McLab Tutorial
www.sable.mcgill.ca/mclab

Laurie Hendren, Rahul Garg and Nurudeen Lameed

Other McLab team members:
Andrew Casey, Jesse Doherty, Anton Dubrau, Jun Li, Amina Aslam, Toheed Aslam, Maxime Chevalier-Boisvert, Soroush Radpour, Oliver Savary , Maja Frydrychowicz, Clark Verbrugge

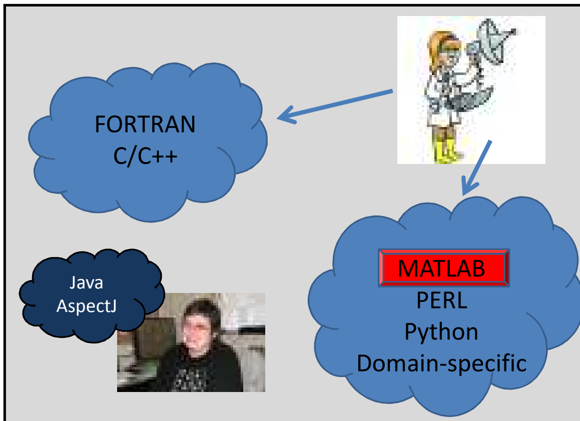
Sable Research Group
School of Computer Science
McGill University, Montreal, Canada

Tutorial Overview

- Why MATLAB?
- Introduction to MATLAB – challenges
- Overview of the McLab tools
  - Introduction to the front-end and extensions
  - IRs, Flow analysis framework and examples
  - Back-ends including the McVM virtual machine
- Wrap-up

Nature Article: "Why Scientific Computing does not compute

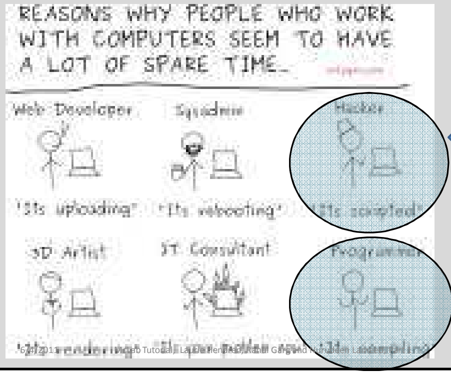
- 38% of scientists spend at least 1/5th of their time programming.
- Codes often buggy, sometimes leading to papers being retracted. Self-taught programmers.
- Monster codes, poorly documented, poorly tested, and often used inappropriately.
- 45% say scientists spend more time programming than 5 years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

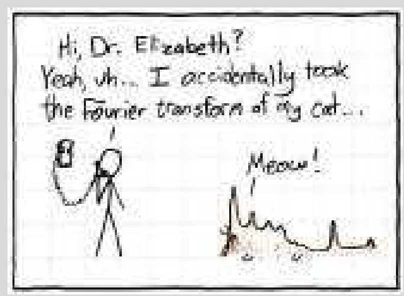
FORTRAN

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

Interpreted ...

Potentially large runtime overhead in both time and space

No types and "flexible" syntax



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

No formal standards for MATLAB

Culture Clash

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, 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" research problems.

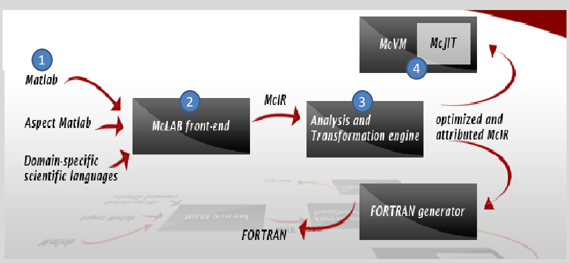
6/4/2011 McLab Tutorial, Laurie Hendren, Rahul Garg and Nurudeen Lameed, Part 1 Intro - 13

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.

6/4/2011 McLab Tutorial, Laurie Hendren, Rahul Garg and Nurudeen Lameed, Part 1 Intro - 14

Overview of McLab/Tutorial



6/4/2011 McLab Tutorial, Laurie Hendren, Rahul Garg and Nurudeen Lameed, Part 1 Intro -15

McLab Tutorial
www.sable.mcgill.ca/mclab

Part 2 – Introduction to MATLAB
- Functions and Scripts
- Data and Variables
- Other Tricky "Features"

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
```


Basic Structure of a MATLAB function (2)

```
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(@sin,3)
a = 0.1080
b = 1.8919

>> [a,b] = ProdSum(@(x)(x),3)
a = 6
b = 6

>> magic(3)
ans = 8 1 6
      3 5 7
      4 9 2

>>ProdSum(ans,3)
ans=96
```


Basic Structure of a MATLAB function (3)

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

```
>> ProdSum([10,20,30],'a')
??? For colon operator with char operands, first and
last operands must be char.
Error in ==> ProdSum at 4
 for i = 1:n

>> ProdSum([10,20,30],i)
Warning: Colon operands must be real scalars.
> In ProdSum at 4
ans = 1

>> ProdSum([10,20,30],[3,4,5])
ans =    6000
```


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)
  - 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
```


Copy Semantics

```
function [ r ] = CopyEx( a, b )
  for i=1:length(a)
    a(i) = sin(b(i));
    c(i) = cos(b(i));
  end
  r = a + c;
end
```

```
>> m = [10, 20, 30]
m = 10   20   30

>> n = 2 * a
n = 20   40   60

>> CopyEx(m,n)
ans = 1.3210    0.0782   -1.2572

>> m = CopyEx(m,n)
m = 1.3210   0.0782  -1.2572
```


Variables and Data in MATLAB

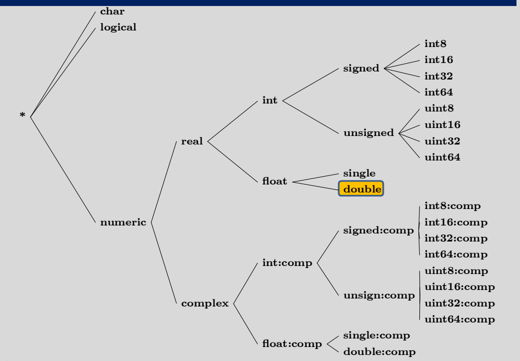
Examples of base types

```
>> clear
>> a = [10, 20, 30]
a = 10   20   30

>> b = int32(a)
b = 10   20   30

>> c = isinteger(b)
c = 1

>> d = complex(int32(4),int32(3))
d = 4 + 3i
```

```
>> whos
Name  Size Bytes  Class    Attributes
  a    1x3   24    double
  b    1x3   12    int32
  c    1x1    1    logical
  d    1x1    8    int32     complex

>> isinteger(c)
ans = 0
>> isnumeric(a)
ans = 1
>> isnumeric(c)
ans = 0
>> isreal(d)
ans = 0
```

6/4/2011 McLab Tutorial, Laurie Hendren, Rahul Garg and Nurudeen Lameed Matlab - 13

MATLAB base data types



6/4/2011 McLab Tutorial, Laurie Hendren, Rahul Garg and Nurudeen Lameed Matlab - 14

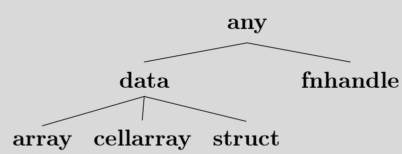
Data Conversions

- double + double → double
- single + double → double
- double:complex + double → double:complex
- int32 + double → int32

- logical + double → error, not allowed
- int16 + int32 → error, not allowed
- int32:complex + int32:complex → error, not defined

6/4/2011 McLab Tutorial, Laurie Hendren, Rahul Garg and Nurudeen Lameed Matlab - 15

MATLAB types: high-level



6/4/2011 McLab Tutorial, Laurie Hendren, Rahul Garg and Nurudeen Lameed Matlab - 16

Cell array and struct example

```
>> students = {'Nurudeen', 'Rahul', 'Jesse'}
students =  'Nurudeen'  'Rahul'  'Jesse'

>> cell = students(1)
cell =   'Nurudeen'

>> contents = students{1}
contents =Nurudeen

>> whos
  Name       Size    Bytes   Class
  cell       1        128     cell
  contents   1x8       16     char
  students   1x3      372     cell
```

```
>> s = struct('name', 'Laurie',
        'student', students)
s = 1x3 struct array with fields:
     name
     student

>> a = s(1)
a = name: 'Laurie'
    student: 'Nurudeen'

>> a.age = 21
a = name: 'Laurie'
    students: 'Nurudeen'
    age:  21
```

6/4/2011 McLab Tutorial, Laurie Hendren, Rahul Garg and Nurudeen Lameed Matlab - 17

Local variables

- Variables are not explicitly declared.
- Local variables are allocated in the current 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.

6/4/2011 McLab Tutorial, Laurie Hendren, Rahul Garg and Nurudeen Lameed Matlab - 18

Global and Persistent Variables

- Variables can be declared to be global.
  - `global x;`
- Persistent declarations are allowed within function bodies only (not allowed in scripts or read-eval-print loop).
  - `persistent y;`
- A persistent or global declaration of x should cover all defs and uses of x in the body of the 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).

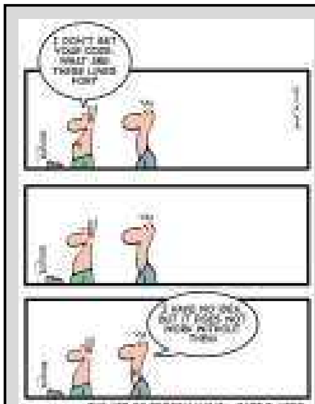
Variable Lookup

- 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.

Local/Global Example

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

```
>> clear

>> global sum

>> sum = 0;

>> ProdSumGlobal([10,20,30],3)
ans = 6000

>> sum
sum = 60

>> whos
  Name    Size    Bytes  Class    Attributes
  ans     1x1     8      double
  sum     1x1     8      double   global
```

Other Tricky "features" in MATLAB

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.

Kind Example

```
function [ r ] = KindEx( a )
  x = a + i + sum(j)
  f = @sin
  eval('s = 10;')
  r = f(x + s)
end
```

```
>> KindEx (3)
x = 3.0000 + 2.0000i
f = @sin
r = 1.5808 + 3.2912i
ans =  1.5808 + 3.2912
```

- VAR: r, a, x, f
- FN: i, j, sum, sin
- ID: s

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

"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

McLab Tutorial
www.sable.mcgill.ca/mclab

Part 3 – McLab Frontend
- Frontend organization
- Introduction to Beaver
- Introduction to JastAdd

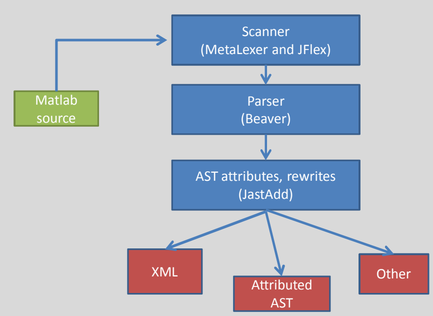
McLab Frontend

- Tools to parse MATLAB-type languages
  - Quickly experiment with language extensions
  - Tested on a lot of real-world Matlab code
- Parser generates ASTs
- Some tools for computing attributes of ASTs
- A number of static analyses and utilities
  - Example: Printing XML representation of AST

Tools used

- Written in Java (JDK 6)
- MetaLexer and JFlex for scanner
- Beaver parser generator
- JastAdd "compiler-generator" for computations of AST attributes
- Ant based builds
- We typically use Eclipse for development
  - Or Vim ☺

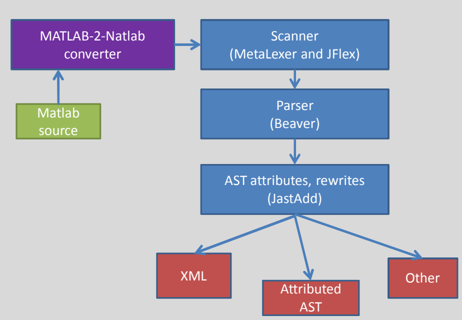
Frontend organization



Natlab

- Natlab is a clean subset of MATLAB
  - Not a trivial subset though
  - Covers a lot of "sane" MATLAB code
- MATLAB to Natlab translation tool available
  - Written using ANTLR
  - Outside the scope of this tutorial
- Forms the basis of much of our semantics and static analysis research

Frontend with MATLAB-to-Natlab



How is Natlab organized?

- Scanner specifications
  - src/metalexer/shared_keywords.mlc
- Grammar files
  - src/parser/natlab.parser
- AST computations based on JastAdd
  - src/natlab.ast
  - src/*jadd, src/*jrag
- Other Java files
  - src/*java

MetaLexer

- A system for writing extensible scanner specifications
- Scanner specifications can be modularized, reused and extended
- Generates JFlex code
  - Which then generates Java code for the lexer/scanner
- Syntax is similar to most other lexers
- Reference: "MetaLexer: A Modular Lexical Specification Language. Andrew Casey, Laurie Hendren" by Casey, Hendren at AOSD 2011.

If you already know Beaver and JastAdd…

Then take a break.
Play Angry Birds.
Or Fruit Ninja.

Beaver

- Beaver is a LALR parser generator
- Familiar syntax (EBNF based)
- Allows embedding of Java code for semantic actions
- Usage in Natlab: Simply generate appropriate AST node as semantic action

Beaver Example

```
Stmt stmt =
        expr.e {: return new ExprStmt(e); :}
|       BREAK {: return new BreakStmt(); :}
|       FOR   for_assign.a  stmt_seq.s END
                    {: return new ForStmt(a,s); :}
```


Beaver Example

Java type

```
Stmt stmt =
        expr.e {: return new ExprStmt(e); :}
|       BREAK {: return new BreakStmt(); :}
|       FOR   for_assign.a  stmt_seq.s END
                    {: return new ForStmt(a,s); :}
```

Beaver Example

Node name in grammar

```
Stmt stmt =
        expr.e {: return new ExprStmt(e); :}
|       BREAK {: return new BreakStmt(); :}
|       FOR   for_assign.a  stmt_seq.s END
                    {: return new ForStmt(a,s); :}
```


Beaver Example

Identifier for node

```
Stmt stmt =
        expr.e {: return new ExprStmt(e); :}
|       BREAK {: return new BreakStmt(); :}
|       FOR   for_assign.a  stmt_seq.s END
                    {: return new ForStmt(a,s); :}
```


Beaver Example

Java code for semantic action

```
Stmt stmt =
        expr.e {: return new ExprStmt(e); :}
|       BREAK {: return new BreakStmt(); :}
|       FOR   for_assign.a  stmt_seq.s END
                    {: return new ForStmt(a,s); :}
```


JastAdd: Motivation

- You have an AST
- Each AST node type represented by a class
- Want to compute attributes of the AST
  - Example: String representation of a node
- Attributes might be either:
  - Inherited from parents
  - Synthesized from children

JastAdd

- JastAdd is a system for specifying:
  - Each attribute computation specified as an aspect
  - Attributes can be inherited or synthesized
  - Can also rewrite trees
  - Declarative philosophy
  - Java-like syntax with added keywords
- Generates Java code
- Based upon "Reference attribute grammars"

How does everything fit?

- JastAdd requires two types of files:
  - .ast file which specifies an AST grammar
  - .jrag/.jadd files which specify attribute computations
- For each node type specified in AST grammar:
  - JastAdd generates a class derived from ASTNode
- For each aspect:
  - JastAdd adds a method to the relevant node classes

JastAdd AST File example

```
abstract BinaryExpr: Expr ::=
        LHS:Expr RHS:Expr
PlusExpr: BinaryExpr;
MinusExpr: BinaryExpr;
MTimesExpr: BinaryExpr;
```

6/4/2011 McLab Tutorial, Laurie Hendren, Rahul Garg and Nurudeen Lameed Frontend-19

JastAdd XML generation aspect

```
aspect AST2XML{
..
eq BinaryExpr.getXML(Document d, Element e){
        Element v = d.getElement(nameOfExpr);
        getRHS().getXML(d,v);
        getLHS().getXML(d,v);
        e.add(v);
        return true;
}
…
```

6/4/2011 McLab Tutorial, Laurie Hendren, Rahul Garg and Nurudeen Lameed Frontend-20

Aspect declaration

```
aspect AST2XML{
..
eq BinaryExpr.getXML(Document d, Element e){
        Element v = d.getElement(nameOfExpr);
        getRHS().getXML(d,v);
        getLHS().getXML(d,v);
        e.add(v);
        return true;
}
…
```

6/4/2011 McLab Tutorial, Laurie Hendren, Rahul Garg and Nurudeen Lameed Frontend-21

“Equation” for an attribute

```
aspect AST2XML{
eq BinaryExpr.getXML(Document d, Element e){
        Element v = d.getElement(nameOfExpr);
        getRHS().getXML(d,v);
        getLHS().getXML(d,v);
        e.add(v);
        return true;
}
…
```

6/4/2011 McLab Tutorial, Laurie Hendren, Rahul Garg and Nurudeen Lameed Frontend-22

Add to this AST class

```
aspect AST2XML{
..
eq BinaryExpr.getXML(Document d, Element e){
        Element v = d.getElement(nameOfExpr);
        getRHS().getXML(d,v);
        getLHS().getXML(d,v);
        e.add(v);
        return true;
}
…
```

6/4/2011 McLab Tutorial, Laurie Hendren, Rahul Garg and Nurudeen Lameed Frontend-23

Method name to be added

```
aspect AST2XML{
..
eq BinaryExpr.getXML(Document d, Element e){
        Element v = d.getElement(nameOfExpr);
        getRHS().getXML(d,v);
        getLHS().getXML(d,v);
        e.add(v);
        return true;
}
…
```

6/4/2011 McLab Tutorial, Laurie Hendren, Rahul Garg and Nurudeen Lameed Frontend-24

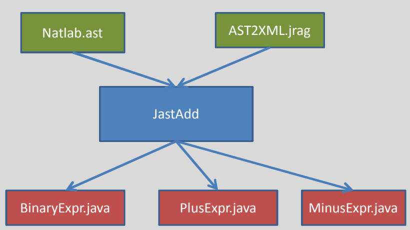
```
aspect AST2XML{
..
eq BinaryExpr.getXML(Document d, Element e){
        Element v = d.getElement(nameOfExpr);
        getRHS().getXML(d,v);
        getLHS().getXML(d,v);
        e.add(v);
        return true;
}
…
```

Attributes can be parameterized


```
aspect AST2XML{
..
eq Binar            ocument d, Element e){
        Ele              Element(nameOfExpr);
        getRHS().getXML(d,v);
        getLHS().getXML(d,v);
        e.add(v);
        return true;
}
…
```

Compute for children

JastAdd weaving



Overall picture recap

- Scanner converts text into a stream of tokens
- Tokens consumed by Beaver-generated parser
- Parser constructs an AST
- AST classes were generated by JastAdd
- AST classes already contain code for computing attributes as methods
- Code for computing attributes was weaved into classes by JastAdd from aspect files

Adding a node

- Let's assume you want to experiment with a new language construct:
- Example: parallel-for loop construct
  - parfor i=1:10 a(i) = f(i) end;
- How do you extend Natlab to handle this?
- You can either:
  - Choose to add to Natlab source itself
  - (Preferred) Setup a project that inherits code from Natlab source directory

Steps

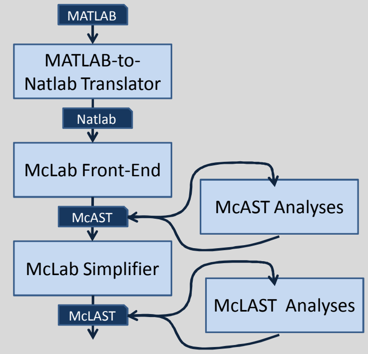
- Write the following in your project:
  - Lexer rule for "parfor"
  - Beaver grammar rule for parfor statement type
  - AST grammar rule for PforStmt
  - attributes for PforStmt according to your requirement
  - eg. getXML() for PforStmt in a JastAdd aspect
  - Buildfile that correctly passes the Natlab source files and your own source files to tools
  - Custom main method and jar entrypoints

McLab Tutorial
www.sable.mcgill.ca/mclab

Part 4 – McLab Intermediate Representations

- High-level McAST
- Lower-level McLAST
- Transforming McAST to McLAST

Big Picture



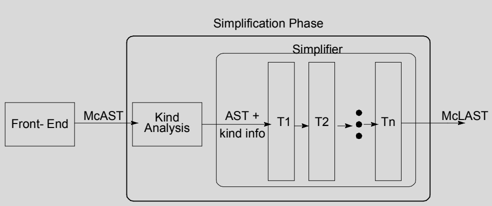
McAST

- High-level AST as produced from the front-end.
- AST is implemented via a collection of Java classes generated from the JastAdd specification file.
- Fairly complex to write a flow analysis for McAST because of:
  - arbitaly complex expressions, especially lvalues
  - ambiguous meaning of parenthesized expressions such as a(i)
  - control-flow embedded in expressions (&&, &, ||, |)
  - MATLAB-specific issues such as the "end" expression and returning multiple values.

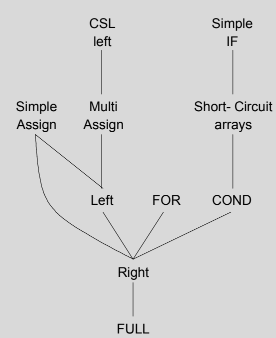
McLAST

- Lower-level AST which:
  - has simpler and explicit control-flow;
  - simplifies expressions so that each expression has a minimal amount of complexity and fewer ambiguities; and
  - handles MATLAB-specific issues such as "end" and comma-separated lists in a simple fashion.
- Provides a good platform for more complex flow analyses.

Simplification Process



Dependences between simplifications



Expression Simplification

Aim: create simple expressions with at most one operator and simple variable references.

```
foo(x) + a(y(i))   →   t1 = foo(x);
                        t2 = y(i);
                        t3 = a(t2);
                        t1 + t3
```

Aim: specialize parameterized expression nodes to array indexing or function call.

6/4/2011 McLab Tutorial, Laurie Hendren, Rahul Garg and Nurudeen Lameed, Part 4 IR - 7

Short-circuit simplifications

- && and || are always short-circuit
- & and I are **sometimes** short-circuit
  - if (exp1 & exp2) is short-circuit
  - t = exp1 & exp2 is not short-circuit
- replace short-circuit expressions with explicit control-flow

6/4/2011 McLab Tutorial, Laurie Hendren, Rahul Garg and Nurudeen Lameed, Part 4 IR - 8

"end" expression simplification

Aim: make "end" expressions explicit, extract from complex expressions.

```
A(2,f(end))  →  A(2,f(EndCall(A,2,2)))
                   →
                     t1 = EndCall(A,2,2);
                     t2 = f(t1);
                     A(2,t2)
```

6/4/2011 McLab Tutorial, Laurie Hendren, Rahul Garg and Nurudeen Lameed, Part 4 IR - 9

L-value Simplification

Aim: create simple l-values.

```
A(a+b,2).e(foo()) = value;   →   t1 = a+b;
                                  t2 = foo();
                                  A(t1,2).e(t2) = value;
```

Note: no mechanism for taking the address of location in MATLAB. Further simplification not possible, while still remaining as valid MATLAB.

6/4/2011 McLab Tutorial, Laurie Hendren, Rahul Garg and Nurudeen Lameed, Part 4 IR - 10

if statement simplification

Aim: create if statements with only two control flow paths.

```
if E1              →   if E1
  body1();                body1();
elseif E2              else
  body2();                if E2
else                        body2();
  body3();                else
end                         body3();
                          end
                        end
```

6/4/2011 McLab Tutorial, Laurie Hendren, Rahul Garg and Nurudeen Lameed, Part 4 IR - 11

for loop simplification

Aim: create for loops that iterate over a variable incremented by a fixed constant.

```
for i = 1:2:n
  % BODY
end
```

```
for i = E        →   t1=E;
  % BODY             t2=size(t1);
end                  t3=prod(t2(2:end));
                     for t4 = 1:t3
                       i = t1(t4);
                       % BODY
                     end
```

6/4/2011 McLab Tutorial, Laurie Hendren, Rahul Garg and Nurudeen Lameed, Part 4 IR - 12

McLab Tutorial
www.sable.mcgill.ca/mclab

Part 5 – Introduction to the McLab Analysis Framework

•Exploring the Main Components

•Creating a Simple Analysis

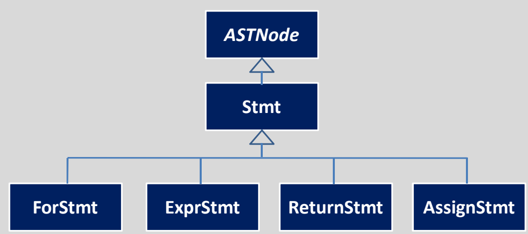
•Depth-first and Structural Analyses

•Example: Reaching Definition Analysis

McLab Analysis Framework

- A simple static flow analysis framework for MATLAB-like languages
- Supports the development of intra-procedural forward and backward flow analyses
- Extensible to new language extensions
- Facilitates easy adaptation of old analyses to new language extensions
- Works with McAST and McLAST (a simplified McAST)

McAST & Basic Traversal Mechanism



- Traversal Mechanism:
  - Depth-first traversal
  - Repeated depth-first traversal

Exploring the main components for developing analyses

The interface *NodeCaseHandler*

- Declares all methods for the action to be performed when a node of the AST is visited:

```
public interface NodeCaseHandler {
   void caseStmt(Stmt node);
   void caseForStmt(ForStmt node);
   void caseWhileStmt(WhileStmt node);
   …
}
```


The class *AbstractNodeCaseHandler*

```
public class AbstractNodeCaseHandler implements
   NodeCaseHandler {
   …
   void caseStmt(Stmt node) {
        caseASTNode(node);
   }
   …
}
```

- Implements the interface *NodeCaseHandler*
- Provides default behaviour for each AST node type except for the root node (*ASTNode*)

The analyze method

- Each AST node also implements the method *analyze* that performs an analysis on the node:

```
public void analyze(NodeCaseHandler handler)
        handler.caseAssignStmt(this);
}
```


Creating a simple analysis

Creating a Traversal/Analysis:

- Involves 3 simple steps:

1. Create a concrete class by extending the class *AbstractNodeCaseHandler*
2. Provide an implementation for *caseASTNode*
3. Override the relevant methods of *AbstractNodeCaseHandler*

An Example: StmtCounter

- Counts the number of statements in an AST

Analysis development Steps:

1. Create a concrete class by extending the class *AbstractNodeCaseHandler*
2. Provide an implementation for *caseASTNode*
3. Override the relevant methods of *AbstractNodeCaseHandler*

An Example: StmtCounter

1. Create a concrete class by extending the class *AbstractNodeCaseHandler*

```
public class StmtCounter extends
   AbstractNodeCaseHandler {
   private int count = 0;
   … // defines other internal methods
}
```


An Example: StmtCounter --- Cont’d

2. Provide an implementation for *caseASTNode*

```
public void caseASTNode( ASTNode node){
   for(int i=0; i<node.getNumChild(); ++i) {
     node.getChild(i).analyze(this);
   }
}
```

An Example: StmtCounter --- Cont'd

3. Override the relevant methods of *AbstractNodeCaseHandler*

```
public void caseStmt(Stmt node) {
    ++count;
    caseASTNode(node);
}
```


An Example: StmtCounter --- Cont'd

```
public class StmtCounter extends AbstractNodeCaseHandler {
    private int count = 0;
    private StmtCounter() { super(); }
    public static int countStmts(ASTNode tree) {
        tree.analyze(new StmtCounter());
    }
    public void caseASTNode( ASTNode node){
        for(int i=0; i<node.getNumChild(); ++i) {
                node.getChild(i).analyze(this);}
    }
    public void caseStmt(Stmt node) {
        ++count; caseASTNode(node);
    }
}
```


Tips: Skipping Irrelevant Nodes

For many analyses, not all nodes in the AST are relevant; to skip unnecessary nodes override the handler methods for the nodes. For Example:

```
public void caseExpr(Expr node) {
    return;
}
```

Ensures that all the children of *Expr* are skipped

Analyses Types: Depth-first and Structural Analyses

Flow Facts: The interface *FlowSet*

- The interface *FlowSet* provides a generic interface for common operations on flow data

```
public  interface FlowSet<D> {
  public FlowSet<D> clone();
  public void copy(FlowSet<? super D> dest);
  public void union(FlowSet<? extends D> other);
  public void intersection(FlowSet<? extends D> other);
    …
}
```

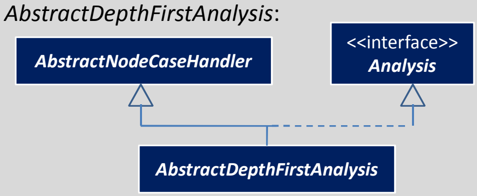

The *Analysis* interface

- Provides a common API for all analyses
- Declares additional methods for setting up an analysis:

```
public interface Analysis<A extends FlowSet> extends
    NodeCaseHandler {
    public void analyze();
    public ASTNode getTree();
    public boolean isAnalyzed();
    public A newInitialFlow();
    …
}
```

Depth-First Analysis

- Traverses the tree structure of the AST by visiting each node in a depth-first order
- Suitable for developing flow-insensitive analyses
- Default behavior implemented in the class *AbstractDepthFirstAnalysis*:



Creating a Depth-First Analysis:

- Involves 2 steps:

1. Create a concrete class by extending the class *AbstractDepthFirstAnalysis*
   a) Select a type for the analysis's data
   b) Implement the method *newInitialFlow*
   c) Implement a constructor for the class

2. Override the relevant methods of *AbstractDepthFirstAnalysis*

Depth-First Analysis: NameCollector

- Associates all names that are assigned to by an assignment statement to the statement.
- Collects in one set, all names that are assigned to
- Names are stored as strings; we use *HashSetFlowSet<String>* for the analysis's flow facts.
- Implements *newInitialFlow* to return an empty *HashSetFlowSet<String>* object.

Depth-First Analysis: NameCollector --- Cont'd

1. Create a concrete class by extending the class *AbstractDepthFirstAnalysis*

```
public class NameCollector extends
    AbstractDepthFirstAnalysis
    <HashSetFlowSet<String>> {
    private int HashSetFlowSet<String> fullSet;

    public NameCollector(ASTNode tree) {
        super(tree); fullSet = newInitialFlow();
    }
    … // defines other internal methods
}
```


Depth-First Analysis: NameCollector --- Cont'd

2. Override the relevant methods of *AbstractDepthFirstAnalysis*

```
private boolean inLHS = false;

public void caseName(Name node) {
    if (inLHS)
        currentSet.add(node.getID());
}
```


Depth-First Analysis: NameCollector --- Cont'd

2. Override the relevant methods of *AbstractDepthFirstAnalysis*

```
public void caseAssignStmt(AssignStmt node) {
    inLHS = true;
    currentSet = newInitialFlowSet();
    analyze(node.getLHS());
    flowSets.put(node, currentSet);
    fullSet.addAll(currentSet);
    inLHS = false;
}
```

Depth-First Analysis: NameCollector --- Cont'd

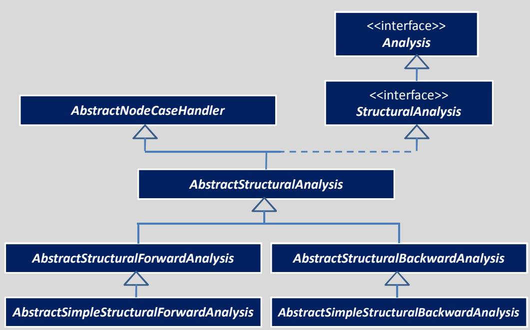
2. Override the relevant methods of *AbstractDepthFirstAnalysis*

```
public void caseParameterizedExpr
(ParameterizedExpr node) {
    analyze(node.getTarget());
}
…
```


Structural Analysis

- Suitable for developing flow-sensitive analyses
- Computes information to approximate the runtime behavior of a program.
- Provides mechanism for:
  - analyzing control structures such as *if-else*, *while* and *for* statements;
  - handling *break* and *continue* statements
- Provides default implementations for relevant methods
- May be forward or backward analysis

Structural Analysis Class Hierarchy



The interface *StructuralAnalysis*

- Extends the *Analysis* interface
- Declares more methods for structural type analysis:

```
public interface StructuralAnalysis<A extends
    FlowSet> extends Analysis<A> {
    public Map<ASTNode, A> getOutFlowSets();
    public Map<ASTNode, A> getInFlowSets();
    public void merge(A in1, A in2, A out);
    public void copy(A source, A dest);
    …
}
```


Developing a Structural Analysis

- Involves the following steps:
  1. Select a representation for the analysis's data
  2. Create a concrete class by extending the class: *AbstractSimpleStructuralForwardAnalysis* for a forward analysis and *AbstractSimpleStructuralBackwardAnalysis* for a backward analysis
  3. Implement a suitable constructor for the analysis and the method *newInitialFlow*
  4. Implement the methods *merge* and *copy*
  5. Override the relevant node case handler methods and other methods

Example: Reaching Definition Analysis

Example: Reaching Definition Analysis

For every statement *s*, for every variable *v* defined by the program, compute the set of all definitions or assignment statements that assign to *v* and that *may* reach the statement *s*

A definition *d* for a variable *v* reaches a statement *s*, if there exists a path from *d* to *s* and *v* is not re-defined along that path.

Reach Def Analysis: An Implementation Step 1

Select a representation for the analysis's data:

HashMapFlowSet<String, Set<ASTNode>>

We use a map for the flow data: An entry is an ordered pair (*v*, *defs*)

where *v* denotes a variable and

defs denotes the set of definitions for *v* that may reach a given statement.

Reach Def Analysis: An Implementation Step 2

Create a concrete class by extending the class: *AbstractSimpleStructuralForwardAnalysis* for a forward analysis:

```
public class ReachingDefs extends
   AbstractSimpleStructuralForwardAnalysis
  <HashMapFlowSet<String, Set<ASTNode>>> {
  …
}
```


Reach Def Analysis: An Implementation Step 3

Implement a suitable constructor and the method *newInitialFlow* for the analysis:

```
public ReachingDefs(ASTNode tree) {
   super(tree);
   currentOutSet = newInitialFlow(); }

public HashMapFlowSet<String, Set<ASTNode>>
   newInitialFlow() {
   return new
   HashMapFlowSet<String,Set<ASTNode>>(); }
```


Reach Def Analysis: An Implementation Step 4a

Implement the methods *merge* and *copy*:

```
public void merge
(HashMapFlowSet<String, Set<ASTNode>> in1,
   HashMapFlowSet<String, Set<ASTNode>> in2,
   HashMapFlowSet<String, Set<ASTNode>> out) {
        union(in1, in2, out);
 }
public void
copy(HashMapFlowSet<String, Set<ASTNode>> src,
    HashMapFlowSet<String, Set<ASTNode>> dest) {
    src.copy(dest);
}
```


Reach Def Analysis: An Implementation Step 4b

```
public void
union (HashMapFlowSet<String, Set<ASTNode>> in1,
   HashMapFlowSet<String, Set<ASTNode>> in2,
   HashMapFlowSet<String, Set<ASTNode>> out) {
   Set<String> keys = new HashSet<String>();
    keys.addAll(in1.keySet()); keys.addAll(in2.keySet());
    for (String v: keys) {
      Set<ASTNode> defs = new HashSet<ASTNode>();
      if (in1.containsKey(v)) defs.addAll(in1.get(v));
      if (in2.containsKey(v)) defs.addAll(in2.get(v));
      out.add(v, defs);
   }
}
```

Reach Def Analysis: An Implementation Step 5a

Override the relevant node case handler methods and other methods :

override caseAssignStmt(AssignStmt node)

```
public void caseAssignStmt(AssignStmt node) {
    inFlowSets.put(node, currentInSet.clone() );
    currentOutSet =
        new HashMapFlowSet<String, Set<ASTNode> > ();

    copy(currentInSet, currentOutSet);
    HashMapFlowSet<String, Set<ASTNode>> gen =
        new  HashMapFlowSet<String, Set<ASTNode>> ();
    HashMapFlowSet<String, Set<ASTNode> > kill =
        new HashMapFlowSet<String, Set<ASTNode>> ();
```


Reach Def Analysis: An Implementation Step 5b

```
// compute out = (in - kill) + gen
  // compute kill
    for( String s : node.getLValues() )
      if (currentOutSet.containsKey(s))
        kill.add(s, currentOutSet.get(s));
      // compute gen
    for( String s : node.getLValues()){
      Set<ASTNode> defs = new HashSet<ASTNode>();
      defs.add(node);
      gen.add(s, defs);
    }
```


Reach Def Analysis: An Implementation Step 5c

```
    // compute (in - kill)
    Set<String> keys = kill.keySet();
    for (String s: keys)
      currentOutSet.removeByKey(s);
    // compute (in - kill) + gen
    currentOutSet = union(currentOutSet, gen);

    // associate the current out set to the node
    outFlowSets.put( node, currentOutSet.clone() );
}
```

McLab Tutorial
www.sable.mcgill.ca/mclab

Part 6 – Introduction to the McLab Backends

- MATLAB-to-MATLAB
- MATLAB-to-Fortran90 (McFor)
- McVM with JIT

MATLAB-to-MATLAB

- We wish to support high-level transformations, as well as refactoring tools.
- Keep comments in the AST.
- Can produce .xml or .m files from McAST or McLAST.
- Design of McLAST such that it remains valid MATLAB, although simplified.

MATLAB-to-Fortran90

- MATLAB programmers often want to develop their prototype in MATLAB and then develop a FORTRAN implementation based on the prototype.
- 1st version of McFOR implemented by Jun Li as M.Sc. thesis.
  - handled a smallish subset of MATLAB
  - gave excellent performance for the benchmarks handled
  - provided good insights into the problems needed to be solved, and some good initial solutions.
- 2nd version of McFOR currently under development.
  - fairly large subset of MATLAB, more complete solutions
  - provide a set of analyses, transformations and IR simplifications that will likely be suitable for both the FORTRAN generator, as well as other HLL.
- e-mail hendren@cs.mcgill.ca to be put on the list of those interested in McFor.

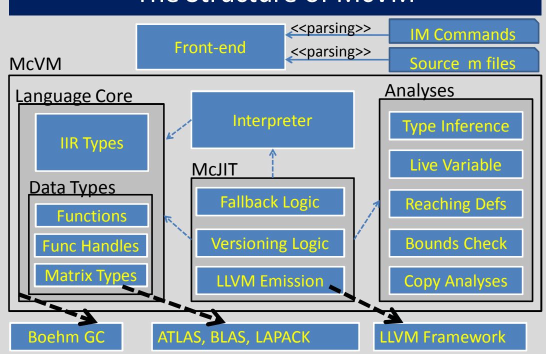
McVM-McJIT

- Whereas the other back-ends are based on static analyses and ahead-of-time compilation, the dynamic nature of MATLAB makes it more suitable for a VM/JIT.
- MathWorks' implementation does have a JIT, although technical details are not known.
- McVM/McJIT is an open implementation aimed at supporting research into dynamic optimization techniques for MATLAB.

McVM Design

- A basic but fast interpreter for the MATLAB language
- A garbage-collected JIT Compiler as an extension to the interpreter
- Easy to add new data types and statements by modifying only the interpreter.
- Supported by the LLVM compiler framework and some numerical computing libraries.
- Written entirely in C++; interface with the McLab front-end via a network port.

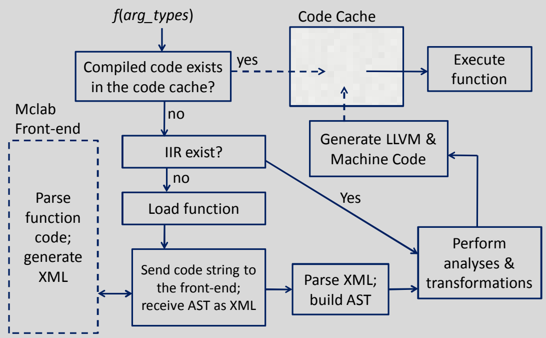
The Structure of McVM



Supported Types

- Logical Arrays
- Character Arrays
- Double-precision floating points
- Double-precision complex number matrices
- Cell arrays
- Function Handles

McJIT: Executing a Function



Type Inference

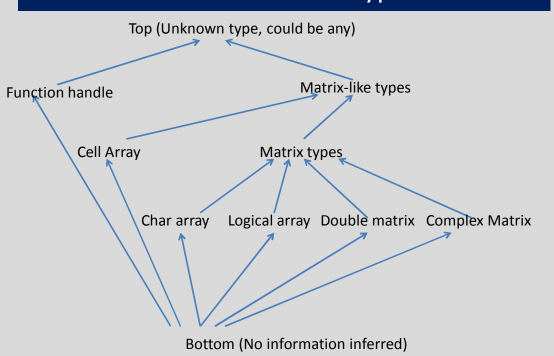
- It is a key performance driver for the JIT Compiler:
  - the type information provided are used by the JIT compiler for function specialization.

Type Inference

- It is a forward flow analysis: propagates the set of possible types through every possible branch of a function.
- Assumes that:

  for each input argument *arg*, there exist some possible types
- At every program point *p*, infers the set of possible types for each variable
- May generate different results for the same function at different times depending on the types of the input arguments

Lattice of McVM types



Internal Intermediate Representation

- A simplified form of the Abstract Syntax Tree (AST) of the original source program
- It is machine independent
- All IIR nodes are garbage collected

IIR: A Simple MATLAB Program

.m file

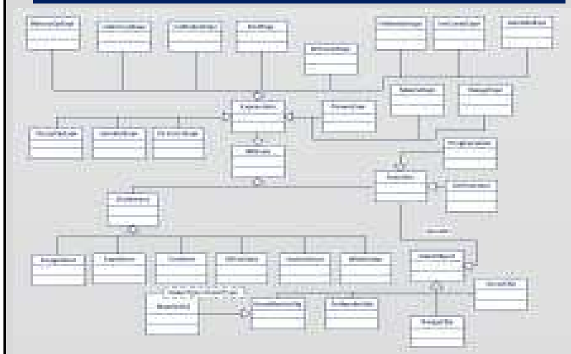
```
function a = test(n)
  a =  zeros(1,n);
  for i = 1:n
    a(i) = i*i;
  end
end
```

IIR form

```
function [a] = test(n)
  a = zeros(1, n);
  $t1 = 1; $t0 = 1;
  $t2 = $t1; $t3 = n;
  while True
    $t4 = ($t0 <= $t3);
    if ~$t4
      break;
    end
    i = $t0;
    a(i) = (i * i);
    $t0 = ($t0 + $t2);
  end
end
```

6/4/2011 McLab Tutorial, Laurie Hendren, Rahul Garg and Nurudeen Lameed Backends- 13

McVM Project Class Hierarchy (C++ Classes)



6/4/2011 McLab Tutorial, Laurie Hendren, Rahul Garg and Nurudeen Lameed Backends- 14

Running McVM

```
bear:~/mcvm2.8/mclab/mcvm-llvm2.8/debug> ./mcvm -jit_enable true -start_dir ~/pldi11_mclabtutorial/
*****************************************************
        McVM - The McLab Virtual Machine v1.0
Visit http://www.sable.mcgill.ca for more information.
*****************************************************

>: c = test(10);
Compiling function: "test"
>: c
ans =
matrix of size 1x10
       1       4       9      16      25      36      49      64      81     100

>:
```

6/4/2011 McLab Tutorial, Laurie Hendren, Rahul Garg and Nurudeen Lameed Backends- 15

McLab Tutorial
www.sable.mcgill.ca/mclab

Part 7 – McVM implementation example: if/else construct

- Implementation in interpreter
- Implementation in JIT compiler

Before we start

- McVM is written in C++, but "clean" C++ ☺
- Nearly everything is a class
- Class names start in capital letters
- Typically one header and one implementation file for each class
- Method names are camel cased (getThisName)
- Members are usually private and named m_likeThis

Before we start …

- Makefile provided
  - Handwritten, very simple to read or edit
- Scons can also be used
- ATLAS/CLAPACK is not essential. Alternatives:
  - Intel MKL, AMD ACML, any CBLAS + Lapacke (eg. GotoBLAS2 + Lapacke)
- Use your favourite development tool
  - I use Eclipse CDT, switched from Vim
- Virtualbox image with everything pre-installed available on request for private use

Implementing if/else in McVM

1. A new class to represent if/else
2. XML parser
3. Loop simplifier
4. Interpreter
5. Various analysis
   i. Reach-def, live variable analysis
   ii. Type checking
6. Code generation

1. A class to represent If/Else

- Class IfElseStmt
- We will derive this class from "Statement"
- Form two files: ifelsestmt.h and ifelsestmt.cpp
- Need fields to represent:
  - Test expression
  - If body
  - Else body

Ifelsestmt.h

- class IfElseStmt: public Statement
- Methods:
  - copy(), toString(), getSymbolUses(), getSymbolDefs()
  - getCondition(), getIfBlock(), getElseBlock()
- Private members:
  - Expression *m_pCondition;
  - StmtSequence *m_pIfBlock;
  - StmtSequence *m_pElseBlock;

Modify statements.h

- Each statement has a field called m_type
- This contains a type tag
- Tag used throughout compiler for switch/case
- enum StmtType{

```
    IF_ELSE,
    SWITCH,
    FOR,
    ….
};
```


2. Modify XML Parser

- Look in parser.h, parser.cpp
- Before anything happens, must parse from XML generated by frontend
- XML parser is a simple recursive descent parser
- Add a case to parseStmt()
  - Look at the element name in the XML
  - If it is "IfStmt", it is a If/Else
- Write a parseIfStmt() function

3. Modify transform loops

- McVM simplifies for-loops to a lower level construct
- To achieve this, we need to first find loops
- Done via a depth first search in the tree
- So add a case to this search to say:
  - Search in the if block
  - Search in the else block
  - Return
- transform_loops.cpp

4. Add to interpreter

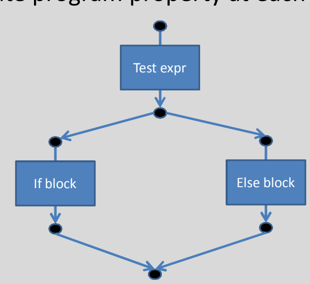
- Always implement in interpreter before implementing in JIT compiler
- It is a simple evaluator: no byte-code tricks, no direct-threaded dispatch etc.
- Add a case to statement evaluation:
  - Evaluate test condition
  - If true, evaluate if block
  - If false, evaluate else block
- interpreter.cpp :
  - Case in execStatement()
  - Calls evalIfElseStmt()

Moment of silence .. Or review

- At this point, if/else has been implemented in the interpreter
- If you don't enable JIT compilation, then you can now run if/else
- Good checkpoint for testing and development

Flow analysis recap

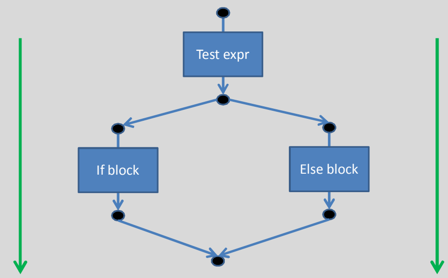
- Compute program property at each program point



Flow analysis recap

- We want to compute property at each program point
- Typically want to compute a map of some kind at each program point
- Program points are not inside statements, but just before and after
- Usually unions computed at join points
- Can be forward or backwards depending on the analysis

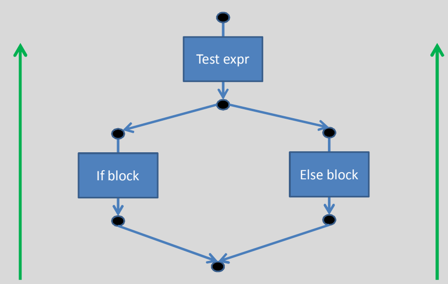
Reaching definitions analysis



McVM reach-defs analysis

- Look in analysis_reachdefs (.h/.cpp)
- getReachDefs() is an overloaded function to compute reach-defs
- ReachDefInfo class to store analysis info
- If/Else:
  – Record reach-defs for test expression
  – Compute reach-defs for if and else blocks by calling getReachDefs() for StmtSequence
  – Compute union at post-if/else point

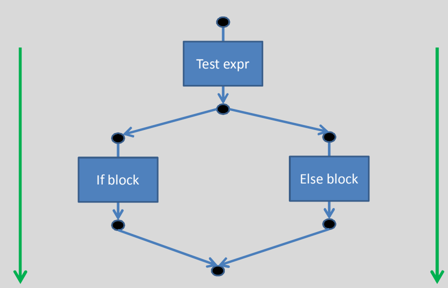
Live variable analysis



McVM live vars analysis

- Look in analysis_livevars (.h/.cpp)
- getLiveVars() is an overloaded function
- LiveVarInfo is a class to store live-vars info
- If/Else:
  – Information flows backwards from post-if/else
  – Flow live-vars through the if and else blocks
  – Compute union at post-test expression
  – Record live-vars info of test expression

Type inference analysis



Type inference

- Look in analysis_typeinfer (.h/.cpp)
- inferTypes() is an overloaded function to perform type inference for most node-types
- For If/else:
  - Infer type of test expression
  - Infer type of if and else blocks
  - Merge information at post-if/else point

Flow analysis tips

- We define a few typedefs for data structures like maps, sets
  - eg: VarDefSet: typedef of set of IIRNode* with appropriate comparison operators and allocator
- When trying to understand flow analysis code, start from code for assignment statements
- Pay attention to statements like return and break

Code generation and LLVM

- LLVM is based upon a typed SSA representation
- LLVM can either be accessed through a C++ API, or you can generate LLVM byte-code directly
- We use the C++ API
- Much of the complexity of the code generator due to SSA representation required by LLVM
- However, we don't do an explicit SSA conversion pass

Code generation in McVM

- SSA conversion is not explicitly represented in the IR
- SSA conversion done while doing code generation
- Assignment instructions are usually not generated directly if Lvalue is a symbol
- In SSA form, values of expressions are important, not what they are assigned to
- We store mapping of symbols to values in an execution environment

Compiling if/else

- Four steps:
  - Compile test expression
  - Compile if block (compStmtSeq)
  - Compile else block (compStmtSeq)
  - Call matchBranchPoints() to do appropriate SSA book-keeping at merge point
- Rest of the code is book-keeping for LLVM
- Such as forming proper basic blocks when required

McLab Tutorial
www.sable.mcgill.ca/mclab

Part 8 – Wrap Up

- Summary
- Ongoing and Future Work
- Further Sources

Tutorial Summary

- MATLAB is a popular language and an important PLDI research area.
- McLab aims to provide tools to support such research.
  - Front-end: extensible scanner, parser, attributes
    - example extension: AspectMatlab
  - IR and analysis framework:
    - two levels of IR, high-level McAST and lower-level McLAST
    - structure-based flow analysis framework
  - Back-ends: MATLAB, McVM with McJIT and McFor

Ongoing and Future Work

- MATLAB refactoring tools:
  - code cleanup
  - refactoring towards Fortran generation
  - include static call graph and interprocedural analysis framework
- MATLAB extensions:
  - AspectMatlab
  - Typing Aspects

Back-end (McVM/McJIT)

- On-stack replacement
- Dynamic optimizations – correct choice of inlining and basic block positioning.
- Optimizations for multicore systems
- Compilation to GPUs and mixed CPU/GPU systems
- Portability and performance across multiple CPU and GPU families

Where to look for more info

- www.sable.mcgill.ca
  - /software
    - currently have McVM and AspectMatlab on the web site
    - can ask for McLab front-end and analysis framework, we will also add to the web site soon
  - /publications
    - papers and thesis, in particular
    - MetaLexer (Andrew Casey)
    - McLab Front-end and Analysis Framework (Jesse Doherty)
    - McVM (Maxime Chevalier-Boisvert)
    - McFor (1st version Jun Li, 2nd version Anton Dubrau)
    - tutorials, starting with this one

Keep in Touch

- main web site:
http://www.sable.mcgill.ca/mclab
- mailing list:
mclab-list@sable.mcgill.ca
- bug reports:
https://svn.sable.mcgill.ca/mclab-bugzilla/
- people:
hendren@cs.mcgill.ca, rahul.garg@mail.mcgill.ca, nurudeen.lameed@mail.mcgill.ca