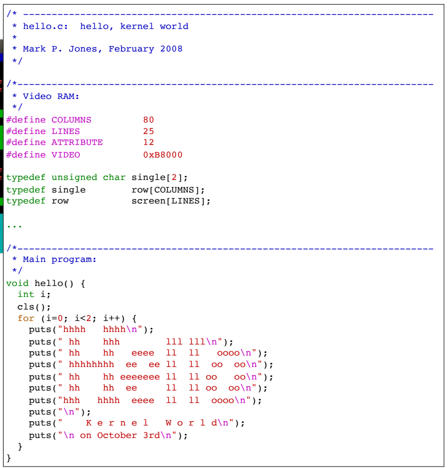
- it's a "design space", not a linear "spectrum"
- determining where any specific language fits is subjective

Domain specific languages in the hello demo

hello.c

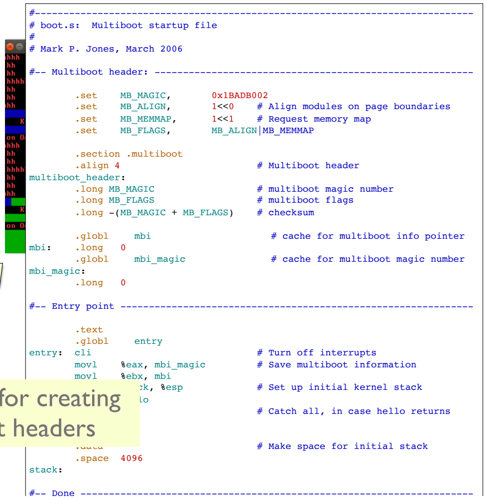
A language for specifying application functionality





```
/* ------------------------------------------------------------------------
 * hello.c:  hello, kernel world
 *
 * Mark P. Jones, February 2008
 */

/*-------------------------------------------------------------------------
 * Video RAM:
 */
#define COLUMNS         80
#define LINES           25
#define ATTRIBUTE       12
#define VIDEO           0xB8000

typedef unsigned char single[2];
typedef single        row[COLUMNS];
typedef row           screen[LINES];

...

/*-------------------------------------------------------------------------
 * Main program:
 */
void hello() {
  int i;
  cls();
  for (i=0; i<2; i++) {
    puts("hhhh   hhhh\n");
    puts(" hh    hhh        lll lll\n");
    puts(" hh    hh   eeee  ll  ll   oooo\n");
    puts(" hhhhhhhh  ee  ee ll  ll  oo  oo\n");
    puts(" hh    hh eeeeeee ll  ll oo   oo\n");
    puts(" hh    hh  ee     ll  ll oo  oo\n");
    puts("hhh   hhhh  eeee  ll  ll  oooo\n");
    puts("\n");
    puts("    K e r n e l   W o r l d\n");
    puts("\n on October 3rd\n");
  }
}
```

Domain specific languages in the hello demo



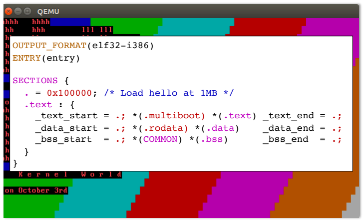
```
#-------------------------------------------------------------------------
# boot.s:  Multiboot startup file
#
# Mark P. Jones, March 2006

#-- Multiboot header: ----------------------------------------------------

        .set    MB_MAGIC,       0x1BADB002
        .set    MB_ALIGN,       1<<0    # Align modules on page boundaries
        .set    MB_MEMMAP,      1<<1    # Request memory map
        .set    MB_FLAGS,       MB_ALIGN|MB_MEMMAP

        .section .multiboot
        .align 4                        # Multiboot header
multiboot_header:
        .long MB_MAGIC                  # multiboot magic number
        .long MB_FLAGS                  # multiboot flags
        .long -(MB_MAGIC + MB_FLAGS)    # checksum

        .globl    mbi                     # cache for multiboot info pointer
mbi:    .long   0
        .globl    mbi_magic               # cache for multiboot magic number
mbi_magic:
        .long   0

#-- Entry point ----------------------------------------------------------

        .text
        .globl    entry
entry:  cli                             # Turn off interrupts
        movl    %eax, mbi_magic         # Save multiboot information
        movl    %ebx, mbi
                  ck, %esp              # Set up initial kernel stack
                   lo
                                        # Catch all, in case hello returns

                                        # Make space for initial stack
        .space  4096
stack:

#-- Done -----------------------------------------------------------------
```

hello.c

boot.s

A language for creating multiboot headers

Domain specific languages in the hello demo



```
OUTPUT_FORMAT(elf32-i386)
ENTRY(entry)

SECTIONS {
  . = 0x100000; /* Load hello at 1MB */
  .text : {
    _text_start = .; *(.multiboot) *(.text) _text_end = .;
    _data_start = .; *(.rodata) *(.data)    _data_end = .;
    _bss_start  = .; *(COMMON) *(.bss)      _bss_end  = .;
  }
}
```

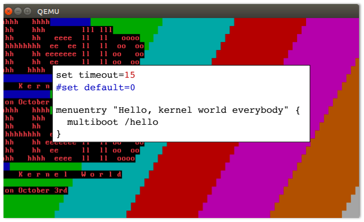
hello.c

boot.s

hello.ld

A language for describing executable file layout

Domain specific languages in the hello demo



```
set timeout=15
#set default=0

menuentry "Hello, kernel world everybody" {
  multiboot /hello
}
```

hello.c

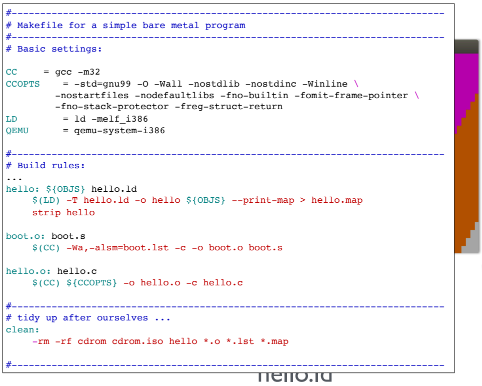
boot.s

hello.ld

grub.cfg

A language for configuring the boot process

Domain specific languages in the hello demo



```
#-------------------------------------------------------------------------
# Makefile for a simple bare metal program
#-------------------------------------------------------------------------
# Basic settings:

CC       = gcc -m32
CCOPTS     = -std=gnu99 -O -Wall -nostdlib -nostdinc -Winline \
          -nostartfiles -nodefaultlibs -fno-builtin -fomit-frame-pointer \
          -fno-stack-protector -freg-struct-return
LD         = ld -melf_i386
QEMU       = qemu-system-i386

#-------------------------------------------------------------------------
# Build rules:
...
hello: ${OBJS} hello.ld
     $(LD) -T hello.ld -o hello ${OBJS} --print-map > hello.map
     strip hello

boot.o: boot.s
     $(CC) -Wa,-alsm=boot.lst -c -o boot.o boot.s

hello.o: hello.c
     $(CC) ${CCOPTS} -o hello.o -c hello.c

#-------------------------------------------------------------------------
# tidy up after ourselves ...
clean:
     -rm -rf cdrom cdrom.iso hello *.o *.lst *.map

#-------------------------------------------------------------------------
```

hell

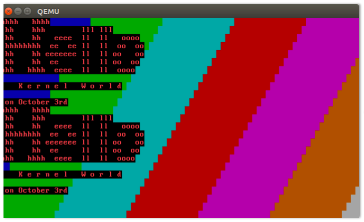
Makefile

A language for describing the build process

grub.cfg

hello.ld

Domain specific languages in the hello demo



hello.c

Makefile

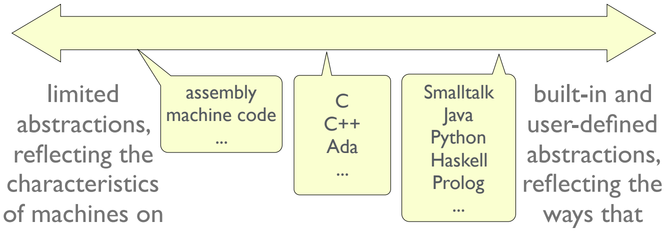
boot.s

hello.ld

grub.cfg

Five "domain specific languages" (DSLs), each serving a different role, but working together to describe a complete program

Low Level vs High Level



limited abstractions, reflecting the characteristics of machines on which programs are executed

assembly
machine code
...

C
C++
Ada
...

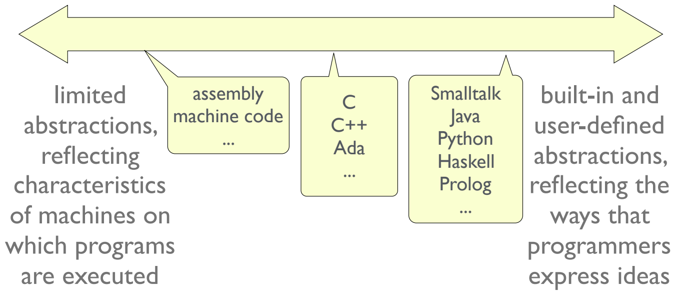
Smalltalk
Java
Python
Haskell
Prolog
...

built-in and user-defined abstractions, reflecting the ways that programmers express ideas

Again, in general:

- it's a "design space", not a linear "spectrum"
- determining where any specific language fits is subjective

Low Level vs High Level



- Historically, low-level development has tended to focus on the use of lower-level languages. Why is this?
- How can we expect to write bare metal programs using languages that intentionally abstract away from hardware?

Invalid classifications

- Confusing languages with implementations:
  - Compiled vs Interpreted
  - Fast vs Slow
- These are properties of implementations, not languages!
- Inherently subjective classifications:
  - Readability
  - Familiarity
  - Ease of use
- These are judgements that individual programmers make based on their experience, background, and preferences ...

Choosing an existing language

- Factors that might influence the choice of a particular language for a given project include:
  - Availability of implementation for target environment
  - Availability of trained programmers
  - Availability of documentation
  - Availability of tools (IDEs, debuggers, ...)
  - Availability of libraries
  - Developer / customer / platform requirements
  - Familiarity / experience
  - ...

Designing a new language - Why?

- Why design a new language?
  - Explore ideas without concern for backward compatibility
  - Address a need that is not met by current designs
  - Learn general principles about programming languages
  - Have some fun!

Designing a new language - How?

- How to design a new language?
  - Identify a need / shortcoming with existing languages
  - Start from a clean slate (uncommon)
  - Improve / borrow from existing languages
  - Write out a language definition
  - Evaluate the design:
    - Write programs
    - Develop tools (compilers, interpreters, etc...)
    - Formalize and prove properties
    - ...
  - Refine, revise, repeat!

Language design is **not** (yet) a precise science!

A language for low-level programming

- We've spent the past eight weeks studying bare-metal development and microkernel design and implementation
- How might we design a language for this domain?
- Is a new language even necessary?
- If so:
  - What features should the language provide?
  - How should we evaluate the new design?

C is great … what more could you want?

- Programming in C gives systems developers:
  - Good (usually predictable) performance characteristics
  - Low-level access to hardware when needed
  - A familiar and well-established notation for writing imperative programs that will get the job done
- What can you do in modern languages that you can't already do with C?
- Do you really need the fancy features of newer object-oriented or functional languages?
- Are there any downsides to programming in C?

How could a different language help? (1)

- Increase programmer productivity (reduce development time)
  - Reduce boilerplate (duplicating code is error prone and increases maintenance costs)
  - Reduce cross cutting concerns (when the implementation of a single feature is "tangled" with the implementations of other features and spread across the source code, making the code harder to read and harder to maintain)
  - …

Example: bitdata types

base$_{22}$	size$_6$	~	r	w	x

From L4Ka::Pistachio, a mature L4 implementation in C++ from the University of Karlsruhe, Germany

```
class mempage_t {
public:
    union {
        struct {
            BITFIELD7(word_t,
                     execute              : 1,
                     write                : 1,
                     read                 : 1,
                     reserved             : 1,
                     size                 : 6,
                     base                 : L4_FPAGE_BASE_BITS,
                     : BITS_WORD - L4_FPAGE_BASE_BITS - 10
                );
        } x __attribute__((packed));
        word_t raw;
    };
};
```

BITFIELD macro adjusts for variations between C/C++compilers …

Permission values inlined

macros for sizes

gcc specific attribute: "a variable or structure field should have the smallest possible alignment"

Example: bitdata types

base$_{22}$	size$_6$	~	r	w	x

From pork, implemented in C (no reliance on non standard features)

```
typedef unsigned Perms;

#define R  (4)
#define W  (2)
#define X  (1)

typedef unsigned Fpage;

static inline Fpage fpage(unsigned base, unsigned size, Perms perms) {
  return alignTo(base, size) | (size<<4) | perms;
}

static inline unsigned fpageMask(Fpage fp) {
  return fpmask[(fp>>4)&0x3f];
}

static inline unsigned fpageStart(Fpage fp) {
  return fp & ~fpageMask(fp);
}

static inline unsigned fpageEnd(Fpage fp) {
  return fp | fpageMask(fp);
}
```

Constants for individual permission bits

`Fpage` is a synonym for `unsigned`, which could prevent type errors from being detected

"constructor"

Bit-level layout is hard-coded via shift and mask constants in functions that are expected to be inlined for fast execution

"selectors"

Example: bitdata types

base$_{22}$	size$_6$	~	r	w	x

```
block fpage {
    field_high  base_address    20
    field       size            6
    padding                     1
    field       read            1
    field       write           1
    field       exec            1
}
```

The designers of seL4 use a *lot* of types like this ... so they created a "bitfields" DSL for describing bitdata types

A parser, written in Python, reads .bf files and generates C code for manipulating data structures (also used for verification work)

Still translates to C code that doesn't distinguish between the types of different fields

Why write boring code, when you can write a more interesting program to write it for you?

Example: bitdata types

base$_{22}$	size$_6$	~	r	w	x

Using Habit, a functional language for low-level systems programming

```
bitdata Perms = Perms [ r, w, x :: Bool ]

bitdata Fpage = Fpage [ base :: Bit 22 | size :: Bit 6
                            | reserved :: Bit 1 | perms :: Perms ]
```

Bit-level data layout

Mimics familiar box notation for bitdata types

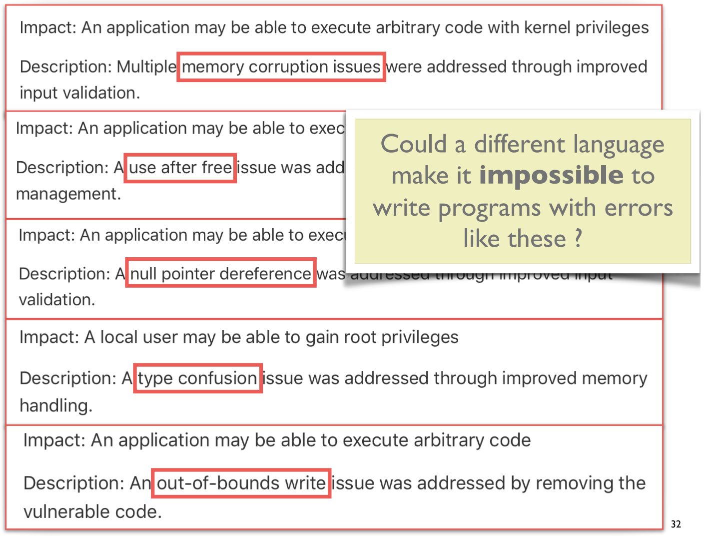
Rich type system: `Bit 22`, `Bit 6`, `Bit 1`, `Perms`, and `Page` are *distinct* types.

Mixing these incorrectly will trigger a compile time error!

Relying on language support complicates the compiler … but simplifies the application code …

How could a different language help? (2)

- Improve software quality (eliminate avoidable bugs)
  - Type confusion ... for example:
    - confusing physical and virtual addresses
    - confusing boolean and unsigned: `(v & 0x81 == 0x1)` gives the wrong result because of precedence, but could have been avoided by checking types
  - Unchecked runtime exceptions (divide by zero, null pointer dereference, out of bounds array access, ...)
    - using `(v & 0x3fff)` to calculate a 10 bit index for a page table ... will actually produce a 14 bit value ...
  - Memory bugs (e.g., use after free, space leak, ...)



Impact: An application may be able to execute arbitrary code with kernel privileges

Description: Multiple memory corruption issues were addressed through improved input validation.

Impact: An application may be able to exec

Description: A use after free issue was add management.

Impact: An application may be able to exec

Description: A null pointer dereference was addressed through improved input validation.

Impact: A local user may be able to gain root privileges

Description: A type confusion issue was addressed through improved memory handling.

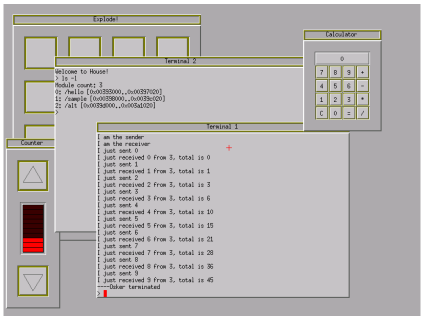
Impact: An application may be able to execute arbitrary code

Description: An out-of-bounds write issue was addressed by removing the vulnerable code.

Could a different language make it **impossible** to write programs with errors like these ?

House (2005)

Kernel, GUI, drivers, network stack, and apps



```
Welcome to House!
> ls -l
Module count: 3
0: /hello [0x00393000..0x00397020]
1: /sample [0x00398000..0x0039c020]
2: /alt [0x0039d000..0x003a1020]
>
```

```
I am the sender
I am the receiver
I just sent 0
I just received 0 from 3, total is 0
I just sent 1
I just received 1 from 3, total is 1
I just sent 2
I just received 2 from 3, total is 3
I just sent 3
I just received 3 from 3, total is 6
I just sent 4
I just received 4 from 3, total is 10
I just sent 5
I just received 5 from 3, total is 15
I just sent 6
I just received 6 from 3, total is 21
I just sent 7
I just received 7 from 3, total is 28
I just sent 8
I just received 8 from 3, total is 36
I just sent 9
I just received 9 from 3, total is 45
----Osker terminated
>
```

Boots and runs in a bare metal environment

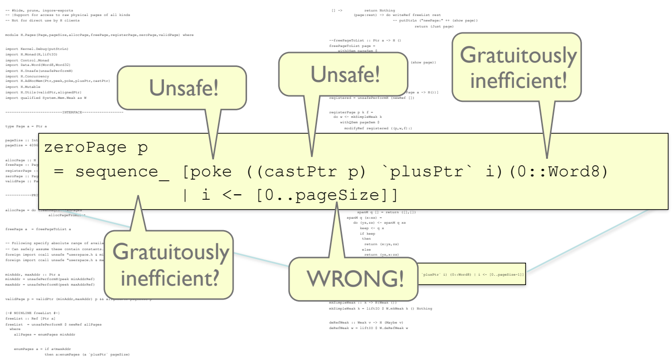
… all written in Haskell, a "purely functional" programming language that is known for:

- type safety
- memory safety
- high-level abstractions

Managing pages in House

```
-- |Support for access to raw physical pages of all kinds

module H.Pages(Page,pageSize,allocPage,freePage,registerPage,zeroPage,validPage)
    where

import Kernel.Debug(putStrLn)
import H.Monad(H,liftIO)
import Control.Monad
import Data.Word(Word8,Word32)
import H.Unsafe(unsafePerformH)
import H.Concurrency
import H.AdHocMem(Ptr,peek,poke,plusPtr,castPtr)
import H.Mutable
import H.Utils(validPtr,alignedPtr)
import qualified System.Mem.Weak as W

--------------------------INTERFACE-------------------

type Page a = Ptr a

pageSize :: Int
pageSize = 4096 -- bytes

allocPage :: H (Maybe (Page a))
freePage :: Page a -> H ()        -- caller must ensure arg is valid
registerPage :: Page a -> b -> (Page a -> H()) -> H ()
zeroPage :: Page a -> H()
validPage :: Page a -> Bool

------------PRIVATE IMPLEMENTATION FOLLOWS---------------
```

Managing pages in House



```
zeroPage p
  = sequence_ [poke ((castPtr p) `plusPtr` i)(0::Word8)
                | i <- [0..pageSize]]
```

It all type checks, it all runs, it must be right ...

A summary of the "House Experience"

- Many positives ...
- But also some serious negatives:
  - Large, untrusted runtime system
  - Reliance on unsafe operations for essential low-level primitives
  - Weak type system
  - Resource management issues
  - Performance concerns
- Can we keep the positives but eliminate the negatives?

The Habit programming language

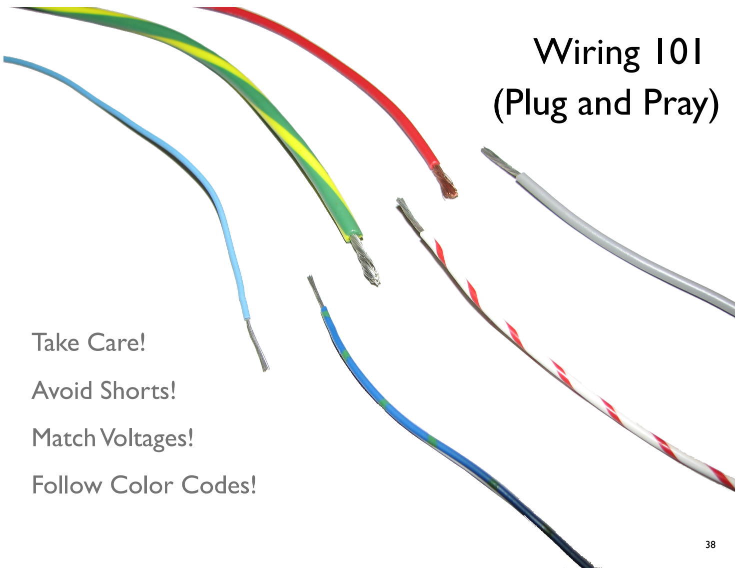
- “a dialect of Haskell that is designed to meet the needs of high assurance systems programming”

Habit = **Ha**skell + **bit**s

- Habit, like Haskell, is a functional programming language
- For people trained in using C, the syntax and features of Habit may be unfamiliar
- I won’t assume much familiarity with functional programming
- We'll use Habit as an example to show how **types** can detect and **prevent** common types of programming errors

---

Wiring 101
(Plug and Pray)



Take Care!

Avoid Shorts!

Match Voltages!

Follow Color Codes!

Plug and Play

Simple, fast connections

Enforce correct usage

Guarantee safety

Higher-level interfaces

Can we emulate this strategy in software, ensuring correct usage and preventing common types of bugs *by construction?*

Division

- You can divide an integer by an integer to get an integer result
- In Habit:

“has type” 1st arg 2nd arg result

```
div :: Int ⟶ Int ⟶ Int
```

- This is a lie!
- **Correction**: You can divide an integer by a **non-zero integer** to get an integer result
- In Habit:

```
div :: Int ⟶ NonZero Int ⟶ Int
```

- But where do `NonZero Int` values come from?

Where do `NonZero` values come from?

- **Option 1**: Integer literals - numbers like `1`, `7`, `63`, and `128` are clearly all `NonZero` integers
- **Option 2**: By checking at runtime

```
nonzero :: Int ⟶ Maybe (NonZero Int)
```

Values of type `Maybe t` are either:
- `Nothing`
- `Just x` for some `x` of type `t`

- These are the only two ways to get a `NonZero Int`!
- `NonZero` is an **abstract datatype**

Examples using `NonZero` values

- Calculating the average of two values:

```
ave     :: Int ⟶ Int ⟶ Int
ave n m = (n + m) `div` 2
```

a non zero literal

- Calculating the average of a list of integers:

```
average :: List Int ⟶ Maybe Int
average nums
  = case nonzero (length nums) of
      Just d  ⟶  Just (sum nums `div` d)
      Nothing ⟶  Nothing
```

checked!

- Key point: If you forget the check, your code will not compile!

Null pointer dereferences

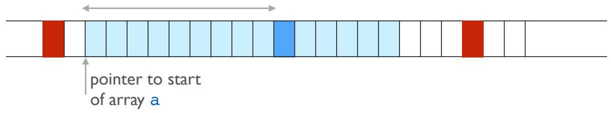
- In C, a value of type `T*` is a pointer to an object of type `T`
- But this may be a lie!
- A `null` pointer has type `T*`, but does NOT point to an object of type `T`
- Attempting to read or write the value pointed to by a `null` pointer is called a "**null pointer dereference**" and often results in system crashes, vulnerabilities, or memory corruption
- Described by Tony Hoare (who introduced null pointers in the ALGOL W language in 1965) as his "billion dollar mistake"

Pointers and reference types

- Lesson learned: We should distinguish between:
  - References (of type `Ref t`) that are guaranteed to point to values of type `t`
  - Physical addresses (of type `PhysAddr t`)
  - Pointers (of type `Ptr t`): either a reference <u>or</u> a `null`
- C groups all these types together as `t*`
- In Habit, they are distinct: `Ptr t = Maybe (Ref t)`
- You can only read or write values via a reference
- Code that tries to read via a pointer will fail to compile!
- Goodbye null pointer dereferences!
- Goodbye physical/virtual address confusion!

Arrays and out of bounds indexes

- Arrays are collections of values stored in contiguous locations in memory



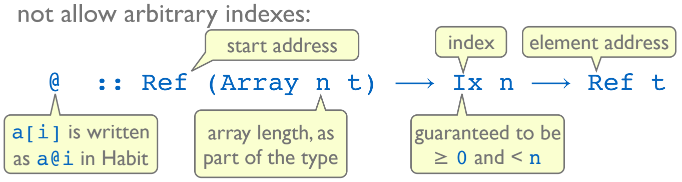
- Address of `a[i]` = start address of `a` + `i`*(size of element)
- Simple, fast, … and dangerous!
- If `i` is not a valid index (an "out of bounds index"), then an attempt to access `a[i]` could lead to a system crash, memory corruption, buffer overflows, …
- A common path to "arbitrary code execution"

Array bounds checking

- The designers of C knew that this was a potential problem … but chose not to address it in the language design:
  - We would need to store a length field in every array
  - We would need to check for valid indexes at runtime
  - This would slow program execution
- The designers of Java knew that this was a potential problem … and chose to address it in the language design:
  - Store a length field in every array
  - Check for valid indexes at runtime
- Performance <u>**OR**</u> Safety … pick **one**! ☹

Arrays in Habit

- Key idea: make array size part of the array and index type, do not allow arbitrary indexes:



start address

index

element address

```
@   :: Ref (Array n t) ⟶ Ix n ⟶ Ref t
```

`a[i]` is written as `a@i` in Habit

array length, as part of the type

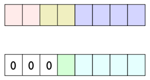
guaranteed to be ≥ `0` and < `n`

- Fast, no need for a runtime check, no need for a stored length
- `Ix n` is another abstract type: 😀

```
maybeIx :: Int ⟶ Maybe (Ix n)
modIx   :: Int ⟶ Ix n
incIx   :: Ix n ⟶ Maybe (Ix n)
```

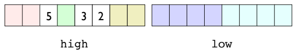
Bit twiddling

- Given two 32 bit input values:
  - base:
  - limit:



(Each box is one nibble (4 bits), least significant bits on the right)

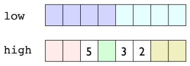
- Calculate a 64 bit descriptor:

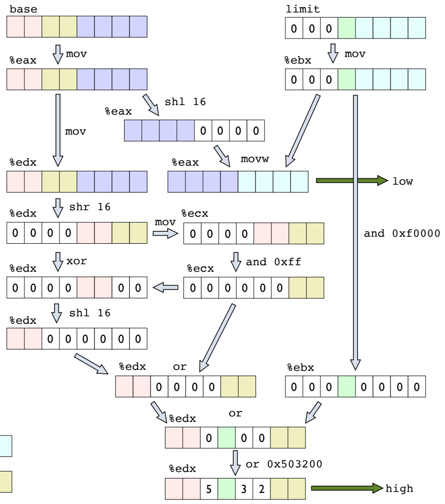


- Needed for the calculation of IA32 Global Descriptor Table (GDT) entries

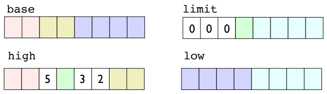
In assembly

```
movl    base, %eax
movl    limit, %ebx

mov     %eax, %edx
shl     $16, %eax
mov     %bx, %ax
movl    %eax, low

shr     $16, %edx
mov     %edx, %ecx
andl    $0xff, %ecx
xorl    %ecx, %edx
shl     $16,%edx
orl     %ecx, %edx
andl    $0xf0000, %ebx
orl     %ebx, %edx
orl     $0x503200, %edx
movl    %edx, high
```





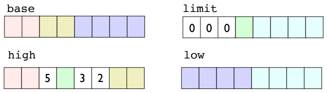
In C



```
low  = (base << 16)          // purple
     | (limit & 0xffff);     // blue
high = (base & 0xff000000)   // pink
     | (limit & 0xf0000)     // green
     | ((base >> 16) & 0xff) // yellow
     | 0x503200;             // white
```

- Examples like this show why we use high-level languages instead of assembly!
- But let's hope we don't get those offsets and masks wrong …
- Because there is not much of a safety net if we mess up …

In Habit



- Programmer describes layout in a type definition:

```
bitdata GDT
  = GDT [ pink   :: Bit 8 | 0x5  :: Bit 4
        | green  :: Bit 4 | 0x32 :: Bit 8
        | yellow :: Bit 8 | purple, blue :: Bit 16 ]
```

- Compiler tracks types and automatically figures out appropriate offsets and masks:

```
makeGDT :: Unsigned ⟶ Unsigned ⟶ GDT
makeGDT (pink # yellow # purple)  -- base
        (0 # green # blue)        -- limit
   = GDT [pink|green|yellow|purple|blue]

silly    :: GDT ⟶ Bit 8
silly gdt = gdt.pink + gdt.yellow
```

Additional examples

- Layout, alignment, and initialization of memory-based tables and data structures
- Tracking (and limiting) side effects
  - ensuring some sections of code are "read only"
  - identifying/limiting code that uses privileged operations
  - preventing code that sleeps while holding a lock
  - isolating functions that can only be used during initialization
  - …
- Reusable methods for concise and consistent input validation…
- …

Summary

- The art of language design:
  - drawing inspiration from prior work
  - tastefully adding/subtracting/refining
  - evaluating and iterating (e.g., by comparing programs and reflecting on improvements in productivity and quality)
- DSLs are designed to meet the needs of specific application domains by providing features that reflect the notations, patterns, and challenges of programming in that domain
- But what are the benefits and costs of modern languages?
  - Can advanced abstractions be put to good use?
  - Is it still possible to get acceptable performance?

---

methods for lang design

- consider the purpose
- what can it do differently?
- compare with existing langs
- user surveys
- Turing completeness/ expressivity
- Formal semantics
- Writing programs
- Taking courses on language
- start simple, expand from there
- orthogonality

general goals

- consistency
- different concepts look different, reduce confusion
- simple syntax
- type system
- safety and security
- usable errors/diagnostics
- portability
- human readable specification
- simplicity
- tutorials
- avoid ambiguity
- expandable: libraries, macros, modularity
- supporting infrastructure (libraries, docs, ...)
- fun! experimentation
- debugging, testing, ...
- power/weight ratio
- generic/polymorphic code
- toggle language features and extensions
- abstraction, and target level of abstraction

LLP specific goals

- specific data structures (page tables, etc.)
- efficiency
- transparency around performance
- static error checking
- linkage to assembly/ machine lang
- hints to compiler (pragmas)
- exception handlers
- compiler intrinsics
- concurrency
- bitfields
-

methods for evaluation

- learn quickly
- surveys
- verbosity
- profiling implementations
- compiler correctness
- community
- comparative testing, user study
- metacircular interpreter
- safety analysis