Principles of Programming Languages

Lecture 06

Implementation of Block Structured Languages
Activations and Environment

- Aspects of Subroutines: Static vs Dynamic
  - Static subroutine: code (``reentrant’’)
    - Exactly one subroutine
  - Subroutine in execution: activation record, AR, activation
    - Many activations of same code possible

- State of a program in execution
  - A collection of activations
    - In a stack or in a heap
  - Contextual relationships among the activations
    - ``environment access’’ pointers & ``control’’ pointers

- Activation contents
  - Fixed: program code (shared)
  - Variable: activation
    - Instruction pointer (ip, ra) —also resumption address or return address
    - Control Pointer (dl) —also control link, dynamic link
    - Environment pointer (ep, fp) —also access link, frame pointer
      - Local environment (this activation)—fp
      - Nonlocal environment (other activations)—sl, static link
Assume static binding

Calling trace: P, R, Q, T, S, Q, T, S

d1 = dynamic link to caller at RT  
s1 = static link to statically enclosing activation

Notes d1 and s1 can be far apart

C SC 520 Principles of Programming Languages
**Static Nesting Level** \((snl)\) and **Distance** \((sd)\)

\[snl\] (name declaration)  
= \# of contour lines surrounding declaration

\[snl\] (name reference)  
= \# contour lines surrounding reference

**Static Distance of a Symbol Occurrence**
\[sd\] (name occurrence)  
= \# of contours crossed outward from occurrence to declaration  
= \(snl\) (name’s occurrence)  
- \(snl\) (that name’s decl.)
Symbol Table Computes $snl$

- Symbol table maps an occurrence of $x$ to
  - line #
  - $snl$ (declaration)
  - Offset among declarations
- Each name $x$ has an "address": (line #, $snl$, offset)
- Scanner keeps track of
  - Contour boundaries crossed (e.g. +1 for { & -1 for })
  - Current name declarations in scope
- Scanner can therefore
  - Identify declaration controlling a name occurrence
  - Replace a name occurrence by pointer to symbol table line #
Symbol Table (cont.)

Assume for simplicity all variables are int and occupy one address

<table>
<thead>
<tr>
<th></th>
<th>name</th>
<th>sn</th>
<th>offset</th>
<th>type &amp;c</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>P</td>
<td>0</td>
<td>0</td>
<td>int ()</td>
</tr>
<tr>
<td>2</td>
<td>a</td>
<td>1</td>
<td>0</td>
<td>int</td>
</tr>
<tr>
<td>3</td>
<td>Q</td>
<td>1</td>
<td>1</td>
<td>int (int)</td>
</tr>
<tr>
<td>4</td>
<td>R</td>
<td>1</td>
<td>2</td>
<td>int (int)</td>
</tr>
<tr>
<td>5</td>
<td>b</td>
<td>2</td>
<td>0</td>
<td>int</td>
</tr>
<tr>
<td>6</td>
<td>c</td>
<td>2</td>
<td>1</td>
<td>int</td>
</tr>
<tr>
<td>7</td>
<td>d</td>
<td>2</td>
<td>2</td>
<td>int</td>
</tr>
<tr>
<td>8</td>
<td>S</td>
<td>2</td>
<td>3</td>
<td>void (int)</td>
</tr>
<tr>
<td>9</td>
<td>T</td>
<td>2</td>
<td>4</td>
<td>int (int)</td>
</tr>
<tr>
<td>10</td>
<td>e</td>
<td>3</td>
<td>0</td>
<td>int</td>
</tr>
<tr>
<td>11</td>
<td>b</td>
<td>3</td>
<td>0</td>
<td>int</td>
</tr>
<tr>
<td>12</td>
<td>b</td>
<td>2</td>
<td>0</td>
<td>int</td>
</tr>
<tr>
<td>13</td>
<td>c</td>
<td>2</td>
<td>1</td>
<td>int</td>
</tr>
</tbody>
</table>
Activation Record Stack

- temps
- saved registers
- ra
- dl
- sl
- locals
- arguments
- temps
- locals

- stack pointer = next available location
- return address
- dynamic link
- static link
- frame pointer = currently executing AR

higher addresses
Activation Record Stack

- Model an AR by a struct (pretend all data are `int` for simplicity)

  ```
  struct AR {
    int arg[n];
    int local[m];
    AR* sl;
    AR* dl;
    CODE* ra;
    void* rx;  // register save area
    ...
  }
  ```

- Temps are pushed on top (``frame extension'') during execution in the activation record and are abandoned on return

- Assume stack growth to *higher* addresses (in reality usually the other way)
Registers

- Special purpose registers: $ra, sp, fp$
- General purpose registers divided into two classes
  - **Caller-saves**: transient values unlikely to be needed across calls
    - Caller assumes nothing valuable in caller-saves set & can be used at will (“destroyed”)
    - **Ex**: temp values during expression evaluation in caller
    - Caller saves these during calling sequence and they are restored after subroutine return
  - **Callee-saves**: used for local variables and indexes, etc.
    - Caller assumes these registers will not be destroyed by callee
    - **Ex**: register holding pointer during a list scan
    - Callee saves these in the prologue just after call, and restores in the epilogue just before return
Compiler Code Generation

- What is generated at CT (to be executed at RT):
  - Upon reference to a variable name \( x \)?
  - In caller before call to subroutine \( Q \) & after return to caller—the calling sequence?
  - In callee before execution of body?
    - Prologue
  - In callee before return to caller?
    - Epilogue

- Assume we are generating code inside body of a subroutine named \( P \)
  - Compiler maintains a level counter during code generation:
    \[ \text{curr	extunderscore level} = \text{current static nesting level of site where code is being generated (body of } P) \]
Access (Reference) to $x$ inside $P$

- From symbol table compiler can compute:
  - $x \rightarrow snl(x), \text{offset}(x)$
  - $P \rightarrow snl(P)$
  - $\text{curr\_level} = snl(P) + 1$ (level at which ref occurs)
  - $sd(x) = \text{curr\_level} - snl(x)$
- Generate code to compute $l$-value into $lv$:
  - $ap = \text{activation record ptr}$
    - $ap = fp$;
    - $for(i = 0; i < sd(x); i++) \ ap = ap->sl$;
    - $lv = ap + offset(x)$;
  - Use $lv$ on LHS of assignment, $*lv$ on RHS
Call $Q$ inside $P$

- Calling sequence for `call $Q$' in source
- Assume arguments passed by value

```
sp->arg[1] = value of argument 1; —transmit args

... 
sp->arg[n] = value of argument n;
fp->ra = resume; —set point to resume execution in caller
sp->dl = fp; —set callee's return link
fp->ry = ry; ...; —save caller-saves registers
ap = fp; —find AR of callee $Q$'s declaration
for(i = 0; i < sd(Q); i++) ap = ap->sl ;
sp->sl = ap; —set callee's static link
fp = sp; —switch to new environment
goto entrypoint(Q); —from symbol table, after $Q$ is compiled

resume: ...
```

- note stack has not been pushed (callee's responsibility)
Prologue Code for Subroutine Q

- Code executed just after caller jumps to callee
- Note compiler knows size of AR for Q
  \[sp = sp + size(AR \text{ of } Q)\]; —push stack frame for current activation
  \[fp->rx = rx; \ldots\]; —save any callee-saves registers
  - now \(sp\) points to next available stack location
  - now \(fp\) points to subroutine frame base
- Push could be done by caller (caller knows name of Q at CT)
  - But this will not work for closures (see below) where caller does not know name of callee at CT
Epilogue code for Subroutine $Q$

- Code executed just before return to caller
- Note compiler knows size of AR for $Q$

```
rx = fp->rx;                — restore any callee-saves registers
sp = sp - size(AR of Q);   — pop stack frame for current activation
fp = fp->dl;               — make caller’s activation current one
ry = fp->ry; ... ;         — restore caller-saves registers
goto fp->ra;               — resume execution in caller just after point of call
```

- now $sp$ points to next available stack location
- now $fp$ points to frame base of caller
Display Method

- Linked list replaced by array!
- Replace traversal of static chain by a single memory reference—more efficient calculation of non-local environment references.
- At CT, the maximum static nesting level is known; possible snl values are 1 .. maxsnl.
- The display is an array $D$ of maxsnl elements.
- $D[i] = fp$ for that part of the environment that is in an AR at snl $i$. 

Equivalent static chain

Executing at snl = 4

$D[1]$

$D[2]$

$D[3]$

$D[4]$

$D[maxsnl]$
Access (Reference) to $x$ inside $P$

- Generate code to compute $l$-value into $lv$:
  
  $$lv = *D[^{snl}(x)] + offset(x)$$

- Use $lv$ on LHS of assignment, $*lv$ on RHS
Call $\mathcal{Q}$ inside $\mathcal{P}$

Before call $\mathcal{Q}$
Executing at $snl = u$ in $\mathcal{P}$
Subr. $\mathcal{Q}$ defined at $snl = d \leq u$

After call $\mathcal{Q}$
Executing at $snl = d + 1$
(body of $\mathcal{Q}$ one level deeper)
$D[d+1]$ overwritten to point to new AR
Call $Q$ inside $P$ (cont.)

- $Q$ defined at $snl\ d \Rightarrow$ new AR executes at $snl\ d+1 \Rightarrow$
  - $D[d+1]$ points to new AR for $Q$
- Old $D[d+1]$ (dotted link) destroyed
- Saved in caller’s AR (since part of caller’s display)
  - New AR field $fp->disp$
- Other elements $D[i]$ where $i \geq d + 2$ left alone
  - An AR deeper in the stack might need them upon return
Call \( Q \) inside \( P \) (cont.)

- Calling sequence for ``call \( Q \)'' in source
- Let \( u = snl(P) \) & \( d = snl(Q) \)
  - Note \( fp == D[u] \)
  - \( sp->arg[1] = value \ of \ argument \ 1 \); —transmit args
    
    . . .
  - \( sp->arg[n] = value \ of \ argument \ n; \)
  - \( fp->ra= resume; \) —set return point in caller
  - \( sp->dl = fp; \) —set callee’s return link
  - \( fp->ry = ry; ...; \) —save caller-saves registers
  - \( fp->disp = D[d+1]; \) —save caller’s display entry to reset on return
  - \( D[d+1] = sp; \) —set display for callee; \( D[1..d] \) are shared
  - \( fp = sp; \) —switch to callee environment
  - \( goto \) \( entrypoint(Q); \) —from symbol table, after \( Q \) is compiled

\( resume: \ ... \)
Prologue/Epilogue code for Subroutine $Q$

- **Prologue same as before**

  \[
  sp = sp + \text{size(AR of } Q) \; ; \quad \text{—push stack frame for current activation}
  \]

  \[
  fp->rx = rx; \ldots; \quad \text{—save any callee-saves registers}
  \]

- **Epilogue restores caller’s display**
  
  - Let $u = \text{snl } (Q)$ —this is known to compiler
  
  \[
  rx = fp->rx; \; \ldots; \quad \text{—restore callee-save registers}
  \]

  \[
  sp = sp - \text{size(AR of } Q) \; ; \quad \text{—pop stack frame for current activation}
  \]

  \[
  fp = fp->dl; \quad \text{—make caller’s activation current}
  \]

  \[
  D[u] = fp->disp; \quad \text{—restore caller’s display}
  \]

  \[
  ry = fp->ry; \; \ldots \; ; \quad \text{—restore caller-saves registers}
  \]

  \[
  \text{goto } fp->ra; \quad \text{—resume execution in caller just after point of call}
  \]
Costs: Static Chain vs Display

- Compare count of memory references
  - Exclude argument transmission, reg. saves (common to both)
  - Assume \( fp, \ sp \) held in registers
- Analyze calling sequence for static chain

<table>
<thead>
<tr>
<th>instruction</th>
<th># refs</th>
</tr>
</thead>
<tbody>
<tr>
<td>( fp-&gt;ra= ) resume;</td>
<td>1</td>
</tr>
<tr>
<td>( sp-&gt;dl = fp; )</td>
<td>1</td>
</tr>
<tr>
<td>( sp-&gt;ry = ry; \ldots )</td>
<td>-</td>
</tr>
<tr>
<td>( ap = fp; )</td>
<td>0</td>
</tr>
<tr>
<td>( \text{for}(i = 0; i &lt; sd(Q); i++)) ( \text{ap = ap-&gt;sl;} )</td>
<td>( sd(Q) )</td>
</tr>
<tr>
<td>( sp-&gt;sl = ap; )</td>
<td>1</td>
</tr>
<tr>
<td>( fp = sp; )</td>
<td>0</td>
</tr>
<tr>
<td>( \text{goto } entrypoint(Q); )</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>( sd(Q)+3 )</td>
</tr>
</tbody>
</table>
Costs (cont.)

- Comparison by # memory references:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Static chain</th>
<th>Display</th>
</tr>
</thead>
<tbody>
<tr>
<td>Access local l-value</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Access non-local x l-value</td>
<td>$sd(x)$</td>
<td>2</td>
</tr>
<tr>
<td>Call $Q$</td>
<td>$sd(Q) + 3$</td>
<td>5</td>
</tr>
<tr>
<td>$Q$ Prologue</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$Q$ Epilogue</td>
<td>3</td>
<td>5</td>
</tr>
</tbody>
</table>

- Need lots of $sd(x) > 2$ & $sd(Q) > 2$ to make worth it
Funargs (Procedure/Function Arguments)

- Consider call $U(T)$;
  (both $U$ and $T$ are visible in body of $R$)
- $T$ is not visible to $U$ $\Rightarrow$ no $T$ activation in the static chain of $U$ $\Rightarrow$ at the call $T(2)$ in $U$, cannot locate definition environment of $T$!
- How is the call $F(2)$ implemented?
  - Must work for any $F$ actual
- What is passed to $U$ in $U(T)$?
Funargs (cont.)

- Consider call \( F(2) \); Previous calling sequence cannot be used. Missing information shown in blue:

\[
\begin{align*}
sp->arg[1] &= \text{value of argument 1}; \quad \ldots \quad \text{—transmit args} \\
fp->ra &= \text{resume}; \quad \text{—set return point in caller} \\
sp->dl &= fp; \quad \text{—set callee’s return link} \\
fp->ry &= ry; \quad \ldots \quad \text{—save caller-saves registers} \\
ap &= fp; \quad \text{—find AR of callee F’s declaration} \\
\text{for}(i = 0; i < sd(F); i++) ap = ap->sl; \\
sp->sl &= ap; \quad \text{—set callee’s static link} \\
f &= sp; \quad \text{—switch to new environment} \\
goto \text{entrypoint}(F); \quad \text{—from symbol table, after F is compiled} \\
resume: \ \ldots
\end{align*}
\]

Don’t know what F really is at CT and don’t know \( sd(F) \) and \( \text{entrypoint}(F) \)
Calling a Formal: \( F(\ldots); \) inside \( U(F) \)

- \( \text{sd}(F) \) is unknown at CT
- At RT, the actual functional argument need not even be in \( U \)'s static chain \( \Rightarrow \) it is inaccessible from the current AR
- \( \therefore \) environment of definition of each funarg must be passed to \( U \) as part of actual argument
- A \textit{funarg} or \textit{closure} is a pair \((ip, ep)\) where:
  - \( ip \) = entry address of the actual argument procedure
  - \( ep \) = reference to most recent activation of definition environment of actual argument procedure
Closure Implementation

- A closure is a pair of references:

  ```c
  struct CL {
    CODE* ip;    // —instruction pointer (entrypoint)
    AR* ep;     // —environment pointer
  }
  ```

- Closure \( f \) is built in caller when a named procedure is passed as an actual

- \( f \) is copied to callee \( u \) as actual corresponding to formal \( F \) : effectively ``\( F = f \)"

- When \( u \) calls \( F \), the static link in the new activation is set by \( sp->sl = F.ep \) and the jump is by \( goto F.ip \)
Call \( F \) inside \( \mathcal{U} \)

- Calling sequence for `\texttt{``call F''}` in source where \( F \) is a function formal

  \[
  \text{sp} \rightarrow \text{arg}[1] = \textit{value of argument 1}; \quad \text{—transmit args}
  \]

  \[
  \ldots
  \]

  \[
  \text{sp} \rightarrow \text{arg}[n] = \textit{value of argument n};
  \]

  \[
  \text{fp} \rightarrow \text{ra} = \text{resume}; \quad \text{—set return point to resume execution}
  \]

  \[
  \text{sp} \rightarrow \text{dl} = \text{fp}; \quad \text{—set callee’s return link}
  \]

  \[
  \text{fp} \rightarrow \text{ry} = \text{ry}; \ldots; \quad \text{—save caller-save registers}
  \]

  \[
  \text{ap} = \text{fp}; \quad \text{—find AR of callee \( Q \)’s declaration}
  \]

  \[
  \text{for}(i = 0; i < \text{sd}(Q); i++) \; \text{ap} = \text{ap} \rightarrow \text{sl} ;
  \]

  \[
  \text{sp} \rightarrow \text{sl} = F.\text{ep}; \quad \text{—set callee’s static link}
  \]

  \[
  \text{fp} = \text{sp}; \quad \text{—switch to new environment}
  \]

  \[
  \text{goto } F.\text{ip}; \quad \text{—entrypoint of code of actual}
  \]

  \[
  \text{resume: } \ldots
  \]
Constructing and Passing Closure

- Consider call $U(T)$ in AR for $R$
- Case: actual proc $T$ is visible, named proc & so is $U$

```
sp->arg[1].ip = entrypoint(T);
fp->ra = resume;  // set return point to resume execution
sp->dl = fp;     // set callee's return link
fp->ry = ry;    // ... // save caller-save registers
ap = fp;        // find AR of argument $T$'s declaration
for (i = 0; i < sd(T); i++) ap = ap->sl;
sp->arg[1].ep = ap; // environment of $T$ set in callee
ap = fp; for (i = 0; i < sd(U); i++) ap = ap->sl;
sp->sl = ap;    // set callee's static link
fp = sp;        // switch to new environment
goto entrypoint(U); // from symbol table
resume: ... 
```
Prologue/Epilogue Code

- Same as for "named" calls, since code is generated once for each possible named actual such as $\mathbf{T}$
- Information for allocation/deallocation known at CT for $\mathbf{T}$
Calls with Formal Procedures: Cases

- Let \( F, F' \) name formal functional parameters and let \( u \) name a visible, actual proc
- Discuss implementation of calling sequences for each of:
  - \( u(F) \);
  - \( F(T) \);
  - \( F(F') \);
Calls with Formal Procedures: \( F(T) \)

- Call to a formal proc with an actual visible named proc

\[
\begin{align*}
\text{sp} \rightarrow \text{arg}[1].\text{ip} &= \text{entrypoint}(T); \\
\text{ap} &= \text{fp}; & \text{—find AR of argument } T\text{’s declaration} \\
\text{for}(i = 0; i < sd(T); i++) \text{ ap} &= \text{ap} \rightarrow \text{sl} ; \\
\text{sp} \rightarrow \text{arg}[1].\text{ep} &= \text{ap}; & \text{—environment of } T \text{ set in callee} \\
\text{fp} \rightarrow \text{ra} &= \text{resume}; & \text{—set return point to resume execution} \\
\text{sp} \rightarrow \text{dl} &= \text{fp}; & \text{—set callee’s return link} \\
\text{fp} \rightarrow \text{ry} &= \text{ry} \ ; \ldots; & \text{—save caller-save registers} \\
\text{ap} &= \text{fp}; \text{ for}(i = 0; i < sd(F); i++) \text{ ap} &= \text{ap} \rightarrow \text{sl} ; \\
\text{sp} \rightarrow \text{sl} &= \text{ap}; & \text{—set callee’s static link} \\
\text{sp} \rightarrow \text{sl} &= F.\text{ep}; & \text{—set callee’s static link} \\
\text{fp} &= \text{sp}; & \text{—switch to new environment} \\
\text{goto } F.\text{ip}; & \text{—from closure of } F \\
\text{resume} &: \ldots
\end{align*}
\]
Challenge

- Can we implement functional parameters using the display?
  - Where does $F$ get its display? (No static chain to unravel given only a starting environment $F.ep$)
  - How is display restored upon return?
Blocks

- Extend existing environment: `{ int x; . . . }`
- Special case of subroutine:
  - No parameters
  - No name
  - Called in one place—where defined
  - Statically prior env. (surrounding block) == dynamically prior

```c
void function B();
{
    float x, y;
    x = z; y = 3;
    w = y;
}

... block
```

```c
... block
```
Block Activation/Deactivation

• A block is like a procedure, but
  ■ Nameless (because called from only one place)
  ■ Parameterless
  ■ Defined at its point of invocation (inline text)
  ■ Same static binding rules apply (static link == dynamic link)

sp→sl = fp;  —set callee’s return link
fp = sp;  —switch to new environment
sp = sp + size(AR of B);  —push stack frame for block activation

entrypoint(B): ...
  ... Body of B ...
  sp = sp - size(AR of B);  —pop stack frame for current activation
  fp = fp→sl;  —reactivate containing block
resume: ...

• Why are references in body of B resolved correctly?
• Can remove need for new AR by allowing caller’s AR to grow and shrink
Exercise

- Show how to handle block entry/exit with using the display method

Before block $B$ entry
Executing at $snl = u$ in $P$

After block $B$ entry
Executing at $snl = u + 1$
(body of $B$ one level deeper)
$D[u+1]$ overwritten to point to new AR
Solution to Exercise:

\[
\begin{align*}
\text{fps}-\to\text{disp} &= \text{D}[u+1]; & \text{— save caller’s display entry} \\
\text{D}[u+1] &= \text{sp}; & \text{— set callee’s display} \\
\text{fp} &= \text{sp}; & \text{— switch to new environment} \\
\text{sp} &= \text{sp} + \text{size(AR of } B); & \text{— push stack frame for block activation} \\
\end{align*}
\]

\text{entrypoint(}B): \ldots

\ldots \text{Body of } B \ldots

\begin{align*}
\text{sp} &= \text{sp} - \text{size(AR of } B); & \text{— pop stack frame for current activation} \\
\text{fp} &= \text{D}[u]; & \text{— reactivate containing block} \\
\end{align*}

\begin{align*}
\text{fp} &= \text{sp} - \text{size(AR of } P); & \text{— reactivate containing block} \\
\text{D}[u+1] &= \text{fp}-\to\text{disp}; & \text{— restore caller’s display} \\
\end{align*}

\text{resume: } \ldots
Non-local `goto`'s

- \( sd(L) = snl(\text{use } L) - snl(\text{def } L) = snl(D)+1 - snl(L) \)
- \( ap = fp; \)
- \( \text{for}(i = 0; i < \text{sd}(L); i++) \)
  \( \quad ap = ap->sl; \)
- \( fp = ap; \)
- \( sp = fp + \text{size}(\text{AR of } A); \)
- \( \text{goto address}(L); \)
- What if display is used? How restore environment of \( A \)?
Label Scope Rules Vary

/* In C, labels have entire function as scope */
#include <stdio.h>
main()
{   int i = 3;  int n = 10;  printf("before forward jump i = %d\n", i);
    goto fore;
back:   printf("after back jump i = %d\n", i);
    if (n < 3) {
        int i = 7;  int j = 13;
        fore:   i = i + 1;
            printf("after forward jump i = %d\n", i);
            printf("after forward jump j = %d\n", j);
            goto back;
    }
else   {   int i = 99;  printf("after else i = %d\n", i);
    }
    printf("before return i = %d\n", i);
}
Label Scope Rules (cont.)

opu> cc labels.c
opu> a.out
before forward jump i = 3
after forward jump i = 1
after forward jump j = 0
after back jump i = 3
after else i = 99
before return i = 3
opu>
Returned Subroutines

```c
main()
{
    int(int)  makemult(int n)
    {
        int t(int x){ return n*x; };return t;
    }
    int(int)  f;
    int  y;
    f = makemult(3);
    y = f(2);
}
```
Returned Subroutines (cont.)

- Before call to `makemult(3)`
  - null pointer = `null`
  - frame pointer = `fp`

- At prologue of `makemult(3)`

- At return from `makemult(3)`
Returned Subroutines (cont.)

- After assignment \( f = \ldots \)

- At prologue of call \( f(2) \)

Note: static and dynamic links differ.
Returned Subroutines (cont.)

- After assignment $y = f(2)$

The AR for $\text{makemult}(3)$ is never ``popped'' while $\text{main}()$ is active

- $\text{main}$ activation refers to a function value $f$
- functional value $f$ requires definition of $n$ in $\text{makemult}$ AR
- function $f$ in environment can be called again many times
  - So lifetime of $\text{makemult}$ AR is lifetime of $\text{main}$

- ARs now managed on a heap, along with closures