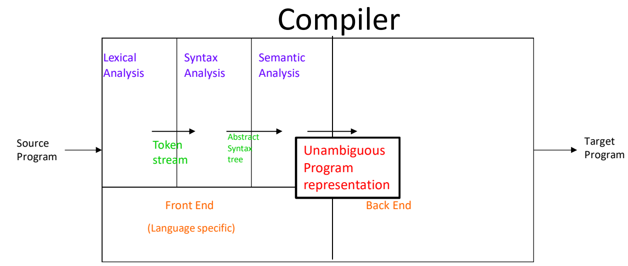
# Compiler Design

## I.R. and Symbol Tables

Amey Karkare
Department of Computer Science and Engineering
IIT Kanpur
karkare@iitk.ac.in

# 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
- IR should be simple and light weight while allowing easy expression of optimizations and transformations.

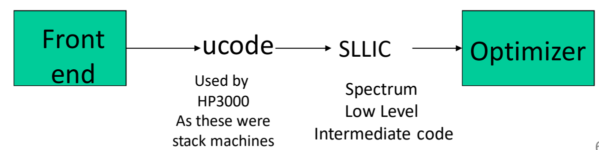
## Issues in IR Design

- 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[10][20]; use a[i][j+2]

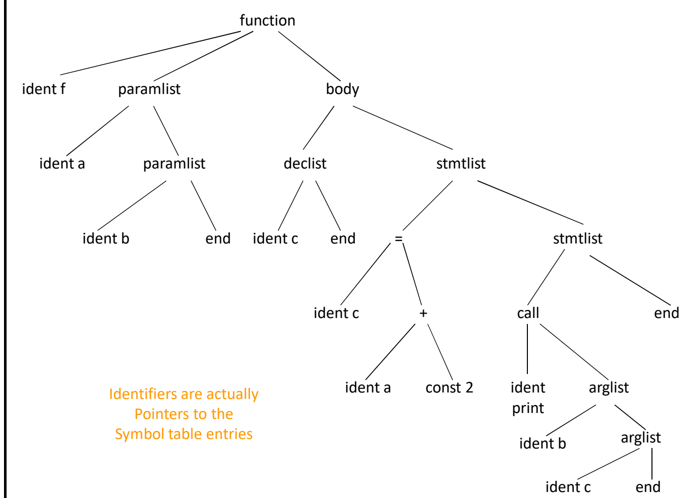
| HIR | MIR | LIR |
|---|---|---|
| t1←a[i,j+2] | t1← j+2 | r1← [fp-4] |
| | t2← i*20 | r2← r1+2 |
| | t3← t1+t2 | r3← [fp-8] |
| | t4← 4*t3 | r4← r3*20 |
| | t5← addr a | r5← r4+r2 |
| | t6← t4+t5 | r6← 4*r5 |
| | t7←*t6 | r7←fp-216 |
| | | f1← [r7+r6] |

# 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

9




10

- 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

11

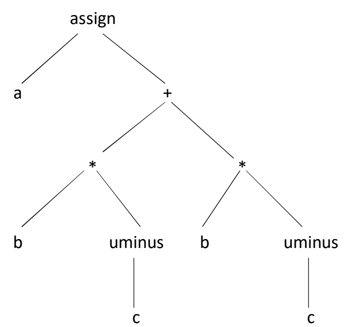
# Abstract Syntax Tree/DAG

- 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

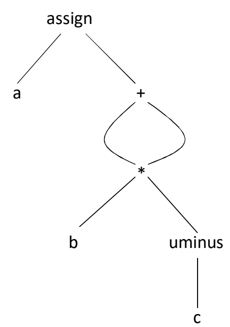
12

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

Abstract syntax tree




Directed Acyclic Graph






# Postfix notation

- Linearized representation of a syntax tree
- List of nodes of the tree
- Nodes appear immediately after its children
- The postfix notation for an expression E is defined as follows:
  - If E is a variable or constant then the postfix notation is E itself
  - If E is an expression of the form $E_1$ op $E_2$ where op is a binary operator then the postfix notation for E is
    - $E_1'$ $E_2'$ op where $E_1'$ and $E_2'$ are the postfix notations for $E_1$ and $E_2$ respectively
  - If E is an expression of the form ($E_1$) then the postfix notation for $E_1$ is also the postfix notation for E



# Postfix notation …

- No parenthesis are needed in postfix notation because
  - the position and parity of the operators permit only one decoding of a postfix expression
- Postfix notation for

  a = b * -c + b * - c

  is

  a b c - * b c - * + =



# 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 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

21

## Data Structures for Symbol Table

- 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 bookkeeping
- Hash table
  - The advantages are obvious

22

## 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)

23

## 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

24

- 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

25

## 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

26

## 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)

27

- most closely nested scope rule can be implemented in data structures discussed
  - give each procedure a unique number
  - blocks must also be numbered
  - procedure number is part of all local declarations
  - name is represented as a pair of number and name
- names are entered in symbol table in the order they occur
- most closely nested rule can be created in terms of following operations:
  - lookup: find the most recently created entry
  - insert: make a new entry
  - delete: remove the most recently created entry

28

## Symbol table structure

- Assign variables to storage classes that prescribe scope, visibility, and lifetime
  - scope rules prescribe the symbol table structure
  - scope: unit of static program structure with one or more variable declarations
  - scope may be nested
    - Pascal: procedures are scoping units
    - C: blocks, functions, files are scoping units
- Visibility, lifetimes, global variables
- Automatic or stack storage
- Static variables

29

## 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

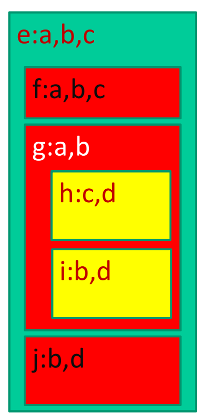
| Name | Type |
|---|---|
| name | character string |
| class | enumeration |
| size | integer |
| type | enumeration |

30

## 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.
```



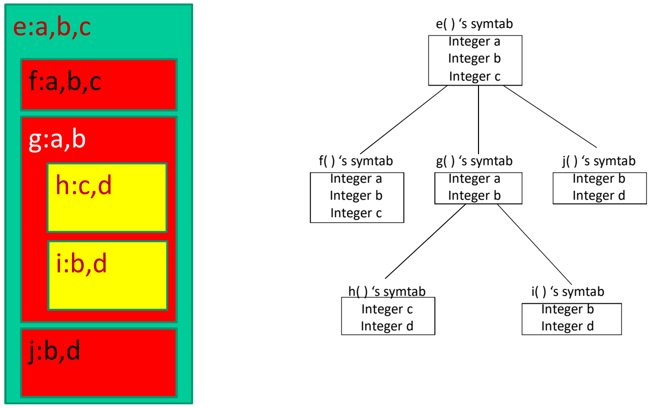

31

## 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

32

## Global Symbol table structure

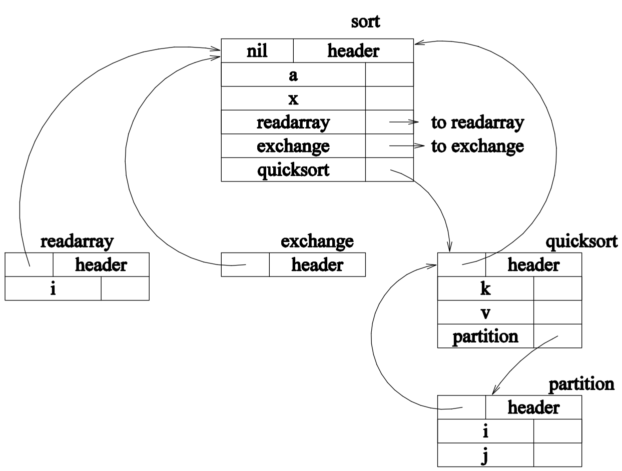



33

## 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.
```

34




35

## Storage binding and symbolic registers

- Translates variable names into addresses
- This process must occur before or during code generation
- each variable is assigned an address or addressing method
- each variable is assigned an offset with respect to base which changes with every invocation
- variables fall in four classes: global, global static, stack, local (non-stack) static

36

- global/static: fixed relocatable address or offset with respect to base as global pointer
- stack variable: offset from stack/frame pointer
- allocate stack/global in registers
- registers are not indexable, therefore, arrays cannot be in registers
- assign symbolic registers to scalar variables
- used for graph coloring for global register allocation



a: global  b: local  c[0..9]: local
gp: global pointer  fp: frame pointer

| MIR | LIR | LIR |
|---|---|---|
| a ← a*2 | r1 ← [gp+8] | s0 ← s0*2 |
| | r2 ← r1*2 | |
| | [gp+8] ← r2 | |
| b ← a+c[1] | r3 ← [gp+8] | s1 ← [fp-28] |
| | r4 ← [fp-28] | s2 ← s0+s1 |
| | r5 ← r3+r4 | |
| | [fp-20]←r5 | |
| | Names bound to locations | Names bound to symbolic registers |



## Local Variables in Frame

- assign to consecutive locations; allow enough space for each
  - may put word size object in half word boundaries
  - requires two half word loads
  - requires shift, or, and
- align on double word boundaries
  - wastes space
  - machine may allow small offsets



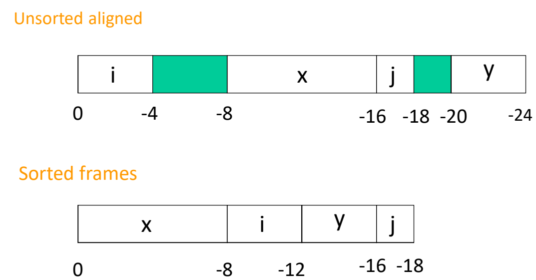
- sort variables by the alignment they need
- store largest variables first
  - automatically aligns all the variables
  - does not require padding
- store smallest variables first
  - requires more space (padding)
  - for large stack frame makes more variables accessible with small offsets

## How to store large local data structures

- Requires large space in local frames and therefore large offsets
- If large object is put near the boundary other objects require large offset either from fp (if put near beginning) or sp (if put near end)
- Allocate another base register to access large objects
- Allocate space in the middle or elsewhere; store pointer to these locations from at a small offset from fp
- Requires extra loads



```
int i;
double float x;
short int j;
float y;
```






## Symbol Table Creation



## Declarations

P → D

D → D ; D

D → id : T

T → integer

T → real

# Declarations

For each name create symbol table entry with information like type and relative address

P → {offset=0} D

D → D ; D

D → id : T

enter(id.name, T.type, offset);
offset = offset + T.width

T → integer

T.type = integer; T.width = 4

T → real

T.type = real; T.width = 8



# Declarations …

T → array [ num ] of $T_1$

T.type = array(num.val, $T_1$.type)

T.width = num.val x $T_1$.width

T → ↑$T_1$

T.type = pointer($T_1$.type)

T.width = 4



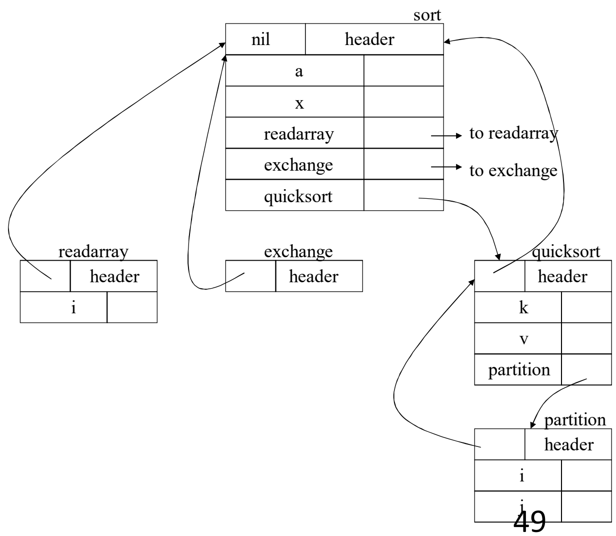
# Keeping track of local information

- when a nested procedure is seen, processing of declaration in enclosing procedure is temporarily suspended
- assume following language
  P → D
  D → D ;D | id : T | proc id ;D ; S
- a new symbol table is created when procedure declaration
  D → proc id ; $D_1$ ; S is seen
- entries for $D_1$ are created in the new symbol table
- the name represented by id is local to the enclosing procedure



# Example

```
program sort;
    var a : array[1..n] of integer;
        x : integer;
    procedure readarray;
        var i : integer;
         ……
    procedure exchange(i,j:integers);
         ……
    procedure quicksort(m,n : integer);
        var k,v : integer;
            function partition(x,y:integer):integer;
                var i,j: integer;
                    ……
         ……
begin{main}
   ……
end.
```




49

## Creating symbol table: Interface

- mktable (previous)
  create a new symbol table and return a pointer to the new table. The argument previous points to the enclosing procedure
- enter (table, name, type, offset)
  creates a new entry
- addwidth (table, width)
  records cumulative width of all the entries in a table
- enterproc (table, name, newtable)
  creates a new entry for procedure name. newtable points to the symbol table of the new procedure
- Maintain two stacks: (1) symbol tables and (2) offsets
- Standard stack operations: push, pop, top

50

## Creating symbol table …

D → proc id;
  {t = mktable(top(tblptr));
  push(t, tblptr); push(0, offset)}
 $D_1$; S
  {t = top(tblptr);
  addwidth(t, top(offset));
  pop(tblptr); pop(offset);
  enterproc(top(tblptr), id.name, t)}

D → id: T
  {enter(top(tblptr), id.name, T.type, top(offset));
  top(offset) = top (offset) + T.width}

51

## Creating symbol table …

P →
  {t=mktable(nil);
  push(t,tblptr);
  push(0,offset)}
 D
  {addwidth(top(tblptr),top(offset));
  pop(tblptr); // save it somewhere!
  pop(offset)}

D → D ; D

52

## Field names in records

T → record
{t = mktable(nil);
push(t, tblptr); push(0, offset)}
D end
{T.type = record(top(tblptr));
T.width = top(offset);
pop(tblptr); pop(offset)}

53

## Names in the Symbol table

S → id := E
{p = lookup(id.place);
if p <> nil then emit(p := E.place)
else error}

E → id
{p = lookup(id.name);
if p <> nil then E.place = p
else error}

54

## Addressing Array Elements

- Arrays are stored in a block of consecutive locations
- assume width of each element is w
- ith element of array A begins in location
  base + (i - low) x w
  where base is relative address of A[low]
- the expression is equivalent to
  i x w + (base-low x w)
  → i x w + const

55

## 2-dimensional array

- storage can be either row major or column major
- in case of 2-D array stored in row major form address of $A[i_1, i_2]$ can be calculated as

  $base + ((i_1 - low_1) \times n_2 + i_2 - low_2) \times w$

  where $n_2 = high_2 - low_2 + 1$
- rewriting the expression gives

  $((i_1 \times n_2) + i_2) \times w + (base - ((low_1 \times n_2) + low_2) \times w)$
  $\rightarrow ((i_1 \times n_2) + i_2) \times w + constant$
- this can be generalized for $A[i_1, i_2, \ldots, i_k]$

56

## Example

- Let A be a 10x20 array, low indices at 1.
  therefore, $n_1 = 10$ and $n_2 = 20$
  and assume $w = 4$
- code to access A[y,z] is

  $t_1 = y * 20$

  $t_1 = t_1 + z$

  $t_2 = 4 * t_1$

  $t_3 = addr(A) - 84$ $\{((low_1 X n_2)+low_2)Xw)=(1*20+1)*4=84\}$

  $t_4 = t_2 + t_3$

  $x = t_4$

57