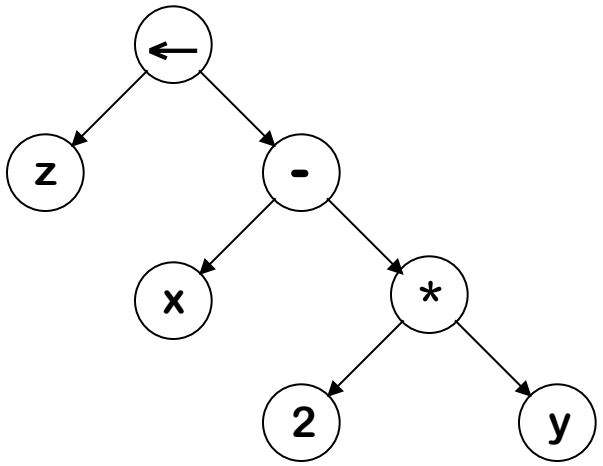
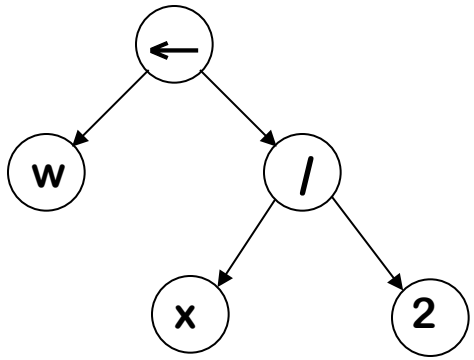
Intermediate Representations
Part II

Directed Acyclic Graph

A directed acyclic graph (DAG) is an AST with a unique node for each value



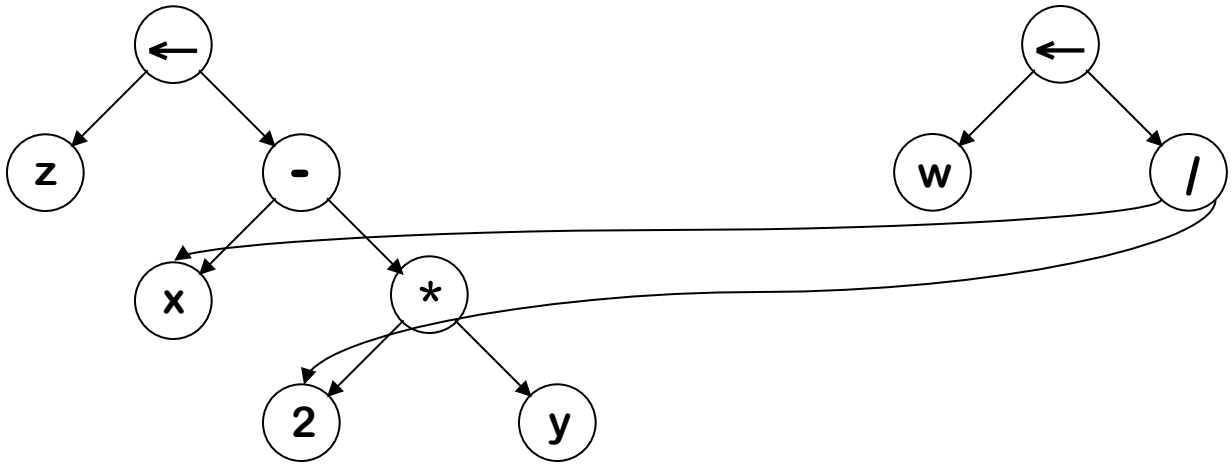


z ← x - 2

w ← x / 2

Directed Acyclic Graph

A directed acyclic graph (DAG) is an AST with a unique node for each value



z ← x - 2

w ← x / 2

Stack Machine Code

Originally used for stack-based computers, now Java

- Example:

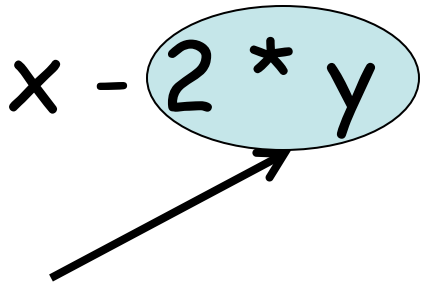
x - 2 * y becomes

```
push x
push 2
push y
multiply
subtract
```

Stack Machine Code

- Operations take operands from a stack
- Compact form
- A form of one-address code
- Introduced names are *implicit*, not *explicit*
- Simple to generate and execute code

Stack Machine Code Advantages



Result is stored in a temporary! Explicit name for result.

```
push 2
push y
multiply
push x
subtract
```

Multiply pops two items off of stack and pushes result! Implicit name for result

Three Address Code

Different representations of three address code

- In general, three address code has statements of the form:

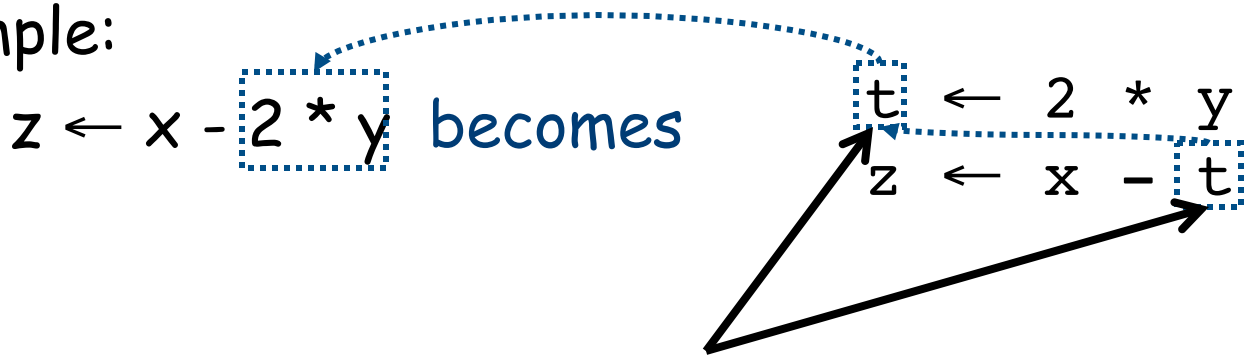
$$x \leftarrow y \ op \ z$$

With 1 operator (*op*) and

(at most) 3 names (x, y, & z)

Three Address Code

Example:



Explicit name for result.

Three Address Code Advantages

- Resembles many real (RISC) machines
- Introduces a new set of names
- Compact form

Three Address Code: Quadruples

Naïve representation of three address code

- Table of k * 4 small integers

```
load   r1, y
loadI  r2, 2
mult   r3, r2, r1
load   r4, x
sub    r5, r4, r3
```

RISC assembly code

Destination

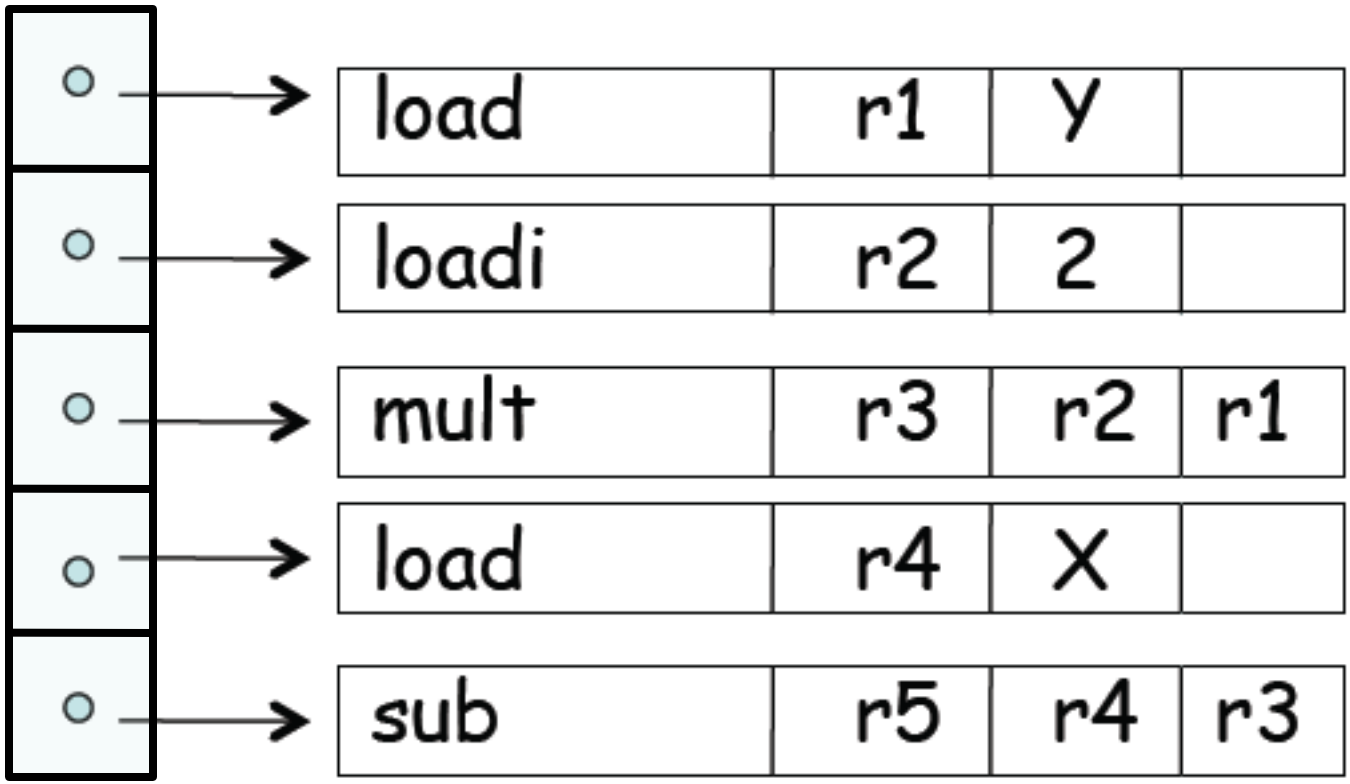
Two operands

load	1	y	
loadi	2	2	
mult	3	2	1
load	4	x	
sub	5	4	3

Quadruples

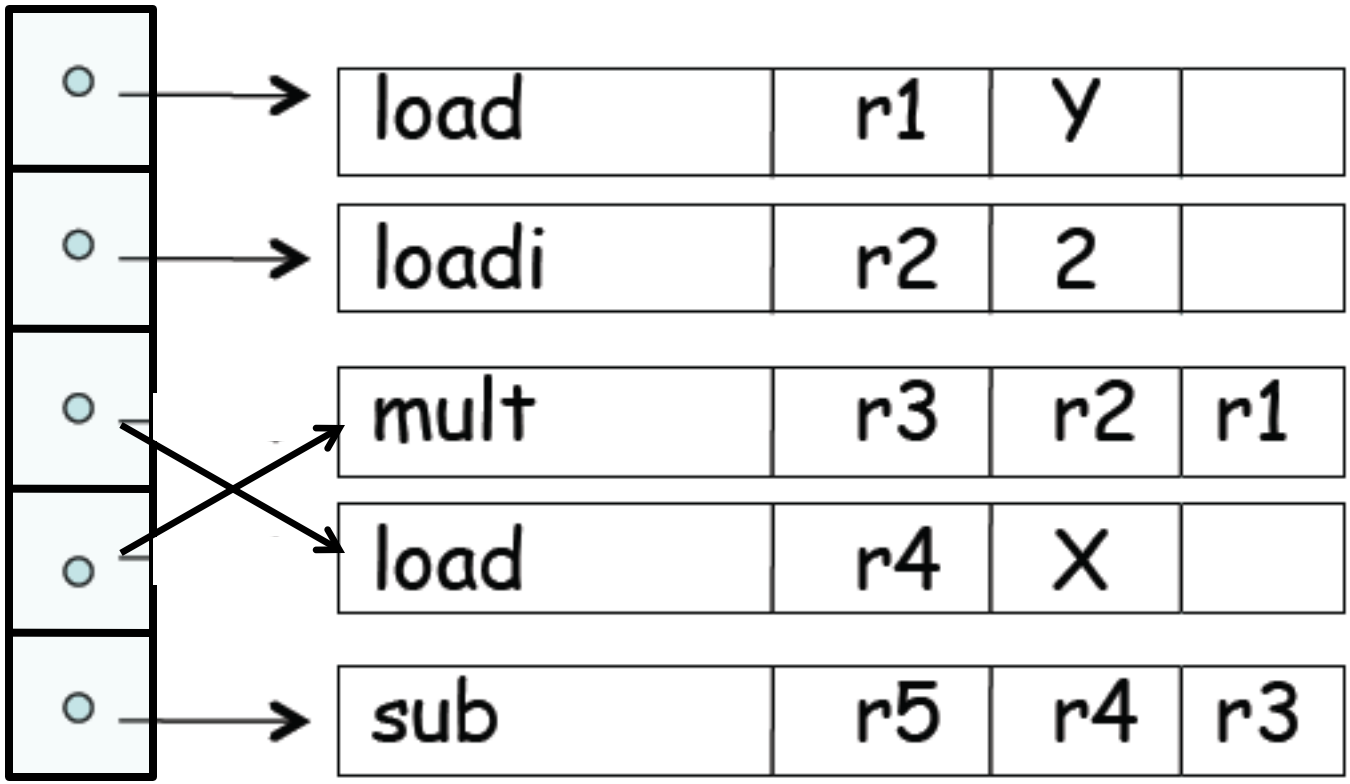
Three Address Code: Array of Pointers

- Index causes level of indirection
- Easy (and cheap) to reorder
- Easy to add (delete) instructions



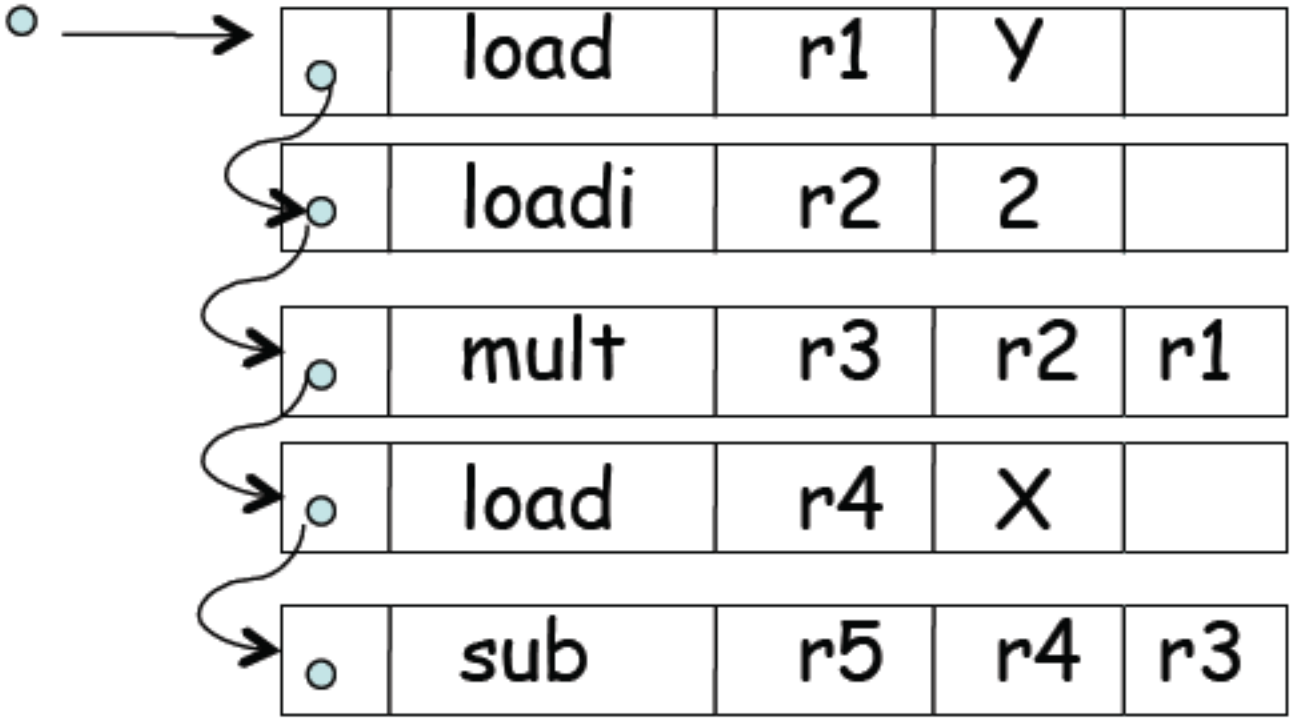
Three Address Code: Array of Pointers

- Index causes level of indirection
- Easy (and cheap) to reorder
- Easy to add (delete) instructions



Three Address Code: Array of Pointers

- No additional array of indirection
- Easy (and cheap) to reorder than simple table
- Easy to add (delete) instructions

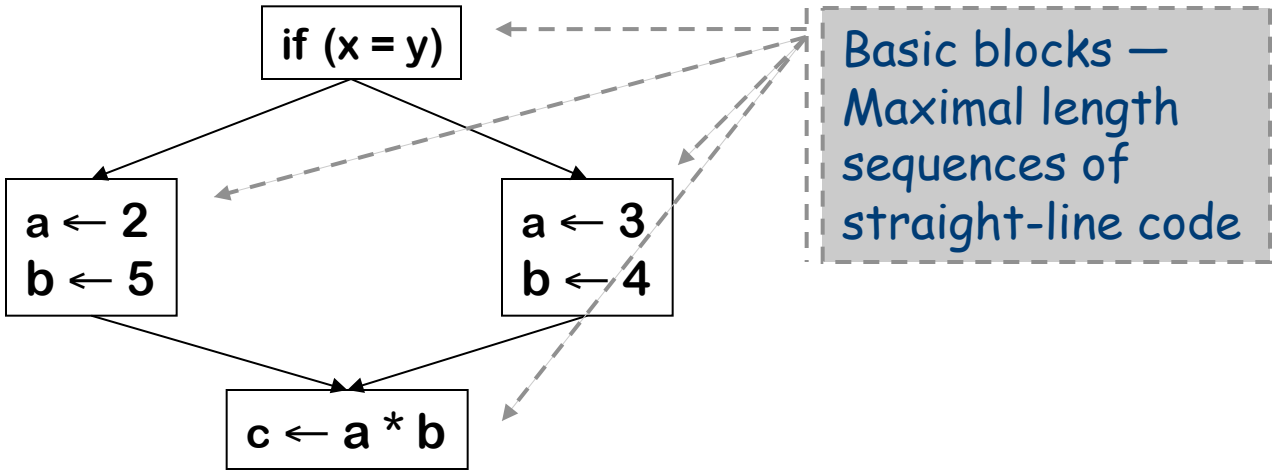


Control-flow Graph

Models the transfer of control in the procedure

- Nodes in the graph are basic blocks
  - → Can be represented with quads or any other linear representation
- Edges in the graph represent control flow

Example



Control-Flow Graphs

- Node: an instruction or sequence of instructions (a **basic block**)
  - → Two instructions i, j in same basic block *iff* execution of i *guarantees* execution of j
- Directed edge: *potential* flow of control
- Distinguished start node *Entry*
  - → First instruction in program

Identifying Basic Blocks

- Input: sequence of instructions *instr(i)*
- Identify **leaders**:
  first instruction of basic block
- Iterate: add subsequent instructions to basic block until we reach another leader

Basic Block Partition Algorithm


```
leaders = instr(1)          // first instruction

for i = 1 to |n|            // iterate thru all instrs
   if instr(i) is a branch
       leaders = leaders ∪ targets of instr(i)
       leaders = leaders ∪ instr(i+1) // instr after
   branch

worklist = leaders
While worklist not empty
   x = first instruction in worklist
   worklist = worklist - {x}
   block(x) = {x}
   for (i = x + 1; i <= |n| && i not in leaders; i++)
       block(x) = block(x) ∪ {i}
```

Static Single Assignment Form

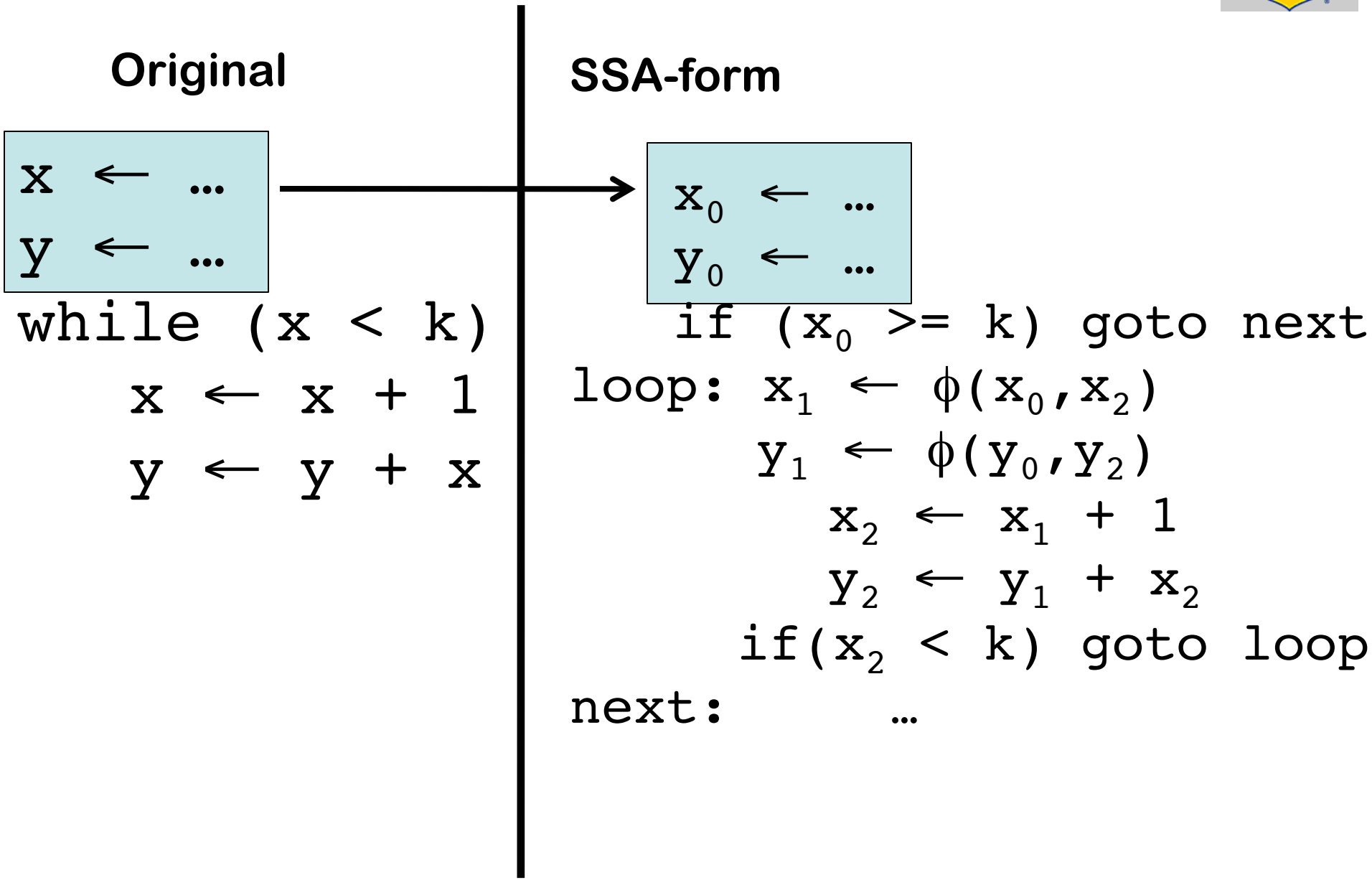
Original

```
x ← …
y ← …
while (x < k)
    x ← x + 1
    y ← y + x
```

SSA-form

```
    x0 ← …
    y0 ← …
    if (x0 >= k) goto next
loop: x1 ← φ(x0,x2)
      y1 ← φ(y0,y2)
         x2 ← x1 + 1
         y2 ← y1 + x2
      if(x2 < k) goto loop
next:      …
```

Static Single Assignment Form



Static Single Assignment Form

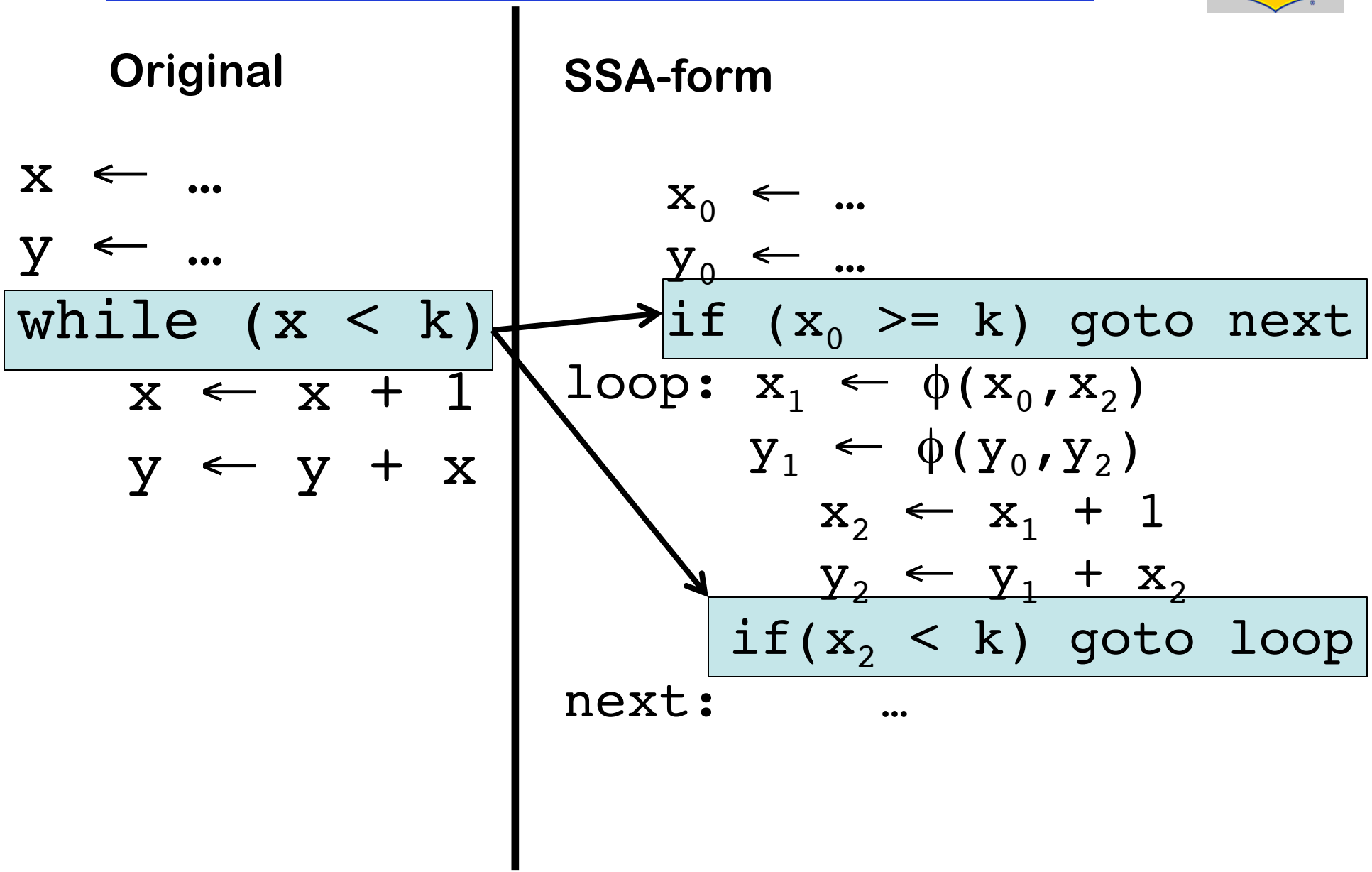
Original

```
x ← …
y ← …
while (x < k)
    x ← x + 1
    y ← y + x
```

SSA-form

```
    x0 ← …
    y0 ← …
    if (x0 >= k) goto next
loop: x1 ← ϕ(x0,x2)
      y1 ← ϕ(y0,y2)
      x2 ← x1 + 1
      y2 ← y1 + x2
    if(x2 < k) goto loop
next:    …
```

Static Single Assignment Form



Static Single Assignment Form

Original

```
x ← …
y ← …
while (x < k)
    x ← x + 1
    y ← y + x
```

This is new! Inserted at points where diff. control flow merge.

SSA-form

```
    x0 ← …
    y0 ← …
    if (x0 >= k) goto next
loop: x1 ← φ(x0,x2)
      y1 ← φ(y0,y2)
        x2 ← x1 + 1
        y2 ← y1 + x2
      if(x2 < k) goto loop
next:     …
```

Static Single Assignment Form

Original

```
x ← …
y ← …
while (x < k)
    x ← x + 1
    y ← y + x
```

SSA-form

```
    x0 ← …
    y0 ← …
    if (x0 >= k) goto next
loop: x1 ← φ(x0,x2)
      y1 ← φ(y0,y2)
      x2 ← x1 + 1
      y2 ← y1 + x2
    if(x2 < k) goto loop
next:     …
```

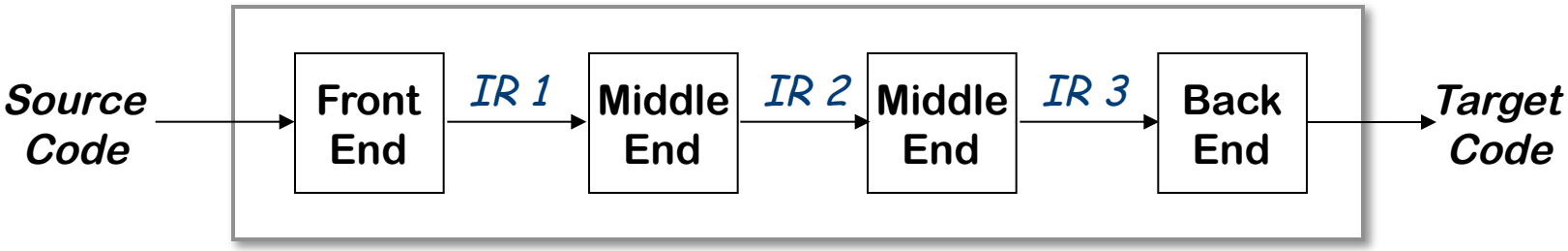
Keeps single assign. property.

Static Single Assignment Form Advantages

Strengths of SSA-form

- Sharper analysis because values never redefined
- Simplifies and improves optimizations
- (Sometimes) faster algorithms

Using Multiple Representations



- Repeatedly lower the level of the intermediate representation
  - → Each intermediate representation is suited towards certain optimizations
- Example: the Open64 compiler
  - → WHIRL intermediate format
    - Consists of 5 different *IR*s that are progressively more detailed and less abstract

Memory Models

Two major models

- Register-to-register model
  - → Keep all values that can legally be stored in a register in registers
  - → Ignore machine limitations on number of registers
  - → Compiler back-end must insert loads and stores
- Memory-to-memory model
  - → Keep all values in memory
  - → Only promote values to registers directly before they are used
  - → Compiler back-end can remove loads and stores
- Compilers for RISC machines usually use register-to-register
  - → Reflects programming model
  - → Easier to determine when registers are used

The Rest of the Story…

Representing the code is only part of an *IR*

There are other necessary components

- Symbol table
- Constant table
  - → Representation, type
  - → Storage class, offset
- Storage map
  - → Overall storage layout
  - → Overlap information
  - → Virtual register assignments

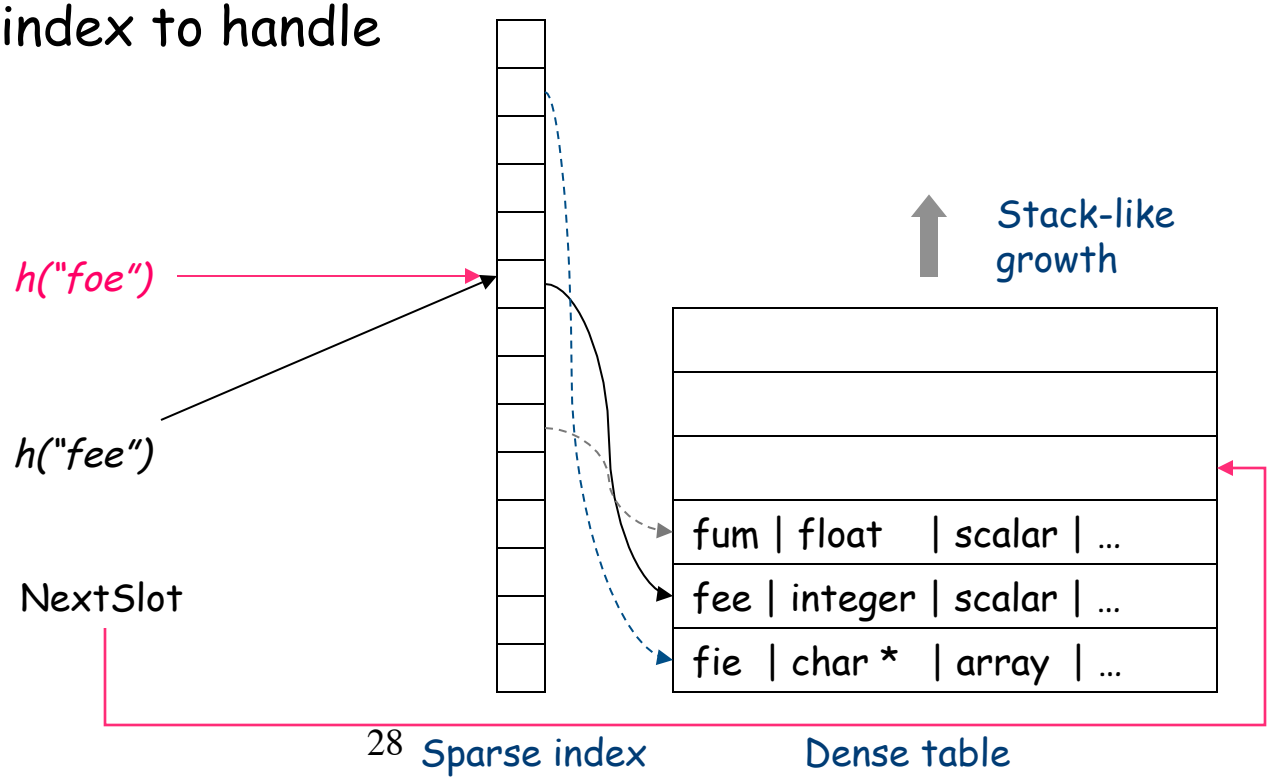
Symbol Tables

Classic approach to building a symbol table uses hashing

- Personal preference: a two-table scheme
  - → Sparse index to reduce chance of collisions
  - → Dense table to hold actual data
    - Easy to expand, to traverse, to read & write from/to files
- Use chains in index to handle collisions

See §B.3 in EaC for a longer explanation

Collision occurs when h() returns a slot in the sparse index that is already full.

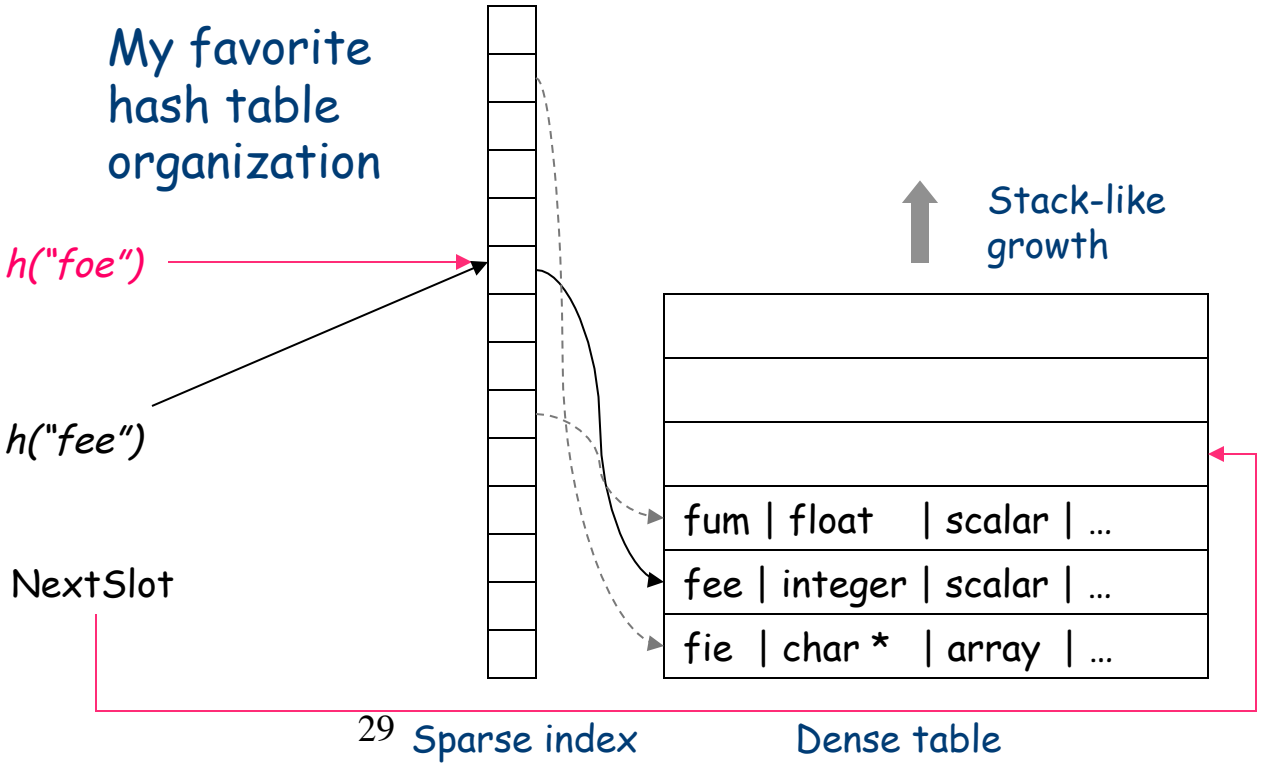


Hash-less Symbol Tables

Classic approach to building a symbol table uses hashing

- Some concern about worst-case behavior
  - → Collisions in the hash function can lead to linear search
  - → Some authors advocate "perfect" hash for keyword lookup
- Automata theory lets us avoid worst-case behavior

Collision occurs when h() returns a slot in the sparse index that is already full.



Hash-less Symbol Tables

One alternative is Paige & Cai's *multiset discrimination*

- Order the name space offline
- Assign indices to each name
- Replace the names in the input with their encoded indices

Digression on page 241 of EaC

Using DFA techniques, we can build a guaranteed linear-time replacement for the hash function *h*

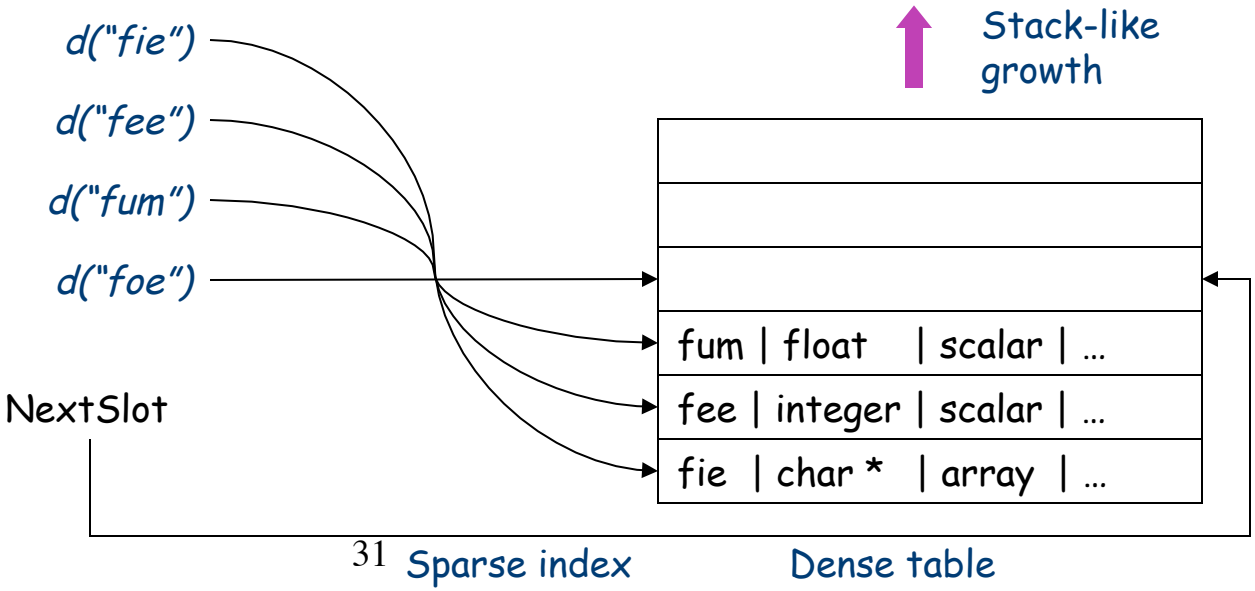
- DFA that results from a list of words is acyclic
  - → RE looks like $r_1 \mid r_2 \mid r_3 \mid \ldots \mid r_k$
  - → Could process input twice, once to build DFA, once to use it
- We can do even better

Hash-less Symbol Tables

Classic approach to building a symbol table uses hashing

- Some concern about worst-case behavior
  - → Collisions in the hash function can lead to linear search
  - → Some authors advocate "perfect" hash for keyword lookup
- Automata theory lets us avoid worst-case behavior

Replace the hash function, *h*, and the sparse index with an efficient direct map, *d*, …



Hash-less Symbol Tables

Incremental construction of an acyclic DFA

- To add a word, run it through the DFA
  - → At some point, it will face a transition to the error state
  - → At that point, start building states & transitions to recognize it
- Requires a memory access per character in the key
  - → If DFA grows too large, memory access costs become excessive
  - → For small key sets (e.g., names in a procedure), not a problem
- Optimizations
  - → Last state on each path can be explicit
    - Substantial reduction in memory costs
    - Instantiate when path is lengthened
  - → Trade off granularity against size of state representation
  - → Encode capitalization separately
    - Bit strings tied to final state?