



COMP 412
FALL 2009

Intermediate Representations

Comp 412

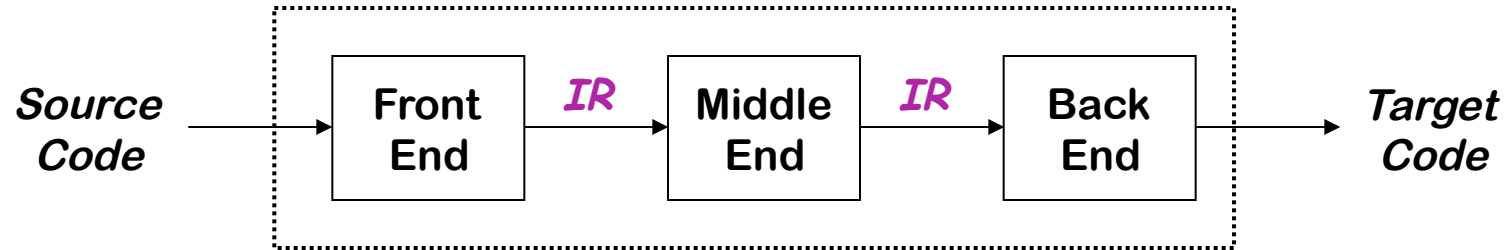
Most of the material in
this lecture comes
from Chapter 5 of EaC

Copyright 2009, Keith D. Cooper & Linda Torczon, all rights reserved.

Students enrolled in Comp 412 at Rice University have explicit permission to make copies of these materials for their personal use.

Faculty from other educational institutions may use these materials for nonprofit educational purposes, provided this copyright notice is preserved.

Intermediate Representations



- Front end - produces an intermediate representation (*IR*)
- Middle end - transforms the *IR* into an equivalent *IR* that runs more efficiently
- Back end - transforms the *IR* into native code
- *IR* encodes the compiler's knowledge of the program
- Middle end usually consists of several passes

Intermediate Representations



- Decisions in *IR* design affect the speed and efficiency of the compiler
- Some important *IR* properties
 - Ease of generation
 - Ease of manipulation
 - Procedure size
 - Freedom of expression
 - Level of abstraction
- The importance of different properties varies between compilers
 - Selecting an appropriate *IR* for a compiler is critical

Types of Intermediate Representations



Three major categories

- Structural
 - Graphically oriented
 - Heavily used in source-to-source translators
 - Tend to be large
- Linear
 - Pseudo-code for an abstract machine
 - Level of abstraction varies
 - Simple, compact data structures
 - Easier to rearrange
- Hybrid
 - Combination of graphs and linear code
 - Example: control-flow graph

Examples:
Trees, DAGs

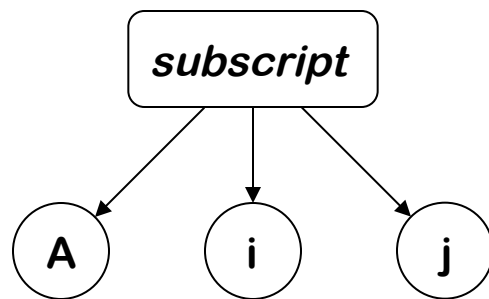
Examples:
3 address code
Stack machine code

Example:
Control-flow graph



Level of Abstraction

- The level of detail exposed in an *IR* influences the profitability and feasibility of different optimizations.
- Two different representations of an array reference:



High level AST:
Good for memory
disambiguation

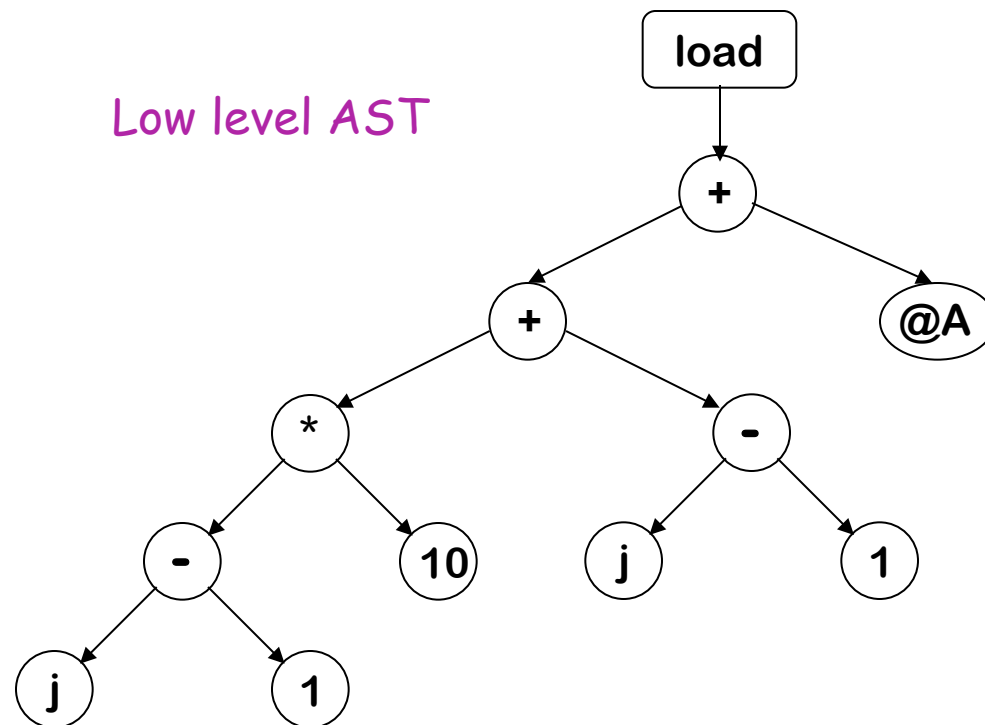
```
loadI 1      => r1
sub   rj, r1 => r2
loadI 10     => r3
mult  r2, r3 => r4
sub   ri, r1 => r5
add   r4, r5 => r6
loadI @A     => r7
add   r7, r6 => r8
load  r8     => rAij
```

Low level linear code:
Good for address calculation



Level of Abstraction

- Structural *IRs* are usually considered high-level
- Linear *IRs* are usually considered low-level
- Not necessarily true:



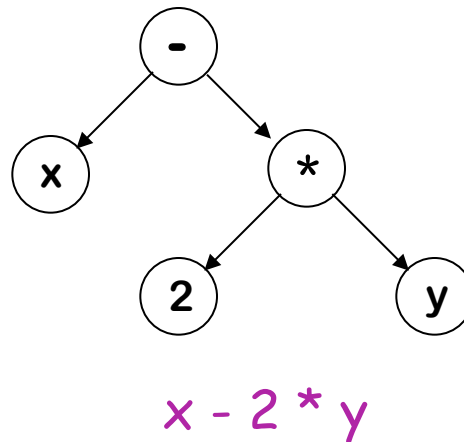
`loadArray A, i, j`

High level linear code



Abstract Syntax Tree

An abstract syntax tree is the procedure's parse tree with the nodes for most non-terminal nodes removed

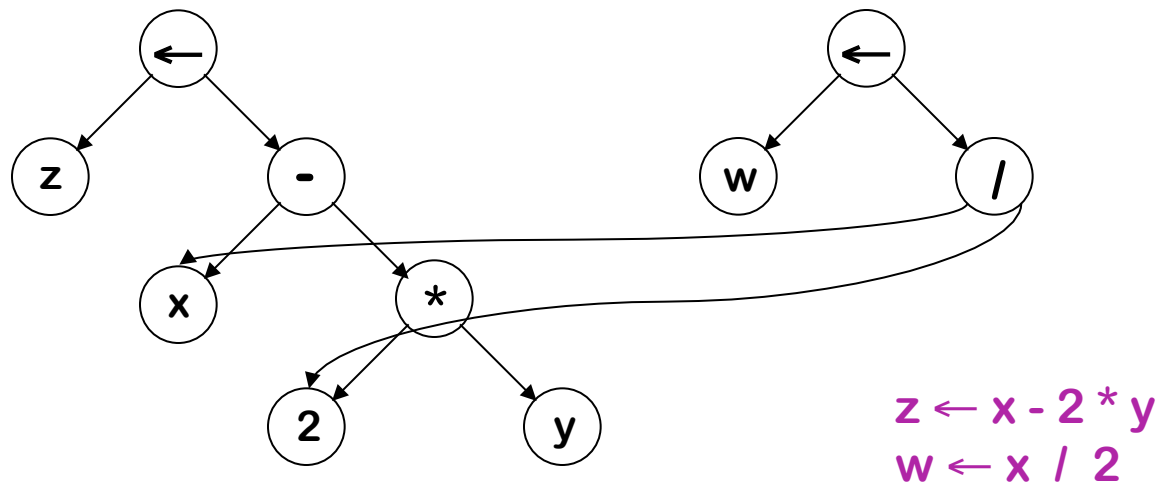


- Can use linearized form of the tree
 - Easier to manipulate than pointers
 - $x \ 2 \ y \ * \ -$ in postfix form
 - $- \ * \ 2 \ y \ x$ in prefix form
- S-expressions (Scheme, Lisp) are (essentially) ASTs



Directed Acyclic Graph

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



$$z \leftarrow x - 2 * y$$
$$w \leftarrow x / 2$$

- Makes sharing explicit
- Encodes redundancy

With two copies of the same expression, the compiler might be able to arrange the code to evaluate it only once.



Stack Machine Code

Originally used for stack-based computers, now Java

- Example:

$x - 2 * y$

becomes

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

Advantages

- Compact form
- Introduced names are *implicit*, not *explicit*
- Simple to generate and execute code

Useful where code is transmitted over slow communication links (*the net*)

Implicit names take up no space, where explicit ones do!



Three Address Code

Several different representations of three address code

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

$$x \leftarrow y \text{ op } z$$

With 1 operator (op) and, at most, 3 names (x, y, & z)

Example:

$$z \leftarrow x - 2 * y$$

becomes

$$\begin{aligned} t &\leftarrow 2 * y \\ z &\leftarrow x - t \end{aligned}$$

Advantages:

- Resembles many real machines
- Introduces a new set of names *
- Compact form



Three Address Code: Quadruples

Naive representation of three address code

- Table of $k * 4$ small integers
- Simple record structure
- Easy to reorder
- Explicit names

The original FORTRAN compiler used "quads"

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

RISC assembly code

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

Quadruples



Three Address Code: Triples

- Index used as implicit name
- 25% less space consumed than quads
- Much harder to reorder

(1)	load	y	
(2)	loadI	2	
(3)	mult	(1)	(2)
(4)	load	x	
(5)	sub	(4)	(3)

Implicit names occupy no space

Remember, for a long time, 640Kb was a lot of RAM



Three Address Code: Indirect Triples

- List first triple in each statement
- Implicit name space
- Uses more space than triples, but easier to reorder

Stmt List	Implicit Names	Indirect Triples		
(100)	(100)	load	y	
(105)	(101)	loadI	2	
	(102)	mult	(100)	(101)
	(103)	load	x	
	(104)	sub	(103)	(102)

- Major tradeoff between quads and triples is compactness versus ease of manipulation
 - In the past compile-time space was critical
 - Today, speed may be more important



Two Address Code

- Allows statements of the form

$$x \leftarrow x \text{ op } y$$

Has 1 operator (op) and, at most, 2 names (x and y)

Example:

$$z \leftarrow x - 2 * y$$

becomes

$$t_1 \leftarrow 2$$

$$t_2 \leftarrow \text{load } y$$

$$t_2 \leftarrow t_2 * t_1$$

$$z \leftarrow \text{load } x$$

$$z \leftarrow z - t_2$$

- Can be very compact

Problems

- Machines no longer rely on destructive operations
- Difficult name space
 - Destructive operations make reuse hard
 - Good model for machines with destructive ops (PDP-11)

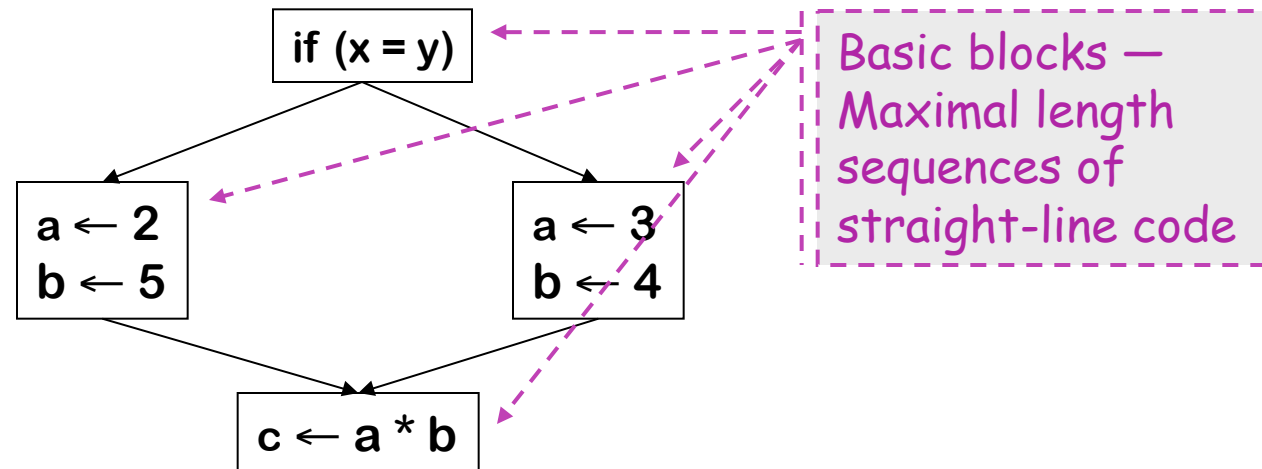


Control-flow Graph

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





Static Single Assignment Form

- The main idea: each name defined exactly once
- Introduce ϕ -functions to make it work

Original

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

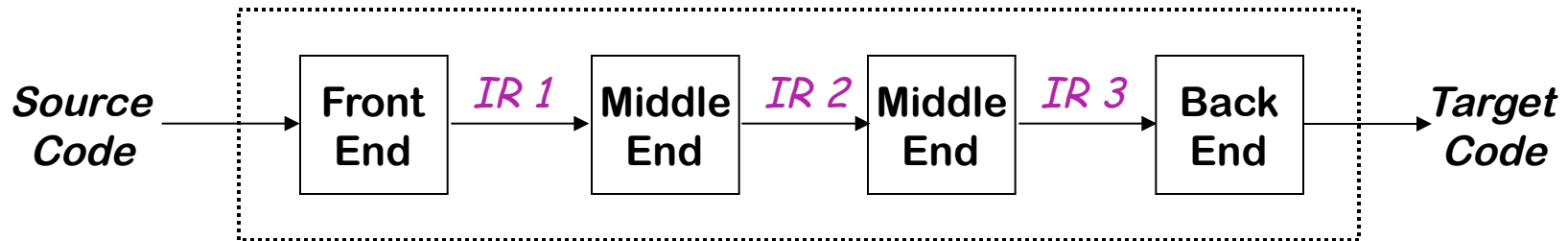
SSA-form

```
x0 ← ...
Y0 ← ...
if (x0 ≥ k) goto next
loop:  x1 ←  $\phi(x_0, x_2)$ 
      Y1 ←  $\phi(Y_0, Y_2)$ 
      x2 ← x1 + 1
      Y2 ← Y1 + x2
      if (x2 < k) goto loop
next:  ...
```

Strengths of SSA-form

- Sharper analysis
- ϕ -functions give hints about placement
- (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 *IRs* 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



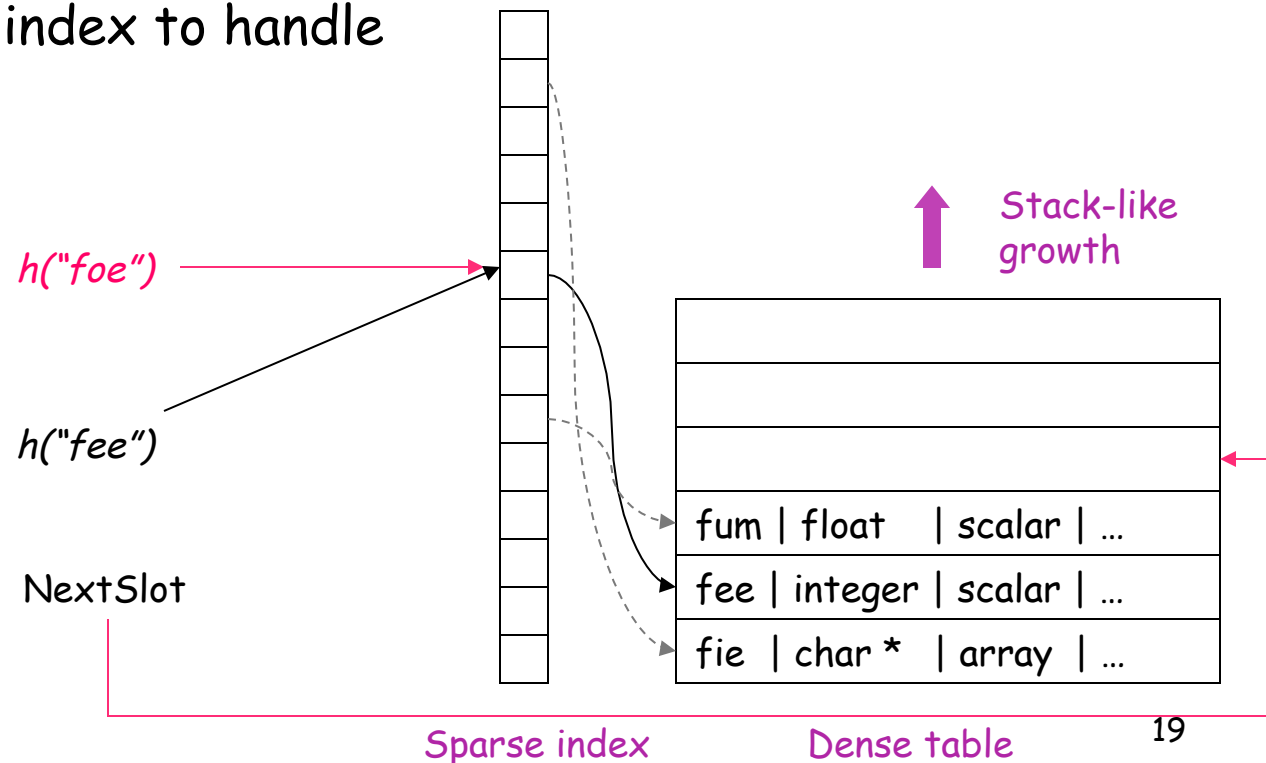
Symbol Tables

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 SB.3 in EaC for a longer explanation

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



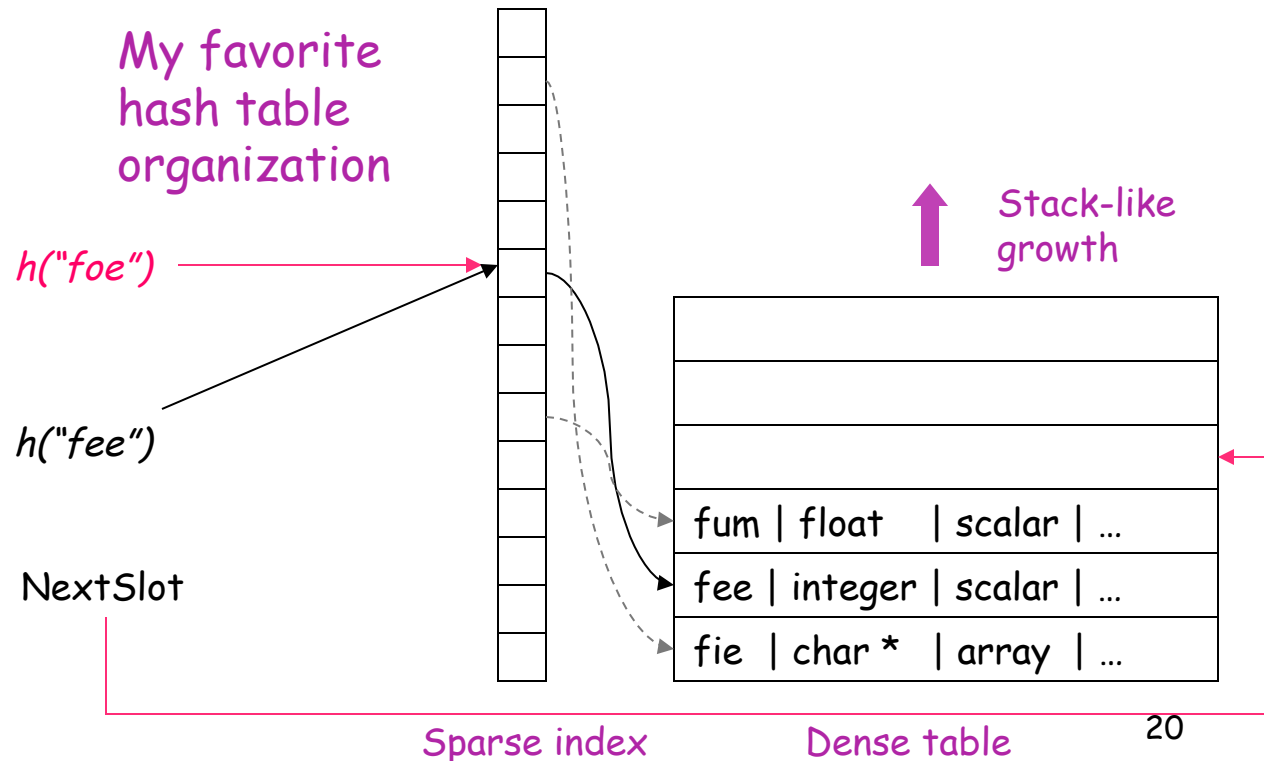


Hash-less Symbol Tables

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 $r_1 | r_2 | r_3 | \dots | r_k$
 - Could process input twice, once to build DFA, once to use it
- We can do even better

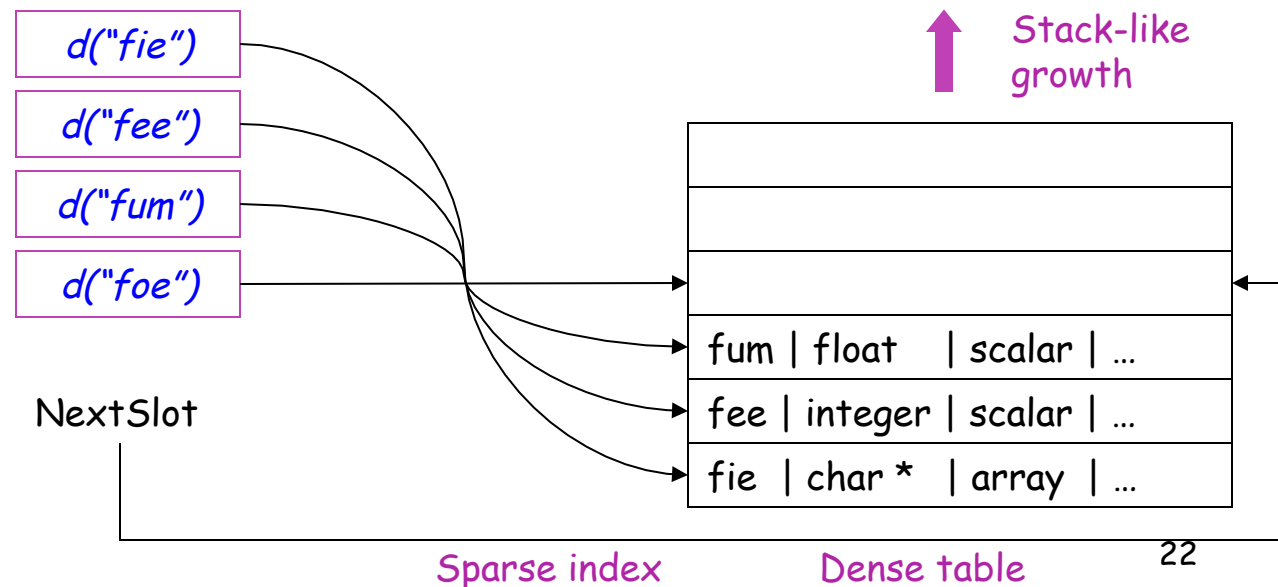


Hash-less Symbol Tables

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?