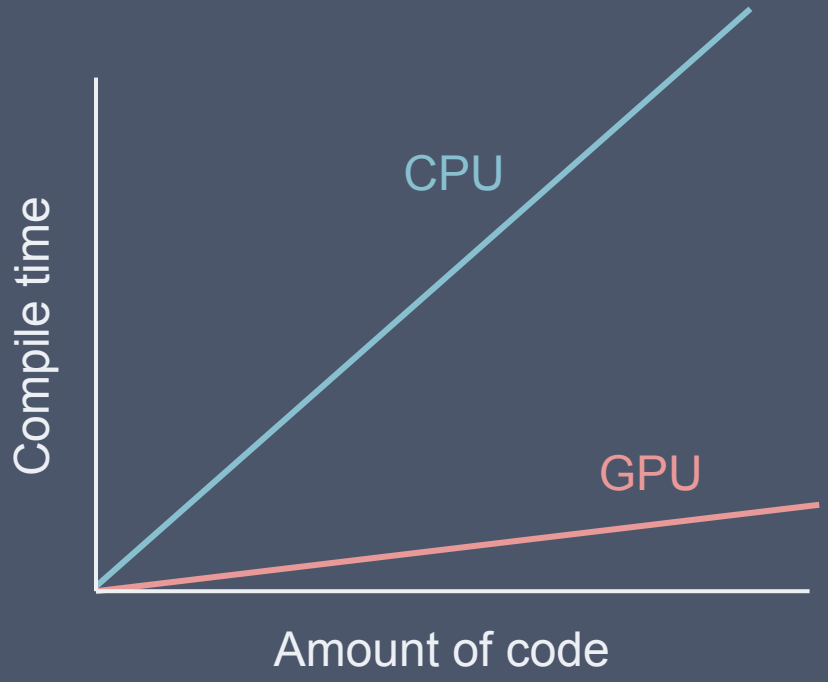
Motivation



*in theory

Compiler anatomy overview

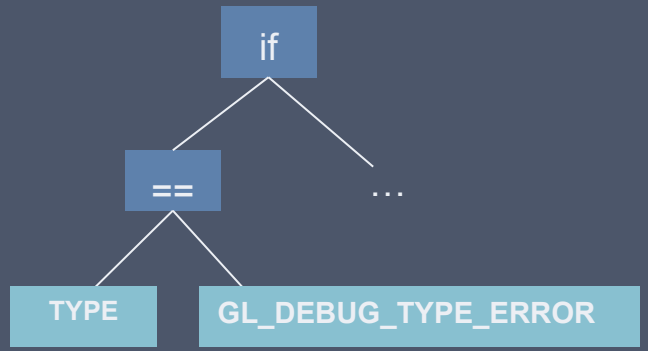
Front End

- Tokenizer

```
if (type == GL_DEBUG_TYPE_ERROR) {
```

[“if”, “(“, “type”, “==”,
“GL_DEBUG_TYPE_ERROR”, “)” , “{“,]

- Parser



- Semantic Analysis*

Back End

- Optimisation*
- Instruction Selection

```
mov %1 , type
cmp %1 , GL_DEBUG_TYPE_ERROR
jne skip
...
skip:
```

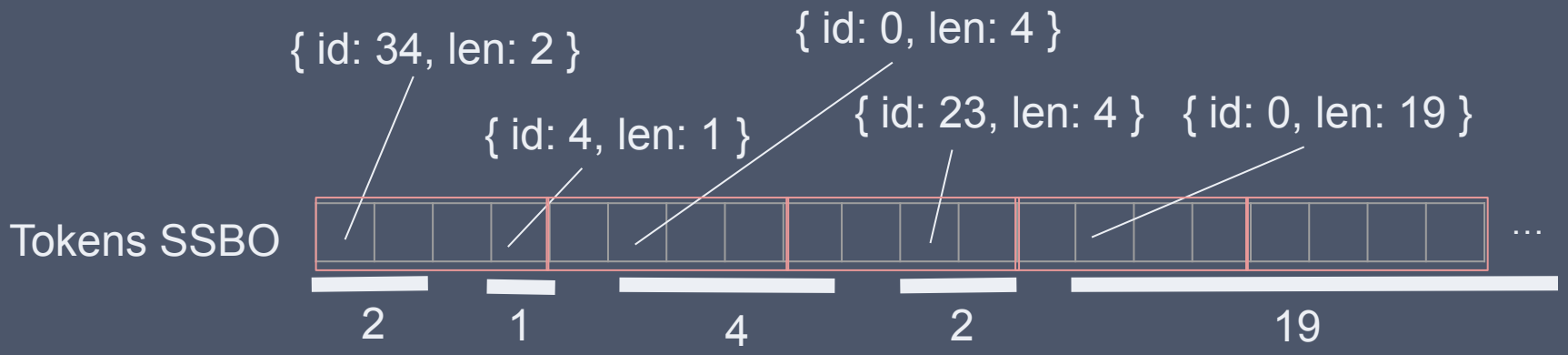
- Register allocation

```
mov eax, type
cmp eax, GL_DEBUG_TYPE_ERROR
jne skip
...
skip:
```

Tokenizer

```
if (type == GL_DEBUG_TYPE_ERROR) {
```

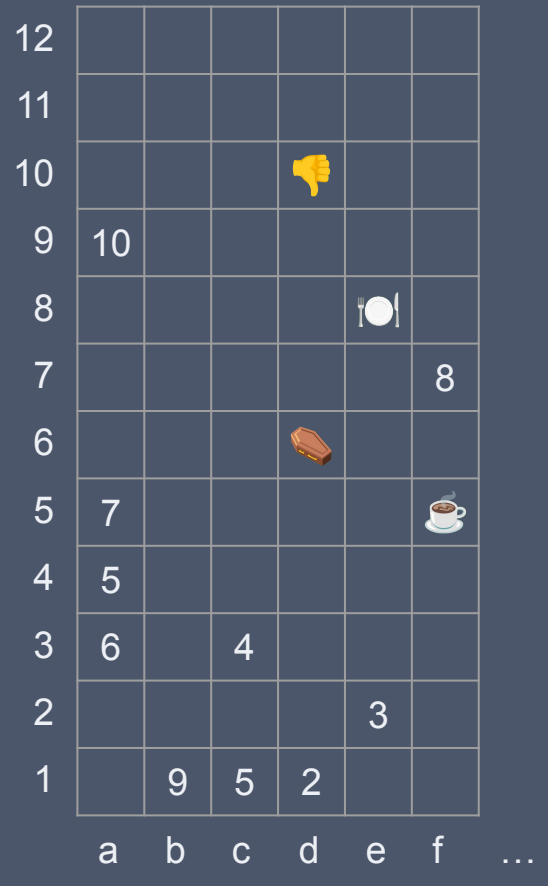
```
struct Token {
    uint id;
    uint len;
};
```



Pattern matching

```
struct ParseTreeItem {
    uint nextRow;
    uint final;
};
```

- ☕ = “decaf”
- ⚰️ = “dead”
- 🍽️ = “cafe”
- 👎 = “bad”

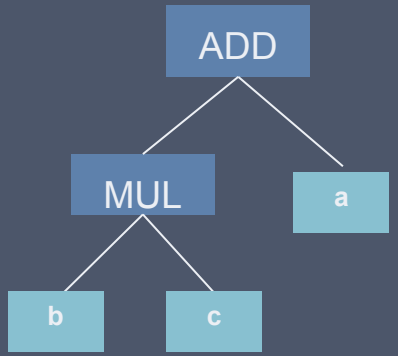


- Empty => { nextRow: 0, final: 0 }
- Number => { nextRow: n, final: 0 }
- Emoji => { nextRow: 0, final: id }

Parser

```
struct Token {
    uint id;
    uint len;
};
```

Tokens SSBO

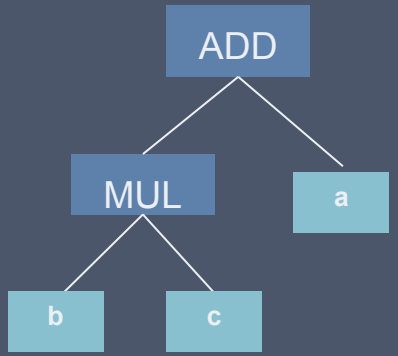



```
PRIMARY_EXPRESSION
   : ADD
   | MUL
   ;

MUL
    : PRIMARY_EXPRESSION ‘*’ PRIMARY_EXPRESSION
    ;

ADD
    : PRIMARY_EXPRESSION ‘+’ PRIMARY_EXPRESSION
    ;
```

Parser

```
struct Token {
    uint id;
    uint len;
};
```

Tokens SSBO




```
PRIMARY_EXPRESSION
   : ADD
   | MUL
   ;

MUL
    : PRIMARY_EXPRESSION '*' PRIMARY_EXPRESSION
    ;

ADD
    : { '*' } PRIMARY_EXPRESSION '+' PRIMARY_EXPRESSION
    ;
```

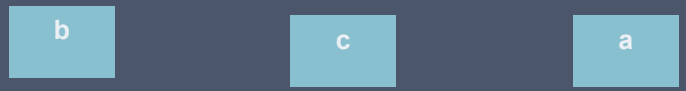
Parser

```
struct Token {
    uint id;
    uint len;
};
```

Invocation 0 | Invocation 1 | Invocation 2

Tokens SSBO


```
PRIMARY_EXPRESSION
  : ADD
  | MUL
  ;

MUL
   : PRIMARY_EXPRESSION '*' PRIMARY_EXPRESSION
   ;

ADD
   : { '*' } PRIMARY_EXPRESSION '+' PRIMARY_EXPRESSION
   ;
```



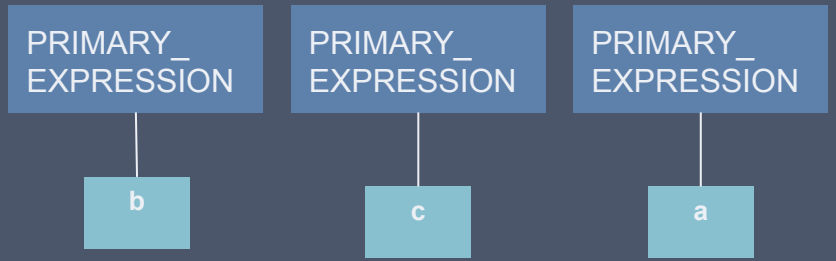
Parser

```
struct Token {
    uint id;
    uint len;
};
```

Invocation 0 | Invocation 1 | Invocation 2

Tokens SSBO


```
PRIMARY_EXPRESSION
  : ADD
  | MUL
  ;

MUL
   : PRIMARY_EXPRESSION '*' PRIMARY_EXPRESSION
   ;

ADD
   : { '*' } PRIMARY_EXPRESSION '+' PRIMARY_EXPRESSION
   ;
```



AST SSBO

Atomic overflow pointer ->

0	{ type: PE, childIndexes: [], codeLocation: 0 }
1	
2	{ type: PE, childIndexes: [], codeLocation: 5 }
3	
4	{ type: PE, childIndexes: [], codeLocation: 9 }
5	
6	
7	
8	

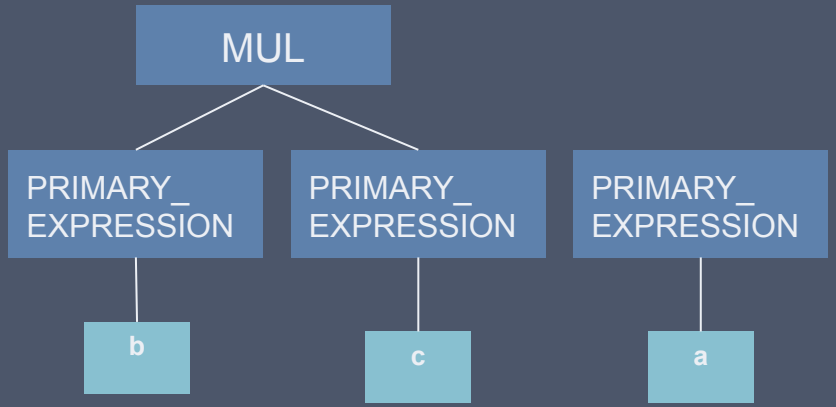
0
1
2
overflow

Parser

```
struct Token {
    uint id;
    uint len;
};
```

Invocation 0 | Invocation 1 | Invocation 2

Tokens SSBO




```
PRIMARY_EXPRESSION
   : ADD
   | MUL
   ;

MUL
   : PRIMARY_EXPRESSION '*' PRIMARY_EXPRESSION
   ;

ADD
   : { '*' } PRIMARY_EXPRESSION '+' PRIMARY_EXPRESSION
   ;
```

AST SSBO

Atomic overflow pointer ->

0	{ type: PE, childIndexes: [], codeLocation: 0 }
1	
2	{ type: PE, childIndexes: [], codeLocation: 5 }
3	{ type: MUL, childIndexes: [2, 4], codeLocation: -1 }
4	{ type: PE, childIndexes: [], codeLocation: 9 }
5	
6	
7	
8	

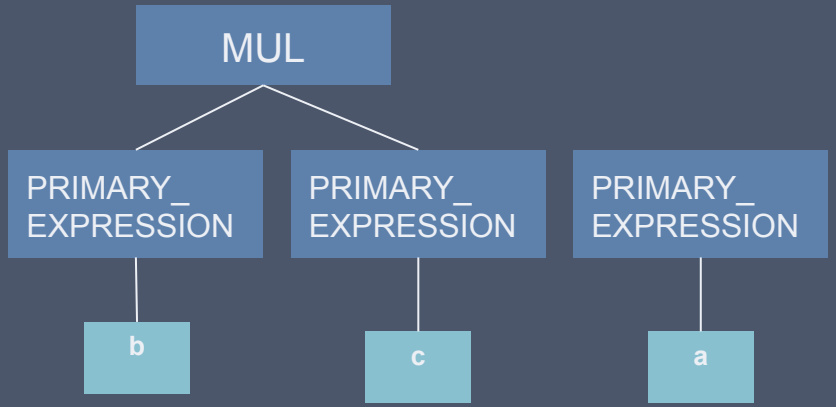
0
1
2
overflow

Parser

```
struct Token {
    uint id;
    uint len;
    int astNodeLocation;
};
```

Invocation 0 | Invocation 1 | Invocation 2

Tokens SSBO




```
PRIMARY_EXPRESSION
  : ADD
  | MUL
  ;

MUL
  : PRIMARY_EXPRESSION '*' PRIMARY_EXPRESSION
  ;

ADD
  : { '*' } PRIMARY_EXPRESSION '+' PRIMARY_EXPRESSION
  ;
```

AST SSBO

0	{ type: PE, childIndexes: [], codeLocation: 0 }
1	
2	{ type: PE, childIndexes: [], codeLocation: 5 }
3	{ type: MUL, childIndexes: [2, 4], codeLocation: -1 }
4	{ type: PE, childIndexes: [], codeLocation: 9 }
5	
6	
7	
8	

0 1 2 overflow

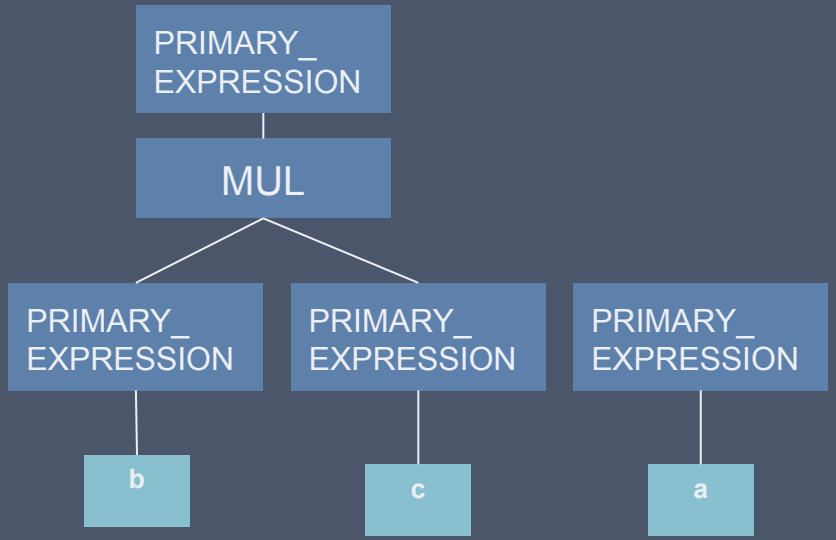
Atomic overflow pointer ->

Parser

```
struct Token {
    uint id;
    uint len;
    int astNodeLocation;
};
```

Invocation 0 | Invocation 1 | Invocation 2

Tokens SSBO




```
PRIMARY_EXPRESSION
  : ADD
  | MUL
  ;

MUL
   : PRIMARY_EXPRESSION '*' PRIMARY_EXPRESSION
   ;

ADD
   : { '*' } PRIMARY_EXPRESSION '+' PRIMARY_EXPRESSION
   ;
```

AST SSBO

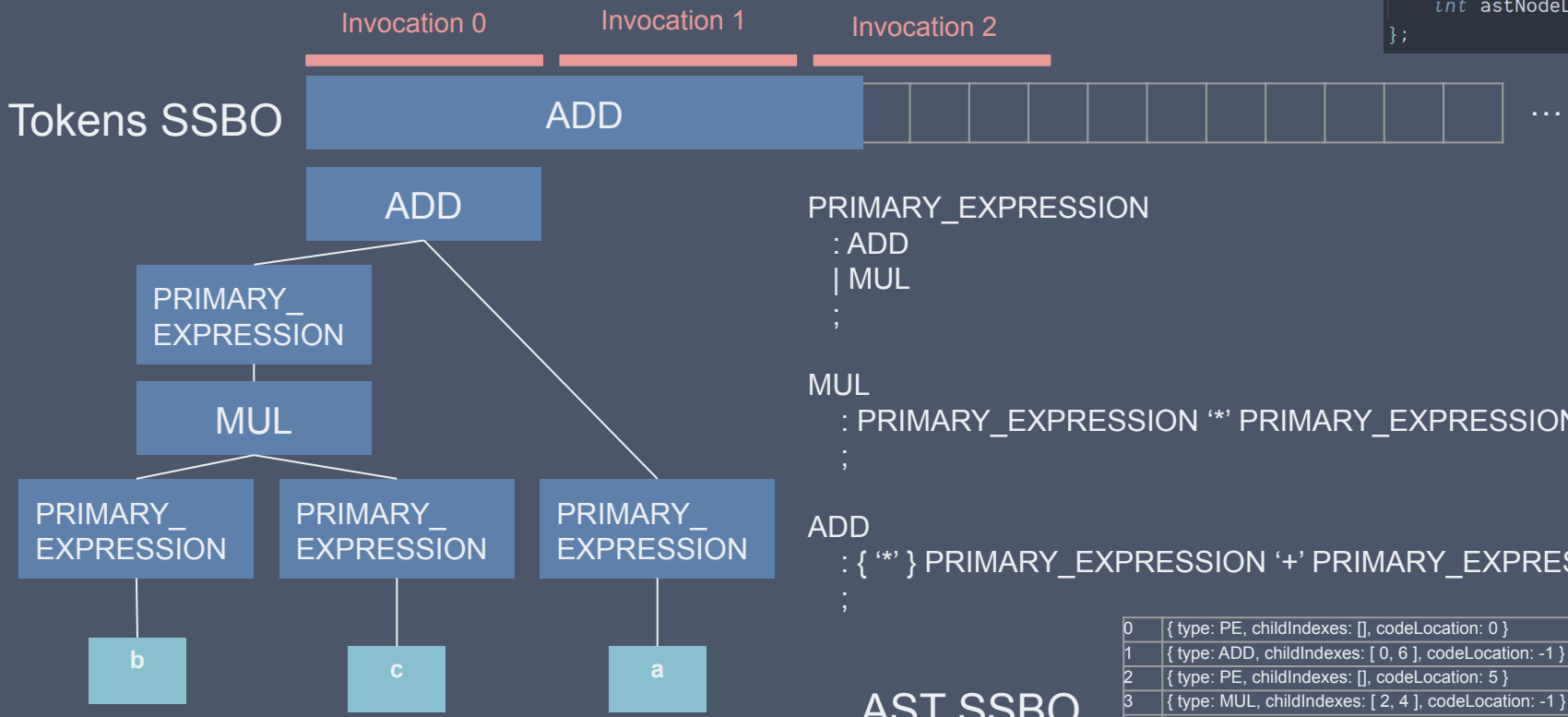
0	{ type: PE, childIndexes: [], codeLocation: 0 }
1	
2	{ type: PE, childIndexes: [], codeLocation: 5 }
3	{ type: MUL, childIndexes: [2, 4], codeLocation: -1 }
4	{ type: PE, childIndexes: [], codeLocation: 9 }
5	
6	{ type: PE, childIndexes: [3], codeLocation: -1 }
7	
8	

0
1
2
overflow

Atomic overflow pointer ->

Parser

```
struct Token {
    uint id;
    uint len;
    int astNodeLocation;
};
```




```
PRIMARY_EXPRESSION
  : ADD
  | MUL
  ;

MUL
  : PRIMARY_EXPRESSION '*' PRIMARY_EXPRESSION
  ;

ADD
  : { '*' } PRIMARY_EXPRESSION '+' PRIMARY_EXPRESSION
  ;
```

AST SSBO

0	{ type: PE, childIndexes: [], codeLocation: 0 }
1	{ type: ADD, childIndexes: [0, 6], codeLocation: -1 }
2	{ type: PE, childIndexes: [], codeLocation: 5 }
3	{ type: MUL, childIndexes: [2, 4], codeLocation: -1 }
4	{ type: PE, childIndexes: [], codeLocation: 9 }
5	
6	{ type: PE, childIndexes: [3], codeLocation: -1 }
7	
8	

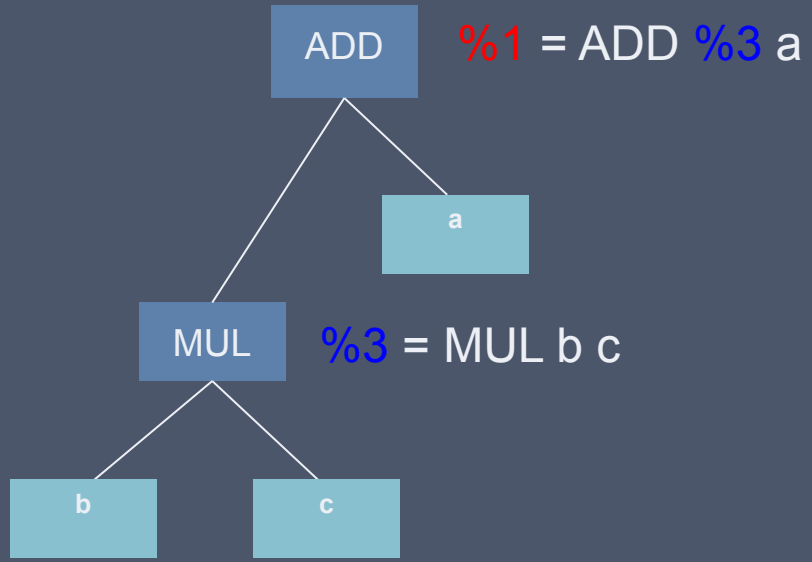
Atomic overflow pointer ->

0 1 2 overflow

IR codegen

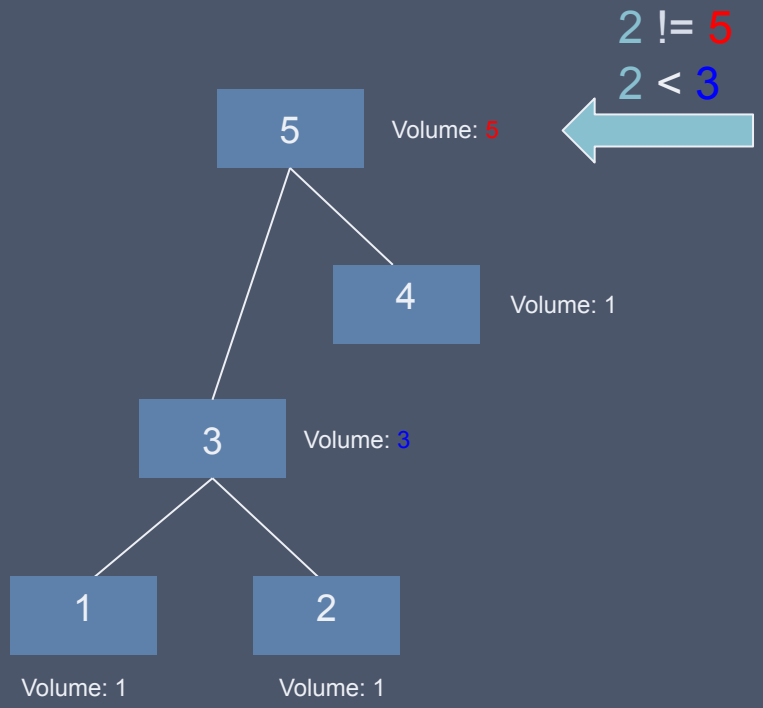
AST SSBO

0	{ type: PE, childIndexes: [], codeLocation: 0 }
1	{ type: ADD, childIndexes: [0, 6], codeLocation: -1 }
2	{ type: PE, childIndexes: [], codeLocation: 5 }
3	{ type: MUL, childIndexes: [2, 4], codeLocation: -1 }
4	{ type: PE, childIndexes: [], codeLocation: 9 }
5	
6	{ type: PE, childIndexes: [3], codeLocation: -1 }
7	
8	




```
PRIMARY_EXPRESSION
   : ADD
   | MUL
   ;

MUL
    : $1 PRIMARY_EXPRESSION '*' $2 PRIMARY_EXPRESSION
    ; < %SELF = MUL %1 %2 >

ADD
    : { '*' } $1 PRIMARY_EXPRESSION '+' $2 PRIMARY_EXPRESSION
    ; < %SELF = ADD %1 %2 >
```

Tree Traversal Algorithm

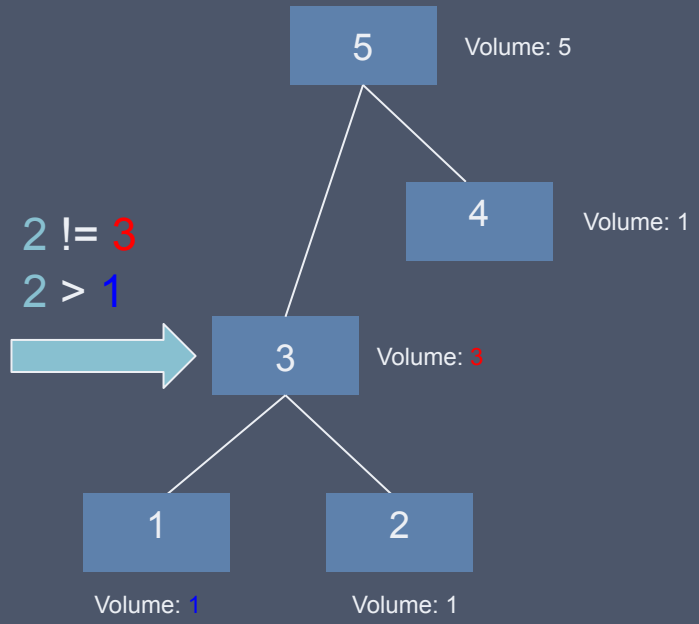


Initial volume := Invocation ID = 2

Current volume: 2

Each node has a dedicated invocation.

Tree Traversal Algorithm

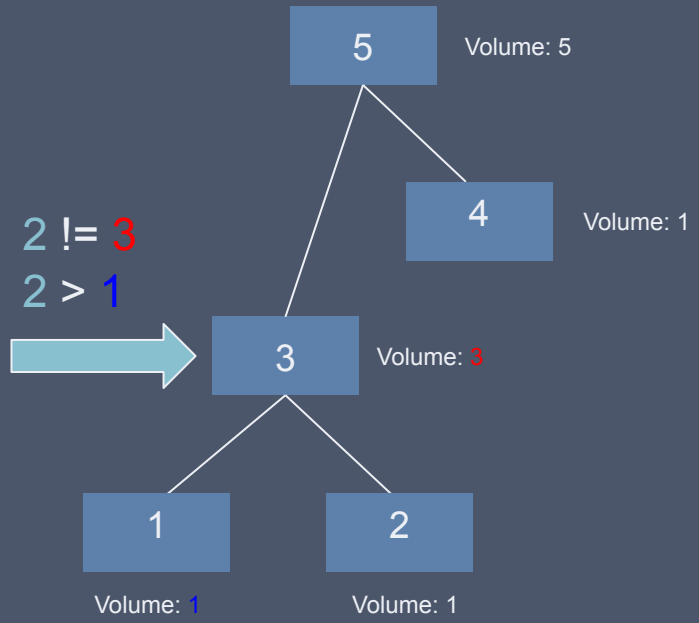


Initial volume := Invocation ID = 2

Current volume: 2

Each node has a dedicated invocation.

Tree Traversal Algorithm

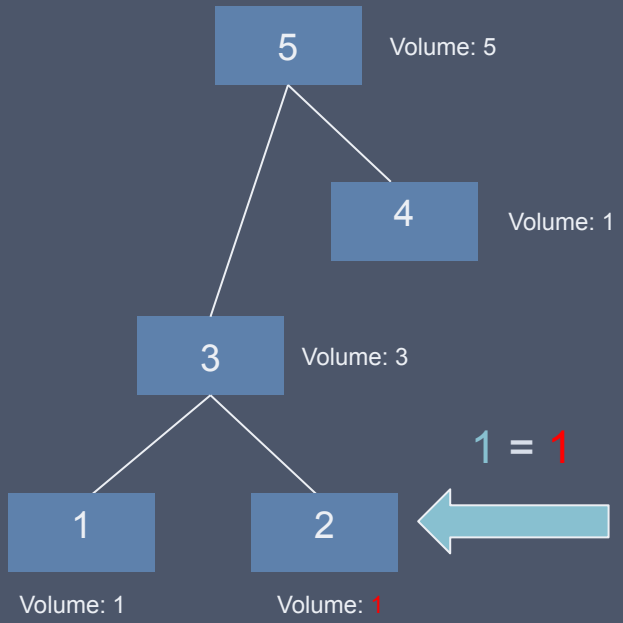


Initial volume := Invocation ID = 2

Current volume: 2 - 1

Each node has a dedicated invocation.

Tree Traversal Algorithm



Initial volume := Invocation ID = 2

Current volume: 1

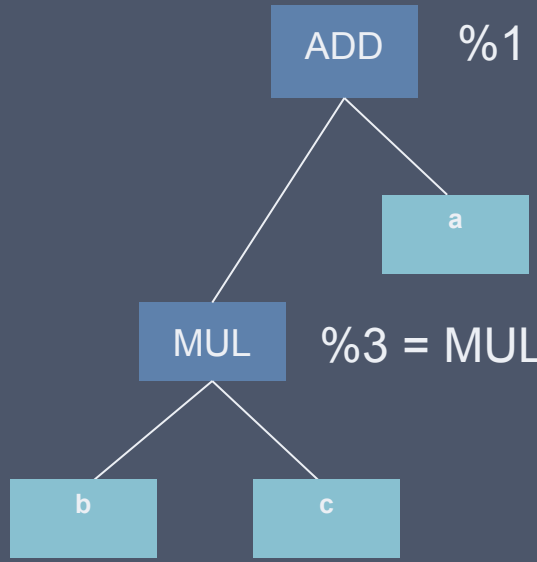
Each node has a dedicated invocation.

IR codegen

AST SSBO

0	{ type: PE, childIndexes: [], codeLocation: 0 }
1	{ type: ADD, childIndexes: [0, 6], codeLocation: -1 }
2	{ type: PE, childIndexes: [], codeLocation: 5 }
3	{ type: MUL, childIndexes: [2, 4], codeLocation: -1 }
4	{ type: PE, childIndexes: [], codeLocation: 9 }
5	
6	{ type: PE, childIndexes: [3], codeLocation: -1 }
7	
8	

```
PRIMARY_EXPRESSION
   : ADD
   | MUL
   ;

MUL
    : $1 PRIMARY_EXPRESSION '*' $2 PRIMARY_EXPRESSION
    ; < %SELF = MUL %1 %2 >

ADD
    : { '*' } $1 PRIMARY_EXPRESSION '+' $2 PRIMARY_EXPRESSION
    ; < %SELF = ADD %1 %2 >
```

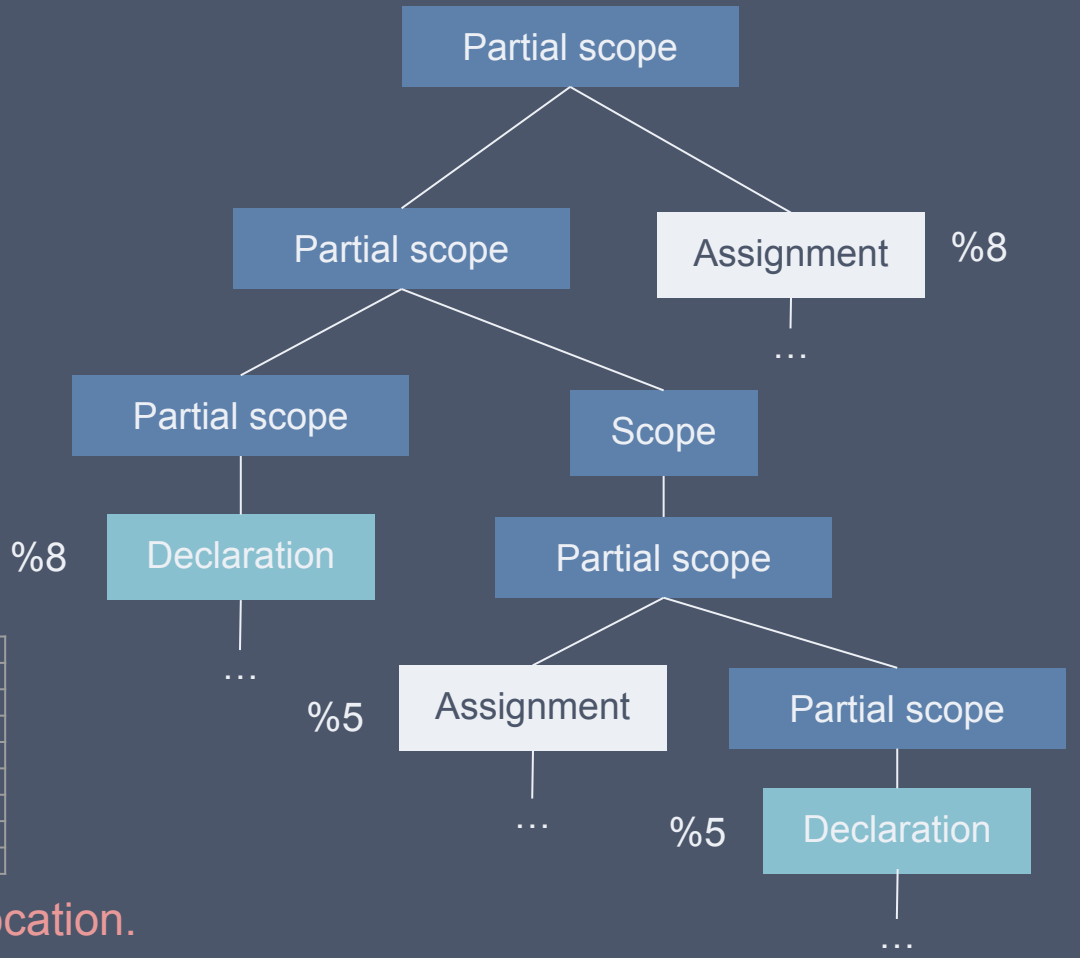


IR SSBO

```
%3 = MUL b c
%1 = ADD %3 a
```

Types and identifiers

```
int a = 0;

{
    int a = 1;
    a = 2;
}

a = 3;
```



Types SSBO

VReg ID	Type
1	
2	
3	
4	
5	{ base: I32, ptrDepth: 0, … }
6	
7	
8	{ base: I32, ptrDepth: 0, … }

Each node has a dedicated invocation.

Type propagation

```
PRIMARY_EXPRESSION
   : ADD
   | MUL
   ;

MUL
    : $1 PRIMARY_EXPRESSION '*' $2 PRIMARY_EXPRESSION
    ; < %SELF = MUL %1 %2 `%SELF := %1 | %2` >

ADD
    : { '*' } $1 PRIMARY_EXPRESSION '+' $2 PRIMARY_EXPRESSION
    ; < %SELF = ADD %1 %2 `%SELF := %1 | %2` >
```

IR SSBO

```
%1 = MUL b c
%2 = ADD %1 a
```

Types SSBO

VReg ID	Type
1	{ base: I32, ptrDepth: 0, … }
2	
3	
4	
5	
6	
7	
8	

Each IR instruction has a dedicated invocation.

Type propagation

```
PRIMARY_EXPRESSION
   : ADD
   | MUL
   ;

MUL
    : $1 PRIMARY_EXPRESSION ‘*’ $2 PRIMARY_EXPRESSION
    ; < %SELF = MUL %1 %2 `%SELF := %1 | %2` >

ADD
    : { ‘*’ } $1 PRIMARY_EXPRESSION ‘+’ $2 PRIMARY_EXPRESSION
    ; < %SELF = ADD %1 %2 `%SELF := %1 | %2` >
```

IR SSBO

```
%1 = MUL b c
%2 = ADD %1 a
```

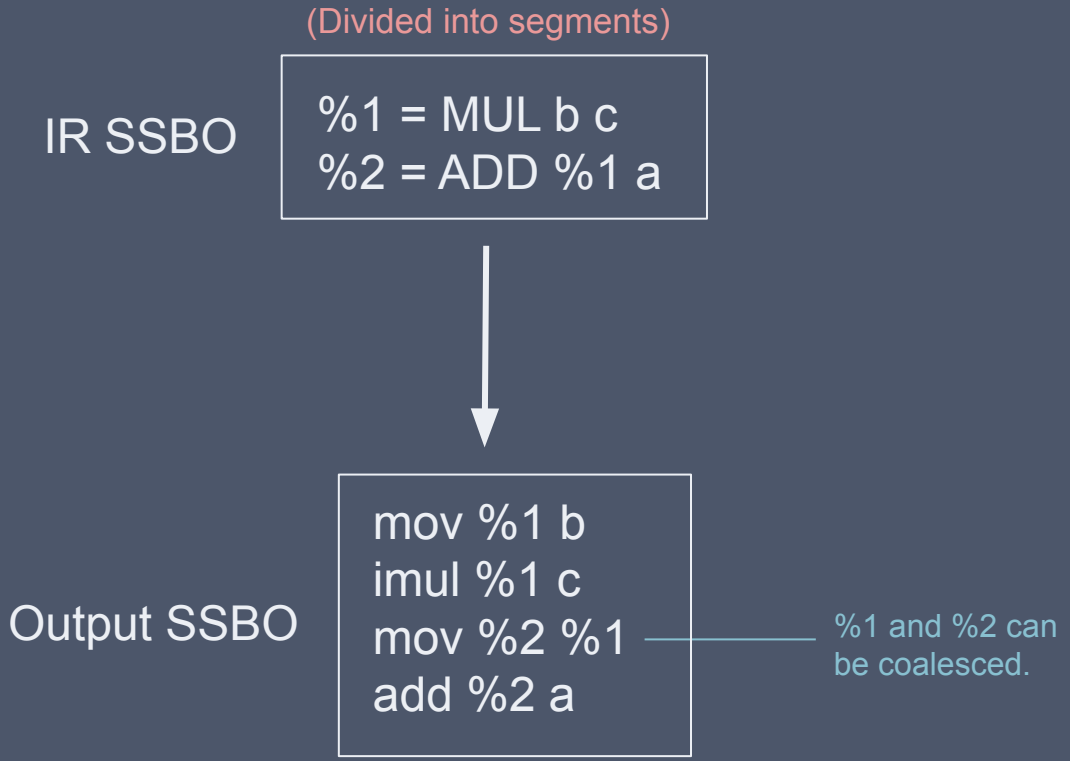
Types SSBO

VReg ID	Type
1	{ base: I32, ptrDepth: 0, … }
2	{ base: I32, ptrDepth: 0, … }
3	
4	
5	
6	
7	
8	

Each IR instruction has a dedicated invocation.

Instruction lowering

```
%a:i32 = ADD %b:i32 %c:i32 ->
   mov %a %b
   add %a %c
   ;

%a:f32 = ADD %b:f32 %c:f32 ->
   mov %a %b
   addss %a %c
   ;

%a:f32 = MUL %b:f32 %c:f32 ->
   mov %a %b
   mulss %a %c
   ;

%a:i32 = MUL %b:i32 %c:i32 ->
   mov %a %b
   imul %a %c
   ;
```



Register allocation

```
struct LiveInterval {
    uint start;
    uint end;
};
```

```
atomicMin(liveIntervals[vreg].start, instPos);
atomicMax(liveIntervals[vreg].end, instPos);
```

```
0     mov %counter, 20
1     loop:
2     cmp %counter, 0
3     je break:

4     mov %a, 0x4
5     mov %b, 0x1
6     add %a, %b
7     mov %c, %b

8     sub %counter, 1
9     jmp loop:
10    break:
```

%a

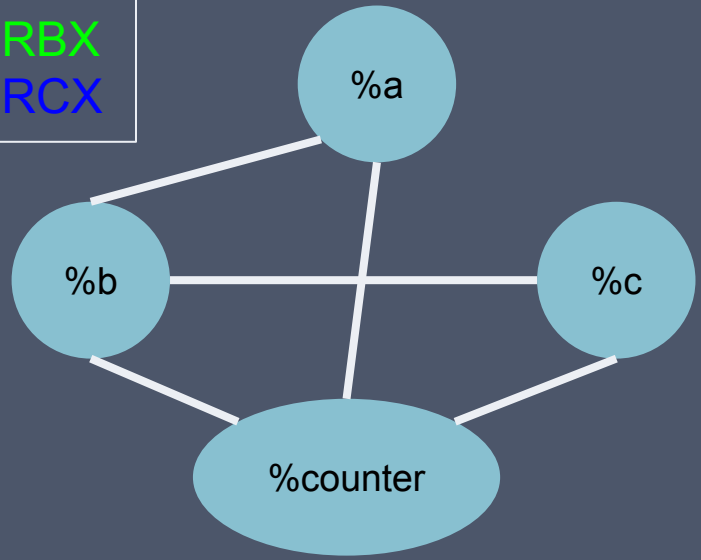
%b

%c

%counter

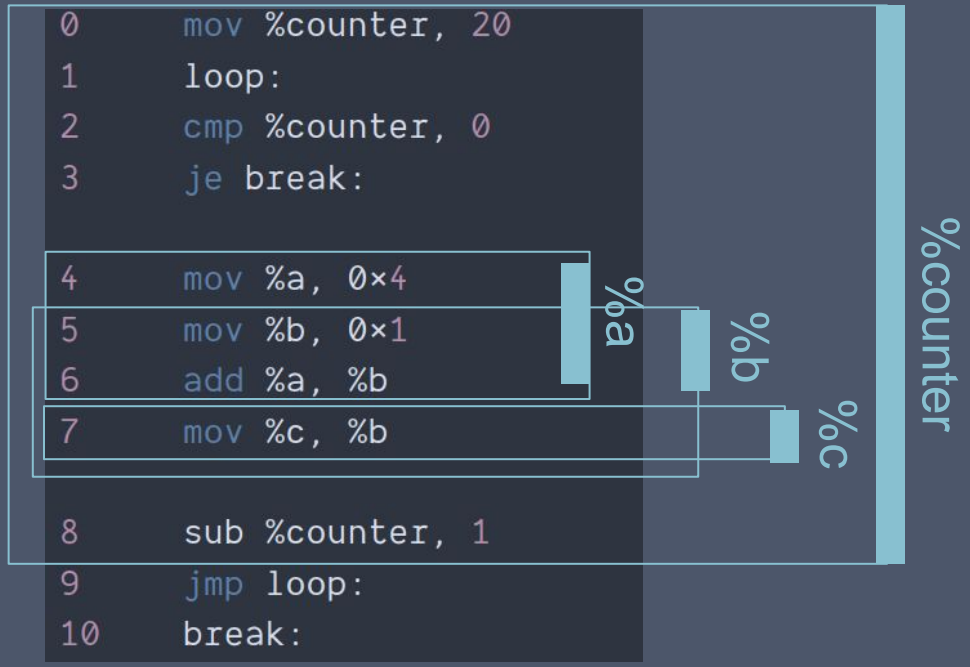
I64 pool

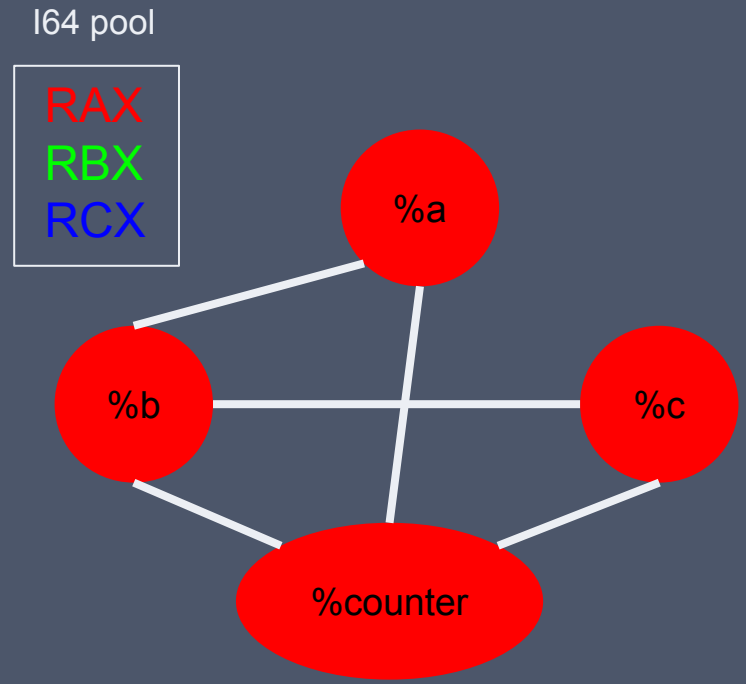
RAX
RBX
RCX



Each VReg has a dedicated invocation.

Register allocation





Greedy allocation

Each VReg has a dedicated invocation.

Register allocation

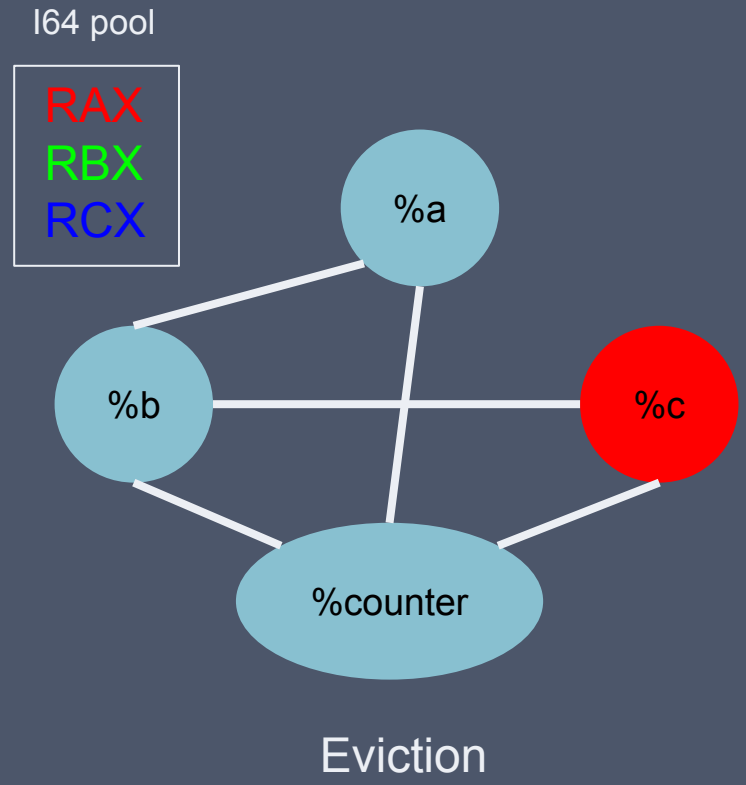
```
0     mov %counter, 20
1     loop:
2     cmp %counter, 0
3     je break:

4     mov %a, 0×4
5     mov %b, 0×1
6     add %a, %b
7     mov %c, %b

8     sub %counter, 1
9     jmp loop:
10    break:
```

%a

%b

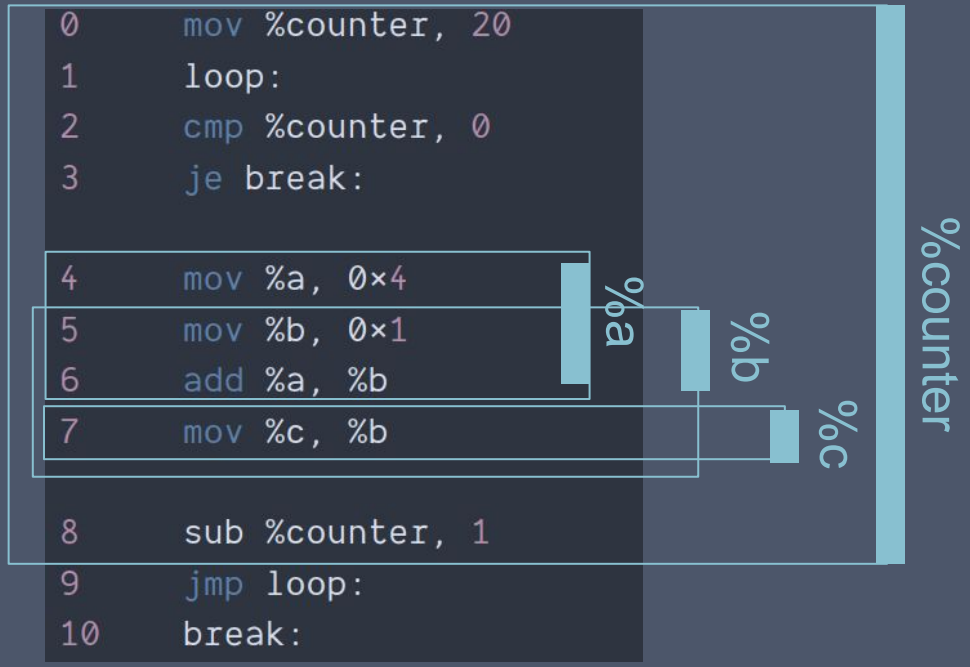
%c

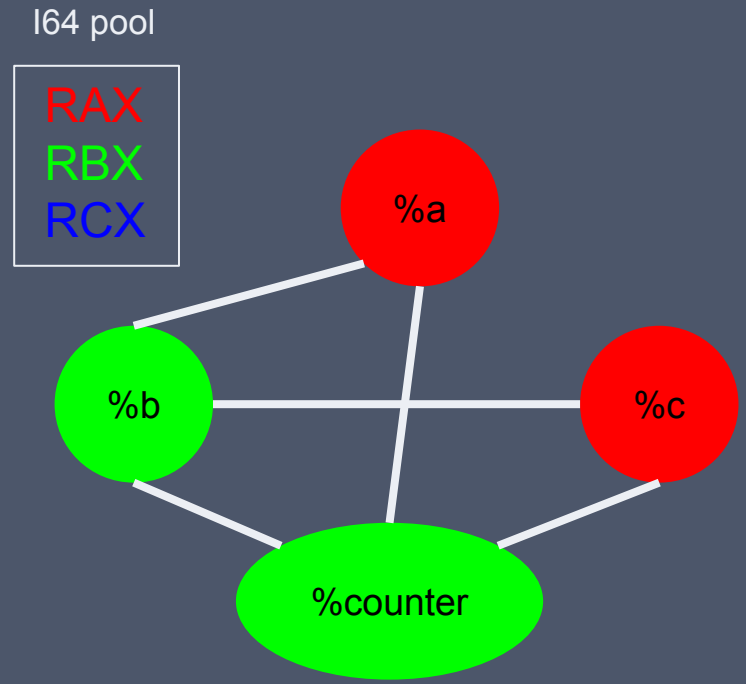
%counter



Each VReg has a dedicated invocation.

Register allocation





Greedy allocation

Each VReg has a dedicated invocation.

Register allocation

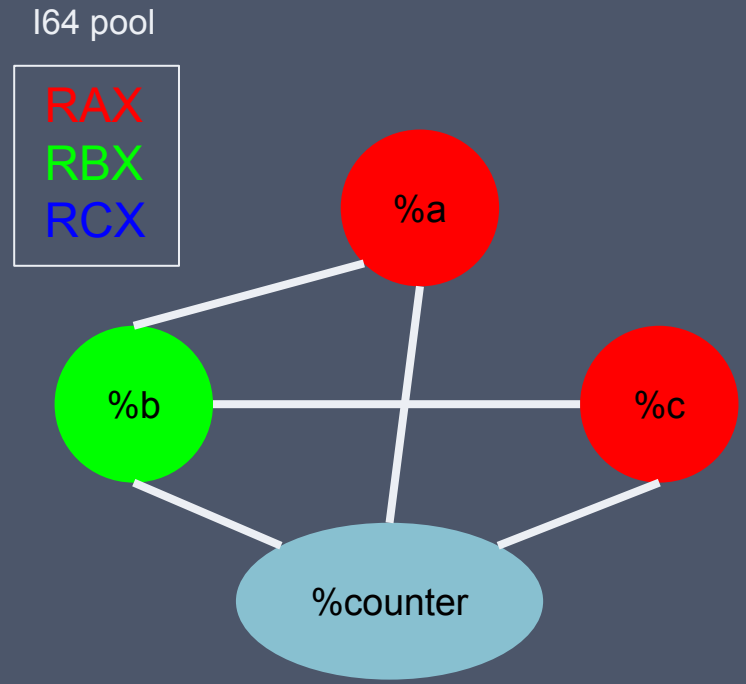
```
0     mov %counter, 20
1     loop:
2     cmp %counter, 0
3     je break:

4     mov %a, 0×4
5     mov %b, 0×1
6     add %a, %b
7     mov %c, %b

8     sub %counter, 1
9     jmp loop:
10    break:
```

%a

%b

%c

%counter



Eviction

Each VReg has a dedicated invocation.

Register allocation

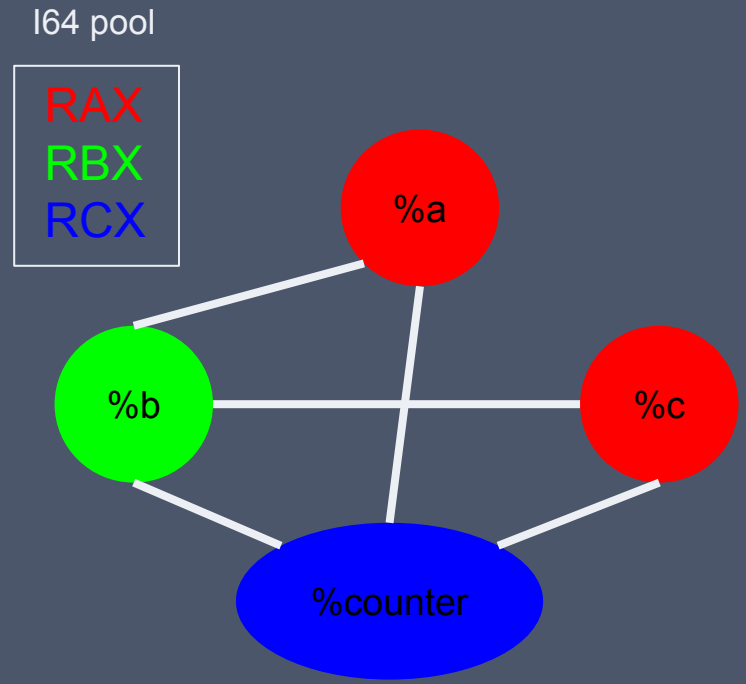
```
0     mov %counter, 20
1     loop:
2     cmp %counter, 0
3     je break:

4     mov %a, 0×4
5     mov %b, 0×1
6     add %a, %b
7     mov %c, %b

8     sub %counter, 1
9     jmp loop:
10    break:
```

%a

%b

%c

%counter



Greedy allocation

Each VReg has a dedicated invocation.

TODO

- Structs and unions
- Enums
- Functions
- Register spilling
- Full x86_64 codegen coverage
- Full C types coverage
- Implicit casting and type precedence
- Float literals
- Inline ASM
- Basic optimisations
- Complex numbers
- Fixing the dozens of edge cases