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 we approach the design of new languages?

• 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

1955
1965
1975
1985
1995
2005
2015

Lisp
Simula
Smalltalk
Java
Haskell
Scala
C
Python
C#
JavaScript
C#
Go
Swift

Classifying programming languages

Classification by "Paradigm"

Imperative

Fortran
BASIC
COBOL
Ada
Pascal
PHP
Go
Rust

Object-Oriented

C
Smalltalk
Pascal
Python

Functional

Lisp
Haskell
F#
Scala
Clojure

Expressivity

Limited expressivity
HTML
regexps
grub.cfg
loader scripts

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

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

Domain specific languages in the hello demo

hello.c

"hello, kernel world"

Mark P. Jones, February 2008

/* ------------------------------------------------------------------------*/
/* 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

        
        
        
        
        
        
        
        
        ...
    }
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 characteristics of machines on which programs are executed.
- Built-in and user-defined abstractions, reflecting the ways that programmers express ideas.

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?

- 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

```c
class mempage_t {
public:
  union {
    struct {
      BITFIELD7(word_t, base_address: 20, execute: 1, write: 1, reserved: 1, size: 6, base: L4_FPAGE_BASE_BITS, x: BITS_WORD - L4_FPAGE_BASE_BITS - 10); 
    } x __attribute__((packed));
    word_t raw;
  }
};
```

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

```c
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);
}
```

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

```c
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);
}
```

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

```c
bitdata Perms = Perms { r, w, x :: Bool }
bitdata Fpage = Fpage { base :: Bit 22 | size :: Bit 6 | reserved :: Bit 1 | perms :: Perms }
```

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

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

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

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

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

Mimics familiar box notation for bitdata types

Rich type system: Bit 22, Bit 6, Bit 1, Perms, and Fpage are distinct types. Mixing these incorrectly will trigger a compile time error!
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, ...)

Managing pages in House

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

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”

\[
\text{Habit} = \text{Haskell} + \text{bits}
\]

• 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

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:

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

• This is a lie!

• Correction: You can divide an integer by a non-zero integer to get an integer result

• In Habit:

\[
\text{div} :: \text{Int} \rightarrow \text{NonZero Int} \rightarrow \text{Int}
\]

• 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:

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

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

• Calculating the average of a list of integers:

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

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

\[
\text{Just d} \rightarrow \text{Just (sum nums \div d)}
\]

\[
\text{Nothing} \rightarrow \text{Nothing}
\]

• 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 $t^*$.
- Described by T on Tony Hoare (who introduced null pointers in the ALGOL W language in 1965) as his “billion dollar mistake”.

Arrays and out of bounds indexes

- Arrays are collections of values stored in contiguous locations in memory.
- Address of $a[i] = \text{start address of } a + i^\text{(size of element)}$.
- 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:
  - Fast, no need for a runtime check, no need for a stored length.
  - $\text{Ix } n$ is another abstract type:
    - $\text{maybeIx} :: \text{Int } \rightarrow \text{Maybe } (\text{Ix } n)$.
    - $\text{modIx} :: \text{Int } \rightarrow \text{Ix } n$.
    - $\text{incIx} :: \text{Ix } n \rightarrow \text{Maybe } (\text{Ix } n)$.

Bit twiddling

- Given two 32 bit input values:
  - base: $\begin{array}{cccccc} \hline \text{low} & \text{mid} & \text{high} \\ \hline \end{array}$.
  - limit: $\begin{array}{cccccc} \hline \text{low} & \text{mid} & \text{high} \\ \hline \end{array}$.
- Calculate a 64 bit descriptor:
  - $\begin{array}{cccccccccccccc} \hline \text{low} & \text{mid} & \text{high} \\ \hline \end{array}$.
- Needed for the calculation of IA32 Global Descriptor Table (GDT) entries.
In assembly

```
movl  base, teax
movl  limit, tebx
mov   teax, tecx
shl   $16, teax
mov   teax, tebx
shr   $16, teax
andl  $0xf0000, tebx
orl   %ecx, teax
shl   $16, %edx
xorl  %ecx, %edx
andl  $0xff, %ecx
mov    %edx, %ecx
shr   $16, %edx
movl  %eax, low
shl   $16, %eax
movl  limit, %ebx
movl  base, %eax
```

In C

```
low = (base << 16) // purple
     | (limit & 0xffff); // blue
high = (base & 0xfff000000) // pink
     | (limit & 0xff0000) // green
     | ((base >> 16) & 0xff) // yellow
     | 0x503200; // white
```

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?