Backend Code Generation

Fan Long

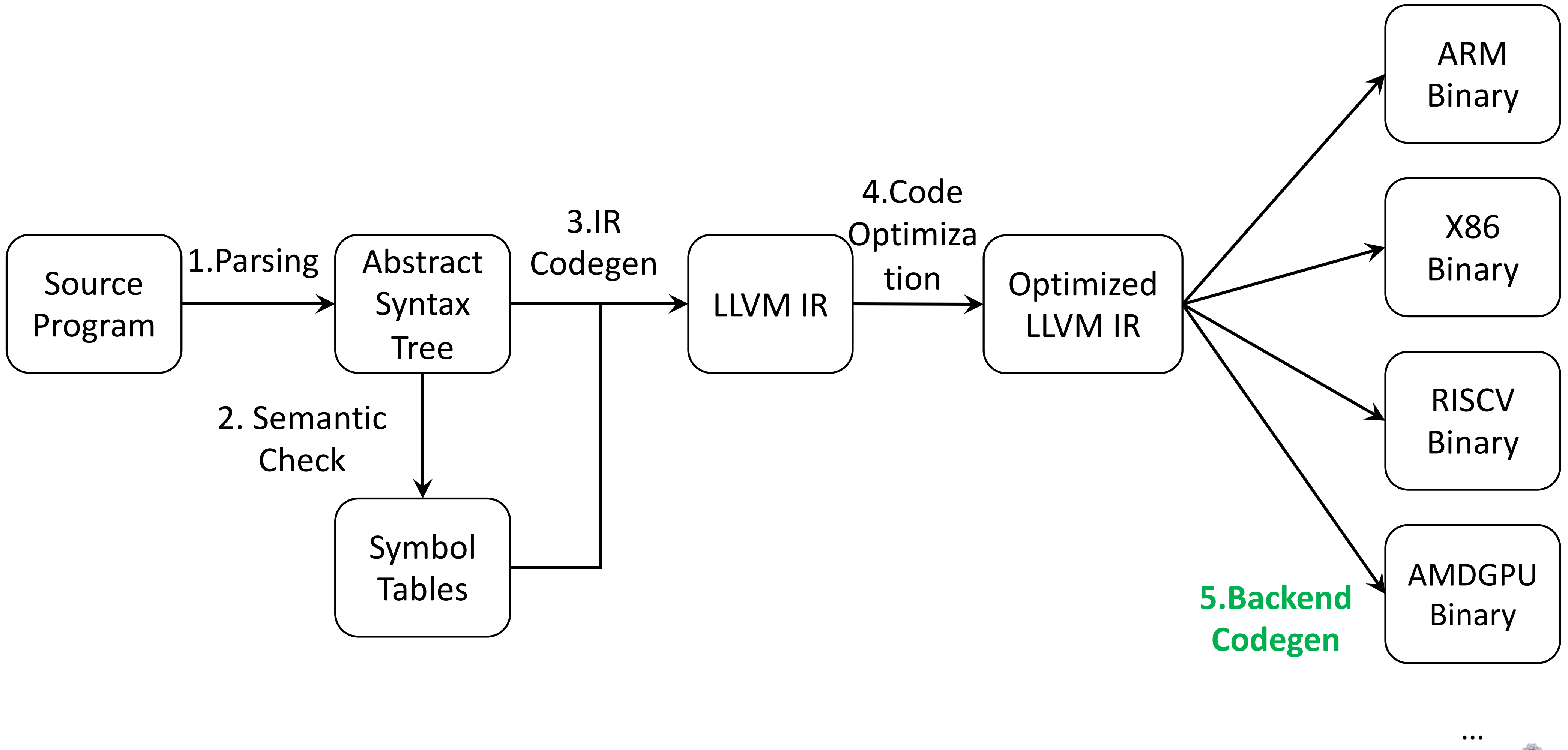
University of Toronto

Backend Code Generation

- Map IR (e.g., LLVM IR) to some real machine code. Assume
  - Program has passed semantic analysis.
  - All type conversions are explicit in the IR.
  - All address calculations (e.g., array subscripts) are explicit in the IR.
  - Control flow graph (IR branches) is correct.
- Code Generation Actions
  - Perform final storage allocation. Assign hardware addresses (i.e., offsets in activation records) for all variables.
  - Transform IR to machine instructions. Usually, one linear pass over IR.
  - Mechanisms to handle branch address fixups, hardware register allocation.

UNIVERSITY OF TORONTO

Typical LLVM Compiler Workflow



UNIVERSITY OF TORONTO

Challenges

- **Runtime Environment**: How to manage runtime memory and provide execution environments.
  - Stack memory management.
  - Global memory management.
  - Dynamic memory management.
  - Runtime library implementation.
- **Code Generation**: How to translate IR to the target machine code.
  - Instruction selection.
  - Register allocation.
- Machine Dependent Optimizations

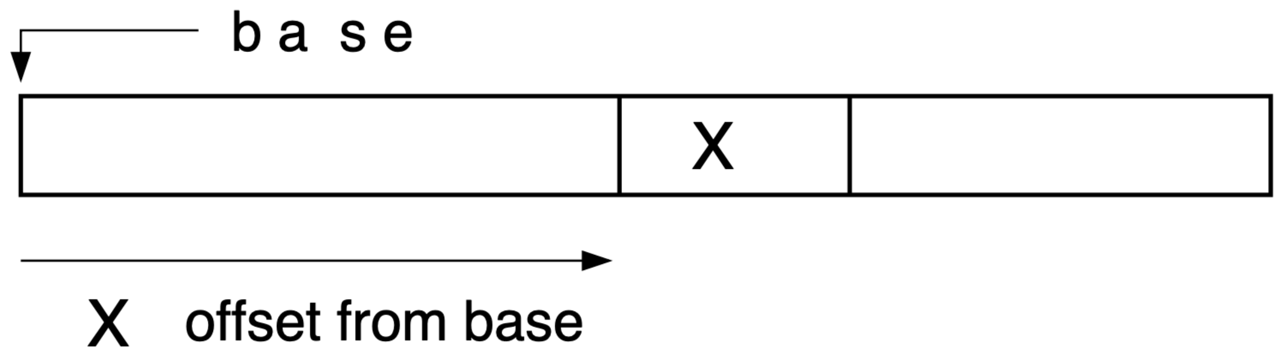
UNIVERSITY OF TORONTO

Runtime Environment

- **Alloca** instructions → Stack memory in the target architectures.
- **Global Variables** in LLVM IR → Data section in the binary.
- **Dynamic Allocations** → System calls or runtime libraries.
- Challenges:
  - More complicated data structures (arrays, struct, unions) require the complier to provide a *mapping algorithm* to for access to information stored in the data structure (e.g. elements of an array, fields in a record).
  - How to organize stack memory when handling chains of function calls?
  - How to manage the dynamically allocated memory? Garbage collection?

Addressing for Structured Data

- The compiler must implement access to non-scalar data objects. e.g. arrays, struct, unions
- *General case:* Access the contents of a data object given only:
  - The address of the beginning of the object in memory.
  - A *compatible* type declaration for the object.
  - *Compatible* is a language-specific property.
- Must implement the most general case of object access allowed in the language.



Array Acess Mapping

- In most modern languages (except Fortran), arrays are laid out in memory in *row major order*, i.e., the rightmost subscript of consecutive array elements varies most rapidly.
- Because arrays can be passed as parameters and/or allocated in dynamic storage, the compiler needs to be able to calculate the address of an arbitrary element of an array given only the base address of the array and information from the declaration of the array.
- If the basic unit of addressing (e.g., bytes) is different from the size of the array element (e.g., words, double words) then each subscript must be *scaled* by the size of the array element in address units.
- Generic array declaration:

$$typeA[\, L_1 : U_1 \ , \ L_2 : U_2 \ , \ \ldots L_n : U_n \,]$$

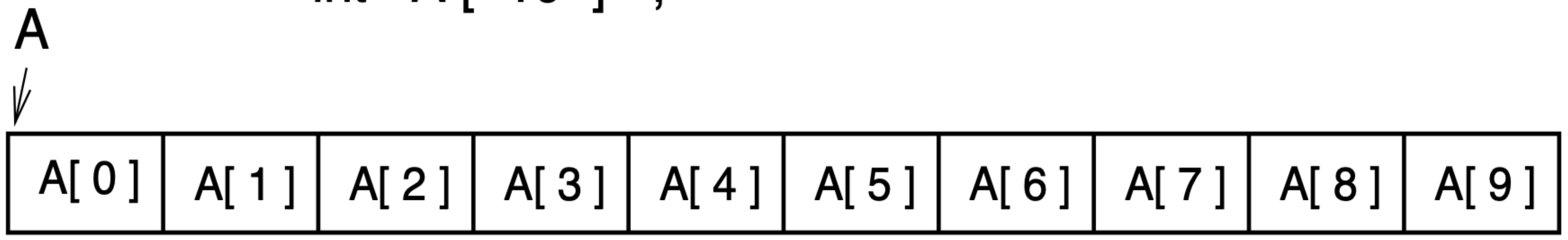
- Generic array reference:

$$A[\, E_1 \ , \ E_2 \ , \ \ldots E_n \,]$$

Array Access Mapping

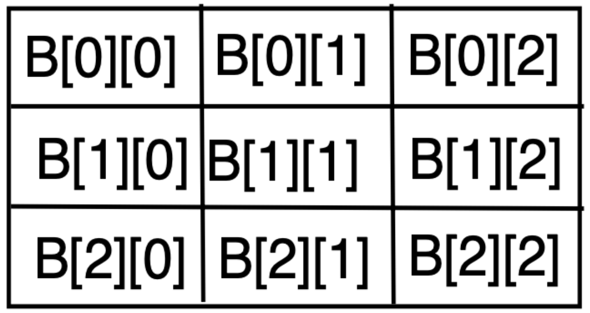
One Dimensional Array

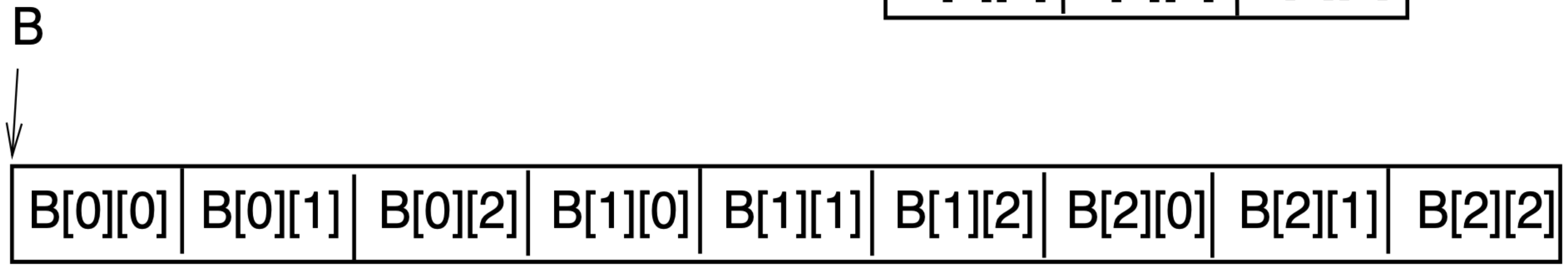
```
int A[ 10 ] ;
```



Two Dimensional Array

```
int B [ 3 ] [ 3 ] ;
```





Array Access Mapping

- Define: base address of an array

$$base = addr(A[\,L_1\ ,\ L_2\ ,\ \ldots\, L_n\,]\,)$$

- Define: stride in the *i-th* dimension

$$stride_i = U_i\ -\ L_i\ +\ 1$$

- Then the address of an arbitrary element $A[E_1\ , E_2\ , \ldots E_n\ ]$ is

$$addr(A[\,E_1\ ,\ E_2\ ,\ \ldots\, E_n\,]) = base + \sum_{i=1}^{n}((E_i - L_i)\cdot \prod_{j=i+1}^{n} stride_j)$$

- To simplify, let: $mult_n$ be the size of an array element (in address units)

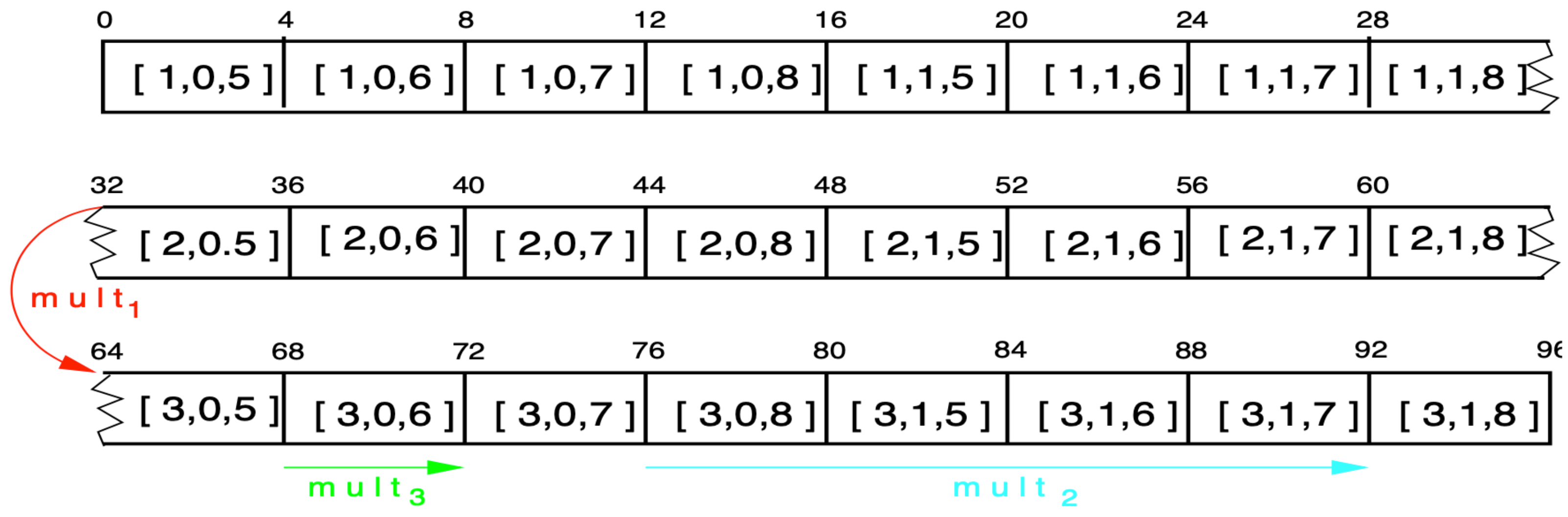
$mult_i\ =\ \prod_{j=i+1}^{n} stride_j\ =\ stride_{i+1}\cdot mult_{i+1}$ then

$$addr(A[\,E_1\ ,\ E_2\ ,\ \ldots\, E_n\,]) = base + \sum^{n}((E_i - L_i)\cdot mult_i)$$

Array Subscripting Example

real B[1 : 3 , 0 : 1 , 5 : 8] ;

i	L_i	U_i	$stride_i$	$mult_i$
0				96
1	1	3	3	32
2	0	1	2	16
3	5	8	4	4



Calculating Array Subscripts Efficiently

- The obvious array subscript calculation based on

$$addr(A[\,E_1\,,\,E_2\,,\,\ldots\,E_n\,]) = base + \sum_{i=1}^{n}((E_i - L_i)\cdot mult_i)$$

Can be done using two registers as

$$\begin{aligned} R_a &\leftarrow base \\ R_b &\leftarrow (\,E_1 - L_1\,) \\ R_b &\leftarrow R_b \cdot mult_1 \\ R_a &\leftarrow R_a + R_b \\ &\ldots \end{aligned}$$

- Can be simplified further if all of the *Li* are zero (e.g. as in C)
- Horners Rule (to optimize array subscript calculation)

$A(x) = a_0 + a_1 x + a_2 x^2 + a_3 x^3 + \ldots$ may be written as

$A(x) = a_0 + x(a_1 + x(a_2 + x(a_3 + \ldots)))$.

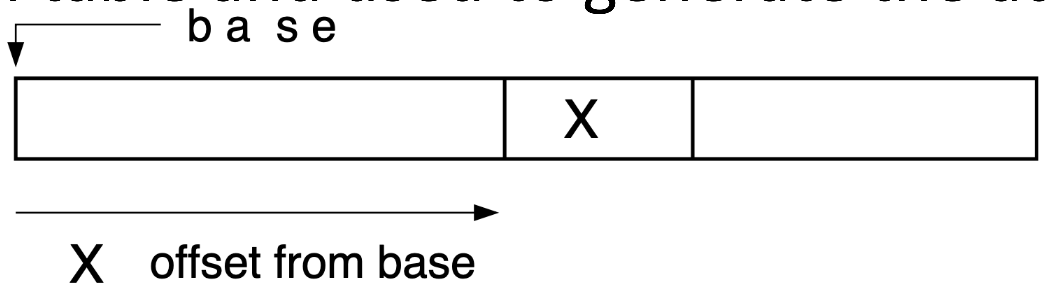
Record and Structure

- It must usually be possible to access any field of a structure given only the base address of the structure and a *compatible* declaration for the structure.
- There are several possible definitions for *compatible declaration*
  - Strict Equivalence The declaration used to access the structure must be the one that was used to declare the structure. (Pascal).
  - Left to Right Equivalence The declaration used to access the structure needs to match the structure declaration only up to the field being accessed. (C, Pl/I)

    Implies: Must map structure fields in left-to-right order. Position of field can not depend on following fields.
  - Major Minor Equivalence Access to any embedded substructure in a structure must be possible given only the address of the substructure and a declaration for the substructure. (most languages)
    Implies: Must use same algorithm for structures and contained substructures
    Position of fields in a substructure can only depend on the substructure.

Record and Structure

- A record or struct is a collection of *fields*. Storage must be allocated for the structure and the layout of the fields must be computed.
- Storage for records and structures can be *packed* or *unpacked*. Data items in a structure can be aligned on their natural hardware boundaries or they can be *unaligned*. Packing usually implies unaligned data. Access to unaligned data is usually much slower than access to aligned data.
- Once the record or structure has been mapped, the offset of each field relative to the start of the record or structure is stored in the compilers symbol table and used to generate the address of the field.



Storage Mapping for Records and Structures

- For most languages, the fields of a structure are laid out in memory in left to right order. *Internal fill* is introduced between scalar items as required so that scalar items are properly aligned on their natural hardware boundaries.
- Most structure storage mapping algorithms try to achieve
  - Proper storage alignment for all scalar data items.
  - Economical use of space, i.e., minimize internal fill.
  - Use *no* internal mapping information (embedded pointers)
  - Allow access to field given only base address and compatible declaration.
  - Support Major/Minor equivalence.
  - Support Left/Right equivalence if required by the programming language.
- If a structure is packed (data items are unaligned) then the fields in a record are laid out in order with no intervening fill

Structure Storage Mapping Algorithm

- A storage mapping algorithm for a structure with *n* fields computes

  $align$ the alignment factor for the entire structure

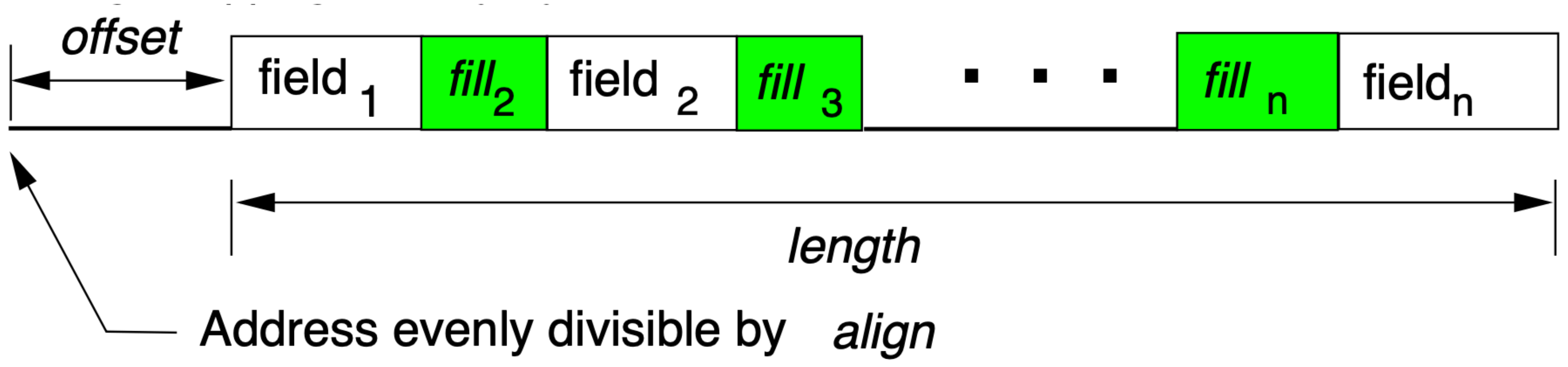
  $offset$ the offset for the entire structure.

  $fill_i$ the amount of fill in front of $field_i, \ 2 \leq i \leq n$

  $length$ the total size of the structure.

  $$length = length_1 + \sum_{i=2}^{n}(length_i + fill_i)$$

- Storage mapping memory layout



Structure Storage Mapping Algorithm

- The reflexive mod function (*rmod*(*x* , *a*)) is defined as:

$$rmod(x, a) = \begin{cases} mod(x, a) & \text{if } a > 0, x > 0 \\ 0 & \text{if } mod(|x|, a) = 0 \\ |a| - mod(|x|, |a|) & \text{otherwise} \end{cases}$$

- Simple mapping algorithm:

```
$align = max_{i=1}^{n} align_i$
$offset = 0$
$length = length_1$
for( i = 2 ; i <= $n$ ; i++ ) {
    $fill_i = rmod(-length, align_i)$
    $length = length + fill_i + length_i$
}
```

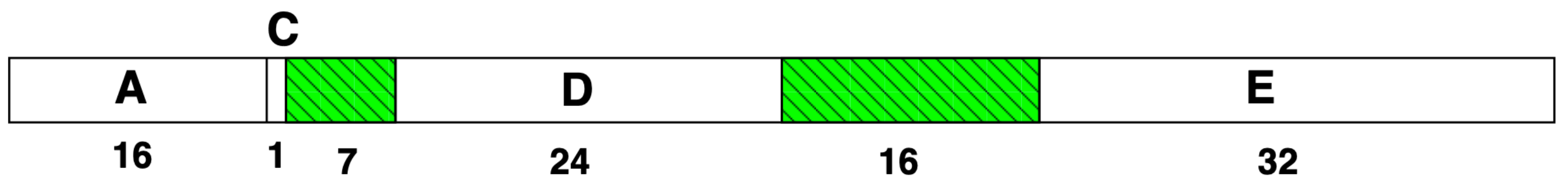
Structure Storage Mapping Algorithm

- A simple algorithm sometimes used for Cobol. Structure is an ordered collection of fields. Substructure is ignored. $length_i$ and $align_i$ determined only by $field_i$.
- This algorithm adds just enough fill in front of each field to make the field properly aligned in memory.
- Works for left-to-right equivalence, but not major-minor equivalence.

Simple Mapping Algorithm Example

```
struct {
      char A[ 2 ] ;              /* item 1 align=8 length=16 */
      struct {
            unsigned C:1 ;        /* item 2 align=1, length=1 */
            char D[ 3 ] ;       /* item 3 align=8, length=24 */
            int E ;              /* item 4 align=32, length=32 */
      } B ;
} X ;
```

name	*i*	*align*	*length*	*offset*	$align_i$	$length_i$	$fill_i$
A	1				8	16	
		32	16	0			0
C	2				1	1	
		32	17	0			0
D	3				8	24	
		32	48	0			7
E	4				32	32	
		32	96	0			16



Depth-First Structure Mapping Algorithm

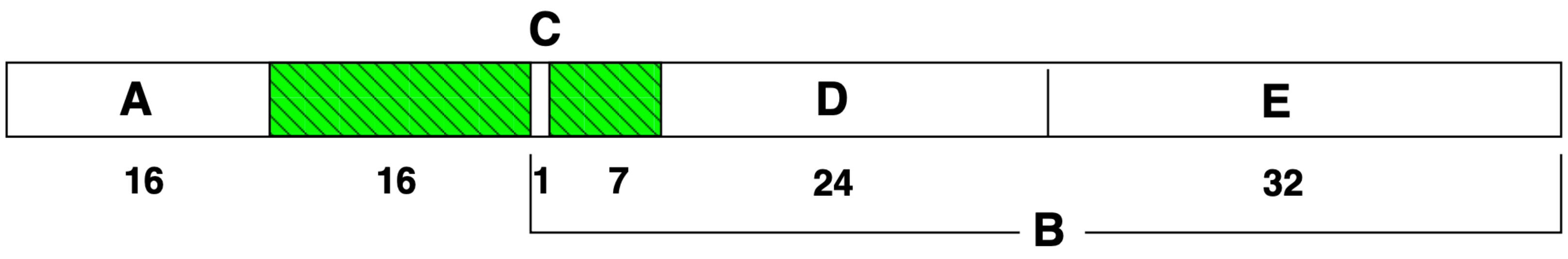
- Most practical structure mapping algorithms try to satisfy Major-Minor equivalence.
- To support Major/Minor equivalence, each embedded record or structure must be mapped separately and independently of its surroundings.
- Structures are mapped *inside/out*, i.e., the most deeply nested sub records or structures are mapped first. The outermost (main) record or structure is mapped only after all embedded records or structures have been mapped.
- Use the simple algorithm but apply it separately to each sub record or structure, processing embedded structures depth first.

UNIVERSITY OF TORONTO

Depth-First Structure Mapping Example

```
struct {    char A[ 2 ] ;                          /* item 1,1 align=8 length=16 */
            struct {        unsigned C:1 ;         /* item 2,1 align=1, length=1 */
                            char D[ 3 ] ;          /* item 2,2 align=8, length=24 */
                            int E ;                /* item 2,3 align=32, length=32 */
            } B ;                                  /* item 1,2 */
} X ;
```

name	*i*	*align*	*length*	*offset*	$align_i$	$length_i$	$fill_i$	$offset_i$
C	2,1				1	1	0	0
		32	1	0				
D	2,2				8	24	7	0
		32	32	0				
E	2,3				32	32	0	
		32	64	0				0
A	1,1				8	16		0
		32	16	0			0	
B	1,2				32	64	16	0
		32	96	0			0	



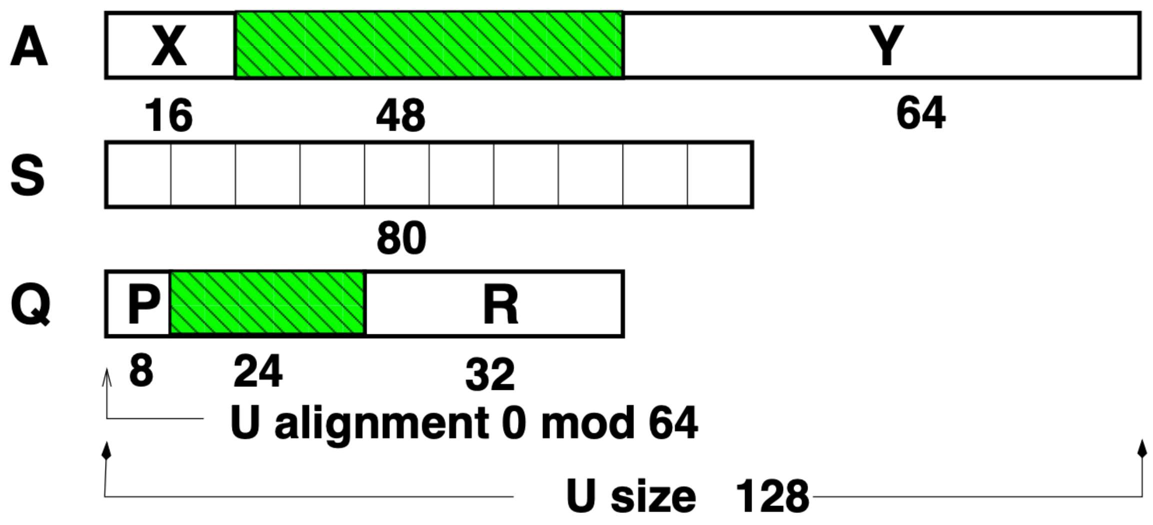
Unions

- The algorithm used for records and structures can also be used for unions and similar constructs like Pascal variant records.
- Strategy:
  - Process each alternative as a separate (implicit) record/structure declaration.
  - Alignment of the entire union is the alignment of the most constrained alternative.
  - Length of the union is the length of the longest alternative.
  - It may be necessary to add fill before alternatives at the start of a union to satisfy the alignment constraints for the alternative. This *might* affect the length of the union.
  - Record separately in the symbol table the offset of each field in each alternative relative to the start of the union.

Union Examples

```
union {
        struct {        short X ;   double Y ;   } A ;
        char S[ 10 ] ;
        struct {        char P ;    int R ;      } Q ;
} U ;
```

Union Field Mappings



Local Storage

- The variables declared in a procedure or function are the *local storage* of the procedure or function.
- In most modern languages, local storage is allocated when the procedure or function is called and deallocated when the procedure or function returns.
- In programming languages that allow recursion, there may be many instances of the local storage for a procedure or function allocated at a given point in the program execution. Each instance is associated with a particular call of the procedure or function.

Local Storage

- In almost all languages, storage allocation for local variables follows a *stack-like* (LIFO) discipline, i.e. the most recently allocated storage is the first to be deallocated.

- Therefore, conceptually at least, a stack is used to manage the allocation and deallocation of local storage.

Local Storage - Activation Records

- The term *activation record* refers to the local storage that is allocated when a function or procedure is called.
- Typically one (or more) hardware registers are used to address the activation record for each procedure or function.

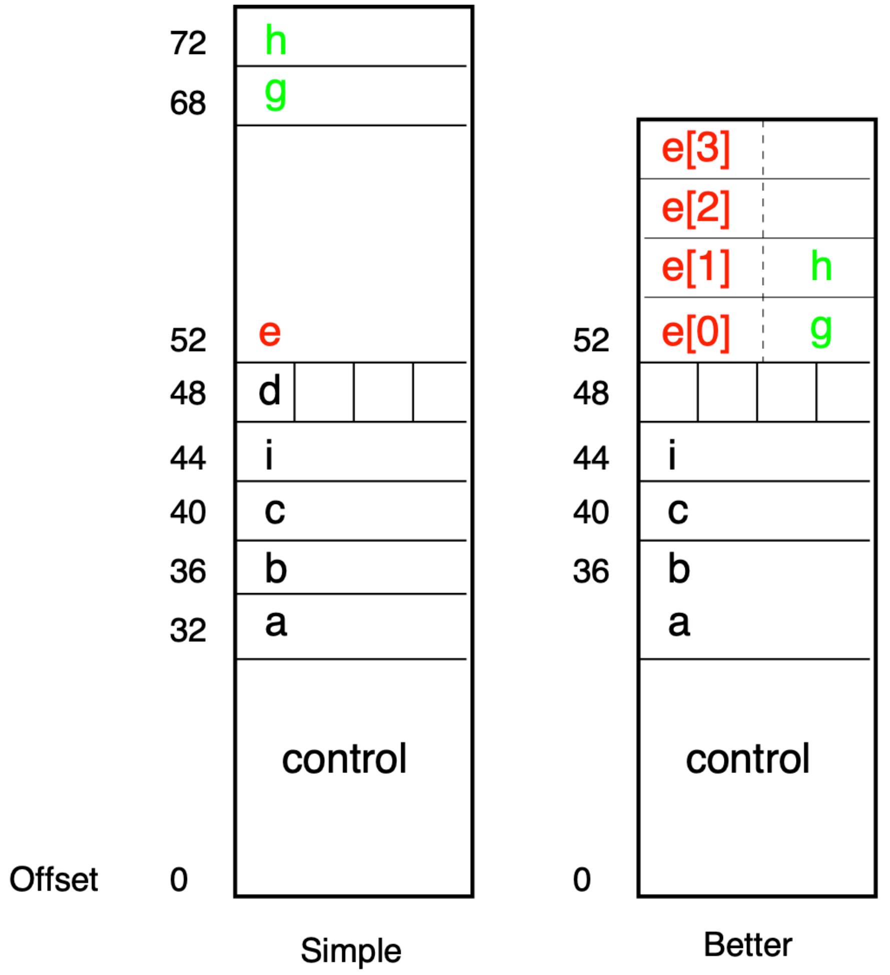
UNIVERSITY OF TORONTO

Local Storage - Activation Records

- A typical activation record contains
  - The return address for the call
  - Control information required to maintain addressing of local storage.
  - Actual parameters passed to this particular call of the procedure or function
  - Storage for the local variables declared in the procedure or function.
  - Pointers to storage for variables that were dynamically allocated, .e.g arrays with dynamic bounds. Storage for the dynamic arrays may also be in the activation record.
  - Temporary storage required for expression evaluation.
  - Storage for variables from *micro scopes* contained in the procedure or function.

Activation Record Example

```
int  F(  int *  a ,  float  b )   {
        int   c , i ;
        char  d[ 4 ] ;
        . . .
        if  a < 0  then {
             float  e[ 4 ] ;
               . . .
        }
        . . .
        while  i < a  do {
             int  * g , h  ;
               . . .
        }
      . . .
}
```



Dynamically Allocated Storage

- Many languages allow the programmer to dynamically allocate storage for arbitrary data structures. e.g. **new** in C++.
- The compiler typically turns programming language storage allocation constructs into calls on a compiler generated internal function that actually manages the storage.
- Possible storage allocation strategies:
  - Let the underlying OS do all the storage management.
  - Allocate a large block of storage at program initialization, and suballocate parts of it to the program on request. One typical approach is to allocate a large block of memory and grow the stack of activation records from one end and dynamically allocate memory from the other end.

Dynamically Allocated Storage

- **Option 1**: Programmers are responsible for deallocating memory (C, C++).
  - Easy to implement as compilers.
  - Runtime library only needs to wrap OS system calls for managing dynamic allocation heap.
  - Error-prone for novice programmers. Memory leak, double free, dangling pointers, etc.
- **Option 2**: Deallocations are managed by language runtime via garbage collection (GC) (Java, C#, etc.).
  - Need to implement a runtime garbage collector
  - Performance overhead and stalls to program execution

Dynamic Memory Management

- Management of dynamically created runtime storage is a central issue in the implementation of many programming languages.
- Most implementations allocate storage in a *heap* with some mechanism for reclaiming memory that is no longer in use.
- GC automatically manages dynamically allocated memory:
  - Identify dynamically allocated memory that is no longer in use.
  - Recycle that memory for later reuse
- Some programming language require Garbage Collection (Java, Lisp), many others allow it.

Garbage Collection Design Issues

- Roots and pointer finding. Is it possible to find all pointers (roots) to allocated objects?
- Mobility of data. It is only possible to move a data object if it is possible to (eventually) update *all pointers* to the object to point at the new location.
- Immediacy of storage reclamation. Is recyclable memory made available immediately or only periodically.
- Program tolerance to interruption and/or arbitrary delay. Can mutation and collection proceed concurrently?
- Processing Cost – Time and Space Overhead
- Recovering cyclic data structures.

Basic Principle

- The key concept for garbage collection is reachability. A program can only access some object if it has a memory address (i.e. pointer) for the object. An object that has been allocated but is no longer reachable, is *garbage* and may be recycled.
- *Roots* are base locations that may hold the address of dynamically allocated objects. Possible roots include: processor registers, all kinds of variables and temporaries in activation records.
- Dynamically allocated memory can only be accessed through some chain of pointers starting with some root. An object is *reachable* if and only if there is some path from a root to the object.
- Two separate issues:
  - identifying reachable objects
  - managing and recycling memory efficiently.

Classic Dynamic Memory Design Approaches

- *Reference Counting* - Every allocated object contains a *reference count*. This count is incremented when a new pointer to the object is created and decremented when a pointer to the object is destroyed. A reference count of zero implies the object has no pointers pointing at it and so it can be reclaimed. Incremental algorithm.
- *Mark and Sweep Collection* - A garbage collector algorithm starts at all roots recursively marking all nodes that can be reached. When this algorithm finishes, any nodes that are not marked can be reclaimed.
- *Conservative marking* , takes more space, but use a simpler algorithm to identify pointers. Strategy:
  - Sweep all available memory, test each word for the *isPointer* property
  - Unless the collector can (heuristically) decide that a word does not contain a pointer, assume that it is a pointer.

Reference Counting

- Used in many languages and applications:
  - Smalltalk, InterLisp, Java, Adobe PhotoShop, C++, Rust
- Storage management runs concurrently intermixed with program execution.
- Invariant: the reference count of each node is equal to the number of pointers pointing to the node.
- Can do optimizations on pointers local to a routine to reduce the amount of reference counting.

Reference Counting - Disadvantages

- High processing cost. Every pointer assignment requires reference count updating.
- Correctness is difficult. Missing even one reference counter update could lead to errors.
- Difficult to deal with embedded pointer assignments, e.g. bulk assignment of a structure containing pointers.
- Reference count roughly doubles the size of every pointer.
- Reference counting schemes can not reclaim cyclical data structures, e.g., a doubly linked list.
- Need some strategy to deal with reference counter overflow.

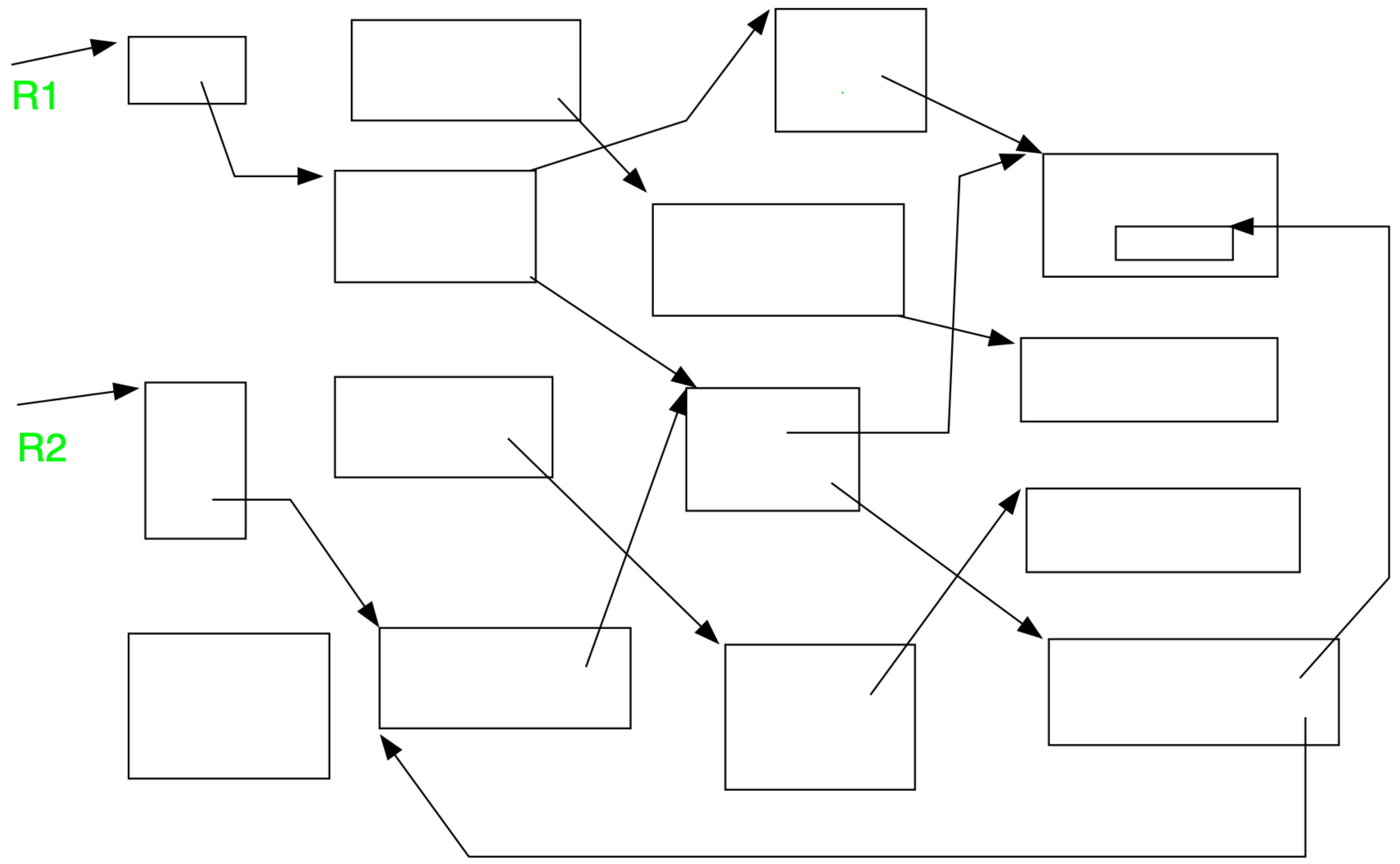
Naive Mark Sweep Algorithm

- Basic algorithm:
  - Suspend program processing.
  - Start at all roots. The algorithm must be able to find **all** roots, e.g. even roots in registers.
  - Trace nodes reachable from the roots and mark them.
  - If trace reaches a node that is already marked, stop tracing on that path.
  - After marking is complete, make another sweep through the heap unmarked nodes can be reclaimed. Reset mark on marked nodes for next sweep.
  - Restart program execution.
- Mark and Sweep is usually triggered asynchronously when the program makes a request for memory that cannot be satisfied.

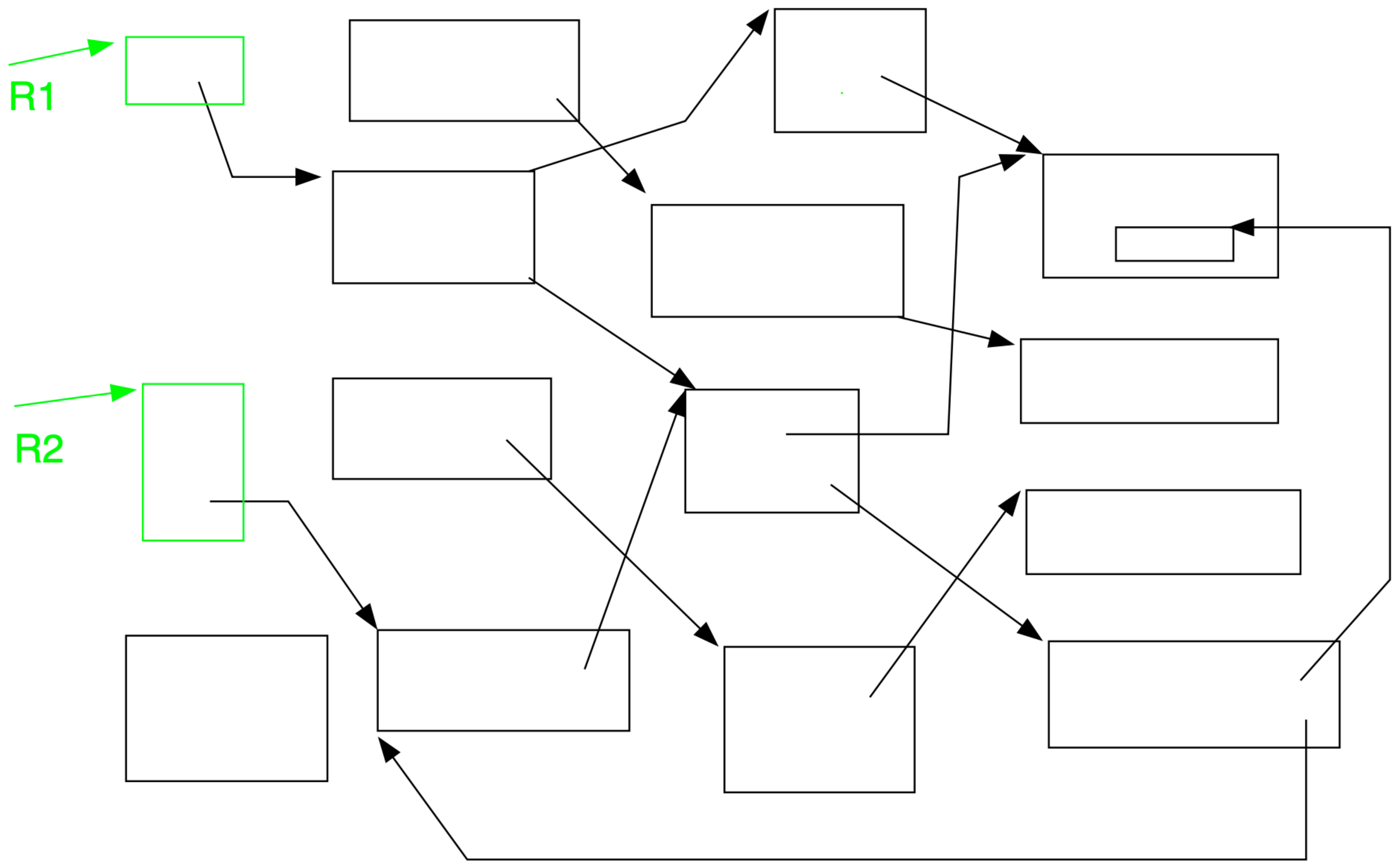
Naive Mark Sweep Algorithm

- Advantages:
  - Cycles handled easily.
  - No overhead on pointer manipulation.
  - No space overhead on pointers.
  - Handles bulk assignment correctly.
- Disadvantages:
  - Stop - Start algorithm. Unsuitable for interactive applications.
  - Guaranteeing that all roots can be found is *hard*
  - Need to disable collector in critical sections.
  - Time is proportional to size of heap, not number of active cells. Algorithm will tend to *thrash* as heap occupancy increases.
  - Without optimizations leads to highly fragmented memory.

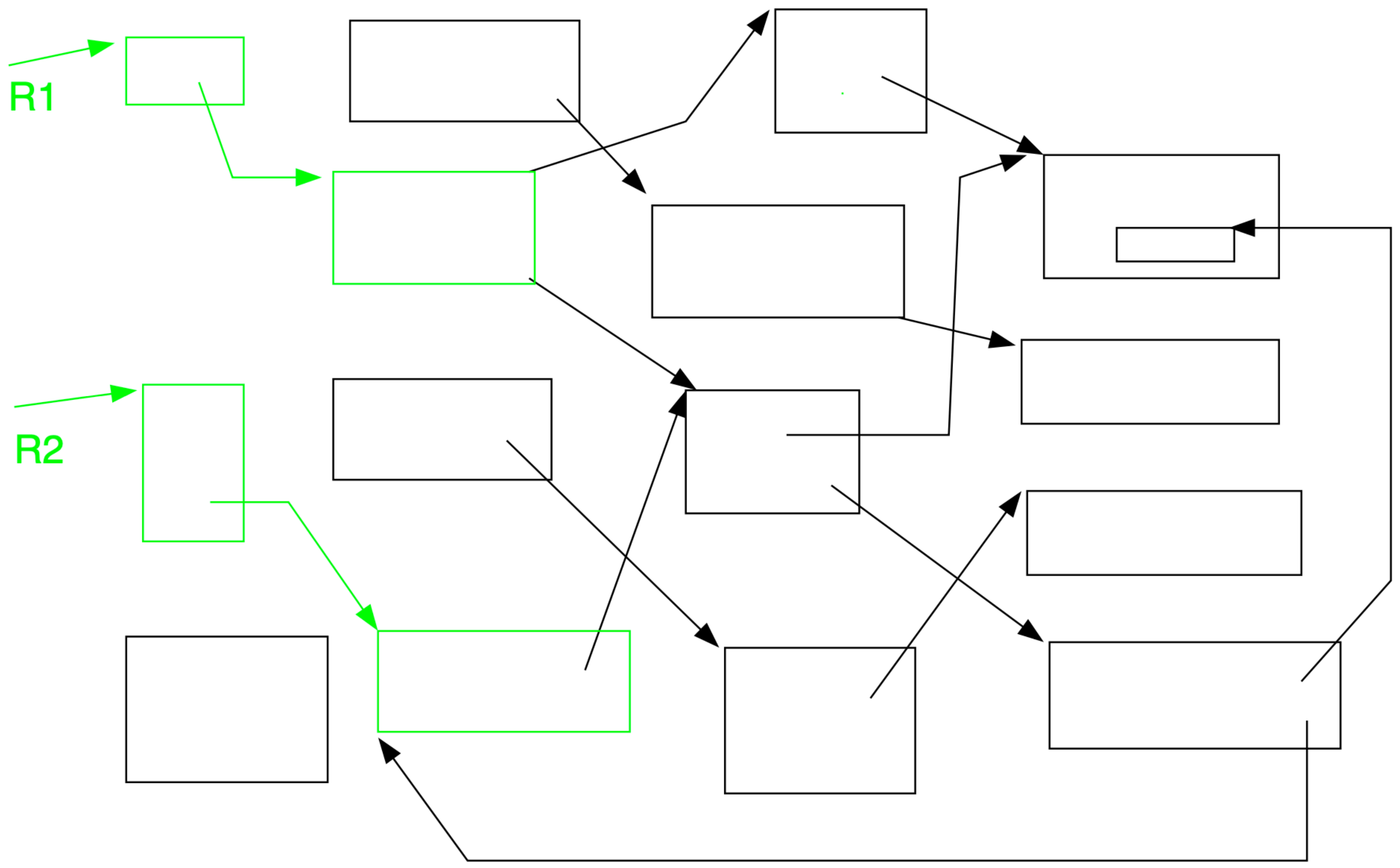
Mark Sweep Example



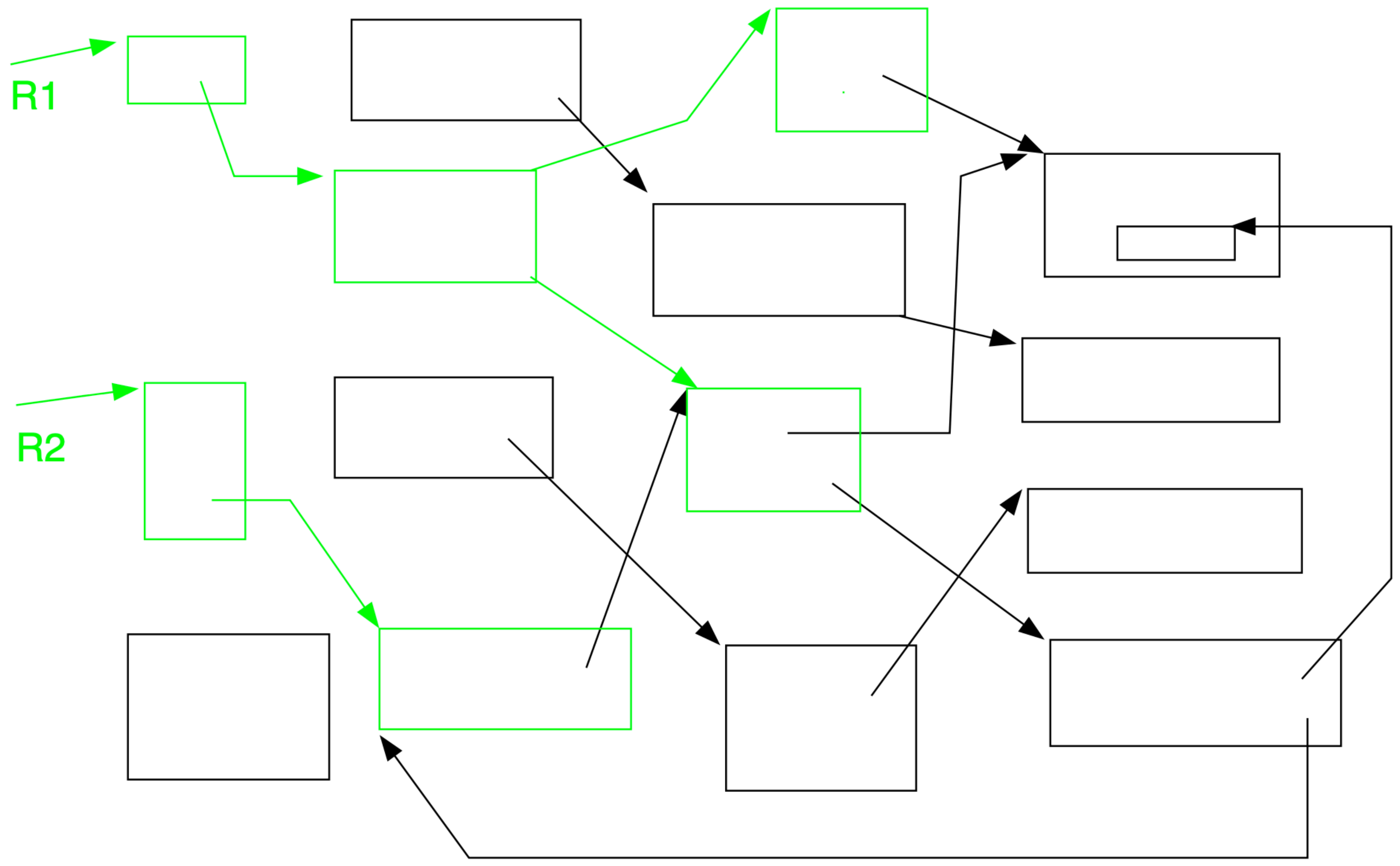
Mark Sweep Example



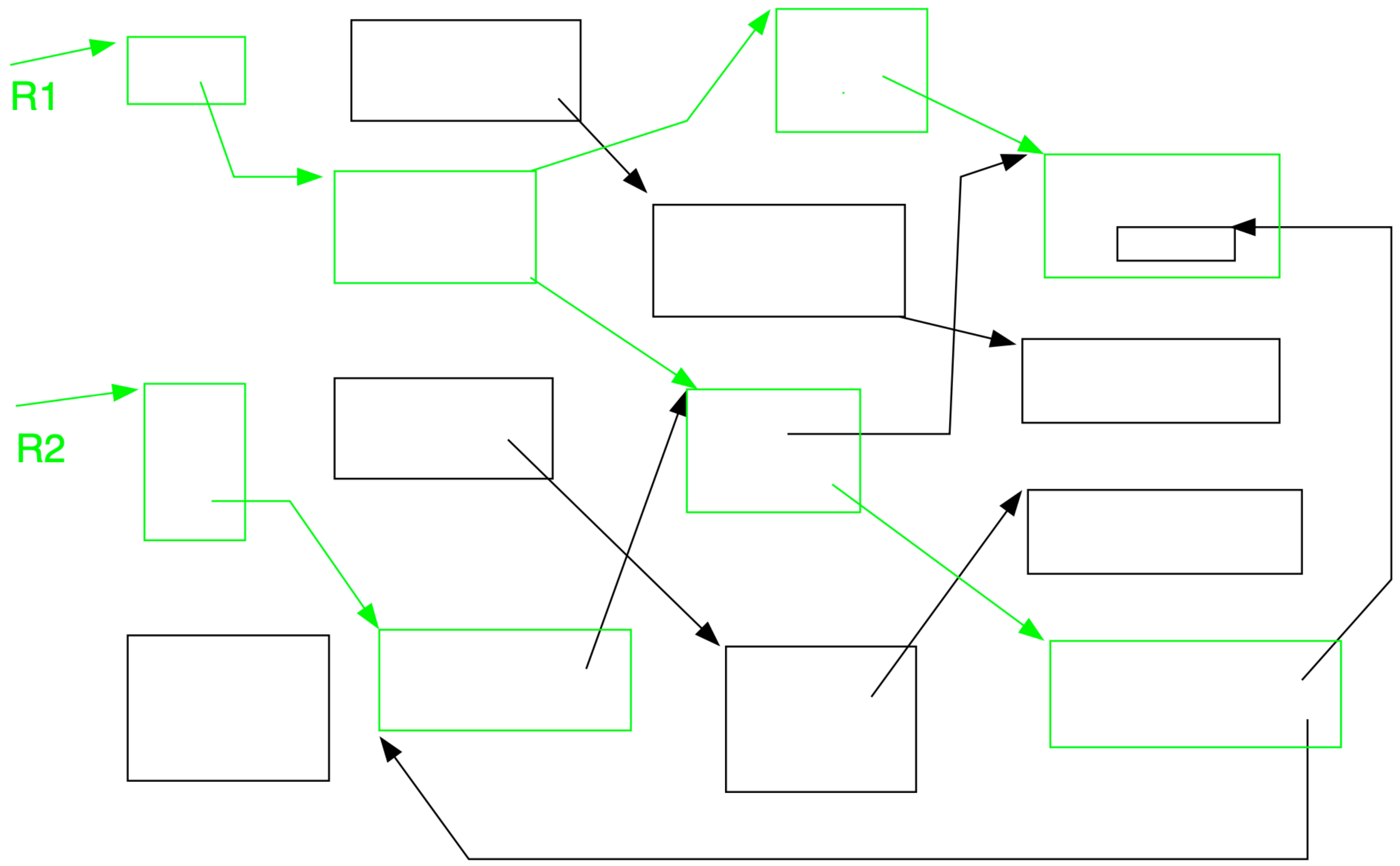
Mark Sweep Example



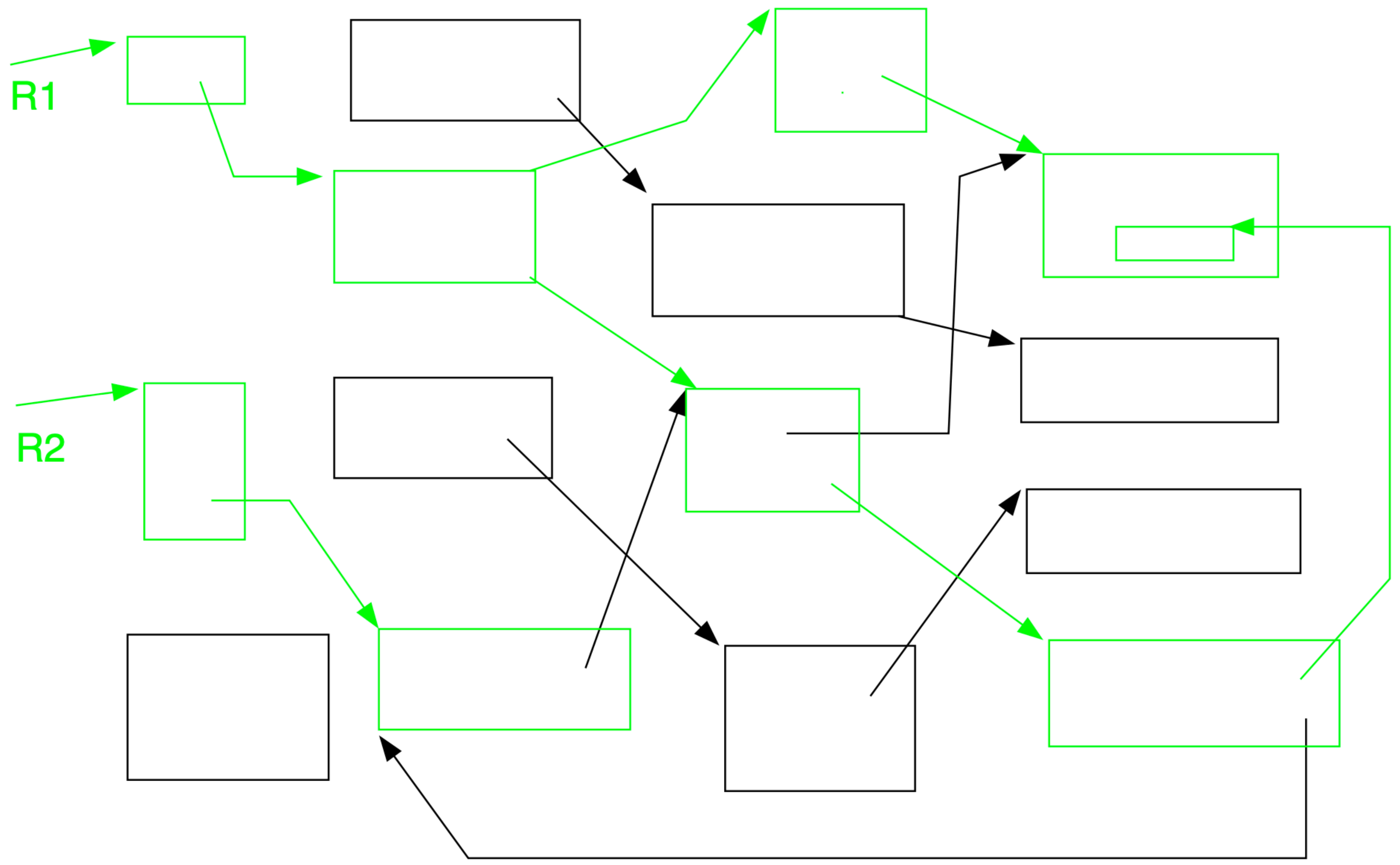
Mark Sweep Example



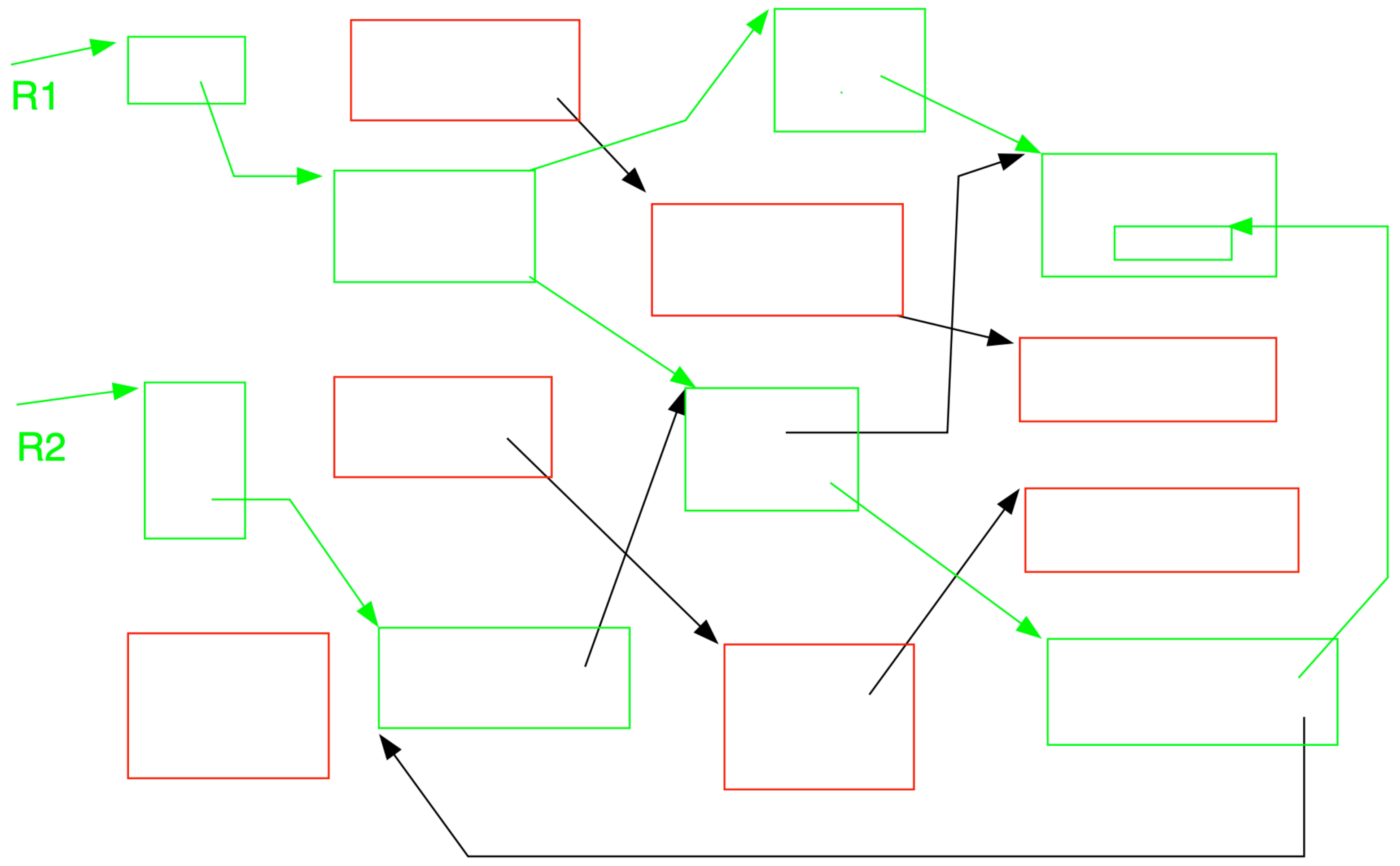
Mark Sweep Example



Mark Sweep Example



Mark Sweep Example

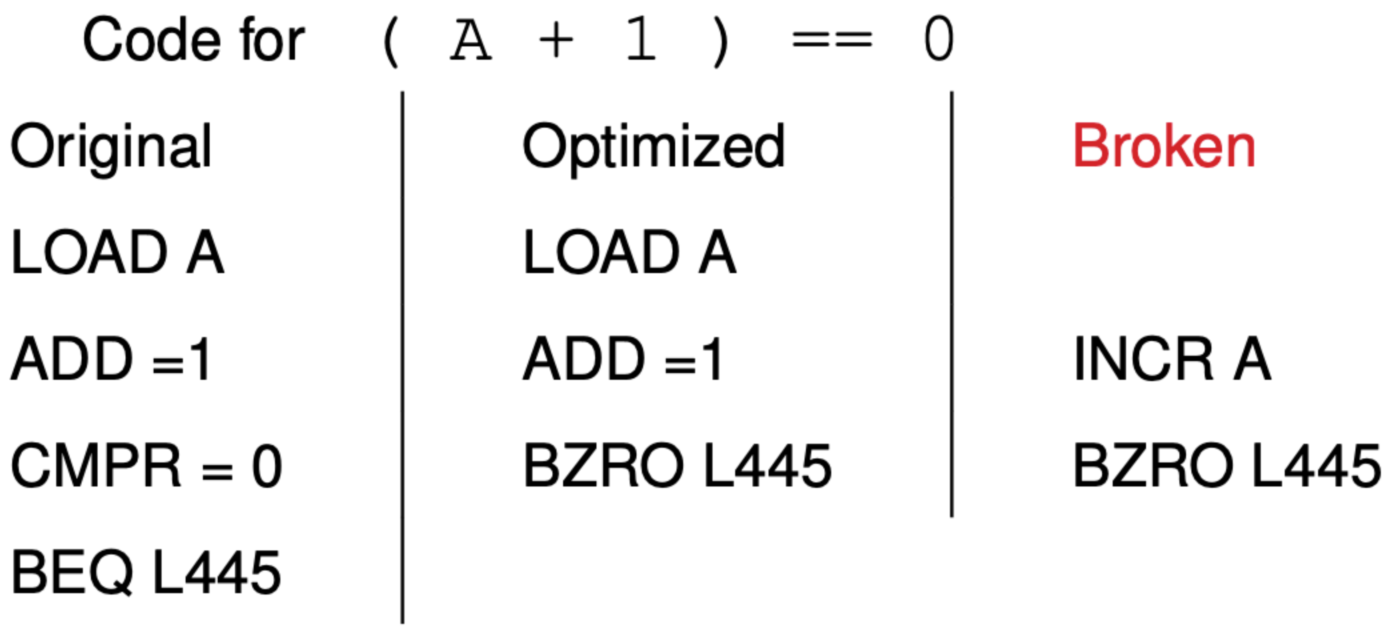


Code Generation – Correctness and Consistency

- The code generation design must correctly implement all of the programming language.
- All of the individual code generation designs choices must lead to a *complete* and *consistent* whole.
- The key issue is that the *entire* hardware state at any point in code generation must be correct for the code generation that follows. State includes not only named registers but also internal state (e.g. condition codes).

Code Generation – Correctness and Consistency

- Need to be careful that optimizations don’t introduce errors in generated code. Example: using test of condition code to avoid compare against zero is broken if expression gets optimized so that condition code isn’t set.



Code for `( A + 1 ) == 0`

Original	Optimized	Broken
LOAD A	LOAD A	
ADD =1	ADD =1	INCR A
CMPR = 0	BZRO L445	BZRO L445
BEQ L445		

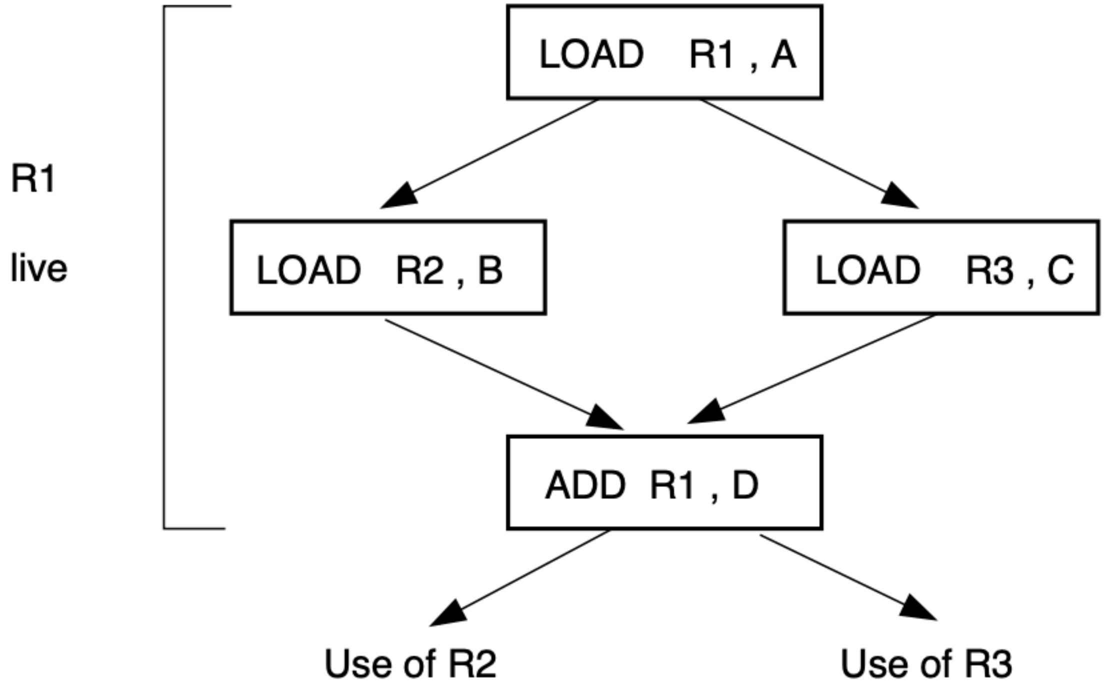
Code Generation Design

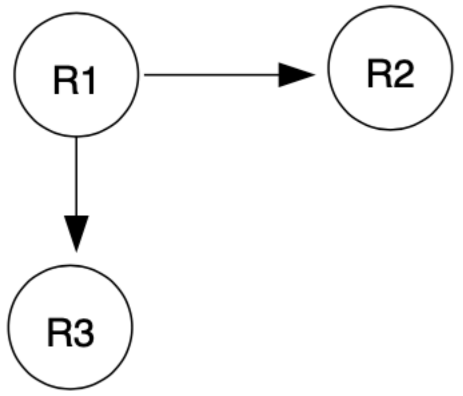
- Instruction Selection
  - Map each IR instruction to one or more machine instructions
  - Interacts with register allocation, i.e., instructions require specific registers
- Register Allocation
  - Map large number of virtual registers in IR to a limited number of hardware registers.
  - Manage register spills for those that cannot fit into hardware registers.
  - Some registers will be dedicated to specific purposes like the display, routine call and return. Often determined by hardware/OS conventions.
  - Optimizations: minimize registers used and minimize spills.

Register Allocation

- IR typically has infinite number of virtual registers (R_i s).
- Perform a Live Variable Analysis on the virtual regiters.
  - A register is live from where it is given a value to where it is last used.
- Build an *interference graph* of the virtual registers
  - A register X interferes with a register Y if X lives at the point of definition for Y.
  - An interference graph has a node for each virtual register and an edge between each pair of registers that interfere with one another.
  - If there are *N* registers available, any node with less than *N* edges connected to it can be assigned a register.

Inteference Graph Example





Register Allocation by Graph Coloring

- Attempt to color the interference graph with *N* colors where *N* is the number of available registers.
  - Nodes in the graph that are connected by an edge cannot have the same color.
  - A successful coloring of the graph corresponds to an assignment of the pseudo registers to real registers.
  - If *N*-coloring succeeds all registers have been assigned.
  - If *N*-coloring fails, find a region with high register requirements and *spill* some pseudo registers into memory.
- Graph Coloring is an NP-hard problem. Apparently linear heuristics have been developed by Chaitin, Chow and others.

Register Spilling

- Selection of registers to be spilled should take into account the cost of loading and storing registers as well as the gain from having data in a register.
- Spilling strategy:
  - Pick a register *R* to be spilled. Allocate a memory location (*Rtmp*) to spill into. Usually in the current activation record.
  - Before each operation that reads R insert a LOAD from the activation record.
  - After each operation that writes to R insert a STORE to the activation record.
  - Recalculate the interference graph and try recoloring
  - Multiple spills might be required to achieve colorability.

Register Spilling Considerations

- Spilling a register R reduces its live range and thus its interference with other registers.
- Any choice of register to spill is correct, but the choice may impact program performance.
- Possible spill heuristics
    - Avoid spilling in inner loops.
    - Spill registers with the largest number of conflicts.
    - Spill registers with the smallest number of definitions and uses.
    - ...

Code Generation Template

- For each possible instruction in the IR, design a template describing target machines instructions to implement the quadruple.
- Templates will have substitutable parameters for operands and results.
- Template expansion mechanism uses conditional selection to deal with possible operand/result locations and possibly operand type dependent code.
- Use conditional selection to generate special case code for local optimizations.

Code Generation Template

- Instruction selection can involve deep decision trees, e.g. large IBM mainframes have at least 17 instructions which perform addition, choice is based on operand types and operand locations.

- The simple versions of this approach are unable to use the *context* of the quadruple to optimize code selection
  For example, could generate INC J for J = J + 1

Template Example

- LLVM IR: %dest = add nsw %left, %right

- leftOperand and rightOperand could be
  - Literal constants
  - LLVM virtual registers
  - Memory (because virtual registers spilled into the activation record)

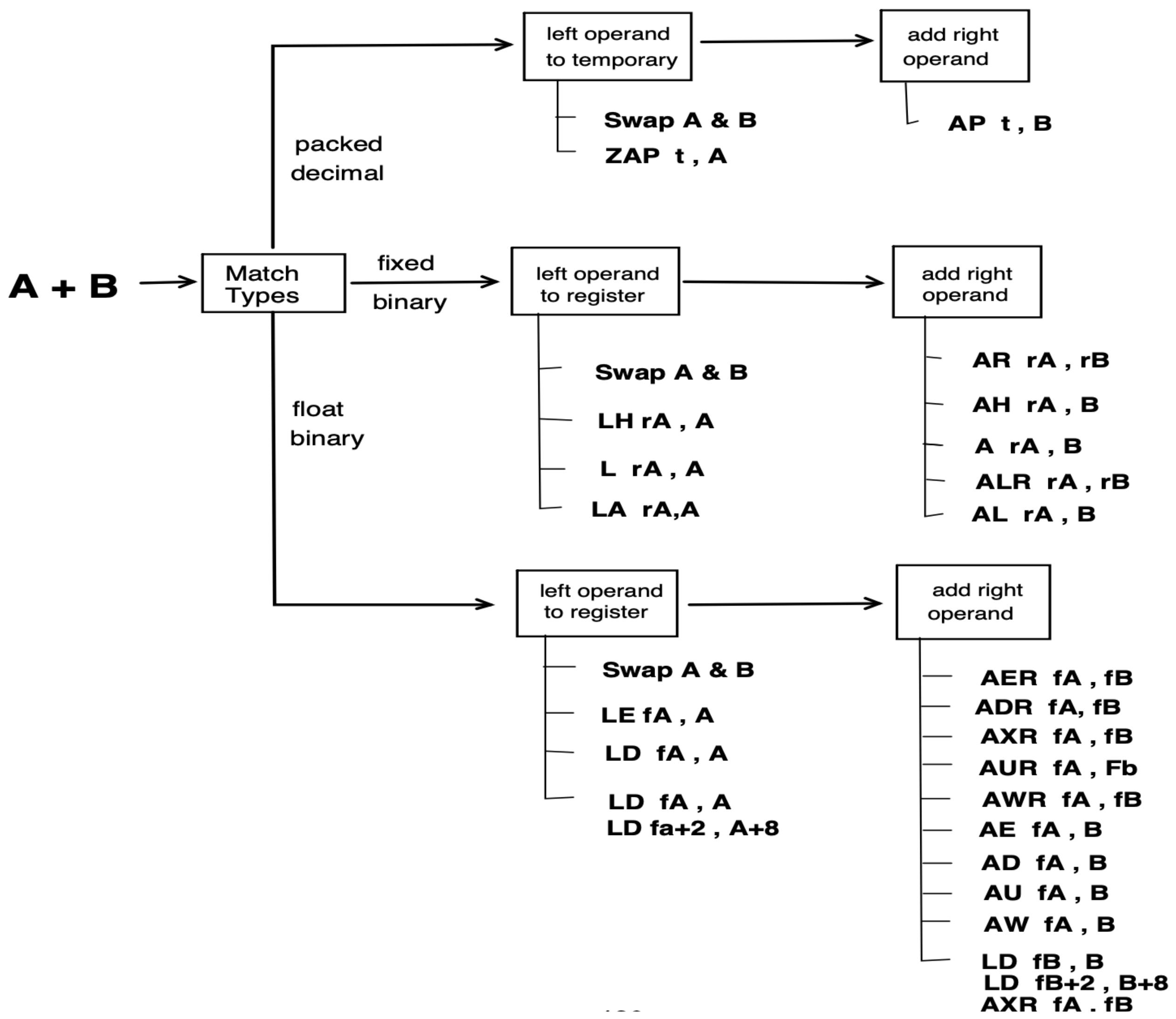
- Result could be
  - LLVM virtual registers
  - Memory (because virtual registers spilled into the activation record)

Template Example for add IR

- Perform addition of left and right, result decribes output.
- Assumes registers can be overwritten, addition is commutative/

$left$	$right$		
	literal	register	memory
literal	$result =$ $left + right$	LIT $tmp_{reg}, = left$ ADDR $right_{reg}, tmp_{reg}$ $result = right_{reg}$	LIT $tmp_{reg}, left$ ADD $tmp_{reg}, right$ $result = tmp_{reg}$
register	LIT $tmp_{reg}, right$ ADDR $left_{reg}, tmp_{reg}$ $result = left_{reg}$	ADDR $left_{reg}, right_{reg}$ $result = left_{reg}$	ADD $left_{reg}, right$ $result = left_{reg}$
memory	LIT $tmp_{reg}, right$ ADD $tmp_{reg}, left$ $result = tmp_{reg}$	ADD $right_{reg}, left$ $result = right_{reg}$	LOAD $tmp_{reg}, left$ ADD $tmp_{reg}, right$ $result = tmp_{reg}$

Example – IBM 370 Code Gen for A + B



```
Swap A & B
ZAP  t , A

AP  t , B

Swap A & B
LH rA , A
L  rA , A
LA  rA,A

AR  rA , rB
AH  rA , B
A  rA , B
ALR  rA , rB
AL  rA , B

Swap A & B
LE fA , A
LD  fA , A
LD  fA , A
LD fa+2 , A+8

AER  fA , fB
ADR  fA, fB
AXR  fA , fB
AUR  fA , Fb
AWR  fA , fB
AE  fA , B
AD  fA , B
AU  fA , B
AW  fA , B
LD  fB , B
LD  fB+2 , B+8
AXR  fA . fB
```

Implementing Code Selection

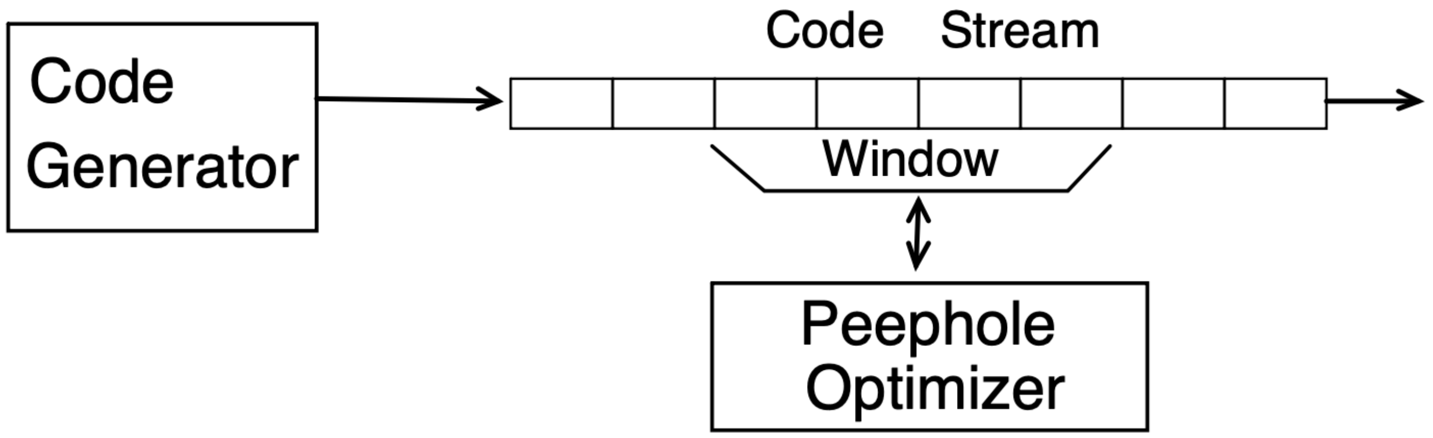
- For complicated (CISC) target machines, instruction selection during code generation can involve very complicated decision processes.
- RISC machines pose a different set of problems, instruction selection is often trivial, but optimal instruction ordering and optimizing around load/store and branch latencies is a major issue.
- Peephole optimization for local optimization on the fly.

Code Generation Optimizations

- Generating *locally good* code is always a winning strategy.
- Take advantage of constant information known to the compiler.
- Use targeting to encourage results to end up in desirable locations (e.g. registers for parameter passing).
- Use special instructions (e.g. literal instructions, register to register instructions) wherever possible for faster code.
  - Use literal instructions (e.g. AddImmediate) for faster arithmetic.
  - Use register-to-register instructions for speed.
  - Be clever about using instructions in unobvious ways.

Peephole Optimizations

- Peephole optimization is a technique for making local improvements to the code generated by a simple code generator.
- Peephole optimizer examine the code stream as it is generated. Uses a 2 .. 4 instruction window to search for particular instruction sequences that it can optimize. Usually table/template driven.
- Issues: Peephole optimizer must be aware of control flow. Can't resolve branch addresses until after peephole optimization.



Peephole Optimization Examples

```
ST    3,X     ⇒   ST  3,X     System/370
L     3,X

MOV   X,R1    ⇒   INC  X      PDP-11
ADD   #1,R1
MOV   R1,X

L     2,Y     ⇒   MVC  Z,Y    System/370
ST    2,Z

BR    * +1    ⇒               All
```

Q/A?

UNIVERSITY OF TORONTO