Compiler Construction

Problem Statement 4
Guideline Slides

NTNU

Norwegian University of Science and Technology

Symbol Tables

— The task 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

— The hash table in the standard library is not really usable so a separate hash table implementation has been provided for this exercise.

— This is not a high performance solution but for this sake of this exercise this is adequate.

Using the 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 caller program to take care of.
- There is the symbol_t struct which should be used for this

symbol_t struct

The struct which you have to make use of for the symbol table located in ir.

```
typedef struct s {
  char *name;          <----(1)
  symtype\_t type;     <----(2)
  node\_t *node;       <----(3)
  size\_t seq;         <----(4)
  size\_t nparms;      <----(5)
  tlhash\_t *locals;   <----(6)
}
```

symbol_t struct

The struct which you have to make use of for the symbol table located in ir.

- (1)$\rightarrow$ Will be Text(name related to the particular symbol)
- (2)$\rightarrow$ Will be the enumeration (the type i.e. whether it is a function or a global or a local variable or whether it is a parameter
- (3)$\rightarrow$ root node (of type function)
- (4)$\rightarrow$ Sequencing number (for everything but global variables)
- (5)$\rightarrow$ Parameter count (for functions)
- (6)$\rightarrow$ Hash table of local names

What to do

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

What to do

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
- Create a global index of string literals

What to do

Thing #3 to do

- Destroy the whole structure that you have created i.e. to implement the destroy symtab tree.
- This depends on your implementation

A global index of string literals

— Strings are only used onece and that happens in the node that represents them

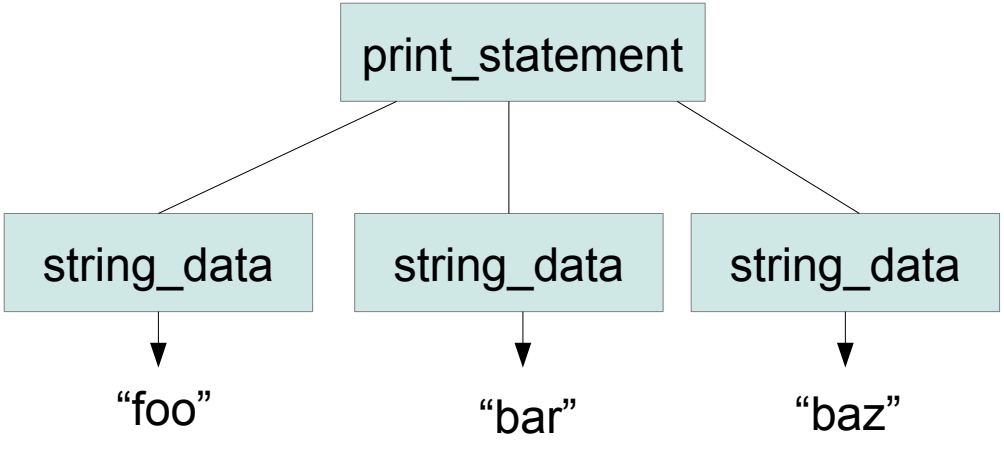
— The node presently contains a pointer to the string at the data element.

— When the time comes to generate code, it would be nice to display all the strings at once

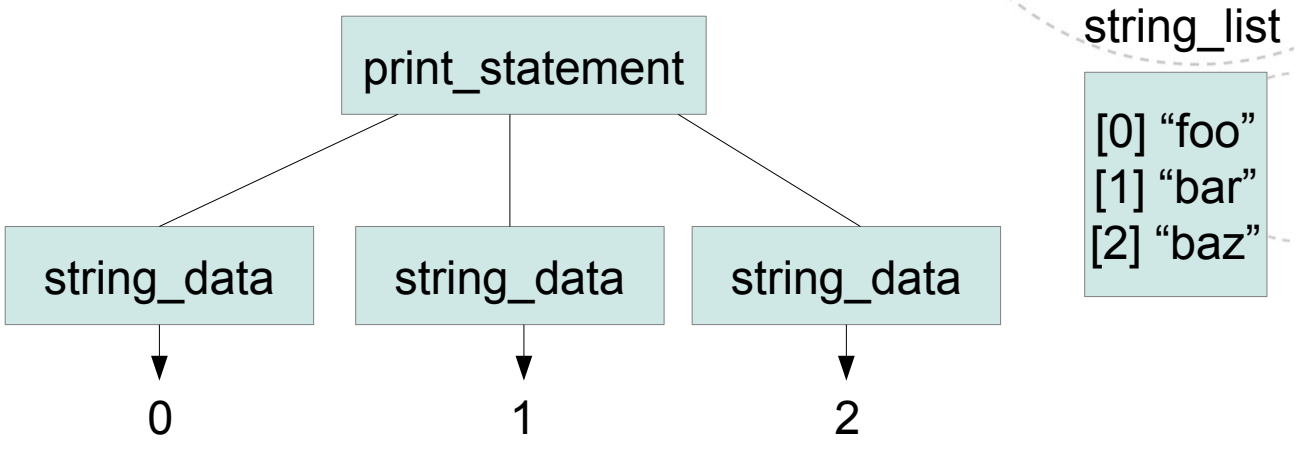
— Therefore:

- 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

For example:


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

Becomes:



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

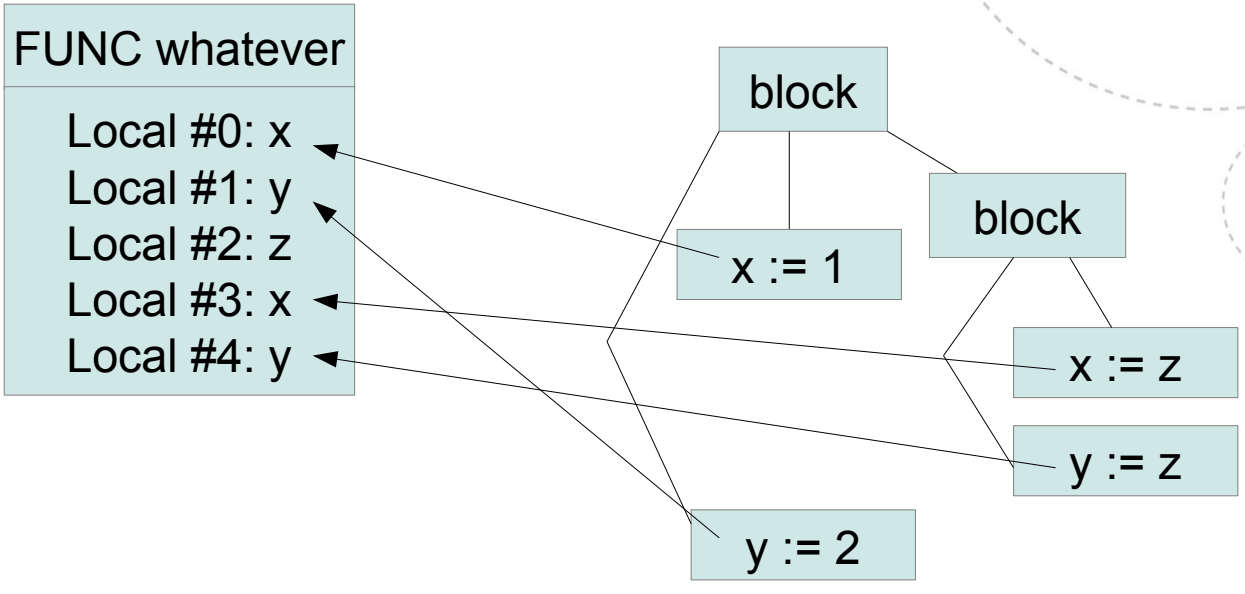
Local name tables

- 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

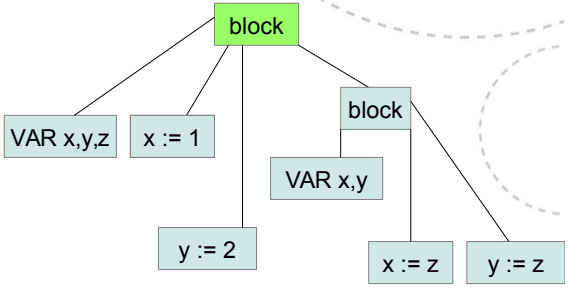
In other words



Avoiding name clashes among local variables

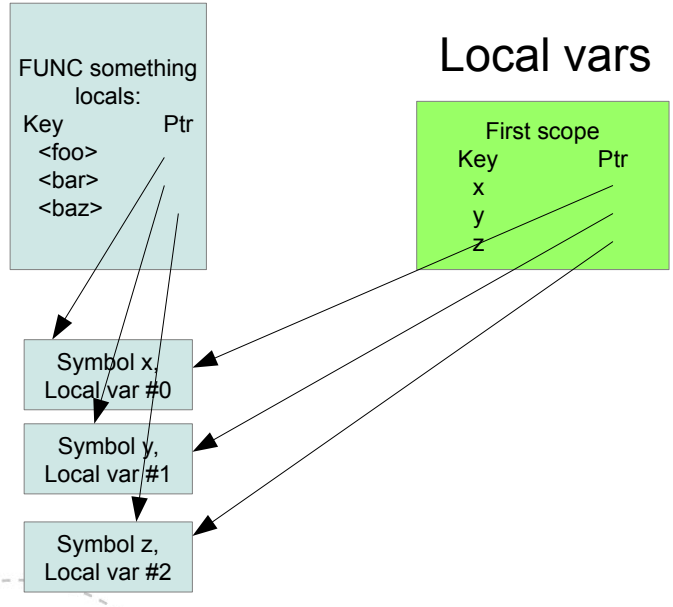
FUNC something
locals:
Key Ptr

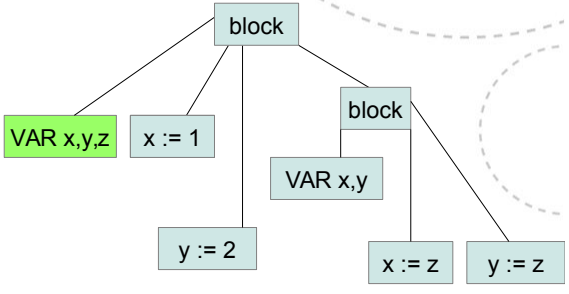
New scope!



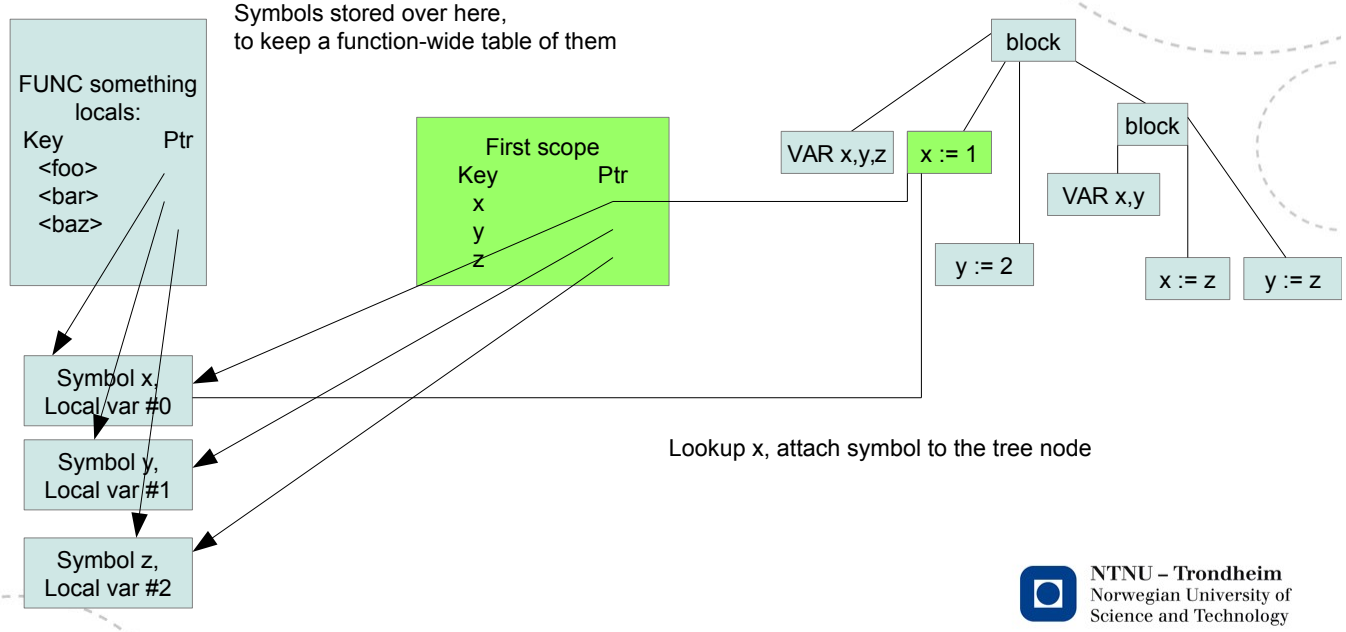
First scope
Key Ptr

Avoiding name clashes among local variables

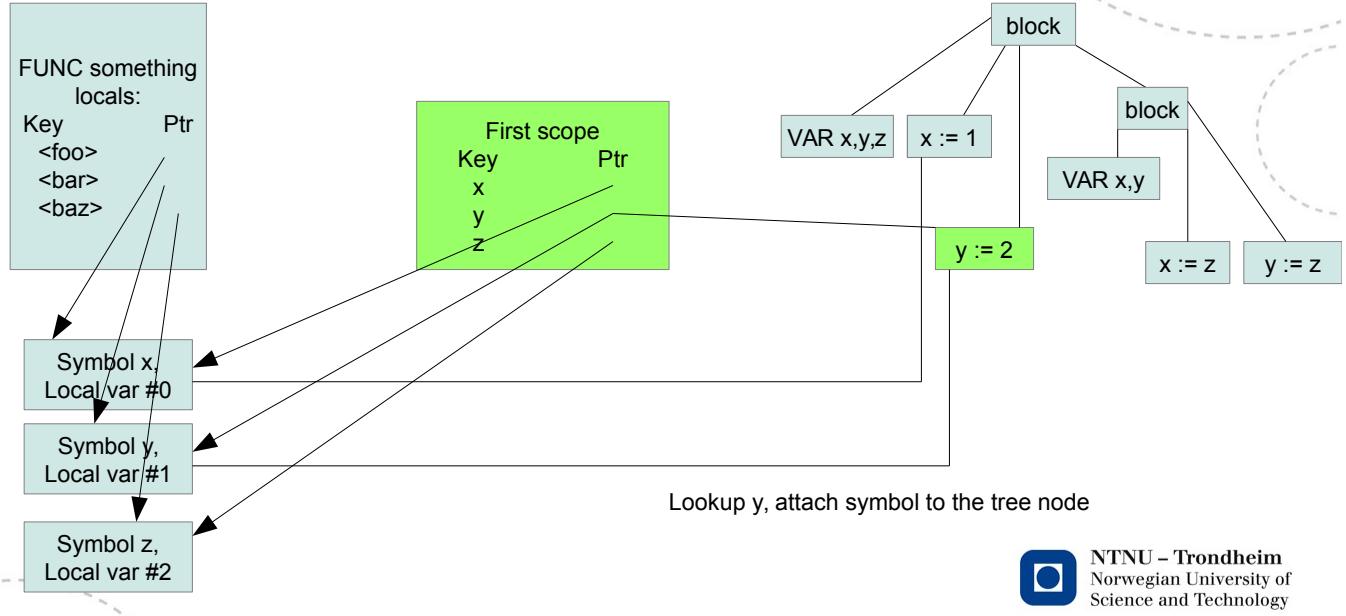




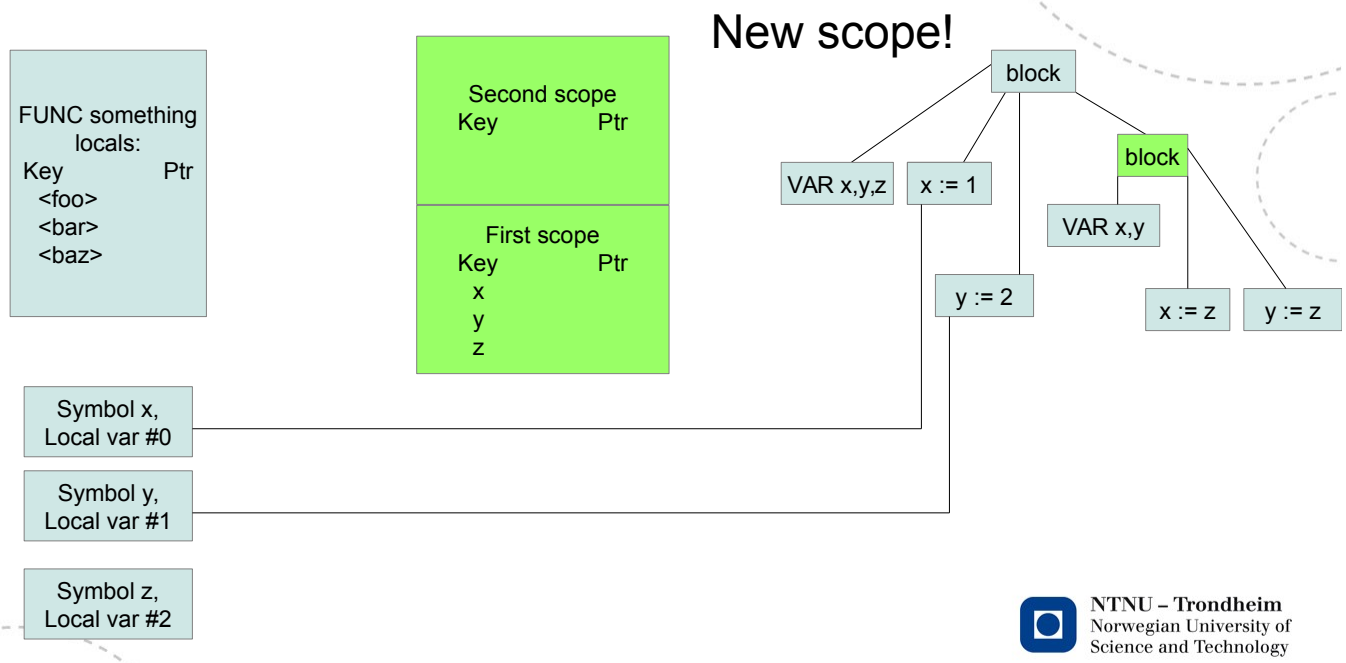
Avoiding name clashes among local variables



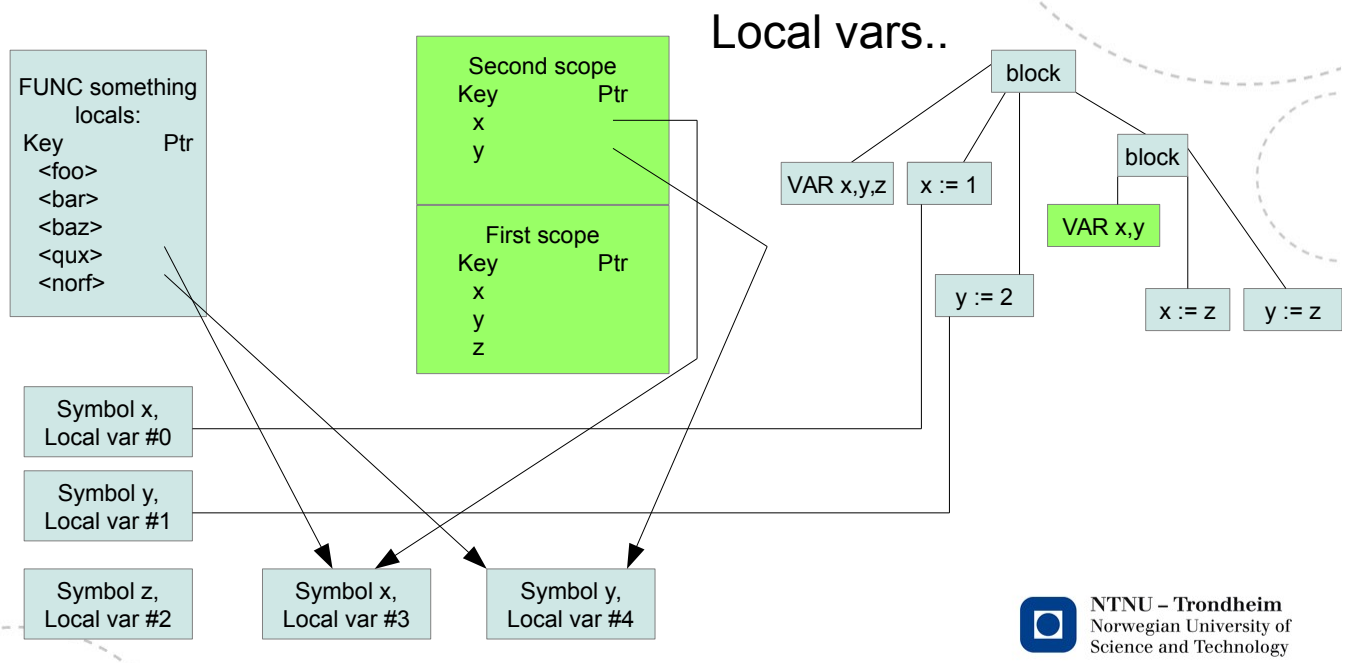
Avoiding name clashes among local variables



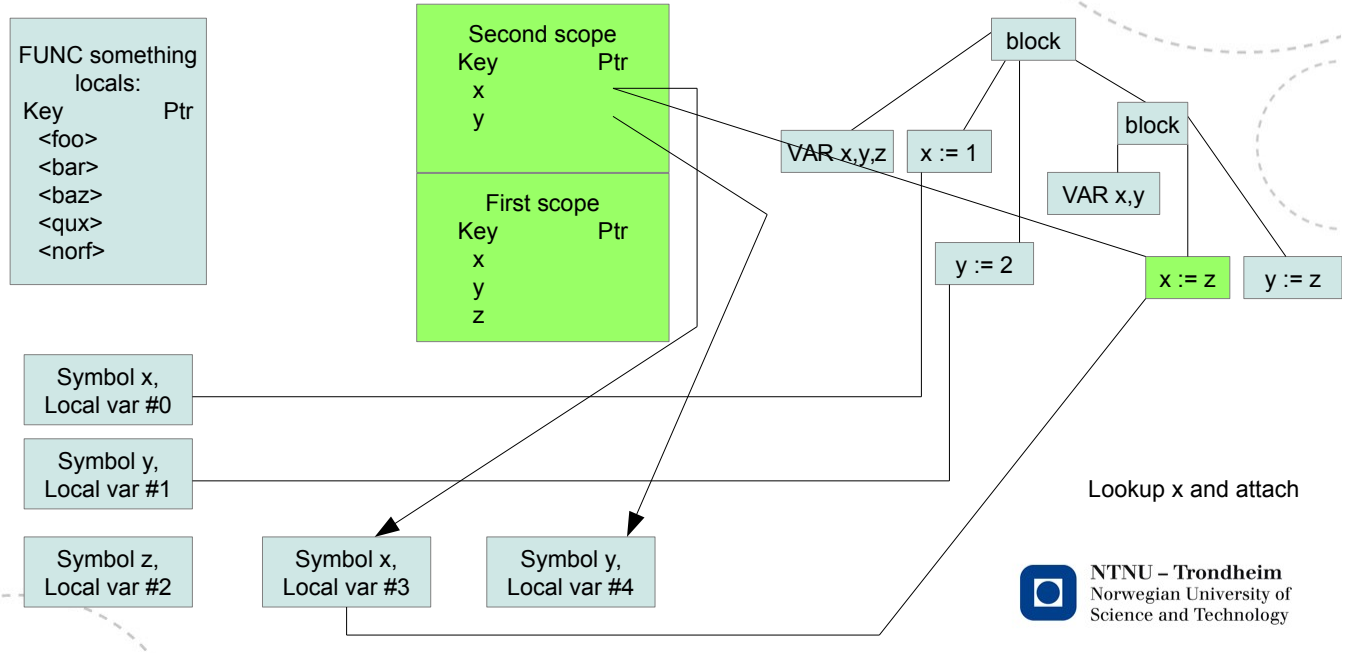
Avoiding name clashes among local variables



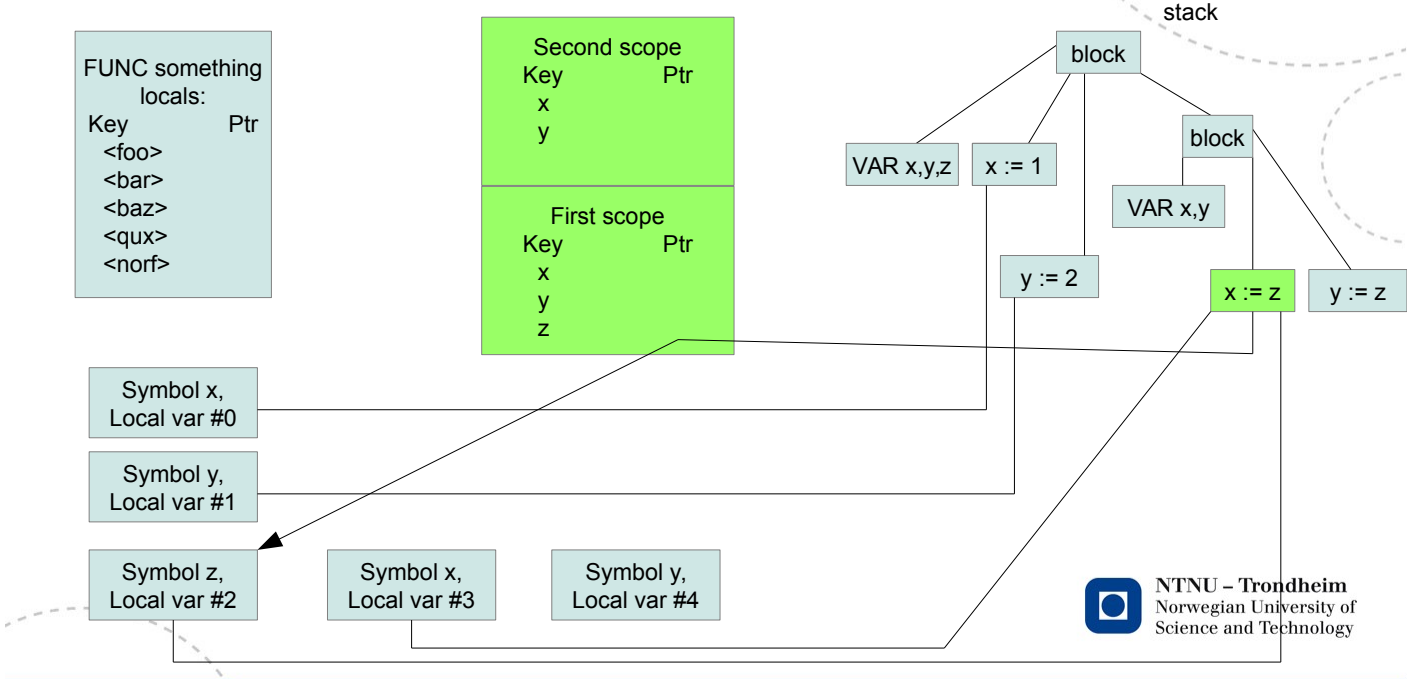
Avoiding name clashes among local variables



Avoiding name clashes among local variables

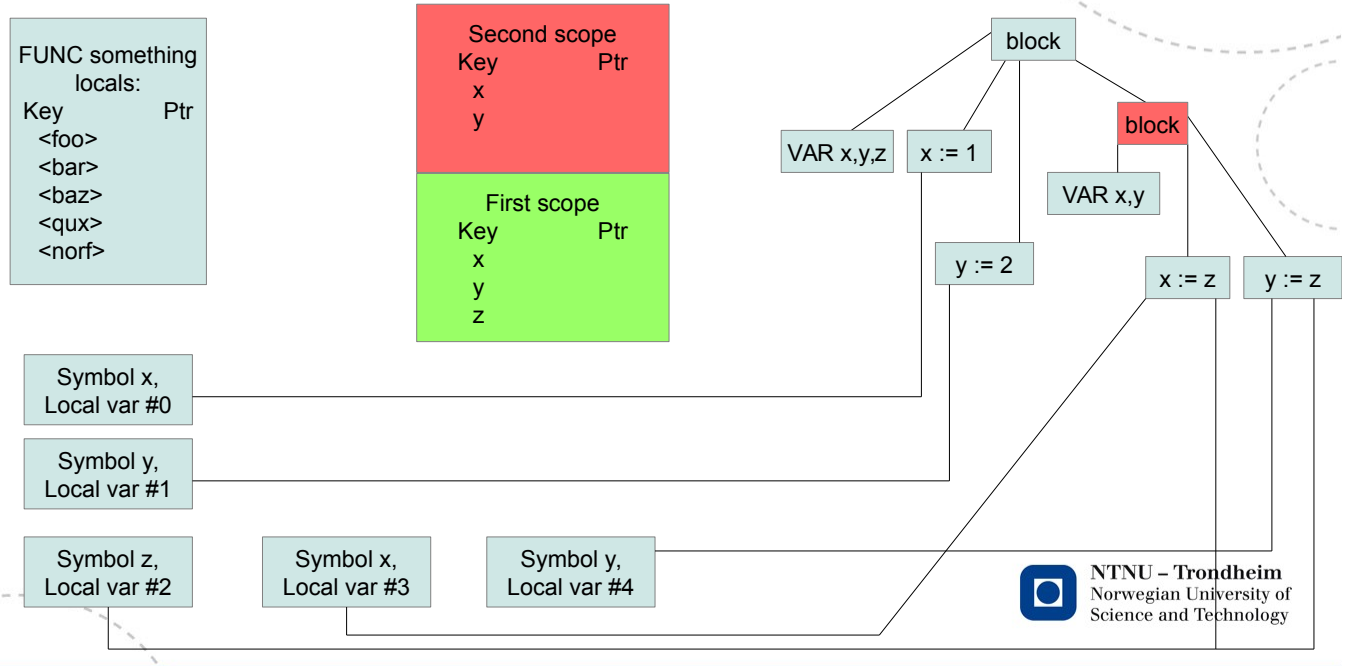


Avoiding name clashes among local variables



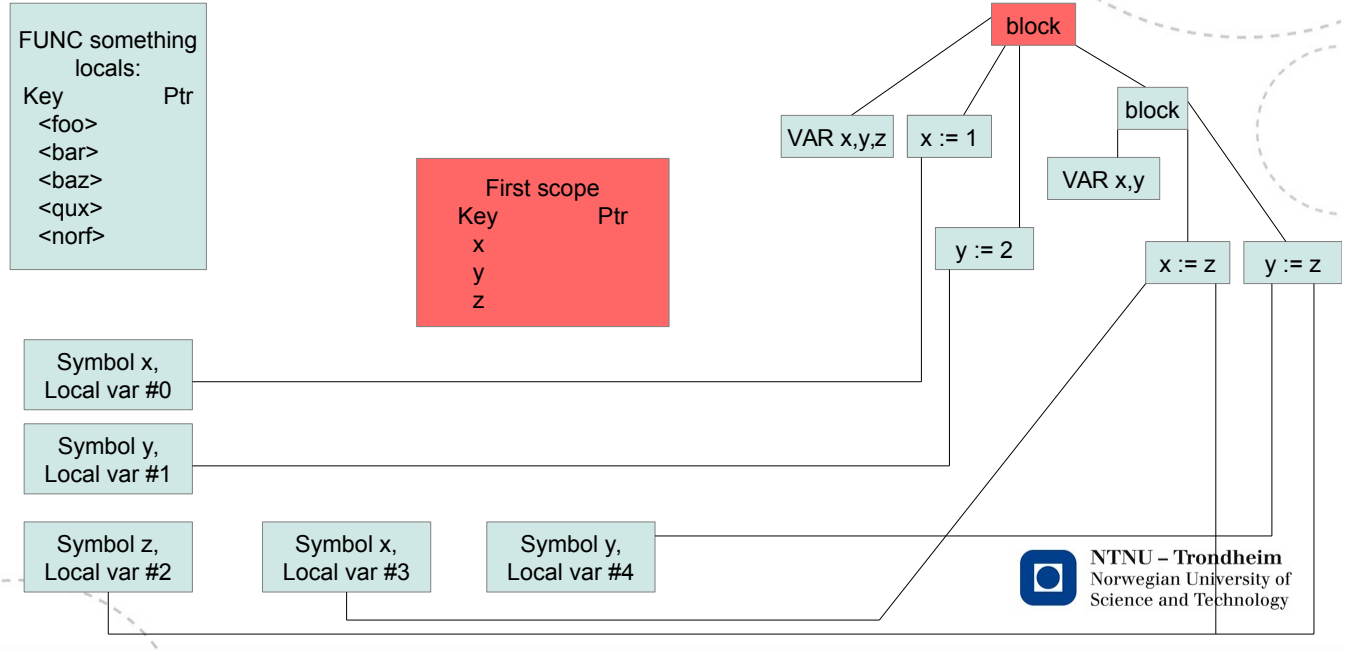
Avoiding name clashes among local variables

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

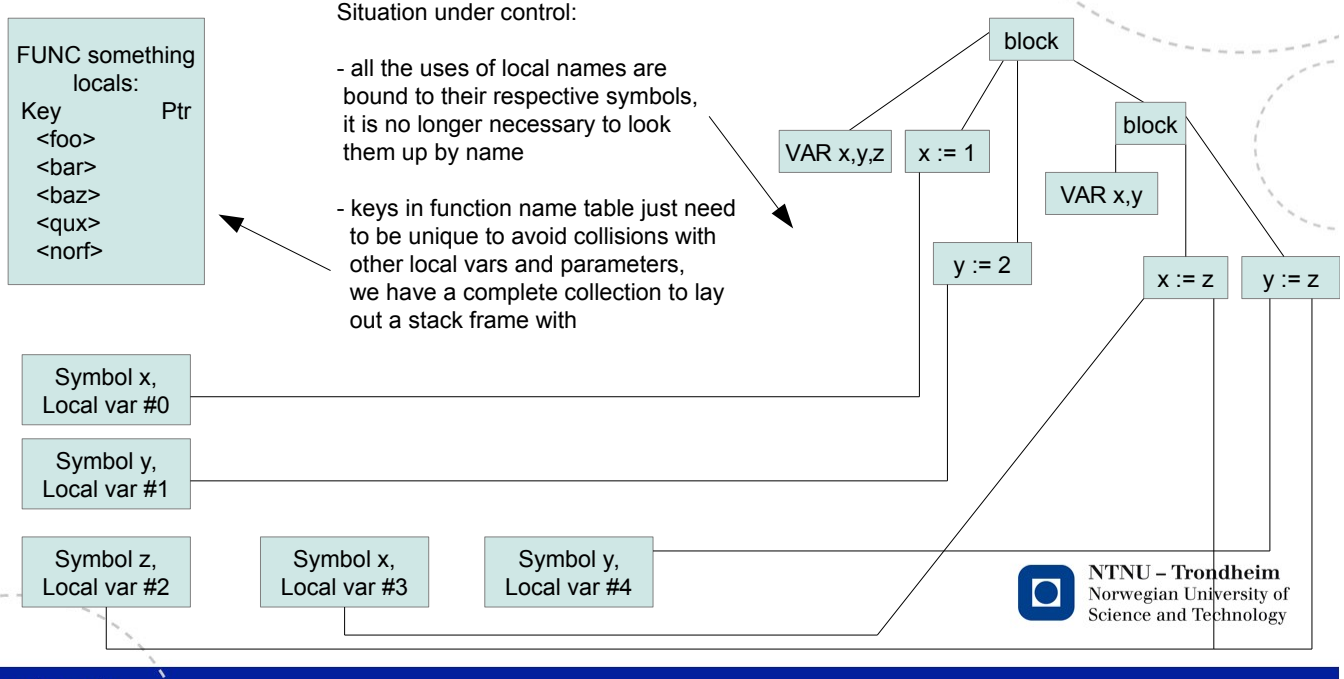


Avoiding name clashes among local variables

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



Avoiding name clashes among local variables



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

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

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