CS536
Intermediate Code/Lang/Represention

A Sahu

CSE, IIT Guwahati

Outline

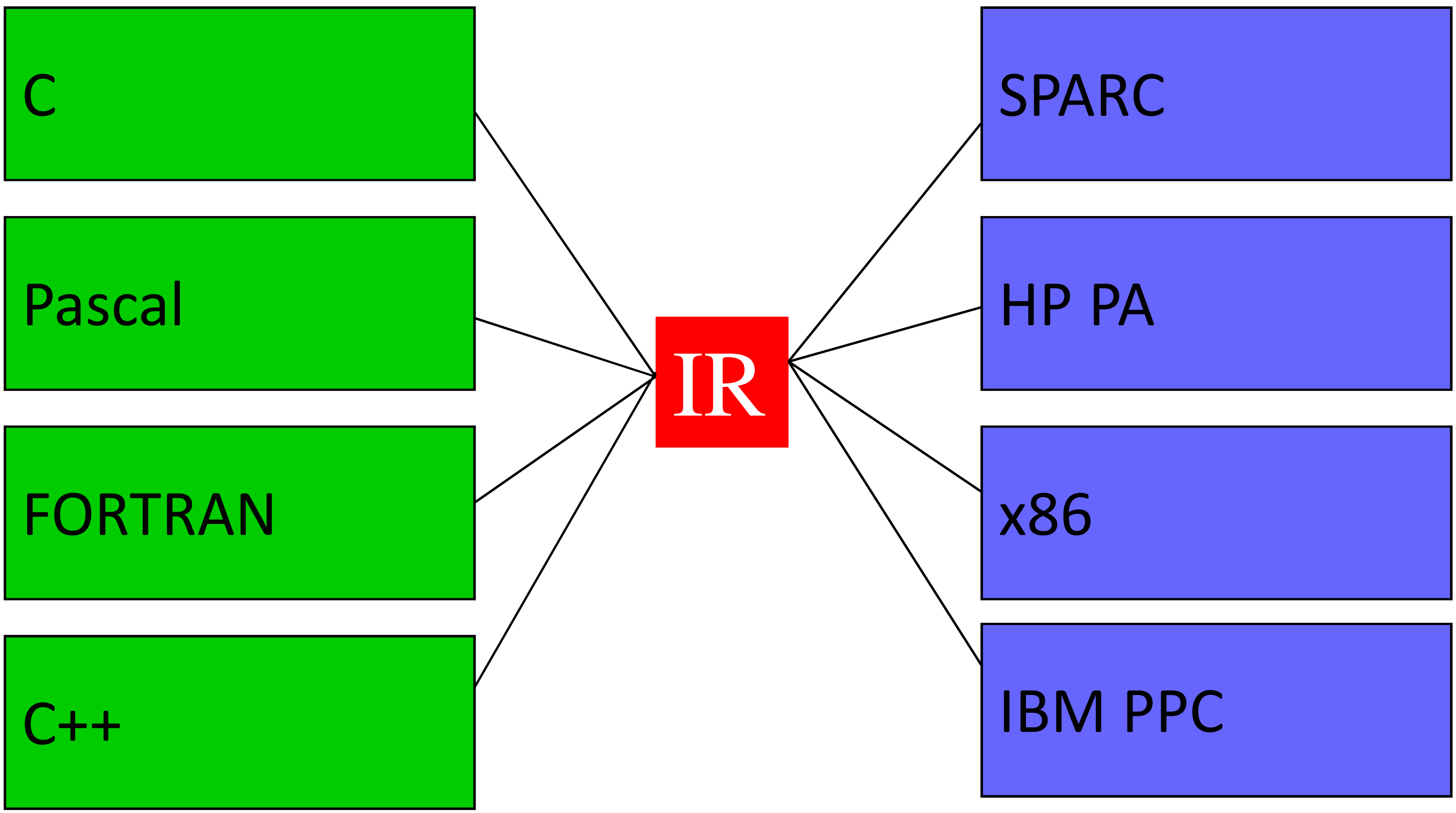
- Variants of Syntax Tree: DAG, ST
- Tree address codes
- Address and Instructions

Intermediate Representation (IR)

- A kind of abstract machine language
- that can express the target machine operations
- without committing to too much machine details.

Why IR ?

With IR



Intermediate Representations

- Intermediate representations span the gap between the source and target languages:
  - closer to target language;
  - (more or less) machine independent;
  - allows many optimizations to be done in a machine-independent way.
- Implementable via syntax directed translation, so can be folded into the parsing process.

Types of Intermediate Languages

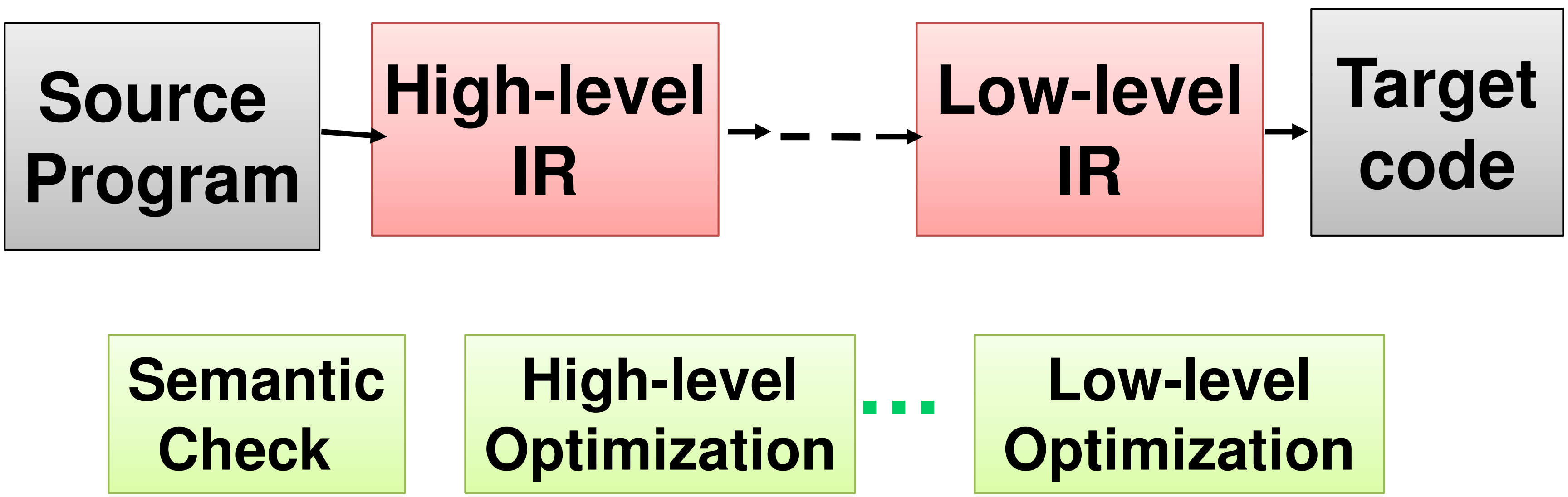
- <u>**High Level IR**</u> **(**e.g., AST):
  - closer to the source language
  - easy to generate from an input program
  - code optimizations may not be straightforward.
- Low Level IR (e.g., 3-address code, RTL):
  - closer to the target machine;
  - easier for optimizations, final code generation;

Advantages of Using an Intermediate Language

- ***Retargeting*** –
  – Build a compiler for a new machine
  – By attaching a new code generator to an existing front-end.
- ***Optimization*** –
  – Reuse intermediate code optimizers in compilers
  – for different languages and different machines.

Note: the terms "intermediate code", "intermediate language", and "intermediate representation" are all used interchangeably.

Multiple-Level IR



Using Multiple-level IR

- Translating from one level to another in the compilation process
  - Preserving an existing technology investment
  - Some representations may be more appropriate for a particular task.

Commonly Used IR

- Possible IR forms
  - Graphical representations: such as syntax trees, AST (Abstract Syntax Trees), DAG
  - Postfix notation
  - Three address code
  - SSA (Static Single Assignment) form
- IR should have individual components that describe simple things

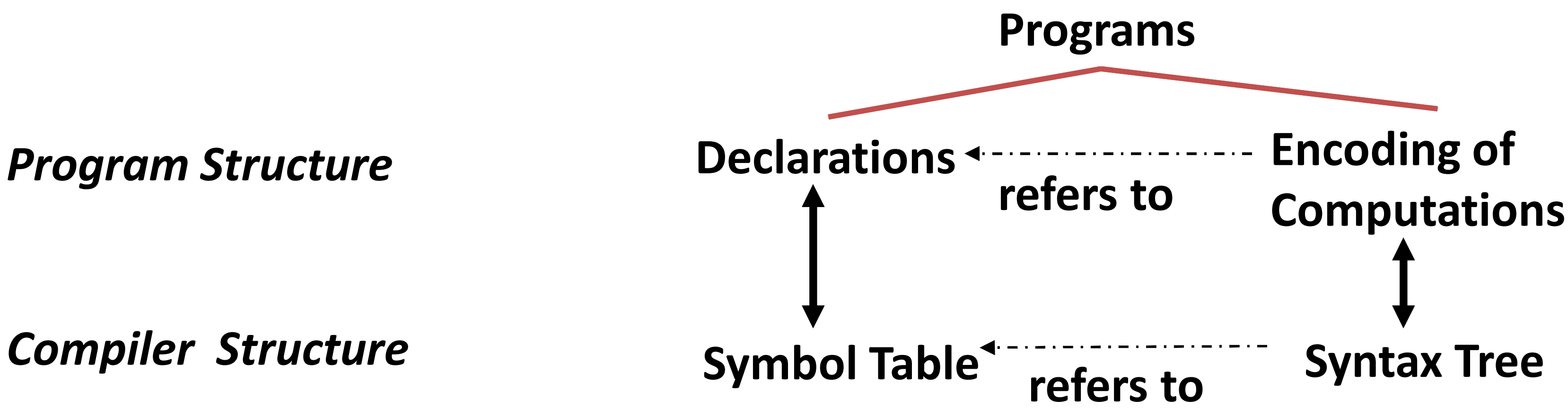
Intermediate Languages Types

- Graphical IRs:
  - Abstract Syntax trees,
  - DAGs,
  - Control Flow Graphs
- Linear IRs:
  - Stack based (postfix)
  - Three address code (quadruples)

Graphical IRs

- Abstract Syntax Trees (AST)
  - Retain essential structure of the parse tree,
  - Eliminating unneeded nodes
- Directed Acyclic Graphs (DAG)
  - Compacted AST to avoid duplication
  - Smaller footprint as well
- Control flow graphs (CFG)
  - Explicitly model control flow

Syntax Trees



A *syntax tree* shows the structure of a program by abstracting away irrelevant details from a parse tree.

– Each node represents a computation to be performed;

– The children of the node represents what that computation is performed on.

Syntax trees decouple parsing from subsequent processing.

Syntax Trees: Example

Grammar :

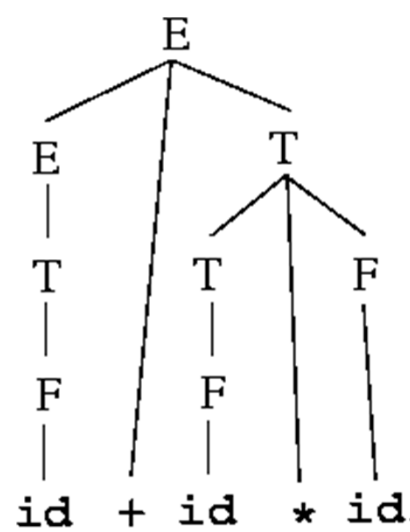
E → E + T | T

T → T * F | F

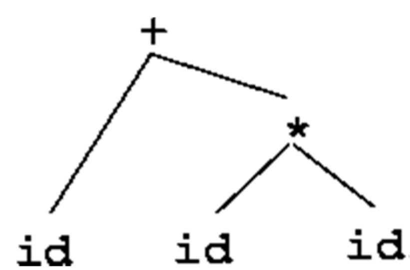
F → (E) | **id**

Input: **id + id * id**

Parse tree:

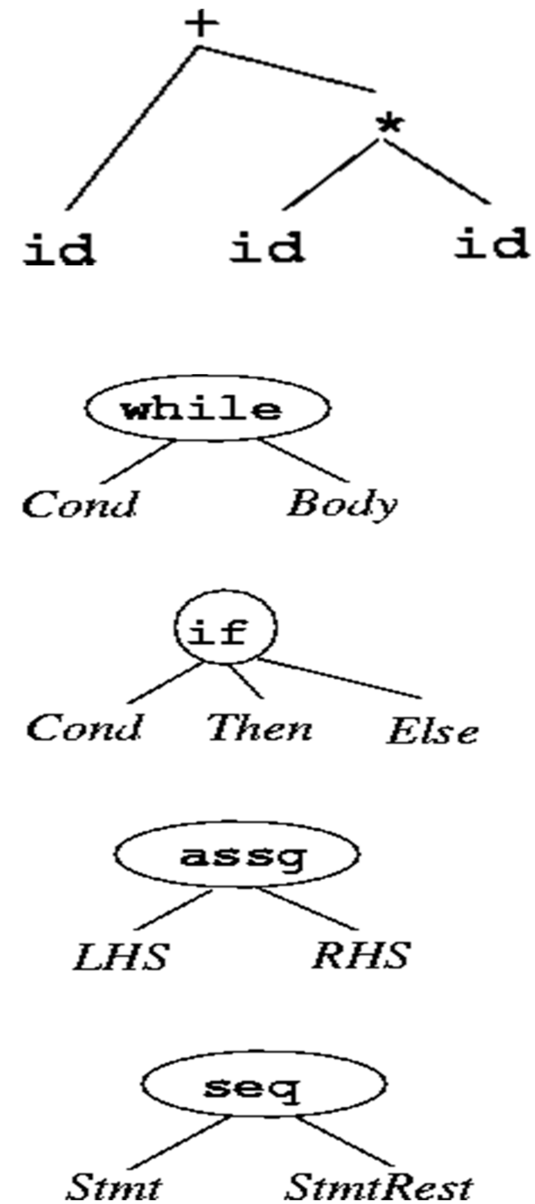


Syntax tree:



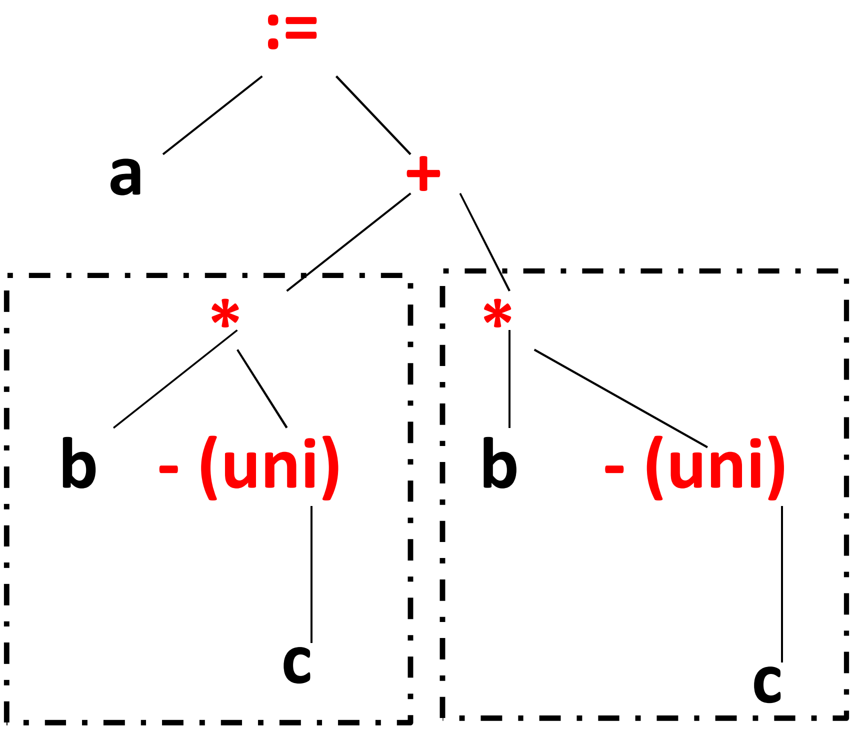
Syntax Trees: Structure

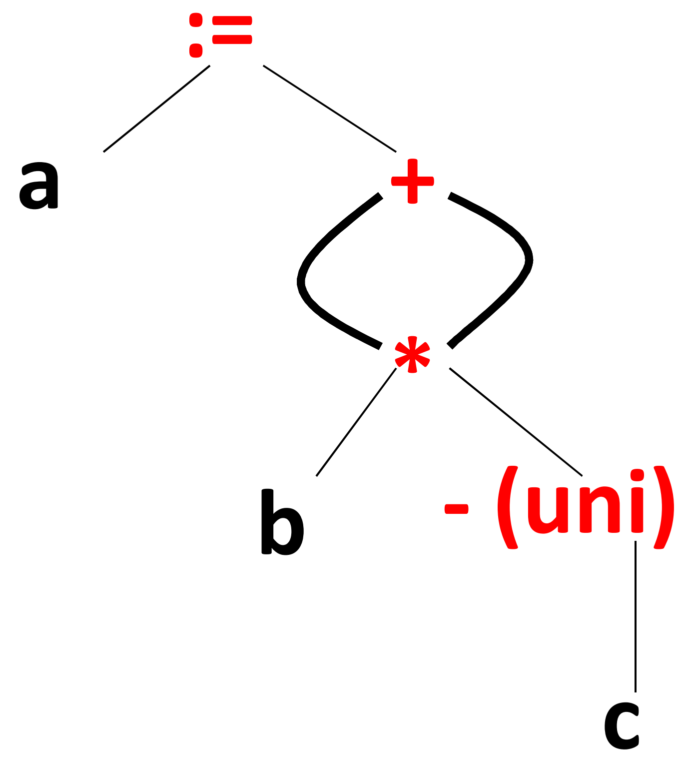
- Expressions:
  - leaves: identifiers or constants;
  - internal nodes are labeled with operators;
  - the children of a node are its operands.
- Statements:
  - a node's label indicates what kind of statement it is;
  - the children correspond to the components of the statement.



ASTs and DAGs:

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

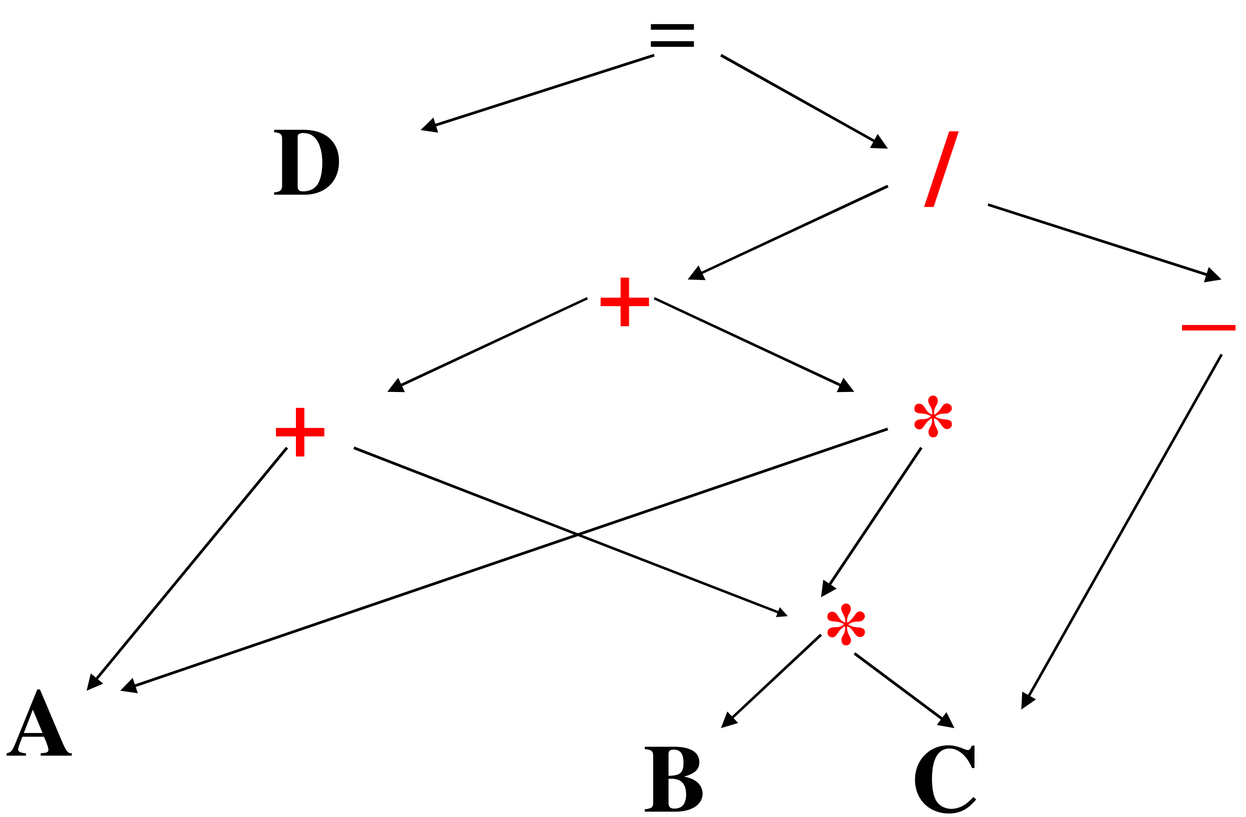




DAG Representation

A variant of syntax tree.

Example: D = ((A+B*C) + (A*B*C))/ -C



Linearized Intermediate Code-I

- Stack based (one address) – compact

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

Linearized Intermediate Code-II: Postfix Notation (PN)

- A mathematical notation wherein every operator follows all of its operands.
- Examples: PN of expression of

  9* (5+2) is 952+*

- How about (a+b)/(c-d) ?

  ab+cd-/

Postfix Notation (PN) – Cont'd

Form Rules:

1. If E is a variable/constant, the PN of E is E itself
2. If E is an expression of the form E_1 op E_2, the PN of E is $E_1'E_2'$op (E_1' and E_2' are the PN of E_1 and E_2, respectively.)
3. If E is a parenthesized expression of form (E_1), the PN of E is the same as the PN of E_1.

Linearized Intermediate Code-III: Three address Code

- Three address (quadruples) – **up to three operands, one operator**

```
t1 <- 2
t2 <- y
t3 <- t1 * t2
t4 <- x
t5 <- t4 - t1
```

Three-Address Statements

- A popular form of intermediate code used in optimizing compilers is three-address statements.
- Source statement:

$$\mathbf{x = a + b * c + d}$$

- Three address statements with temp t_1 and t_2:

$$t_1 = b * c$$

$$t_2 = a + t_1$$

$$x = t_2 + d$$

Three Address Code

- The general form **x := y op z**
  - x, y, and z are names, constants, compiler-generated temporaries
  - **op** stands for any operator such as +,-,…
- The expr: **x*5-y** might be translated as

  t1 := x * 5

  t2 := t1 - y

An Intermediate Instruction Set

- **Assignment:**
  - x = y <u>*op*</u> z // (<u>*op*</u> binary);
  - x = <u>*op*</u> y *//(<u>op</u> unary);*
  - x = y; *//copy instruction*
- **Jumps:**
  - if (x <u>*op*</u> y) goto L (L a label); *//Cond Jump*
  - goto L *//Uncond Jump*
- **Pointer and indexed assignments:**
  - x = y[z]   y[z] = x   x = &y
  - x = *y   *y = x.

An Intermediate Instruction Set

- **Procedure call/return:**
  - param x, k (x is the k^{th} param)
  - retval x
  - call p
  - enter p
  - leave p
  - return
  - retrieve x

```
param x1
param x2
…
param xn
call p, n  //p(x1,x2,..xn)
```

- **Type Conversion:**
  - x = cvt_A_to_B y (A, B base types) e.g.: cvt_int_to_float
- **Miscellaneous :**
  - label L

Three Address Code: Example

- **Source:**

```
if ( x + y*z > x*y + z)
    a = 0;
```

- **Three Address Code:**

```
tmp1 = y*z
tmp2 = x+tmp1                // x + y*z
tmp3 = x*y
tmp4 = tmp3+z                // x*y + z
if (tmp2 <= tmp4) goto L
a = 0
L:
```

Three Address Code: Example

```
do i=i+1; while (a[i] < v);
```

```
L:    t1 = i + 1
      i = t1
      t2 = i * 8
      t3 = a[t2]
      if t3 < v goto L
```

Symbolic labels

```
100:  t1 = i + 1
101:  i = t1
102:  t2 = i * 8
103:  t3 = a[t2]
104:  if t3 < v goto 100
```

Position numbers

Syntax-Directed Translation Into Three-Address

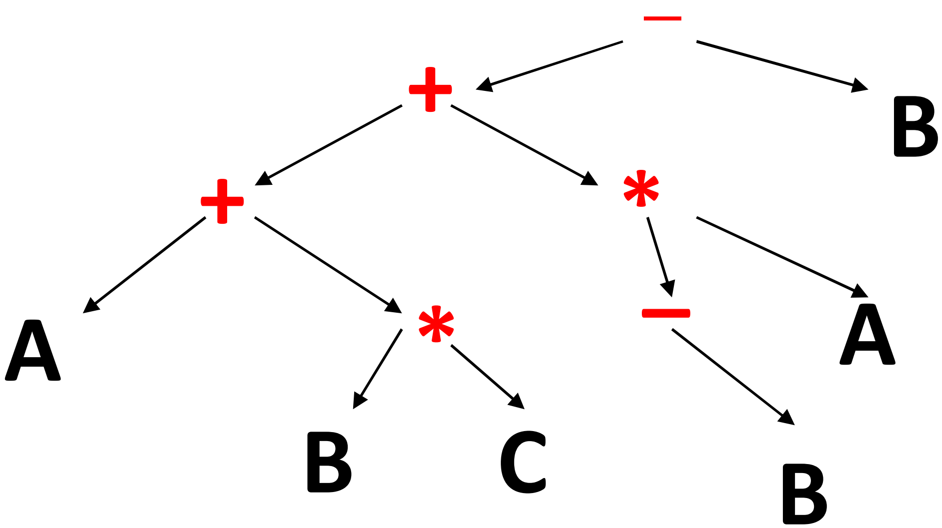
- **Temporary**
- In general, when generating three-address statements, the compiler has to create new temporary variables (temporaries) as needed.
- We use a function ***newtemp( )*** that returns a new temporary each time it is called

Syntax-Directed Translation Into Three-Address

- The syntax-directed definition for ***E*** in a production **id :=** ***E*** has two attributes:
  1. ***E.place*** - the location (variable name or offset) that holds the value corresponding to the nonterminal
  2. ***E.code*** - the sequence of three-address statements representing the code for the nonterminal

Syntax tree vs. Three address code

Expression: (A+B*C) + (-B*A) - B



T1 := B * C

T2 = A + T1

T3 = - B

T4 = T3 * A

T5 = T2 + T4

T6 = T5 – B

Three address code is a linearized representation of a syntax tree (or a DAG) in which explicit names (temporaries) correspond to the interior nodes of the graph.