RUTGERS
THE STATE UNIVERSITY
OF NEW JERSEY

CS415 Compilers

Typing, Symbol Tables, Intermediate Representations, Code Generation

These slides are based on slides copyrighted by Keith Cooper, Ken Kennedy & Linda Torczon at Rice University

Review - Syntax Directed Translation

Ad-hoc syntax-directed translation

- Associate pieces of code with each production
- At each reduction, the corresponding code is executed
- Allowing arbitrary code provides complete flexibility
  - → Includes ability to do tasteless & bad things

Review - Calculate Total Number of Cycles

$Block_0$	→	$Block_1$ Assign	
	|	Assign	
Assign	→	Ident = Expr ;	cost ← cost + COST(store);
$Expr_0$	→	$Expr_1$ + Term	cost ← cost + COST(add);
	|	$Expr_1$ - Term	cost ← cost + COST(sub);
	|	Term	
$Term_0$	→	$Term_1$ * Factor	cost ← cost + COST(mult);
	|	$Term_1$ / Factor	cost ← cost + COST(div);
	|	Factor	
Factor	→	(Expr)	
	|	Number	cost ← cost + COST(loadi);

	|	Identifier	{ i ← hash(Identifier);
			if (Table[i].loaded = false)
			then {
			cost ← cost + COST(load);
			Table[i].loaded ← true;
			}
			}

This looks cleaner & simpler than the AG sol'n ! "cost" and Table[ ] are global variables

Review - Attribute Grammars

What is an attribute grammar?

- A context-free grammar augmented with a set of rules
- Each symbol in the derivation has a set of values, or *attributes*
- The rules specify how to compute a value for each attribute

Example grammar

Number	→	Sign List
Sign	→	<u>+</u>
	|	<u>-</u>
List	→	List Bit
	|	Bit
Bit	→	0
	|	1

This grammar describes signed binary numbers

We would like to augment it with rules that compute the decimal value of each valid input string

Review – Signed Binary Number Example

Add rules to compute the decimal value of a signed binary number

Productions			Attribution Rules
Number	$\rightarrow$	*Sign List*	*List.pos* $\leftarrow$ *0* *If Sign.neg* *then Number.val* $\leftarrow$ – *List.val* *else Number.val* $\leftarrow$ *List.val*
Sign	$\rightarrow$	$\underline{+}$	*Sign.neg* $\leftarrow$ *false*
	|	$\underline{-}$	*Sign.neg* $\leftarrow$ *true*
$List_0$	$\rightarrow$	$List_1$ *Bit*	$List_1.pos \leftarrow List_0.pos + 1$ $Bit.pos \leftarrow List_0.pos$ $List_0.val \leftarrow List_1.val + Bit.val$
	|	*Bit*	*Bit.pos* $\leftarrow$ *List.pos* *List.val* $\leftarrow$ *Bit.val*
Bit	$\rightarrow$	0	*Bit.val* $\leftarrow$ *0*
	|	1	$Bit.val \leftarrow 2^{Bit.pos}$

Symbol	Attributes
Number	val
Sign	neg
List	pos, val
Bit	pos, val

SDT V.S. Attribute Grammar

Relationship between SDT and attribute grammars

Similarities

- Both rules & actions associated with productions
- Application order determined by tools
- (Somewhat) abstract names for symbols

Differences

- SDT actions applied as a unit; not true for AG rules
- Anything goes in *ad-hoc* actions; AG rules are (purely) functional
- AG rules are higher level than *ad-hoc* actions

Typical Uses (Semantic Analysis)

- Building a symbol table
  - → Enter declaration information as processed
  - → At end of declaration syntax, do some post processing
  - → Use table to check errors as parsing progresses

assumes table is global

- Simple error checking/type checking
  - → Define before use → lookup on reference
  - → Dimension, type, ... → check as encountered
  - → Type conformability of expression → bottom-up walk
  - → Procedure interfaces are harder
    - Build a representation for parameter list & types
    - Check actual vs. formal parameter list
    - Positional or keyword associations

Error Recovery in Shift-Reduce Parsers

The problem: parser encounters an invalid token

Goal: Want to parse the rest of the file

Basic idea:

- Assume something went wrong while trying to find handle for nonterminal A
- Pretend handle for A has been found; pop "handle", skip over input to find terminal that can follow A

Restarting the parser:

- find a restartable state on the stack (has transition for nonterminal A)
- move to a consistent place in the input (token that can follow A)
- perform (error) reduction (for nonterminal A)
- print an informative message

Error Recovery in YACC

Yacc's (bison's) error mechanism (note: version dependent!)

- designated token **error**
- used in error productions of the form

  $A \rightarrow$ **error** α // basic case
- α specifies synchronization points

When error is discovered

- pops stack until it finds state where it can shift the **error** token
- resumes parsing to match α

  special cases:
  - α = w, where w is string of terminals: skip input until w has been read
  - $\alpha = \varepsilon$: skip input until state transition on input token is defined
- error productions can have actions

Error Recovery in YACC

```
cmpdstmt: BEG stmt_list END
stmt_list : stmt
          | stmt_list ';' stmt
          | error { yyerror("\n***Error: illegal statement\n");}
```

This should

- throw out the erroneous statement
- synchronize at ";" or "END" (implicit: $\alpha = \varepsilon$)
- writes message "***Error: illegal statement" to stderr

Example: begin a & 5 | hello ; a := 3 end

↑ ↑ resume parsing

***Error: illegal statement

Review - Types and Type Systems

Type system: Each language construct (operator, expression, statement, …) is associated with a type expression. The type system is a collection of rules for assigning type expressions to these constructs.

Type expressions for
- basic types: `integer, char, real, boolean, typeError`
- constructed types, e.g., one-dimensional arrays: `array(lb, ub, elem_type)` , where elem_type is a type expression

A type checker implements a type system. It computes or "constructs" type expressions for each language construct

Types and Type Systems

Programmers may define their own types and give them names:

```
type my_int is int;
 ...
int a;
my_int b;
 ...
  ... a + b ...
```

Type names can also be part of the type expression language.
Note: type names and type variables are different!

Type Equivalence

Structural -- type equivalence: type names are expanded
Name -- type equivalence: type names are not expanded

Example:

```
type A is  array(1..10) of integer;
type B is  array(1..10) of integer;
a : A;
b : B;
c, d: array(1..10) of integer;
e: array(1..10) of integer;
```

Answer: structural equivalence:
name equivalence:

Type Equivalence

Structural -- type equivalence: type names are expanded

Name -- type equivalence: type names are not expanded

Example:

```
type A is  array(1..10) of integer;
type B is  array(1..10) of integer;
a : A;
b : B;
c, d: array(1..10) of integer;
e: array(1..10) of integer;
```

Answer: structural equivalence: `(a, b, c, d, e)`

name equivalence: `(a); (b); (c, d, e);`

Syntax Directed Translation Scheme (SDT)

Revisit our type inference rule for "+".

```
exp  :  exp  '+'  exp  {  if ($1 == integer && $3 == integer)
                              $$ = integer;
                          else {
                              $$ = typeError;
                              printf("\n***Error: illegal operand types\n");
                          } }
```

PROJECT HINT: The definition of type expression as C types (structs) should be done in attr.h . attr.c may contain helper functions. The assignment of type expression C types to terminals and nonterminals of the grammar is done in parse.y.

Lexically-scoped Symbol Tables

§ 5.5 in EaC

The problem

- The compiler needs a distinct record for each declaration
- Nested lexical scopes admit duplicate declarations

The interface

- insert(*name, level*) – creates record for *name* at *level*
- lookup(*name, level*) – returns pointer or index
- delete(*level*) – removes all names declared at *level*

Many implementation schemes have been proposed (see § B.4)

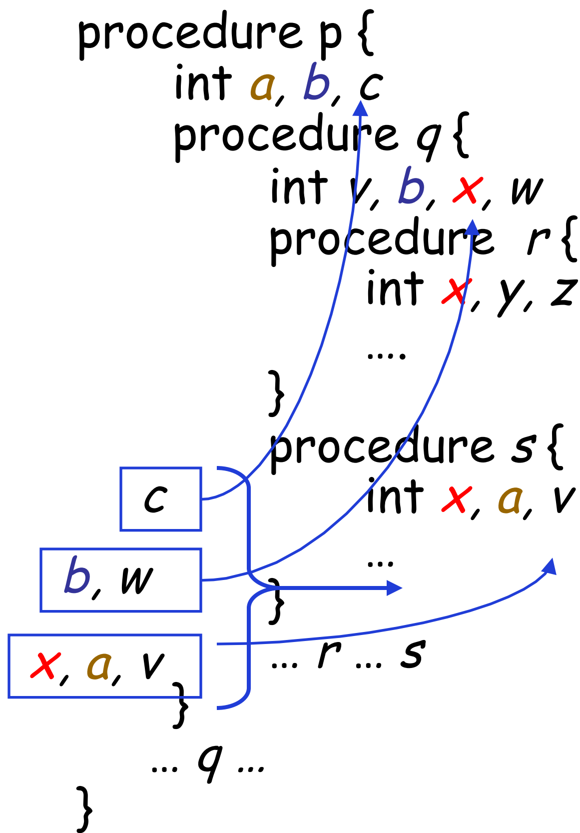
- We'll stay at the conceptual level
- Hash table implementation is tricky, detailed, & fun

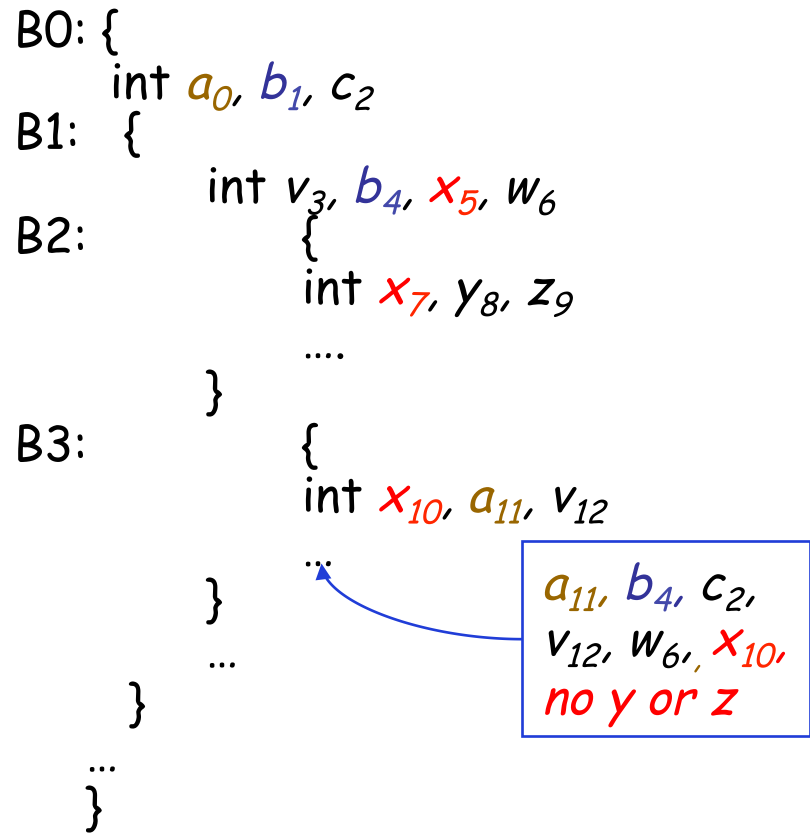
Example

```
procedure p {
      int a, b, c
      procedure q {
            int v, b, x, w
            procedure  r {
                  int x, y, z
                  ....
            }
            procedure s {
                  int x, a, v
                  ...
            }
            ... r ... s
      }
      ... q ...
}
```

```
B0: {
      int a, b, c
B1:   {
            int v, b, x, w
B2:               {
                  int x, y, z
                  ....
            }
B3:               {
                  int x, a, v
                  ...
            }
            ...
      }
      ...
}
```

Example



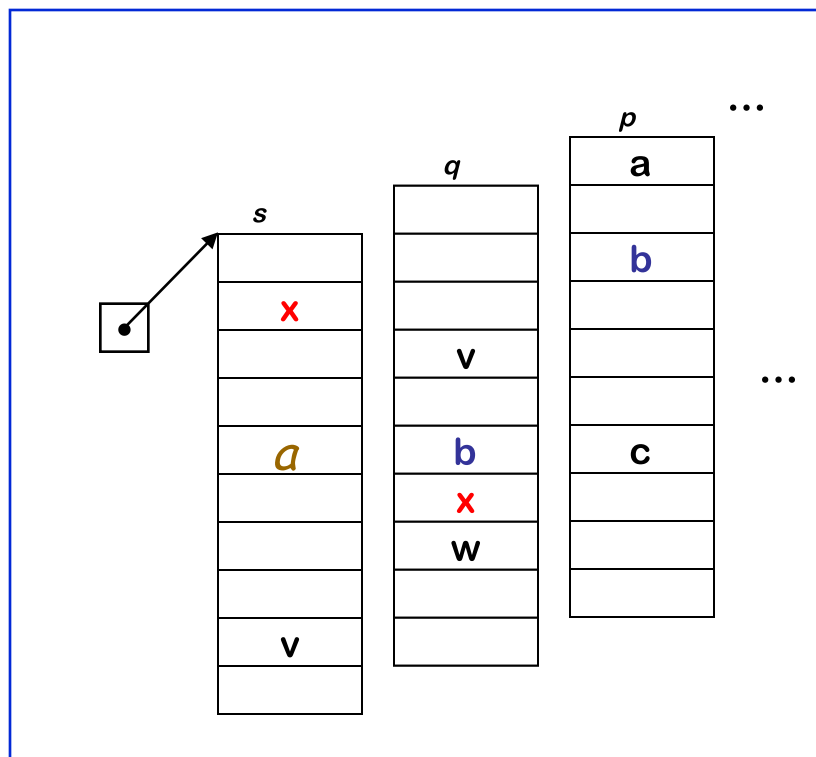


Picturing it as a series of Algol-like procedures

Lexically-scoped Symbol Tables

High-level idea

- Create a new table for each scope
- Chain them together for lookup



"Chain of tables" implementation

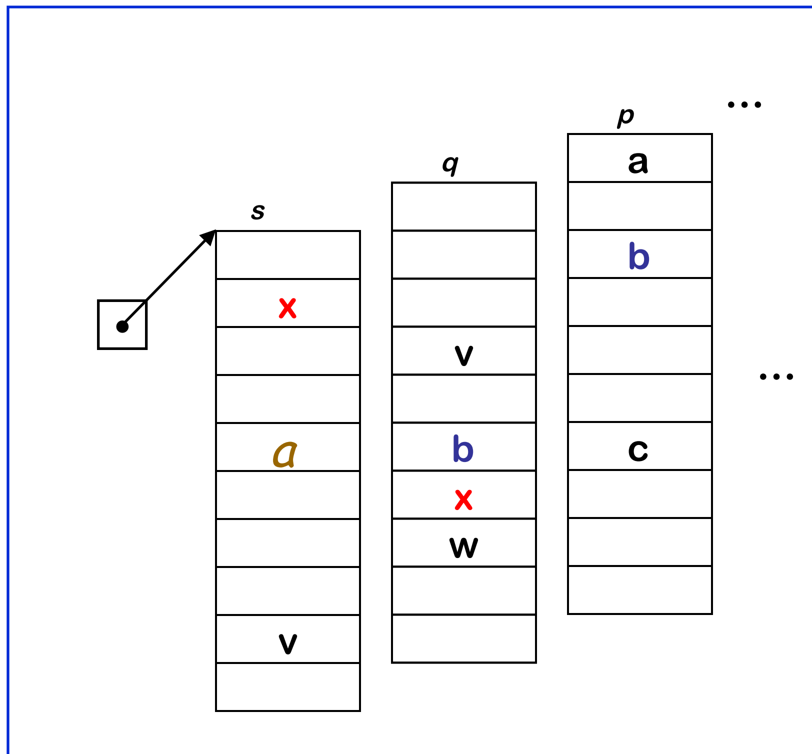
- *insert*() may need to create table
- it always inserts at current level
- *lookup*() walks chain of tables & returns first occurrence of name
- *delete*() throws away table for level *p*, if it is top table in the chain

Individual tables can be hash tables.

Lexically-scoped Symbol Tables

High-level idea

- Create a new table for each scope
- Chain them together for lookup



Remember

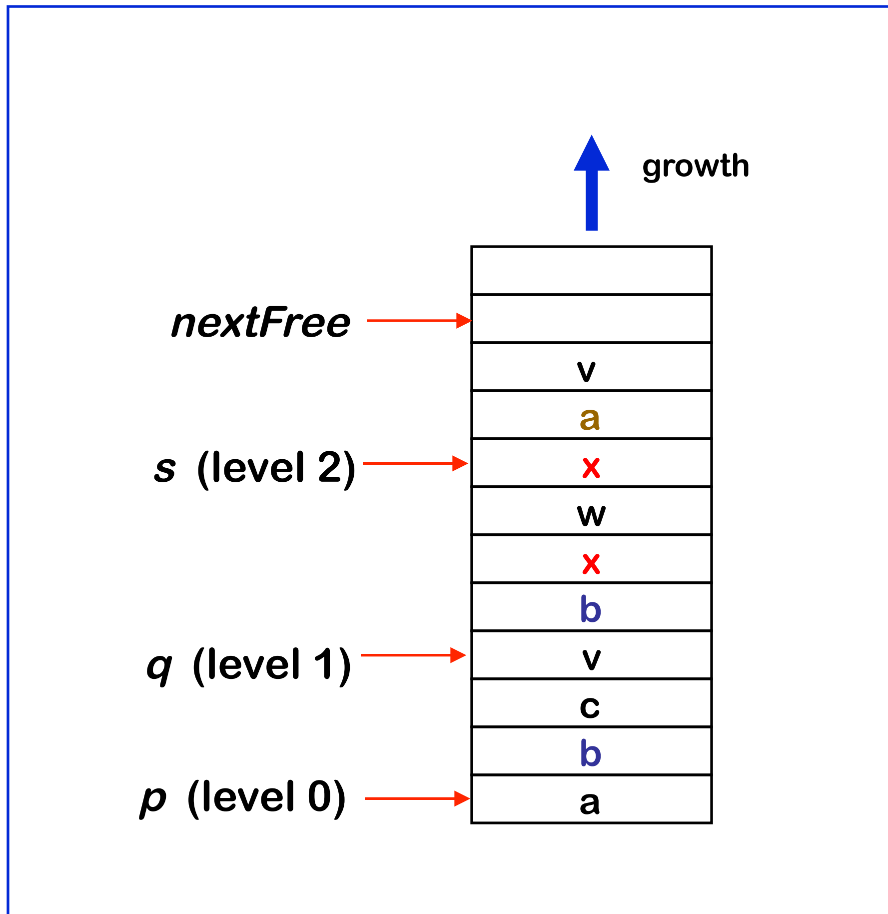
a_{11}, b_4, c_2, v_{12}, w_6, x_{10}, no y or z

the names visible in s

If we add the subscripts, the relationship between the code and the table becomes clear

Implementing Lexically Scoped Symbol Tables

Stack organization



Implementation

- **insert () creates new level pointer if needed and inserts at nextFree**
- **lookup () searches linearly from nextFree-1 forward**
- **delete () sets nextFree to the equal the start location of the level deleted.**

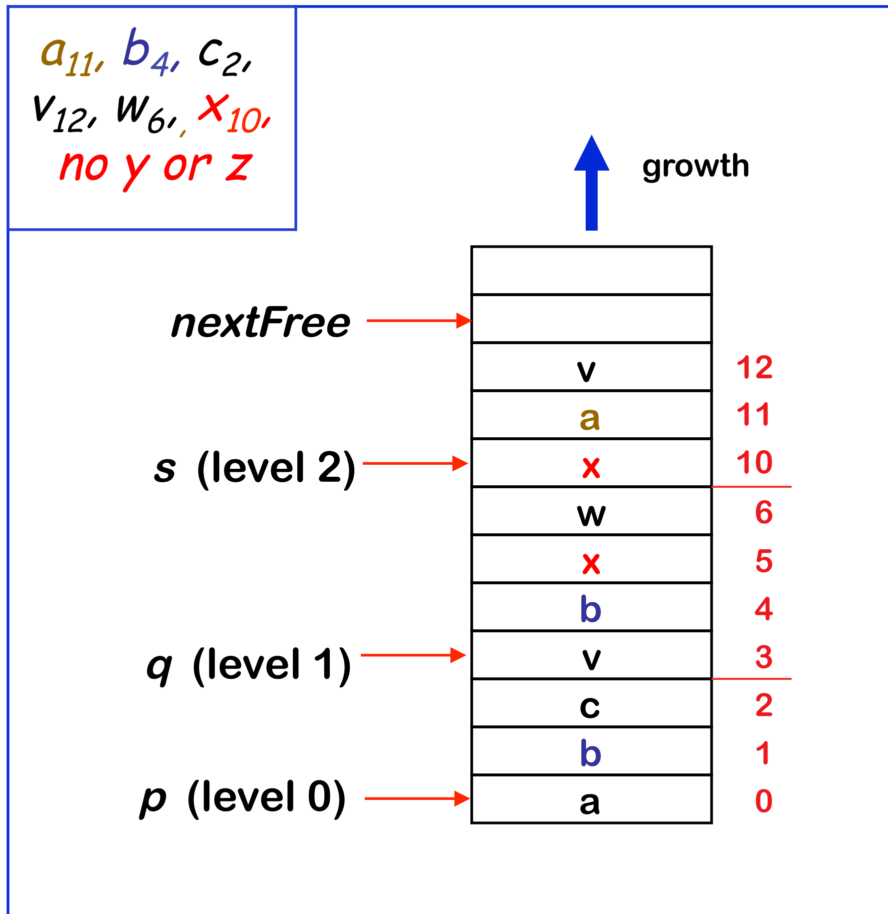
Advantage

- **Uses <u>much</u> less space**

Disadvantage

- **Lookups can be expensive**

Stack organization



Implementation

- **insert ()** creates new level pointer if needed and inserts at nextFree
- **lookup ()** searches linearly from nextFree-1 forward
- **delete ()** sets nextFree to the equal the start location of the level deleted.

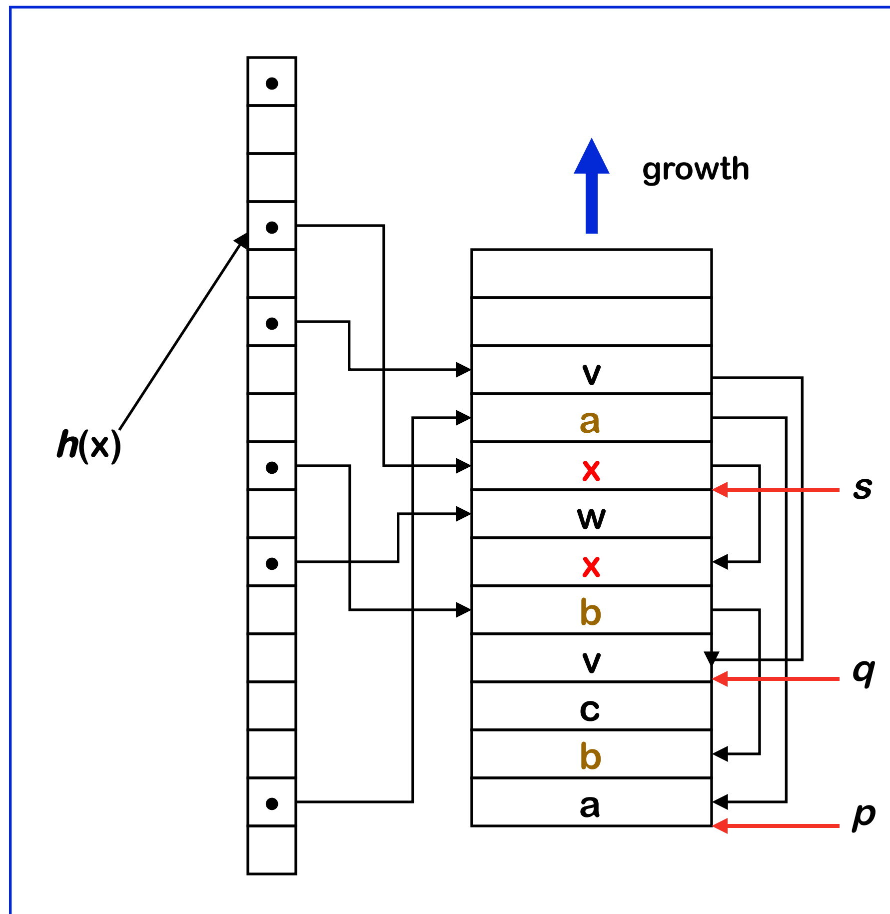
Advantage

- Uses much less space

Disadvantage

- Lookups can be expensive

Threaded stack organization



Implementation

- insert () puts new entry at the head of the list for the name
- lookup () goes direct to location
- delete () processes each element in level being deleted to remove from head of list

Advantage

- lookup is fast

Disadvantage

- delete takes time proportional to number of declared variables in level

Implementing Lexically Scoped Symbol Tables

Threaded stack organization

a_{11}, b_4, c_2, v_{12}, w_6, x_{10}, no y or z

Implementation

- insert () puts new entry at the head of the list for the name
- lookup () goes direct to location
- delete () processes each element in level being deleted to remove from head of list

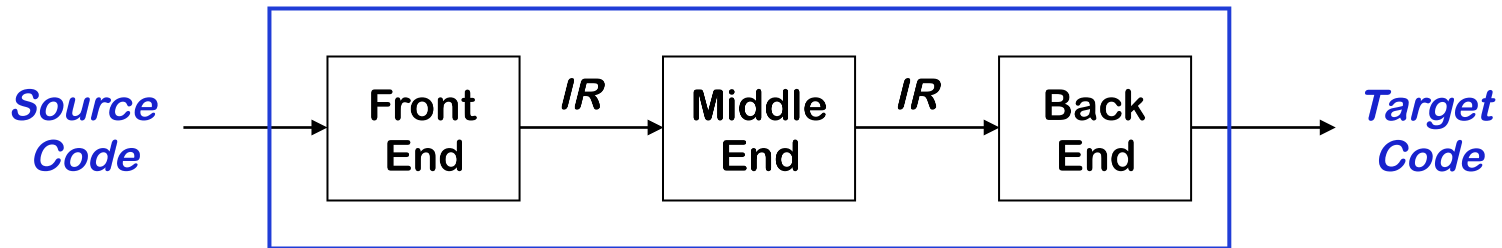
Advantage

- lookup is fast

Disadvantage

- delete takes time proportional to number of declared variables in level

Intermediate Representations
(EaC Chaper 5)



- 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
  - → Size
  - → 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

Examples:
Trees, DAGs

- Linear
  - → Pseudo-code for an abstract machine
  - → Level of abstraction varies
  - → Simple, compact data structures
  - → Easier to rearrange

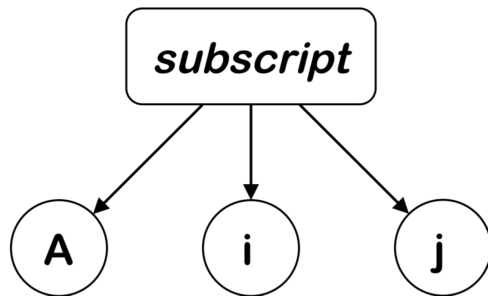
Examples:
3 address code
Stack machine code

- Hybrid
  - → Combination of graphs and linear 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:



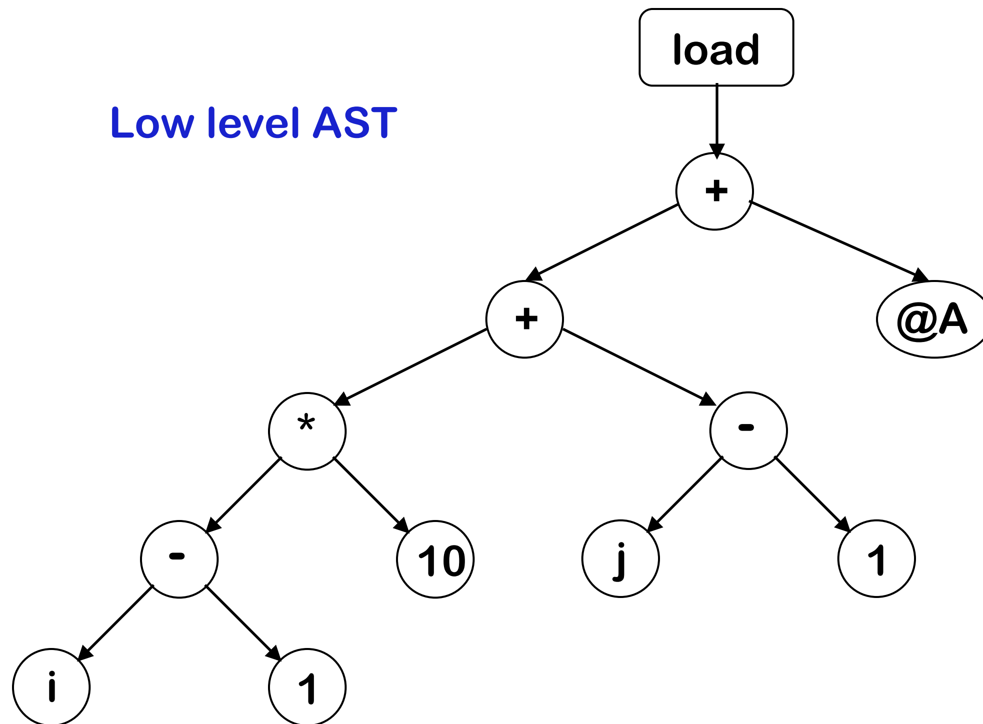
High level AST:
Good for memory disambiguation

```
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:

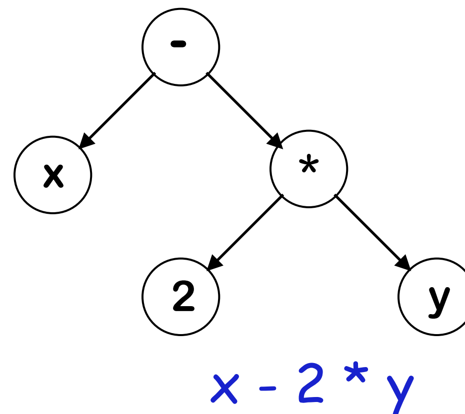



```
loadArray A,i,j
```

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



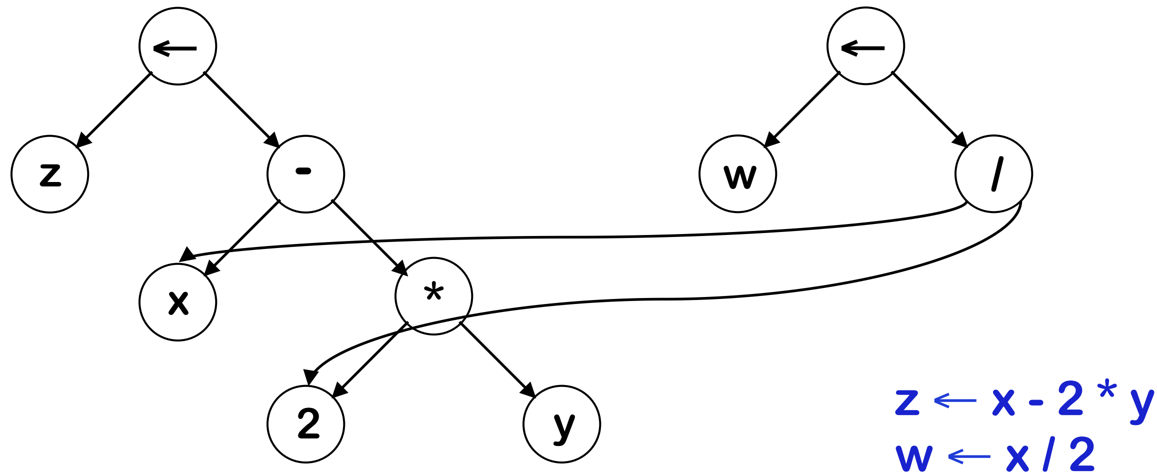
- Can use linearized form of the tree
  - Easier to manipulate than pointers

    x 2 y * - in postfix form

    - * 2 y x in prefix form

Directed Acyclic Graph

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



z ← x - 2 * y
w ← x / 2

- Makes sharing explicit
- Encodes redundancy

Same expression twice means that the compiler might arrange to evaluate it just 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

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

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

Three Address Code

Several different representations of three address code

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

  $x \leftarrow y \; \underline{op} \; z$

  With 1 operator ($\underline{op}$) and, at most, 3 names (x, y, z)

Example:

$z \leftarrow x - 2 * y$ becomes

```
t ← 2 * y
z ← x - t
```

Advantages:

- Resembles many 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
- 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	2

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)

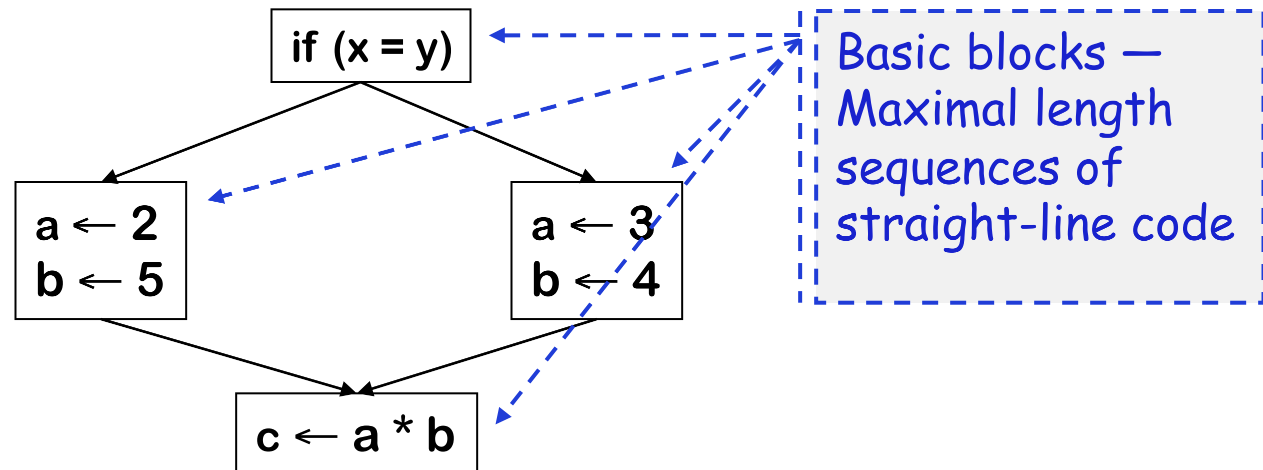
Implicit names take no space!

Control-flow Graph (CFG)

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 (SSA)

- The main idea: each name defined exactly once in program
- 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 ← φ(x0,x2)
           y1 ← φ(y0,y2)
           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

Next class

Code generation

Read EaC: Chapter 7.1 – 7.5