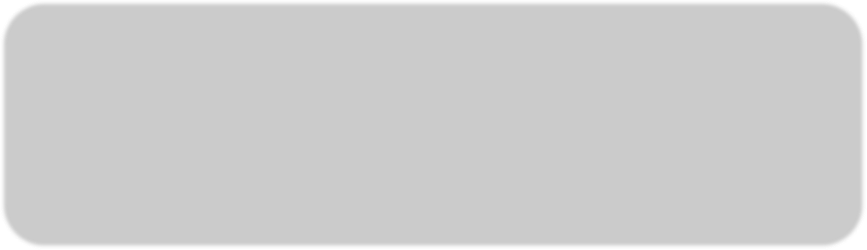
**Compiler Principles**

**Run-Time Environments**



Source

Language

**Analysis**

Intermediate Language

Symbol Table

Code Optimizer

Code Generator

Lexical Analyzer Syntax Analyzer Semantic Analyzer Intermediate Code Generator

**Synthesis**

Target

Language

### Before discussing code generation, we need to understand what we are trying to generate.

* There are a number of standard techniques for structuring executable code that are widely used.
* **Run-time environment** is created by the compiler to support target programs.

##### allocation of storage locations for the objects

* + mechanisms to access variables
  + linkages between procedures
  + mechanisms for passing parameters
  + interfaces to the operating system
  + input/output devices

– …

low end

|  |
| --- |
| Code |
| Static Data |
| Stack |
|  |
| Heap |

high end

Memory locations for code are

determined at compile time.

Locations of static data can also be

determined at compile time.

Data objects allocated at run-time. (Activation Records)

Other dynamically allocated data objects at run-time. (For example, malloc area in C).

* An execution of a procedure starts at the beginning of the procedure body;
* When the procedure is completed, it returns the control to the point immediately after the place where that procedure is called.
* Each execution of a procedure is called as its **activation**.
* **Lifetime** of an activation of a procedure is the sequence of the steps between the first and the last steps in the execution of that procedure (including the other procedures called by that procedure).
* If a and b are procedure activations, then their lifetimes are either non-overlapping or are nested. (single thread)
* If a procedure is recursive, a new activation can begin before

an earlier activation of the same procedure has ended.

### We can use a tree (called **activation tree**) to show the way control enters and leaves activations.

* In an activation tree:
  + Each node represents an activation of a procedure.
  + The root represents the activation of the main program.
  + The node *a* is a parent of the node *b* iff the control flows from *a* to *b*.
  + The node *a* is left to the node *b* iff the lifetime of *a*

occurs before the lifetime of *b*.

##### program main; procedure s;

begin ... end; procedure p;

procedure q; begin ... end;

begin q; s; end; begin p; s; end;

enter main

enter p

enter q exit q enter s exit s

exit p enter s exit s

exit main

**A Nested Structure**

Sequence of procedure is depth-first traversal of the activation tree

### main

p s

Now control is in s; what procedures are live? What is their order?



q s

### Activation Tree

* The activation tree depends on run-time behavior.
* The activation tree may be different for every program input.

why?

### Since activations are properly nested, a can track currently active procedures.

stack

What data structure?

* The flow of the control in a program corresponds to a depth-first traversal of the activation tree that:
  + starts at the root,
  + visits a node before its children, and
  + recursively visits children at each node in a left-to-right order.
* A stack (called **control stack**) can be used to keep track of live procedure activations.
  + An activation record is pushed onto the control stack as the activation starts.
  + That activation record is popped when that activation ends.
* When node *n* is at the top of the control stack, the stack contains the nodes along the path from *n* to the root.
* Information needed by a single execution of a procedure is managed using a contiguous block of storage called **activation record**.
* An activation record is allocated when a procedure is entered, and it is de-allocated when that procedure exits.
* Size of each field can be determined at compile time (Although actual location of the activation record is determined at run-time).
  + Except that if the procedure has a local variable and its size depends on a parameter, its size is determined at the run time.

# Activation Records Cont’d

store the returned values to the calling procedure. In practice, we may use a machine register for the return value.

|  |
| --- |
| return value |
| actual parameters |
| control link |
| access link |
| saved machine status |
| local data |
| temporaries |

used by the calling procedure to supply parameters to the called procedure

points to the activation record of the caller

used to refer nonlocal data, e.g., in other activation records

holds information about the state of the machine

before the procedure is called.

holds data that local to an execution of a procedure

temporary variables

# Activation Records Example

##### stack

|  |
| --- |
| main |
| p  a: |
| q  b: |

program main; procedure p; var a:real; procedure q;

var b:integer; begin ... end; begin q; end; procedure s; var c:integer; begin ... end; begin p; s; end;

main

p s

q

**Activation Records for Recursive Procedures**

program main; procedure p;

function q(a:integer):integer; begin

if (a=1) then q:=1; else q:=a+q(a-1); end;

begin q(3); end; begin p; end;

main p q(3)

q(2)

|  |
| --- |
| main |
| p |
| q(3)  a:3 |
| q(2)  a:2 |
| q(1)  a:1 |

q(1)

#### The following function computes Fibonacci numbers recursively. Assume that the initial call is f(4).

**function f(n:integer):integer;**

**var s:integer; var t:integer; begin**

**if (n<2) then f:=1;**

**else**

**begin**

**s:=f(n-1);**

**t:=f(n-2);**

**f:=s+t;**

**end;**

1. Draw the complete activation tree.
2. Draw the activation records in the stack the first time f(0) is about to return?

**end;**

a)

f(3)

f(4)

b)

f(2)

|  |
| --- |
| f(4)  n: 4  s:  t: |
| f(3)  n: 3  s:  t: |
| f(2)  n: 2  s: 1  t: |
| f(0)  n: 0  s:  t: |

f(2)

f(1) f(0)

f(1)

f(1) f(0)

* + Who allocates an activation record of a procedure?
    - Some part is created by the caller of that procedure before that procedure is entered.
    - Some part of the activation record of a procedure is created by that procedure itself immediately after that procedure is entered.

### Who de-allocates?

* + - Callee de-allocates the part allocated by Callee.
    - Caller de-allocates the part allocated by Caller.

**Creation of An Activation Record Cont’d**



Caller’s Activation Record

Caller’s Responsibility

|  |
| --- |
| return value |
| actual parameters |
| control link |
| access link |
| saved machine status |
| local data |
| temporaries |
| return value |
| actual parameters |
| control link |
| access link |
| saved machine status |
| local data |
| temporaries |

Callee’s Activation Record

Callee’s Responsibility

|  |
| --- |
| actual parameters |
| return value |
| control link |
| access link |
| saved machine status |
| local data  pointer to a pointer to b |
| temporaries |
| array a array b |

Variable-length data is allocated after temporaries, and there is a link from local data to that array.

#### The same variable name can be used in different parts of the program.

* + The scope rules of the language determine which declaration of a name applies when the name appears in the program.
  + An occurrence of a variable (a name) is:
    - **local**: If that occurrence is in the same procedure in which that name is declared.
    - **non-local**: Otherwise (i.e., it is declared outside of that procedure)

procedure p; var b:real; procedure q;

var a: integer; a is local

begin a := 1; b := 2; end; b is non-local

begin ... end;

### Scope rules of a language determine the treatment of references to nonlocal names.

* + Scope rules:
    - Lexical scope (static scope)
      * Most-closely nested rule is used.
      * Determines the declaration that applies to a name by examining the program text alone at compile- time.
      * Pascal, C, ...
    - Dynamic scope
      * Determines the declaration that applies to a name at run-time.
      * Lisp, APL, ...
  + The scope of a declaration in a block- structured language is given by the *mostly closed rule*.
  + Each procedure (block) will have its own activation record.

###### procedure

* + - begin-end blocks
      * (treated same as procedure without creating most part

of its activation record)

#### A procedure may access to a nonlocal name using:

* + - access links in activation records, or
    - displays (an efficient way to access to nonlocal names)

#### Access links are implemented by adding pointers to each activation record.

###### If procedure p is nested immediately within procedure q in the source code, then the access link in any activation of p points to the most recent activation of q.

* + Access links form a chain from the activation record at the top of the stack to a sequence of activations at lower nesting depths. Along this chain are all the activations whose data and procedures are accessible to the currently executing procedure.

program main; var a:int; procedure p; var d:int;

|  |
| --- |
| main  access link  a: |
| q(1)  access link i,b: |
| q(0)  access link i,b: |
| s access link  c: |
| p access link  d: |

begin a:=1; end; procedure q(i:int); var b:int; procedure s;

var c:int; begin p; end; begin

if (i<>0) then q(i-1) else s;

end;

begin q(1); end;

Access

Links

Bottom

Top

* Access links must be passed with

procedure parameters.

program main;

procedure p(procedure a); begin a; end;

procedure q; procedure s;

begin ... end;

begin p(s) end; begin q; end;

Bottom

Top

|  |
| --- |
| main  access link |
| q access link |
| p  access link a: s |
| s access link |

program main; var a:int; procedure p;

var b:int; procedure q();

var c:int;

begin

c:=a+b; end;

begin q; end; begin p; end;

addrC := offsetC(curr\_pos) pos := traceback(curr\_pos) addrB := offsetB(pos)

pos := traceback(pos) addrA := offsetA(pos) ADD addrA, addrB, addrC

|  |
| --- |
| main  access  link  a: |
| p  access link  b: |
| q  access link  c: |

* An array of pointers to activation records can be used to access

activation records.

* This array is called as displays.
* For each level, there will be an array entry.

Current activation record at level 1 Current activation record at level 2 Current activation record at level 3

|  |
| --- |
| 1: |
| 2: |
| 3: |
|  |

## Accessing Nonlocal Variables using Display

program main; var a:int; procedure p;

var b:int; procedure q();

var c:int;

begin

c:=a+b; end;

begin q; end; begin p; end;

D[1]

D[2]

|  |
| --- |
| main  access  link  a: |
| p  access link  b: |
| q  access link  c: |

D[3]

addrC := offsetC(D[3]) addrB := offsetB(D[2]) addrA := offsetA(D[1]) ADD addrA, addrB, addrC

### A nonlocal name’s binding is not determined lexically, but inherits that of the caller.

* + Example:

int x = 0;

int f() { return x; }

int main() { int x = 1; return f(); }

##### Lexical scope: return 0

* Dynamic scope: return 1
  + Stack-based method:
    - traverses the runtime stack, checking each activation

record for the first value of the identifier

##### Table-based method:

* + - uses a table to associate each name with its current meaning
    - when a procedure is activated, locate each local name to the entry in the table and store the name’s previous meaning in the activation record

### Bad program readability, understanding of the program relies on a simulation of the it

* + Not suitable for static type checking
  + Therefore, dynamic scope is not commonly used
  + A value that outlives the procedure that creates it cannot be kept in the activation record

##### Example: int\* func() { return new int[10]; }

* + Languages with dynamically allocated data use a heap to store dynamic data
* The heap is an area of memory which is dynamically allocated.
* Like a stack, it may grow and shrink during runtime.
* Unlike a stack it is not a LIFO

more complicated to manage

* In a typical programming language implementation we will have both heap and stack allocated memory coexisting.

Let both stack and heap share the same memory area, but grow towards each other from opposite ends!

SB



Stack memory area

ST Stack

grows downward

Heap can expand upward

HT

Heap memory area

HB

### Allocation: When a program requests memory for a variable or object, the memory manager produces a chunk of contiguous heap memory of the requested size

* + Deallocation: The memory manager returns de-allocated space to the pool of free space
  + Problem: Heap is fragmented into too many small and noncontiguous chunks
  + Best-fit algorithm: allocates the requested memory in the smallest available chunk that is large enough.
  + First-fit algorithm: places an object in the first (lowest-address) chunk in which it fits

##### takes less time than best-fit to place objects

* + - overall performance is worse than best-fit

**Stack** is LIFO allocation => ST moves up/down everything above ST is in use/allocated. Below is free memory. This is easy! But …

**Heap** is not LIFO, how to manage free space in the “middle” of the heap?

SB

HT

Allocated

ST reuse?

Free

Free

Mixed: Allocated and

Free

HB

How to manage free space in the “middle” of the heap?

=> keep track of free blocks in a data structure: the “free list”.

For example we could use a linked list pointing to free blocks.

freelist

HT

Next

Free

**But where do we store this data structure?**

Free

Next

HB

Next

Free

Where do we find the memory to store a freelist data structure?

=> Since the free blocks are not used by the program,

memory manager can use them for storing the freelist itself.

HT

|  |
| --- |
|  |
| *free block size* |
| *next free* |
|  |
|  |

HF

HF

HB