From Code To Computer:
What happens when you hit "go"

Jacob Wilkins

Scientific Computing Department, STFC

May 12, 2021

Overview

What is code?

- A code is just a string of letters and numbers.
- It has to be processed just as anything else does.

```
#"Hej Verden!"o|
print ([]and(0and"Hola mon!"or"Helo Byd!")or"Hai
    dunia!")
```

Python – Hai dunia! – Malay

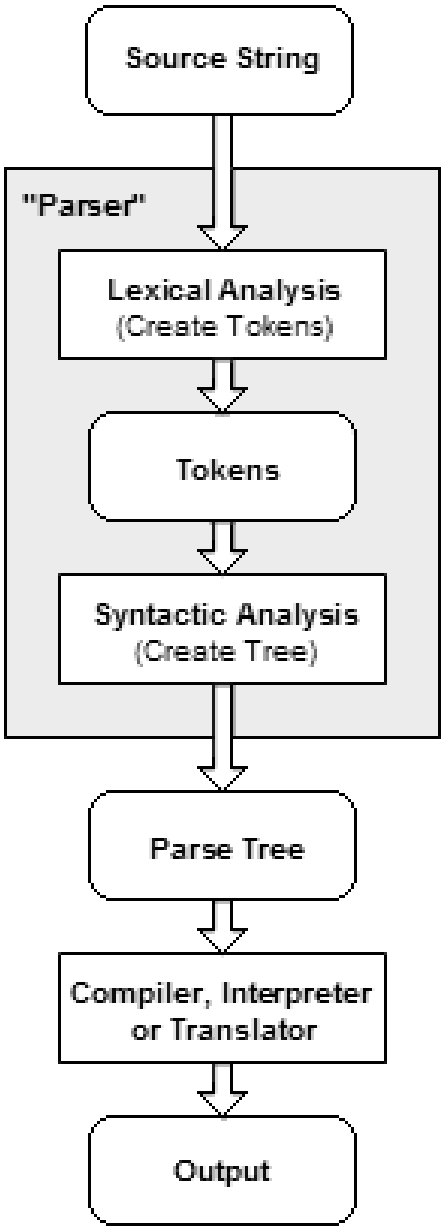
Perl – Helo Byd! – Welsh

Ruby – Hola món! – Catalan

Haystack – Hej Verden! – Danish

https://codegolf.stackexchange.com/questions/146544/hello-world-in-multiple-languages

What happens first?



https://en.wikipedia.org/wiki/Parsing

Our example

- We're going to mostly look at what happens to the following example code:

```
int main () {
    int a = 3;
    int b = 2;
    int x = (a + b) * 2;
    return x;
}
```

Lexers

- A lexer transforms the strings of your code to something the machine can read.
- Recognise the "grammar" of the language and separate it into fundamental, described pieces.
- These pieces are often called "tokens" and in simple terms correspond to parts of speech.

Possible Token	Part of Speech	Example
Identifier	Proper Noun	x, colour, myStr
Operator	Verb	$+$, `pow`, myFunc
Separator	Punctuation	;, :, $\{\}$
Literal	Noun	2, "string", True

Lexers

The line:

x	=	(	a	+	b	)	*	2	;
x	becomes	first	a	add	b	then	times	2	.
PN	V	P	PN	V	PN	P	V	N	P

Might become:

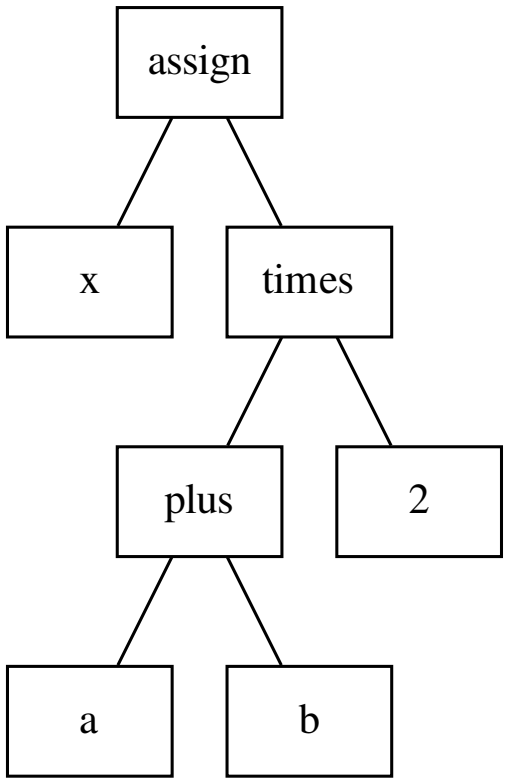
```
[(identifier, x), (operator, =), (separator, (),
(identifier, a), (operator, +), (identifier, b),
(separator, )), (operator, *), (literal, 2),
(separator, ;)]
```

Great, now what?

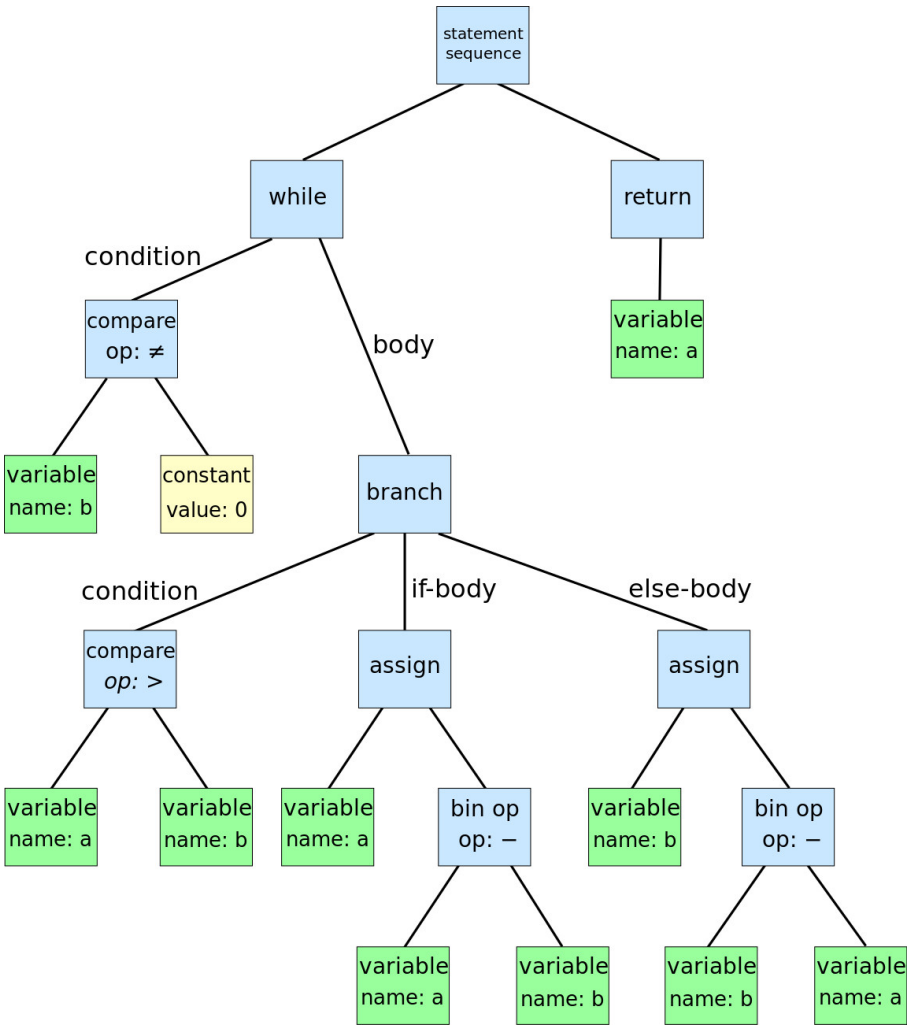
- So, we have broken our code down into chunks, but we still have work to do.
- We need some order of operations.
- We need to know which variable goes with which operation.

Parsers

- Parsers transform these tokens into a tree
- Tree determines order of operations.



Parsers




```
while b != 0
  if a > b
    a := a − b
  else
    b := b − a
return a
```

https://en.wikipedia.org/wiki/Abstract_syntax_tree

Intermediate Representation

- Need to reorder tree into something computer can use.
- Leads to some language which looks like code.
- Different methods of implementation.

Language: Python

```
0  LOAD_NAME          0 (a)
2  LOAD_NAME          1 (b)
4  BINARY_ADD
6  LOAD_CONST         0 (2)
8  BINARY_MULTIPLY
10 STORE_NAME         2 (x)
12 LOAD_CONST         1 (None)
14 RETURN_VALUE
```

Language: GCC (C)

```
(set (reg:SI 87 [ _1 ])
     (plus:SI (reg:SI 90)
              (reg:SI 91)))
(set (reg:SI 92)
     (ashift:SI (reg:SI 87 [ _1 ])
                (const_int 1 [0x1])))
(set (mem/c:SI (plus:DI (reg/f:DI 82 virtual-stack-vars)
                        (const_int -4 [0xfffffffffffffffc]))
                        [1 x+0 S4 A32])
     (reg:SI 92))
```

Intermediate Representation

- Other advantages too:
- For something like GCC or the .NET Framework:
  - Compile many languages
  - Don't want copied code (e.g. optimisation)
    $\Rightarrow$ Compile to some common language
- For something like Java or Python:
  - Feeding code into virtual machine
  - Virtual machine in other (probably lower-level) language

Intermediate Representation

- Often similar to the rendering which can be read by machine.
- Might break down code.

`x = (-b + sqrt(b`$^2 - 4 * a * c))/(2 * a)$; $\Rightarrow$

```
t1:=b*b
t2:=4*a
t3:=t2*c
t4:=t1-t3
t5:=sqrt(t4)
t6:=0-b
t7:=t5+t6
t8:=2*a
t9:=t7/t8
x:=t9
```

Intermediate Representation

- Often similar to the rendering which can be read by machine.
- Might break down code.
- Can contain optimisation hints.

```
(insn 9 8 10 2 (parallel [
            (set (reg:SI 87 [ _1 ])
                (plus:SI (reg:SI 90)
                    (reg:SI 91)))
            (clobber (reg:CC 17 flags))
        ]) "test.c":6 -1
     (expr_list:REG_EQUAL (plus:SI (mem/c:SI (plus:DI (reg/f:DI 82
            virtual-stack-vars)
                (const_int -12 [0xfffffffffffffff4])) [1 a+0 S4 A32])
            (mem/c:SI (plus:DI (reg/f:DI 82 virtual-stack-vars)
                (const_int -8 [0xfffffffffffffff8])) [1 b+0 S4 A32]))
        (nil)))
```

Intermediate Representation

- Often similar to the rendering which can be read by machine.
- Might break down code.
- Can contain optimisation hints.
- Might go through several optimising iterations.

test.c.229r.expand	test.c.230r.vregs	test.c.231r.into_cfglayout
test.c.232r.jump	test.c.244r.reginfo	test.c.264r.outof_cfglayout
test.c.265r.split1	test.c.267r.dfinit	test.c.268r.mode_sw
test.c.269r.asmcons	test.c.273r.ira	test.c.274r.reload
test.c.278r.split2	test.c.282r.pro_and_epilogue	test.c.285r.jump2
test.c.298r.stack	test.c.299r.alignments	test.c.301r.mach
test.c.302r.barriers	test.c.306r.shorten	test.c.307r.nothrow
test.c.308r.dwarf2	test.c.309r.final	test.c.310r.dfinish

Assembly

- Assembly is the lowest level “human readable” code.
- Machine specific.
- Deals explicitly with elements of the chip.

```
main:
.LFB0:
        .cfi_startproc
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq    %rsp, %rbp
        .cfi_def_cfa_register 6
        movl    $3, -12(%rbp)
        movl    $2, -8(%rbp)
        movl    -12(%rbp), %edx
        movl    -8(%rbp), %eax
        addl    %edx, %eax
        addl    %eax, %eax
        movl    %eax, -4(%rbp)
        movl    -4(%rbp), %eax
        popq    %rbp
        .cfi_def_cfa 7, 8
        ret
        .cfi_endproc
```

Interpreted Machine Code

- And interpreted languages?
- Most have a huge switch statement selecting the operation.
- Call functions from interpreter.
- The following is taken from ceval.c in core CPython:

```
case TARGET(BINARY_FLOOR_DIVIDE){...}
case TARGET(BINARY_MODULO): {...}
case TARGET(BINARY_ADD): {
    PyObject *right = POP();
    PyObject *left = TOP();
    PyObject *sum;
    if (PyUnicode_CheckExact(left) &&
            PyUnicode_CheckExact(right)) {
        sum = unicode_concatenate(tstate, left, right, f, next_instr);
        /* unicode_concatenate consumed the ref to left */
    }
    else {
        sum = PyNumber_Add(left, right);
        Py_DECREF(left);
    }
    Py_DECREF(right);
    SET_TOP(sum);
    if (sum == NULL)
        goto error;
    DISPATCH();
}
```

Machine Code

- But computers don't deal with assembly.
- As its name suggests an assembler assembles the assembly into machine code.
- Each number translates to a CPU instruction.

```
5fa: 55                    push   %rbp
5fb: 48 89 e5              mov    %rsp,%rbp
5fe: c7 45 f4 03 00 00 00  movl   $0x3,-0xc(%rbp)
605: c7 45 f8 02 00 00 00  movl   $0x2,-0x8(%rbp)
60c: 8b 55 f4              mov    -0xc(%rbp),%edx
60f: 8b 45 f8              mov    -0x8(%rbp),%eax
612: 01 d0                 add    %edx,%eax
614: 01 c0                 add    %eax,%eax
616: 89 45 fc              mov    %eax,-0x4(%rbp)
619: 8b 45 fc              mov    -0x4(%rbp),%eax
61c: 5d                    pop    %rbp
61d: c3                    retq
```

Why does it change?

- What is optimisation?
- Why does a compiler change my code?

Why does it change?

- What is optimisation?
- Why does a compiler change my code?
- We're going to look at several ways in which a compiler might change your code.

Optimisation

- Compilers have many collective years of experience dealing with code.
- Recognise common patterns and do the smart thing.

Optimisation

- Compilers have many collective years of experience dealing with code.
- Recognise common patterns and do the smart thing.

`gcc -O0 -march=haswell`

```
int main () {
    int a = 3;
    int b = 2;
    int x = (a +
        b) * 2;
    return x;
}
```

```
main:
        push    rbp
        mov     rbp, rsp
        mov     DWORD PTR [rbp-4], 3
        mov     DWORD PTR [rbp-8], 2
        mov     edx, DWORD PTR [rbp-4]
        mov     eax, DWORD PTR [rbp-8]
        add     eax, edx
        add     eax, eax
        mov     DWORD PTR [rbp-12], eax
        mov     eax, DWORD PTR [rbp-12]
        pop     rbp
        ret
```

Optimisation

- Compilers have many collective years of experience dealing with code.
- Recognise common patterns and do the smart thing.

`gcc -O3 -march=haswell`

```
int main () {
    int a = 3;
    int b = 2;
    int x = (a +
        b) * 2;
    return x;
}
```

```
main:
    mov eax, 10
    ret
```

Optimisation - Another Example

```
int countSetBits (int x) {
    int count = 0;
    while (x != 0) {
        count++;
        x &= x-1;
    }
    return count;
}
```

Optimisation - Another Example

`gcc -O0 -march=haswell`

```
int countSetBits (int
    x) {
    int count = 0;
    while (x != 0) {
        count++;
        x &= x-1;
    }
    return count;
}
```

```
countSetBits(int):
        push    rbp
        mov     rbp, rsp
        mov     DWORD PTR [rbp-20], edi
        mov     DWORD PTR [rbp-4], 0
.L3:
        cmp     DWORD PTR [rbp-20], 0
        je      .L2
        inc     DWORD PTR [rbp-4]
        mov     eax, DWORD PTR [rbp-20]
        dec     eax
        and     DWORD PTR [rbp-20], eax
        jmp     .L3
.L2:
        mov     eax, DWORD PTR [rbp-4]
        pop     rbp
        ret
```

Optimisation - Another Example

`gcc -O3 -march=haswell`

```
int countSetBits (int
    x) {
    int count = 0;
    while (x != 0) {
        count++;
        x &= x-1;
    }
    return count;
}
```

```
countSetBits(int):
        xor     eax, eax
        popcnt  eax, edi
        ret
```

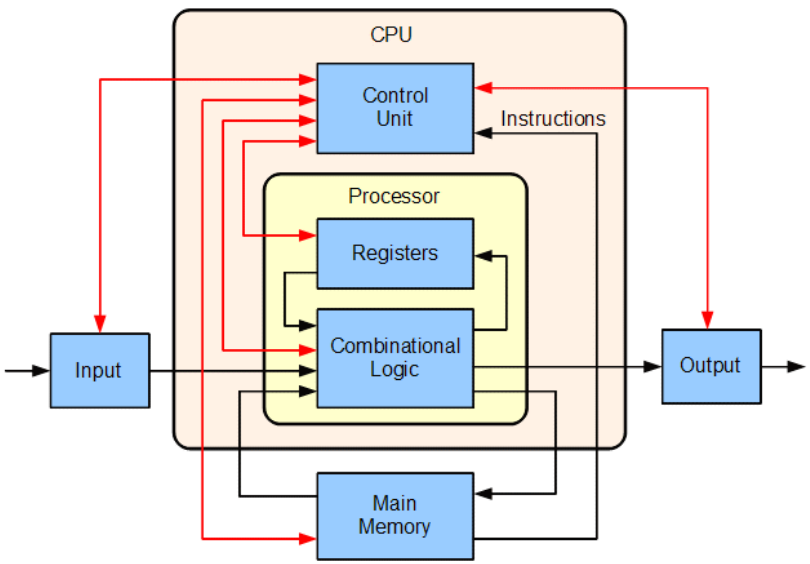
Hitting the chip

- People like to quote the Hz of a CPU (e.g. 2.4GHz)
- Cycle rate like a heartbeat.
- Simplistically RISC CPUs try to run ∼1 instruction per "cycle".

Image by Poil: https://en.wikipedia.org/wiki/Central_processing_unit

Beyond the CPU

- Modern CPUs are complex.
- Multiple layers of memory for faster access.
- Different controllers manage copying memory.



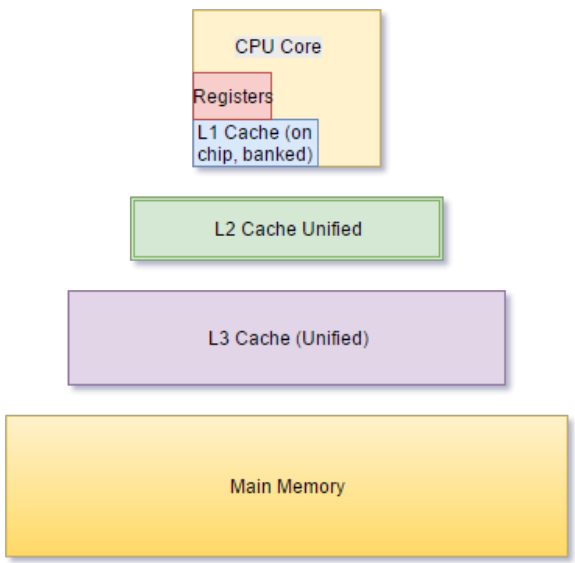


Image by Lambtron: https://en.wikipedia.org/wiki/Central_processing_unit
Image by Kbbuch: https://en.wikipedia.org/wiki/Cache_hierarchy

Stored Up For a Rainy Day

- Cache exists to speed up your calculations.
- Preloads data from memory to make job easier.
- Doesn't just load one value, loads and fills a "cache line"
- Cache misses require loading again from main memory.
- Caches speed up data reuse!

Type	Cycles	Time	Size
L1 CACHE	$\sim$ 4 cycles	2.1 – 1.2 ns	$\sim$ 64 KiB
L2 CACHE	$\sim$ 10 cycles	5.3 – 3.0 ns	$\sim$ 256 KiB
L3 CACHE	$\sim$ 40 cycles	21.4 – 12.0 ns	$\sim$ 4 MiB
Main Memory	—	$\sim$ 60 ns	$\sim$ 8 GiB
SSD	—	$\sim$ 50 μs	$\sim$ 256 GiB
HDD	—	$\sim$ 10 ms	$\sim$ 1 TiB

Table: Memory Access Speed on a Core i7 Xeon 5500 (approximate)
https://software.intel.com/sites/products/collateral/hpc/vtune/performance_analysis_guide.pdf

Through the pipeline

- Modern CPUs "pipeline" data.
- Rather than doing one job at a time, they try to do as much as possible.
- Caches are key in doing this by avoiding memory access.
- May change the order of code to do this better.

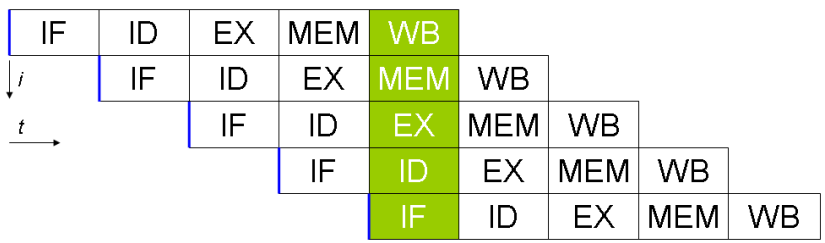


Image by Poil: https://en.wikipedia.org/wiki/Central_processing_unit

Into the “modern” age

- Modern computers are parallel.
- Compiler will change code to exploit vector processing.

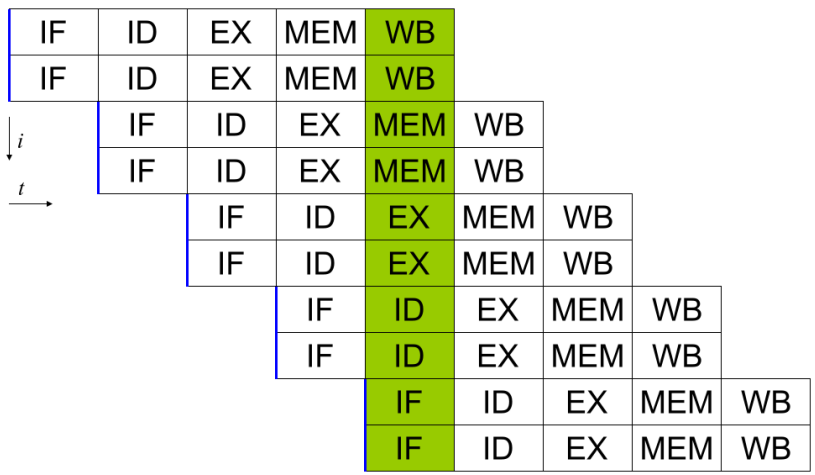


Image by Poil: https://en.wikipedia.org/wiki/Central_processing_unit

Vectorisation Example

Source

```
int testFunction(int* input, int length) {
  int sum = 0;
  for (int i = 0; i < length; ++i) {
    sum += input[i];
  }
  return sum;
}
```

Vectorisation Example

gcc -O0 -march=haswell

```
testFunction(int*, int):
        push    rbp
        mov     rbp, rsp
        mov     QWORD PTR [rbp-24], rdi
        mov     DWORD PTR [rbp-28], esi
        mov     DWORD PTR [rbp-4], 0
        mov     DWORD PTR [rbp-8], 0
.L3:
        mov     eax, DWORD PTR [rbp-8]
        cmp     eax, DWORD PTR [rbp-28]
        jge     .L2
        mov     eax, DWORD PTR [rbp-8]
        cdqe
        lea     rdx, [0+rax*4]
        mov     rax, QWORD PTR [rbp-24]
        add     rax, rdx
        mov     eax, DWORD PTR [rax]
        add     DWORD PTR [rbp-4], eax
        inc     DWORD PTR [rbp-8]
        jmp     .L3
.L2:
        mov     eax, DWORD PTR [rbp-4]
        pop     rbp
        ret
```

gcc -O3 -march=haswell

```
testFunction(int*, int):
        test    esi, esi
        jle     .L7
        lea     eax, [rsi-1]
        cmp     eax, 6
        jbe     .L8
        mov     edx, esi
        mov     rax, rdi
        vpxor   xmm0, xmm0, xmm0
        shr     edx, 3
        sal     rdx, 5
        add     rdx, rdi
.L5:
        vpaddd  ymm0, ymm0, YMMWORD PTR
              [rax]
        add     rax, 32
        cmp     rax, rdx
        jne     .L5
        vmovdqa xmm1, xmm0
        vextracti128    xmm0, ymm0, 0x1
        mov     edx, esi
        vpaddd  xmm0, xmm1, xmm0
        and     edx, -8
        vpsrldq xmm1, xmm0, 8
        vpaddd  xmm0, xmm0, xmm1
        vpsrldq xmm1, xmm0, 4
        vpaddd  xmm0, xmm0, xmm1
        vmovd   eax, xmm0
        test    sil, 7
        je      .L11
        vzeroupper
    [...]
```

You might be smart. Compilers are smarter

- Compilers have many years experience.
- Compilers have looked at CPU instruction sets.
- Compilers know many tricks.

```
return x*2;
return x*8;
return x*32;
return x*7;
return x*3;
return x*65599;
return (x << 16) +
       (x << 6) - x;
```

```
.x2:
lea     eax, [rdi+rdi]
.x8:
lea     eax, [0+rdi*8]
.x32:
sal     eax, 5
.x7:
lea     eax, [0+rdi*8]
sub     eax, edi
.x3:
lea     eax, [rdi+rdi*2]
.x65599:
imul    eax, edi, 65599
.x65599:
imul    eax, edi, 65599
```

Exploring for yourself

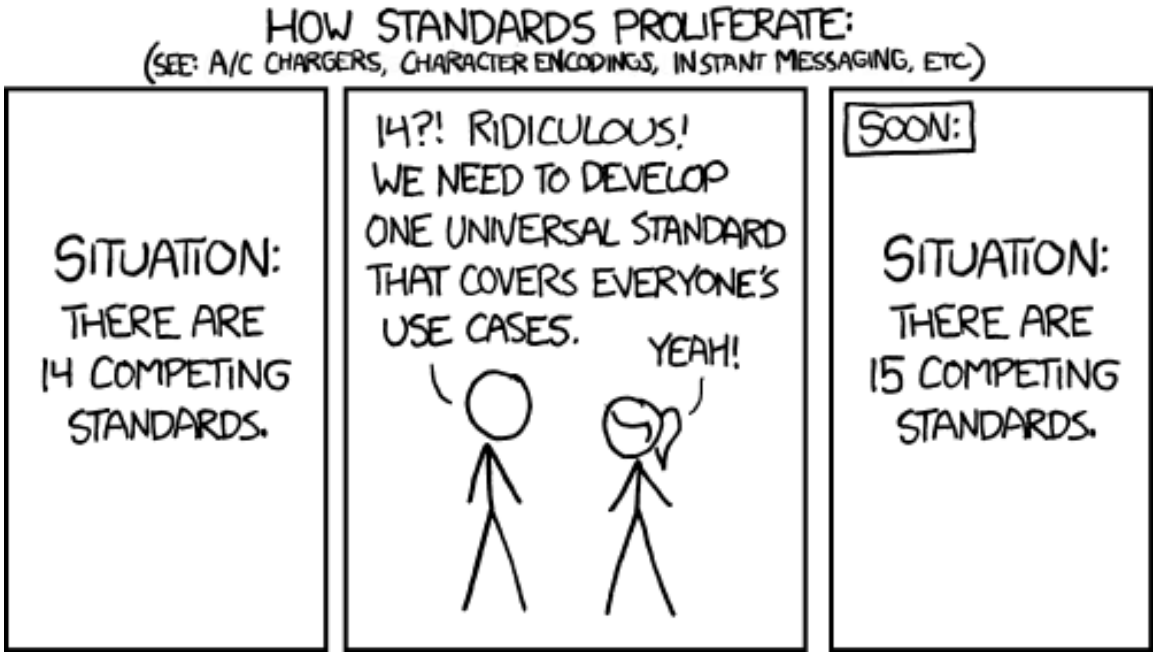
- Excellent talk by Matt Godbolt:
  KEYNOTE: What Everyone Should Know About How Amazing Compilers Are - Matt Godbolt [C++ on Sea 2019]
  `https://www.youtube.com/watch?v=w0sz5WbS5AM`
- Playing with his "Compiler Explorer"
  `godbolt.org`

Menagerie of Languages

- So why is there a zoo of languages?
- Why don't we just have one language we all use to talk to computers?

Menagerie of Languages

- So why is there a zoo of languages?
- Why don't we just have one language we all use to talk to computers?



Not everything is a nail

- C
  - Low-level
  - Full control
  - Strict typing

Not everything is a nail

- C
  - Low-level
  - Full control
  - Strict typing
- Fortran
  - Restrictive
  - Abstracts many tasks
  - Array handling

Not everything is a nail

- C
  - Low-level
  - Full control
  - Strict typing
- Fortran
  - Restrictive
  - Abstracts many tasks
  - Array handling

  ‘
- Python
  - Flexible typing
  - Dynamic allocation
  - Interpreted

Comparison

C

```c
#include <math.h>
int len = 10;
int arr[10] = {2};
for(int i=0; i<len; i++) {pow(arr[i],i);}
```

Fortran

```fortran
integer, dimension(0:9) :: arr
arr = 2
do i = lbound(arr), ubound(arr)
     arr(i) = arr(i)**i
end do
!OR
arr = [(2**i, i=0,9)]
```

Python

```python
arr = [2**i for i in range(10)]
```

Other philosophies/needs

- Only looked at procedural imperative languages...
- Many different philosophies.

Haskell - Functional Language

```
factorial :: Integral -> Integral
factorial 0 = 1
factorial n = n * factorial (n-1)
```

J - Array Programming Language

```
factorial =: */ &: >: @: i.
NB. product after increment to list (0..N-1)
```

LISP - List programming language

```
(defun factorial (n)
   (if (zerop n) 1 (* n (factorial (1- n)))))
```

Matlab - Built-In

```
factorial(N)
```

Different languages for different purposes

- Languages are designed for different purposes.
- Choose the right tool for the job.

Summary

- Compilers have hundreds of combined years of experience.
- Don't try to outsmart the compiler
  - Straightforward, clear code helps humans and the compiler.
  - Use the compiler as the tool it's meant to be.
  - Learn how to use your compiler effectively to help you!
- Learn how to use things by taking them apart.
- Try to choose the right tool for the job before you start.
  - Working around rather than with your language.
  - Not using the core features of your language.
  - Language not able to do what you want.