We need to understand how a program executes at runtime before we can generate code for it.

- Run-time environment
  - Procedure abstraction
  - Run-time storage management
- Code generation

Intermediate representation

Back end
- Type checker (semantic analysis)
- Parser (syntactic analysis)
- Scanner (lexical analysis)

We are now.

The procedure abstraction

1. Control abstraction
   - Procedure provides many benefits
   - Well-defined entry, exits
   - Control abstraction
2. Name space
   - Names in the intermediate representation
3. External interface
   - Local names are protected from outside
   - New name space within procedure
4. Separate compilation
   - Enables software designs, systems
   - Protection for both caller and callee
5. Procedure abstraction
   - Keeps compiled times reasonable
   - Compile procedures independently
   - Allows us to build large programs

Where are we now?
Procedures Inhakes

The linkage convention is the interface used for performing procedure calls. The interface is also used for the associated activation record or frame (at run time) assumption that each procedure activation has an associated activation record or frame (at run time).

- locals stored in frame
- return address
- return value
- parameters
- access link
- call by reference parameter passing
- call by reference parameter passing

Assumptions:

- locals stored in frame
- can always expand an allocated block
- RISC architecture
- call by reference parameter passing

procedure

procedure p

pre-call

post-return

procedure q

procedure r

procedure s
Procedure Inherras

Data Areas
At runtime, that code manipulates the frame as
At compile time, we generate the code to do this

<table>
<thead>
<tr>
<th>Return</th>
<th>Call</th>
</tr>
</thead>
<tbody>
<tr>
<td>jump to return address</td>
<td></td>
</tr>
<tr>
<td>restore parent’s FP</td>
<td></td>
</tr>
<tr>
<td>unextend basic frame</td>
<td></td>
</tr>
<tr>
<td>restore state</td>
<td></td>
</tr>
<tr>
<td>store return value</td>
<td></td>
</tr>
<tr>
<td>copy return value</td>
<td></td>
</tr>
<tr>
<td>return sequence</td>
<td></td>
</tr>
</tbody>
</table>

Return Sequence

Call Sequence

Call
Caller
Callee

Procedure Inherras
What names are visible in Procedure B?

What names are visible in Procedure C?

Name spaces of languages

LISP

C

Java

Algo1, Pascal, Modula

most recently invoked procedure

dynamic scope

Nested lexical scope

C

Algo1, Pascal, Modula

most recently invoked procedure

dynamic scope

Nested lexical scope

LISP

C

Java

Algo1, Pascal, Modula

most recently invoked procedure

dynamic scope

Nested lexical scope
Run-time storage organization

- Code and static may be separate or intertwined
- Allows both stack and heap maximum freedom

Typical memory layout

Control stack
- Dynamic slice of activation tree
- Return addresses

Data space
- Some data is dynamically allocated in code
- Variable size data must be dynamically allocated
- Extem size data may be statically allocated

Logically Memory Space

Typical Memory Layout

Free

High

Low

Symbol

Variable

Type

C
S
H
I
D
Run-time storage organization

- Explicit allocation, explicit/deallocation
- Lifetimes
- Call-by-reference, pointers, lead to non-local
- Addresses assigned at run-time

Downward exposure:
- Dynamically allocated variables
  - If lifetimes are limited
    - If sizes are fixed
  - Addresses compiled into code

(Head)

Local variables
  - Duplicates
  - Runtime storage ensures universal access, handles
  - Lifetime to fixed size objects
  - Addresses assigned (relocated at link-time)

(Stack)

Global variables
  - (base address)
  - Each variable must be assigned a storage class

When do local variables go?
When can we allocate them on a stack?

Downward exposure:
- Lexical scoping
- Dynamic scoping
  - E.g., pass by reference

Upward exposure:
- Lexical scoping
  - Called procedures may reference my variables

Key issue is lifetime of local names
- With only downward exposure, the compiler cannot

Upward exposure:
- Continue passing style
  - Functions that return functions
    - can I return a reference to my variables?
Heap management

Functionalities:

- Allocate a block of at least k bytes if alloc() is called.
- Places the block pointed to by p.
- freed(p) frees the block pointed to by p.
- Places the block pointed to by freed().
- Removes it from the pool.
- Places the block pointed to by alloc().
- Places a block of at least k bytes into the free pool.

Potential problems:

- Wasted space if alloc() returns blocks that are larger than requested, the excess space is wasted.
- Fragmentation after a series of alloc and free commands, the free space pool becomes fragmented, preventing allocation of large blocks.
- Initialization requires cooperation among processes.
- Global variables
- Local variables
- Need to determine offset in frame
- Stored on stack in frame
- May use access links or displays
- May access variables declared in enclosing scopes
- Memory management
- Functionality:
  - all oc(k) allocates a block of at least k bytes.
  - free(p) frees the block pointed to by p.
  - freed(p) frees the block pointed to by p.
  - alloc() allocates a block of at least k bytes.

Heap management

Issues:

A heap management scheme must balance these issues:

- Speed – alloc and free should be inexpensive.
- Fragmentation – preventing allocation of large blocks.
- Initialization – after a series of alloc and free commands, the free space pool becomes fragmented, preventing allocation of large blocks.
- Wasted space – if alloc returns blocks that are larger than requested, the excess space is wasted.

How does the code find data at run-time?

Lexical scoping:

- Local variables
- Global variables
- Initialization requires cooperation among processes.
- Visible everywhere
- Need to determine offset in frame
- Stored on stack in frame
- May use access links or displays
- May access variables declared in enclosing scopes

Access to non-local data

Knuth, Volume 1, §2.7
### Garbage Collection Approach

Automatically free unused memory

- No pointer to a block done with it

#### Advantages

- Simplifies programmers' life
- Avoids dangling references to "free" memory
- Avoids memory leaks

#### Disadvantages

- May require a program to halt during collection
- May be costly to perform
- May be conservatice

- May not find all free memory

#### Garbage Collection Algorithms

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reference Counting</td>
<td>Update pointers where necessary</td>
</tr>
<tr>
<td></td>
<td>Copy all live data to new area, free rest</td>
</tr>
<tr>
<td></td>
<td>Other concurrent or halt program</td>
</tr>
<tr>
<td>Copying Collector</td>
<td>Mark all accessible data, free rest</td>
</tr>
<tr>
<td></td>
<td>Start with pointers in registers, frames</td>
</tr>
<tr>
<td></td>
<td>Half program during collection</td>
</tr>
<tr>
<td></td>
<td>Mark and scan</td>
</tr>
<tr>
<td></td>
<td>Reclaiming memory may cause more collection</td>
</tr>
<tr>
<td></td>
<td>When count reaches zero, reclaim memory</td>
</tr>
<tr>
<td></td>
<td>Record # references to each piece of memory</td>
</tr>
<tr>
<td></td>
<td>Concurrent with program execution</td>
</tr>
</tbody>
</table>

- Garbage collection algorithms
- (e.g., pointer arithmetic in C)