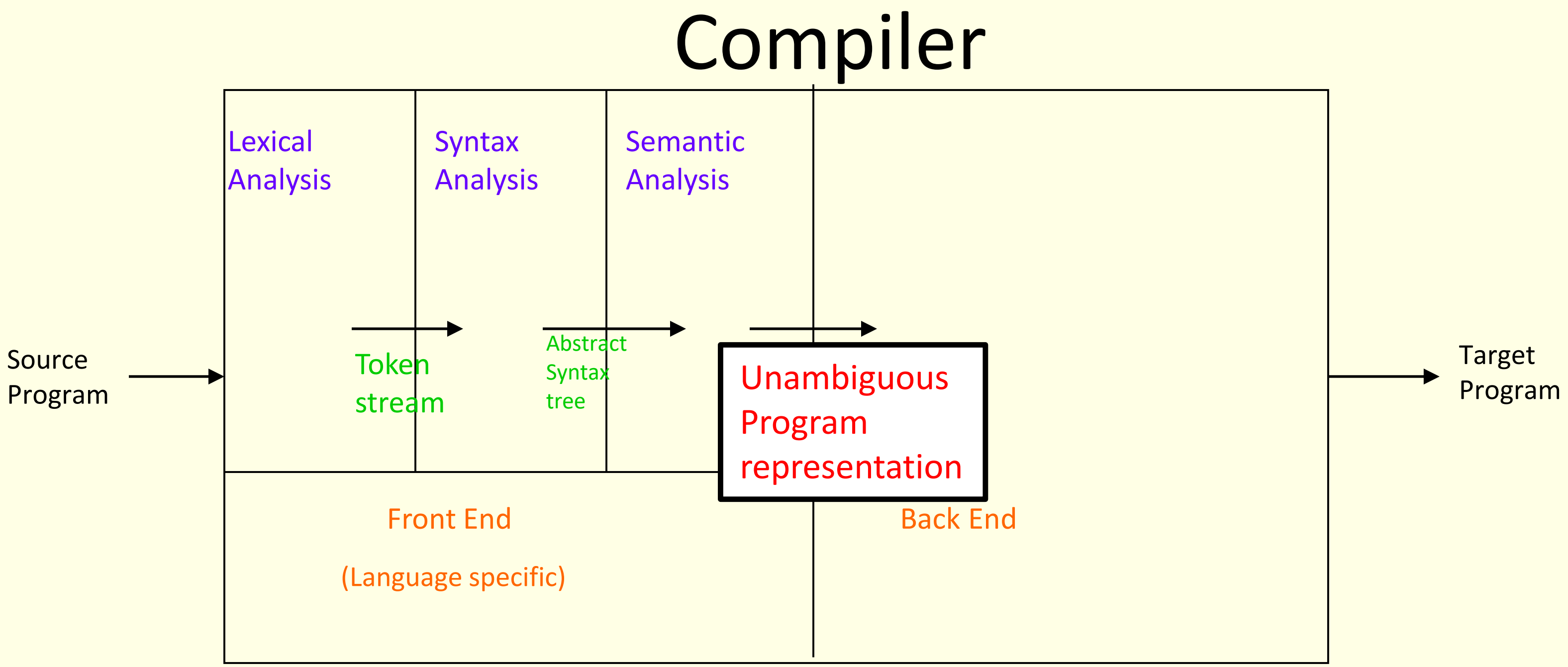
# Principles of Compiler Design

## Intermediate Representation



# Intermediate Representation Design

- More of a wizardry rather than science
- Compiler commonly use 2-3 IRs
- HIR (high level IR) preserves loop structure and array bounds
- MIR (medium level IR) reflects range of features in a set of source languages
  - language independent
  - good for code generation for one or more architectures
  - appropriate for most optimizations
- LIR (low level IR) low level similar to the machines

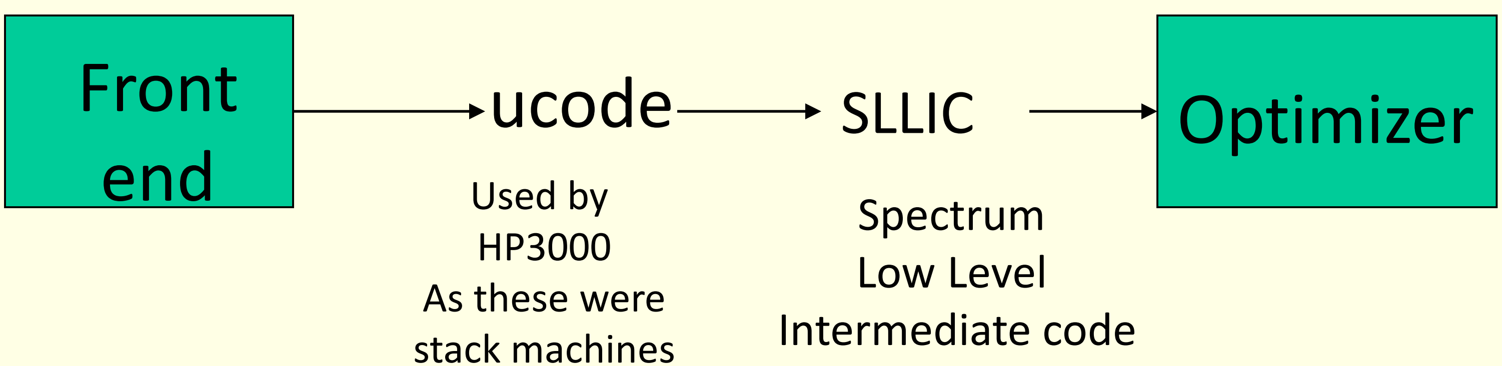
- Compiler writers have tried to define Universal IRs and have failed. (UNCOL in 1958)
- There is no standard Intermediate Representation. IR is a step in expressing a source program so that machine understands it
- As the translation takes place, IR is repeatedly analyzed and transformed
- Compiler users want analysis and translation to be fast and correct
- Compiler writers want optimizations to be simple to write, easy to understand and easy to extend

# Issues in IR Design

- source language and target language
- porting cost or reuse of existing design
- whether appropriate for optimizations
- U-code IR used on PA-RISC and Mips. Suitable for expression evaluation on stacks but less suited for load-store architectures
- both compilers translate U-code to another form
  - HP translates to very low level representation
  - Mips translates to MIR and translates back to U-code for code generator

# Issues in new IR Design

- how much machine dependent
- expressiveness: how many languages are covered
- appropriateness for code optimization
- appropriateness for code generation
- Use more than one IR (like in PA-RISC)



# Issues in new IR Design …

- Use more than one IR for more than one optimization
- represent subscripts by list of subscripts: suitable for dependence analysis
- make addresses explicit in linearized form:
  - suitable for constant folding, strength reduction, loop invariant code motion, other basic optimizations

# float a[20][10];
# use a[i][j+2]

HIR

t1← a[i,j+2]

MIR

t1← j+2
t2← i*20
t3← t1+t2
t4← 4*t3
t5← addr a
t6← t4+t5
t7←*t6

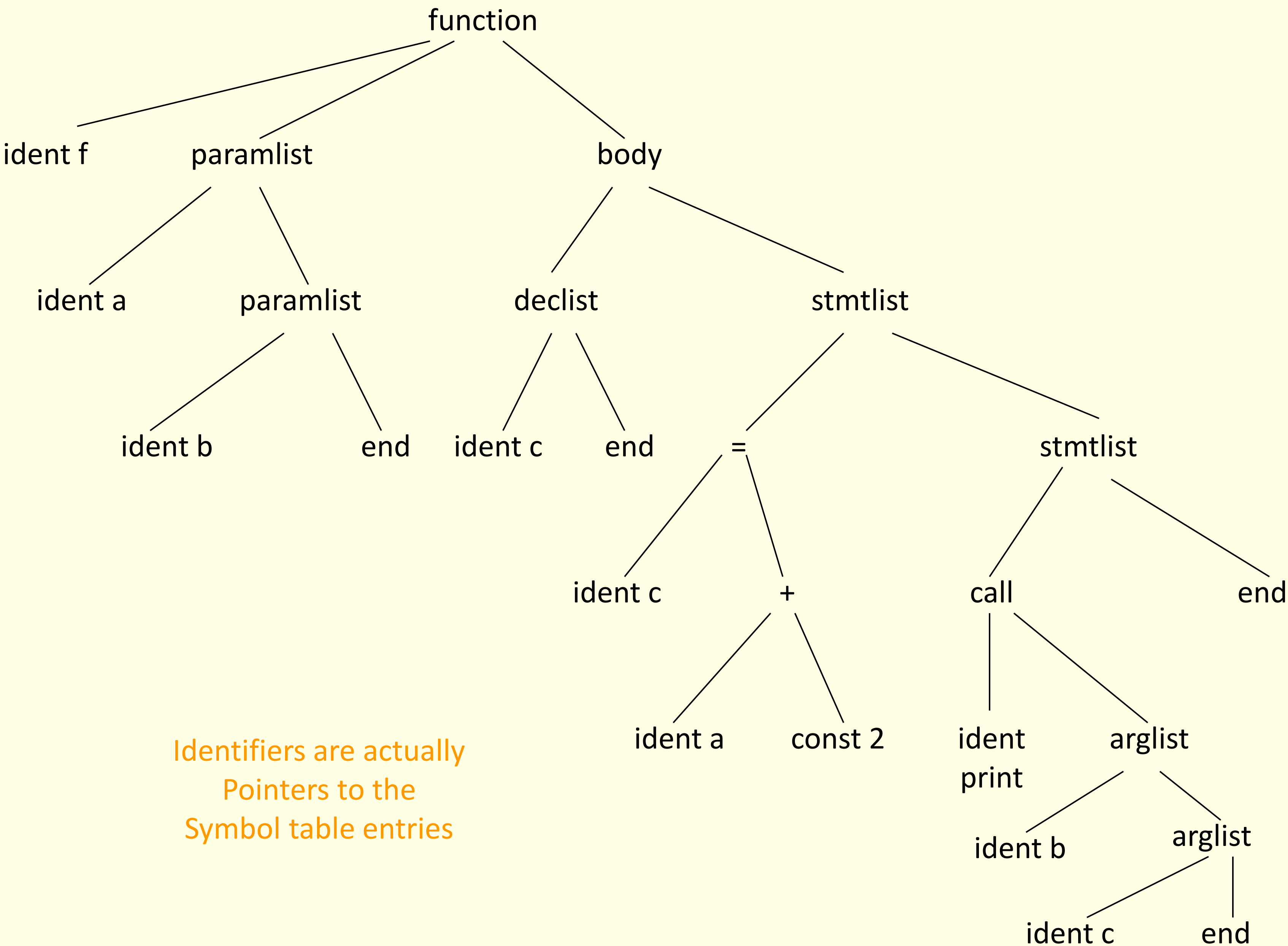
LIR

r1← [fp-4]
r2← r1+2
r3← [fp-8]
r4← r3*20
r5← r4+r2
r6← 4*r5
r7←fp-216
f1← [r7+r6]

# High level IR

```
int f(int a, int b) {
    int c;
    c = a + 2;
    print(b, c);
}
```

- Abstract syntax tree
  - keeps enough information to reconstruct source form
  - keeps information about symbol table




Identifiers are actually
Pointers to the
Symbol table entries

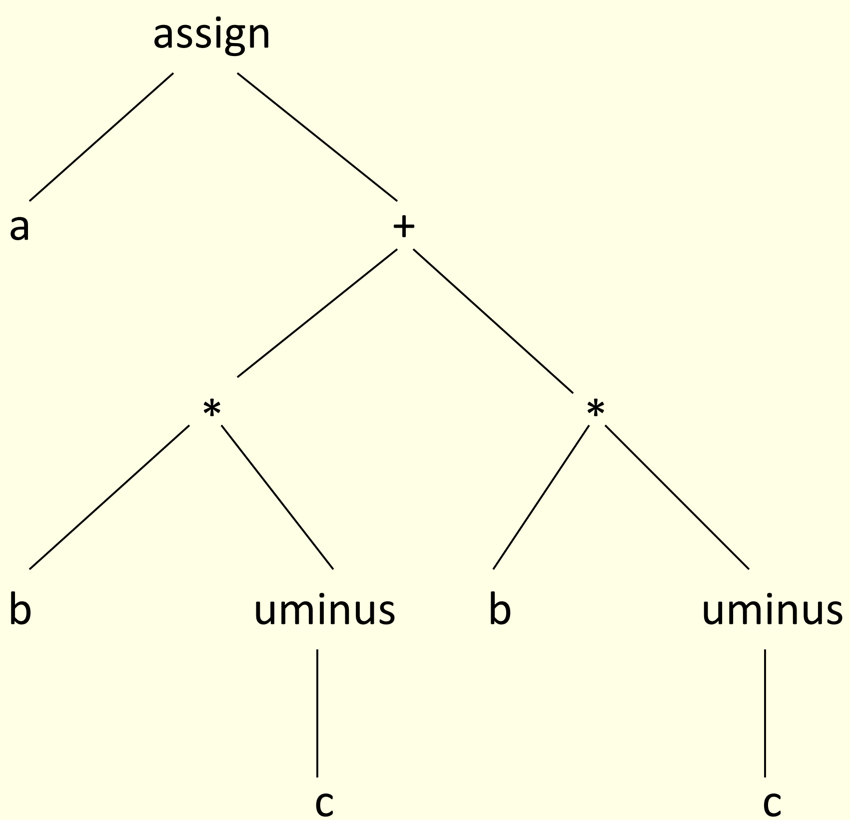
- Medium level IR
  - reflects range of features in a set of source languages
  - language independent
  - good for code generation for a number of architectures
  - appropriate for most of the optimizations
  - normally three address code
- Low level IR
  - corresponds one to one to target machine instructions
  - architecture dependent
- Multi-level IR
  - has features of MIR and LIR
  - may also have some features of HIR

# Abstract Syntax Tree/DAG

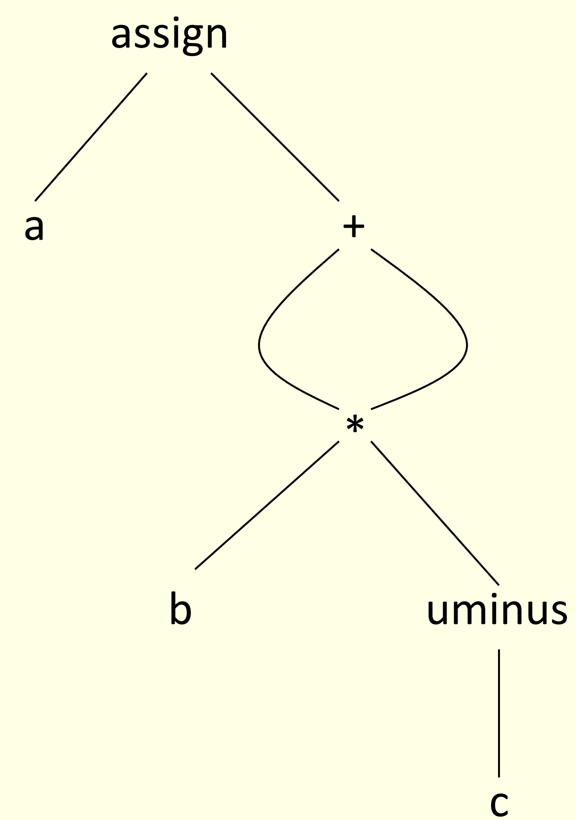
- Condensed form of a parse tree
- useful for representing language constructs
- Depicts the natural hierarchical structure of the source program
  - Each internal node represents an operator
  - Children of the nodes represent operands
  - Leaf nodes represent operands
- DAG is more compact than abstract syntax tree because common sub expressions are eliminated

# a := b * -c + b * -c

Abstract syntax tree




Directed Acyclic Graph



# Three address code

- A linearized representation of a syntax tree where explicit names correspond to the interior nodes of the graph
- Sequence of statements of the general form

$$X := Y \text{ op } Z$$

  - X, Y or Z are names, constants or compiler generated temporaries
  - op stands for any operator such as a fixed- or floating-point arithmetic operator, or a logical operator
  - Extensions to handle arrays, function call

# Three address code …

- Only one operator on the right hand side is allowed
- Source expression like x + y * z might be translated into

  $t_1 := y * z$

  $t_2 := x + t_1$

  where $t_1$ and $t_2$ are compiler generated temporary names
- Unraveling of complicated arithmetic expressions and of control flow makes 3-address code desirable for code generation and optimization
- The use of names for intermediate values allows 3-address code to be easily rearranged

# Three address instructions

- Assignment
  - x = y op z
  - x = op y
  - x = y
- Jump
  - goto L
  - if x relop y goto L
- Indexed assignment
  - x = y[i]
  - x[i] = y
- Function
  - param x
  - call p,n
  - return y
- Pointer
  - x = &y
  - x = *y
  - *x = y

# Other IRs

- SSA: Single Static Assignment
- RTL: Register transfer language
- Stack machines: P-code
- CFG: Control Flow Graph
- Dominator Trees
- DJ-graph: dominator tree augmented with join edges
- PDG: Program Dependence Graph
- VDG: Value Dependence Graph
- GURRR: Global unified resource requirement representation. Combines PDG with resource requirements
- Java intermediate bytecodes
- The list goes on ......

# Symbol Table

- Compiler uses symbol table to keep track of scope and binding information about names
- changes to table occur
  - if a new name is discovered
  - if new information about an existing name is discovered
- Symbol table must have mechanism to:
  - add new entries
  - find existing information efficiently

# Symbol Table

- Two common mechanism:
  - linear lists
    - simple to implement, poor performance
  - hash tables
    - greater programming/space overhead, good performance
- Compiler should be able to grow symbol table dynamically
  - If size is fixed, it must be large enough for the largest program

# Data Structures for SymTab

- List data structure
  - simplest to implement
  - use a single array to store names and information
  - search for a name is linear
  - entry and lookup are independent operations
  - cost of entry and search operations are very high and lot of time goes into book keeping
- Hash table
  - The advantages are obvious

# Symbol Table Entries

- each entry corresponds to a declaration of a name
- format need not be uniform because information depends upon the usage of the name
- each entry is a record consisting of consecutive words
  - If uniform records are desired, some entries may be kept outside the symbol table (e.g. variable length strings)

# Symbol Table Entries

- information is entered into symbol table at various times
  - keywords are entered initially
  - identifier lexemes are entered by lexical analyzer
  - attribute values are filled in as information is available
- a name may denote several objects in the same block

```
int x;
struct x {float y, z; }
```

  - lexical analyzer returns the name itself and not pointer to symbol table entry
  - record in the symbol table is created when role of the name becomes clear
  - in this case two symbol table entries will be created

- attributes of a name are entered in response to declarations
- labels are often identified by colon (:)
- syntax of procedure/function specifies that certain identifiers are formals
- there is a distinction between token id, lexeme and attributes of the names
  - it is difficult to work with lexemes
  - if there is modest upper bound on length then lexemes can be stored in symbol table
  - if limit is large store lexemes separately

# Storage Allocation Information

- information about storage locations is kept in the symbol table
  - if target is assembly code then assembler can take care of storage for various names
- compiler needs to generate data definitions to be appended to assembly code
- if target is machine code then compiler does the allocation
- for names whose storage is allocated at runtime no storage allocation is done
  - compiler plans out activation records

# Representing Scope Information

- entries are declarations of names
- when a lookup is done, entry for appropriate declaration must be returned
- scope rules determine which entry is appropriate
- maintain separate table for each scope
- symbol table for a procedure or scope is compile time equivalent an activation record
- information about non local is found by scanning symbol table for the enclosing procedures
- symbol table can be attached to abstract syntax of the procedure (integrated into intermediate representation)

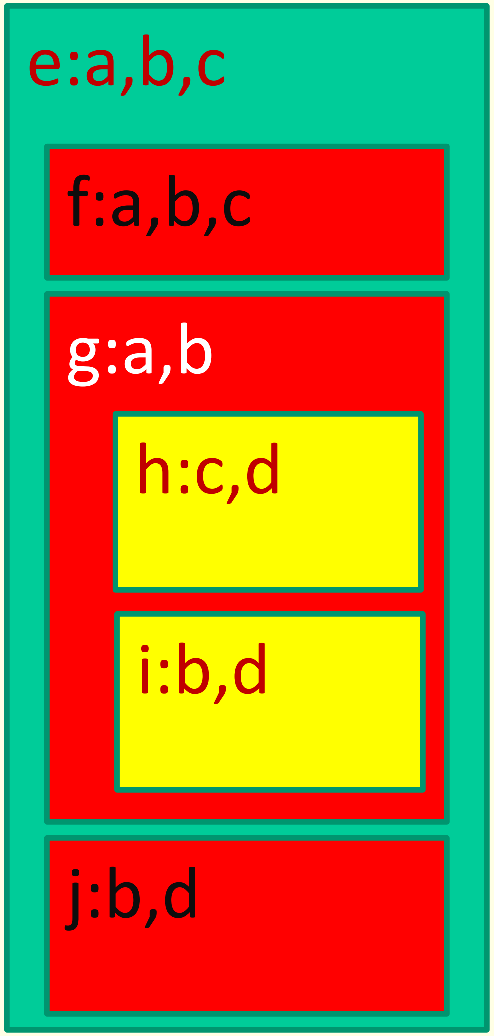
# Symbol attributes and symbol table entries

- Symbols have associated attributes
- typical attributes are name, type, scope, size, addressing mode etc.
- a symbol table entry collects together attributes such that they can be easily set and retrieved
- example of typical names in symbol table

| Name | Type |
|---|---|
| name | character string |
| class | enumeration |
| size | integer |
| type | enumeration |

# Nesting structure of an example Pascal program

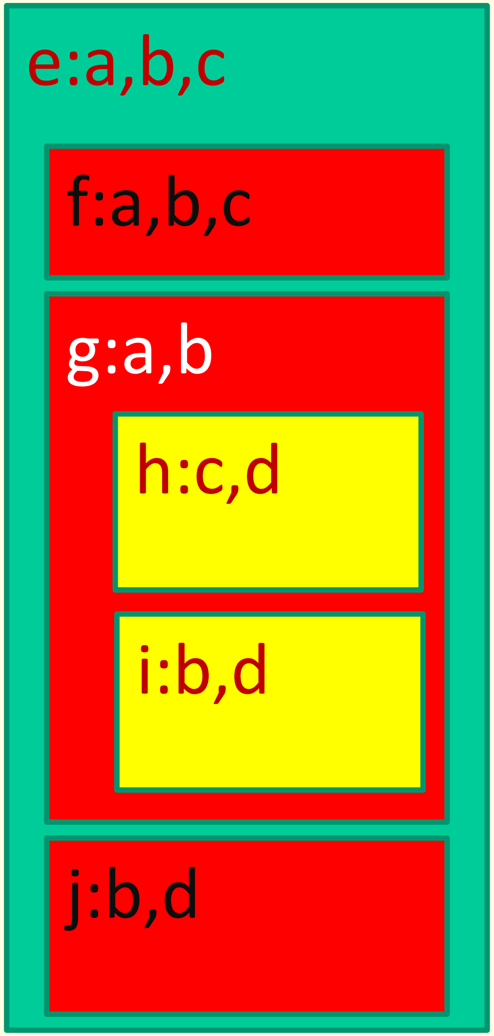
```
program e;
  var a, b, c: integer;

  procedure f;
    var a, b, c: integer;
    begin
       a := b+c
    end;

  procedure g;
    var a, b: integer;

    procedure h;
      var c, d: integer;
      begin
         c := a+d
      end;

    procedure i;
      var b, d: integer;
      begin
         b:= a+c
      end;
    begin
     ....
    end
  procedure j;
    var b, d: integer;
    begin
       b := a+d
    end;

begin
   a := b+c
end.
```

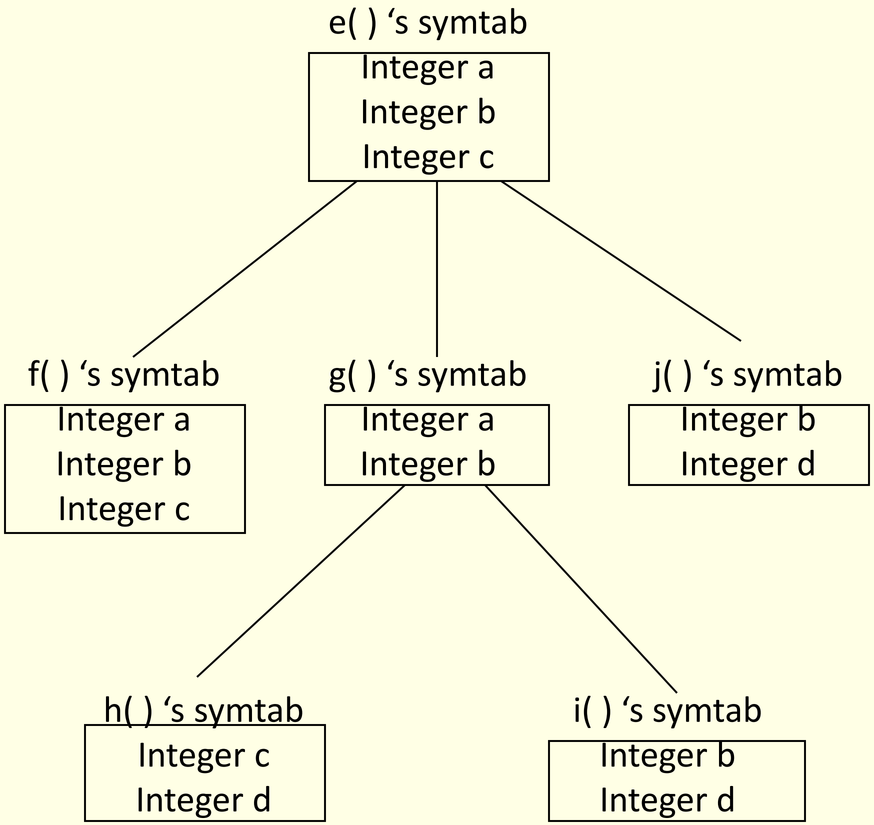


# Global Symbol table structure

- scope and visibility rules determine the structure of global symbol table
- for Algol class of languages scoping rules structure the symbol table as tree of local tables
  - global scope as root
  - tables for nested scope as children of the table for the scope they are nested in

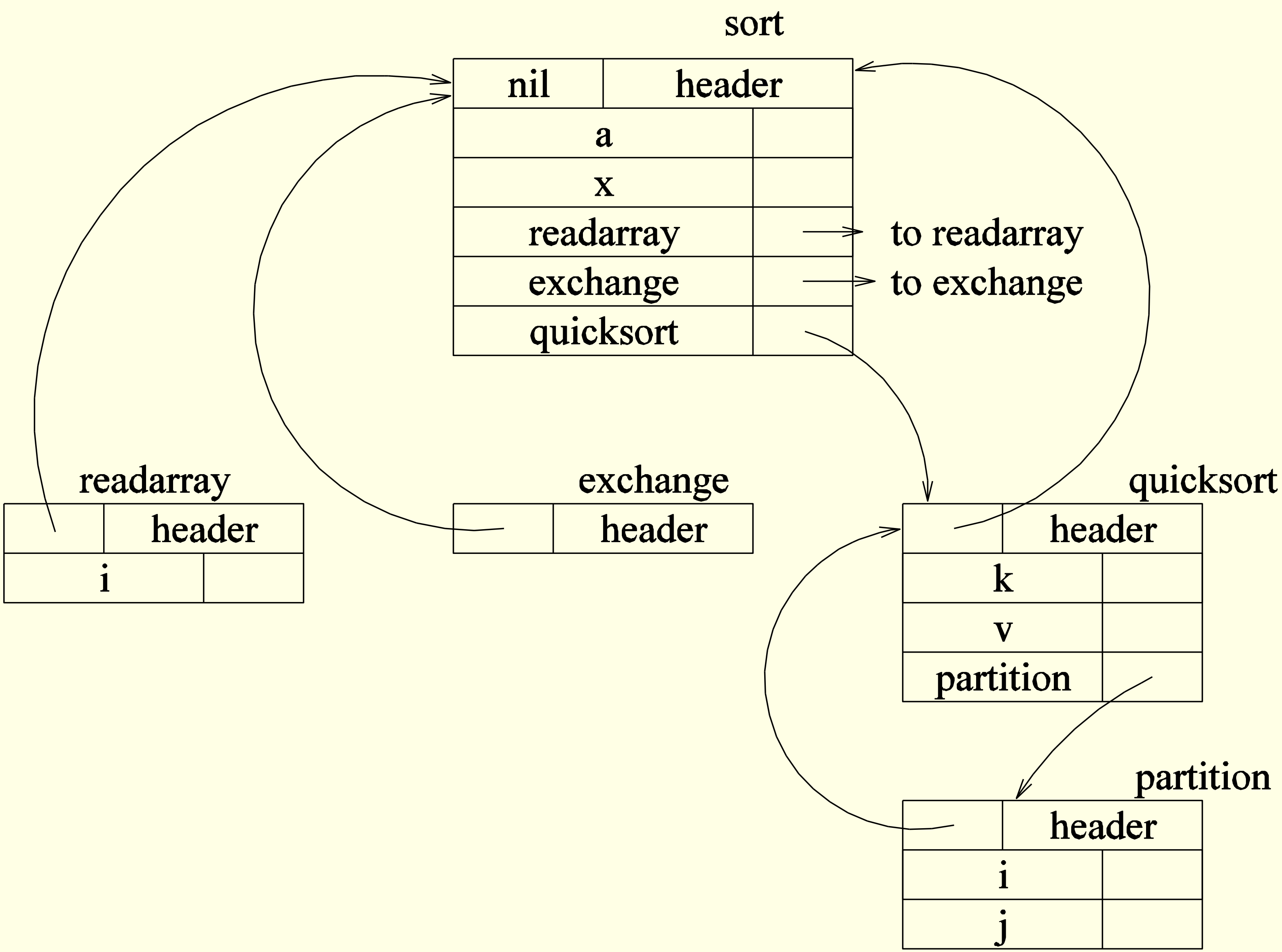
# Global Symbol table structure



# Example

```
program sort;
  var a : array[0..10] of integer;

    procedure readarray;
      var i :integer;
            :
    procedure exchange(i, j
                  :integer)
           :

procedure quicksort (m, n :integer);
      var i :integer;
    function partition (y, z
                  :integer) :integer;
         var i, j, x, v :integer;
               :
      i:= partition (m,n);
      quicksort (m,i-1);
      quicksort(i+1, n);
               :
begin{main}
    readarray;
    quicksort(1,9)
end.
```


sort
nil
header
a
x
readarray
exchange
quicksort
to readarray
to exchange
readarray
header
i
exchange
header
quicksort
header
k
v
partition
partition
header
i
j