DECOMPILING JAVA

PART II: CONTROL FLOW

David Foster
12th Grade
Chamblee High School

OBJECTIVE

The objective of this project is to derive a method for analyzing patterns of control flow in Java bytecode. This analysis, called *control flow analysis* (CFA), would permit control flow constructs in bytecode to be translated into equivalent control flow constructs in Java source code.

CONTINUATION

This project is a continuation of last year's project, which involved writing a Java-language decompiler. The particular algorithm described in this paper, *control flow analysis*, is a critical and highly complex piece of the decompiler's implementation that was not finalized at the time last year's project was presented. Even though this analysis represents only one of the decompiler's many algorithms, it is sufficiently complex to merit its own paper.

DESIGN GOALS

- **It is desired that CFA be capable of analyzing all patterns of control flow in Java bytecode that are directly expressible in Java source code.** This covers all patterns of control flow that could be found in a `.class` file generated by a Java compiler.

- **It is not required that CFA deal with *untranslatable constructs* for which there is no direct translation to Java source code.** Such constructs include:
    - unstructured locking,
    - partially overlapping exception-handlers,
    - inline subroutines (that use `jsr` or `ret`),
    - exceptional regions with multiple startpoints (aka *headers*), and
    - synchronized regions with multiple startpoints (aka *headers*).

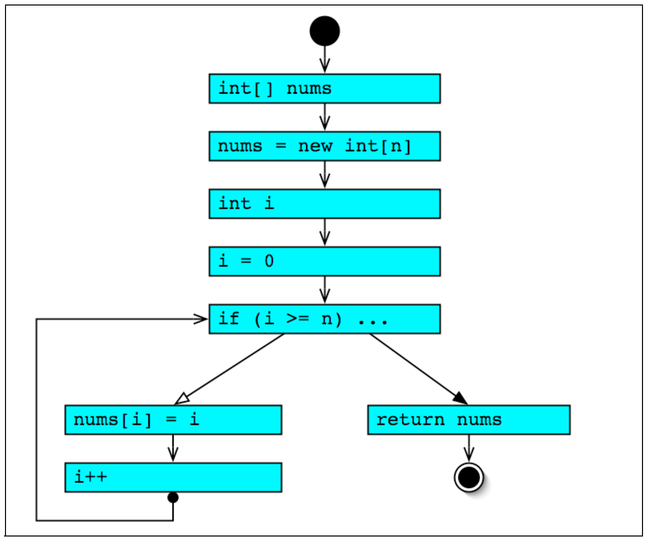
TABLE OF CONTENTS

FORMAT OF THE INPUT TO CFA

Control flow analysis takes a representation of program code as its input. This representation consists of a *graph* of *commands*.

Graphs of Commands

A *graph of commands* consists of a set of commands, the branches that link them together, and the set of exception handlers (possibly empty) that protect them.



Graph of Commands (Example)

The set of *commands* and *branches* in the diagram above collectively form one *graph of commands*.

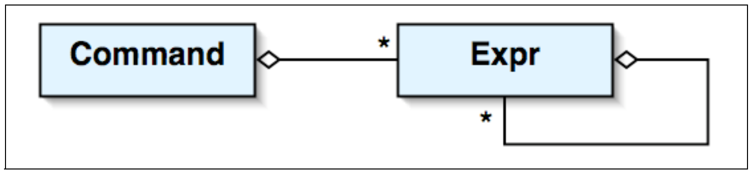
NOTE: **Only *reducible*[1] graphs of commands may be used as input for CFA.**

Since the Java language is a structured programming language, it does not contain constructs for creating irreducible control flow constructs (although such constructs are possible in Java `.class` files).

[1] A *reducible* graph of commands is one in which every *cycle* (loop) has a unique startpoint (sometimes called a *header*).

Commands

Commands are programmatic abstractions for the individual instructions contained in program code. These were first introduced in last year's research for creating a Java decompiler program. Commands are capable of performing basic operations such as assigning a value to a local variable or evaluating an expression.



Anatomy of a Command

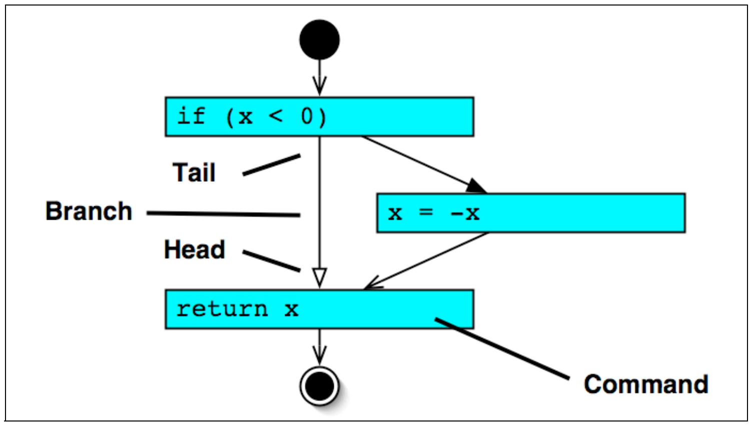
Types of Commands

Control flow analysis requires every command in its input to be one of the following types:

Type	Written Form	Description
AssignCommand	`assign`	Change the value of one variable to match that of another.
EvalCommand	`eval`	Evaluates an expression and discards the result. *Special Use:* constructor invocations
IfCommand	`if`	Performs a conditional branch, based on the value of a boolean expression.
SwitchCommand	`switch`	Performs a conditional branch, based on the value of an integral expression.
ReturnCommand	`return`	Returns to the caller of the method, passing a return value if the method is non-`void`.
ThrowCommand	`throw`	Throws an exception.
MonitorEnterCommand	`monitorenter`	Acquires a synchronization lock on an object.
MonitorExitCommand	`monitorexit`	Releases a lock held on an object.
InfiniteLoopCommand	`infinite_loop`	Loops forever. Equivalent to: `while (true) {}`

Branches

Commands on their own, however, are not sufficient to represent control flow in program code. To do so, we introduce the concept of *branches* between commands. After the computer executes a command, it decides which command to execute next by picking one of the *branches* leading from the original command and following it to the command to be executed next.



Anatomy of a Graph of Commands

Each branch is traversable in the direction leading from the branch's *tail* to its *head*. A branch is said to *lead from* its tail and to *lead to* its head. It is permissible for there to be more than one branch with the same (*head*, *tail*) pair. It is also permissible for a branch's *head* and *tail* to be equal (i.e. the branch links a node to itself).

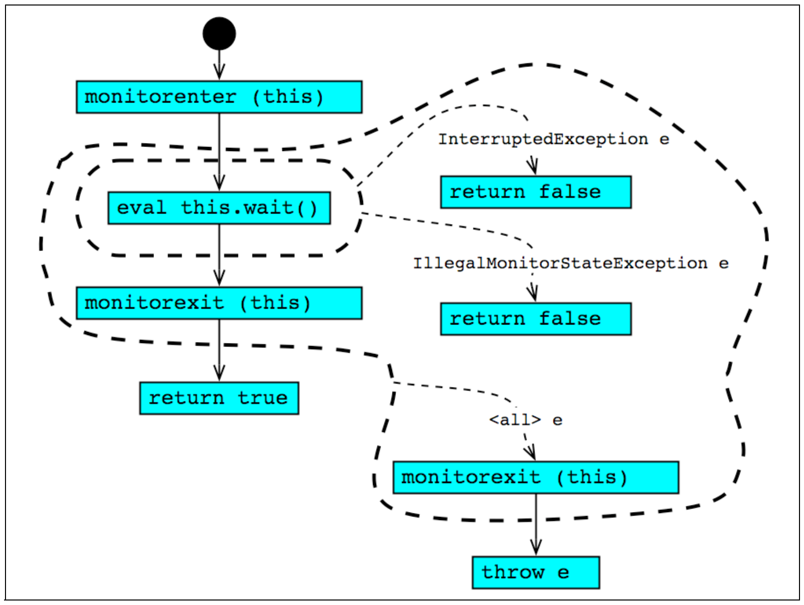
Exception Handlers

More complex command-graphs may also include regions protected by *exception handlers*. In diagrams, such regions are denoted by dashed lines surrounding groups of commands.

<u>Constraints on Exceptional Constructs</u>

- It is not permitted for any two *exception handlers* to exist that have partially (but not completely) overlapping protected regions. More formally, there must not exist any pair of exception handlers (**A**,**B**) such that (**A**.region ∩ **B**.region) ∉ {Ø, **A**.region, **B**.region}. `try-catch` constructs in the Java language are always structured in a way that conforms to this restriction.

Example of a Complex Graph of Commands



Complex Graph of Commands (Example)

In this diagram:

- The black circle denotes the *startpoint* of the method.
- Colored rectangles denote *commands*.
- Arrows denote *branches* between different commands.
- Thick dashed loops surround regions protected by one or more *exception handlers*.
- Dashed arrows leading from think dashed loops denote *exceptional branches* leading from the region protected by an *exception handler* to its *handlerPC*.

PHASE 1A: PREPROCESSING STAGE

The first phase of control flow analysis involves partitioning the input graph of commands into a set of *structured subgraphs*.

Preprocessing Stage

Before the input graph of commands is partitioned, a preprocessing stage is run that:

1. gathers information about exceptional control flow constructs,
2. gathers information about synchronized control flow constructs, and
3. removes all `monitorexit` commands so that successive stages need not worry about them.

1. Gathering Information about Exceptional Control Flow Constructs

It can be deduced that:

- The semantics of an exception-handler and a catch-clause are the same: to protect a region of code, catching all exceptions thrown within the protected region that match the type of exception it declares it can handle. Therefore, **an individual exception handler corresponds to an individual catch-clause of a Java-language try-catch-statement.**
- Catch-clauses *inherit* the region of code they protect from the try-statement they are associated with. Exception-handlers, on the other hand, *specify* the region of code they themselves protect. Therefore, **multiple exception-handlers that specify the same protected-region correspond to catch-clauses that are associated with a *common* try-catch-statement.**
  - Try-statements in Java source code protect all statements that their associated try-block lexically encloses (and no others). Therefore, a try-statement **A** will lexically enclose a try-statement **B** if and only if **B**'s protected region is a proper subset of **A**'s protected region.

In consideration of the preceding deductions, **we define an *exception handler group* to be a set of exception-handlers that protect the same region of commands**. With this definition, the members of an exception handler group correspond to catch-clauses associated with a common try-catch-statement. Therefore, **an individual *exception handler group* corresponds to an individual try-catch-statement.**

In light of this:

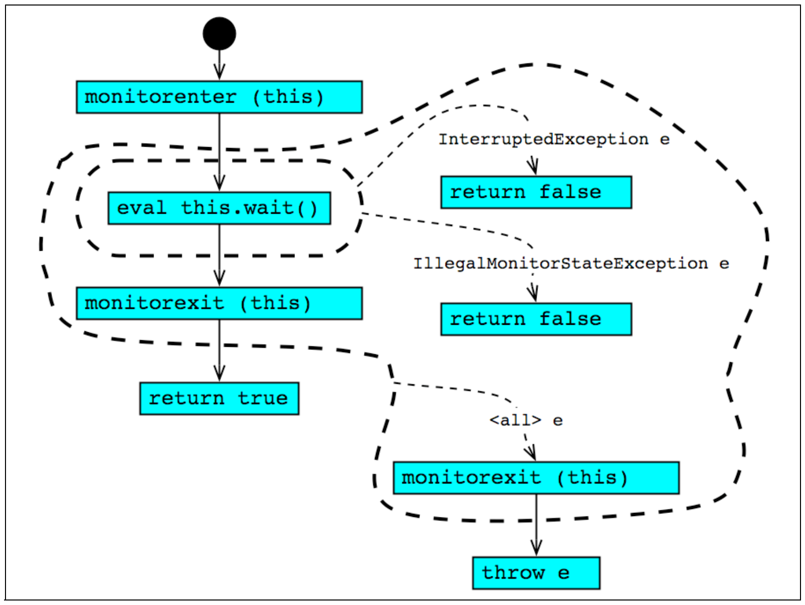
- **The region that an exception-handler-group A protects will be a proper superset of the region that an exception-handler-group B protects if and only if the try-catch-statement that A corresponds to lexically encloses the try-catch-statement that B corresponds to.**

- More formally, given exception-handler-groups **A** and **B**,
  (**A**.protectedRegion ⊃ **B**.protectedRegion) iff
  (**A**.tryStatement lexically encloses **B**.tryStatement).

1.1 Organizing Exception Handlers into Groups

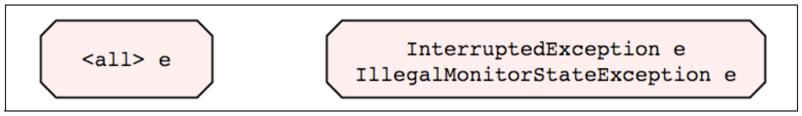
CFA organizes exception-handlers into groups to discover which exception-handlers (or equivalently, which catch-clauses) are associated with which try-catch-statements.

Exception handlers are organized into groups based of the regions of commands they protect. Multiple exception handlers protecting the same set of commands will be organized into the same *exception handler group*.



Graph of Commands (Example Input)

In the above command-graph, the exception handlers for `InterruptedException e` and `IllegalMonitorStateException e` would be organized into the same group (since they protect the same region) and the exception handler for `<all> e` would be placed into its own separate group.



Set of Exception Handler Groups (Example Output)

1.2 Arranging Exception Handler Groups into a Nesting Hierarchy

After all *exception handler groups* have been formed, CFA arranges them into a *nesting hierarchy* based on the command-regions they protect.

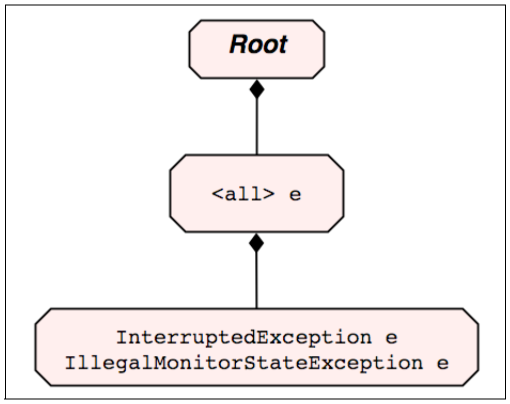
Given an *exception handler group* **A** and another group **B** in such a *nesting hierarchy*:

- If **A**.region ⊃ **B**.region, **A**'s node will be designated as an ancestor of **B**'s node.
- If **A**.region ⊂ **B**.region, **A**'s node will be designated as a descendant of **B**'s node.
- If (**A**.region ∩ **B**.region) ∉ {Ø, **A**.region, **B**.region}, then **A** and **B**'s regions partially intersect, which is a violation of our input constraints. Therefore such a case need not be handled.

With this definition of a *nesting hierarchy*'s structure, it can be deduced that:

- If (**A**.node is an ancestor of **B**.node), **A**.tryStatement lexically encloses **B**.tryStatement.
- If (**A**.node is a descendant of **B**.node), **B**.tryStatement lexically encloses **A**.tryStatement.

In our example, the *exception handler groups* would be organized into the following *nesting hierarchy*:



Nesting Hierarchy (Example Output)

2. Gathering Information About Synchronized Control Flow Constructs

It can be deduced that:

- The semantics of a *synchronized-statement* that lexically encloses a region **R** invoked on object **T** in Java source code are:
  - obtain a lock on **T** when execution enters **R**,
  - hold the lock on **T** while execution remains within **R**, and
  - release the lock on **T** when execution exits **R** (no matter how it exited).
- The semantics of a *monitorenter* command $\mathbf{C_1}$ invoked on an expression **T** are:
  - obtain a lock on **T** upon the successful execution of $\mathbf{C_1}$, and
  - hold the lock on **T** until execution encounters (and successfully executes) a *monitorexit* command $\mathbf{C_2}$ invoked on **T**, releasing the lock on **T**.
- No command other than a *monitorenter* command can obtain a lock on an object.
- No command other than a *monitorexit* command can release a lock on an object.

Therefore, **the semantics of a *synchronized-statement* in Java source code can only be expressed in terms of monitorenter and monitorexit commands.** It should be noted, however, that *monitorenter* and *monitorexit* commands could be used to express semantics that are not possible in Java source code (i.e. *untranslatable constructs*).

Untranslatable Construct: Unstructured Locking

The Java Virtual Machine (JVM) Specification (2nd Edition) explicitly states:

- **"Although a compiler for the *Java* programming language normally guarantees *structured use of locks*, there is no assurance that all code submitted to the Java virtual machine will obey this property.** Implementations of the Java virtual machine are permitted but not required to enforce [...] *structured locking*." (§8.13)
- *Structured locking* describes the restriction that, "during a method invocation, every *unlock* operation on [a lock] **L** must match some preceding *lock* operation on **L**." (§8.13)

Therefore, since the JVM permits *unstructured locking*, and since the Java language lacks any construct to perform *unstructured locking*, ***unstructured locking* is an *untranslatable construct*.** However, **since *unstructured locking* is an *untranslatable construct*, our design goals do not require that we handle it.**

Identifying Synchronized Regions

For those synchronized constructs that can be represented in Java source code, **CFA identifies all cases where there exists a command-region *R* that is "sandwiched" (along all execution paths) between:**

- **an individual *monitorenter* command C_1 and**
- **a set of one or more *monitorexit* commands $C_2[]$ that are invoked on a common target-object *T*.**

In such cases, CFA designates there to be a *synchronized region* that is associated with *R*, C_1, and $C_2[]$.

If, after identifying all such *synchronized regions*, there exist remaining *monitorenter* and *monitorexit* commands that are not associated with any *synchronized region*, an assertion-failure error is raised.

3. Removing *monitorexit* Commands

The last task of the preprocessing stage is to remove all *monitorexit* commands. This is to relieve successive stages from needing to handle them.

PHASE 1B: PARTITIONING STAGE

Now that information about the exceptional and synchronized constructs in the method has been analyzed, we are ready to partition the input graph into *structured subgraphs*.

Different types of *structured subgraphs* are used to describe to different kinds of control flow patterns found in the input graph of commands. For example, a *SinglyRootedSubgraph* (which is a type of structured subgraph) is capable of describing the control flow patterns present in a chunk of a Java-language *statement block*.

Partitioning via Parsing

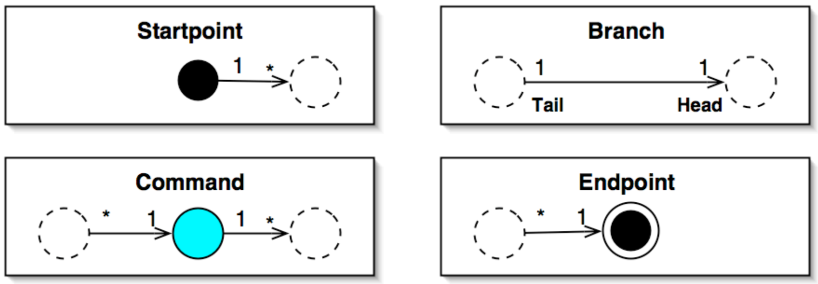
The method for partitioning the input graph into structured subgraphs can be thought of as a form of *parsing*. In this context, the *input string* is the graph of commands, the *nonterminals* are *structured subgraphs*, the *terminals* are commands, and the partitioned output graph is an *abstract syntax tree*.

The grammar described in this section will match any *reducible* graph of commands that conforms to our input constraints.

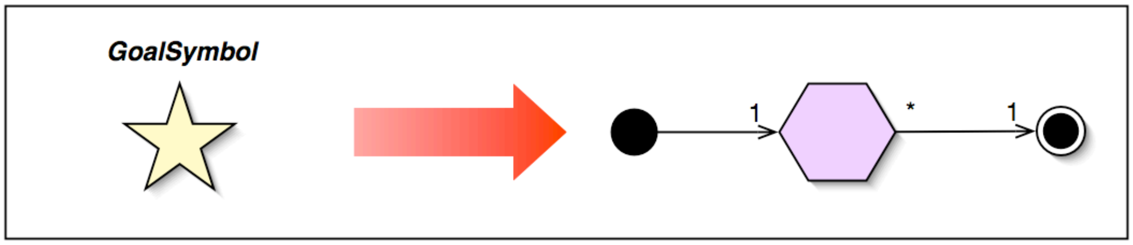
Production Rules

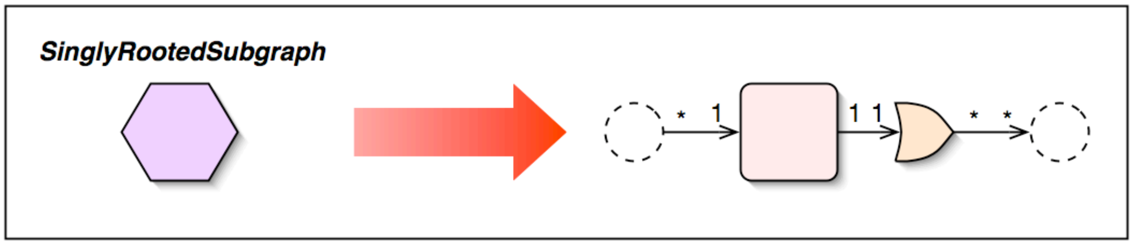
Since our *input string* is non-linear, the *production rules* for our *generative grammar* cannot be easily described using standard notations such as Backus-Naur form (BNF). Instead, we will use an informal notation that relies on diagrams and natural language to describe the production rules.

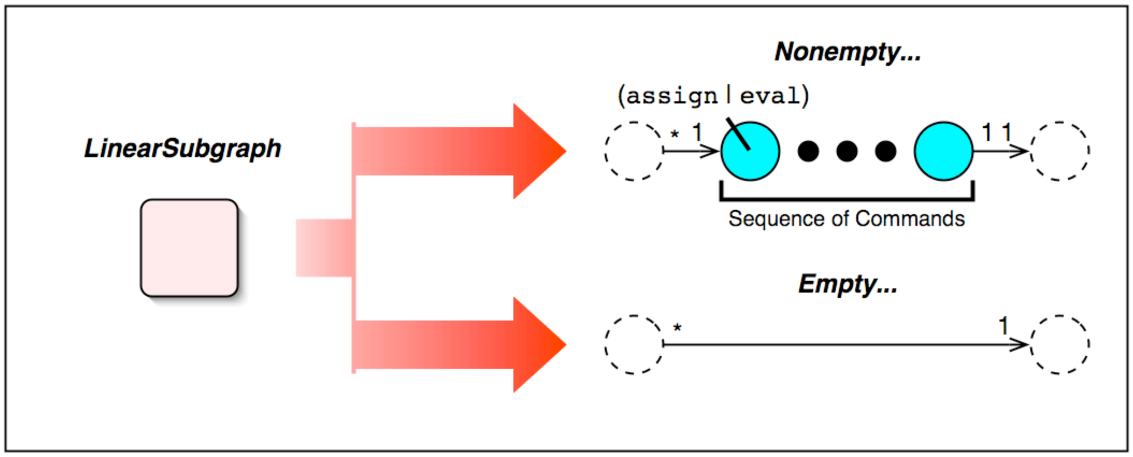
Terminals

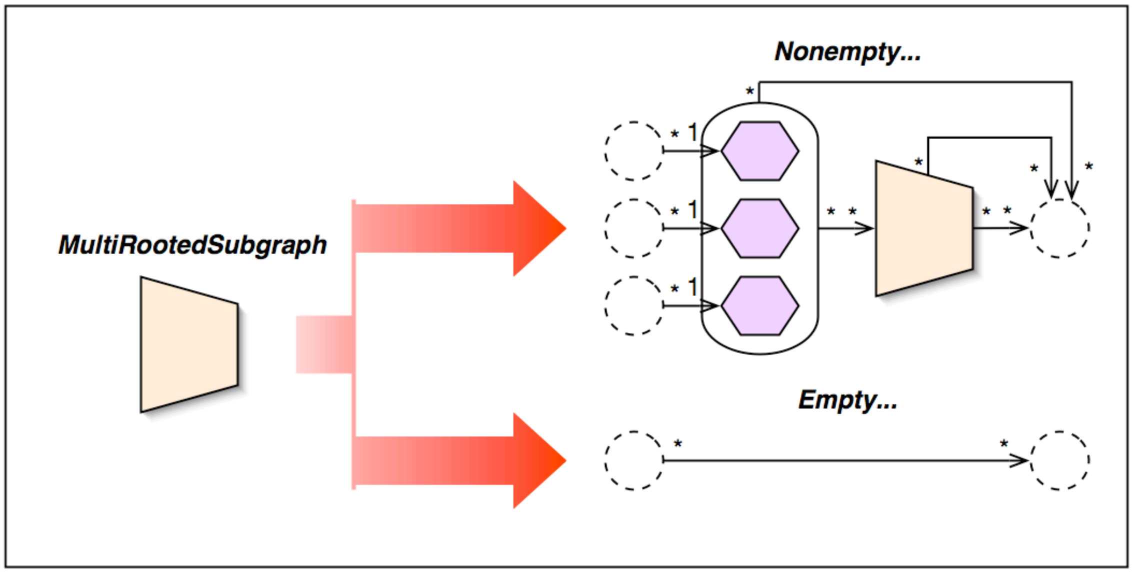


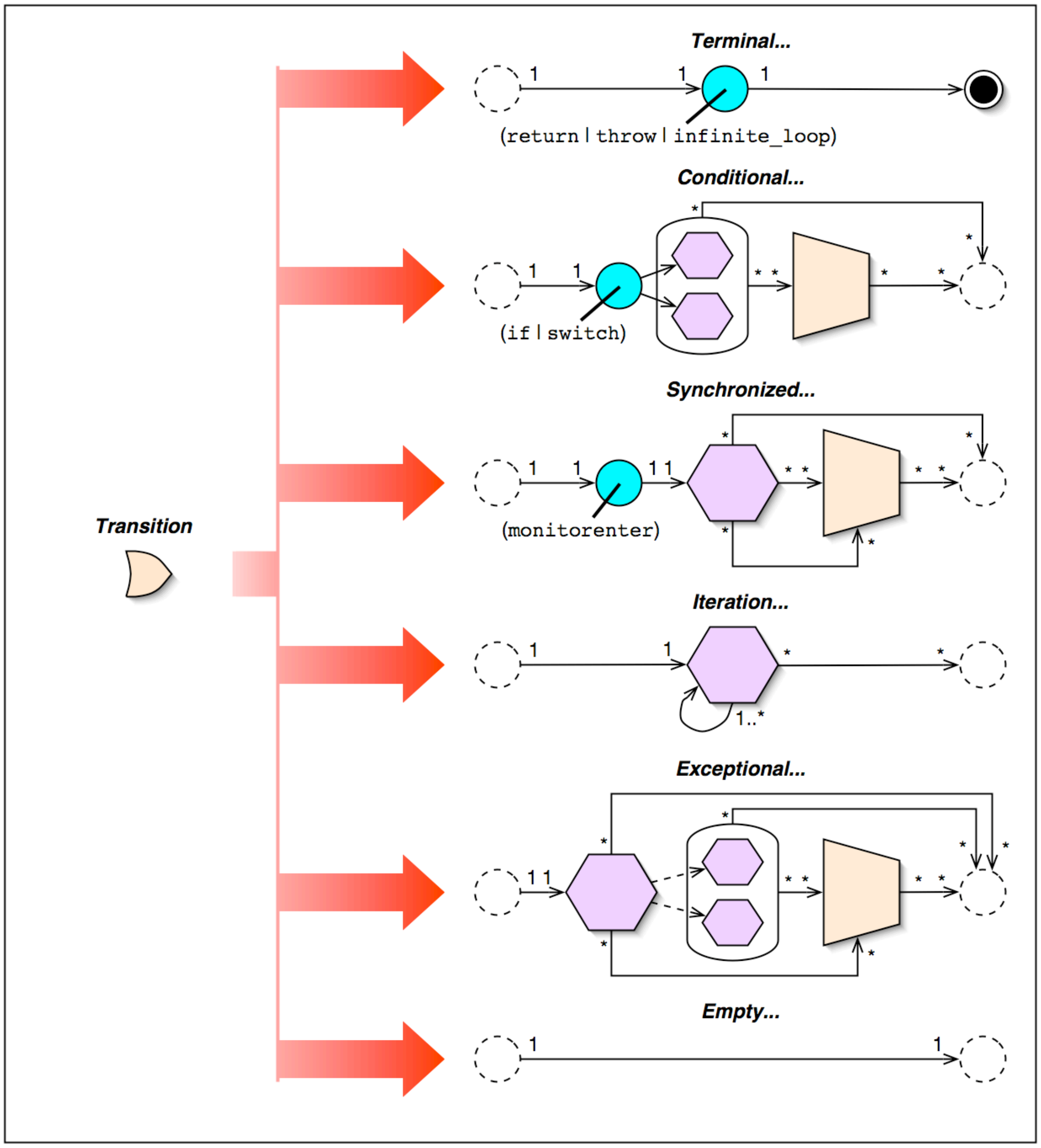
Nonterminals










Transition
Terminal...
1
1
1
(return | throw | infinite_loop)
Conditional...
1
1
(if | switch)
Synchronized...
1
1
1 1
(monitorenter)
Iteration...
1
1
1..*
Exceptional...
1 1
Empty...
1
1

Additional Constraints

Some of the production rules have additional constraints that are not shown in the above diagrams.

ConditionalTransition

- The set of commands in each contained *SinglyRootedSubgraph* is equal to the set of commands *dominated*[2] by the branch entering the subgraph.

SynchronizedTransition

- The set of commands in the contained *SinglyRootedSubgraph* is equal to the set of commands sandwiched between the transition's `monitorenter` command its corresponding `monitorexit` command (which was determined during the preprocessing stage).

IterationTransition

- All branches leading into the contained *SinglyRootedSubgraph* target the initial command of the subgraph.

ExceptionalTransition

- For the first contained *SinglyRootedSubgraph*:
    - The set of commands in the subgraph is equal to the set of commands protected by set of *exception handlers* that the *ExceptionalTransition* is associated with.
- For every other contained *SinglyRootedSubgraph*:
    - The *initial command* of the subgraph is equal to the *handlerPC* of the individual *exception handler* that the subgraph corresponds to.
    - The set of commands in the subgraph is equal to the set of commands *dominated* by the branch entering the subgraph.

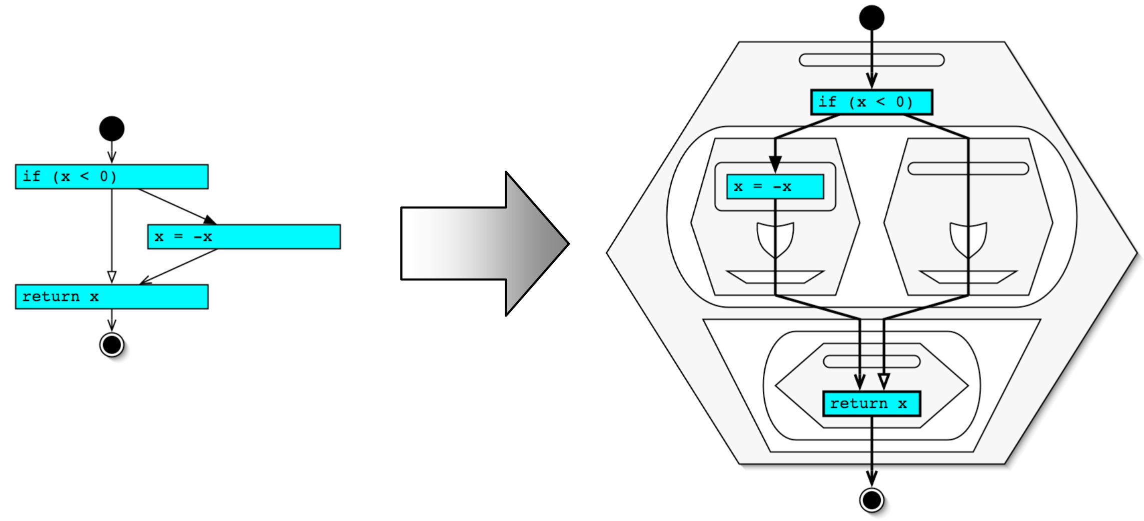
NonemptyMultiRootedSubgraph

- All contained *SinglyRootedSubgraphs* must be nonempty; no contained *SinglyRootedSubgraph* may contain both an *EmptyLinearSubgraph* and an *EmptyTransition*.

NOTE: Since all `monitorexit` commands were removed during the preprocessing stage (Phase 1A), they do not appear in any of the production rules.

[2] A command $\mathbf{C_1}$ is said to *dominate* a command $\mathbf{C_2}$ if and only if there exists at least one path from the *startpoint* to $\mathbf{C_2}$ and every such path passes through $\mathbf{C_1}$ at least once.

Applying Phase 1A and 1B to an Example



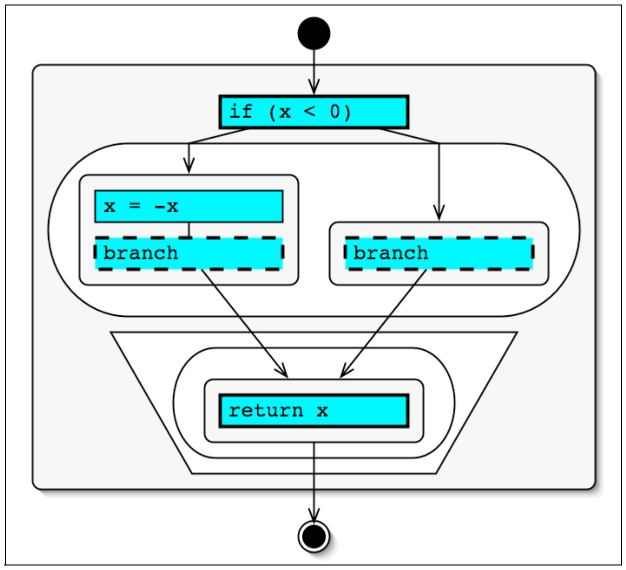
For brevity, some *Transition* nonterminals ( ) are not shown in the example's abstract syntax tree.

PHASE 2: CONSTRUCTION OF FLOW BLOCKS

The second phase of control flow analysis involves transforming the *abstract syntax tree* from the previous phase into a hierarchy of *flow blocks*, the decompiler's native representation for the control flow patterns described by the abstract syntax tree.

Flow Blocks

A flow block is roughly equivalent to a *SinglyRootedSubgraph*, except that its representation of the control flow patterns (present in a chunk of a statement block) differs slightly.



Hierarchy of Flow Blocks (Example)

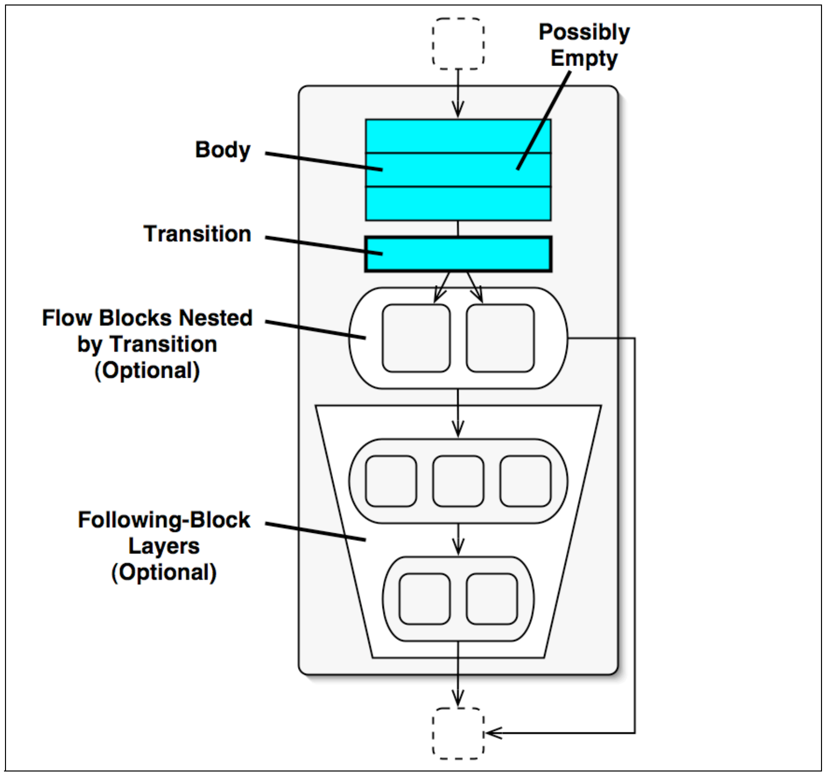
<u>Analogous Constructs: SinglyRootedSubgraphs vs. Flow Blocks</u>

SinglyRootedSubgraph	Flow Block
LinearSubgraph	*body*
Transition	*transition*
MultiRootedSubgraph	*following block layers*

Anatomy of a Flow Block

The components of a *flow block* include:

- a **body** – a linear sequence of commands (possibly empty) composed of *AssignCommands* and *EvalCommands*.
- a **transition** – a control-flow-altering construct that is composed of:
  - an optional *embedded command* and/or
  - an optional list of *contained flow blocks*
- a list of **following block layers** (possibly empty)
  - An individual *block layer* within this list of following block layers is composed of a list of *member flow blocks*.


Possibly Empty
Body
Transition
Flow Blocks Nested by Transition (Optional)
Following-Block Layers (Optional)

Anatomy of a Flow Block

Transitions

A *transition* is merely a non-linear CF construct. It therefore stands in contrast to the *body*, which is a strictly linear CF construct. Only through transitions can a program transfer control between different flow blocks.

Types of Transitions

Listed below are the types of *transitions* that can be put into a *flow block*.

Type	Behavior	Nesting?
BranchTransition	*structural*	*non-nesting*
IfTransition	functional	nesting
SwitchTransition	functional	nesting
ReturnTransition	functional	*non-nesting*
ThrowTransition	functional	*non-nesting*
SynchronizedTransition	functional	nesting
WhiteTrueTransition	*structural*	nesting
TryTransition	*structural*	nesting

Functional transitions are those transitions that embed a command; *structural transitions* are those that do not. In diagrams, *functional transitions* are drawn with a solid border whereas *structural transitions* will be drawn with a dashed border.

Nesting transitions are those transitions that can contain *flow blocks*; *non-nesting transitions* are those that cannot.

Following Block-Layers

A flow-block's *following block-layers* nests the set of commands *dominated* by the flow-block's *transition* that are not nested by the *transition*'s set of contained blocks. Each block-layer within a set of *following block-layers* dominates every *block-layer* "below" it. Branching may not occur between different *member flow blocks* contained by an individual *block layer*.

Restrictions on Branching Between Flow Blocks (and Transitions)

In this section, whenever we say "a flow block $\mathbf{B_1}$ branches to a flow block $\mathbf{B_2}$," we mean, "$\mathbf{B_1}$'s transition branches to $\mathbf{B_2}$."

Given a flow block **B** with transition **T** and following block-layers **L[]** = {$\mathbf{L_1}$, $\mathbf{L_2}$, ..., $\mathbf{L_n}$}:

- **T** (and only **T**) may branch to any of its *contained* flow blocks.
- Only those flow blocks that are *nested* by **T** may branch to **T**.
  Such branches are termed *backward branches* and **T** must be an *IterationTransition*.
- Only those flow blocks *nested* by **T** may branch to a flow block *contained* by $\mathbf{L_1}$.
- The only flow blocks not nested by $\mathbf{L_i}$ that a block nested by $\mathbf{L_i}$ can branch to are:
    - flow blocks nested by $\mathbf{L_j}$, where ($\mathbf{i < j}$),
    - a flow block that **B** may branch to.

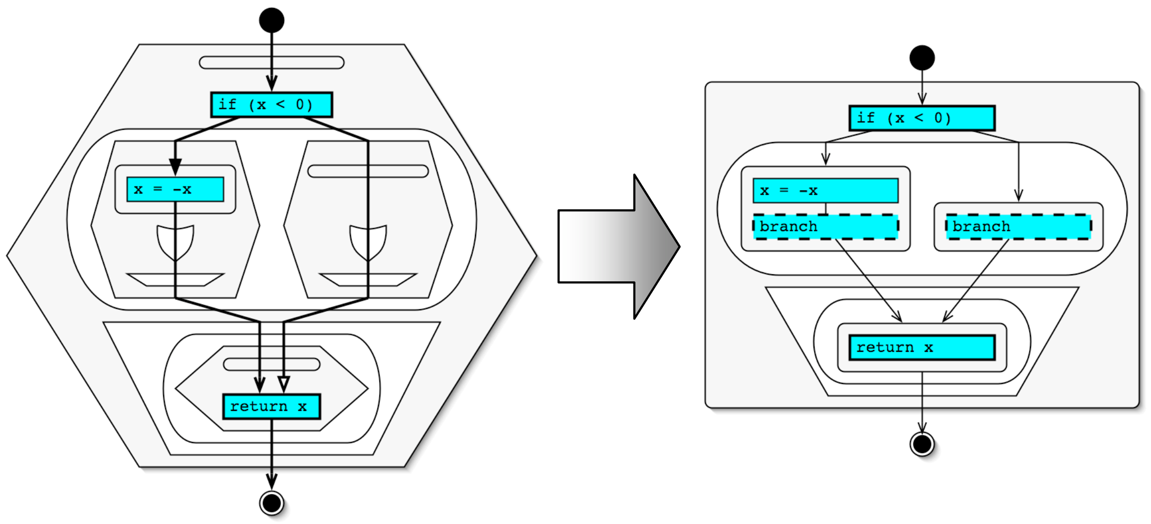
Transforming Structured Subgraphs into Flow Blocks

It is fairly straightforward to transform the *abstract syntax tree* from the previous phase into the *flow blocks* of this phase. Informal transformation rules are given below:

```
GoalSymbol[sr_subgraph] => {block}
    {block: sr_subgraph => block}

SinglyRootedSubgraph[linear_subgraph, transition] => {block}
    {block: FlowBlock[
        body: linear_subgraph => body,
        transition: transition => transition,
        following block layers: transition => FBL]}

LinearSubgraph[command_list] => {body}
    {body: Body[command_list]}

Transition => {transition, FBL}
    TerminalTransition[return]
        {transition: ReturnTransition; FBL: Ø}
    TerminalTransition[throw]
        {transition: ThrowTransition; FBL: Ø}
    TerminalTransition[infinite_loop]
        {transition: WhiteTrueTransition[loop_block = Ø]; FBL: Ø}
    ConditionalTransition[if, sr_subgraphs[2], mr_subgraph]
        {transition: IfTransition[
                    then block: sr_subgraphs[0] => block,
                    else block: sr_subgraphs[1] => block];
         FBL: mr_subgraph => FBL}
    ConditionalTransition[switch, sr_subgraphs[], mr_subgraph]
        {transition: SwitchTransition[
                    default block: *sr_subgraphs*[0] => block,
                    case blocks: *sr_subgraphs*[1..*sr_subgraphs*.length] ]
        FBL: *mr_subgraph* => FBL}
**SynchronizedTransition[***sr_subgraph*, *mr_subgraph***]**
        {transition: **SynchronizedTransition[**
                    sync block: *sr_subgraph* => block];
        FBL: *mr_subgraph* => FBL}
**IterationTransition[***sr_subgraph***]**
        {transition: **WhileTrueTransition[**
                    loop block: *sr_subgraph* => block];
        FBL: Ø}
**ExceptionalTransition**[*protected_sr_subgraph*,
                    *other_ sr_subgraphs,*
                    *mr_subgraph***]**
        {transition: **TryTransition[**
                    try block: *protected_sr_subgraph* => block,
                    catch blocks: *other_ sr_subgraphs* => block[]];
        FBL: *mr_subgraph* => FBL}
**EmptyTransition**
        {transition: **BranchTransition<***target***>**; FBL: Ø}

**MultiRootedSubgraph[***layers***]** => {FBL}
    {FBL: **FollowingBlockLayers[***layers***]**}
```

With these rules, any hierarchy of *structured subgraphs* can be transformed into a hierarchy of nested *flow blocks*.

Applying Phase 2 to an Example



PHASE 3A: CONSTRUCTION OF BLOCK SEGMENTS

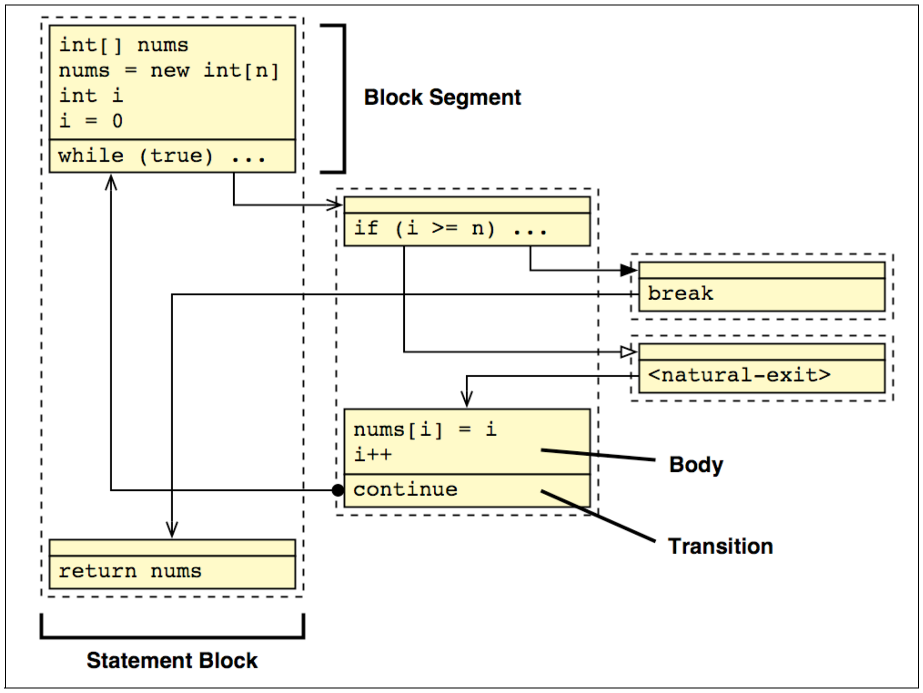
In the third phase, the hierarchy of *flow blocks* from the previous phase is transformed into a hierarchy of *statement blocks*, which contain *block segments*.

Statement Blocks

The *statement blocks* manipulated by CFA are analogous to statement blocks in Java source code. CFA *statement blocks*, unlike their Java counterparts, are not composed of statements, but rather, are composed of *block segments*.

Block Segments

A *block segment* is a chunk of a CFA statement block. Block segments consist of a linear *body*, containing a series of sequentially executed commands, and a non-linear *transition* that may transfer control to other block-segments. Transitions may optionally embed commands or contain other block-segments. Block segments maintain a reference to the next segment (if there is one) in the same block.



Hierarchy of Statement Blocks Which Contain Block Segments (Example)

Block Segments vs. Flow Blocks

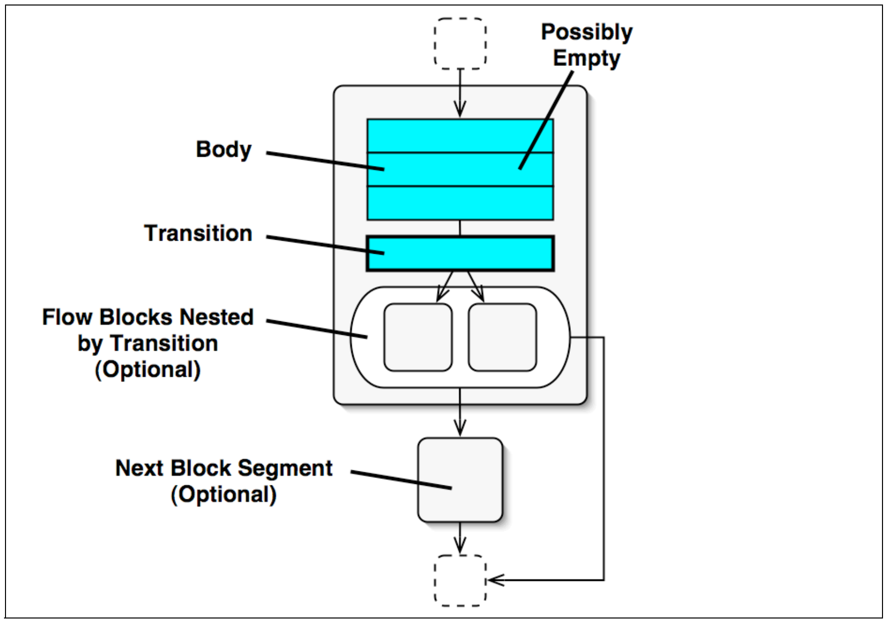
Block segments, like *flow blocks*, have a *body* and a *transition*. *Block segments*, however:

- lack *following block layers*,
- are associated with the *next block-segment* in the same statement-block, and
- can contain ***BlockTransitions*** (a new type of *transition*).

Anatomy of a Block Segment

Block segment are associated with:

- a **body**, which consists of a linear sequence of commands (possibly empty);
- a **transition**, which is a non-linear construct that may contain
    - an optional embedded command and/or
    - an optional list of contained *block segments*; and
- the **next block-segment** (if present) in the same statement-block.


Possibly Empty
Body
Transition
Flow Blocks Nested by Transition (Optional)
Next Block Segment (Optional)

Anatomy of a Block Segment

Block Transitions

A *block transition* is a structural nesting transition that always contains exactly one *block segment*. The Java-language construct most similar to a *block transition* is referred to, in this paper, as a *block statement*.[3] In the Java grammar, a *block statement* corresponds to the symbol `Statement`, rewritten as a `StatementWithoutTrailingSubstatement`, and then as a `Block`.[4] In less formal terms, a *block statement* is denoted in source code by a pair of curly braces on their own (that are not part of another statement).

[3] This construct has no official name.

[4] Gostling, James; Joy, Bill; Steele, Guy; and Bracha, Gilad. The Java Language Specification (2nd Edition). Addison-Wesley Pub Co., 1986. §14.5.

Transforming Flow Blocks into Block Segments

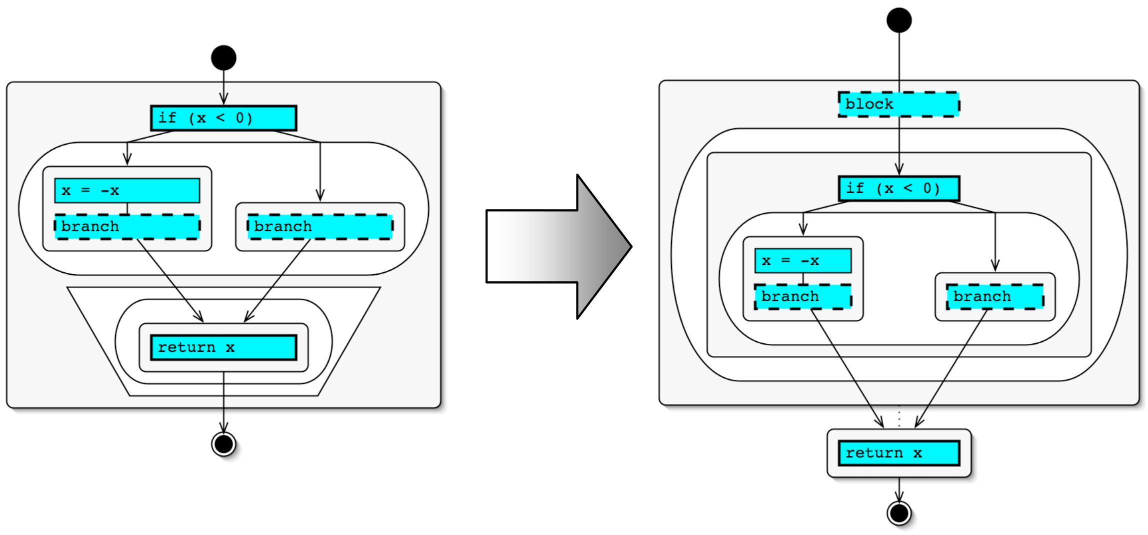
Translating a *flow block* into a *block segment* is fairly straightforward. The *body* and *transition* of the original *flow block* can be copied verbatim into the new *block segment*. The *following block layers*, however, must be dealt with specially. Informal transformation rules are given below.

FlowBlock[*body*=**B**, *transition*=**T**, *following-blocks*={}]
    => **BlockSegment**[*body*=**B**, *transition*=**T**, *next-block-segment*=Ø]

FlowBlock[*body*=**B**, *transition*=**T**, *following-blocks*={**FB**$_1$, …, **FB**$_n$}]
    => **BlockSegment**[
        body = {},
        transition =
            BlockTransition[
                FlowBlock[
                    body=**B**,
                    transition=**T**,
                    following-blocks={**FB**$_1$, …, **FB**$_{n-1}$}
                ]
            ],
        next-block-segment=**FB**$_n$
    ]

Using these transformation rules, all *flow blocks* with *following block layers* are converted into a set of nested *block segments* with *block transitions*.

Applying Phase 3A to an Example



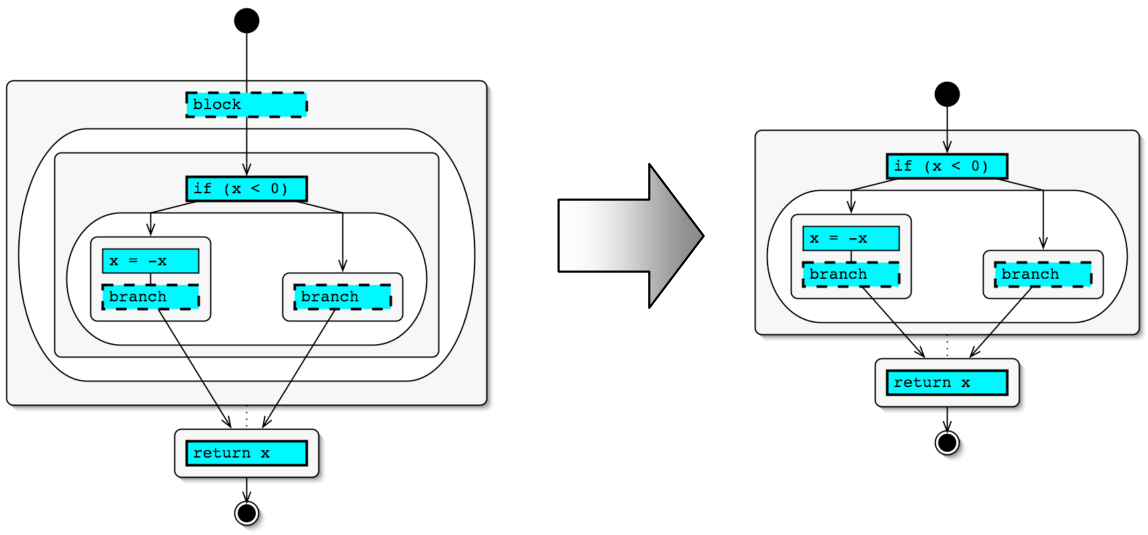
PHASE 3B: STYLISTIC TRANSFORMATIONS

The second half of phase three (which is optional) involves transforming the hierarchy of *block segments* in ways that reduce redundancies or improve the "stylistic" quality of the code.

Since this subphase consists mostly of "stylistic" transformations on the set of *block segments* and because it is optional, it will not be covered in detail. Suffice to say there are a number of transformations which:

- reduce the number of *block transitions* (by removing them where they are unnecessary);
- reduce the nesting depth of *block segments*;
- identify looping patterns involving *while(true) transitions* that can be rewritten in terms of *while transitions*, *do-while transitions*, or *for transitions*; and
- identify *finally clauses* of *try transitions*.

Applying Phase 3B to an Example



In this example, an unnecessary *block transition* was eliminated.

ENDGAME: SOURCE CODE OUTPUT

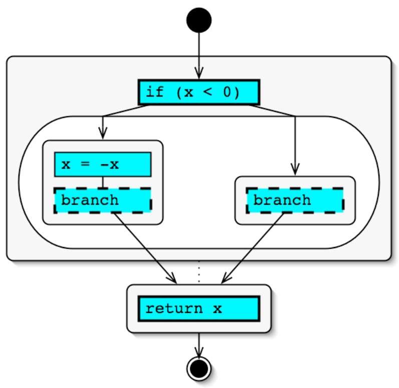
Upon the completion of phase three, control flow analysis is complete.

Now, with only minimal effort, one can translate a hierarchy of *block segments* into Java-language source-code. This translation must deal with:

- the insertion of *local variable declaration statements*
- the generation of names for those variables whose names are not specified in the `.class` file
- the generation of labels for those statements that need them (especially *iteration statements*, *block statements*, and *switch statements*)

This translation process, having little to do with control flow analysis, will not be discussed here.

Transforming an Example into Source Code



```
int abs(int x) {
     if (x < 0) {
          x = -x;
     }
     return x;
}
```

PROTOTYPE IMPLEMENTATION & TESTING

To ensure the algorithm detailed in this paper was valid, an implementation of it was created and integrated into last year's decompiler program. This implementation was then was tested on `.class` files from the standard library of the Java 2 Platform Standard Edition v.1.4.1.

Testing Procedure

The following testing procedure was automated so that it could be efficiently applied to *every* class in the J2SE v.1.4.1 standard library.

To test an individual `.class` file **C**, the automated testing unit would:

1. Run the prototype implementation, passing the name of **C** as input.
2. If the decompiler exited with an error, a **Failure** was recorded.
3. If the decompiler produced output successfully, a **Success** was recorded.

Admittedly, this testing procedure cannot identify *semantic* errors in the decompiler's output; it can only verify that none of the decompiler's internal assertions were violated.

To counter this "blind spot" for semantic errors in the automated tests, a few *manual* followup tests were run, where the decompiled source code of several classes was compared (by hand) with the original source code.

Results

As of April 4, 2005, the decompiler's complete set of algorithms (including control flow analysis) work successfully on 99.80% of the 6949 classes in the Java 2 Platform Standard Edition v1.4.1 (according to the automated tests). In none of the 14 unprocessable classes was the failure the fault of the control flow analysis algorithm (i.e., none of the assertion violations that occurred were due to the CFA algorithm being unaware of some type of control flow construct).

SUMMARY & CONCLUSION

The method of control flow analysis presented in this paper can be used on all reducible input command-graphs consisting of only those control flow patterns that are directly representable using Java-language control flow constructs (i.e., that lack *untranslatable constructs*).

Note that the control flow analysis algorithm detailed in this paper is designed for decompilers of *structured programming languages*[5] (such as Java). Adjustments would need to be made to this algorithm before it could be used by decompilers of *unstructured programming languages* (such as C).

APPLICATIONS OF THE RESEARCH

Control flow analysis is an integral algorithm for decompilers and for other programs that must decompose reducible graphs in order to identify high-level control flow structures within them.

Decompilation technology, in general, has several practical applications:

- Decompilers are often used to modify and maintain legacy systems for which source code is not available.
- The Rigi Project[6] uses decompilation technology to visualize the structure of legacy software for which documentation is either nonexistent or of poor quality.
- *Binary recompilers* are used to decompile, optimize, and recompile legacy binaries in order to incorporate modern compiler optimizations into them. "Estimating that compiler technology has contributed about 5% per year in system performance improvements over the last decade or more, the compounded performance loss to legacy binaries may be quite significant."[7]

[5] In this context, a *structured programming language* refers to a language in which all code structures (such as subroutines or code blocks) have a unique startpoint. Typically, any programming language that lacks a GOTO statement fulfills this classification.
[6] For more information on the Rigi Project, visit http://www.rigi.csc.uvic.ca/.
[7] Mudge, Trevor; Reinhardt, Steve; and Tyson, Gary. "Binary Recompilation and Combined Compiler/Architecture Enhancements Studies." http://www.eecs.umich.edu/~jringenb/binary_recomp.html

- Other types of recompilers are used for retargeting legacy code for new *processor architectures*.[8] This avoids the cost of writing an entirely new program, which would incur additional development and testing costs.
- Other types of recompilers are used to retargeting legacy code for new *source languages*. This decreases costs[9] associated with maintaining legacy code because of the increased pool of developers and expertise available for the new source language. For example, programs have already been developed for translating COBOL code into Java code.[10]

[8] ResQSoft is one such program for retargeting legacy code. More information on it can be found at http://www.resqsoft.com/engineer.html.

[9] "[A]pproximately 80% of most application budgets are allocated to software maintenance" according to Barbara Errickson-Connor of ASG (*http://www.aboutlegacycoding.com/default.htm?AURL=%2FArticles%2FV%2FV60104%2Easp*).

[10] Corporola, for example, has written **Cobol2Java**, a COBOL to Java source code translator. See http://www.corporola.com/product/cobol2java.html for more information.

REFERENCES

Aho, Alfred V.; Sethi, Ravi; and Ullman, Jeffrey D. Compilers: Principles, Techniques, and Tools. Addison-Wesley Pub Co., 1986.

Aho, Alfred V. and Ullman, Jeffrey D. Principles of Compiler Design. Addison-Wesley Pub Co., 1977.

Diestel, Reinhard. Graduate Texts in Mathematics: Graph Theory. Springer, 2000.

Gostling, James; Joy, Bill; Steele, Guy; and Bracha, Gilad. The Java Language Specification (2nd Edition). Addison-Wesley Pub Co., 2000.

Lindholm, Tim and Yellin, Frank. The Java Virtual Machine Specification (2nd Edition). Addison-Wesley Pub Co., 1999.

Nolan, Godfrey. Decompiling Java. APress, 1994.