

Location for Activation Record

- Depending upon language, activation record can be created in the static, stack or heap area
- Creation in Static Area:
 - Early languages, like FORTRAN
 - Address of all arguments, local variables etc. are preset at compile time itself
 - To pass parameters, values are copied into these locations at the time of invoking the procedure and copied back on return
 - There can be a single activation of a procedure at a time
 - Recursive procedures cannot be implemented



Location for Activation Record

- Creation in Stack Area:
 - Used for languages like C, Pascal, Java etc.
 - As and when a procedure is invoked, corresponding activation record is pushed onto the stack
 - On return, entry is popped out
 - Works well if local variables are not needed beyond the procedure body
- For languages like LISP, in which a full function may be returned, activation record created in the heap



Processor Registers

- Also a part of the runtime environment
- Used to store temporaries, local variables, global variables and some special information
- Program counter points to the statement to be executed next
- Stack pointer points to the top of the stack
- Frame pointer points to the current activation record
- Argument pointer points to the area of the activation record reserved for arguments

Environment Types

- Stack based environment without local procedures – common for languages like C
- Stack based environment with local procedures – followed for block structured languages like Pascal



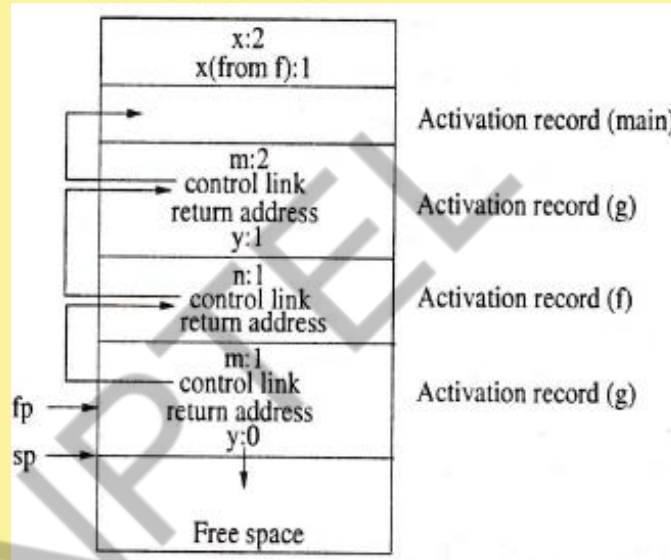
Environment without Local Procedures

- For languages where all procedures are global
- Stack based environment needs two things about activation records
 - Frame pointer: Pointer to the current activation record to allow access to local variables and parameters
 - Control link / Dynamic link: Kept in current activation record to record position of the immediately preceding activation record



Environment without Local Procedures

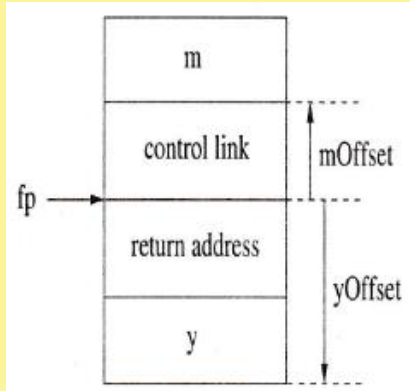
```
int x = 2;
void f( int n ) {
    static int x = 1;
    g(n);
    x--;
}
void g( int m ) {
    int y = m - 1;
    if ( y > 0 ) {
        f(y);
        x--;
    }
}
main() {
    g(x); return 0;
}
```



Snapshot of program execution after main has called g, g has called f and f has in turn called g

Accessing Variables

- Parameters and local variables found by offset from the current frame pointer
- Offsets can be calculated statically by the compiler
- Consider procedure g with parameter m and local variable y



$mOffset = \text{size of control link} = +4 \text{ bytes}$

$yOffset = -(\text{size of } y + \text{size of return address}) = -6$

Hence, m and y can be accessed by $4(fp)$ and $-6(fp)$

Activation Record Creation

At a call	
Caller	Callee
<ol style="list-style-type: none"> 1. Allocate basic frame 2. Store parameters 3. Store return address 4. Save caller-saved registers 5. Store self frame pointer 6. Set frame pointer for child 7. Jump to child 	<ol style="list-style-type: none"> 1. Save callee saved registers, state 2. Extend frame for locals 3. Initialize locals 4. Fall through to code
At a return	
Caller	Callee
<ol style="list-style-type: none"> 1. Copy return value 2. Deallocate basic frame 3. Restore caller-saved registers 	<ol style="list-style-type: none"> 1. Store return value 2. Restore callee-saved registers, state 3. Unextend frame 4. Restore parent's frame pointer 5. Jump to return address

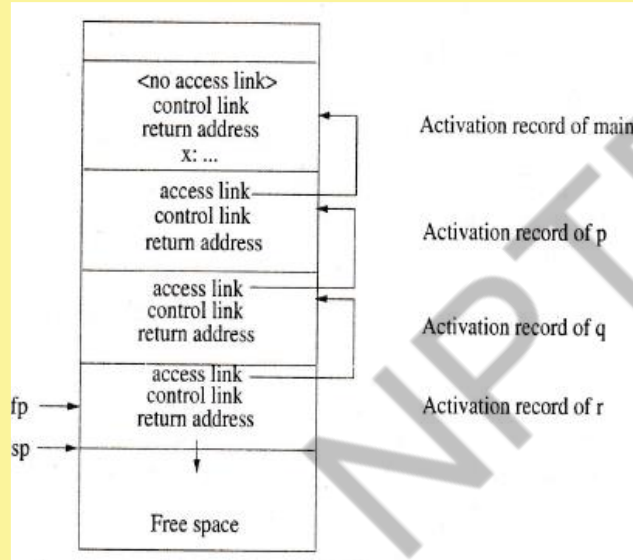
Environment with Local Procedures

- For supporting local procedures, variables may have various scopes
- To determine the definition to be used for a reference to a variable, it is needed to access non-local, non-global variables
- These definitions are local to one of the procedures nesting the current one – need to look into the activation records of nesting procedures
- Solution is to keep extra bookkeeping information, called *access link*, pointing to the activation record for the defining environment of a procedure



Environment with Local Procedures

```
program chaining;  
procedure p;  
var x: integer;  
procedure q;  
  procedure r  
  begin  
    x := 2;  
    ...  
    if ... Then p;  
  end {of r}  
begin  
  r;  
end {of q}  
begin  
  q;  
end {of p}  
begin {of main}  
  p;  
end.
```



- Current procedure r
- To locate definition of x, it has to traverse through the activation records using access links
- When the required procedure containing definition of x is reached, it is accessed via offset from the corresponding frame pointer

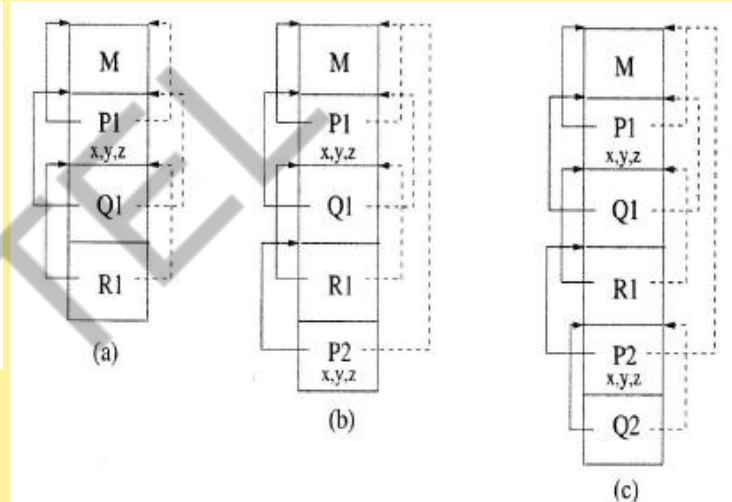
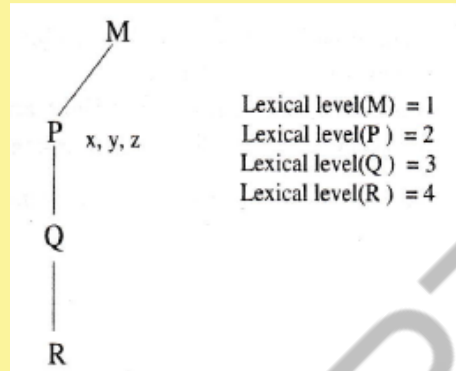
Compiler's Responsibility

- Proper code to access the correct definitions:
 - Find difference d between the lexical nesting level of declaration of the name and the lexical nesting level of the procedure referring to it
 - Generate code for following d access links to reach the right activation record
 - Generate code to access the variable through offset mechanism



Example

```
program M;  
  procedure P;  
    var x, y, z;  
    procedure Q;  
      procedure R;  
        begin  
          ... z = P; ...  
        end R;  
      begin  
        ... y = R; ...  
      end Q;  
    begin  
      ... x = Q; ...  
    end P;  
  begin  
    ... P; ...  
  end M;
```



DISPLAY

- Difficulty in non-local definitions is to search by following access links
- Particularly for virtual paging environment, certain portion of the stack containing activation records may be swapped out, access may be very slow
- To access variables without search, *display* is used



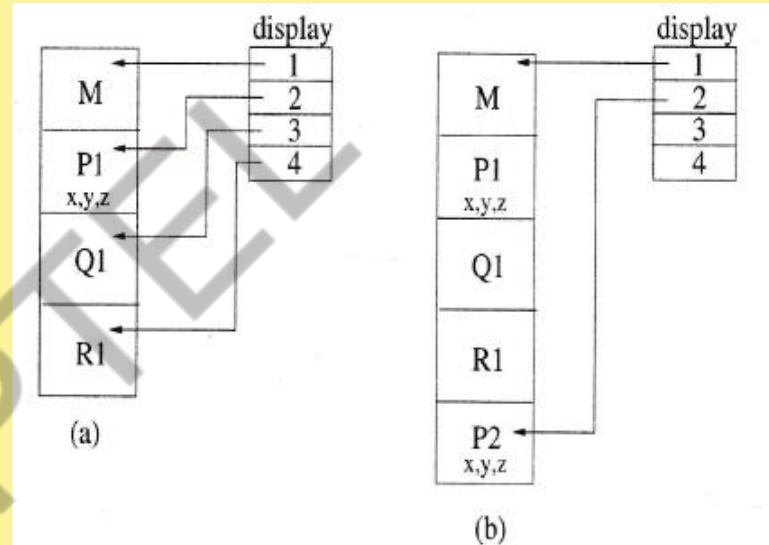
Display

- Display d is a global array of pointers to activation records, indexed by the lexical nesting depth
- Element $d[i]$ points to the most recent activation of the block at nesting depth i
- A nonlocal X is found as follows:
 - If the most closely nested declaration of X is at nesting depth i , then $d[i]$ points to activation record containing the location for X
 - Use relative address within the activation record to access X



Example

- Maximum nesting depth 4, so 4 entries in the display
- In Fig (a), M has called P, P has called Q and Q has in turn called R
- Compiler knows that x is in procedure P at lexical level 2
- Code is generated to access second entry of the display to reach the activation record of P directly
- Same is in Fig (b)



Maintaining Display

- When a procedure P at nesting depth i is called, following actions are taken:
 - Save value of $d[i]$ in the activation record for P
 - Set $d[i]$ to point to new activation record
- When a procedure P finishes:
 - $d[i]$ is reset to the value stored in the activation record of P



Example

1. Program X
2. var x, y, z;
3. Procedure P
4. var a;
5. begin (of P)
6. a = Q
7. end (of P)
8. Procedure Q
9. Procedure R
10. begin (of R)
11. P
12. end (of R)
13. begin (of Q)
14. R
15. end (of Q)
16. begin (of X)
17. P
18. Q
19. end (of X)

- Show the snapshots of the stack of activation records at the time of executing line nos. 6, 11, 14, 17, 18
- Show the corresponding displays

Conclusion

- Data structure activation record contains necessary information to control program execution
- Compiler writer must generate appropriate code for operations
- A small array, display, helps in the process
- Display management becomes a part of compiler's responsibility





NPTEL ONLINE CERTIFICATION COURSES

**Thank
you!**

Compiler Design

Intermediate Code Generation

Santanu Chattopadhyay

Electronics and Electrical Communication Engineering



- ☐ Intermediate Languages
- ☐ Intermediate Language Design Issues
- ☐ Intermediate Representation Techniques
- ☐ Statements in Three-Address Code
- ☐ Implementation of Three-Address Instructions
- ☐ Three-Address Code Generation
- ☐ Conclusion

Intermediate Code

- Compilers are designed to produce a representation of input program in some hypothetical language or data structure
- Representations between the source language and the target machine language programs
- Offers several advantages
 - Closer to target machine, hence easier to generate code from
 - More or less machine independent, makes it easier to retarget the compiler to various different target processors
 - Allows variety of machine-independent optimizations
 - Can be implemented via syntax-directed translation, can be folded into parsing by augmenting the parser



Intermediate Languages

- Can be classified into – High-level representation and Low-level representation

High-level Representation

- Closer to source language program
- Easy to generate from input program
- Code optimization difficult, since input program is not broken down sufficiently

Low-level Representation

- Closer to target machine
- Easy to generate final code from
- Good amount of effort in generation from the source code



Intermediate Language Design Issues

- Set of operators in intermediate language must be rich enough to allow the source language to be implemented
- A small set of operations in the intermediate language makes it easy to retarget
- Intermediate code operations that are closely tied to a particular machine or architecture can make it harder to port
- A small set of intermediate code operations may lead to long instruction sequences for some source language constructs. Implies more work during optimization

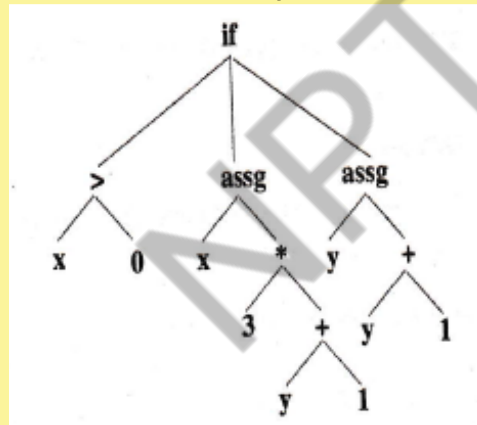


Intermediate Representation Techniques

- High-level Representation
 - Abstract Syntax Trees
 - Directed Acyclic Graphs
 - P-code
- Low-level Representation

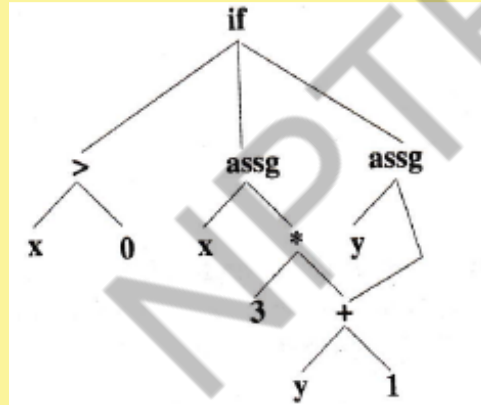
Abstract Syntax Tree

- Compact form of parse tree
- Represents hierarchical structure of a program
- Nodes represent operators, children of a node the operands
- Example: “if $x > 0$ then $x = 3 * (y + 1)$ else $y = y + 1$ ”



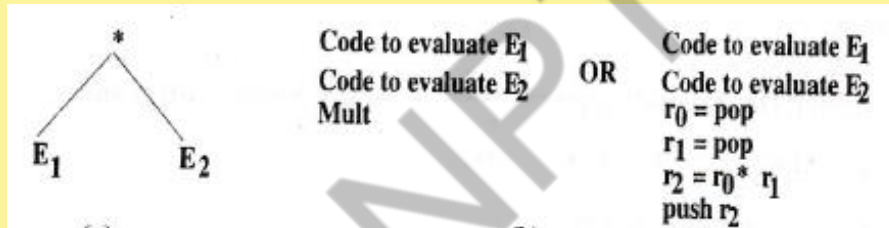
Directed Acyclic Graph (DAG)

- Similar to syntax tree
- Common subexpressions represented by single node



P-code

- Used for stack based virtual machines
- Operands are always found on the top of the stack
- May need to push operands to the stack first
- Syntax tree to P-code:



Low-level Representation – Three Address Code

- Sequence of instructions of the form “ $x = y \text{ op } z$ ”
- Only one operator permitted in the right hand side
- Due to its simplicity, offers better flexibility in terms of target code generation and code optimization

$x = y * z + w * a$

$t_1 = y * z$
 $t_2 = w * a$
 $x = t_1 + t_2$

Statements in Three-Address Code

- Intermediate languages usually have the following types of statements
 - Assignment
 - Jumps
 - Address and Pointer Assignments
 - Procedure Call/Return
 - Miscellaneous

Assignment Statement

- Three types of assignment statements
 - $x = y \text{ op } z$, op being a binary operator
 - $x = \text{op } y$, op being a unary operator
 - $x = y$
- For all operators in the source language, there should be a counterpart in the intermediate language



Jump Statement

- Both conditional and unconditional jumps are required
 - goto L, L being a label
 - if x relop y goto L

Indexed Assignment

- Only one-dimensional arrays need to be supported
- Arrays of higher dimensions are converted to one-dimensional arrays
- Statements to be supported
 - $x = y[i]$
 - $x[i] = y$

Address and Pointer Assignments

- Statements required are of following types
 - $x = \&y$, address of y assigned to x
 - $x = *y$, content of location pointed to by y is assigned to x
 - $x = y$, simple pointer assignment, where x and y are pointer variables



Procedure Call/Return

- A call to the procedure $P(x_1, x_2, \dots, x_n)$ is converted as
param x1
param x2
...
param xn
- A procedure is implemented using the following statements
enter f, Setup and initialization
leave f, Cleanup actions (if any)
return
return x
retrieve x, Save returned value in x

