```
In [1]: import { requireCytoscape, requireCarbon } from "./lib/draw";

        requireCarbon();
        requireCytoscape();
```

Transpilers and Compilers

Where Were We?

1. Language primitives (i.e., building blocks of languages)
2. Language paradigms (i.e., combinations of language primitives)
3. **Building a language** (i.e., designing your own language)

- Last time: interpreters and evaluation
- This time: **compilers** and **transpilers**

Review

- We've seen two components of language implementation: parsers and interpreters.
- Let's review these before moving to the topic for today: *compilers* and *transpilers*.

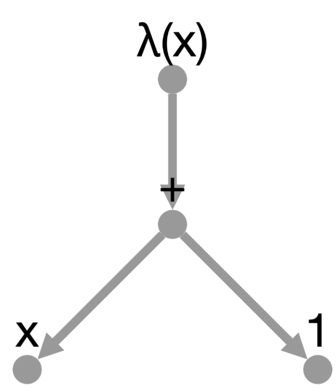
```
In [2]: import { draw, treeLayout } from "./lib/draw";
        import * as T from "./lib/lambdats/token";
        import * as E from "./lib/lambdats/expr";
        import * as I from "./lib/lambdats/substInterp";
        import * as Parser from "./lib/lambdats/parser";

        function drawProg(prog: string|E.Expr): void {
            if (typeof prog === 'string') {
                draw(E.cytoscapify(Parser.parse(prog)), 800, 350, treeLayout);
            } else {
                draw(E.cytoscapify(prog), 800, 350, treeLayout);
            }
        }
```

Component: Parsing

- Input: string
- Output: AST

```
In [3]: const inputString = "λx => x + 1";
        const outputAST = Parser.parse(inputString);
        drawProg(outputAST);
```

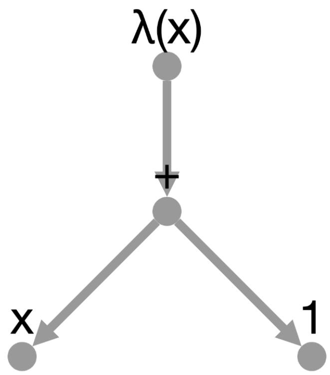


Component: Interpreter

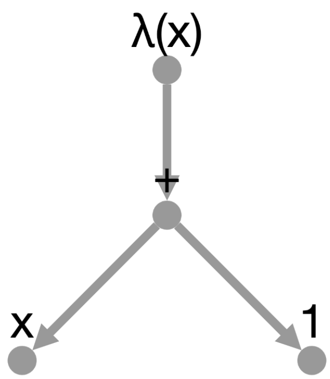
- Input: AST
- Output: AST that is a *value*.
  - A *value* is an AST that can no longer be reduce.
  - For LambdaTS, this means that the root of the AST is either a `NumericConstant` or a `FunctionExpr`.

In [4]:
```
const inputAST = Parser.parse("λx => x + 1");
console.log("Input");
drawProg(inputAST);
const outputAST = I.interpret(inputAST);
console.log("Output");
drawProg(outputAST);
```

Input

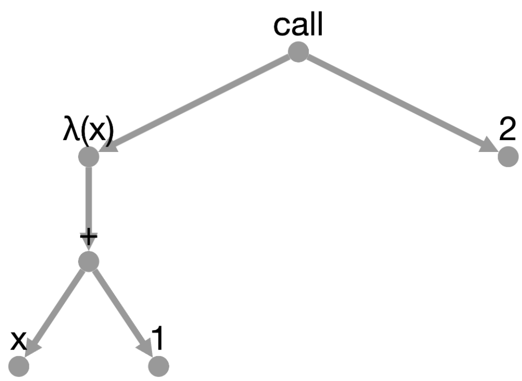


Output

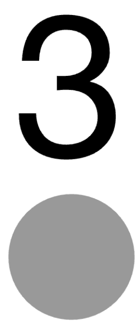


```
In [5]: const inputAST = Parser.parse("(λx => x + 1)(2)");
console.log("Input");
drawProg(inputAST);
const outputAST = I.interpret(inputAST);
console.log("Output");
drawProg(outputAST);
```

Input



Output



Compiler / Transpiler

Today, we will look at **compilers**, a program that converts an expression/AST in a **source** language into an expression/AST in a **target** language.

- Traditionally, the source language is some high-level language and the target language is assembly, i.e., a low-level language.
- When the **source** langauge and **target** language are both high-level programming langauges, the compiler is also called a **transpiler**.

LambdaTS Expression/AST

```
export type Expr = NumericConstant | BinaryExpr | ConditionalExpr | FunctionExpr | Identifier | CallExpr;
export type BinaryExpr = { tag: "BINARY";
    operator: BinaryOperator;
    left: Expr;
    right: Expr;
};
// Note (e) is not part of the AST
export type ConditionalExpr = { tag: "CONDITIONAL";
    condExpr: Expr;
    thenExpr: Expr;
    elseExpr: Expr;
};
export type FunctionExpr = { tag: "FUNCTION";
    parameter: string;
    body: Expr;
};
export type CallExpr = { tag: "CALL";
    func: Expr;
    argument: Expr;
};
```

A Second AST

- Transpiling does not make sense if there is only 1 AST.
- We'll need another AST. Let's add **let bindings** to LambdaTS to create LambdaTS2.
- We will then write a **transpiler** that translates expressions in LambdaTS2 to LambdaTS.

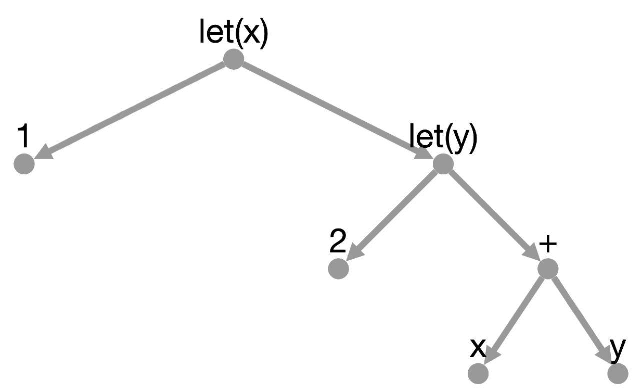
Let Expressions

- We want to have local variables.
- Note that TypeScript has let statements and not let expressions.
- Let expressions are the functional version of let statements.

In [6]:
```
let x = 1;
let y = 2;
x + y
```

3

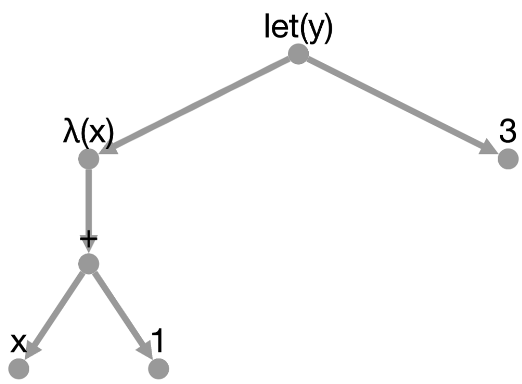
In [7]:
```
// LambdaTS2 program
drawProg(Parser.parse("let x = 1 in let y = 2 in x + y"));
```



In [8]:
```
let y = (x) => x + 1;
3
```

3

In [9]:
```
// LambdaTS2 Program
drawProg(Parser.parse("let y = λx => x + 1 in 3"));
```



LambdaTS2 AST

In [10]:
```
type Expr = T.NumericConstant | BinaryExpr | ConditionalExpr | FunctionExpr | T.Identifier | CallExpr | LetExpr;

// New language construct
type LetExpr = { tag: "LET";  // let x = left in right
    name: string;
    left: Expr;
    right: Expr;
};

type BinaryExpr = { tag: "BINARY";
    operator: T.BinaryOperator;
    left: Expr;
    right: Expr;
};
type ConditionalExpr = { tag: "CONDITIONAL";
    condExpr: Expr;
    thenExpr: Expr;
    elseExpr: Expr;
};
type FunctionExpr = { tag: "FUNCTION";
    parameter: string;
    body: Expr;
};
type CallExpr = { tag: "CALL";
    func: Expr;
    argument: Expr;
};
```

LambdaTS2 Interpreter?

- We could write an interpreter for LambdaTS2.
- Alternatively, we could translate a LambdaTS2 AST into a LambdaTS AST and use the LambdaTS interpreter.
- This is what happens with TypeScript---we use `tsc` to transpile a TypeScript AST into a JavaScript AST and use the JavaScript interpreter node.

Transpiling

Example 1

In [11]:
```
let x = 2;
x + 1
```

3

In [12]:
```
((x) => x + 1)(2)
```

3

Example 2

In [13]:
```
let x = 2;
let y = 3;
x + y
```

5

In [14]:
```
// Step one
((x) => {
    let y = 3;
    return x + y;
})(2)
```

5

In [15]:
```
// Step two, recursive translation of the body
((x) => ((y) => x + y)(3))(2)
```

5

Exercise

In [16]:
```
let x = 2;
let y = 3;
let z = 4;
x + y + z;
```

9

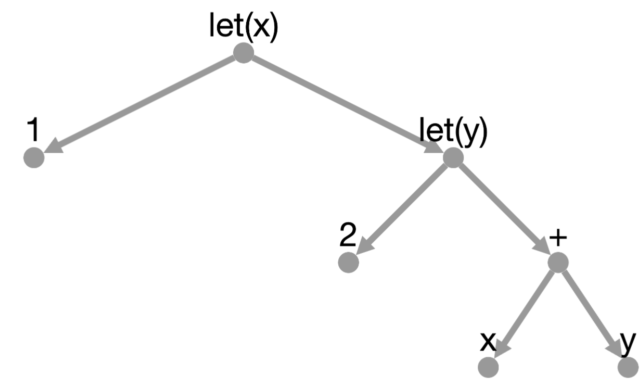
In [18]:
```
// Expr is LambdaTS2 AST
// E.Expr is LambdaTS AST
function transpile(e: Expr): E.Expr {
    switch (e.tag) {
        case "LET": {
            // We don't have "LET" in LambdaTS
            // We translate let x = left in right into
            //                (λx => right)(left)
            return E.mkCallExpr(E.mkFunctionExpr(e.name, transpile(e.right)), transpile(e.left));
        }

        // All of theses cases are not interesting
        case "NUMBER": {
            return T.mkNumericConstant(e.value);
        }
        case "BINARY": {
            return E.mkBinaryExpr(transpile(e.left), e.operator, transpile(e.right));
        }
        case "CONDITIONAL": {
            return E.mkConditionalExpr(transpile(e.condExpr), transpile(e.thenExpr), transpile(e.elseExpr));
        }
        case "FUNCTION": {
            return E.mkFunctionExpr(e.parameter, transpile(e.body));
        }
        case "IDENTIFIER": {
            return T.mkIdentifier(e.name);
        }
        case "CALL": {
            return E.mkCallExpr(transpile(e.func), transpile(e.argument));
        }
        default: {
            throw Error("Shouldn't happen");
        }
    }
}
```

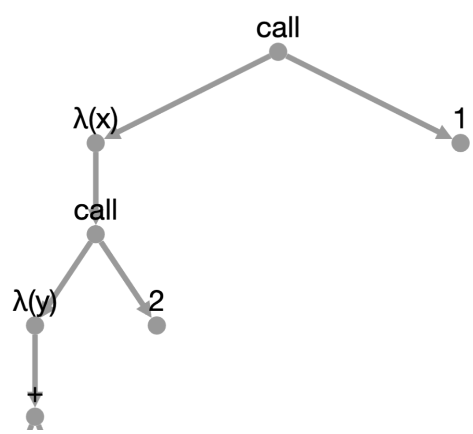
Example 1

```
In [19]: const inputAST = Parser.parse("let x = 1 in let y = 2 in x + y");
console.log("Input (in LambdaTS2)");
drawProg(inputAST);
const outputAST = transpile(inputAST);
console.log("Output (in LambdaTS)");
drawProg(outputAST);
console.log("Interpret (in LambdaTS)");
drawProg(I.interpret(outputAST));
```

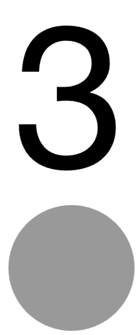
Input (in LambdaTS2)



Output (in LambdaTS)



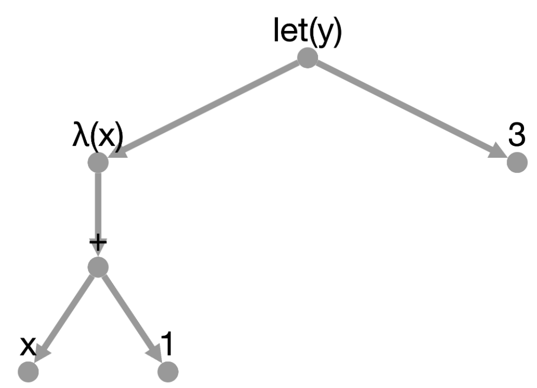
Interpret (in LambdaTS)



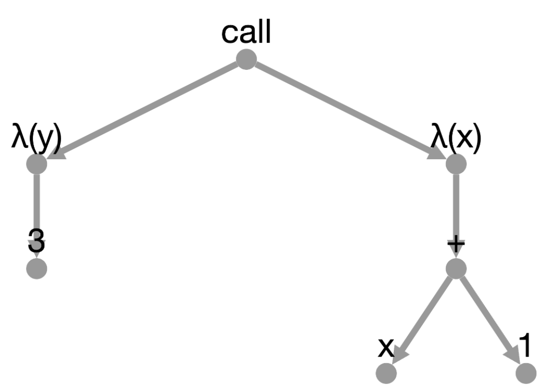
Example 2

In [20]:
```
const inputAST = Parser.parse("let y = λx => x + 1 in 3");
console.log("Input (in LambdaTS2)");
drawProg(inputAST);
const outputAST = transpile(inputAST);
console.log("Output (in LambdaTS)");
drawProg(outputAST);
console.log("Inerpret (in LambdaTS)");
drawProg(I.interpret(outputAST));
```

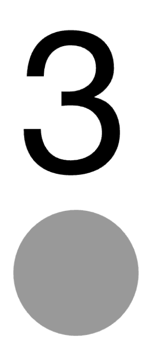
Input (in LambdaTS2)



Output (in LambdaTS)



Inerpret (in LambdaTS)



That's It!

- We have seen how to translate an AST in one language into an AST in another language.
- `transpile` translates a LambdaTS2 AST into a LambdaTS AST.
- This is the similar to what `tsc` is doing: translating a TypeScript AST into a JavaScript AST.
- Once we do so, we can use the interpreter for the other language to run our programs.

- Every TypeScript language feature is transpiled into a JavaScript AST.

Same Language Translations

- Transpilers take ASTs in one language and convert them into ASTs in another language.
- Now we'll look at same language source code translations.
- This can be used to optimize code and is what a compiler might perform.

Example 1: Compare these two pieces of code

In [21]:
```
// Version 1
const width = 5
const height = 4
const area = width * height
console.log(area)
```

20

In [22]:
```
// Version 2
console.log(20)
```

20

Tradeoffs

1. Version 1 is more interpretable.
2. Version 2 gets rid of extra arithmetic.

Example 2: Compare these two pieces of code

In [23]:
```
// Version 1
let flag = true;
let version;
if (flag) {
    version = "1.1";
} else {
    version = "1.2";
}
```

In [24]:
```
// Version 2
let version = "1.1";
// version = "1.2";  // We are using comments to toggle between version number
```

Tradeoffs

1. Version 1 uses code to toggle and is easier to extend.
2. Version 2 doesn't have branching overhead.

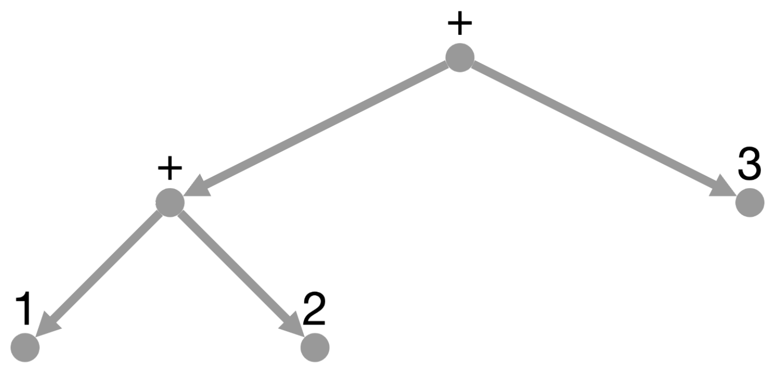
Constant Folding: Example of Semantics Preserving Translation

Get best of both worlds:

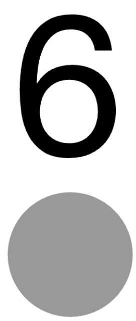
1. Let user write interpretable code.
2. Use syntax rewriting to get code that eliminates code that can be statically simplified.

In [25]:
```
console.log("original");
drawProg("1 + 2 + 3");
console.log("constant folded");
drawProg("6");
```

original



constant folded



Constant Folding on Arithmetic

In [26]:
```
function constantFoldBinary(eLeft: E.Expr, operator: T.BinaryOperator, eRight: E.Expr): E.Expr {
    if (eLeft.tag === "NUMBER" && eRight.tag === "NUMBER") {
        // If both the left and right are numbers, then we can evaluate them and return the appropriate constant.
        // Question: how does this compare with the interpreter for binary expressions?
        // Hint: look at the return type of constantFoldBinary
        switch (operator) {
            case "+": {
                return T.mkNumericConstant(eLeft.value + eRight.value);
            }
            case "-": {
                return T.mkNumericConstant(eLeft.value - eRight.value);
            }
            case "*": {
                return T.mkNumericConstant(eLeft.value * eRight.value);
            }
            case "/": {
                return T.mkNumericConstant(eLeft.value / eRight.value);
            }
        }
    } else {
        // If either one of the expressions is not a numeric constant, we cannot evaluate the expression.
        return E.mkBinaryExpr(eLeft, operator, eRight);
    }
}
```

In [27]:

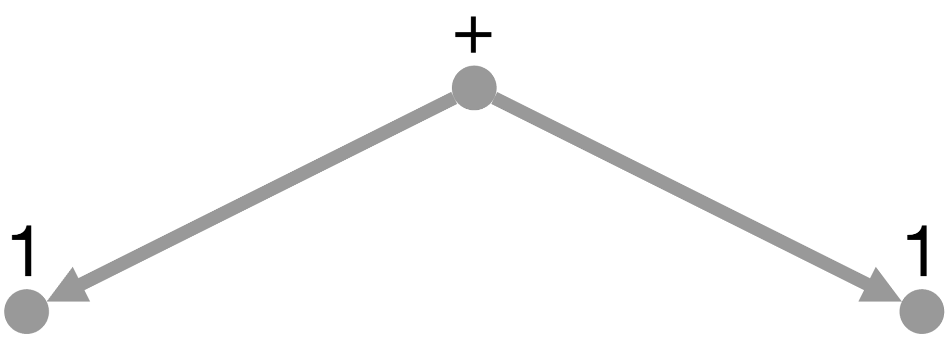
```
function constantFold(e: E.Expr): E.Expr {
    switch (e.tag) {
        case "NUMBER": {
            return T.mkNumericConstant(e.value); // Nothing to do
        }
        case "BINARY": {
            // Note: what can be improved?
            return constantFoldBinary(e.left, e.operator, e.right);
        }
        case "CONDITIONAL": {
            // Will handle later
            return E.mkConditionalExpr(constantFold(e.condExpr), constantFold(e.thenExpr), constantFold(e.elseExpr));
        }
        case "FUNCTION": {
            // Recursively perform constant folding
            return E.mkFunctionExpr(e.parameter, constantFold(e.body))
        }
        case "IDENTIFIER": {
            return T.mkIdentifier(e.name); // Nothing to do
        }
        case "CALL": {
            // Recursively perform constant folding
            return E.mkCallExpr(constantFold(e.func), constantFold(e.argument));
        }
        default: {
            throw Error("Shouldn't happen");
        }
    }
}
```

Example 1

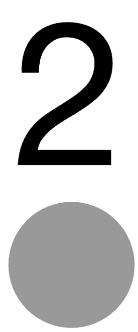
In [28]:

```
const ex1 = "1 + 1";
console.log("Original", E.exprToString(Parser.parse(ex1)));
drawProg(ex1);
console.log("Optimized", E.exprToString(constantFold(Parser.parse(ex1))));
drawProg(constantFold(Parser.parse(ex1)));
```

Original (1+1)



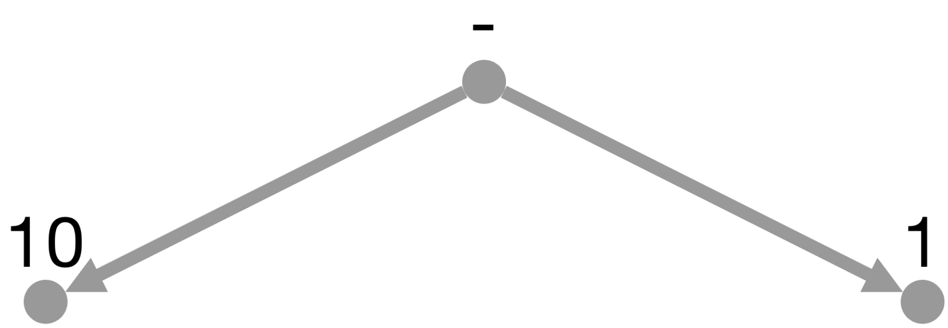
Optimized 2



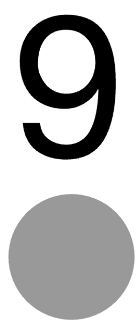
Example 2

In [29]:
```
const ex2 = "10 - 1";
console.log("Original", E.exprToString(Parser.parse(ex2)));
drawProg(ex2);
console.log("Optimized", E.exprToString(constantFold(Parser.parse(ex2))));
drawProg(constantFold(Parser.parse(ex2)));
```

Original (10-1)



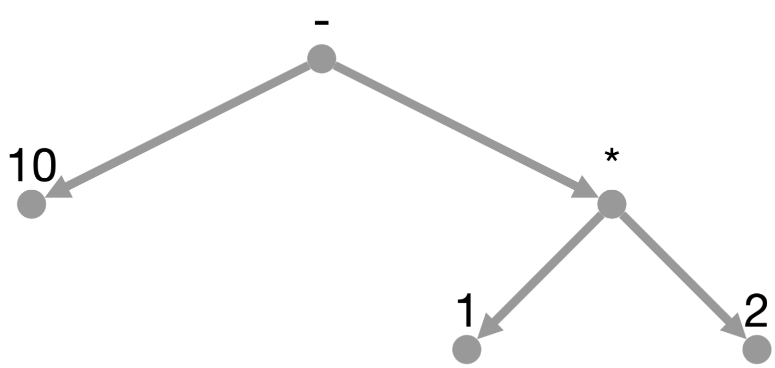
Optimized 9



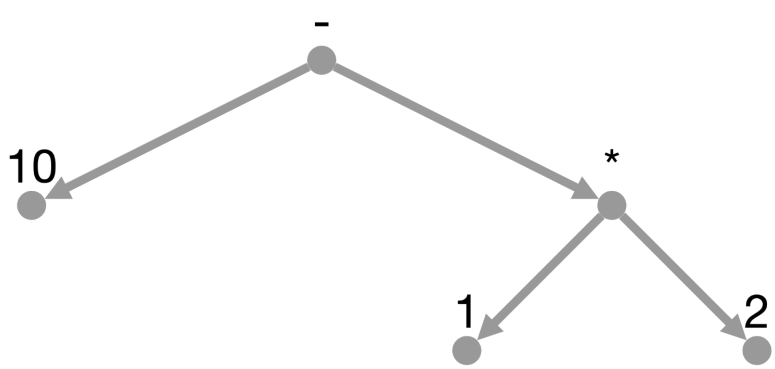
Example 3

In [30]:
```
const ex3 = "10 - (1 * 2)";
console.log("Original", E.exprToString(Parser.parse(ex3)));
drawProg(ex3);
console.log("Optimized", E.exprToString(constantFold(Parser.parse(ex3))));
drawProg(constantFold(Parser.parse(ex3)));
```

Original (10-(1*2))



Optimized (10-(1*2))



Hmm ... what went wrong?

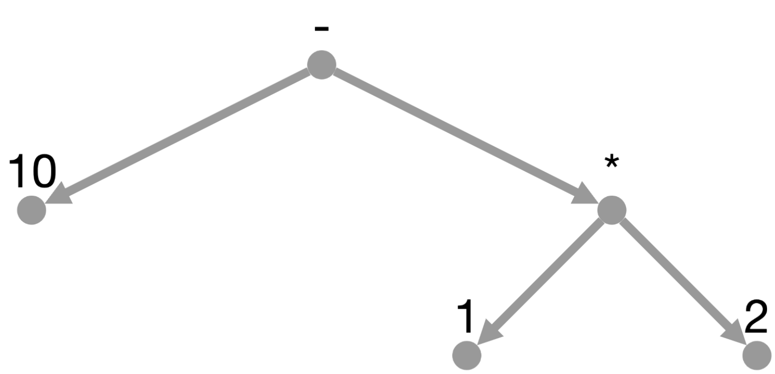
1. 10 is a numeric constant
2. 1*2 is a binary expression
3. Hence, our constant fold operation cannot reduce it

In [31]:
```
function constantFold2(e: E.Expr): E.Expr {
    switch (e.tag) {
        case "NUMBER": {
            return T.mkNumericConstant(e.value); // Nothing to do
        }
        case "BINARY": {
            // Note: what can be improved?
            return constantFoldBinary(constantFold2(e.left), e.operator, constantFold2(e.right));
        }
        case "CONDITIONAL": {
            // Will handle later
            return E.mkConditionalExpr(constantFold2(e.condExpr), constantFold2(e.thenExpr), constantFold2(e.elseExpr));
        }
        case "FUNCTION": {
            // Recursively perform constant folding
            return E.mkFunctionExpr(e.parameter, constantFold2(e.body))
        }
        case "IDENTIFIER": {
            return T.mkIdentifier(e.name); // Nothing to do
        }
        case "CALL": {
            // Recursively perform constant folding
            return E.mkCallExpr(constantFold2(e.func), constantFold2(e.argument));
        }
        default: {
            throw Error("Shouldn't happen");
        }
    }
}
```

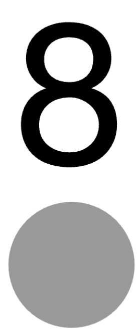
Example 3: Again

In [32]:
```
const ex3 = "10 - 1 * 2";
console.log("Original", E.exprToString(Parser.parse(ex3)));
drawProg(ex3);
console.log("Optimized", E.exprToString(constantFold2(Parser.parse(ex3))));
drawProg(constantFold2(Parser.parse(ex3)));
```

Original (10-(1*2))



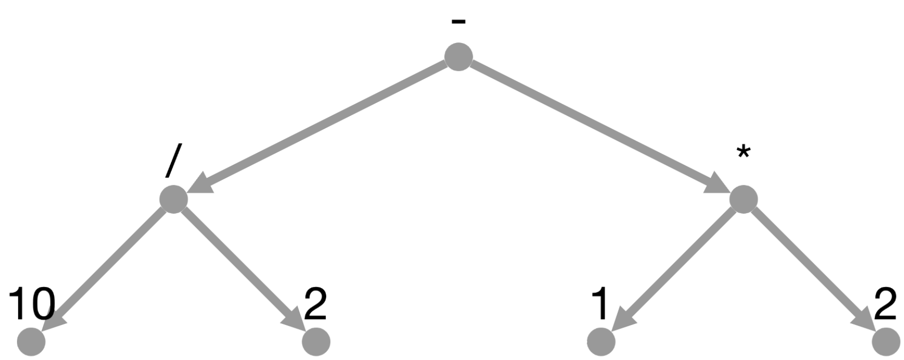
Optimized 8



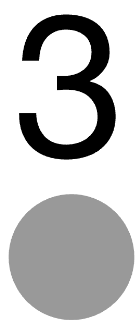
Example 4

```
In [33]: const ex4 = "(10 / 2)- 1 * 2";
console.log("Original", E.exprToString(Parser.parse(ex4)));
drawProg(ex4);
console.log("Optimized", E.exprToString(constantFold2(Parser.parse(ex4))));
drawProg(constantFold2(Parser.parse(ex4)));
```

Original ((10/2)-(1*2))



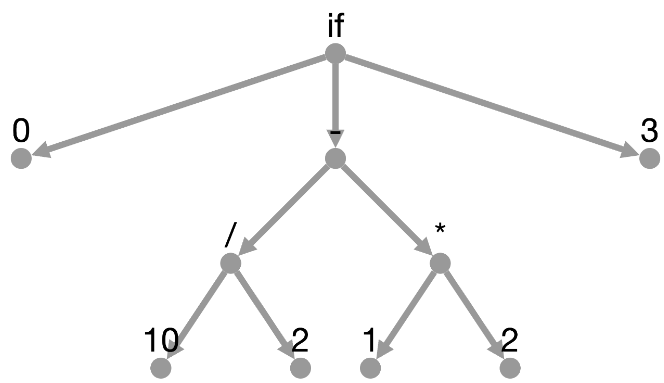
Optimized 3



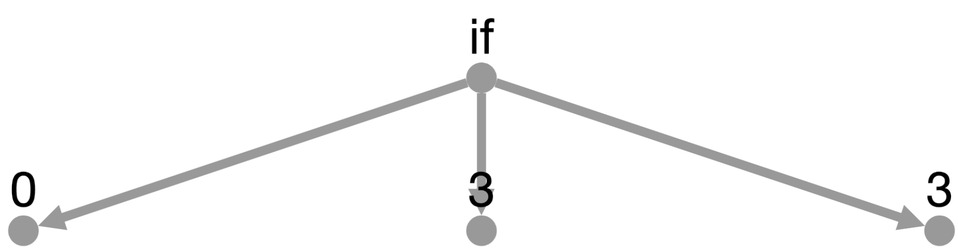
Example 5

In [34]:
```
const ex5 = "0 ? (10 / 2)- 1 * 2 : 3";
console.log("Original", E.exprToString(Parser.parse(ex5)));
drawProg(ex5);
console.log("Optimized", E.exprToString(constantFold2(Parser.parse(ex5))));
drawProg(constantFold2(Parser.parse(ex5)));
```

Original (0?((10/2)-(1*2)):3)



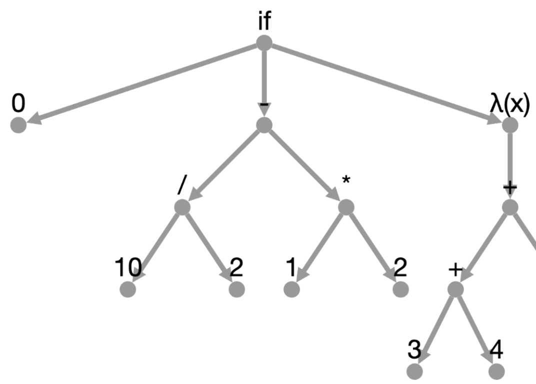
Optimized (0?3:3)



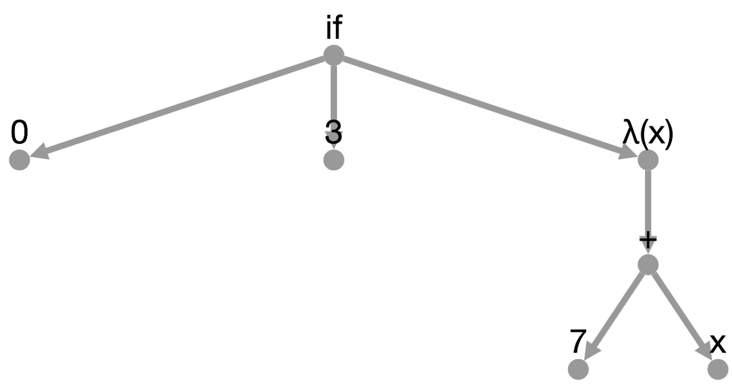
Example 6

In [35]:
```
const ex6 = "0 ? (10 / 2)- 1 * 2 : λx => 3 + 4 + x";
console.log("Original", E.exprToString(Parser.parse(ex6)));
drawProg(ex6);
console.log("Optimized", E.exprToString(constantFold2(Parser.parse(ex6))));
drawProg(constantFold2(Parser.parse(ex6)));
```

Original (0?((10/2)-(1*2)):(λx=>((3+4)+x)))



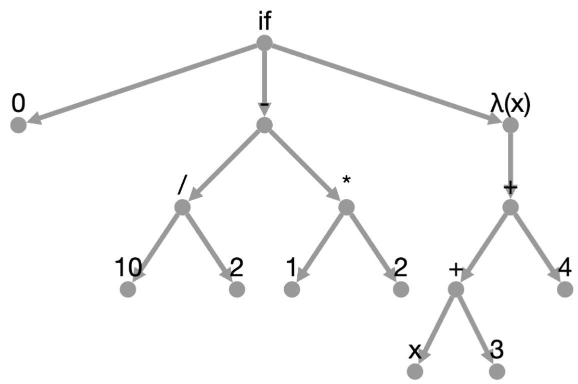
Optimized (0?3:(λx=>(7+x)))



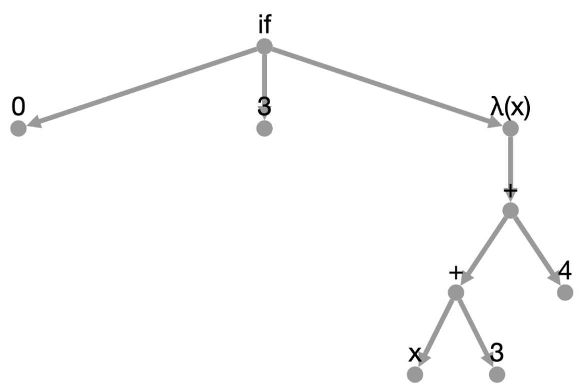
Example 7

In [36]:
```
const ex7 = "0 ? (10 / 2)- 1 * 2 : λx => x + 3 + 4";
console.log("Original", E.exprToString(Parser.parse(ex7)));
drawProg(ex7);
console.log("Optimized", E.exprToString(constantFold2(Parser.parse(ex7))));
drawProg(constantFold2(Parser.parse(ex7)));
```

Original (0?((10/2)-(1*2)):(λx=>((x+3)+4)))



Optimized (0?3:(λx=>((x+3)+4)))



Hmmm ... that didn't work

1. x + 3 is an expression
2. so therefore our constant folder did not work

Aside: soundness vs. completeness

1. Both `constantFold` and `constantFold2` are **sound**, i.e., they do the correct thing
2. Neither `constantFold` nor `constantFold2 are **complete**, i.e., perform all constant foldings
3. In general, cannot be both sound and complete. Languages make the decision to be sound.
4. This soundness vs. completeness issue most often appears in **type-checking**.

Constant Folding with Conditionals

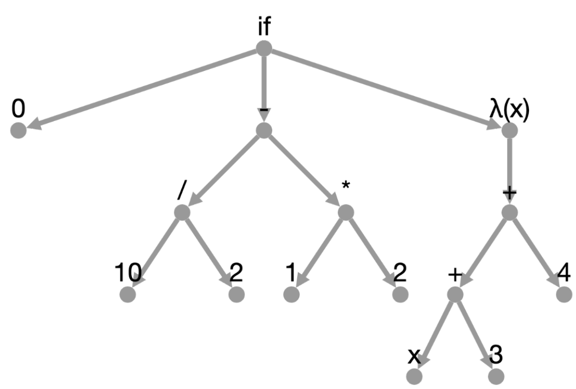
In [37]:
```
function constantFoldConditional(eCondExpr: E.Expr, eThenExpr: E.Expr, eElseExpr: E.Expr): E.Expr {
    if (eCondExpr.tag === "NUMBER") {
        return eCondExpr.value === 0 ? eElseExpr : eThenExpr;
    } else {
        return E.mkConditionalExpr(eCondExpr, eThenExpr, eElseExpr);
    }
}
```

In [38]:
```
function constantFold3(e: E.Expr): E.Expr {
    switch (e.tag) {
        case "NUMBER": {
            return T.mkNumericConstant(e.value); // Nothing to do
        }
        case "BINARY": {
            // Note: what can be improved?
            return constantFoldBinary(constantFold3(e.left), e.operator, constantFold3(e.right));
        }
        case "CONDITIONAL": {
            // Will handle later
            return constantFoldConditional(constantFold3(e.condExpr), constantFold3(e.thenExpr), constantFold3(e.elseExp
        }
        case "FUNCTION": {
            // Recursively perform constant folding
            return E.mkFunctionExpr(e.parameter, constantFold3(e.body))
        }
        case "IDENTIFIER": {
            return T.mkIdentifier(e.name); // Nothing to do
        }
        case "CALL": {
            // Recursively perform constant folding
            return E.mkCallExpr(constantFold3(e.func), constantFold3(e.argument));
        }
        default: {
            throw Error("Shouldn't happen");
        }
    }
}
```

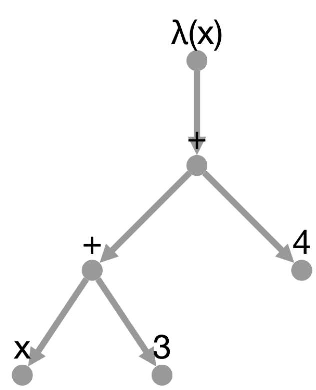
Example 7 Again

In [39]:
```
const ex7 = "0 ? (10 / 2)- 1 * 2 : λx => x + 3 + 4";
console.log("Original", E.exprToString(Parser.parse(ex7)));
drawProg(ex7);
console.log("Optimized", E.exprToString(constantFold3(Parser.parse(ex7))));
drawProg(constantFold3(Parser.parse(ex7)));
```

Original (0?((10/2)-(1*2)):(λx=>((x+3)+4)))



Optimized (λx=>((x+3)+4))



Summary

1. Constant folding is one of many optimizations that many compilers perform.
2. Many "hand-performed" optimizations can be done with a compiler.
3. Obviously, there are limits to compiler optimization. Compiler optimizations are sound, but not necessarily complete.

Making Your Code Lazy: Code transformation that changes the semantics of your code

Problem: suppose you want to transpile Haskell code to TypeScript code. The problem is that Haskell is a lazy language but TypeScript has eager semantics. How would you do this?

Related problem: suppose we have an interpreter for lambdaTS that has eager semantics, but we want it to have lazy semantics. What's an alternative to writing another interpreter for lambdaTS?

In [40]:
```
function toLazyExpr(e: E.Expr): E.Expr {
    switch (e.tag) {
        case "NUMBER": {
            return T.mkNumericConstant(e.value);
        }
        case "BINARY": {
            return E.mkBinaryExpr(toLazyExpr(e.left), e.operator, toLazyExpr(e.right));
        }
        case "CONDITIONAL": {
            return E.mkConditionalExpr(toLazyExpr(e.condExpr), toLazyExpr(e.thenExpr), toLazyExpr(e.elseExpr));
        }
        case "FUNCTION": {
            return E.mkFunctionExpr(e.parameter, toLazyExpr(e.body))
        }
        case "IDENTIFIER": {
            // Evaluate the thunk with an arbitrary input (thunk's argument is never used).
            return E.mkCallExpr(T.mkIdentifier(e.name), T.mkNumericConstant(42));
        }
        case "CALL": {
            // Delay the evaluation of the argument of a call by wrapping it in a thunk.
            const thunk = E.mkFunctionExpr("unit", toLazyExpr(e.argument));
            return E.mkCallExpr(toLazyExpr(e.func), thunk);
        }
        default: {
            throw Error("Shouldn't happen");
        }
    }
}
```

Example

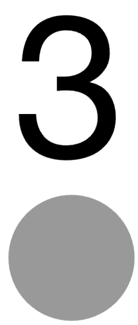
In [41]:
```
E.exprToString(toLazyExpr(Parser.parse("(λx => 3)(4 + (λy => y))")))
```

(λx=>3)((λunit=>(4+(λy=>y(42)))))

In [43]:
```
try {
    // Should error
    drawProg(I.interpret(Parser.parse("(λx => 3)(4 + (λy => y))")));
} catch(err) {
    console.log(err);
}
```

```
Error: Attempting [object Object] + [object Object]
    at interpretBinop (/Users/dehuang/Documents/teaching/csc600/lectures/lib/lambdats/substInterp.js:74:15)
    at interpret (/Users/dehuang/Documents/teaching/csc600/lectures/lib/lambdats/substInterp.js:84:20)
    at Object.interpret (/Users/dehuang/Documents/teaching/csc600/lectures/lib/lambdats/substInterp.js:104:29)
    at evalmachine.<anonymous>:4:32
    at evalmachine.<anonymous>:10:3
    at sigintHandlersWrap (node:vm:270:12)
    at Script.runInThisContext (node:vm:127:14)
    at Object.runInThisContext (node:vm:307:38)
    at Object.execute (/Users/dehuang/Documents/teaching/csc600/lectures/node_modules/tslab/dist/executor.js:162:38)
    at JupyterHandlerImpl.handleExecuteImpl (/Users/dehuang/Documents/teaching/csc600/lectures/node_modules/tslab/dist/jupyter.js:219:38)
```

In [45]:
```
// The lazy program does not error!
drawProg(I.interpret(toLazyExpr(Parser.parse("(λx => 3)(4 + (λy => y))"))));
```



Summary

1. Compilers and transpilers are programs that convert ASTs in one language into ASTs in another language.
2. Compilers/transpilers can be used for code optimization.
3. Compilers/transpilers can be used to convert between languages with different semantics.

In [ ]:
```
```