```
print(a + { a += 1;  a })
```

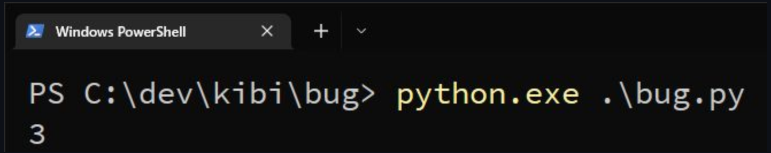
```
var a = 1

print(a + { a += 1;  a })

// -> 4
// a = 1

print(a + { a += 1;  a })

// -> 4
// a = 1

print(1 + { a += 1;  a })

// -> 4
// a = 2

print(1 + { a })

// -> 4
// a = 2

print(1 + 2)

// -> 4
// a = 2

print(3)

// -> 4
```

```python
a = 1

def f():
    global a
    a += 1
    return a

print(a + f())
```

```
var a = 1


print(a + {
    a += 1;
    a
})
```

```python
a = 1

def f():
    global a
    a += 1
    return a

print(a + f())
```

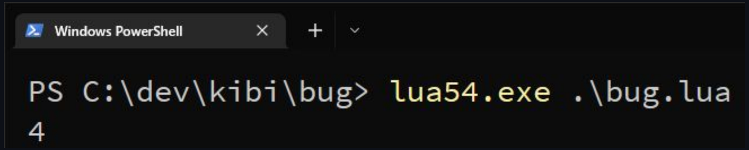
```python
a = 1

def f():
    global a
    a += 1
    return a

print(a + f())
```



```
PS C:\dev\kibi\bug> python.exe .\bug.py
3
```

```lua
local a = 1

function f()
    a = a + 1
    return a
end

print(a + f())
```

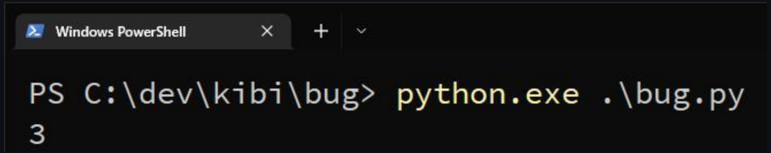
```lua
local a = 1

function f()
    a = a + 1
    return a
end

print(a + f())
```



Windows PowerShell

```
PS C:\dev\kibi\bug> lua54.exe .\bug.lua
4
```

```python
a = 1

def f():
    global a
    a += 1
    return a

print(a + f())
```



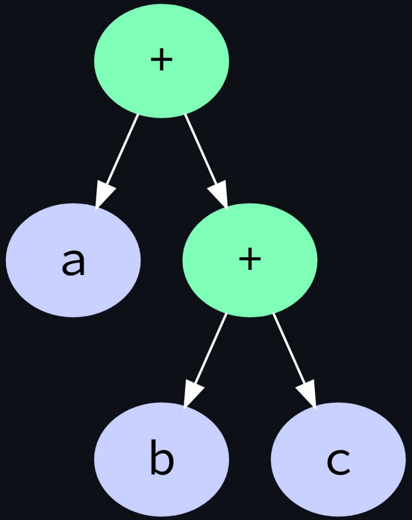
```
PS C:\dev\kibi\bug> python.exe .\bug.py
3
```

source text

```
"a + (b + c)"
```

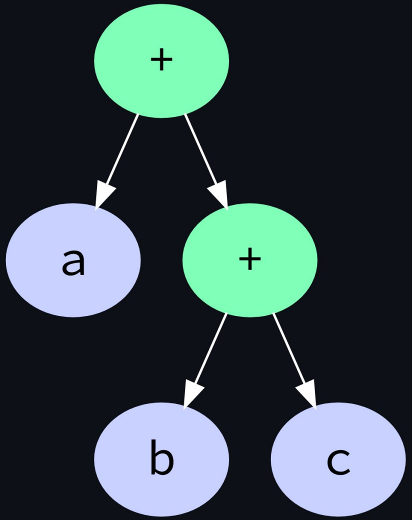
$$\begin{tikzcd} \boxed{\text{source text}} \arrow[r, "\text{parser}"] & \boxed{\begin{array}{c}\text{abstract}\\ \text{syntax tree}\end{array}} \end{tikzcd}$$

```
"a + (b + c)"
```

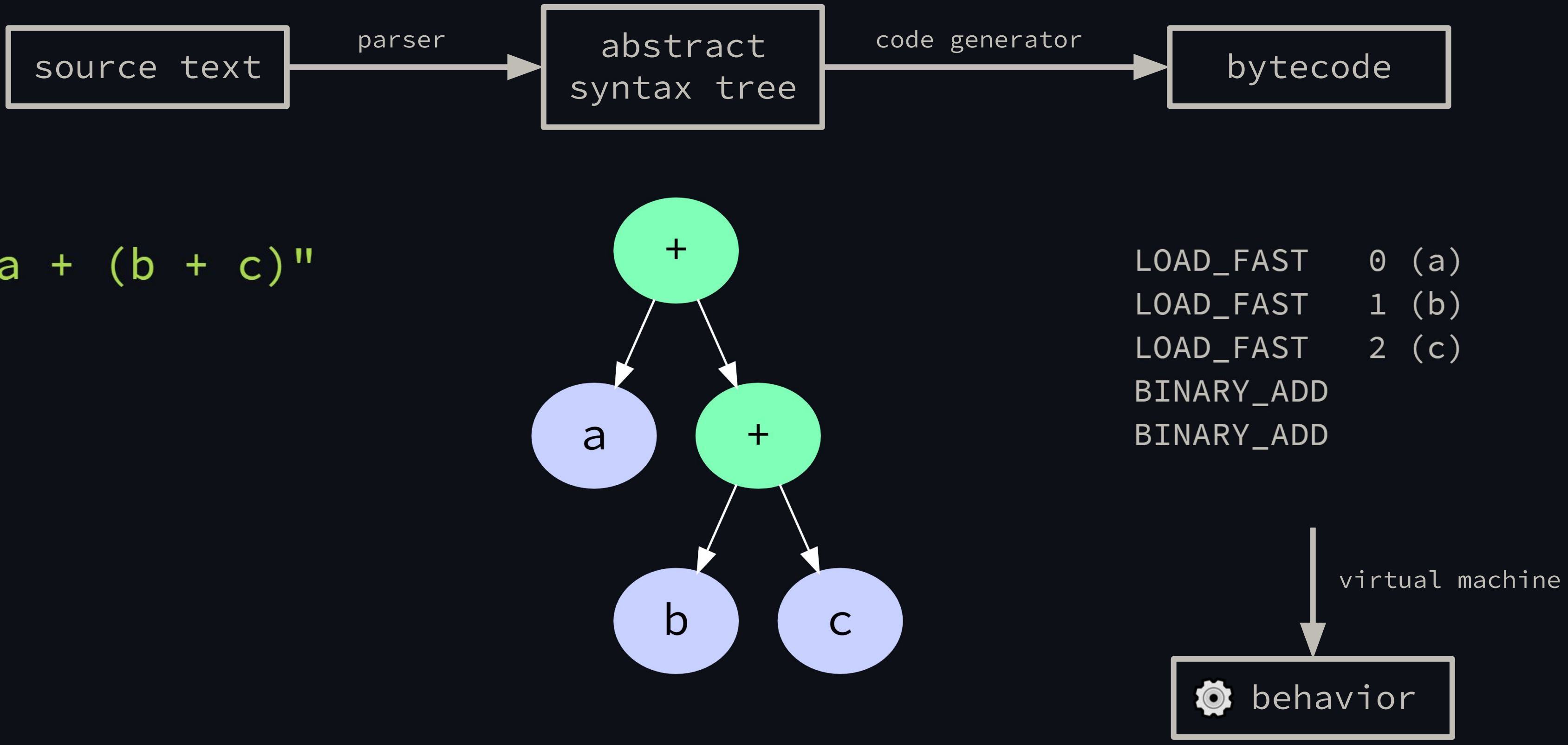


$$\begin{tikzcd}
\boxed{\text{source text}} \arrow[r, "\text{parser}"] & \boxed{\text{abstract syntax tree}} \arrow[r, "\text{code generator}"] & \boxed{\text{bytecode}}
\end{tikzcd}$$

```
"a + (b + c)"
```



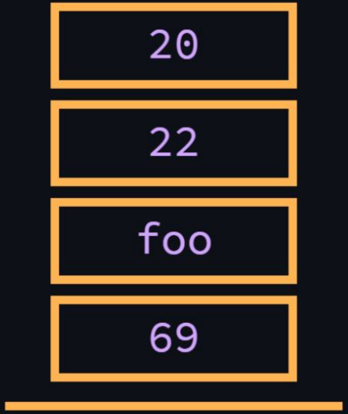
```
LOAD_FAST    0 (a)
LOAD_FAST    1 (b)
LOAD_FAST    2 (c)
BINARY_ADD
BINARY_ADD
```



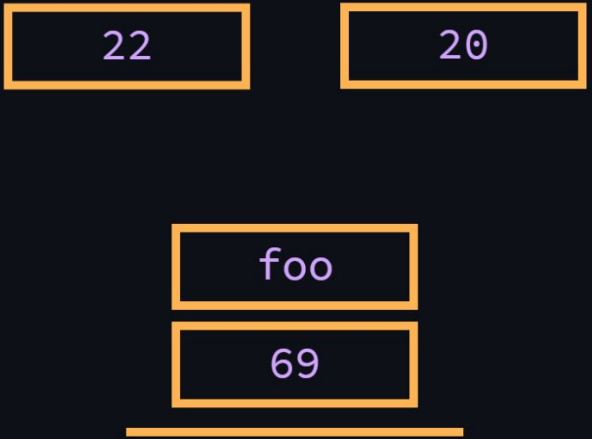
"a + (b + c)"

```
LOAD_FAST    0 (a)
LOAD_FAST    1 (b)
LOAD_FAST    2 (c)
BINARY_ADD
BINARY_ADD
```

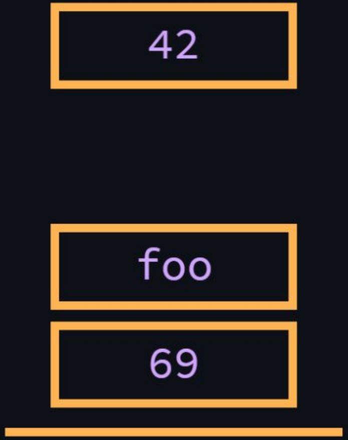
BINARY_ADD



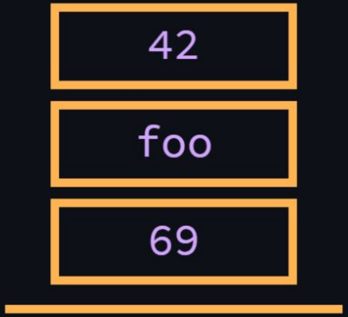
BINARY_ADD



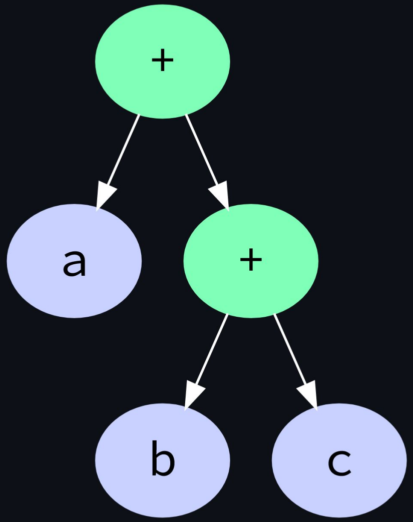
BINARY_ADD

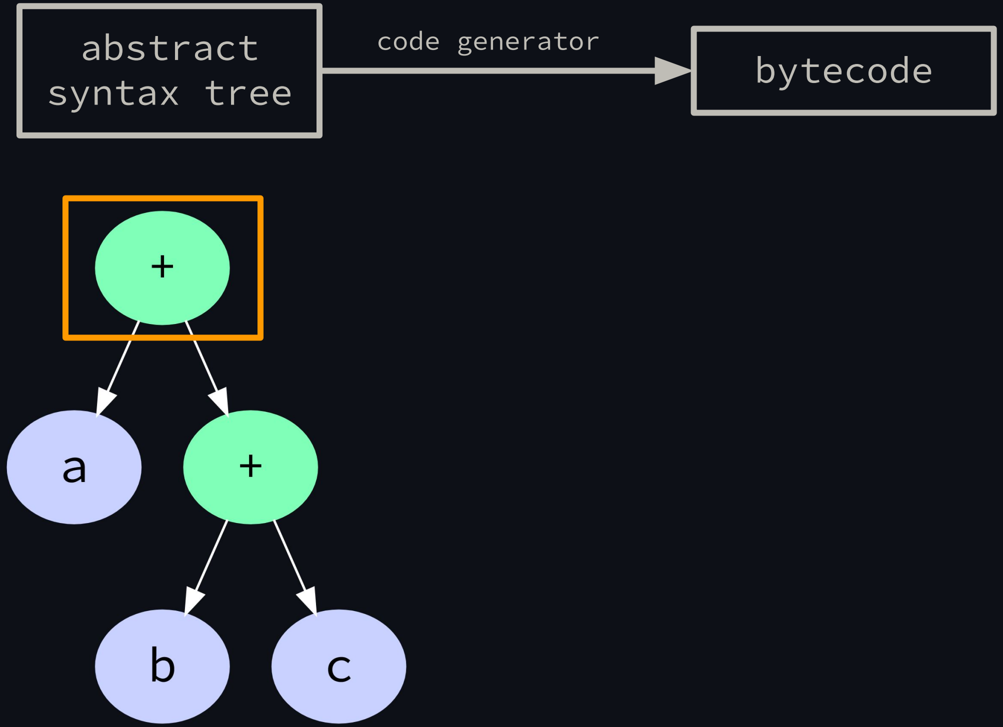


BINARY_ADD

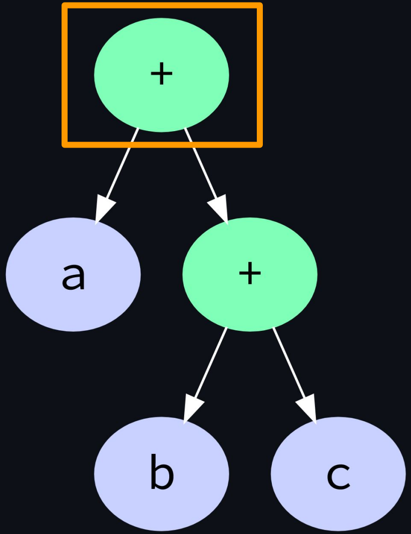


$$\begin{tikzcd}\boxed{\text{abstract syntax tree}} \arrow[r, "\text{code generator}"] & \boxed{\text{bytecode}}\end{tikzcd}$$

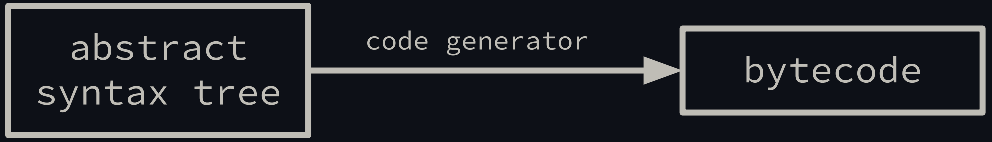


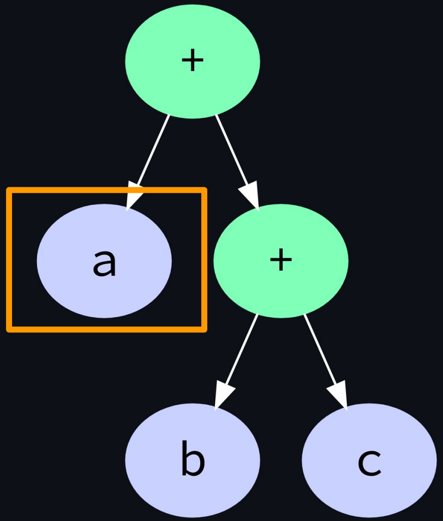

abstract syntax tree
code generator
bytecode
+
a
+
b
c

$$\begin{tikzcd}\boxed{\text{abstract syntax tree}} \arrow[r, "\text{code generator}"] & \boxed{\text{bytecode}}\end{tikzcd}$$

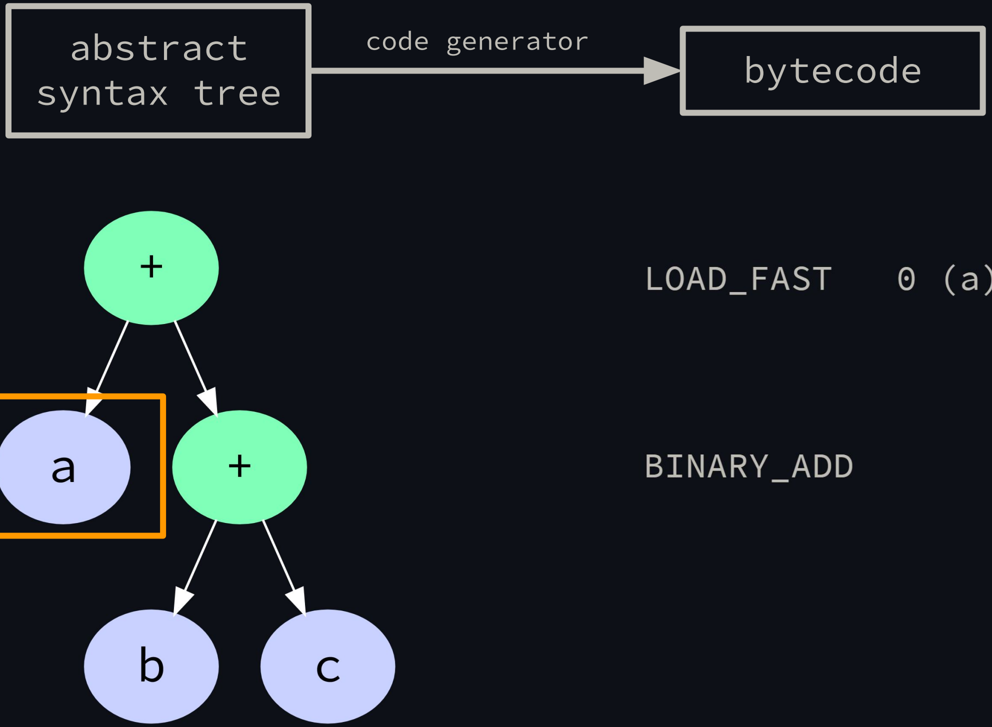


```
BINARY_ADD
```



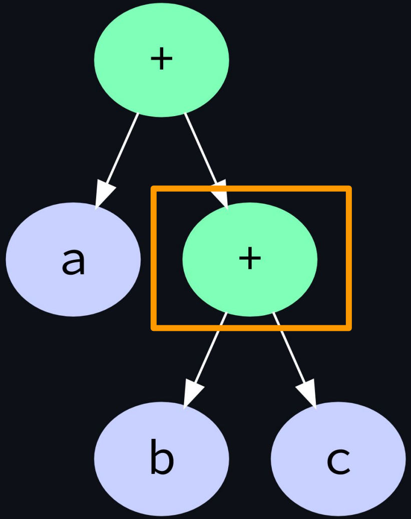


```
BINARY_ADD
```

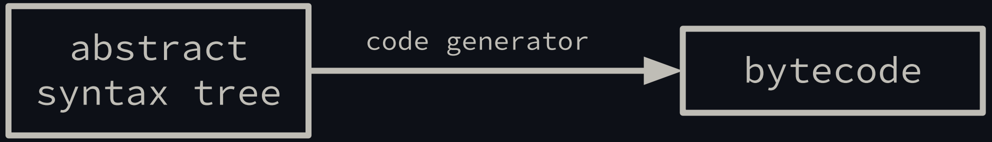


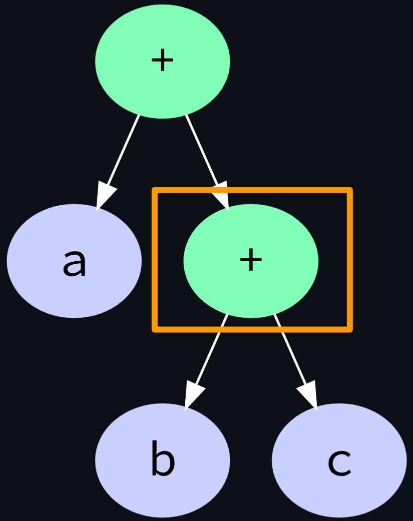
```
LOAD_FAST    0 (a)


BINARY_ADD
```

$$\begin{tikzcd}\boxed{\text{abstract syntax tree}} \arrow[r, "\text{code generator}"] & \boxed{\text{bytecode}}\end{tikzcd}$$

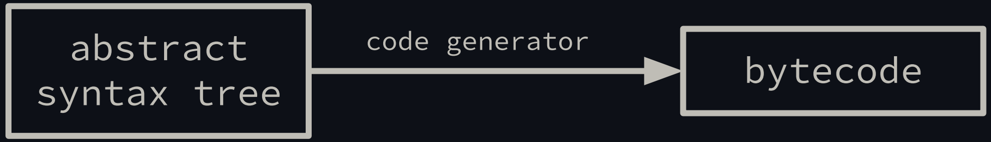


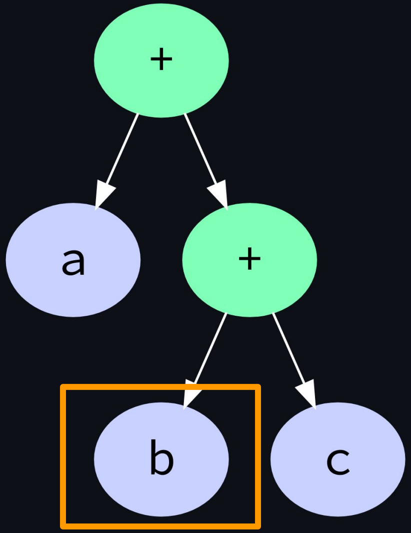
```
LOAD_FAST    0 (a)


BINARY_ADD
```



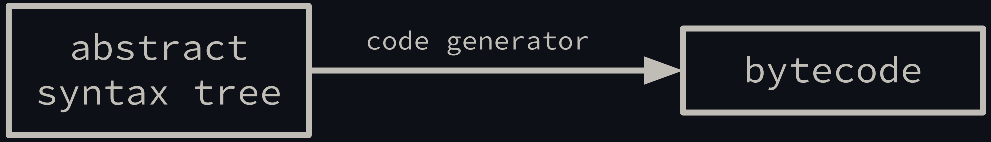


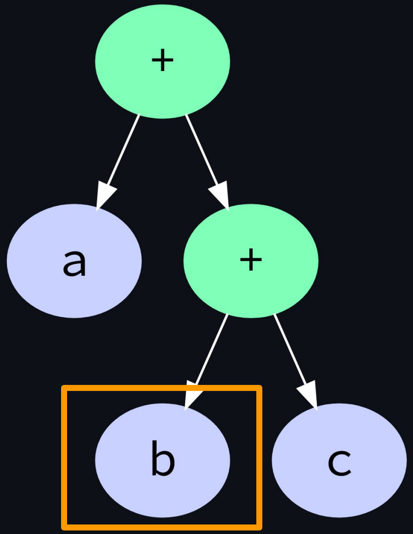
```
LOAD_FAST    0 (a)


BINARY_ADD
BINARY_ADD
```





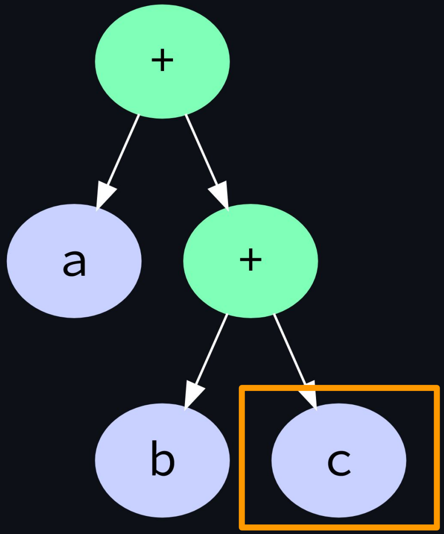
```
LOAD_FAST    0 (a)


BINARY_ADD
BINARY_ADD
```





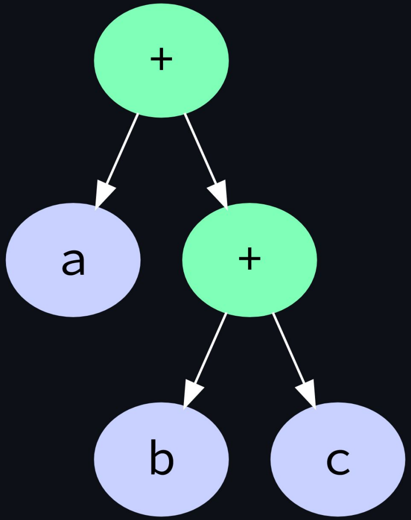
```
LOAD_FAST    0 (a)
LOAD_FAST    1 (b)

BINARY_ADD
BINARY_ADD
```

$$\begin{tikzcd}\boxed{\text{abstract syntax tree}} \arrow[r, "\text{code generator}"] & \boxed{\text{bytecode}}\end{tikzcd}$$

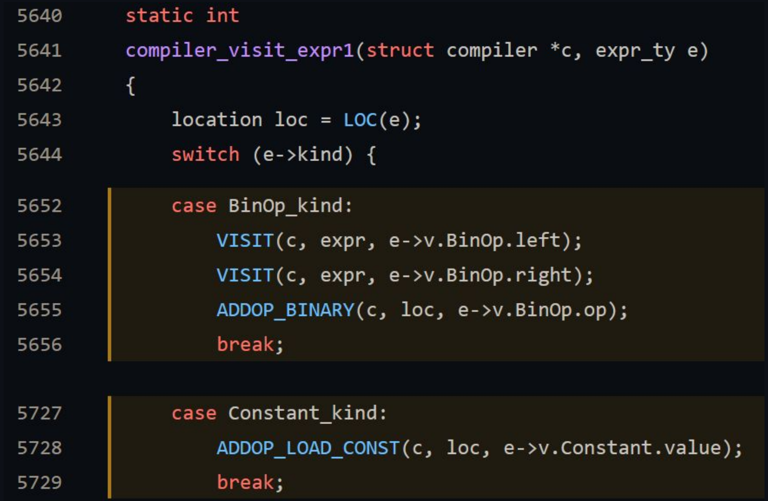


```
LOAD_FAST    0 (a)
LOAD_FAST    1 (b)
LOAD_FAST    2 (c)
BINARY_ADD
BINARY_ADD
```

$$\begin{tikzcd}\boxed{\text{abstract syntax tree}} \arrow[r, "\text{code generator}"] & \boxed{\text{bytecode}}\end{tikzcd}$$



```
LOAD_FAST    0 (a)
LOAD_FAST    1 (b)
LOAD_FAST    2 (c)
BINARY_ADD
BINARY_ADD
```



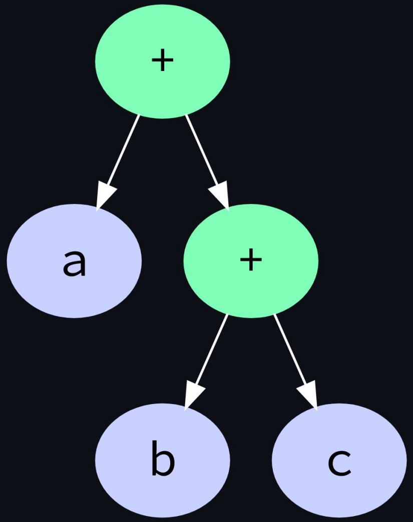
```c
static int
compiler_visit_expr1(struct compiler *c, expr_ty e)
{
    location loc = LOC(e);
    switch (e->kind) {

    case BinOp_kind:
        VISIT(c, expr, e->v.BinOp.left);
        VISIT(c, expr, e->v.BinOp.right);
        ADDOP_BINARY(c, loc, e->v.BinOp.op);
        break;

    case Constant_kind:
        ADDOP_LOAD_CONST(c, loc, e->v.Constant.value);
        break;
```

```python
a = 1

def f():
    global a
    a += 1
    return a

print(a + f())
```

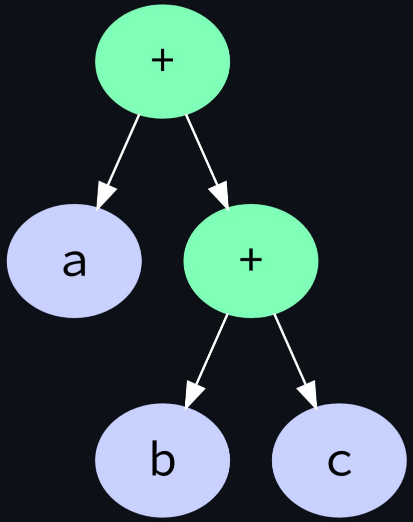
```python
a = 1

def f():
    global a
    a += 1
    return a

print(a + f())
```

```python
a = 1

def f():
    global a
    a += 1
    return a

print(a + f())
```

```
LOAD_GLOBAL     1 (a)
LOAD_GLOBAL     2 (f)
CALL_FUNCTION   0
BINARY_ADD
```

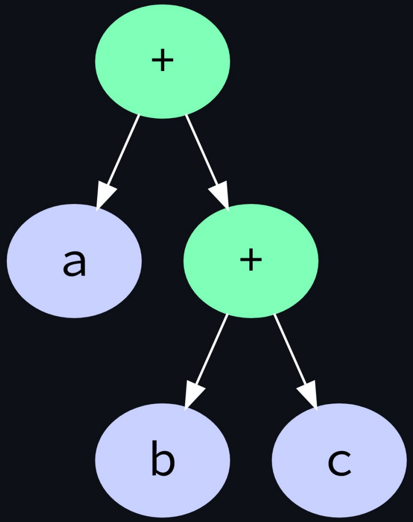


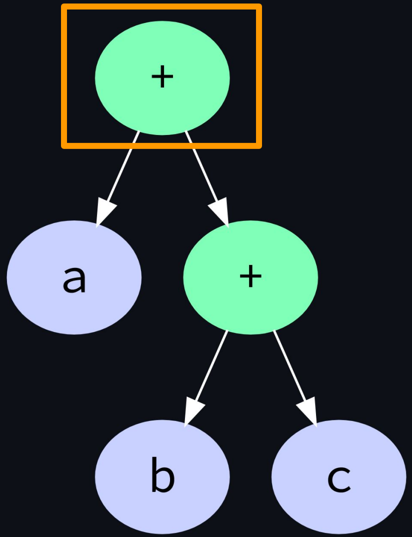
```
LOAD_FAST    0 (a)
LOAD_FAST    1 (b)
LOAD_FAST    2 (c)
BINARY_ADD
BINARY_ADD
```

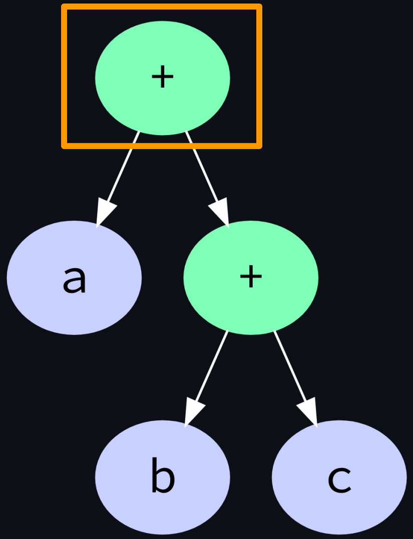


```
LOAD_FAST    0 (a)
LOAD_FAST    1 (b)
LOAD_FAST    2 (c)
BINARY_ADD
BINARY_ADD
```

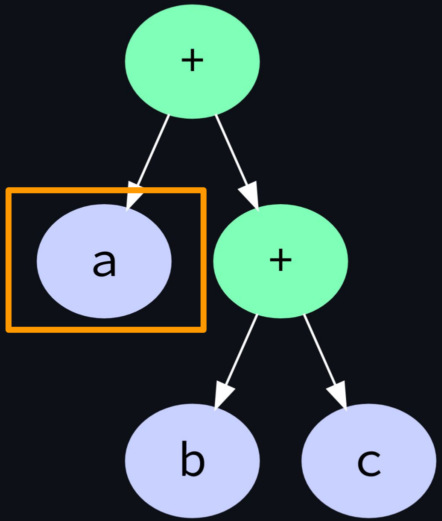
```
ADD r3, r1(b), r2(c)
ADD r4, r0(a), r3
```


+
a
+
b
c

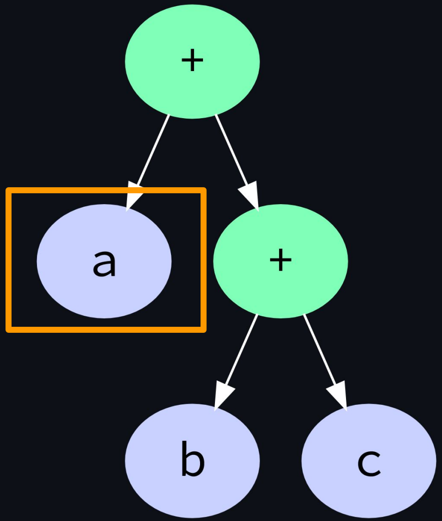

+
a
+
b
c



```
ADD r?, r?, r?
```

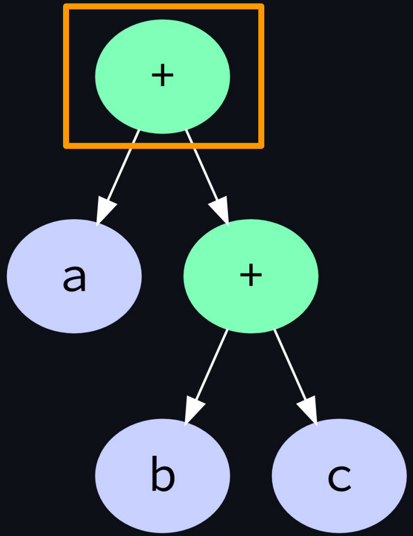


```
ADD r?, r?, r?
```



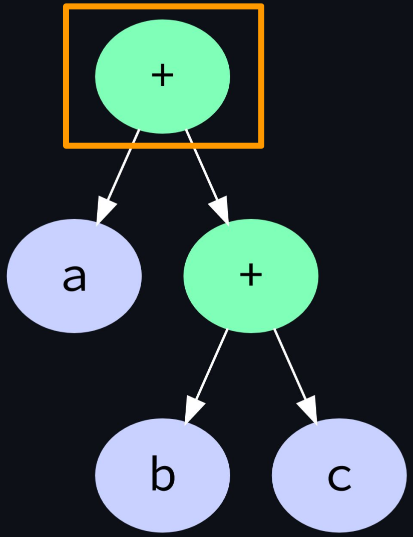
```
a: r0
b: r1
c: r2
```

```
ADD r?, r?, r?
```



```
a: r0
b: r1
c: r2
```

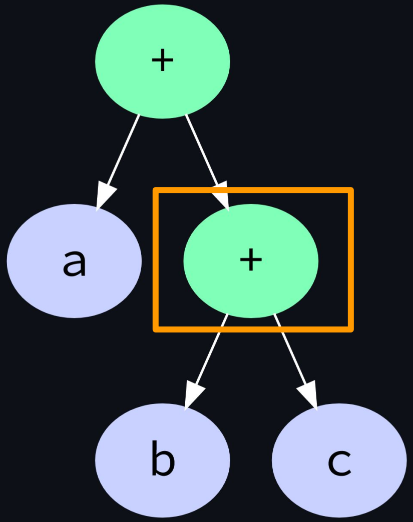
```
ADD r?, r?, r?
```



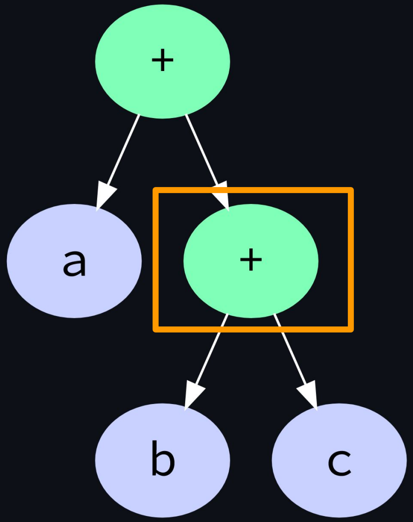
```
a: r0
b: r1
c: r2
```

```
ADD r?, r0(a), r?
```

```
a: r0
b: r1
c: r2
```

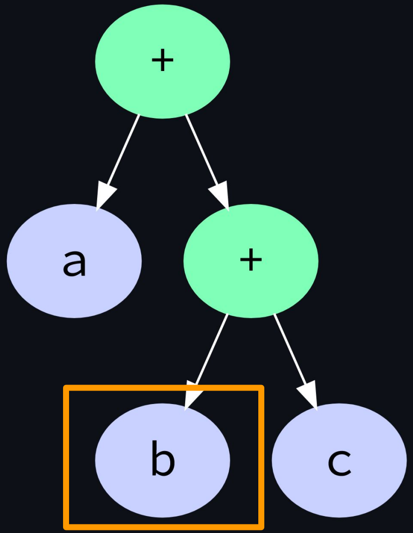


```
ADD r?, r0(a), r?
```



```
a: r0
b: r1
c: r2
```

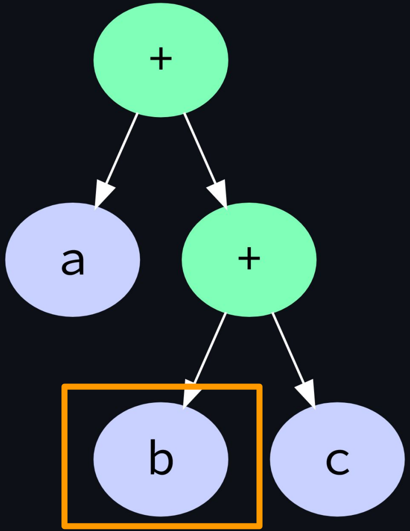
```
ADD r?, r?, r?
ADD r?, r0(a), r?
```



```
a: r0
b: r1
c: r2
```

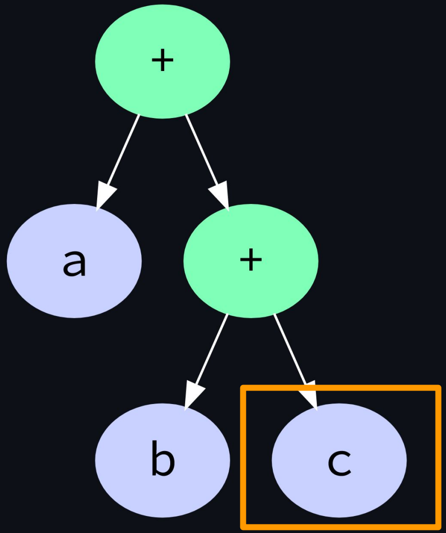
```
ADD r?, r?, r?
ADD r?, r0(a), r?
```

```
a: r0
b: r1
c: r2
```



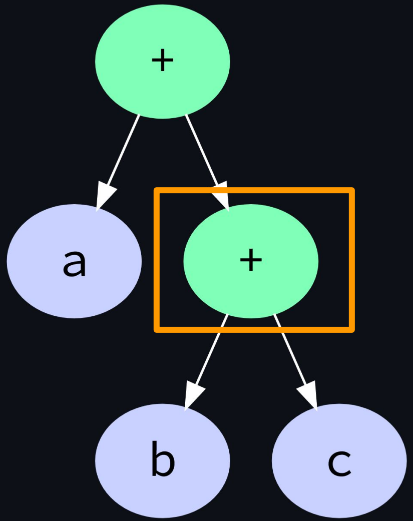
```
ADD r?, r1(b), r?
ADD r?, r0(a), r?
```

```
a: r0
b: r1
c: r2
```

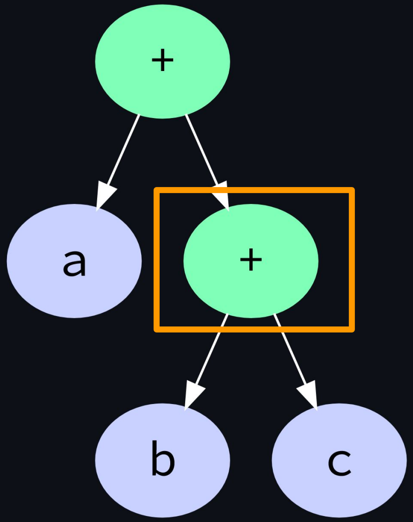


```
ADD r?, r1(b), r2(c)
ADD r?, r0(a), r?
```

```
a: r0
b: r1
c: r2
```

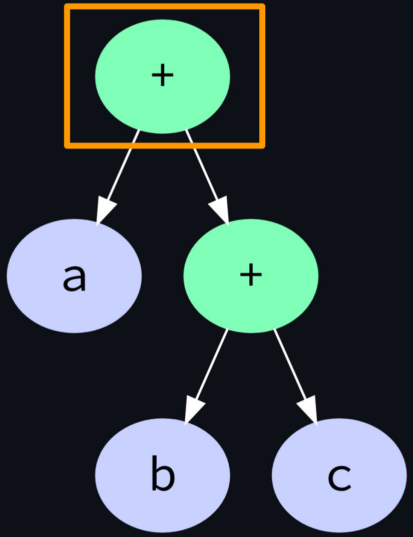


```
ADD r?, r1(b), r2(c)
ADD r?, r0(a), r?
```



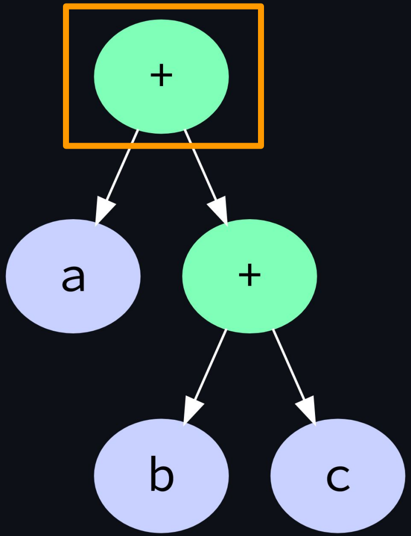
```
a: r0
b: r1
c: r2
```

```
ADD r3, r1(b), r2(c)
ADD r?, r0(a), r?
```



```
a: r0
b: r1
c: r2
```

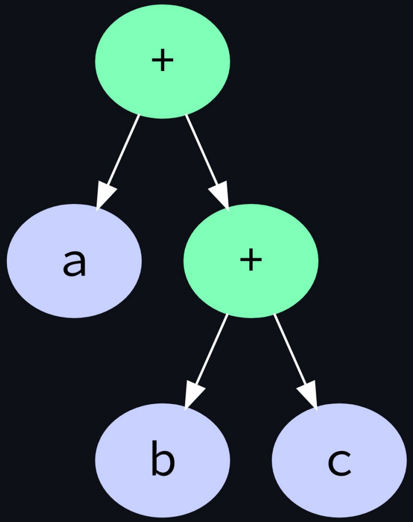
```
ADD r3, r1(b), r2(c)
ADD r?, r0(a), r3
```



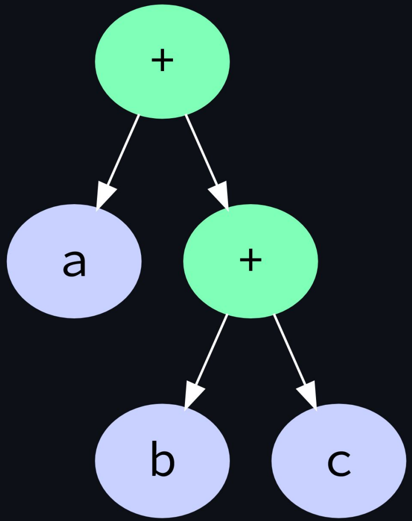
```
a: r0
b: r1
c: r2
```

```
ADD r3, r1(b), r2(c)
ADD r4, r0(a), r3
```

```
a: r0
b: r1
c: r2
```



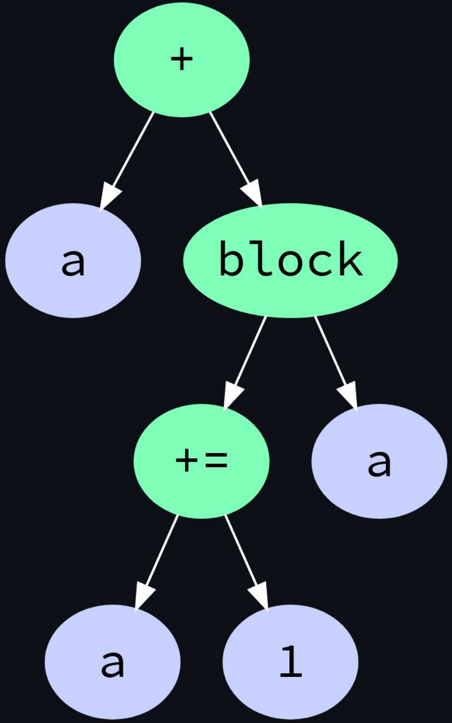
```
ADD r3, r1(b), r2(c)
ADD r4, r0(a), r3
```



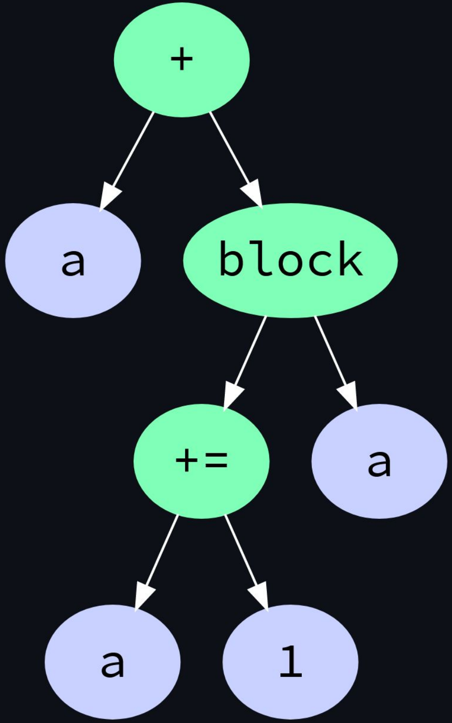
```
LOAD_FAST    0 (a)
LOAD_FAST    1 (b)
LOAD_FAST    2 (c)
BINARY_ADD
BINARY_ADD
```

```
ADD r3, r1(b), r2(c)
ADD r4, r0(a), r3
```

```
a + { a += 1;  a }
```

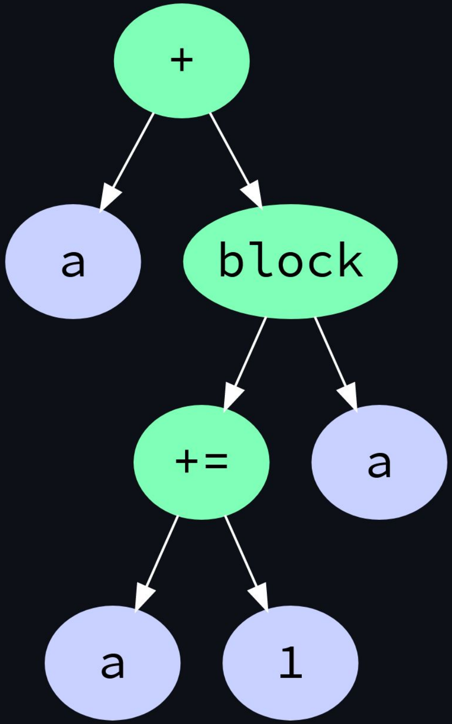


```
a + { a += 1;  a }
```



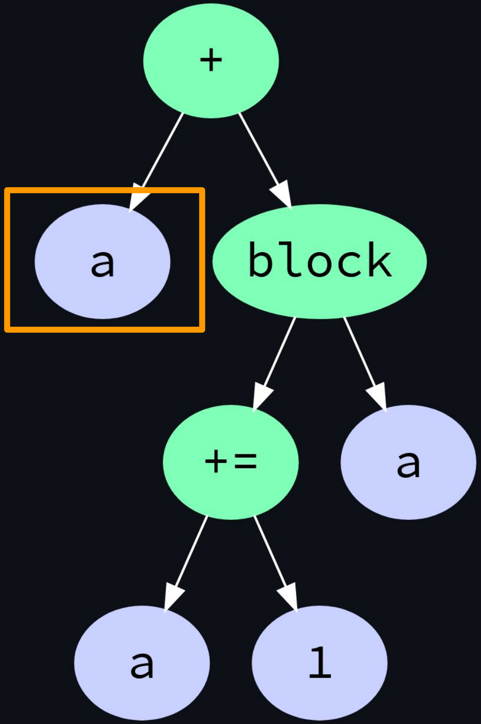
```
ADD r0(a), r0(a), #1
ADD r1, r0(a), r0(a)
```

```
a + { a += 1;  a }
```



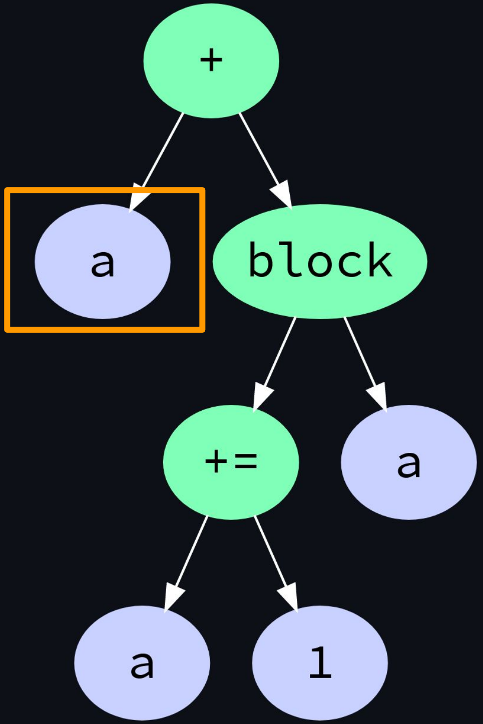
```
ADD r0(a), r0(a), #1
ADD r1, r0(a), r0(a)
```

```
a + { a += 1;  a }
```

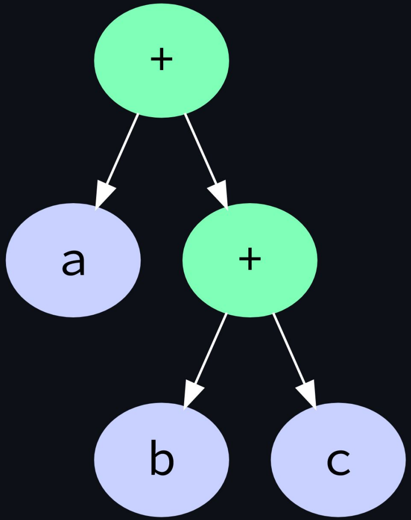


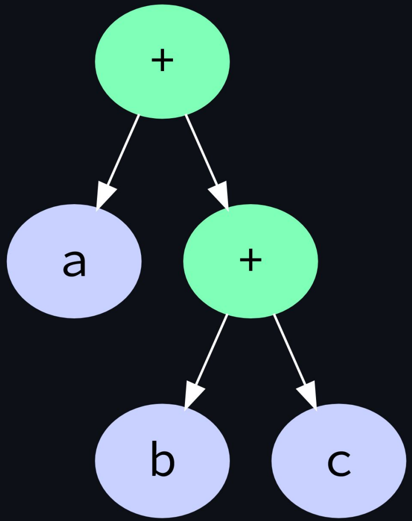
```
ADD r0(a), r0(a), #1
ADD r1, r0(a), r0(a)
```

```
a + { a += 1;  a }
```

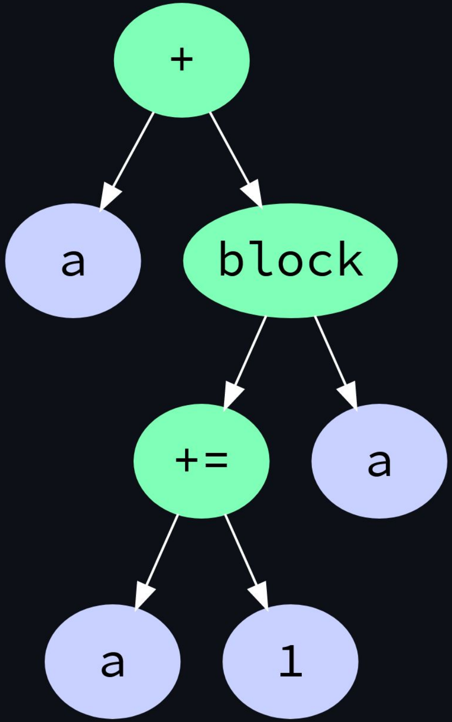


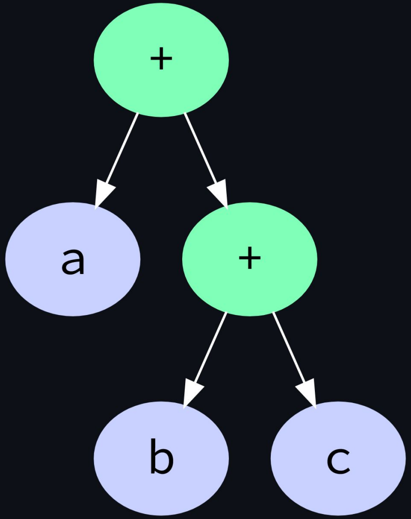
```
COPY r1, r0(a)
ADD r0(a), r0(a), #1
ADD r1, r1, r0(a)
```

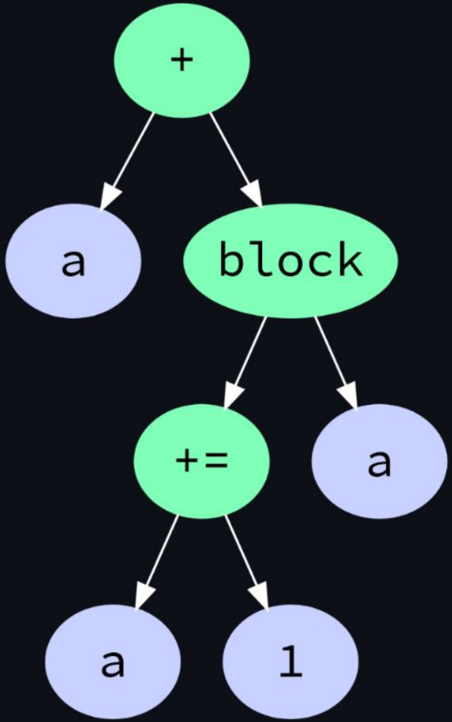

+
a
+
b
c

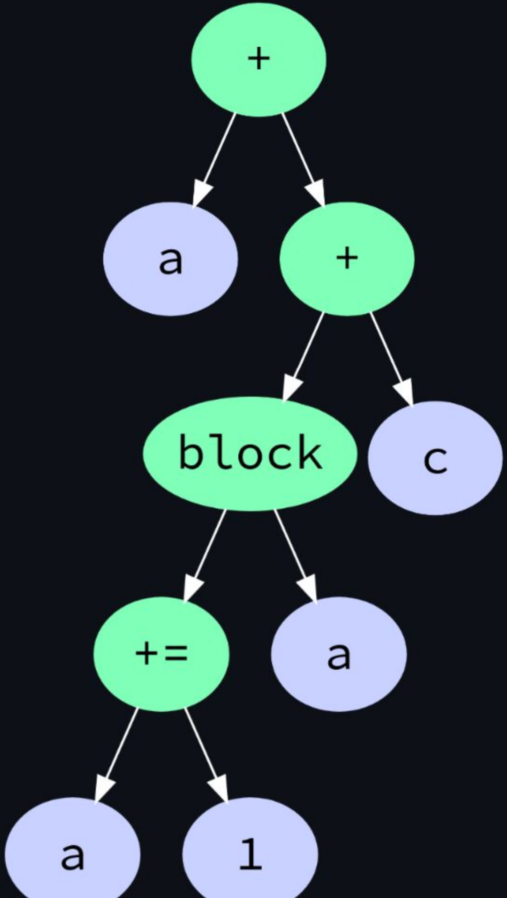


```
COPY r3, r0(a)
COPY r4, r1(b)
COPY r5, r2(c)
ADD r4, r4, r5
ADD r3, r3, r4
```


+
a
block
+=
a
a
1


+
a
+
b
c


+
a
block
+=
a
a
1

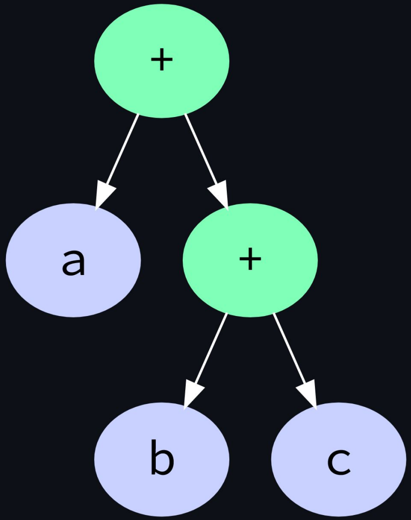

+
a
+
block
c
+=
a
a
1

```
binary op:  a + { a += 1; a }
binary op:  a + { a += 1; a }
tuple:      (a, { a += 1; a })
```

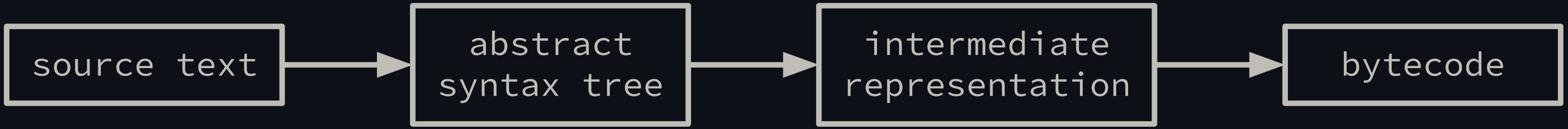
```
binary op:  a + { a += 1; a }
tuple:      (a, { a += 1; a })
call:       foo(a, { a += 1; a })
```

$$\begin{tikzcd}\boxed{\text{source text}} \arrow[r, "\text{parser}"] & \boxed{\text{abstract syntax tree}} \arrow[r, "\text{code generator}"] & \boxed{\text{bytecode}}\end{tikzcd}$$

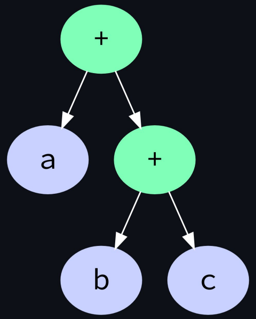
"a + (b + c)"



```
ADD r3, r1(b), r2(c)
ADD r4, r0(a), r3
```



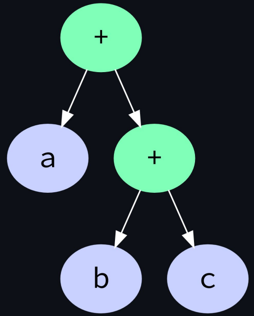
"a + (b + c)"



```
ADD r3, r1(b), r2(c)
ADD r4, r0(a), r3
```

$$\begin{tikzcd}\boxed{\text{source text}} \arrow[r] & \boxed{\text{abstract syntax tree}} \arrow[r] & \boxed{\text{intermediate representation}} \arrow[r] & \boxed{\text{bytecode}}\end{tikzcd}$$

```
"a + (b + c)"
```

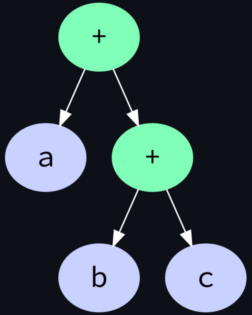


```
i3 = add b, c
i4 = add a, i3
```

```
ADD r3, r1(b), r2(c)
ADD r4, r0(a), r3
```

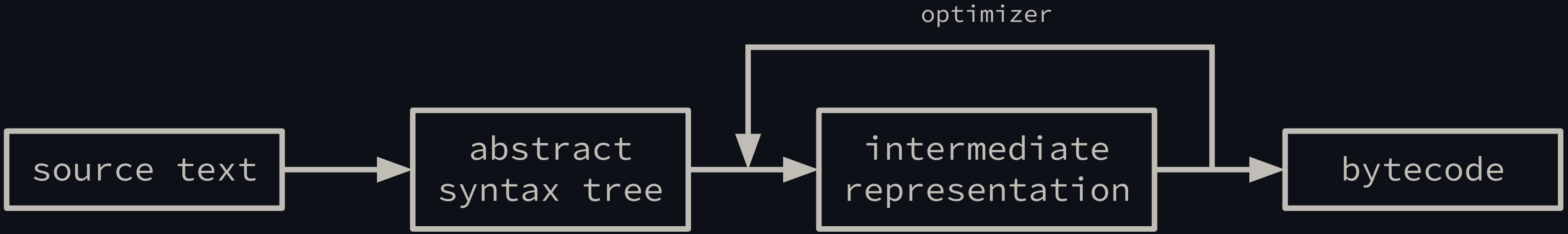
$$\begin{tikzcd}\boxed{\text{source text}} \arrow[r] & \boxed{\begin{array}{c}\text{abstract}\\ \text{syntax tree}\end{array}} \arrow[r] & \boxed{\begin{array}{c}\text{intermediate}\\ \text{representation}\end{array}} \arrow[r] & \boxed{\text{bytecode}}\end{tikzcd}$$

```
"a + (b + c)"
```

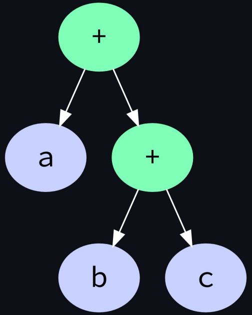


```
i0 = a
i1 = b
i2 = c
i3 = add i1, i2
i4 = add i0, i3
```

```
ADD r3, r1(b), r2(c)
ADD r4, r0(a), r3
```

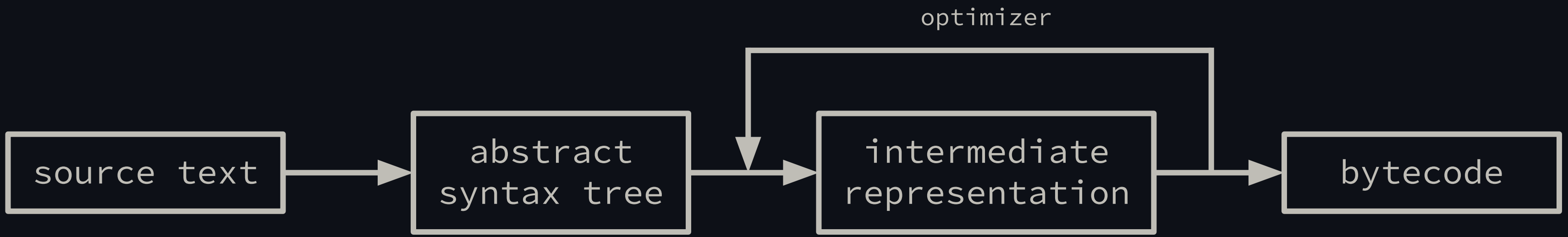


```
"a + (b + c)"
```

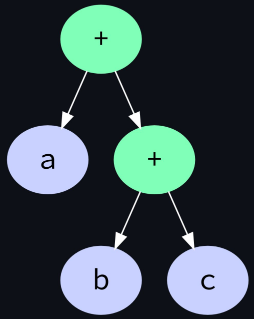


```
i0 = a
i1 = b
i2 = c
i3 = add i1, i2
i4 = add i0, i3
```

```
ADD r3, r1(b), r2(c)
ADD r4, r0(a), r3
```

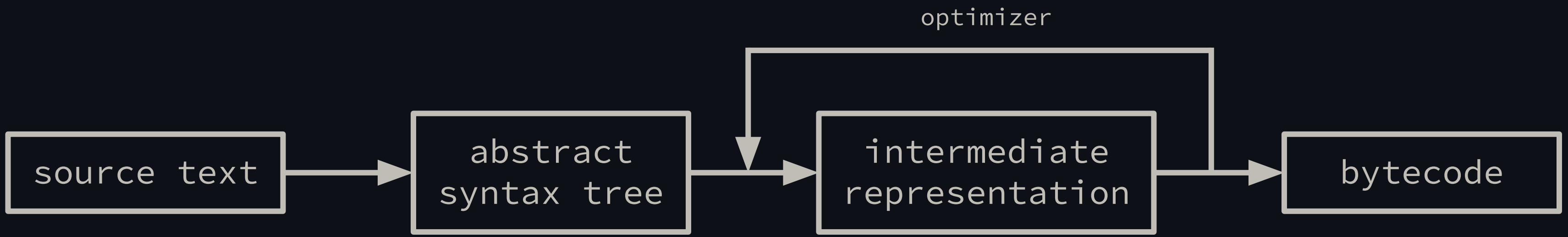


```
"a + (b + c)"
```

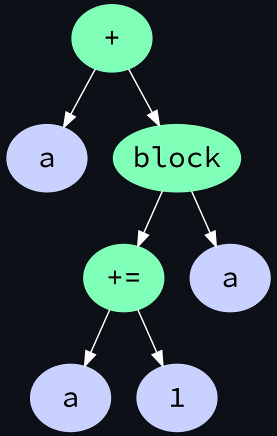


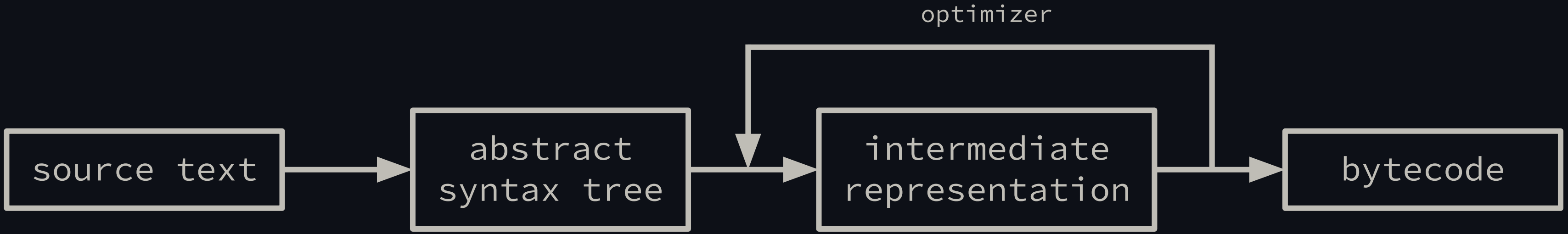
```
i3 = add b, c
i4 = add a, i3
```

```
ADD r3, r1(b), r2(c)
ADD r4, r0(a), r3
```

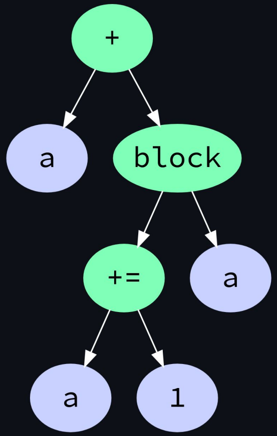

optimizer
source text
abstract syntax tree
intermediate representation
bytecode

"a + (b + c)"

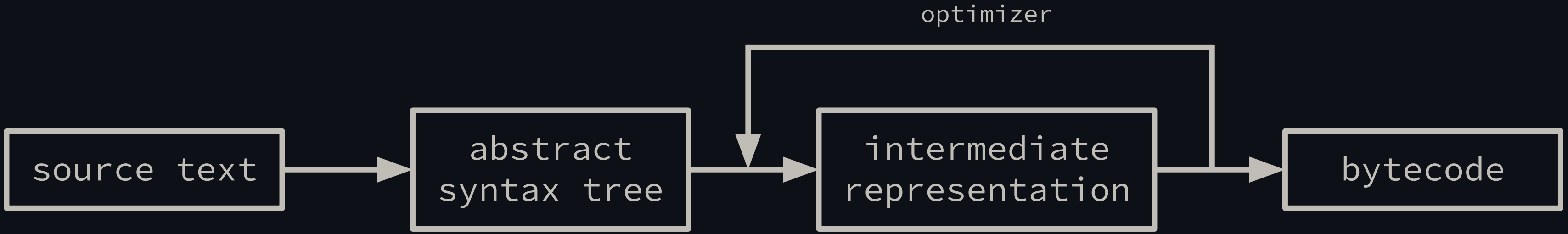

+
a
block
+=
a
a
1



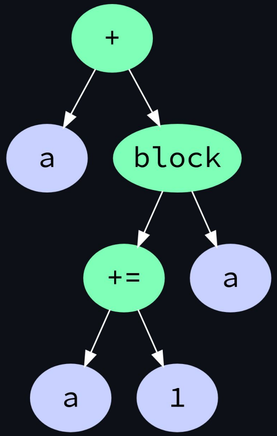
```
"a + (b + c)"
```



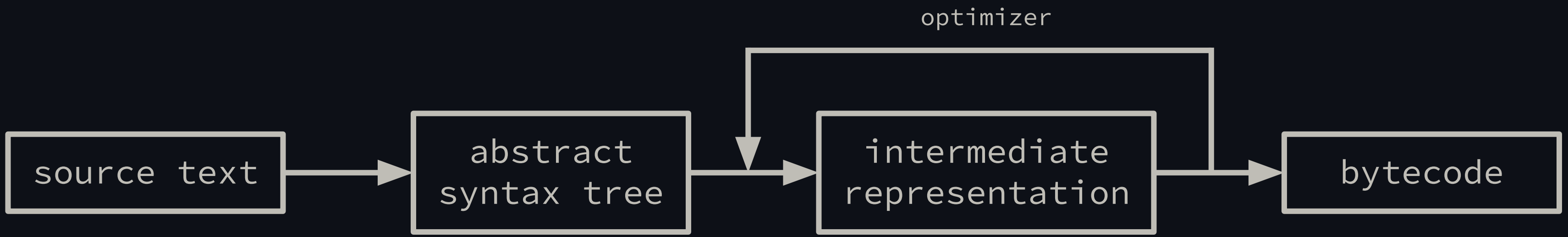
```
i0 = a
a  = add a, #1
i1 = a
i2 = i0 + i1
```



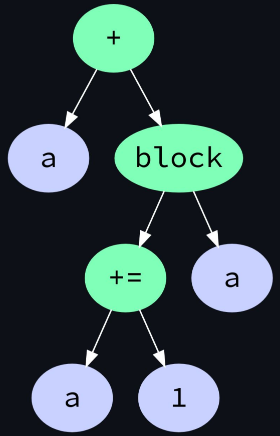
"a + (b + c)"



```
i0 = a
a  = add a, #1
i2 = i0 + a
```



```
"a + (b + c)"
```



```
i0 = a
a  = add a, #1
i2 = i0 + a
```

```
COPY r1, r0(a)
ADD  r0(a), r0(a), #1
ADD  r1, r1, r0(a)
```

```
i0 = a
a  = add a, #1
i1 = a
i2 = i0 + i1
```

$\xrightarrow{???}$

```
i0 = a
a  = add a, #1
i2 = i0 + a
```

```
i0 = a

...

i5 = add i0, #7
```

```
i0 = a

... // no `a = …`

i5 = add i0, #7
```

```
i0 = a

... // no `a = …`

i5 = add i0, #7
```

```
i0 = a

...

i5 = add a, #7
```

```
i0 = a

... // no `a = …`

i5 = add i0, #7
```

```
i0 = a

... // no `… = i0`

i5 = add a, #7
```

```
i0 = a
... // no `a = …`
i5 = add i0, #7
```

```
i0 = a
... // no `… = i0`
i5 = add a, #7
```

```
// yoink
...
i5 = add a, #7
```

```
a + { a += 1;  a }
```

```
a + { a += 1;  a }
```

```
i0 = a
a  = add a, #1
i1 = a
i2 = add i0, i1
```

```
a + { a += 1;  a }
```

```
i0 = a
a  = add a, #1
i1 = a
i2 = add i0, a
```

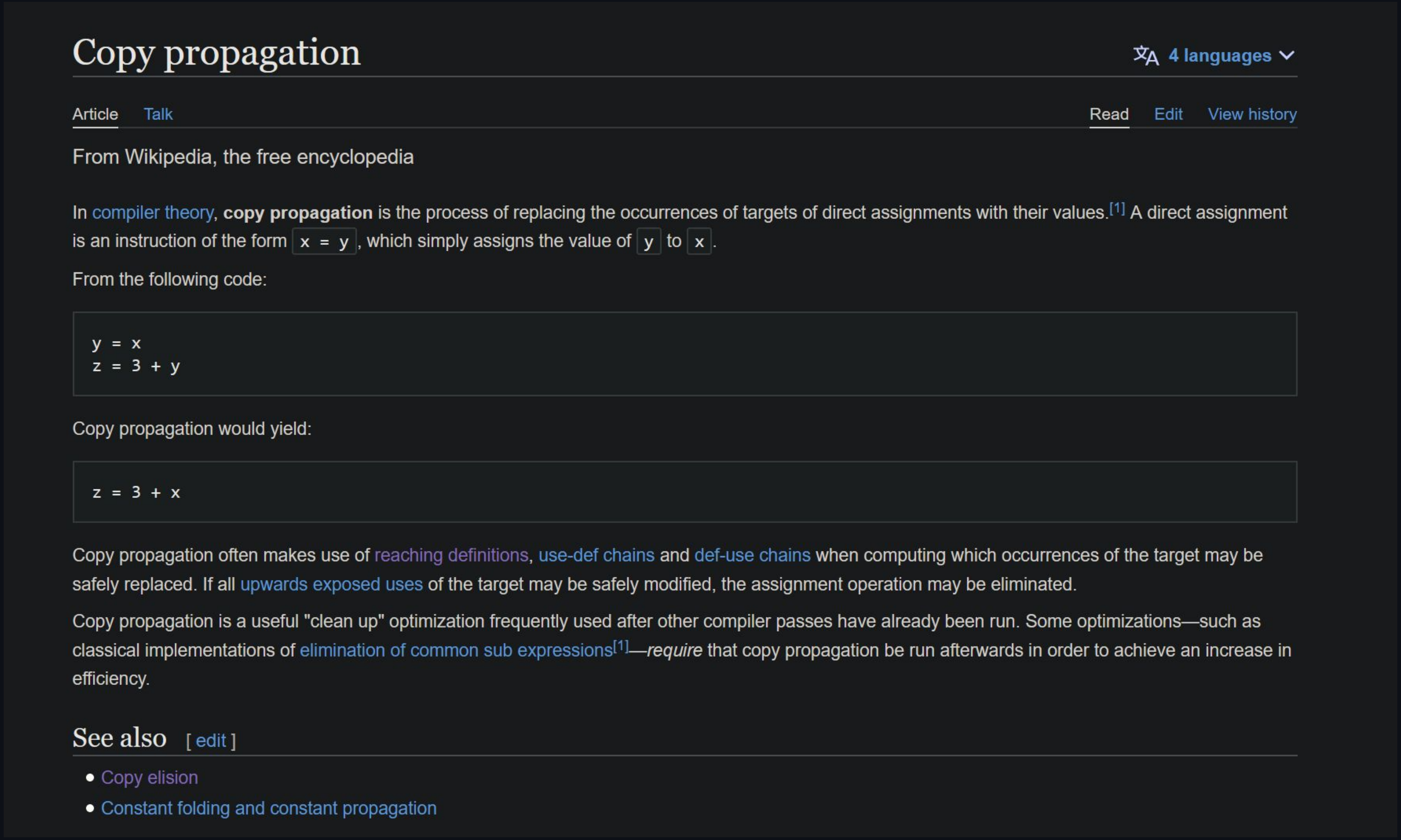
```
a + { a += 1;  a }
```

```
i0 = a
a  = add a, #1
i2 = add i0, a
```

```
i0 = a

...

i5 = add i0, #7
```

```
i0 = a

...

...

...

i5 = add i0, #7
```

```
    i0 = a

    ...

    if i2 then L7 else L8
L7: a = add a, #1
L8: ...

    i5 = add i0, #7
```



Copy propagation

4 languages

Article Talk Read Edit View history

From Wikipedia, the free encyclopedia

In compiler theory, **copy propagation** is the process of replacing the occurrences of targets of direct assignments with their values.[1] A direct assignment is an instruction of the form `x = y`, which simply assigns the value of `y` to `x`.

From the following code:

```
y = x
z = 3 + y
```

Copy propagation would yield:

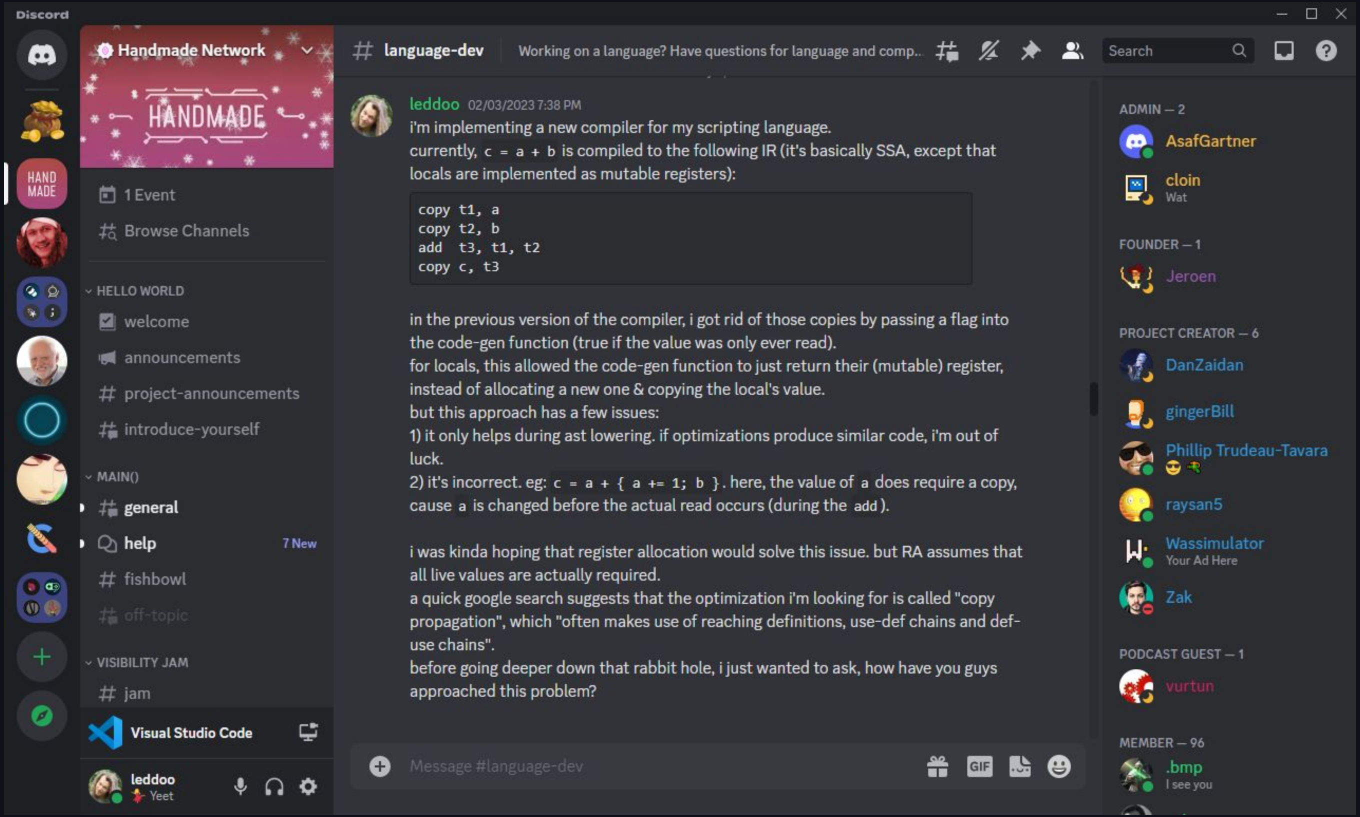
```
z = 3 + x
```

Copy propagation often makes use of reaching definitions, use-def chains and def-use chains when computing which occurrences of the target may be safely replaced. If all upwards exposed uses of the target may be safely modified, the assignment operation may be eliminated.

Copy propagation is a useful "clean up" optimization frequently used after other compiler passes have already been run. Some optimizations—such as classical implementations of elimination of common sub expressions[1]—*require* that copy propagation be run afterwards in order to achieve an increase in efficiency.

See also [edit]

- Copy elision
- Constant folding and constant propagation



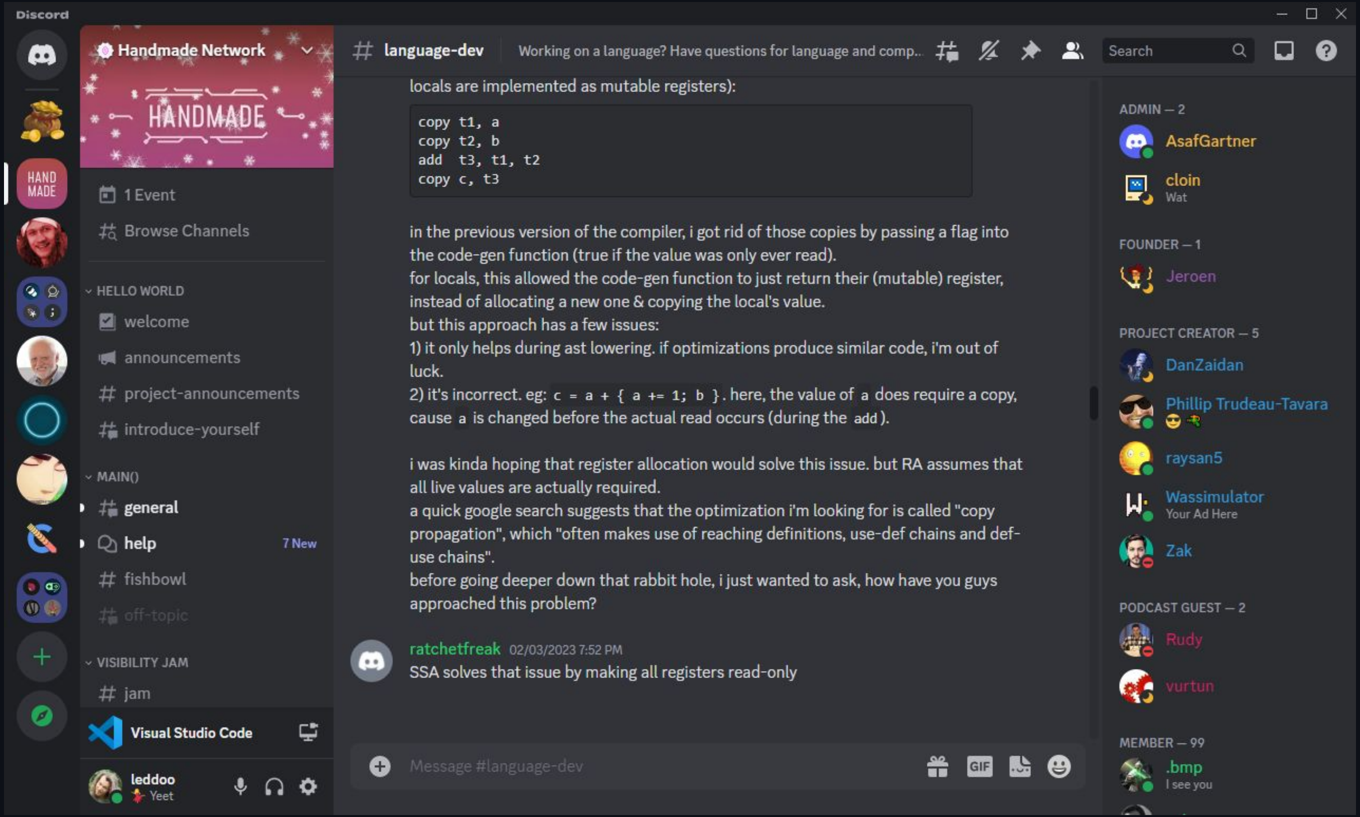
leddoo 02/03/2023 7:38 PM

i'm implementing a new compiler for my scripting language.
currently, `c = a + b` is compiled to the following IR (it's basically SSA, except that locals are implemented as mutable registers):

```
copy t1, a
copy t2, b
add  t3, t1, t2
copy c, t3
```

in the previous version of the compiler, i got rid of those copies by passing a flag into the code-gen function (true if the value was only ever read).
for locals, this allowed the code-gen function to just return their (mutable) register, instead of allocating a new one & copying the local's value.
but this approach has a few issues:
1) it only helps during ast lowering. if optimizations produce similar code, i'm out of luck.
2) it's incorrect. eg: `c = a + { a += 1; b }`. here, the value of `a` does require a copy, cause `a` is changed before the actual read occurs (during the `add`).

i was kinda hoping that register allocation would solve this issue. but RA assumes that all live values are actually required.
a quick google search suggests that the optimization i'm looking for is called "copy propagation", which "often makes use of reaching definitions, use-def chains and def-use chains".
before going deeper down that rabbit hole, i just wanted to ask, how have you guys approached this problem?
]



locals are implemented as mutable registers):

```
copy t1, a
copy t2, b
add  t3, t1, t2
copy c, t3
```

in the previous version of the compiler, i got rid of those copies by passing a flag into the code-gen function (true if the value was only ever read).
for locals, this allowed the code-gen function to just return their (mutable) register, instead of allocating a new one & copying the local's value.
but this approach has a few issues:
1) it only helps during ast lowering. if optimizations produce similar code, i'm out of luck.
2) it's incorrect. eg: `c = a + { a += 1; b }`. here, the value of `a` does require a copy, cause `a` is changed before the actual read occurs (during the `add`).

i was kinda hoping that register allocation would solve this issue. but RA assumes that all live values are actually required.
a quick google search suggests that the optimization i'm looking for is called "copy propagation", which "often makes use of reaching definitions, use-def chains and def-use chains".
before going deeper down that rabbit hole, i just wanted to ask, how have you guys approached this problem?

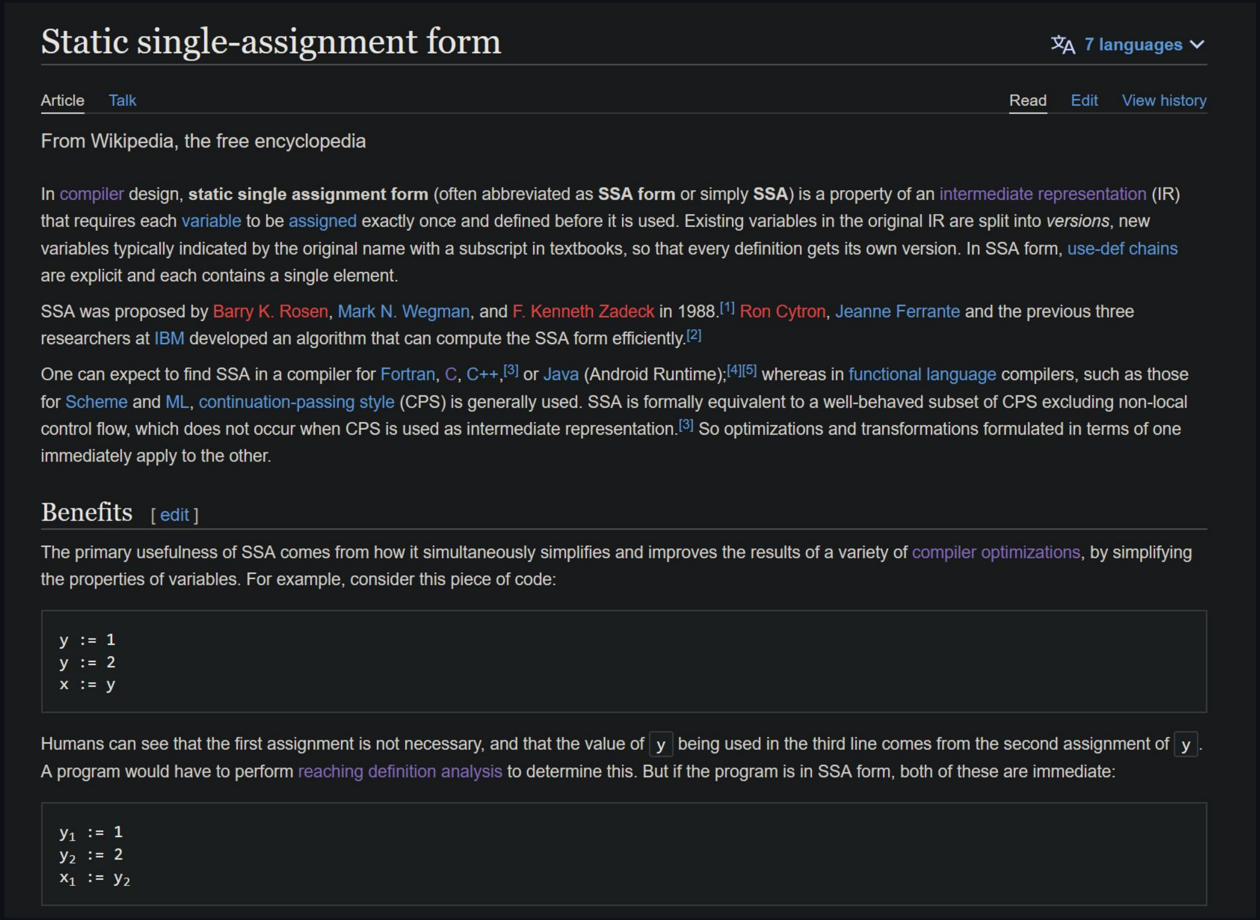
ratchetfreak 02/03/2023 7:52 PM
SSA solves that issue by making all registers read-only

```
a  = 1

i0 = a

...

...

i5 = add i0, #7
```

```
a  := 1

i0 := a

...

...

i5 := add i0, #7
```

```
a  := 1

i0 := a

a  := a + 1   ??

...

i5 := add i0, #7
```

```
a1 := 1

i0 := a1

a2 := a1 + 1

...

i5 := add i0, #7
```



Static single-assignment form

文A 7 languages

Article Talk Read Edit View history

From Wikipedia, the free encyclopedia

In compiler design, **static single assignment form** (often abbreviated as **SSA form** or simply **SSA**) is a property of an intermediate representation (IR) that requires each variable to be assigned exactly once and defined before it is used. Existing variables in the original IR are split into *versions*, new variables typically indicated by the original name with a subscript in textbooks, so that every definition gets its own version. In SSA form, use-def chains are explicit and each contains a single element.

SSA was proposed by Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck in 1988.[1] Ron Cytron, Jeanne Ferrante and the previous three researchers at IBM developed an algorithm that can compute the SSA form efficiently.[2]

One can expect to find SSA in a compiler for Fortran, C, C++,[3] or Java (Android Runtime);[4][5] whereas in functional language compilers, such as those for Scheme and ML, continuation-passing style (CPS) is generally used. SSA is formally equivalent to a well-behaved subset of CPS excluding non-local control flow, which does not occur when CPS is used as intermediate representation.[3] So optimizations and transformations formulated in terms of one immediately apply to the other.

Benefits [edit]

The primary usefulness of SSA comes from how it simultaneously simplifies and improves the results of a variety of compiler optimizations, by simplifying the properties of variables. For example, consider this piece of code:

```
y := 1
y := 2
x := y
```

Humans can see that the first assignment is not necessary, and that the value of `y` being used in the third line comes from the second assignment of `y`. A program would have to perform reaching definition analysis to determine this. But if the program is in SSA form, both of these are immediate:

```
y_1 := 1
y_2 := 2
x_1 := y_2
```

```
var a = 0

if foo:
     a = 1
end

print(a)
```

```
var a1 = 0

if foo:
     a = 1
end

print(a)
```

```
var a1 = 0

if foo:
     a2 = 1
end

print(a)
```

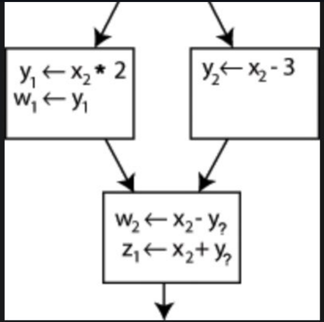
```
var a1 = 0

if foo:
     a2 = 1
end

print(a?)
```

```
var a1 = 0

if foo:
     a2 = 1
end

print(a2 if foo)
```

```
var a1 = 0

if foo:
     a2 = 1
end

print(a2 if foo else a1)
```

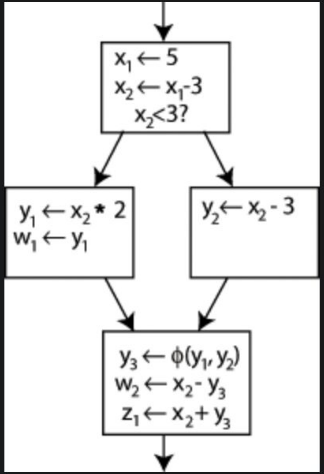
```
var a1 = 0

if foo:
     a2 = 1
end

a3 = a2 if foo else a1

print(a3)
```

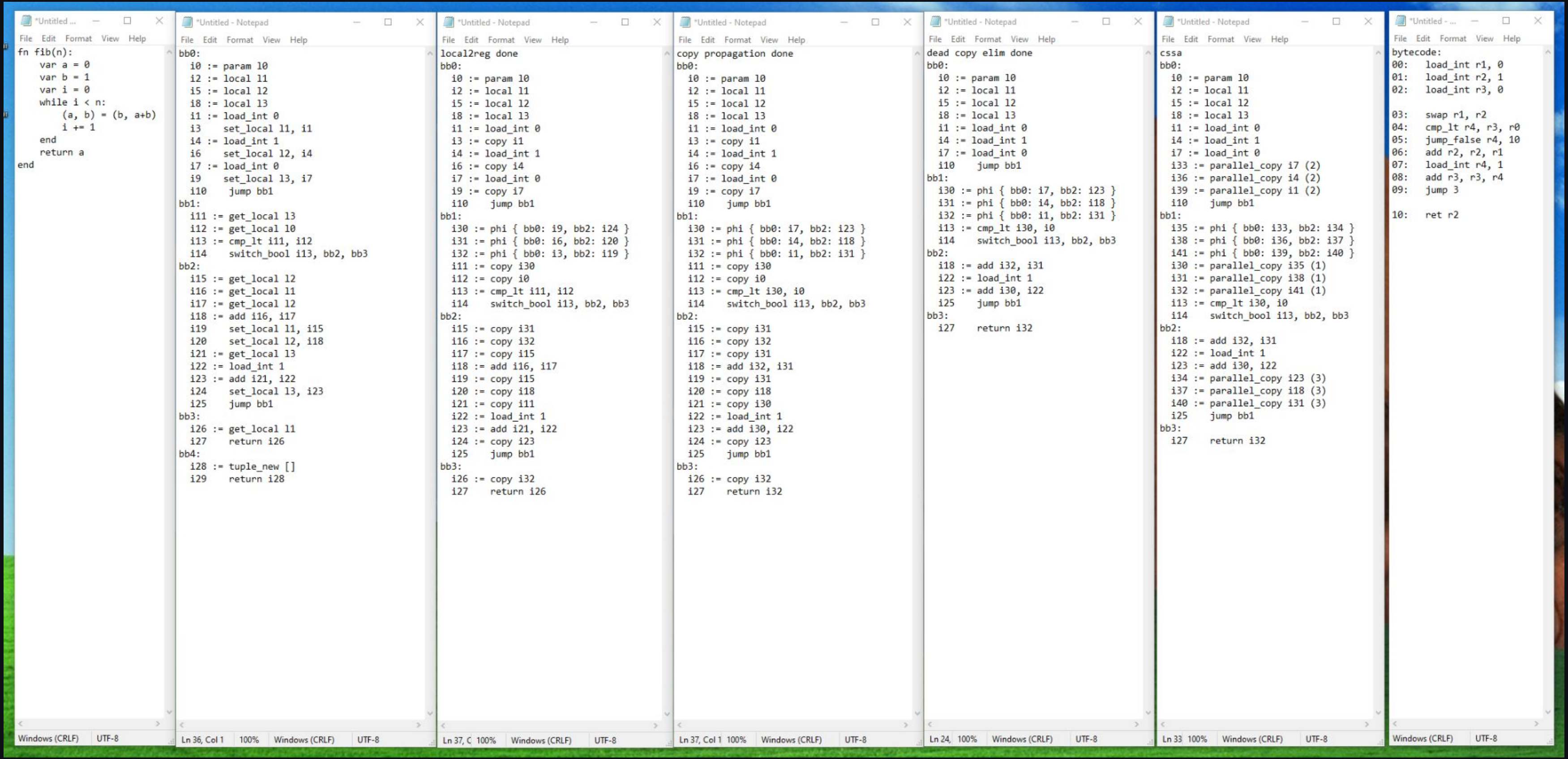
```
var a1 = 0

if foo:
     a2 = 1
end

a3 = phi(a1, a2)

print(a3)
```



It is clear which definition each use is referring to, except for one case: both uses of *y* in the bottom block could be referring to either y_1 or y_2, depending on which path the control flow took.

To resolve this, a special statement is inserted in the last block, called a *Φ (Phi) function*. This statement will generate a new definition of *y* called y_3 by "choosing" either y_1 or y_2, depending on the control flow in the past.





```
fn fib(n):
    var a = 0
    var b = 1
    var i = 0
    while i < n:
        (a, b) = (b, a+b)
        i += 1
    end
    return a
end
```

```
bb0:
  i0 := param l0
  i2 := local l1
  i5 := local l2
  i8 := local l3
  i1 := load_int 0
  i3    set_local l1, i1
  i4 := load_int 1
  i6    set_local l2, i4
  i7 := load_int 0
  i9    set_local l3, i7
  i10    jump bb1
bb1:
  i11 := get_local l3
  i12 := get_local l0
  i13 := cmp_lt i11, i12
  i14    switch_bool i13, bb2, bb3
bb2:
  i15 := get_local l2
  i16 := get_local l1
  i17 := get_local l2
  i18 := add i16, i17
  i19    set_local l1, i15
  i20    set_local l2, i18
  i21 := get_local l3
  i22 := load_int 1
  i23 := add i21, i22
  i24    set_local l3, i23
  i25    jump bb1
bb3:
  i26 := get_local l1
  i27    return i26
bb4:
  i28 := tuple_new []
  i29    return i28
```

```
local2reg done
bb0:
  i0 := param l0
  i2 := local l1
  i5 := local l2
  i8 := local l3
  i1 := load_int 0
  i3 := copy i1
  i4 := load_int 1
  i6 := copy i4
  i7 := load_int 0
  i9 := copy i7
  i10    jump bb1
bb1:
  i30 := phi { bb0: i9, bb2: i24 }
  i31 := phi { bb0: i6, bb2: i20 }
  i32 := phi { bb0: i3, bb2: i19 }
  i11 := copy i30
  i12 := copy i0
  i13 := cmp_lt i11, i12
  i14    switch_bool i13, bb2, bb3
bb2:
  i15 := copy i31
  i16 := copy i32
  i17 := copy i15
  i18 := add i16, i17
  i19 := copy i15
  i20 := copy i18
  i21 := copy i11
  i22 := load_int 1
  i23 := add i21, i22
  i24 := copy i23
  i25    jump bb1
bb3:
  i26 := copy i32
  i27    return i26
```

```
copy propagation done
bb0:
  i0 := param l0
  i2 := local l1
  i5 := local l2
  i8 := local l3
  i1 := load_int 0
  i3 := copy i1
  i4 := load_int 1
  i6 := copy i4
  i7 := load_int 0
  i9 := copy i7
  i10    jump bb1
bb1:
  i30 := phi { bb0: i7, bb2: i23 }
  i31 := phi { bb0: i4, bb2: i18 }
  i32 := phi { bb0: i1, bb2: i31 }
  i11 := copy i30
  i12 := copy i0
  i13 := cmp_lt i30, i0
  i14    switch_bool i13, bb2, bb3
bb2:
  i15 := copy i31
  i16 := copy i32
  i17 := copy i31
  i18 := add i32, i31
  i19 := copy i31
  i20 := copy i18
  i21 := copy i30
  i22 := load_int 1
  i23 := add i30, i22
  i24 := copy i23
  i25    jump bb1
bb3:
  i26 := copy i32
  i27    return i32
```

```
dead copy elim done
bb0:
  i0 := param l0
  i2 := local l1
  i5 := local l2
  i8 := local l3
  i1 := load_int 0
  i4 := load_int 1
  i7 := load_int 0
  i10    jump bb1
bb1:
  i30 := phi { bb0: i7, bb2: i23 }
  i31 := phi { bb0: i4, bb2: i18 }
  i32 := phi { bb0: i1, bb2: i31 }
  i13 := cmp_lt i30, i0
  i14    switch_bool i13, bb2, bb3
bb2:
  i18 := add i32, i31
  i22 := load_int 1
  i23 := add i30, i22
  i25    jump bb1
bb3:
  i27    return i32
```

```
cssa
bb0:
  i0 := param l0
  i2 := local l1
  i5 := local l2
  i8 := local l3
  i1 := load_int 0
  i4 := load_int 1
  i7 := load_int 0
  i33 := parallel_copy i7 (2)
  i36 := parallel_copy i4 (2)
  i39 := parallel_copy i1 (2)
  i10    jump bb1
bb1:
  i35 := phi { bb0: i33, bb2: i34 }
  i38 := phi { bb0: i36, bb2: i37 }
  i41 := phi { bb0: i39, bb2: i40 }
  i30 := parallel_copy i35 (1)
  i31 := parallel_copy i38 (1)
  i32 := parallel_copy i41 (1)
  i13 := cmp_lt i30, i0
  i14    switch_bool i13, bb2, bb3
bb2:
  i18 := add i32, i31
  i22 := load_int 1
  i23 := add i30, i22
  i34 := parallel_copy i23 (3)
  i37 := parallel_copy i18 (3)
  i40 := parallel_copy i31 (3)
  i25    jump bb1
bb3:
  i27    return i32
```

```
bytecode:
00:    load_int r1, 0
01:    load_int r2, 1
02:    load_int r3, 0

03:    swap r1, r2
04:    cmp_lt r4, r3, r0
05:    jump_false r4, 10
06:    add r2, r2, r1
07:    load_int r4, 1
08:    add r3, r3, r4
09:    jump 3

10:    ret r2
```