Popular Ways to Debug

- Using “printf”
  - Often mocked, viewed as unscientific
  - In fact, just an easy way to apply dynamic/log analysis and perform experiments
  - If you ask the right questions, *print* can be a great debugging tool
    - Supports scientific debugging
    - I have no lessons here, other than to print intelligently, not blindly grope around

Popular Ways to Debug

- Using a debugger
  - Usually thought of as “more scientific”
  - It can be!
  - A debugger is good when you want to:
    - Inspect closely what happens to some values
    - Slow down and carefully watch things during one part of a run
    - Get information across a lot of state at once
  - Hard to make guidelines, but in general printf is for “across time” and debuggers are for “across state”

Basic Assertions

```
if (divisor == 0) {
    printf("Division by zero!");
    abort();
}
```

Specific Assertions

assert (divisor != 0);

Implementation

```
void assert (int x)
{
  if (!x)
  {
    printf("Assertion failed!\n");
    abort();
  }
}
```

Execution

```
$ my-program
Assertion failed!
Abort (core dumped)
$
```

Better Diagnostics

```
$ my-program
divide.c:37:
   assertion ‘divisor != 0’ failed
Abort (core dumped)
$ _
```

Assertions as Macros

```
#ifndef NDEBUG
#define assert(ex) \
((ex) ? 1 : (cerr << __FILE__ << ":" << __LINE__ \
          << ": assertion ‘" #ex "’ failed\n", \
               abort(), 0))
#else
#define assert(x) ((void) 0)
#endif
```

Ensuring Sanity (A Time Class)

```
class Time {
public:
    int hour();     // 0..23
    int minutes();  // 0..59
    int seconds();  // 0..60 (incl. leap seconds)

    void set_hour(int h);
    …
}
```

Any time from 00:00:00 to 23:59:60 is valid

Ensuring Sanity

```
void Time::set_hour(int h)
{
    // precondition
    assert (0 <= hour() && hour() <= 23) &&
           (0 <= minutes() && minutes() <= 59) &&
           (0 <= seconds() && seconds() <= 60);
    …
    // postcondition
    assert (0 <= hour() && hour() <= 23) &&
           (0 <= minutes() && minutes() <= 59) &&
           (0 <= seconds() && seconds() <= 60);
}
```

Ensuring Sanity

```
bool Time::sane()
{
    return (0 <= hour() && hour() <= 23) &&
           (0 <= minutes() && minutes() <= 59) &&
           (0 <= seconds() && seconds() <= 60);
}

void Time::set_hour(int h)
{
    assert (sane());  // precondition
    ...
    assert (sane());  // postcondition
}
```

Ensuring Sanity

```
bool Time::sane()
{
    return (0 <= hour() && hour() <= 23) &&
           (0 <= minutes() && minutes() <= 59) &&
           (0 <= seconds() && seconds() <= 60);
}
```

- sane() is the *invariant* of a Time object:
  - holds *before* every public method
  - holds *after* every public method

Ensuring Sanity

```
bool Time::sane()
{
    return (0 <= hour() && hour() <= 23) &&
           (0 <= minutes() && minutes() <= 59) &&
           (0 <= seconds() && seconds() <= 60);
}
```

```
void Time::set_hour(int h)
{
    assert (sane());
    ...
    assert (sane());
}
```

same for set_minute(), set_seconds(), etc.

Locating Infections

- **Precondition failure = infection *before* method**
- **Postcondition failure = infection *within* method**
- **All assertions pass = no infection**

```
void Time::set_hour(int h)
{
   assert (sane());  // precondition
   …
   assert (sane());  // postcondition
}
```

Relative Debugging

- **Rather than checking a spec, we can also compare against a *reference run:***
  - **The environment has changed–e.g. ports or new interpreters**
  - **The code has changed**
  - **The program has been reimplemented**

Relative Assertions

- We compare two program runs
- A *relative assertion* compares variable values across the two runs:
  - assert \ p1::perimeter@polygon.java:65 == \ p0::perimeter@polygon.java:65
- Specifies when and what to compare

Concepts

- Assertions catch infections before they propagate too far
- Assertions check preconditions, postconditions and invariants
- Assertions can serve as specifications
- A program can serve as reference to be compared against

System Invariants

- Some properties of a program must hold over the entire run:
  - must not access data of other processes
  - must handle mathematical exceptions
  - must not exceed its privileges
- Typically checked by hardware and OS

Memory Invariants

- Even within a single process, some invariants must hold over the entire run
  - code integrity
  - data integrity
- This is a major issue in C and C++

Heap Misuse

s = malloc(30)

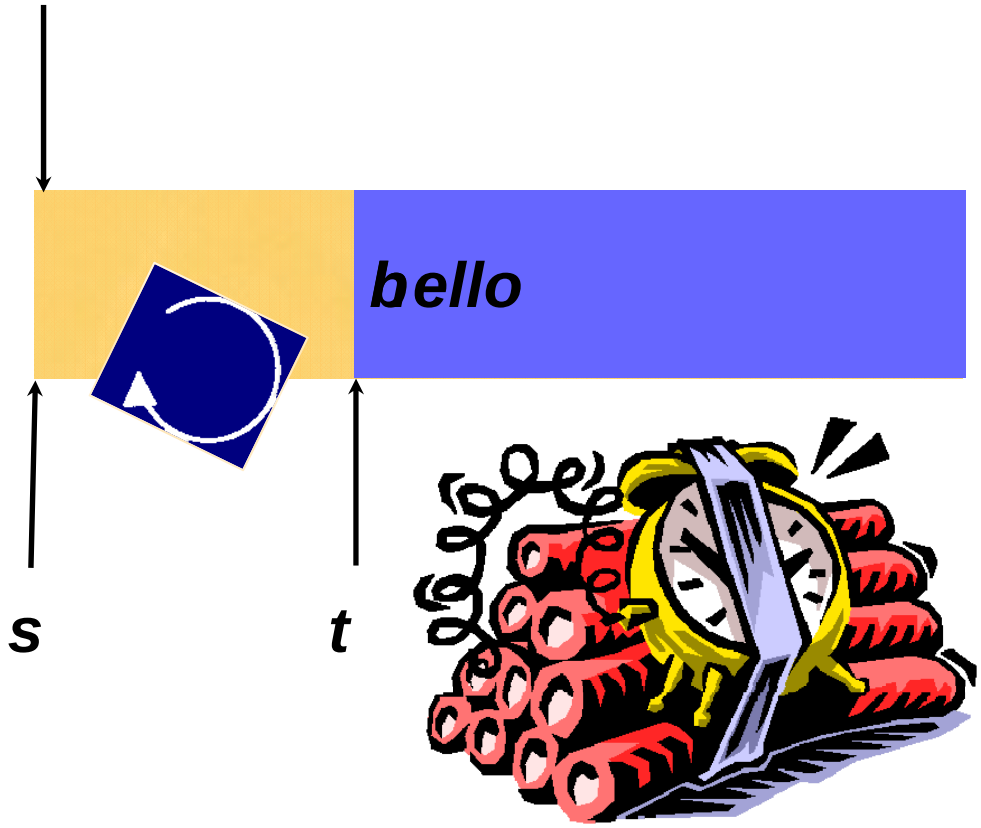
free(s)

t = malloc(20)

strcpy(t, “hello”)

s[10] = ‘b’

free(s)



Heap Assertions

- **The *GNU C runtime library* provides a simple check against common errors:**

```
$ MALLOC_CHECK_=2 myprogram myargs
free() called on area that was already free'd()
Aborted (core dumped)
$ _
```

Heap Assertions

s = malloc(30)

free(s)

free(s)

free() called on area that was already free'd()
Aborted (core dumped)

Array Assertions

- **The *Electric Fence* library checks for array overflows:**

```
$ gcc -g -o sample-with-efence sample.c -lefence
$ ./sample-with-efence 11 14
Electric Fence 2.1
Segmentation fault (core dumped)
$ _
```

Array Assertions

s = malloc(30)

s[30] = 'x'

Segmentation fault (core dumped)

Memory Assertions

- The *Valgrind* tool checks *all* memory accesses:

```
$ valgrind sample 11 14
Invalid read of size 4
  at 0x804851F: shell_sort (sample.c:18)
  by 0x8048646: main (sample.c:35)
  by 0x40220A50: __libc_start_main (in /lib/libc.so)
  by 0x80483D0: (within /home/zeller/sample)
```

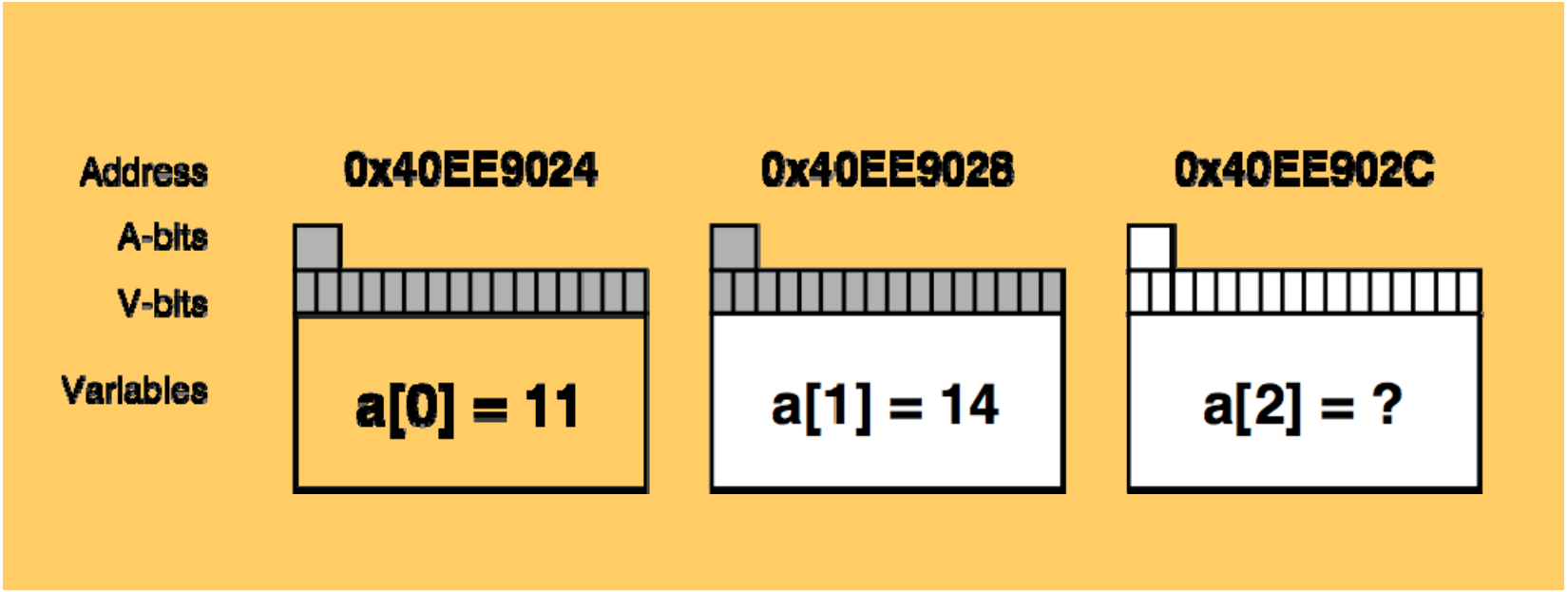
- Valgrind works as an *interpreter* for x86 code

Valgrind Checks

- Use of uninitialized memory
- Accessing free'd memory
- Accessing memory beyond malloc'd block
- Accessing inappropriate stack areas
- Memory leaks: allocated area is not free'd
- Passing uninitialized memory to system calls

Shadow Memory

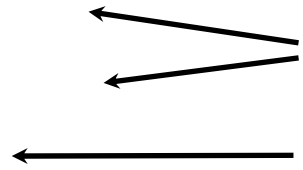
- V-bit set = corresponding bit is initialized
- A-bit set = corresponding byte is accessible



V-Bits

- When a bit is first written, its V-bit is set
- Simple read accesses to uninitialized memory do not result in warnings:

```
struct S { int x; char c; };
struct S s1, s2;
s1.x = 42;
s1.c = 'z';
s2 = s1;
```



5 bytes initialized

8 bytes copied
(no warning)

V-Bits Warnings

- **Reading uninitialized data causes a warning if**
  - **a value is used to generate an *address***
  - **a *control flow decision* is to be made**
  - **a value is passed to a *system call***

A-Bits

- When the program starts, all global data is marked “accessible” (= A-bits are set)
- malloc() sets A-bits for the area returned; free() clears them
- Local variables are “accessible” on entry and “non-accessible” on exit
- Accessing “non-accessible” data ⇒ error

Overhead

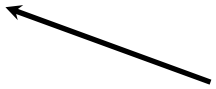
Tool	GNU C Library	Electric Fence	Valgrind
Space	2 bytes/ malloc	1 page/malloc	100 %
Time	negligible	negligible	2500 %

Preventing Misuse

- **CYCLONE is a C dialect which prevents common pitfalls of C**
- **Most important feature: *special pointers***

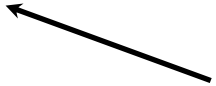
Non-NULL Pointers

int getc (FILE @fp);



fp may not be NULL

extern FILE *fp;
char c = getc(fp);

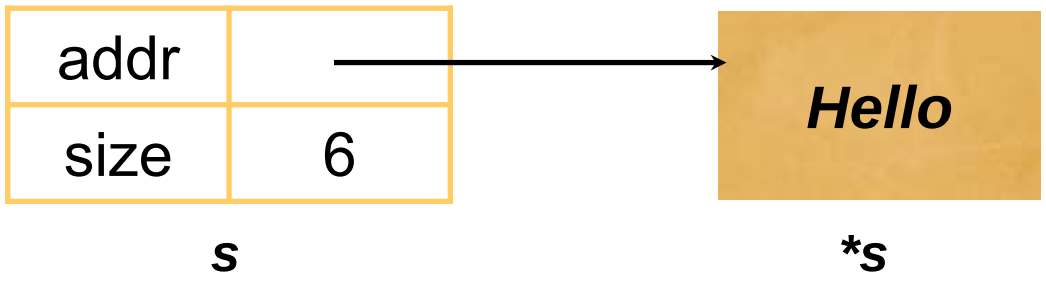


warning: NULL check inserted

Fat Pointers

- **A fat pointer holds address *and* size**
- **All accesses via a fat pointer are automatically bounds-checked**

int strlen(const char? s)



CYCLONE Restrictions

- NULL checks are inserted
- Pointer arithmetic is restricted
- Pointers must be initialized before use
- Dangling pointers are prevented through region analysis and limitations on free()
- Only “safe” casts and unions are allowed

Production Code

- **Should products ship with active assertions?**

Things to Check

Critical results. **If lives, health, or money depend on a result, it had better be checked.**

External conditions. **Any conditions which are not within our control must be checked for integrity.**

Points to Consider

- **The more active assertions, the greater the chance to catch infections.**
- **The sooner a program fails, the easier it is to track the defect.**
- **Defects that escape into the field are the hardest to track.**

More to Consider

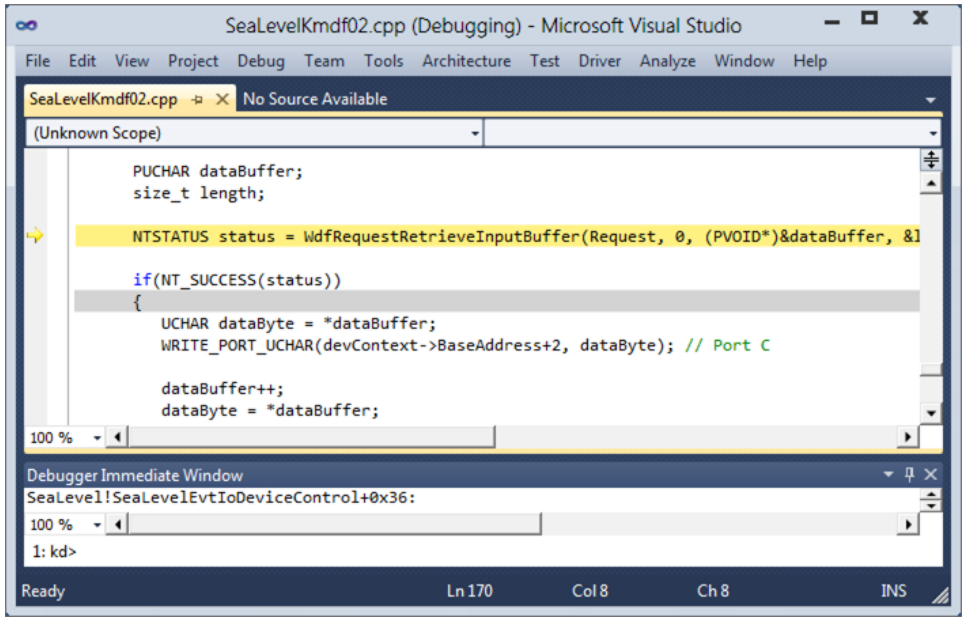
- **By default, failing assertions are not user-friendy.**
  - → ***Handle assertions in a user-friendly way***
- **Assertions impact performance.**
  - → ***First measure; then turn off only the most time-consuming assertions***

Concepts

- To check memory integrity, use specialized tools to detect errors at run time
- Apply such tools before any other method
- To fully prevent memory errors, use another language (or dialect, e.g. Cyclone)
- Turning assertions off seldom justifies the risk of erroneous computation

Debuggers

- Let you “take over” a program and run it under more control than usual
- Command line (gdb) and visual debuggers are both popular and widely used



GDB

- There are many other debuggers out there
- Won't spend a lot of time on GDB specifics
- Major features (common to many debuggers or becoming more common)
  - Single step through a program line at a time
    - GDB: **step** and **next**
      - **next** skips over functions called in a line
  - Inspect memory locations/values
  - Set breakpoints – places to stop execution
    - Can be conditional (break at line XX if y > z)
  - Set watchpoints – events to watch for in execution
  - Change values in memory
  - GDB can also "run a program backwards" a little bit now!

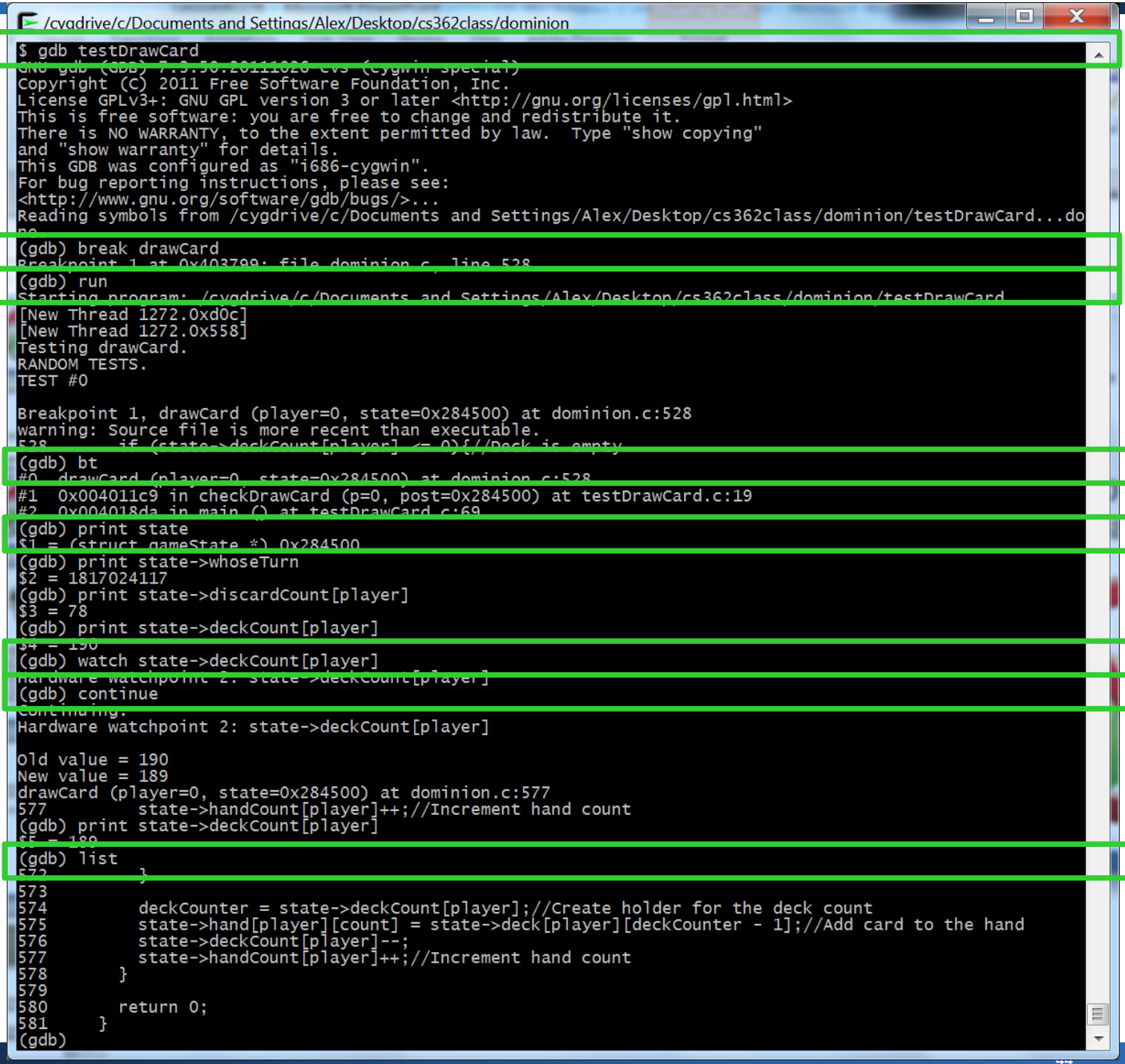
GDB

- Si
  de
  - ault
  - ther

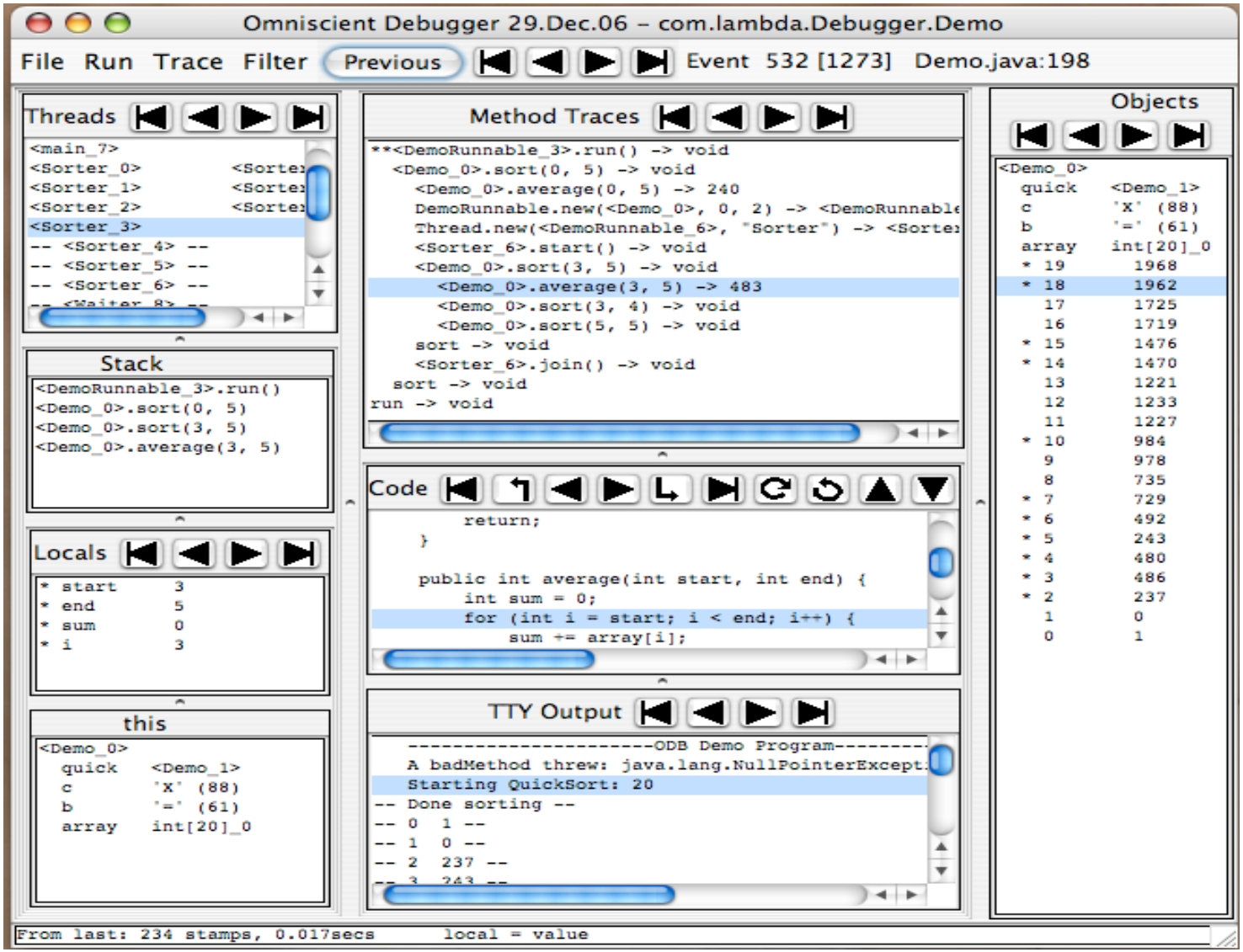
/cygdrive/c/Documents and Settings/Alex/Desktop/cs362class/dominion

```
Alex@groce /cygdrive/c/Documents and Settings/Alex/Desktop/cs362class/dominion
$ ./badTestDrawCard.exe
Testing drawCard.
RANDOM TESTS.
Segmentation fault (core dumped)

Alex@groce /cygdrive/c/Documents and Settings/Alex/Desktop/cs362class/dominion
$ gdb ./badTestDrawCard.exe
GNU gdb (GDB) 7.3.50.20111026-cvs (cygwin-special)
Copyright (C) 2011 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "i686-cygwin".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>...
Reading symbols from /cygdrive/c/Documents and Settings/Alex/Desktop/cs362class/
dominion/badTestDrawCard.exe...done.
(gdb) run
Starting program: /cygdrive/c/Documents and Settings/Alex/Desktop/cs362class/dom
inion/badTestDrawCard.exe
[New Thread 1980.0x46c]
[New Thread 1980.0xf2c]
Testing drawCard.
RANDOM TESTS.

Program received signal SIGSEGV, Segmentation fault.
0x00403348 in drawCard (player=145, state=0x2844f0) at dominion.c:534
534             state->deck[player][i] = state->discard[player][i];
(gdb) bt
#0  0x00403348 in drawCard (player=145, state=0x2844f0) at dominion.c:534
#1  0x004011a4 in checkDrawCard (p=145, post=0x2844f0) at badTestDrawCard.c:14
#2  0x0040141d in main () at badTestDrawCard.c:38
(gdb) print i
$1 = 0
(gdb) print player
$2 = 145
(gdb)
```

GDB


/cygdrive/c/Documents and Settings/Alex/Desktop/cs362class/dominion
$ gdb testDrawCard
Copyright (C) 2011 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law. Type "show copying"
and "show warranty" for details.
This GDB was configured as "i686-cygwin".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>...
Reading symbols from /cygdrive/c/Documents and Settings/Alex/Desktop/cs362class/dominion/testDrawCard...do
(gdb) break drawCard
(gdb) run
[New Thread 1272.0xd0c]
[New Thread 1272.0x558]
Testing drawCard.
RANDOM TESTS.
TEST #0
Breakpoint 1, drawCard (player=0, state=0x284500) at dominion.c:528
warning: Source file is more recent than executable.
(gdb) bt
#1 0x004011c9 in checkDrawCard (p=0, post=0x284500) at testDrawCard.c:19
(gdb) print state
(gdb) print state->whoseTurn
$2 = 1817024117
(gdb) print state->discardCount[player]
$3 = 78
(gdb) print state->deckCount[player]
(gdb) watch state->deckCount[player]
(gdb) continue
Hardware watchpoint 2: state->deckCount[player]
Old value = 190
New value = 189
drawCard (player=0, state=0x284500) at dominion.c:577
577   state->handCount[player]++;//Increment hand count
(gdb) print state->deckCount[player]
(gdb) list
573
574   deckCounter = state->deckCount[player];//Create holder for the deck count
575   state->hand[player][count] = state->deck[player][deckCounter - 1];//Add card to the hand
576   state->deckCount[player]--;
577   state->handCount[player]++;//Increment hand count
578   }
579
580   return 0;
581   }
(gdb)

Omniscient Debugger



How does it work?

- ODB records a *trace* of the entire execution history
- Slows down programs by a factor of 10
- Records about 100 MB/s
- Now available in commercial tools:
  - RETROVUE
  - CODEGUIDE

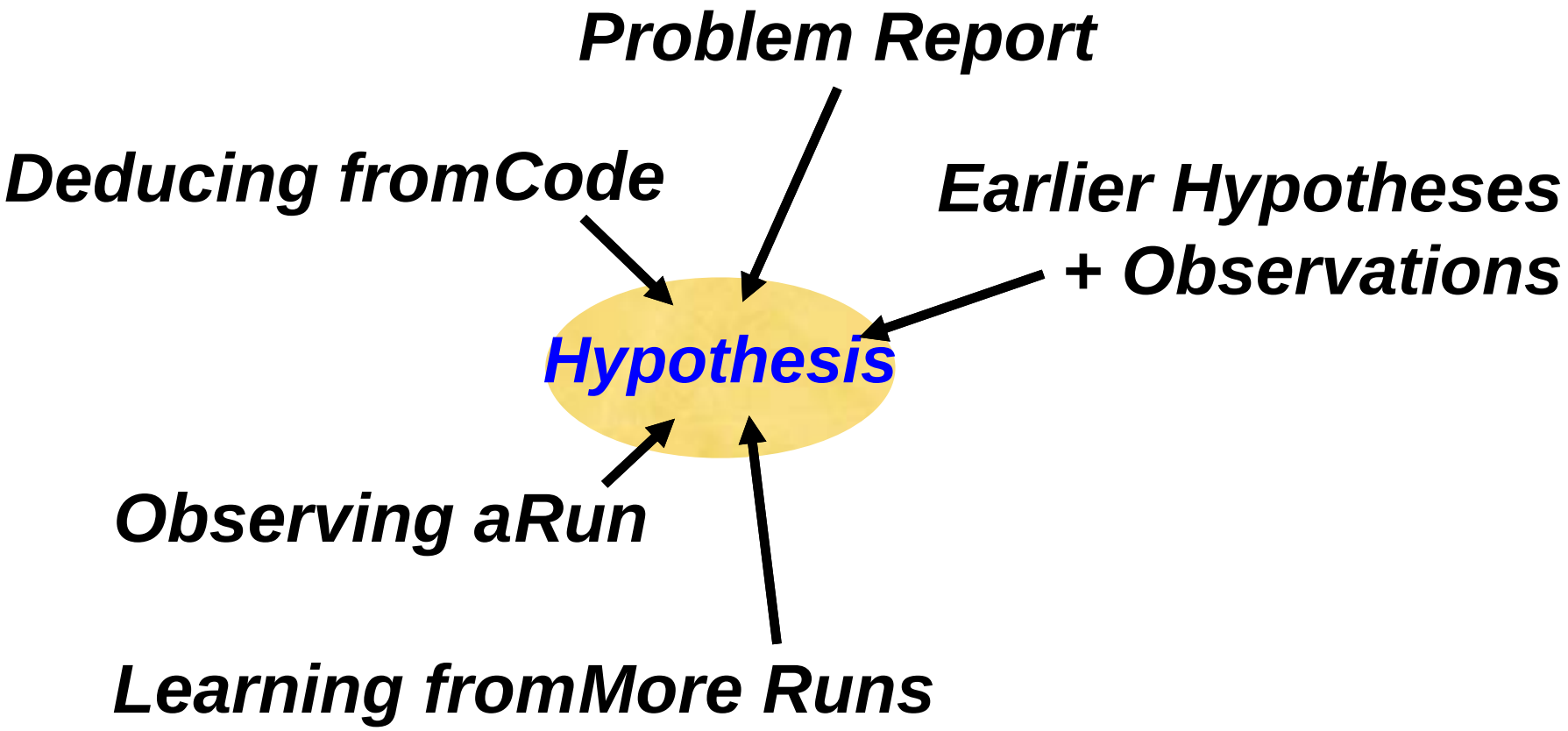
Exploring the Past

A typical debugging session looks like this:

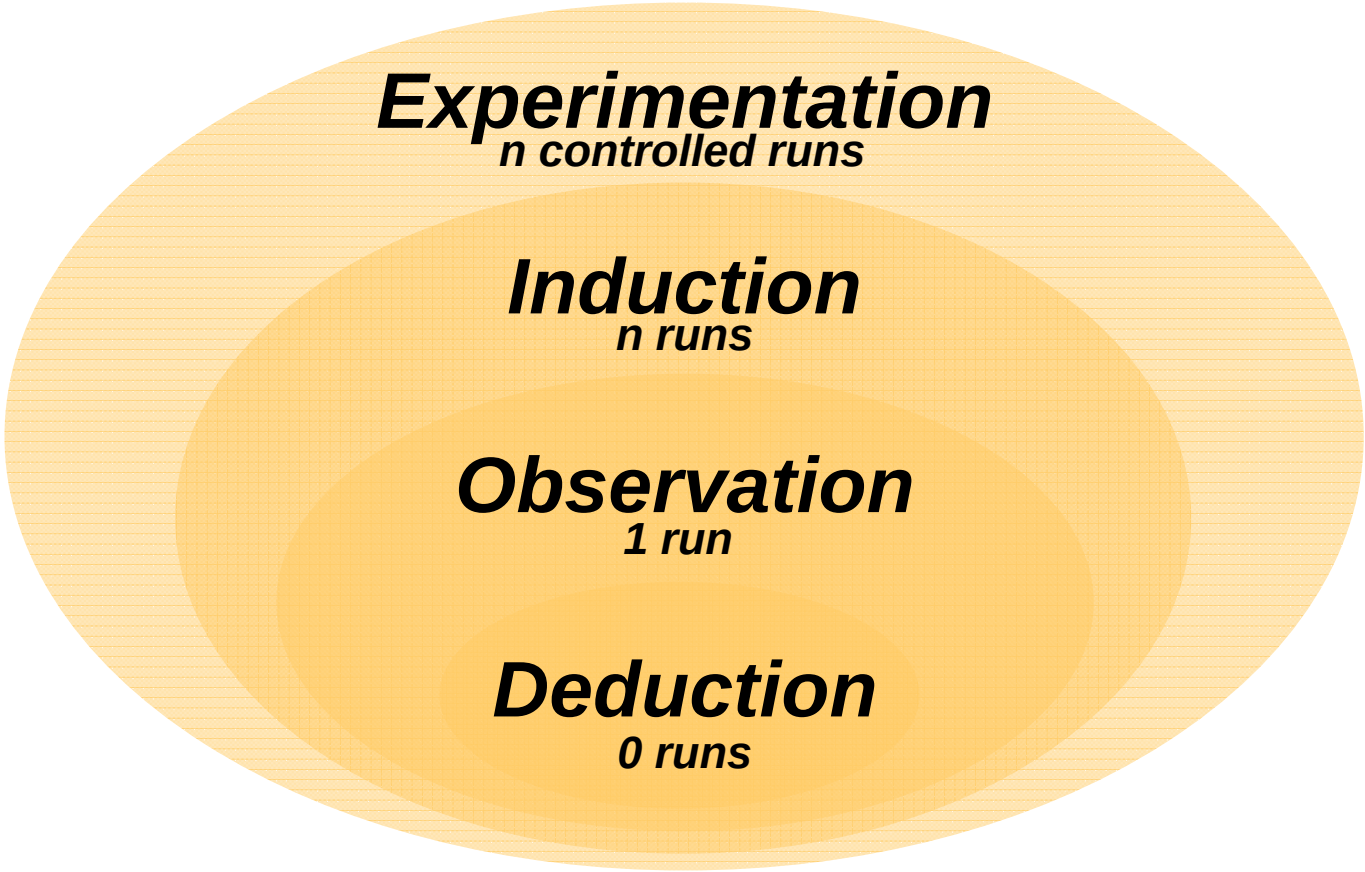
1. **Set a breakpoint**
2. **Start program, reaching breakpoint**
3. **Step, Step, Step, …**
4. **Oops! I've gone too far!**

Deducing Errors

Obtaining a Hypothesis



Reasoning about Runs



Reasoning about Runs

Effects of Statements

Write. A statement can change the program state (i.e. write to a variable)

Control. A statement may determine which statement is executed next (other than unconditional transfer)

Affected Statements

Read. A statement can read the program state (i.e. from a variable)

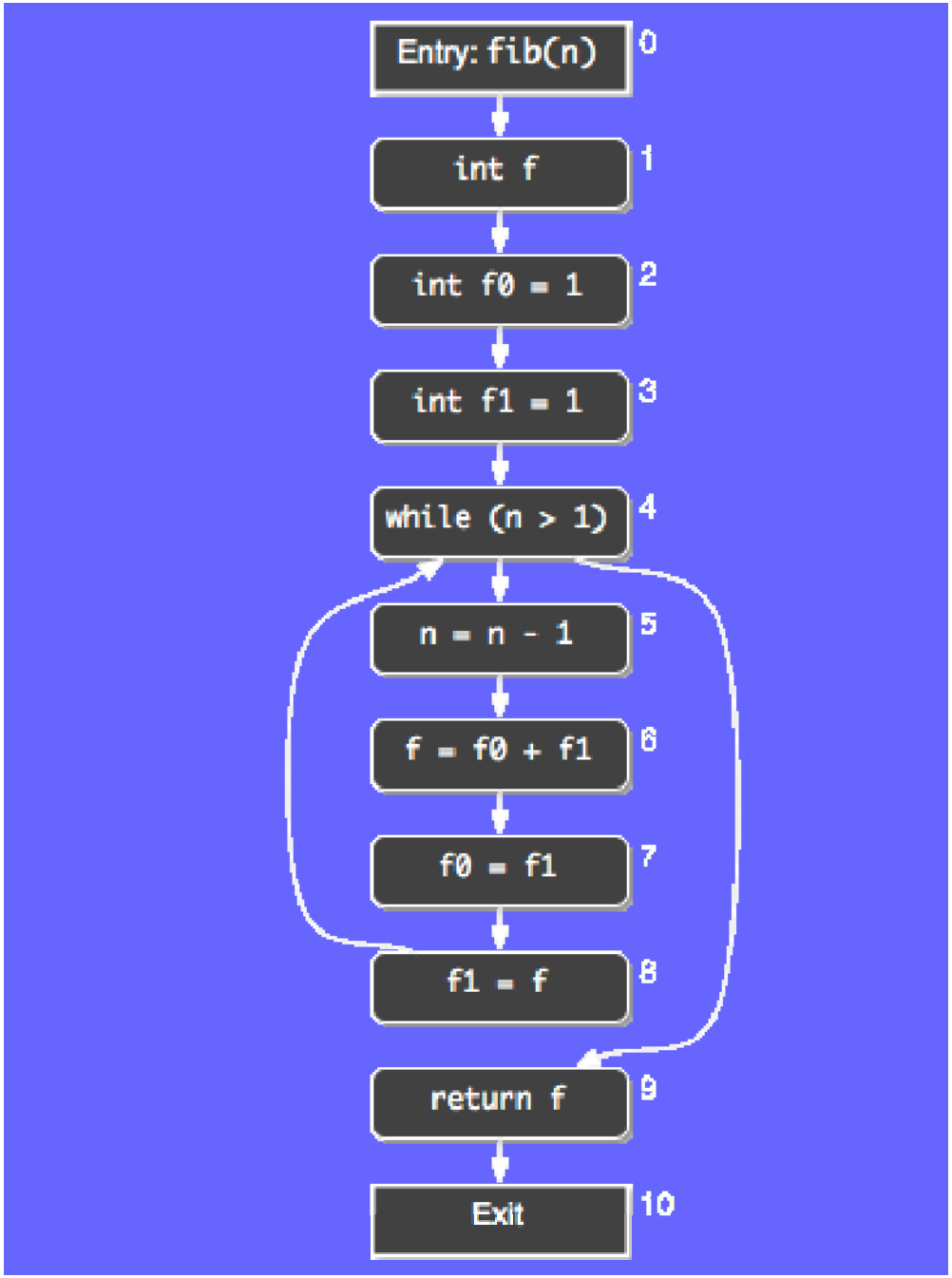
Execution. To have any effect, a statement must be executed.

Effects in fibo.c

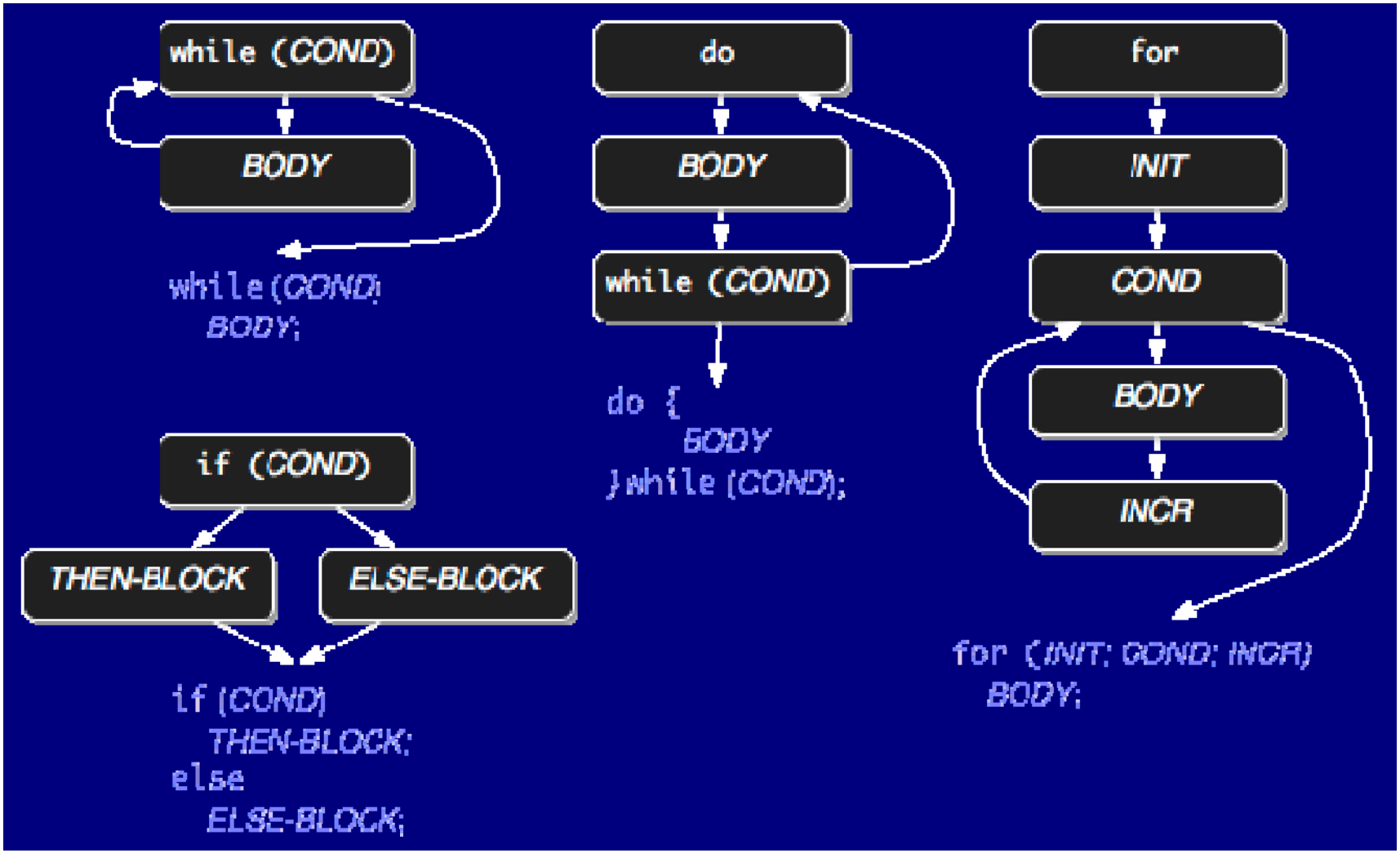
	Statement	Reads	Writes	Controls
0	fib(n)		n	1-10
1	int f		f	
2	f0 = 1		f0	
3	f1 = 1		f1	
4	while (n > 1)	n		5-8
5	n = n - 1	n	n	
6	f = f0 + f1	f0, f1	f	
7	f0 = f1	f1	f0	
8	f1 = f	f	f1	
9	return f	f	<ret>	

Control Flow

```
int fib(int n)
{
   int f, f0 = 1, f1 = 1;

   while (n > 1) {
            n = n - 1;
            f = f0 + f1;
            f0 = f1;
            f1 = f;
   }

   return f;
}
```



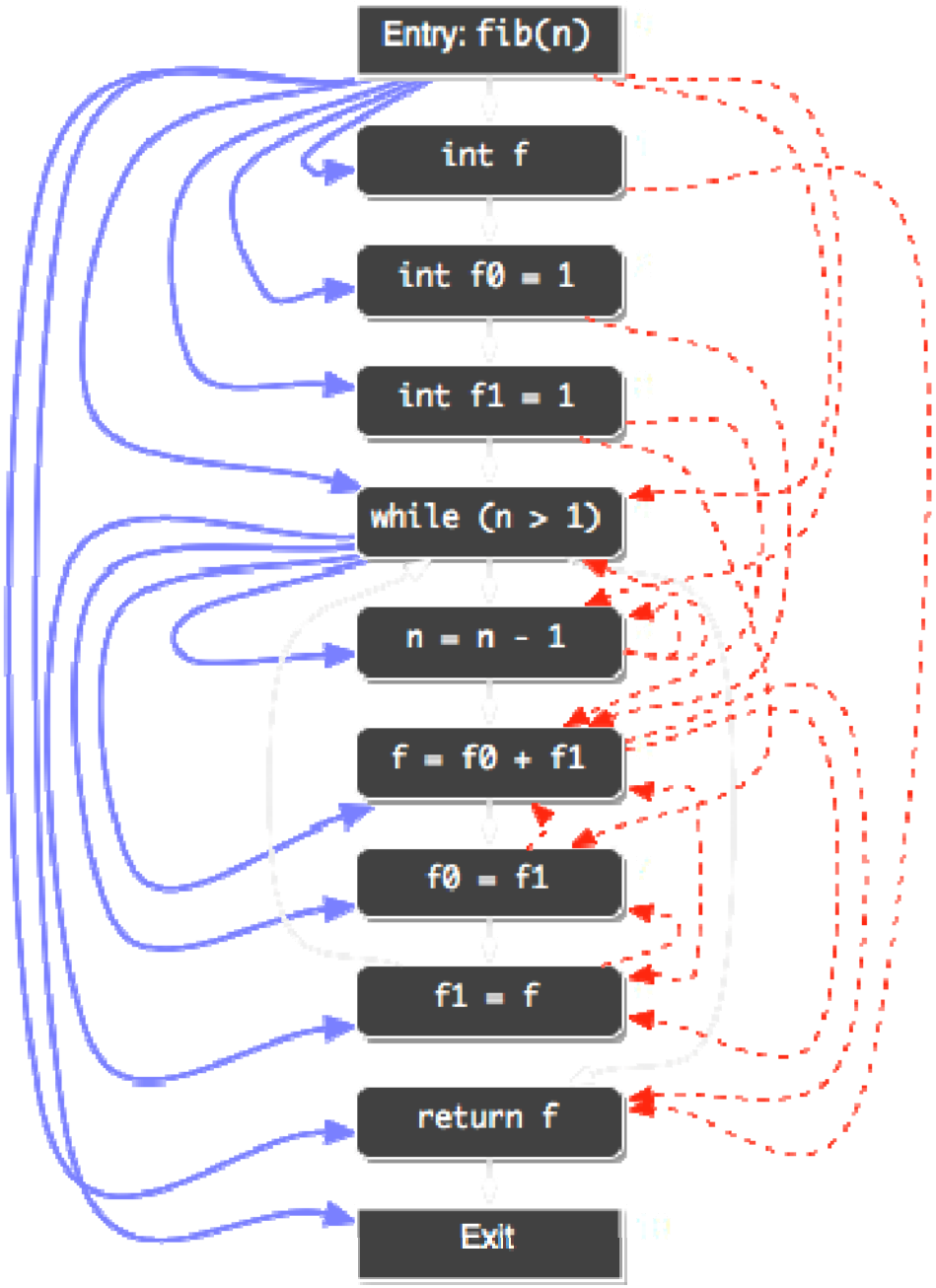
Control Flow Patterns



Dependences

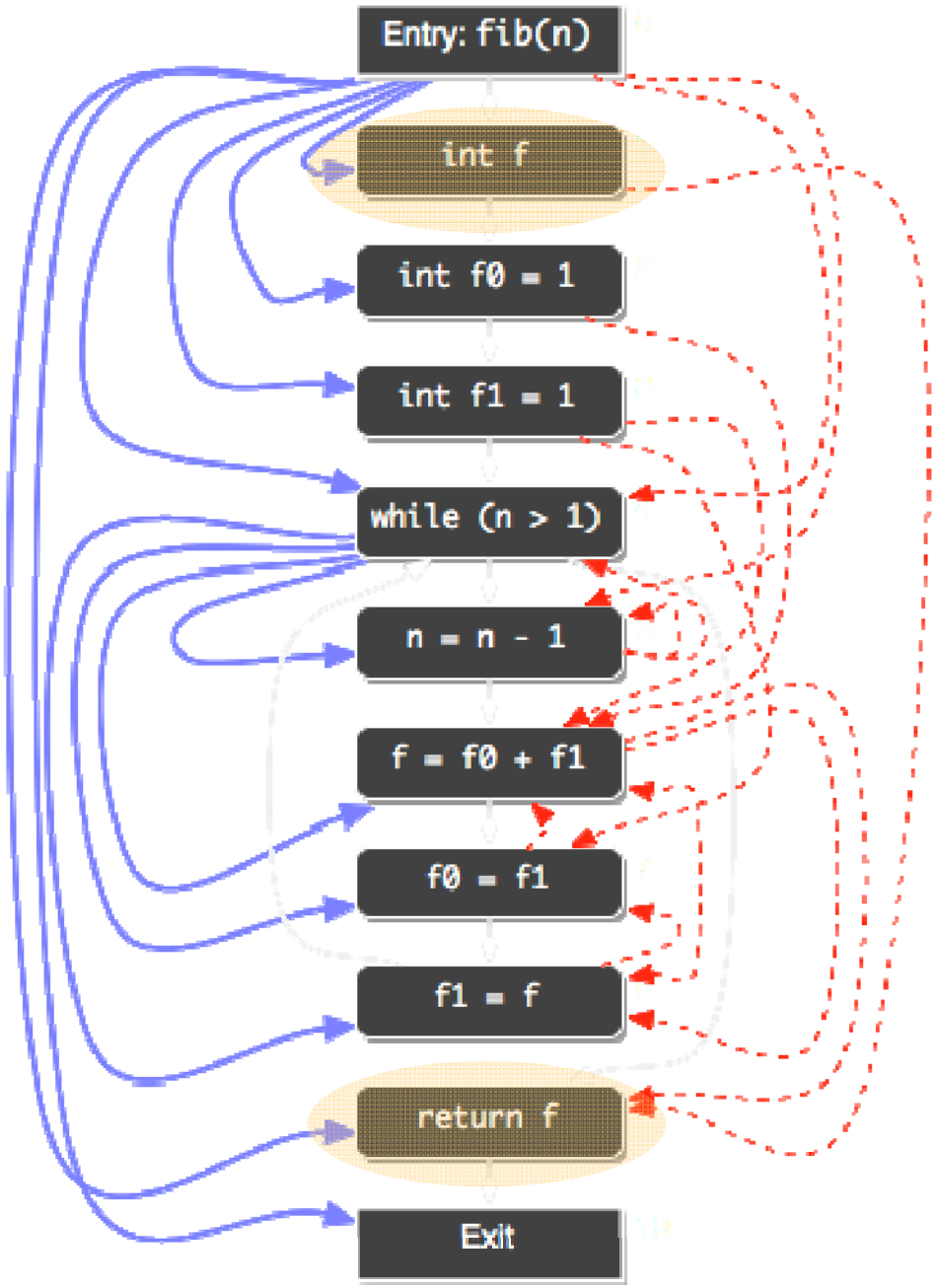
Data dependency:

Control dependency:

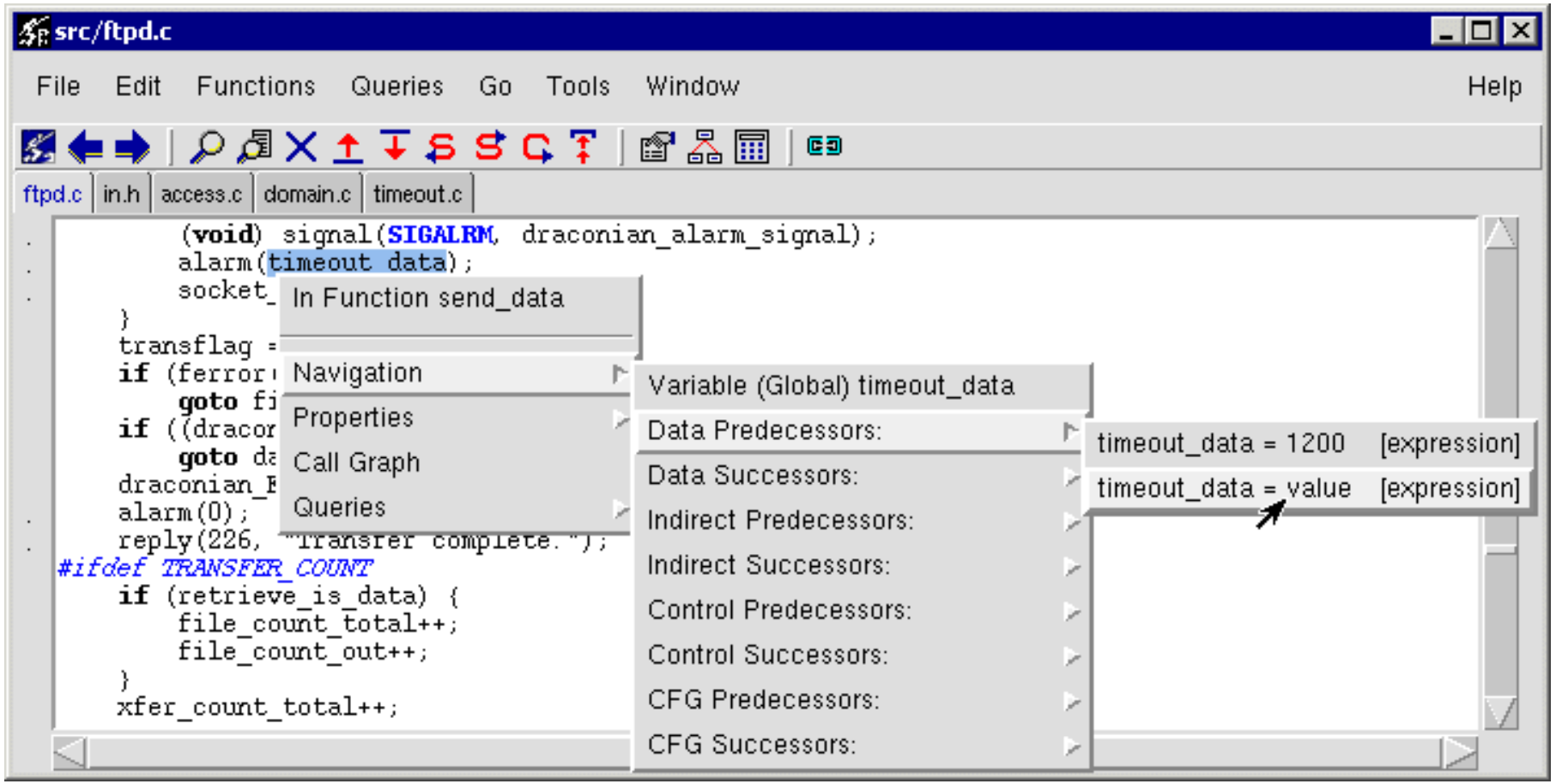


Dependences

- Following the dependences, we can answer questions like
- Where does this value go to?
- Where does this value come from?



Navigating along Dependences



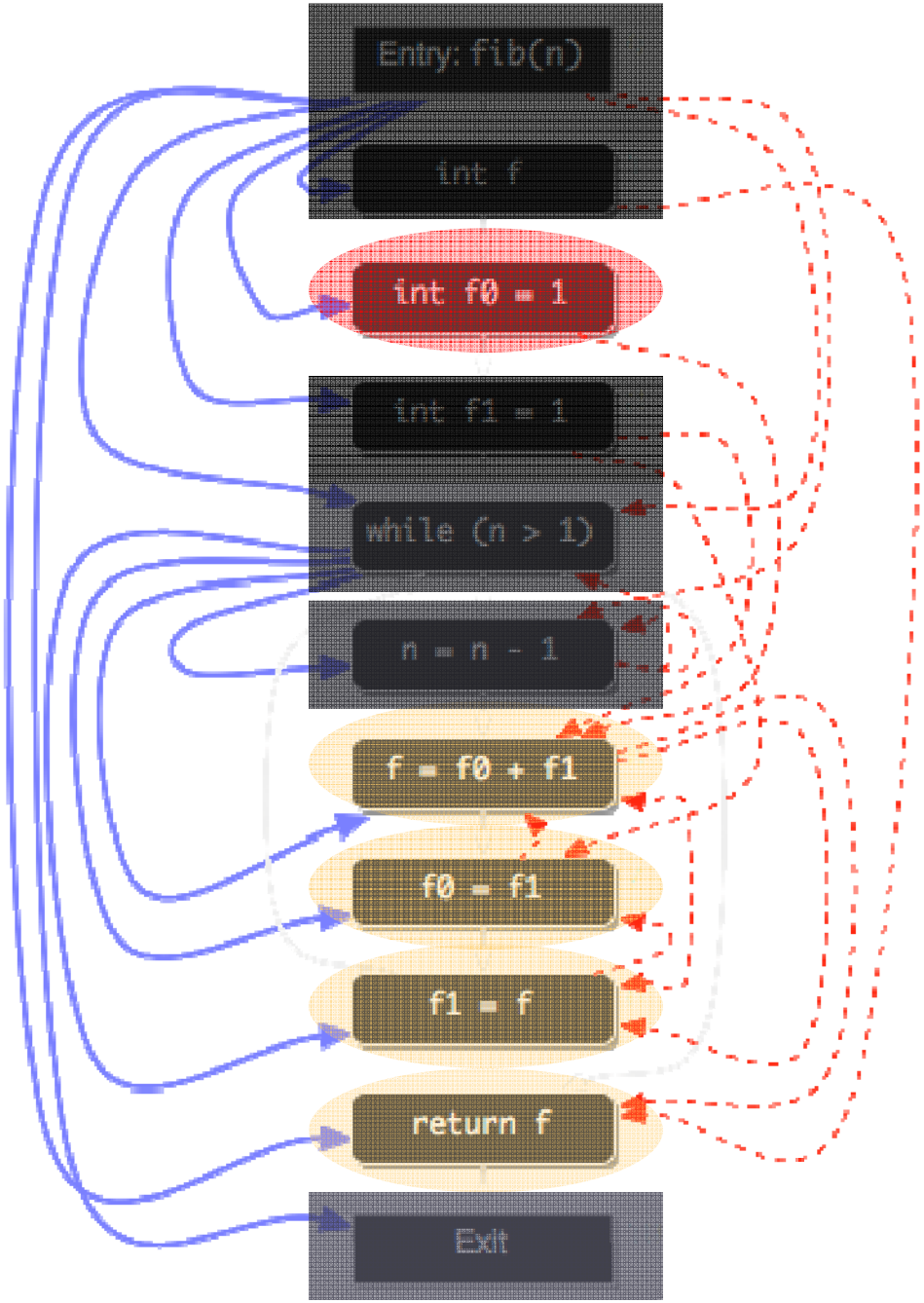
Program Slicing

- A *slice* is a subset of the program
- Allows programmers to *focus on what's relevant* with respect to some statement S:
  - All statements influenced by S
  - All statements that influence S

Forward Slice

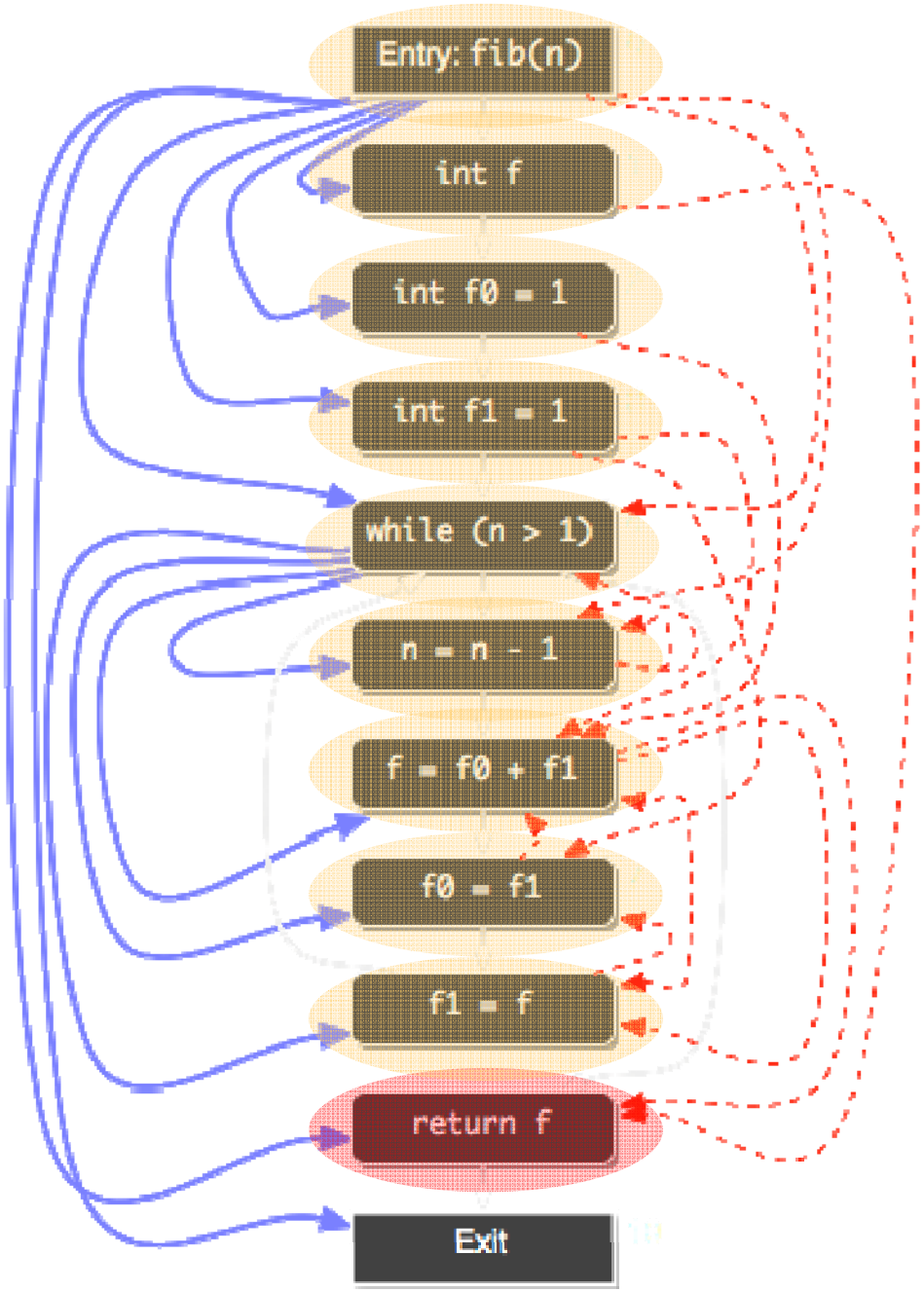
- Given a statement A, the forward slice contains all statements whose read variables or execution could be influenced by A
- Formally:

$$S^F(A) = \{B \mid A \rightarrow^F B\}$$



Backward Slice

- Given a statement B, the backward slice contains all statements that could influence the read variables or execution of B
- Formally:



Two Slices

```
int main() {
  int a, b, sum, mul;
  sum = 0;
  mul = 1;
  a = read();
  b = read();
  while (a <= b) {
    sum = sum + a;
    mul = mul * a;
    a = a + 1;
  }
  write(sum);
  write(mul);
}
```

- Slice Operations:
- Backbones
- Dices
- Chops

← ***Backward slice of sum***

← ***Backward slice of mul***

Backbone

```
a = read();
b = read();
while (a <= b) {
```

```
  a = a + 1;
```

- Contains only those statement that occur in both slices
- Useful for focusing on common behavior

Two Slices

```
int main() {
  int a, b, sum, mul;
  sum = 0;
  mul = 1;
  a = read();
  b = read();
  while (a <= b) {
    sum = sum + a;
    mul = mul * a;
    a = a + 1;
  }
  write(sum);
  write(mul);
}
```

- Slice Operations:
- Backbones
- Dices
- Chops

Backward slice of sum

Backward slice of mul

Dice

```
sum = 0;
```

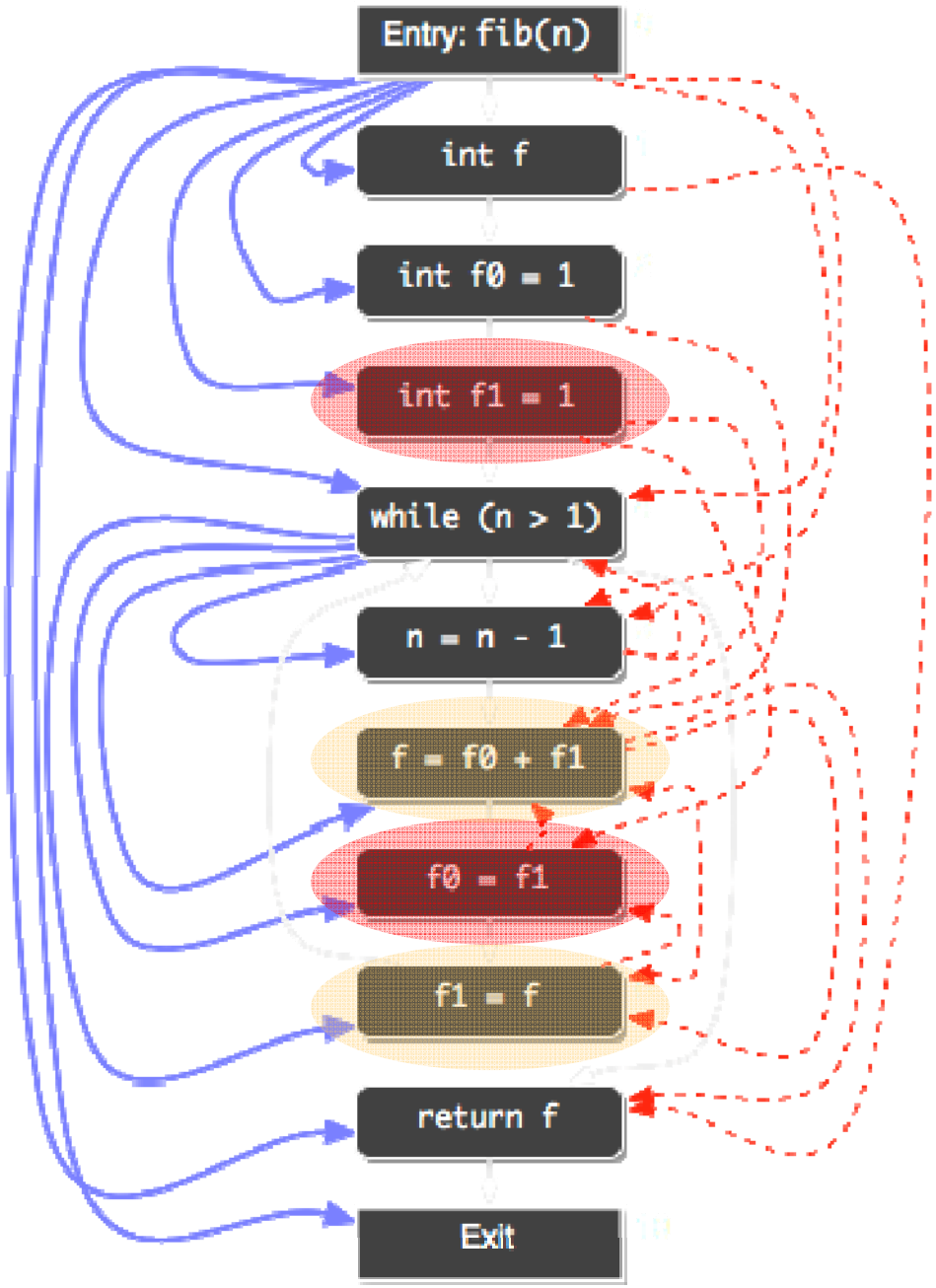
```
 sum = sum + a;
```

```
write(sum);
```

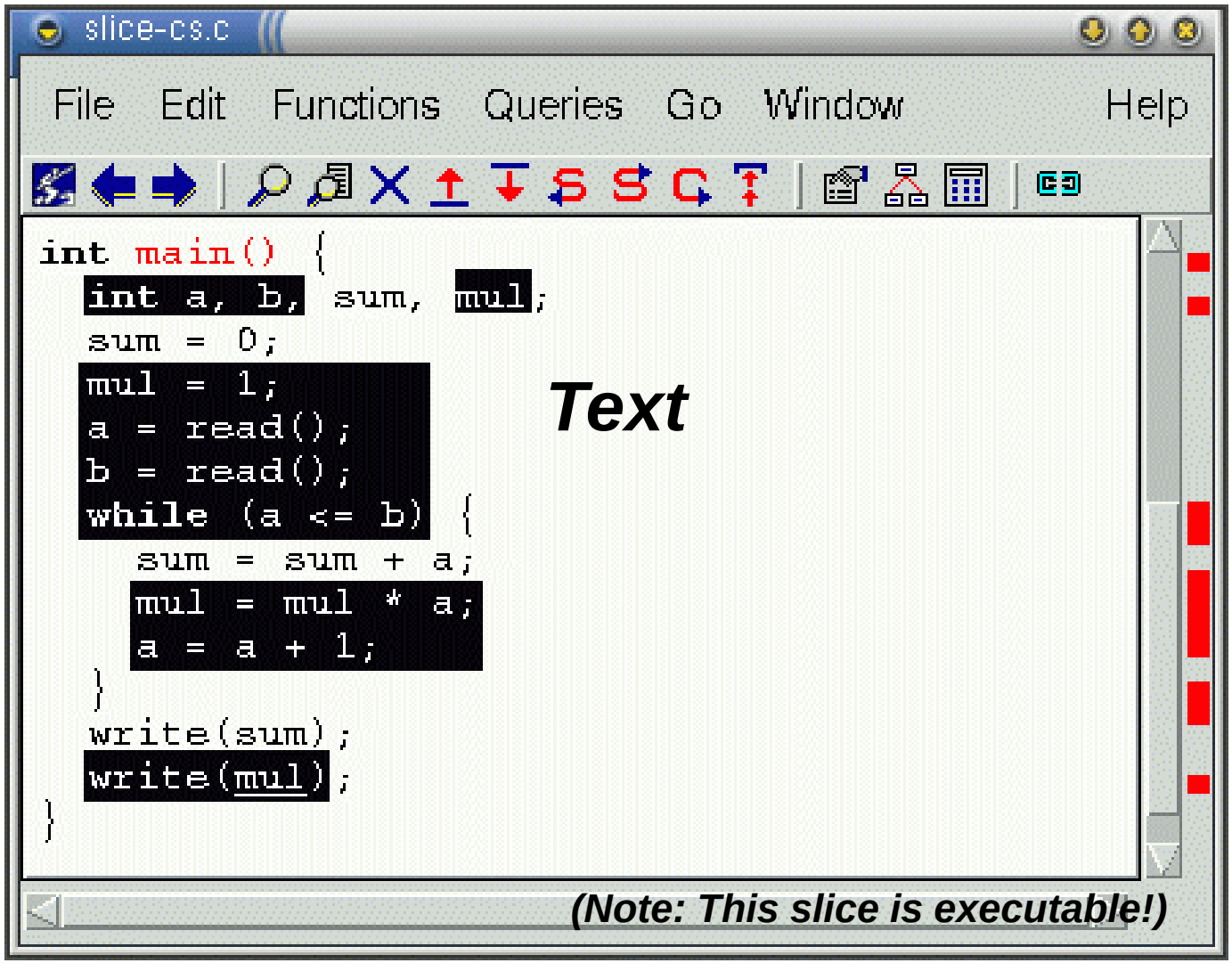
- Contains only the difference between two slices
- Useful for focusing on differing behavior

Chop

- Intersection between a forward and a backward slice
- Useful for determining influence paths within the program



Leveraging Slices



Concepts

- To reason about programs, use
  - deduction (0 runs)
  - observation (1 run)
  - induction (multiple runs)
  - experimentation (controlled runs)
- To slice a program, follow dependences from a statement S to find all statements that
  - could be influenced by S (forward slice)
  - could influence S (backward slice)

Concepts

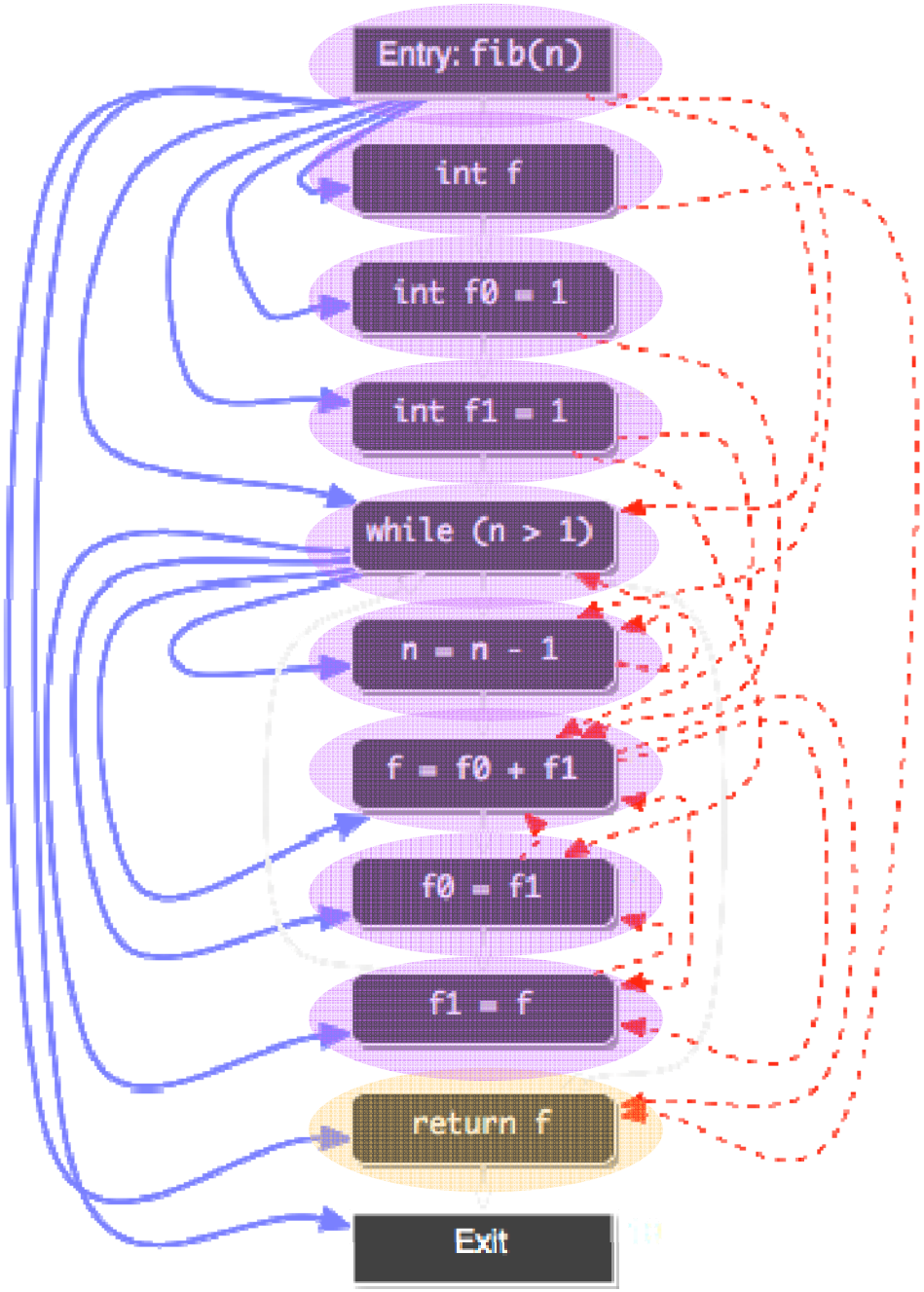
- Using deduction alone includes a number of risks, including code mismatch, abstracting away, and imprecision.
- Any deduction is limited by the halting problem and must thus resort to conservative approximation.
- For debugging, deduction is best combined with actual observation.

Dynamic Slicing

- **Static slices apply to *all* program runs:**
  - **General + reusable, but imprecise**
- **A *dynamic slice* applies to *a single run:***
  - **Specific and precise**

Static Slicing

- Given a statement B, the backward slice contains all statements that could influence the read variables or execution of B
- Formally:



```
1 n = read();
2 a = read();
3 x = 1;
4 b = a + x;
5 a = a + 1;
6 i = 1;
7 s = 0;
8 while (i <= n) {
9     if (b > 0)
10        if (a > 1)
11            x = 2;
12    s = s + x;
13    i = i + 1;
14 }
15 write(s);
```

```
1 n = read(); // n = 2
2 a = read(); // a = 0
3 x = 1;
4 b = a + x;
5 a = a + 1;
6 i = 1;
7 s = 0;
8 while (i <= n) {
9     if (b > 0)
10        if (a > 1)
11            x = 2;
12    s = s + x;
13    i = i + 1;
14 }
15 write(s);
```

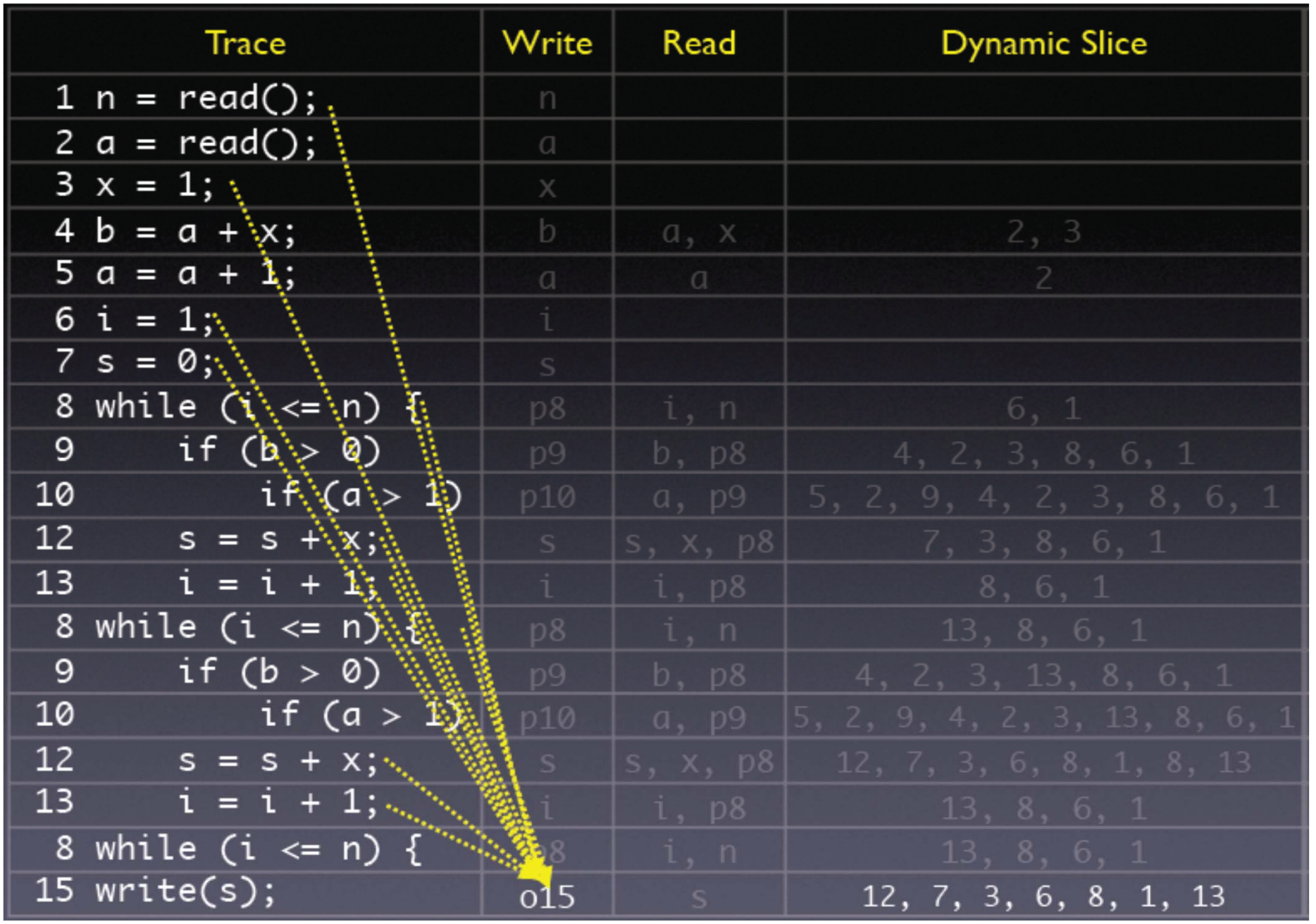
Static slice for (s, 15) **Dynamic slice for (s, 15)**

```
 1 n = read();
 2 a = read();
 3 x = 1;
 4 b = a + x;
 5 a = a + 1;
 6 i = 1;
 7 s = 0;
 8 while (i <= n) {
 9     if (b > 0)
10         if (a > 1)
11             x = 2;
12     s = s + x;
13     i = i + 1;
14 }
15 write(s);
```

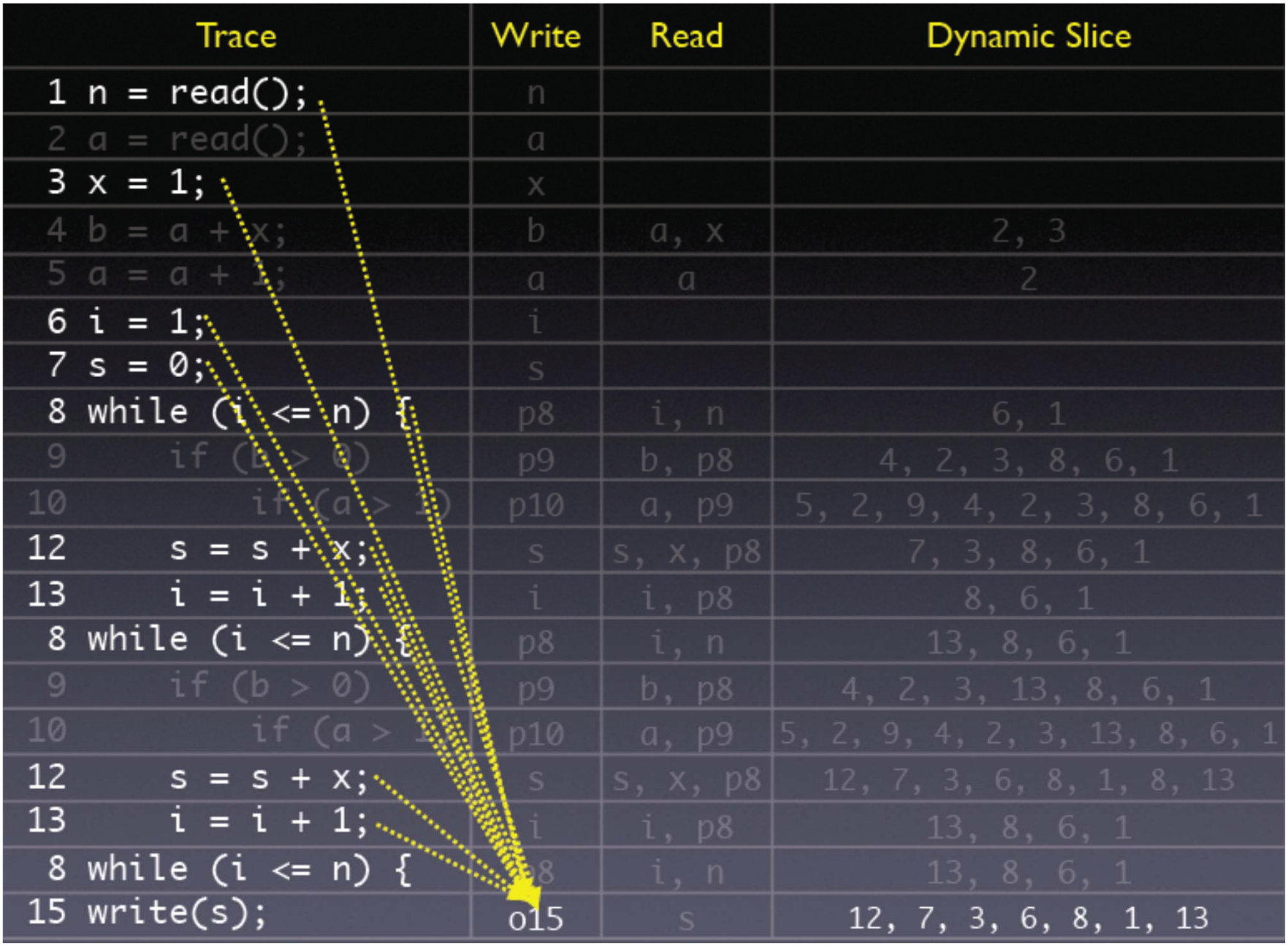
1. Obtain a *trace* of the execution
2. Get the variables that are read and written
3. Assign an empty slice to each written variable
4. Compute the slices from start to end:

$$SynSlice(n) = \bigcup_{i} (SynSlice(n_i) \cup Line(n_i))$$

Trace	Write	Read	Dynamic Slice
1 n = read();	n	$DynSlice(w) = \bigcup_i (DynSlice(r_i) \cup \{line(r_i)\})$	
2 a = read();	a		
3 x = 1;	x		
4 b = a + x;	b	a, x	2, 3
5 a = a + 1;	a	a	2
6 i = 1;	i		
7 s = 0;	s		
8 while (i <= n) {	p8	i, n	6, 1
9 if (b > 0)	p9	b, p8	4, 2, 3, 8, 6, 1
10 if (a > 1)	p10	a, p9	5, 2, 9, 4, 2, 3, 8, 6, 1
12 s = s + x;	s	s, x, p8	7, 3, 8, 6, 1
13 i = i + 1;	i	i, p8	8, 6, 1
8 while (i <= n) {	p8	i, n	13, 8, 6, 1
9 if (b > 0)	p9	b, p8	4, 2, 3, 13, 8, 6, 1
10 if (a > 1)	p10	a, p9	5, 2, 9, 4, 2, 3, 13, 8, 6, 1
12 s = s + x;	s	s, x, p8	12, 7, 3, 6, 8, 1, 13
13 i = i + 1;	i	i, p8	13, 8, 6, 1
8 while (i <= n) {	p8	i, n	13, 8, 6, 1
15 write(s);	o15	s	12, 7, 3, 6, 8, 1, 13



Trace	Write	Read	Dynamic Slice
1 n = read();	n		
2 a = read();	a		
3 x = 1;	x		
4 b = a + x;	b	a, x	2, 3
5 a = a + 1;	a	a	2
6 i = 1;	i		
7 s = 0;	s		
8 while (i <= n) {	p8	i, n	6, 1
9 if (b > 0)	p9	b, p8	4, 2, 3, 8, 6, 1
10 if (a > 1)	p10	a, p9	5, 2, 9, 4, 2, 3, 8, 6, 1
12 s = s + x;	s	s, x, p8	7, 3, 8, 6, 1
13 i = i + 1;	i	i, p8	8, 6, 1
8 while (i <= n) {	p8	i, n	13, 8, 6, 1
9 if (b > 0)	p9	b, p8	4, 2, 3, 13, 8, 6, 1
10 if (a > 1)	p10	a, p9	5, 2, 9, 4, 2, 3, 13, 8, 6, 1
12 s = s + x;	s	s, x, p8	12, 7, 3, 6, 8, 1, 8, 13
13 i = i + 1;	i	i, p8	13, 8, 6, 1
8 while (i <= n) {	p8	i, n	13, 8, 6, 1
15 write(s);	o15	s	12, 7, 3, 6, 8, 1, 13



Trace	Write	Read	Dynamic Slice
1 n = read();	n		
2 a = read();	a		
3 x = 1;	x		
4 b = a + x;	b	a, x	2, 3
5 a = a + 1;	a	a	2
6 i = 1;	i		
7 s = 0;	s		
8 while (i <= n) {	p8	i, n	6, 1
9 if (b > 0)	p9	b, p8	4, 2, 3, 8, 6, 1
10 if (a > 1)	p10	a, p9	5, 2, 9, 4, 2, 3, 8, 6, 1
12 s = s + x;	s	s, x, p8	7, 3, 8, 6, 1
13 i = i + 1;	i	i, p8	8, 6, 1
8 while (i <= n) {	p8	i, n	13, 8, 6, 1
9 if (b > 0)	p9	b, p8	4, 2, 3, 13, 8, 6, 1
10 if (a > 1)	p10	a, p9	5, 2, 9, 4, 2, 3, 13, 8, 6, 1
12 s = s + x;	s	s, x, p8	12, 7, 3, 6, 8, 1, 8, 13
13 i = i + 1;	i	i, p8	13, 8, 6, 1
8 while (i <= n) {	p8	i, n	13, 8, 6, 1
15 write(s);	o15	s	12, 7, 3, 6, 8, 1, 13

```
1 n = read();
2 a = read();
3 x = 1;
4 b = a + x;
5 a = a + 1;
6 i = 1;
7 s = 0;
8 while (i <= n) {
9     if (b > 0)
10        if (a > 1)
11            x = 2;
12    s = s + x;
13    i = i + 1;
14 }
15 write(s);
```

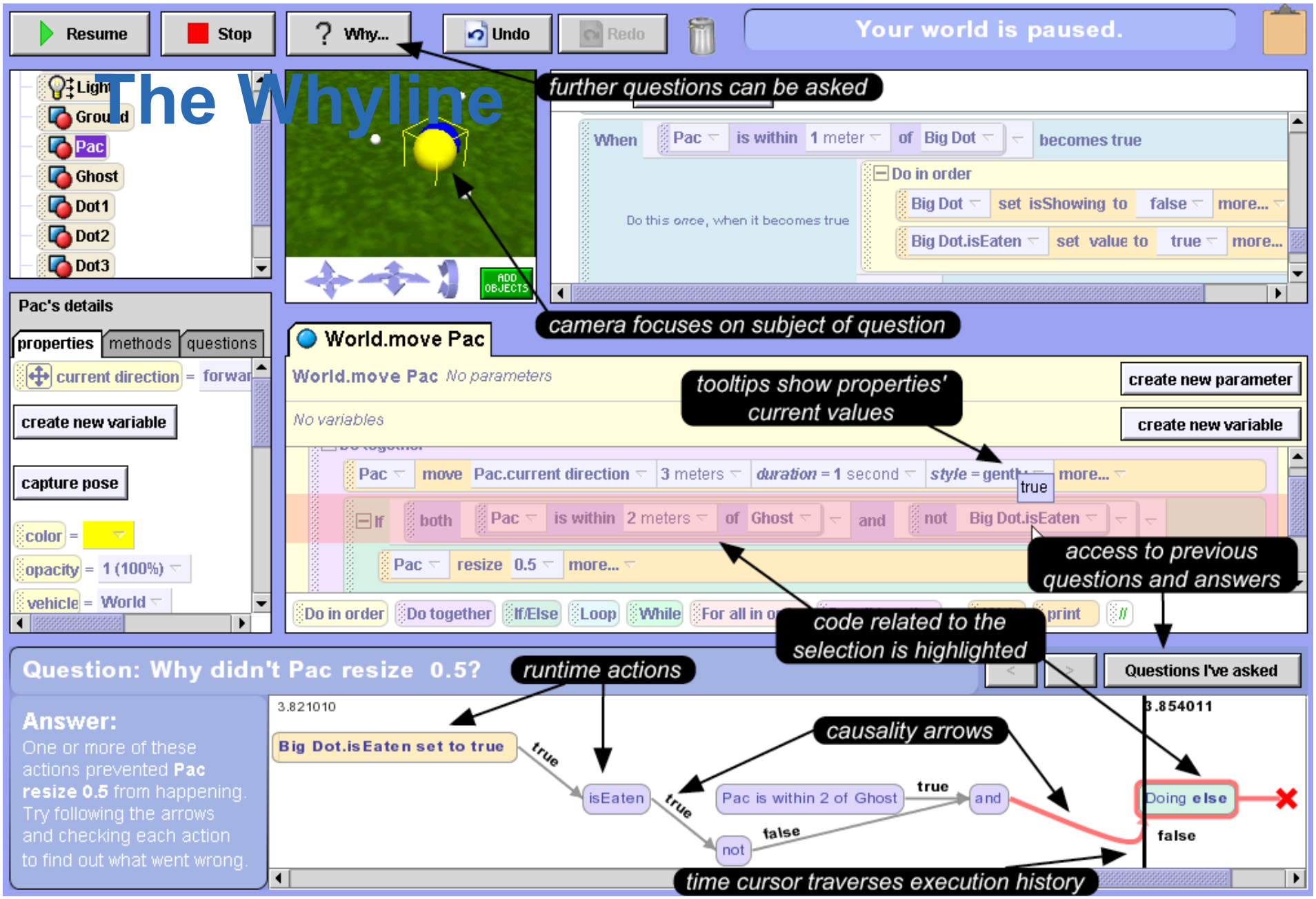
```
1 n = read(); // n = 2
2 a = read(); // a = 0
3 x = 1;
4 b = a + x;
5 a = a + 1;
6 i = 1;
7 s = 0;
8 while (i <= n) {
9     if (b > 0)
10        if (a > 1)
11            x = 2;
12    s = s + x;
13    i = i + 1;
14 }
15 write(s);
```

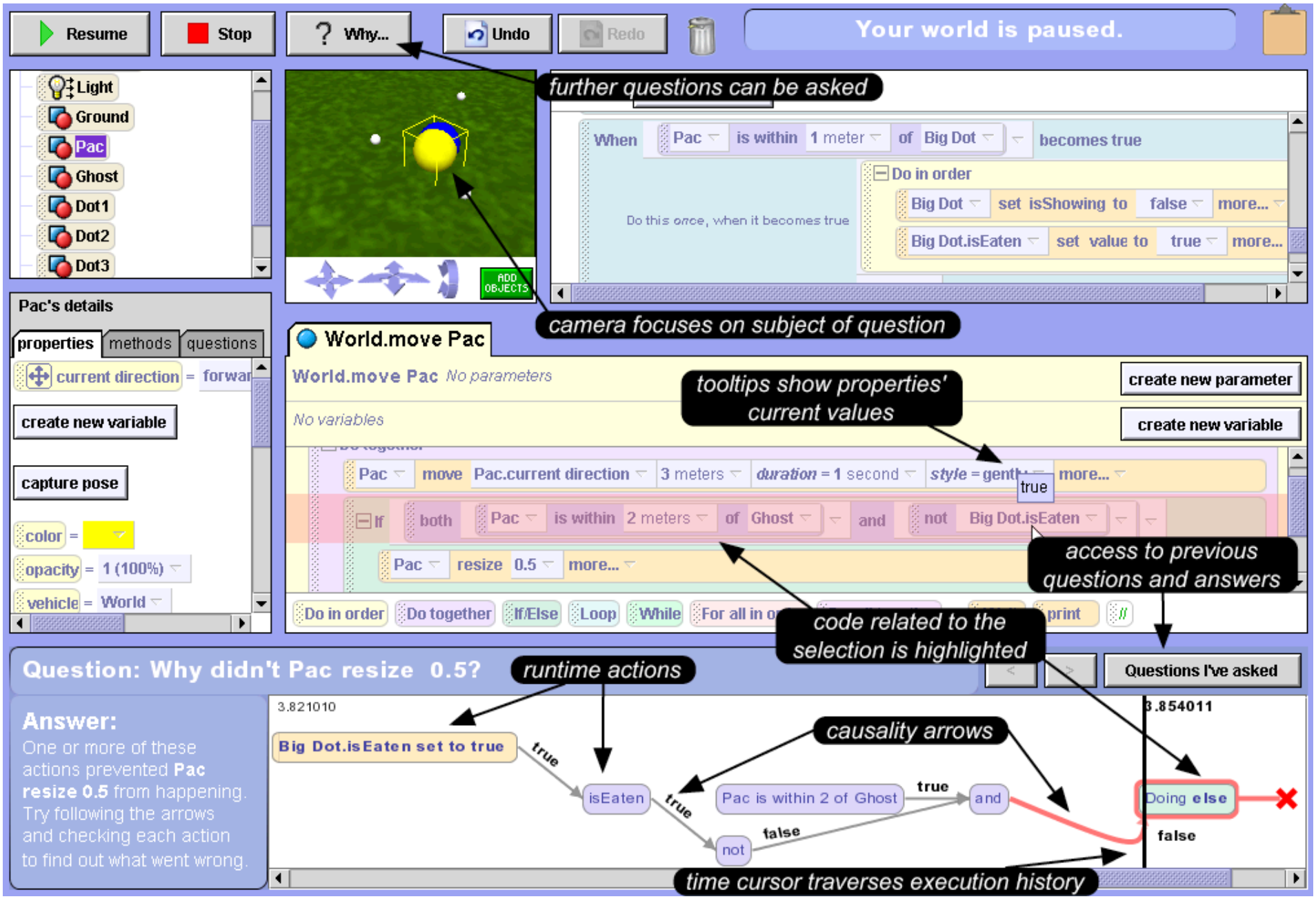
Static slice for (s, 15) **Dynamic slice for (s, 15)**

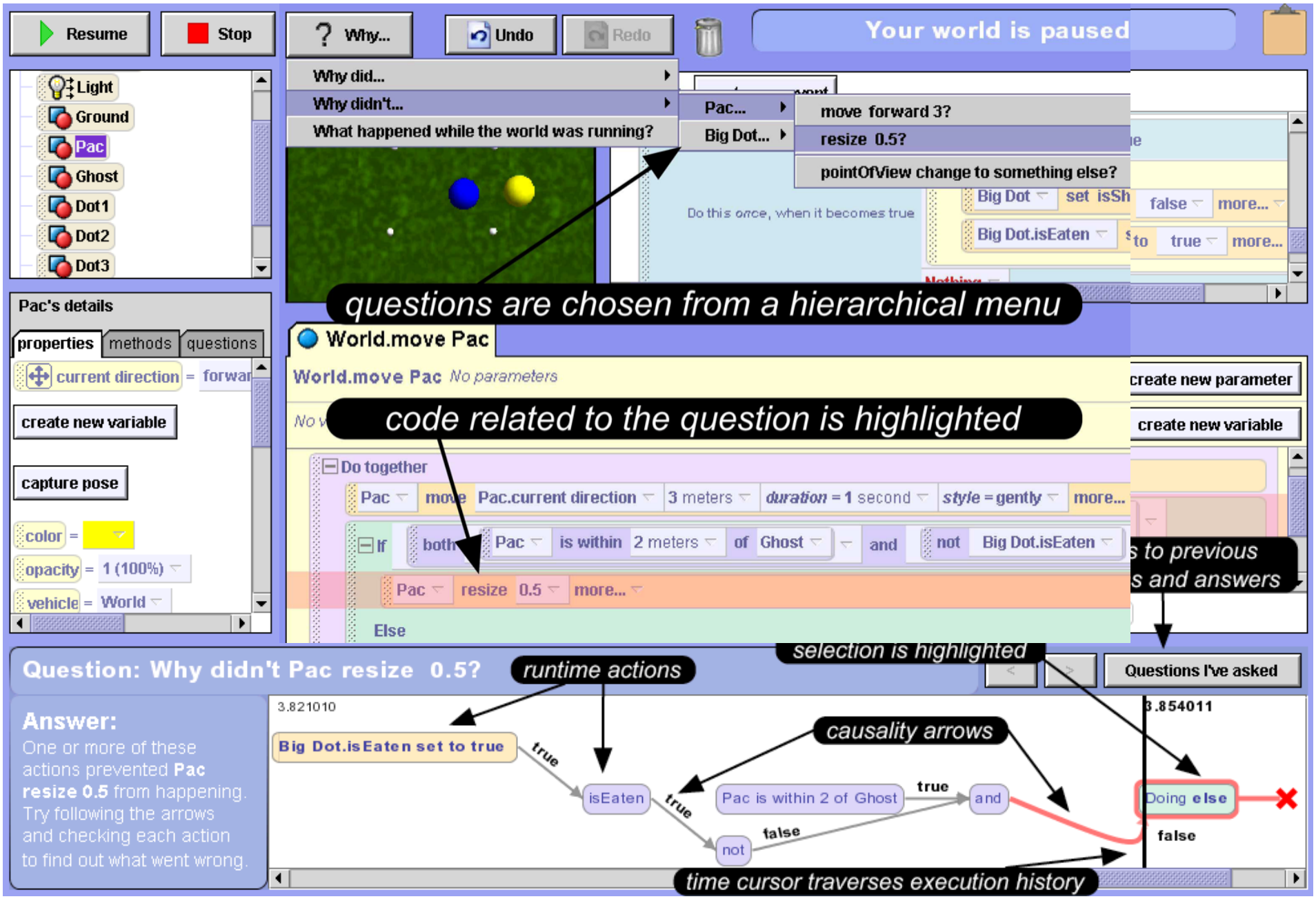
Discussion

- Dynamic slices are much more precise than static slices (applied to the one run, that is)
- From some variable, a backward slice encompasses on average
  - 30% of the *entire* program (static slice)
  - 5% of the *executed* program (dynamic slice)
- Overhead as in omniscient debugging

The Whyline


Resume
Stop
Why...
Undo
Redo
Your world is paused.
further questions can be asked
When Pac is within 1 meter of Big Dot becomes true
Do in order
Do this once, when it becomes true
Big Dot set isShowing to false more...
Big Dot.isEaten set value to true more...
Pac
Ghost
Dot1
Dot2
Dot3
ADD OBJECTS
camera focuses on subject of question
Pac's details
properties
methods
questions
current direction = forwar
create new variable
capture pose
color =
opacity = 1 (100%)
vehicle = World
World.move Pac
World.move Pac No parameters
No variables
tooltips show properties' current values
create new parameter
create new variable
Pac move Pac.current direction 3 meters duration = 1 second style = gentl more...
true
If both Pac is within 2 meters of Ghost and not Big Dot.isEaten
Pac resize 0.5 more...
access to previous questions and answers
Do in order
Do together
If/Else
Loop
While
print
code related to the selection is highlighted
Question: Why didn't Pac resize 0.5?
runtime actions
Questions I've asked
Answer:
One or more of these actions prevented Pac resize 0.5 from happening. Try following the arrows and checking each action to find out what went wrong.
3.821010
3.854011
Big Dot.isEaten set to true
true
isEaten
true
causality arrows
Pac is within 2 of Ghost
true
and
Doing else
not
false
false
time cursor traverses execution history


Resume
Stop
Why...
Undo
Redo
Your world is paused.
further questions can be asked
Light
Ground
Pac
Ghost
Dot1
Dot2
Dot3
When Pac is within 1 meter of Big Dot becomes true
Do in order
Do this once, when it becomes true
Big Dot set isShowing to false more...
Big Dot.isEaten set value to true more...
ADD OBJECTS
camera focuses on subject of question
Pac's details
properties
methods
questions
current direction = forwar
create new variable
capture pose
color =
opacity = 1 (100%)
vehicle = World
World.move Pac
World.move Pac No parameters
No variables
create new parameter
create new variable
tooltips show properties' current values
Pac move Pac.current direction 3 meters duration = 1 second style = gentl more...
true
If both Pac is within 2 meters of Ghost and not Big Dot.isEaten
Pac resize 0.5 more...
access to previous questions and answers
Do in order
Do together
If/Else
Loop
While
print
code related to the selection is highlighted
Question: Why didn't Pac resize 0.5?
runtime actions
Questions I've asked
Answer:
One or more of these actions prevented Pac resize 0.5 from happening. Try following the arrows and checking each action to find out what went wrong.
3.821010
3.854011
Big Dot.isEaten set to true
true
isEaten
true
causality arrows
Pac is within 2 of Ghost
true
and
Doing else
not
false
false
time cursor traverses execution history

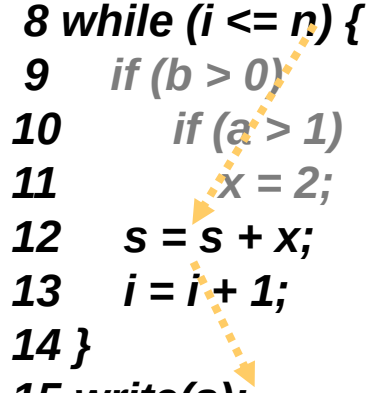

Resume
Stop
Why...
Undo
Redo
Your world is paused
Why did...
Why didn't...
What happened while the world was running?
Pac...
Big Dot...
move forward 3?
resize 0.5?
pointOfView change to something else?
Light
Ground
Pac
Ghost
Dot1
Dot2
Dot3
Pac's details
properties
methods
questions
current direction = forwar
create new variable
capture pose
color =
opacity = 1 (100%)
vehicle = World
questions are chosen from a hierarchical menu
World.move Pac
World.move Pac No parameters
code related to the question is highlighted
Do together
Pac move Pac.current direction 3 meters duration = 1 second style = gently more...
If both Pac is within 2 meters of Ghost and not Big Dot.isEaten
Pac resize 0.5 more...
Else
create new parameter
create new variable
Do this once, when it becomes true
Big Dot.isEaten
false
more...
true
Question: Why didn't Pac resize 0.5?
runtime actions
selection is highlighted
Questions I've asked
Answer:
One or more of these actions prevented Pac resize 0.5 from happening. Try following the arrows and checking each action to find out what went wrong.
3.821010
3.854011
Big Dot.isEaten set to true
isEaten
Pac is within 2 of Ghost
and
not
Doing else
causality arrows
time cursor traverses execution history

“Why did” questions

- Take the dynamic slice of the variable
- Follow at most two dependencies
- If programmer wants, follow dependencies transitively

“Why did s = 2 in Line 15?”

```
1 n = read(); // n = 2
2 a = read(); // a = 0
3 x = 1;
4 b = a + x;
5 a = a + 1;
6 i = 1;
7 s = 0;
8 while (i <= n) {
9     if (b > 0)
10        if (a > 1)
11            x = 2;
12    s = s + x;
13    i = i + 1;
14 }
15 write(s);
```



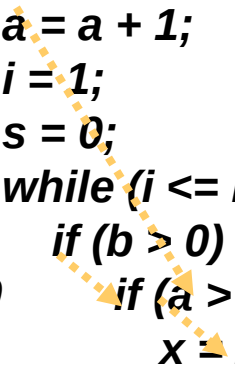
“Because s = 1 and i = 2”

“Why didn’t” questions

- Follow back control dependencies to closest controlling statement(s)
- Do a “why did” question on each
- Again, follow at most two dependencies

“Why didn’t x = 2 in Line 11?”

```
 1 n = read(); // n = 2
 2 a = read(); // a = 0
 3 x = 1;
 4 b = a + x;
 5 a = a + 1;
 6 i = 1;
 7 s = 0;
 8 while (i <= n) {
 9     if (b > 0)
10         if (a > 1)
11             x = 2;
12     s = s + x;
13     i = i + 1;
14 }
15 write(s);
```



“Because a = 1 and b = 1”

Discussion

- The WHYLINE combines
  - omniscient debugging
  - static slicing
  - dynamic slicing
- in an attractive package, showcasing the state of the art in interactive debugging

Tracking Infections

1. Start with the infected value as seen in the failure
2. Follow back the dependencies
3. Observe and judge origins – are they sane?
4. If some origin is infected, repeat at Step 2
5. All origins are sane? Here's the infection site!

Concepts

- Omniscient debugging allows for simple exploration of the entire execution history
- Dynamic slicing tells the origin of a value
- To track down an infection, follow dependencies and observe origins, repeating the process for infected origins