Copyright Notice

• These slides are distributed under the Creative Commons Attribution 3.0 License

• You are free:
  • to share—to copy, distribute and transmit the work
  • to remix—to adapt the work

• under the following conditions:
  • Attribution: You must attribute the work (but not in any way that suggests that the author endorses you or your use of the work) as follows: “Courtesy of Mark P. Jones, Portland State University”

The complete license text can be found at http://creativecommons.org/licenses/by/3.0/legalcode
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
- Unsurprisingly, most drivers today prefer to drive modern cars

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"

Imperative
- Fortran
- BASIC
- COBOL
- C
- Ada
- Pascal
- PHP
- Go
- Rust

Object-Oriented
- Simula
- C++
- Smalltalk
- Java
- Python
- JavaScript
- C#
- Swift

Functional
- Lisp
- Haskell
- F#
- Scala
- Clojure

Expressivity

Limited expressivity
- HTML
- regexps
- grub.cfg
- loader scripts
- ...

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

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

applicable in a wide variety of domains

C  C++  Java  Python  C#  Haskell  Go  Rust

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

A language for specifying application functionality

```c
void hello() {
    int i;
    cls();
    for (i=0; i<2; i++) {
        puts("hhhh    hhhhh\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 on October 3rd")
    }
}
```
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, 0x1 << 0 # Align modules on page boundaries
.set MB_MEMMAP, 0x1 << 1 # Request memory map
.set MB_FLAGS, MB_ALIGN | MB_MEMMAP

.section .multiboot
.align 4
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 %esi, msi
    x, %esp # Set up initial kernel stack
    lo # Catch all, in case hello returns
    hlt # Make space for initial stack

.data
.space 4096
.stack:

-- Done ---------------------------------------------------------------------
```

A language for describing executable file layout

```
hello.c
```

```
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 = ;
}
```

A language for creating multiboot headers

```
boot.s
```
Domain specific languages in the hello demo

### hello.c

```c
set timeout=15
#set default=0
menuentry "Hello, kernel world everybody" {
    multiboot /hello
}
```

### boot.s

```
A language for configuring the boot process
```

### hello.ld

```
A language for describing the build process
```

### grub.cfg

```
A language for configuring the boot process
```

### Makefile

```
# 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 = 1d -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
```

# Domain specific languages in the hello demo

```bash
# 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 = 1d -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
```
Domain specific languages in the hello demo

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

Low Level vs High Level

- Low Level:
  - limited abstractions, reflecting the characteristics of machines on which programs are executed
- High Level:
  - 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

- Limited abstractions, reflecting characteristics of machines on which programs are executed.
- Assembly machine code...
- Built-in and user-defined abstractions, reflecting the ways that programmers express ideas.
- C
- C++
- Ada...
- Smalltalk
- Java
- Python
- Haskell
- Prolog...

• 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 - 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

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));
        } x __attribute__((packed));
        word_t raw;
    };
};

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

Permission values inlined

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

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);
}

"selector"
Example: bitdata types

block fpage {
  field_high  base_address  20
  field       size          6
  padding     reserved     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)

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

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

Example: bitdata types

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 ]

Mimics familiar box notation for bitdata types

Bit-level data layout

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 execute arbitrary code
Description: A use after free issue was addressed through improved memory management.

Impact: An application may be able to execute arbitrary code
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

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

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

.allocPage = do cleanRegisteredPages

                    -- caller must ensure arg is valid
                registerPage :: Page a -> b -> (Page a -> H()) -> H ()
                zeroPage :: Page a -> H ()
                validPage :: Page a -> Bool
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 = Haskell + bits

• 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: `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

  \[
  \text{nonzero} :: \text{Int} \rightarrow \text{Maybe (NonZero Int)}
  \]

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:

  \[
  \text{ave} :: \text{Int} \rightarrow \text{Int} \rightarrow \text{Int}
  \]

  \[
  \text{ave} \ n \ m = (n + m) \ `\text{div}` \ 2
  \]

- Calculating the average of a list of integers:

  \[
  \text{average} :: \text{List Int} \rightarrow \text{Maybe Int}
  \]

  \[
  \text{average} \ \text{nums} = \text{case nonzero (length nums) of}
  \]

  \[
  \begin{align*}
  \text{Just} \ d & \rightarrow \text{Just} \ (\text{sum} \ \text{nums} \ `\text{div}` \ d) \\
  \text{Nothing} & \rightarrow \text{Nothing}
  \end{align*}
  \]

- 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 $\text{Ref } t$) that are guaranteed to point to values of type $t$
  • Physical addresses (of type $\text{PhysAddr } t$)
  • Pointers (of type $\text{Ptr } t$): either a reference or a null
• C groups all these types together as $t*$
• In Habit, they are distinct: $\text{Ptr } t = \text{Maybe (Ref } t)$
• 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 $a[i] = \text{start address of } a + i \times (\text{size of element})$
- 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 OR Safety ... pick one! 😞
Arrays in Habit

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

\[ @ :: \text{Ref (Array } n \ t) \rightarrow \text{Ix } n \rightarrow \text{Ref } t \]

- Fast, no need for a runtime check, no need for a stored length

- Ix n is another abstract type:

  \[ \text{maybeIx :: Int } \rightarrow \text{Maybe (Ix } n) \]
  \[ \text{modIx :: Int } \rightarrow \text{Ix } n \]
  \[ \text{incIx :: Ix } n \rightarrow \text{Maybe (Ix } n) \]

Bit twiddling

- Given two 32 bit input values:
  - base: \[
  \begin{array}{cccccc}
  \_ & \_ & \_ & \_ & \_ & \_ \\
  \end{array}
  \]
  - limit: \[
  \begin{array}{cccccc}
  0 & 0 & 0 & \_ & \_ & \_ \\
  \end{array}
  \]

- Calculate a 64 bit descriptor:

\[
\begin{array}{cccccc}
5 & 3 & 2 & \_ & \_ & \_ \\
\_ & \_ & \_ & \_ & \_ & \_ \\
\end{array}
\]

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

```
In C
```

```
low = (base <= 16)       // purple
    | (limit & 0xffffffff); // blue
high = (base & 0xff0000000) // 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 …
• Programmer describes layout in a type definition:

```haskell
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:

```haskell
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