Language Extensions
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

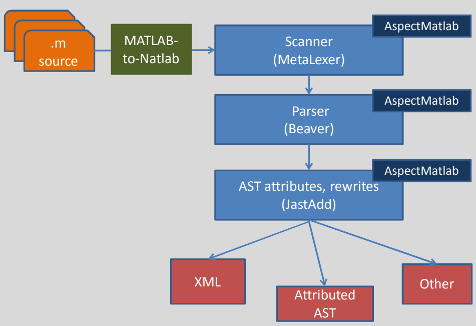
Laurie Hendren
McGill University

Leverhulme Visiting Professor
Department of Computer Science
University of Oxford

Overview

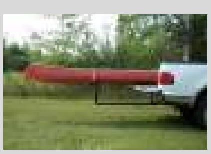
- How does one make language extensions for MATLAB using McLab?
- MetaLexer
- Aspects for MATLAB
- ["Types" for MATLAB]

McLab Extensible Front-end

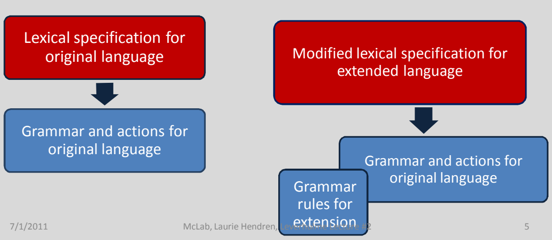


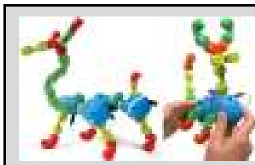
MetaLexer

- Modular Lexer Generator
- M.Sc. thesis, Andrew Casey
- AOSD 2011
- www.sable.mcgill.ca/metalexer

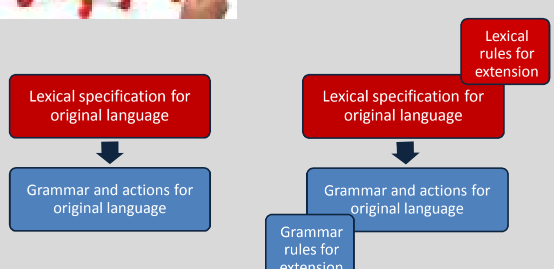


Given a front-end specification for a language (i.e. MATLAB), current method to implement a front-end for an extension of that language (i.e. AspectMatlab)?





Desired Modular MetaLexer Approach



We also want to be able to combine lexical specifications for diverse languages.

- Java + HTML
- Java + Aspects (AspectJ)
- Java + SQL
- MATLAB + Aspects (AspectMatlab)

Scanning AspectJ

```
package foo;

aspect Aspect {
    before() : (execution(* Clazz.* (..)) || if(Clazz.flag)) {
        System.out.println("Hello");
    }

    class Clazz {
        static boolean flag = false;

        public static void foo() {
            flag = !flag;
        }
    }
}
```

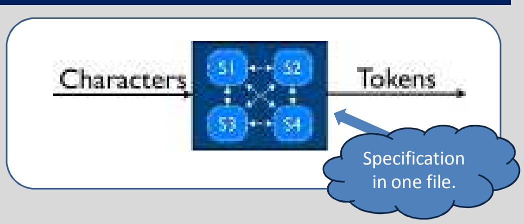

Would like to be able to reuse and extend lexical specification modules

- Nested C-style comments
- Javadoc comments
- Floating-point constants
- URL
- regular expressions
- …

First, let's understand the traditional lexer tools (lex, flex, jflex).

- programmer specifies regular expressions + actions
- tools generate a finite automaton-based implementation
- states are used to handle different language contexts

JFlex Lexing Structure



- **Lexing rules associated with a state.**
- **Changing states associated with action code.**

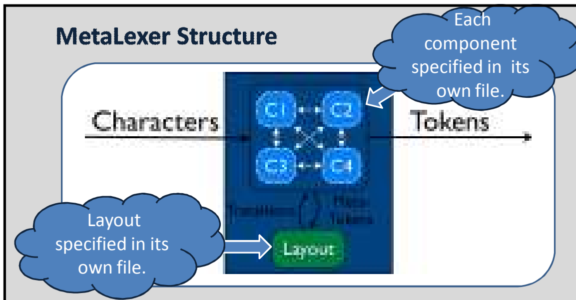

```
%%
%class Lexer
 Identifier  = [: jletter :]  [: jletterdigit :]*
 ...
%state STRING
%%
<YYINITIAL> {
  "abstract"       { return symbol(sym.ABSTRACT); }
  { Identifier }   { return symbol(sym.IDENTIFIER); }
  \"               { string.setLength(0); yybegin(STRING); }
  ...
}

<STRING> {
  \"               { yybegin(YYINITIAL); return ...; }
  [^\n\r\"\\]+     { string.append( yytext() ); }
  \\t              { string.append('\t'); }
  ...
}
```

Current (ugly) method for extending jflex specifications - copy&modify

- Copy jflex specification.
- Insert new scanner rules into copy.
  - Order of rules matters!
- Introduce new states and action logic for converting between states.

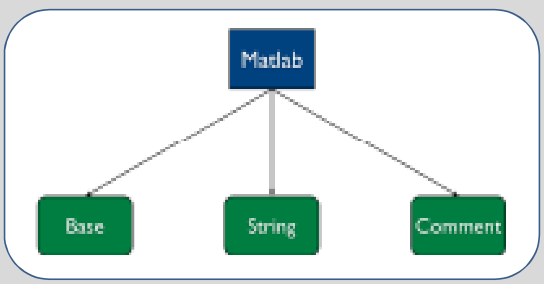
- Principled way of weaving new rules into existing rules.
- Modular and abstract notion of state and changing between states.

MetaLexer Structure

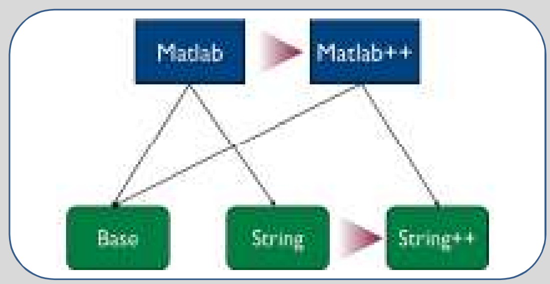


- Components define lexing rules associated with a state, rules produce meta-tokens.
- Layout defines transitions between components, state changes by meta-lexer (regular expressions + matching pairs of start/end symbols).

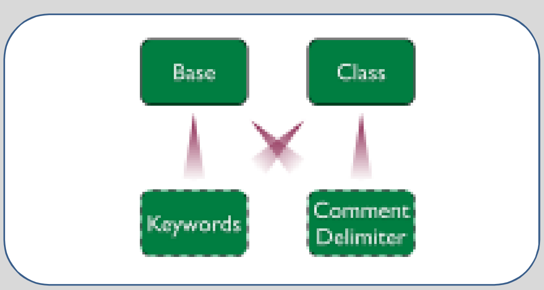
Example Structure of a MetaLexer Specification for MATLAB



Extending a MetaLexer Specification for Matlab

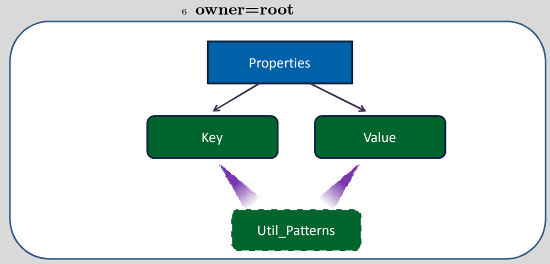


Sharing component specifications with MetaLexer



Scanning a properties file

```
#some properties
name=properties
date=2009/09/21

#some more properties
owner=root
```



util_properties.mlc helper component

```
%component util_patterns
%helper

lineTerminator = [\r\n] | "\r\n"
otherWhitespace = [ \t\f\b]
 identifier  = [a-zA-Z][a-zA-Z0-9_]*
comment = #[^\r\n]*
```


key.mlc component

```
%component key
%extern "Token symbol(int)"
%extern "Token symbol(int, String)"
%extern "void error(String) throws LexerException"

%%

%%inherit util_patterns
{lineTerminator} {: /*ignore*/ :}
{otherWhitespace} {: /*ignore*/ :}
"=" {: return symbol(ASSIGN); :} ASSIGN
%:
{ identifier } {: return symbol(KEY, yytext()); :}
{comment} {: /*ignore*/ :}
%:
<<ANY>> {: error("Unexpected char '"+yytext()+"'"); :}
<<EOF>> {: return symbol(EOF); :}
```

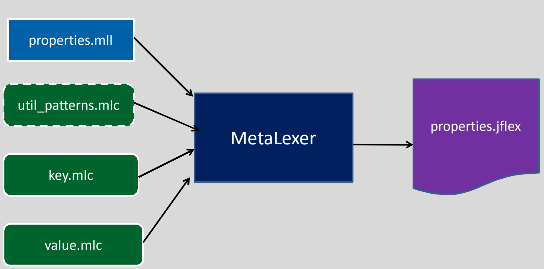

value.mlc component

```
%component value
%extern "Token symbol(int, String, int,  int ,  int ,  int)"
%append{
      return symbol(VALUE, text, startLine, startCol,
                                            endLine, endCol);
%append}

%%

%%inherit util_patterns
{lineTerminator} {: :} LINE_TERMINATOR
%:
%:
<<ANY>> {: append(yytext()); :}
<<EOF>> {: :} LINE_TERMINATOR
```


properties.mll layout

```
package properties;
%%
import static  properties.TokenTypes.*;
%%
%layout properties
%option public "%public"
 ...
%lexthrow "LexerException"
%component key
%component value
%start key
%%
%%embed
%name key_value
%host key
%guest value
%start ASSIGN
%end LINE_TERMINATOR
```

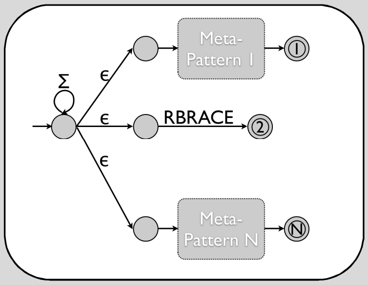

MetaLexer is implemented and available: www.sable.mcgill.ca/metalexer



Key problems to solve:

- How to implement the meta-token lexer?
- How to allow for insertion of new components, replacing of components, adding new embeddings (metalexer transitions).
- How to insert new patterns into components at specific points.

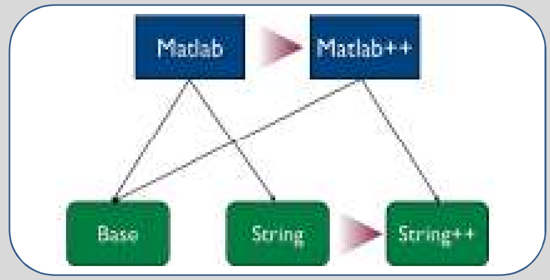
Implementing the meta-token lexer



Recognize a meta-pattern, i.e. when to go to a new component and when to return.

Recognize the matching suffix.

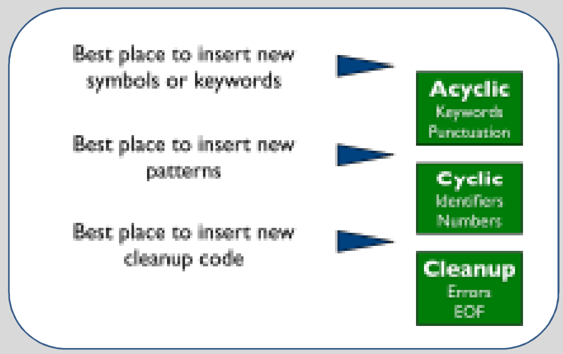
Implementing inheritance (structured weaving).



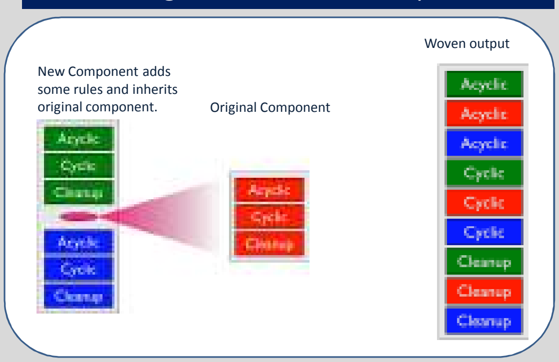
Implementing MetaLexer layout inheritance

- **Layouts can inherit other layouts**
- **%inherit directive put at the location at which the inherited transition rules (embeddings) should be placed.**
- **each %inherit directive can be followed by:**
  - **%unoption**
  - **%replace**
  - **%unembed**
  - **new embeddings**

Implementing MetaLexer component inheritance



Weaving in an inherited component



Results:

Applied to three projects with complex scanners:

- **AspectJ (abc and extensions)**
- **Matlab (Annotations and AspectMatlab extensions)**
- **MetaLexer**

AspectJ and Extensions

```
%%embed
%name perclause
%host aspect_decl
%guest pointcut
%start [PERCFLOW PERCFLOWBELOW PERTARGET
        PERTHIS] LPAREN
%end RPAREN
%pair LPAREN, RPAREN

%%embed
%name pointcut
%host java, aspect
%guest pointcut
%start POINTCUT
%end SEMICOLON
```


MetaLexer scanner implemented in MetaLexer

- 1st version of MetaLexer written in JFlex, one for components and one for layouts.
- 2nd version implemented in MetaLexer, many shared components between the component lexer and the layout lexer.

Related Work for MetaLexer

- Ad-hoc systems with separate scanner/ LALR parser
  - Polyglot
  - JastAdd
  - abc
- Recursive-descent scanner/parser
  - ANTLR and systems using ANTLR
- Scannerless systems
  - Rats! (PEGs)
- Integrated systems
  - Copper (modified LALR parser which communicates with DFA-based scanner)

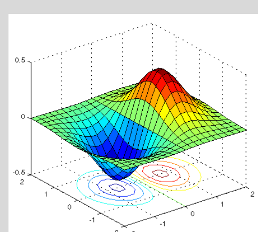
Metalexer Conclusions

- MetaLexer allows one to specify modular and extensible scanners suitable for any system that works with JFlex.
- Two main ideas: meta-lexing and component/layout inheritance.
- Used in large projects such as abc, McLab and MetaLexer itself.
- Available at: www.sable.mcgill.ca/metalexer

AspectMatlab

- Simple Aspect-Oriented extension to MATLAB
- M.Sc. thesis, Toheed Aslam
- Analysis by Jesse Doherty, applications by Anton Dubrau, extensions by Olivier Savary-Belanger
- AOSD 2010
- www.sable.mcgill.ca/mclab

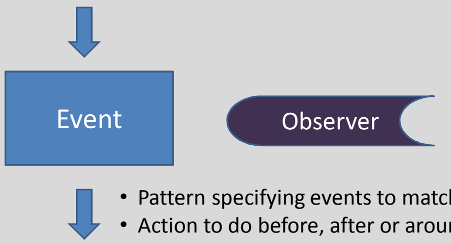
Why AspectMatlab?



- Test the McLab framework for extensibility
- Bring a simple and relevant version of AOP to scientists.

- simple language constructs
- focus on arrays and loops

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

also,
new value

a(i) = b(j,k)

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

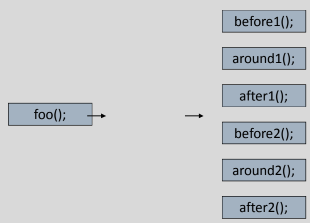
```
actions
   aCountCall : before pCall : (name, args)
      this.count = this.count + 1;
      disp(['calling ',name,' with args(',args, ')']);
   end

   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
```

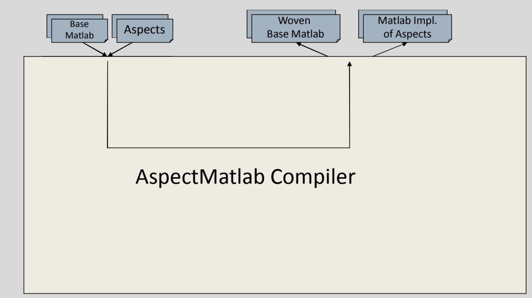
```
actions
   actcall : around pCallFoo : (args)
      % proceed not called, so varargout is set
      varargout{1} = bar(args{1}, args{2});
   end
end
```

Actions Weaving Order

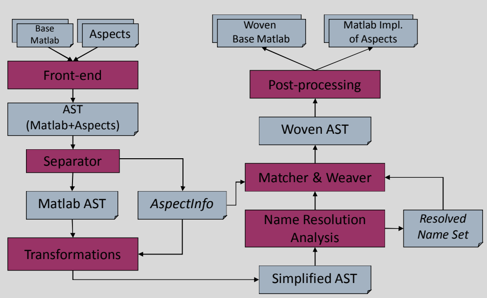



```
actions
  before1 : before pCallFoo
  …
  around1 : around pCallFoo
  …
  after1 : after pCallFoo
  …
  before2 : before pCallFoo
  …
  around2 : around pCallFoo
  …
  after2 : after pCallFoo
  …
end
```


Compiler Structure

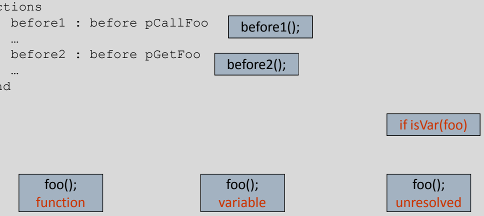


Compiler Structure



Name Resolution Analysis

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



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)*

Conclusions

- McLab supports extensions to MATLAB
- We developed MetaLexer to support modular and extensible lexers, and then used it in McLab.
- We designed and implemented AspectMatlab as an exercise in using McLab for extensions, and also to provide simple and relevant AOP for scientists.

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

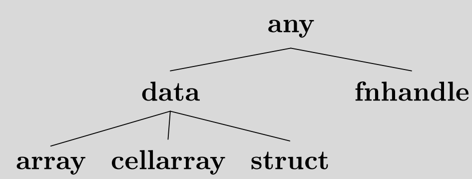
```
function y = sturm(X,BC,F,G,R)
% STURM Solve the Sturm-Liouville equation:
%   d( F*dY/dX )/dX - G*Y = R using linear finite elements.
% INPUT:
%   X - a one-dimensional grid-point array of length N.
%   BC - is a 2 by 3 matrix [A1, B1, C1 ; An, Bn, Cn]
...
% Alex Pletzer: pletzer@pppl.gov (Aug.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

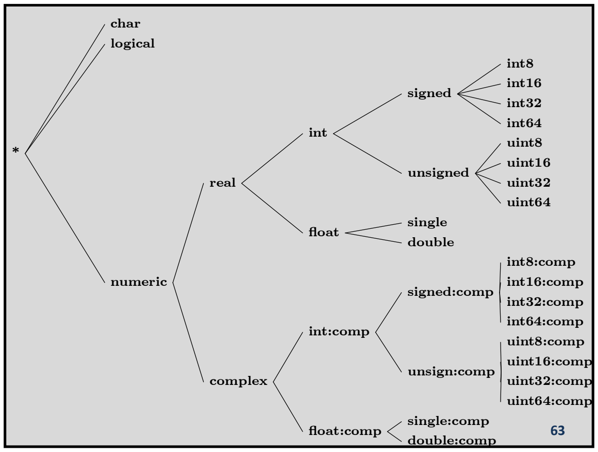
```
>> Ex1(3)
ans = 0.8415    0.9093    0.1411

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

61

High-level types in MATLAB



62

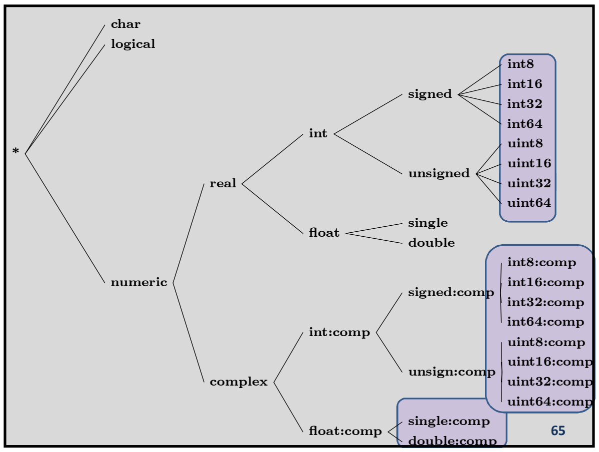


63

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
```

64



65

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

66

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

67