

Implementation of Languages

SIMPLESEM

specify semantics in an abstract assembler language

static structure: data memory D, code memory C, instruction pointer ip

instructions:

- set target, source
- set 10, D[20]
- set 15, read
- set write, D[50]
- set 99, $D[15] + D[33 * C[41]] - 8$
- set D[10], D[20]
- jump 47
- jumpt 47, $D[3] > D[8]$
- jump D[13]
- halt

Language with Simple Statements

```

main()
{
    int i, j;           code memory           data memory
                        +-----+           +-----+
    get(i, j);          |0  set 0, read         |0  cell for i
                        |1  set 1, read         |1  cell for j
    while (i != j)     |2  jumpt 8, D[0] = D[1]
        if (i > j)     |3  jumpt 6, D[0] <= D[1]
            i -= j;    |4  set 0, D[0] - D[1]
        else           |5  jump 7
            j -= i;    |6  set 1, D[1] - D[0]
                        |7  jump 2
    print(i);          |8  set write, D[0]
}                      |9  halt

```

Simple Routines

```
int i=1, j=2, k=3;
```

```
alpha()
```

```
{  
    int i=4, l=5;  
    ...  
    i += k + l;  
    ...  
};
```

```
beta()
```

```
{  
    int k=6;  
    ...  
    i = j + k;  
    alpha();  
    ...  
};
```

```
main()
```

```
{  
    ...  
    beta();  
    ...  
}
```

Memory Layout (Simple Routines)

code memory		data memory
	...	0 8 //i global
014	set 6, 16	1 2 //j
015	jump 100	2 3 //k
	...	3 125 //ret. alpha
049	halt	4 12 //i
	...	5 5 //l
058	set 4, D[4]+D[2]+D[5]	6 16 //ret. beta
	...	7 6 //k
099	jump D[3]	...
	...	
122	set 0, D[1]+D[7]	
123	set 3, 125	
124	jump 50	ip = 59
	...	
149	jump D[6]	
	...	
	// ...	

Separate Compilation

```
/* file 1 */
```

```
int i=1, j=2, k=3;
extern beta();
main()
{
    ...
    beta();
    ...
}
```

```
/* file 2 */
```

```
extern int k;
alpha()
{
    int i=4, l=5;
    ...
    i += k + l;
    ...
}
```

```
/* file 3 */
```

```
extern int i,j;
extern alpha();
beta()
{
    int k = 6;
    ...
    i = j + k;
    alpha();
    ...
}
```

Recursion and Result Values

```
int n;
int fact()
{
    int loc;
    if (n > 1) {
        loc = n--;
        return loc * fact();
    }
    else
        return 1;
}

main()
{
    get(n);
    if (n >= 0)
        print(fact());
    else
        print("input error");
}
```

Stack with Dynamic Link

data memory

```
+-----+
|0  B    //CURRENT
|1  C    //FREE
|   ...           stack grows downwards
|                   holds activation records
|
|A  ... //return pointer (caller's activation record)
|   ... //dynamic link
|   ... //local variables
|
|B  ... //return pointer (current activation record)
|   A   //dynamic link
|   ... //local variables
|
|C  ... //free memory
|
```


Invocation and Return

set 1, D[1]+1	allocate memory for result
set D[1], ip+4	set return pointer
set D[1]+1, D[0]	set dynamic link
set 0, D[1]	set CURRENT
set 1, D[1]+AR	set FREE
jump addr	jump to routine
set 1, D[0]	set FREE
set 0, D[D[0]+1]	set CURRENT
jump D[D[1]]	jump to stored return pointer

Nested Blocks

```

int f() {
    int x,y,w;          //1
    while(...) {
        int x,z;       //2
        while(...) {
            int y; ... //3
        }
        if(...) {
            int x,w; ... //4
        }
    }
    if(...) {
        int a,b,c,d; //5
        ...
    }
}

```

activation record of f

```

+-----+
| return pointer
| dynamic link
| x in //1
| y in //1
| w in //1
| x in //2, a in //5
| z in //2, b in //5
| y in //3, x in //4, c in //5
| w in //4, d in //5
+-----+

```

Nested Routines and Static Link

<pre> int x,y,z; void f1() { int t,u; void f2() { int x,w; void f3() { int y,w,t; ... } ... } ... } </pre>	<pre> sketch of run-time stack +----- A global environment (x,y,z), no dyn.link, stat.link needed B act.record for f1 (t,u) with dyn.link: A, stat.link: A C act.record for f2 (x,w) with dyn.link: B, stat.link: B D act.record for f3 (y,w,t) with dyn.link: C, stat.link: C E act.record for f2 (x,w) with dyn.link: D, stat.link: B !! </pre>
--	--

Computing Frame Pointer

each variable statically bound to a pair $\langle d, o \rangle$ where

d is the **distance** between current and addressed static scope

e.g., $d=2$ when accessing u (defined in f_1) within f_3 ,

o is the **offset** of the variable within the activation record

computing the **frame pointer**:

$$fp(d) = \text{if } d=0 \text{ then } D[0] \text{ else } D[fp(d-1)+stat.link.offset]$$

examples: $fp(0) = D[0]$,

$fp(1) = D[D[0]+stat.link.offset]$,

$fp(2) = D[D[D[0]+stat.link.offset]+stat.link.offset]$

adresse of variable bound to $\langle d, o \rangle$ is $D[fp(d)+o]$

Routine Invocation with Static Link

set 1, D[1]+1	allocate memory for result
set D[1], ip+5	set return pointer
set D[1]+1, D[0]	set dynamic link
set D[1]+2, fp(d)	set static link (stat.link.offset = 2)
set 0, D[1]	set CURRENT
set 1, D[1]+AR	set FREE
jump addr	jump to routine

Dynamic Array

example in Ada: `type V is array (Integer range <>) of Integer;`
`A: V(1..N);` -- N not statically known

compiler allocates in activation record space for **descriptor**
(holds index range and pointer to memory of array)

when executing the array declaration at run-time:

extend current activation record with space for array,
initialize descriptor appropriately

example for access `A(I) := 0;`

`set[D[D[0]+m] + D[D[0]+s]], 0`

(m = offset of descriptor of A; s = offset of local variable I)

Dynamic Scoping

<code>void sub2() {</code>	sketch of run-time stack
<code>declare x;</code>	+-----
<code>...</code>	A link = none (act.record for main)
<code>}</code>	x = ...
<code>void sub1() {</code>	y = ...
<code>declare y;</code>	z = ...
<code>sub2();</code>	B link = A (act.record for sub1)
<code>...</code>	y = ...
<code>}</code>	C link = B (act.record for sub2)
<code>void main() {</code>	z = ...
<code>declare x,y,z;</code>	
<code>sub1();</code>	
<code>sub2();</code>	dynamic search for name along link
<code>}</code>	

Heap

local data on stack are lost when returning from a routine invocation

non-local data needed after return must be allocated on the **heap**

data memory

```
+-----+
| stack: grows downwards,
|         holds activation records removed on return
|
|
| heap:  grows upwards,
|         holds data alive until program termination
+-----+
```


Parameter Passing

call by reference: pass l-value of actual parameter to callee

call by copy: parameter is local variable of callee

call by value: copy r-value into variable of callee

call by result: copy result back into variable of caller

call by value-result: call by value on invocation
and call by result on return (value copied twice)

call by name: substitute actual parameter
for each occurrence of formal parameter in callee

Implementation of Call by Reference

formal parameter is reference to actual parameter in callee's activation record
(offset off)

pass l-value if actual parameter is a variable $\langle d, o \rangle$ in caller:

set $D[0]+off, fp(d)+o$

pass r-value if actual parameter is call-by-reference parameter $\langle d, o \rangle$ in caller:

set $D[0]+off, D[fp(d)+o]$

assignment to a call-by-reference parameter: $x = 0;$

set $D[D[0]+off], 0$

Call by Reference vs. Value Result

these calling conventions differ if

two formal parameters are aliases

example: actual parameters: $a[i]$ und $a[j]$
formal parameters: x und y
body of routine: $x = 0; y++;$

a formal parameter and a variable visible to both (caller and callee) are alias

example: actual parameter: a
formal parameter: x
body of routine: $a = 1; x = x + a;$

Call by Name

```
swap(int a, b) {  
    int temp = a;  
    a = b;  
    b = temp;  
};  
...  
swap(i, a[i]);  
...  
  
temp = i;  
i = a[i];  
a[i] = temp;  
  
| int c; /* global variable */  
| swap(int a, b) {  
|     int temp = a; a = b;  
|     b = temp; c++;  
| };  
| y() {  
|     int c, d;  
|     swap(c, d);  
| };  
  
| temp = c; c = d;  
| d = temp;  
| c++;           !!
```

Routine as Parameter

```

1  int u, v;      |   7  void b(routine x) |  19  void main()
2  void a()      |   8  {                 |  20  {
3  {             |   9      int u, v, y;  |  21      b(a);
4      int y;    |  10      void c() {   |  22  };
5      ...      |  11          ...     |
6  };           |  12          y = ...; |
              |  13          ...     |
              |  14      };         |
              |  15      x();       |
              |  16      b(c);     |
              |  17      ...       |
              |  18  }             |

```

routine parameter must include static link