Intermediate Code Generation

Given: Results of parsing...

- Abstract Syntax Tree
- Embed generation directly in parser

Option 1:

Generate Final Code Directly

- SPARC Assembler, or
- Machine Code

Option 2:

Generate Intermediate Code ("IR code")

... Then generate final code

"Final Code Generation Phase"

```
    goto  L3
    ...
L3: goto  L7

    t1 := x + y
    t2 := t1
```

Why?

Intermediate Code Generation

Given: Results of parsing...

- Abstract Syntax Tree
- Embed generation directly in parser

Option 1:

Generate Final Code Directly

- SPARC Assembler, or
- Machine Code

Option 2:

Generate Intermediate Code ("IR code")

... Then generate final code

"Final Code Generation Phase"

```
    goto  L3
    ...
L3: goto  L7

    t1 := x + y
    t2 := t1
```

Why?

- Retargetting the compiler
  - Got a new CPU architecture? Replace Final Code Generator
- Break code generation task into two smaller tasks
- Optimization
  - "Machine Independent" Optimizations

Approach #1

Syntax-Directed Translations

$E_0 \rightarrow E_1 + E_2$	E_0.place := NewTemp () E_0.code := E_1.code ‖ E_2.code ‖ IR (E_0.place, ‘:=’, E_1.place, ‘+’, E_2.place)

CFG Rule

“IR” - A routine to create IR instructions

Attributes (Synthesized, Inherited)

Approach #2

We have parsed the program and built an in-memory representation
(Abstract Syntax Tree)
We will create methods to walk this AST and emit code

Intermediate Representation (Variations)

Source Code

```
a = b * -c + b  * -c;
```

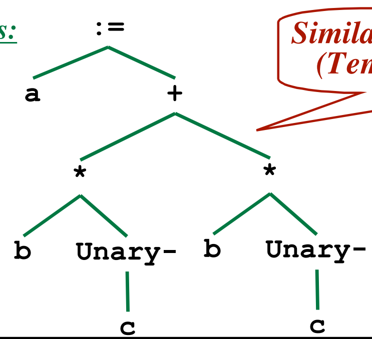
Closer to SPARC code

Three-Address Code:

Each instruction has (up to) 3 operands.

```
t1  := -c              neg    c      ⇒ t1
t2  := b * t1          mult   b,t1   ⇒ t2
t3  := -c              neg    c      ⇒ t3
t4  := b * t3          mult   b,t3   ⇒ t4
t5  := t2 + t4         add    t2,t4  ⇒ t5
a  := t5               move   t5     ⇒ a
```

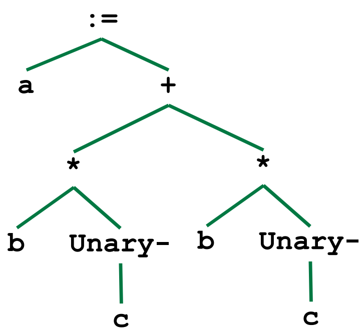
Tree Representations:



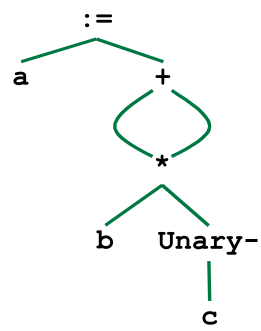
Similar to the AST we already have. (Temporaries are ignored, here)

Graphical Representations

Tree:

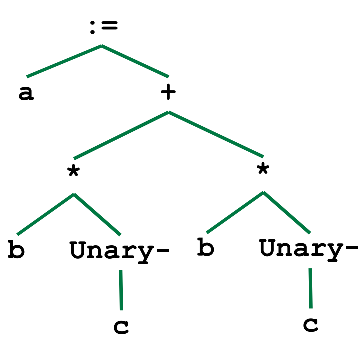


DAG:

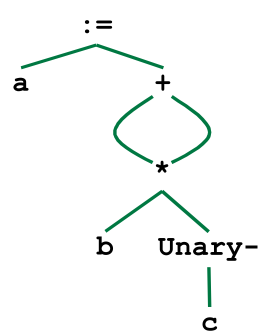


Graphical Representations

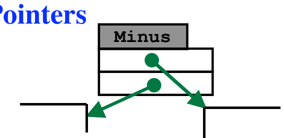
Tree:



DAG:



Structures and Pointers



0	id	b	-
1	id	c	-
2	unary-	1	-
3	mult	0	2
4	id	b	-
5	id	c	-
6	unary-	5	-
7	mult	4	6
8	add	3	7
9	id	a	-
10	assign	9	8

An array of fixed-sized records

Q: How to build DAG's (i.e., trees with shared, common parts)?

A: When you are about to allocate a new node; look to see if you already have one with the same info.

Three-Address Instructions

Idea:

One operation
Three "addresses" (fields, args), at most
Two operands
One result
(Some instructions have only 0, 1, or 2 addresses)
Much closer to machine language

Examples:

Notation used in Textbook

```
t1  := -c              neg    c     ⇒ t1
t2  := b * t1          mult   b,t1  ⇒ t2
t3  := -c              neg    c     ⇒ t3
t4  := b * t3          mult   b,t3  ⇒ t4
t5  := t2 + t4         add    t2,t4 ⇒ t5
a  := t5               move   t5    ⇒ a
```

- Looks like source code, but...
- NOTE: Lots of temp variables
  Tend to create many
  Will try to eliminate during optimization

Source:

```
a = (b * -c) + (b * -c);
```

Translation #1:

```
t1  := -c
t2  := b * t1
t3  := -c
t4  := b * t3
t5  := t2 + t4
a  := t5
```

Translation #2:

```
t1  := -c
t2  := b * t1
t5  := t2 + t2
a  := t5
```

The shared sub-expression is computed only once

Types of Three-Address Instructions

Binary Operations

```
x := y + z
x := y * z
...etc...
```

Want different instructions for
- Integer
- Floating-point

Notational Variations

```
x := y + z (int)
x := y + z (float)
```

$x := y +_i z$
$x := y +_f z$

```
x := iadd(y,z)
x := fadd(y,z)
```

```
iadd  y,z ⇒ x
fadd  y,z ⇒ x
```

Unary Operations

```
x := -y
x := IntToReal(y)
```

Notational Variations

```
x := -y (int)
```

$x := -_i\ y$

```
x := ineg(y)
```

```
ineg  y ⇒ x
```

Assignment / Move

```
x := y
```


Jump / Goto / Unconditional Branch

```
goto Lab_58
```

Label

```
Lab_43:
```

A place holder
Strictly for labeling the target of a branch
Acts like a nop
Labels are simple strings
(Assembler/Linker will assign addresses)

```
Lab_43:
        x := y + 3
        z := x * 2
        if z > w then goto Lab_43
```

Function "`NewLabel()`" returns....

```
"Lab_1"
"Lab_2"
"Lab_3"
...
```

Conditional Jumps / Branches

```
if x < y then goto Lab_57
```

Conditions: <, <=, >, >=, =, !=
Data Types: integer, floating

```
if x < y then... (int)       gotoiLT
if x <= y then... (int)      gotoiLE
  ...
if x < y then... (float)     gotofLT
if x <= y then... (float)    gotofLE
  ...
```

Procedure / Function Calls

```
param  x      ->  ld    ...,%o0
param  y      ->  ld    ...,%o1
call   foo    ->  call  foo
                  nop
result x      ->  st    %o0,...
```

Array Accessing

```
x := y[i]
x[i] := y
```

Address / Pointer Manipulation

```
x := &y      Load address
x := *y      Load indirect
*x := y      Store indirect
```

You can design the exact instruction set in any way that facilitates compilation.

But you must be careful to be clear and precise about the IR instructions' meanings.

More IR instructions ⇒ Easier to compile to.
...but more work during final code generation

Fewer IR instructions ⇒ < Reverse >

Representing 3-Address Statements

- Quadruples ("Quads")
- Triples
- Indirect Triples

```
t0 := x + y
t1 := t0 * z
a := t1
```

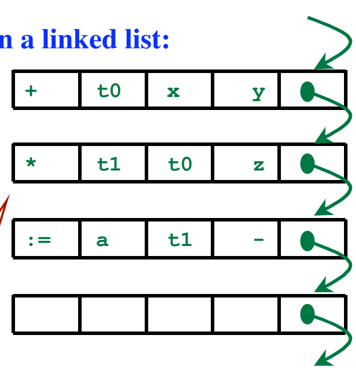
Quadruples

Store each field directly

... In an array:

0	+	t1	x	y
1	*	t2	t1	z
2	:=	a	t2	-
3				

... In a linked list:



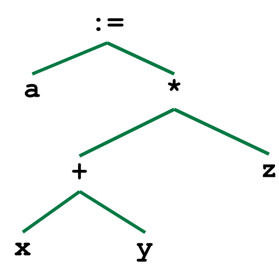
Less Space

Easier to re-order

Triples

Don't store the result directly.
Implicitly associate a temporary result with each triple.

```
t0  := x + y
t1  := t0 * z
a  := t1
```



Avoids creating the temporaries.
Saves storage.
Difficult to re-order instructions.

0	+	x	y
1	*	0	z
2	:=	a	1
3			

The following instruction is difficult

```
x[i]  := y
```

It takes 2 triples.

Indirect Triples

Get around the re-ordering problem
... by introducing another data structure.

0:	100
1:	101
2:	102
3:	103

100:	+	x	y
101:	*	100	z
102:	:=	a	101
103:	:=	b	w

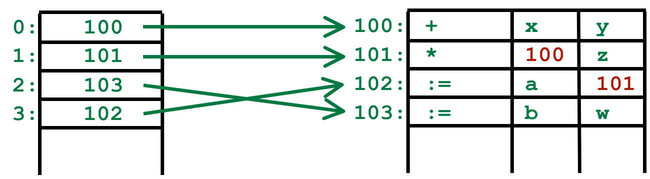
Quadruples
Less indirection, simpler
Easier to manipulate, reorder

Triples

Indirect Triples
About same amount of space as quadruples
May save space when lots of shared sub-expressions
More complex

Indirect Triples

Get around the re-ordering problem
    ... by introducing another data structure.



Quadruples
    Less indirection, simpler
    Easier to manipulate, reorder
Triples
Indirect Triples
    About same amount of space as quadruples
    May save space when lots of shared sub-expressions
    More complex

Translating Expressions

Idea: Use Syntax-directed translations

For each expression, we'll synthesize two attributes:

E.code
    This is the code we will generate for expression E.
        (It is a sequence of all the IR instructions in the translation.)
    When executed (at runtime), these instructions will compute the
        value of the expression and place the value into some variable.

E.place
    The name of the variable (often a temporary variable)
        into which this code will move the final result value when executed.

For each statement, we will synthesize one attribute:

S.code
    The IR code for this source statement.

Call "NewTemp" to create a new temporary variable
```
t = NewTemp();
```
Call "IR" to create a new 3-address instruction
```
IR (t, ":=", x, "+", y)  ⇒  "t := x + y"
```

Goal

Take a source statement and produce a sequence of IR quads:

Example:

```
x := y + z;
```

IR Quads:

```
t1 := y + z
x := t1
```

Example:

```
x := (y + z) * (u + v);
```

IR Quads:

```
t1 := y + z
t2 := u + v
t3 := t1 * t2
x := t3
```


Synthesized Code and Place Attributes

Consider

$E_0 \rightarrow E_1 + E_2$

Work bottom-up.

Assume we already have

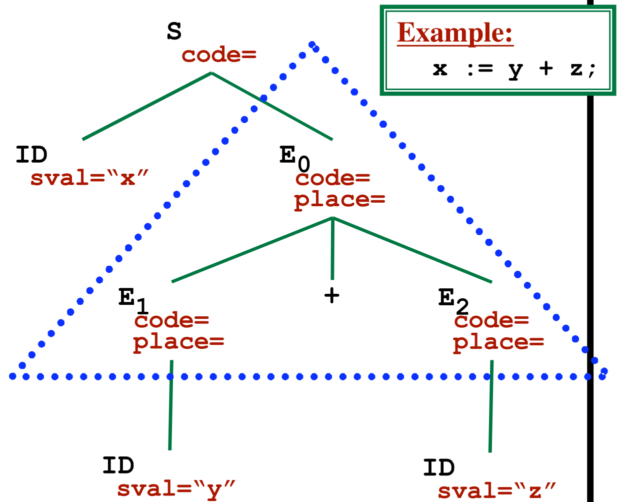
E_1.place = "y"
E_1.code = " "
E_2.place = "z"
E_2.code = " "

Then, use the rules to compute

E_0.place
E_0.code

Example:

```
x := y + z;
```



$E_0 \rightarrow E_1 + E_2$ E_0.place := NewTemp ()
E_0.code := E_1.code || E_2.code ||
IR (E_0.place, ':=', E_1.place, '+', E_2.place)

Assume we already have
E_1.place = "y"
E_1.code = " "
E_2.place = "z"
E_2.code = " "

Then, use the rules to compute
E_0.place = "t1"
E_0.code = "t1 := y + z"

$E_0 \rightarrow E_1 + E_2$	E_0.place := NewTemp () E_0.code := E_1.code || E_2.code || IR (E_0.place, ':=', E_1.place, '+', E_2.place)
$E_0 \rightarrow E_1 * E_2$	E_0.place := NewTemp () E_0.code := E_1.code || E_2.code || IR (E_0.place, ':=', E_1.place, '*', E_2.place)
$E_0 \rightarrow$ **ID**	E_0.place := ID.svalue E_0.code := " "
$E_0 \rightarrow$ **-** E_1	E_0.place := NewTemp () E_0.code := E_1.code || IR (E_0.place, ':=', '-', E_1.place)
$E_0 \rightarrow$ (E_1)	E_0.place := E_1.place E_0.code := E_1.code
S → **ID** **:=** E **;**	S.code := E.code || IR (ID.svalue, ':=', E.place)

Generating Code for "IF" Statement

```
if E then S1 else S2 endIf

          <code for E>
          if E.place = 0 goto Label_Else
          <code for S1>
          goto Label_End
Label_Else:
          <code for S2>
Label_End:
```


Generating Code for "IF" Statement

```
if E then S1 else S2 endIf

          <code for E>
          if E.place = 0 goto Label_Else
          <code for S1>
          goto Label_End
Label_Else:
          <code for S2>
Label_End:
```


$S_0 \rightarrow$ if E then S_1 else S_2 endIf

label_else = NewLabel ()
label_end = NewLabel ()
S_0.code := E.code ||
    IR ('**if**', E.place, '**=0 goto** ', label_else) ||
    S_1.code ||
    IR ('**goto**', label_end) ||
    IR (label_else, '**:**') ||
    S_2.code ||
    IR (label_end, '**:**')

Compile-Time vs. Run-Time

"*Static*" versus "*Dynamic*"

Source Text ("Syntactic", "Lexical")

```
procedure foo (...) is begin ... end;
```

Only 1 static (compile-time) copy of "foo"

Run-time Activations

At any moment at runtime, "foo" may have

- Zero activations
- One activations
- Many activations (if "foo" is recursive)

Standard Terminology

"Routine"
  "Procedure" - Returns no result
  "Function" - Returns a result

Other Terminology

"Procedures" - may or may not return a result (PCAT)
  "Void-Procedure"
  "Non-void Procedure"

"Functions" - may or may not return a result (C)
  "Void-Function"
  "Non-void Function"

Terminology

Static

A routine is called a a place in the program

e.g., "quicksort" is called on lines 16, 17, and 23

"***formal parameters***" Like variables

"***actual arguments***" Expressions (which yield a value when executed)

Runtime

When a routine is called, it is a new "*activation*" (or "*invocation*")

The routine is "*invoked*".

The "*lifetime of an activation*"

From the moment of invocation to the moment of return

The "*Caller*"

The "*Callee*" (the "*called*" routine)

Static:

"quicksort" calls "partition" on line 12.

Dynamic:

This activation of quicksort calls quicksort with arguments 4 and 7.

CS-322 Code Generation-Part 1

Nested Activations

At runtime...

Assume p calls f

Then: f must return before p returns

Assumptions:

- Single thread of control
- No errors / exceptions
- No suspended closures
- No gotos.

Call and **Return** are like nested parentheses

```
(( (() (()()) ()) ( () ) ) () () )
```

Add print statements to routines

```
procedure foo ()
    print ("foo entered");
    ...
    print ("foo returns");
    return;
```

© Harry H. Porter, 2006

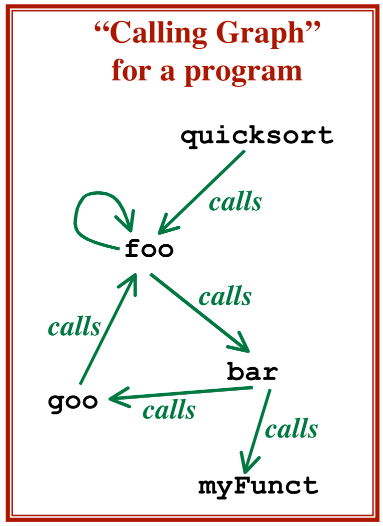
Recursion

Recursive routines

```
foo entered
   foo entered
      foo entered
         ...
      foo returns
   foo returns
foo returns
```

Mutually recursive routines (“Indirectly recursive”)

```
foo entered
   bar entered
      goo entered
         foo entered
            ...
         foo returns
      goo returns
   bar returns
foo returns
```



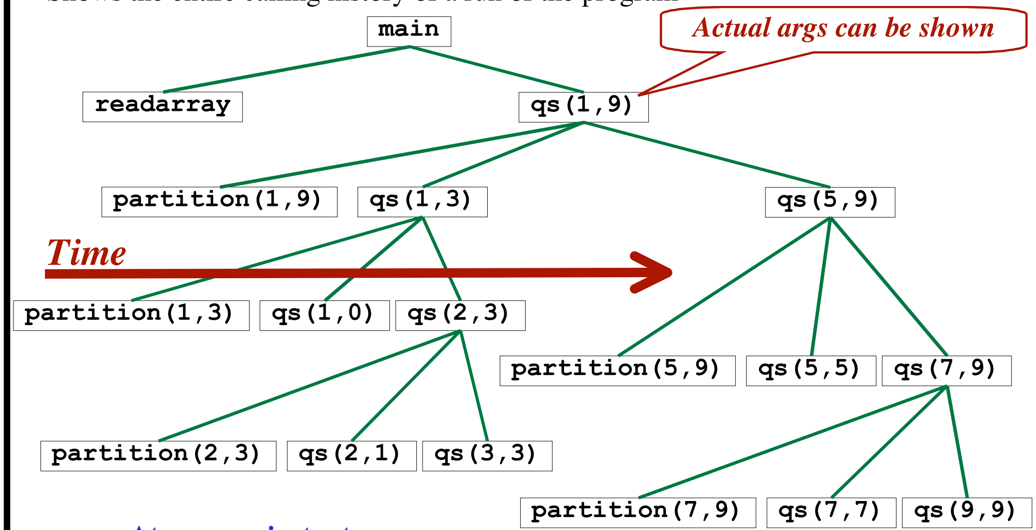
The Quicksort Program

```
program is
  var a: array of integer := ...;
  procedure readArray () is
    var i: integer := 0;
    begin
      for i := 1 to 9 do read ( a[i] ); end;
    end;
  procedure partition (y, z: integer): integer is
    var i,j,x,v: integer := 0;
    begin ... end;
  procedure quicksort (m, n: integer) is
    var i: integer := 0;
    begin
      if n>m then
        i := partition (m, n);
        quicksort (m, i-1);
        quicksort (i+1, n);
      end;
    end;
  begin
    a[0]  := -9999;
    a[10] :=  9999;
    readArray ();
    quicksort (1,9);
  end;
```

This code is in `examples/sort.pcat`

The Activation Tree

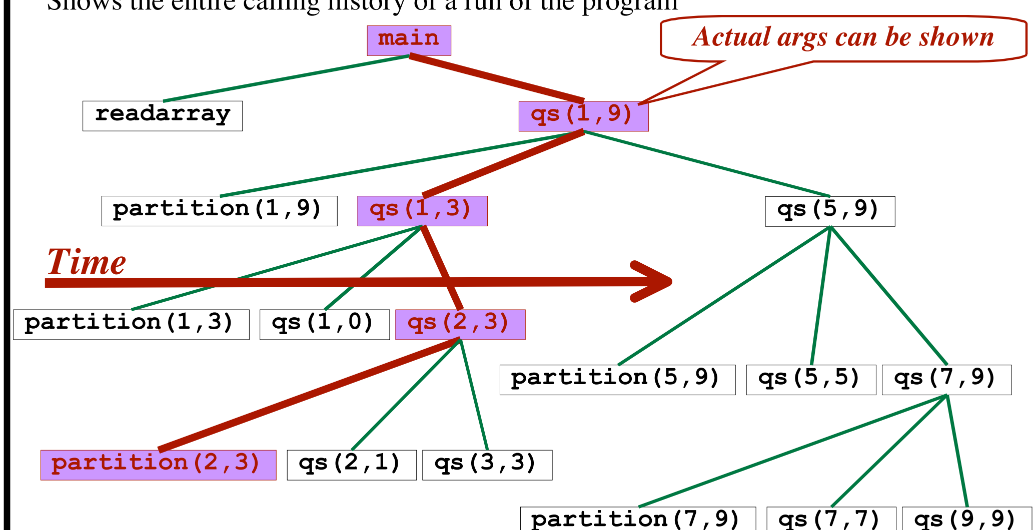
Shows the entire calling history of a run of the program



At any one instant...
A path from the root to a node
shows the current "*activation stack*"

The Activation Tree

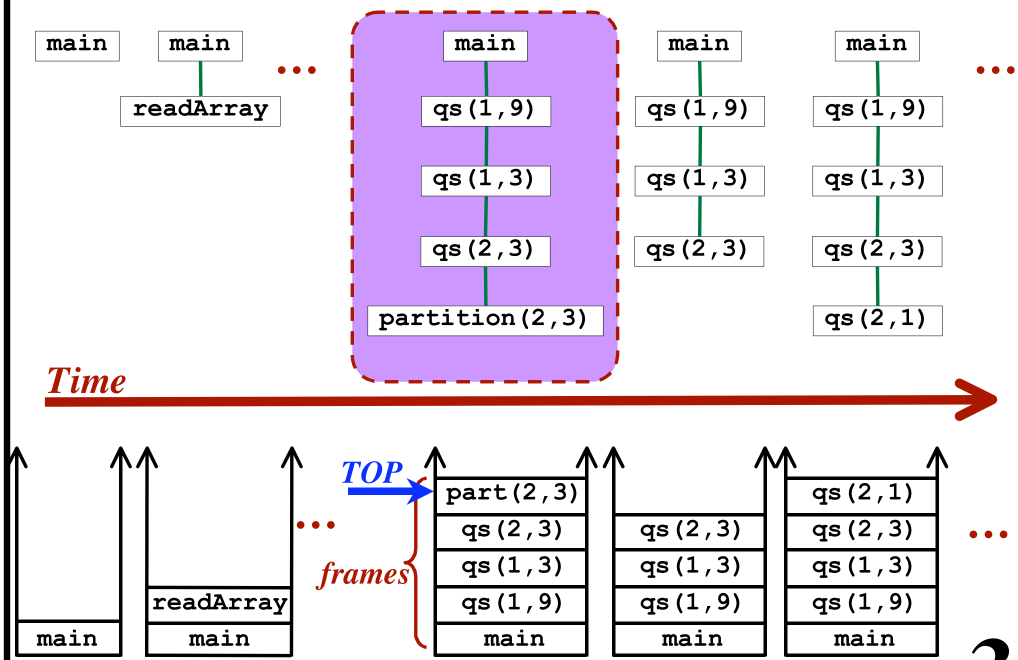
Shows the entire calling history of a run of the program



At any one instant...
A path from the root to a node
shows the current "*activation stack*"

Activation Stack

Also called:
Control Stack
Runtime Stack
Frame Stack



Each live routine has an activation record (a “frame”) on the stack

Enter a new routine?
- **PUSH a frame onto the stack**

Return from a routine?
- **POP the top frame**

Want to access the local variables of the current routine?
- **Find them in the frame on the TOP of the stack.**

Declarations

There may be several declarations for some variable name,

```
procedure foo (...) is
  var i: integer := ...;
  procedure bar (...) is
    var i: integer := ...;
    begin ... end;
  begin ... end;
```

"Scope Rules"

The scope of a declaration

The part of the program (static) where we can use the declared name.

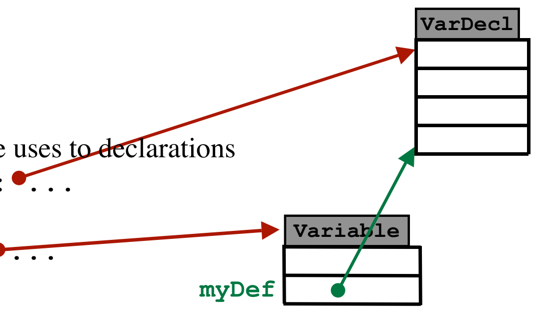
"Local"

"Non-Local"

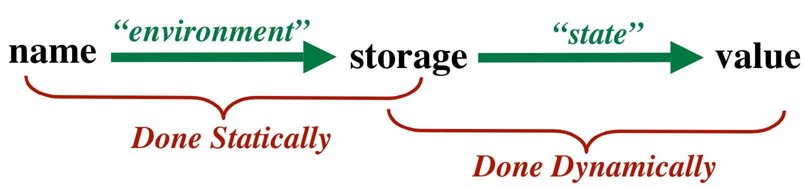
Symbol Table

Match variable uses to declarations

```
var x: ...
...
... x ...
```



Binding of Names



L-Values

R-Values

Assignments change state.

Declarations change the environment.

Some languages allow the environment

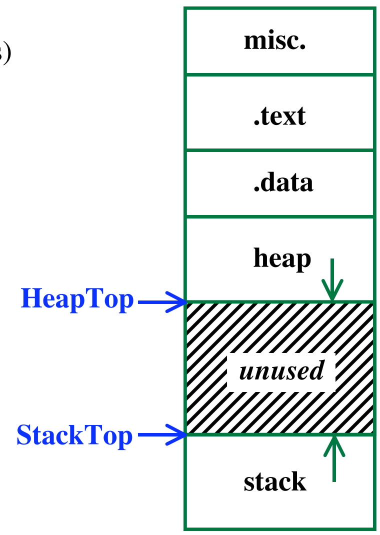
to be changed dynamically!

Memory

.text segment
will be read-only
code (includkng library routines)
fixed (constant) data
.asciz strings

.data segment
read/write
used to hold (global) data
“display registers”

.bss segment
read/write
all bytes initialized to zero
(we won’t use)



Dangling References

“A pointer to storage that has been freed / deallocated”

```
p := new (...)
...
delete p;
...
...p...
```

```
main () {
  int *p;
  p = foo();
  ... p ...
}
int * foo () {
  int i = 123;
  return &i;
}
```

Consequences?

- Clobber random memory locations
  Weird program behavior
- Read / write the wrong data
  Normal program behavior, but incorrect results
- Caught by the runtime system
  Catastrophic crash (core dump)
  Graceful failure w/ message
- No affect; program functions correctly
  Is the program still wrong?

Dangling References

Problem:
The programmer explicitly frees / deallocates data
...but the programmer frees data "too soon".

Solution:
Don't let programmer free data!

Problem:
The program uses more memory than necessary.

Solution:
The runtime system identifies objects that
cannot possibly be used again by the program
"Garbage" objects
The runtime system reclaims this space
The "Garbage Collector"

The Heap

Simplest Organization
Allocate space at the end (top) of the heap
Abort when "StackTop < HeapTop"
Never free / release space

Explicit Freeing of Space
malloc / calloc / free in "C"
Object creation / deletion in "C++"
Keep a list of free chunks of memory ("free list")
e.g., Linked list of memory areas
"Free" adds memory to this list
To allocate memory...
First check the free list
If possible, use the free list
Otherwise, grow the heap
NEVER MOVE ALLOCATED CHUNKS OF MEMORY
Program relies on pointers remaining valid. ("C", "C++")

Automatic Garbage Collection

Java, Smalltalk, Haskell, Lisp, ...

Memory Management Subsystem ("Heap Manager")

Program periodically asks for memory (allocates objects)
Memory manager returns (a pointer to) a new chunk of space

Space running low?

Memory manager identifies objects that are no longer in use

"garbage"

Object is reachable? Not garbage
Object is unreachable? Garbage

Reclaims space used by garbage objects

Fragmentation?

Move objects around ("***compaction***")

To make large chunks of memory available

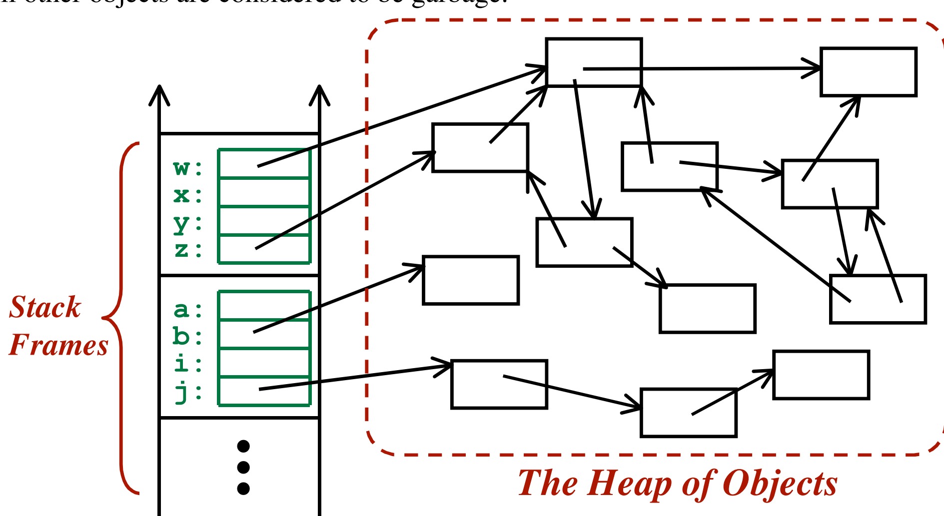
Memory manager must re-adjust pointers when objects are moved.
Memory manager must be able to identify all pointers at runtime.

Tight coupling with the language implementation

The program doesn't know objects have been moved!!!

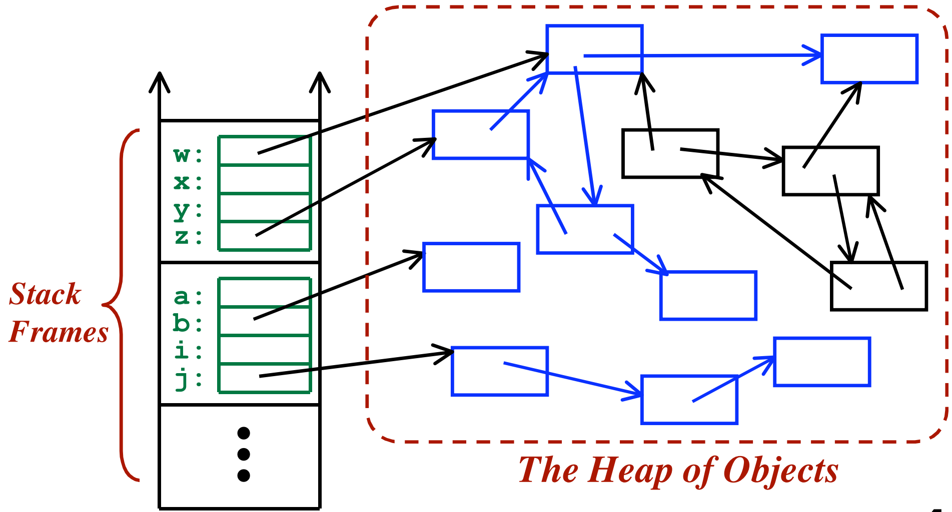
Garbage

Memory is organized into "objects" which point to each other.
Objects are allocated at random during program execution.
Some objects are "**reachable**" from the program variables.
All other objects are considered to be garbage.



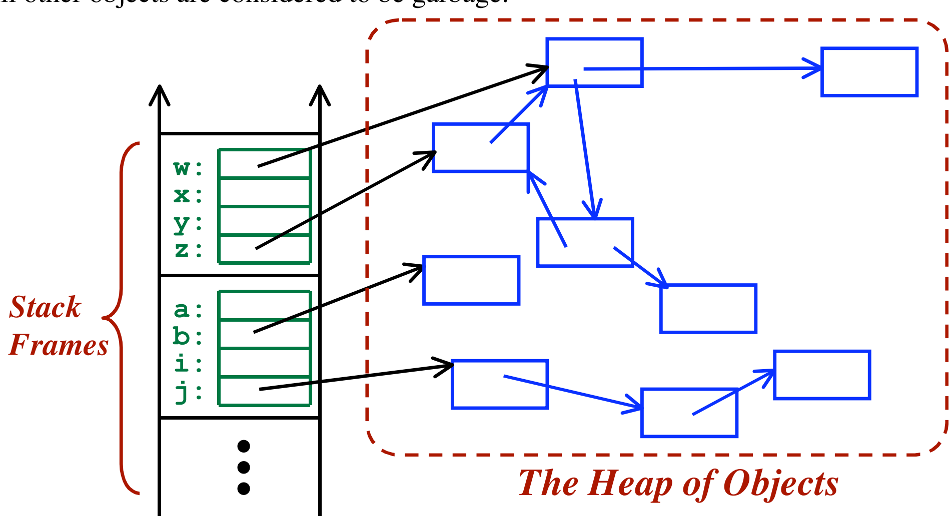
Garbage

Memory is organized into “objects” which point to each other.
Objects are allocated at random during program execution.
Some objects are “**reachable**” from the program variables.
All other objects are considered to be garbage.



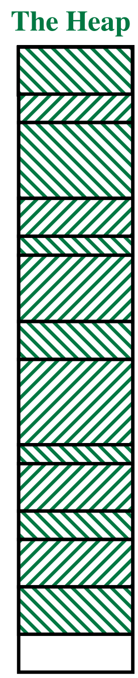
Garbage

Memory is organized into “objects” which point to each other.
Objects are allocated at random during program execution.
Some objects are “**reachable**” from the program variables.
All other objects are considered to be garbage.



Fragmentation

Variable-sized chunks of memory are allocated.
Some chunks are freed (in more-or-less random order).
The resulting free space become “fragmented”.
Need to allocate more space?
    Adequate space is available
    ... but it is not contiguous!

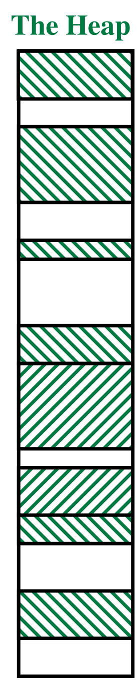


CS-322 Code Generation-Part 1

Fragmentation

Variable-sized chunks of memory are allocated.
Some chunks are freed (in more-or-less random order).
The resulting free space become “fragmented”.
Need to allocate more space?
    Adequate space is available
    ... but it is not contiguous!

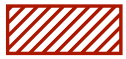
Several objects are freed.



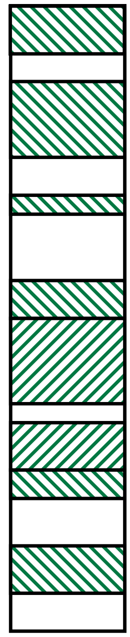
Fragmentation

Variable-sized chunks of memory are allocated.
Some chunks are freed (in more-or-less random order).
The resulting free space become “fragmented”.
Need to allocate more space?
    Adequate space is available
    ... but it is not contiguous!

Want to allocate this object:



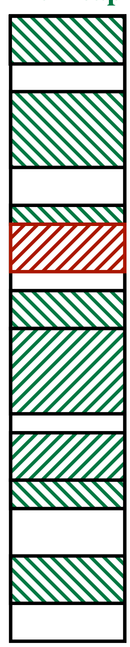
The Heap



Fragmentation

Variable-sized chunks of memory are allocated.
Some chunks are freed (in more-or-less random order).
The resulting free space become “fragmented”.
Need to allocate more space?
    Adequate space is available
    ... but it is not contiguous!

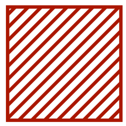
The Heap



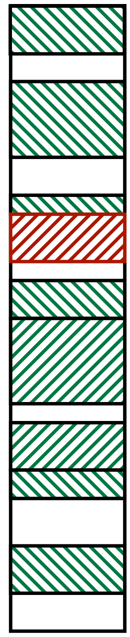
Fragmentation

Variable-sized chunks of memory are allocated.
Some chunks are freed (in more-or-less random order).
The resulting free space become “fragmented”.
Need to allocate more space?
    Adequate space is available
    ... but it is not contiguous!

Want to allocate this object:



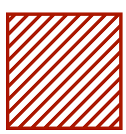
The Heap



Fragmentation

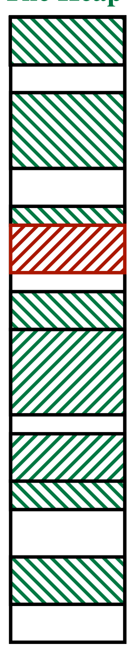
Variable-sized chunks of memory are allocated.
Some chunks are freed (in more-or-less random order).
The resulting free space become “fragmented”.
Need to allocate more space?
    Adequate space is available
    ... but it is not contiguous!

Want to allocate this object:



“Compact” Memory!

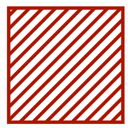
The Heap



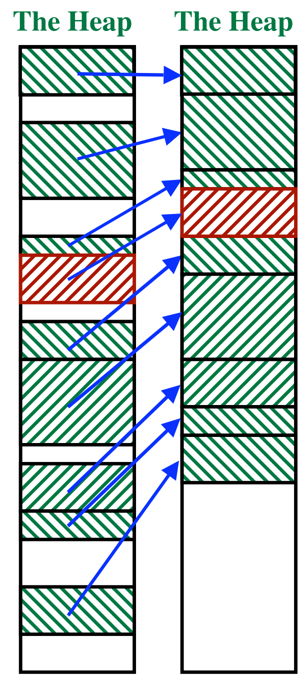
Fragmentation

Variable-sized chunks of memory are allocated.
Some chunks are freed (in more-or-less random order).
The resulting free space become “fragmented”.
Need to allocate more space?
- Adequate space is available
- ... but it is not contiguous!

Want to allocate this object:



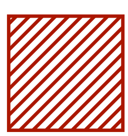
“Compact” Memory!



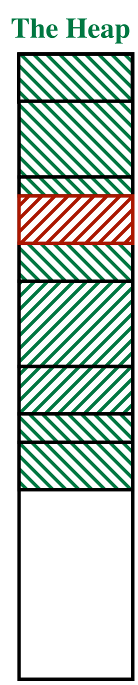
Fragmentation

Variable-sized chunks of memory are allocated.
Some chunks are freed (in more-or-less random order).
The resulting free space become “fragmented”.
Need to allocate more space?
- Adequate space is available
- ... but it is not contiguous!

Want to allocate this object:

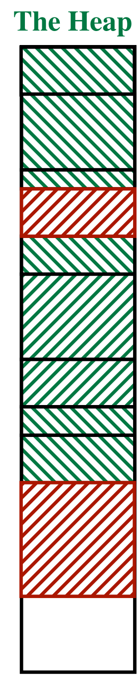


“Compact” Memory!



Fragmentation

Variable-sized chunks of memory are allocated.
Some chunks are freed (in more-or-less random order).
The resulting free space become "fragmented".
Need to allocate more space?
    Adequate space is available
    ... but it is not contiguous!



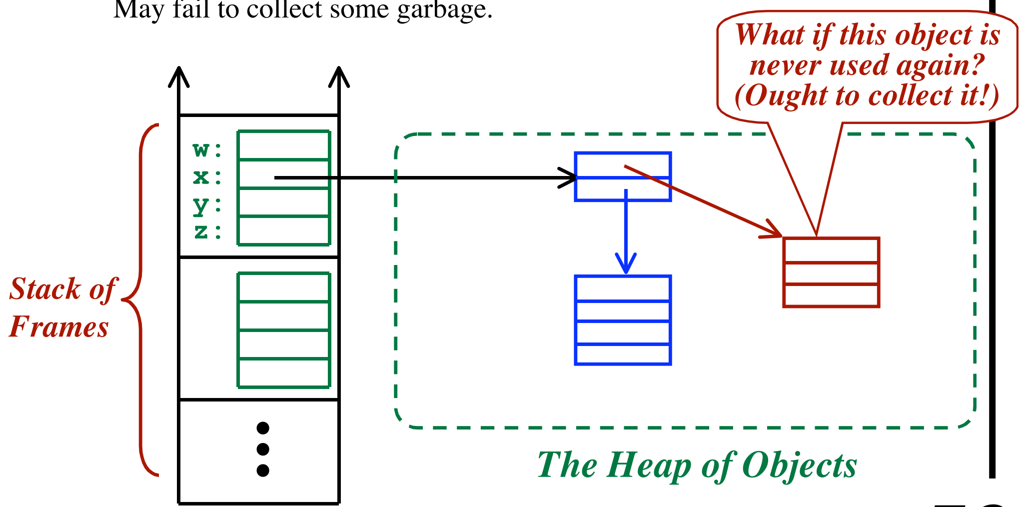
Pointer Problems

The program frees memory "too soon".
- Dangling references
    - Program crashes
    - Program accesses/overwrites other data
        - Incorrect / weird behavior

The program fails to free unused memory.
- The program frees memory "too late".
- The program slowly eats up memory.
    - Finally, the program runs out of memory.
        - "memory exhausted" error
- A Debugging Nightmare
    - Happens with long running programs
        - Difficult to replicate the bug
    - Where exactly is the "error"
        - Errors of omissions are hard to localize!

Pointer Problems

Automatic Garbage Collection
- Dangling references no longer possible.
- Freeing memory...
  - The Garbage Collector is conservative.
  - May fail to collect some garbage.



CS-322 Code Generation-Part 1

Pointer Problems

Automatic Garbage Collection
- Dangling references no longer possible.
- Freeing memory...
  - The Garbage Collector is conservative.
  - May fail to collect some garbage.