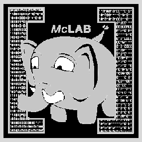
*Mc*LAB: Compiler Tools for MATLAB



Amina Aslam
Toheed Aslam
Andrew Casey
Maxime Chevalier- Boisvert
Jesse Doherty
Anton Dubrau
Rahul Garg
Maja Frydrychowicz
Nurudeen Lameed
Jun Li
Soroush Radpour
Olivier Savary

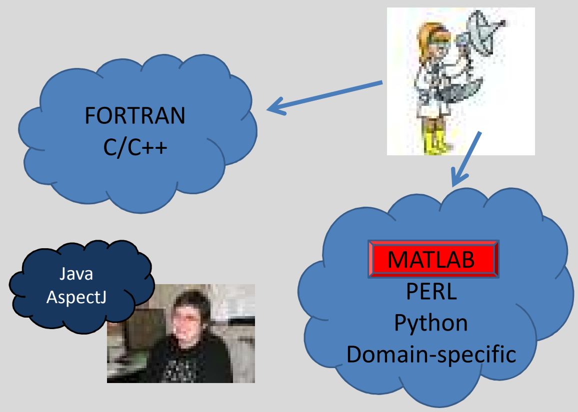
Laurie Hendren
McGill University
Notes for COMP 621 – Week 2

Overview – PART I

- Why MATLAB?
- Introduction to MATLAB – challenges
- Overview of the McLAB tools
- Resolving names in MATLAB

Nature Article: "Why Scientific Computing does not compute" [Merali, Oct 2010]

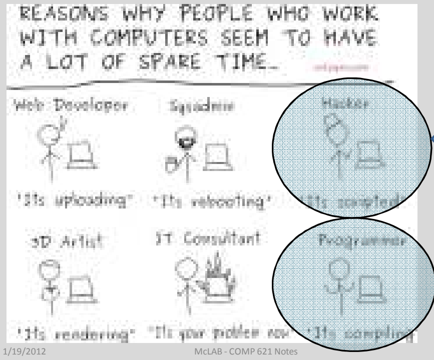
- 38% of scientists spend at least 1/5th of their time programming.
- Codes often buggy, sometimes leading to papers being retracted. Self-taught programmers.
- Monster codes, poorly documented, poorly tested, and often used inappropriately.
- 45% say scientists spend more time programming than 5 years ago.



A lot of MATLAB programmers!

- Started as an interface to standard FORTRAN libraries for use by students.... but now
  - 1 million MATLAB programmers in 2004, number doubling every 1.5 to 2 years.
  - over 1200 MATLAB/Simulink books
  - used in many sciences and engineering disciplines
- Even more "unofficial" MATLAB programmers including those using free systems such as Octave or SciLab.

Why do Scientists choose MATLAB?



MATLAB

FORTRAN

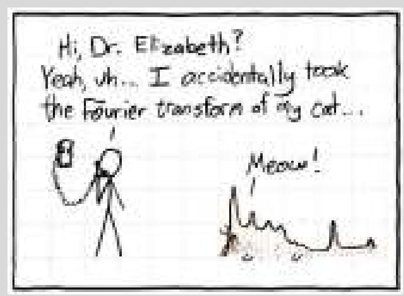
Implications of choosing a dynamic, "scripting" language like MATLAB....

Many run-time decisions ...

Potentially large runtime overhead in both time and space

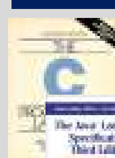
No types and "flexible" syntax

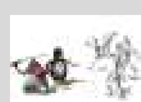
http://imgs.xkcd.com/comics/fourier.jpg

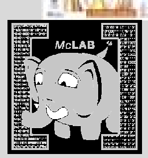


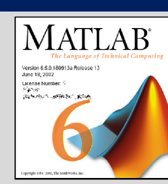
Most semantic (syntactic) checks made at runtime ... No static guarantees

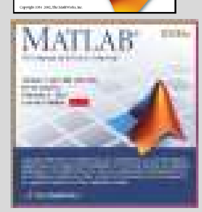
No formal standards for MATLAB











Culture Gap

Scientists / Engineers

- Comfortable with informal descriptions and "how to" documentation.
- Don't really care about types and scoping mechanisms, at least when developing small prototypes.
- Appreciate libraries, convenient syntax, simple tool support, and interactive development tools.

Programming Language / Compiler Researchers

- Prefer more formal language specifications.
- Prefer well-defined types (even if dynamic) and well-defined scoping and modularization mechanisms.
- Appreciate "harder/deeper/more beautiful" programming language/compiler research problems.

Goals of the McLab Project

- Improve the understanding and documentation of the semantics of MATLAB.
- Provide front-end compiler tools suitable for MATLAB and language extensions of MATLAB.
- Provide a flow-analysis framework and a suite of analyses suitable for a wide range of compiler/soft. eng. applications.
- Provide back-ends that enable experimentation with JIT and ahead-of-time compilation.

Enable PL, Compiler and SE Researchers to work on MATLAB

Brief Introduction to MATLAB

Functions and Scripts in MATLAB

Basic Structure of a MATLAB function

```
function [ prod, sum ] = ProdSum( a, n )
  prod = 1;
  sum = 0;
  for i = 1:n
    prod = prod * a(i);
    sum = sum + a(i);
  end;
end
```

```
>> [a,b] = ProdSum([10,20,30],3)
a = 6000
b = 60

>>  ProdSum([10,20,30],2)
ans = 200

>> ProdSum('abc',3)
ans =941094

>> ProdSum([97 98 99],3)
ans = 941084
```


Primary, nested and sub-functions

Primary Function

Nested Function

Sub-Function

```
% should be in file NestedSubEx.m
function [ prod, sum ] = NestedSubEx( a, n )
  function [ z ] = MyTimes( x, y )
    z = x * y;
  end
  prod = 1;
  sum = 0;
  for i = 1:n
    prod = MyTimes(prod, a(i));
    sum = MySum(sum, a(i));
  end;
end

function [z] = MySum ( x, y )
  z = x + y;
end
```


Basic Structure of a MATLAB script

```
% stored in file ProdSumScript.m
prod = 1;
sum = 0;
for i = 1:n
  prod = prod * a(i);
  sum = sum + a(i);
end;
```

```
>> clear
>> a = [10, 20, 30];
>> n = 3;
>> whos
  Name   Size   Bytes   Class
  a      1x3    24      double
  n      1x1    8       double
>> ProdSumScript()
>> whos
  Name   Size   Bytes   Class
  a      1x3    24      double
  i      1x1    8       double
  n      1x1    8       double
  prod   1x1    8       double
  sum    1x1    8       double
```

Directory Structure and Path

- Each directory can contain:
  - `.m` files (which can contain a script or functions)
  - a `private/` directory
  - a package directory of the form `+pkg/`
  - a type-specialized directory of the form `@int32/`
- At run-time:
  - current directory (implicit 1st element of path)
  - directory of last called function
  - path of directories
  - both the current directory and path can be changed at runtime (`cd` and `setpath` functions)

Function/Script Lookup Order (call in the body of a function f)

```
function f
 ...
 foo(a);
 ...
end
```

- Nested function (in scope of f)
- Sub-function (in same file as f)
- Function in /private sub-directory of directory containing f.
- 1st matching function, based on function name and type of first argument, looking in type-specialized directories, looking first in current directory and then along path.
- 1st matching function/script, based on function name only, looking first in current directory and then along path.

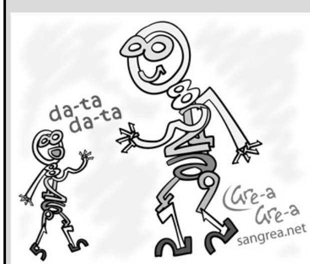
Function/Script Lookup Order (call in the body of a script s)

```
% in s.m
...
foo(a);
...
```

- Function in /private sub-directory of directory of last called function (not the /private sub-directory of the directory containing s).
- 1st matching function/script, based on function name, looking first in current directory and then along path.

```
dir1/          dir2/
  f.m            s.m
  g.m            h.m
  private/       private/
    foo.m          foo.m
```


Variables and Data in MATLAB



MATLAB types: high-level

```
            any
     data          fnhandle
array cellarray struct
```


Variables

- Variables are not explicitly declared.
- Local variables are allocated in the current workspace. Global and persistent variables in a special workspace.
- All input and output parameters are local.
- Local variables are allocated upon their first definition or via a load statement.
  - `x = ...`
  - `x(i) = ...`
  - `load ('f.mat', 'x')`
- Local variables can hold data with different types at different places in a function/script.

Variable Workspaces

- There is a workspace for global and persistent variables.
- There is a workspace associated with the read-eval-print loop.
- Each function call creates a new workspace (stack frame).
- A script uses the workspace of its caller (either a function workspace or the read-eval-print workspace).

1/19/2012 McLAB - COMP 621 Notes Matlab - 25

Variable Lookup

- If the variable has been declared global or persistent in the function body, look it up in the global/persistent workspace.
- Otherwise, lookup in the current workspace (either the read-eval-print workspace or the top-most function call workspace).
- For nested functions, use the standard scoping mechanisms.

1/19/2012 McLAB - COMP 621 Notes Matlab - 26

Other Tricky "features" in MATLAB

1/19/2012 McLAB - COMP 621 Notes

Irritating Front-end "Features"

- keyword **end** not always required at the end of a function (often missing in files with only one function).
- command syntax
  - `length('x')` or `length x`
  - `cd('mydirname')` or `cd mydirname`
- arrays can be defined with or without commas:
  [10, 20, 30] or [10 20 30]
- sometimes newlines have meaning:
  - **a = [10 20 30**
    40 50 60]; // defines a 2x3 matrix
  - **a = [10 20 30 40 50 60]; // defines a 1x6 matrix**
  - **a = [10 20 30;**
    40 50 60]; // defines a 2x3 matrix
  - **a = [10 20 30; 40 50 60]; // defines a 2x3 matrix**

1/19/2012 McLAB - COMP 621 Notes Matlab - 28

“Evil” Dynamic Features

- not all input arguments required

```
function [ prod, sum ] = ProdSumNargs( a, n )
  if nargin == 1 n = 1; end;
  ...
end
```

- do not need to use all output arguments
- eval, evalin, assignin
- cd, addpath
- load

1/19/2012 McLAB - COMP 621 Notes Matlab - 29

Evil Feature of the Day - Looking up an identifier

Old style general lookup - interpreter

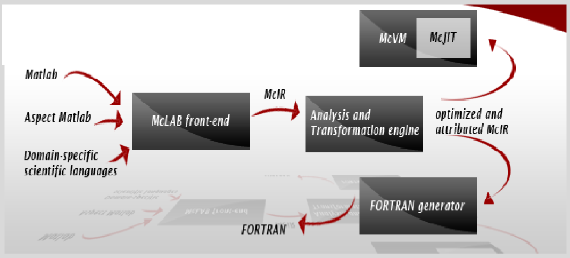
- First lookup as a variable.
- If a variable not found, then look up as a function.

MATLAB 7 lookup - JIT

- When function/script first loaded, assign a "kind" to each identifier. VAR – only lookup as a variable, FN – only lookup as a function, ID – use the old style general lookup.
- How is the kind assignment done. What impact does it have on the semantics?

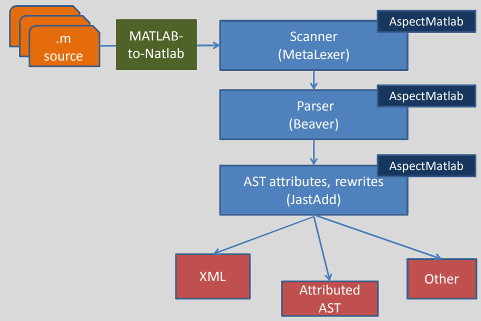
1/19/2012 McLAB - COMP 621 Notes Matlab - 30

McLab – Overall Structure



1/19/2012 McLAB - COMP 621 Notes 31

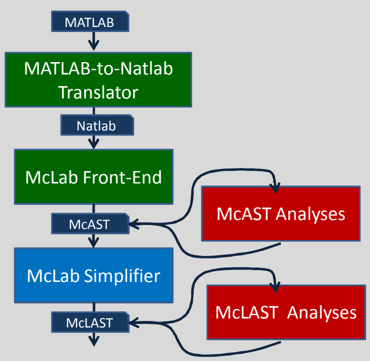
McLab Extensible Front-end



1/19/2012 McLAB - COMP 621 Notes 32

Analysis Engine

Analyses are written using an Analysis Framework that supports forward and backward flow analysis over McAST and McLAST.

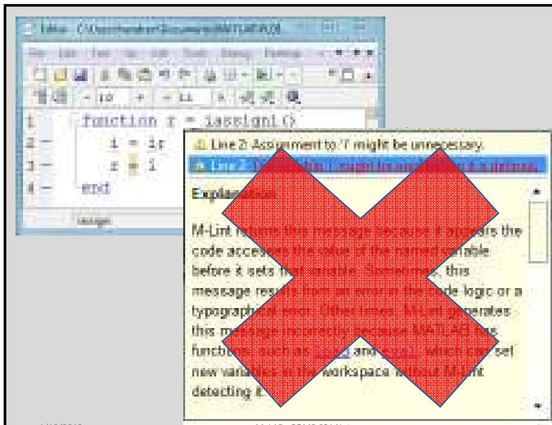


1/19/2012 McLAB - COMP 621 Notes 33

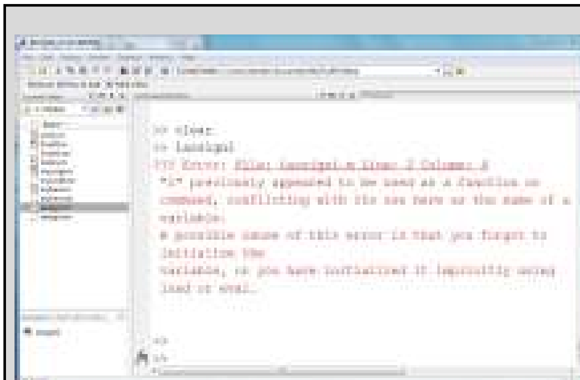
How does MATLAB resolve Names?

- No official specification
- Motivating example

1/19/2012 McLAB - COMP 621 Notes 34



1/19/2012 McLAB - COMP 621 Notes 35



1/19/2012 McLAB - COMP 621 Notes 36

Read-Eval-Print Loop

Evil Feature of the Day - Recap

Old style general lookup - interpreter

- First lookup as a variable.
- If a variable not found, then look up as a function.

MATLAB 7 lookup - JIT

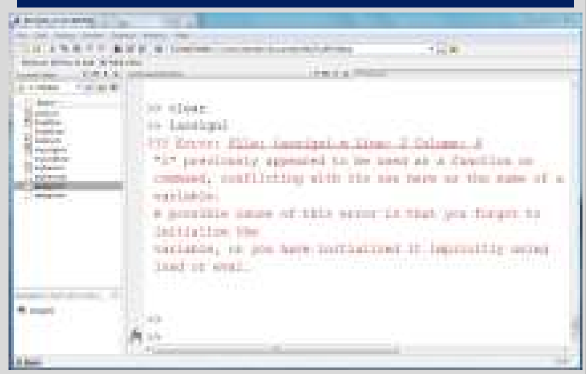
- When function/script first loaded, statically assign a "kind" to each identifier. VAR – only lookup as a variable, FN – only lookup as a function, ID – use the old style general lookup.
- Compile-time error if, within the body of a function or script, an identifier has kind VAR in one place and FN in another.

Does the kind analysis change the semantics?

Yes, in two ways!

1. New compile-time errors, so programs that would previously execute will not.
2. Different binding at run-time for some identifiers which are assigned a kind of VAR or FN.

Compile-time kind error



Different lookup with old vs MATLAB 7 semantics

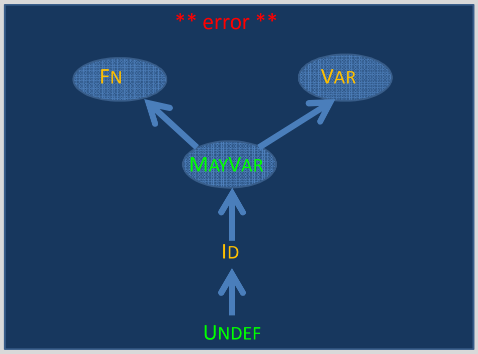
```
1 function [ r ] = KindEx( a )
2   x = a + sum(j);
3   eval('sum = ones(10);');
4   r = sum(x);
5 end
```

- Old interpreter semantics:
  - sum, line 2, named function
  - sum, line 4, local variable
- MATLAB 7 semantics gives a static kind of FN to sum
  - sum, line 2, named function
  - sum, line 4, named function

Our approach to the Kind Analysis Problem

- Identify that a kind analysis is needed to match MATLAB 7 semantics.
- Specify and implement a kind assignment algorithm that matches the observed behaviour of MATLAB 7. (both for functions and for scripts)
- Identify any weaknesses in the MATLAB 7 approach and suggest two more clearly defined alternatives, one flow-sensitive and one flow-insensitive.
- Determine if the alternatives could be used without significant change to the behaviour of existing MATLAB programs.

Kind Abstraction



Kind Analysis

1. Collect all identifiers used in function/script and set initial kind approximations for each identifier.
2. Traverse AST applying analysis rules to identifiers.
3. Traverse AST making final kind assignment.

Steps 1 and 3 are different for scripts and functions, step 2 uses the same rules.

Step 2: Kind Analysis Rules

Definition of identifier *x*:

$$kind[x] \leftarrow kind[x] \bowtie \text{VAR}$$

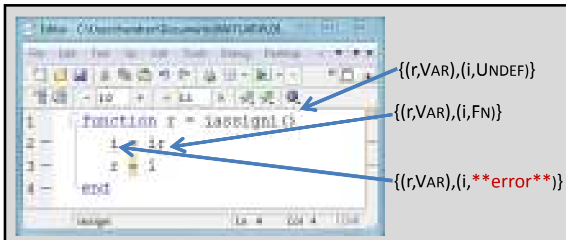
Use of identifier *x*:

if $((kind[x] \in \{\text{ID}, \text{UNDEF}\})$&exists_lib(x,lib))
  $kind[x] \leftarrow \text{FN}$
else
  $kind[x] \leftarrow kind[x] \bowtie \text{ID}$

Kind Analysis for Functions

- Initial values: input and output parameters are initialized to VAR, all other identifiers are initialized as UNDEF.
- Final values:

```
for each id occurrence in f do
  if fkind[id] in {ID, MAYVAR}
    id.kind = ID
  else  /* fkind[id] in {VAR, FN} */
    id.kind = fkind[id]
```

if $((kind[x] \in \{\text{ID}, \text{UNDEF}\})$&exists_lib(x,lib))
  $kind[x] \leftarrow \text{FN}$
else
  $kind[x] \leftarrow kind[x] \bowtie \text{ID}$

READ RULE

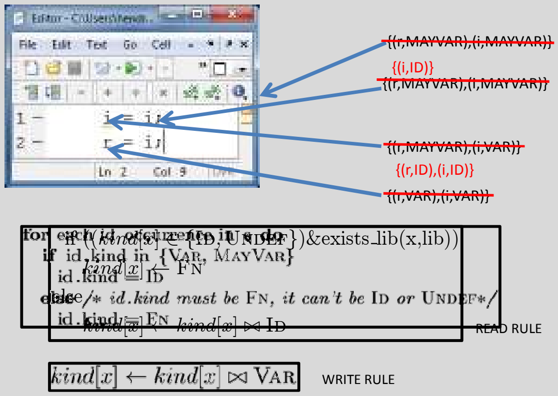
$kind[x] \leftarrow kind[x] \bowtie \text{VAR}$ WRITE RULE

Kind Analysis for Scripts

- Initial values: all identifiers are initialized to MAYVAR
- Final values:

```
for each id occurrence in s do
  if id.kind in {VAR, MAYVAR}
    id.kind = ID
  else /* id.kind must be FN, it can't be ID or UNDEF*/
    id.kind = FN
```

- Note: most identifiers will be mapped to ID



Problems with MATLAB 7 kind analysis

- apparently not clearly documented, in some ways just a side-effect of a JIT implementation decision
- without a clear specification, confusing for the programmer and compiler/tool developer
- loses almost all information about variables in scripts
- some strange anomalies due to a "traversal-sensitive" analysis

Examples of Anomalies

if (exp)
... = sum(10); (sum,FN)
else
sum(10) = ...; *error*
	if (~exp)
	sum(10) = ... ; (sum,VAR)
	else
	... = sum(10); (sum,VAR)

size(size(10)) = ... (size,VAR) (size, VAR)	t = size(10); (size,FN) size(t) = ... *error*

Flow-sensitive Analysis

if (exp)
... = sum(10); (sum,FN)
else
sum(10) = ...; (sum, VAR)
// merge, *error*
	size(size(10)) =
	(size,FN)
	error

- Apply a flow-sensitive analysis that merges at control-flow points.
- Consider explicit loads to be definitions -
  `load ('f.mat', 'x')`
- Map final kinds for scripts using the same algorithm as for functions.

Flow-insensitive Analysis

if (exp)
... = sum(10);
else
sum(10) = ...;
(sum,VAR)
	size(size(10)) =
	(size,VAR)

1. Assign VAR to identifiers that are defined on lhs, or declared global or persistent.
2. Assign FN to identifiers which have a handle taken or used in command syntax.
3. Assign FN to identifiers that have no assignment yet, and which are found in the library.

error if assigned both FN and VAR

Results: What is the distribution of kinds for functions/scripts in real MATLAB programs?

Various-sized benchmarks from a wide variety of application areas

Benchmark Category	# Benchmarks
Single (1 file)	2051
Small (2-9 files)	848
Medium (10-49 files)	113
Large (50-99 files)	9
Very Large (≥ 100 files)	2
Total	3024

Send benchmarks or links to hendren@cs.mcgill.ca

Results for Functions - number of identifiers with each Kind

Kind	MATLAB 7	Flow-Sens.	Flow-Insens.
VAR	107388	107401	107406
FN	75533	75533	75533
ID	2369	2335	2335
error	1	3	0
warn	0	9	7
Total	185291	185291	185291

11698 functions

Results for Scripts – number of identifier instances with each Kind

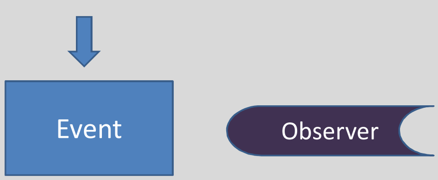
Kind	MATLAB 7 raw	MATLAB 7 post-process	Flow-sens.	Flow-Insens
VAR	153444	0	153954	153954
FN	1	1	3	3
ID	69022	222466	68410	68410
error	0	0	0	0
warn	0	0	100	100
Total	222467	222467	222467	222467

2035 scripts

Overview – PART II

- What is an Aspect
- AspectMatlab
- Typing Aspects

What is an Aspect?



- Pattern specifying events to match.
- Action to do before, after or around the matched events.
- Action can use context information from the matched event.

Example: Profiling Array Sparsity

0 0 0 9 0 0 0 0 0 0
0 0 0 0 0 0 5 0 0 0
0 1 0 0 0 0 0 3 0 0
0 0 0 0 4 0 0 0 0 0
0 0 7 0 0 0 0 0 0 0

- Capture the sparsity and size at each operation on the whole array.
- Capture the number of indexed references to each array.
- Print out a summary for each array, allowing the programmer to identify good candidates to implement as sparse arrays.

Background - MATLAB Class

```
classdef myClass
  properties
     …
  end

  methods
     …
  end
end
```

data
count = 0;

helper functions
function x=getCount(this)
x = this.count;
end

Aspect Definition

```
aspect myAspect
  properties
     …
  end
  methods
     …
  end
  patterns
     …
  end
  actions
     …
  end
end
```

data
count = 0;

helper functions
function x=getCount(this)
x = this.count;
end

pointcuts
foocalls : call(foo);

advice
foocounter : before foocalls
this.count = this.count + 1;
end

Function and Operator Patterns

```
patterns
   pCallFoo : call(foo);
   pExecBar : execution(bar);

   pCallFoo2args : call(foo(*,*));
   pExecutionMain : mainexecution();
end
```

```
patterns
   plusOp : op(+);
   timesOp : op(.*) || op(*);
   matrixOps: op(matrix);
   allButMinus: op(all) & ~op(-);
end
```


Array Patterns

a(i) = b(j,k)

also,
new value

Context Info
name
indices
object (value)
line number
location
file name

```
patterns
   pSetX : set(a);
   pGetX : get(b);

   arraySet : set(*);
   arrayWholeGet : get(*());
   arrayIndexedGet : get(*(..));
end
```


Loop Patterns

```
t1 = [1,3,5,7,9,…,n];
for   t2 = 1:numel(t1)
   i = t1(t2);
   …
   …
   …
end
```

```
patterns
   pLoopI : loop(i);
   pLoopHeadI : loophead(i);
   pLoopBodyI : loopbody(i);
end
```


Scope Patterns

```
patterns
   pWithinFoo : within(function, foo);
   pWithinBar : within(script, bar);
   pWithinMyClass : within(class, myClass);
   pWithinLoops : within(loops, *);
   pWithinAllAbc : within(*, abc);
end
```

Compound Patterns

- Logical combinations of primitive patterns

```
patterns
   pCallFoo  : call(foo) & within(loops, *);
   pGetOrSet : (get(*) | set(*)) & within(function, bar);
end
```


Before & After Actions

```
actions
   aCountCall : before pCall
      this.count = this.count + 1;
      disp('calling a function');
   end

   aExecution : after executionMain
      total = this.getCount();
      disp(['total calls: ', num2str(total)]);
   end
end
```


Context Exposure

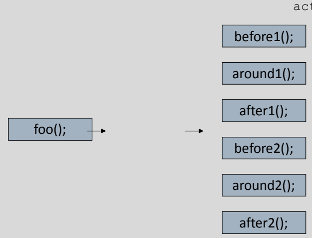
```
actions
   aCountCall : before pCall : (name, args)
      this.count = this.count + 1;
      disp(['calling ',name,' with args(',args, ')']);
   end

   aExecution : after executionMain : (file)
      total = this.getCount();
      disp(['total calls in ',file,': ',num2str(total)]);
   end
end
```


Around Actions

```
actions
   actcall : around pCallFoo : (args)
      disp(['before foo call with args(', args , ')']);
      proceed();
      disp(['after foo call with args(', args , ')']);
   end
end
```

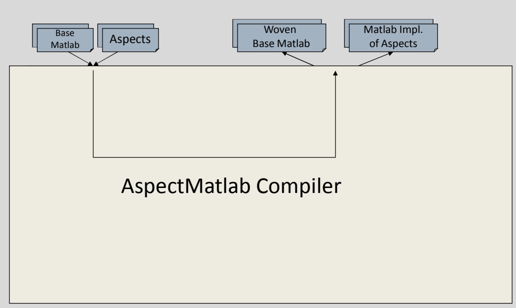
```
actions
   actcall : around pCallFoo : (args)
      % proceed not called, so varargout is set
      varargout{1} = bar(args{1}, args{2});
   end
end
```


Actions Weaving Order

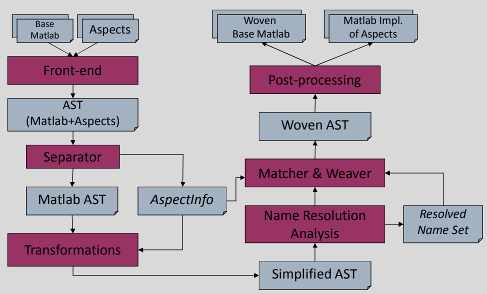



```
actions
   before1 : before pCallFoo
   …
   around1 : around pCallFoo
   …
   after1 : after pCallFoo
   …
   before2 : before pCallFoo
   …
   around2 : around pCallFoo
   …
   after2 : after pCallFoo
   …
end
```


Compiler Structure



Compiler Structure



Name Resolution Analysis

```
patterns
   pCallFoo : call(foo);
   pGetFoo : get(foo);
end
actions
   before1 : before pCallFoo
   …
   before2 : before pGetFoo
   …
end
```

before1();

before2();

if isVar(foo)

foo(); function

foo(); variable

foo(); unresolved

Scientific Use Cases

- Domain-Specific Profiling of Programs
  - Tracking array sparsity
  - Tracking array size-growing operations
  - Counting floating-point operations
- Extending Functionality
  - Interpreting loop iteration space
  - Adding units to computations

Related Work for AspectMatlab

- AspectJ *(Kiczales et al., ECOOP '01)*
  - abc *(The de Moor and Hendren gang, AOSD '05)*
  - Array pointcuts *(Chen et al., JSES '07)*
  - Loop pointcuts *(Harbulot et al., AOSD '06)*
- AspectCobol *(Lammel et al., AOSD '05)*
- Domain-Specific Aspects in Matlab *(Cardoso et al., DSAL workshop held at AOSD '10)*

Typing Aspects

- Types for MATLAB, somewhat in the spirit of aspects.
- Designed by what programmers might want to say.
- Checked at run-time, but some static analysis could be done.

Simple Example MATLAB function

```
function [ r ] = Ex1( n )
% Ex1(n) creates a vector of n values containing
%   the values [sin(1), sin(2), ..., sin(n)]
  for i=1:n
    r(i) = sin(i);
  end
end
```

```
>> Ex1(3)
ans = 0.8415     0.9093     0.1411

>> Ex1(2.3)
ans = 0.8415     0.9093
```

```
>> Ex1(int32(3))
??? Undefined function or method 'sin' for input
    arguments of type 'int32'.
Error in ==> Ex1 at 5
  r(i) = sin(i);

>> Ex1('c')
??? For colon operator with char operands, first
    and last operands must be char.
Error in ==> Ex1 at 4
  for i=1:n

>> Ex1(@sin)
??? Undefined function or method '_colonobj' for
    input arguments of type 'function_handle'.
Error in ==> Ex1 at 4
  for i=1:n
```



```
>> Ex1(complex(1,2))
Warning: Colon operands must be real scalars.
> In Ex1 at 4
ans = 0.8415

>> Ex1(true)
Warning: Colon operands should not be logical.
> In Ex1 at 4
ans = 0.8415

>> Ex1([3,4,5])
ans = 0.8415    0.9093    0.1411
```


MATLAB prorammers often expect certain types

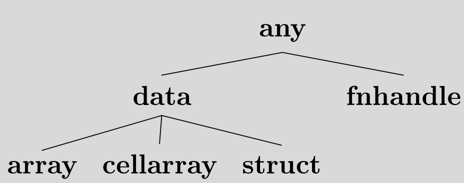
```
function y = sturm(X,BC,F,G,R)
% STURM Solve the Sturm-Liouville equation:
%   d( F*dY/dX )/dX - G*Y = R using linear finite elements.
% INPUT:
%   X - a one-dimensional grid-point array of length N.
%   BC - is a 2 by 3 matrix [A1, B1, C1 ; An, Bn, Cn]
...
% Alex Pletzer: pletzer@pppl.gov (Aug. 97/July 99).
...
```

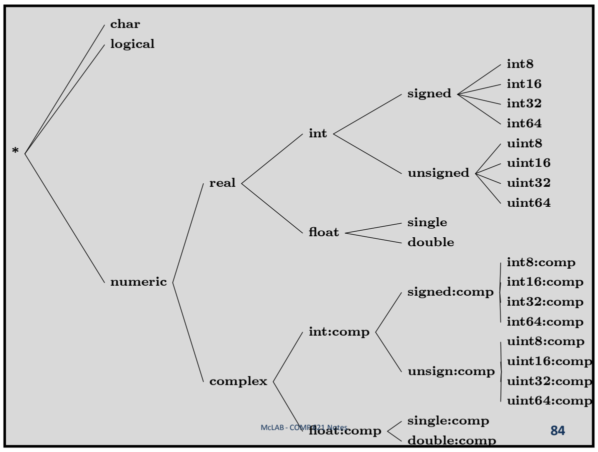


```
1 function [ r ] = Ex1( n )
2 % Ex1(n) creates a vector of n values containing
3 %   the values [sin(1), sin(2), ..., sin(n)]
4   atype('n','scalar of Float');
5   for i=1:n
6     r(i) = sin(i);
7   end
8   atype('r','array [n.value] of n.basetype');
9 end
```

```
>> Ex1(3)
ans = 0.8415    0.9093    0.1411

>> Ex1('c')
Type error in Ex1.m, Line 4: Expecting 'n' to have
type 'scalar of float', but got the type
'scalar of char'.
```

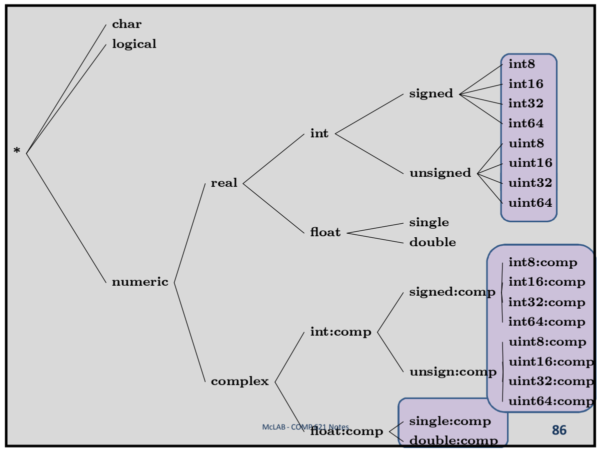

High-level types in MATLAB





Simple Example

```
function [ r ] = foo( a, b, c, d )
  atype('a', 'array [...] of int');
  atype('b', 'array[*,*]');
  atype('c', 'array[*,*,...]) of complex');
  atype('d', 'scalar of uint32');
  % ...
  % body of foo
  % ...
  atype('r','array[a.dims] of int');
end
```

Capturing reflective information

```
function [ r ] = foo( a )
  atype('a','any');
  % ...
  % body of foo
  % ...
  atype('r','a.type');
end
```

- a.type
- a.value
- a.dims
- a.basetype

Capturing dimensions and basetype

```
function [ r ] = foo( a, b )
  atype('a','array[<n>,<m>] of real');
  atype('b','array[a.m,<p>] of a.basetype');
  % ...
  % body of foo
  % ...
  atype('r','array[a.m,b.p] of a.basetype');
end
```

- <n> can be used as a dimension spec
- value of n is instantiated from the runtime dimension
- repeated use in same atype statement implies equality

Conclusions

- MATLAB is an important language, but presents challenges for compiler writers
- McLAB provides toolkits for analysis and transformation of MATLAB
- McLAB is extensible, AspectMatlab is one extension
- Typing Aspects is another possible extension.