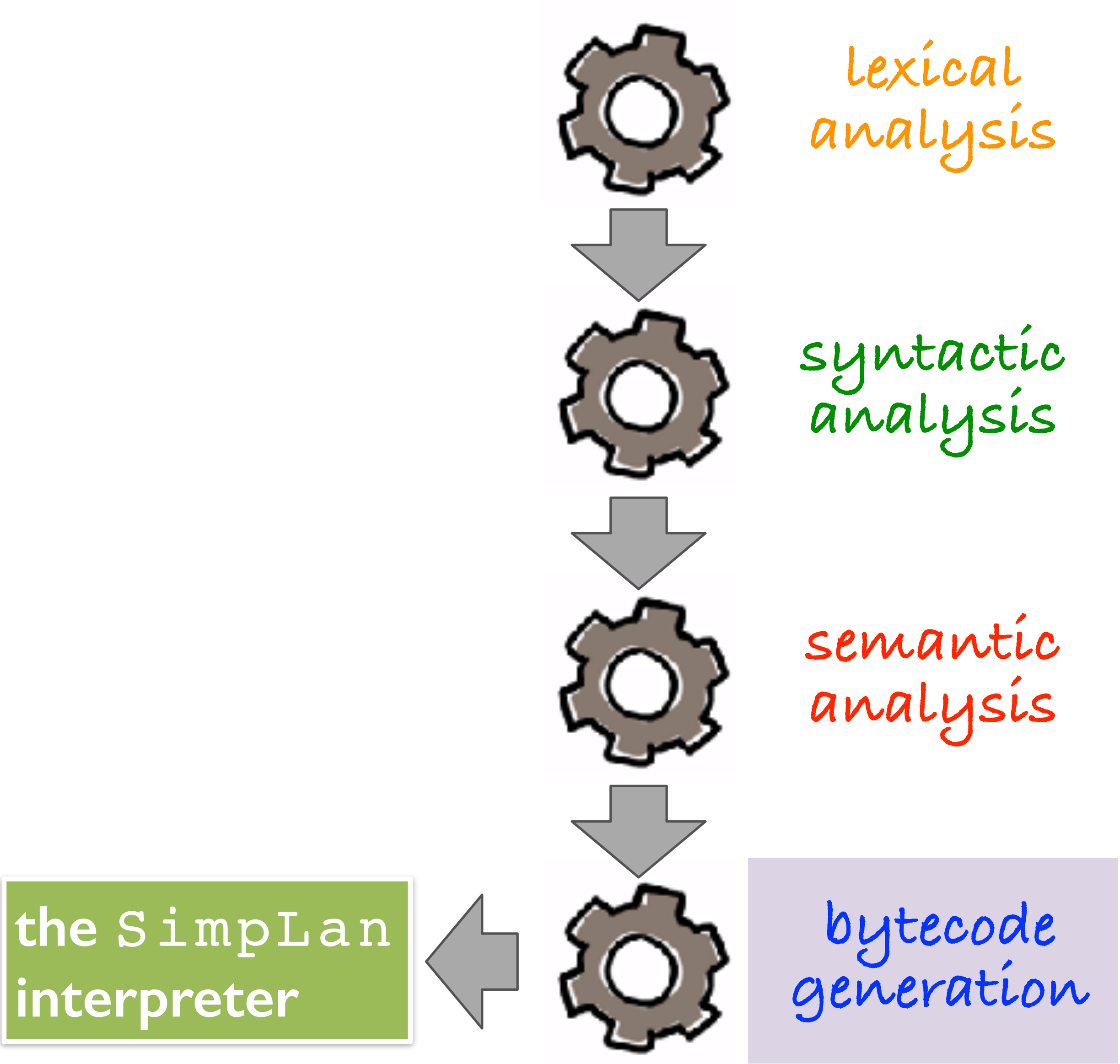
ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA
DIPARTIMENTO DI
INFORMATICA - SCIENZA E INGEGNERIA

CODE GENERATION

COSIMO LANEVE
`cosimo.laneve@unibo.it`

CORSO 72671
COMPLEMENTI DI LINGUAGGI DI PROGRAMMAZIONE

THIS LECTURE



lexical analysis

syntactic analysis

semantic analysis

bytecode generation

the `SimpLan` interpreter

OUTLINE

1. management of run-time resources
2. correspondence between static (compile-time) and dynamic (run-time) structures
3. storage organisation (in particular, memory management)
4. code generation (for stack machines)
5. the assembly language
6. a simple source language (`SimpLan`) and its stack-machine implementation

reference:

- Torben Morgensen: **Basics of Compiler Design**, Chapter 7

RUN-TIME ENVIRONMENTS

before discussing code generation, we need to understand what we are trying to generate

- there are a number of standard techniques for structuring executable code that are widely used

remark: the execution of a program is **initially under the control of the operating system** — when a program is invoked:

1. the OS allocates space for the program
2. the code is loaded into part of the space
3. the OS jumps to the entry point (i.e., "`main`")

MEMORY LAYOUT

traditionally, pictures of machine organization have areas for different kinds of data:

- delimitated by lines

these pictures are **simplifications**

- not all memory needs to be contiguous

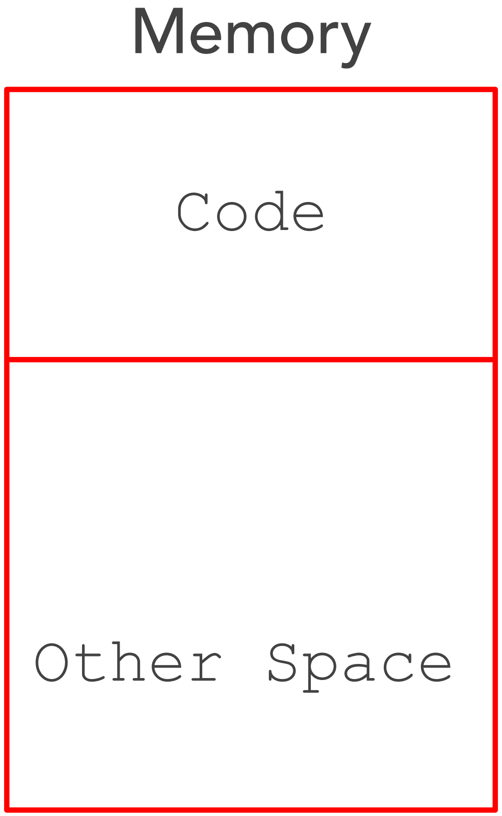
what is "other space"?

- holds all data for the program

`Other Space = data space`

the compiler is responsible for:

- **generating** code
- **managing** the use of the data space



CODE GENERATION GOALS AND ASSUMPTIONS

two goals:

1. *correctness*
2. *speed*

most complications in code generation come from trying to be fast as well as correct

assumptions:

1. **execution is sequential**; control moves from one point in a program to another in a well-defined order
2. when a procedure is called, **control eventually returns to the point immediately after the call**

ACTIVATIONS AND LIFETIME OF THE VARIABLES

an invocation of function `f` is an activation of `f`

the **lifetime of an activation** of `f` is

- all the steps to execute `f`
- including all the steps in `f` calls

the **lifetime of a variable** `x` is the **portion of execution** in which `x` is defined

note that

- lifetime is a **dynamic (run-time) concept**
- **scope** is a **static concept**

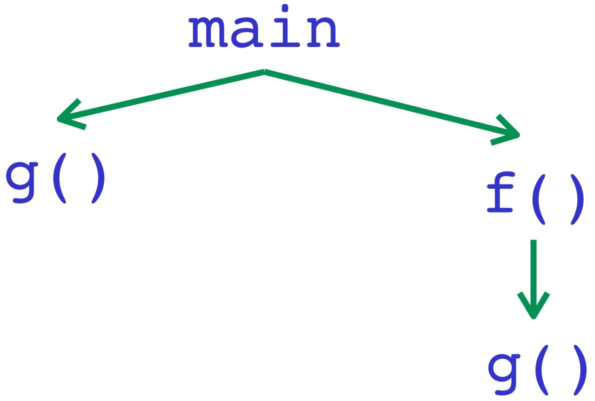
ACTIVATION TREES

trees that indicate the activation lifetime of functions

example:

```
int g() { return 1 ; }
int f() { return g() ; }
void main() {
           g() ; f() ;
}
```

the activation tree



assumption (2) requires that when `f` calls `g`, then `g` returns before `f` continues

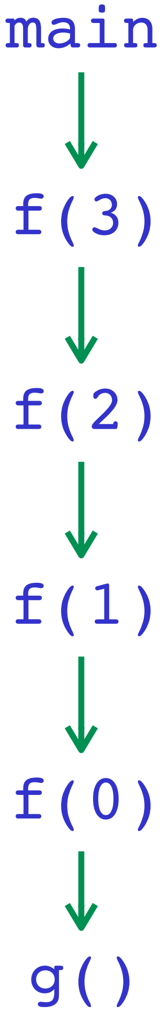
- lifetimes of procedure activations are properly nested
- the **activation tree**
  - invocations in sequence are depicted as **brother nodes**
  - nested invocations are depicted as **father-son nodes**

EXAMPLE

compute the activation tree for

```
int g() { return 1; }
int f(int x) {
    if (x == 0) { return g(); }
    else { return f(x - 1); }
}
void main() { f(3); }
```

the activation tree



main

f(3)

f(2)

f(1)

f(0)

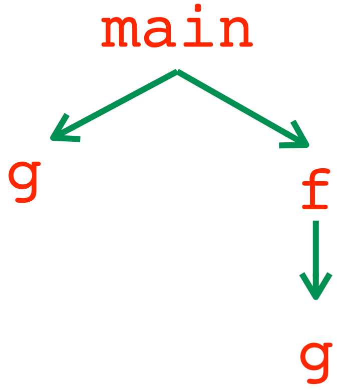
g()

REMARKS

- the activation tree **depends on the run-time behaviour**
- the activation tree **may be different for every program input**
- since activations are nested, a **stack** can track currently active procedures

```
int g() { return 1; }
int f() { return g(); }
void main() {
    g();
    f();
}
```

Activation Tree



Stack

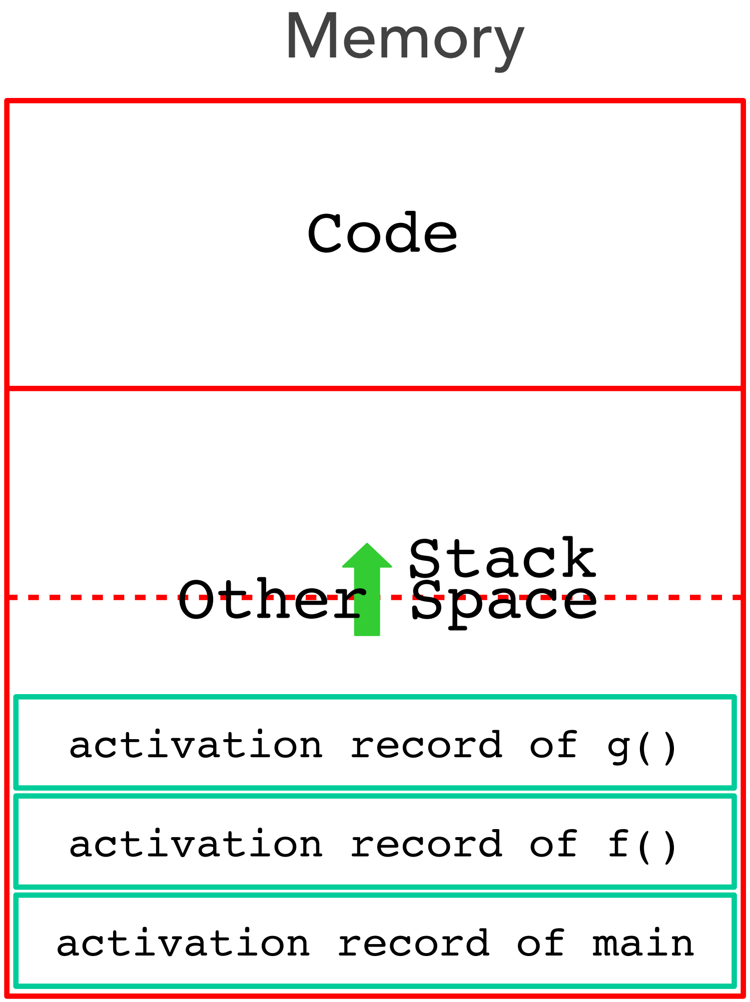
main

main g

main f

main f g

REVISED MEMORY LAYOUT



the information needed to manage one function activation is called an **activation record** (AR) or **frame**

- the program starts with the AR of `main`
- then `main` invokes `f`
- then `f` calls `g`
- **remark**: `g`'s activation record contains a mix of info about `f` and `g`

this is a **stack** of activation records

- the AR have different dimensions
- the dimension depends on the formal parameters and on the local variables
- you need to implement the stack by means of pointers

WHAT IS IN `G`'S AR WHEN `F` CALLS `G`?

- `f` is "**suspended**" until `g` completes, when this happens, `f` resumes
- `g`'s AR contains information needed to resume execution of `f`
- `g`'s AR may also contain:
  - `g`'s return value (needed by `f`)
  - actual parameters to `g` (supplied by `f`)
  - space for `g`'s local variables

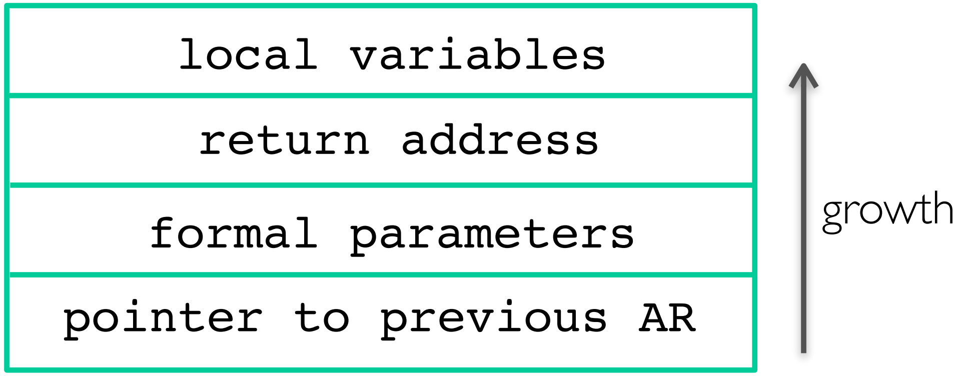
THE CONTENTS OF A TYPICAL AR FOR g

1. space for g's **return value**
2. **actual parameters**
3. **pointer to the previous activation record**
    - the **control link** — points to AR of caller of g
4. **machine status** prior to calling g
    - contents of **registers & program counter**
5. **local variables**
6. other **temporary values**

THE EXERCISE, REVISITED

```
int g() { return 1; }
int f(int x) {
     int y = 1 ;
     if (x == 0) { return g(); }
     else { return f(x - y); (**) }
}
void main() { f(3); (*) }
```

AR for `f`:



local variables

return address

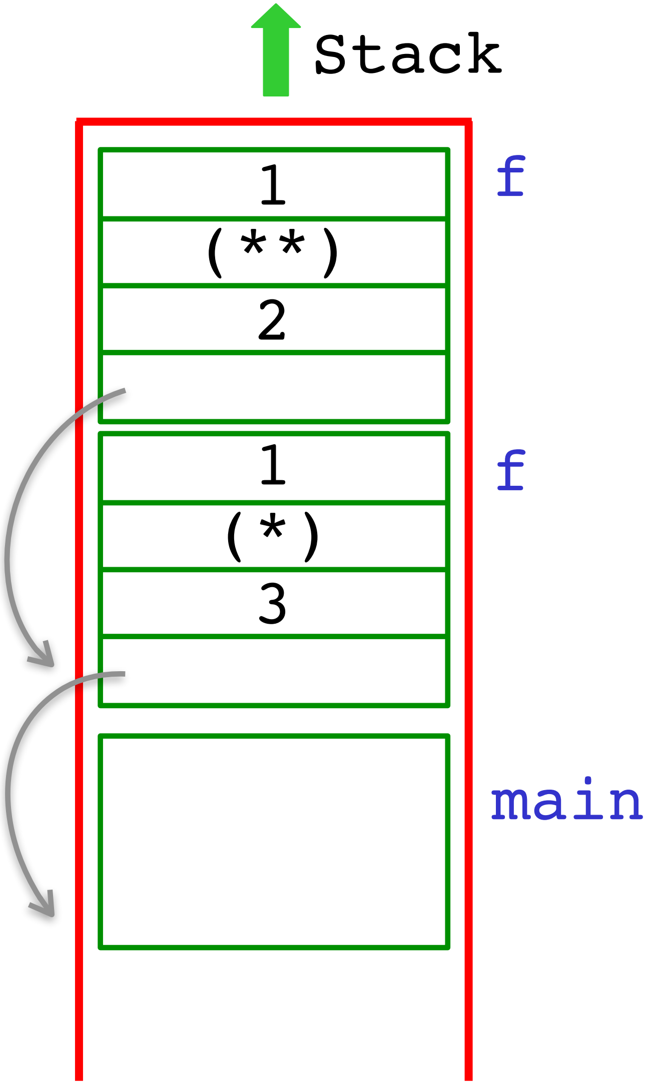
formal parameters

pointer to previous AR

remarks

- this is a possible structure of AR
- there is no space for the **return value** because it will be stored in a register (in our machine)
- otherwise it is stored below the "pointer to previous AR". WHY?

STACK AFTER TWO CALLS TO `f`



- `main` has no argument or local variables and its result is never used; **its AR is uninteresting**
- `(*)` and `(**)` are **return addresses** of the invocations of `f`
- this is only one of many possible AR designs
  - it works for many programming languages

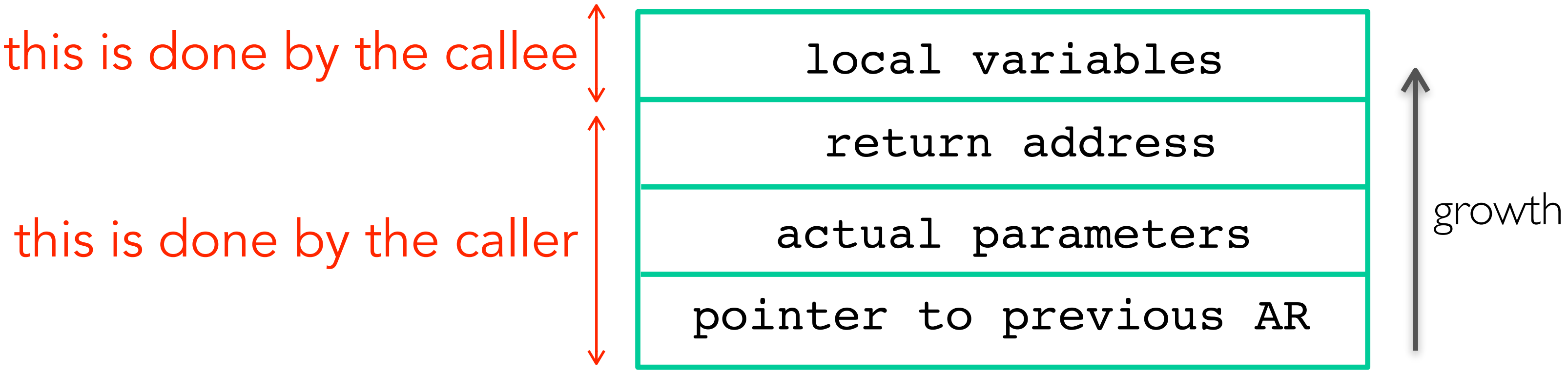
THE MAIN POINT

the compiler must **determine**, at compile-time, the **layout** of activation records and **generate code** that correctly accesses locations in the activation record

therefore **the AR layout and the code generator must be designed together!**

WHO BUILDS THE ACTIVATION RECORDS?

both the caller and the callee



this is done by the callee

this is done by the caller

local variables

return address

actual parameters

pointer to previous AR

growth

GLOBAL VARIABLES

all references to a global variable point to the same element

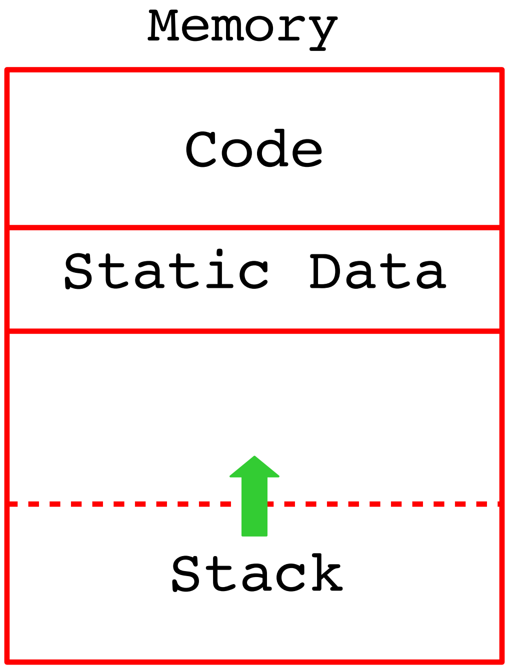
- **it is wrong** to store a global variable in an AR

global variables are assigned a **fixed address** once

- variables with fixed address are "**statically allocated**"

depending on the language, there may be other statically allocated values

memory layout with static data:



VARIABLES DECLARED IN OUTER SCOPES

- references to a **variable declared in an outer scope**
  - should point to a variable stored in **another activation record**
- **to which activation record** ?
  - according to the **most closely nested** rule, an activation record should point to the most recent activation record of its immediately enclosing scope
- use **access links** …

ACTIVATION RECORDS WITH ACCESS LINKS

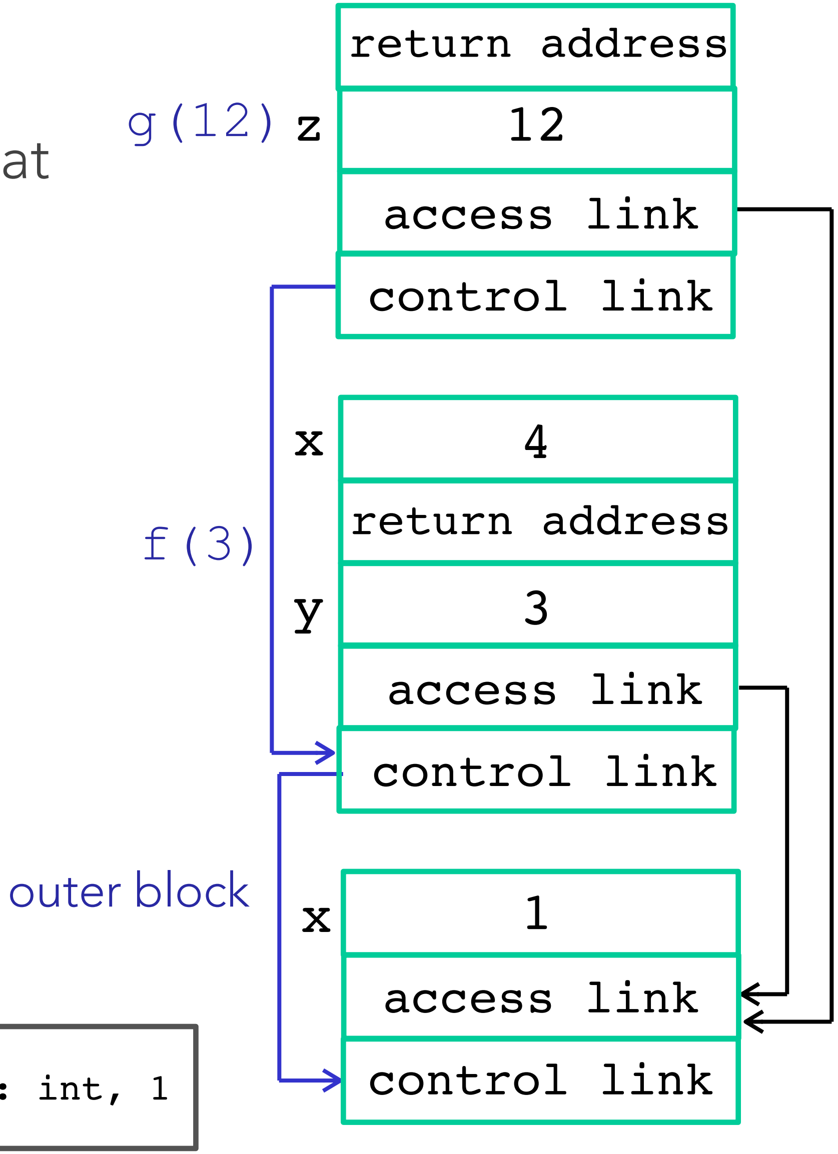
the access links are set to the **AR of the enclosing syntactical block**

- for function body, this is the block that contains the function declaration

```
int x = 1;
{
   int g(int z){ return x+z; }
   int f(int y){ int x = y+1;
                 return g(y*x); }
   f(3);
   . . .
}
```

the symbol table at this point

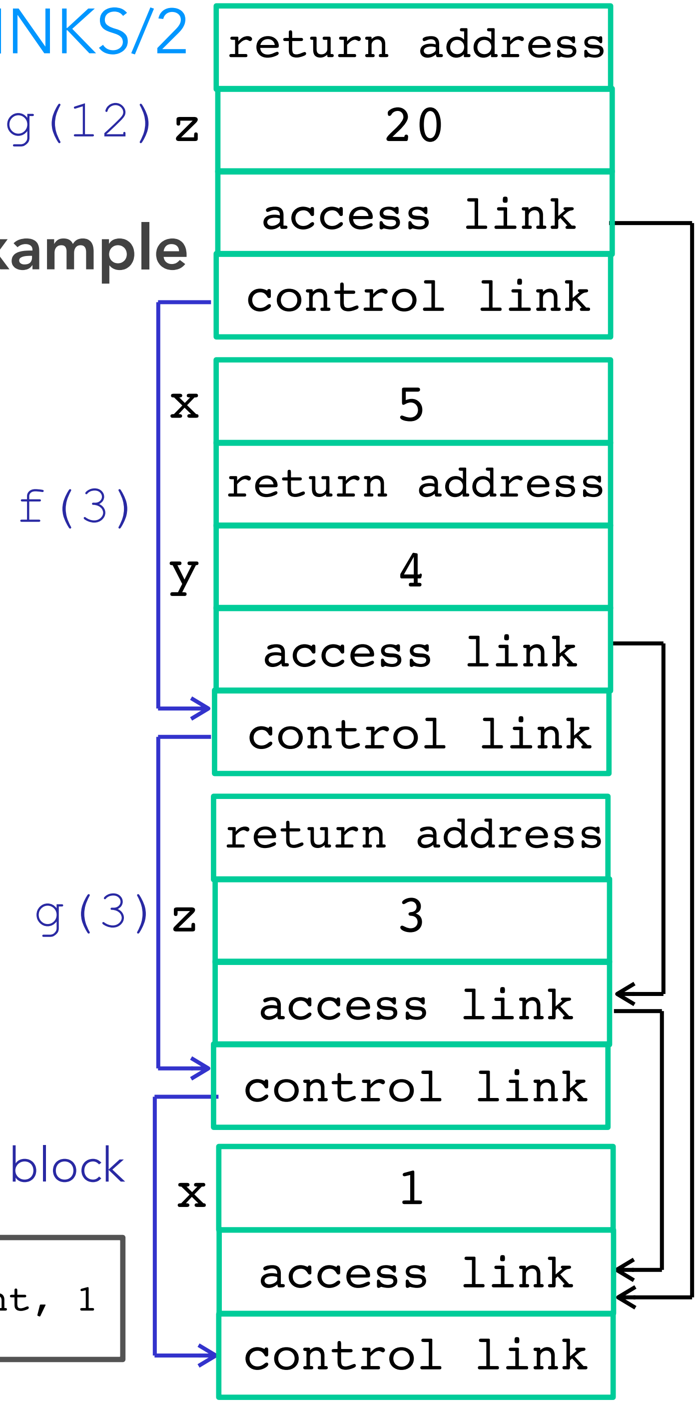
```
y: int, 3   →   g: int→int, 2   →   x: int, 1
x: int, 3       f: int→int, 2
```



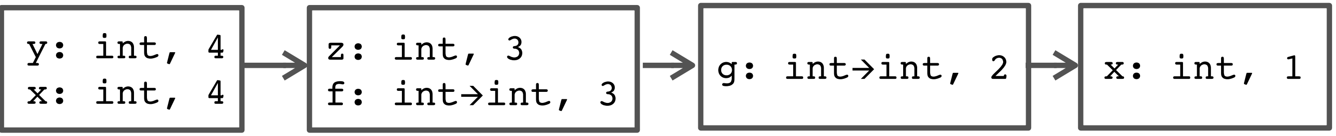
ACTIVATION RECORDS WITH ACCESS LINKS/2

the access links are set to the **AR of the enclosing syntactical block — another example**

```
int x = 1;
{ int g(int z){
      int f(int y){ int x = y+1;
                    return g(y*x);
      }
      return f(x+z);
  }
  g(3);
}
. . .
```



the symbol table at this point


y: int, 4
x: int, 4

z: int, 3
f: int→int, 3

g: int→int, 2

x: int, 1

SETTING THE ACCESS LINK

the value of the access link of a new activation record is established as follows:

- an inner block is entered or a function **declared in the current scope** is called:

  ```
  ACCESS_LINK = address of ACCESS_LINK in current AR
  ```

- a function calls itself recursively or calls another function **declared in the enclosing syntactical block**:

  ```
  ACCESS_LINK = value of ACCESS_LINK of the current AR
  ```

- **in general**, call to a function **outside the current scope**:

  ```
  ACCESS LINK = follow the chain of ACCESS_LINKs for
                  the difference between current
                  nesting level and that of function
                  declaration; let AR' be the activation
                  record
                  address of ACCESS_LINK of AR'
  ```

HEAP STORAGE

a value that **outlives the procedure** that creates it cannot be kept in the AR

```
Bar foo() { return new Bar() ; }
```

- the `Bar` value must survive deallocation of `foo`'s AR

languages with **dynamically allocated data** use a **heap** to **store dynamic data**

- the ARs in the stack are deallocated when the control exits from the corresponding scope
- **what about data in the heap?**

GARBAGE COLLECTION

what about data in the heap?

- they can be removed when they become “**garbage**”
- at a given point `p` in the execution of a program, a memory location `m` is **garbage** if no continuation of `p` can access location `m`

garbage collection:

- detects garbage during program execution
- is invoked when more memory is needed
- **the decision is made by the run-time system**, not by the program

GARBAGE COLLECTION

in some programming languages, deallocation is under the responsibility of the programmer

example of deallocation in `C`

```
ptr = lst ; flag = true ;
while (ptr!=NULL  && flag){
    if (ptr->val == 1) { ptr = ptr->next ; flag = false ; }
    else { previous = ptr; ptr = ptr->next ; free(previous) ; }
    }
```

problem: in case of **sharing** (more pointers to the same location) the cell cannot be actually freed (**dangling pointers**)

- in the above example, `lst` is a dangling pointer when `lst->val != 1`

other languages have **implicit** garbage collection algorithms

GARBAGE COLLECTION ALGORITHM/MARK AND SWEEP

mark-and-sweep algorithm

- assume **tag bits** associated with data
- assume that the addresses of locations created by a program are collected into a **table**
- **the algorithm**:
  1. set all tag bits to `0`
  2. start from each location used directly in the program (look for them from the AR) and follow all links, changing tag bit to `1`
  3. consider as garbage all cells with tag = `0`

GARBAGE COLLECTION ALGORITHM/REFERENCE COUNTING

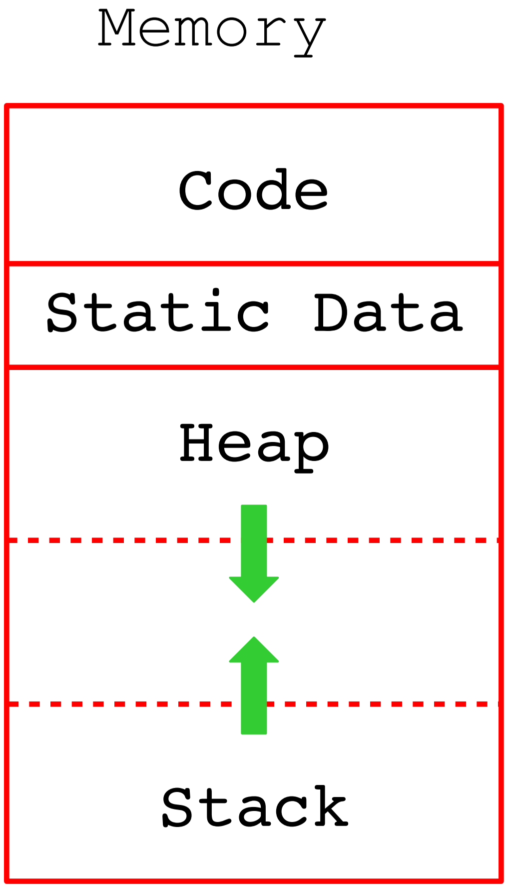
reference counting algorithm

- assume each datum in memory has an associated **reference counter**
- **the algorithm**:
  1. when a datum is allocated in memory, initialize the counter to `0`
  2. when a pointer to a datum is set, increments the counter
  3. when a pointer to a datum is reset, decrement the counter
  4. when the counter turns to be `0`, the datum is garbage

RECAPS

- the **code area** contains **object code**
    - for most languages, **fixed size and read only**
- the **static area** contains **data** (not code) with fixed addresses (e.g., global data)
    - **fixed size**, may be **readable or writable**
- the **stack** contains an AR for each currently active procedure
    - each AR usually has **fixed size** (for a given procedure), contains locals
- the **heap** contains all other data
- both the **heap and the stack grow**
    - you must take care that they **don't grow into each other**
    - **solution**: start heap and stack at **opposite ends of memory** and let them grow towards each other

MEMORY LAYOUT WITH HEAP



CODE GENERATION

several possible code generations!

- it depends on virtual machine and its bytecode instructions

a simple virtual machine: a stack machine with registers

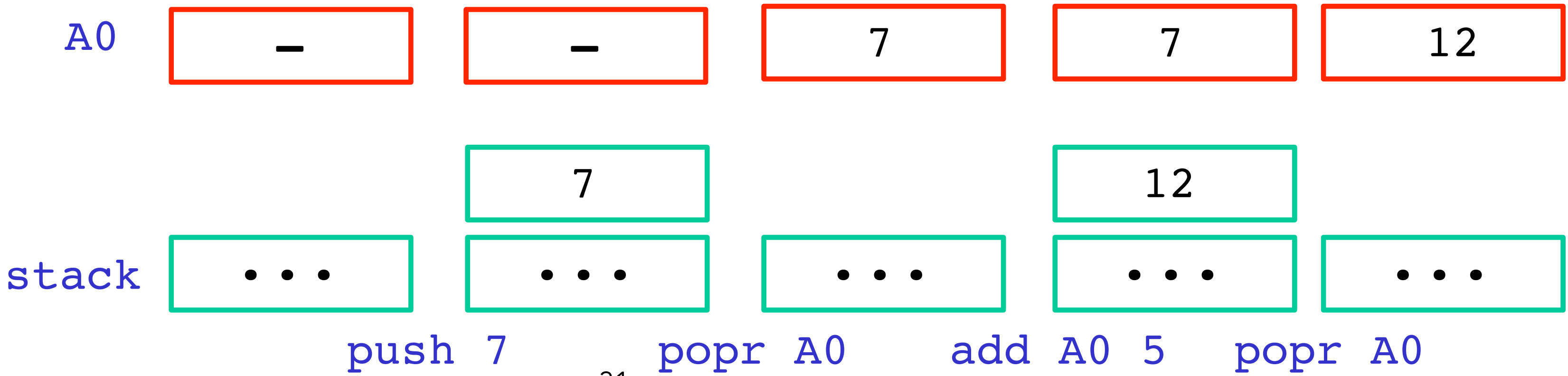
- stack machines are very common because they use the stack of activation records
- the **Java Virtual Machine** is a stack machine with registers
- the **Simple Virtual Machine** (SVM) is a stack machine with registers

CODE GENERATION FOR STACK MACHINES WITH REGISTERS

stack machines with (finitely many) **registers** are a simple evaluation model

- use the **stack** (of activation records) and registers to store **values** for intermediate results
- **example**: consider the instructions
  - `push n` : places the integer `n` on top of the stack
  - `popr A0` : pops the value on top of the stack and stores it in the register `A0`
  - `add A0 m` : adds the value of `A0` to `m` and puts the result back on the stack
  - a program that computes `7 + 5` and stores the result in `A0`:

```
push 7
popr A0
add A0 5
popr A0
```



A0 – – 7 7 12

7 12

stack • • • • • • • • • • • • • • •

push 7 popr A0 add A0 5 popr A0

IN STACK MACHINES WITH REGISTERS

each operation takes operands either from the stack or from registers and puts results on the stack or in a register

- this means a **simple** compilation scheme

the **location** of the **operands** is either **implicit** (the stack) or **explicit** (the registers)

- when the operands are implicit, they are always on **top of the stack**
- therefore there is **no need** to specify operands explicitly
- there is **no need** to specify the location of the result

FROM STACK MACHINES TO ASSEMBLY LANGUAGE

we consider a compiler that generates code for a stack machine with registers

- we want to run the resulting code on some processor
- we will define a machine, called `SVM` (**Simple Virtual Machine**) and an **interpreter** for the `SVM` machine code
- the interpreter will be in `Java`

our **assembly language**

- has **arithmetic operations** that use registers for operands and results
  - we will use `A0`, `RA`, `FP`, `SP`, `AL`, `T1`
- use **load** and **store** instructions for moving values between stack/memory and registers

A SAMPLE OF ASSEMBLY INSTRUCTIONS

- `add R1 R2`
    - `push(R1 + R2)      // the value of R1 plus the value`
      `// of R2 is stored on the stack`
- `addi R1 n`
    - `push(R1 + n)       // the value of R1 plus n is`
      `// stored on the stack`
- `sub R1 R2`
    - `push(R1 - R2)      // the value of R1 plus the value`
      `// of R2 is stored on the stack`
- `storei R1 n        // n is stored in R1`
- `move R1 R2         // the value of R1 is stored in R2`
- `pushr R1         // pushes the value of R1 on the stack`
- `popr R1     // pops the stack, the value is stored in R1`

A BASIC LANGUAGE

a basic language with integers and integer operations

$$
\begin{array}{l}
\mathtt{P} \rightarrow \mathtt{D\ ;\ P\ |\ E} \\
\mathtt{D} \rightarrow \mathtt{T\ id(ARGS)\ =\ E} \\
\mathtt{ARGS} \rightarrow \mathtt{T\ id,\ ARGS\ |\ T\ id} \\
\mathtt{E} \rightarrow \mathtt{Int\ |\ Id\ |\ if\ (E_1\ ==\ E_2)\ then\ E_3\ else\ E_4} \\
\qquad \mathtt{|\ E_1\ +\ E_2\ |\ E_1\ -\ E_2\ |\ id(E_1, \ldots, E_n)}
\end{array}
$$

T is only int

in this language there is no variable declaration — there are formal parameters only!

- `Int` are integer constants, `Id` are the identifiers,
- the rightmost expression is the "main"
- there is recursion — the program computing the product of two numbers:

```
int product(int m, int n) =
      if (n == 0) then 0
      else if (n == 1) then m
      else m + product(m, n-1) ;
product(5,7)
```

CODE GENERATION STRATEGY

for each expression `e` we generate assembly code that:

- computes the value of `e` and stores it in `A0` (`A0` is a special register called **accumulator**)
- **preserves** `SP` and **the contents of the stack**

we define a **code generation function**

`cgen(SymbolTable Γ, Node e)`

whose result is the code generated for `e`

invariant: the result of computing an expression **is always in the accumulator**

- **after** computing an expression **the stack is as before**

the invariant must be satisfied by `cgen(Γ,e)`

CODE GENERATION FOR CONSTANTS AND ADD

- the code to **evaluate a constant** simply copies it into the accumulator:

```
cgen($\Gamma$, n) = storei A0 n
```

  - this preserves the stack, as required

- the code to **evaluate an add expression** is:

```
cgen($\Gamma$, $e_1$ + $e_2$) = cgen($\Gamma$, $e_1$)
                                pushr A0
                                cgen($\Gamma$, $e_2$)
                                popr T1
                                add A0 T1
                                popr A0
```

this is ok, but be carefull!
you need to verify that the
old value of T1 is useless!

  - this code preserves the stack, as required
  - **possible optimization**: put the result of e_1 directly in register `T1` ?

ANOTHER CODE FOR ADD

possible optimization: put the result of `e`$_1$ directly in `T1`

```
cgen(Γ, e₁ + e₂) =
                cgen(Γ, e₁)
                move A0 T1    // T1 ← A0
                cgen(Γ, e₂)
                add A0 T1
                popr A0
```

this is wrong!

try to generate code for : `3 + (7 + 5)`

REMARKS ABOUT CODE GENERATION

1. the code for **+** is a template with "**holes**" for code for evaluating e_1 and e_2
2. stack-machine code generation is **recursive**
3. code for $e_1 + e_2$ consists of code for e_1 and e_2 **glued together**
4. code generation can be written as a **recursive-descent visit** of the AST

CODE GENERATION FOR SUB

- the code is

```
cgen($\Gamma$, $e_1$ - $e_2$) =
                    cgen($\Gamma$, $e_1$)
                    pushr A0
                    cgen($\Gamma$, $e_2$)
                    popr T1
                    sub T1 A0
                    popr A0
```

  - this code preserves the stack, as required
  - the old value of `T1` is useless

CODE GENERATION FOR CONDITIONAL — ATTEMPT 1

we need a **flow control instructions**

`beq R1 R2 label`

- branch to `label` if `R1 = R2`
- another branching instruction: `b label` (**unconditional jump** to `label`)

```
cgen(Γ, if ($e_1$==$e_2$) then $e_3$ else $e_4$) =
                          cgen(Γ, $e_1$)
                          pushr A0
                          cgen(Γ, $e_2$)
                          popr T1
                          beq A0 T1 true_branch
                    false_branch:
                          cgen(Γ, $e_4$)
                          b end_if
                    true_branch:
                          cgen(Γ, $e_3$)
                    end_if:
```

CODE GENERATION FOR CONDITIONAL — SOLUTION

we have added labels

the code for if (e_1 = e_2) then e_3 else e_4 :

```
cgen(Γ, if (e1 == e2) then e3 else e4) =
                   ~~false_branch = newlabel()~~;
                   true_branch = newlabel();
                   end_if = newlabel();
                   cgen(Γ, e1)
                   pushr A0
                   cgen(Γ, e2)
                   popr T1
                   beq A0 T1 true_branch
                 ~~false_branch~~:
                   cgen(Γ, e4)
                   b end_if
                 true_branch:
                   cgen(Γ, e3)
                 end_if:
```

the labels must be fresh
to avoid name clashes

false_branch is useless!

RECAP OF THE BYTECODE

the bytecode language is the following one (up-to now)

```
bytecode =   ( storei R1 n
               | add R1 R2
               | sub R1 R2
               | addi R1 n
               | pushr R1
               | popr R1
               | move R1 R2
               | beq R1 R2 label
               | b label
               | label: )*           ;
```

we have used the registers `A0, T1`

THE ACTIVATION RECORD

code for function calls and function definitions depends on the **layout of the activation record**

a very simple AR suffices for this language:

- the result is always in the **accumulator**
  - no need to store the result in the AR
- the activation record **holds actual parameters**
  - for `f(x₁,…,xₙ)` push `xₙ,…,x₁` on the stack
  - these are the only variables in this language
- the stack discipline guarantees that on function exit `SP` is the same as it was on function entry
  - there is **no need** to store `SP` in the AR
- we need to store the **return address**

THE ACTIVATION RECORD (CONT.)

- we need to implement the stack of activation records
  - the AR must store a pointer to an address of caller's AR
  - this is the **chain of control links** used before
  - this pointer is stored in the register `FP` (**frame pointer**)
  - we take `FP` to point **below** the position of the first parameter of the called function (to the old value of `FP`)
  - `FP` is used by generated code to locate AR elements, e.g. parameters, based on offsets

THE ACTIVATION RECORD (CONT.)

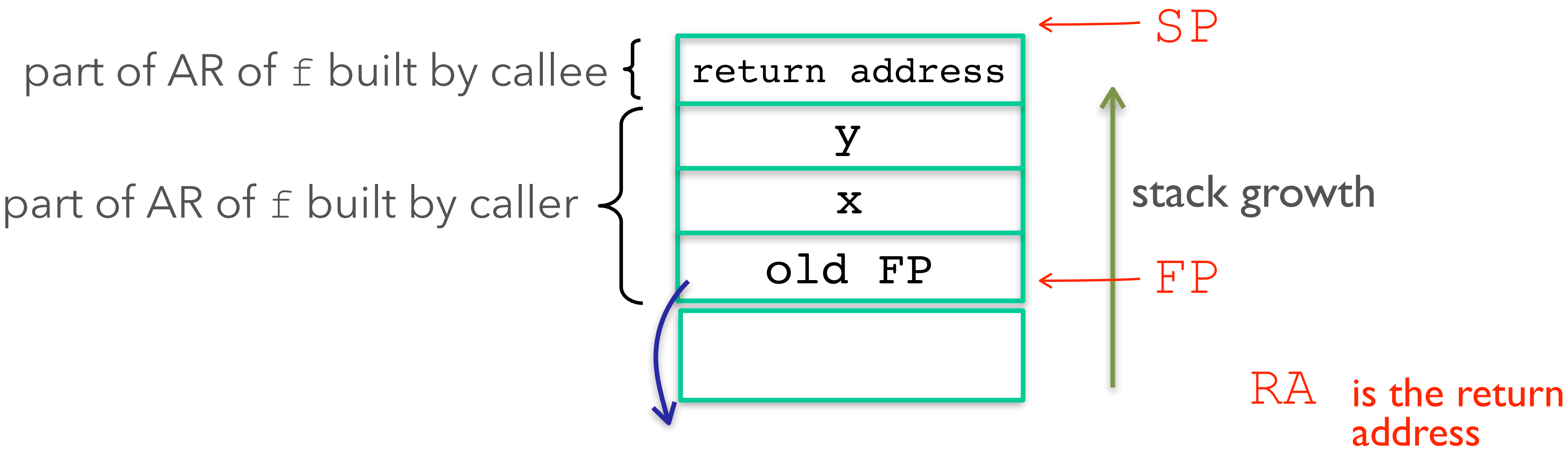
what about the **access link**?

- access links **are not needed** because we do not have nested declarations
- we just have local parameter declarations and functions that are all declared in the global scope that is allocated statically

THE ACTIVATION RECORD: SUMMARY

for our simple language, an AR with the caller's frame pointer (control link), the actual parameters, and the return address suffices

consider a call to `f(x,y)`, the AR will be:



CODE GENERATION FOR VARIABLES

in our simple language the “variables” of a function are just its parameters

- they are all in the AR
- pushed by the **caller**

problem: because the stack grows when intermediate results are saved, the variables are **not at a fixed offset** from `SP`

- example: `int f(x) = 3+x`

solution: use the **frame pointer**

- it always points to the first variable

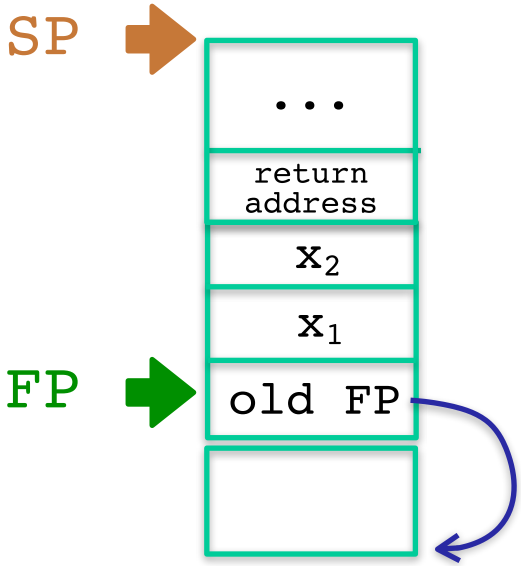
CODE GENERATION FOR VARIABLES: EXAMPLE

we use the instruction

```
store R1 offset(R2)
```

that stores in `R1` the value at address `R2+offset`

for $\texttt{int f}(x_1, x_2) = \texttt{e}$ the activation and frame pointer are set up as follows:



- $\mathbf{x}_1$ is at `FP - 1`
- $\mathbf{x}_2$ is at `FP - 2`
- thus, the access to $\mathbf{x}_i$ is

$$\texttt{cgen}(\Gamma, x_i) = \texttt{store A0 } \underline{\texttt{lookup}(\Gamma, x_i)\texttt{.offset(FP)}}$$

with $z = i$

gives the offset of x_i inside the AR

- the offset of a parameter needs to be **inserted in its symbol table entry**

CODE GENERATION FOR FUNCTION CALL

the calling sequence consists of instructions (of both caller and callee) that set up a function invocation

new instruction: `jsub label`

- **jump to label**, **save** address of next instruction in `RA`

```
cgen(Γ,f(e_1,…,e_n)) = pushr FP
                       cgen(Γ, e_1)
                       pushr A0
                        .  .  .
                       cgen(Γ, e_n)
                       pushr A0
                       move SP FP
                       addi FP n+1
                       jsub lookup(Γ, f).label
```

store in `FP` the address of the old value of `FP` which is at `SP+n+1`
this must be done **after** having evaluated the actual parameters! WHY?

the label is set when the symbol table is created for function definitions

- the caller saves its value of the frame pointer
- then it saves the actual parameters in reverse order
- then it saves the return address in register `RA`
- the AR so far is `n+1` words

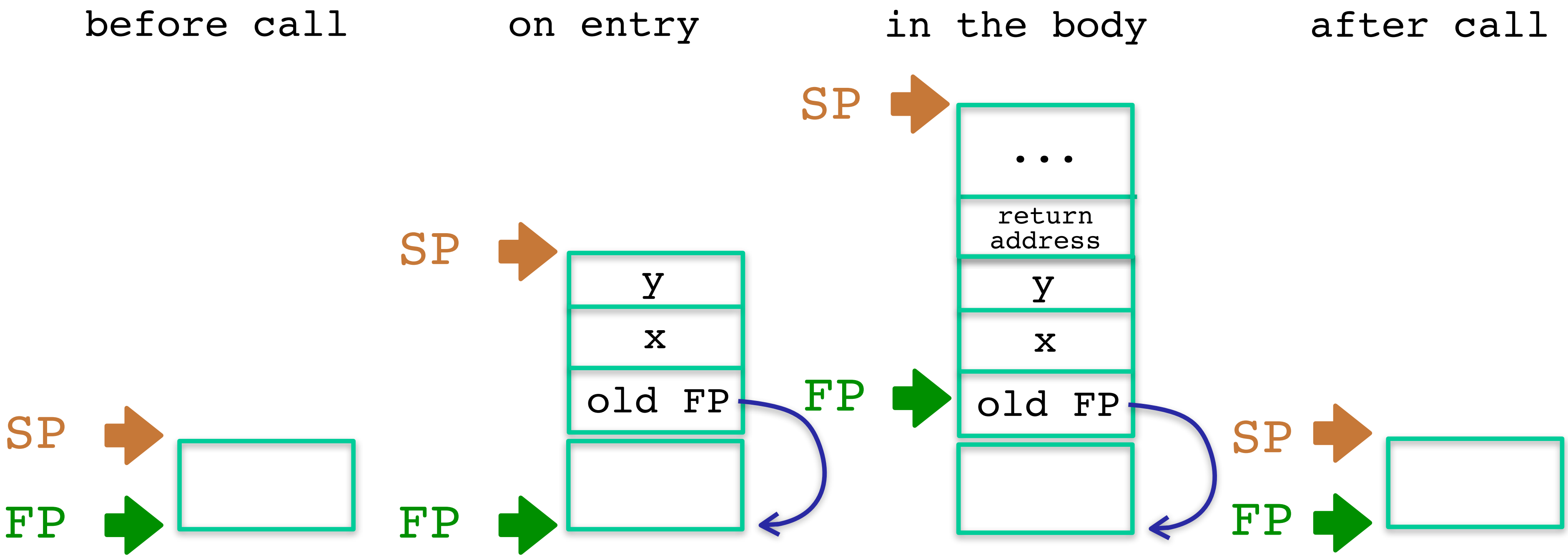
CODE GENERATION FOR FUNCTION DEFINITION

new instruction: `rsub RA // jump to address in RA`

```
cgen(Γ,int f(int x₁,…,int xₙ) = e) =
                lookup(Γ,f).label:
                                  pushr RA
                                  cgen(Γ,e)
                                  popr RA
                                  addi SP n
                                  popr FP
                                  rsub RA
```

- the frame pointer does point to the bottom of the frame
- the callee pops the return address, the actual arguments and the saved value of the frame pointer
- `n` is the number of formal parameters of `f` // the `addi` is equivalent to pop n times
- the return value is left in `A0`

CALLING SEQUENCE: EXAMPLE FOR `F(X,Y)`



because **the stack grows when intermediate results are saved**, ARs **are not adjacent** on stack!

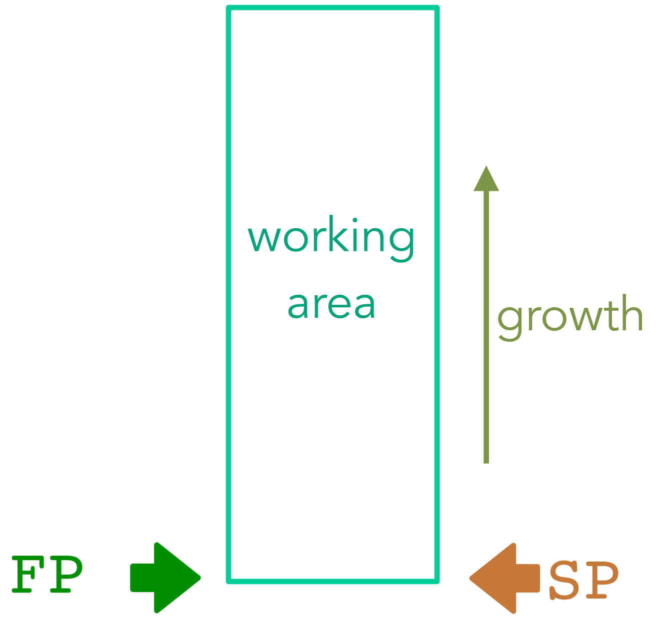
- example: `int f(int x, int y) = (x+3)+g(y)`
- upon execution, the value of `x+3` is on the stack between the AR of `f` and the AR of `g`

CODE GENERATION FOR PROGRAMS

what is cgen(∅, D_1; … ; D_n; E) ?

we are in a very simple case:

- there is no declaration of variables
- D_i are all function definitions



```
cgen(∅, D1; … ; Dn; E) =
            storei SP max_value
            storei FP max_value
            cgen(Γ,E)               // where  ∅ ⊢ D1; … ; Dn : Γ
            halt                    // end of the program
            cgen(∅, D1; … ; Dn) // the sequence of fun declar.

cgen(Γ, D; D') = cgen(Γ, D)    // D is a single function
                 cgen(Γ', D') // Γ ⊢ D : Γ' a single function
```

SUMMARY

the **activation record layout** must be designed together with the code generator

code generation can be done by recursive traversal of the AST

to simplify the presentation we have not discusses the access links!

but **access links can be easily added** (as discussed in “memory management”):

- check the difference in the nesting level between the caller and the declaration of the callee, and follow the already settled access links accordingly

RECAP OF THE BYTECODE

the bytecode language is the following one

```
bytecode =  ( storei R1 n
            | add R1 R2
            | sub R1 R2
            | addi R1 n
            | pushr R1
            | popr R1
            | move R1 R2
            | beq R1 R2 label
            | b label
            | label:
            | store R1 offset(R2)
            | jsub label
            | rsub R1 )*          ;
```

in red: new instructions and new registers

we have used the registers `A0, T1, SP, RA, FP`

EXAMPLE: CODE GENERATION FOR PRODUCT

```
cgen(Γ, n) = storei A0 n

cgen(Γ, e1 + e2) = cgen(Γ, e1)
                   pushr A0
                   cgen(Γ, e2)
                   popr T1
                   add A0 T1
                   popr A0

cgen(Γ, if (e1 == e2) then e3 else e4) =
               true_branch = newlabel();
               end_if = newlabel();
               cgen(Γ, e1)
               pushr A0
               cgen(Γ, e2)
               popr T1
               beq A0 T1 true_branch
               cgen(Γ, e4)
               b end_if
  true_branch: cgen(Γ, e3)
       end_if:
```

```
cgen(Γ,x) = store A0 lookup(Γ,x).offset(FP)

cgen(Γ,f(e1,…,en)) = pushr FP
                     cgen(Γ, e1)
                     pushr A0
                     . . .
                     cgen(Γ, en)
                     pushr A0
                     move SP FP
                     addi FP n+1
                     jsub lookup(Γ, f).label

cgen(Γ,int f(int x1,…,int xn) = e) =
   lookup(Γ,f).label: pushr RA
                      cgen(Γ,e)
                      popr RA
                      addi SP n
                      popr FP
                      rsub RA
```

```
int product(int m, int n) = if (n == 0) then 0
                            else if (n == 1) then m
                            else m + product(m, n-1) ;
product(5,7)
```

EXAMPLE: CODE GENERATION FOR PRODUCT

```
         storei SP max_value
         storei FP max_value
         pushr FP
         storei A0 5
         pushr A0
         storei A0 7
         pushr A0
         move SP FP
         addi FP 3
         jsub Prod
         halt
```

cgen(Γ, product(5,7))

```
int product(int m, int n) = if (n == 0) then 0
                            else if (n == 1) then m
                            else m + product(m, n-1) ;
product(5,7)
```

```
  Prod: pushr R
         store A0 -2(FP)
         pushr A0
         storei A0 0
         popr T1
         beq A0 T1 true_E
         store A0 -2(FP)
         pushr A0
         storei A0 1
         popr T1
         beq A0 T1 true_I
         store A0 -1(FP)
         b end_I
 true_I: store A0 -1(FP)
         pushr A0
         pushr FP
         store A0 -1(FP)
         pushr A0
         store A0 -2(FP)
         pushr A0
         storei A0 1
         popr T1
         sub T1 A0
         popr A0
         pushr A0
         move SP FP
         addi FP 3
         jsub Prod
         popr T1
         add A0 T1
         popr A0
  end_I: b end_E
 true_E: storei A0 0
  end_E: popr RA
         addi SP 2
         popr FP
         rsub RA
```

cgen(Γ, if_esterno)

cgen(Γ, if_interno)

else
if interno

then
if interno

THE BASIC LANGUAGE WITH ACCESS LINKS AND STATEMENTS

the basic language grows . . .

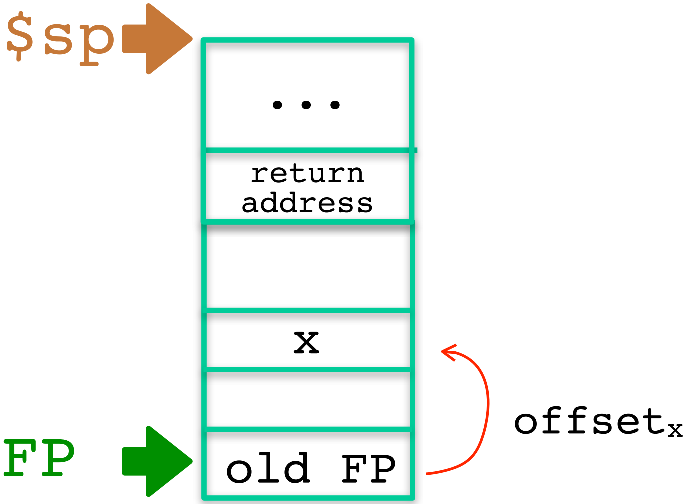
```
P     →  D ; P | E | S
D     →  T id(ARGS) = P
ARGS  →  id, ARGS | id
E     →  int | id | if (E1 == E2) then E3 else E4
          | E1 + E2 | E1 - E2 | id(E1,…,En)
S     →  ( id = E ; | id(E1,…,En) ; )+
T     →  int | void
```

- `int` and `void` are the **types**
- there are **nested declarations** of functions
- **example**:

```
void foo(int x,int y) = void gee(int z) x = x+z ;
                        if (y == 0) then skip ;
                        else ( gee(y) ; foo(x,y-1) ; )
foo(0,10) ;
```

← skip is e.g. x=x;

STATEMENTS



we discuss assignments `x = e ;`

- **simple case:** `x` is in the current RA
- `offset`$_x$ is at an offset with respect to the current value of `FP`
- we use a new instruction:

  ```
  load R1 offset(R2)
  ```

  that loads in the address `R2+offset` the value at `R1`

- thus

```
cgen(Γ, x = e;) =
          cgen(Γ, e)
          load A0 lookup(Γ, x).offset(FP)
```

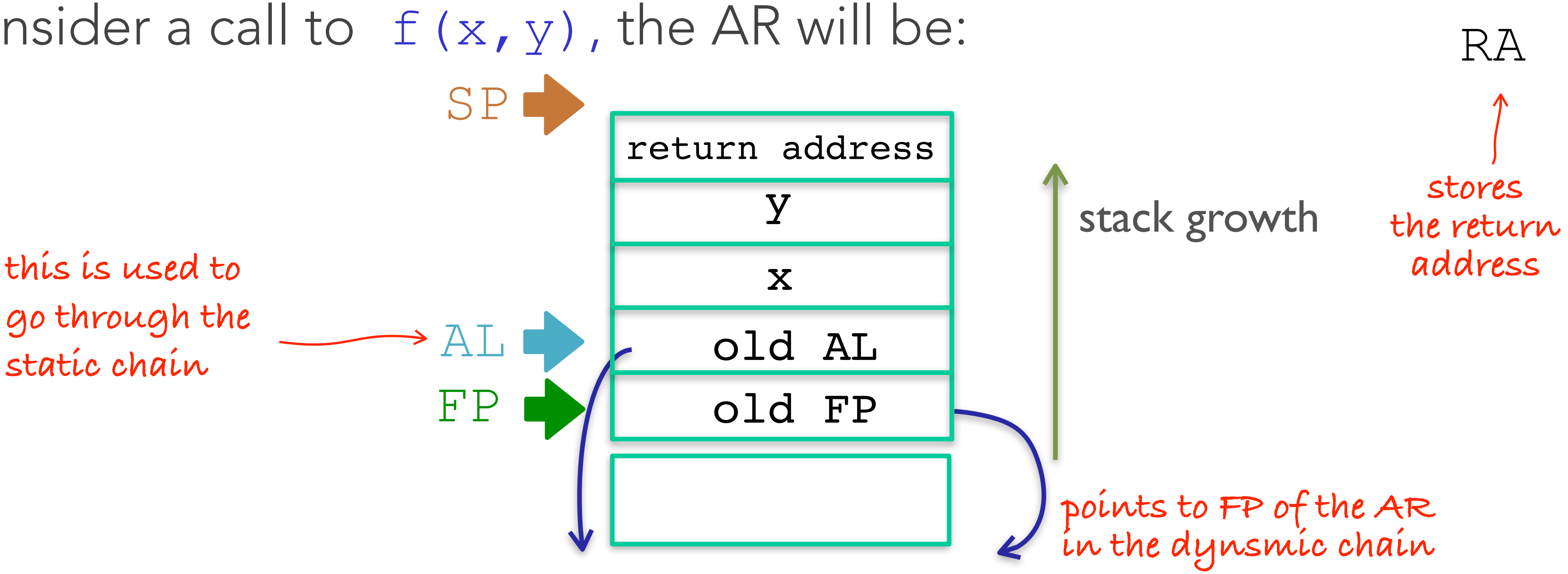
where is offset$_x$?

THE ACCESS TO GLOBAL VARIABLES: ACCESS LINKS

the AR for our language has

- the caller's frame pointer,
- the **access link** to the enclosing environment (in the static chain),
- the actual parameters,
- and the return address

consider a call to `f(x,y)`, the AR will be:



SP

return address

y

x

old AL

old FP

AL

FP

stack growth

RA

stores the return address

this is used to go through the static chain

points to FP of the AR in the dynsmic chain

THE ACCESS TO GLOBAL VARIABLES: ACCESS LINKS

```
cgen(Γ,x) =
          move AL T1
          for (i=0;
               i < nesting_level -
                   lookup(Γ, x).nesting_level;
               i++) store T1 0(T1) ;
          subi T1 lookup(Γ, x).offset
          store A0 0(T1)
```

gives the nesting level of `x` wrt the current one

this is the dereferentiation of AL

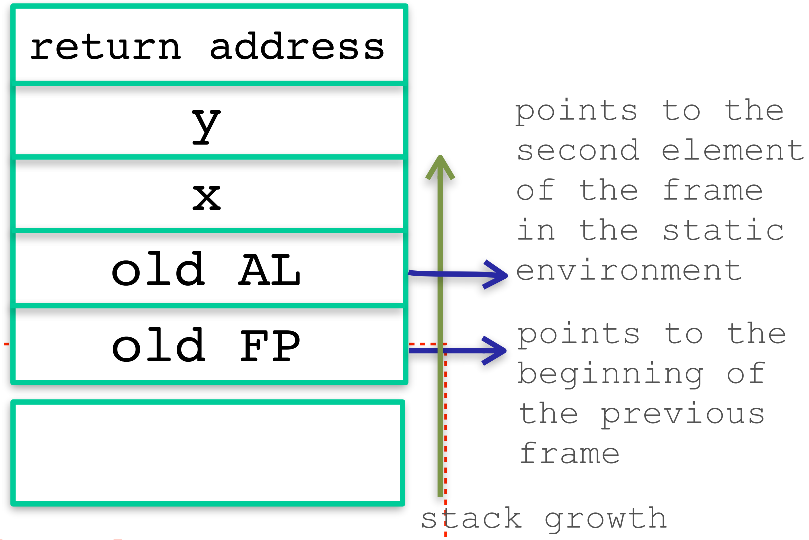
```
cgen(Γ, x = e;) =
               cgen(Γ, e)
               move AL T1
               for (i=0;
                    i < nesting_level -
                        lookup(Γ, x).nesting_level;
                    i++) store T1 0(T1) ;
               subi T1 lookup(Γ, x).offset
               load A0 0(T1)
```

we do not use anymore `FP`!

exercise: define `cgen(Γ, S ; S')`

CODE GENERATION FOR FUNCTION CALL WITH ACCESS LINKS

the code for the invocation in slide 51 must be refined … managing access links



```
cgen(Γ, f(e1,…,en)) =
        pushr FP
        move SP FP
        addi FP 1
        move AL T1
        for ( i=0;
          i < nesting_level -
              lookup(Γ,f).nesting_level;
          i++ ) store T1 0(T1) ;
        pushr T1
        cgen(Γ, e1)
        pushr A0
          . . .
        cgen(Γ, en)
        pushr A0
        move FP AL
        subi AL 1
        jsub lookup(Γ, f).label
```

management of the access link

the label is taken from the symbol table

NEW CODE GENERATION FOR FUNCTION DEFINITION

there may be several functions called in the same way

- the label of the first instruction must be taken from the symbol table!

```
cgen($\Gamma$, T f($T_1$ $x_1$,…,$T_n$ $x_n$) = e) =
        lookup($\Gamma$,f).label:
              pushr RA
              cgen($\Gamma$, e)
              popr RA
              addi SP n
              pop
              store FP 0(FP)
              move FP AL
              subi AL 1
              pop
              rsub RA
```

the label is set when the symbol table is created for function definitions

two pop: one for AL and the other for FP

ADVANCED TOPICS

- the implementation of **higher order functions**
- the implementation of **oo languages** and of **dynamic dispatch**

HIGHER-ORDER FUNCTIONS

in some languages, functions can be passed as parameters

```
fun bool f( x:(int,int)→bool, a:int, b:int){
     ...
     bool z = x(a,b) ;
     ...
}
```

function `f` should prepare the activation record for the execution of `x(a,b)`

- **PROBLEM**: `f` has no knowledge about how to set the “access link”
- should point to **most recent AR where the called function is declared** (based on difference with nesting level of such a declaration, computed at compile time)

HIGHER-ORDER FUNCTIONS (CONT.)

solution:

- the caller of `f`, that passes the actual value for the parameter `x`, should also pass a pair containing:
  1. **the address of the code of the function `g` that is actually passed (usual value of identifier `g`)**
  2. **the address of the most recent AR in which `g` is declared (additional context information)**
- this pair is called **closure** of the function to be passed
- the value of `x` is set to such a pair (values of identifiers of function type are pairs)
- when `x(a,b)` is executed the access link will be set to the second element of this pair

CODE GENERATION FOR OO LANGUAGES

two issues:

1. how are **objects represented in memory**?
2. how is **dynamic dispatch** implemented?

example

```
class A {
    int a = 0;
    int d = 1;
    int f(){ return a + d; }
}

class  B extends A {
    int b = 2;
    int f(){ return a ; }          //override
    int g(){ return a - b; }
}

class  C extends A {
    int c = 3;
    int h(A x) { return x.f()*c; }
}

A u = new B() ; C w = new C() ; w.h(u) ;
```

dynamic dispatch!

- fields of `A` are inherited by classes `B` and `C`
- all methods in all classes refer to `a`
- for methods to work correctly in `A`, `B`, and `C` objects, field `a` must be in the same "place" in each object

PROBLEM 1: OBJECT LAYOUT

an object is like a `struct` in **C**

- the reference `foo.field` is an index into a `foo` struct at an offset corresponding to `field`
- an objects is stored in a contiguous memory
  - each field stored at a fixed offset in the object
  - the offset needs to be inserted in its symbol table entry
- on object creation, the corresponding layout is instantiated in the heap (it will be either eventually removed by the garbage collector or with an explicit delete operation)

PROBLEM 1: OBJECT REPRESENTATION OF SUBCLASSES

remark: given a layout for class `A`, a layout for subclass `B` can be defined by **extending the layout** of `A` with additional slots for the additional fields of `B`

- this leaves the layout of `A` **unchanged** (layout of `B` is an extension of it)

PROBLEM 2: DYNAMIC DISPATCH

consider again our example

```
class A {
   int a = 0;
   int d = 1;
   int f(){ return a + d; }
}

class  B extends A {
   int b = 2;
   int f(){ return a ; }          //override
   int g(){ return a - b; }
}

class  C extends A {
   int c = 3;
   int h(A x) { return x.f()*c; }
}
```

- `e.g()` calls method `g` of `B` if `e` yields a `B` object
- `e.f()` calls method `f` of `A` if `e` yields an `A` or `C` object (`f` is inherited in the case of `C`) calls method `f` of `B` if `e` yields a `B` object (even if static type of `e` is `A`)

the implementation of methods and dynamic dispatch **strongly resembles the implementation of fields**

PROBLEM 2: DYNAMIC DISPATCH/DISPATCH TABLES

every class has a fixed set of methods (including inherited methods)

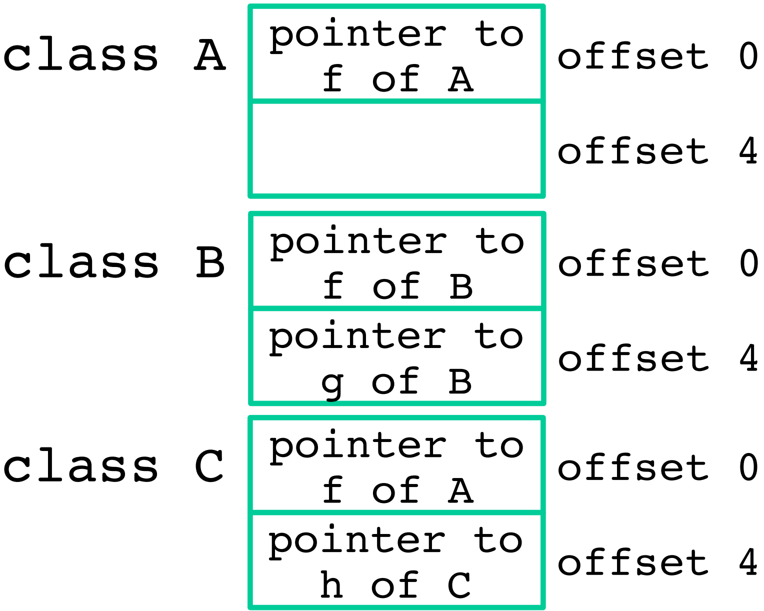
a dispatch table indexes these methods

- an array of method addresses ← this is also called VIRTUAL TABLE
- a method `f` lives at a fixed offset in the dispatch table for a class and all of its subclasses

example: the dispatch table for class `A` has only 1 method

the tables for `B` and `C` extend the table for `A`

because methods can be overridden, the code for `f` is not the same in every class, but is always at the same offset



class A	pointer to f of A	offset 0
		offset 4
class B	pointer to f of B	offset 0
	pointer to g of B	offset 4
class C	pointer to f of A	offset 0
	pointer to h of C	offset 4

USING DISPATCH TABLES

the dispatch pointer in an object of class `C` points to the dispatch table for class `C`

every method `f` of class `C` is assigned an offset O_f in the dispatch table at compile time

- the offset is inserted in the symbol table entry of method `f` of class `C` as usual

to implement a dynamic dispatch `e.f()` we

- let O_f be the offset of the method `f` in the dispatch-table associated to the static type of `e`
- evaluate `e`, obtaining an object `o` (that could be of any subclass)
- let `D` be the dispatch-table of `o`
- execute the method pointed by $D[O_f]$

THE SVM GRAMMAR

point to MEMORY[]

points to CODE[]

use **MEMORY** to store data; use registers **SP, RA, RV, FP, HP, IP,**

```
assembly: ( 'load' REG NUMBER '(' REG ')' // = memory[NUMBER+REG_r]<-REG_l
    | 'store' REG NUMBER '(' REG ')'// = REGleft <- memory[NUMBER + REGright]
    | 'storei' REG NUMBER           // = REG <- NUMBER
    | 'move' REG REG                // = REGleft <- REGright
    | 'add' REG REG                 // = top <- REGleft + REGright
    | 'addi' REG NUMBER             // = top <- REGleft + NUMBER
    | 'sub' REG REG                 // = top <- REGleft - REGright
    | 'subi' REG NUMBER             // = top <- REGleft - NUMBER
    | 'mul' REG REG                 // = top <- REGleft * REGright
    | 'muli' REG NUMBER             // = top <- REGleft * NUMBER
    | 'div' REG REG                 // = top <- REGleft / REGright
    | 'divi' REG NUMBER             // = top <- REGleft / NUMBER
    | 'push' (n=NUMBER | l=LABEL)   // = memory[sp] = number|label , sp = sp-1
    | 'pushr' REG                   // = memory[sp] = REG , sp = sp-1
    | 'pop'                         // = sp = sp+1
    | 'popr' REG                    // = REG <- memory[SP+1] == STORE REG 0(SP)
    | 'b' LABEL                     // = ip = LABEL
    | 'beq' REG REG LABEL           // = if REGleft == REGright => ip = LABEL
    | 'bleq' REG REG LABEL          // = if REGleft <= REGright => ip = LABEL
    | 'jsub' LABEL
    | 'rsub' REG                    // = REG = RA
    | l=LABEL ':'
    | 'halt'                        //terminate the execution
    )* ;
```

in SVM.g4

COMMENTS ABOUT SVM

- example: the PLUS node

```
public String codeGeneration() {
    return left.codeGeneration()+
           "pushr A0 \n" +
           right.codeGeneration()+
           "popr T1 \n" +
           "add A0 T1 \n" +
           "popr A0 \n" ;
}
```

THE SVM GRAMMAR

- the code in the `SimpLan` compiler for invoking functions

Dichiarazioni Locali
Return Address
PARAMETRI ATTUALI
Access Link
Frame Pointer

```
public String codeGeneration() {
     String parCode="";
     for (int i = 0; i < parameters.size() ; i = i+1)
             parCode += parameters.get(i).codeGeneration() + "pushr A0\n" ;

     String getAR="";
     for (int i=0; i < nesting - entry.getnesting() ; i++)
             getAR+="store T1 0(T1) \n";

     return  "pushr FP \n"
             + "move SP FP \n"
             + "addi FP 1 \n"
             + "move AL T1\n"
             + getAR
             + "pushr T1 \n"
             + parCode
             + "move FP AL \n"
             + "subi AL 1 \n"
             + "jsub " + entry.getlabel() + "\n" ;
 }
```

FUNCTION DEFINITION IN SIMPLAN

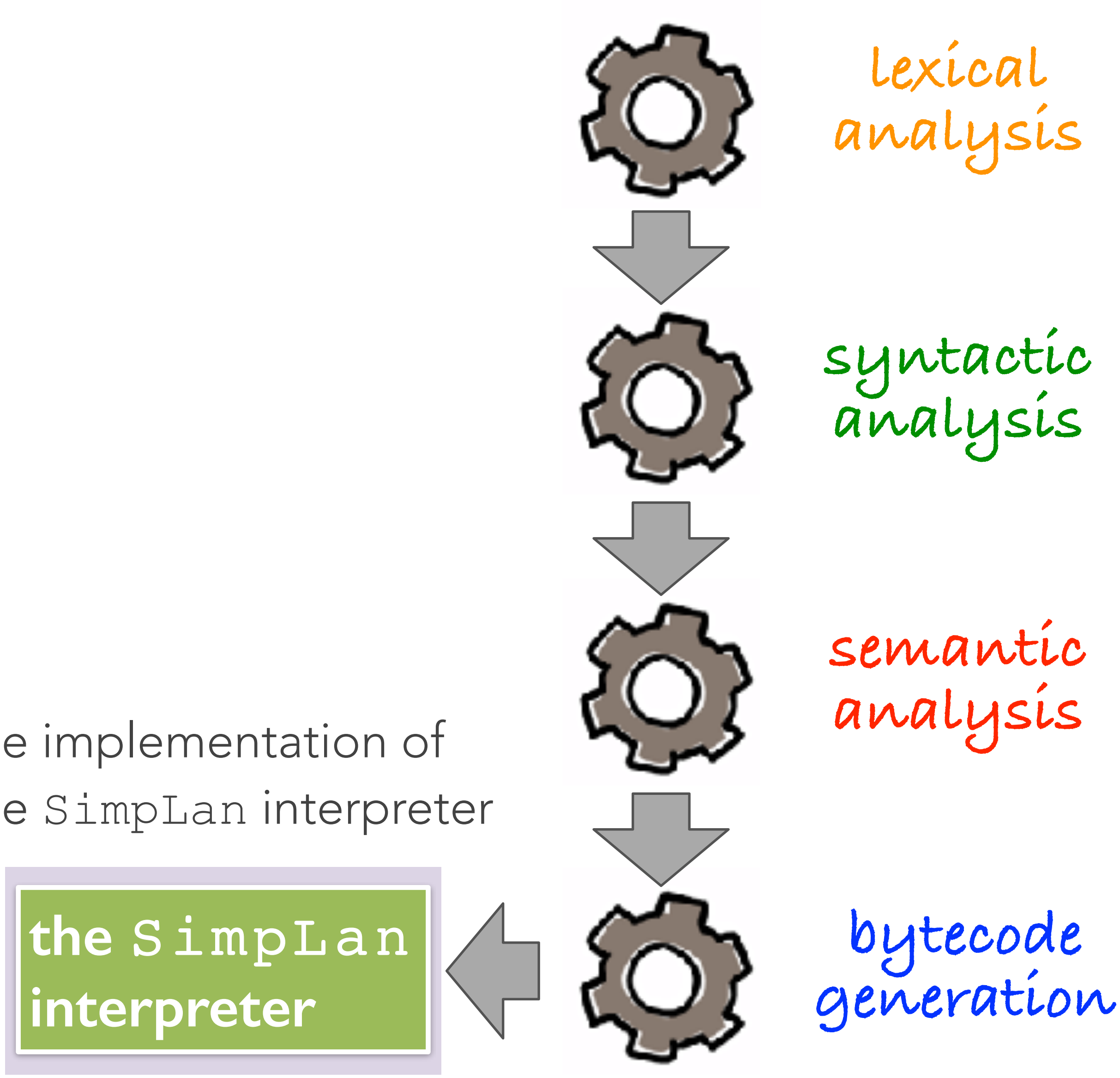
```
   cgen(Γ,int f(int $x_1$,…,int $x_n$) = e) =
add(Function,lookup(Γ,f).label:
                              pushr RA
                              cgen(Γ,e)
                              popr RA
                              addi SP n
                              popr SP
                              popr FP
                              rsub RA
) push lookup(Γ,f).label
```

this is put into a file/string called `Function` that collects all the functions and and will be justapoxed to the code of the main expression

this is what has been done in the SimpLan prototype

see the code!

NEXT LECTURE



lexical analysis

syntactic analysis

semantic analysis

bytecode generation

the `SimpLan` interpreter

the implementation of
the `SimpLan` interpreter