Telescoping Languages

Domain Specific Languages for the Price of C (or Fortran)

Arun Chauhan

Indiana University

Collaborators

- **Ken Kennedy**
- Bradley Broom
- Keith Cooper
- Rob Fowler
- John Garvin
- Chuck Koelbel
- Cheryl McCosh
- John Mellor-Crummey
- Linda Torczon

A True Story

- The world of Digital Signal Processing
  - almost everyone uses MATLAB
  - a large number uses MATLAB exclusively
  - almost everyone hates writing C code
  - readily tradeoff running time for development time
  - often forced to rewrite programs in C

A True Story

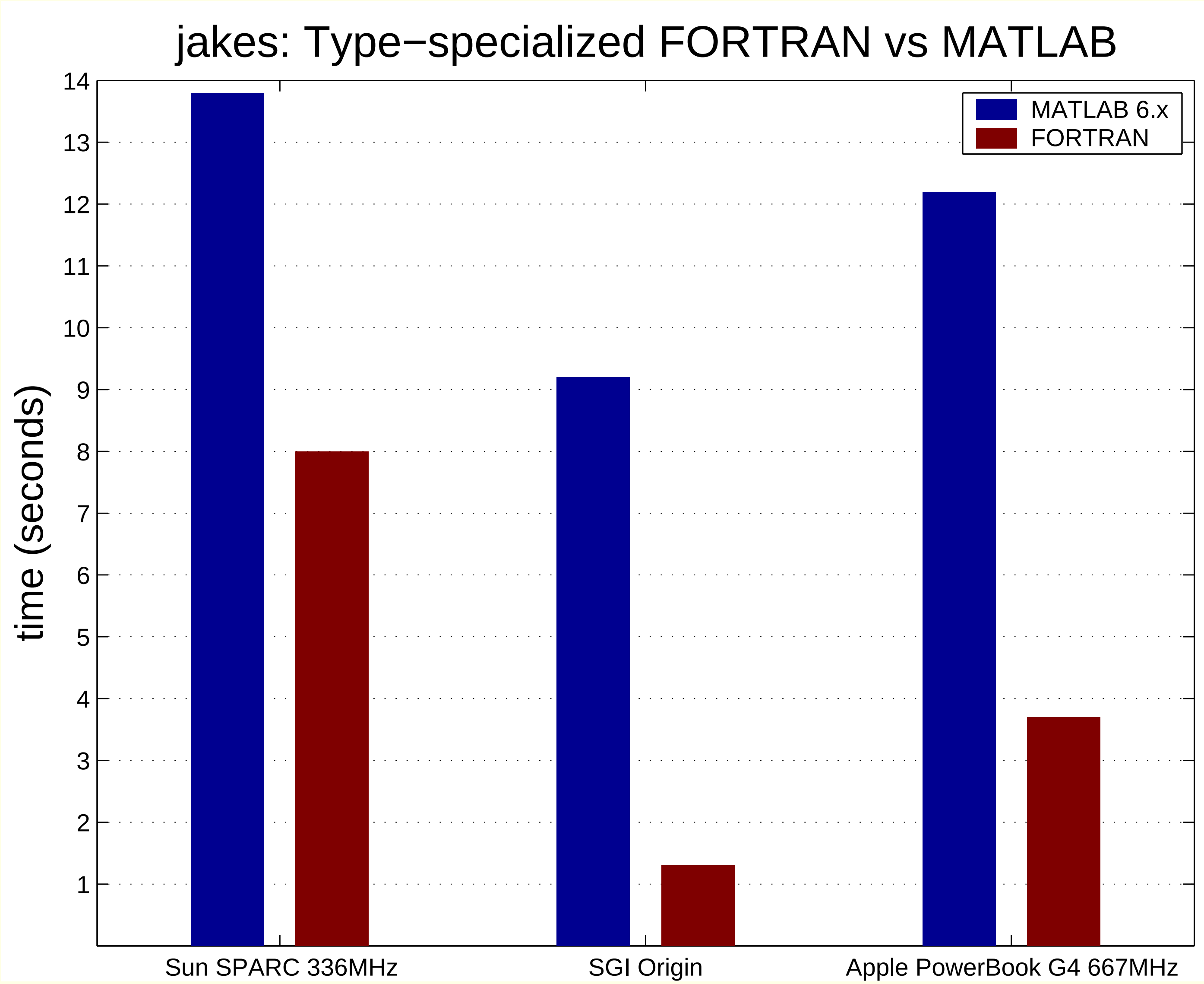
- The world of Digital Signal Processing
  - almost everyone uses MATLAB
  - a large number uses MATLAB exclusively
  - almost everyone hates writing C code
  - readily tradeoff running time for development time
  - often forced to rewrite programs in C
- Linear algebra through MATLAB
  - ARPACK prototyped in MATLAB
  - recoded in Fortran for performance

A True Story

- The world of Digital Signal Processing
  - almost everyone uses MATLAB
  - a large number uses MATLAB exclusively
  - almost everyone hates writing C code
  - readily tradeoff running time for development time
  - often forced to rewrite programs in C
- Linear algebra through MATLAB
  - ARPACK prototyped in MATLAB
  - recoded in Fortran for performance

The productivity connection

The Performance Gap



It’s the Compilers

“We did not regard language design as a difficult problem, merely a simple prelude to the real problem: designing a compiler that could produce efficient programs.”

–John Backus, the “Father of Fortran”

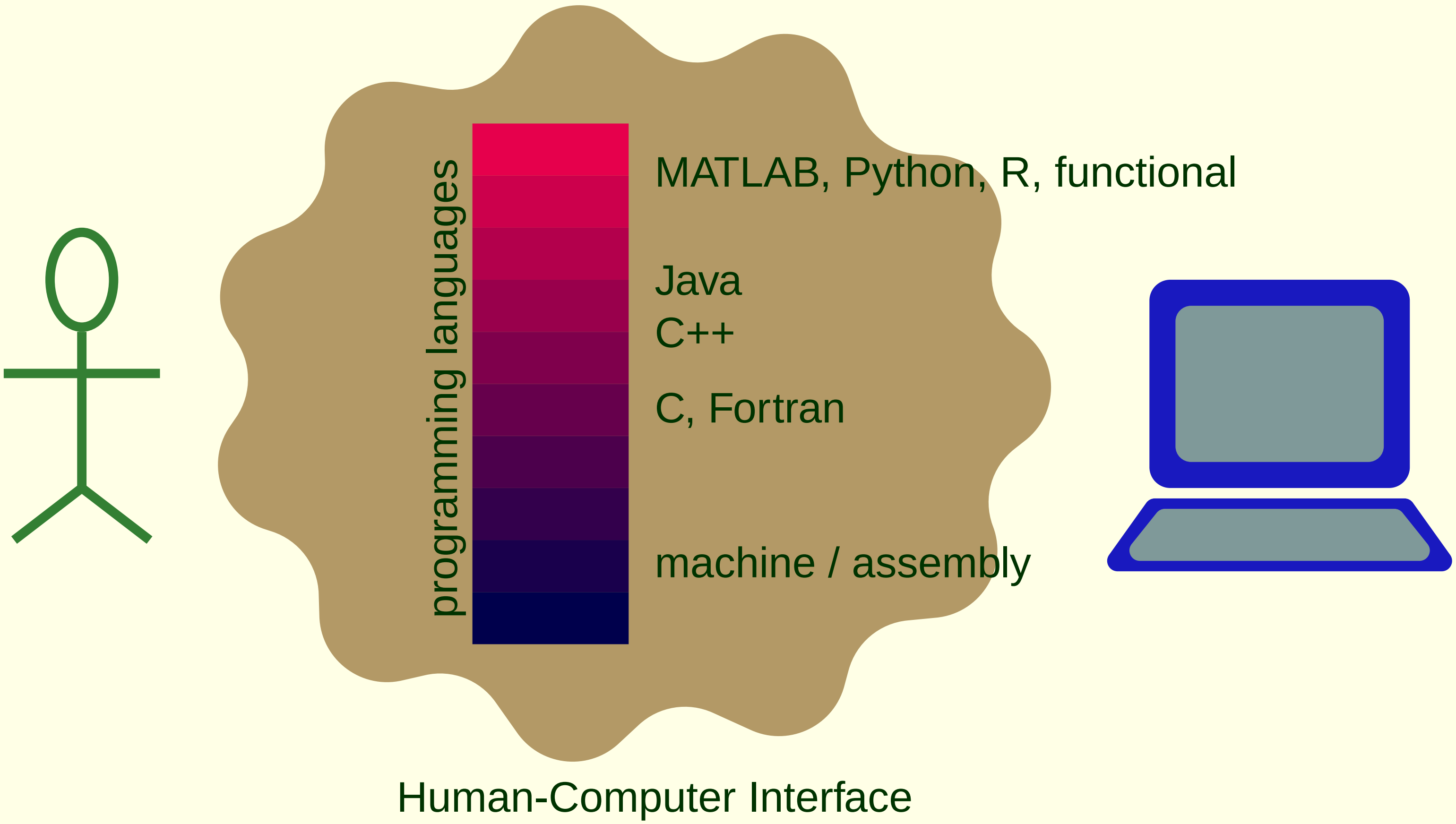
It’s the Compilers

“We did not regard language design as a difficult problem, merely a simple prelude to the real problem: designing a compiler that could produce efficient programs.”

–John Backus, the “Father of Fortran”

Effective and **Efficient** compilation

The Big Picture



Fundamental Observation

- Libraries are the key in optimizing high-level scripting languages

`a = x * y` $\Rightarrow$ `a = mclMtimes(x, y)`

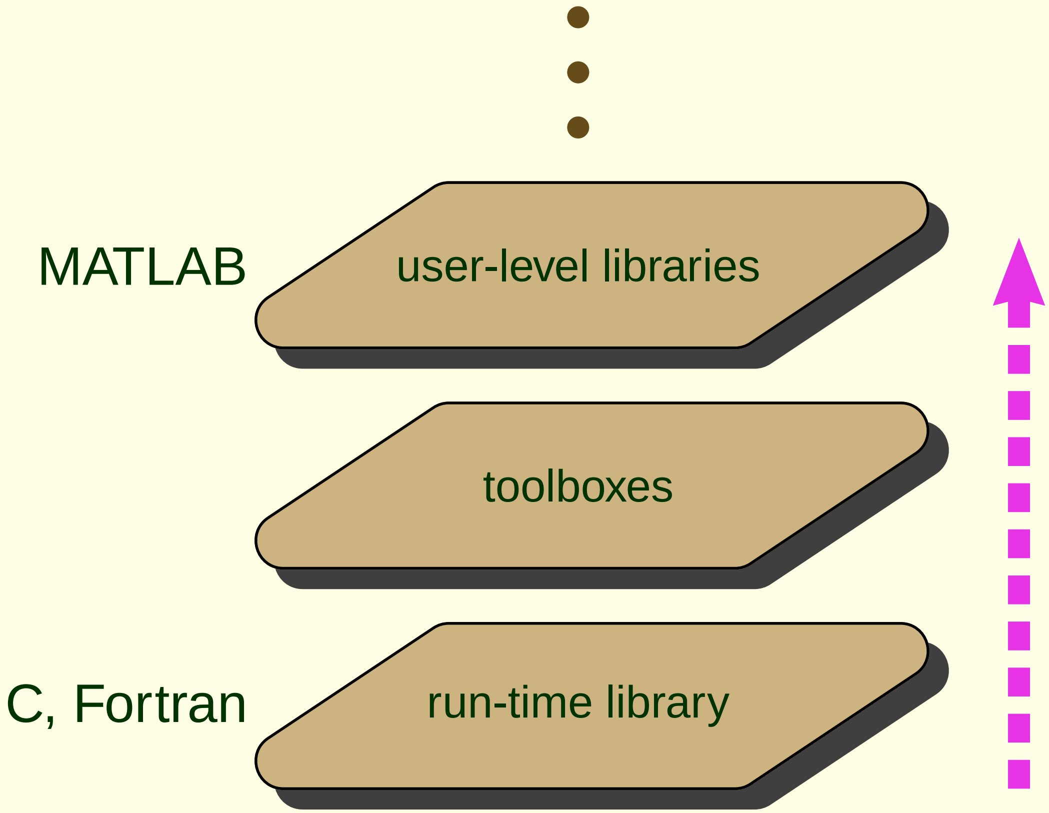
Fundamental Observation

- Libraries are the key in optimizing high-level scripting languages

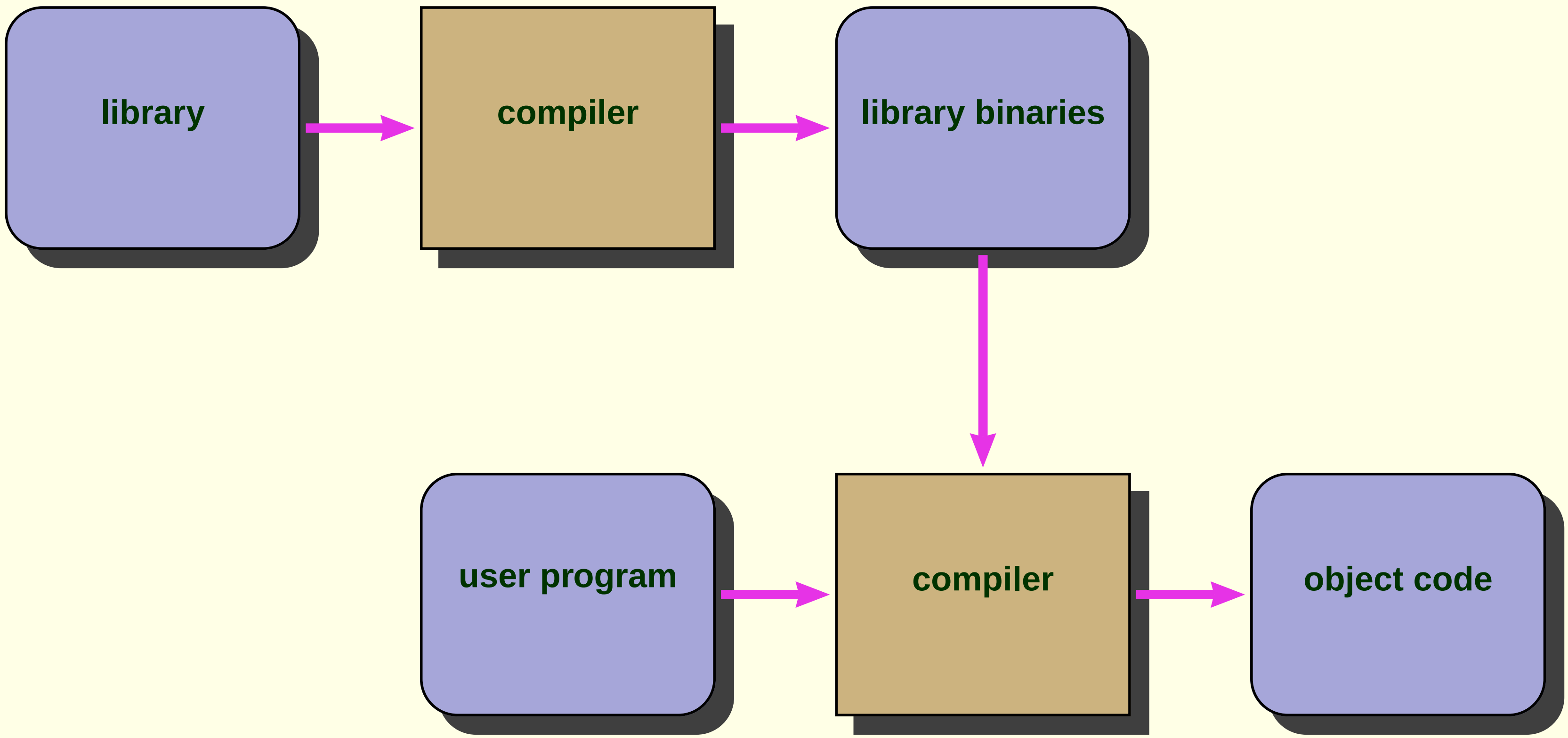
```
a = x * y  ⇒  a = mclMtimes(x, y)
```

- Libraries practically **define** high-level scripting languages
  - high-level operations are often “syntactic sugar”
    - runtime libraries implement operations
  - a large effort in HPC is toward writing libraries
  - domain-specific libraries make scripting languages useful and popular

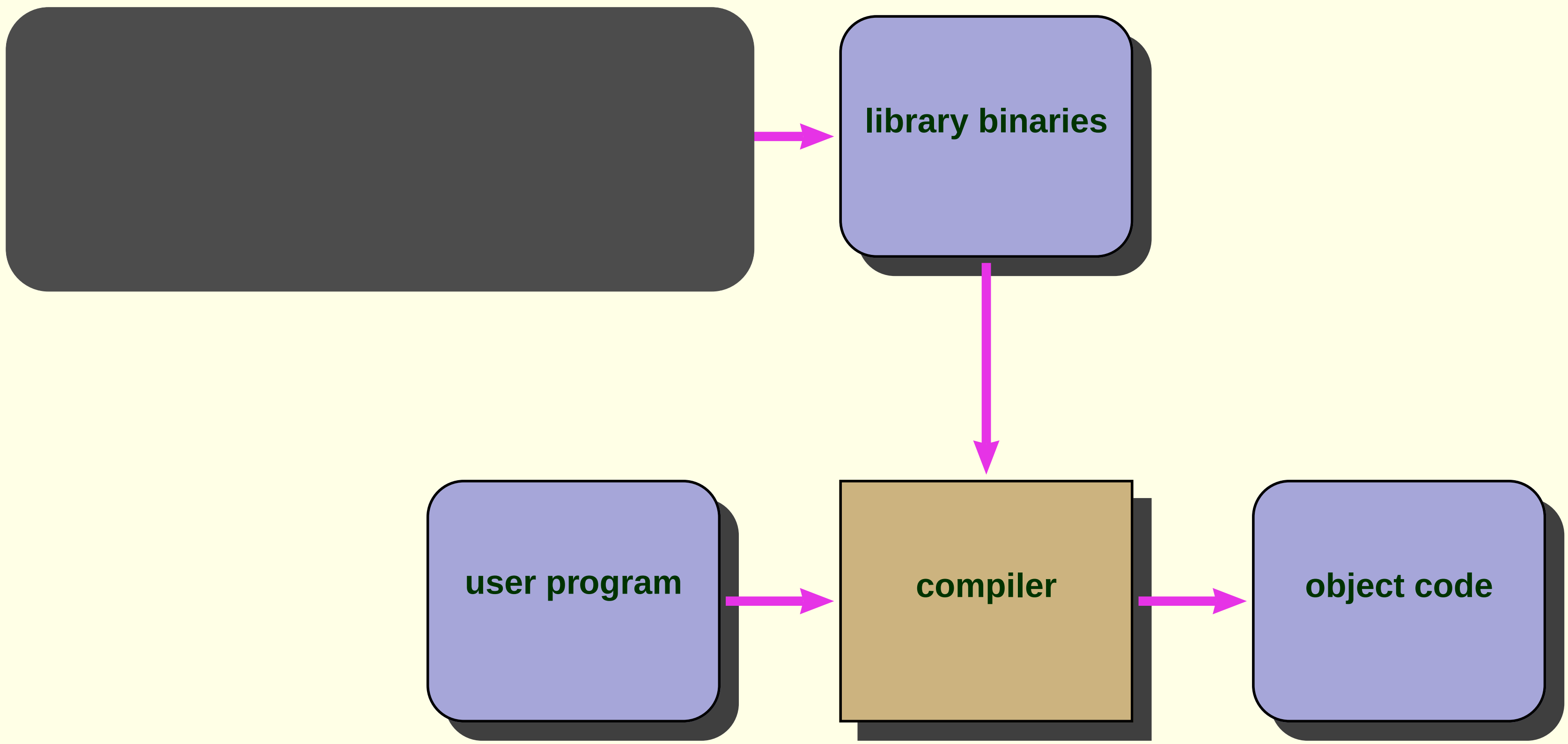
Hierarchy of Libraries



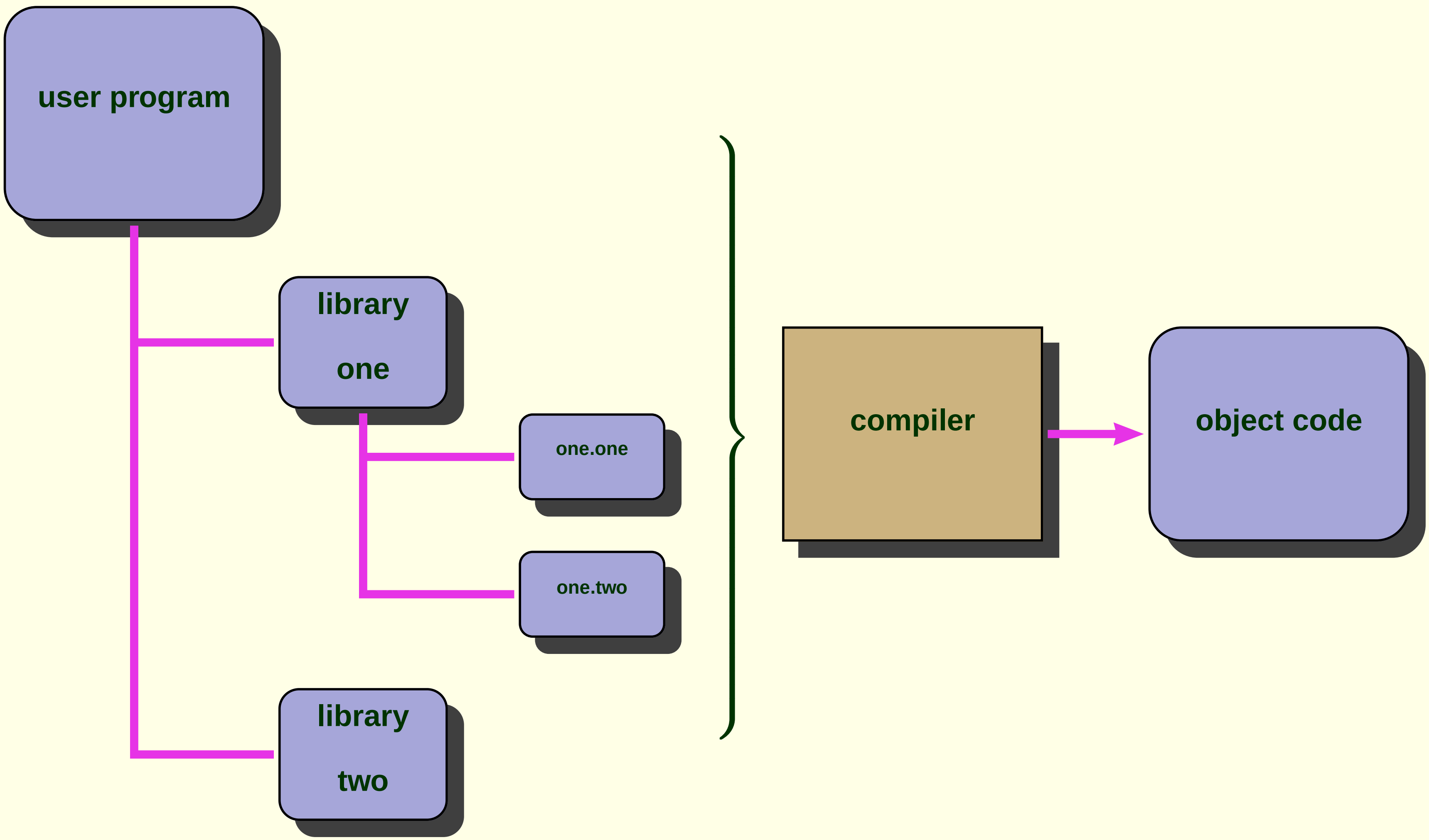
Libraries as Black Boxes



Libraries as Black Boxes

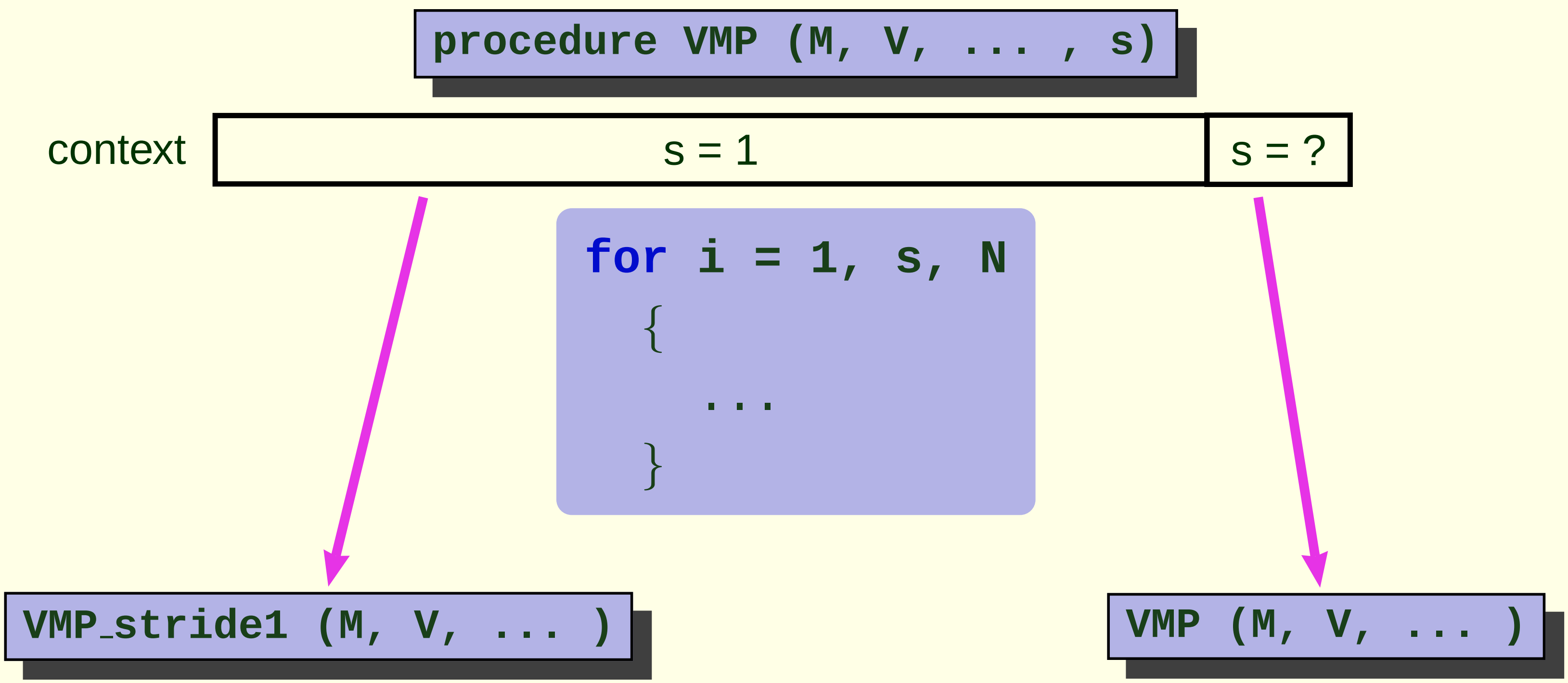


Whole Program Compilation



Motivating Example

- Specialization
- Speculate contexts
  - utilize library writers' specialized knowledge



Telescoping Languages: Entities

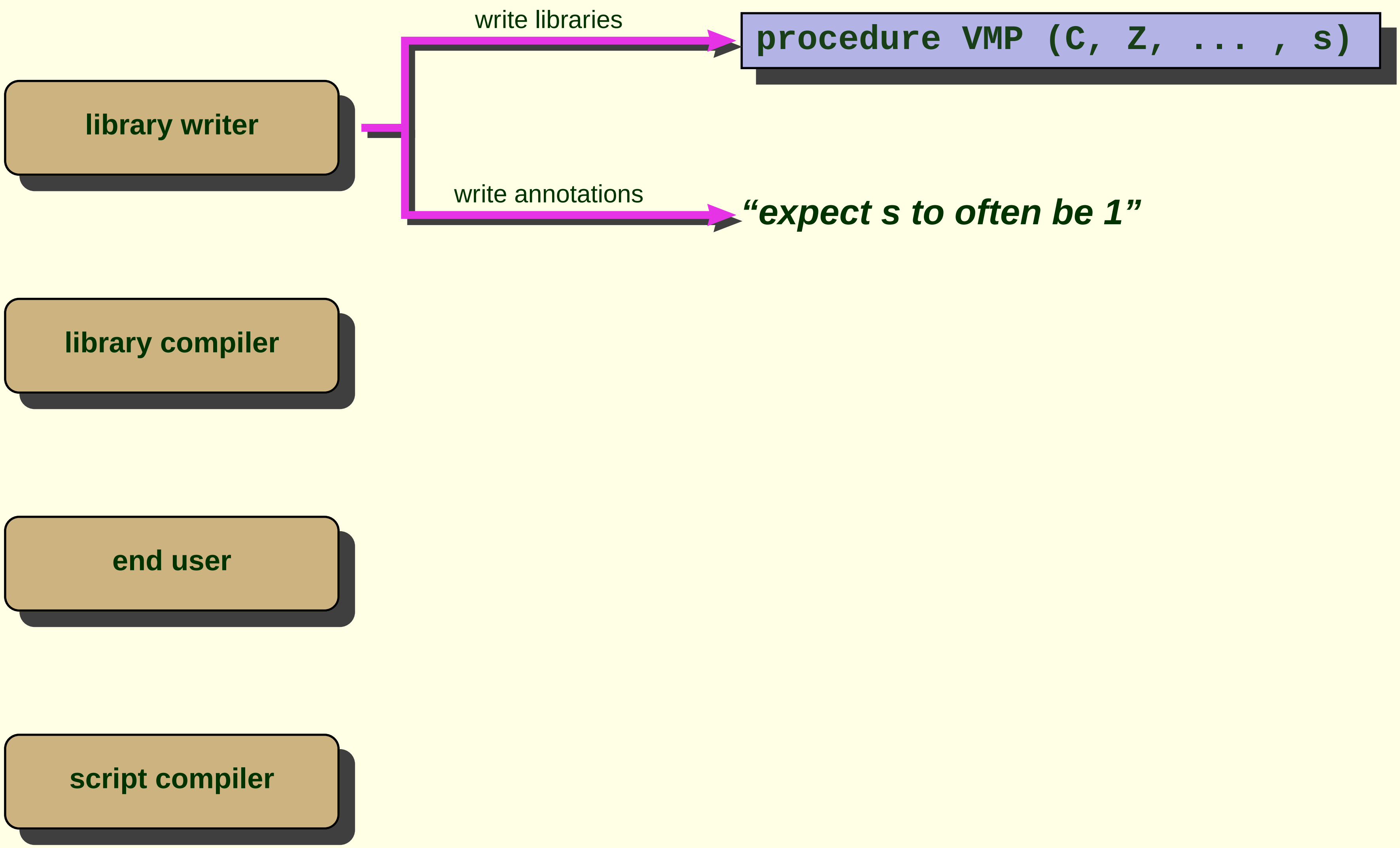
library writer

library compiler

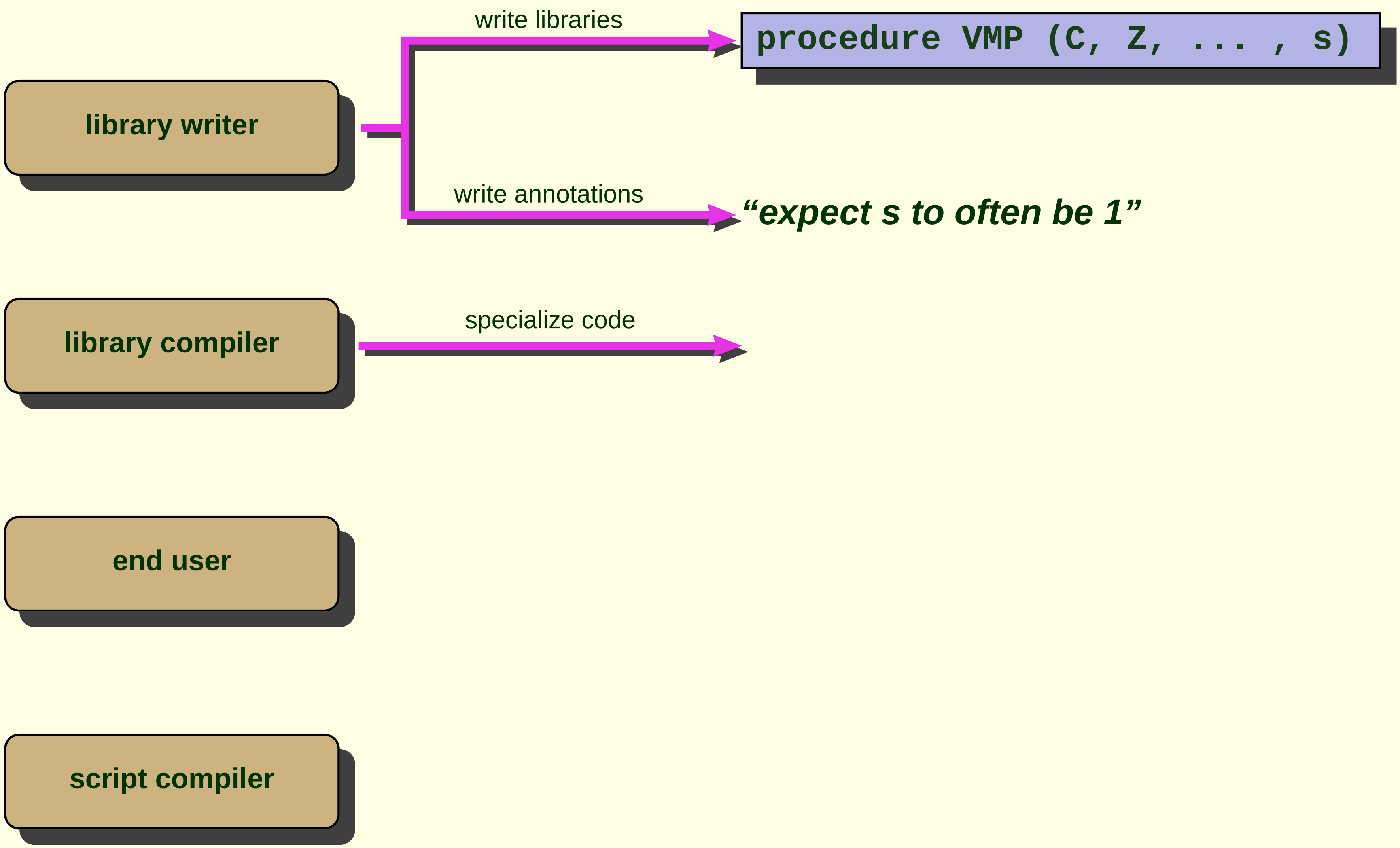
end user

script compiler

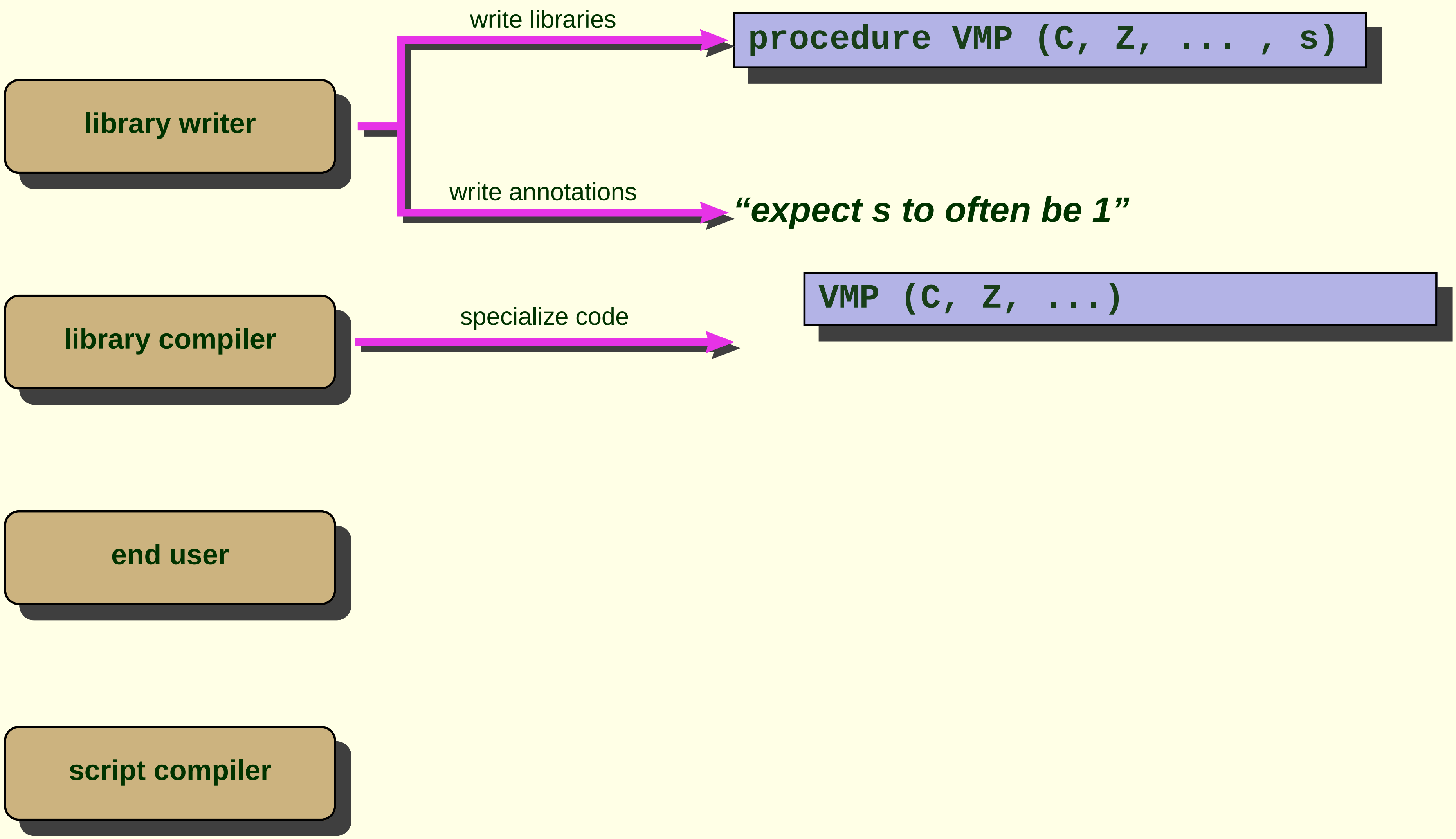
Telescoping Languages: Entities



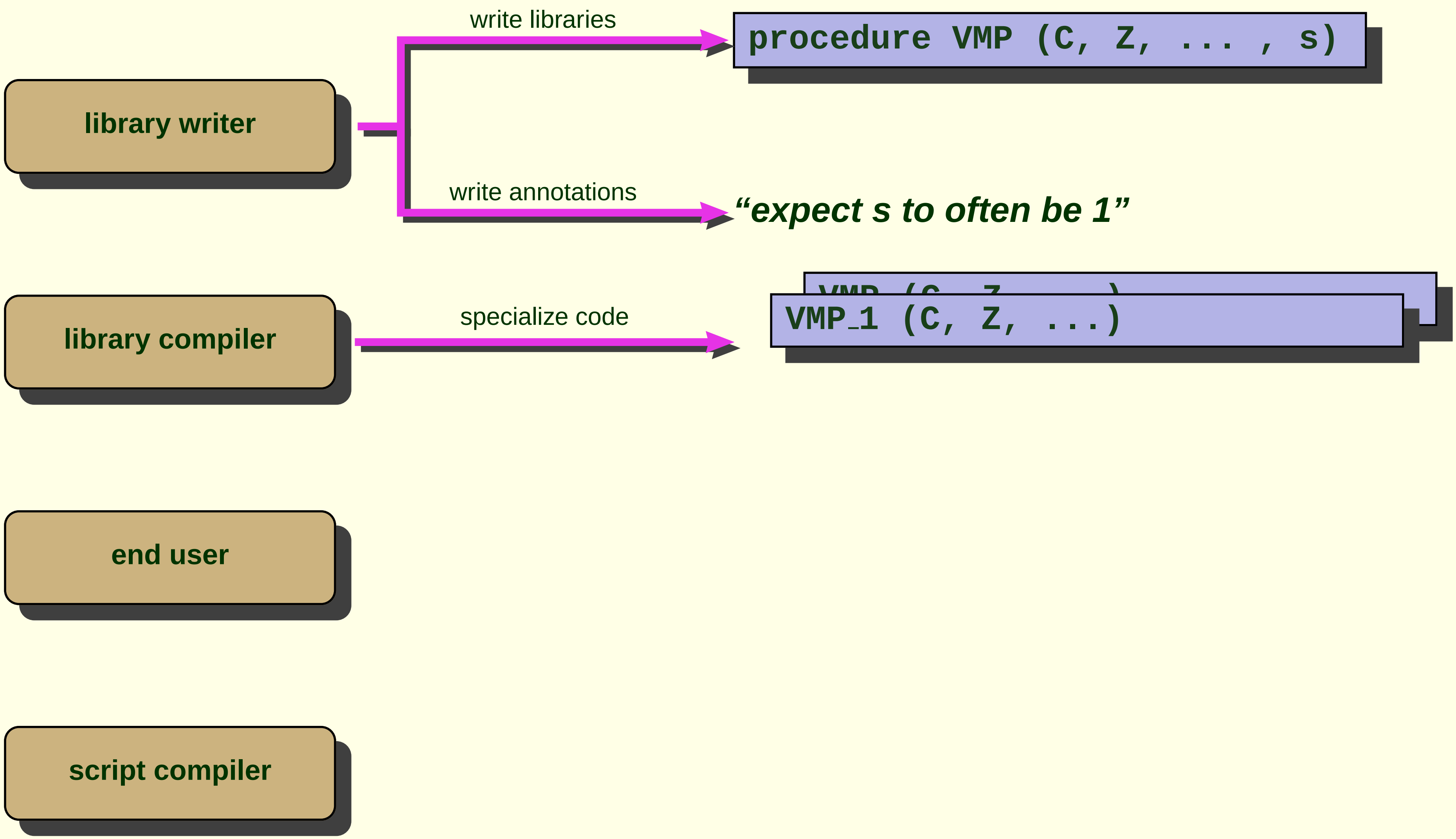
Telescoping Languages: Entities



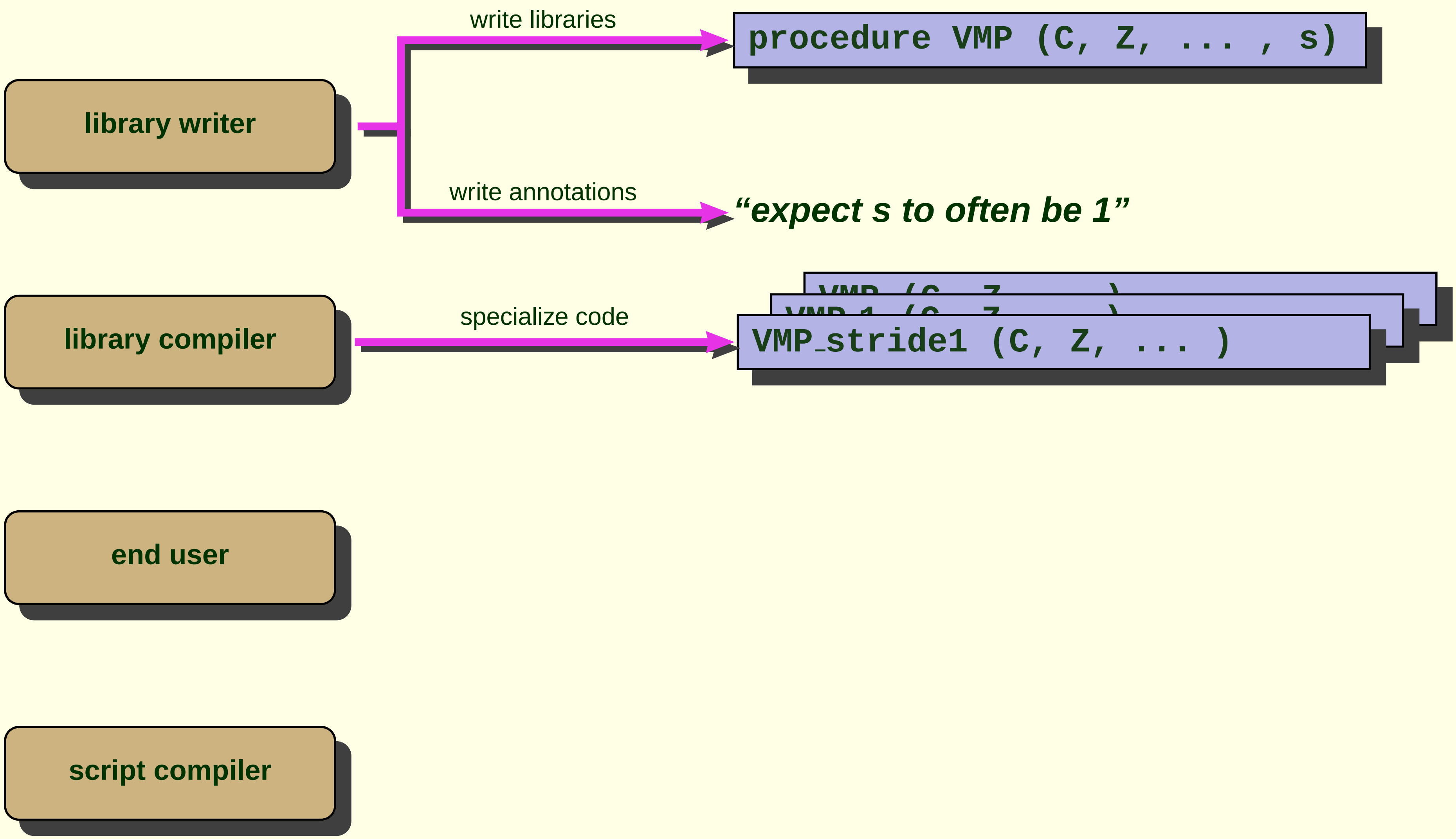
Telescoping Languages: Entities



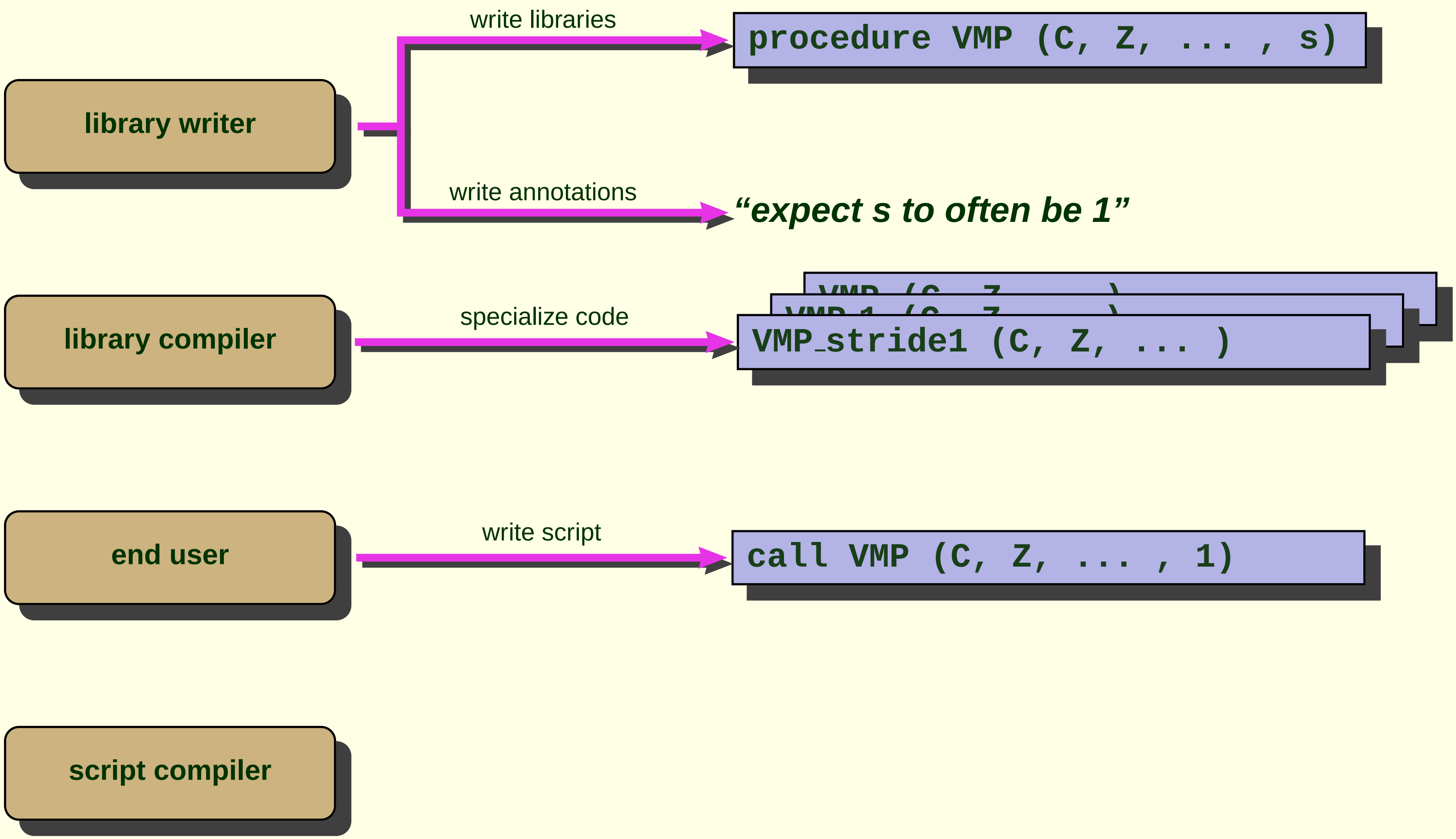
Telescoping Languages: Entities



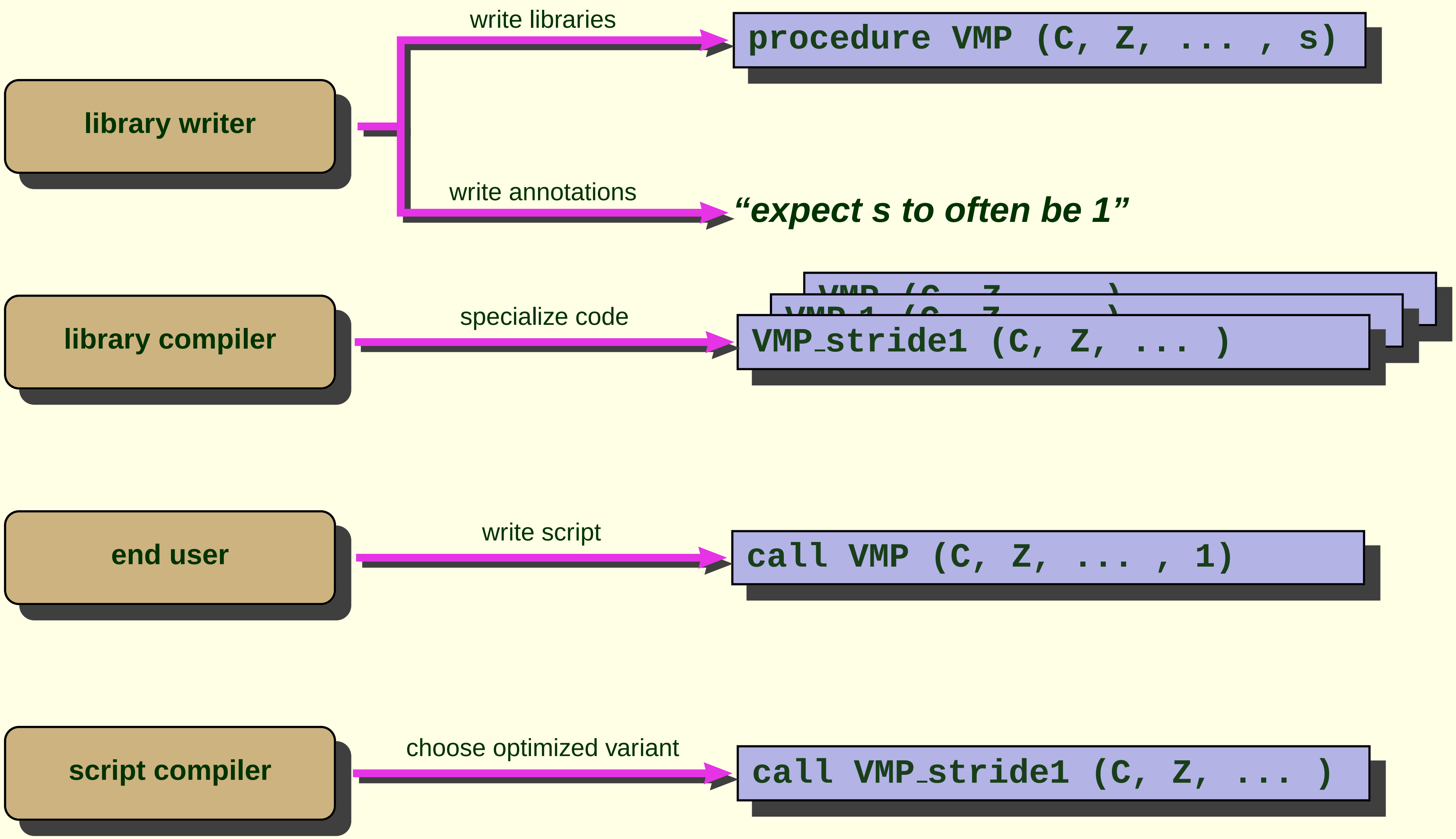
Telescoping Languages: Entities



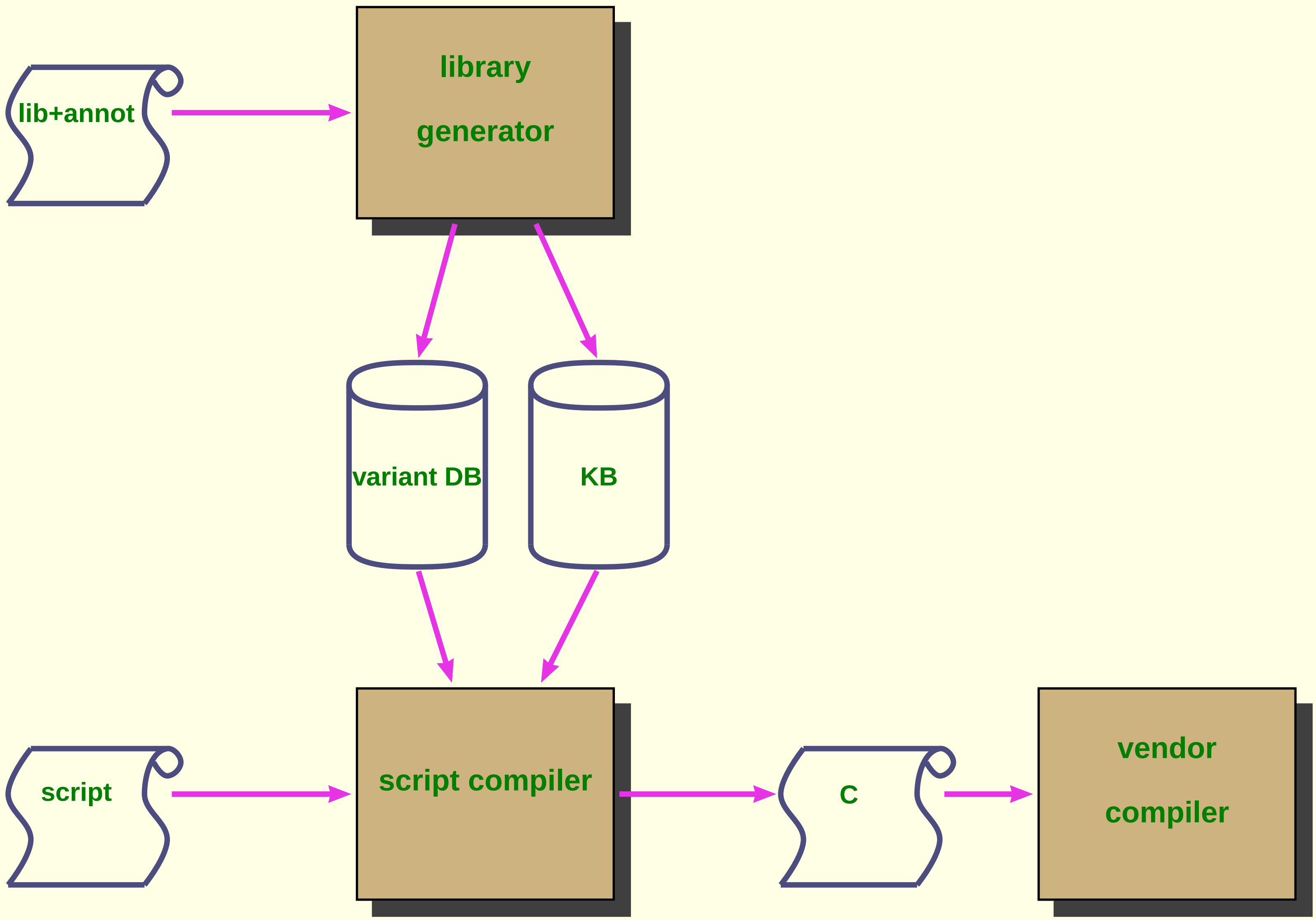
Telescoping Languages: Entities



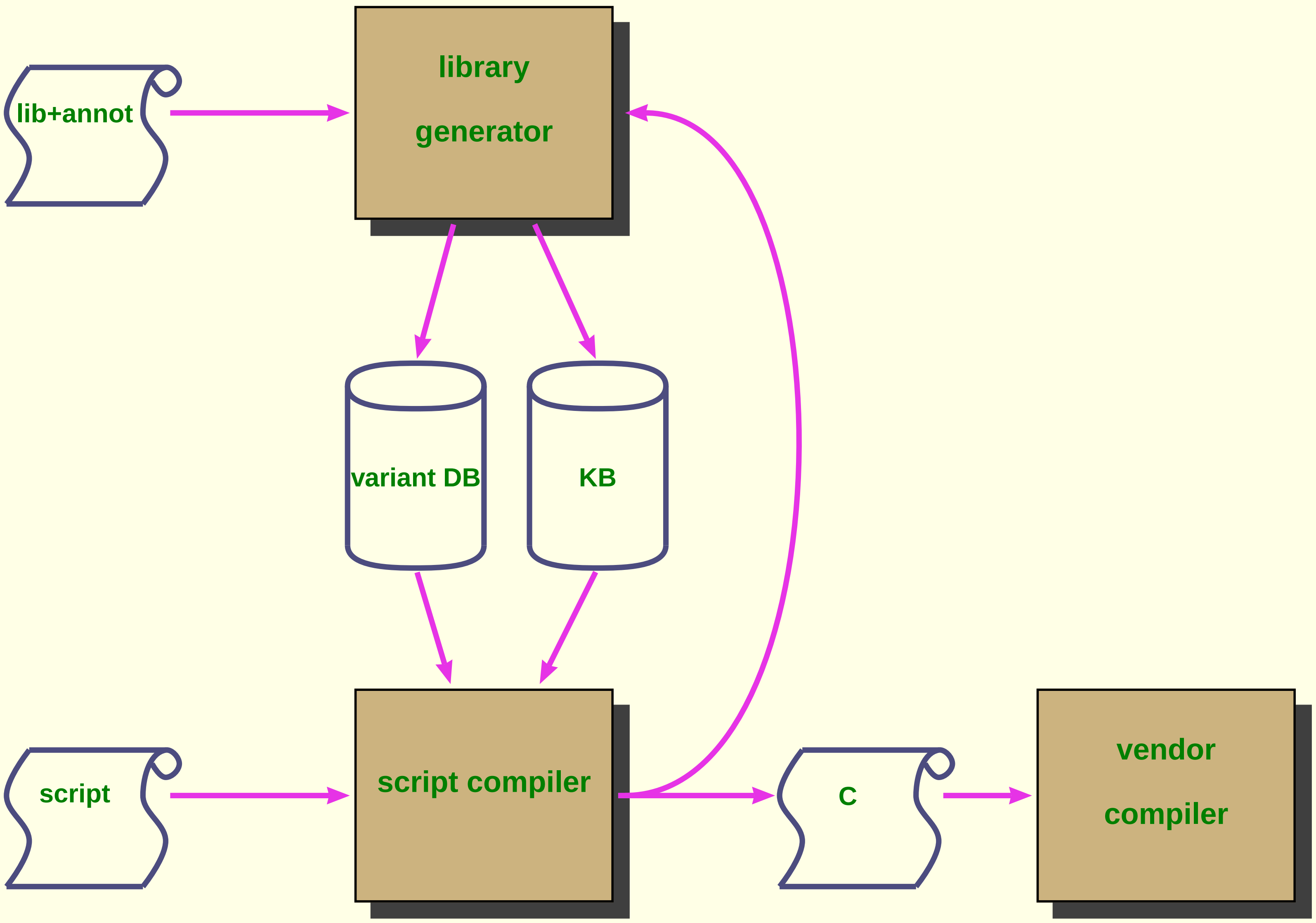
Telescoping Languages: Entities



Overall Telescoping System



Overall Telescoping System



Telescoping Languages Approach

- Pre-compile libraries to minimize end-user compilation time
- Annotate libraries to capture specialized knowledge of library writers
- Generate specialized variants based on interesting contexts
- Link appropriate versions into the user script

Telescoping Languages Approach

- Pre-compile libraries to minimize end-user compilation time
- Annotate libraries to capture specialized knowledge of library writers
- Generate specialized variants based on interesting contexts
- Link appropriate versions into the user script

analogous to offline indexing by search engines

Library Compiler: Some Issues

- Dealing with high-level scripting languages
  - parsing and analyzing a library procedure written in a scripting language
  - translating into an intermediate language (C, Fortran)
- High-level transformations
  - identifying useful transformations
- Enabling the library writer to express library properties
  - stating facts about library procedures
  - describing specializations

Inferring Types

- type $\equiv <\tau, \delta, \sigma, \psi>$
  - $\tau =$ intrinsic type, e.g., int, real, complex, etc.
  - $\delta =$ array dimensionality, 0 for scalars
  - $\sigma = \delta$-tuple of positive integers
  - $\psi =$ "structure" of an array
- Examples
  - x is scalar, integer

    $\Rightarrow$ type of x $= <$int, 0, $\perp$, $\perp>$
  - y is 3-D $10 \times 5 \times 20$ dense array of reals

    $\Rightarrow$ type of y $= <$real, 3, $<$10,5,20$>$, dense$>$

Relevant Optimizations

"It is a capital mistake to theorize before one has data. Insensibly one begins to twist facts to suit theories, instead of theories to suit facts."

–Sir Arthur Conon Doyle in a *A Scandal in Bohemia*

Study of DSP Applications

- MATLAB applications from the ECE department
  - real applications being used in the DSP and image processing group
- Looked for high-level transformations
- Discovered
  - two novel procedure-level optimizations
  - relevance of several well known transformation techniques

Procedure Strength Reduction

```
for i = 1:N
    ...
    f (c1, c2, i, c3);
    ...
end
```

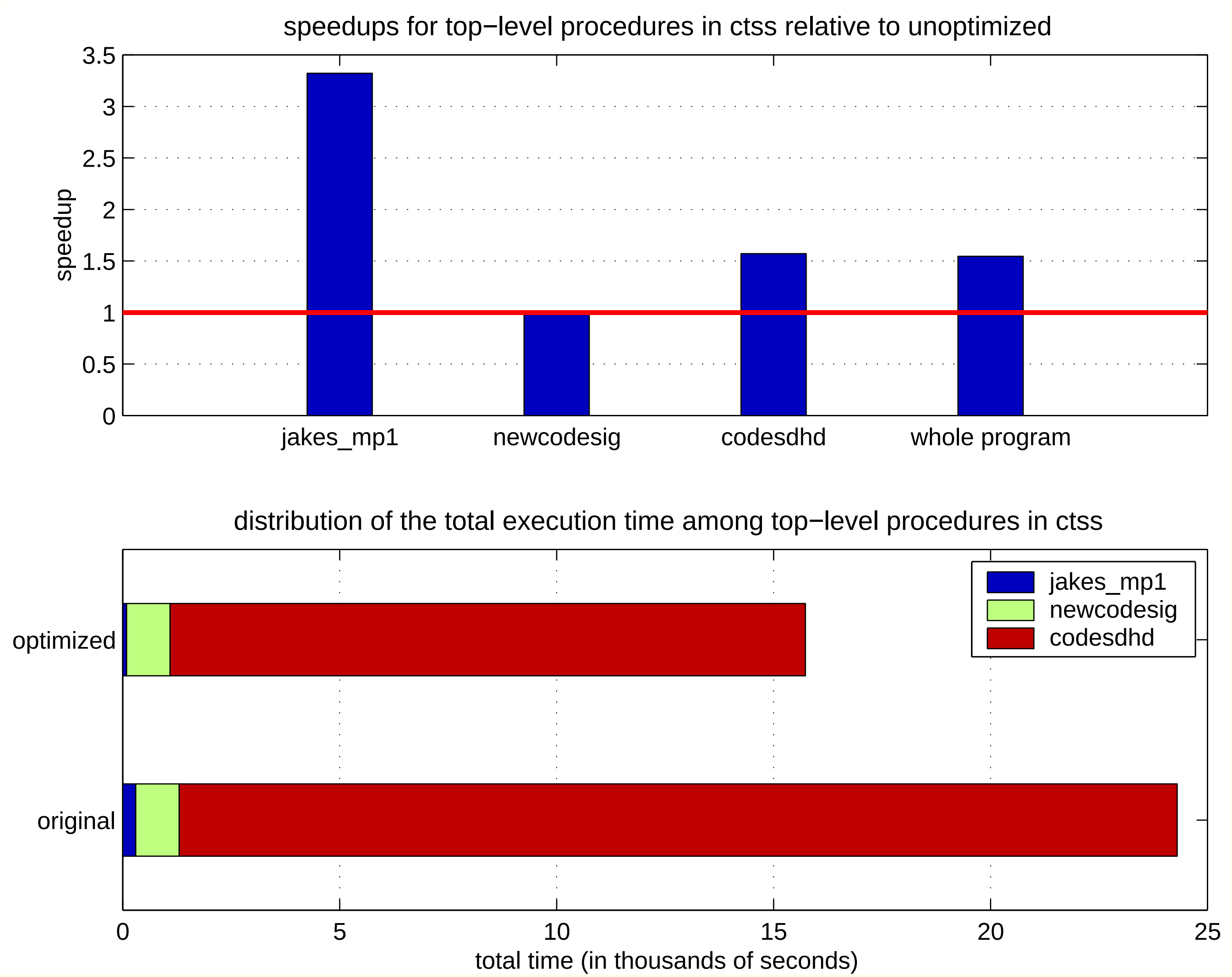
Procedure Strength Reduction

```
for i = 1:N
   ...
   f (c₁, c₂, i, c₃);
   ...
end
```

$\Rightarrow$

```
f_init (c₁, c₂, c₃);
for i = 1:N
   ...
   f_iter (i);
   ...
end
```

Speedup by PSR



Procedure Vectorization

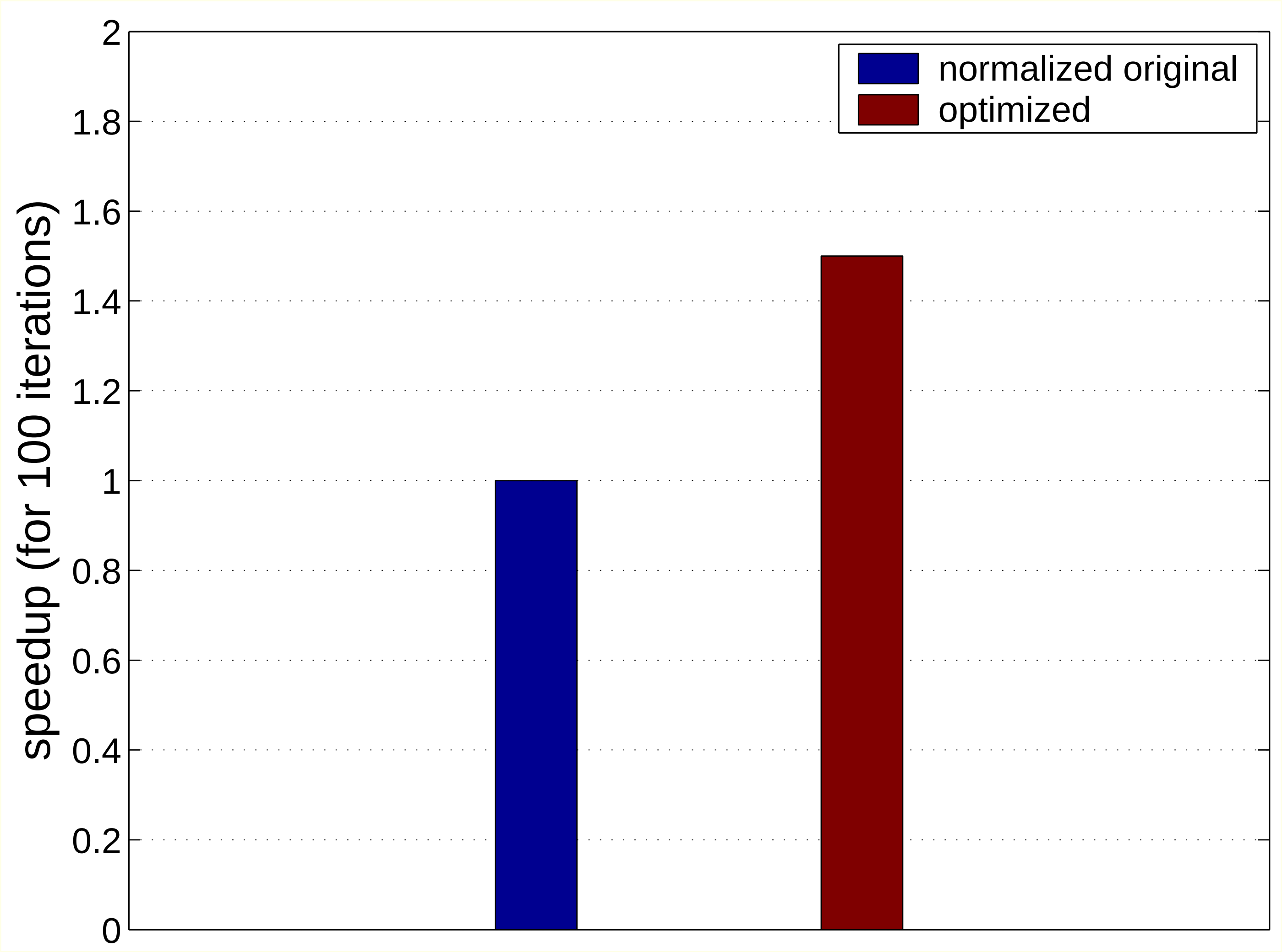
```
for i = 1:N
    α
    f (c₁, c₂, i, A[i]);
    β
end
...
function f (a₁, a₂, a₃, a₄)
    <body of f>
```

Procedure Vectorization

```
for i = 1:N
    α
    f (c1, c2, i, A[i]);
    β
end
...
function f (a1, a2, a3, a4)
    <body of f>
```

```
for i = 1:N
    α
end
f_vect (c1, c2, [1:N], A)
for i = 1:N
    β
end
...
function f_vect (a1, a2, a3, a4)
    for i = 1:N
        <body of f>
    end
```

Applying to `jakes`



High-payoff Optimizations

- Loop vectorization
- Library identities
- Common subexpression elimination
- Beating and dragging along
- Constant propagation

High-payoff Optimizations

- **Loop vectorization**
- **Library identities**
- Common subexpression elimination
- Beating and dragging along
- Constant propagation

High-payoff Optimizations

- **Loop vectorization**
- **Library identities**
- Common subexpression elimination
- Beating and dragging along
- Constant propagation

Loop Vectorization

```
function z = jakes_mp1 (blength, speed, bnumber, N_Paths)
....
for k = 1:N_Paths
   ....
   xc = sqrt(2)*cos(omega*t_step*j') ...
      + 2*sum(cos(pi*np/Num_osc).*cos(omega*cos(2*pi*np/N)*t_step.*jp));
   xs = 2*sum(sin(pi*np/Num_osc).*cos(omega*cos(2*pi*np/N)*t_step.*jp));

   % for j = 1 : Num
   %  xc(j) = sqrt(2) * cos (omega * t_step * j);
   %  xs(j) = 0;
   %  for n = 1 : Num_osc
   %     cosine = cos(omega * cos(2 * pi * n / N) * t_step * j);
   %     xc(j) = xc(j) + 2 * cos(pi * n / Num_osc) * cosine;
   %     xs(j) = xs(j) + 2 * sin(pi * n / Num_osc) * cosine;
   %  end
   % end
   ....
end
```

High-payoff Optimizations

- **Loop vectorization**
- **Library identities**
- Common subexpression elimination
- Beating and dragging along
- Constant propagation

Library Identities

```
function [s, r, j_hist] = min_sr1 (xt, h, m, alpha)
  ...
  while ~ok
    ...
    invsr = change_form_inv (sr0, h, m, low_rp);
    big_f = change_form (xt-invsr, h, m);
    ...
    while iter_s < 3*m
      ...
      invdr0 = change_form_inv (sr0, h, m, low_rp);
      sssdr = change_form (invdr0, h, m);
      ...
    end
    ...
    invsr = change_form_inv (sr0, h, m, low_rp);
    big_f = change_form (xt-invsr, h, m);
    ...
    while iter_r < n1*n2
      ...
      invdr0 = change_form_inv (sr0, h, m, low_rp);
      sssdr = change_form (invdr0, h, m);
      ...
    end
    ...
  end
```

XML-based Language

- Enables library writers to express transformations of interest
- Can specify type-based specializations
- Powerful enough to specify library indentities

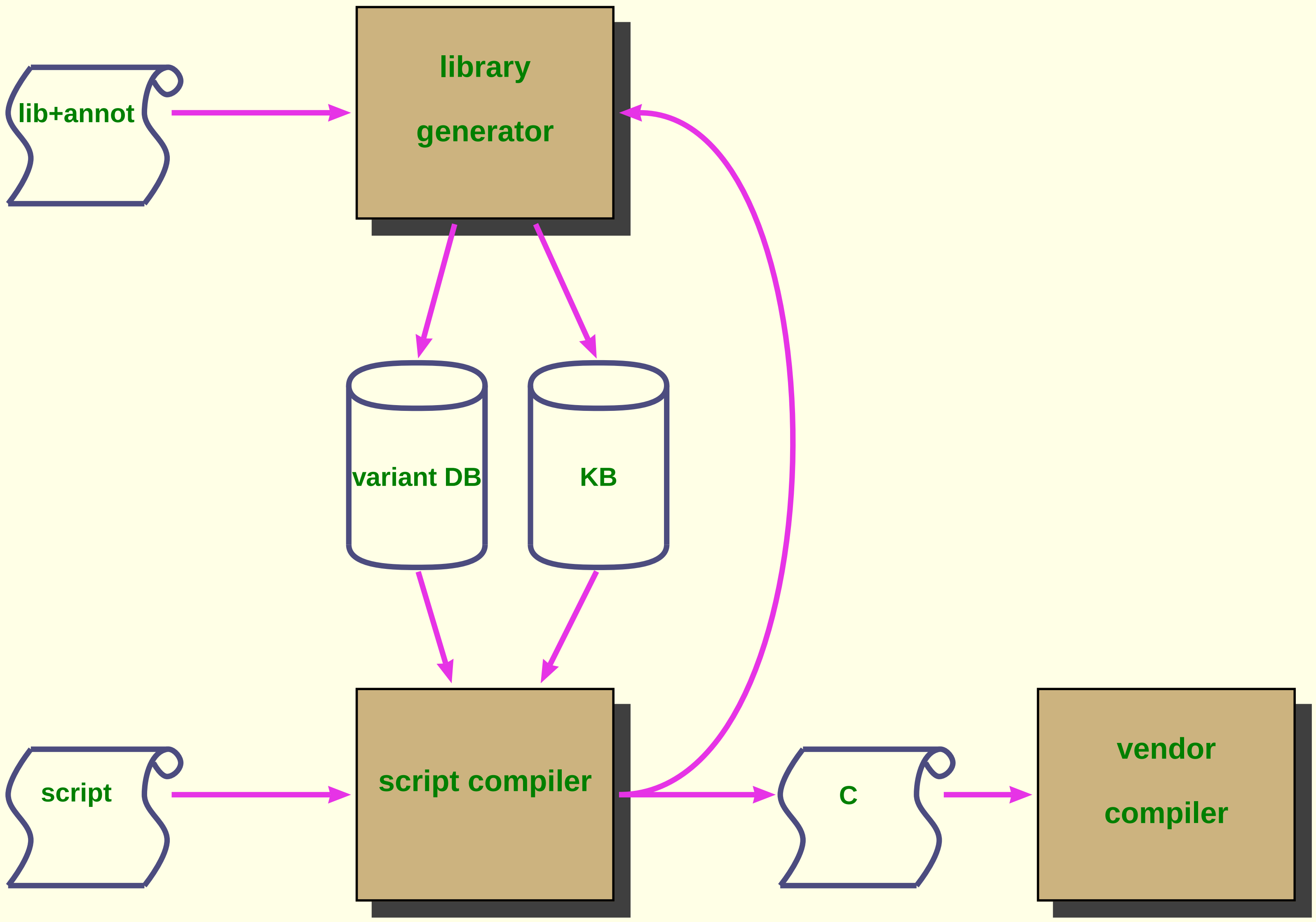
XML-based Language

- Enables library writers to express transformations of interest
- Can specify type-based specializations
- Powerful enough to specify library indentities
- Serves as a driver for the source-level optimization phase

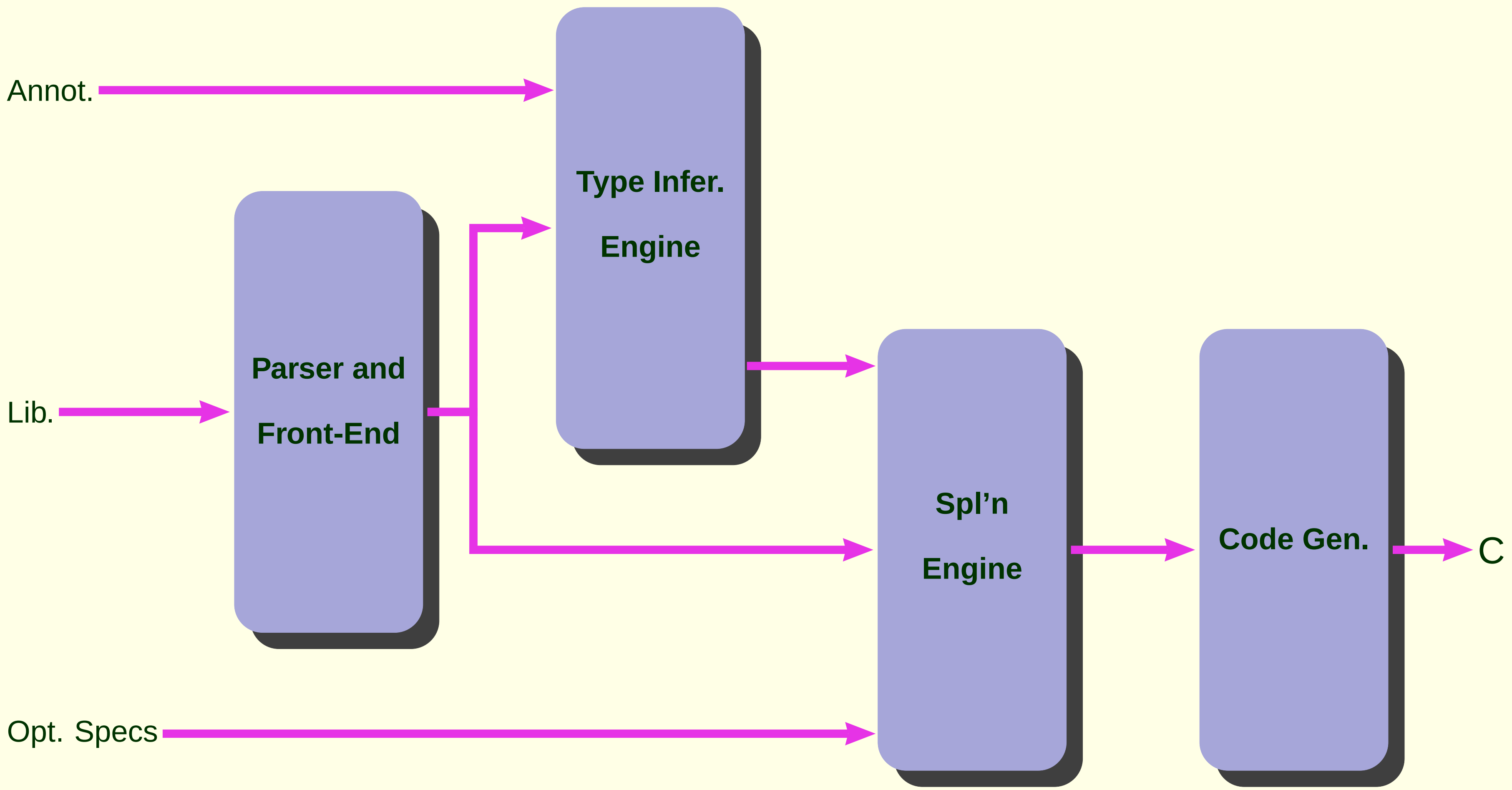
Example: Type-based Specialization

```
<specialization>
  <context>
    <type var="x" dims="0"/>
    <type var="y" dims="0"/>
  </context>
  <match>
    <simpleStmt>
      <function> generic_ADD </function>
      <input> <var>x</var> <var>y</var> </input>
      <output> <var>z</var> </output>
    </simpleStmt>
  </match>
  <substitute>
    <simpleStmt>
      <function> scalar_ADD </function>
      <input> <var>x</var> <var>y</var> </input>
      <output> <var>z</var> </output>
    </simpleStmt>
  </substitute>
</specialization>
```

Overall Telescoping System



Architecture of the Library Compiler



Meanwhile, Elsewhere . . .

- Compiling MATLAB
  - FALCON, MaJIC (UIUC, Cornell)
  - MATCH (Northwestern)
- Parallelizing MATLAB
  - CONLAB (Sweden), Otter (Oregon State), MENHIR (Irisa), . . .
  - *P (MIT)
- Annotations
  - Broadway (UT Austin)
- High-level programming systems
  - POOMA, ROSE (LLNL)
- Automatic library generation
  - ATLAS (UTK), FFTW (MIT)

Concluding Remarks

- Need to raise the level of interface with computers
  - scripting languages raise the level of programming interface
- Scripting languages provide higher abstraction in programming languages but incur performance penalties
- Libraries need to be at the core of a compilation strategy for scripting languages
  - speculative specialization
  - incorporating expert knowledge of library writers
- Experience with MATLAB indicates that a library-centered approach pays off

Future Directions

- Parallel computation
  - speculation or specification of data distribution?
  - library identities
- Dynamically evolving systems (such as the computation grid)
  - speculatively specializing on possible scenarios
  - dynamically switching library versions
  - pre-building schedules
  - self-learning systems through feedback
- Library compilation ideas in other domains
  - VLSI design
  - component-based systems

Other Possible Directions

- Developing annotation language
- Refining techniques to speculatively optimize code
  - database techniques
- Time and space trade-offs in library generation
  - machine learning techniques
- Diversifying the source language systems
  - R, Python, Perl, etc.
- Self-learning systems
  - extracting general contexts from examples
  - incorporating feedback through maintenance-mode runs

`http://www.cs.indiana.edu/~achauhan/`