CS 242: Programming Languages

Will Crichton
Course staff

Will

Jayden

Danny

Sicheng
fun innerproduct(a, b, n):
    c := 0
    for i := 1 step 1 until n do
        c := c + a[i] * b[i]
    return c
fun innerproduct(a, b, n):
    c := 0
    for i := 1 step 1 until n do
        c := c + a[i] * b[i]
    return c

• Statements operate on invisible state
• Computes word-at-a-time by repetition of assignment/modification
• Requires names for arguments, iterator, return value
fun innerproduct(a, b, n):
    c := 0
    for i := 1 step 1 until n do
        c := c + a[i] * b[i]
    return c

• Statements operate on invisible state
• Computes word-at-a-time by repetition of assignment/modification
• Requires names for arguments, iterator, return value

let innerproduct = zip |> (map *) |> (reduce +)
fun innerproduct(a, b, n):
    c := 0
    for i := 1 step 1 until n do
        c := c + a[i] * b[i]
    return c

- Statements operate on invisible state
- Computes word-at-a-time by repetition of assignment/modification
- Requires names for arguments, iterator, return value

let innerproduct = zip |> (map *) |> (reduce +)

- Built from composable functions (map, reduce, pipe)
- Operates on whole conceptual units (lists), no repeated steps
- No names for arguments or temporaries
public class Person {
    private final String firstName;
    private final String lastName;
    private final Integer age;
    public Person(String firstName,
                   String lastName,
                   Integer age) {
        this.firstName = firstName;
        this.lastName = lastName;
        this.age = age;
    }
    public String getFirst() {
        return firstName;
    }
    public String getLast() {
        return lastName;
    }
    public Integer getAge() {
        return age;
    }
    public Boolean valid() {
        return age > 18;
    }
}
public class Person {
  private final String firstName;
  private final String lastName;
  private final Integer age;
  public Person(String firstName, String lastName, Integer age) {
    this.firstName = firstName;
    this.lastName = lastName;
    this.age = age;
  }
  public String getFirst() {
    return firstName;
  }
  public String getLast() {
    return lastName;
  }
  public Integer getAge() {
    return age;
  }
  public Boolean valid() {
    return age > 18;
  }
}

public static List<String> validByAge(List<Person> in) {
  List<Person> people = new ArrayList<Person>();
  for (Person p: in) {
    if (p.valid()) people.add(p);
  }
  Collections.sort(people, new Comparator<Person>() {
    public int compare(Person a, Person b) {
      return a.age() - b.age();
    }
  });
  List<String> ret = new ArrayList<String>();
  for (Person p: people) {
    ret.add(p.first);
  }
  return ret;
}

List<Person> input = new ArrayList<Person>();
input.add(new Person("John", "Valid", 32));
input.add(new Person("John", "InValid", 17));
input.add(new Person("OtherJohn", "Valid", 19));
List<Person> output = validByAge(input)
public class Person {
    private final String firstName;
    private final String lastName;
    private final Integer age;
    public Person(String firstName, String lastName, Integer age) {
        this.firstName = firstName;
        this.lastName = lastName;
        this.age = age;
    }
    public String getFirst() {
        return firstName;
    }
    public String getLast() {
        return lastName;
    }
    public Integer getAge() {
        return age;
    }
    public Boolean valid() {
        return age > 18;
    }
}

public static List<String> validByAge(List<Person> in) {
    List<Person> people = new ArrayList<Person>();
    for (Person p: in) {
        if (p.valid()) people.add(p);
    }
    Collections.sort(people, new Comparator<Person>() {
        public int compare(Person a, Person b) {
            return a.age() - b.age();
        }
    });
    List<String> ret = new ArrayList<String>();
    for (Person p: people) {
        ret.add(p.first);
    }
    return ret;
}

List<Person> input = new ArrayList<Person>();
input.add(new Person("John", "Valid", 32));
input.add(new Person("John", "Invalid", 17));
input.add(new Person("OtherJohn", "Valid", 19));

List<Person> output = validByAge(input)

case class Person(val first: String, val last: String, val age: Int) {
    def valid: Boolean = age > 18
}

def validByAge(in: List[Person]) =
    in.filter(_.valid).sortBy(_.age).map(_.first)

validByAge(List(
    Person("John", "Valid", 32),
    Person("John", "Invalid", 17),
    Person("OtherJohn", "Valid", 19)))
Fact #1: Developers are reading, not writing


<table>
<thead>
<tr>
<th>Project</th>
<th>Comprehension</th>
<th>Navigation</th>
<th>Editing</th>
<th>Others</th>
</tr>
</thead>
<tbody>
<tr>
<td>Average</td>
<td>57.62%</td>
<td>23.96%</td>
<td>5.02%</td>
<td>13.40%</td>
</tr>
</tbody>
</table>
Fact #2: Software ecosystems are complex
Fact #2: Software ecosystems are complex

1. Learn some CI/CD Tool
   - Jenkins
   - Travis CI
   - TeamCity
   - Drone
   - Circle CI

2. Learn how to monitor software and infrastructure
   - AWS
   - Google Cloud
   - Azure
   - Digital Ocean
   - Heroku

3. Cloud Providers
   - ELK Stack
   - Graylog
   - Splunk
   - Papertrail

4. Infrastructure Monitoring
   - Nagios
   - Icinga
   - Datadog
   - Zabbix
   - Monit

5. Application Monitoring
   - AppDynamics
   - New Relic

6. Logs Management
   - ELK Stack
   - Graylog
   - Splunk
   - Papertrail

7. Containers
   - Docker
   - rkt
   - LXC

8. Configuration Management
   - Ansible
   - Salt
   - Chef
   - Puppet

9. Container Orchestration
   - Kubernetes
   - Docker Swarm
   - Mesos
   - Nomad

10. Infrastructure Provisioning
    - Terraform
    - Cloud Formation

Keep Exploring
std::move_if_noexcept

Defined in header `<utility>`

```cpp
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;
```

```cpp
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;
```

`move_if_noexcept` obtains an rvalue reference to its argument if its move constructor does not throw exceptions or if there is no copy constructor (move-only type), otherwise obtains an lvalue reference to its argument. It is typically used 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**
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 by \( P \).
Fact #3: Dennard scaling is dead
Memory usage continues to scale

The table shows the memory usage of various processes, with the Slack Helper and Google Chrome Helper processes consuming significant amounts of memory.

- Slack Helper: 738.8 MB
- Slack Helper: 419.8 MB
- Slack Helper: 399.4 MB
- Slack Helper: 366.4 MB
- Google Chrome Helper: 317.8 MB
- Google Chrome Helper: 57.2 MB
- Google Chrome Helper: 54.9 MB
- Google Chrome Helper: 53.5 MB
- Google Chrome Helper: 45.2 MB
- Google Chrome Helper: 44.5 MB
- Google Chrome Helper: 44.0 MB
- Google Chrome Helper: 43.8 MB
- Google Chrome Helper: 43.6 MB
- Google Chrome Helper: 43.5 MB
- Google Chrome Helper: 43.3 MB
- Google Chrome Helper: 43.2 MB
- Google Chrome Helper: 43.1 MB
- Google Chrome Helper: 42.8 MB
- Google Chrome Helper: 40.0 MB
- Google Chrome Helper: 30.1 MB
- Google Chrome Helper: 18.8 MB
- Google Chrome Helper: 18.4 MB
- Google Chrome Helper: 17.3 MB

The total memory usage is around 700 MB.

3 GB!!!
Huge perf gap for most programs

<table>
<thead>
<tr>
<th></th>
<th>Immutable</th>
<th>Mutable</th>
<th>Double Only</th>
<th>No Objects</th>
<th>In C</th>
<th>Transposed</th>
<th>Tiled</th>
<th>Vectorized</th>
<th>BLAS MxM</th>
<th>BLAS Parallel</th>
</tr>
</thead>
<tbody>
<tr>
<td>ms</td>
<td>17,094,152</td>
<td>77,826</td>
<td>32,800</td>
<td>15,306</td>
<td>7,530</td>
<td>2,275</td>
<td>1,388</td>
<td>511</td>
<td>196</td>
<td>58</td>
</tr>
</tbody>
</table>

- 219.7x
- 2.2x
- 3.4x
- 2.8x
- 3.5x
- 2.4x
- 2.1x
- 1.7x
- 2.7x
- 219.7x
- 522x
- 1117x
- 2271x
- 7514x
- 12316x
- 33453x
- 87042x
- 296260x

| Cycles/OP    | 8,358    | 38       | 16         | 7         | 4     | 1         | 1/2     | 1/5        | 1/11     | 1/36         |

Saman Amarasinghe, MIT 6.172 “Performance Engineering”. 2009
Maybe you don't need Rust and WASM to speed up your JS

Vyacheslav Egorov on 03 feb 2018
Maybe you don't need Rust and WASM to speed up your JS
Maybe you don't need Rust and WASM to be efficient.
Maybe you don't need Rust and WASM to speed up the Web, Swift and Emscripten could be enough.
You don't need Rust and WASM to become a software developer.
I trawled V8 issue tracker and found few active issues like these:

- **Issue 6391**: StringCharCodeAt slower than Crankshaft;
- **Issue 7092**: High overhead of String.prototype.charCodeAt in typescript test;
- **Issue 7326**: Performance degradation when looping across character codes of a string;
I trawled V8 issue tracker and found few:

- Issue 6391: StringCharCodeAt slow.
- Issue 7092: High overhead of String.prototype.getMethod.
- Issue 7326: Performance degradation

<table>
<thead>
<tr>
<th>Overhead Symbol</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>doQuickSort</td>
<td>17.92%</td>
</tr>
<tr>
<td>BuiltInArgumentsAdaptor::doQuickSort</td>
<td>11.20%</td>
</tr>
<tr>
<td>BuiltInArgumentsAdaptor::doQuickSort</td>
<td>7.17%</td>
</tr>
<tr>
<td>BuiltInArgumentsAdaptor::doQuickSort</td>
<td>4.99%</td>
</tr>
<tr>
<td>BuiltInArgumentsAdaptor::doQuickSort</td>
<td>3.68%</td>
</tr>
<tr>
<td>BuiltInArgumentsAdaptor::doQuickSort</td>
<td>2.73%</td>
</tr>
<tr>
<td>BuiltInArgumentsAdaptor::doQuickSort</td>
<td>2.11%</td>
</tr>
<tr>
<td>BuiltInStringEqual</td>
<td>1.09%</td>
</tr>
<tr>
<td>SourceMapConsumer::parseMapping</td>
<td>1.06%</td>
</tr>
<tr>
<td>SourceMapConsumer::parseMapping</td>
<td>1.60%</td>
</tr>
<tr>
<td>v8::internal::StringTable::LookupUtf8StringIfNullOrUndefined</td>
<td>3.26%</td>
</tr>
<tr>
<td>v8::internal::StringTable::LookupUtf8StringIfNullOrUndefined</td>
<td>2.72%</td>
</tr>
<tr>
<td>v8::internal::StringTable::LookupUtf8StringIfNullOrUndefined</td>
<td>1.21%</td>
</tr>
<tr>
<td>BuiltInStringCharAt</td>
<td>1.16%</td>
</tr>
<tr>
<td>BuiltInStringCharAt</td>
<td>1.04%</td>
</tr>
<tr>
<td>BuiltInStringCharAt</td>
<td>0.18%</td>
</tr>
<tr>
<td>BuiltInStringPrototypeIsLoose</td>
<td>0.05%</td>
</tr>
<tr>
<td>BuiltInStringPrototypeIsLoose</td>
<td>0.03%</td>
</tr>
<tr>
<td>BuiltInStringPrototypeIsLoose</td>
<td>0.02%</td>
</tr>
<tr>
<td>BuiltInStringPrototypeIsLoose</td>
<td>0.01%</td>
</tr>
<tr>
<td>BuiltInScavenger::ScavengerObject</td>
<td>0.76%</td>
</tr>
<tr>
<td>BuiltInScavenger::ScavengerObject</td>
<td>0.72%</td>
</tr>
<tr>
<td>v8::internal::String::IteratingStringHasher::reg</td>
<td>0.68%</td>
</tr>
<tr>
<td>v8::internal::String::IteratingStringHasher::reg</td>
<td>0.65%</td>
</tr>
<tr>
<td>v8::internal::String::IteratingStringHasher::reg</td>
<td>0.64%</td>
</tr>
<tr>
<td>v8::internal::IncrementalMarking::RecordWriteSlow</td>
<td>0.50%</td>
</tr>
</tbody>
</table>
Fact #4: Bugs are not improving

“The majority of vulnerabilities fixed and with a CVE assigned are caused by developers inadvertently inserting memory corruption bugs into their C and C++ code.”

Microsoft Security Response Center. “A proactive approach to more secure code.” 2019
Facts of modern software development

1. Developers are mostly comprehending/debugging code
2. Software ecosystems are growing in complexity
3. CPU hardware performance is stalling
4. Bugs aren’t getting any better
1. How did we get here?

2. What is the way out?
Alan Turing

“On Computable Numbers” 1937

Alonzo Church

“A set of postulates for the foundation of logic”, 1932
Lambda calculus

\[
\begin{align*}
[x \rightarrow y] x & = y \\
[x \rightarrow y] z & = z \\
[x \rightarrow y] \lambda z \cdot x & = \lambda z \cdot y \\
[x \rightarrow y] \lambda y \cdot x y & = \lambda y' \cdot y y' \\
[x \rightarrow y] x (\lambda x \cdot x) & = y (\lambda x \cdot x)
\end{align*}
\]
Lambda calculus

\[
\begin{align*}
[x \to y] x &= y \\
[x \to y] z &= z \\
[x \to y] \lambda z \cdot x &= \lambda z \cdot y \\
[x \to y] \lambda y \cdot x y &= \lambda y' \cdot y y' \\
[x \to y] x (\lambda x \cdot x) &= y (\lambda x \cdot x)
\end{align*}
\]
John Backus,
“History of FORTRAN” 1978
Prior to FORTRAN, most source language operations were not machine operations. Large inefficiencies in looping and computing addresses were masked by time spent in floating point subroutines.

The advent of the 704 with built-in floating point and indexing radically altered the situation. …It increased the problem of generating efficient programs by an order of magnitude by speeding up floating point operations by a factor of ten and thereby leaving inefficiencies nowhere to hide. This caused us to regard the design of the translator as the real challenge, not the simple task of designing the language.
Algon introduced into programming languages such terms as type, declaration, identifier, for statement, while, if then else, switch, the begin end delimiters, block, call by value and call by name, typed procedures, declaration scope, dynamic arrays, side effects, global and local variables.

Algon was strongly derived from FORTRAN and its contemporaries. The logic, arithmetic and data organizations were close to those then being designed into real computers. Certain simple generalizations of computer instructions such as switch, for statement, and if statements were included because their semantics and computer processing were straight-forward consequences of single statement processing.
C fits firmly in the traditional procedural family typified by Fortran and Algol 60. It is ‘close to the machine’ in that the abstractions it introduces are readily grounded in the concrete data types and operations supplied by conventional computers.

The most important [historical accident] has been the tolerance of C compilers to errors in type. C evolved from typeless languages. It did not suddenly appear to its earliest users as an entirely new language with its own rules; instead we continually had to adapt existing programs as the language developed, and make allowance for an existing body of code.

Ken Thompson, Dennis Ritchie
“The Development of the C Language” 1993
Our first main task was to carry out resonance absorption calculations related to the construction of Norway's first nuclear reactor. Monte Carlo simulation methods were successfully introduced instead in 1949-1950. The necessity of using simulation and the need of a language for system description was the direct stimulus for SIMULA.

When writing simulation programs we had observed that processes often shared a number of common properties, both in data attributes and actions, but were structurally different in other respects so that they had to be described by separate declarations. Such partial similarity fairly often applied to processes in different simulation models, indicating that programming effort could be saved by somehow preprogramming the common properties.
[Simula’s] class concept allowed me to map my application concepts into the language constructs in a direct way. The way Simula classes can act as coroutines made the inherent concurrency of my application easy to express.

The implementation of Simula, however, did not scale in the same way and as a result the whole project came close to disaster. The cost arose from several language features and their interactions: run-time type checking, guaranteed initialization of variables, concurrency support, and garbage collection of both user-allocated objects and procedure activation records.

Bjarne Stroustrup
The team originally considered using C++, but rejected it for several reasons. They decided that C++‘s complexity led to developer errors. The language's lack of garbage collection meant that programmers had to manually manage system memory, a challenging and error-prone task. The team also worried about the C++ language's lack of portable facilities for security, distributed programming, and threading. Finally, they wanted a platform that would port easily to all types of devices.

– Wikipedia??
My initial goal for Python was to serve as a second language for people who were C or C++ programmers, but who had work where writing a C program was just not effective.

Maybe it was something you'd do only once. It was the sort of thing you'd prefer to write a shell script for, but when you got into the writing details, you found that the shell was not the ideal language— you needed more data structures, more namespaces, or maybe more performance. The first sound bite I had for Python was, "Bridge the gap between the shell and C."
Turing languages

1957 – FORTRAN
1959 – ALGOL
1962 – SIMULA

1972 – C
1979 – C++

1991 – Python
1995 – Java
My own research in artificial intelligence [in 1958]… involved representing information about the world by sentences in a suitable formal language and a reasoning program that would decide what to do by making logical inferences. Representing sentences by list structure seemed appropriate and a list-processing language also seemed appropriate for programming the operations involved in deduction.

…One needs a notation for functions, and it seemed natural to use the λ-notation of Church (1941). I didn't understand the rest of his book, so I wasn't tempted to try to implement his more general mechanism for defining functions.
The languages people use to communicate with computers differ in their intended aptitudes, towards either a particular application area, or a particular phase of computer use (high level programming, program assembly, job scheduling, etc). The question arises, do the idiosyncrasies reflect basic logical properties of the situations that are being catered for? Or are they accidents of history and personal background that may be obscuring fruitful developments?

ISWIM is an attempt at a general purpose system for describing things in terms of other things, that can be problem-oriented by appropriate choice of "primitives." A possible first step in the research program is 1700 doctoral theses called "A Correspondence between x and Church's λ-notation."

Peter Landin, "The Next 700 Programming Languages" 1966
The principal aims in designing ML were to make it impossible to prove non-theorems yet easy to program strategies for performing proofs.

A strategy—or recipe for proof—could be something like “induction on \( f \) and \( g \), followed by assuming antecedents and doing case analysis, all interleaved with simplification”. This is imprecise—analysis of what cases? what kind of induction, etc, etc.—but these in turn may well be given by further recipes, still in the same style.

Robin Milner, “A Metalanguage for Interactive Proof in LCF” 1978
The simplicity and elegance of functional programming captivated the present authors. Lazy evaluation— with its direct connection to the pure, call-by-name lambda calculus, the remarkable possibility of representing and manipulating infinite data structures, and addictively simple and beautiful implementation techniques—was like a drug.
Church languages

1959 — LISP
1966 — ISWIM
1978 — ML
1990 — Haskell

Turing languages

1957 — FORTRAN
1959 — COBOL, ALGOL
1962 — SIMULA

1972 — C, Smalltalk
1979 — C++
1991 — Python
1995 — Java

1972 — C, Smalltalk
1972 — Prolog
1978 — ML
1990 — Haskell
Church languages are intimidating

```
Couldn't match type `k0' with `b'
because type variable `b' would escape its scope
This (rigid, skolem) type variable is bound by
the type signature for
    groupBy :: Ord b => (a -> b) -> Set a -> Set (b, [a])
The following variables have types that mention k0
```
PL theory can be really dense

\[ \Sigma_0; \Gamma_1 \vdash A = D \bar{v} : \text{Set}_n \quad \Gamma = \Gamma_1(x : A)\Gamma_2 \]

Data \( D \Delta : \text{Set}_n \) where \( c_i \Delta_i \bar{j}_i \mid b_i \in \Sigma_0 \quad \exists k. [\bar{j}_k \mid b_k] \neq [] \)

\[
\begin{align*}
\Delta'_i &= \Delta_i(\bar{j}_i : \mathbb{I})[\bar{v} / \Delta] \\
\bar{q}_i &= \hat{\Delta}_i[\bar{j}_i \mid b_i][\bar{v} / \Delta] \\
\rho_i &= \mathbb{I}_{\Gamma_1} \uplus [c_i \bar{q}_i / x] \\
\rho'_i &= \rho_i \uplus \mathbb{I}_{\Gamma_2} \\
\Theta_i &= \text{BOUNDARY}(\bar{j}_i); \Theta \\
\Sigma_{i-1}; \Gamma_1 \Delta'_i(\Gamma_2 \rho_i) \vdash f \bar{q} \rho'_i := Q_i : C \rho'_i \mid \Theta \sim \Sigma_i \quad i = 1 \ldots n
\end{align*}
\]

\( \Delta_{hc} = (r : \mathbb{I})(u : \mathbb{I} \rightarrow \text{Partial } r (D \bar{v}))(u_0 : D \bar{v} [r \mapsto u \ i0]) \)

\[
\begin{align*}
\rho_{hc} &= \mathbb{I}_{\Gamma_1} \uplus [\text{hcomp } r u u_0 / x] \\
\rho'_{hc} &= \rho_{hc} \uplus \mathbb{I}_{\Gamma_2} \\
\Sigma_n; \Gamma_1 \Delta_{hc}(\Gamma_2 \rho_{hc}) &\vdash f \bar{q} \rho'_{hc} := Q_{hc} : C \rho'_{hc} \mid (r = 1); \Theta \sim \Sigma_{n+1}
\end{align*}
\]

\[ \Sigma_0; \Gamma \vdash f \bar{q} := \text{case}_x \left\{ \begin{array}{c}
c_1 \bar{q}_1 \mapsto Q_1; \ldots; c_n \bar{q}_n \mapsto Q_n \\
\text{hcomp } r u u_0 \mapsto Q_{hc}
\end{array} \right\} : C \mid \Theta \sim \Sigma_{n+1} \]

Vezzosi et al. "Cubical Agda: A Dependently Typed Programming Language with Univalence and Higher Inductive Types" ICFP '19
Thesis:
Ideas from Church languages will radically improve how we 1) think, 2) communicate, and 3) program in software design.
Corollary:
Learning these ideas gives you a programming superpower to make correct and reliable software way more productively.
Von Neumann languages (e.g., the FORTRANs, the ALGOLs) constantly keep our noses pressed in the dirt of the separate computation of single words, whereas we should be focusing on the form and content of the overall result. We regard the DO, FOR, WHILE statements and the like as powerful tools, whereas they are in fact weak palliatives that are necessary to make the primitive von Neumann style of programming viable at all.

… While it was perhaps natural and inevitable that languages like FORTRAN and its successors developed out of the von Neumann computer, that such languages have dominated our thinking for twenty years is unfortunate. Their long-standing familiarity will make it hard for us to understand and adopt new programming styles which one day will offer far greater intellectual and computational power.

fun innerproduct(a, b, n):
  c := 0
  for i := 1 step 1 until n do
    c := c + a[i] * b[i]
  return c

• Statements operate on invisible state
• Computes word-at-a-time by repetition of assignment/modification
• Requires names for arguments, iterator, return value

let innerproduct = zip |> (map *) |> (reduce +)

• Built from composable functions (map, reduce, pipe)
• Operates on whole conceptual units (lists), no repeated steps
• No names for arguments or temporaries
Modular design is the key to successful programming. When writing a modular program to solve a problem, one first divides the problem into subproblems, then solves the subproblems, and finally combines the solutions.

The ways in which one can divide up the original problem depend directly on the ways in which one can glue solutions together. Therefore, to increase one’s ability to modularize a problem conceptually, one must provide new kinds of glue in the programming language.

fun innerproduct(a, b, n):
  c := 0
  for i := 1 step 1 until n do
    c := c + a[i] * b[i]
  return c

let innerproduct = zip |> (map *) |> (reduce +)
Python

```python
# Any * Any -> Any
def innerproduct(a, b): ...
```

C

```c
typedef struct {
  float* arr; ...
} vec_t;
float innerproduct(vec_t a, vec_t b)
```

Java

```java
public class Vec<T> {
  public T innerproduct(Vec<T> other);
}
```

Idris

```
innerproduct : Vec n a -> Vec n a -> a
```
Church languages are coming!

- Java
- Objective-C
- C++
- JavaScript
- R
- Python

→ Scala +

→ Swift

→ TypeScript

→ Julia

(with types)
Companies are realizing FP benefits

Messenger used to receive bugs reports on a daily basis; since the introduction of Reason, there have been a total of 10 bugs (that's during the whole year, not per week)! Refactoring speed went from days to hours to dozens of minutes.

  – “Messenger.com Now 50% Converted to Reason” 2017

Being able to encode constraints of your application in the type system makes it possible to refactor, modify, or replace large swaths of code with confidence. Rust's error model forces developers to handle every corner case. [Our system] needs very little attention. We were able to leave it running without any issues through the holiday break.

  – “Rust at OneSignal” 2017
Formal verification is around the corner

val aes_ctr_encrypt_bytes:
  v:variant
  → key:aes_key v
  → n_len:size_nat{n_len <= 16}
  → nonce:lbytes n_len
  → c:size_nat
  → msg:bytes{length msg / 16 + c <= max_size_t} →
  Tot (ciphertext:bytes{length ciphertext == length msg})
The software world is ready for better languages
- The languages we use today have barely changed in 60 years
- Developer spending 5% time writing code
- Huge, buggy, slow software ecosystems

Church languages focus on mathematics of programming
- Advanced type systems and functional features promote modularity, specification, and verification
- Not mainstream due to inefficiencies, lack of application focus

The future is functional
- Better compilers, better understanding of which concepts matter in practice, better awareness are moving Church languages to mainstream
Key concepts and skills: theory

• Formal representation of and reasoning about programs
  - Descriptions of syntax and semantics of languages
  - Mathematical specification of program behavior (type systems)
  - Inductive proofs about programs and languages
  - Formal model of both functional and imperative languages

• You should be able to…
  - Learn more advanced PL theory
  - Read basic PL theory papers (and understand them)
  - Understand the essence of basic computational concepts
Key concepts and skills: practice

• Functional programming techniques
  - Explicit state-passing with expression-oriented programming
  - Abstract patterns with higher-order functions
  - Error handling and inductive data with algebraic data types
  - Generic/modular programming with polymorphism and traits
  - Resource management/concurrency with linear types
  - Verified communication with session types

• You should be able to…
  - Structure low-level code to avoid large classes of bugs
  - Learn new functional programming languages more effectively
  - Recognize opportunities for Church programming in Turing languages
    • “Will, I feel guilty now every time I write a for loop” – CURIS student in my office for the summer
Course prerequisites

• **Theory: CS 103**
  - First-order logic, induction, discrete math
  - Strongly encouraged

• **Systems: CS 107 + 110**
  - 107 absolutely
  - 110 for a bit of the multithreading
Course logistics

• Weekly assignments
  - Get started early!
  - Mixture of written problems and programming problems
  - Expect 15 hr/week including lecture

• No midterm, take-home final exam
  - Assignments are more work to compensate

• Everything available on course website
  - cs242.stanford.edu
  - Including detailed lecture notes and slides
Attending lectures

• Yes, everything is recorded through SCPD
  - Yes, you can watch everything from your sofa

• I still recommend you come to class!
  - Put yourself in a learning environment
  - I will always give time for thought and questions

• …But no laptops
  - Don’t distract people if you’re going to attend lectures