Cetus: Source-to-source Compiler Infrastructure for Multicores

14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming '09

PPoPP '09

For most recent version of these slides see

cetus.ecn.purdue.edu/ppopp09tutorial

What is Cetus?

- Source-to-source C compiler written in Java and maintained at Purdue University
- Provides an internal C parser (Antlr)
- Intermediate representations (15K+ lines)
- Compiler passes (20K+ lines and growing)
- Supported by National Science Foundation
- Drivers of the Cetus effort:
  - automatic parallelization
  - many other applications have emerged

Download and License

- Cetus.ecn.purdue.edu
- Modified Artistic License

  Key points:
  - Open source with most of the common rules
  - OK to distribute modifications, if you make them public and let us know
  - Redistribution of Cetus must be free

Why Cetus?

- Wanted source-to-source C translator, because
  - Parallelization techniques tend to be source transformations
  - Allows comparison of original and transformed code
  - Cetus use as an instrumentation tool
- Alternatives did not fit
  - Predecessor (Polaris) only works on Fortran77
  - GCC has no source-to-source capability
  - SUIF is for C; last major update in 2001
  - Open64's 5 IRs are more complex than we needed
- Wanted a compiler written in modern language
  - Polaris and SUIF use old dialects of C++
  - Portability is important for user community
- Best alternative was to write our own

An Initial Example

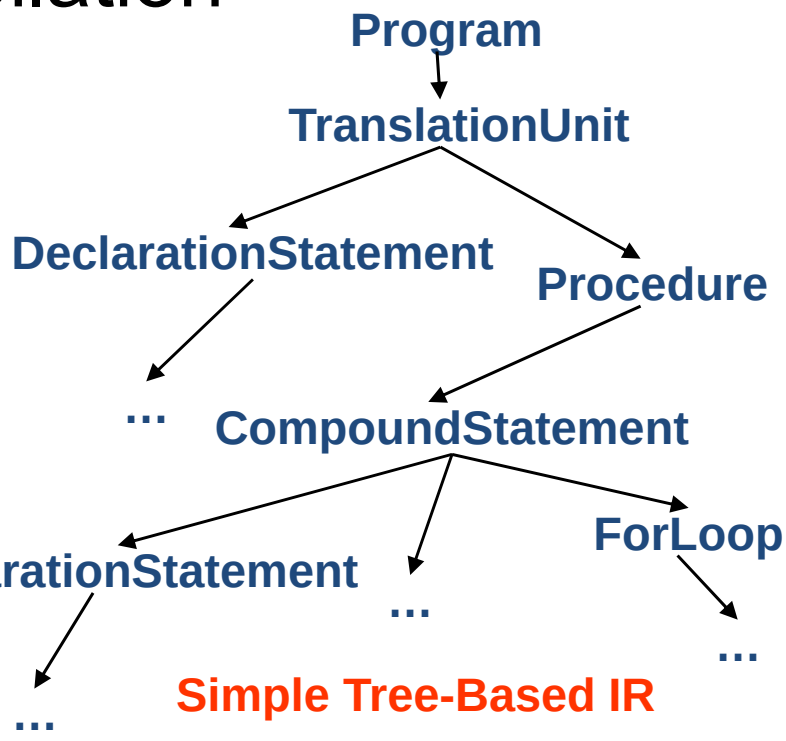
- Cetus source-to-source compilation

Input Source

```
int temp;
int main(void) {
        int i,j,c,a[100];
        c = 2;
        for (i=0; i<100; i++) {
                a[i] = c*a[i]+(a[i]-1);
        }
}
```

Output Source

```
int temp;
int
main (void)
{
  int i, j, c, a[100];
  c = 2;
  for (i = 0; i < 100; i++)
    {
      a[i] = ((c * a[i]) + (a[i] - 1));
    }
}
```



Simple Tree-Based IR

As closely associated with original program structure as possible for regeneration of source code

An Initial Example

- Automatic Parallelization

Input

```
int foo(void)
{
  int i;
  double t, s, a[100];
  for ( i=0; i<50; ++i )
  {
    t = a[i];
    a[i+50] = t +
         (a[i]+a[i+50])/2.0;
    s = s + 2*a[i];
  }
  return 0;
}
```

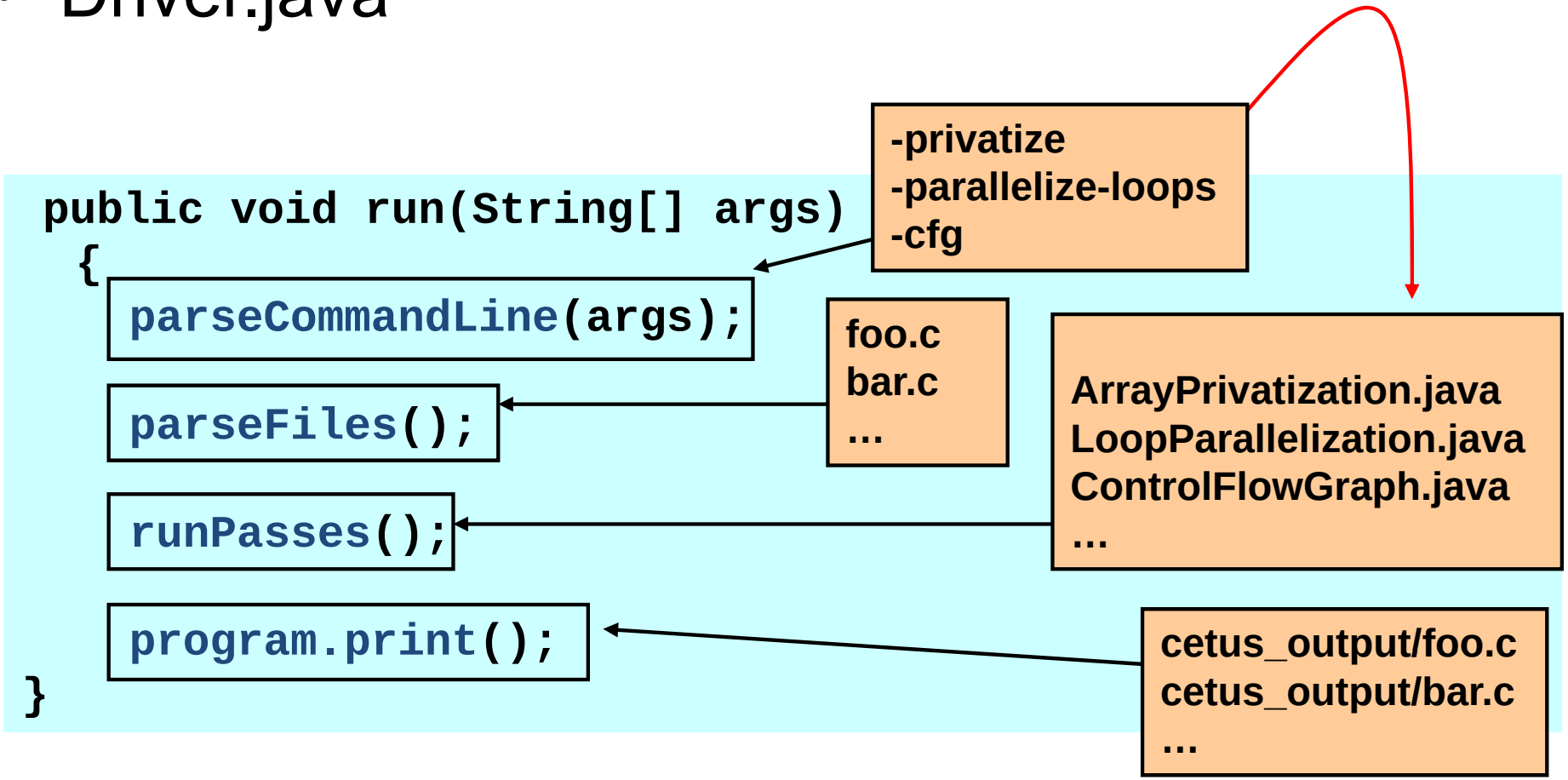
```
$ cetus –parallelize-loops foo.c
```

Output

```
int foo(void )
{
  int i;
  double t, s, a[100];
  #pragma cetus private(i, t)
  #pragma cetus reduction(+: s)
  #pragma cetus parallel
  #pragma omp parallel for reduction(+: s)
                           private(i, t)
  for (i=0; i<50;  ++ i)
  {
    t=a[i];
    a[(i+50)]=(t+((a[i]+a[(i+50)])/2.0));
    s=(s+(2*a[i]));
  }
  return 0;
}
```

Cetus Driver

- Driver.java



Loop instrumentation pass

```
/* File name: MyTool.java */                          Pseudo code
/* Codes are explained in detail later */             User code
public static void InstrumentLoops(Program P)
{
  foreach Procedure proc in P {   → see BreadthFirstIterator
    List<Statement> loops = new ArrayList<Statement>();
    foreach Loop loop in proc     → see DepthFirstIterator
      loops.add(loop);
    String name = proc.getName();
    int i=0;
    for ( Statement loop : loops ) {
      Statement notestmt = CreateNameAnnotation(name, i);
      Statement ticstmt = CreateTimingCall("cetus_tic", i);
      Statement tocstmt = CreateTimingCall("cetus_toc", i++);
      AddInstrumentation(loop, notestmt, ticstmt, tocstmt);
    }
  }
}
```

Cetus Driver - MyTool

- Modify Driver.java to run MyTool

```
public void run_modified(String[] args) {
   parseCommandLine(args);

   parseFiles();

   runPasses();

   program.print();
 }
```

```
public void runPasses() {
   ...
   MyTool my_loop_instrumenter = new MyTool();
   my_loop_instrumenter.InstrumentLoops(program);
   ...
 }
```

Cetus Passes for Program Analysis and Transformation

Analysis Passes

- Pointer alias analysis
  - Currently: very simple flow/context-insensitive alias map
- Symbolic range analysis
  - Generates a map from variables to their valid symbolic bounds at each program point
- Array privatization
  - Computes privatizable scalars and arrays
- Reduction recognition
  - Detects reduction operations

Analysis Passes (cont.)

- Data dependence analysis
  - Performs dependence analysis with Banerjee-Wolfe Test
- Symbolic expression manipulators
  - Normalizes and simplifies expressions
  - Examples
    - **`1+2*a+4-a` → `5+a`** (folding)
    - **`a*(b+c)` → `a*b+a*c`** (distribution)
    - **`(a*2)/(8*c)` → `a/(4*c)`** (division)
- Call Graph and CFG generators
  - CFG provided either at basic-block level or at statement level
- Basic use/def set computation for both scalars and array sections

Transformation Passes

- Induction variable substitution pass (upcoming)
  - Identifies and substitutes induction variables
- Loop parallelizer
  - Depends on induction variable substitution, reduction recognition, and array privatization
  - Performs loop dependence analysis
  - Generates “parallel loop” annotations
- Program normalizers
  - Single call, single declarator, singe return normalizers
- Loop outliner
  - Extracts loops out into separate subroutines

Symbolic Range Analysis

- Computes symbolic ranges for integer variables
  - Performs fine-grain value flow analysis
  - Returns map from statements to sets of ranges
  - Ranges are valid before each statement
- Results are used for symbolic value comparison
  - Comparison under a given set of ranges
  - Useful for compile-time symbolic analysis
- Limitations
  - Conservative handling of side effects
  - Does not model integer overflow

Array Data Flow Analysis Using Symbolic Range Analysis

```
int foo(…)
{
 …
 for (tstep=1; tstep<=NSTEP; tstep++)
 {
   for (i=1; i<N-1; i++)
     for (j=1; j<M-1; j++)
     { /* range:{ 1<=i<=(N-2), 1<=j<=(M-2), 1<=tstep<=NSTEP} */
       A[i][j] = (B[i][j-1]+B[i][j+1]+B[i-1][j]+B[i+1][j])/4.0;
     }

   for (i=1; i<N-1; i++)
     for (j=1; j<M-1; j++)
     { /* range:{ 1<=i<=(N-2), 1<=j<=(M-2), 1<=tstep<=NSTEP} */
       B[i][j] = A[i][j];
     }
 }
}
```

DEF: A[1:N-2][1:M-2]
USE: B[0:N-1][0:M-1]

DEF: B[1:N-2][1:M-2]
USE: A[1:N-2][1:M-2]

Symbolic Range Analysis

```
   int foo(…)
   {
    …
    for (i=0; i<n; i++) {
S1   if ( i>0 ) {
S2    a[i] = …
     } else {
S3    a[i] = …
     }
    }
S4  a[i] = …
    …
   }
```

RangeMap(**foo**)

```
    …
S1  {0<=i<=n-1, n>=i+1}
S2  {1<=i<=n-1, n>=i+1}
S3  {i=0, n>=i+1}
S4  {i>=0, n<=i}
    …
```

compare(**S4**,**n**,**i+1**) → **n < i+1**

compare(**S3**,**i**,**10**) → **i < 10**

compare(**S3**,**n**,**0**) → **n > 0**

Data Dependence Analysis

- Loops are the primary source of parallelism in scientific and engineering applications.
- Compilers detect loops that have independent iterations, i.e. iterations access disjoint data

```
for (i=1; i<n; i++) {
    A[expression1] = …
      … = A[expression2]
}
```

Necessary condition for this loop to be parallel:

expression1 in any iteration i' is different from *expression2* in any other iteration i''

Cetus includes the Banerjee-Wolfe DD test

Symbolic Expression Manipulators

- Cetus provides expression normalizers
  - Simplifies a given expression
    - `1+2*a+4-a` → `5+a` (folding)
    - `a*(b+c)` → `a*b+a*c` (distribution)
    - `(a*2)/(8*c)` → `a/(4*c)` (division)
  - Symbolic operations
    - add(), subtract(), multiply(), divide(), …
- Where is this used?
  - Reduces corner cases in analysis passes
  - Array subscript normalization for DD test
- See cetus.analysis.NormalExpression for more

Array Privatization

The Concept

```
#pragma omp parallel for private(work)
for (i=1; i<n; i++){
   work[1:n] = …
   … = …
   … = work[1:n]
}
```

Dependency: Elements of work read in iteration i' were also written in iteration i'-1.

Each processor is given a separate version of the private data, so there is no sharing conflict

Array Privatization Algorithm

L: Loop, LIVE: Live-out variables
EXIT : Loop exits
PRED: Predecessors
KILL: Killed set due to modified variables
in the section representation
DEF : Defined set
USE: Used set
UEU : Upward-exposed uses
PRI: Private variables
LPRI: Live-out private variables

```
procedure Privatization(L, LIVE)
  // input : L, LIVE
  // output : DEF[L], UEU[L], PRI[L]
  // side effect : L is annotated with PRI[L] and LPRI[L]
  // 1. Privatize inner loops
  foreach direct inner loop l in L
    (DEF[l], USE[l], PRI[l]) = Privatization(l, LIVE)
  // 2. Create CFG of loop body w. collapsed inner loops
  G(N, E) = BuildCFG(L)
  // 3. Compute must-defined set DEF prior to each node
  Iteratively solve data flow equation DEF for node n ∈ N .
    DEFin[n] = ⋂ p∈P RED[n] DEFout[p]
    DEFout[n] = (DEFin[n] − KILL[n]) ∪ DEF[n]
  // 4. Compute DEF[L], UEU[L], PRI[L]
  DEF[L] = ⋂ n∈EXIT[L] DEFout[n]
  UEU[L] = ⋃ n∈N (USE[n] − DEFin[n])
  PRI[L] = CollectCandidates(L, DEF, PRI)
  PRI[L] = PRI[L] − UEU[L]
  LPRI[L] = PRI[L] ⋂ LIVE[L]
  PRI[L] = PRI[L] − LPRI[L]
  AnnotatePrivate(L, PRI[L], LPRI[L])
  // 5. Aggregate DEF and USE over the iteration space
  DEF[L] = AggregateDEF(DEF[L])
  UEU[L] = AggregateUSE(UEU[L])
  return (DEF[L], UEU[L], PRI[L])
end procedure
```

Reduction Parallelization

The Concept

```
#pragma omp parallel for reduction(+:sum)
for (i=1; i<n; i++){
   …
   sum = sum + A[i]
   …
}
```

Dependency: Value of sum written in iteration i'-1 is read in iteration i'.

Each processor will accumulate partial sums, followed by a combination of these parts at the end of the loop.

Reduction Recognition Algorithm

```
procedure RecognizeReductions(L)
  Input : Loop L
  Side-effect : adds reduction annotations for L in the IR
  REDUCTION = {} // set of candidate reduction expressions
  REF = {} // set of non-reduction variables referenced in L
  foreach expr in L
    localREFs = findREF(expr)
    if (expr is AssignmentExpression)
      candidate = lhse(expr)
      increment = rhse(expr) – candidate
      if ( !(getSymbol(candidate) in findREF(increment)) )
        // criterion1 is satisfied
        REDUCTION = REDUCTION ∪ candidate
        localREFs = findREF(increment)
    REF = REF ∪ localREFs // collect non-reduction references for criterion 2
  foreach expr in REDUCTION
    if ( ! (getSymbol(expr) in REF) )
      // criterion 2 is satisfied
      if (expr is ArrayAccess AND expr.subscript is loop-variant)
        CreateAnnotation(sum-reduction, ARRAY, expr)
      else
        CreateAnnotation(sum-reduction, SCALAR, expr)
end procedure
```

lhse/rhse extracts the left/right-hand side expression of an assignment expr. (note, assignments are referred to as expressions rather than statements)

findREF(X) returns the set of USE/DEF references in a given expression X.

"–" is a symbolic subtraction operator.

getSymbol(X) returns the base symbol of expression X (a scalar or an array)

criterion 1: the loop contains one or several assignment expressions of the form sum=sum+increment, where increment is typically a real-valued, loop-variant expression.

criterion 2: sum does not appear anywhere else in the loop.

Induction Variable Substitution

The Concept

```
ind = k
for (i=1; i<n; i++){
   ind  = ind + 2
   A(ind) = B(i)
}
```

simple Induction variables →

```
ind = k
for (i=1; i<n; i++){
   A(k+2*i) = B(i)
}
```

```
ind = k
for (i=1; i<n; i++){
   ind  = ind + i
   A(ind) = B(i)
}
```

generalized Induction variables →

```
for (i=1; i<n; i++){
   A(k+(i**2+i)/2) = B(i)
}
```

Induction Variable Substitution Algorithm

```
procedure SubstituteInductionVariable(iv, L, LIVE)
  input : iv, // induction variable to be substituted
          L // loop to be processed
          LIVE // set of variables that are live-out of L
  side effect : iv is substituted in IR
  inc = FindIncrement(L0) // L0 is the outermost loop of the nest
  Replace(L0 , iv)
  if (iv ∈ LIVE) InsertStatement("iv = inc") // at the end of L
end procedure
```

```
procedure FindIncrement(L)
  // Find the increments incurred by iv from the beginning of L:
  // inc_after[s] is the increment from beginning of loop body
  //     to statement s
  // - inc_into_loop[L] is the increment from beginning of L to beginning
  //   oj  j th iteration (j counts L's iterations from 1 to ub, in steps of 1)
  // - the subroutine returns the total increment added by L
  inc = 0
  foreach statement stmt in L of type Ind, Loop
    if stmt has type Loop
      inc += FindIncrement(stmt)
    else // statement has the form iv = iv + exp
      inc += exp
    inc_after[stmt] = inc
    inc_into_loop[L] = Σ1^(j-1) (inc) // inc may depend on j
    return Σ1^ub (inc)
end procedure
```

```
procedure Replace(L, initialval)
  // Substitute v with the closed-form expression
  val = initialval + inc_into_loop[L]
  foreach statement stmt in L of type Ind, Loop, Use
    if stmt has type Loop
      Replace(stmt, val)
    if stmt has type Loop, Ind
      val += inc_after[stmt]
    if stmt has type Use
      Substitute(stmt, iv, val)
        // replace occurrences of iv in stmt
end procedure
```

Benchmarks and Performance

- Parsing
  - SPEC CPU 2006: 14 codes, 990k lines, 4 ½ minutes
  - SPEC OMP 2001: 3 codes, 17k lines, 15 seconds
  - NAS Parallel Benchmark: 8 codes, 15k lines, 17 seconds
- Runtime validation
  - Parsing → compilation (gcc) → SPEC validation
  - SPEC CPU 2006: 13 validates (1 with K&R function)
- Loop parallelization (#loop, #manual, #auto, #manual and auto)
  - SPEC OMP 2001: 408, 24, 21, 1
  - NAS Parallel Benchmark: 902, 195, 138, 33

Cetus Roadmap

We want community contributions!

- Improved parallelization
  - Enhancements to existing passes
  - Symbolic data dependence test
  - Pointer analysis
  - Locality enhancement techniques
- Cetus application passes
  - OpenMP to MPI translator
  - OpenMP to CUDA translator
  - Autotuning system

Cetus Intermediate Representation (IR)

Code Traversal

Major Parts of Class Hierarchy

Base Classes

Program
TranslationUnit
Declaration
- Procedure
- VariableDeclaration
- ClassDeclaration
- …

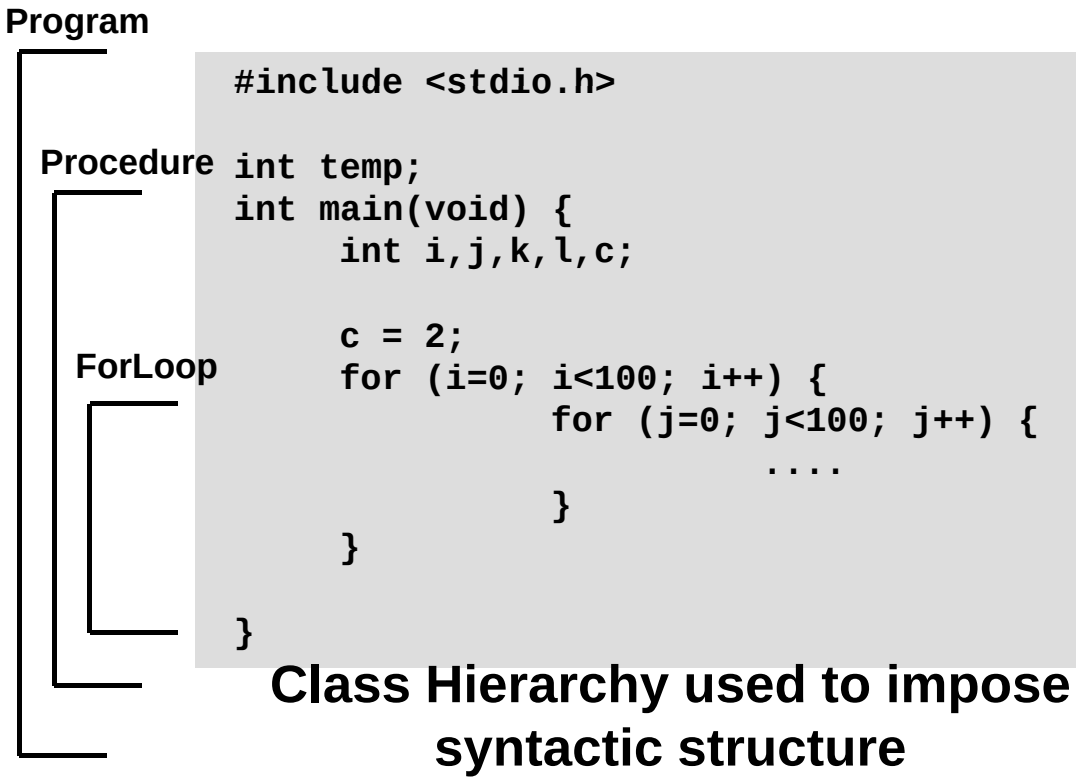
Statement
- IfStatement
- ForLoop
- CompoundStatement
- …

Expression
- BinaryExpression
- UnaryExpression
- FunctionCall
- …

Sub Classes

IRIterator
- DepthFirstIterator
- BreadthFirstIterator
- FlatIterator

Annotation

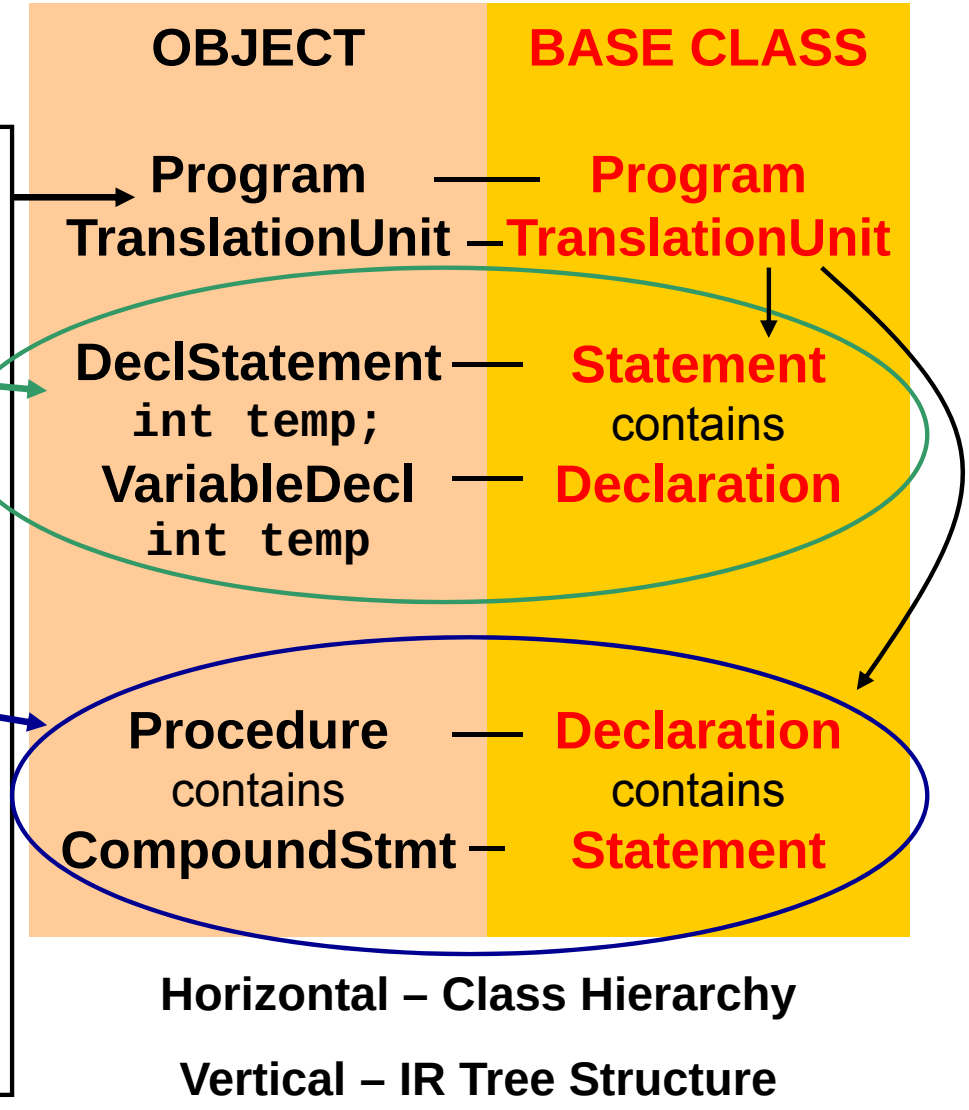


Class Hierarchy used to impose syntactic structure

Cetus Class Hierarchy

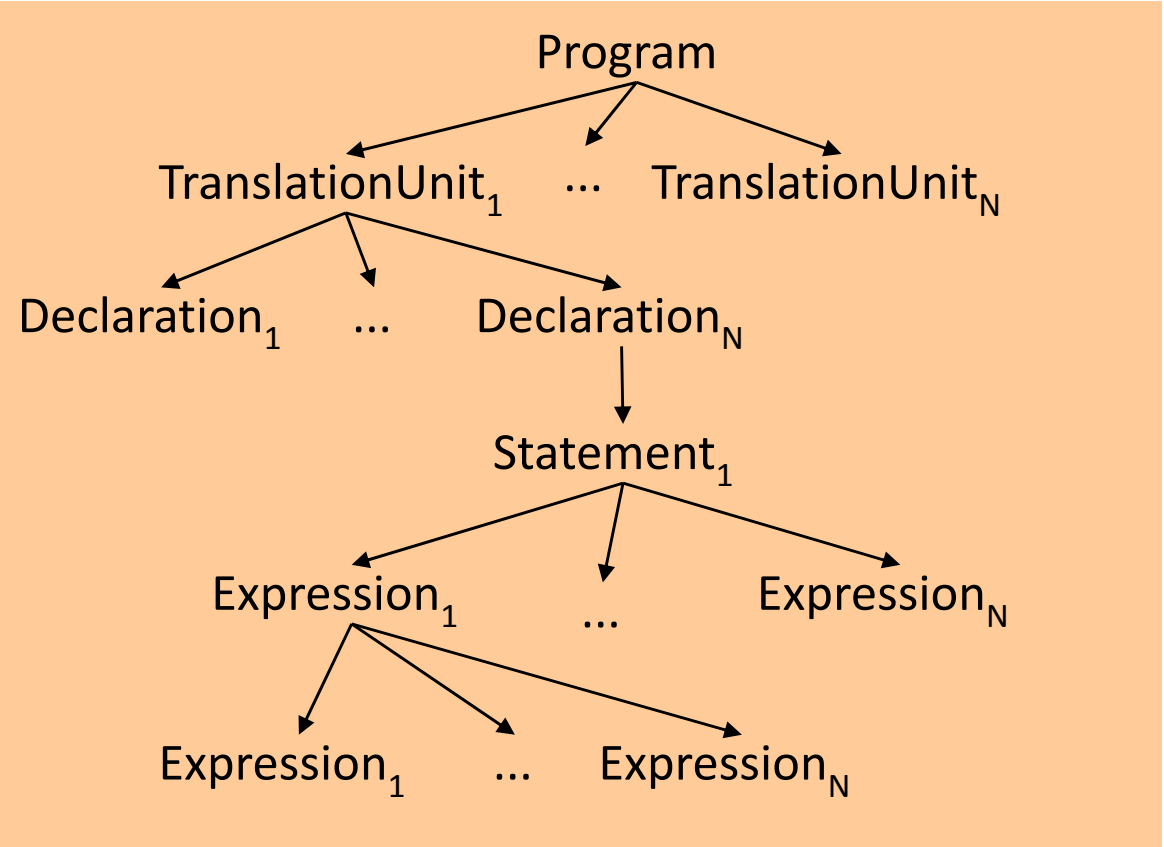
- Class Hierarchy

```
#include <stdio.h>

int temp;
.
.
.
int main(void) {
    int i,j,k,l,c;
    c = 2;
    for (i=0; i<100; i++) {
            for (j=0; j<100; j++) {
                    ....
            }
    }
}
```



Cetus IR

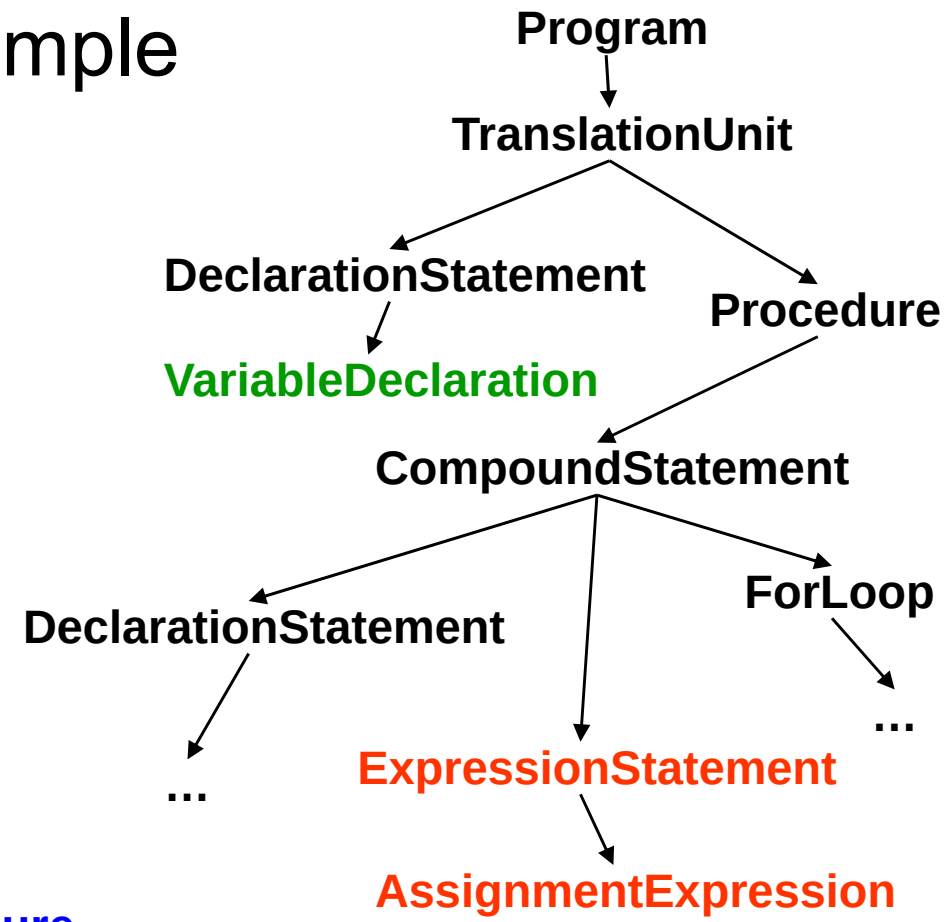
- IR Tree Representation
  - Objects stored in n-ary tree
  - Multiple parent-child relationships
  - All IR objects implement the Traversable interface
  - Traversable object points to parent and children



Cetus IR

- IR Tree Structure Example

```
int temp;
int main(void) {
    int i,j,k,l,c;
    c = 2;
    for (i=0; i<100; i++) {
            ...
    }
}
```



CLASS HIERARCHY IN THIS EXAMPLE

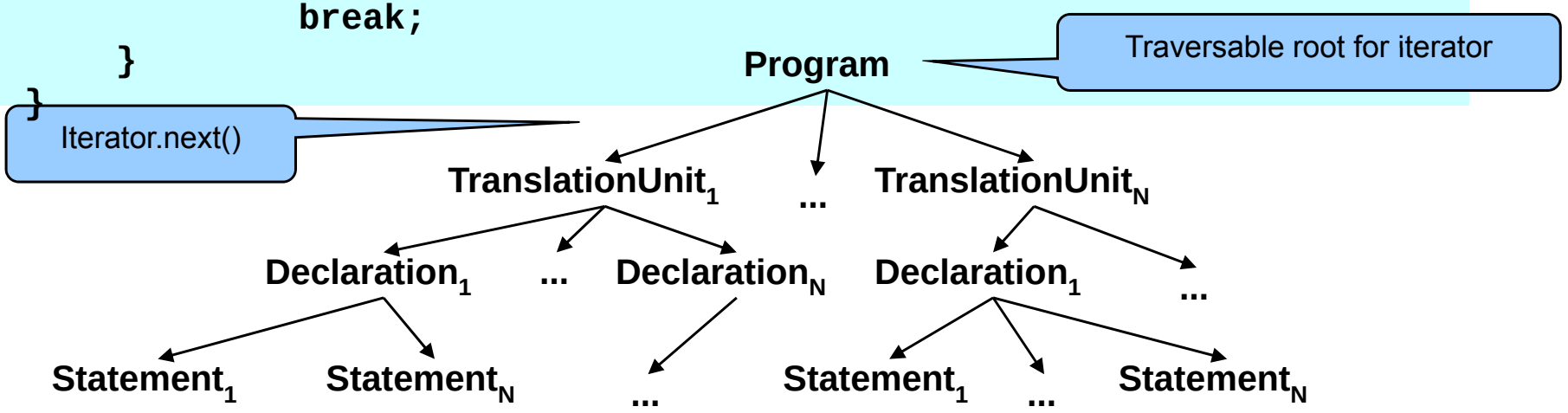
Declaration => VariableDeclaration, Procedure

Statement => DeclarationStatement, CompoundStatement, ForLoop, ExpressionStatement

Cetus – IR Iterators

- BreadthFirst Iteration

```
/* Iterate breadth-first program */
BreadthFirstIterator bfs_iter = new BreadthFirstIterator(program);

for (;;) {
    Object o = null;
    try {
            o = bfs_iter.next();
            System.out.print(o.getclass().getname());
    } catch (NoSuchElementException e) {
            break;
    }
}
```

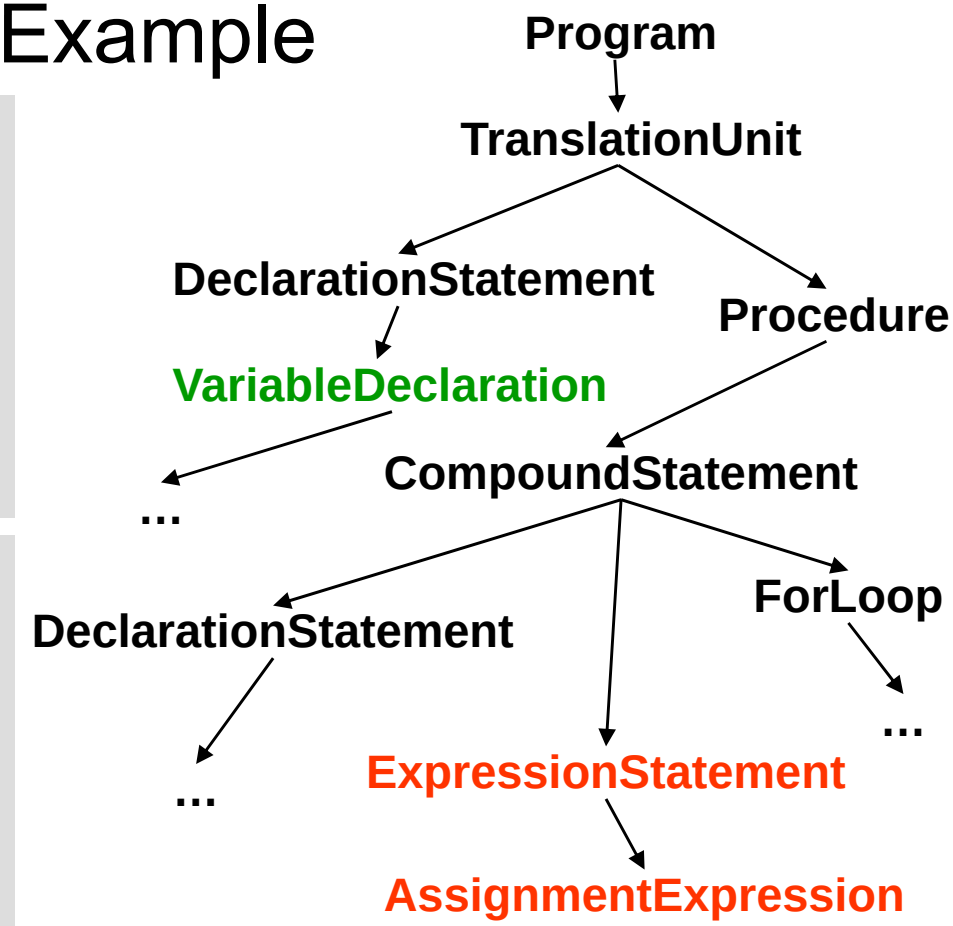


IR Iterators

- IR Tree BreadthFirst Example

```
int temp;
int main(void) {
    int i,j,k,l,c;
    c = 2;
    for (i=0; i<100; i++) {
    }
}
```

```
int temp;
int main(void) {
    int i,j,k,l,c;
    c = 2;
    for (i=0; i<100; i++) {
    }
}
```

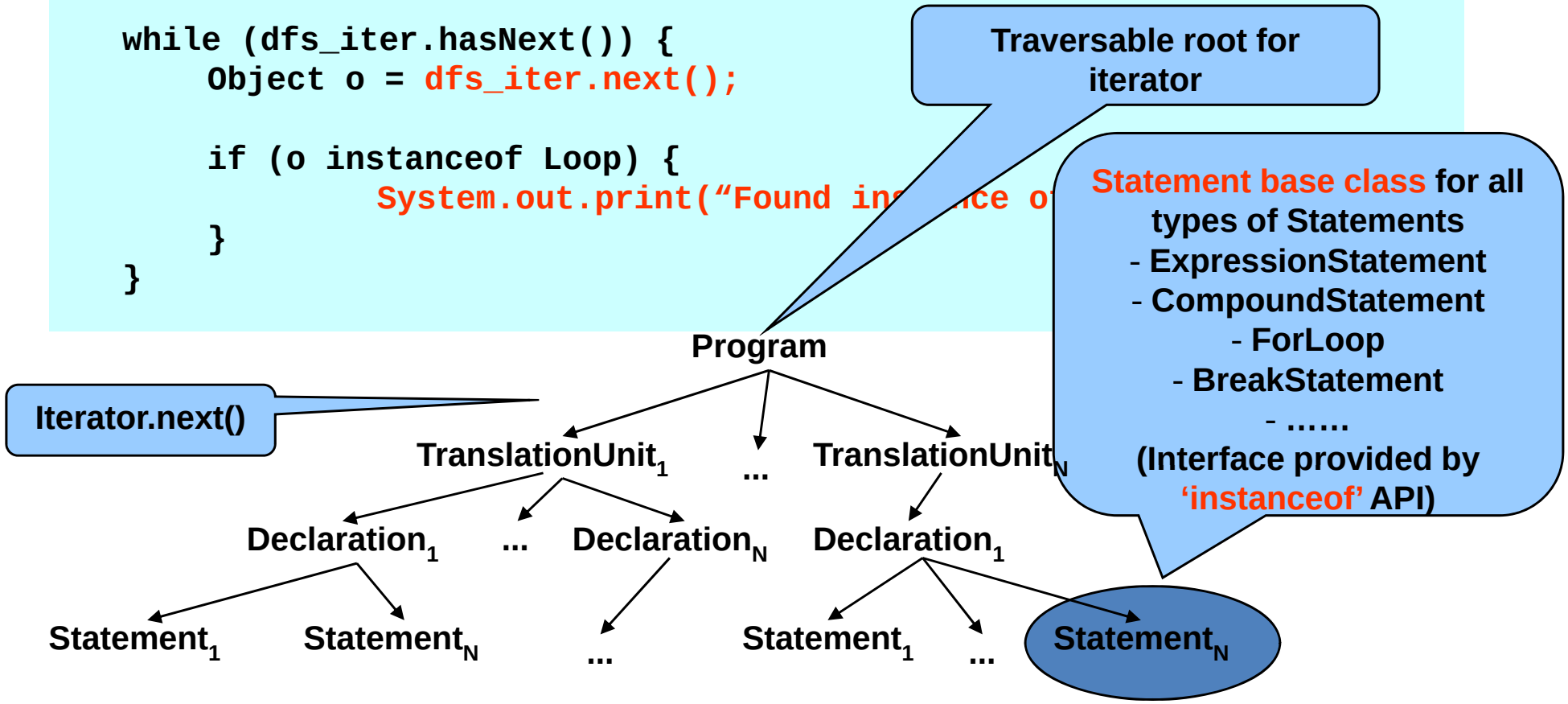


BreadthFirst traversal can be used to traverse upper-level objects in the IR before proceeding to lower-level objects

IR Iterators

- DepthFirst Iteration

```
/* Iterate depth-first program */
DepthFirstIterator dfs_iter = new DepthFirstIterator(program);

while (dfs_iter.hasNext()) {
    Object o = dfs_iter.next();

    if (o instanceof Loop) {
            System.out.print("Found ins ice o
    }
}
```

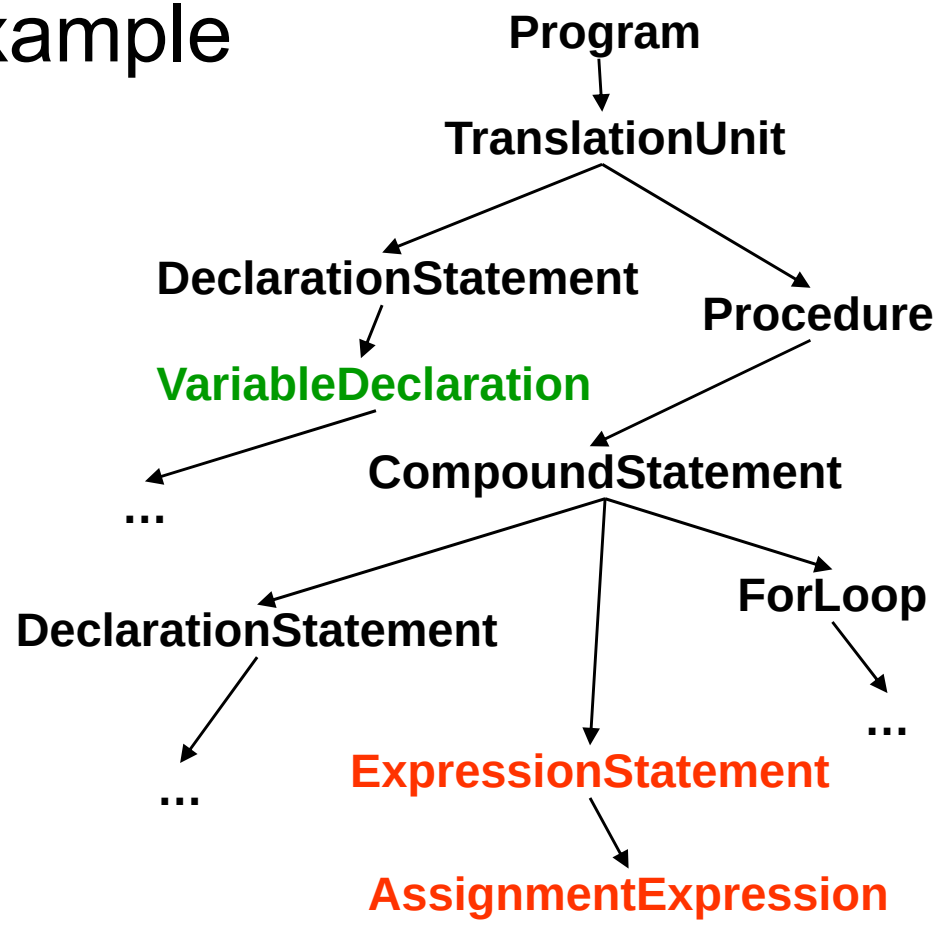


IR Iterators

- IR Tree DepthFirst Example

```
int temp;
int main(void) {
    int i,j,k,l,c;
    c = 2;
    for (i=0; i<100; i++) {
    }
}
```

```
int temp;
int main(void) {
    int i,j,k,l,c;
    c = 2;
    for (i=0; i<100; i++) {
    }
}
```

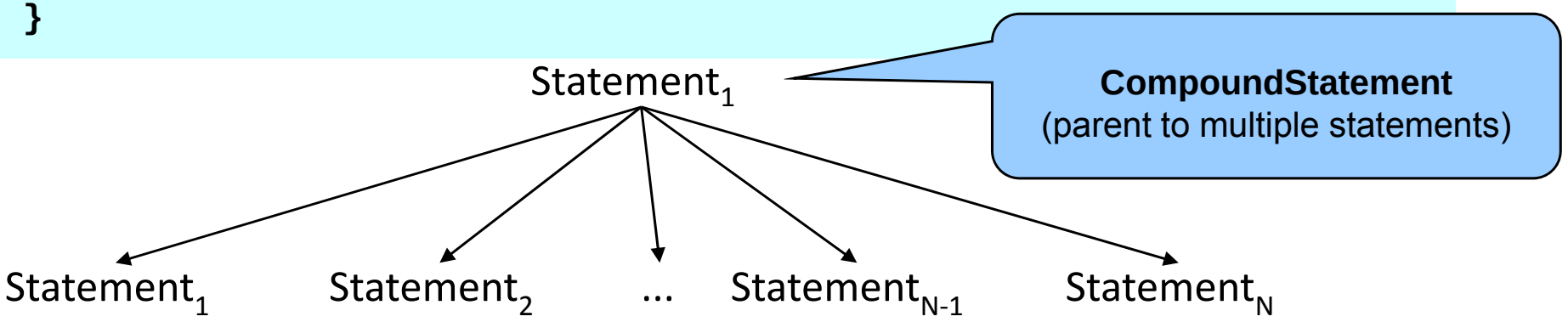


DepthFirst traversal can be used to find outermost loops in a procedure, including those inside compound statements.

IR Iterators

- Flat Iteration (Single level iteration)

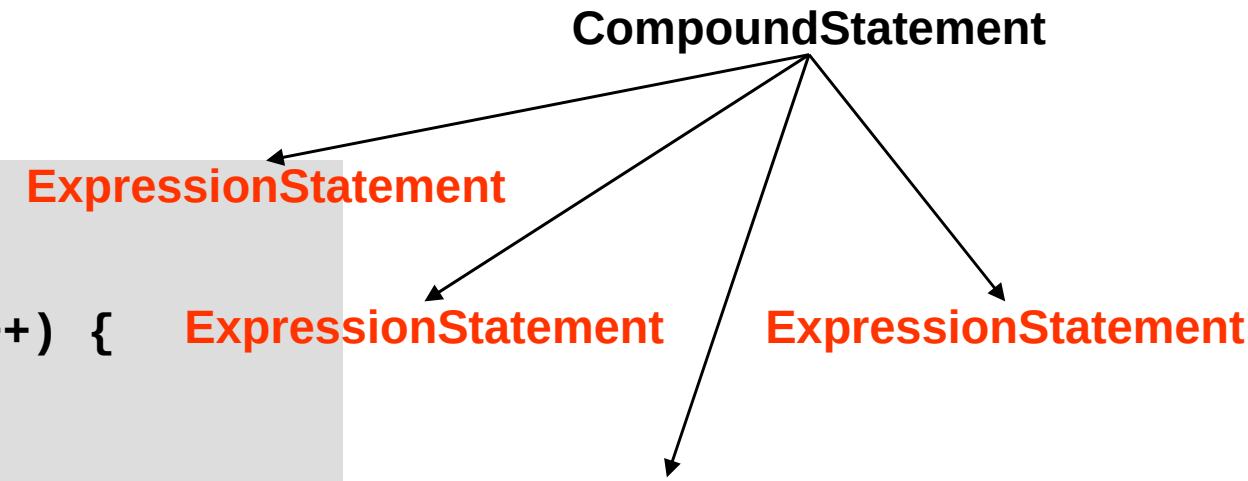
```
/* Iterate single level on compound statement */
CompoundStatement loop_body = (CompoundStatement)loop.getBody();
FlatIterator flat_iter = new FlatIterator(loop_body);

while (flat_iter.hasNext()) {
    Object o = flat_iter.next();

    if(o instanceof ExpressionStatement) {
            Tools.println("Found Expression Statement in Loop", 2);
    }
}
```



IR Iterators

- IR Tree Flat Iteration Example

```
int main(void) {
    int i,j,c,a[100],b;
    for (i=0; i<100; i++) {
            c = a[i];
            a[i] = ...;
            for (j=0; j<100; j++) {
            }
            b = a[i-1];
    }
}
```



Traverse over the outermost statements in a loop body

Compound Statement – ForLoop, ExpressionStatement (collection of Statement objects)

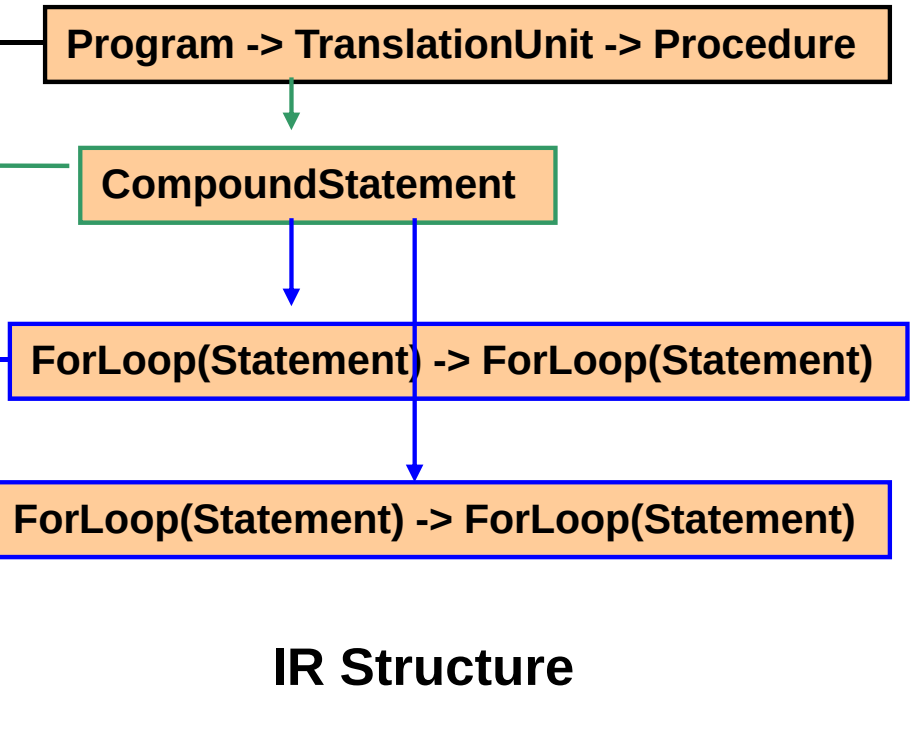
IR Iterators

- Additional functional interface for traversal
  - next(Class c) returns the next object of Class c
    - `iter.next(Statement.class);`
  - next(Set s) returns the next object belonging to any of the classes in Set s
    - `HashSet<Class> of_interest = new HashSet<Class>();`
    - `of_interest.add(AssignmentExpression.class);`
    - `of_interest.add(UnaryExpression.class);`
    - `iter.next(of_interest);`
  - pruneOn(Class c) forces the iterator to skip all descendants of objects of Class c in the IR tree
    - `iter.pruneOn(AssignmentExpression.class);`

IR Traversal – Example

- Loop Nest Identification – Traverse over all *outermost loops* in the program i.e. all loops at the outermost level of a loop nest in the program

```
int main(void) {
    int i,j,k,l;
    float a[100][100],b[100][100], c;

    c = 2.0;
    for (i=0; i<100; i++) {
        for (j=0; j<100; j++) {
            ....
        }
    }
    for (k=0; k<100; k++) {
        for (l=0; l<100; l++) {
            ....
        }
    }
}
```

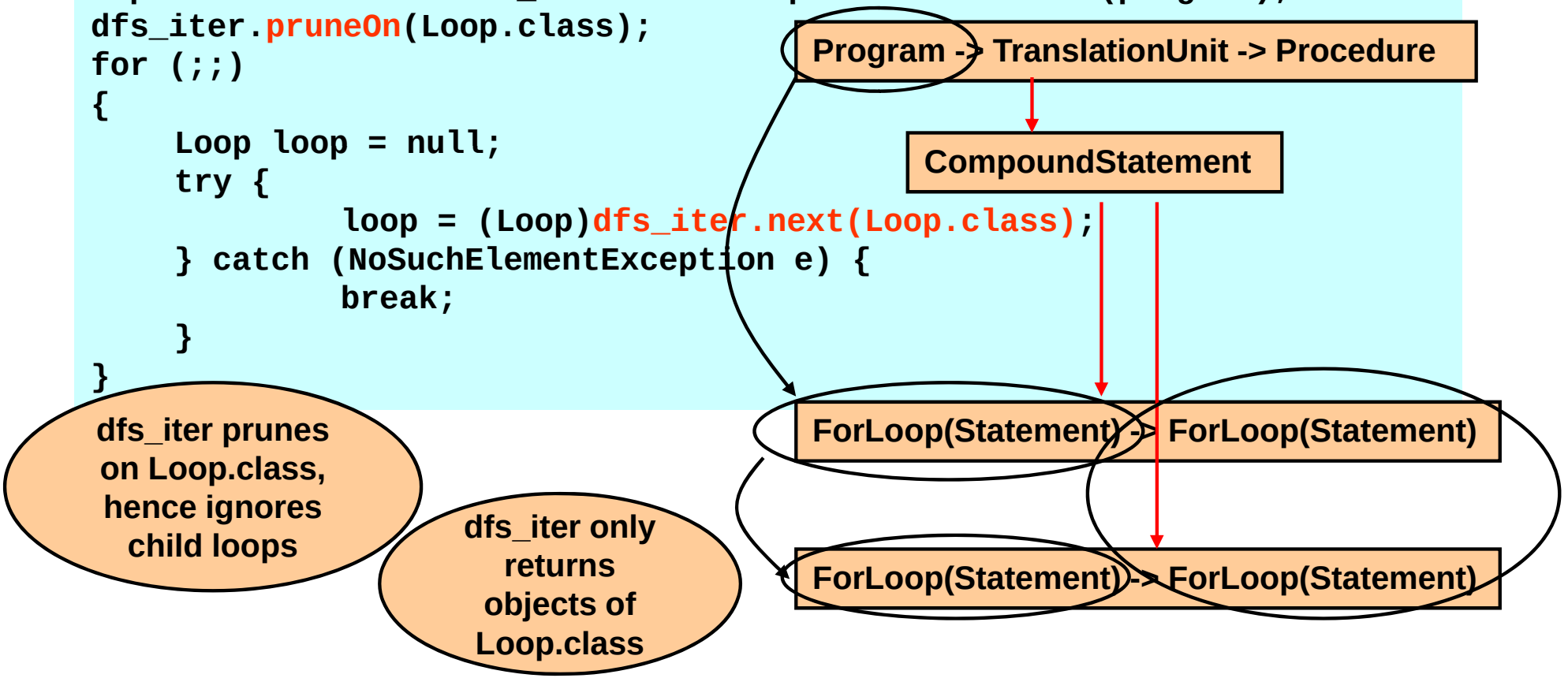


IR Structure

IR Traversal - Example

- Loop nest identification – Iterator interface

```
/* Iterate depth-first over outermost loops in the program */
DepthFirstIterator dfs_iter = new DepthFirstIterator(program);
dfs_iter.pruneOn(Loop.class);
for (;;)
{
    Loop loop = null;
    try {
            loop = (Loop)dfs_iter.next(Loop.class);
    } catch (NoSuchElementException e) {
            break;
    }
}
```



Cetus IR Traversal

- Call Tree Traversal
  - Call Tree is HashMap indexed by Procedure
  - Tree Node contains Procedure and list of callers and callees

// Interface includes

callsSelf(Procedure p)

isLeaf(Procedure p)

isRecursive(Procedure p)

Get Call Graph

```
/* Generate a call graph from
the program IR */
CallGraph cg_generator =
    new CallGraph(program);
HashMap cg =
    cg_generator.getCallGraph();
```

```
/* Obtain the call graph node for
Procedure proc and access its
callers and callees */
...
Node p = cg.get(proc);
```

Get Proc Node

```
ArrayList<Caller> proc_callers =
                        p.getCallers();
ArrayList<Procedure> proc_callees =
                        p.getCallees();
for(Caller c : proc_callers) {
    Statement call_site =
            c.getCallSite();
    Procedure calling_proc =
            c.getCallingProc();
}
```

Access callers and callees

Symbol Table Interface

Accessing Symbol Table

SymbolTable interface

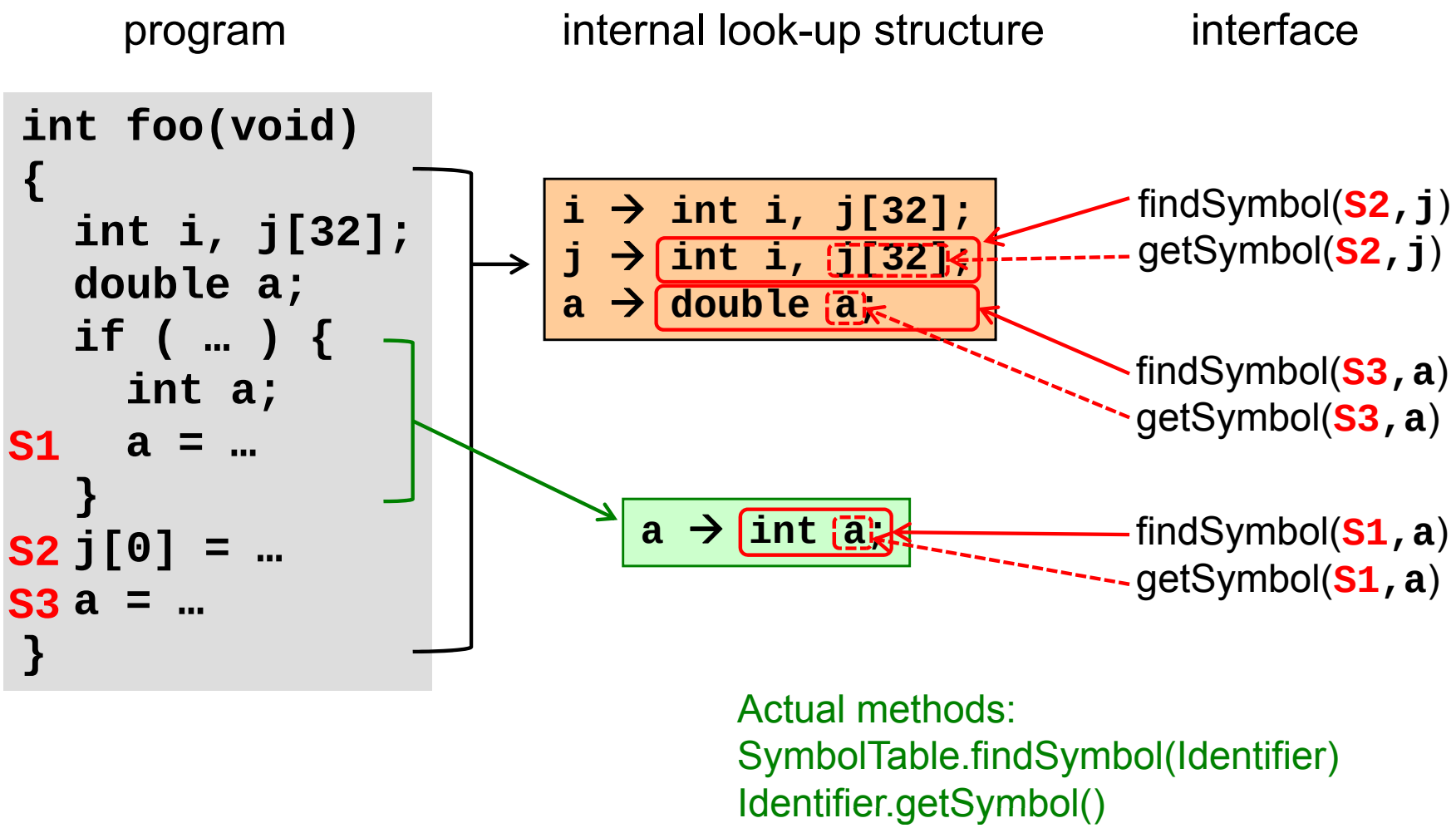
- IR with scope inherits SymbolTable interface
  - TranslationUnit, CompoundStatement, …
- Such an IR object stores a look-up structure
  - From identifier to its declaration
  - Interface functions search/modify this structure
- Methods provided by SymbolTable interface
  - Adding declaration
    - addDeclaration(), addDeclarationBefore(), …
  - Finding declaration of an identifier
    - findSymbol()

Accessing Symbol Table

Symbol interface

- IR representing declarator inherits Symbol interface
  - VariableDeclarator, ProcedureDeclarator, …
- Defines a unique entity for every variable
- Provides fast access to symbol attributes
  - Symbol look-up is preprocessed after parsing
  - Every Identifier expression points to its symbol
- Methods provided by Symbol interface
  - Accessing variables' specifiers
    - getTypeSpecifiers(), getArraySpecifiers()
  - Accessing the symbol name
    - getSymbolName()

Accessing Symbol Table



Declaring Variables

- Find the scope for a new variable
- Create an identifier with a non-conflicting name
  - Use SymbolTable interface
- Prepare specifiers (type, array, pointer, …)
  - Predefined intrinsic types from Specifier class
  - Use UserSpecifier to create user-defined types
- Create variable declaration
- Insert the declaration to the scope

Declaring Variables

Inserting a temporary integer variable in current scope

```
/* Inserts a temporary integer */
void InsertTemp(Traversable t)
{
  while ( !(t instanceof SymbolTable) )
    t = t.getParent();
  SymbolTable symtab = (SymbolTable)t;
  Identifier temp = new Identifier("temp");
  int i = 0;
  while ( symtab.findSymbol(temp) != null )
    temp = new Identifier("temp"+(i++));
  Declarator d = new VariableDeclarator(temp);
  Declaration decl =
    new VariableDeclaration(Specifier.INT,temp);
  symtab.addDeclaration(dd);
}
```

find scope

decide name

create declaration

insert declaration

```
for (i=0; i<100; i++) {
 for (j=0; j<100; j++) {
  double temp;
  …
  a[i][j] = b;
  …
 }
}
```

t

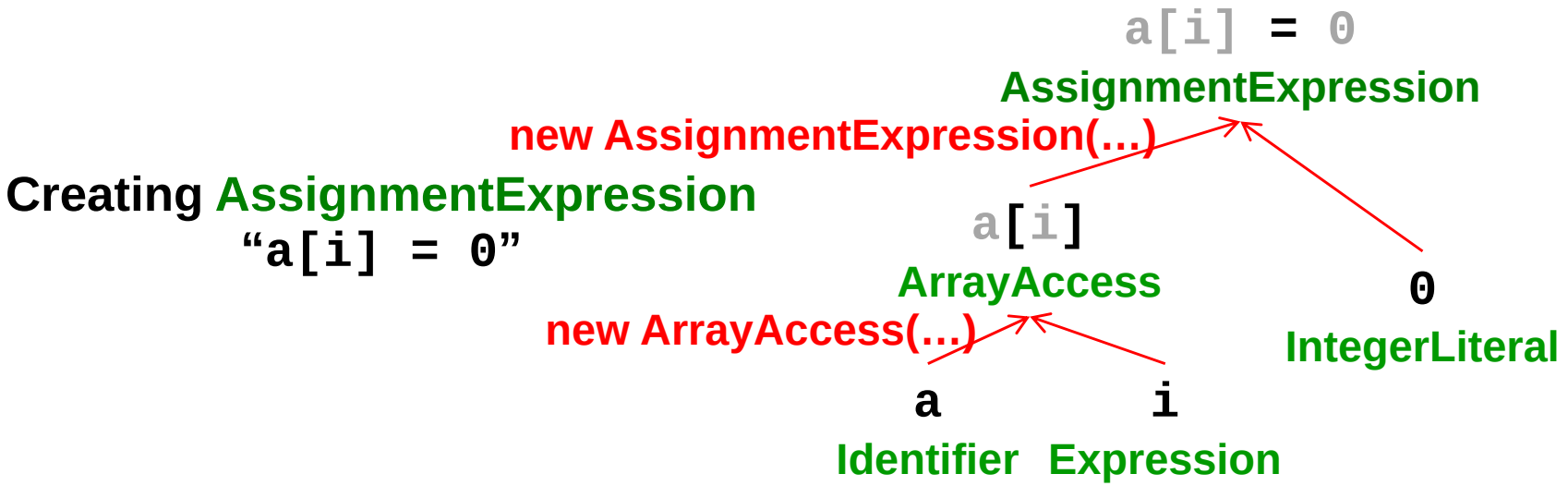
```
symtab =
temp = "temp0"
decl = "int temp0;"
```

```
for (i=0; i<100; i++) {
 for (j=0; j<100; j++) {
  double temp;
  int temp0;
  …
  a[i][j] = b;
  …
 }
}
```

Working on the Program IR

Creating Cetus IR objects

- Equivalent to building a tree from leaves
  - Some IRs are created first then populated
    e.g., CompoundStatement
- IR constructors perform the tasks
- Use cloning if creating from an existing IR object



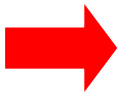
Modifying Cetus IR objects

- Equivalent to inserting/replacing a node in a tree
- IR methods handle most modifications
- Other modifications are usually search/replace
  - Create a new expression or statement
  - Traverse the IR to locate the position for the new node to be inserted or replaced with
  - Perform insertion or replacement
- We will explore the details with examples
  - Loop unrolling
  - Loop instrumentation

Working Example – Loop Unrolling

Loop unrolling by factor of 2

```
for (i=0; i<100; i++) {
  fx[i] = fx[i]+x;
  fy[i] = fy[i]+y;
  fz[i] = fz[i]+z;
}
```



```
for (i=0; i<100; i+=2) {
  fx[i] = fx[i]+x;
  fy[i] = fy[i]+y;
  fz[i] = fz[i]+z;
  fx[i+1] = fx[i+1]+x;
  fy[i+1] = fy[i+1]+y;
  fz[i+1] = fz[i+1]+z;
}
```

Find loop index
Create expression i+=2
Replace expression i++ → i+=2

Create 3 Statements
Find loop index
Search/Replace Expression i → i+1
Insert 3 Statements

```
UnrollLoopBy2(ForLoop L)
{
  /* we will compose          */
  /* assuming L is canonical */
}
```

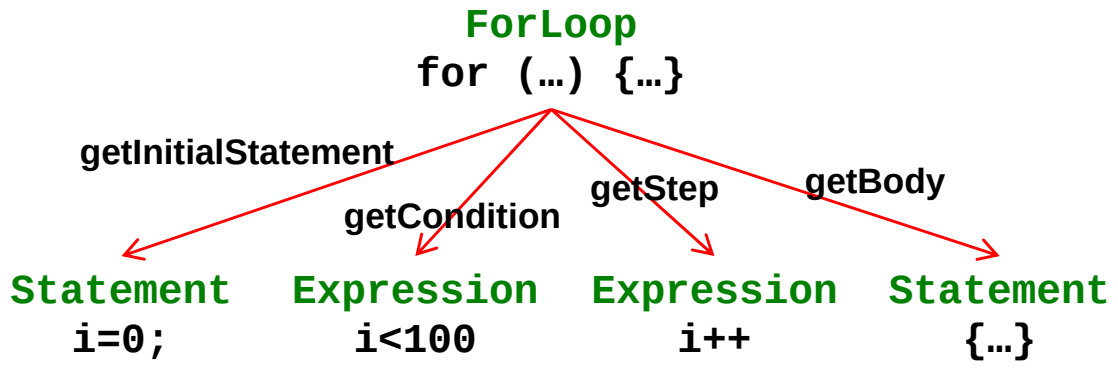
Working on the Program IR

Expressions

Accessing Expressions with IR methods

- IR methods provide direct access to child expressions
  - **ForLoop**: getInitialStatement(), getCondition(), getStep(), …
  - **IfStatement**: getControlExpression, getThenStatement(), …
  - **ExpressionStatement**: getExpression()
  - **ArrayAccess**: getIndex(), getArrayName()
  - **FunctionCall**: getArguments(), getName(), …
  - **BinaryExpression**: getLHS(), getRHS()

```
for (i=0; i<100; i++) {
  …
}
```



Accessing Expressions with IR methods

Finding loop index in UnrollLoopBy2(L)

```
Expression FindLoopIndex(ForLoop L)
{
(1)Statement stmt = L.getInitialStatement();
(2)Expression expr = stmt.getExpression();
(3)Expression index = expr.getLHS();
   return index;
}
```

ForLoop L

```
for (i=0; i<100; i++) {
  fx[i] = fx[i]+x;
  fy[i] = fy[i]+y;
  fz[i] = fz[i]+z;
}
```

(1) stmt = "i=0;"

(2) expr = "i=0"

(3) index = "i"

Creating Expressions

- Most modifications start from creation
  - Replacing an expression with a new one
  - Inserting a new expression
- Use IR constructors
  - **BinaryExpression**(lhs, op, rhs)
  - **AssignmentExpression**(lhs, op, rhs)
  - **FunctionCall**(name, arguments)
- Children are usually created first
- Do not reuse existing IR objects – use clone()

Creating Expressions

Creating new step expression in UnrollLoopBy2(L)

```
Expression CreateStepBy2(Expression index)
{
(1)Expression index0 = (Expression)index.clone();
(2)Expression step = new IntegerLiteral(2);
(3)Expression expr = new AssignmentExpression(
     index0, AssignmentOperator.ADD, step);
  return expr;
}
```

ForLoop L

```
for (i=0; i<100; i++) {
  fx[i] = fx[i]+x;
  fy[i] = fy[i]+y;
  fz[i] = fz[i]+z;
}
```

FindLoopIndex(L) = "i"

(1) index0 = "i"

(2) step = "2"

(3) expr = "i+=2"

Creating Expressions

Creating new indices for unrolled statements in UnrollLoopBy2(L)

```
Expression CreateIndexPlus1(Expression index)
{
(1)Expression index0 = (Expression)index.clone();
(2)Expression one = new IntegerLiteral(1);
(3)Expression expr = new BinaryExpression(
     index0, BinaryOperator.ADD, one);
  return expr;
}
```

ForLoop L

```
for (i=0; i<100; i++) {
  fx[i] = fx[i]+x;
  fy[i] = fy[i]+y;
  fz[i] = fz[i]+z;
}
```

FindLoopIndex(L) = "i"

(1) index0 = "i"

(2) one = "1"

(3) expr = "i+1"

Modifying Expressions with IR methods

- IR methods provide high-level modifiers
  - **ForLoop**: setCondition(), setStep(), …
  - **IfStatement**: setControlExpression(), setThen(), ...
  - **ArrayAccess**: setIndex(), …
  - **FunctionCall**: setArguments(), addArgument(), …
  - **BinaryExpression**: setLHS(), setRHS()
- Replacing step expression In UnrollLoopBy2(L)

ForLoop L

```
for (i=0; i<100; i++) {
  fx[i] = fx[i]+x;
  fy[i] = fy[i]+y;
  fz[i] = fz[i]+z;
}
```

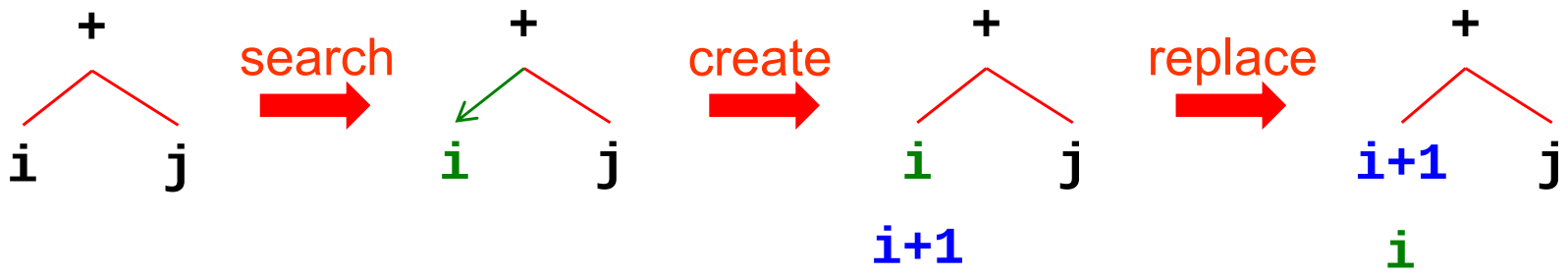
new_step = "i+=2"

```
L.setStep(new_step);
```

```
for (i=0; i<100; i+=2) {
  fx[i] = fx[i]+x;
  fy[i] = fy[i]+y;
  fz[i] = fz[i]+z;
}
```

Searching/Replacing Expressions

- Frequently used in program transformation
  - Replacing identifiers with constants: `foo(i,j)→foo(10,100)`
  - Replacing identifiers with other expressions: `i+j→(i+1)+j`
  - Replacing common sub expressions with temporaries: `i+j→t`
- Steps
  - Locate the expression to be replaced (traversal/compare)
  - Create a new expression with a constructor
  - Replace the old expression with the new one



Searching/Replacing Expressions

Modifying array indices in unrolled statements in UnrollLoopBy2(L)

```
void ReplaceExpr(Expression a, Expression b, Traversable t)
{
  List old_exprs = new ArrayList();
  DepthFirstIterator iter = new DepthFirstIterator(t);
  while ( iter.hasNext() ) {
    Object child = iter.next();
    if ( child.equals(a) ) old_exprs.add(child);
  }                                                    search
  for ( Object old_expr : old_exprs ) {
    Expression new_expr = (Expression)b.clone();
    ((Expression)old_expr).swapWith(new_expr);  replace
  }
}
```

```
for (i=0; i<100; i++) {
  fx[i] = fx[i]+x;
  fy[i] = fy[i]+y;
  fz[i] = fz[i]+z;
}
```

```
t: fx[i] = fx[i]+x;

old_exprs: [i, i] (search)

t: fx[i+1] = fx[i+1]+x; (replace)
```

Working on the Program IR

Statements

Creating Statements

- Use IR constructors
  - **ExpressionStatement**(expr)
  - **ForLoop**(init, condition, step, body)
  - **IfStatement**(condition, thenstmt, elsestmt)
  - **CompoundStatement**() – child statements are added later
- Use clone() from existing statements
- Creating new unrolled statements in UnrollLoopBy2(L)

```
List CreateNewStatements(ForLoop L)
{
  List stmts = new ArrayList();
  FlatIterator iter =
    new FlatIterator(L.getBody());
  while ( iter.hasNext() )
    stmts.add(iter.next().clone());
  return stmts;
}
```

statements from cloning

```
for (i=0; i<100; i++) {
  fx[i] = fx[i]+x;
  fy[i] = fy[i]+y;
  fz[i] = fz[i]+z;
}
```

```
stmts: [fx[i]=fx[i]+x;,
  fy[i]=fy[i]+y;,
  fz[i] = fz[i]+z;]
```

Inserting Statements

- Steps
  - Locate the reference statement (before or after)
  - Locate the parent **CompoundStatement**
  - Use appropriate methods in **CompoundStatement**
    - addStatement(new_stmt)
    - addStatementBefore(ref_stmt, new_stmt)
    - addStatementAfter(ref_stmt, new_stmt)
- Inserting unrolled statements in UnrollLoopBy2(L)

```
void InsertNewStatements(ForLoop L, List stmts)
{
  CompoundStatement parent =
    (CompoundStatement)L.getBody();
  for ( Object stmt : stmts )
    parent.addStatement((Statement)stmt));
}
```

Composing UnrollLoopBy2(ForLoop L)

```
void UnrollLoopBy2(ForLoop L)
{
  Expression index = FindLoopIndex(L);
  Expression new_step = CreateStepBy2(index);
  L.setStep(new_step);
  List new_stmts = CreateNewStatements(L);
  Expression index1 = CreateIndexPlus1(index);
  for ( Object stmt : new_stmts )
    ReplaceExpr(index, index1, (Statement)stmt);
  InsertNewStatements(L, new_stmts);
}
```

index: `i`

new_step: `i+=2`

`for (…; i+=2) {…}`

new_stmts: `[fx[i]=fx[i]+x, …]`

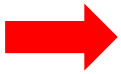
index1: `i+1`

new_stmts: `[fx[i+1]=fx[i+1]+x,…]`

ForLoop L

```
for (i=0; i<100; i++) {
  fx[i] = fx[i]+x;
  fy[i] = fy[i]+y;
  fz[i] = fz[i]+z;
}
```

UnrollLoopBy2(L)



```
for (i=0; i<100; i+=2) {
  fx[i] = fx[i]+x;
  fy[i] = fy[i]+y;
  fz[i] = fz[i]+z;
  fx[i+1] = fx[i+1]+x;
  fy[i+1] = fy[i+1]+y;
  fz[i+1] = fz[i+1]+z;
}
```

Working on the Program IR

Annotations

Creating/Inserting Annotations

- Annotations are used in Cetus for
  - Information exchange between passes
    e.g., set of modified / used variables
  - Information for backend compilers – pragmas
    e.g., OpenMP pragmas
  - Information for code readers – comments
- Annotations can be created with
  - String for comments or raw codes
  - Map (string→object) for internal data exchange

Creating/Inserting Annotations

- Printing of annotations can be customized by
  - Setting built-in print method
    - As comment (default), as pragma, or as raw
  - Extending annotation base class
- Steps (similar to inserting statements)
  - Create the annotation with appropriate data
  - Set an appropriate print method
  - Locate the position for the annotation
  - Insert the annotation

Working Example – Loop Instrumentation

```
int compute(…)
{
 for (i=is; i<ie; i++) {
  …
 }
 …
 for (j=js; j<je; j++) {
  …
 }
}
```

Inserts loop name and timing calls

Find loop
Create function call
Insert function call

```
int compute(…)
{
 /* compute#0 */
 cetus_tic(0);
 for (i=is; i<ie; i++) {
  …
 }
 cetus_toc(0);
 …
 /* compute#1 */
 cetus_tic(1);
 for (j=js; j<je; j++) {
  …
 }
 cetus_toc(1);
}
```

Find loop
Create annotation
Insert annotation

Loop Instrumentation Code

```
public static void InstrumentLoops(Program P)    Pseudo code
{
  foreach Procedure proc in P {   → see BreadthFirstIterator
    List<Statement> loops = new ArrayList<Statement>();
    foreach Loop loop in proc     → see the next slide
      loops.add(loop);                                  find loops
    String name = proc.getName();
    int i=0;
    for ( Statement loop : loops ) {
                                                  create annotation
      Statement notestmt = CreateNameAnnotation(name, i);
      Statement ticstmt = CreateTimingCall("cetus_tic", i);
      Statement tocstmt = CreateTimingCall("cetus_toc", i++);
      AddInstrumentation(loop, notestmt, ticstmt, tocstmt);
                                                  insert statements
    }
  }
}
```

create timing calls

Loop Instrumentation Code

Finding loops

```
void InstrumentLoops(Program P)
{
  …
  Procedure proc = …
  List<Statement> loops =
    new ArrayList<Statement>();
  DepthFirstIterator iter =
    new DepthFirstIterator(proc);
  while ( iter.hasNext() ) {
    Object o = iter.next();
    if ( o instanceof Loop )
      loops.add((Statement)o);
  }
  …
}
```

proc

```
int compute(…)
{
 for (i=is; i<ie; i++) {
  …
 }
 …
 for (j=js; j<je; j++) {
  …
 }
}
```

loops: [
  for(i=is…){…},
  for(j=js;…){…}
]

Loop Instrumentation Code

Creating loop naming annotations as comments

```
/* Creates a comment "name#num" */
Statement CreateNameAnnotation(String name, int num)
{
  Annotation note = new Annotation(name+"#"+num);
  /* note is printed as comment by default */
  Statement notestmt = new AnnotationStatement(note);
  return notestmt;
}
```

```
int compute(…)          proc
{
 for (i=is; i<ie; i++) {
  …
 }
 …
 for (j=js; j<je; j++) {
  …
 }
}
```

From proc.getName() before calling CreateNameAnnotation():
name: **"compute"**

For the first loop:
notestmt: **/* compute#0 */**

For the second loop:
notestmt: **/* compute#1 */**

Loop Instrumentation Code

Creating timing calls

```
/* Creates a function call "fcname(num);" */
Statement CreateTimingCall(String fcname, int num)
{
(1)FunctionCall fc = new FunctionCall(new Identifier(fcname));
(2)fc.addArgument(new IntegerLiteral(num));
(3)Statement fcstmt = new ExpressionStatement(fc);
  return fcstmt;
}
```

```
int compute(…)
{
 for (i=is; i<ie; i++) {
  …
 }
 …
 for (j=js; j<je; j++) {
  …
 }
}
```

(1) fc: cetus_tic()
(2) fc: cetus_tic(0)
(3) fcstmt: cetus_tic(0);

Similarly,
cetus_toc(0);, cetus_tic(1);, cetus_toc(1);

Loop Instrumentation Code

Inserting statements

```
/* Inserts name and tic before loop,
 * and toc after loop */
AddInstrumentation(Statement loop, Statement name,
Statement tic, Statement toc)
{
  CompoundStatement parent =
    (CompoundStatement)loop.getParent();
  parent.addStatementBefore(loop, name);
  parent.addStatementBefore(loop, tic);
  parent.addStatementAfter(loop, toc);
}
```

locate parent compound statement

(1) (2) (3)

```
int compute(…)
{
 for (i=is; i<ie; i++) {
  …
 }
 …
 for (j=js; j<je; j++) {
  …
 }
}
```

```
int compute(…)
{
 /* compute#0 */
 for (i=is; i<ie; i++) {
  …
 }
 …
}
```

(1)

```
int compute(…)
{
 /* compute#0 */
 cetus_tic(0);
 for (i=is; i<ie; i++) {
  …
 }
 cetus_toc(0);
 …
}
```

(2)(3)

```
int compute(…)
{
 /* compute#0 */
 cetus_tic(0);
 for (i=is; i<ie; i++) {…}
 cetus_toc(0);
 /* compute#1 */
 cetus_tic(1);
 for (j=js; j<je; j++) {…}
 cetus_toc(1);
}
```

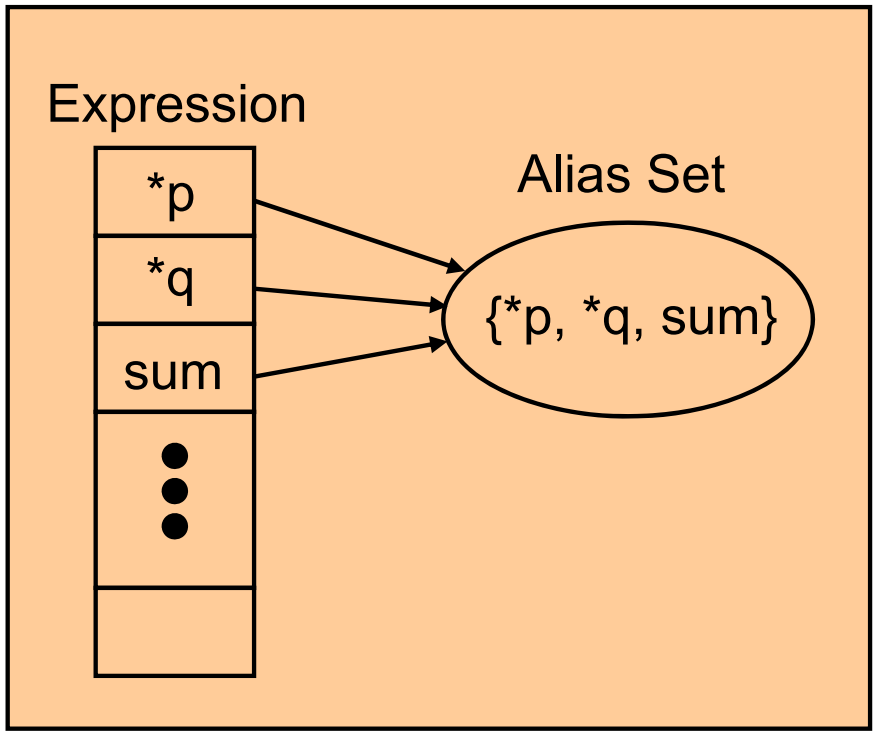
Checking Aliases

- Cetus alias analysis pass performs a conservative intra-procedural flow-insensitive alias analysis for each procedure in the program

Example

```
int i, sum=0;
int *p, *q;
int A[N], B[N];

q = p;
p = &sum;
for (i=0; i<N; i++)
{
  sum += A[i];
  B[i] = *p + *q;
}
```

Alias Map Data Structure



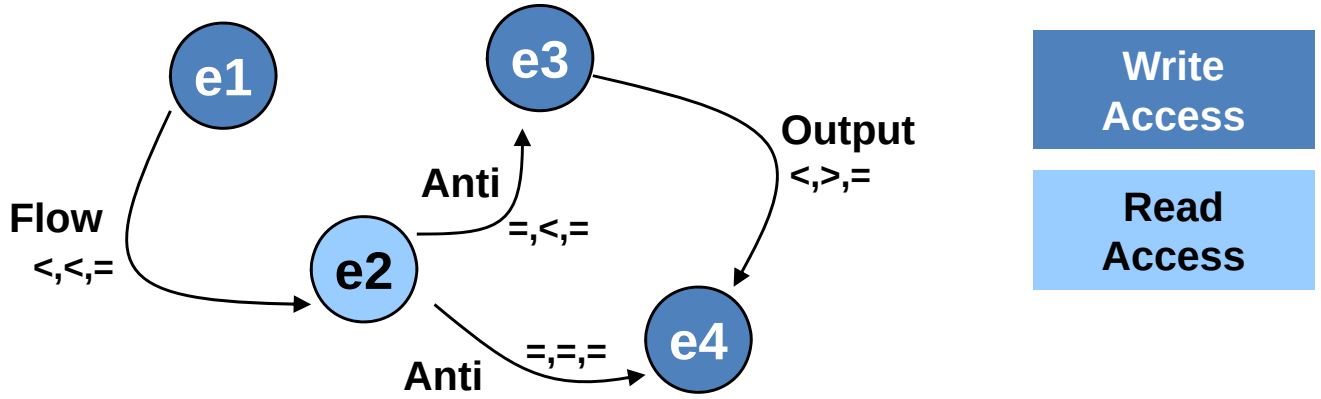
Using Alias Analysis

```
AliasAnalysis alias = new AliasAnalysis(program);
alias.start();

/* 1. returns true if Symbols A and B are aliased */
if (alias.isAliased(A, B)) {
  ...
}

/* 2. returns all Symbols that are aliased to Symbol A
   apply get_alias_set to *p → {*p, *q, sum}  */
Set alias_set = alias.get_alias_set(A);
if (alias_set.contains(A)) {
  ...
}
```

Data Dependence Test Interface

Data Dependence – Current Interface v1.0

- Testing and dependence graph creation

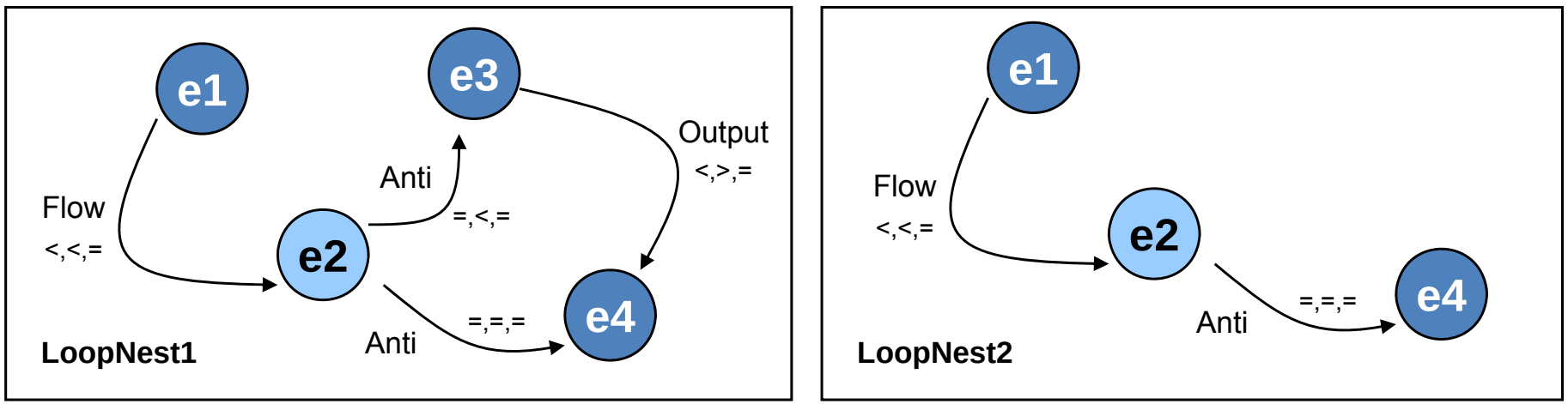


  - Data dependence graph stored as private member within the Dependence Test Driver
  - Invoke dependence testing locally within optimization/analysis pass
  - Obtain a local working copy of loop nest graph for manipulation and information extraction

Data Dependence Interface

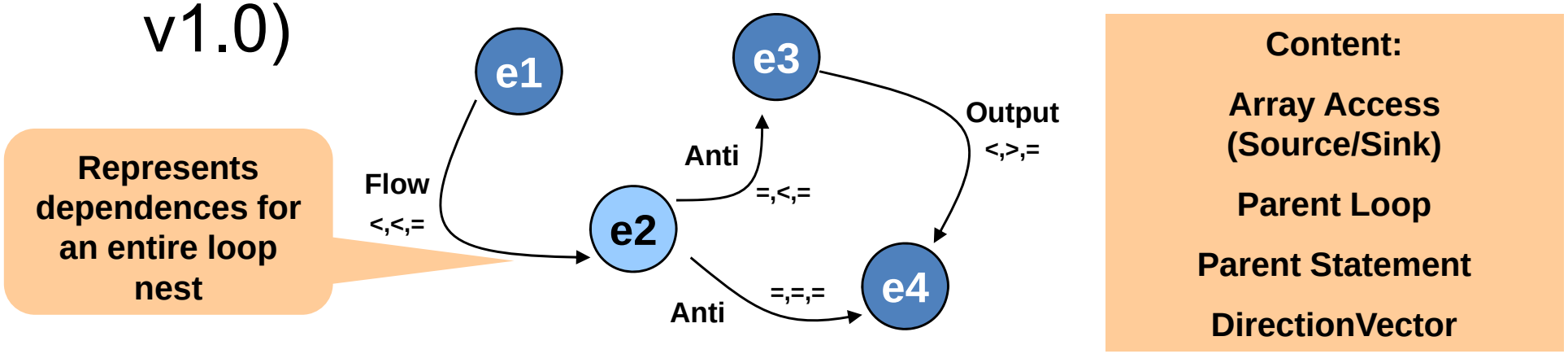
- Getting Data Dependence Information (Cetus v1.0)

```
/* Interface routine runs dependence test behind the scenes and returns
loop-nest based dependence graph */
DependenceAnalysisObject->get_ddgraph();
DependenceAnalysisObject->get_loop_nest_specific_ddgraph(loop);
```



Data Dependence Interface

- Getting Data Dependence Information (Cetus v1.0)




```
/* Data dependence graph (ddgraph) contains all relevant information
Use loop-nest specific graph to check for loop parallelization */
ddgraph->check_if_loop_is_parallel(loop);

/* To extract specific dependence-related information */
ddgraph->get_all_dependence_information_for(expr1, expr2);
ddgraph->get_all_dependence_information_for(stmt1, stmt2);
```

Data Dependence Interface

- Getting Data Dependence Information

```
BreadthFirstIterator bfs_iter = new BreadthFirstIterator(program);
bfs_iter.pruneOn(Loop.class);
DDGraph loop_graph = null;
while (bfs_iter.hasNext()) {
    Object o = bfs_iter.next();
    if (o instanceof Loop) {
            Loop loop = (Loop)o;
     }
```

Find outermost loops

Obtain dependence graph

```
    /* Run dependence analysis on the nest and obtain the
     * dependence graph */
    loop_graph = (new DDTDriver(program)).getDDGraph(
                                DDGraph.summarize, loop);
```

```
    /* Graph is null if loop nest was ineligible for dependence analysis */
    if (loop_graph != null) {
            /* Check if loop in the nest is parallel or not */
            boolean no_dependence = loop_graph.checkParallel(loop);
            addAnnotation(loop, no_dependence);
    }
}
```

Check for parallelism

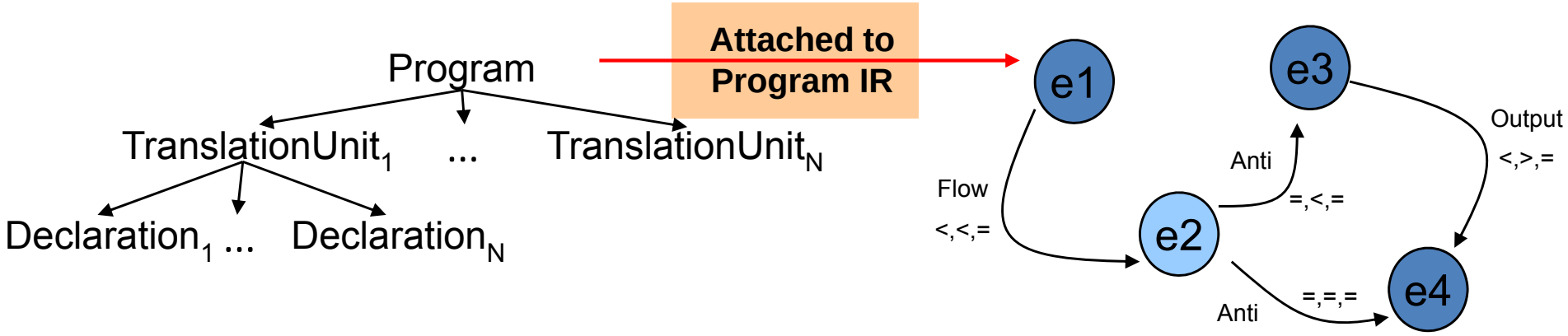
Data Dependence Interface

- Current interface functions in Cetus v1.0:
  // For all loops in a nest
  – checkLoopCarriedDependenceForGraph()
  // For a given loop in the nest
  – checkParallel(loop)
  // For array accesses and statements within the nest
  – getDependenceArcsFromTo(expr1, expr2)
  – getDependenceArcsFromTo(stmt1, stmt2)

- Current Limitations in Cetus v1.0:
  – Passing information between passes not ideal (i.e. transformation that modifies dependence information)
  – Interface to dependence information is local to each pass and not managed as one standard interface
  – Redundant dependence analysis by each dependence information seeking pass

Data Dependence Interface – Future Release

- Getting Data Dependence Information
  - DependenceAnalysis run once per IR configuration and output DD Graph attached to Program IR
  - Program IR related DD graph common to all optimization/analysis passes
  - Loop Parallelization pass and other analysis passes only query dependence graph, do not run dependence analysis
  - DD graph must be regenerated explicitly by pass writer following IR transformation and changes visible to all compiler passes



Data Dependence Interface – Future Release

- Using dependence information for parallelization

```
boolean is_parallel;
DDGraph pdg = program.getDDGraph();
// Check eligibility for dependence testing
if (LoopTools.checkDataDependenceEligibility(loop)==true) {
        /* check if scalar dependences are present using
         * privatization and reduction variable information */
        if (LoopTools.scalarDependencePossible(loop)==true)
                is_parallel = false;
        // check if array loop carried dependences exist
        else if (pdg.checkLoopCarriedDependence(loop)==true)
                is_parallel=false;
        // No dependences
        else
                is_parallel=true;
}
else
        is_parallel=false;

return is_parallel;
```

```
/* After transformation pass such as
Loop-distribution, generate new
dependence graph that is
automatically attached to Program IR,
overwriting the old dependence graph
*/
DDTDriver ddt_driver =
        new DDTDriver(program);
ddt_driver.start();
```

Frequently Asked Questions (& Answers)

- **Having problems compiling Cetus?**
  The source code for Cetus is located in the /src/ directory of your download version. Several options are provided to compile Cetus, these are provided in the Cetus release notes (included in your download), and will be made available via the manual that is under development. One of the most important issues to look out for is the java classpath environment variable. While using each of the build scripts provided with the download, please ensure that you are setting the environment variables correctly to point to the right locations that contain your libraries.

- **How do I run Cetus?**
  Cetus can be run from the command line by invoking cetus.exec.Driver. Details of this command can be found in the release notes, as well as the manual.

- **What tools do I require in order to run Cetus?**
  You will need to have the following on your computer
  - JAVA 2 SDK, SE 1.5.x (or later)
  - ANTLRv2
  - GCC

- **How do I learn more about the Cetus infrastructure?**
  Please refer to the section "Papers describing the infrastructure" on the documentation page.

- **How do I reference Cetus in my work?**
  We would recommend you use the publication that is most relevant to your work from the section on "Papers describing the infrastructure“ on the Cetus website.

Frequently Asked Questions (& Answers)

- **Does Cetus include a C++ frontend?**
  The current release does not support C++. We have a mostly functional C++ parser that is separate from Cetus. The cctreewalker directory in your source directory for Cetus provides dump routines to interface with external C++ parser tools such as Flex/Bison and generate IR. We cannot provide additional information regarding the correctness or scope of this implementation due to insufficient testing. Look out for more information in later releases.

- **Which SDK/JRE is needed?**
  We are using the Java 2 SDK, SE 1.5.0. The documentation and SDK are available from Sun's website.

- **How do I inform the developers of Cetus about bugs that I have come across?**
  We are always looking forward to receiving feedback (good or bad!) from the users of Cetus. Either provide us feedback through the Bugzilla support system or by sending us email through our contact email provided on the homepage.

- **What is Cetus currently being tested on?**
  We are currently working with the following benchmarks on Cetus.
  - SPECCPU 2006
  - SPECOMP 2001
  More information about these suites can be found on the SPEC website at www.spec.org.
  - NPB2.3 (OpenMP C versions)
  More information about the NAS Parallel Benchmark suite can be found at their website.

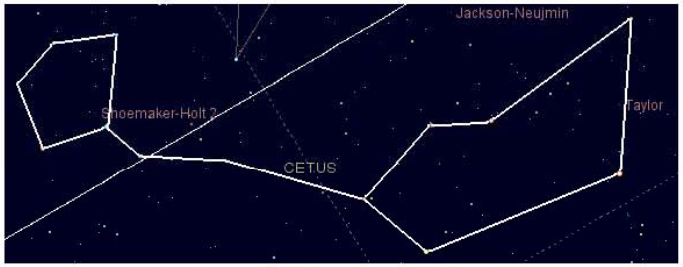
Download Cetus (cetus.ecn.purdue.edu)

PURDUE UNIVERSITY

CETUS

– A SOURCE-TO-SOURCE COMPILER INFRASTRUCTURE FOR C PROGRAMS

- Introduction
- Documentation
- Publications
- People
- Cetus Mailing List
- News
- Download Cetus
- Cetus Support
- FAQ
- Internal



The constellation Cetus is a sea monster. A *C* monster.

Contact us at: cetus@ecn.purdue.edu

Cetus 1.0 (September 21, 2008) Now Available!

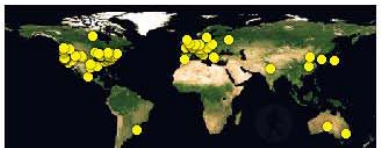
Cetus is a compiler infrastructure for the source-to-source transformation of software programs. It currently supports ANSI C and is under development to support C++. Since its creation nearly 4 years ago, it has grown to over 45,000 lines of Java code, been made available publicly on the web, and has become a basis for several research projects.

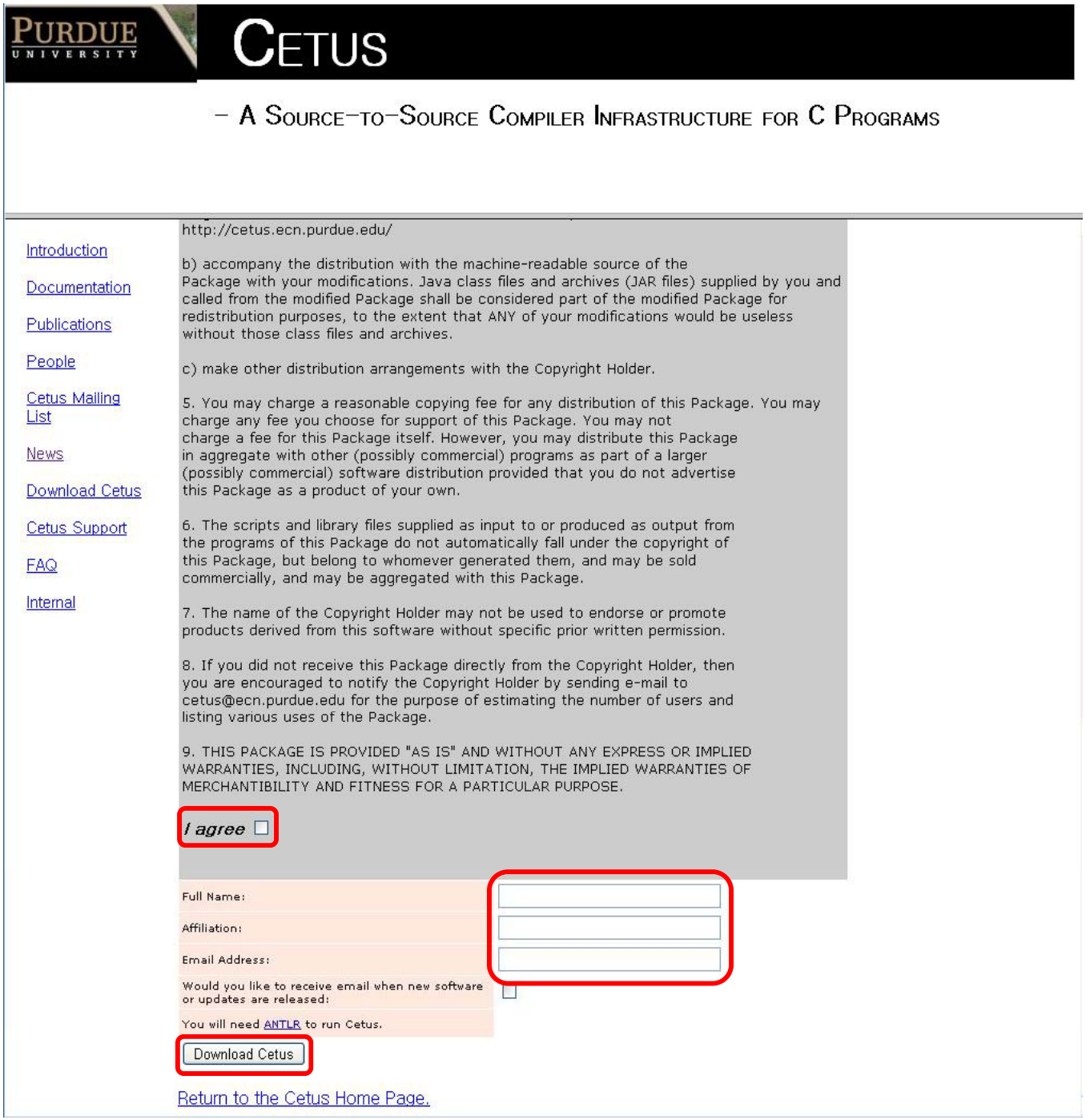
The Cetus infrastructure has users in the following countries:
United States, Brazil, France, Germany, India, Spain, South Korea, Taiwan.

This work is supported in part by the National Science Foundation. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

Visitors

213	3	2	1
17	3	2	1
11	3	2	1
10	3	2	1
8	3	1	1
8	3	1	1
8	3	1	1




PURDUE UNIVERSITY
CETUS
– A SOURCE-TO-SOURCE COMPILER INFRASTRUCTURE FOR C PROGRAMS
Introduction
Documentation
Publications
People
Cetus Mailing List
News
Download Cetus
Cetus Support
FAQ
Internal
http://cetus.ecn.purdue.edu/
b) accompany the distribution with the machine-readable source of the Package with your modifications. Java class files and archives (JAR files) supplied by you and called from the modified Package shall be considered part of the modified Package for redistribution purposes, to the extent that ANY of your modifications would be useless without those class files and archives.
c) make other distribution arrangements with the Copyright Holder.
5. You may charge a reasonable copying fee for any distribution of this Package. You may charge any fee you choose for support of this Package. You may not charge a fee for this Package itself. However, you may distribute this Package in aggregate with other (possibly commercial) programs as part of a larger (possibly commercial) software distribution provided that you do not advertise this Package as a product of your own.
6. The scripts and library files supplied as input to or produced as output from the programs of this Package do not automatically fall under the copyright of this Package, but belong to whomever generated them, and may be sold commercially, and may be aggregated with this Package.
7. The name of the Copyright Holder may not be used to endorse or promote products derived from this software without specific prior written permission.
8. If you did not receive this Package directly from the Copyright Holder, then you are encouraged to notify the Copyright Holder by sending e-mail to cetus@ecn.purdue.edu for the purpose of estimating the number of users and listing various uses of the Package.
9. THIS PACKAGE IS PROVIDED "AS IS" AND WITHOUT ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, WITHOUT LIMITATION, THE IMPLIED WARRANTIES OF MERCHANTIBILITY AND FITNESS FOR A PARTICULAR PURPOSE.
I agree
Full Name:
Affiliation:
Email Address:
Would you like to receive email when new software or updates are released:
You will need ANTLR to run Cetus.
Download Cetus
Return to the Cetus Home Page.

Backup Slide

Loop Parallelization Results with the latest Cetus

NPB	#LOOPS	#MANUAL	#AUTO	#MANUAL ONLY
BT	223	54	104	27
CG	41	22	22	4
EP	11	1	5	1
FT	51	6	6	4
IS	14	1	8	0
LU	171	29	130	10
MG	77	12	7	12
SP	314	70	208	7
TOTAL	902	195	490	65

SPEC OMP 2001	#LOOPS	#MANUAL	#AUTO	#MANUAL ONLY
320.equake	83	11	20	11
330.art	75	5	6	5
332.ammp	250	7	16	7
TOTAL	408	23	42	23