NTNU – Trondheim
Norwegian University of
Science and Technology

Problem set 4

Symbol table(s)

- The task this time is to 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, ps4_skeleton comes with a *hash table* implementation

Hash tables in C

- There is a hash table of sorts in the standard library, but in my opinion, it is awful.
- The provided *tlhash.[h|c]* is a simple implementation that uses CRC32 hashing and modulates the checksum over a count of table entries (each is a linked list).
- This is not a certifiably fantastic high-performance solution, but we won't be hashing that much data anyway.

  (You can provide your own table if you don't like mine, I supply it so that you don't *have* to; implementing hash tables is a topic for a different course.)

Using 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, managing what they point to is for the calling program to care about.

  (*tl* is my shorthand for *typeless*)

Using tlhash.h/c

- A general walkthrough of this wouldn't require keys to be strings, or values to be pointers to structs

  (it's some code I've recycled in various contexts)
- Since that's what we'll be using, though, I've written up a small example that exercises all the functions, in hash_examples.tgz under the 'examples' folder on it's learning.
- Hopefully, you should be able to employ it in a similar manner, just pointing at symbol_t structs instead.

symbol_t structs

```
typedef struct s {
    char *name;
    symtype_t type;
    node_t *node;
    size_t seq;
    size_t nparms;
    tlhash_t *locals;
} symbol_t;
```

- These are what 'entry' in the nodes are meant to point to, so that we can link nodes with names to what they symbolize

symbol_t structs

```
typedef struct s {
    char *name;
    symtype_t type;
    node_t *node;
    size_t seq;
    size_t nparms;
    tlhash_t *locals;
} symbol_t;
```

Text (name)

Enumeration: functions, global vars, parameters, or local vars

Root node (of function)

Sequencing number
(for everything but global vars)

symbol_t structs

```
typedef struct s {
    char *name;
    symtype_t type;
    node_t *node;
    size_t seq;
    size_t nparms;
    tlhash_t *locals;
} symbol_t;
```

Parameter count
(for functions)

Hash table of local names
(for functions)

Thing #1 to do

- Skeleton already initializes a global symbol table (global_names)
- Fill it with symbol structs for functions and global vars, i.e. implement find_globals
- Functions will need their own name table in addition, it can already be filled in with the parameter names
- Functions also link to their tree node (so that we can traverse a function's subtree when knowing its name)
- Number the parameters
- Number functions too

Thing #2 to do

- Traverse each function's subtree, resolve names (and strings) within its scope, i.e. implement bind_names
- This will be a mixture of entering declared names into its local table, and linking used names to the symbol they represent
- Number local variables
- Look up used identifiers first locally, then globally
  - This is a little tricky, more in a moment
- Create a global index of string literals
  - This is not so tricky, but more in a moment

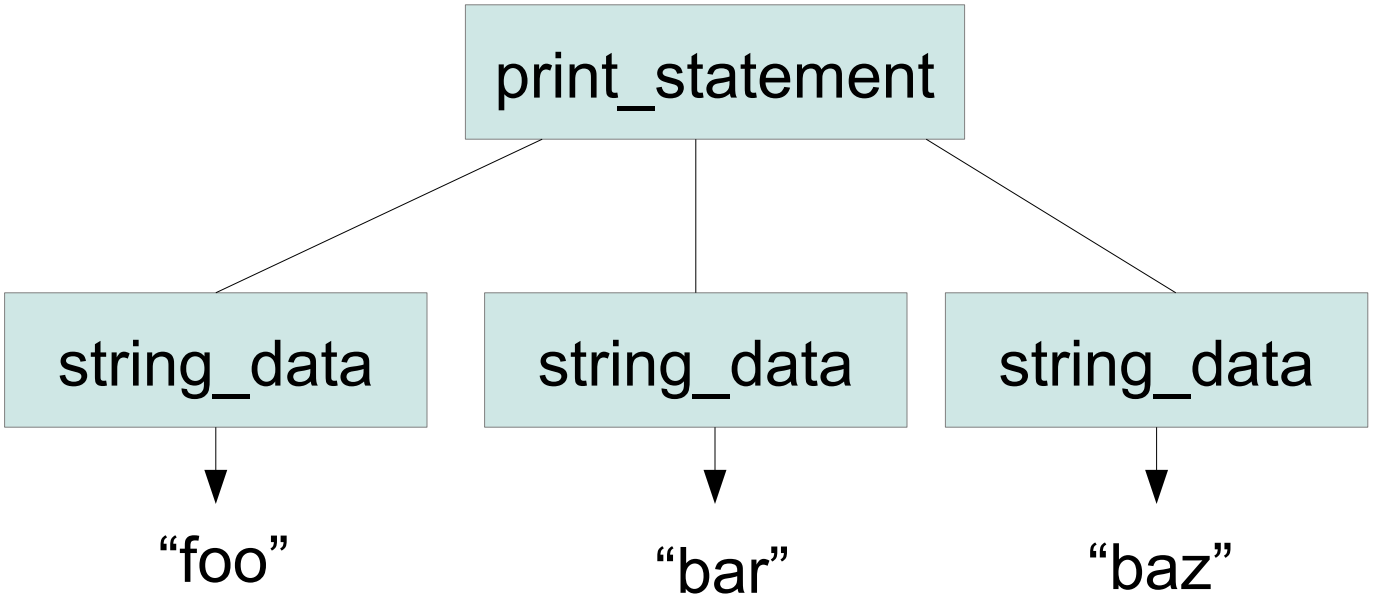
Thing #3 to do

- Take down the whole structure you created, i.e. implement destroy_symtab
- This might depend a bit on how you choose to build it, but you'll know how you did that when you've done it.

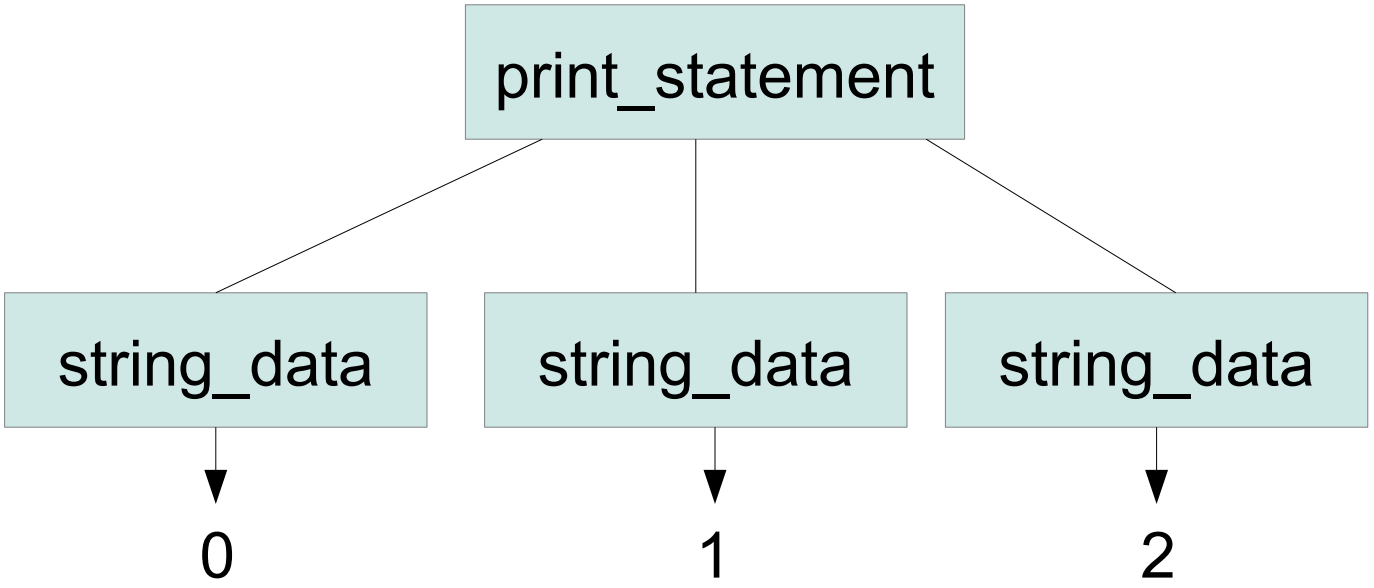
A global index of string literals

- Strings are only used once, i.e. 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, we'll want to blurt out 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

For example:


print_statement
string_data
string_data
string_data
“foo”
“bar”
“baz”

Becomes:



string_list

[0] “foo”
[1] “bar”
[2] “baz”

As usual, I recommend dynamically allocating everything for regularity, but you're the author

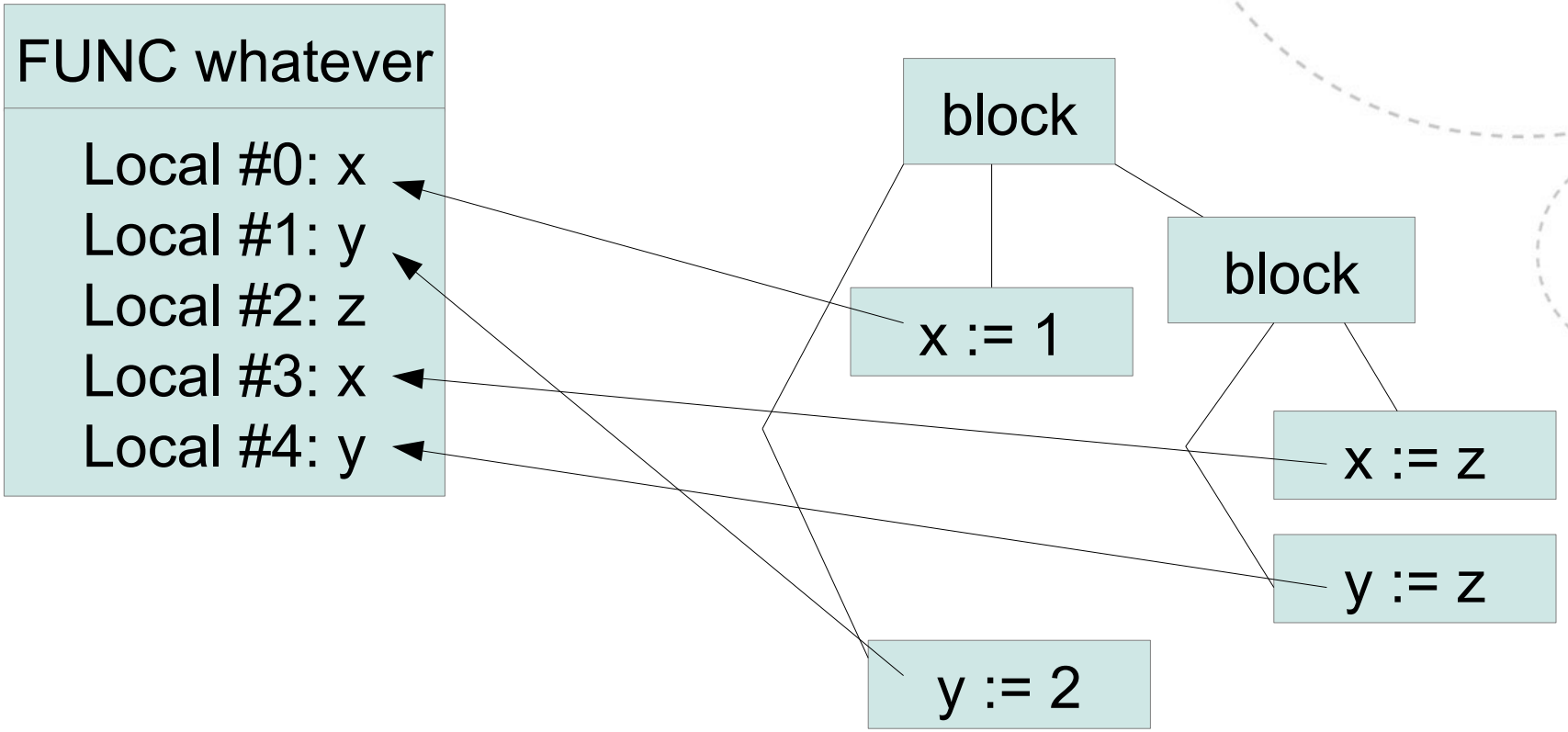
Local name tables

- Houston, there will be a problem
- VSL admits

```
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
```

- 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

In other words



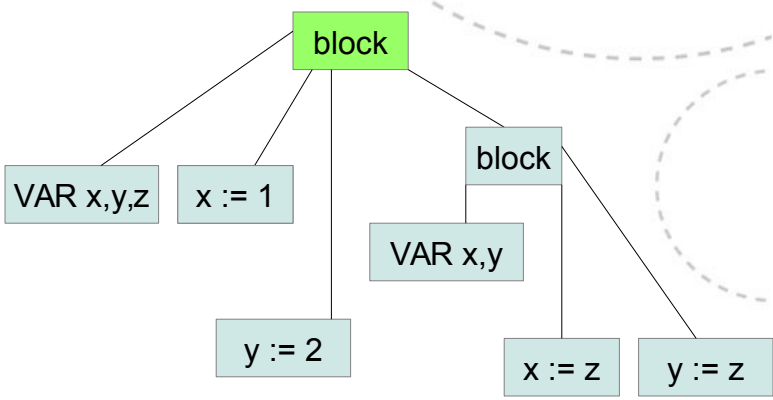
Blocks need a name table

- But only temporarily:
  - While traversing the inner block, looking up “x” should result in the symtab entry for local #3
  - When it's finished, we go back to looking up “x” as the symtab entry for local #0
- We can use a *stack* (yay!) of temporary hash tables
  - Push a new one when a block begins
  - Put in locally declared names, make them point onwards to the real symtab entry
  - Look up names in top-to-bottom order, to resolve closest defining scope
  - Pop the temporary table off your stack when the block has ended
- After each node has been linked to the correct symtab entry, it no longer matters what they are called, *but*
- Number local variables, so that we can tell inner and outer x-s and y-s apart

Avoiding name clashes among local variables

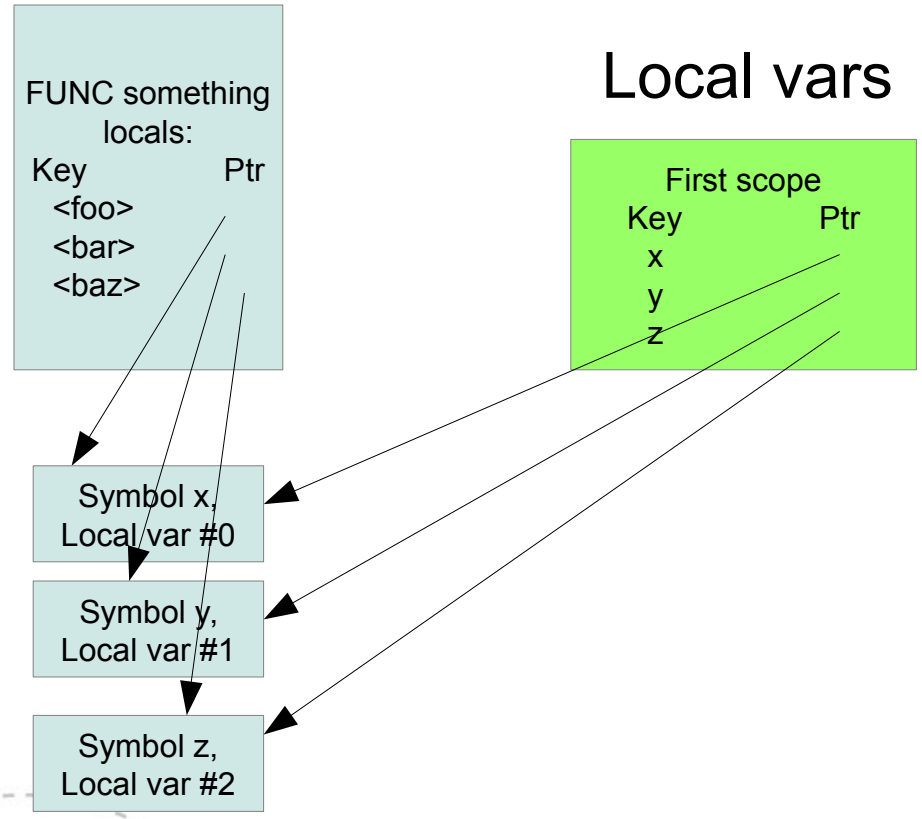
FUNC something
locals:
Key Ptr

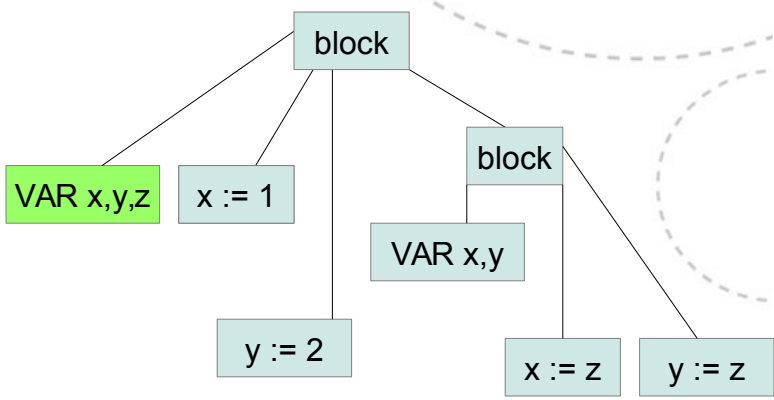
New scope!



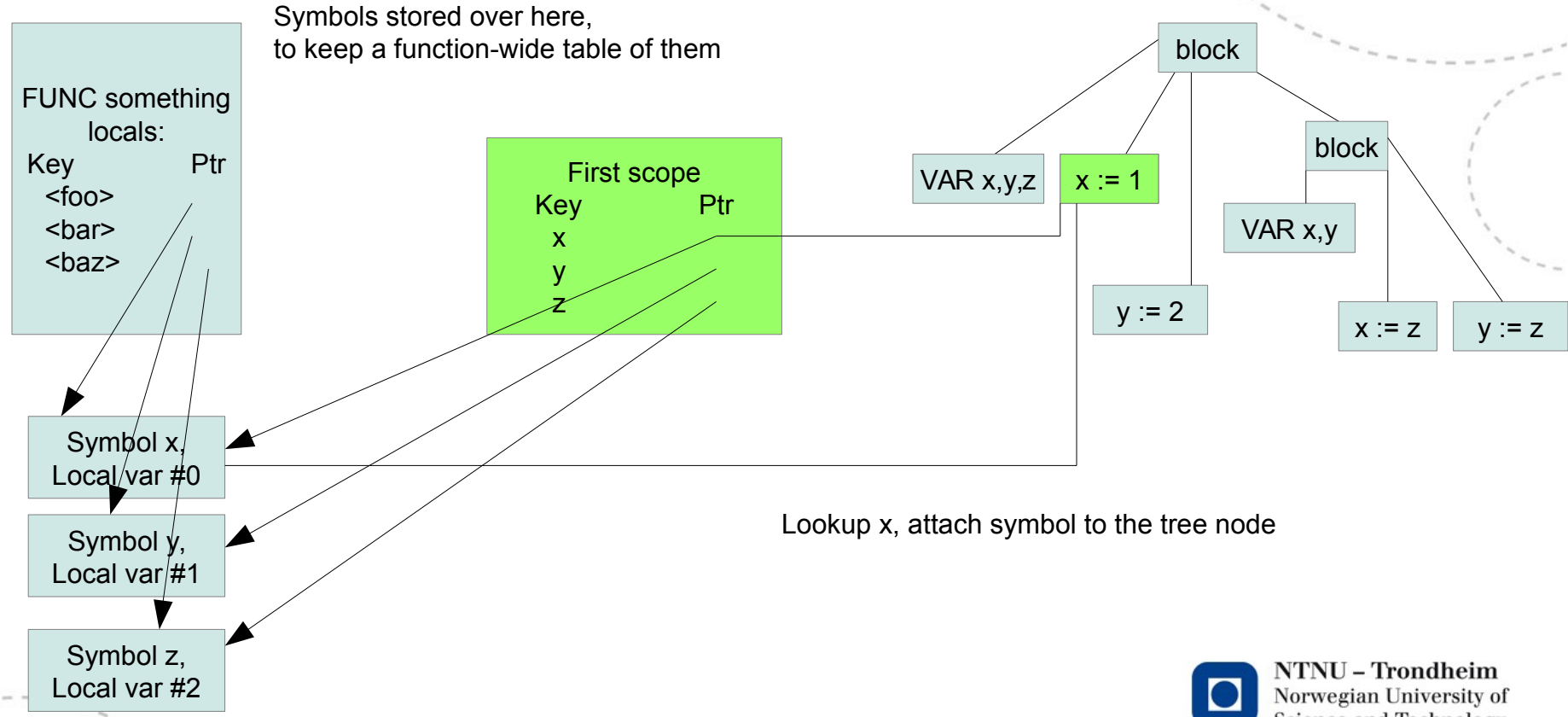
First scope
Key Ptr

Avoiding name clashes among local variables

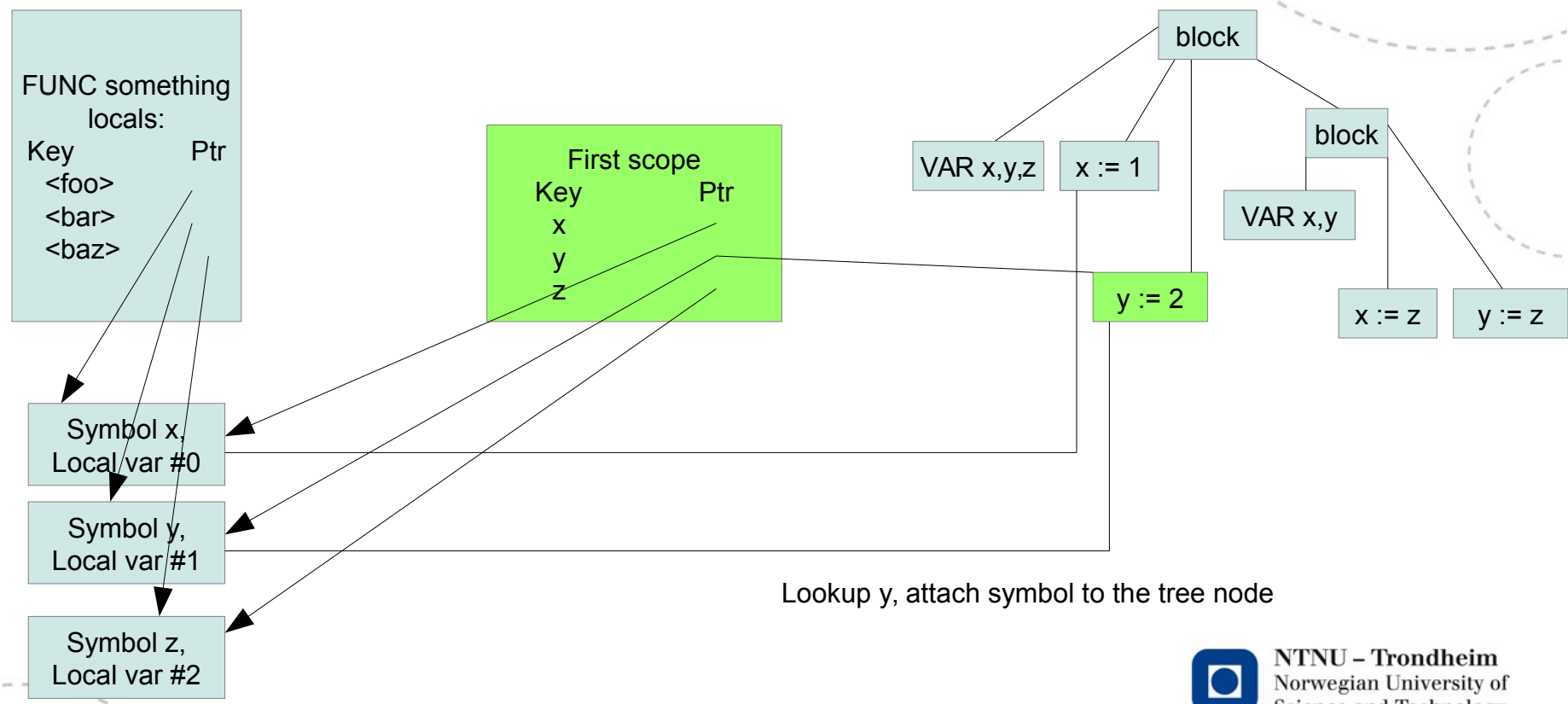




Avoiding name clashes among local variables

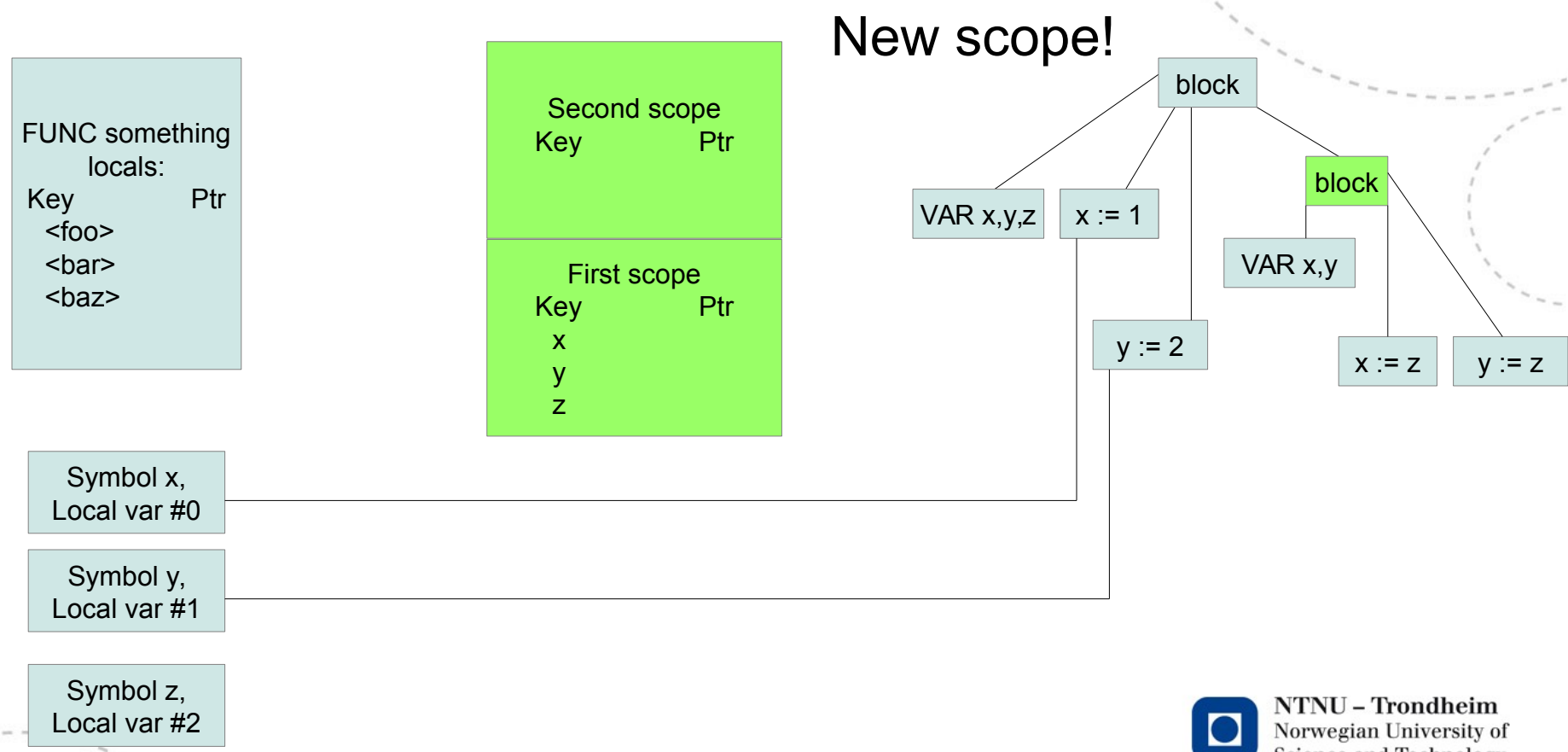


Avoiding name clashes among local variables

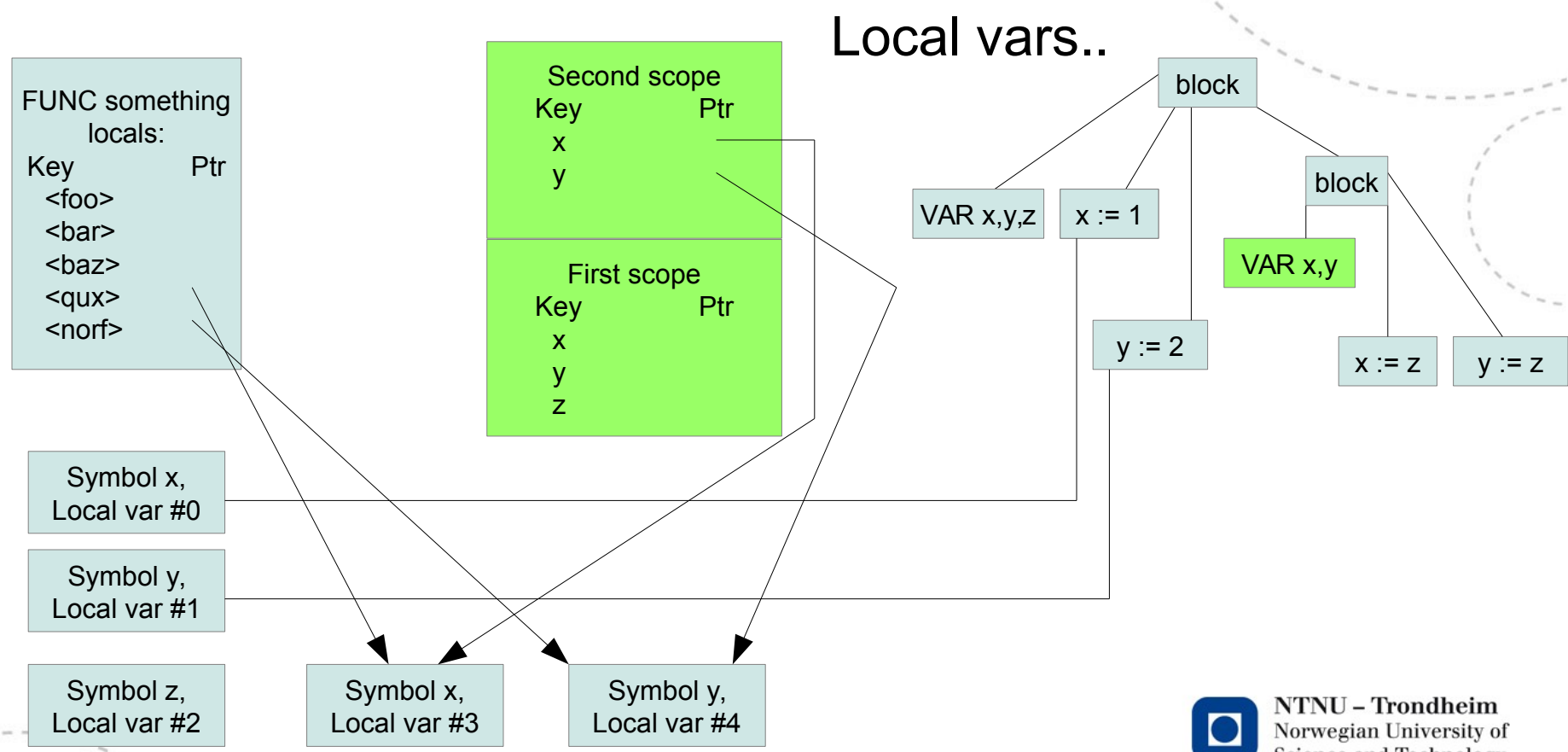


Lookup y, attach symbol to the tree node

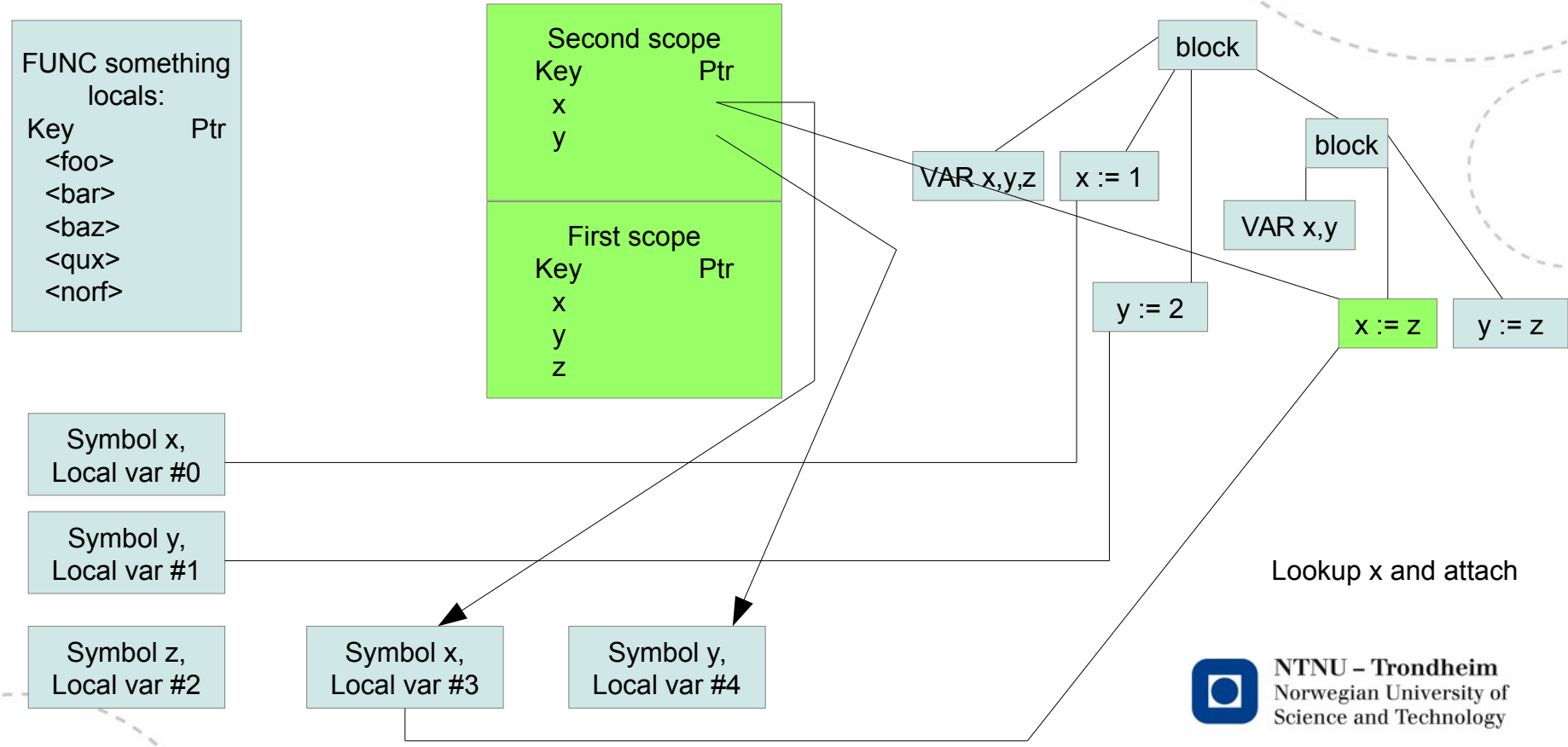
Avoiding name clashes among local variables



Avoiding name clashes among local variables

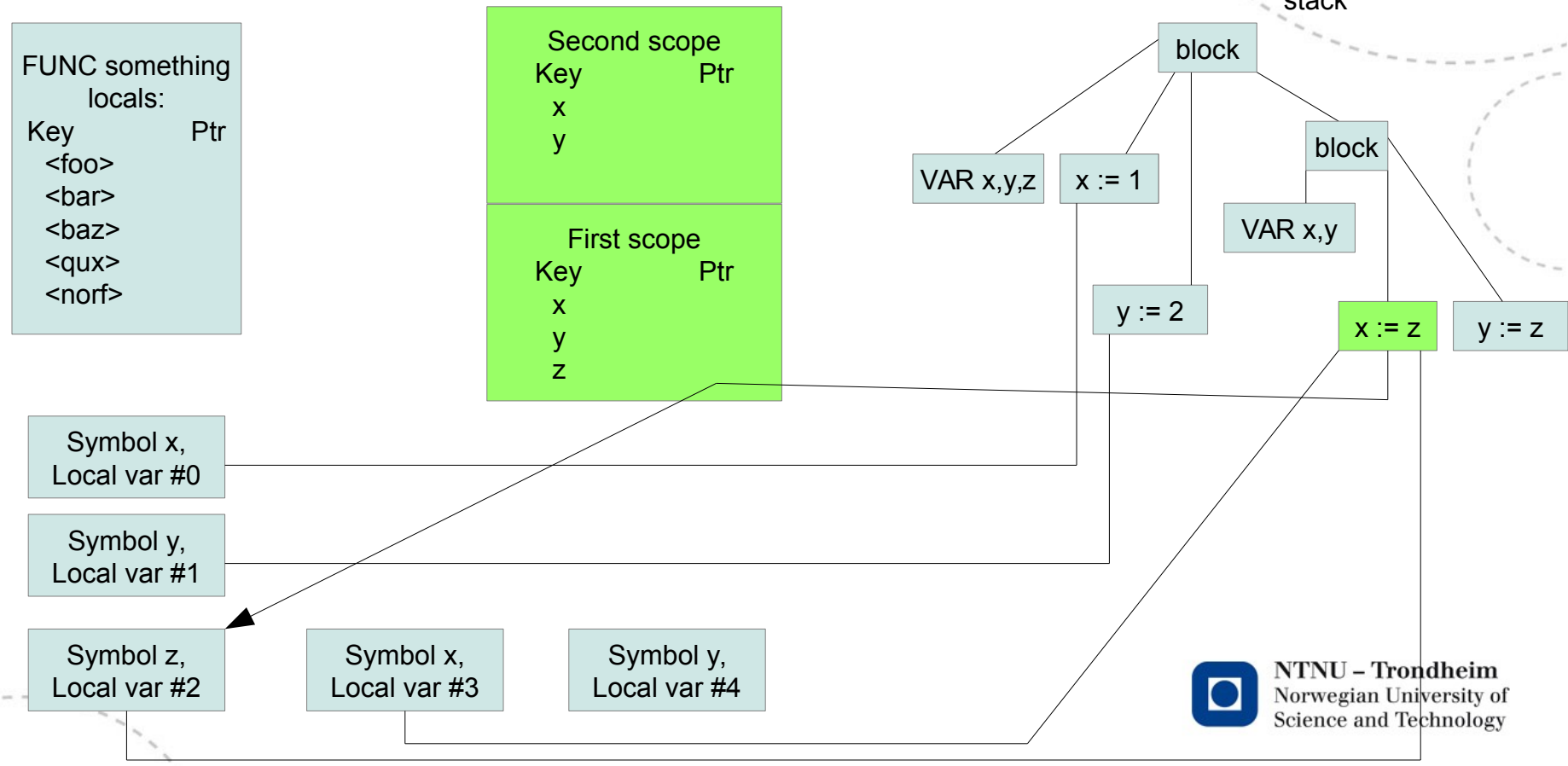


Avoiding name clashes among local variables



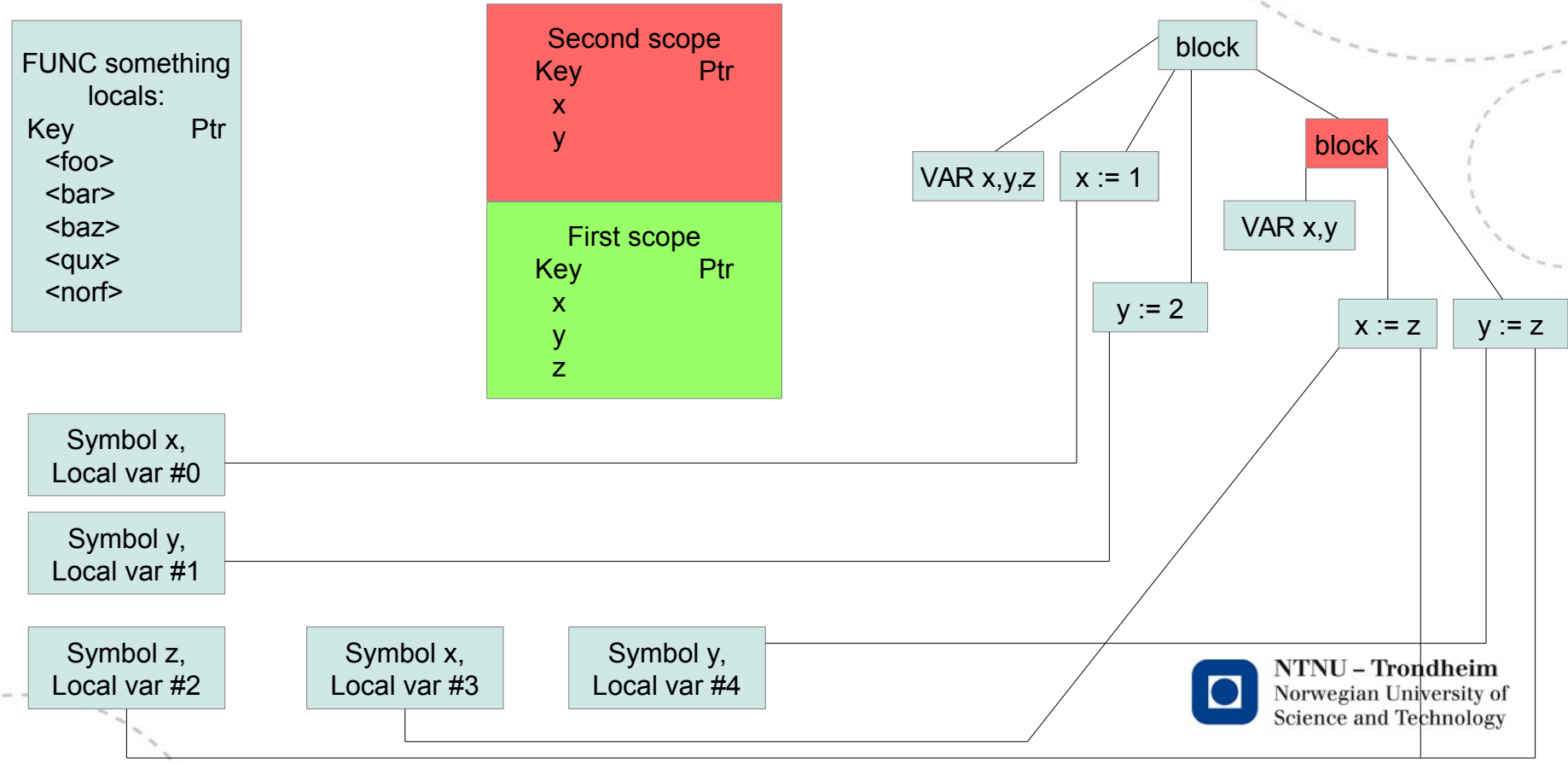
Avoiding name clashes among local variables

Lookup z and attach.
z isn't in inner scope,
must search down the
stack



Avoiding name clashes among local variables

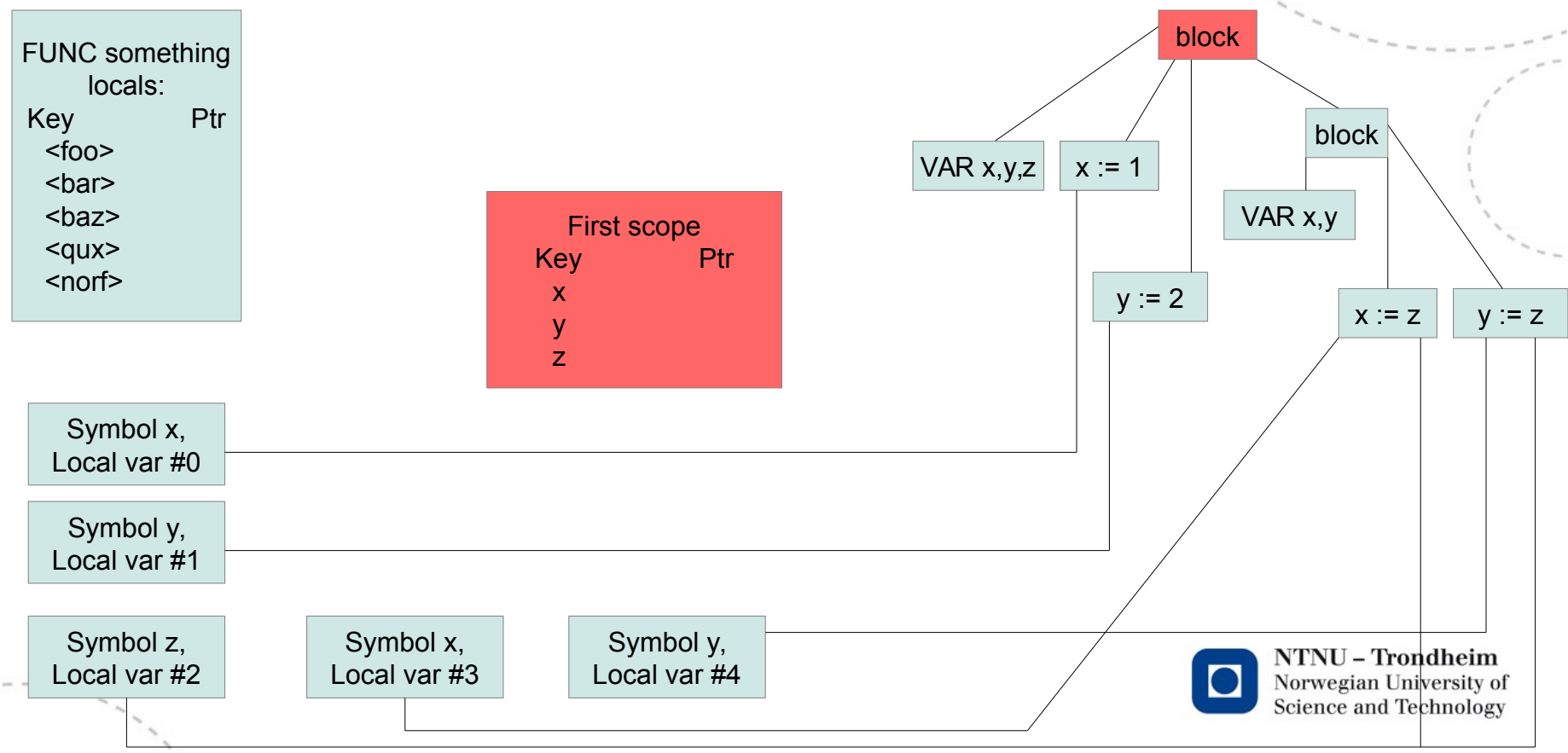
When block is finished, remove temporary scope table from top of stack



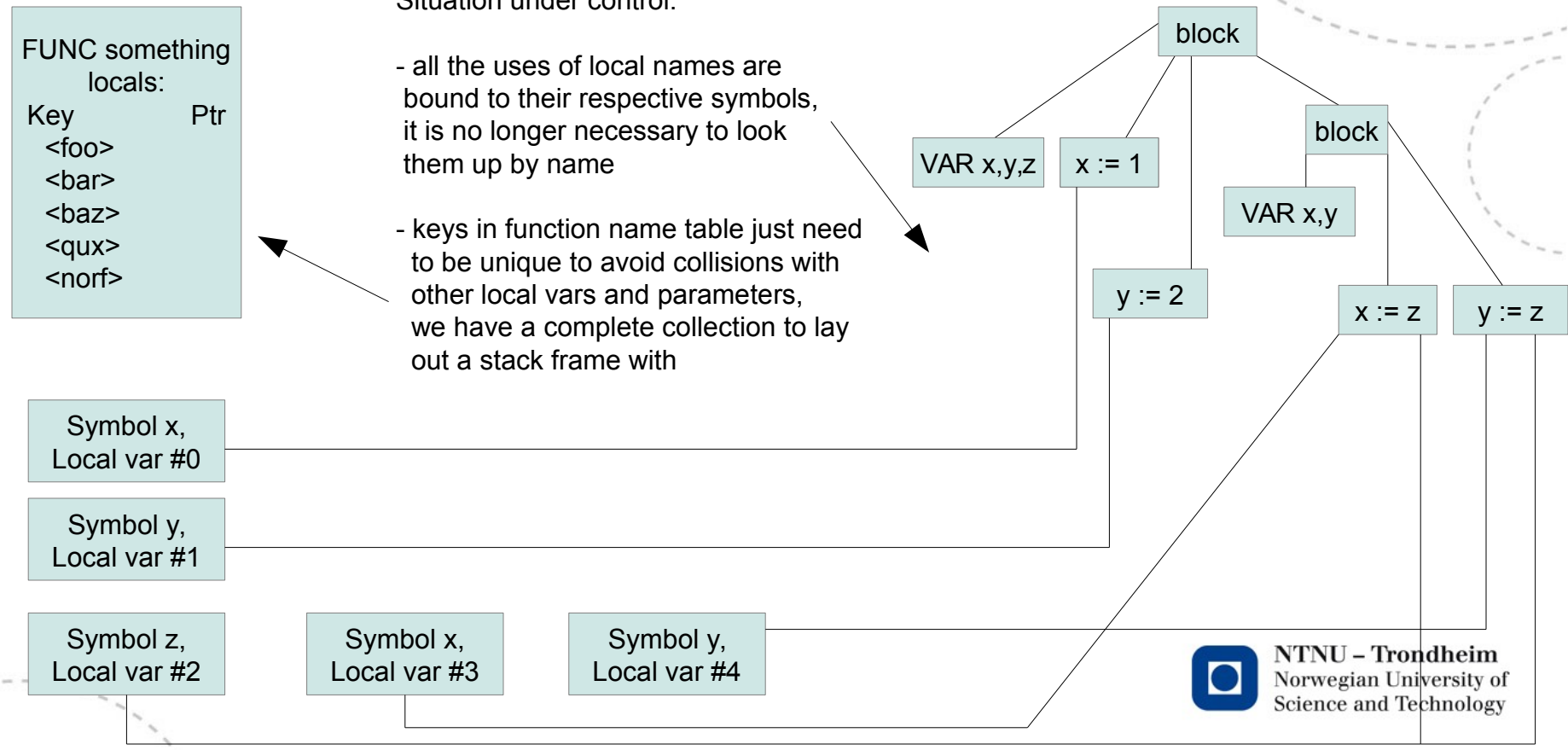
NTNU – Trondheim
Norwegian University of Science and Technology

Avoiding name clashes among local variables

When block is finished, remove temporary scope table from top of stack



Avoiding name clashes among local variables



NTNU – Trondheim
Norwegian University of Science and Technology

(Another way to do it)

- The function symtab entry only really needs to know how many locals to make space for, you can also just count them, and add a field to the symbol_t struct
- That leaves you with the tree nodes as the only path to their symbol_t structs, and several links to each, so it gets messier to remove everything
  - You can also keep a collected list of pointers to locals...
  - ...or keep a list that tracks every symbol you see, and work it out from there…
  - *Etc. etc.*

About the cleanup

- Due to the wonders of virtual memory, the program would work even if it leaked all its memory
- If you meticulously instrument it, you may also find that the generated code from Lex/Yacc leaks (a little)
- I can't bring myself to say that tidy memory management makes no difference, so there's a payoff for giving it a shot
  - Calibrate your effort against the reward, though, perfecting this can take a lot of time depending on how you juggle your pointers, and it is not the main focus of the exercise
  - Non-perfect efforts are better than non-efforts

Semantic errors

- Looking up names, we can now tell whether they were properly declared or not
- It can be helpful to put in an error message or two if you like to test using your own programs
- What to do with incorrect programs isn't *specified*, it is enough work to compile correct ones
  - Whether your compiler exits gracefully or crashes and burns on an incorrect program is up to you

The latest text dump

- print_symbols and print_bindings are already written, 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 text dump looks a little different from the ones I've supplied as guideline
  - Particularly, if you hash differently, elements might come out sorted in different orders, I have not taken the trouble to sort them by sequence numbers

However:

- Up to the order things appear in, the indices of functions, parameters, local variables should match
- 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 chapter, we will use them to calculate addresses in machine-level code
- Please don't invent alternative numbering schemes

By the way

- Putting everything in the already defined routines can make them large an unwieldy to work with
- My own ir.c is littered with various subroutines to handle various cases, just to keep things apart and think of one problem at a time
- You can do that if you like

Scores and all that jazz

- Full marks for all the right numbers
  - Output differences tolerated up to ordering, but the numbering should be correct
  - If you can see where this is going, it is technically *possible* to invent another scheme, but what we're doing roughly emulates a common convention, so that will be fine. Thanks.
- Partial marks for an attempt
  - Depending on how close it comes to working
- Zero is for nothing-at-all
  - I would encourage everyone to try everything

How much does it count, *really*?

- 1/100 out of PS4 is 1/1000 out of TDT4205
- You can, in fact, not numerically exclude yourself from the possibility of an A with this thing alone
  - Don't make an experiment out of it, though, it's not that easy to do *everything* else to perfection
- I can not remember seeing any final mark decided by the finer details of one sub-task
- Don't panic just because there are points

On the other hand

- This one *can* actually be a little bit tricky
- Starting with one evening to go, I would probably struggle to finish it in style
- While I am no super-programmer, some of you surely are
- To be on the safe side, I still suggest that you start super-programming sooner rather than later
  - It never hurts to be done early