NTNU | Norwegian University of Science and Technology

Compiler Construction

Practical Exercise 5: Symbol table construction

Guidelines and hints

Michael Engel

Symbol Tables

- This exercise builds the next step of your compiler, enabling it to perform *semantic analysis* and, in the final code generation exercise, then generate code to address variables and functions

- We organize identifiers and strings so that we can resolve them to memory locations in the finished program

- Variable names and function names are text strings, so we’ll need to index a table based on those

- For this purpose, PE5_skeleton.zip comes with a hash table implementation
  - The hash table in the standard library is not really usable so a separate hash table implementation has been provided for this exercise
  - Ours is not a high performance solution but for this sake of this exercise this is adequate

Using the functions in `tlhash.h/c`

- The interface has functions to handle tlhash_t structs, that is
  - initialize
  - finalize
  - insert
  - lookup
  - remove
  - obtain all keys
  - obtain all values

- Keys and values are just `void` pointers, the caller program needs to manage what they point

- The `symbol_t` struct should be used for this

struct symbol_t

- The struct for the symbol table is located in `ir.h`:

```
typedef struct s {
    char *name;         // string: name related to current symbol

    symtype_t type;     // enum: function, global, local, parameter?

    node_t *node;       // root node (of type function)

    size_t seq;         // sequence number (not for global variables)

    size_t nparms;      // number of parameters (for functions)

    tlhash_t *locals;   // hash table of local names
}
```

Task 1

- The skeleton already initializes a global symbol table (`global_names`)
- Implement `find_globals`:
  - Fill the `global_names` table with symbol structs for
    - functions
    - global variables
- Functions require their own name table
  - this can already be filled in with the parameter names
- Functions also link to their tree node
  - thus we can traverse a function's subtree when we know the function's name
- Number
  - the parameters
  - and also the functions

Task 2

- Traverse each function's subtree
  - resolve names (and strings) within each function's scope
    - implement `bind_names` to do this
  - the subtree construction will consist of a mix of
    - entering declared names into the function's local table and
    - linking the used names to the symbol they represent
- Number local variables
- Look up used identifiers hierarchically
  - first in the local scope
  - if not found in the local scope, look up the identifier in the global scope
- Create a global index of string literals

Task 3

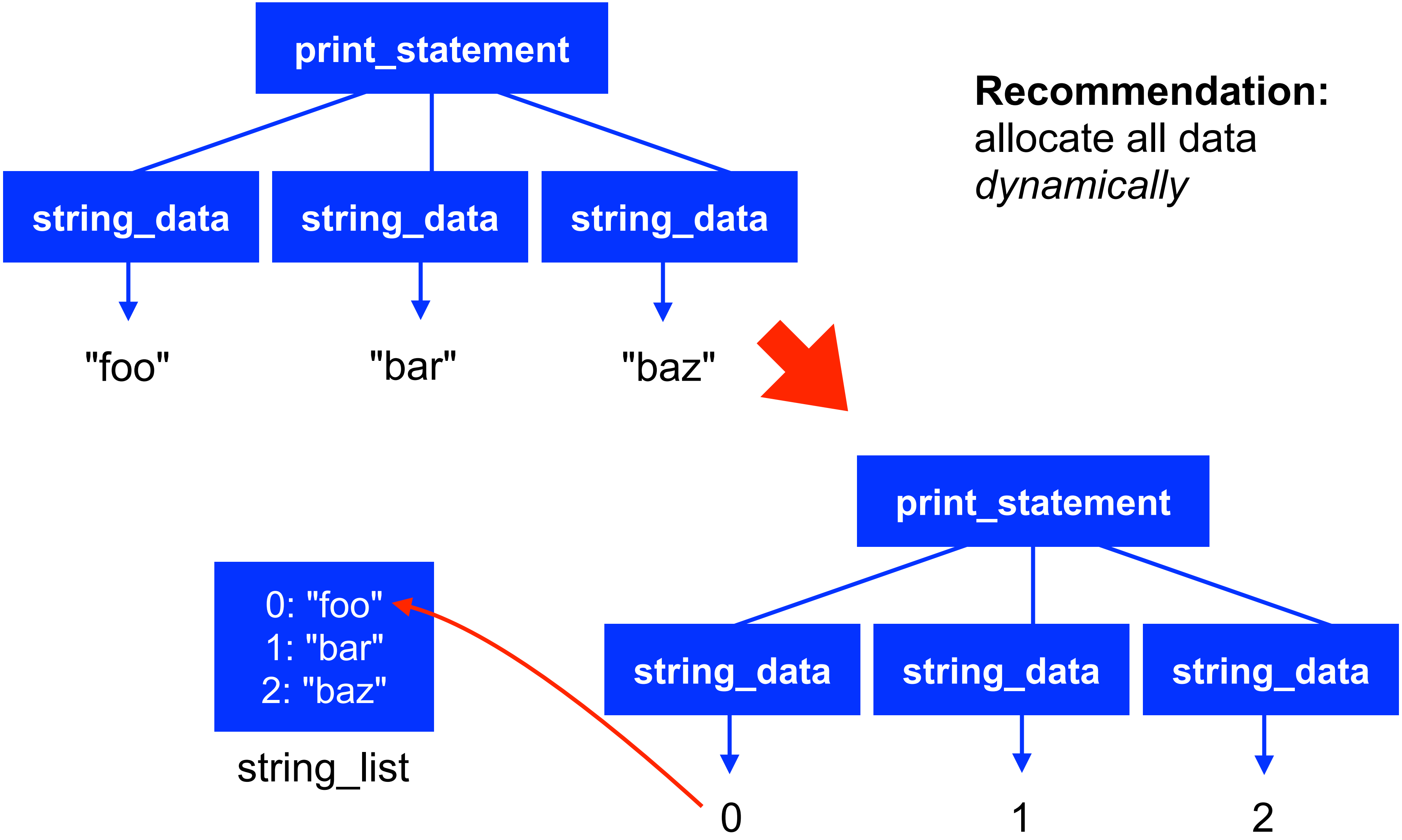
- Destroy the whole structure that you created
  - used when it is no longer needed
  - implement the `destroy_subtree` function to do this
- The exact implementation depends on your implementation

Global string literal index

- Strings are only used once and that happens in the node that represents them
  - The node presently contains a pointer to the string at the data element
  - When the time comes to generate code, it would be nice to display all the strings at once

- Therefore:
  - Take the pointer and put it in the global `string_list`
  - Keep a count of strings (`stringc`)
  - Remember to size up and resize (grow) the table as appropriate
  - Replace the node’s data element with the number of the string it used to hold

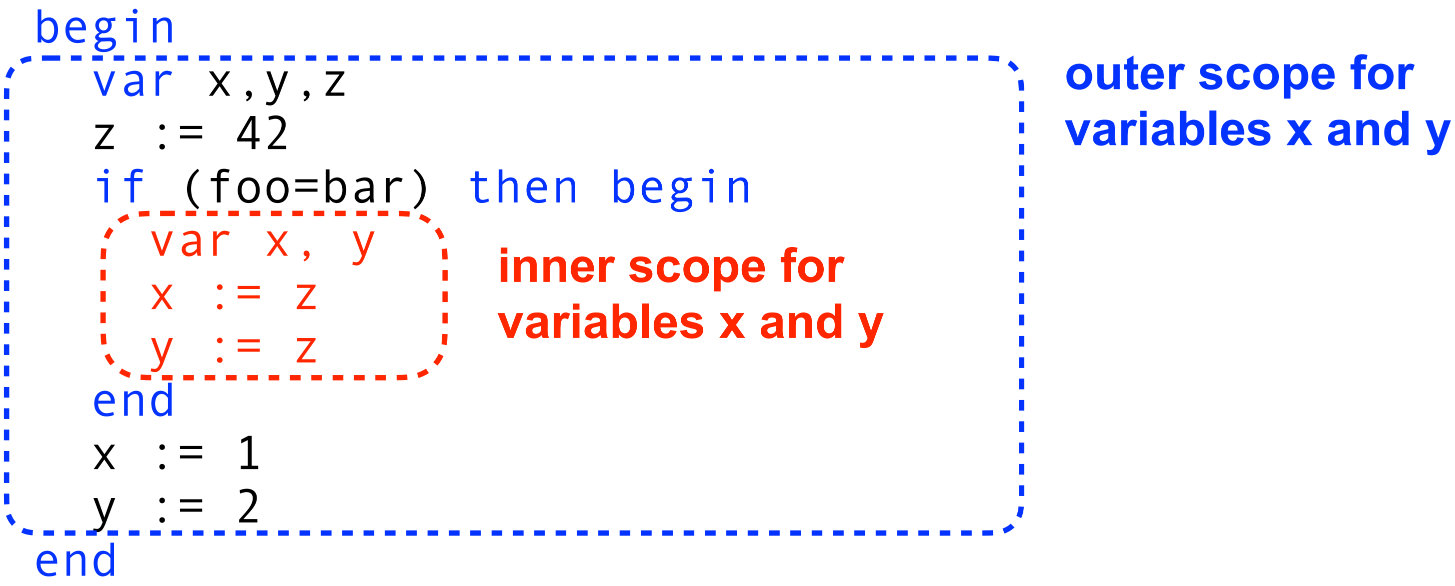
Global string literal index: example



Recommendation: allocate all data *dynamically*

Local name tables

- We have a problem…
- The following code is legal in VSL:



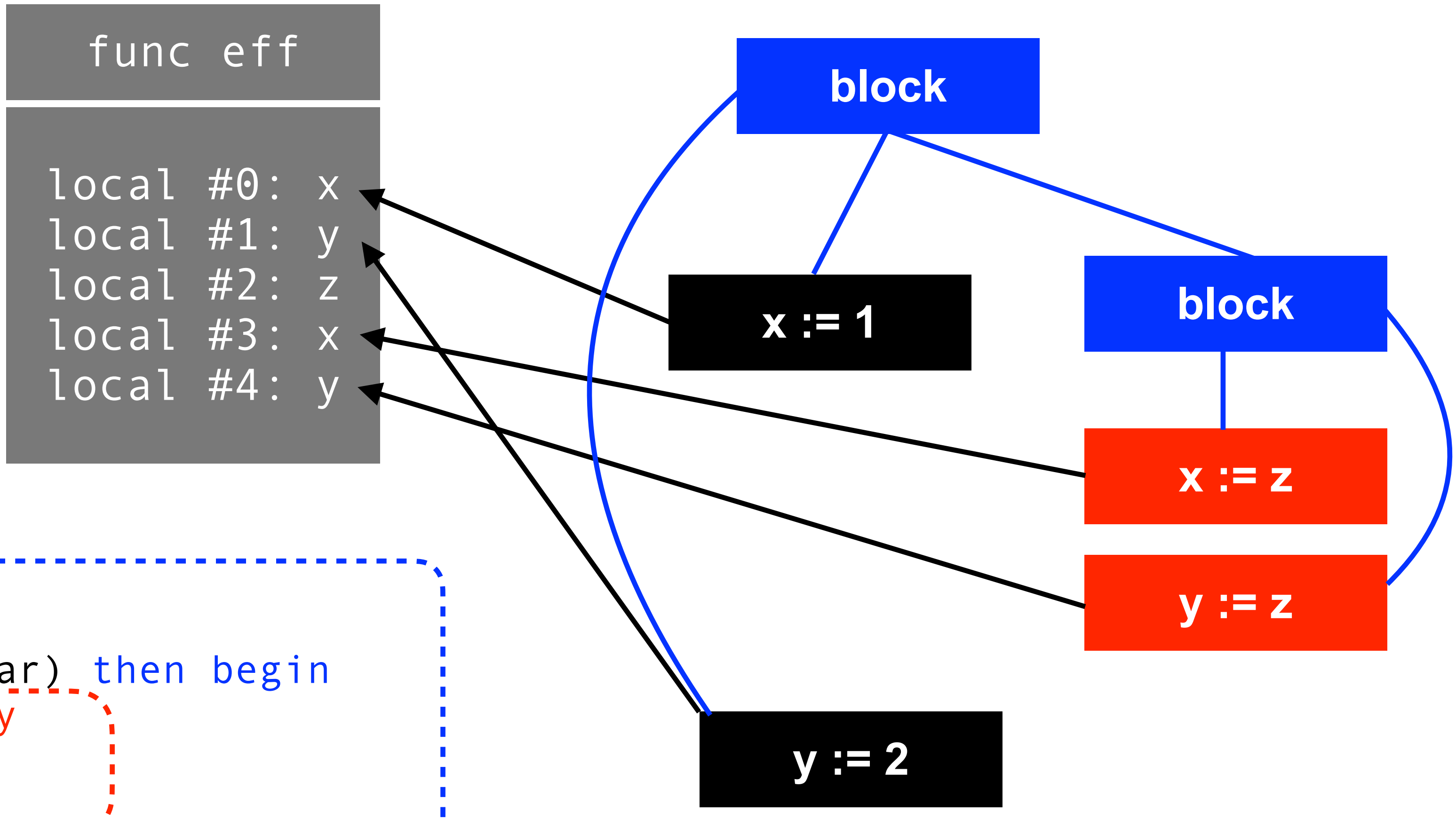
```
begin
   var x,y,z
   z := 42
   if (foo=bar) then begin
      var x, y
      x := z
      y := z
   end
   x := 1
   y := 2
end
```

outer scope for variables x and y

inner scope for variables x and y

- There are outer `x , y` and inner `x , y`: ***these are not the same variables***
- In the end, we want them in a single, local table for the function

Local name table implementation

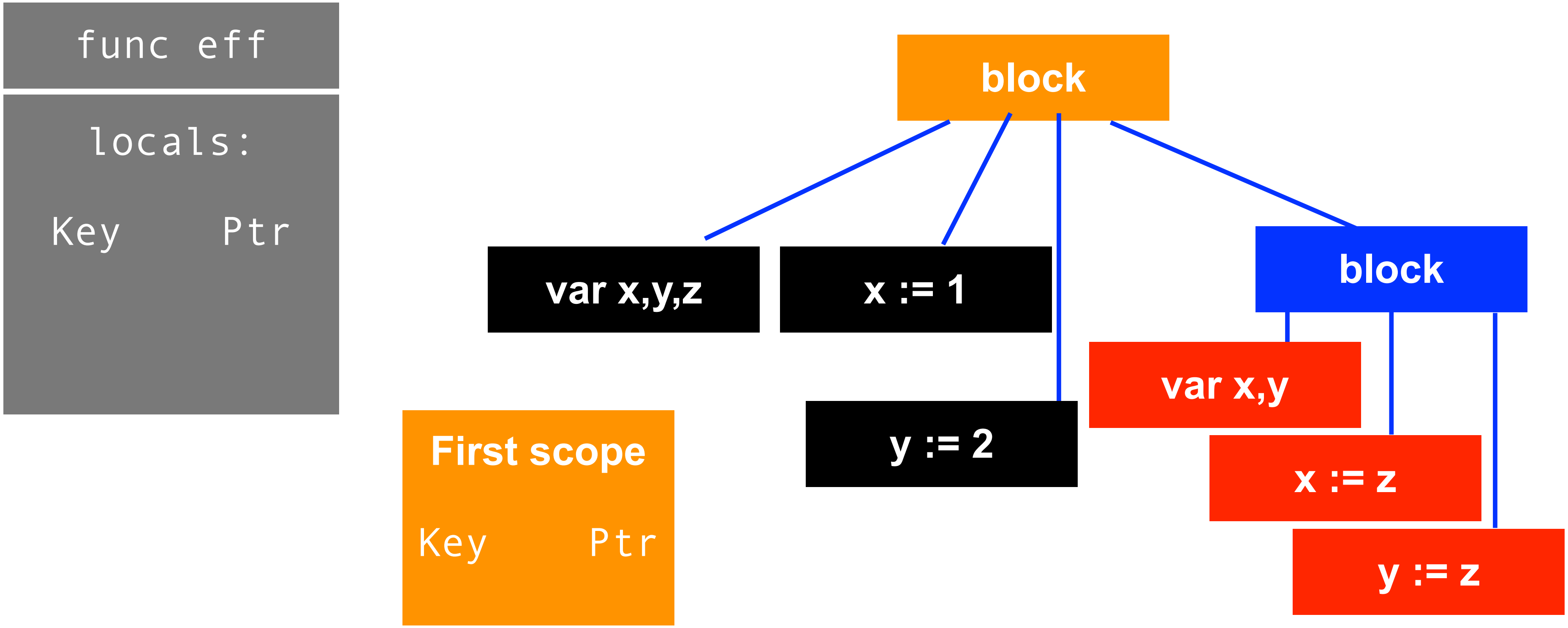


```
func eff

local #0: x
local #1: y
local #2: z
local #3: x
local #4: y
```

```
begin
    var x,y,z
    z := 42
    if (foo=bar) then begin
        var x, y
        x := z
        y := z
    end
    x := 1
    y := 2
end
```

Avoiding name clashes for local vars



func eff

locals:

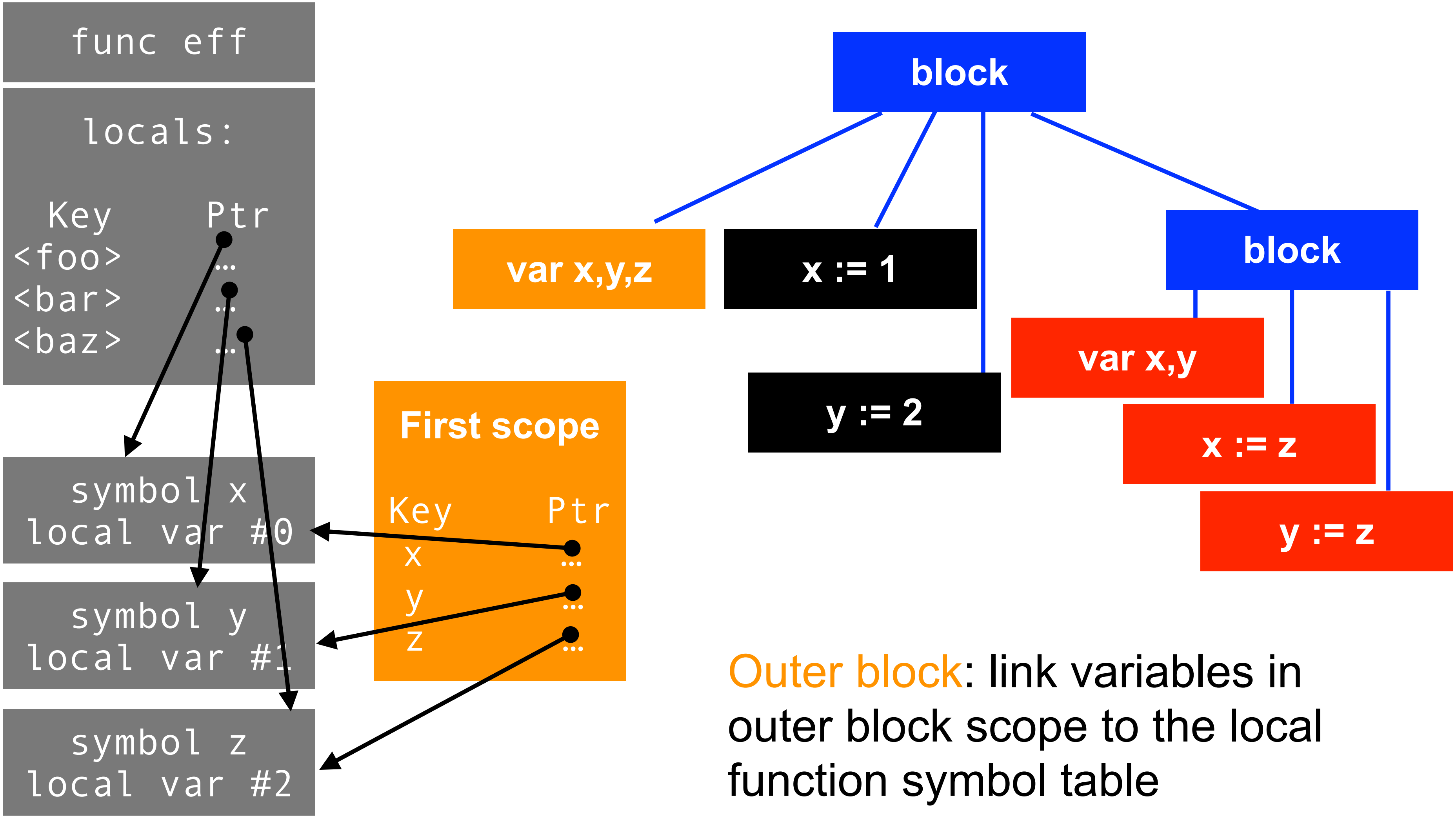
Key Ptr

First scope

Key Ptr

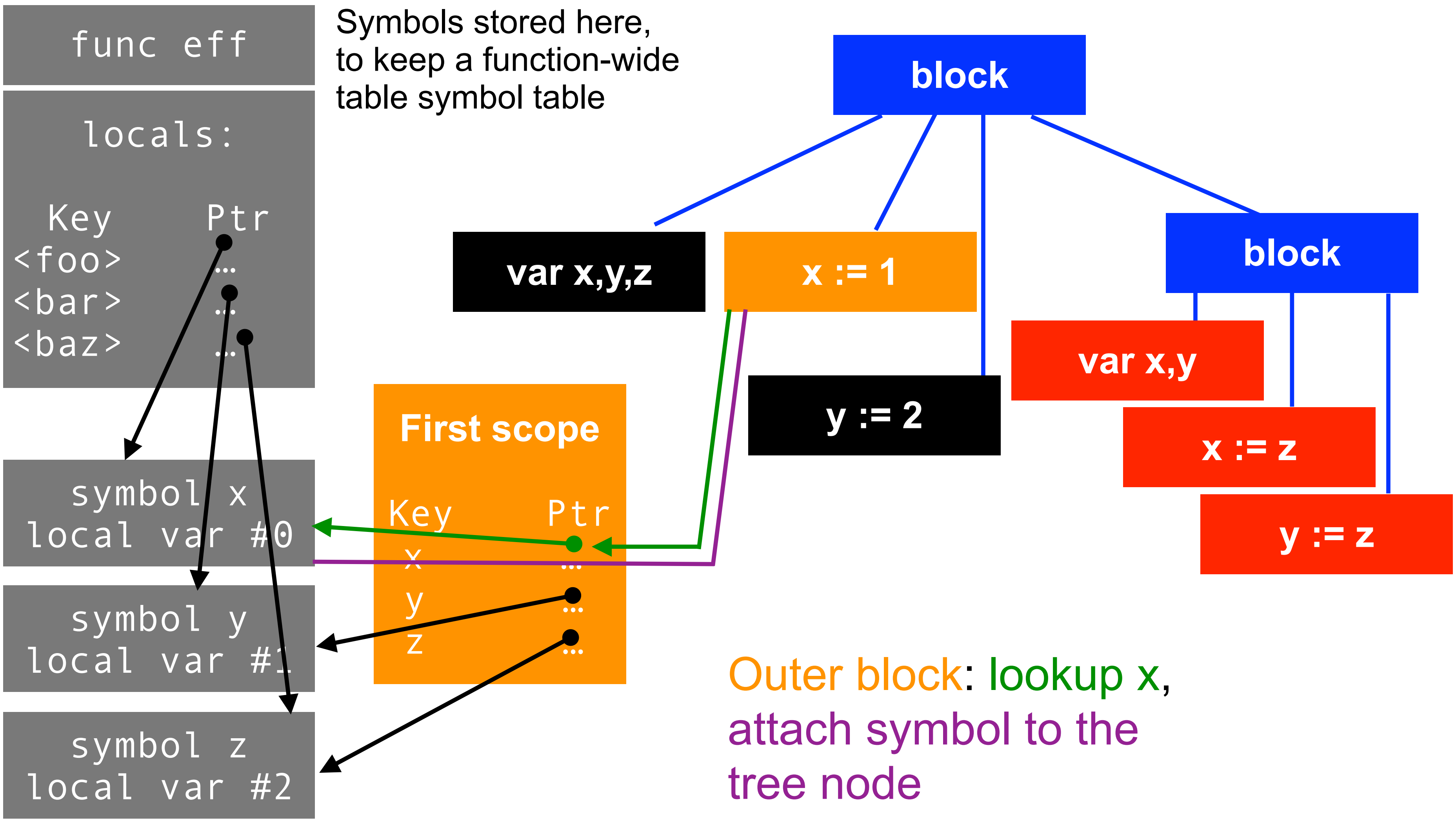
Outer block: create a new scope!

Avoiding name clashes for local vars



Outer block: link variables in outer block scope to the local function symbol table

Avoiding name clashes for local vars



Symbols stored here, to keep a function-wide table symbol table

func eff

locals:

Key Ptr
<foo> …
<bar> …
<baz> …

symbol x
local var #0

symbol y
local var #1

symbol z
local var #2

First scope

Key Ptr
x …
y …
z …

block

var x,y,z

x := 1

y := 2

block

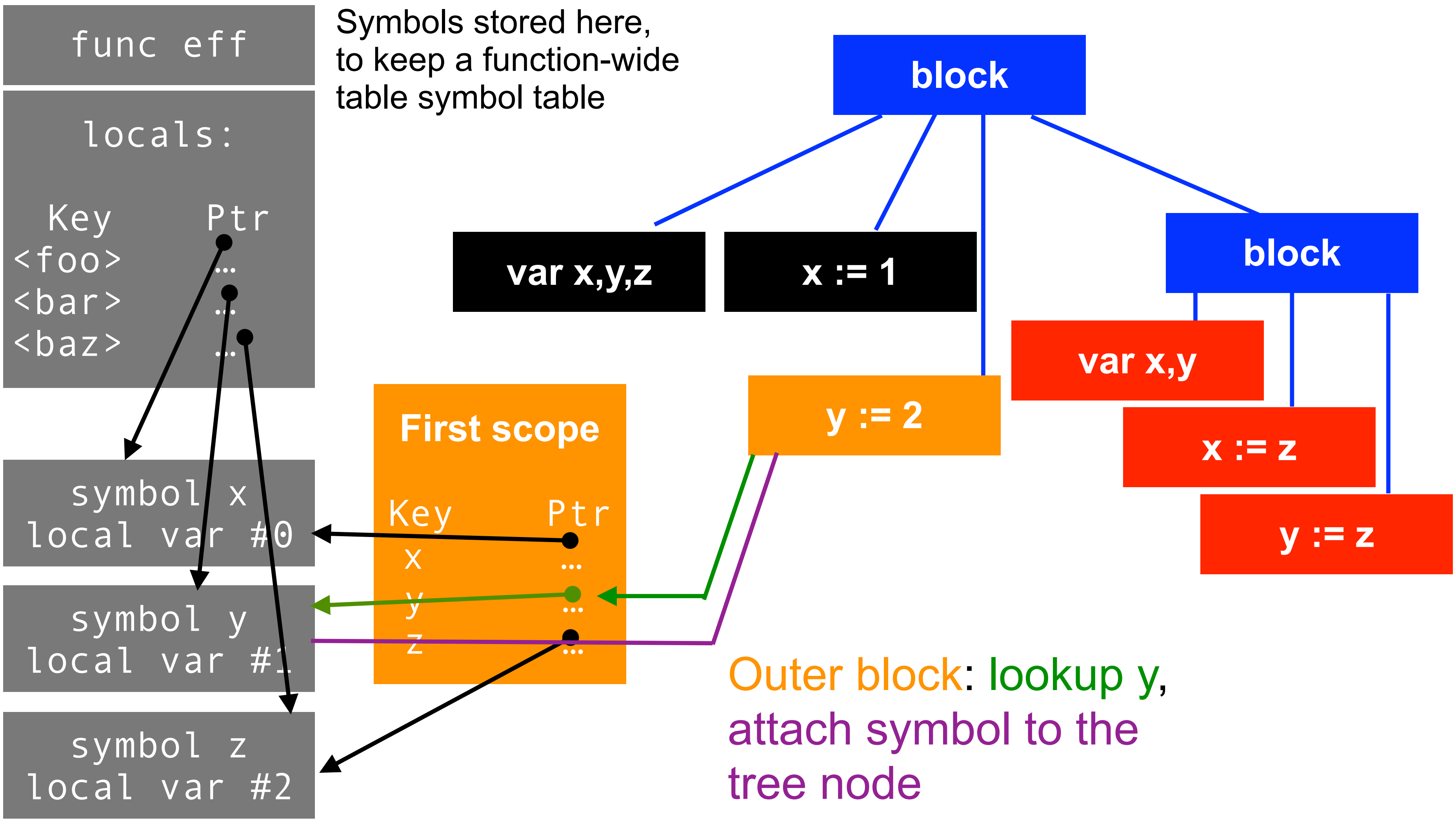
var x,y

x := z

y := z

Outer block: lookup x, attach symbol to the tree node

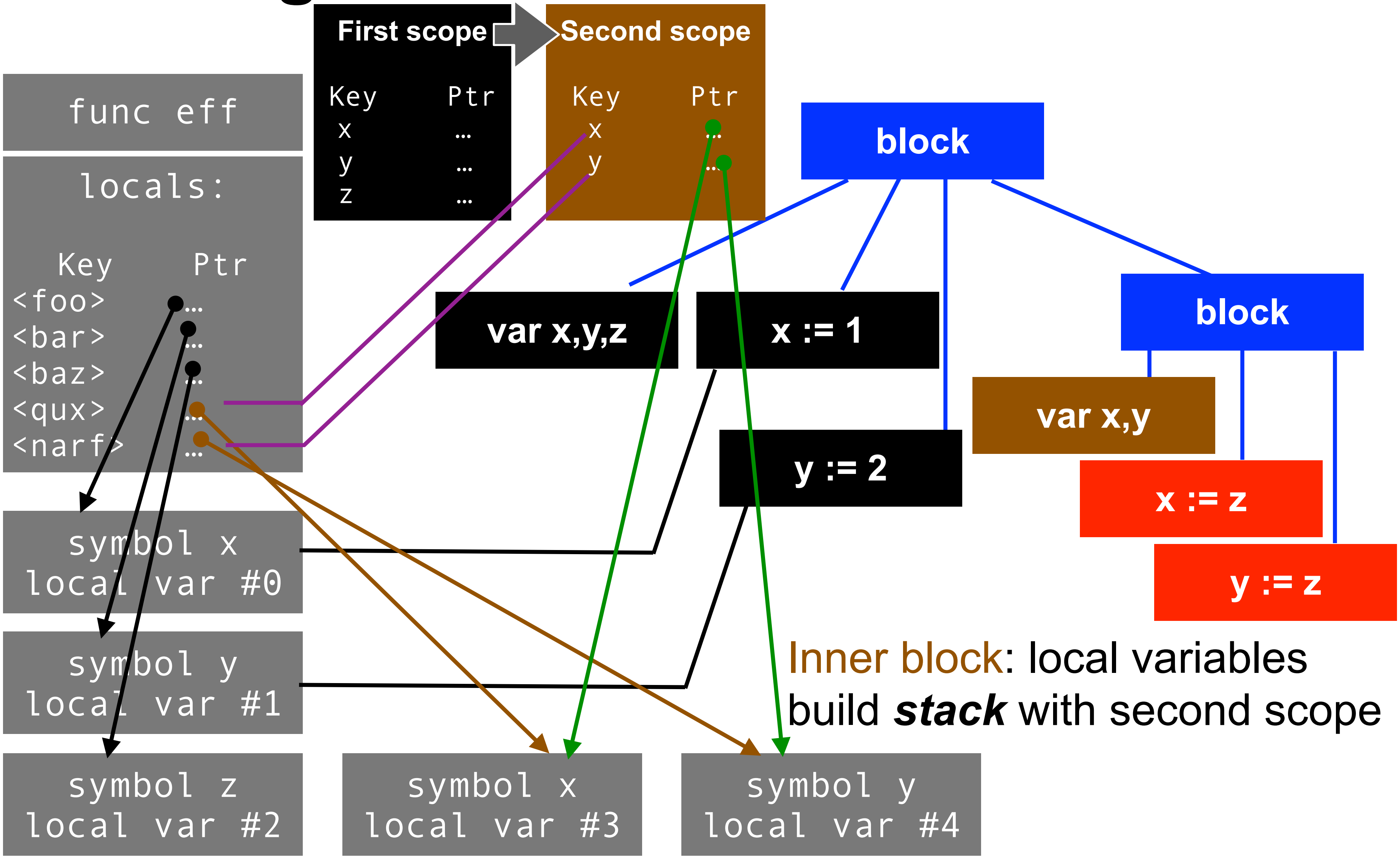
Avoiding name clashes for local vars



Symbols stored here,
to keep a function-wide
table symbol table

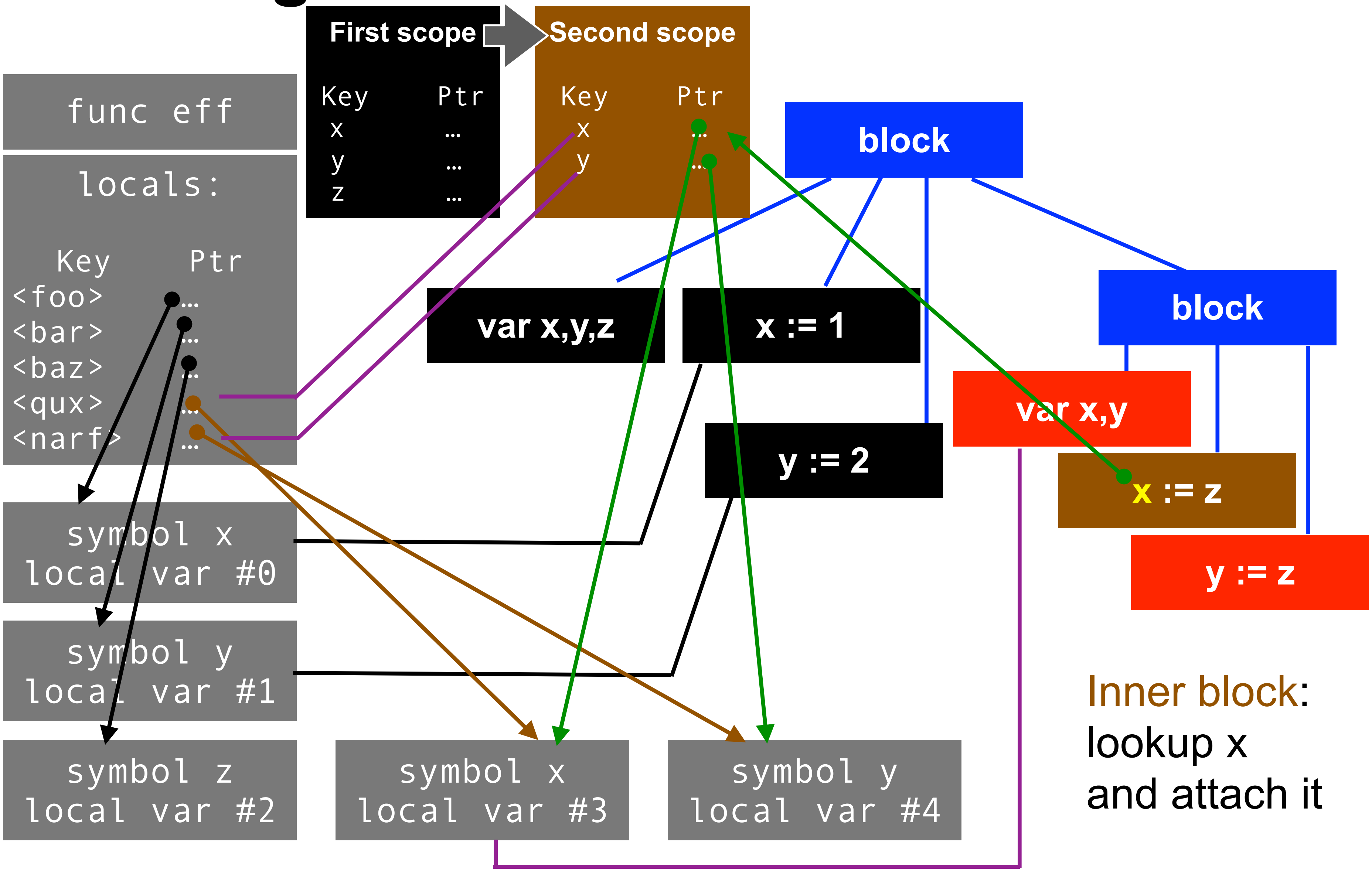
Outer block: lookup y, attach symbol to the tree node

Avoiding name clashes for local vars



Inner block: local variables build ***stack*** with second scope

Avoiding name clashes for local vars



func eff

locals:

Key	Ptr
<foo>	…
<bar>	…
<baz>	…
<qux>	…
<narf>	…

First scope

Key	Ptr
x	…
y	…
z	…

Second scope

Key	Ptr
x	…
y	…

block

var x,y,z

x := 1

y := 2

block

var x,y

x := z

y := z

symbol x
local var #0

symbol y
local var #1

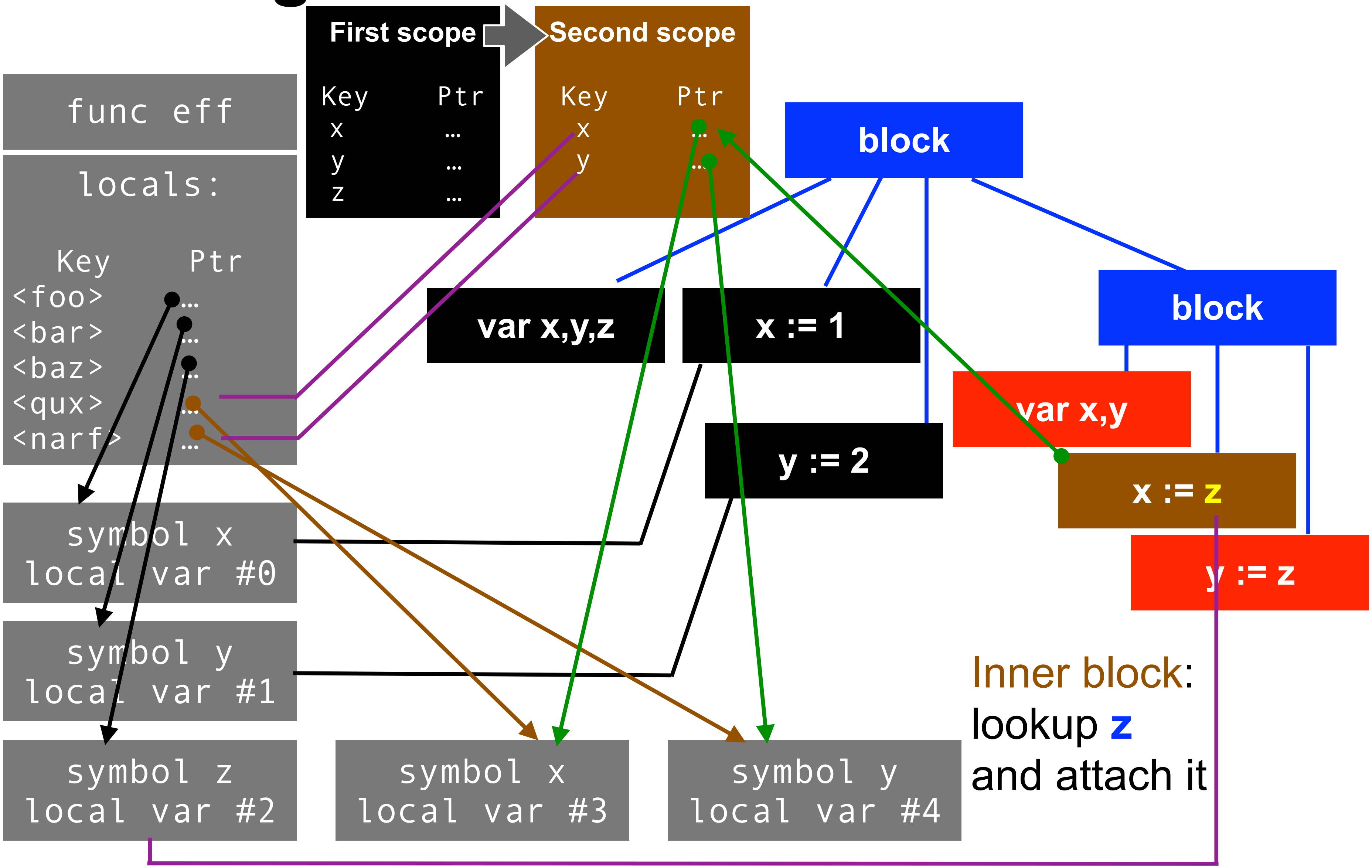
symbol z
local var #2

symbol x
local var #3

symbol y
local var #4

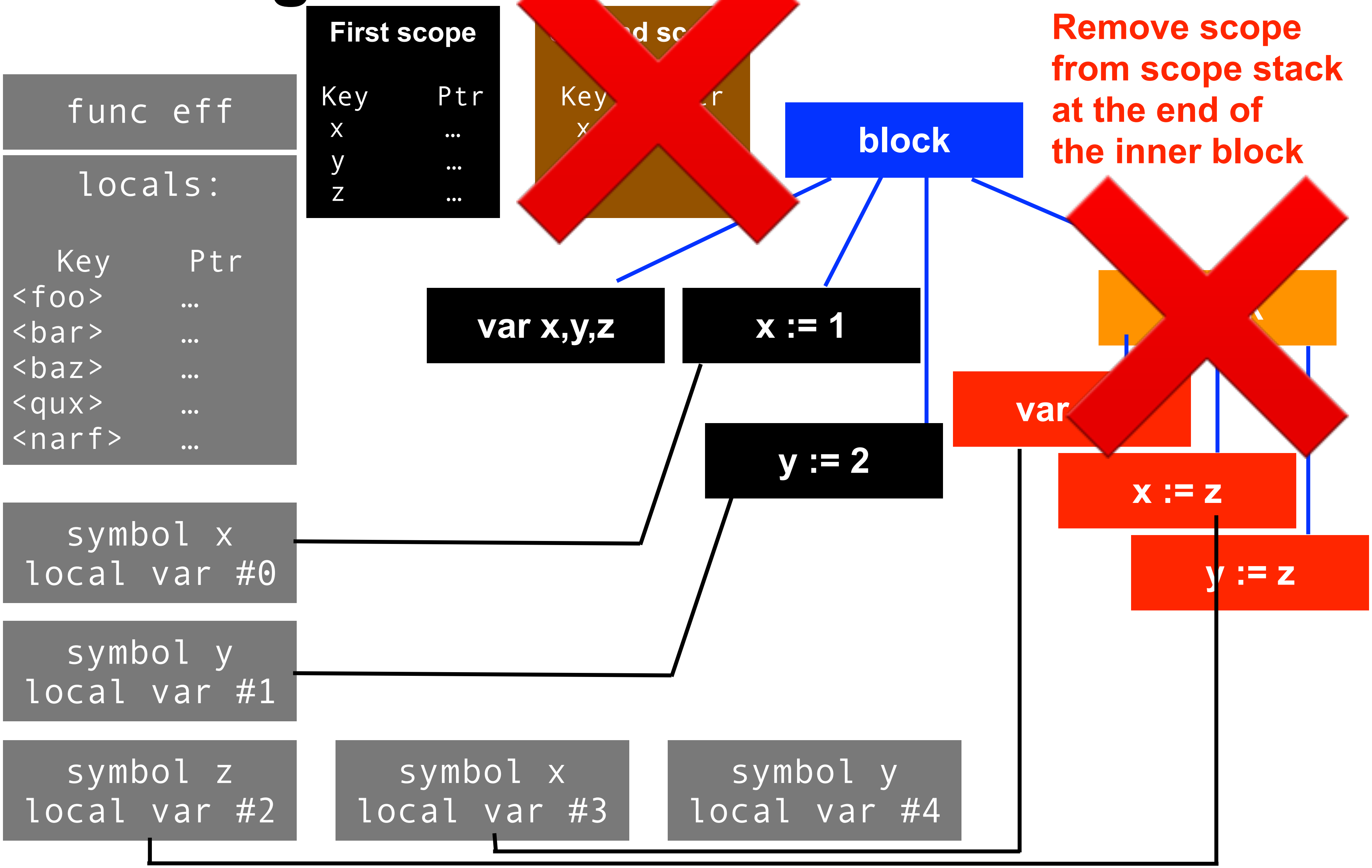
Inner block:
lookup x
and attach it

Avoiding name clashes for local vars



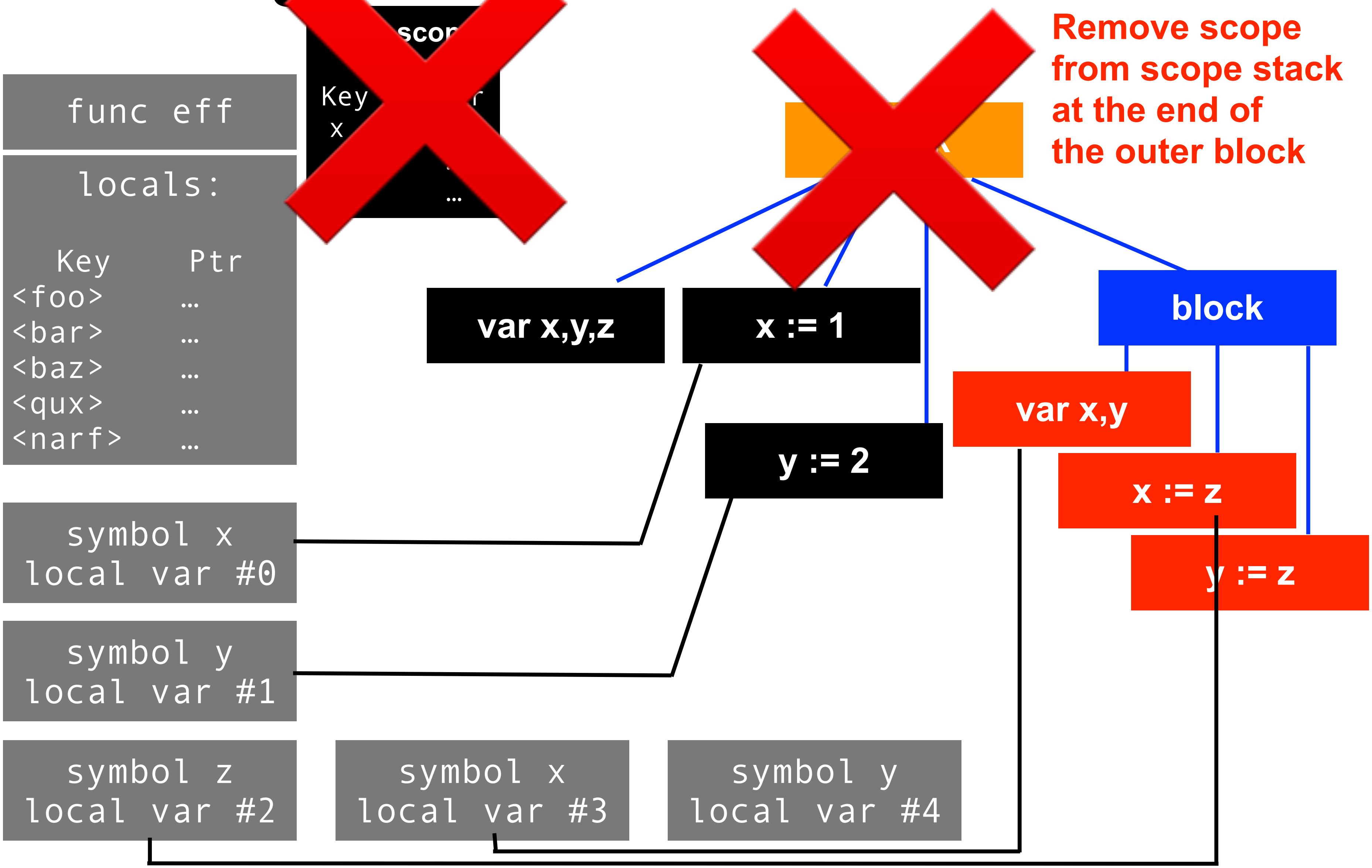
Inner block:
lookup **z**
and attach it

Avoiding name clashes for local vars



Remove scope from scope stack at the end of the inner block

Avoiding name clashes for local vars



func eff

locals:

Key	Ptr
<foo>	…
<bar>	…
<baz>	…
<qux>	…
<narf>	…

scop

Key

x

…

var x,y,z

x := 1

y := 2

block

var x,y

x := z

y := z

symbol x
local var #0

symbol y
local var #1

symbol z
local var #2

symbol x
local var #3

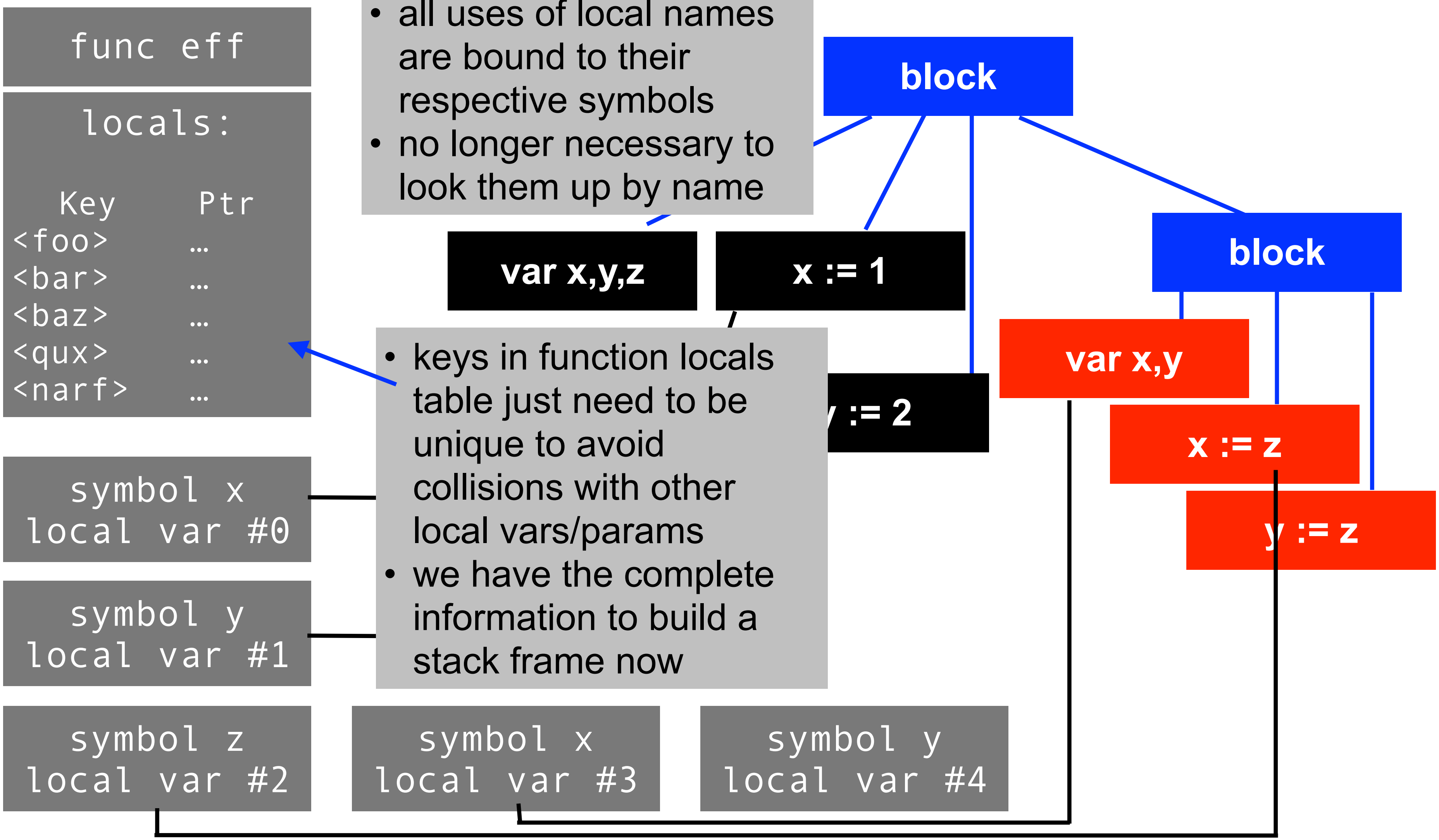
symbol y
local var #4

Remove scope from scope stack at the end of the outer block

Avoiding name clashes for local vars

Cleaned up:

- all uses of local names are bound to their respective symbols
- no longer necessary to look them up by name



func eff

locals:

Key	Ptr
<foo>	…
<bar>	…
<baz>	…
<qux>	…
<narf>	…

symbol x
local var #0

symbol y
local var #1

symbol z
local var #2

symbol x
local var #3

symbol y
local var #4

block

var x,y,z

x := 1

y := 2

block

var x,y

x := z

y := z

- keys in function locals table just need to be unique to avoid collisions with other local vars/params
- we have the complete information to build a stack frame now

Semantic errors

- When looking up names, we can now determine if they are properly declared or not

- It is useful to add error messages indicating a semantic error if you use your own test programs (especially ones to test if the compiler catches the error…)

- What should you do when you encounter an incorrect program?
  - This is not specified
  - Print an error message, crash and burn, format your hard disk 😂…

- "Real" programming languages have different *data types*
  - this would require storing/verifying type information…not in VSL

Blocks need a name table

- Only temporarily:
  - While traversing the inner block, looking up x should return symtab entry #3
  - When the inner block is finished, go back to look up x as symtab entry #0
- We can use a *stack* of temporary hash values
  - Push a new one when a block begins
  - Put in locally declared names, make them point to the real symbol table entry
  - Look up names recursively in bottom-up order to find the closest scope for a name
  - Pop the temporary table off the stack at the end of the block
- After each node has been linked to the correct symbol table entry, its name no longer is relevant ***but…***
- we need to number local variables so we can tell apart variables with the same name on different scopes

Tree dump in current skeleton code

- `print_symbols` and `print_bindings` are already written (in `ir.c`), they are meant to display
  - the string table
  - the names and indices of contents in global and local symbol tables
  - the symtab entries linked from tree nodes

- It could happen that your tree dump looks a bit different from the ones supplied in PE5_sources_and_output.zip
  - Particularly, if you hash differently, elements might come out sorted in different orders,
  - They are not sorted by sequence numbers here

Hints

- ***The indices of functions, parameters, local variables should match*** our example output up to the order things appear in

- Those follow from the structure of the input program, so there's a correct order to count them in, regardless of how you implement it

- These sequence indices are not arbitrary

    - It's not enough that they are unique numbers, so it won't do to keep a single counter and use it for everything

    - In the next (and final) exercise, we will use them to calculate addresses in machine-level code

    - Please don't invent alternative numbering schemes