CS 242:

Programming Languages

9/24/2018

Course staff

Stephanie

Will

Esther

Varun

Today's goals

- Course overview
- Intro to syntax and semantics

Course overview

Problem #1:

What is a programming language?

"A vocabulary and set of grammatical rules for instructing a computer to perform specific tasks."

- *Fundamental of Programming Languages* (Ellis Horowitz)

"A programming language is a notation for writing programs, which are specifications of a computation or algorithm."

- Wikipedia

"Programming languages are the medium of expression in the art of computer programming."

- *Concepts in Programming Languages* (John Mitchell)

"A good programming languge is a conceptual universe for thinking about programming".

- Alan Perlis

When in doubt: majority vote!

http://etc.ch/PvnC

<u>Problem #2:</u>
How do we describe programming languages?

Intel x86 documentation

CVTTPS2DQ—Convert with Truncation Packed Single-Precision Floating-Point Values to Packed Signed Doubleword Integer Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F 5B /r CVTTPS2DQ xmm1, xmm2/m128	RM	V/V	SSE2	Convert four packed single-precision floating-point values from xmm2/mem to four packed signed doubleword values in xmm1 using truncation.
VEX.128.F3.0F.WIG 5B /r VCVTTPS2DQ xmm1, xmm2/m128	RM	V/V	AVX	Convert four packed single-precision floating-point values from xmm2/mem to four packed signed doubleword values in xmm1 using truncation.
VEX.256.F3.0F.WIG 5B /r VCVTTPS2DQ ymm1, ymm2/m256	RM	V/V	AVX	Convert eight packed single-precision floating-point values from ymm2/mem to eight packed signed doubleword values in ymm1 using truncation.
EVEX.128.F3.0F.W0 5B /r VCVTTPS2DQ xmm1 {k1}{z}, xmm2/m128/m32bcst	FV	V/V	AVX512VL AVX512F	Convert four packed single precision floating-point values from xmm2/m128/m32bcst to four packed signed doubleword values in xmm1 using truncation subject to writemask k1.
EVEX.256.F3.0F.W0 5B /r VCVTTPS2DQ ymm1 {k1}{z}, ymm2/m256/m32bcst	FV	V/V	AVX512VL AVX512F	Convert eight packed single precision floating-point values from ymm2/m256/m32bcst to eight packed signed doubleword values in ymm1 using truncation subject to writemask k1.
EVEX.512.F3.0F.W0 5B /r VCVTTPS2DQ zmm1 {k1}{z}, zmm2/m512/m32bcst {sae}	FV	V/V	AVX512F	Convert sixteen packed single-precision floating-point values from zmm2/m512/m32bcst to sixteen packed signed doubleword values in zmm1 using truncation subject to writemask k1.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
FV	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

Converts four, eight or sixteen packed single-precision floating-point values in the source operand to four, eight or

6.7.3.1 Formal definition of `restrict`

1 Let **D** be a declaration of an ordinary identifier that provides a means of designating an object **P** as a restrict-qualified pointer to type **T**.

2 If **D** appears inside a block and does not have storage class **extern**, let **B** denote the block. If **D** appears in the list of parameter declarations of a function definition, let **B** denote the associated block. Otherwise, let **B** denote the block of **main** (or the block of whatever function is called at program startup in a freestanding environment).

3 In what follows, a pointer expression **E** is said to be *based* on object **P** if (at some sequence point in the execution of **B** prior to the evaluation of **E**) modifying **P** to point to a copy of the array object into which it formerly pointed would change the value of **E**.[119] Note that "based" is defined only for expressions with pointer types.

4 During each execution of **B**, let **L** be any lvalue that has **&L** based on **P**. If **L** is used to access the value of the object **X** that it designates, and **X** is also modified (by any means), then the following requirements apply: **T** shall not be const-qualified. Every other lvalue used to access the value of **X** shall also have its address based on **P**. Every access that modifies **X** shall be considered also to modify **P**, for the purposes of this subclause. If **P** is assigned the value of a pointer expression **E** that is based on another restricted pointer object **P2**, associated with block **B2**, then either the execution of **B2** shall begin before the execution of **B**, or the execution of **B2** shall end prior to the assignment. If these requirements are not met, then the behavior is undefined.

5 Here an execution of **B** means that portion of the execution of the program that would correspond to the lifetime of an object with scalar type and automatic storage duration

---

119) In other words, **E** depends on the value of **P** itself rather than on the value of an object referenced

```
will highcpu ~/gcc ❯❯❯ ./cloc-1.78.pl .
   81177 text files.
   80315 unique files.
    4674 files ignored.

github.com/AlDanial/cloc v 1.78  T=92.54 s (827.0 files/s, 121670.8 lines/s)
```

Language	files	blank	comment	code
C	32429	556567	579624	2929550
Ada	5711	277381	368655	783143
C++	22434	169922	226650	747203
PO File	42	269374	363087	715169
Go	3665	72049	109674	542915
C/C++ Header	3165	123298	162805	522181
Markdown	437	50424	0	414990
Bourne Shell	142	77080	63019	409052
Fortran 90	5376	25125	41675	142157
m4	198	8110	2549	71077
Assembly	585	12706	31134	62206
XML	61	6307	543	46731
Windows Module Definition	150	6171	35	46472
Expect	326	6786	12325	28625
HTML	114	523	29	26936
Objective C	525	4940	3162	16966
Perl	23	1470	2234	14135
make	122	2222	1464	13327
Fortran 77	502	1415	4326	11920
Objective C++	247	2407	1567	8178
TeX	4	826	3308	6557
Python	27	1669	1815	6199
Pascal	24	1335	6949	5321
MSBuild script	7	1	0	4675
awk	21	548	702	3877
Fortran 95	98	898	2412	2600
Bourne Again Shell	20	457	689	2034
C#	9	230	506	879
JSON	4	0	0	384
yacc	1	56	37	316
OCaml	1	44	29	285
Standard ML	1	34	28	215
vim script	4	50	71	193
CMake	1	32	31	186
lex	1	34	30	154
Haskell	38	17	0	122
NAnt script	2	17	0	103
Windows Resource File	3	5	3	96
MATLAB	3	13	0	46
SAS	1	14	22	32
Brainfuck	1	3	4	10
SWIG	3	2	9	9
Lisp	1	4	12	8
DOS Batch	2	0	0	4
CSS	1	0	0	1
SUM:	76532	1680566	1991214	7587239

std::move_if_noexcept

Defined in header <utility>

```
template< class T >
typename std::conditional<
    !std::is_nothrow_move_constructible<T>::value && std::is_copy_constructible<T>::value,
    const T&,
    T&&
>::type move_if_noexcept(T& x) noexcept;
```
(since C++ (until C++

```
template< class T >
constexpr typename std::conditional<
    !std::is_nothrow_move_constructible<T>::value && std::is_copy_constructible<T>::value,
    const T&,
    T&&
>::type move_if_noexcept(T& x) noexcept;
```
(since C++

ove_if_noexcept obtains an rvalue reference to its argument if its move constructor does not throw exceptions or if here is no copy constructor (move-only type), otherwise obtains an lvalue reference to its argument. It is typically sed to combine move semantics with strong exception guarantee.

Parameters

x - the object to be moved or copied

Return value

`std::move(x)` or `x`, depending on exception guarantees.

Notes

This is used, for example, by `std::vector::resize`, which may have to allocate new storage and then move or copy elements from old storage to new storage. If an exception occurs during this operation, `std::vector::resize` undoes everything it did to this point, which is only possible if `std::move_if_noexcept` was used to decide whether to use move construction or copy construction. (unless copy constructor is not available, in which case move constructor is used either way and the strong exception guarantee may be waived)

Example

Naming and binding

es refer to objects. Names are introduced by name binding operations. Each occurrence of a name in the program text refers to the *binding* of that name established in
containing the use.

ck is a piece of Python program text that is executed as a unit. The following are blocks: a module, a function body, and a class definition. Each command typed interact
a file given as standard input to the interpreter or specified on the interpreter command line the first argument) is a code block. A script command (a command spec
mand line with the '**-c**' option) is a code block. The file read by the built-in function `execfile()` is a code block. The string argument passed to the built-in function `eval()` an
ode block. The expression read and evaluated by the built-in function `input()` is a code block.

de block is executed in an *execution frame*. A frame contains some administrative information (used for debugging) and determines where and how execution continue
ution has completed.

ope defines the visibility of a name within a block. If a local variable is defined in a block, its scope includes that block. If the definition occurs in a function block, the scop
ained within the defining one, unless a contained block introduces a different binding for the name. The scope of names defined in a class block is limited to the class bloc
ode blocks of methods – this includes generator expressions since they are implemented using a function scope. This means that the following will fail:

```
s A:
a = 42
b = list(a + i for i in range(10))
```

n a name is used in a code block, it is resolved using the nearest enclosing scope. The set of all such scopes visible to a code block is called the block's *environment*.

ame is bound in a block, it is a local variable of that block. If a name is bound at the module level, it is a global variable. (The variables of the module code block are local a

ed in a code block but not defined there, it is a *free variable*.

n a name is not found at all, a `NameError` exception is raised. If the name refers to a local variable that has not been bound, a `UnboundLocalError` exception is raised.
ass of `NameError`.

ollowing constructs bind names: formal parameters to functions, `import` statements, class and function definitions (these bind the class or function name in the defining blo
fiers if occurring in an assignment, `for` loop header, in the second position of an `except` clause header or after `as` in a `with` statement. The `import` statement of the form `f
mes defined in the imported module, except those beginning with an underscore. This form may only be used at the module level.

get occurring in a `del` statement is also considered bound for this purpose (though the actual semantics are to unbind the name). It is illegal to unbind a name that is refe
e; the compiler will report a `SyntaxError`.

assignment or import statement occurs within a block defined by a class or function definition or at the module level (the top-level code block).

ame binding operation occurs anywhere within a code block, all uses of the name within the block are treated as references to the current block. This can lead to error
n a block before it is bound. This rule is subtle. Python lacks declarations and allows name binding operations to occur anywhere within a code block. The local variables
mined by scanning the entire text of the block for name binding operations.

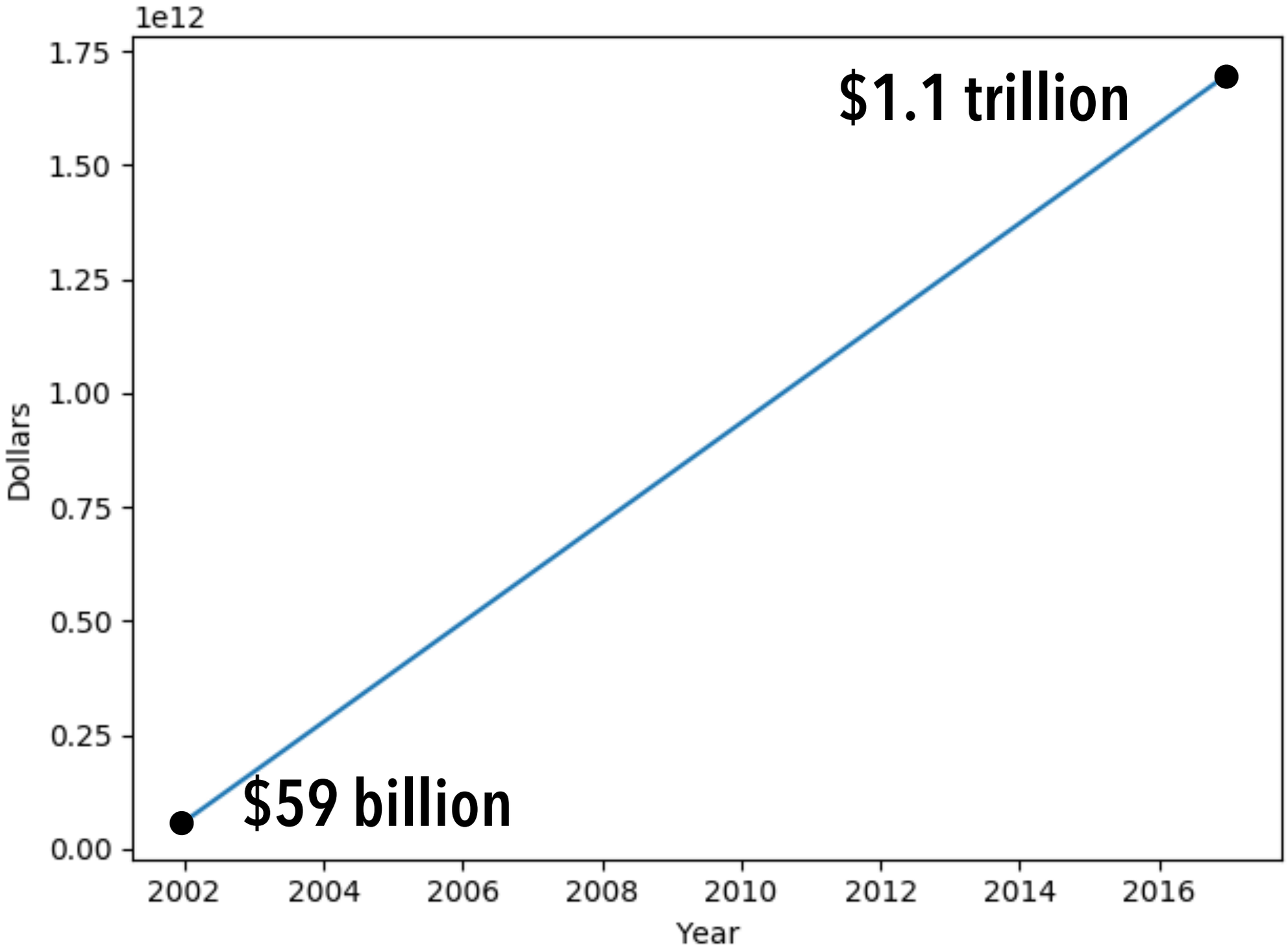
global statement occurs within a block, all uses of the name specified in the statement refer to the binding of that name in the top-level namespace. Names are r
espace by searching the global namespace, i.e. the namespace of the module containing the code block, and the builtins namespace, the namespace of the module
espace is searched first. If the name is not found there, the builtins namespace is searched. The global statement must precede all uses of the name.

uiltins namespace associated with the execution of a code block is actually found by looking up the name `__builtins__` in its global namespace; this should be a diction
case the module's dictionary is used). By default, when in the `__main__` module, `__builtins__` is the built-in module `__builtin__` (note: no 's'); when in any other module,

<u>Problem #3:</u>

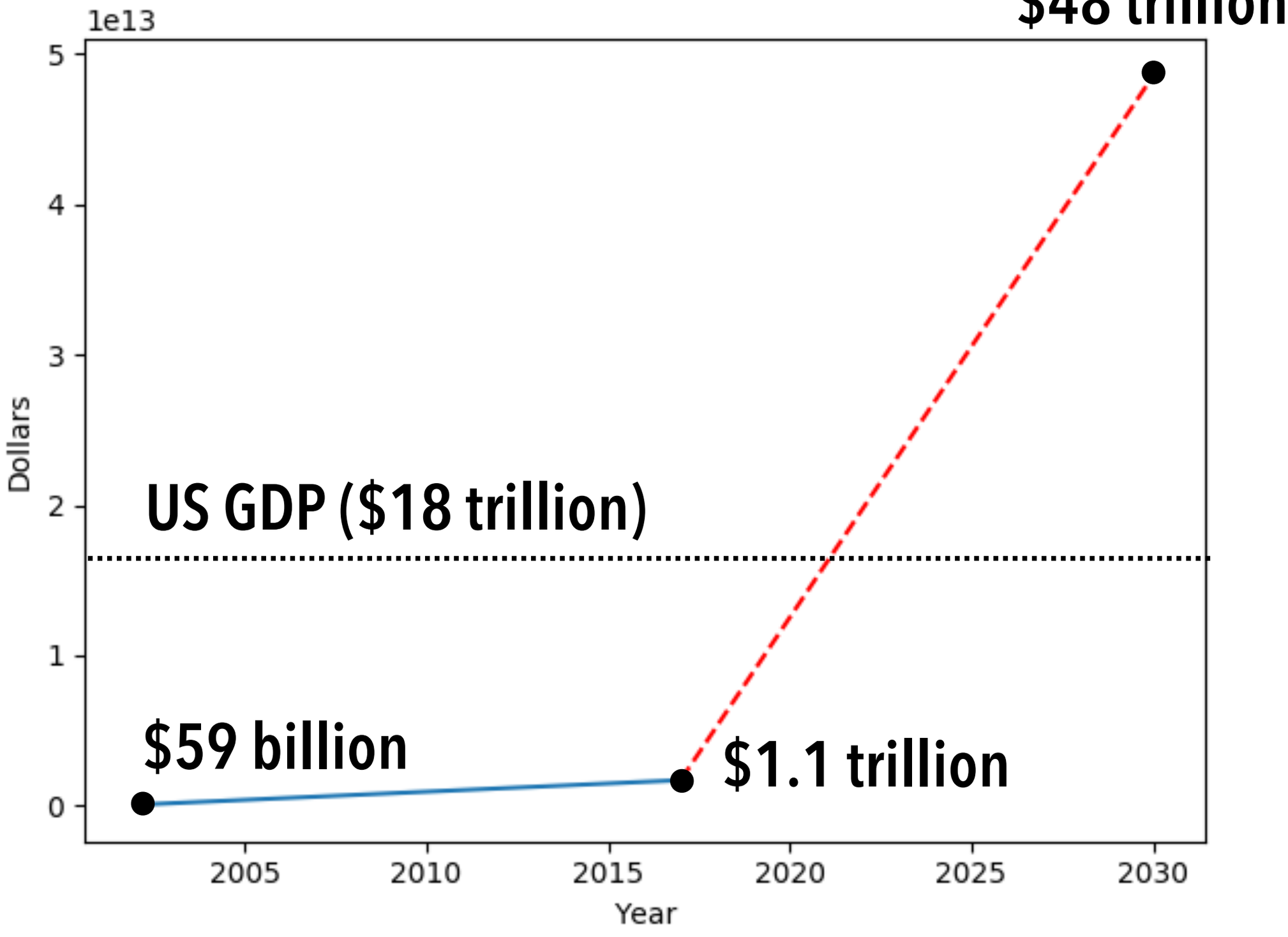
How do we reason about programming languages?

Annual cost of software bugs



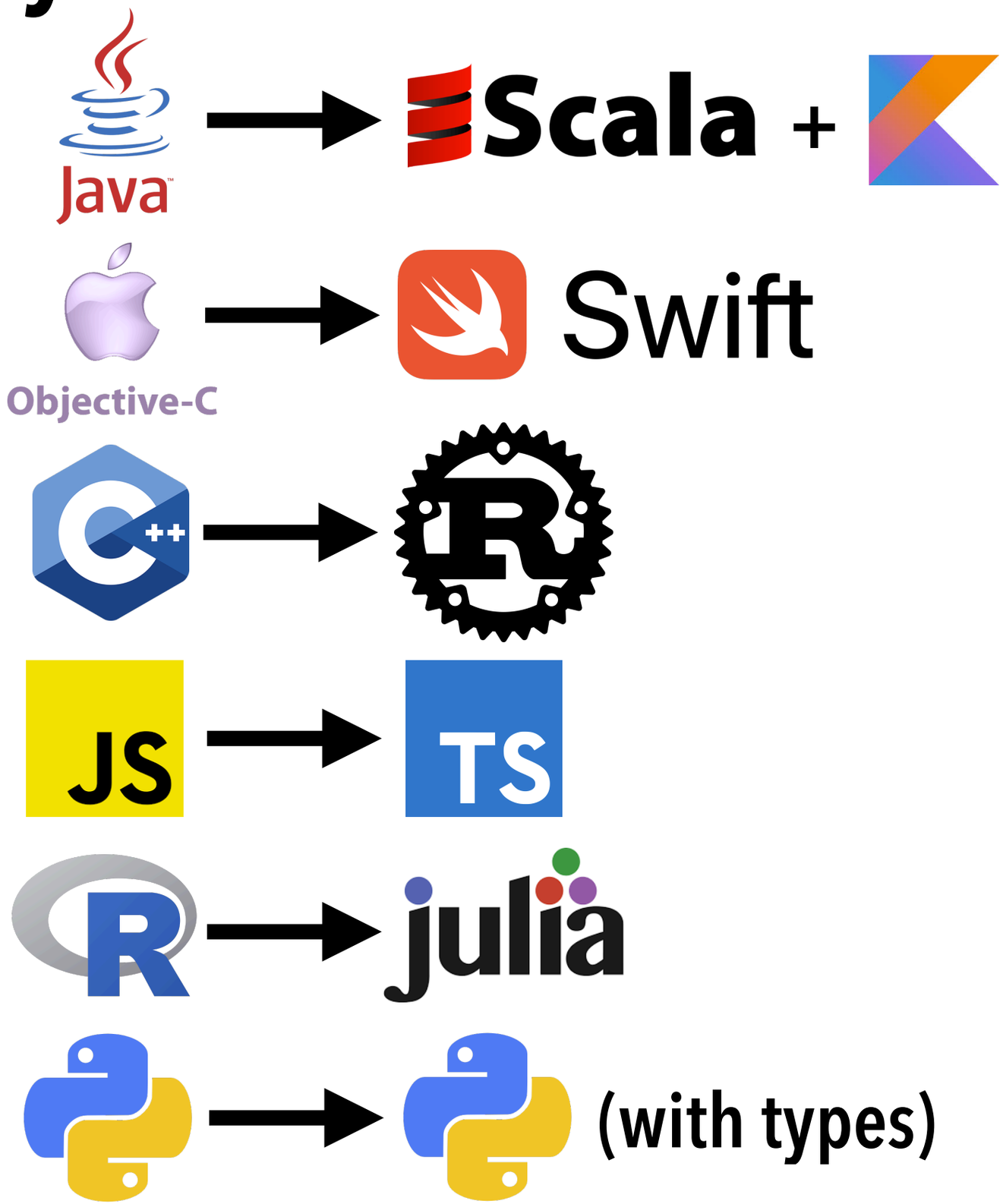
Sources: NIST 2002, Tricentis 2017

Annual cost of software bugs (extrapolated)



Sources: NIST 2002, Tricentis 2017

Type safety matters more than ever



Dennard scaling is dead

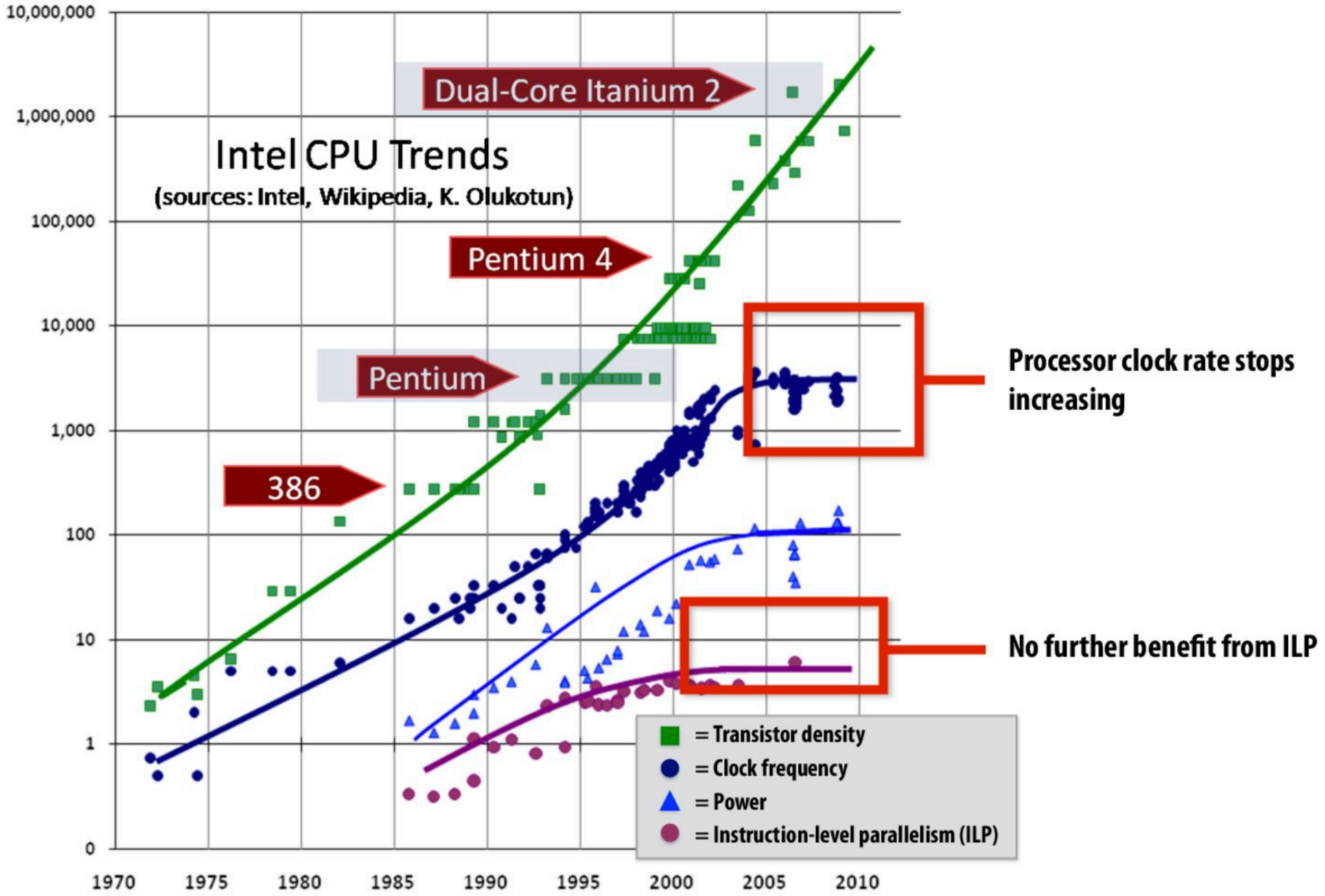


Image credit: "The free Lunch is Over" by Herb Sutter, Dr. Dobbs 2005

High-performance DSLs reign

TensorFlow

```
a = tf.placeholder(tf.int16)
b = tf.placeholder(tf.int16)

# Define some operations
add = tf.add(a, b)
mul = tf.multiply(a, b)

# Launch the default graph.
with tf.Session() as sess:
    # Run every operation with variable input
    print("Addition with variables: %i" % sess.run(add, feed_dict={a: 2, b: 3}))
print("Multiplication with variables: %i" % sess.run(mul, feed_dict={a: 2, b: 3}))
```

Halide

```
Func blur_3x3(Func input) {
  Func blur_x, blur_y;
  Var x, y, xi, yi;

  // The algorithm - no storage or order
  blur_x(x, y) = (input(x-1, y) + input(x, y) + input(x+1, y))/3;
  blur_y(x, y) = (blur_x(x, y-1) + blur_x(x, y) + blur_x(x, y+1))/3;

  // The schedule - defines order, locality; implies storage
  blur_y.tile(x, y, xi, yi, 256, 32)
        .vectorize(xi, 8).parallel(y);
  blur_x.compute_at(blur_y, x).vectorize(x, 8);

  return blur_y;
}
```

Course theme:

Bridging the gap between systems and theory

Course overview

- Theory
  - Formal systems for describing and reasoning about PLs
  - Core vocabulary for describing programming constructs
  - Essentials of the functional programming paradigm

- Systems
  - Apply formal methods to real world languages and systems
  - Reason about memory safety, state machines, assembly, …
  - Compare value of dynamic vs. static typing

Syllabus

- **Weeks 1-3: Theory**
  - Lambda calculus and OCaml
  - The language of programming languages
  - Functional programming basics

- **Weeks 4-5: WebAssembly**
  - Case study on applying formal semantics to low level languages

- **Weeks 5-7: Rust**
  - Memory safety, traits, concurrency, state machines, and communication
  - (Plus a WebAssembly interpreter!)

- **Weeks 8-9: Lua**
  - Dynamic vs. static typing, object systems

Course structure

- **Weekly assignments**
  - Submit up to 3 days late per assignment, 5 late days total over semester
  - Mixed programming/written
- **No midterm**
  - … But there is a final
- **No required readings**
  - Supplementary material in the syllabus
  - All lecture notes/code will be posted online
  - Lecture videos available through SCPD

Prerequisites

- CS 103: induction, first order logic, basic proofs
- CS 110: assembly (ARM or x86), C, concurrency (threads, synchronization primitives)

Intro to syntax and semantics