One aspect of run-time organisation: stack-based storage allocation

- Lifetime and storage
- Basic stack allocation:
  - stack frames
  - dynamic links
- Allocation for nested procedures:
  - non-local variable access
  - static links

**Storage Areas**

- **Static storage**: storage for entities that live throughout an execution.
- **Stack storage**: storage allocated dynamically, but deallocation must be carried out in the opposite order to allocation.
- **Heap storage**: region of the memory where entities can be allocated and deallocated dynamically as needed, in any order.

**Example: Lifetime (1)**

```plaintext
var x, y, z: ...
proc P()
    var p1, p2: ...
    begin ... end
proc Q()
    var q1, q2: ...
    begin ... if ... Q(); ... end
proc R()
    var r1, r2: ...
    begin ... Q() ... end
begin ... P() ... R() ... end
```
Example: Lifetime (2)

![Diagram of lifetime example](https://via.placeholder.com/150)

Example: Lifetime (3)

```java
private static Integer foo(int i) {
    Integer n = new Integer(i);
    return n;
}
```

- The lifetimes of `i` and `n` coincide with the invocation of `foo`.
- The lifetime of the integer object created by `new` starts when `new` is executed and ends when there are no more references to it.
- The integer object thus survives the invocation of `foo`.

Storage Allocation (1)

- **Global variables** exist throughout the program’s run-time.
- Where to store such variables can thus be decided **statically**, at compile (or link) time, once and for all.

Example:

```java
private static String[] tokenTable = ...
```

Storage Allocation (2)

- **Arguments** and **local variables** exist only during a function (or procedure or method) invocation:
  - Function calls are properly nested.
  - In case of **recursion**, a function may be **re-entered** any number of times.
  - Each function activation needs a private set of arguments and local variables.
- These observations suggest that storage for arguments and local variables should be allocated on a **stack**.
Storage Allocation (3)

- When the lifetime does not coincide with procedure/function invocations, *heap allocation* is needed. E.g. for:
  - objects in object-oriented languages
  - function closures in languages supporting functions as first class entities
  - storage allocated by procedures like `malloc` in C.
- Such storage either *explicitly deallocated* when no longer needed, or *automatically reclaimed* by a garbage collector.

Stack Frames

One *stack frame* or *activation record* for each currently active function/procedure/method. Contents:

- Arguments
- Bookkeeping information; e.g.
  - Return address
  - Dynamic link
  - Static link
- Local variables
- Temporary workspace

Defining the Stack

The stack is usually defined by a handful of registers, dictated by the CPU architecture and/or convention. For example:

- **SB**: Stack Base
- **ST**: Stack Top
- **LB**: Local Base

The names vary. Stack Pointer (SP) and Frame Pointer (FP) are often used instead of ST and LB, respectively.

Typical Stack Frame Layout

<table>
<thead>
<tr>
<th>address</th>
<th>contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>LB - argOffset</td>
<td>arguments</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>LB</td>
<td>static link</td>
</tr>
<tr>
<td>LB + 1</td>
<td>dynamic link</td>
</tr>
<tr>
<td>LB + 2</td>
<td>return address</td>
</tr>
<tr>
<td>LB + 3</td>
<td>local variables</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>LB + tempOffset</td>
<td>temporary storage</td>
</tr>
</tbody>
</table>

where

\[
\text{argOffset} = \text{size(arguments)}
\]

\[
\text{tempOffset} = 3 + \text{size(local variables)}
\]

TAM uses this convention. (Word (e.g. 4 bytes) addressing assumed, offsets in words.)
Example: A function \( f \)

(Not quite current MiniTriangle, but language could easily be extended in this way.)

```plaintext
var n: Integer;
...
fun f(x,y: Integer): Integer =
  let
    z: Integer
  in begin
    z := x * x + y * y;
    return n * z
  end
```

Example: Calling \( f \)

Call sequence for \( f(3, 7) \ast 8 \):

```
2015 LOADL 3 ; 1st arg. (x)
2016 LOADL 7 ; 2nd arg. (y)
2017 CALL f
2018 LOADL 8
2019 MUL
```

Address of each instruction explicitly indicated to the left. Address of \( f \) here given symbolically by a label. Corresponds to the address where the code for \( f \) starts, say 2082.

Example: Stack layout on entry to \( f \)

On entry to \( f \); caller’s ST = \( f \)'s LB:

<table>
<thead>
<tr>
<th>address</th>
<th>contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>SB + 42</td>
<td>n: n</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>LB - 2</td>
<td>x: 3</td>
</tr>
<tr>
<td>LB - 1</td>
<td>y: 7</td>
</tr>
<tr>
<td>LB</td>
<td>static link</td>
</tr>
<tr>
<td>LB + 1</td>
<td>dynamic link</td>
</tr>
<tr>
<td>LB + 2</td>
<td>return address = 2018</td>
</tr>
</tbody>
</table>

Ret. addr. = old program counter (PC) = addr. of instruction immediately after the call instruction. New PC = address of first instruction of \( f \) = 2082.

Example: TAM Code for \( f \)

TAM-code for the function \( f \) (at address 2082):

```
LOADL 0
LOAD [LB - 2]; x
LOAD [LB - 1]; y
MUL
STORE [LB + 3]; z
LOAD [SB + 42]; n
LOAD [LB - 1]; y
MUL
LOAD [LB - 1]; y
POP 1 1
MUL
RETURN 1 2
```

RETURN replaces activation record (frame) of \( f \) by result, restores LB, and jumps to ret. addr. (2018).

Note: all variable offsets are static.
Dynamic and Static Links

- **Dynamic Link**: Value to which LB (Local Base) is restored by RETURN when exiting procedure; i.e. addr. of caller’s frame = old LB:
  - “Dynamic” because depends on where function was called from.

- **Static Link**: Base of underlying frame of function that immediately lexically encloses this one.
  - “Static” because depends on the program’s structure and not on its execution.
  - Used to determine addresses of variables of lexically enclosing functions.

Example: Stack Allocation (1)

```
let
  var a: Integer[3];
  var b: Boolean;
  var c: Character;
proc Y ()
  let
    var d: Integer;
    var e: record c: Character, n: Integer end
  in
  ...;
proc Z ()
  let
    var f: Integer
  in
  begin ...; Y(); ... end
in
  begin ...; Y(); ...; Z(); ... end
```

Example: Stack Allocation (2)

Initially LB = SB; i.e., the global variables constitute the frame of the main program.

Call sequence: main → Y (i.e. after main calling Y):

- Global variables:
  - SB: a[0], a[1], a[2], b, c
- Frame of Y:
  - LB: static link, dynamic link, return address, d, e, c, e, n
- ST:

Example: Stack Allocation (3)

Call sequence: main → Z → Y:

- Global variables:
  - SB: a[0], a[1], a[2], b, c
- Frame of Z:
  - static link, dynamic link, return address, f
- Frame of Y:
  - LB: static link, dynamic link, return address, d, e, c, e, n
- ST:
Exercise: Stack Allocation

Global variables

SB →

```
[0]
[1]
[2]
```

Frame of Z

```
static link
dynamic link
return address
f
dynamic link
static link
return address
d
e.c
e.n
```

Frame of Y

```
static link
dynamic link
return address
f
dynamic link
static link
return address
d
e.c
e.n
```

In Y, what is the address of: b? e.c? f?

Non-Local Variable Access (1)

Consider **nested** procedures:

```plaintext
proc P()
  var x, y, z: Integer
  proc Q()
    ... begin ... if ... Q() ... end
    proc R()
      ... begin ... Q() ... end
      begin ... Q() ... end
      begin ... Q() ... R() ... end
  end
end
```

P’s variables are in scope also in Q and R.

But how to access them from Q or R?

Neither global, nor local!

Belong to the **lexically enclosing procedure**.

Non-Local Variable Access (2)

In particular:

- We cannot access x, y, z relative to the stack base (SB) since we cannot (in general) statically know if P was called directly from the main program or indirectly via one or more other procedures.
- I.e., there could be arbitrarily many stack frames **below** P’s frame.

Non-Local Variable Access (3)

- We cannot access x, y, z relative to the local base (LB) since we cannot (in general) statically know if e.g. Q was called directly from P, or indirectly via R and/or recursively via itself.
- I.e., there could be arbitrarily many stack frames **between** Q’s and P’s frames.
Non-Local Variable Access (4)

Answer:
• The *Static Links* in Q’s and R’s frames are set to point to P’s frame on each activation.
• The static link in P’s frame is set to point to the frame of its closest lexically enclosing procedure, and so on.
• Thus, by following the chain of static links, one can access variables at any level of a nested scope.

Non-Local Variable Access (5)

Call sequence: main → ... → P → Q:

![Call sequence diagram]

Global variables
other frames
Frame of P

Frame of Q

Non-Local Variable Access (6)

Call sequence: main → ... → P → R → Q → Q:

![Call sequence diagram]

Non-Local Variable Access (7)

Consider further levels of nesting:

```
proc P()
  var x, y, z: Integer
  proc Q()
    proc R()
      ...
      begin ...if ... R() ... end
      ...
      begin ... Q() ... end
    begin ... R() ... end
  begin ... Q() ... end
```

Note: Q’s variables now in scope in R.

To access, compute the difference between scope levels of the accessing procedure/function and the accessed variable (note: static information), and follow that many static links.
Non-Local Variable Access (8)

Call sequence: main →...→P →Q →R →R:

Global variables
other frames
Frame of P

Frame of Q

Frame of R (1)

Frame of R (2)

Non-Local Variable Access (9)

TAM code, P calling Q: Q’s static link = P’s local base, pushed onto stack prior to call:

LOADA [LB + 0] ; Q’s static link
LOADCA #1_Q ; Address of Q
CALLI

TAM code, R calling itself recursively: copy of R’s static link (as calle’s and caller’s scope levels are the same) pushed onto stack prior to call:

LOAD [LB + 0] ; R’s static link
LOADCA #2_R ; Address of R
CALLI

Non-Local Variable Access (10)

Accessing y in P from within R; scope level difference is 2:

LOAD [LB + 0] ; R’s static link
LOADI 0 ; Q’s static link
LOADI 4 ; y at offset 4 in P’s frame