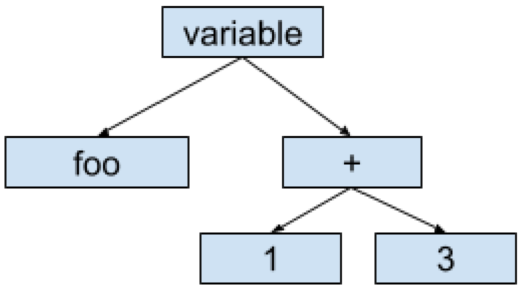
Notes

- An interpreter is a program that scans a source-code into tokens, parses the tokens using context-free grammar rule to transform it to an Abstract Syntax Tree (AST) and then execute the AST by walking over it
- An interpreter has the following phases:
    - Scanning:
        - Lexeme: blob of words, eg. var x = ‘foo bar’ (var, x, =, ‘foo bar’)
        - Each alphabet at the scanning level is a character
        - Token:
            - Identifier
            - symbol (‘=’,etc..)
            - Reserved words (var, etc..)
            - etc..
    - Parsing:
        - Each alphabet at the parsing level is a token
        - An expression is one more tokens
        - Context-free grammar

          Examples
            - Expression:
                - expression -> equality
                - Equality -> comparison ((“!=”|”==”) comparison)*
                - Comparison -> term((“>”|”>=”|”<”|”<=) term)*
                - term-> factor((“-”|”+”) factor)*
                - factor-> unary((“/”|”*”) unary)*
                - literal -> number | string | “true” | “false” | “nil”
                - unary -> (“-” | “!”) unary | primary
                - binary -> expression operator expression
                - operator -> “==” | “!=” | “<” | “<=” | “>” | “>=” | “+” | “-” | “*” | “/”
                - primary -> number | string | “true” | “false” | “nil” | “(“ expression “)”
                - etc..
            - Statements:
                - program -> declaration * EOF
                - statement -> exprStmt | printStmt
                - exprStmt -> expression “;”
                - printStmt -> “print” expression “;”
                - block -> “{“ declaration * “}”
                - declaration -> funDecl | varDecl | statement;
                - function -> identifier “(“ params? “)” block;
                - If -> if “(“ exprStmt “)” statement (else statement)?
                - etc..

    - Resolving lexical scopes:
        - This is needed to ensure the correct implementation of lexical scoping and closures.
        - Semantic analysis is performed to infer meaning from the code.
        - More on this later.
    - Interpreting the tree:
        - The abstract syntax tree is a binary tree data structure that is an output of the parsing phase.

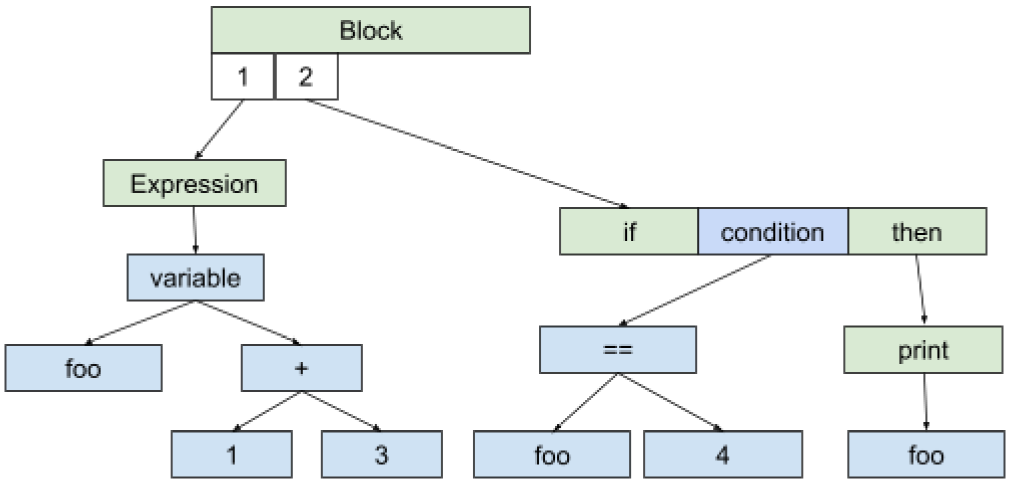
- Here is an example of *variable foo* = *1* + *3* (variable foo (+ 1 3))



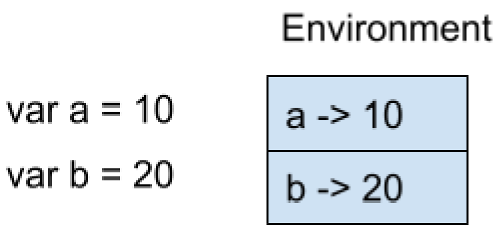
- Here is a slightly more complex example involving statements (in green) and expr (in blue)

```
{
   var foo = 1+3
   if (foo == 4) print foo
}
```

<Block <(var foo (+ 1 3)>, <if (== foo 4) <print foo> >>



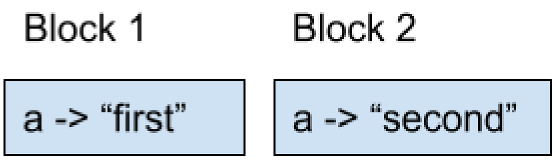
- **Environments:**
  - A data structure that allows bindings of variables to values.



- **Scope:**
  - Defines a region where name maps to certain entities.

- Multiple scopes allows users to enable the same name to refer to different things.
- Lexical scope (static scope) is a specific style of scoping where text of a program shows where scope start and end.
    - Eg

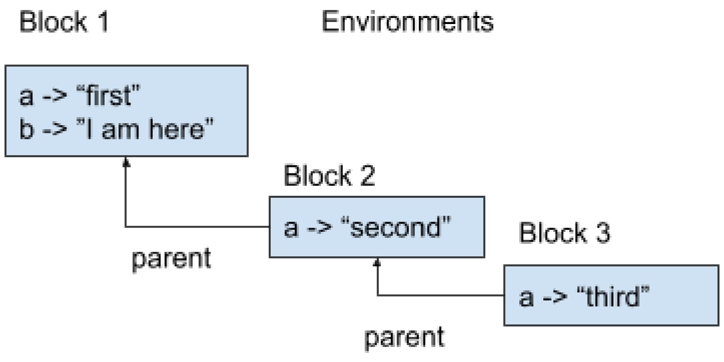
```
// block 1
{
       var a = “first”
}
// var a is removed from scope
// block 2
{
       var a = “second”
}
// var a is removed from scope
```



- **Nested scoping**
    - Allows scope to nest so that the nested block can see variables upstream and if there are variables with the same name, it will not interfere with the surrounding variables.
    - Each time there is a nested block, a new environment is created with the parent environment passed in as reference. Sort of like a stack…
    - If the nested block (e.g. block 2) contains the same variable name, it shadows the outer one, meaning the code inside the block can only see the inner one.

    Eg

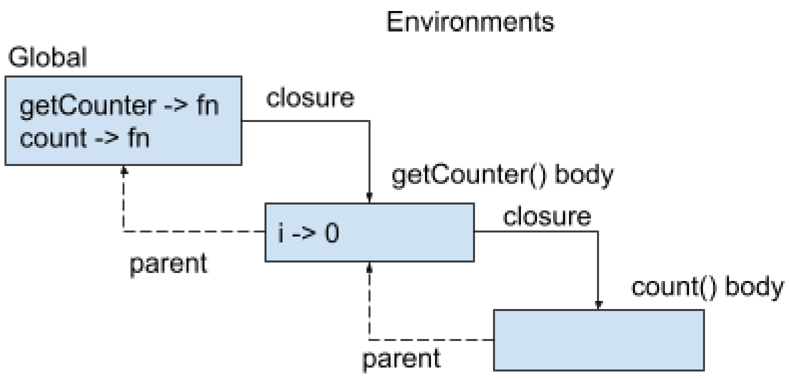
```
{
       var a = “first
       Var b = “i am here”
       {
              var a = “second”
              {
                     var a = ”third”
                     print b
              }
       }
}
```



- How does a nested block find variables declared upstream?
    - From the third block, the program is able to print b due to backward chaining if the variable is not found in the current block
- **Closure**
    - Consider the following code:

```
Line 01: fun getCounter() {
Line 02:   var i = 0;
Line 03:   fun count() {
Line 04:     i = i + 1;
Line 05:    return i;
Line 06:   }
Line 07: }
Line 08:  var count = getCounter()
Line 09: print count() // 1
Line 10: print count() // 2
```

    - Closure is a pattern that allows the function count() to *hang on* to the surrounding var i in getCounter(). That means when count() is executed at line 9 and 10, the count() function should be able to retrieve var *i* that was declared at line 2 and increment the count.



    - When the getCounter function is visited in the AST, the global environment is passed in as closure to the getCounter’s function environment

  - When the count() function is visited in the AST, the getCounter's function environment (which encloses global) is passed as closure count's function environment.
  - When count() is executed, the code represented by *i = i + 1* will resolve to see if *i* exists in the environment inside count(). Since *i* does not exist here, it will backtrack to the getCounter's (enclosing) environment. Since *i* exists here, the *i* value is resolved as i = 0 + 1 (i = 1), when value 1 is assigned to *i*, the code again backtrack from count's environment to the getCounter's environment (where *i* exists) in order to set the *i* value to 1.

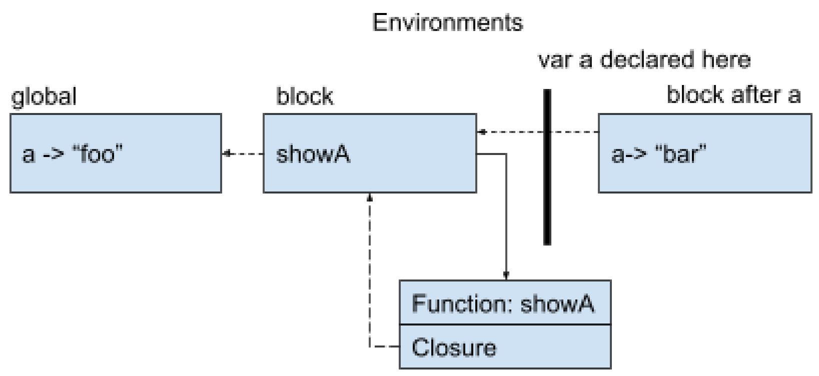
- **Resolver / Semantic Analysis**
  - A resolver is needed to patch a bug with closure where variables in a block can be mutated as shown in the following example:

```
var a = “foo”;
{
  fun showA() {
     print a;
  }
  showA() // Prints “foo”
  a = “bar”
  showA() // Prints “bar” (a bug)
}
```

  - The expected behaviour should be:

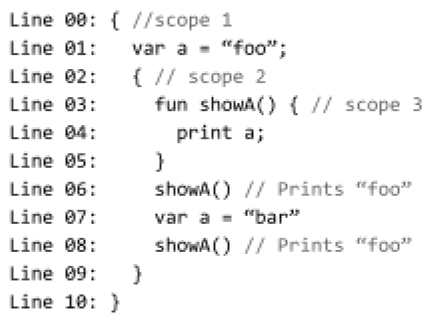
```
var a = “foo”;
{
  fun showA() {
     print a;
  }
  showA() // Prints “foo”
  var a = “bar”
  showA() // Prints “foo”
}
```

  - The reason is that when a function is declared, any variables referenced inside the function block should be ‘frozen’ (as in referring to the last known environment where it is declared) to make it immutable. This is how function closure should work in a language with first-class functions.

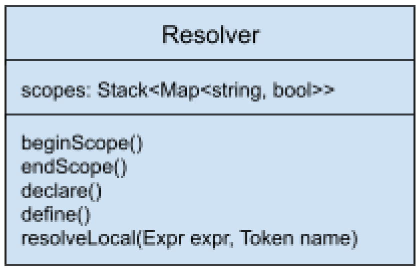


- To solve this bug, a semantic analysis technique can be used to resolve the variables between the parsing and the execution phase

- How it actually works:



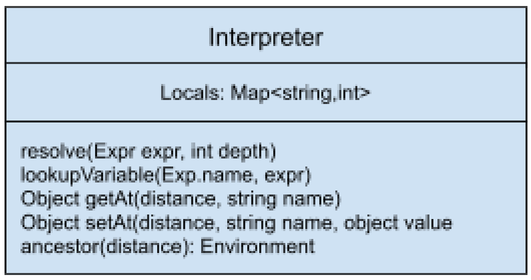
```
Line 00: { //scope 1
Line 01:   var a = "foo";
Line 02:   { // scope 2
Line 03:     fun showA() { // scope 3
Line 04:       print a;
Line 05:     }
Line 06:     showA() // Prints "foo"
Line 07:     var a = "bar"
Line 08:     showA() // Prints "foo"
Line 09:   }
Line 10: }
```



Resolver

scopes: Stack<Map<string, bool>>

beginScope()
endScope()
declare()
define()
resolveLocal(Expr expr, Token name)



Interpreter

Locals: Map<string,int>

resolve(Expr expr, int depth)
lookupVariable(Exp.name, expr)
Object getAt(distance, string name)
Object setAt(distance, string name, object value
ancestor(distance): Environment

- **Resolution steps under resolver:**
    - In line 0, the code will execute beginScope, which will push a new scope into the stack.

Stack (Resolver)	
Map (Scope 1)	{}

    - In line 1, code will execute the declare statement, resolve the initializer statement if it's not empty and finally, the define statement. Since we are at scope 1, the variable a is stored there.

Stack (Resolver)	
Map (Scope 1)	{ a -> true }

- In line 2, the code will execute beginScope in resolver, which will push the 2nd scope into the stack.

Stack (Resolver)	
Map (Scope 1)	{ a -> true }
Map (Scope 2)	{}

- In line 3: the code will execute the declare statement, define statement and resolve function on the statement. In the resolve function, beginScope is first executed, all the parameters are declared and defined

Stack (Resolver)	
Map (Scope 1)	{ a -> true }
Map (Scope 2)	{showA -> true}
Map (Scope 3)	{}

- In line 4, a is resolved. Since a is a variable, resolveLocal(..) function is executed, which does the following thing;
    - Traverse through the scopes from right to left
        - If scope contains variable/function name
            - Call interpreter.resolve(..) with the current scope in path, passing the expr and current depth.
            - The interpreter.resolve(..) will put the key: Expr object, value: depth to the locals map data structure.

Stack (Resolver)	
Map (Scope 1)	{ a-> true}
Map (Scope 2)	{showA -> true}
Map (Scope 3)	{}

locals map (Interpreter)	
< a > inside scope 3	2

- In line 5, the endScope of the function is executed, which will pop the last scope from the stack.

Stack (Resolver)	
Map (Scope 1)	{ a-> true}
Map (Scope 2)	{showA -> true}
~~Map (Scope 3)~~	~~{}~~

- In line 6, the function call is visited, and because the function is stored as a variable, resolveLocal(..) is executed.

locals map (Interpreter)	
< a > inside scope 3	2
< showA > inside scope 2	1

- In line 7, code will execute the declare statement, resolve the initializer statement if it's not empty and finally, the define statement (for shadow var a)

Stack (Resolver)	
Map (Scope 1)	{ a-> true}
Map (Scope 2)	{showA -> true, a-> true}

- In line 8, the same code as line 6 is executed.
- In line 9, endScope is executed, which will pop the last scope from the stack.

Stack (Resolver)	
Map (Scope 1)	{ a-> true}
~~Map (Scope 2)~~	~~{showA -> true, a-> true}~~

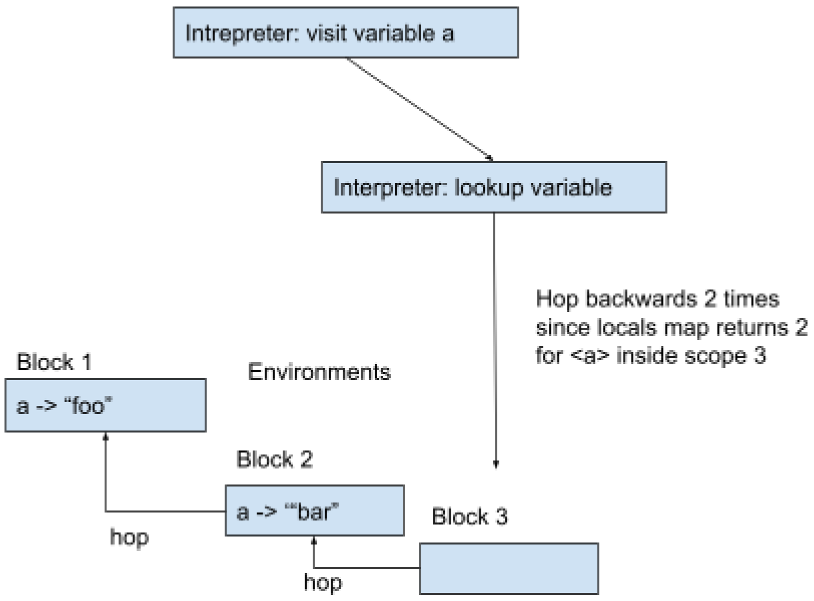
- In line 10, endScope is executed, which will pop the last scope from the stack.

Stack (Resolver)	
~~Map (Scope 1)~~	~~{ a-> true}~~

- **Interpretation steps:**
    - Recall that the locals table for the Interpreter (after running resolver step) is:

locals map (Interpreter)	
< a > inside scope 3	2
…	…

- Line 06: showA()
    - When the body of the function node is visited in the parse tree and the print i node is encountered,
    - The interpreter will look up the variable “a“ inside the function by checking the locals map for the distance (depth) to the ancestor that declared variable ‘a’.
    - Since the distance is 2, the code will ‘hop’ through the environment backward (backtracking) twice to lookup the variable ‘a’
    - The resulting output is ‘foo’
- Line 08: showA()
    - Same as above.


Intrepreter: visit variable a
Interpreter: lookup variable
Hop backwards 2 times
since locals map returns 2
for <a> inside scope 3
Block 1
Environments
a -> “foo”
Block 2
a -> ““bar”
hop
Block 3
hop