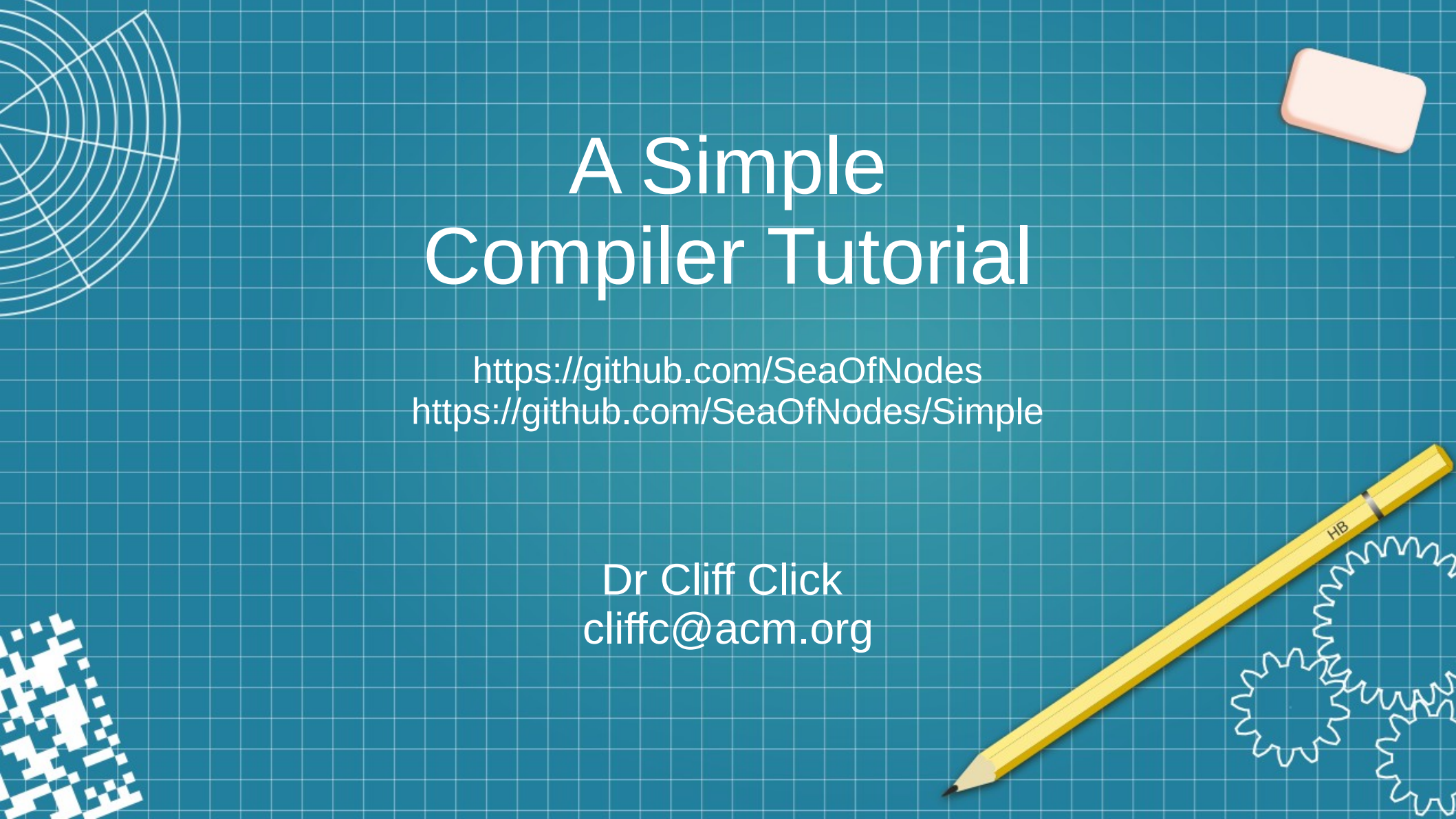


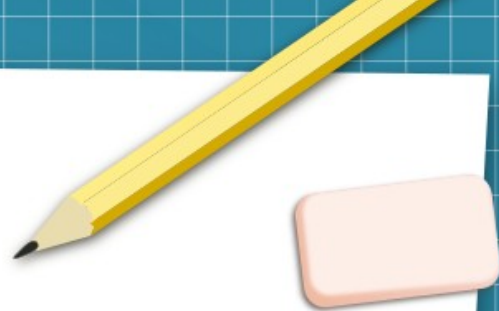


A Simple Compiler Tutorial

<https://github.com/SeaOfNodes>
<https://github.com/SeaOfNodes/Simple>

Dr Cliff Click
cliffc@acm.org

A Simple Compiler Tutorial

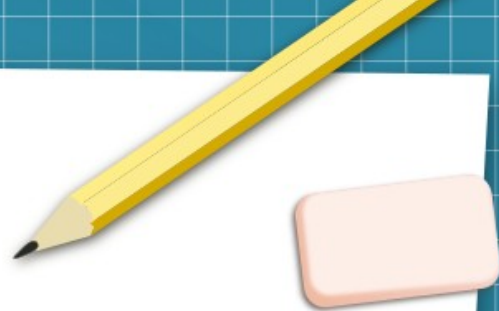


- Open Source! Apache v2

<https://github.com/SeaOfNodes>
<https://github.com/SeaOfNodes/Simple>

- Java repo is active and most complete
 - 24 chapters, 25th is well underway
- Features Sea-of-Nodes IR
 - Not my father's IR! (but might predate you....)

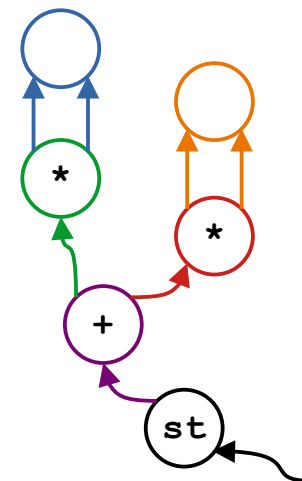
Sea-of-Nodes



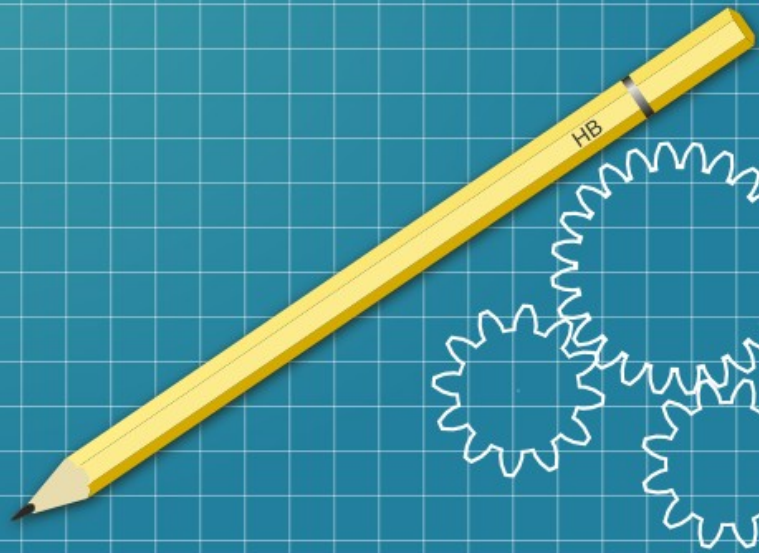
- What is the “Sea of Nodes”?
- A Compiler IR
 - Used in the HotSpot JIT for 25 years
 - Very fast to build and optimize, very robust, excellent code quality
 - Roughly ‘gcc -O2’ quality in 1/100th the time
 - Inspiration for Graal and other compilers
 - Always **SSA**, All The Time
 - Graph based – unusual!
 - Intermingled control and data flow – VERY unusual!

SSA as a Graph

- SSA: Single “name” for all values / points
 - By name I mean: “a graph Node”
 - By “Node” I mean: a C-style ptr to a data structure called “Node”
 - Which you rename as the text-form %p of the pointer: “0xcafebabe”
 - All these name-forms are functionally equivalent
 - **Which is the whole point of SSA in the first place!**
- Non-goal: Minimal SSA, dropping that on the floor
- Build straight from parsing program text
 - But works fine from AST as well



- Chapters 1,2: Expressions are SSA



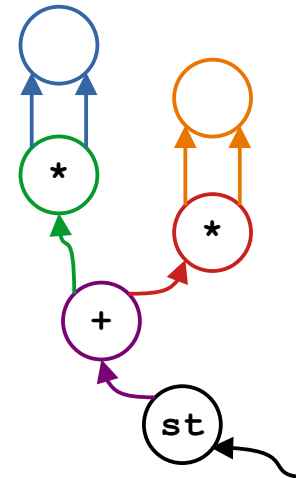
Expressions are SSA

- Standard expression DAG
 - Equivalent forms, up to Alpha-Renaming:
 - Globally unique Names & expressions
 - List of tuples containing opcode+names
 - Pointers to tuples containing opcode+pointers
 - Which is a rename away from a graph

dist=x*x+y*y;

rx	op	...	
ry	op	...	
r0	*	rx	rx
r1	*	ry	ry
r2	+	r0	r1
rd	st	r2	

	op		
	op		
	*		
	*		
	+		
	st		

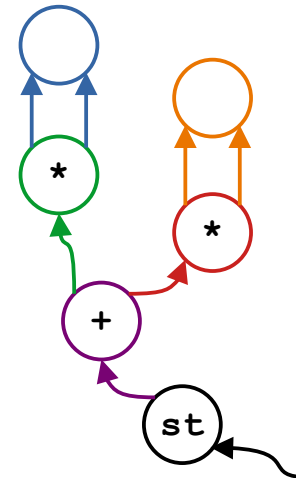
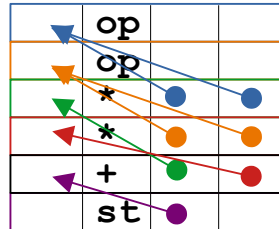


Data Flow

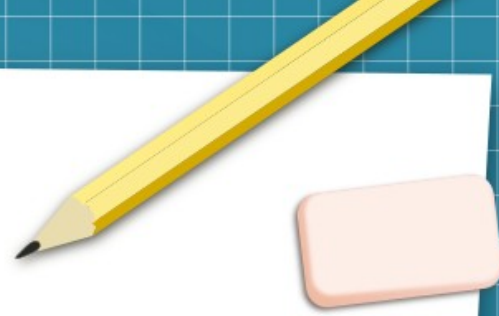
- Node has:
 - use → def edges, ordered, required for semantics
 - opcode, again semantics
 - def → use edges, used for fast optimization
 - unique int, for bitsets for fast graph walking
 - maybe a constant
 - semantic analysis fields

dist=x*x+y*y;

rx	op	...	
ry	op	...	
r0	*	rx	rx
r1	*	ry	ry
r2	+	r0	r1
rd	st	r2	



Let's SSA: $x*x+y*y$



- Text Cursor `dist = x*x + y*y`
 - Ignore parsing ↑
 - Slight diff AST parsing
- Expression stack
 - Oper prec
- Not about parsing!
 - Skipping details

Let's SSA: $x*x+y*y$

- Text Cursor
- Expr stack to empty / null
- Symbol Table!
 - Map from name $\rightarrow$ node
 - Java bytecodes the “name” is just a reg num

`dist = x*x + y*y`



stk

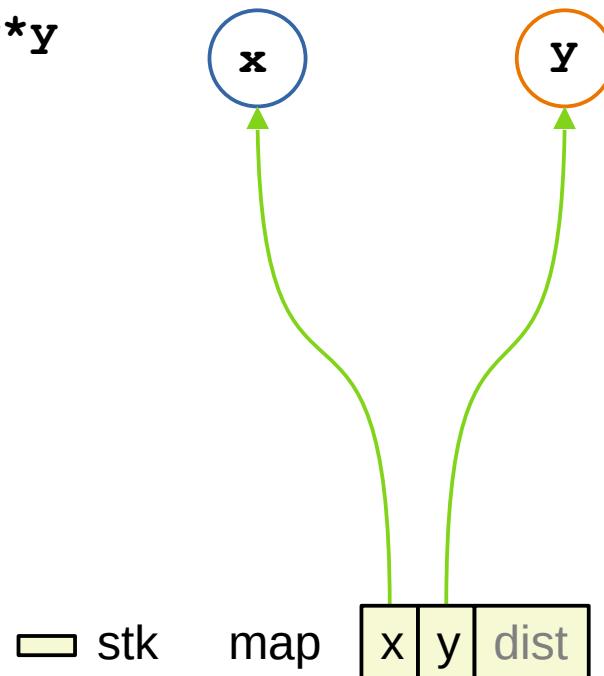
map

x	y	dist
---	---	------

Let's SSA: $x*x+y*y$

- Text Cursor
- Expr stack to empty / null
- Symbol Table!
 - name $\rightarrow$ node
- 'x' and 'y' already mapped
- Parser-only edges are **lime**

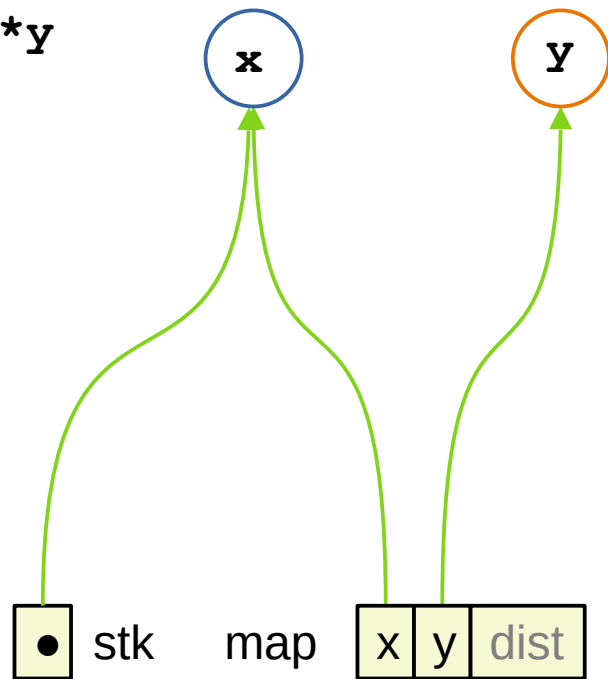
`dist = x*x + y*y`
↑



Let's SSA: $x*x+y*y$

- Parse 'x'
- Lookup 'x' node
- Push node

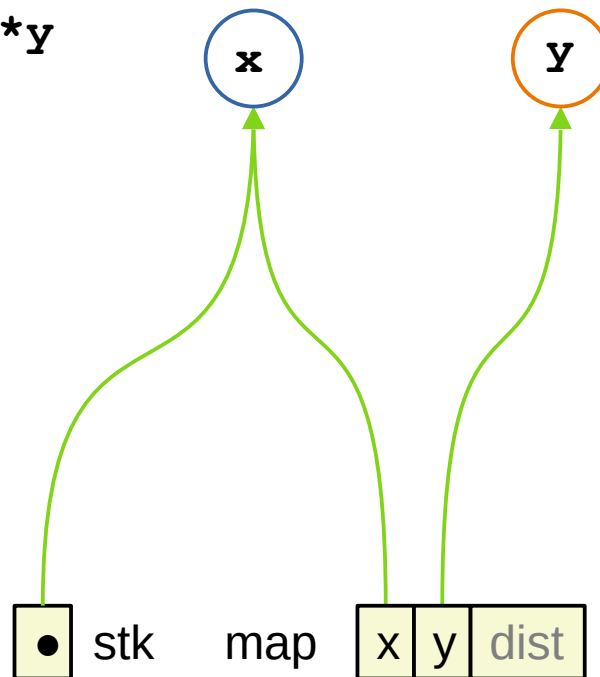
$\text{dist} = x*x + y*y$
↑



Let's SSA: $x*x+y*y$

- Parse '*'
 - Oper!
 - Save for later
- Insert Your Favorite Precedence Parsing Strategy Here...

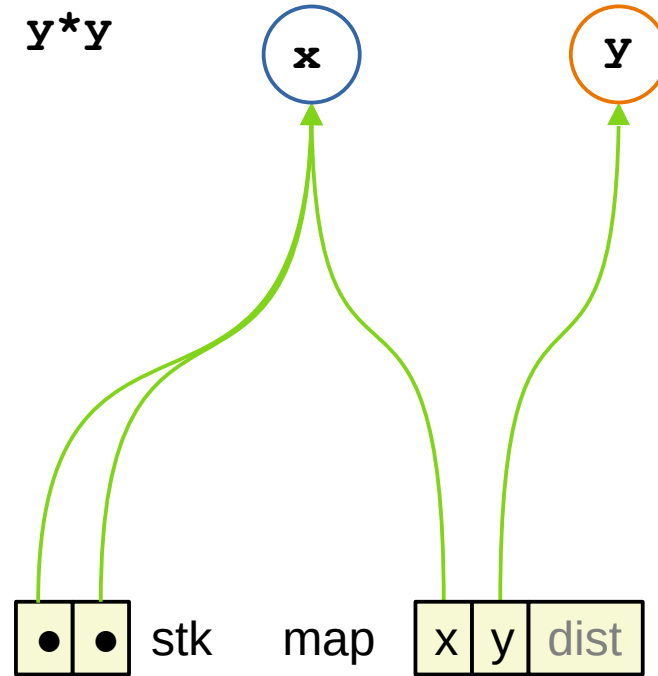
`dist = x*x + y*y`
 ↑



Let's SSA: $x*x+y*y$

- Parse 'x'
- Lookup 'x' node
- Push node

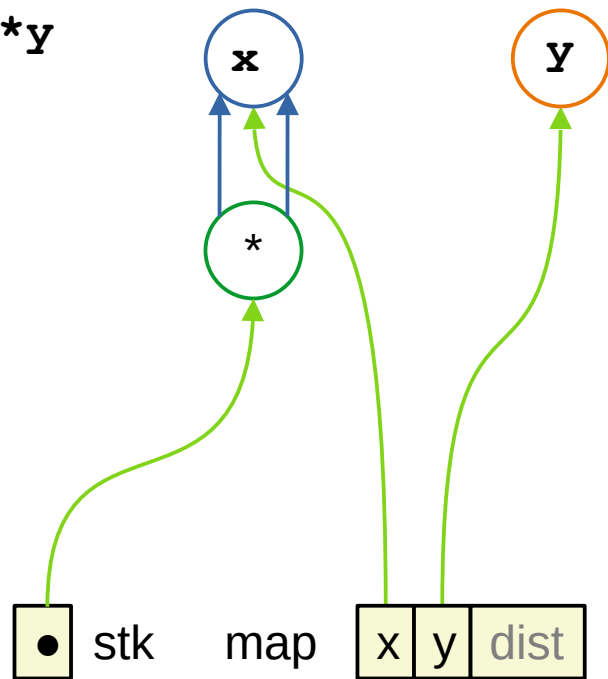
`dist = x*x + y*y`
 ↑



Let's SSA: $x*x+y*y$

- End oper '*'
- Pop 2 nodes
- Build '*' node
- Push on stack

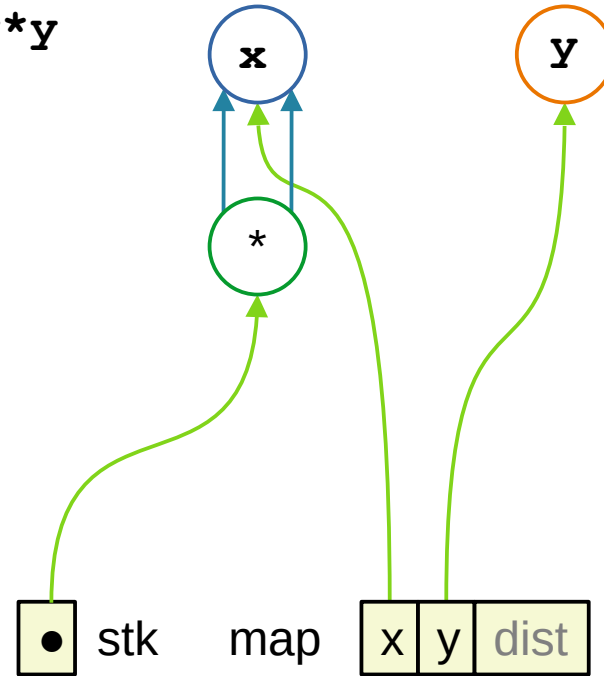
`dist = x*x + y*y`



Let's SSA: $x*x+y*y$

- Parse '+'
 - Oper!
 - Save for later

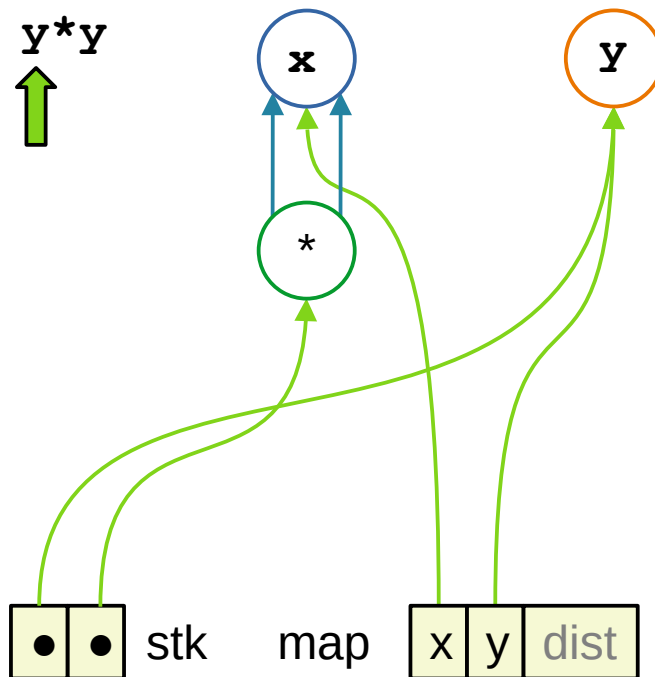
`dist = x*x + y*y`
 ↑



Let's SSA: $x*x+y*y$

- Parse 'y'
- Lookup 'y' node
- Push node

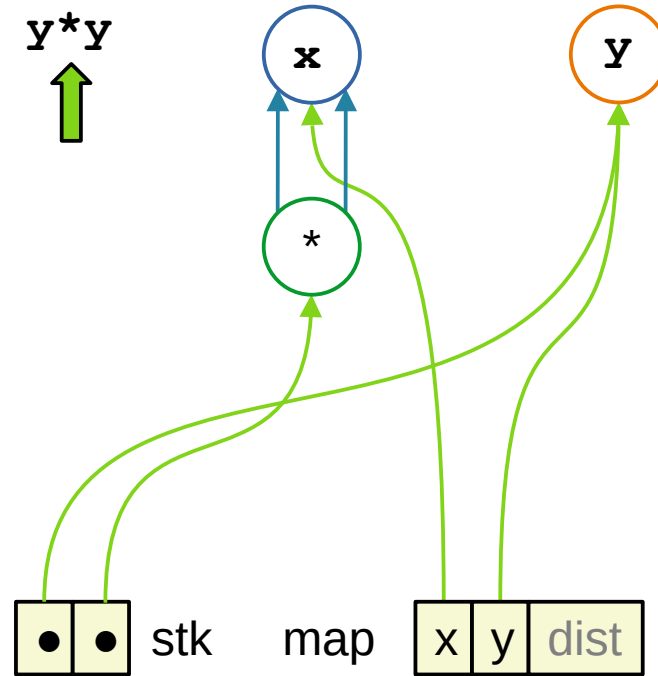
`dist = x*x + y*y`
 ↑



Let's SSA: $x*x+y*y$

- Parse '*'
 - Oper!
 - Save for later

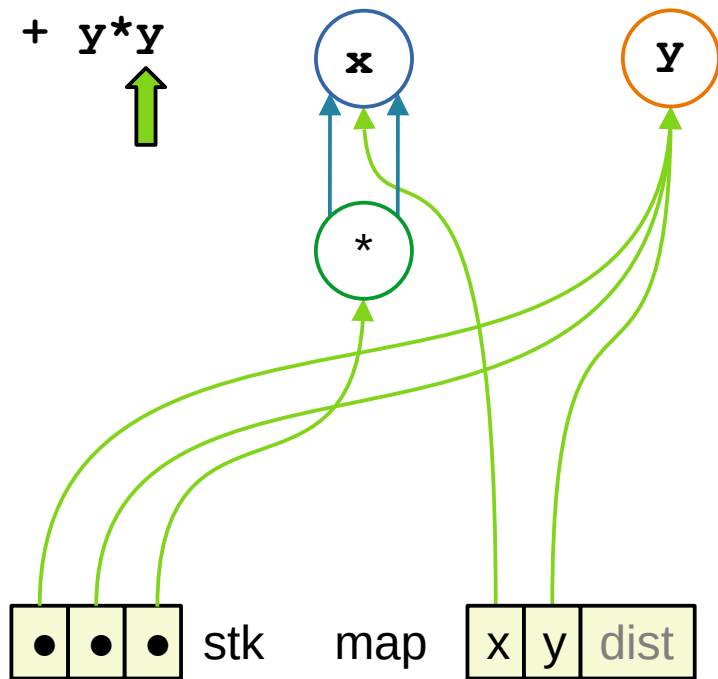
`dist = x*x + y*y`
 ↑



Let's SSA: $x*x+y*y$

- Parse 'y'
- Lookup 'y' node
- Push node

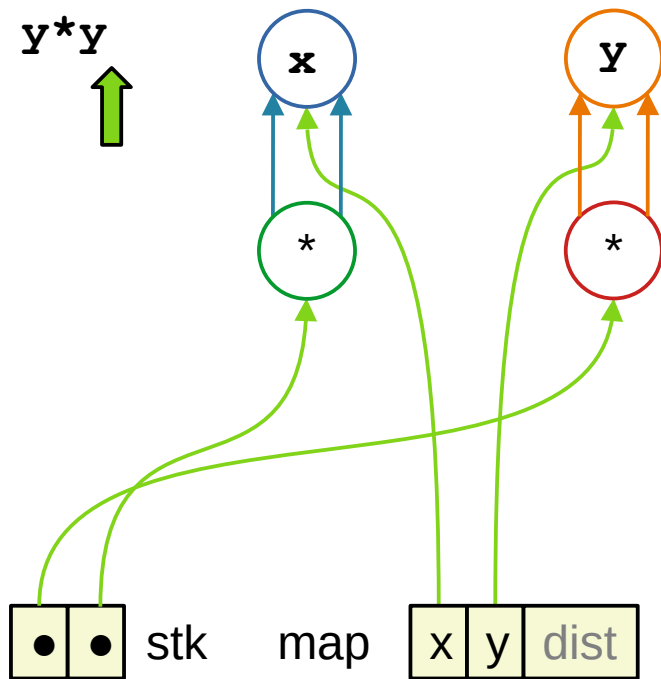
`dist = x*x + y*y`



Let's SSA: $x*x+y*y$

- End oper '*'
- Pop 2 nodes
- Build '*' node
- Push on stack

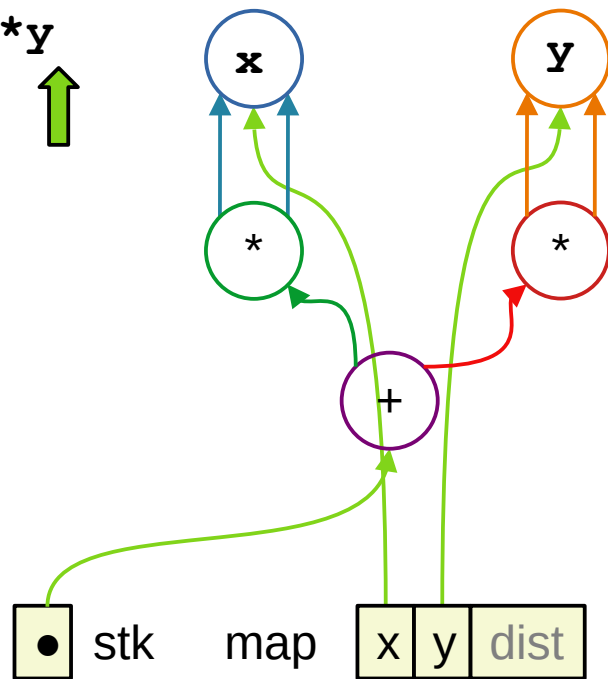
`dist = x*x + y*y`



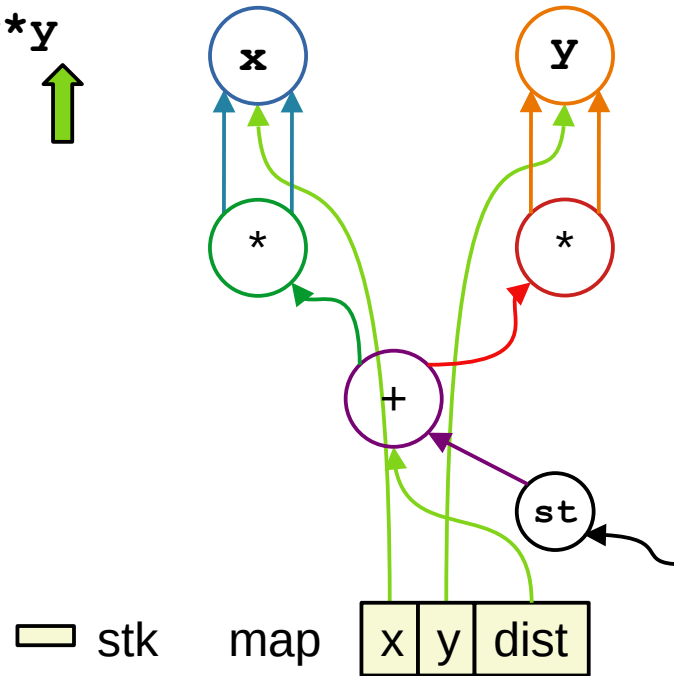
Let's SSA: $x*x+y*y$

- End oper '+'
- Pop 2 nodes
- Build '+' node
- Push on stack

`dist = x*x + y*y`

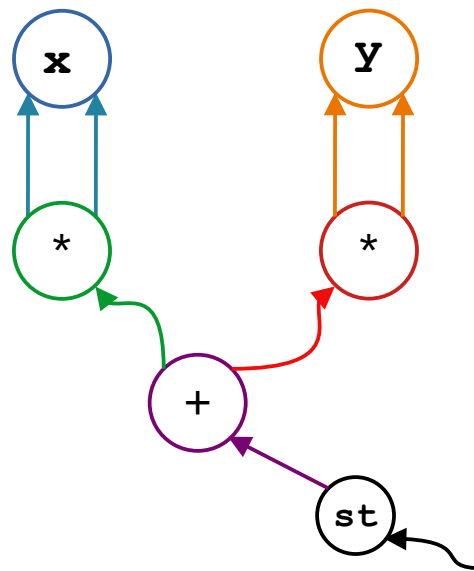


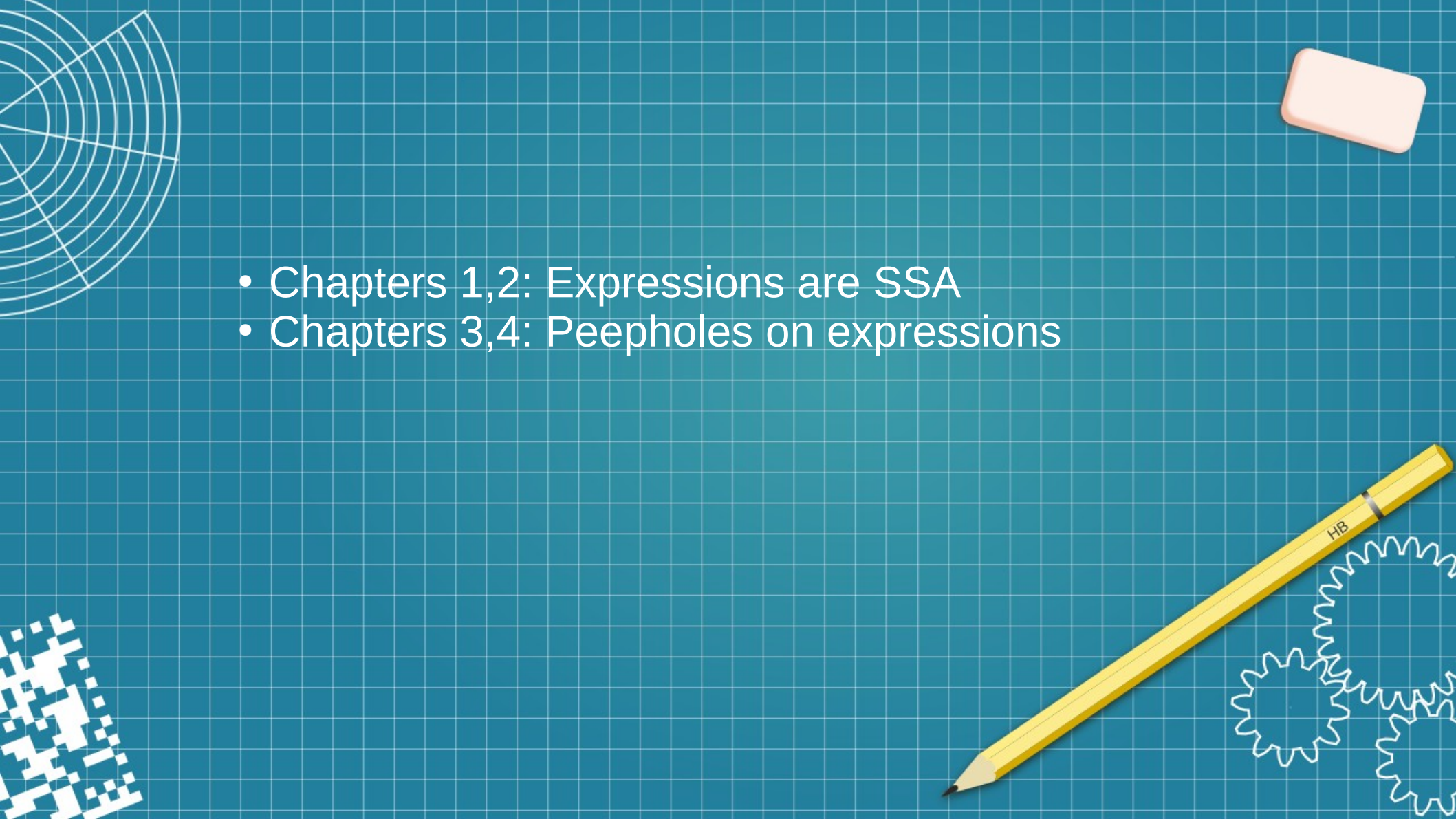
```
dist = x*x + y*y
```



Let's SSA: $x*x+y*y$

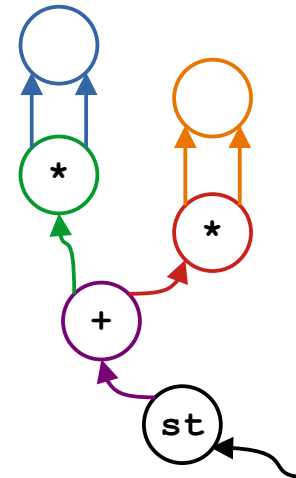
- Hey! This is an expression DAG!
- Yup!
- No control-flow, so exactly the same.
- Its also Use $\rightarrow$ Def edges
 - Easy to flip and get D $\rightarrow$ U



- 
- Chapters 1,2: Expressions are SSA
 - Chapters 3,4: Peephholes on expressions

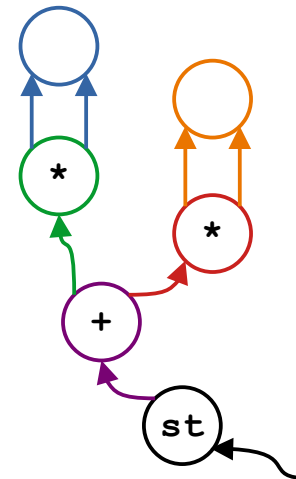
Optimizations

- Peepholes
 - Graph rewrite rules – lots of very simple rules
 - Peek at small region, rewrite it, neighbors on worklist
 - Repeat until worklist empties
 - Fast, linear, 50% of all gains
- Pessimistic data flow analysis (forwards and backwards)
 - Actually, just another Peephole
- CSE / Value-Numbering (yes, another Peephole)
- Optimistic data flow analysis
 - “whole compilation unit”
 - Starting point for the next round of Peepholes
- Other Stuff – not in this talk!
 - Inlining, Loops, code-gen, reg alloc



Peephole Rules – Data

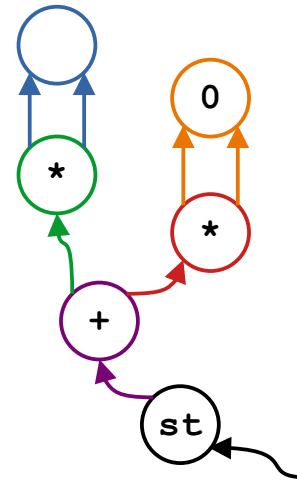
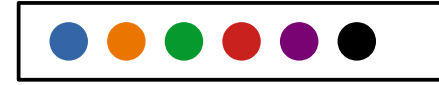
- Major cleanup & gains
- Uses worklist, def → use edges for neighbors
- Many peephole rules tied to opcode
 - Virtual call in C++, Java
 - Constant-prop another v-call
 - Reverse (liveness) v-call
 - Hindley-Milner v-call
- Dead code and CSE same for all



Peephole Rules – Data

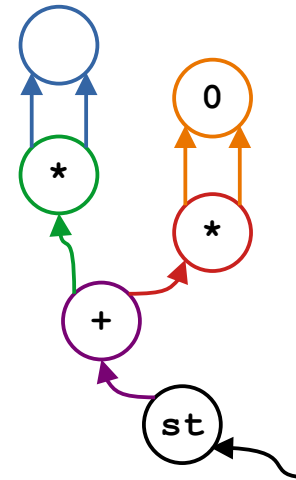
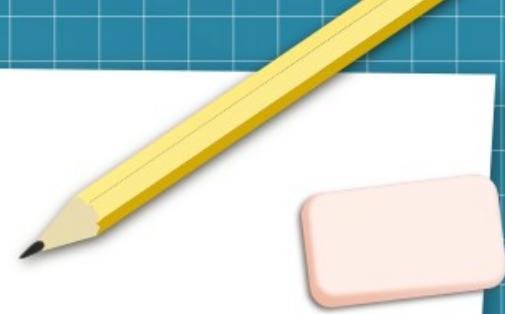
- Example: `y=0; dist=x*x+y*y`

worklist



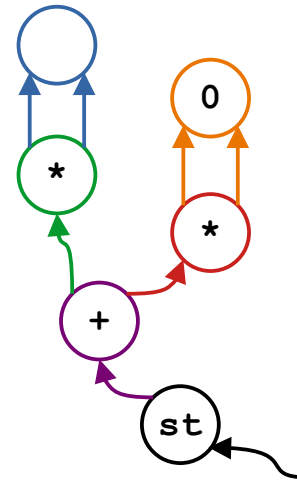
Peephole Rules – Data

- Example: `y=0; dist=x*x+y*y`
- Pop $\textcircled{\text{st}}$, not dead, no CSE
- Call 'ideal', no progress, done.



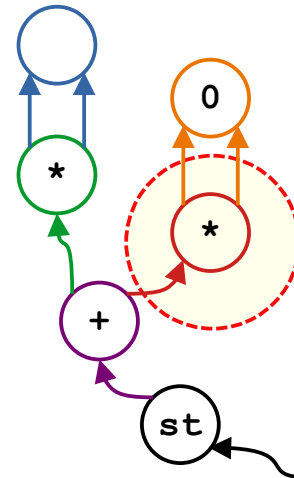
Peephole Rules – Data

- Example: `y=0; dist=x*x+y*y`
- Pop $\oplus$, not dead, no CSE
- Call 'ideal', no progress, done.



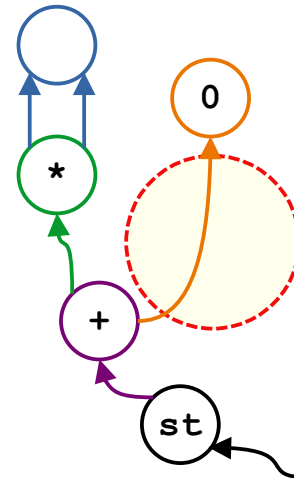
Peephole Rules – Data

- Example: `y=0; dist=x*x+y*y`
- Pop $*$, not dead, no CSE
- Call 'ideal'
 - Multiply of zero has rule



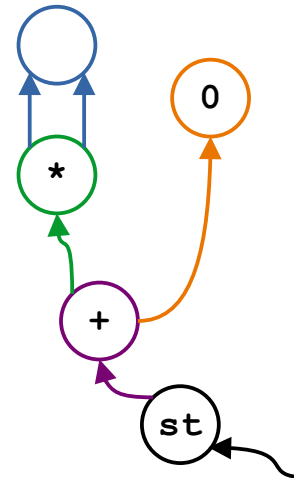
Peephole Rules – Data

- Example: `y=0; dist=x*x+y*y`
- Pop $*$, not dead, no CSE
- Call 'ideal'
 - Multiply of zero has rule
 - Replace with zero



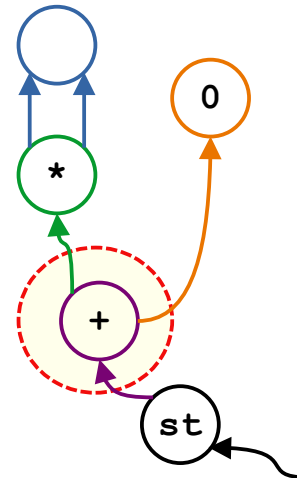
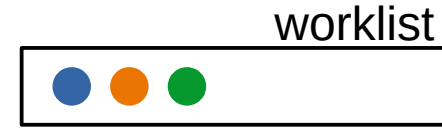
Peephole Rules – Data

- Example: `y=0; dist=x*x+y*y`
- Pop $*$, not dead, no CSE
- Call 'ideal'
 - Multiply of zero has rule
 - Replace with zero
 - Neighbors on list



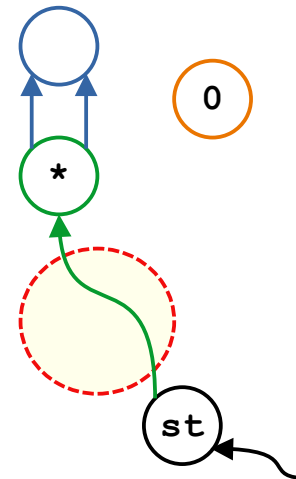
Peephole Rules – Data

- Example: `y=0; dist=x*x+y*y`
- Pop $\oplus$, not dead, no CSE
- Call 'ideal'
 - Add of zero has a rule



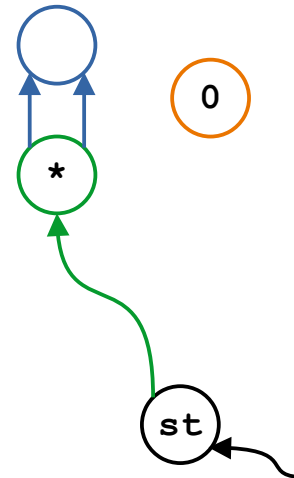
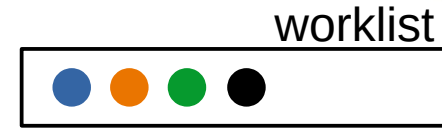
Peephole Rules – Data

- Example: `y=0; dist=x*x+y*y`
- Pop $+$, not dead, no CSE
- Call 'ideal'
 - Add of zero has a rule
 - Replace with other input



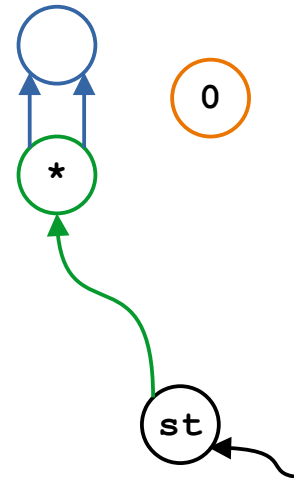
Peephole Rules – Data

- Example: $y=0$; $\text{dist}=x*x+y*y$
- Pop $+$, not dead, no CSE
- Call 'ideal'
 - Add of zero has a rule
 - Replace with other input
 - Neighbors on worklist



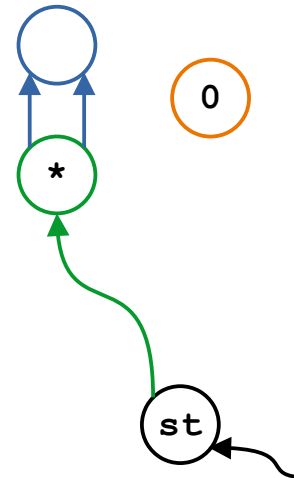
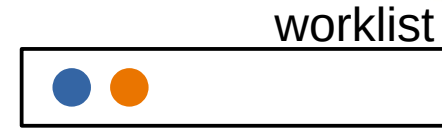
Peephole Rules – Data

- Example: `y=0; dist=x*x+y*y`
- Pop $\textcircled{\text{st}}$, not dead, no CSE
- Call 'ideal', no progress, done.



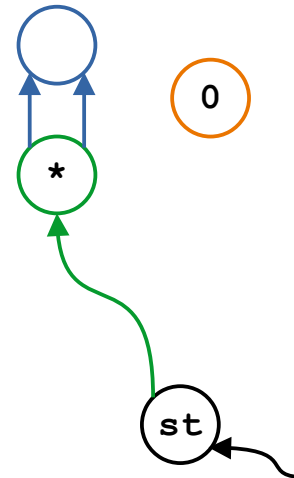
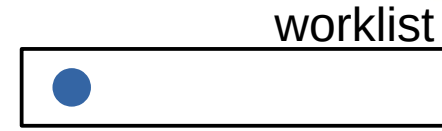
Peephole Rules – Data

- Example: `y=0; dist=x*x+y*y`
- Pop $*$, not dead, no CSE
- Call 'ideal', no progress, done.



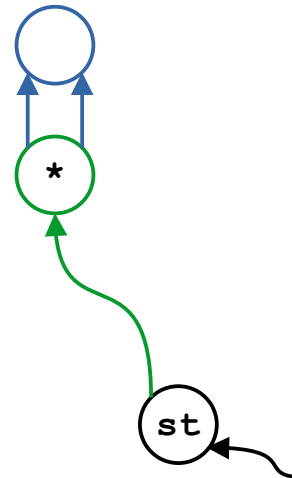
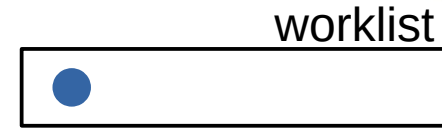
Peephole Rules – Data

- Example: `y=0; dist=x*x+y*y`
- Pop $\textcircled{0}$, dead!



Peephole Rules – Data

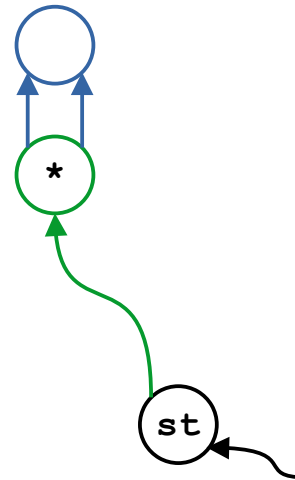
- Example: `y=0; dist=x*x+y*y`
- Pop 0, dead!
 - Delete
 - Neighbors on list, done.

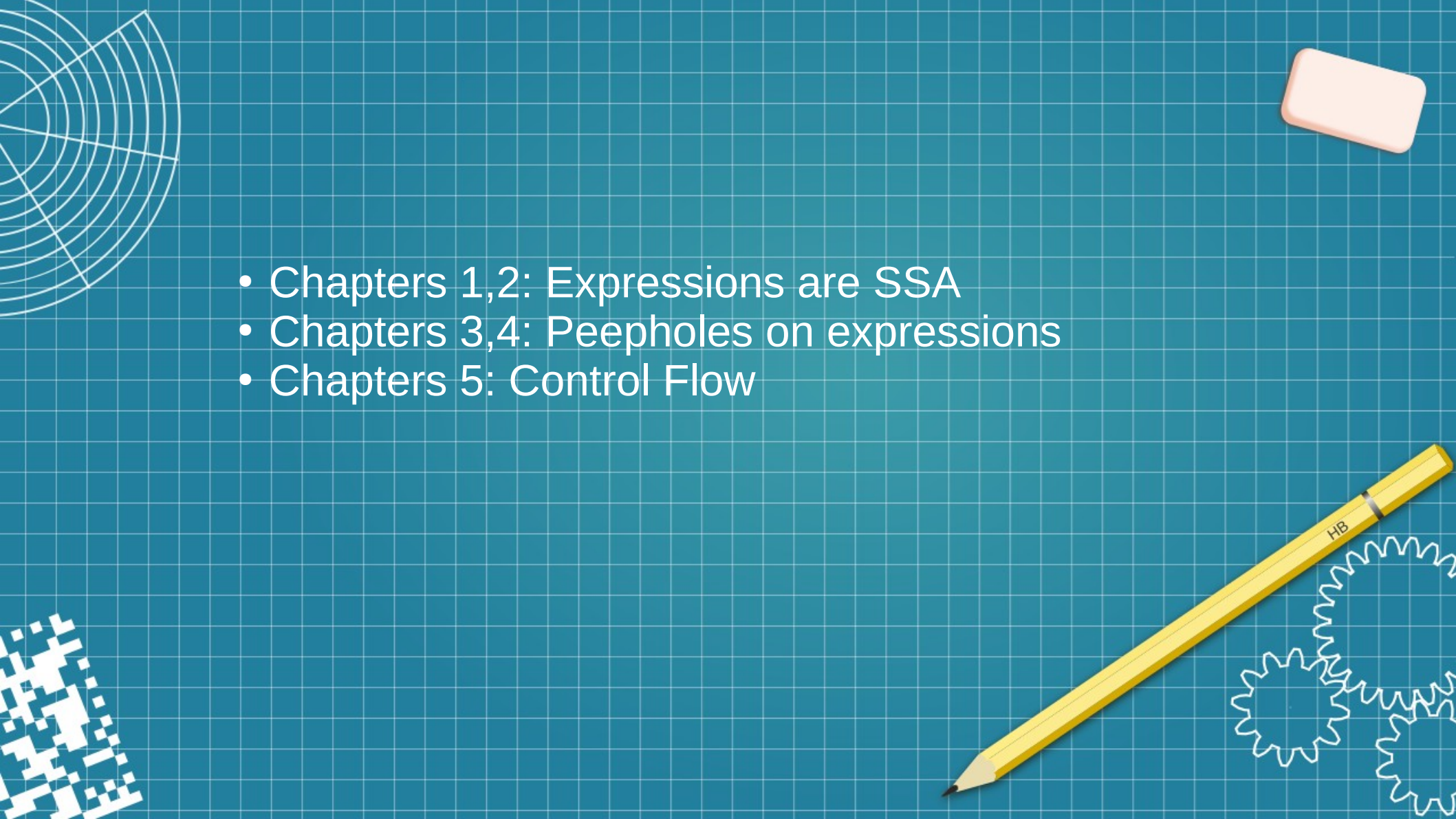


Peephole Rules – Data

- Example: `y=0; dist=x*x+y*y`
- Pop , not dead, no CSE
- Call 'ideal', no progress, done.

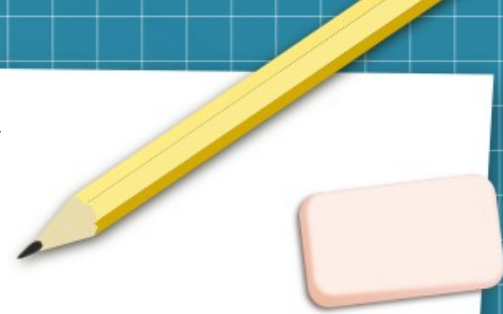
worklist

- 
- Chapters 1,2: Expressions are SSA
 - Chapters 3,4: Peephholes on expressions
 - Chapters 5: Control Flow

Sea of Nodes: Control Flow

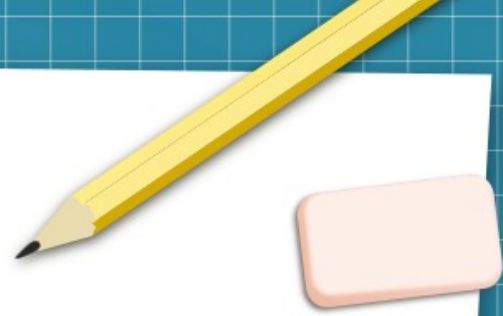
- Normal CFG
 - Embedded in the same Graph Format
 - So **looks** a little different
 - Same Peephole rules apply
- Data nodes lose a “basic block place”
 - Inferred by edges
 - Recovered via straightforward graph-walk during code-gen
 - Not needed for most optimizations
 - Yes needed for bulk rewrites, e.g. loop opts
 - Not for all peepholes, both control and data, cse, inline, dead, cprop, ...



Control Flow Start

- Starting point

```
if( tmp ) x=3 else x=4
```

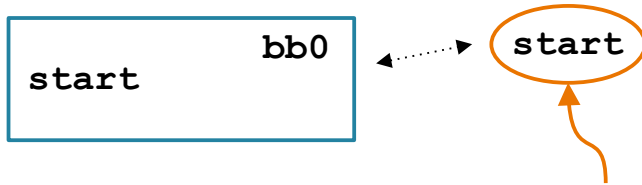


Control Flow Start

- Starting point

```
if( tmp ) x=3 else x=4
```

- First Basic Block ↔ First Control Node



- start is Just another Node

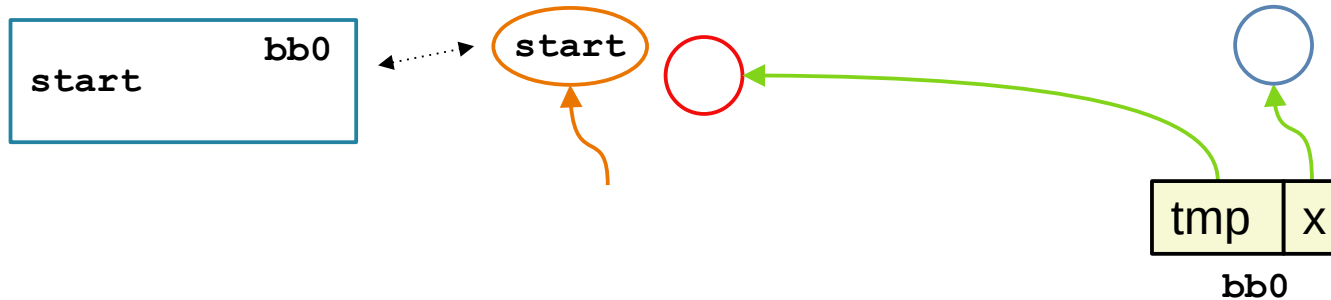
- No inputs, has outputs, has 'start' opcode

Control Flow Start

- Starting point

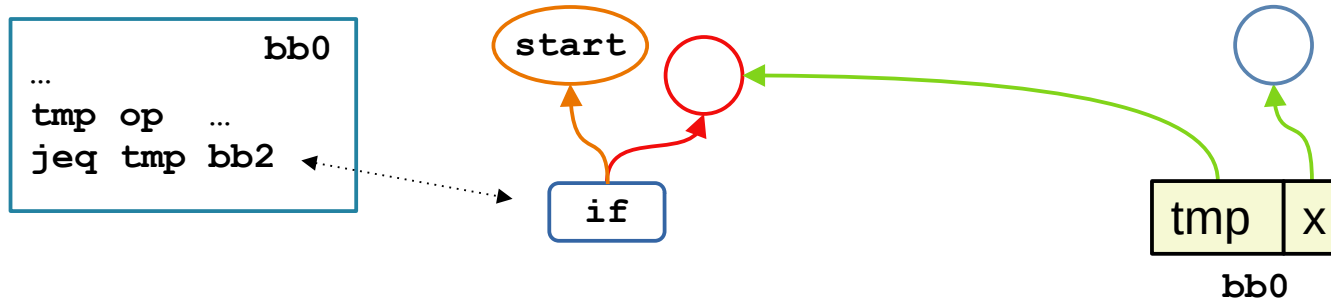
```
if( tmp ) x=3 else x=4
```

- Data – tmp and x come from before



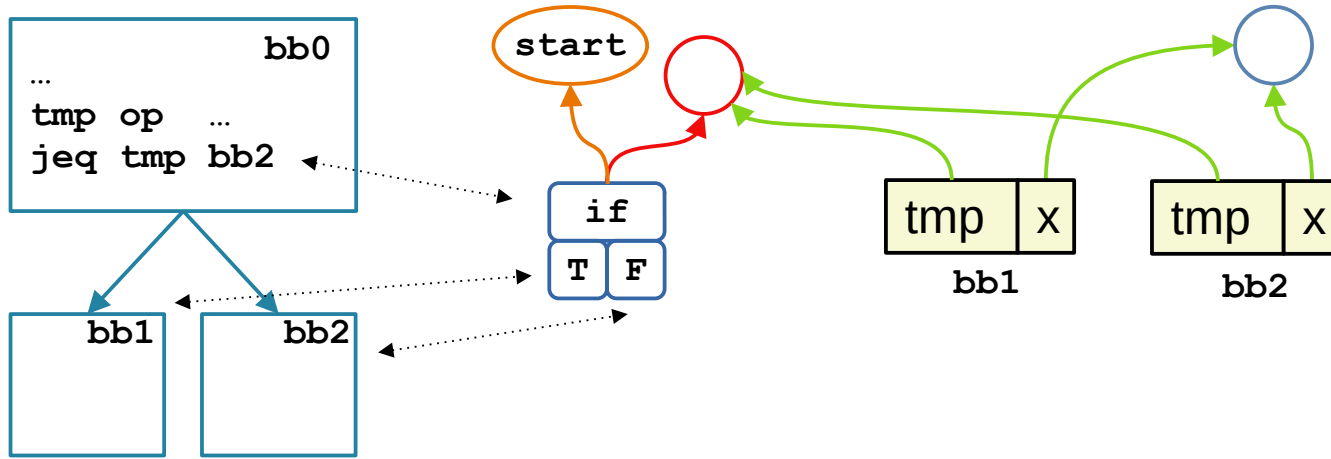
Control Flow Split

- Test and branch: `if( tmp ) x=3 else x=4`
 - Split in CFG $\leftrightarrow$ Split in Sea-of-Nodes



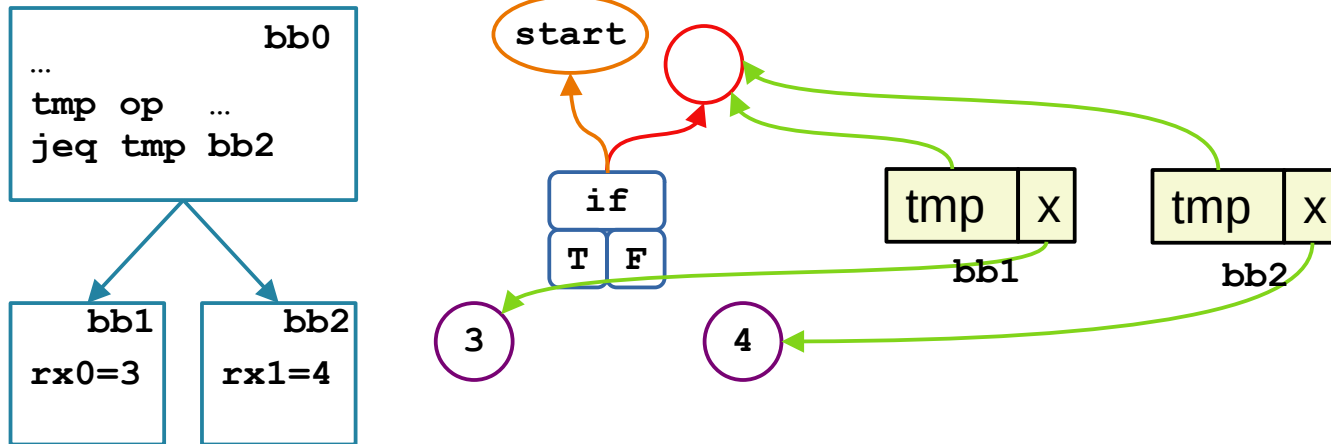
Control Flow Split

- Test and branch: `if( tmp ) x=3 else x=4`
 - Split in CFG $\leftrightarrow$ Split in Sea-of-Nodes



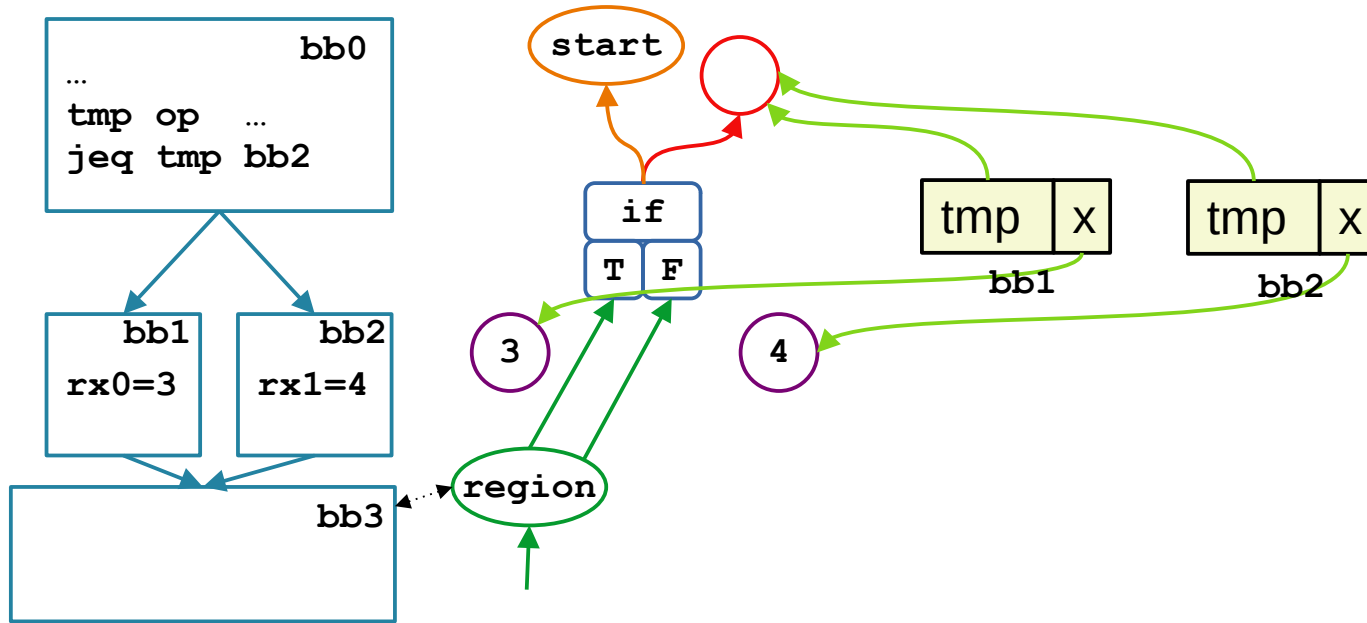
Control Flow Split

- Test and branch: `if( tmp ) x=3 else x=4`
 - Data nodes from either block lose their 'place'

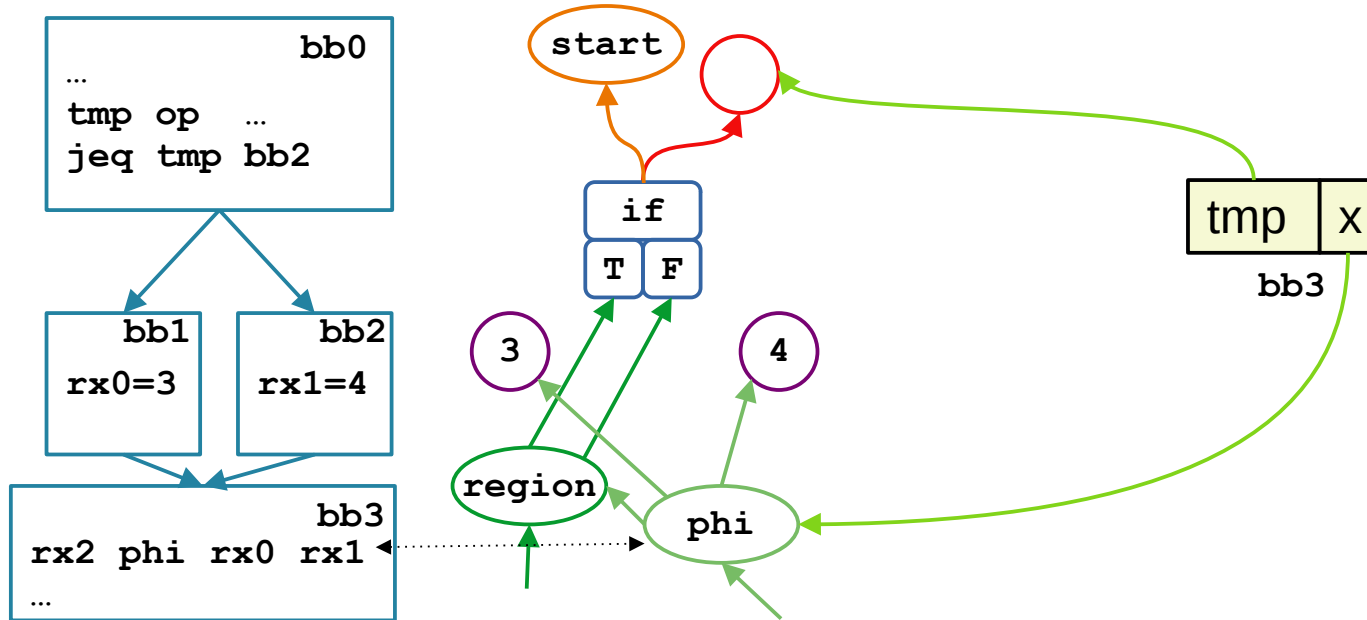


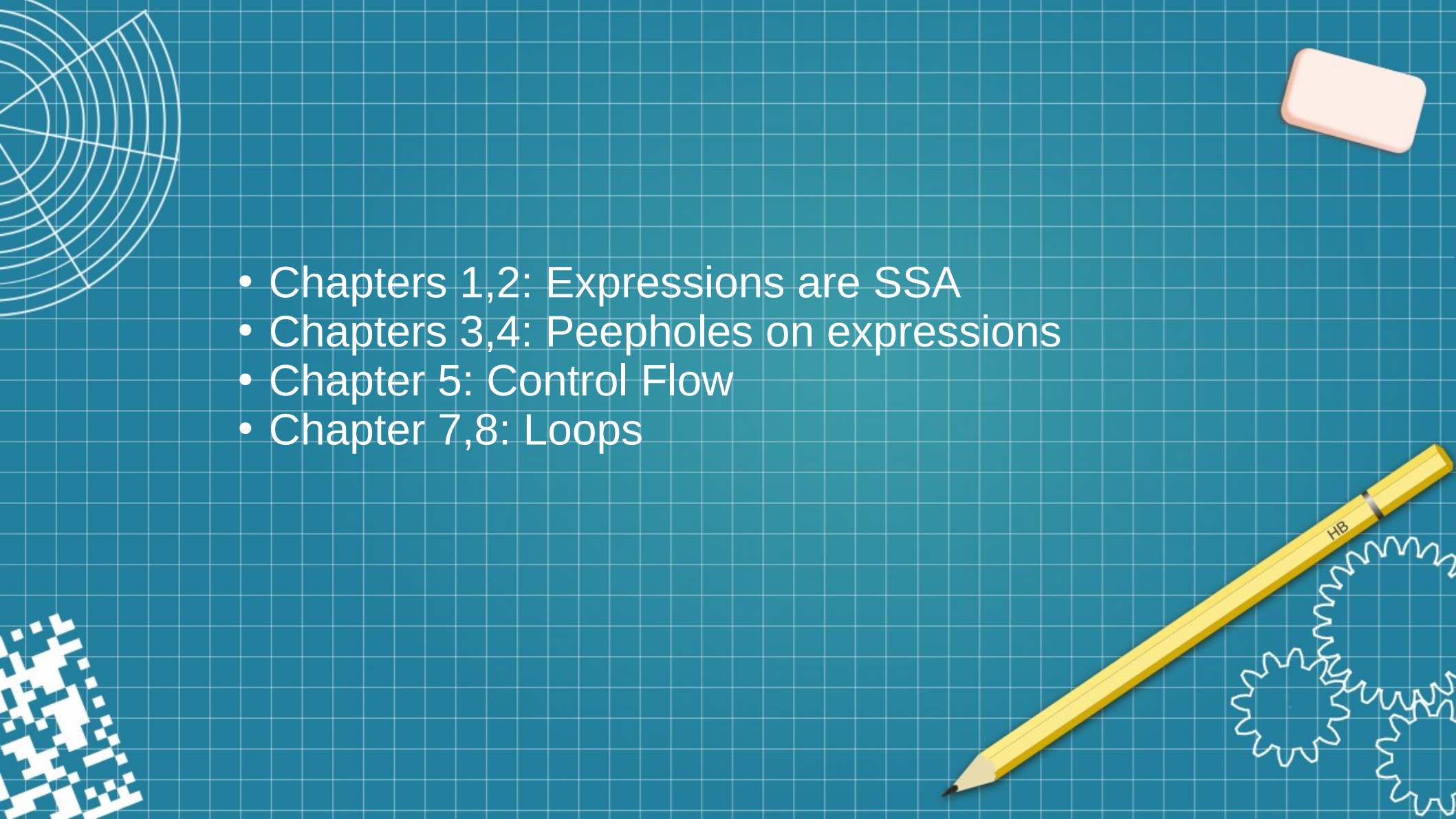
Control Flow Merge

- Region merges control flow (i.e. Basic Block head)
 - SSA Phi nodes inserted both CFG & Sea-of-Nodes



- 



- 
- Chapters 1,2: Expressions are SSA
 - Chapters 3,4: Peephholes on expressions
 - Chapter 5: Control Flow
 - Chapter 7,8: Loops

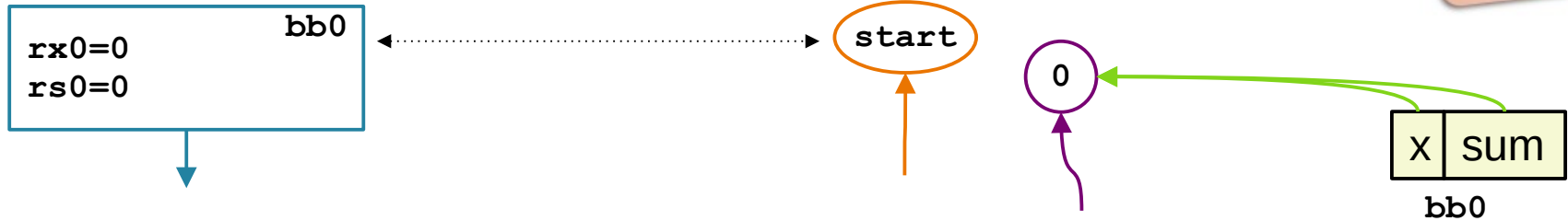
Control Flow Merge - Loops

- Region comes first, otherwise same as before

```
x=sum=0;  
while(x<10)  
    sum += x++;
```

Control Flow Merge - Loops

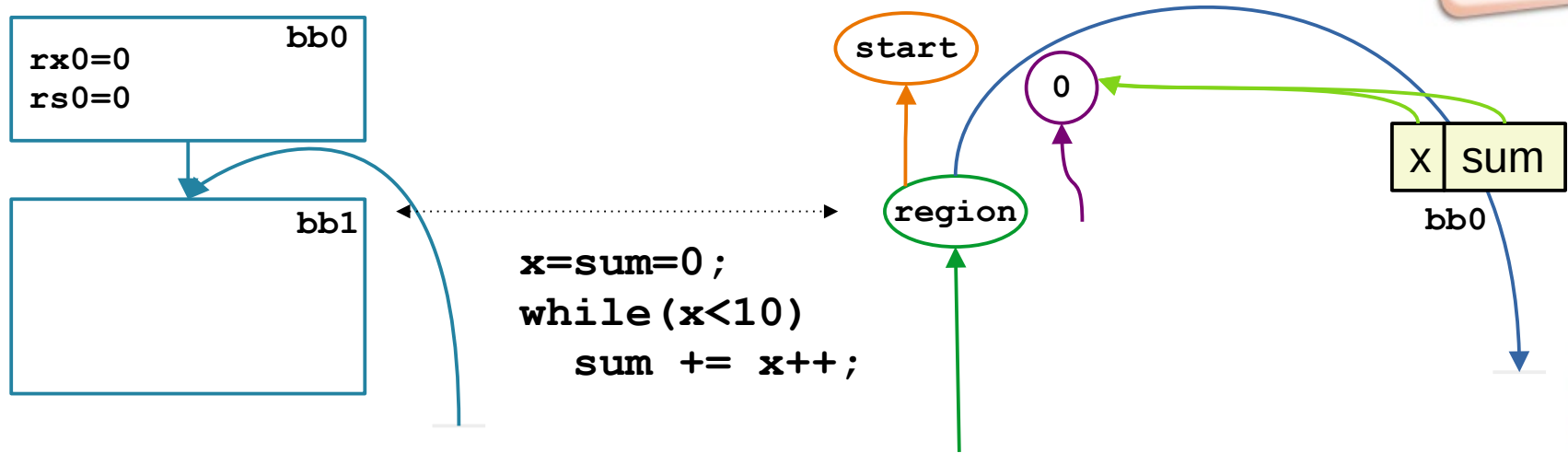
- Region comes first, otherwise same as before



```
x=sum=0;  
while(x<10)  
    sum += x++;
```

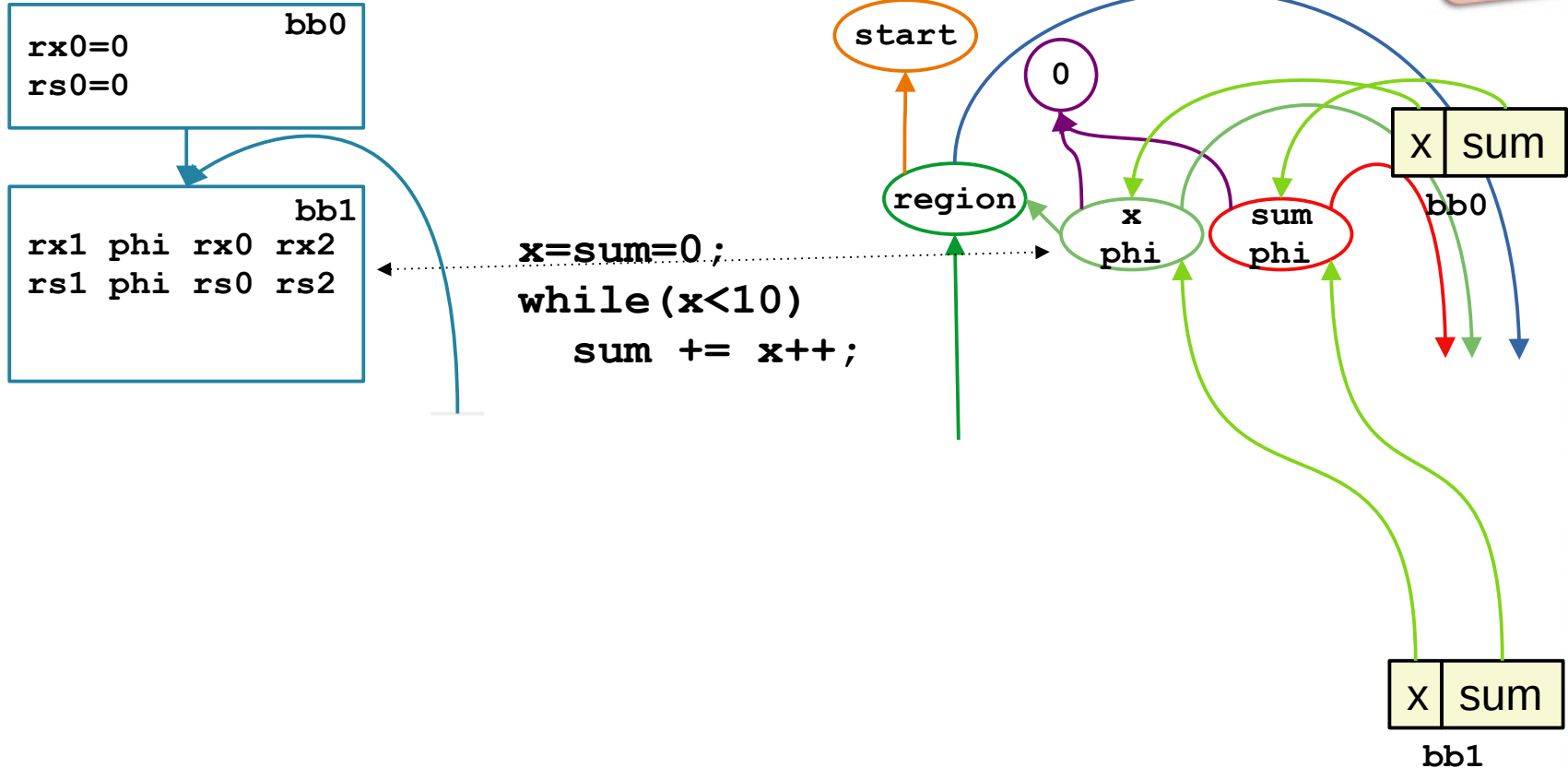
Control Flow Merge - Loops

- Region comes first, otherwise same as before

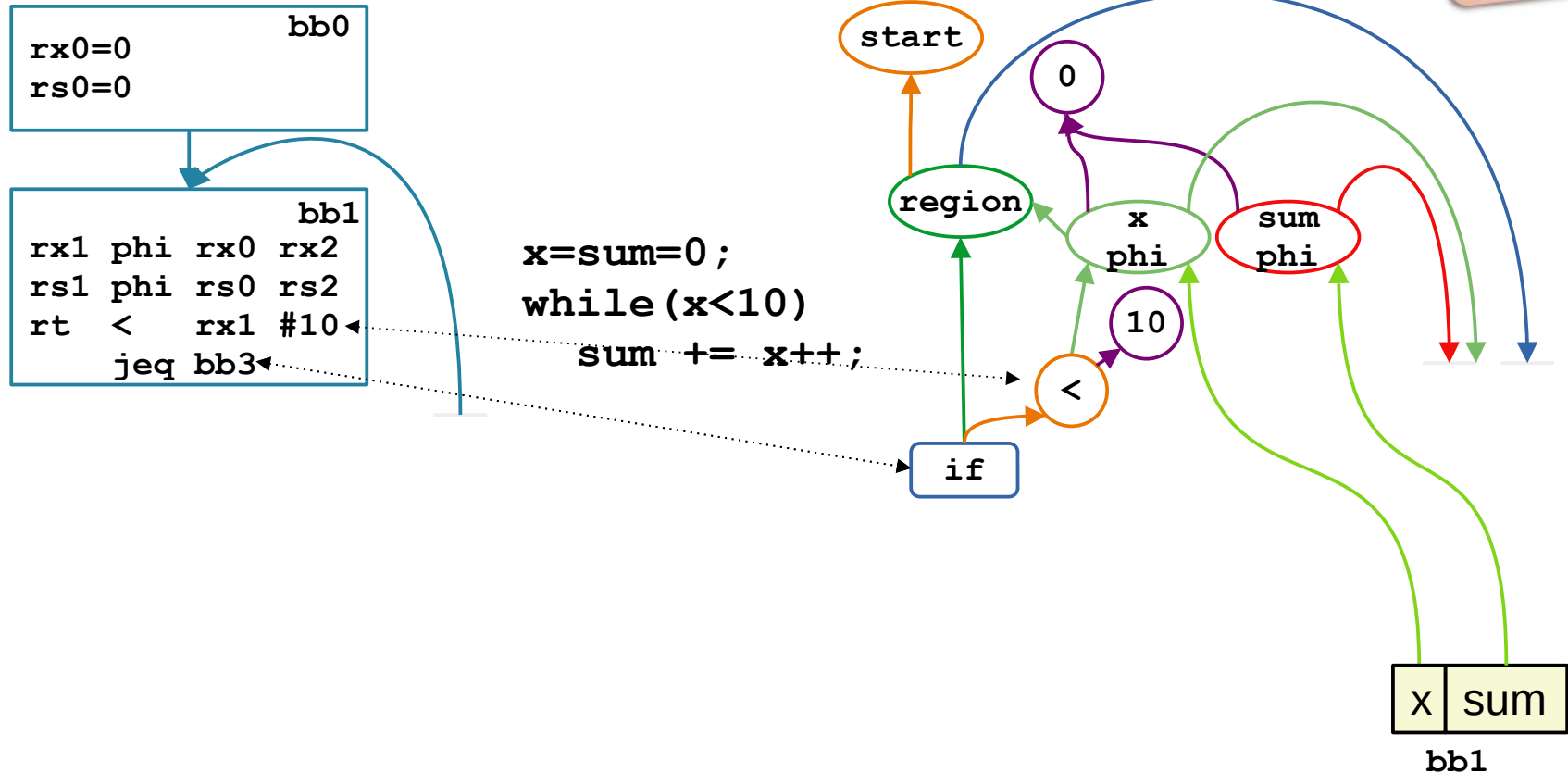


Control Flow Merge - Loops

- Region comes first, otherwise same as before

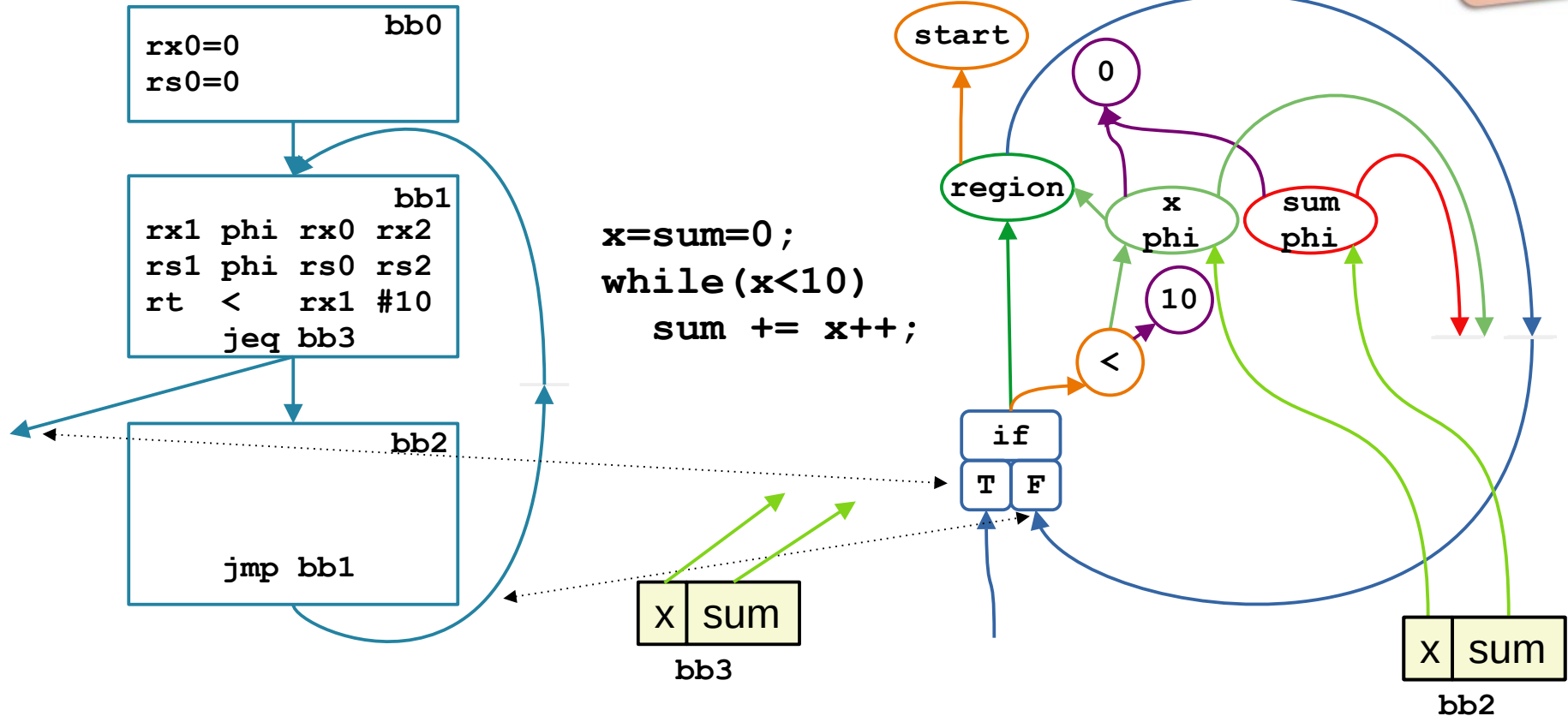


- Filling in basic block 'bb1'



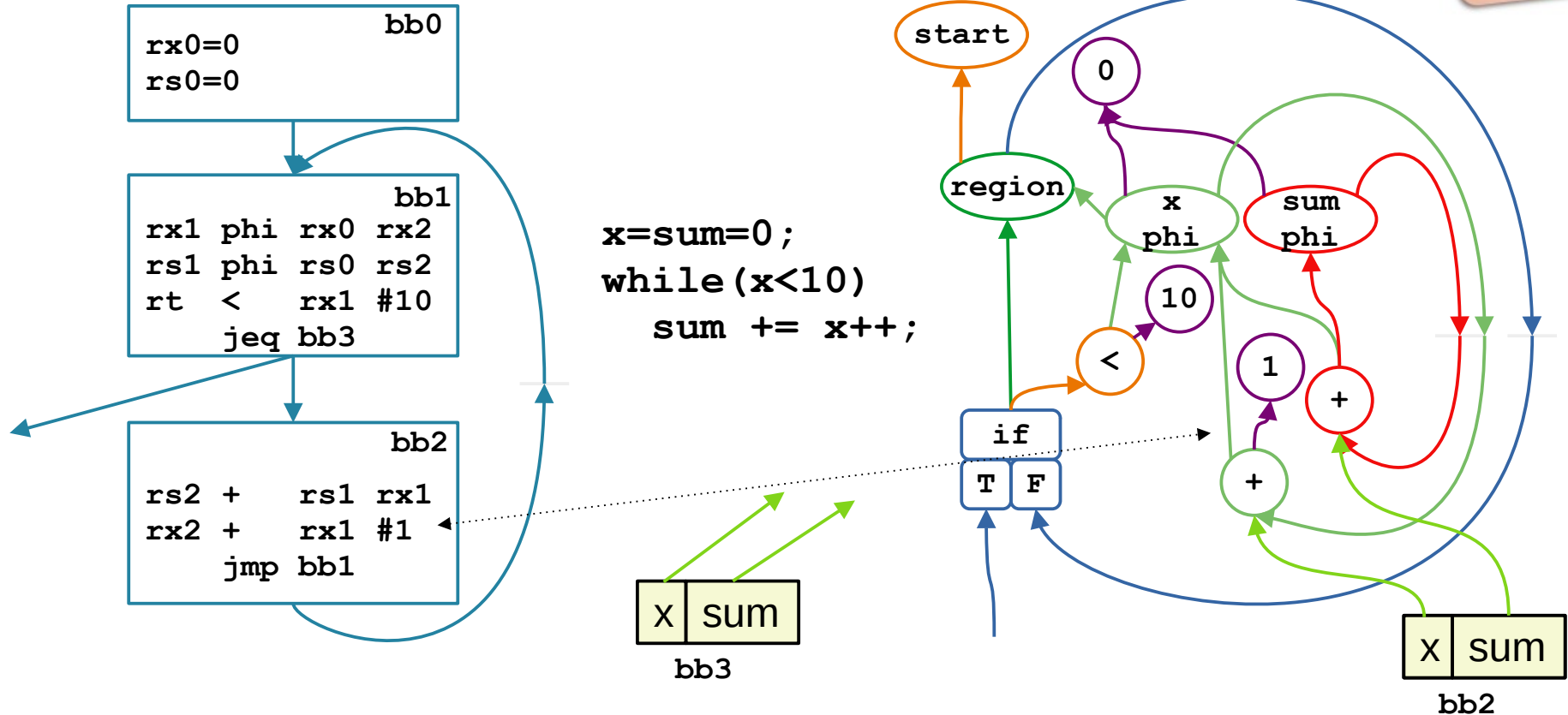
Control Flow Merge - Loops

- Hollow shell of 'bb2'



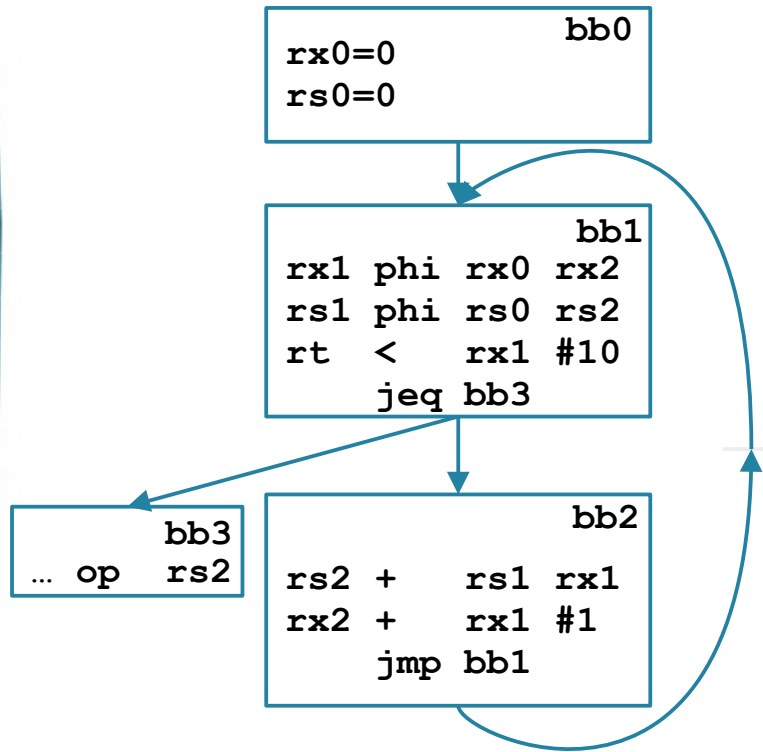
Control Flow Merge - Loops

- Filling in 'bb2'

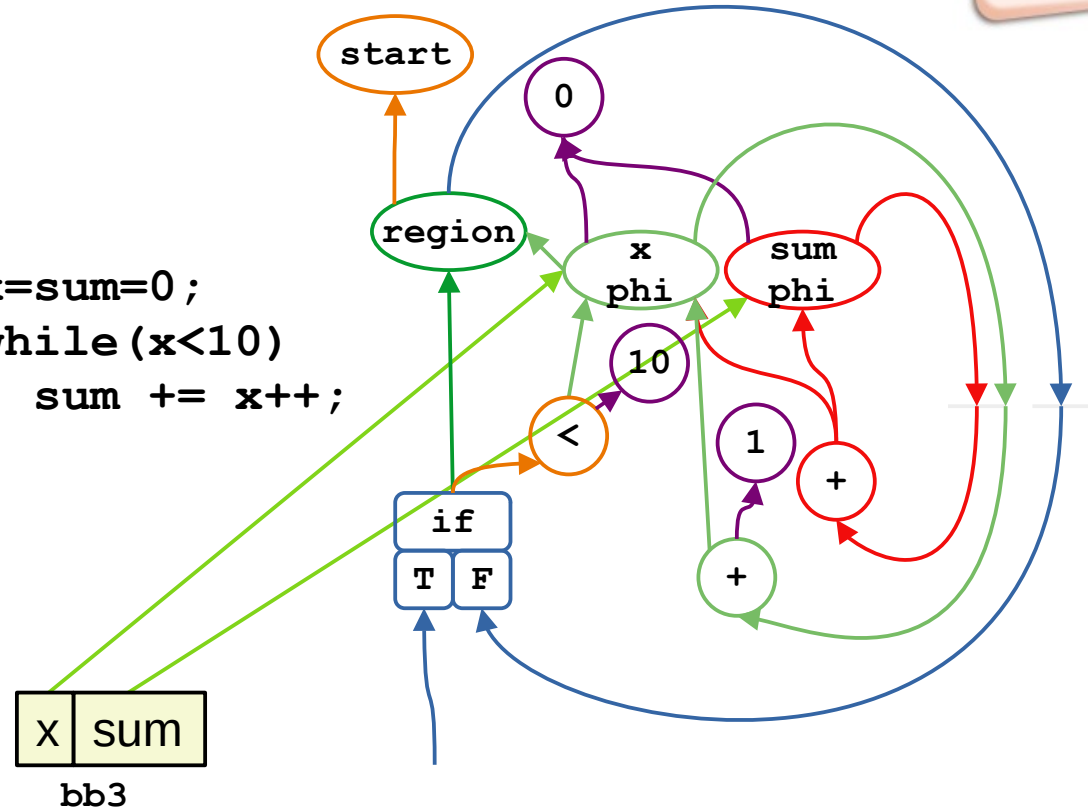


Control Flow Merge - Loops

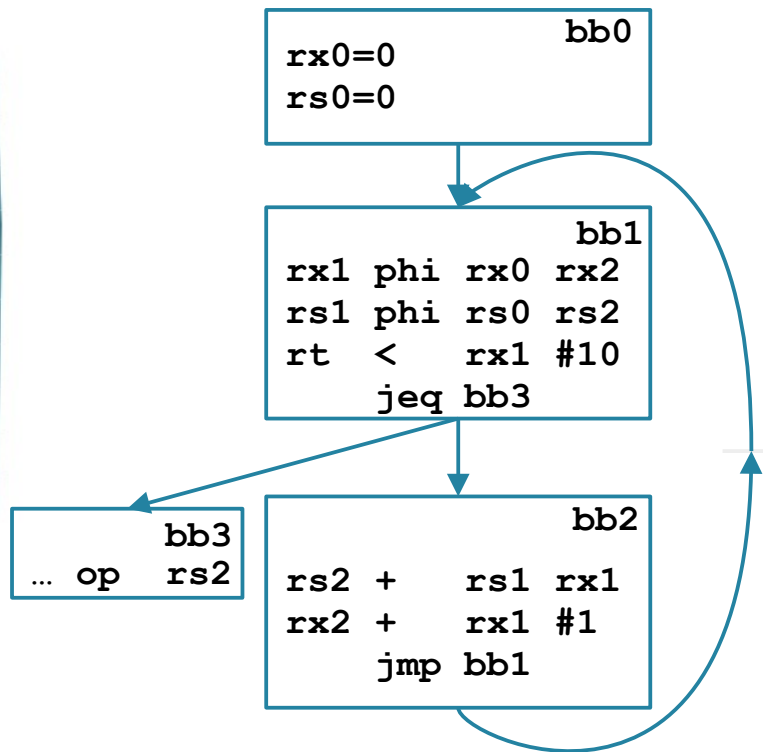
- Starting 'bb3'



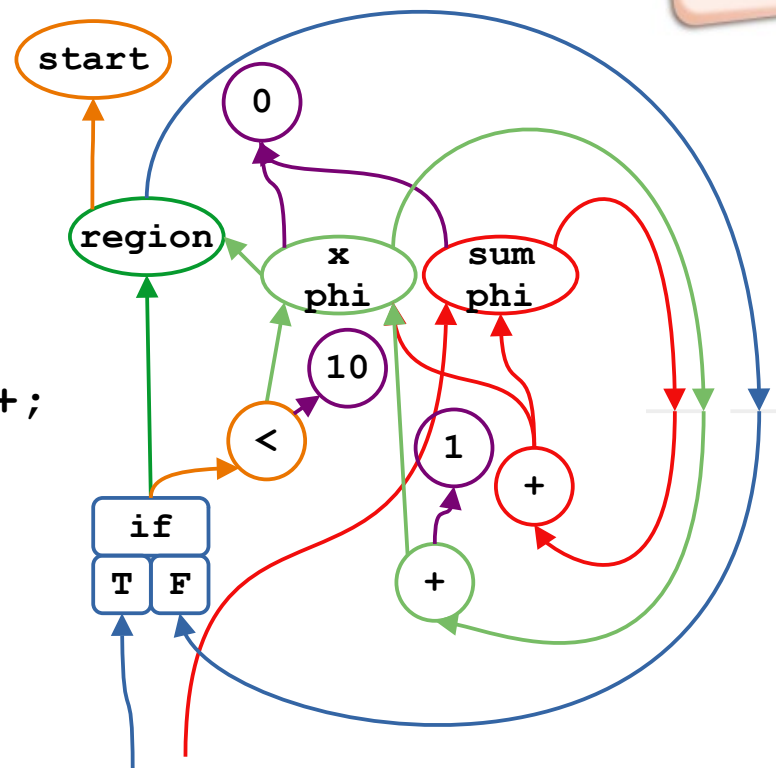
`x=sum=0;`
`while(x<10)`
`sum += x++;`



- Starting 'bb3'

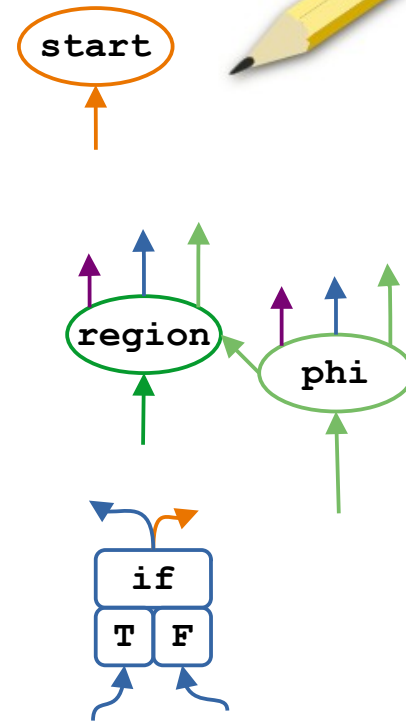


```
x=sum=0;
while (x<10)
    sum += x++;
```



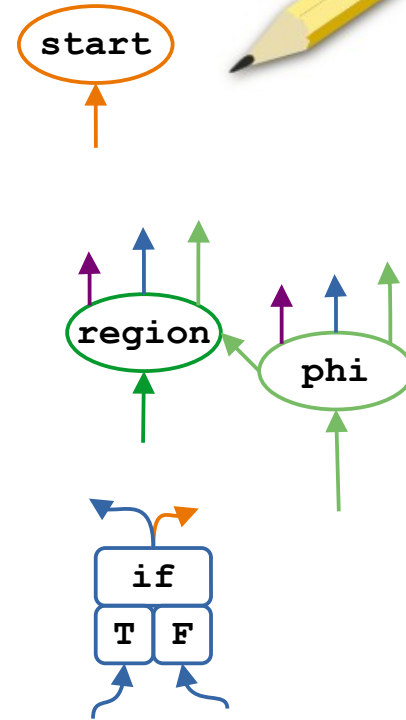
Graph Control Flow

- **start** first basic block
- **region** merges two or more CFG edges
 - Always 1 control user
 - Zero or many Phi users
 - Each phi has data-per-ctrl edge
- **if** splits control two ways
 - One control input, one data input
 - **T** **F** labels the different control outputs

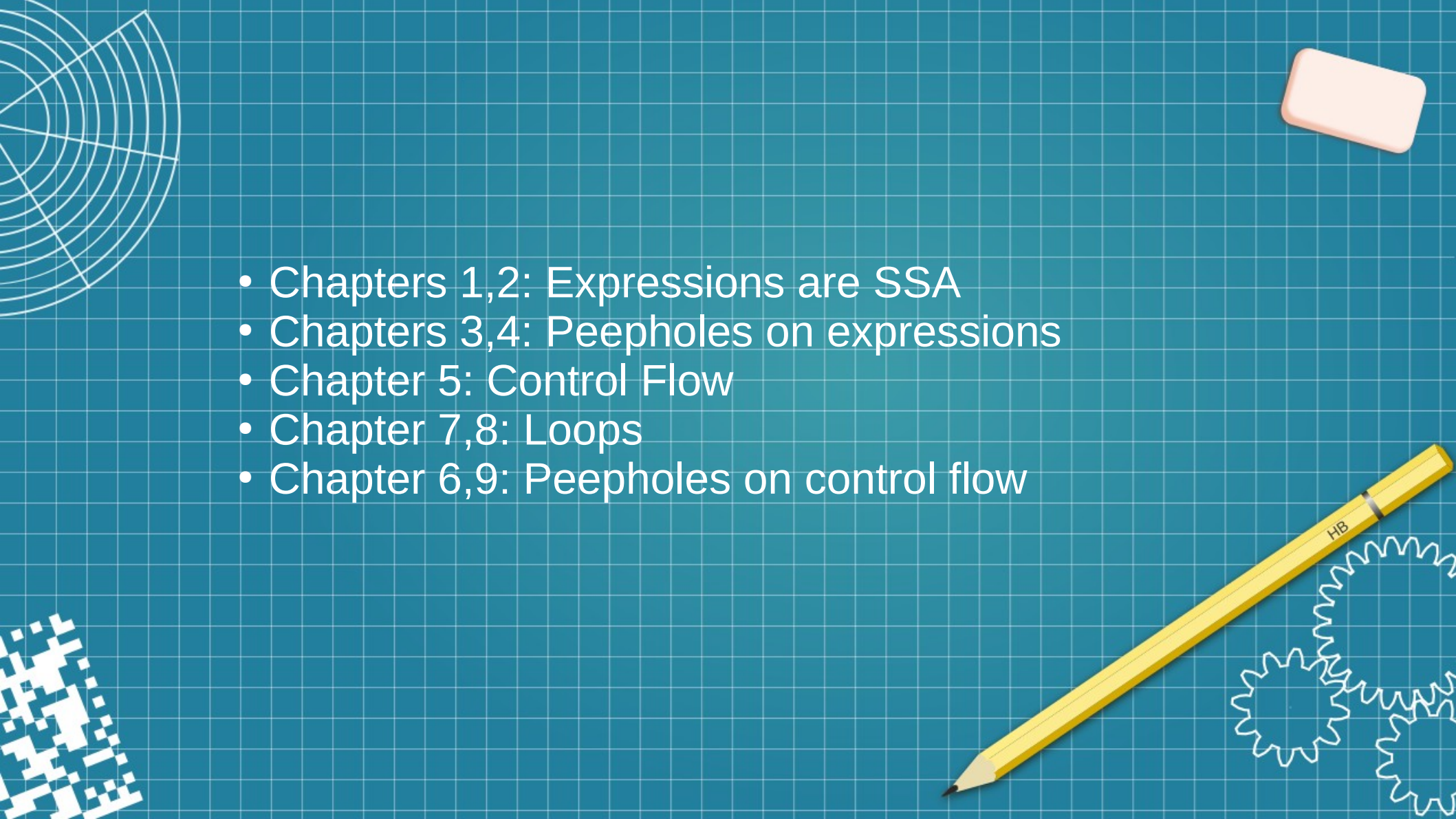


Graph Control Flow

- Can always trace the CFG outlines
 - Every edge is either control or data
- CFG nodes also Peephole like normal
 - The peephole worklist can't tell, doesn't care
 - Dead code elim and constant folding work the same
- Hey, let's Peephole some Control Flow!

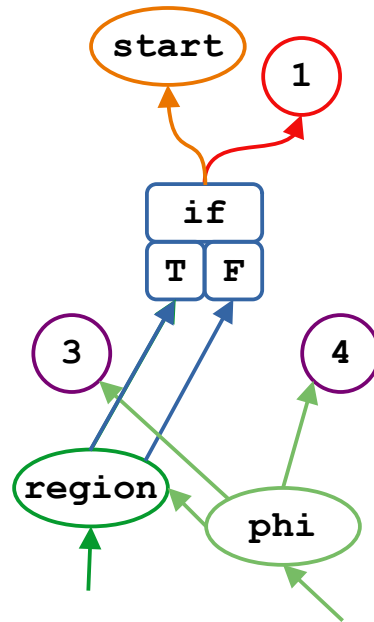


```
tmp=true; if( tmp ) x=3 else x=4
```

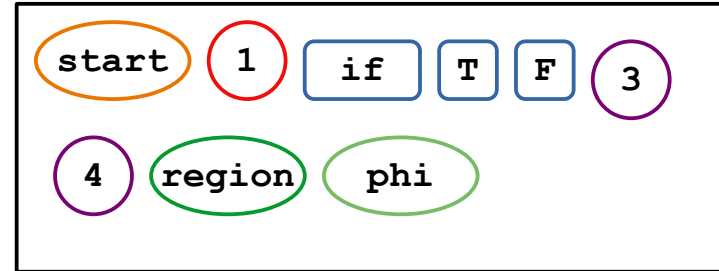
- 
- Chapters 1,2: Expressions are SSA
 - Chapters 3,4: Peephholes on expressions
 - Chapter 5: Control Flow
 - Chapter 7,8: Loops
 - Chapter 6,9: Peephholes on control flow

Peephole Control Flow

- Assume `tmp` is true: 1 `tmp=true; if( tmp ) x=3 else x=4`

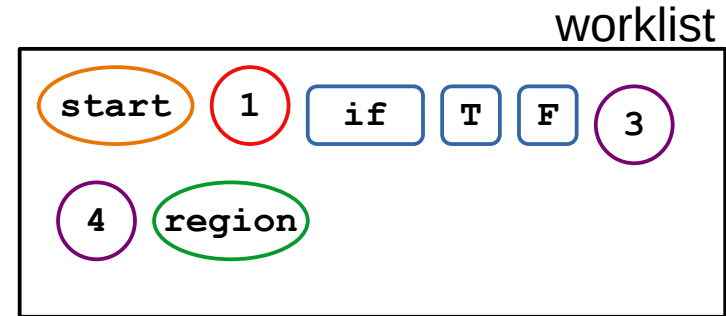
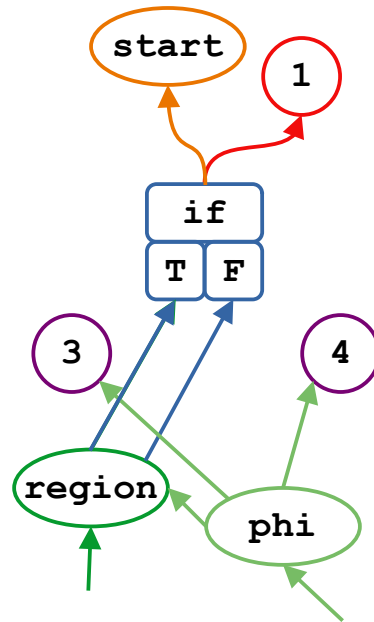


worklist



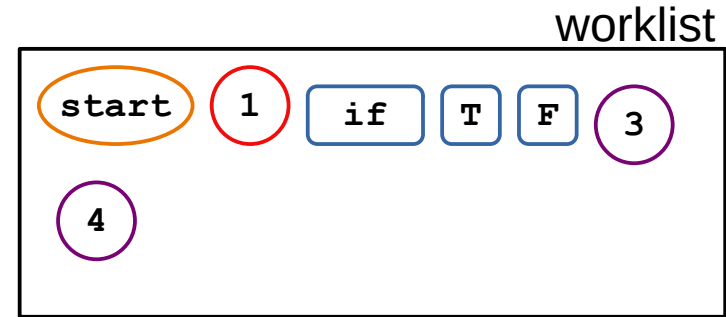
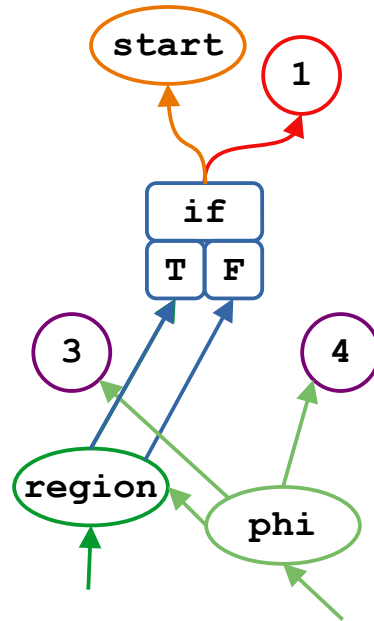
Peephole Control Flow

- Pop `phi`, not dead, no CSE.
- Call ideal, no progress, done.



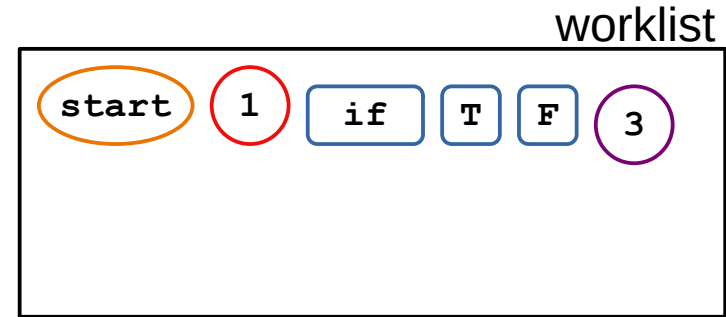
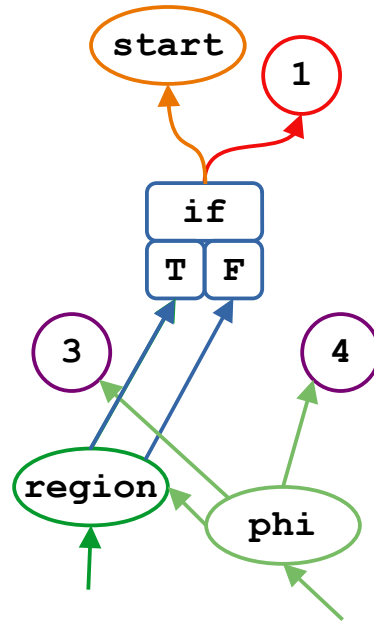
Peephole Control Flow

- Pop region, not dead, no CSE.
- Call ideal, no progress, done.



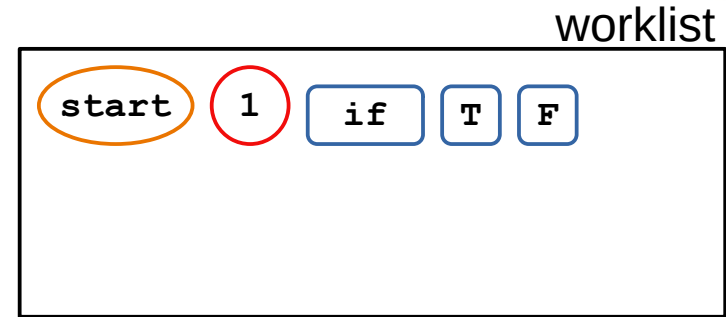
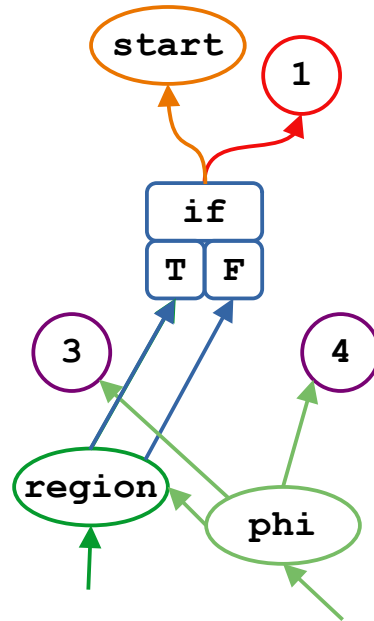
Peephole Control Flow

- Pop ④ , not dead, no CSE.
- Call ideal, no progress, done.



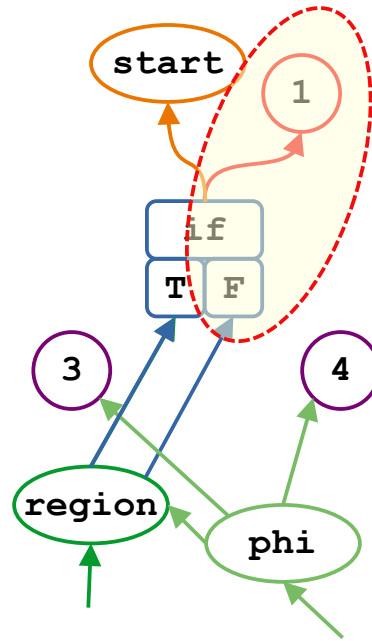
Peephole Control Flow

- Pop ③ , not dead, no CSE.
- Call ideal, no progress, done.

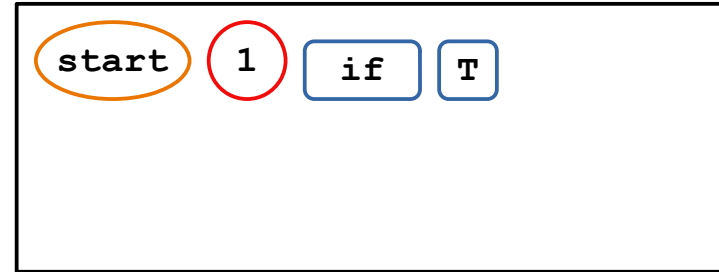


Peephole Control Flow

- Pop `F` , not dead, no CSE.
- Call ideal
 - `if` Predicate is true

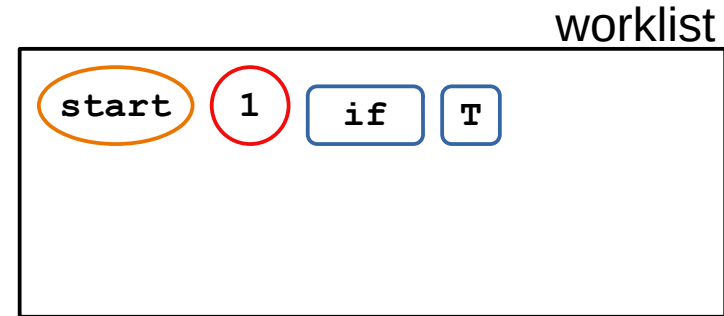
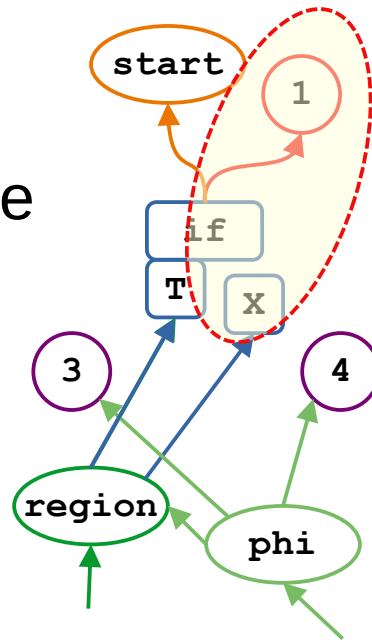


worklist



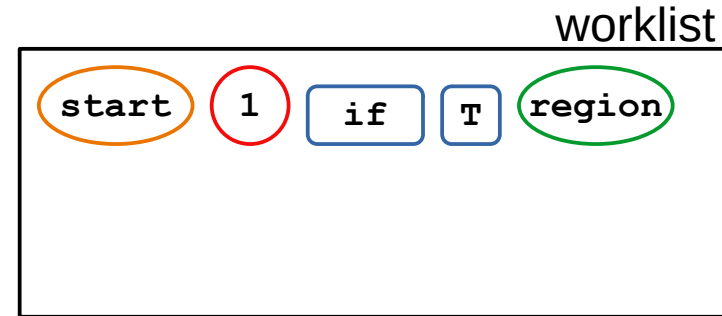
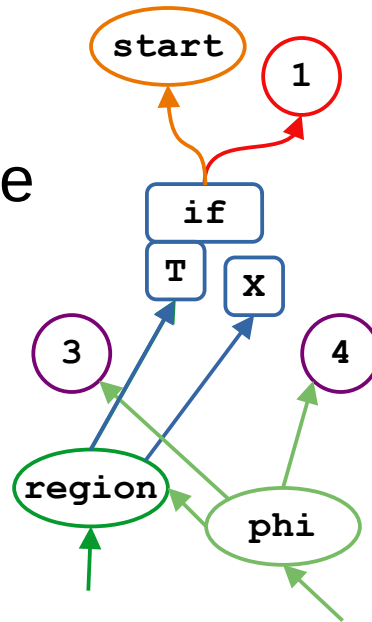
Peephole Control Flow

- Pop `F` , not dead, no CSE.
- Call ideal
 - `if` Predicate is true
 - become `x` the unreachable control



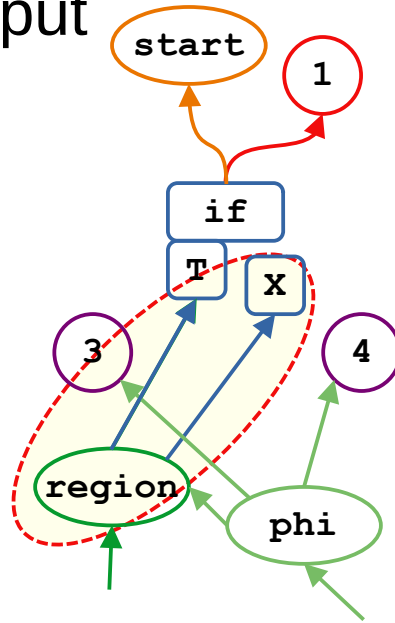
Peephole Control Flow

- Pop `F` , not dead, no CSE.
- Call ideal
 - `if` Predicate is true
 - become `x` the unreachable control
 - Neighbors on list

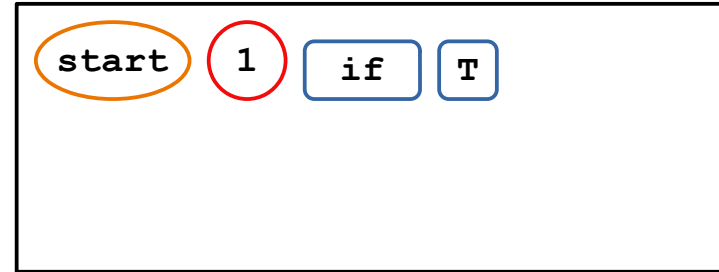


Peephole Control Flow

- Pop region, not dead, no CSE.
- Call ideal
 - Only 1 live input

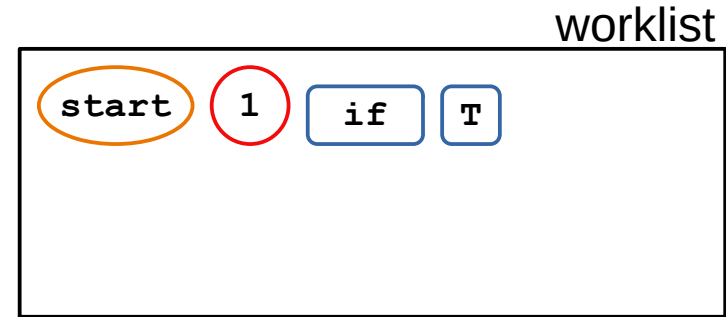
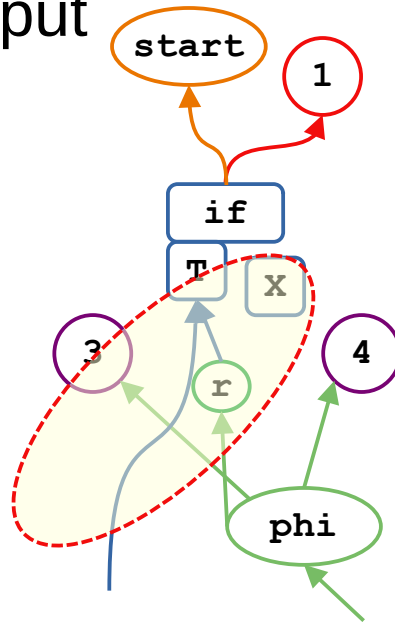


worklist



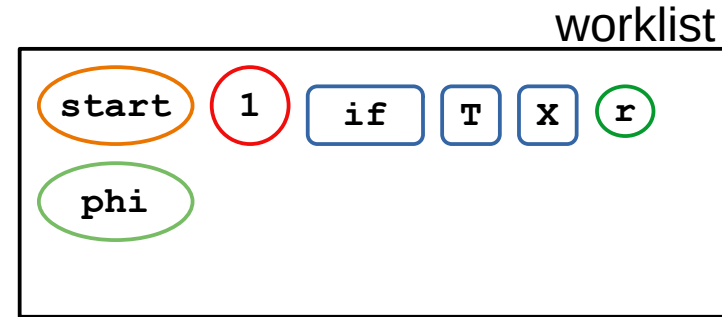
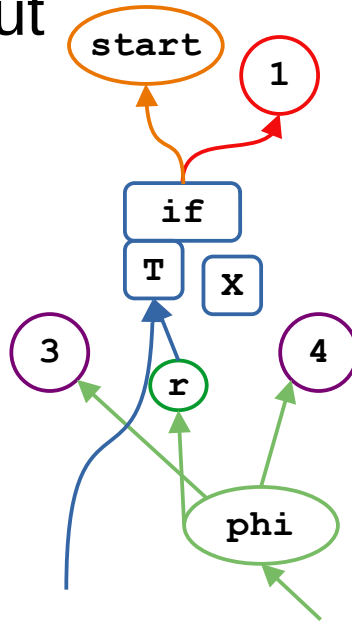
Peephole Control Flow

- Pop region, not dead, no CSE.
- Call ideal
 - Only 1 live input
 - Become it



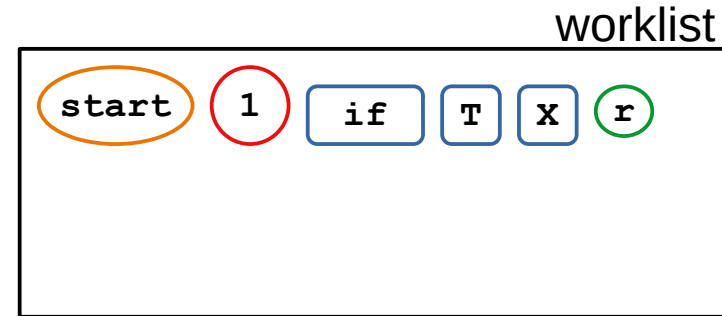
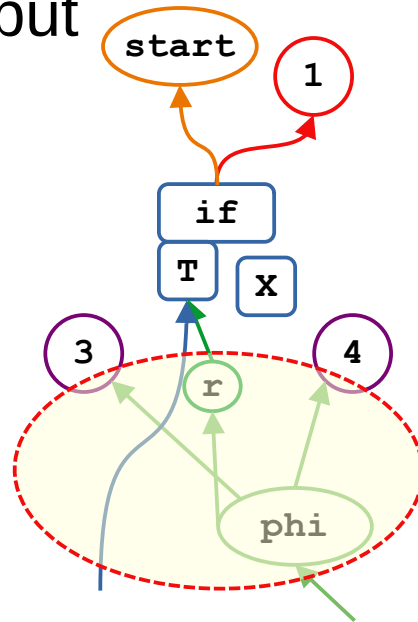
Peephole Control Flow

- Pop region, not dead, no CSE.
- Call ideal
 - Only 1 live input
 - Become it
 - Neighbors on list



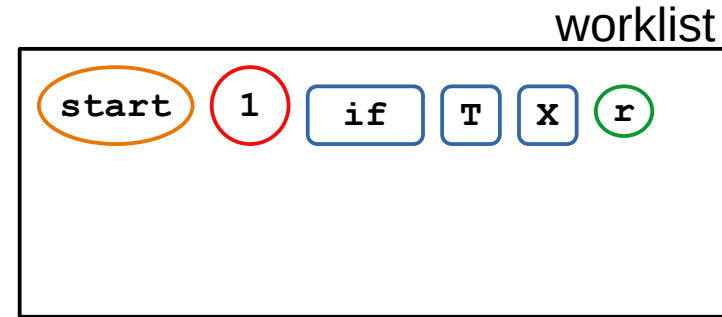
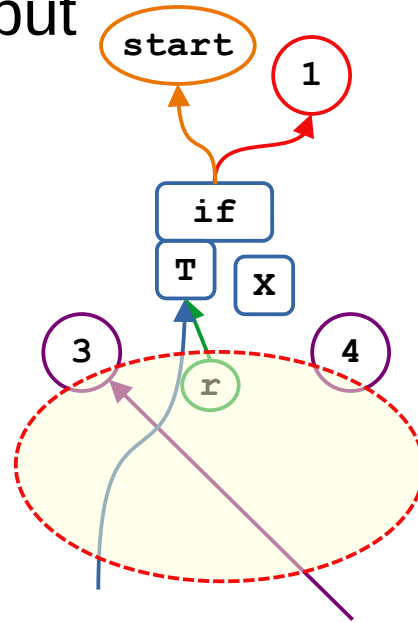
Peephole Control Flow

- Pop phi, not dead, no CSE.
- Call ideal
 - Only 1 live input on r



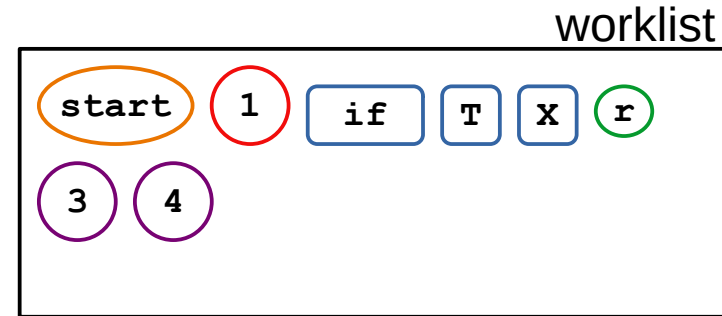
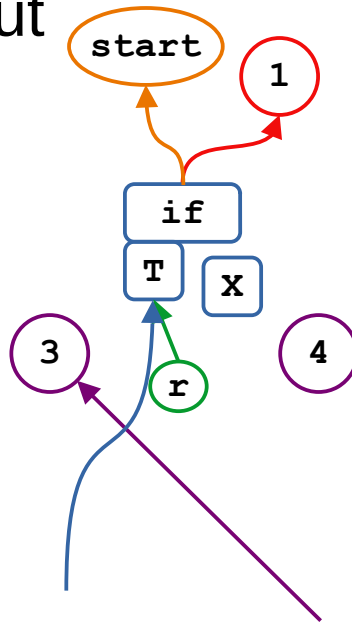
Peephole Control Flow

- Pop phi, not dead, no CSE.
- Call ideal
 - Only 1 live input on r
 - Become it



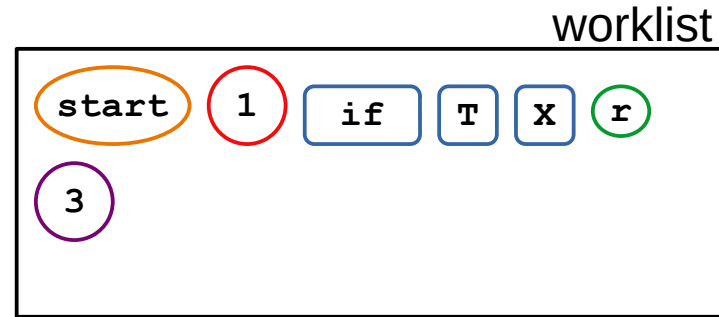
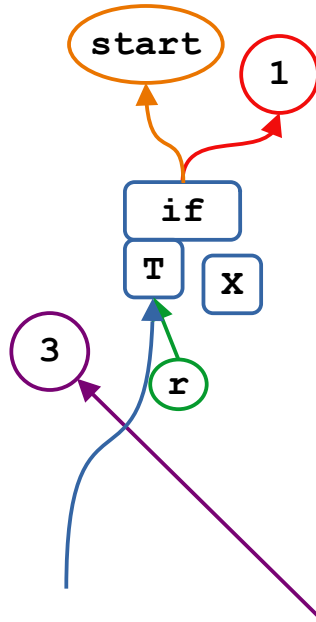
Peephole Control Flow

- Pop phi, not dead, no CSE.
- Call ideal
 - Only 1 live input on r
 - Become it
 - Neighbors on list



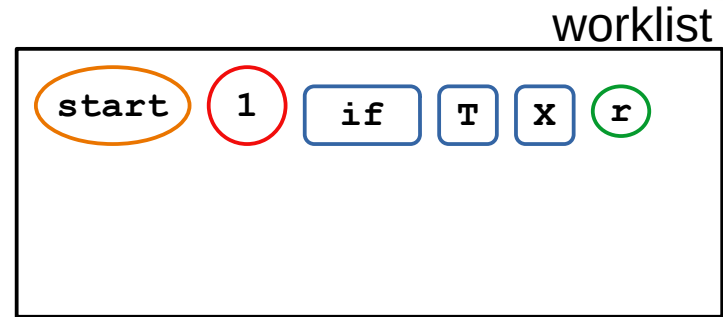
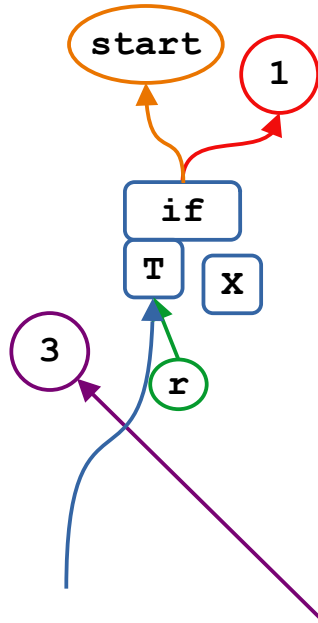
Peephole Control Flow

- Pop 4 , dead
- Neighbors on list



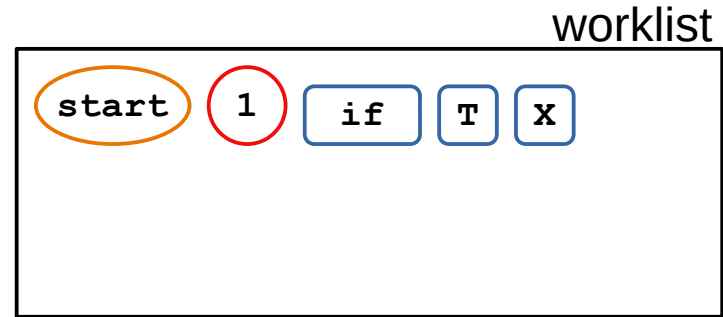
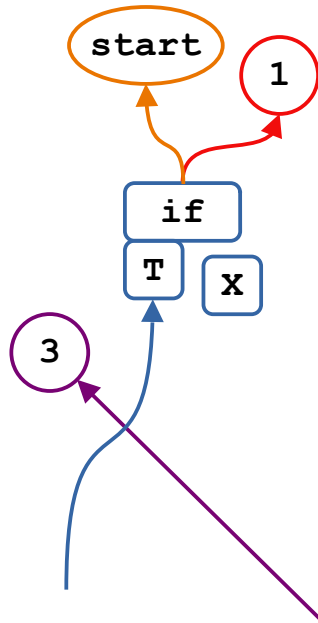
Peephole Control Flow

- Pop 3, not dead, no CSE
- Call ideal, no progress, done.



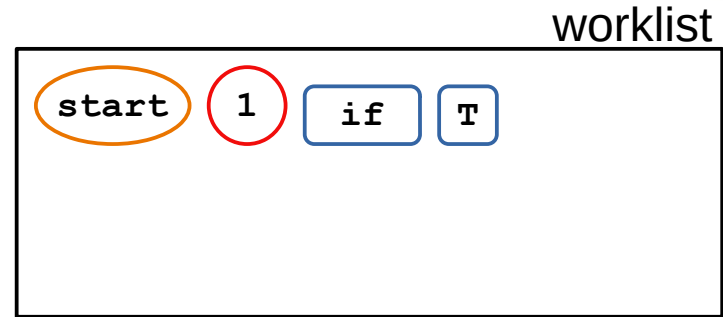
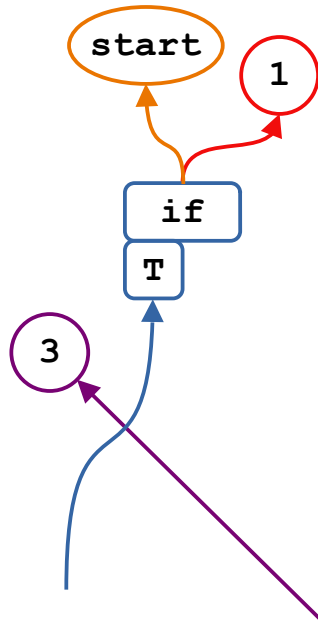
Peephole Control Flow

- Pop $\textcircled{r}$, dead
- Neighbors on list



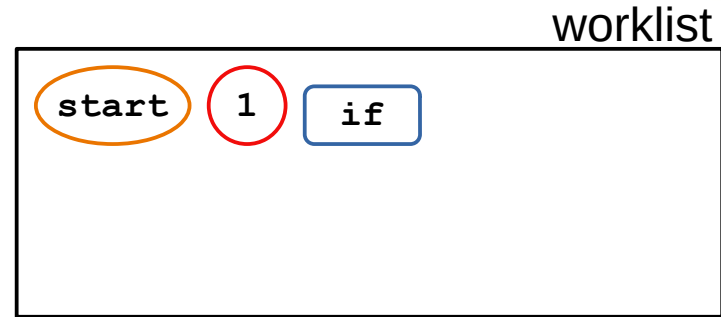
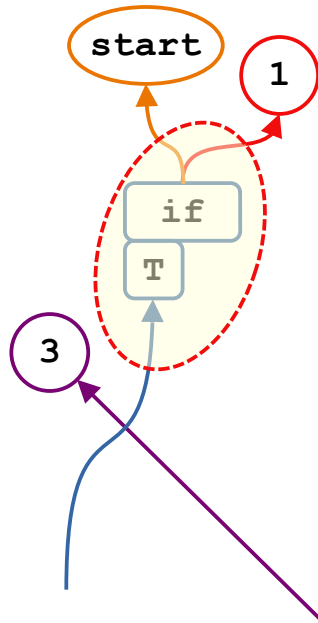
Peephole Control Flow

- Pop x , dead
- Neighbors on list



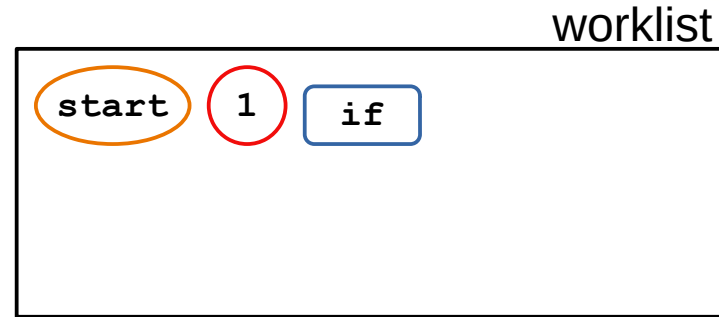
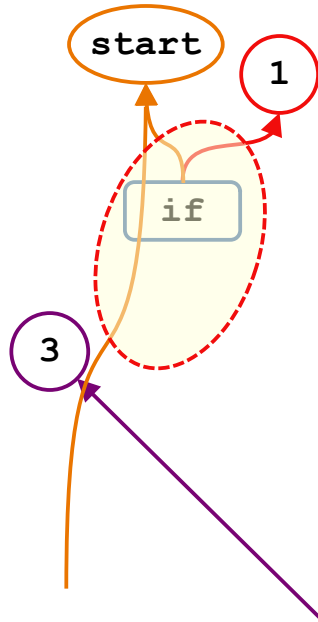
Peephole Control Flow

- Pop `if` `T` , not dead, no CSE.
- Call ideal
 - `if` Predicate is true



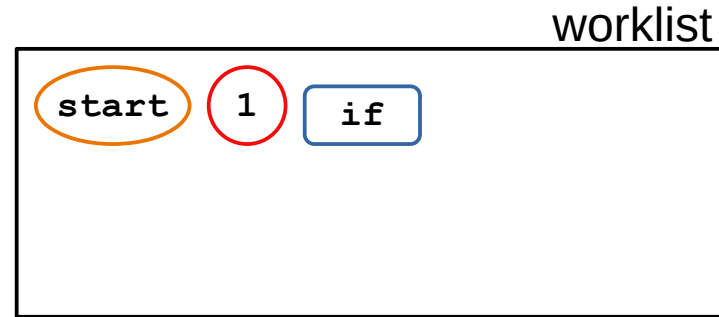
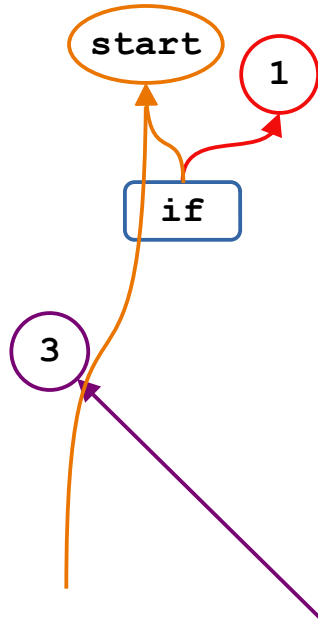
Peephole Control Flow

- Pop $\boxed{T}$, not dead, no CSE.
- Call ideal
 - $\boxed{\text{if}}$ Predicate is true
 - Become $\boxed{\text{if}}$ control



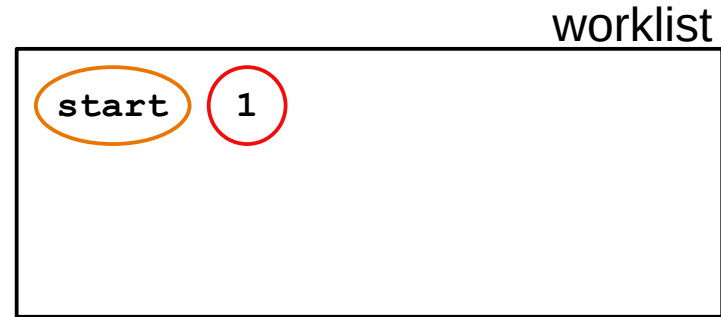
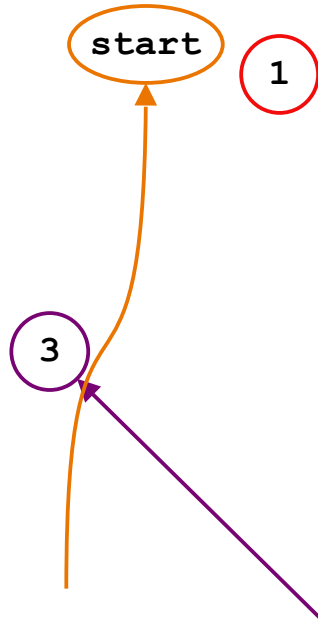
Peephole Control Flow

- Pop $\boxed{T}$, not dead, no CSE.
- Call ideal
 - $\boxed{\text{if}}$ Predicate is true
 - Become $\boxed{\text{if}}$ control
 - Neighbors on list



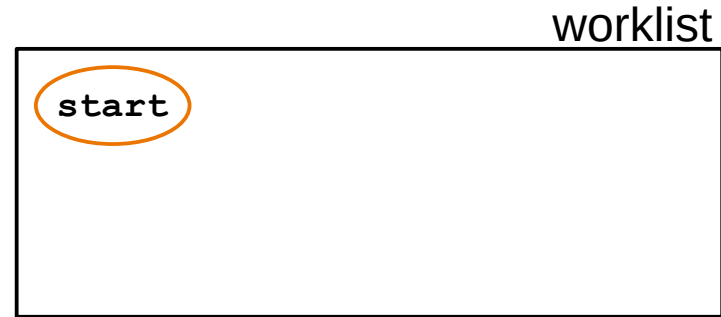
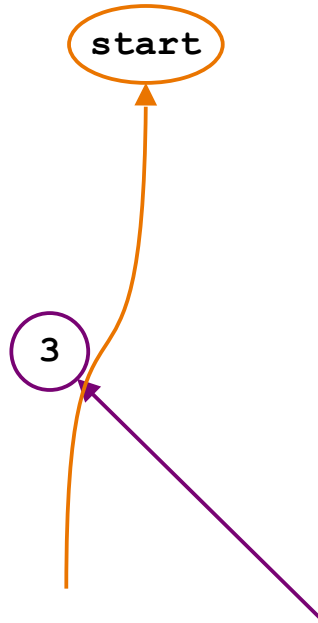
Peephole Control Flow

- Pop `if`, dead.
- Neighbors on list



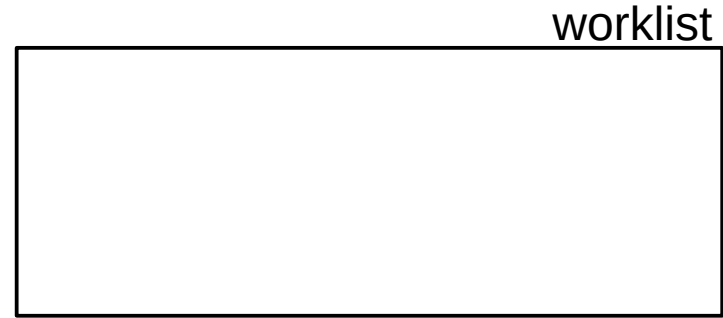
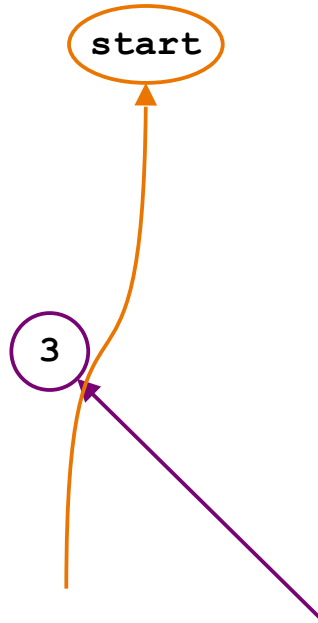
Peephole Control Flow

- Pop ① , dead.
- Neighbors on list



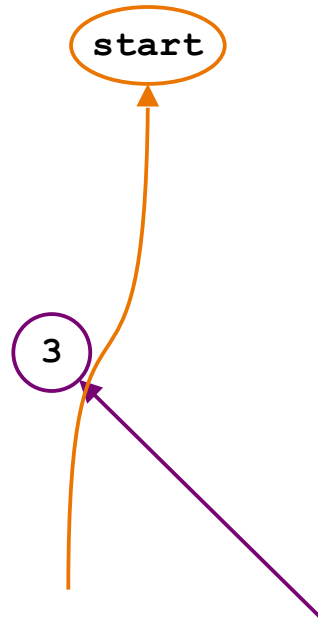
Peephole Control Flow

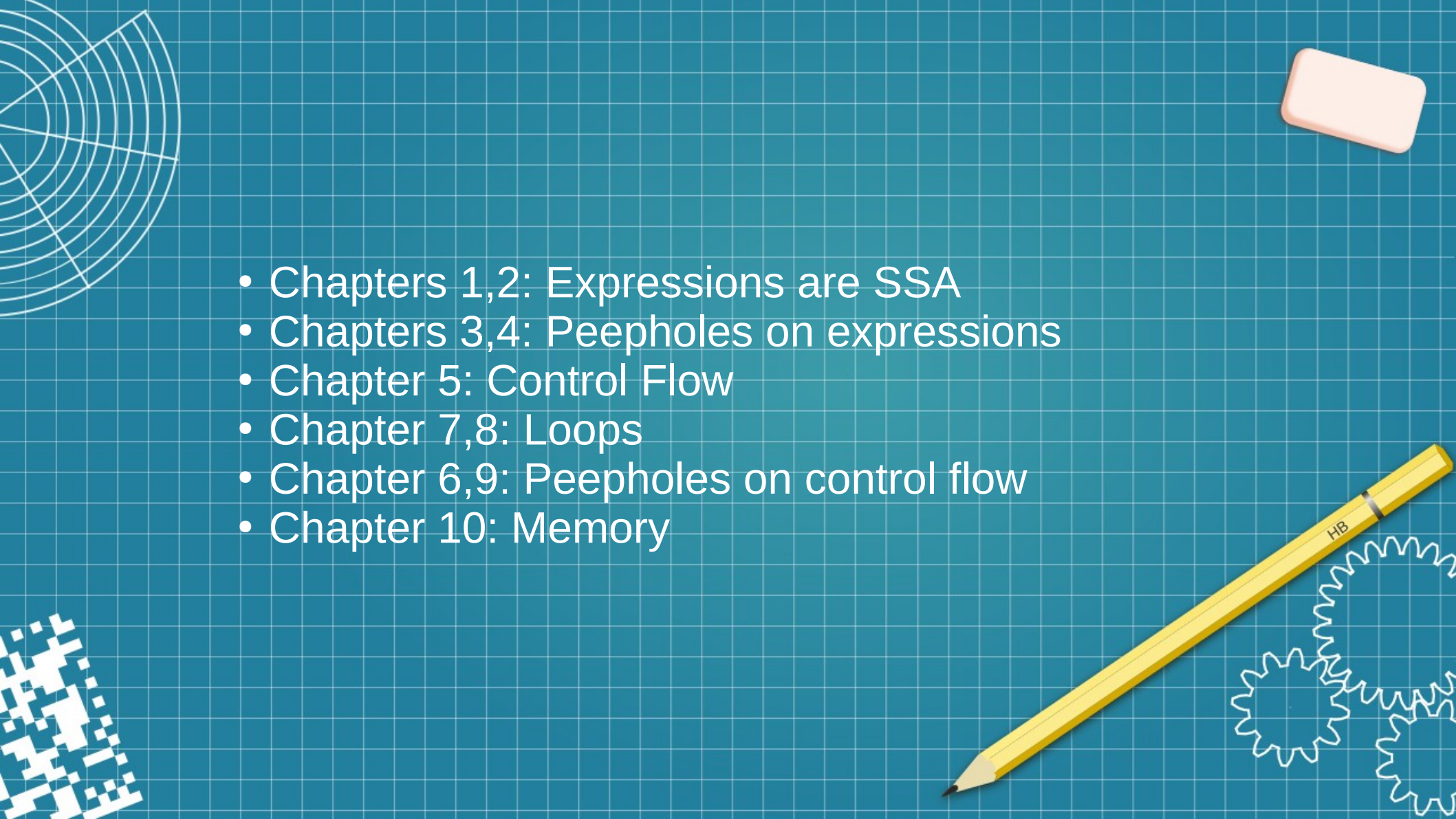
- Pop start, not dead, no CSE.
- Call ideal, no progress, done.



Peephole Control Flow

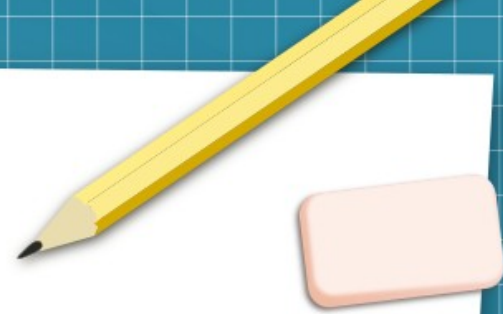
- Peepholes done



- 
- Chapters 1,2: Expressions are SSA
 - Chapters 3,4: Peephholes on expressions
 - Chapter 5: Control Flow
 - Chapter 7,8: Loops
 - Chapter 6,9: Peephholes on control flow
 - Chapter 10: Memory

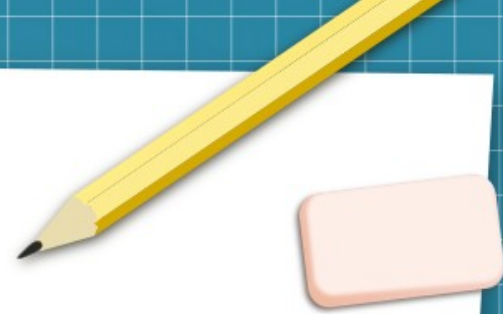
Memory

- SSA-based Memory
 - Really, nothing to do with the Sea-of-Nodes
- Memory is treated as a **value**, like 3.14 or “abc”.
 - Its a Big Fat value, with lots of internal structure
- `start` produces Memory
- Produce and consume Memory:
 - `call` `phi` `st`
- Loads `ld` only consume memory



Memory

- Peephole rules work just the same
 - ϕ rules are unchanged
 - st and ld have some memory-specific ones
 - Support back-to-back ld/st or st/st elimination
 - Support CSE / value-numbering
- **Precision** is based on alias information
 - Correct to use a single Memory value, but very imprecise
- **Equivalence-class** aliasing is much more precise
 - AND can be directly represented in the graph
 - AND comes for free from strongly typed languages

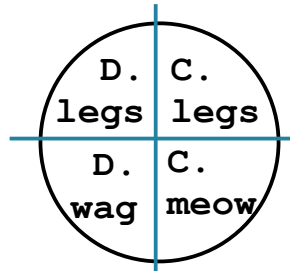
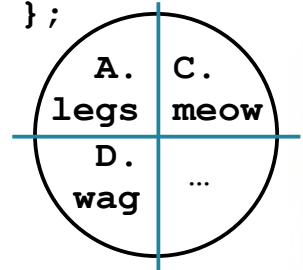


Equivalence Class Aliasing

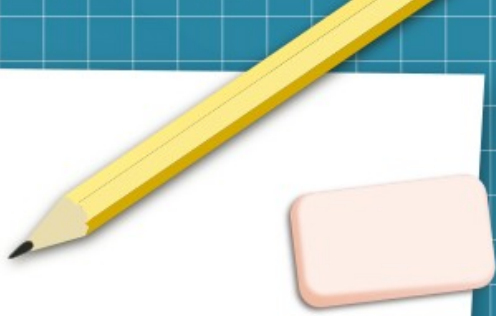
- Memory is broken up into *equivalence classes*
- Things in the same equivalence classes **always** alias
- Things in different equivalence classes **never** alias
- Example:

```
class Animal { int legs };  
class Cat extends Animal { float meow_vol; };  
class Dog extends Animal { float wag_rate; };
```

- Fields `legs`, `meow_vol`, `wag_rate` all different eqvs
- `{cat.legs, dog.legs}` DO alias...
- unless `Animal` is abstract



Example: Loop with Iterator



- What you wrote:

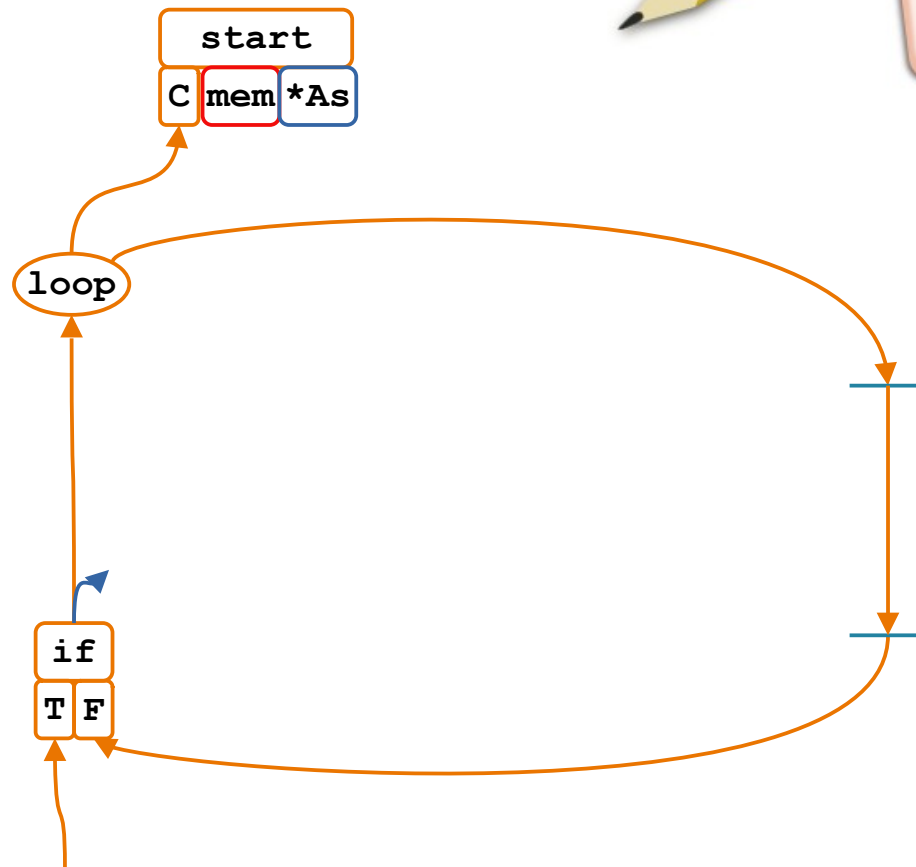
```
int[] As=...;  
for( int i : As )  
    sum += i;
```

- What you got:

```
int[] As=...;  
Iter it=new Iter();  
while( it.x < As.length ) {  
    int i = As[it.x++];  
    sum += i;  
}
```

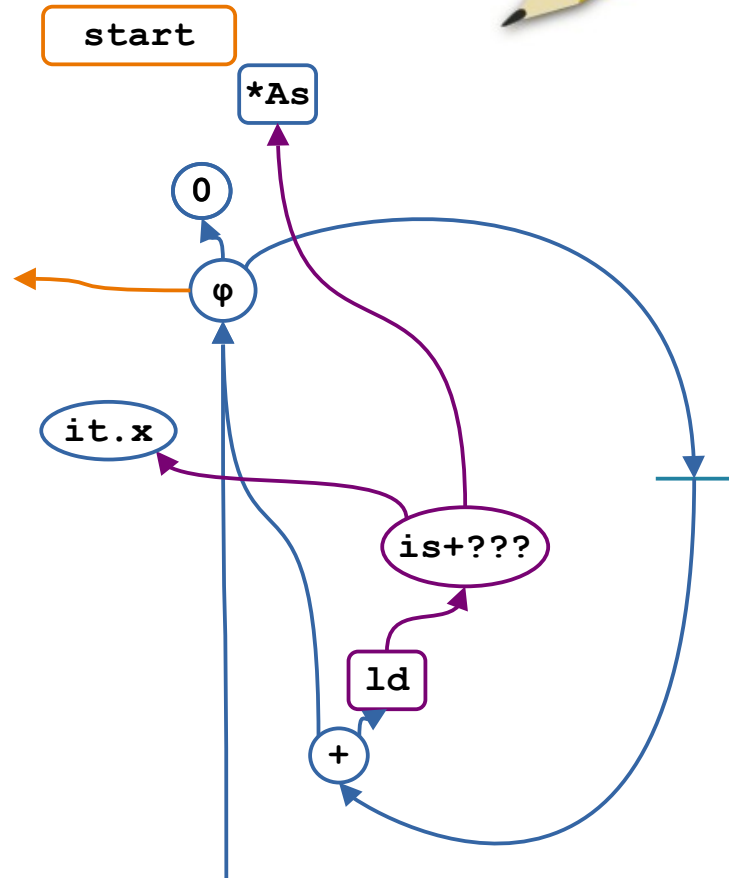
Example: the CFG

```
int[] As=...;  
Iter it=new Iter();  
while( it.x < As.length ) {  
    int i = As[it.x++];  
    sum += i;  
}
```



Example: Data flow

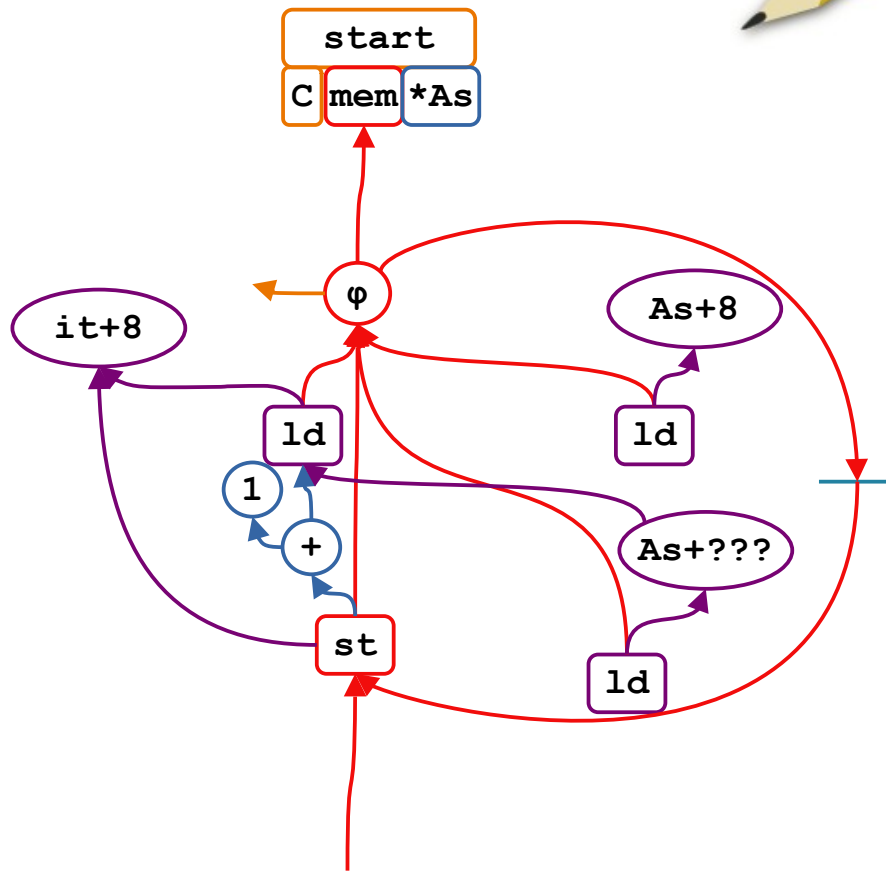
```
int[] As=...;  
Iter it=new Iter();  
while( it.x < As.length ) {  
    int i = As[it.x++];  
    sum += i;  
}
```



ing

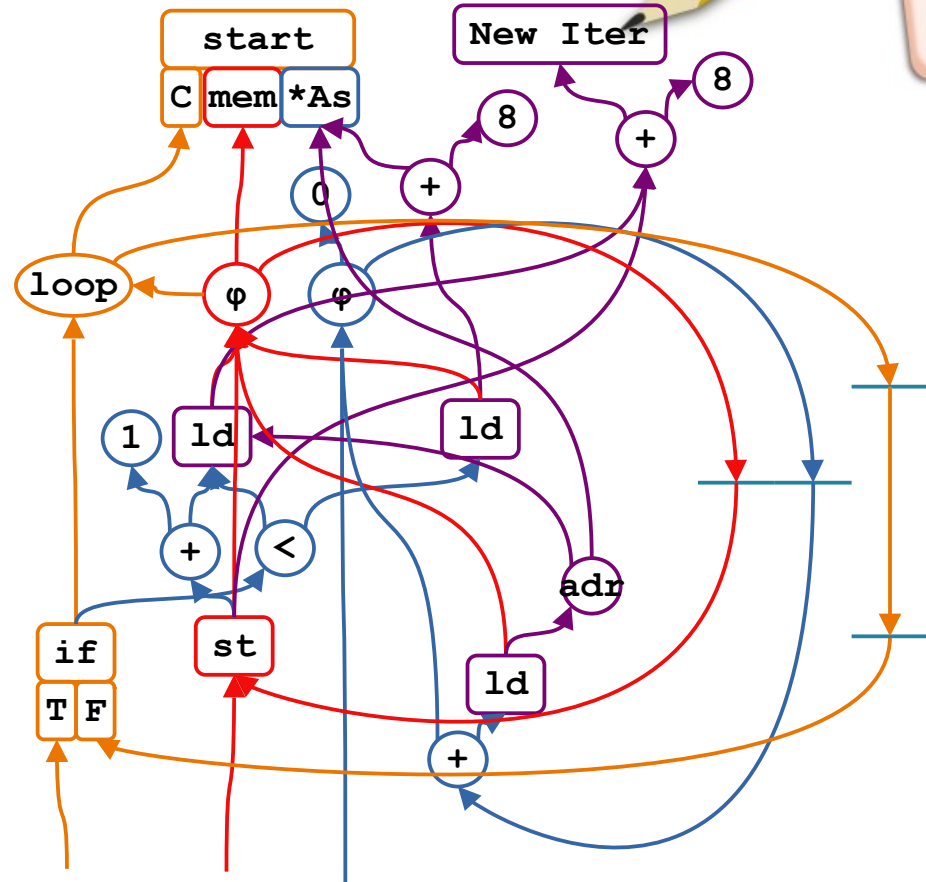
-

```
int[] As=...;
Iter it=new Iter();
while( it.x < As.length ) {
    int i = As[it.x++];
    sum += i;
}
```



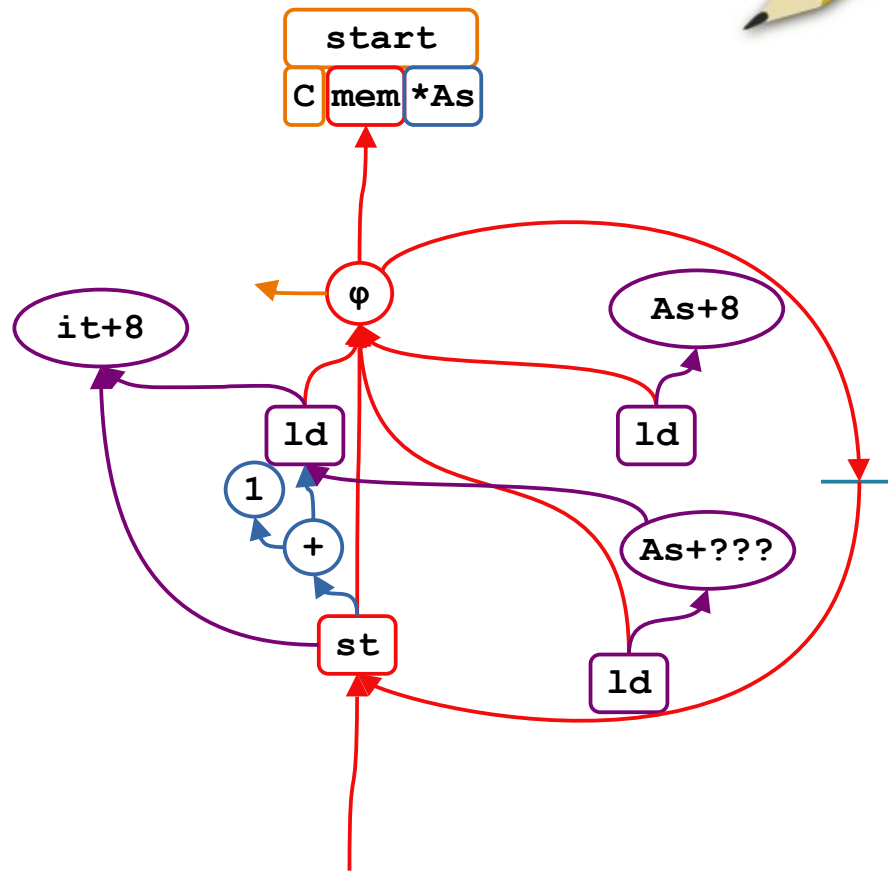
- All together
 - Hard to read

```
int[] As=...;
Iter it=new Iter();
while( it.x < As.length ) {
    int i = As[it.x++];
    sum += i;
}
```



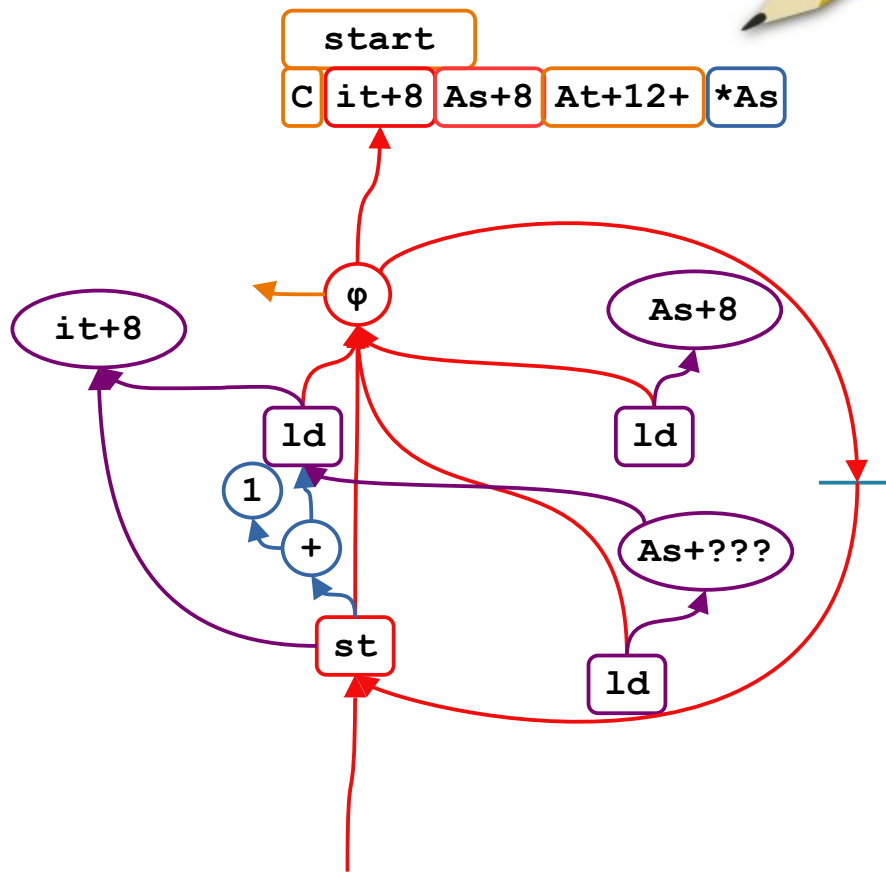
ing

```
int[] As=...;
Iter it=new Iter();
while( it.x < As.length ) {
    int i = As[it.x++];
    sum += i;
}
```



- 

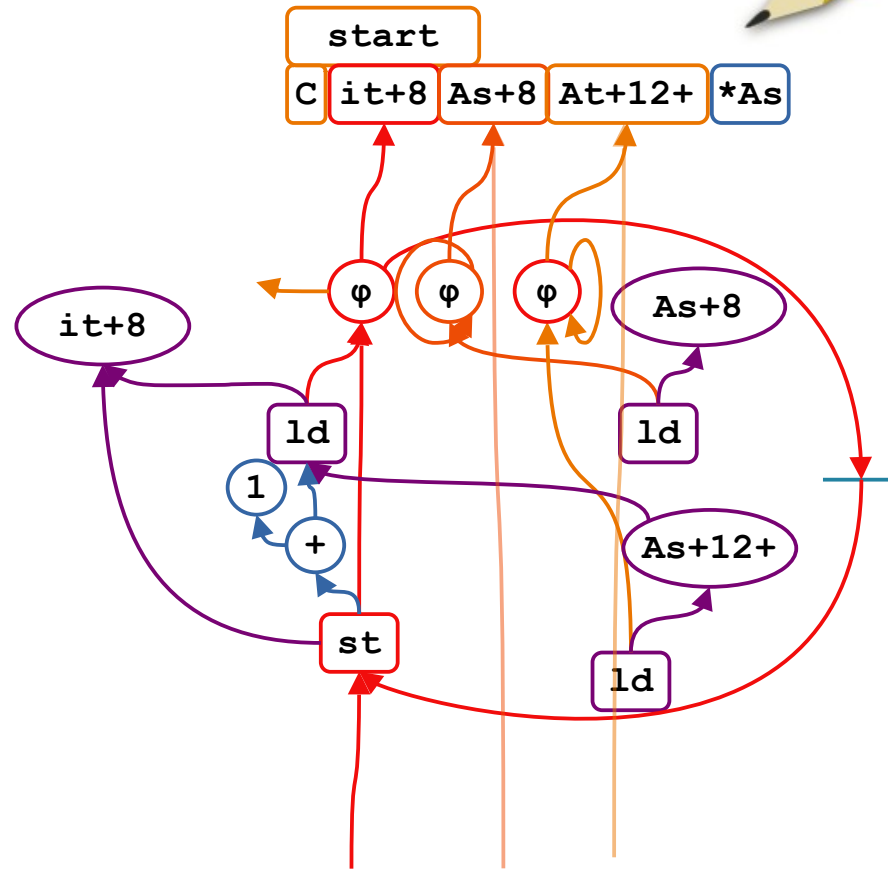
```
int[] As=...;
Iter it=new Iter();
while( it.x < As.length ) {
    int i = As[it.x++];
    sum += i;
}
```



Equivalence Classes

- But `Iter+8` will never alias with `As`; and
- `As+8` will never alias with `As+12+offset`
- Equivalence classes!
 - Separate memory!

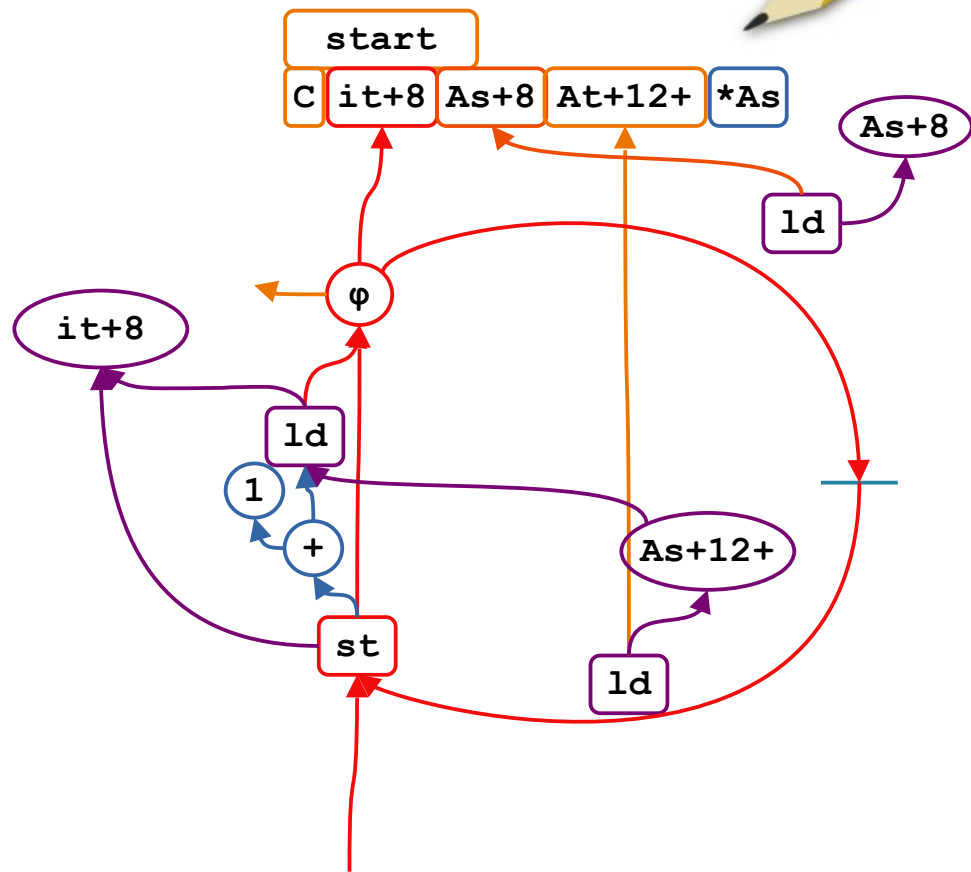
```
int[] As=...;
Iter it=new Iter();
while( it.x < As.length ) {
    int i = As[it.x++];
    sum += i;
}
```



Peepholes

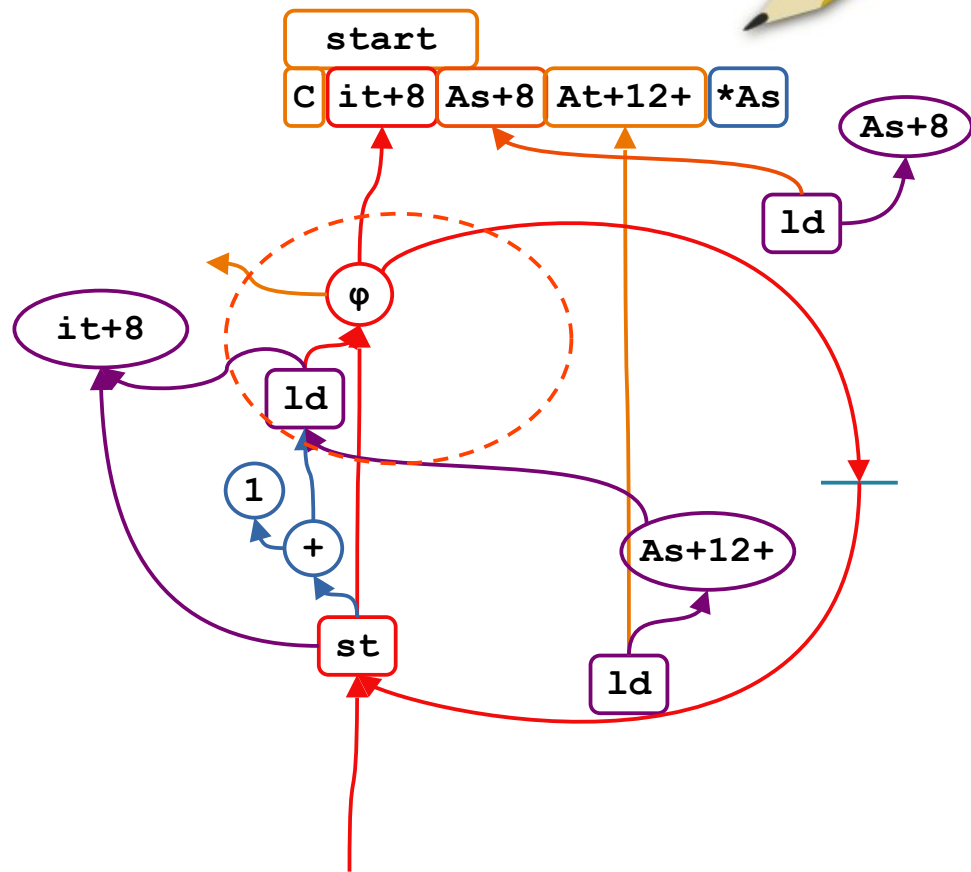
- Peephole remove Phis
- Re-layout graph

```
int[] As=...;  
Iter it=new Iter();  
while( it.x < As.length ) {  
    int i = As[it.x++];  
    sum += i;  
}
```



Peepholes

- Peephole: push ld up thru phi

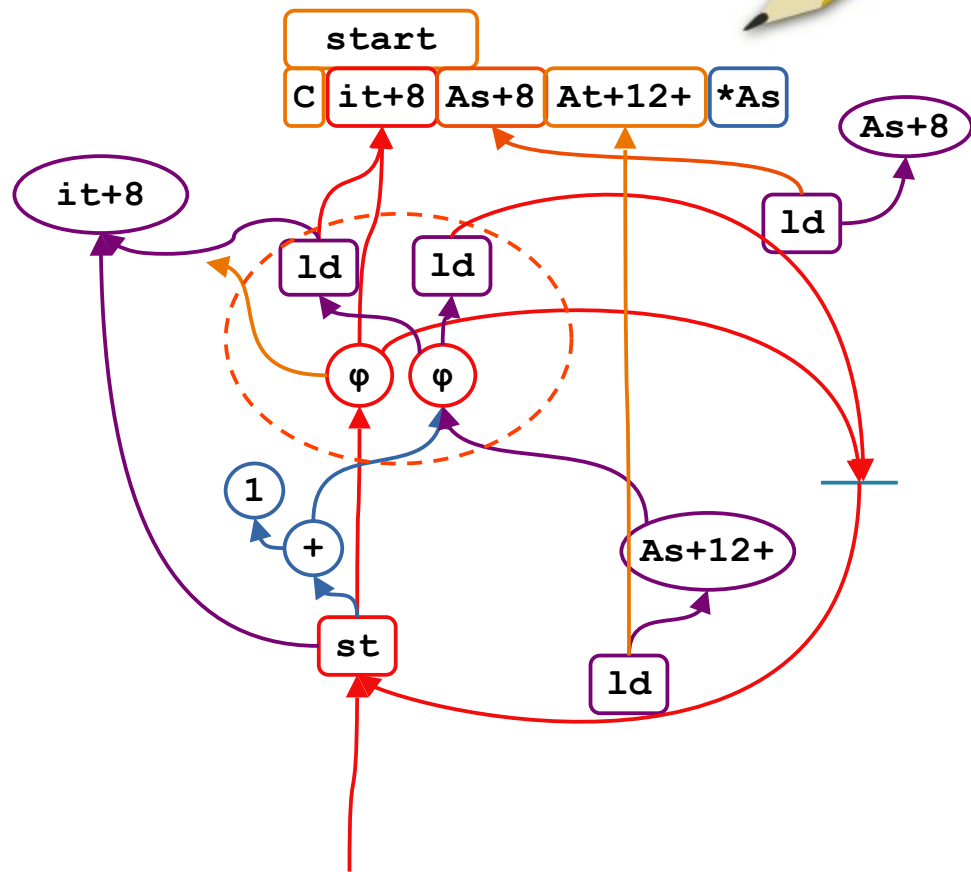


```
int[] As=...;
Iter it=new Iter();
while( it.x < As.length ) {
    int i = As[it.x++];
    sum += i;
}
```

Peepholes

- Peephole: push ld up thru phi

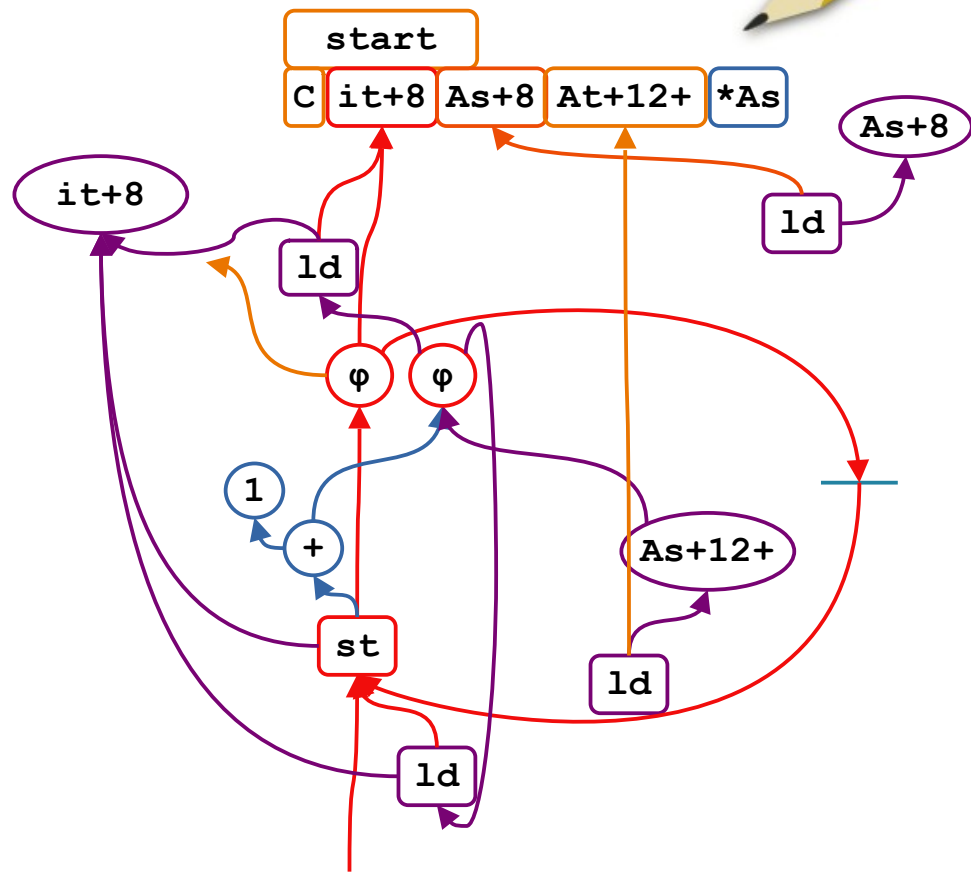
```
int[] As=...;  
Iter it=new Iter();  
while( it.x < As.length ) {  
    int i = As[it.x++];  
    sum += i;  
}
```



Peepholes

- Peephole: push ld up thru phi
- Relay layout graph

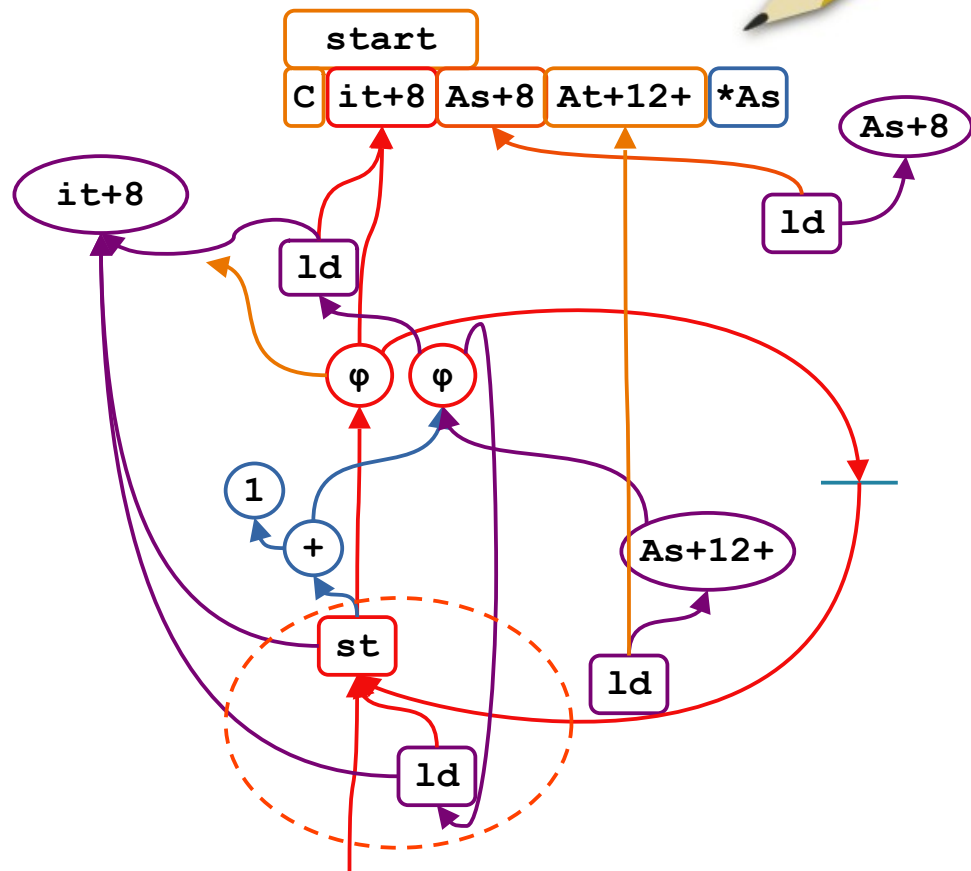
```
int[] As=...;  
Iter it=new Iter();  
while( it.x < As.length ) {  
    int i = As[it.x++];  
    sum += i;  
}
```



Peepholes

- Back-to-back store-load

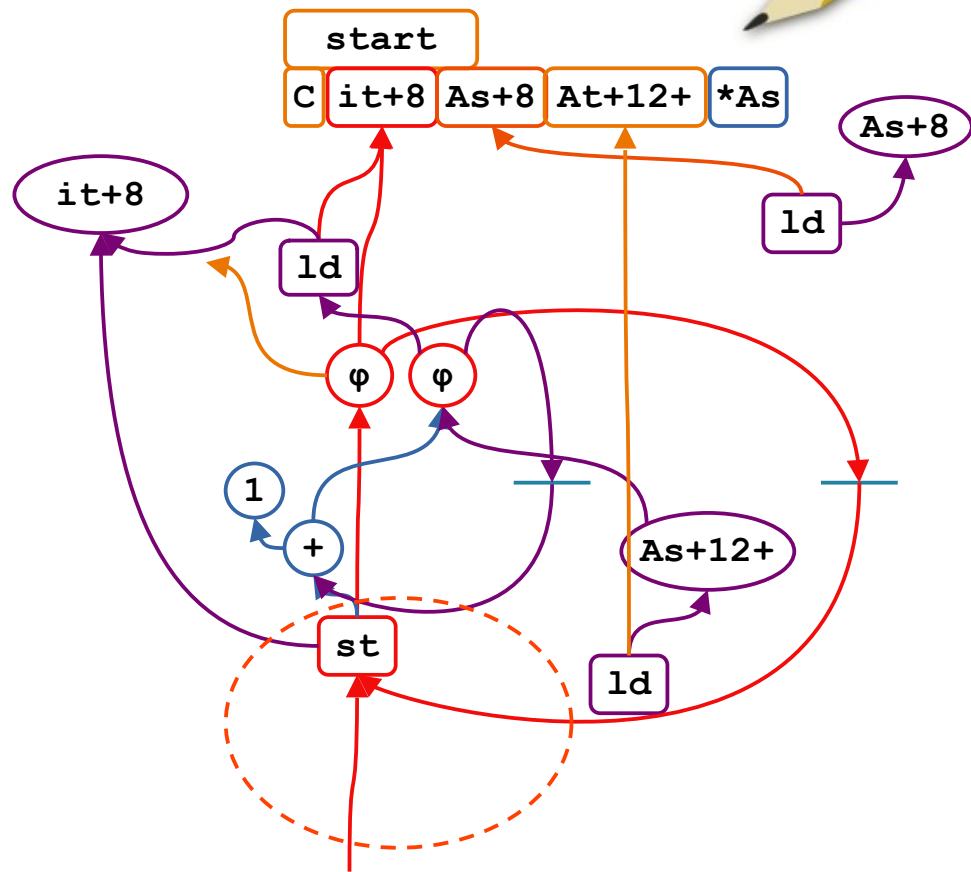
```
int[] As=...;  
Iter it=new Iter();  
while( it.x < As.length ) {  
    int i = As[it.x++];  
    sum += i;  
}
```



Peepholes

- Back-to-back store-load

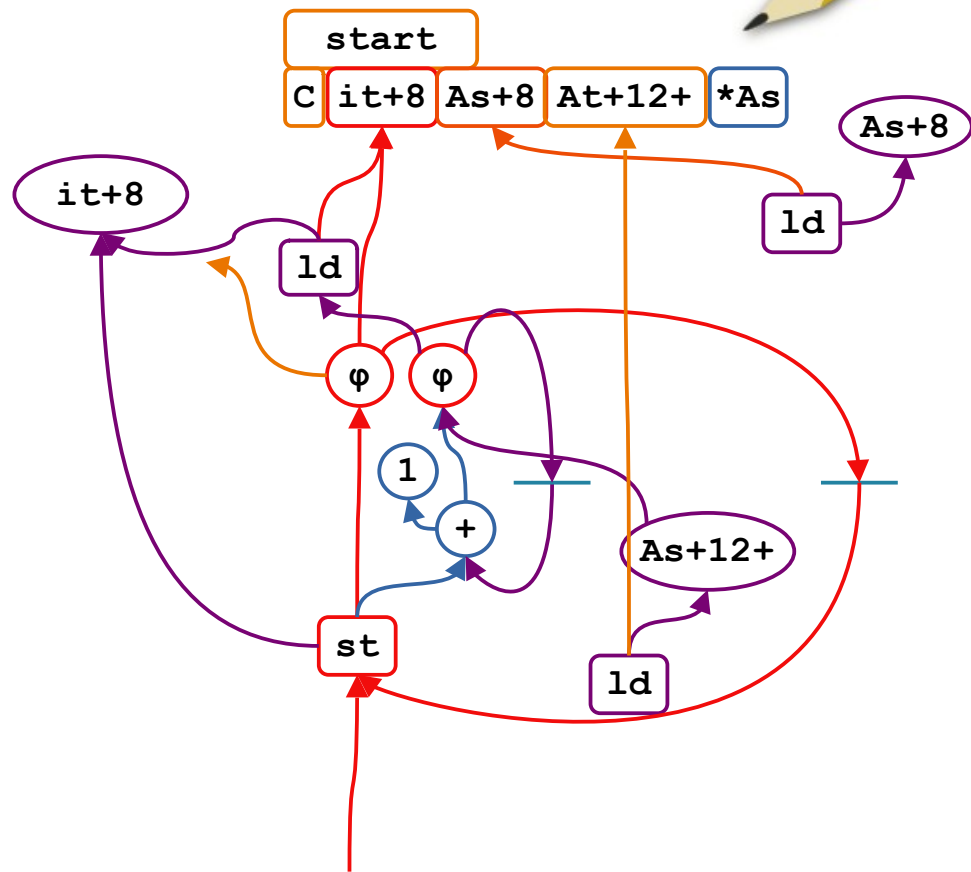
```
int[] As=...;  
Iter it=new Iter();  
while( it.x < As.length ) {  
    int i = As[it.x++];  
    sum += i;  
}
```



Peepholes

- Back-to-back store-load
- Relay layout graph

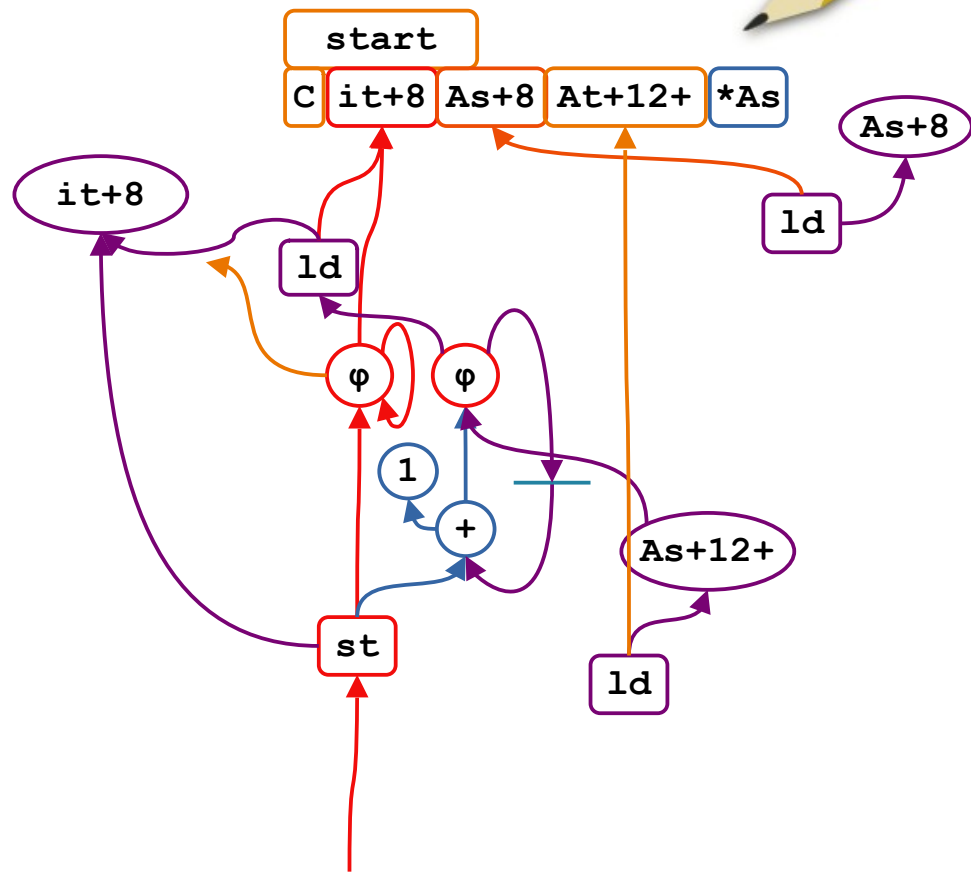
```
int[] As=...;  
Iter it=new Iter();  
while( it.x < As.length ) {  
    int i = As[it.x++];  
    sum += i;  
}
```



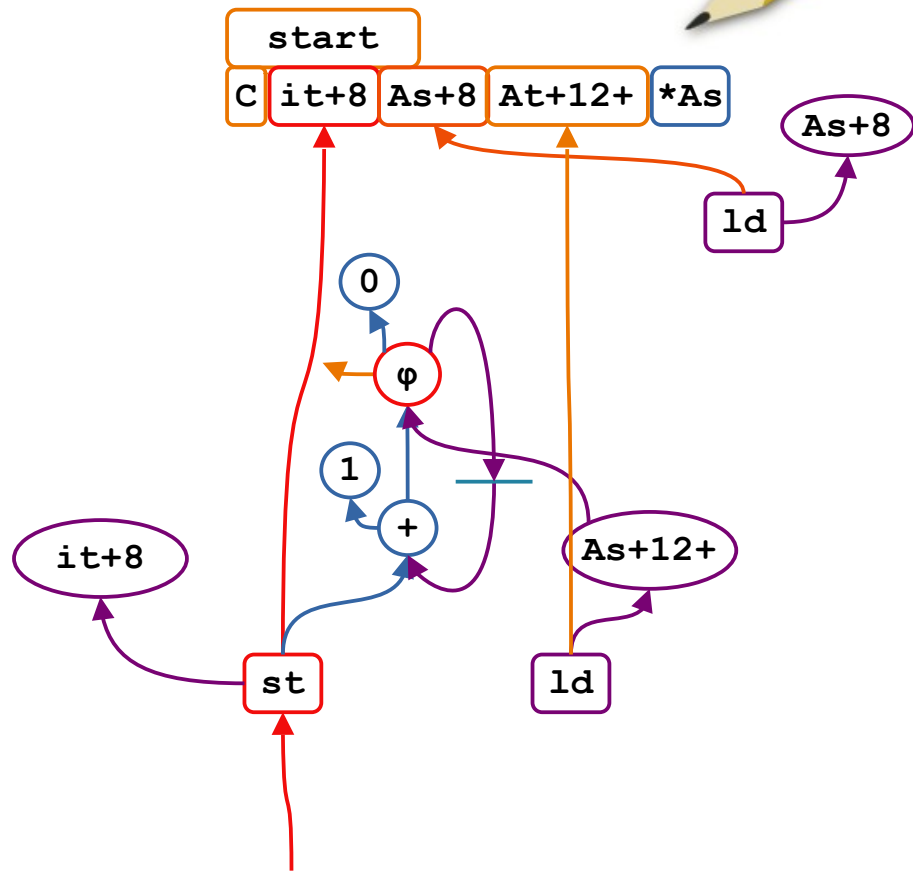
Peepholes

- Back-to-back store-store
- Relay layout graph

```
int[] As=...;  
Iter it=new Iter();  
while( it.x < As.length ) {  
    int i = As[it.x++];  
    sum += i;  
}
```

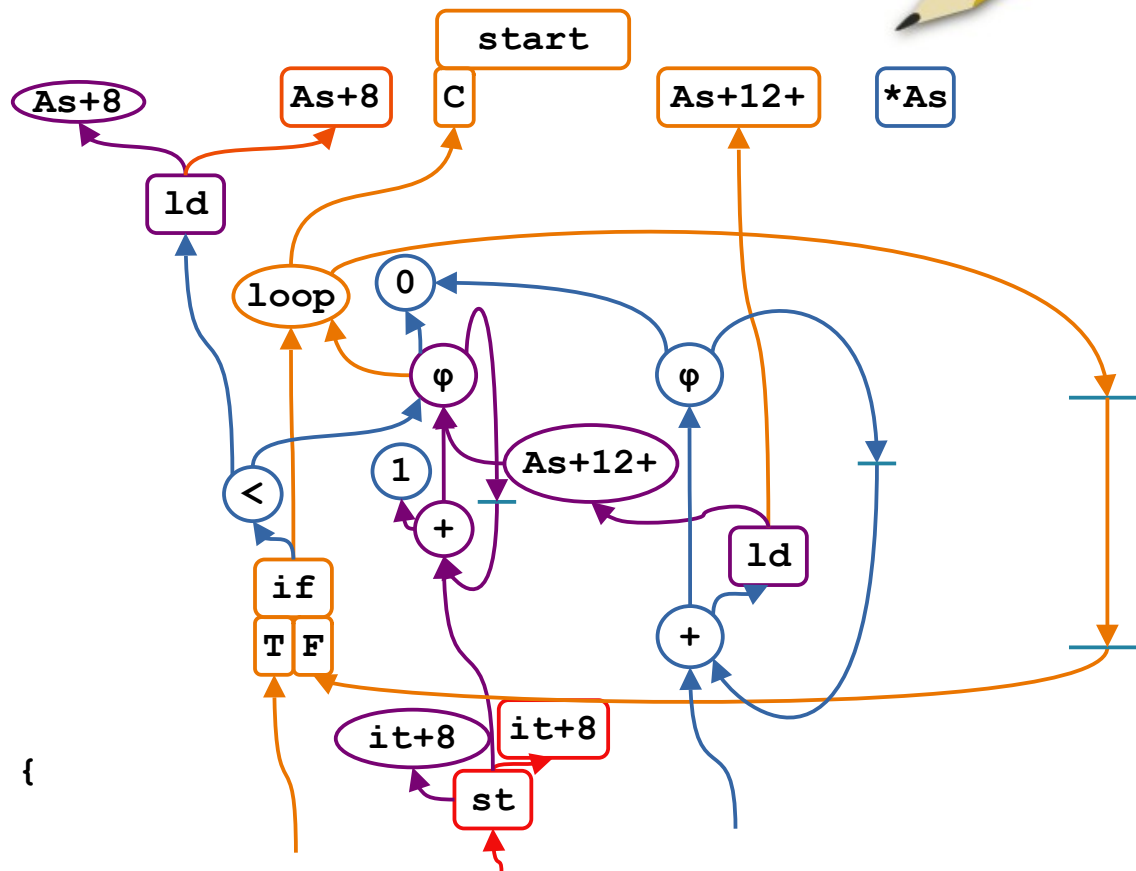


```
int[] As=...;
Iter it=new Iter();
while( it.x < As.length ) {
    int i = As[it.x++];
    sum += i;
}
```



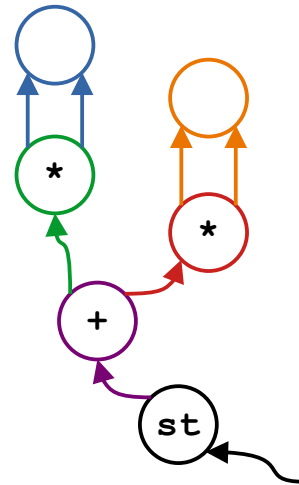
All Together

```
int[] As=...;
Iter it=new Iter();
while( it.x < As.length ) {
    int i = As[it.x++];
    sum += i;
}
```



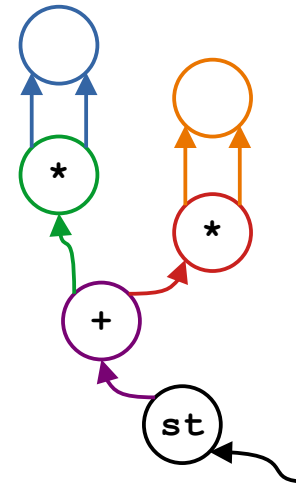
Big Graph

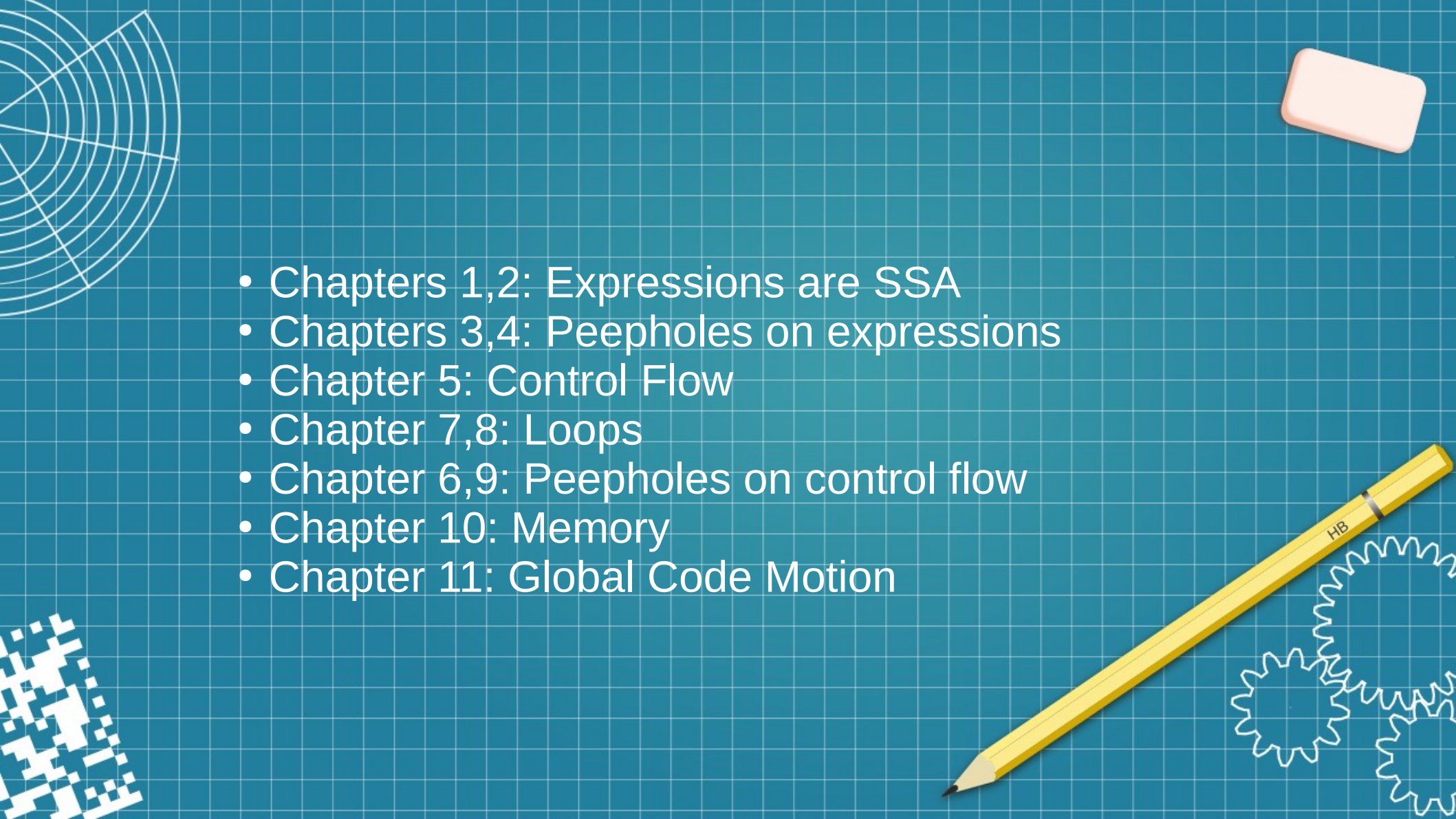
- Like any other IR, scales to size of compilation unit
- Uses graph walks
 - Same as walking a Control Flow Graph
 - Also includes Data!
 - Visit-bit per-node
 - Usually with an external BitSet, using unique int
 - Compilation *speed* is often tied to IR *size*
 - *Smaller IR* → *Faster Compiles*
 - Take the effort to minimize IR footprint!
- Debugging usually involves pretty-prints
 - Same as a CFG-based IR

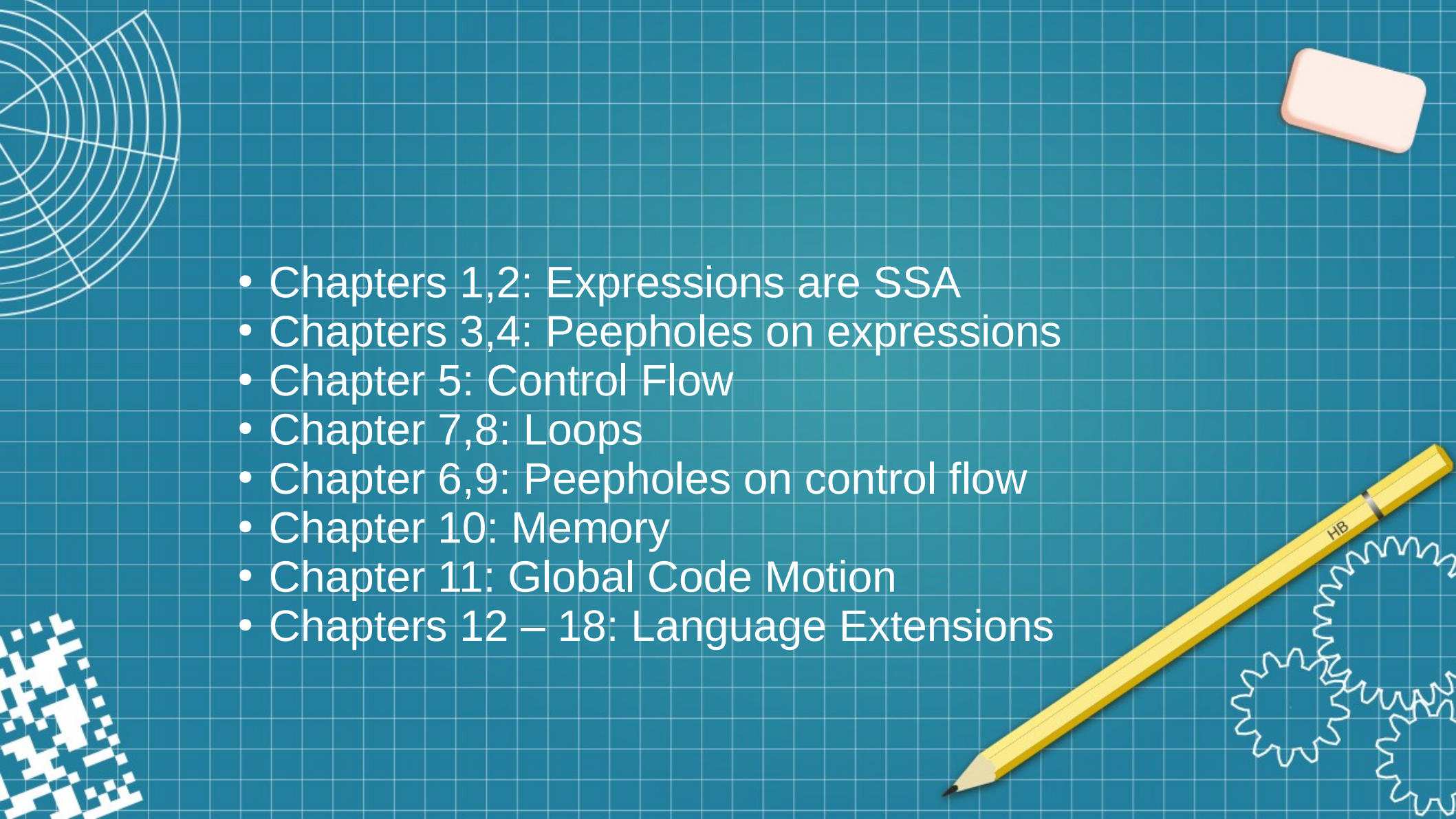


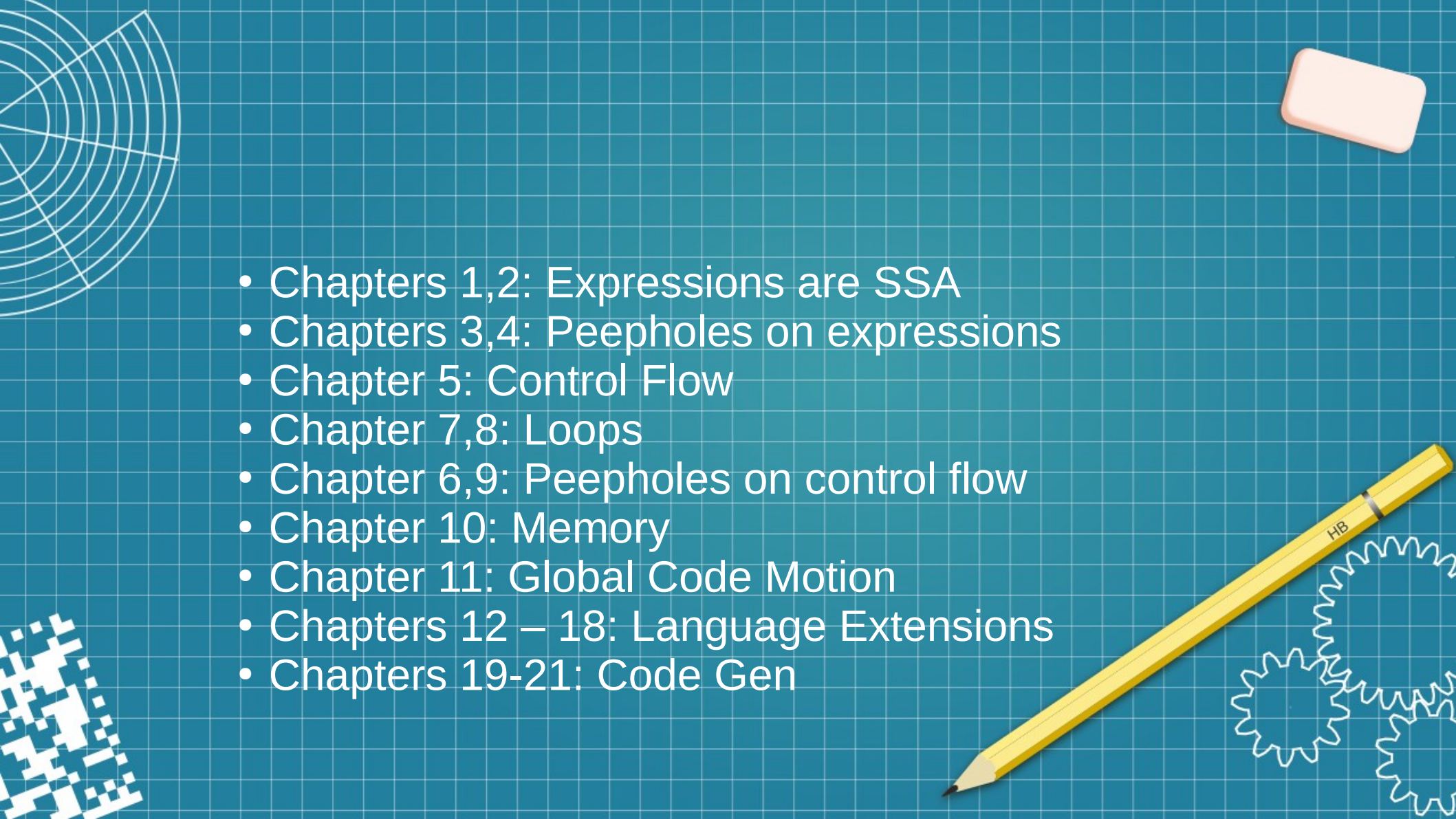
Sea of Nodes

- Graph- and SSA-based IR
 - Mixing Control and Data
- Peepholes: One Cheap Optimization to Rule Them All!
 - Ok, not quite, but covers a LOT of ground
 - Still need e.g. inlining, loop opts, unroll, code-gen
 - Easy to write and debug peepholes
 - BECAUSE they are entirely locally correct
 - Even memory is a local peephole
 - Aliasing is always correct by construction!
 - 25+ yrs of Sun/Oracle engineers extending



- 
- Chapters 1,2: Expressions are SSA
 - Chapters 3,4: Peephholes on expressions
 - Chapter 5: Control Flow
 - Chapter 7,8: Loops
 - Chapter 6,9: Peephholes on control flow
 - Chapter 10: Memory
 - Chapter 11: Global Code Motion

- 
- Chapters 1,2: Expressions are SSA
 - Chapters 3,4: Peephholes on expressions
 - Chapter 5: Control Flow
 - Chapter 7,8: Loops
 - Chapter 6,9: Peephholes on control flow
 - Chapter 10: Memory
 - Chapter 11: Global Code Motion
 - Chapters 12 – 18: Language Extensions

- 
- Chapters 1,2: Expressions are SSA
 - Chapters 3,4: Peephholes on expressions
 - Chapter 5: Control Flow
 - Chapter 7,8: Loops
 - Chapter 6,9: Peephholes on control flow
 - Chapter 10: Memory
 - Chapter 11: Global Code Motion
 - Chapters 12 – 18: Language Extensions
 - Chapters 19-21: Code Gen