

Today's Topics

Last Time

- Abstract machines, run time **models**
- Expression stack (**ES**) model of **expression evaluation**
- Run stack (**RS**) model of **scopes** and **automatic variables**
- Managing the **Run Stack** - the **Dynamic Pointer Stack**, the Run Stack **Display**, (**LL,ON**) addressing
- Maintaining the **Display**, **value** and **reference** parameters

Today

- Modelling **procedures** and **functions**

Then

- Modelling storage layout of **arrays** and **records**
- What is **Semantic Analysis**?

Runtime Model - Procedures

- We normally enter new scopes by calling a **procedure** or **function**
- So far, we have described the **representation** of the new scope, but not the details of how it is constructed
- We can divide the procedure or function call into two components - the *caller setup*, and the *callee prologue*
- In the **caller setup**, we are interested in setting up the **parameters** and making the actual call to transfer control to the procedure (the *callee*)
- In the **callee prologue**, we are interested in establishing the storage and setup of the procedure's new scope

Procedures - Caller Setup

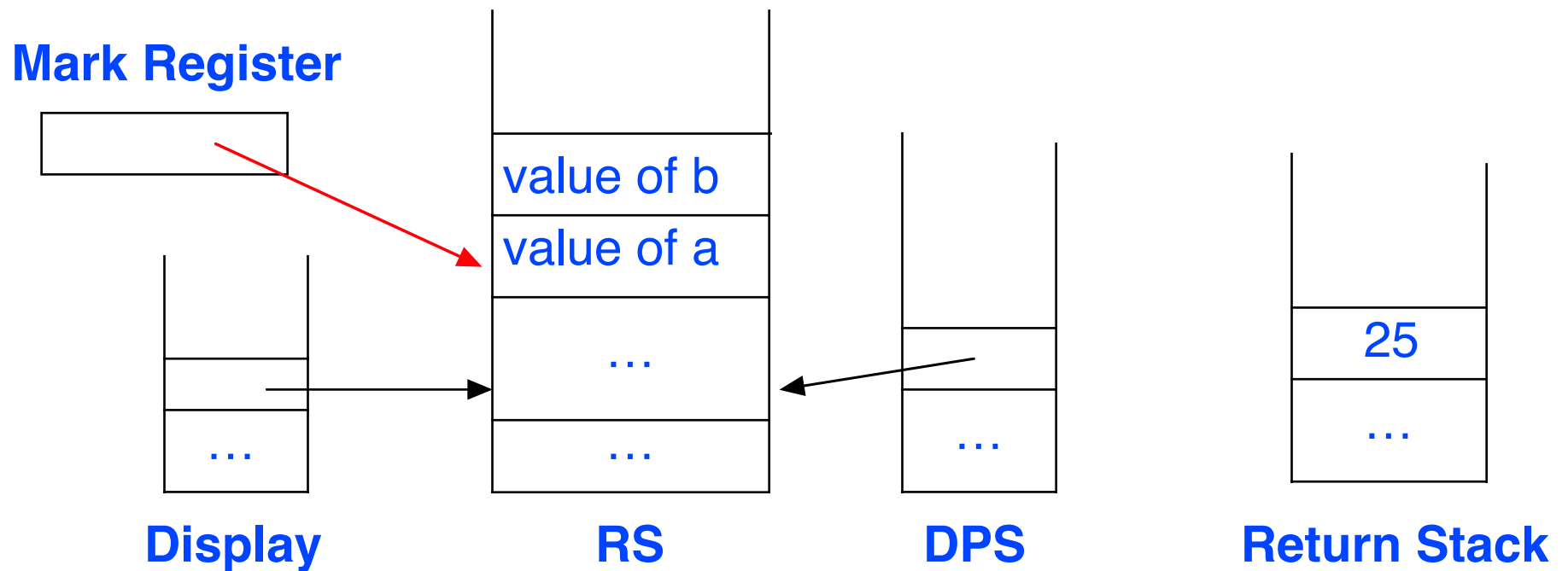
- Since **parameters** are treated as the first variables of the new scope, we have to remember where the top of the **RS** was when we started pushing parameters
- In our abstract machine, a new instruction called **markstack** is used for this purpose
- **Markstack** copies the current value of the runstack pointer to a temporary "**register**" of the machine
- We must also remember the location in the code to return to after the call - this involves yet another stack called the **return stack**

```
p (a, b) →      19: markstack
                  20: push  a
                  21: passparameter
                  22: push  b
                  23: passparameter
                  24: call  p
                  25:  ...
```

Procedures - Caller Setup

`p(a, b) →`

- 19: `markstack`
- 20: `push a`
- 21: `passparameter`
- 22: `push b`
- 23: `passparameter`
- 24: `call p`
- 25: `...`



Procedures - Callee Prologue / Epilogue

- The procedure's *prologue* is responsible for completing the setup of the **Run Stack**, **Display** and **Dynamic Pointer Stack**, and its *epilogue* is responsible for undoing it
- In our abstract machine, the single new instruction *enter* handles the **prologue**, and the new abstract machine instruction *return* handles the **epilogue** and returns control to the caller
- Each of these takes as operand the **lexical level** of the procedure (so they know which **Display** entry to modify), and the number of local variables (so *enter* knows how much storage to reserve on the **Run Stack** for the procedure's scope)

```
procedure p (a,b);           p:   enter   LL,Nlocals
    ...                       ...
end p;                       return  LL
```

Procedures - Callee Prologue / Epilogue

- What exactly does the *enter* instruction do?

DSPPointer += 1

push DPS frame

DPS[DSPPointer].RSPPointer := MarkReg

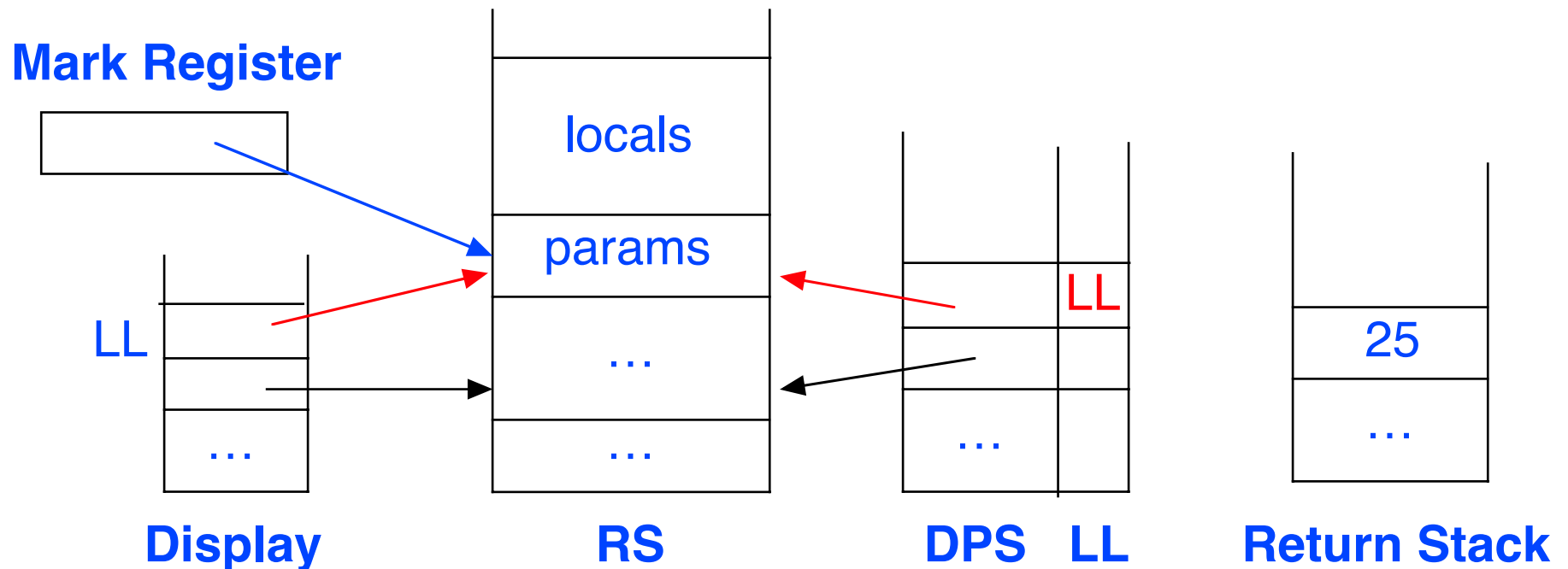
DPS[DSPPointer].LL := LL

Display[LL] := MarkReg

set Display[LL]

RSPPointer += Nlocals

allocate space for local vars



Procedures - Callee Prologue / Epilogue

- The *return* instruction undoes all this and returns control to the caller

$RSPPointer := DPS[DPSPPointer].RSPPointer - 1$

pop DPS frame

$DPSPPointer -= 1$

$Display[LL] := DPSSearch(DPS, DPSPPointer, LL)$

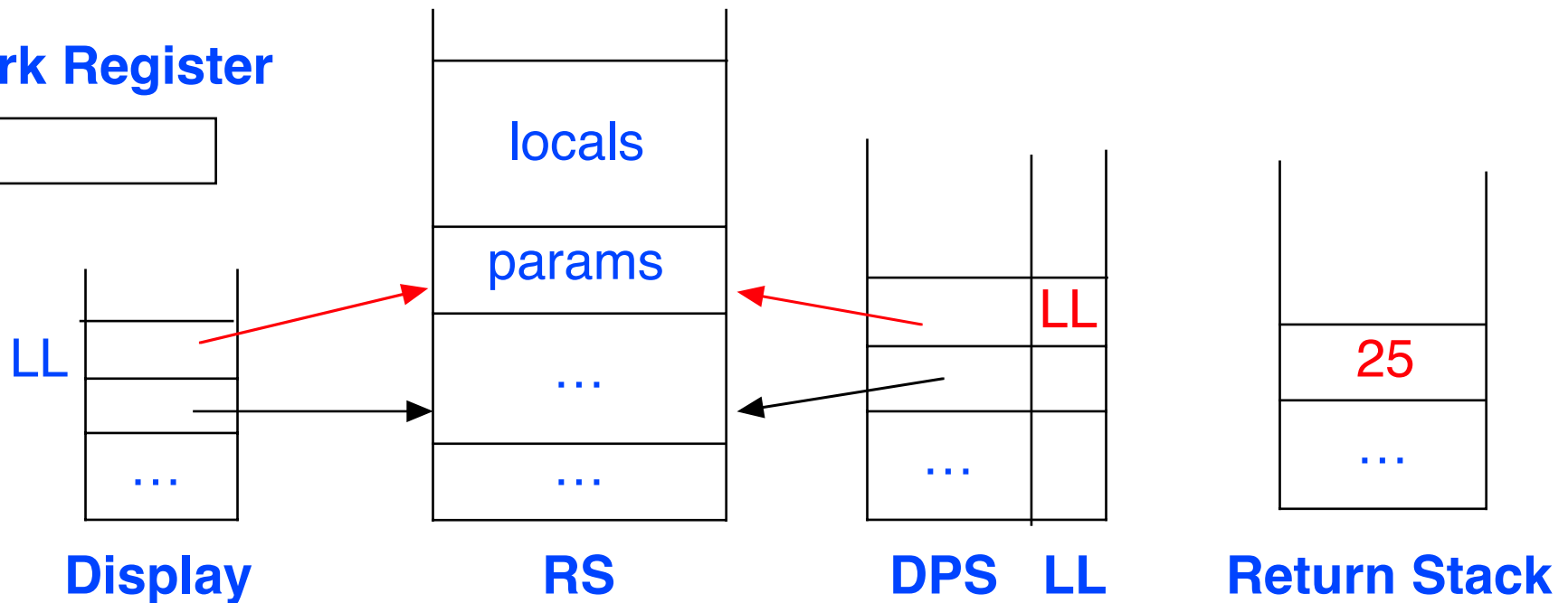
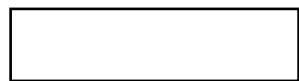
reset Display

$PC := ReturnStack (RetPointer)$

back to caller

$RetPointer -= 1$

Mark Register



Function Results

- Function **results** are normally returned by pushing their value on the **ES** - this is the right place for them, since the result of a function is a value to be used in an expression

```
function f(x) : integer;
```

```
...
```

```
return x;
```

```
...
```

```
push    x  
return  LL
```

- This also works for returning **objects** in **OO** languages, except that we would push a **reference** (address) of the returned object on the **ES**

Function Results

- In some languages (e.g., [Turing](#), [Ada](#), [Modula 3](#)), it is possible to return entire [arrays](#) as values (not objects or references to them, but a copy of the whole thing)
- This can be handled by creating a local array in the *caller* to receive the result, and then passing it by [reference](#) to the function

```
function p(a:int): array 1..100 of int
```

```
bar := p(1)[i] =>
```

```
var result: array 1..100 of int  
p (1, result)  
bar := result[i]
```

Summary

Procedures and Functions

- Caller setup, **prologue** and **epilogue**
- Returning **function** results

Next Week

- **Quiz #2**: Lexical and syntactic structure, grammars, **PDA** and **BNF**, bottom-up and top-down parsing, ambiguity, runtime model of expressions
- Text **chapters 7 - 11** inclusive (**lectures 8 - 13**, to end of ES)

Then

- **Storage layout** model
- Begin **Semantic Analysis**