Compiler Design

Intermediate Representations

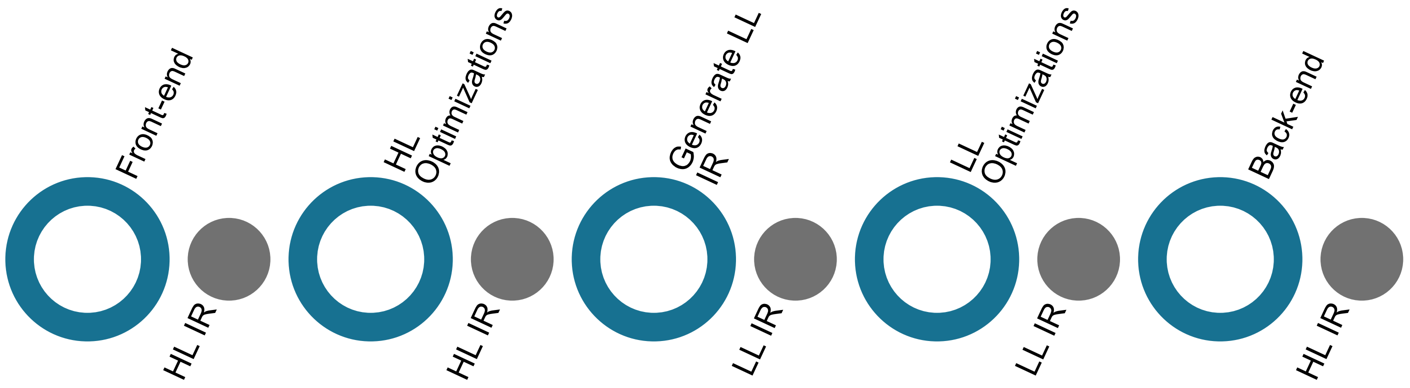
Dr. Nicolai Stawinoga, Dr. Biagio Cosenza | TU Berlin | Wintersemester 2022-23

You are here

Lexical Analysis
Syntax Analysis
Semantic Analysis
Intermediate code gen.
I. C. Optimization
Code Generation
Target code optimization
Target code generation

Intermediate Representations

- The Intermediate Representation (IR) encodes all the knowledge that the compiler has derived about the source program
  - ➢to follow: back-end transforms the code, as represented by the IR, into target code
- Recall: middle-end, may transform the code represented by the IR into equivalent code that may perform more efficiently
  - ➢Typically:



Technische Universität Berlin

About Intermediate Representations

- Why use an intermediate representation?
  - to facilitate retargeting
  - to enable machine independent-code optimizations or more aggressive code generation strategies
- Design issues
  - ease of generation
  - ease of manipulation
  - cost of manipulation
  - level of abstraction
  - size of typical procedure
- Decisions in the IR design have major effects on the speed and effectiveness of compiler

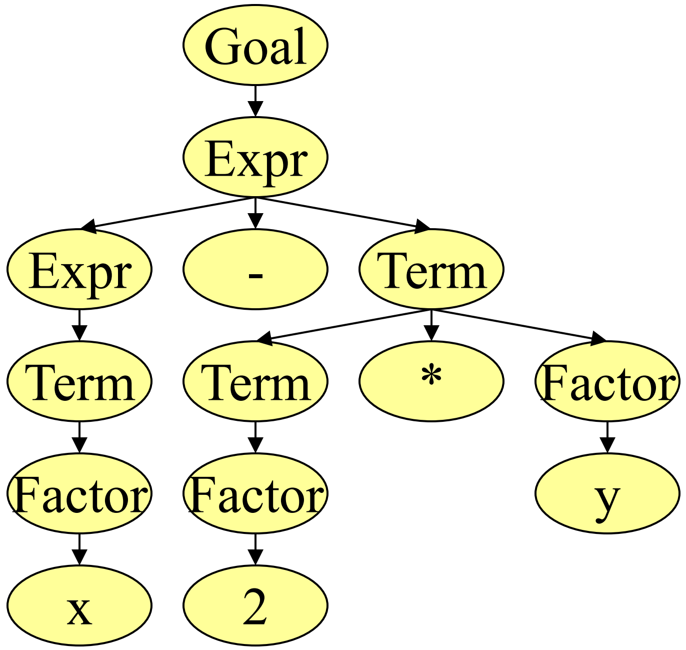
AES Embedded Systems Architecture

About Intermediate Representations

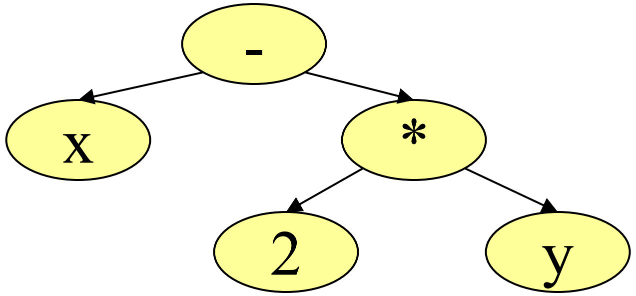
- Level of abstraction:
  - Code representation: AST, 3-address code, stack code, SSA form
  - Analysis representation (may have several at a time): CFG (Control Flow Graph), ...
- Categories of IRs by structure
  - **Structural** (graphical): trees, DAGs; used in source-to-source translators; node and edge structures tend to be large
  - **Linear**: pseudo-code for some abstract machine: three address code, stack machine code
  - **Hybrid** (combination of the above): Control-Flow Graph
- There is no universally good IR
  - the right choice depends on the goals of the compiler!

From Parse Trees to Abstract Syntax Trees

- Why we don't want to use the parse tree?
  - ➢ quite a lot of unnecessary information
- How to convert a parse tree to an abstract syntax tree?
  - ➢ Traverse in postorder (postfix)
  - ➢ Use `mkleaf` and `mknode` where appropriate
  - ➢ Match action with grammar rule



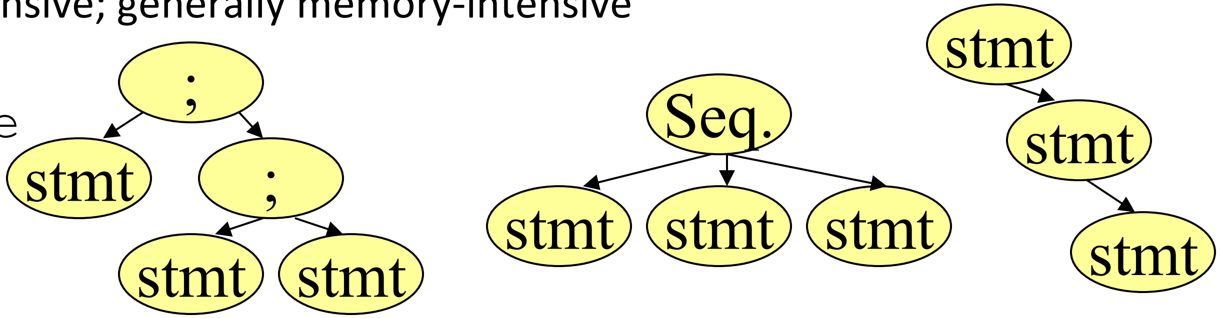
Abstract Syntax Trees



- An Abstract Syntax Tree (AST) is the procedure's parse tree with the non-terminal symbols removed
  - Example: `x - 2*y`
- The AST is a near source-level representation
- Source code can be easily generated
  - perform an in-order tree walk (first the left subtree, then the root, then the right subtree)
  - printing each node as visited
  - Issues: traversals and transformations are pointer-intensive; generally memory-intensive
- Example:

```
Stmt_sequence → stmt; Stmt_sequence
              | stmt
```

  - At least 3 AST versions for "`stmt; stmt; stmt`"



AST Example: Clang AST

- A simple test code

```
int f(int x) {
  int result = (x / 42);
  return result;
}
```

- Compiled with Clang:
  - ➢ `clang -Xclang -ast-dump -fsyntax-only test.cc`

```
TranslationUnitDecl 0x5aea0d0 <<invalid sloc>>
... cutting out internal declarations of clang ...
`-FunctionDecl 0x5aeab50 <test.cc:1:1, line:4:1> f 'int (int)'
  |-ParmVarDecl 0x5aeaa90 <line:1:7, col:11> x 'int'
  `-CompoundStmt 0x5aead88 <col:14, line:4:1>
    |-DeclStmt 0x5aead10 <line:2:3, col:24>
    | `-VarDecl 0x5aeac10 <col:3, col:23> result 'int'
    |   `-ParenExpr 0x5aeacf0 <col:16, col:23> 'int'
    |     `-BinaryOperator 0x5aeacc8 <col:17, col:21> 'int' '/'
    |       |-ImplicitCastExpr 0x5aeacb0 <col:17> 'int'
<LValueToRValue>
    |       | `-DeclRefExpr 0x5aeac68 <col:17> 'int' lvalue
ParmVar 0x5aeaa90 'x' 'int'
    |       `-IntegerLiteral 0x5aeac90 <col:21> 'int' 42
    `-ReturnStmt 0x5aead68 <line:3:3, col:10>
      `-ImplicitCastExpr 0x5aead50 <col:10> 'int'
<LValueToRValue>
        `-DeclRefExpr 0x5aead28 <col:10> 'int' lvalue Var
0x5aeac10 'result' 'int'
```

AST Example: dHPF
High Performance Fortran

- Fortran

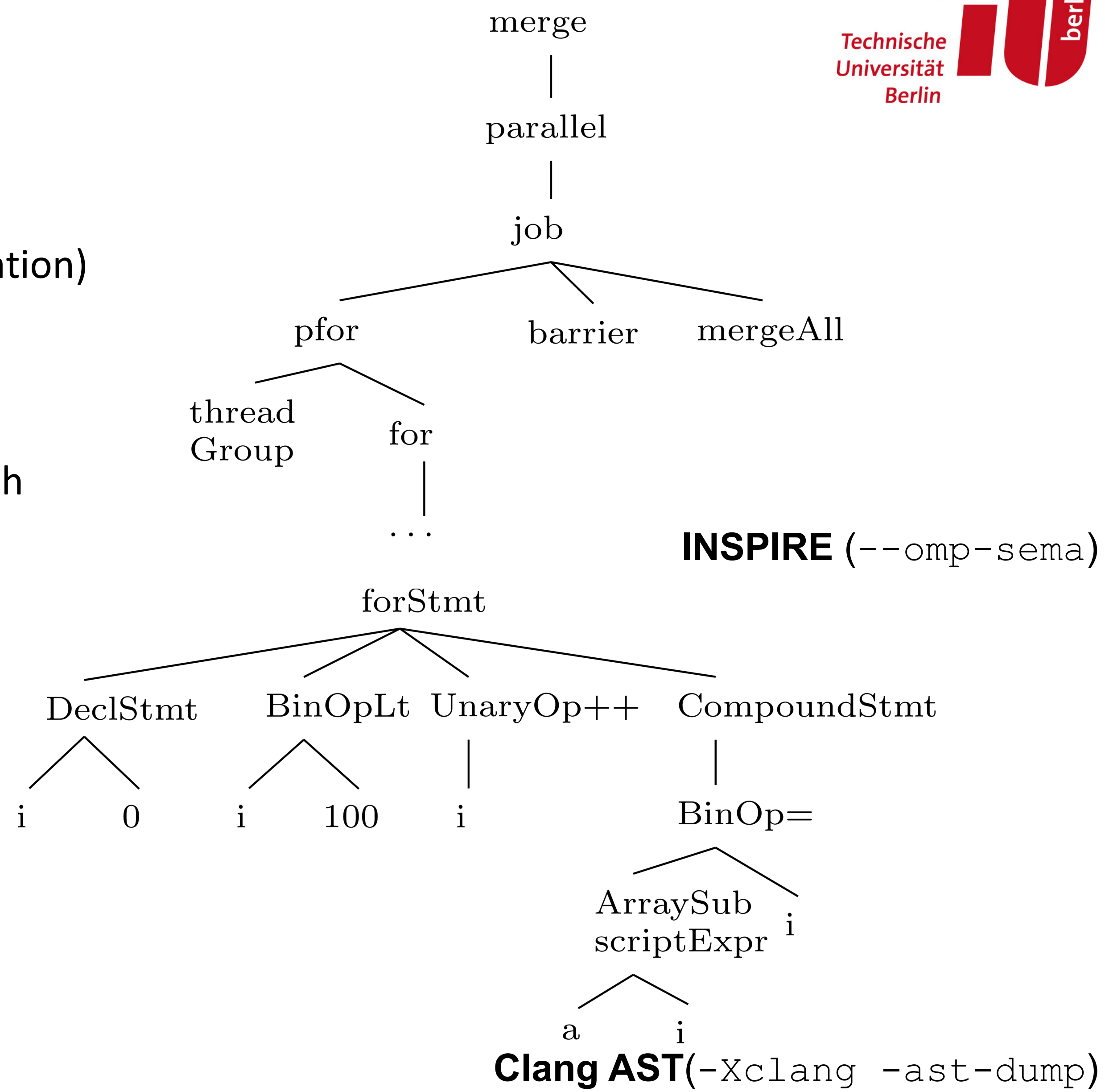
```
PROGRAM MAIN
      REAL A(100), X
!HPF$ PROCESSORS P(4)
!HPF$ DISTRIBUTE A(BLOCK) ONTO P
      FORALL (i=1:100) A(i) = X+1
                 CALL FOO(A)
      END

      SUBROUTINE FOO(X)
      REAL X(100)
!HPF$ INHERIT X
      IF (X(1).EQ.0) THEN
        X = 1
      ELSE
        X = X + 1
      END IF
      RETURN
      END
```

```
(1[GLOBAL]                                 ! 2 components in true-branch
   ((2[PROG_HEDR]                          ! 6 components in true-branch
      (3[VAR_DECL]
       4[PROCESSORS_STMT]
       5[DISTRIBUTE_DECL]
       (6[FORALL_STMT]
          (7[ASSIGN_STAT]
           8[CONTROL_END]
          )
          NULL
       )
       9[PROC_STAT]
       10[CONTROL_END]
      ) NULL
    )
    (11[PROC_HEDR]
       (12[VAR_DECL]
        13[INHERIT_DECL]
        (14[LOGIF_NODE]                    ! both branches are non empty
           (15[ASSIGN_NODE]
            16[CONTROL_END]
           )
           (17[ASSIGN_NODE]
            18[CONTROL_END]
           )
        )
        19[RETURN_STAT]
        20[CONTROL_END]
       ) NULL
    ) NULL
)
```

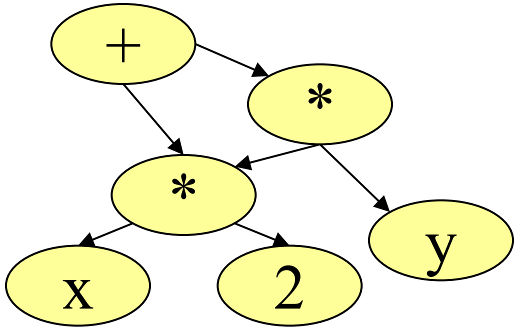
AST Example: Insieme Compiler

- Source-to-source compiler, parallel by design
- INSPIRE (INSieme Parallel Intermediate Representation)
  - ➢ high-level IR
  - ➢ all transformations happens at AST level
  - ➢ output is C/C++ code enabling parallelization with pthreads, MPI, OpenCL
  - ➢ parallelism is expressed in the IR

```
int a [N];
#pragma omp parallel
{
  #pragma omp for
  for(int i=0; i<N; i++){
    a[i]=i;
  }
}
```



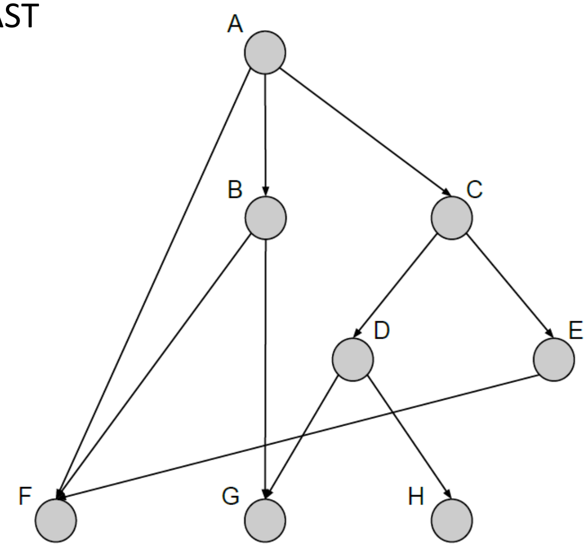
Abstract Semantic Graph (ASGs)



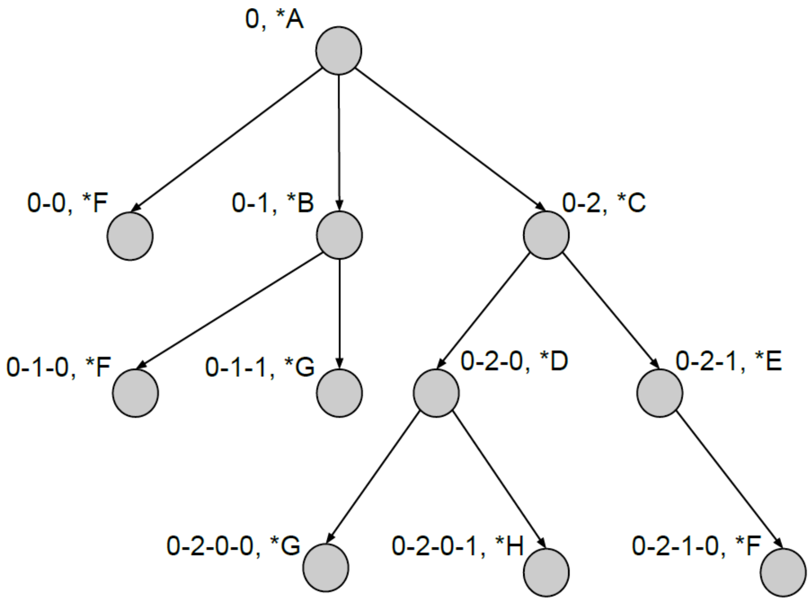
- A DAG (directed acyclic graph) representing an AST
- has a unique node for each value
  - ➢example: the DAG for `x*2+x*2*y`
- Powerful representation, encodes redundancy; but difficult to transform, not useful for showing control-flow
- Construction
  - ➢replace constructors used to build an AST with versions that remember each node constructed by using a table.
  - ➢traverse the code in another representation
- **Exercise**: Construct the AST and ASG for `x=2*x+sin(2*x); z=x/2`

ASG Example: Insieme Compiler

- IR is both a DAG and a tree
  - ➢shared nodes
  - ➢self contained: there is no explicit symbol table
  - ➢can be traversed as AST or ASG
    - if you look at the addresses, they form an AST
    - if you look at the node, an ASG



(a) INSPIRE node DAG



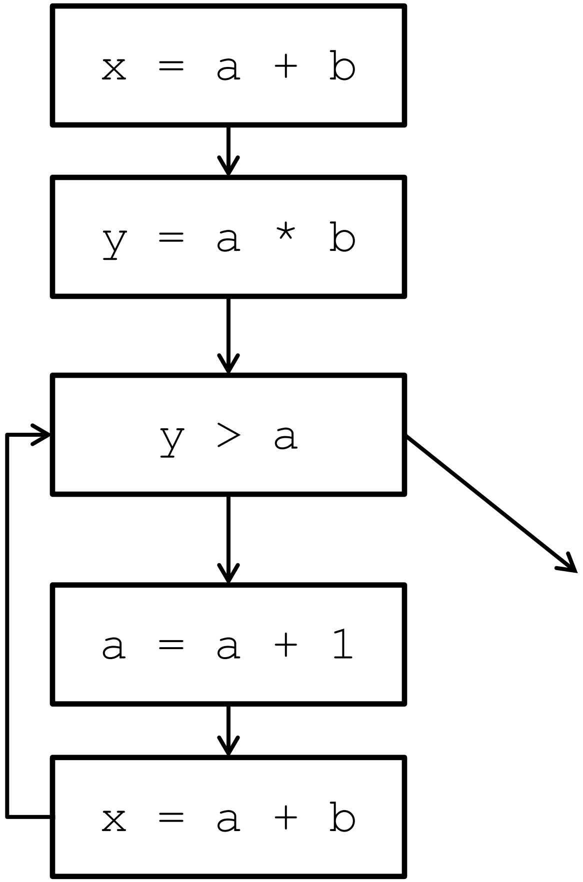
(b) INSPIRE address AST

Auxiliary Graph Representations

- Control-Flow Graph (CFG): models the way that the code transfers control between blocks in the procedure.
  - Node: instruction or code block (“basic block”, code block with no branches)
  - Edge: transfer of control between instructions
  - (Captures loops, if statements, case, goto)
- Data Dependency Graph: encodes the flow of data
  - Node: program statement
  - Edge: connects two nodes if one uses the result of the other
  - Useful in examining the legality of program transformations
- Call Graph: shows dependences between procedures
  - Useful for inter-procedural analysis

Control-Flow Graph Example

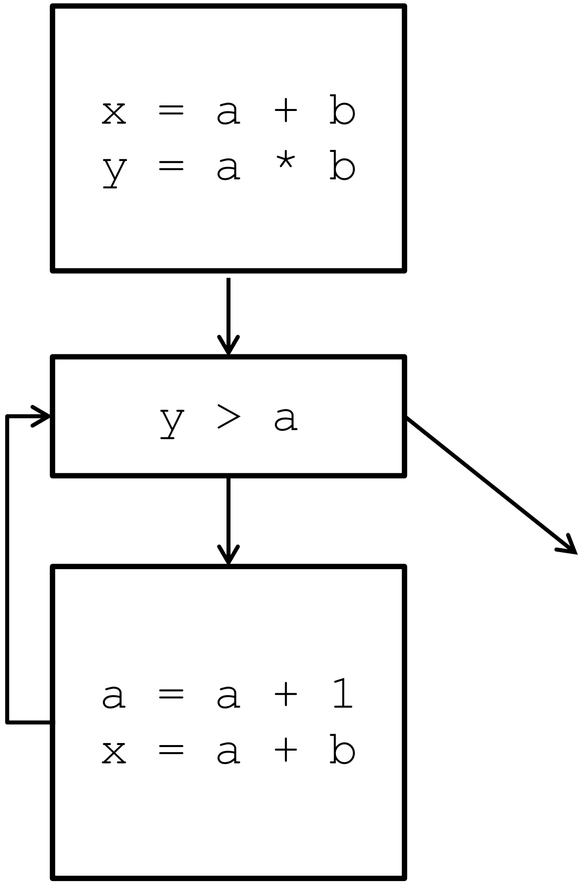
```
x = a + b;
y = a * b;
while (y > a) {
      a = a + 1;
      x = a + b;
}
```



CFG with Basic Blocks

- Basic Block: Straight-line code sequence with no branching (except at entry and exit)
  - ➢ more efficient representation

```
x = a + b;
y = a * b;
while (y > a) {
        a = a + 1;
        x = a + b;
}
```



Exercise (1)

- Draw the Control-Flow Graph of the following

```
1. stmtlist1
2. if (x=y){
3.   stmtlist2
4.   stmtlist3
5. }
6. else {
7.   stmtlist4
8. }
9. stmtlist5
```

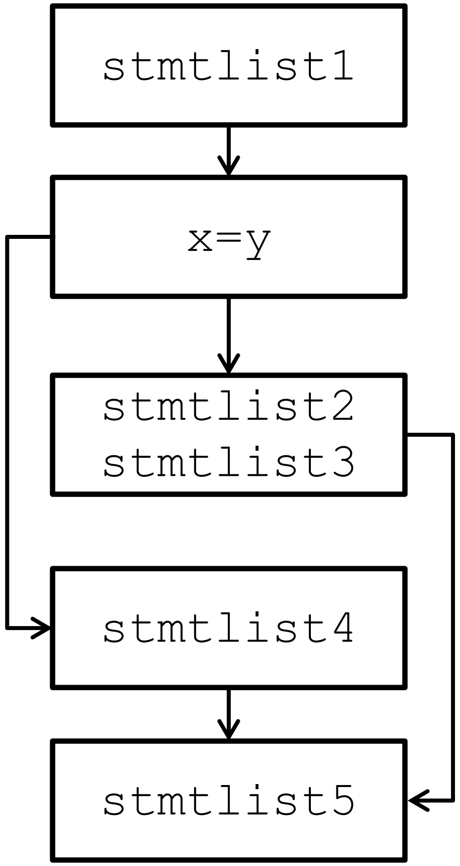
Example with a for loop

```
1. stmtlist1
2. for (int i=0; i<N; i++)
3.   stmtlist2
4. stmtlist3
```

Exercise (1) – CFG for if-then-else

- Draw the Control-Flow Graph of the following

```
1. stmtlist1
2. if (x=y){
3.    stmtlist2
4.    stmtlist3
5. }
6. else {
7.    stmtlist4
8. }
9. stmtlist5
```



Exercise (2)

- Draw the data dependency graph of

```
sum=0
done=0
while !done do
  read j
  if (j>0)
    sum=sum+j
    if (sum>100)
      done=1
    else
      sum=sum+1
    endif
  endif
endwhile
print sum
```

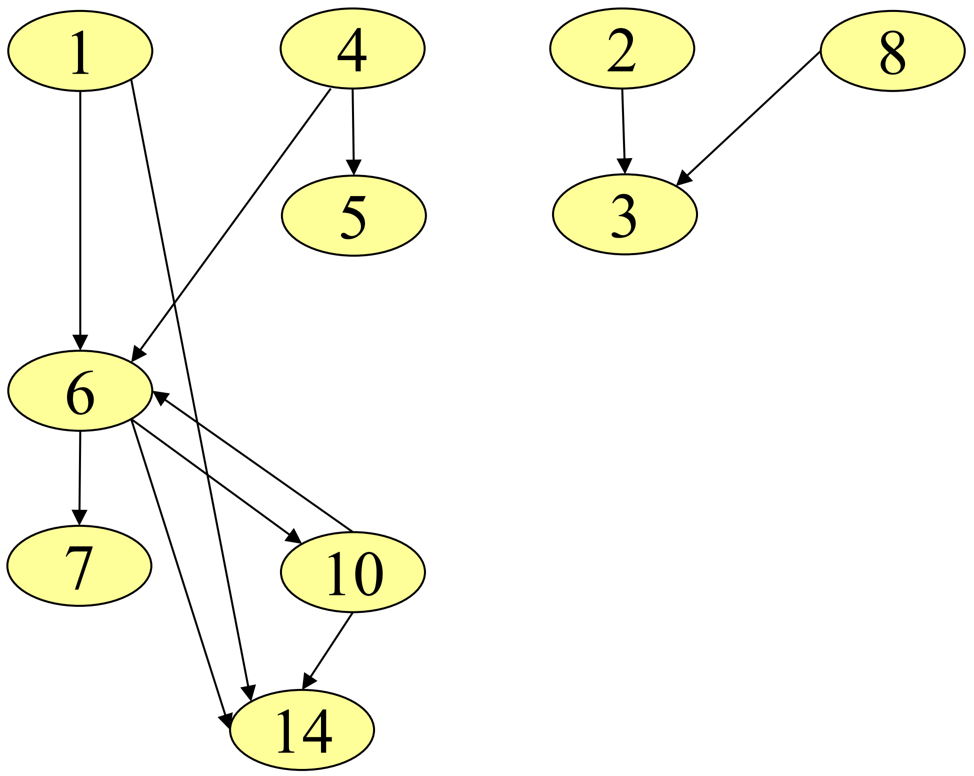
- What about this?

```
do i=1,n
   A(I)=a(3*I+10)
```

Exercise (2)

- Draw the data dependency graph of

```
1.  sum=0
2.  done=0
3.  while !done do
4.    read j
5.    if (j>0)
6.      sum=sum+j
7.      if (sum>100)
8.        done=1
9.      else
10.       sum=sum+1
11.     endif
12.   endif
13.endwhile
14.print sum
```



Exercises (3)

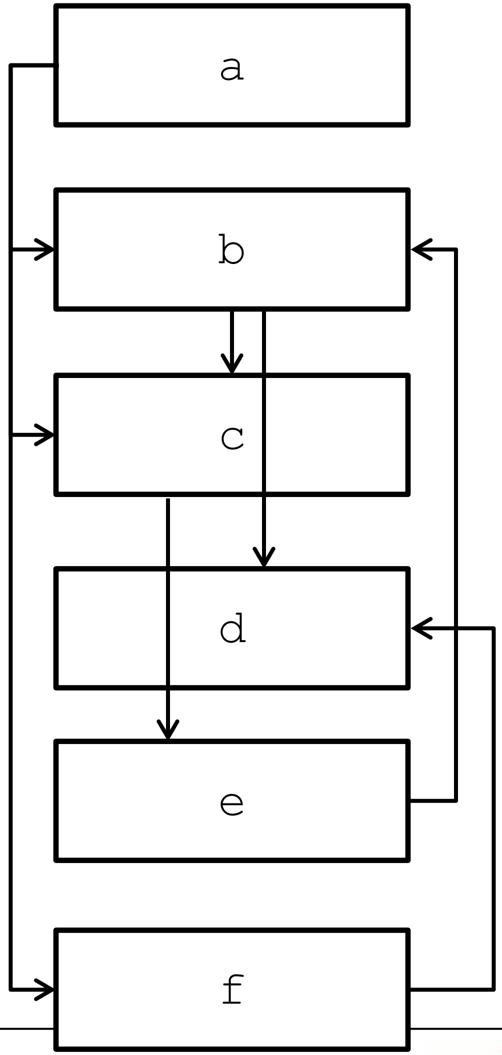
- Draw the Call Graph

1. `void a() { … b() … c() … f() … }`
2. `void b() { … d() … c() … }`
3. `void c() { … e() … }`
4. `void d() { … }`
5. `void e() { … b() … }`
6. `void f() { … d() …}`

Exercises (3) – solution

- Draw the Call Graph

```
1. void a() { … b() … c() … f() … }
2. void b() { … d() … c() … }
3. void c() { … e() … }
4. void d() { … }
5. void e() { … b() … }
6. void f() { … d() …}
```



Three-address Code (TAC / 3AC)

- A term used to describe many different representations: each statement is a single operator and at most three operands
- Example:

```
if (x>y) then z=x-2*y
```

→

```
        t1=load x
        t2=load y
        t3=t1>t2
        if not(t3) goto L
        t4=2*t2
        t5=t1-t4
        z=store t5
L: …
```

- Advantages: compact form, makes intermediate values explicit, resembles many machines

Three-address Code (2)

- Storage considerations: How to store a three-address code?
  - **Quadruples**: a table with 4 small `int` (used by the original Fortran)
  - **Triples**: index used as implicit name (less space, but hard to **reorder**)
  - **Indirect triples**: list triples in a statement list (a trade-off)

Triples' Problem: optimization may change the execution order
Statement array: uses more space than triples, but easier to reorder

```
1. load r1, y
2. loadi r2, 2
3. mult r3,r2,r1
4. load r4, x
5. sub r5,r4,r3
```

Quadruples				Triples				Indirect Triples				
load	1	y	-	(1)	load	y		**(103)**	(100)	load	y	
loadi	2	2	-	(2)	loadi	2		**(101)**	(101)	loadi	2	
mult	3	2	1	(3)	mult	(1)	(2)	**(100)**	(102)	mult	(100)	(101)
load	4	x	-	(4)	load	x		**(102)**	(103)	load	x	
sub	5	4	3	(5)	sub	(4)	(3)	**(104)**	(104)	sub	(103)	(104)

Other Linear Representations

- Two address code is more compact: In general, it allows statements of the form `x = x <op> y`
  - ➢ single operator and at most two operands

```
load  r1,y
loadi r2,2
mult  r2,r1
load  r3,x
sub   r3,r2
```

- One address code (also called stack machine code) is more compact
  - ➢ useful in environments where space is at a premium (has been used to construct bytecode interpreters for Java)
  - ➢ sometime distinguish: zero-address (stack) vs one-address (accumulator)

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

Using Multiples Representations: Examples

- SiMPLE back-end compiler
  - Code: CFG + 3-address code
  - Analysis info: value DAG (represents dataflow in basic block)
- Sun Compilers for SPARC
  - Code: 2 different IRs
  - Analysis info: CFG + dependency graph + ?
  - High-level IRs: linked list of triples
  - Low-level IRs: SPARC assembly like operations
- IBM Compilers for Power, PowerPC:
  - Code: Low-level IR
  - Analysis info: CFG + value graph + dataflow graphs
- dHPF compiler
  - Code: AST
  - Analysis info: CFG + SSA + Value DAG + Call Graph
- GCC, LLVM, Java, PHP
  - Details in the next lectures

Loop-specific Representation: the Polyhedral Model

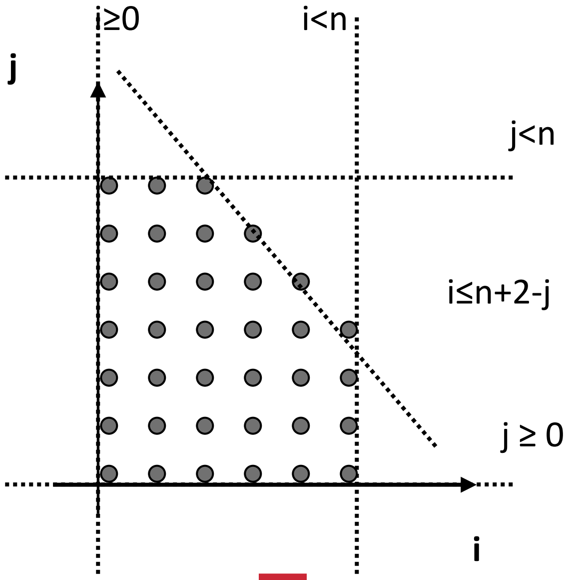
- A representation used to apply code-transformation on affine loops
  - an affine loop is represented with (different) matrices
  - typically used for automatic parallelization and loop transformations in general
- From/to polyhedral representation
  1. from the affine loop (e.g., in the AST) to the polyhedral representation
  2. one or more code transformations are applied on the polyhedral representation
  3. the polyhedra can be also used to understand how to transform the code
  4. a “code generator” goes back from the polyhedral representation to the original loop representation, e.g., transformed AST

Polyhedral Model for Iteration Space

Sketch only.
See lecture on loop opt.

```
for(int i=0; i<n; i++)
  for(int j=0; j<n; j++)
    if(i <= n+2-j)
(s)     b[j] = b[j] + a[i];
```

Code -> AST -> polyhedral model



$$\begin{bmatrix} 1 & 0 \\ -1 & 0 \\ 0 & 1 \\ 0 & -1 \\ -1 & -1 \end{bmatrix} \begin{pmatrix} i \\ j \end{pmatrix} + \begin{pmatrix} 0 \\ n-1 \\ 0 \\ n-1 \\ -n-2 \end{pmatrix} \geq \vec{0}$$

Iteration domain of S

$$\begin{array}{cccc} & & n & \text{constant} \\ \end{array}$$
$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ -1 & 0 & 1 & -1 \\ 0 & 1 & 0 & 0 \\ 0 & -1 & 1 & -1 \\ -1 & -1 & -1 & -2 \end{bmatrix} \begin{pmatrix} i \\ j \\ n \\ 1 \end{pmatrix} \geq \vec{0}$$

Iteration domain with homogenous coord.

Code transformation

polyhedral model -> AST

Symbol Tables

- Introduce a central repository of identifiers
  - ➢ symbol table or sets of symbol tables
- Associate with each production a snippet of code that would execute each time the parser reduces that production - action routines
- Examples:
  - ➢ code that checks if a variable is declared prior to use (on a production like `Factor→id`)
  - ➢ code that checks that each operator and its operands are type-compatible (on a production like `Term → Term*Factor`)
- Allowing arbitrary code provides flexibility
- Evaluation fits nicely with LR(1) parsing
- Symbol tables are retained across compilation (carry on for debugging too)

What information is stored in the symbol table?

- What items to enter in the symbol table?
  - Variable names, defined constants, procedure and function names, literal constants and strings, source text labels, compiler-generated temporaries
- What kind of information might the compiler need about each item?
  - textual name, data type, declaring procedure, storage information
  - depending on the type of the object, the compiler may want to know list of fields (for structures), number of parameters and types (for functions), etc…
- In practice, many different tables may exist
- Symbol table information is accessed frequently: efficiency of access is critical!

Organizing the Symbol Table

- Linear List
  - Simple approach, has no fixed size; but inefficient: a lookup may need to traverse the entire list: this takes **O(n)**
- Binary tree
  - An unbalanced tree would have similar behavior as a linear list (this could arise if symbols are entered in sorted order)
  - A balanced tree (path length is roughly equal to all its leaves) would take **$O(\log_2 n)$** probes per lookup (worst-case). Techniques exist for dynamically rebalancing trees
- Hash table
  - Uses a hash function, `h`, to map names into integers; this is taken as a table index to store information
  - Potentially **O(1)**, but needs inexpensive function, with good mapping properties, and a policy to handle cases when several names map to the same single index

Lexical Scoping

- Many languages introduce independent name scopes
  - C, for example, may have global, static (file), local and block scopes
  - Pascal: nested procedure declarations
  - C++, Java: class inheritance, nested classes
  - C++, Java, Modula: packages, namespaces, modules, etc…
  - Namespaces allow two different entities to have the same name within the same scope: E.g.: In Java, a class and a method can have the same name (Java has six namespaces: packages, types, fields, methods, local variables, labels)
- Problems
  - at point `x`, which declaration of variable `y` is current?
  - as parser goes in and out of scopes, how does it track `y`?
    - allocate and initialize a symbol table for each level

Summary

- Intermediate representations
- Three address codes, quadruples, triples, indirect triples
- Control-Flow Graph, Data-Dependency Graph, Call Graph
- Symbol table
- Readings
  - [ALSU] Construction of syntax tree 5.3.1; Intermediate Representation 6.1, 6.2.1-3