Compilers

Syntax

CodeGen

Runtime Organization

Semantics

Types
• Management of run-time resources

• Correspondence between
  - static (compile-time) and
  - dynamic (run-time) structures

• Storage organization
Execution of a program is initially under the control of the operating system.

When a program is invoked:
- The OS allocates space for the program.
- The code is loaded into part of the space.
- The OS jumps to the entry point (i.e., “main”).
Runtime Organization

Memory

Low Address

Code

Other Space

High Address
• By tradition, pictures of machine organization have:
  - Low address at the top
  - High address at the bottom
  - Lines delimiting areas for different kinds of data

• These pictures are simplifications
  - E.g., not all memory need be contiguous
• **Other Space = Data Space**

• **Compiler is responsible for:**
  - Generating code
  - Orchestrating use of the data area
Compilers

Activations
• Two goals:
  - Correctness
  - Speed

• Complications in code generation come from trying to be fast as well as correct
Two assumptions:

1. Execution is sequential; control moves from one point in a program to another in a well-defined order
   - No concurrency
2. When a procedure is called, control always returns to the point immediately after the call
   - No exceptions
• An invocation of procedure $P$ is an activation of $P$

• The *lifetime* of an activation of $P$ is
  - All the steps to execute $P$
  - Including all the steps in procedures $P$ calls
• The *lifetime* of a variable $x$ is the portion of execution in which $x$ is defined

• Note that
  - Lifetime is a dynamic (run-time) concept
  - Scope is a static concept
• **Observation**
  - When \( P \) calls \( Q \), then \( Q \) returns before \( P \) returns

• **Lifetimes of procedure activations are properly nested**

• **Activation lifetimes can be depicted as a tree**
Class Main {
    g(): Int { 1 };
    f(): Int { g() };
    main(): Int {{ g(); f(); }};
}
Class Main {
  g(): Int { 1 };  
  f(x: Int): Int { if x = 0 then g() else f(x - 1) fi };  
  main(): Int {{f(3); }};
}

Activations

Main
  ↓
  f
  ↓
  f
  ↓
  f
  ↓
  f
  ↓
g
The `powerOfTwo()` function, shown to the right, returns true if its argument is a power of two, false otherwise. What is the activation tree for `powerOfTwo(4)`?

```
isEven(x:Int) : Bool { x % 2 == 0 };
isOne(x:Int) : Bool { x == 1 };
powerOfTwo(x:Int) : Bool {
    if isEven(x) then powerOfTwo(x / 2)
    else isOne(x)
};
```
The `powerOfTwo()` function, shown to the right, returns true if its argument is a power of two, false otherwise. What is the activation tree for `powerOfTwo(4)`?

```swift
isEven(x: Int) -> Bool { x % 2 == 0 }
isOne(x: Int) -> Bool { x == 1 }
powerOfTwo(x: Int) -> Bool {
    if isEven(x) then
        powerOfTwo(x / 2)
    else
        isOne(x)
}
```
• The activation tree depends on run-time behavior

• The activation tree may be different for every program input

• Since activations are properly nested, a stack can track currently active procedures
Class Main {
    g(): Int { 1 };
    f(): Int { g() };
    main(): Int {{ g(); f(); }};
}

Main Stack

Main
Class Main {
    g() : Int { 1 };
    f(): Int { g() };
    main(): Int {{ g(); f(); }};
}

Main

Stack

Main

g
Class Main {
    g() : Int { 1 };
    f() : Int { g() };
    main() : Int {{ g(); f(); }};
}

Activations

Main

Stack

Main

f
Class Main {
    g() : Int { 1 };
    f(): Int { g() };
    main(): Int {{ g(); f(); }};
}

Main
  g      f
  \downarrow  
  \downarrow  
  g

Stack
    Main
        f
        g
• The information needed to manage one procedure activation is called an *activation record (AR)* or *frame*.

• If procedure F calls G, then G’s activation record contains a mix of info about F and G.
F is “suspended” until G completes, at which point F resumes

G’s AR contains information needed to
- Complete execution of G
- Resume execution of F
Activation Records

The contents of a typical AR for G

- Space for G’s return value
- Actual parameters
- Pointer to the previous activation record
  - The control link; points to AR of caller of G
- Machine status prior to calling G
  - Contents of registers and program counter
  - Local variables
- Other temporary values
Class Main {
    g() : Int { 1};
    f(x:Int):Int {if x=0 then g() else f(x - 1)(**)fi};
    main(): Int {{f(3);(*)}};
};

AR for f:

<table>
<thead>
<tr>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>argument</td>
</tr>
<tr>
<td>control link</td>
</tr>
<tr>
<td>return address</td>
</tr>
</tbody>
</table>
Stack after two calls to `f`
• **Main** has no argument or local variables and its result is never used; its AR is uninteresting

• (*) and (***) are return addresses of the invocations of f
  - The return address is where execution resumes after a procedure call finishes

• This is only one of many possible AR designs
  - Would also work for C, Pascal, FORTRAN, etc.
• The compiler must determine, at compile-time, the layout of activation records and generate code that correctly accesses locations in the activation records.

• Thus, the AR layout and the code generator must be designed together!
The picture shows the state after the call to the 2nd invocation of $f$ returns
The advantage of placing the return value 1st in a frame is that the caller can find it at a fixed offset from its own frame.

There is nothing magic about this organization:
- Can rearrange order of frame elements
- Can divide caller/callee responsibilities differently
- An organization is better if it improves execution speed or simplifies code generation
• Real compilers hold as much of the frame as possible in registers
  - Especially the method result and arguments
Compilers

Globals & Heap
• All references to a global variable point to the same object
  - Can’t store a global in an activation record

• Globals are assigned a fixed address once
  - Variables with fixed address are “statically allocated”

• Depending on the language, there may be other statically allocated values
 Globals & Heap

Memory

Code

Static Data

Stack

Low Address

High Address

Alex Aiken
• A value that outlives the procedure that creates it cannot be kept in the AR

```java
method foo() { new Bar }
```

The Bar value must survive deallocation of foo’s AR

• Languages with dynamically allocated data use a heap to store dynamic data
• The code area contains object code
  - For many languages, fixed size and read only
• The static area contains data (not code) with fixed addresses (e.g., global data)
  - Fixed size, may be readable or writable
• The stack contains an AR for each currently active procedure
  - Each AR usually fixed size, contains locals
• Heap contains all other data
  - In C, heap is managed by malloc and free
• Both the heap and the stack grow

• Must take care that they don’t grow into each other

• Solution: start heap and stack at opposite ends of memory and let them grow towards each other
• Low-level details of machine architecture are important in laying out data for correct code and maximum performance

• Chief among these concerns is *alignment*
• Most modern machines are (still) 32 or 64 bit
  - 8 bits in a byte
  - 4 or 8 bytes in a word
  - Machines are either byte or word addressable
Data is *word aligned* if it begins at a word boundary.

"AB" is not word aligned.

Most machines have some alignment restrictions:
- Or performance penalties for poor alignment
  - Sometimes 10 times slower
• Example: A string

"Hello"

Takes 5 characters (without a terminating \0)

• To word align next word, add 3 “padding” characters to the string

• The padding is not part of the string, it’s just unused memory
Compilers
Runtime Organization (cont.)
Scope Rules

Variable $x$ locally declared in $p$

A function $x$

program prg;
  var y : real;
  function x(a : real) : real;
  begin ... end;
procedure p;
  var x : integer;
  begin
    x := 1;
    ...
    end;
begin
  y := x(0.0);
  ...
end.
Access to Nonlocal Names

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

  - **Lexical Scope (Static Scope)**
    
    Determines the declaration that applies to a name by examining the program text alone at compile-time.
    
    Most-closely nested rule is used.
    
    Pascal, C, ..

  - **Dynamic Scope**
    
    Determines the declaration that applies to a name at run-time.
    
    Lisp, APL, ...
Mapping Names to Values

```
var i;
...
i := 0;
...
i := i + 1;
```
Mapping Names to Values

At compile time

- `environment`
  - `name`
  - `storage`

At run time

- `state`
  - `storage`
  - `value`

---

```
var i;
...
i := 0;
...
i := i + 1;
```
The returned value of the called procedure is returned in this field to the calling procedure. In practice, we may use a machine register for the return value.

The field for actual parameters is used by the calling procedure to supply parameters to the called procedure.

The optional control link points to the activation record of the caller.

The optional access link is used to refer to nonlocal data held in other activation records.

The field for saved machine status holds information about the state of the machine before the procedure is called.

The field of local data holds data that local to an execution of a procedure.

Temporary variables is stored in the field of temporaries.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>return value</td>
<td>The returned value of the called procedure is returned in this field.</td>
</tr>
<tr>
<td>actual parameters</td>
<td>The field for actual parameters is used by the calling procedure.</td>
</tr>
<tr>
<td>optional control link</td>
<td>The optional control link points to the activation record of the caller.</td>
</tr>
<tr>
<td>optional access link</td>
<td>The optional access link is used to refer to nonlocal data.</td>
</tr>
<tr>
<td>saved machine status</td>
<td>The field for saved machine status holds information about the machine state.</td>
</tr>
<tr>
<td>local data</td>
<td>The field of local data holds data that local to an execution of a procedure.</td>
</tr>
<tr>
<td>temporaries</td>
<td>Temporary variables is stored in this field.</td>
</tr>
</tbody>
</table>
Activation records (subroutine frames) on the run-time stack hold the state of a subroutine

Calling sequences are code statements to create activations records on the stack and enter data in them

- Caller’s calling sequence enters actual arguments, control link, access link, and saved machine state
- Callee’s calling sequence initializes local data
- Callee’s return sequence enters return value
- Caller’s return sequence removes activation record
### Activation Records (Subroutine Frames)

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Returned value</td>
<td></td>
</tr>
<tr>
<td>Actual parameters</td>
<td></td>
</tr>
<tr>
<td>Optional control link</td>
<td></td>
</tr>
<tr>
<td>Optional access link</td>
<td></td>
</tr>
<tr>
<td>Save machine status</td>
<td></td>
</tr>
<tr>
<td>Local data</td>
<td></td>
</tr>
<tr>
<td>Temporaries</td>
<td></td>
</tr>
</tbody>
</table>

**Top_sp**

- **Caller’s responsibility to initialize**
- **Callee’s responsibility to initialize**
Scope with Nested Procedures

```pascal
program sort(input, output)
    var a : array [0..10] of integer;
    x : integer;
    procedure readarray;
    var i : integer;
    begin ... end;
    procedure exchange(i, j : integer);
    begin x := a[i]; a[i] := a[j]; a[j] := x end;
    procedure quicksort(m, n : integer);
    var k, v : integer;
    function partition(y, z : integer) : integer
    var i, j : integer;
    begin ... exchange(i, j) ... end
    begin
    if (n > m) then begin
        i := partition(m, n);
        quicksort(m, i - 1);
        quicksort(i + 1, n)
    end
    end;
    begin
    ... quicksort(1, 9)
end.
```
Access Links (Static Links)

The access link points to the activation record of the static parent procedure: 
- $s$ is parent of $r$, $e$, and $q$
- $q$ is parent of $p$
To implement access to nonlocal data \( a \) in procedure \( q \), the compiler generates code to traverse \( n_q - n_a \) access links to reach the activation record where \( a \) resides.

- \( n_q \) is the nesting depth of procedure \( q \).
- \( n_a \) is the nesting depth of the procedure containing \( a \).
Access Links

```plaintext
program main;
  var a:int;
  procedure p;
    var d:int;
    a:=1;
  end p;
  procedure q(i:int);
    var b:int;
    procedure s;
      var c:int;
      call p;
    end s;
    if (i<>0) then q(i-1) else s;
  end q;
  q(1);
end main;
```

Address of `a`, when control is in procedure `s`:

```plaintext
@@(top_sp - #m) + #m+ #0 (here, n_s - n_a = 2)
```
Variable Length Data

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

```plaintext
program main;
  procedure p(i:int);
    var c, a[I], b[I]: int;
    c := a(3);
  end p;
  call p(5);
end main;
```

Address of `c`: `top_sp + #1`
Address of `a(3)`: `@ (top_sp + #2) + (#3 - #1) * #1`
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.
Accessing Nonlocal Variables using Display

program main;
var a:int;
procedure p;
var b:int;
call q;
procedure q();
var c:int;
c:=a+b;
end q;
end p;
call p;
end main;

Address of \( c \): \( D[3] + #0 \)
Address of \( a \): \( D[1] + #0 \)

At the time of executing \( q() \)