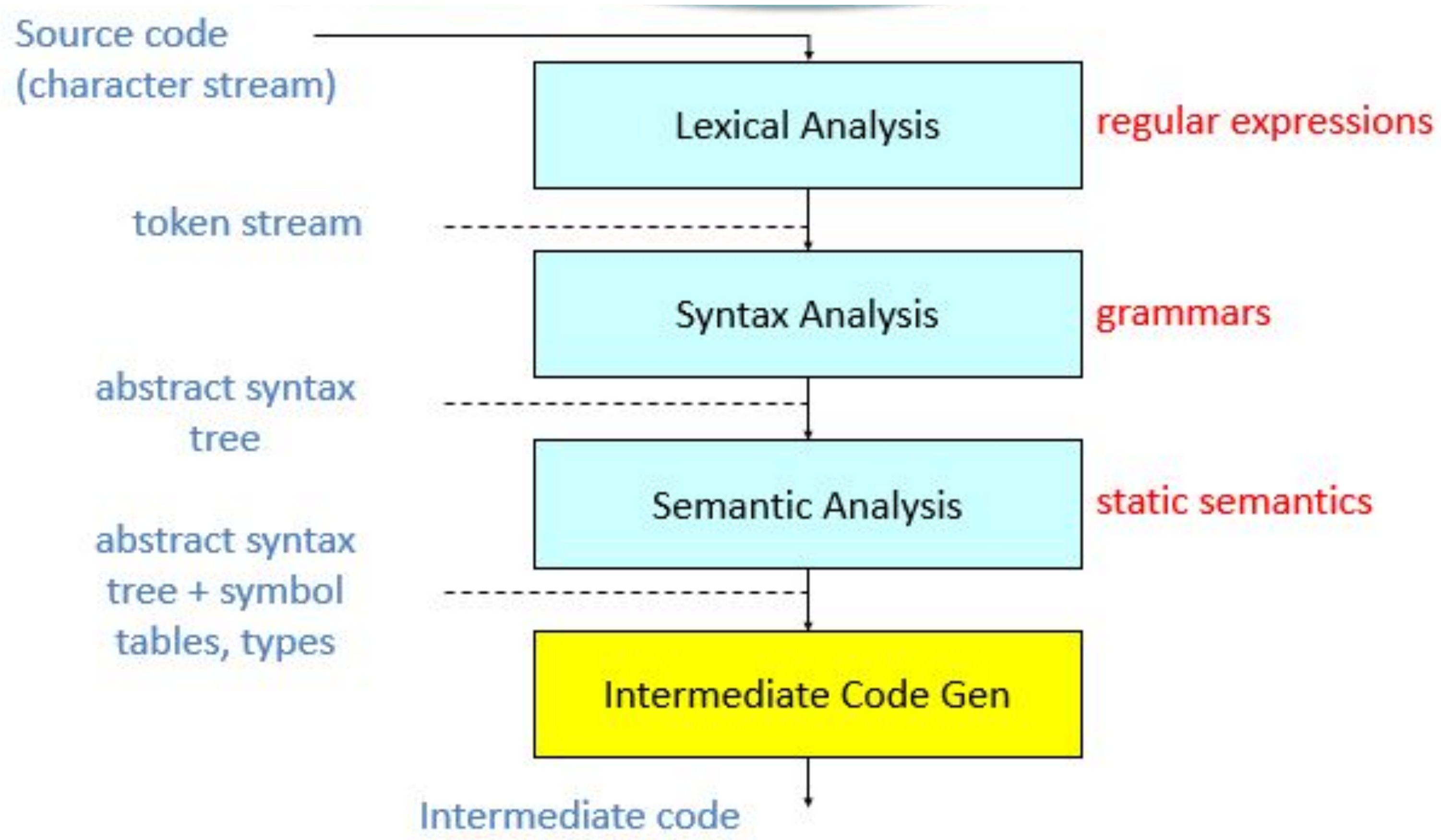
# Intermediate Representation (IR)

## TRANSLATION TO INTERMEDIATE CODE

# Where we are………



Source code
(character stream)
Lexical Analysis
regular expressions
token stream
Syntax Analysis
grammars
abstract syntax
tree
Semantic Analysis
static semantics
abstract syntax
tree + symbol
tables, types
Intermediate Code Gen
Intermediate code

# Translation to Intermediate code

**Intermediate Representation (IR):**

- An abstract machine language
- Expresses operations of target machine
- Not specific to any particular machine
- Independent of source language

**IR code generation not necessary:**

- Semantic analysis phase can generate real assembly code directly.
- Hinders portability and modularity.

# Translation to Intermediate code

Suppose we wish to build compilers for $n$ source languages and $m$ target machines.

**Case 1: no IR**

- Need separate compiler for each source language/target machine combination.
- A total of $n * m$ compilers necessary.
- Front-end becomes cluttered with machine specific details, back-end becomes cluttered with source language specific details.

**Case 2: IR present**

- Need just $n$ front-ends, $m$ back ends.

# Intermediate Representation

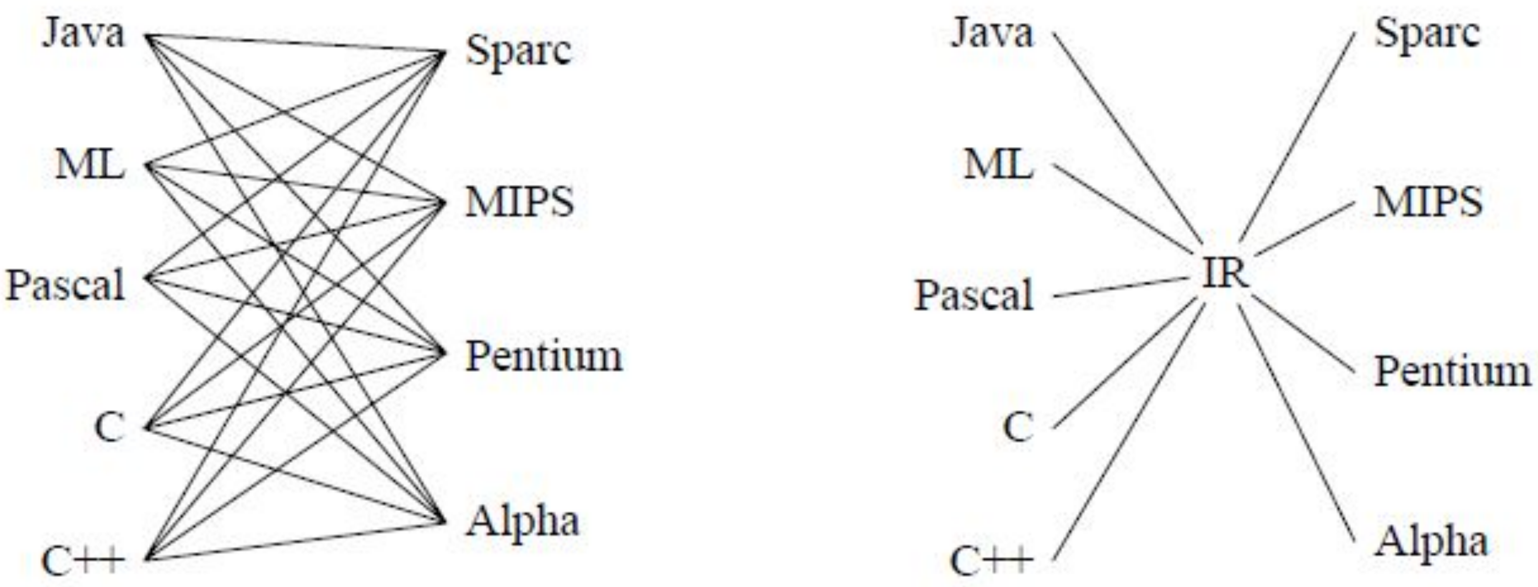



**FIGURE 7.1.** Compilers for five languages and four target machines: (left) without an IR, (right) with an IR.
From *Modern Compiler Implementation in ML*, Cambridge University Press, ©1998 Andrew W. Appel

# Intermediate Representation

Intermediate codes can be represented in a **variety of ways** and they have their own benefits.

**High Level IR**

- High-level intermediate code representation is very close to the source language itself.
- They can be easily generated from the source code and we can easily apply code modifications to enhance performance.
- But for **target machine optimization, it is less preferred.**

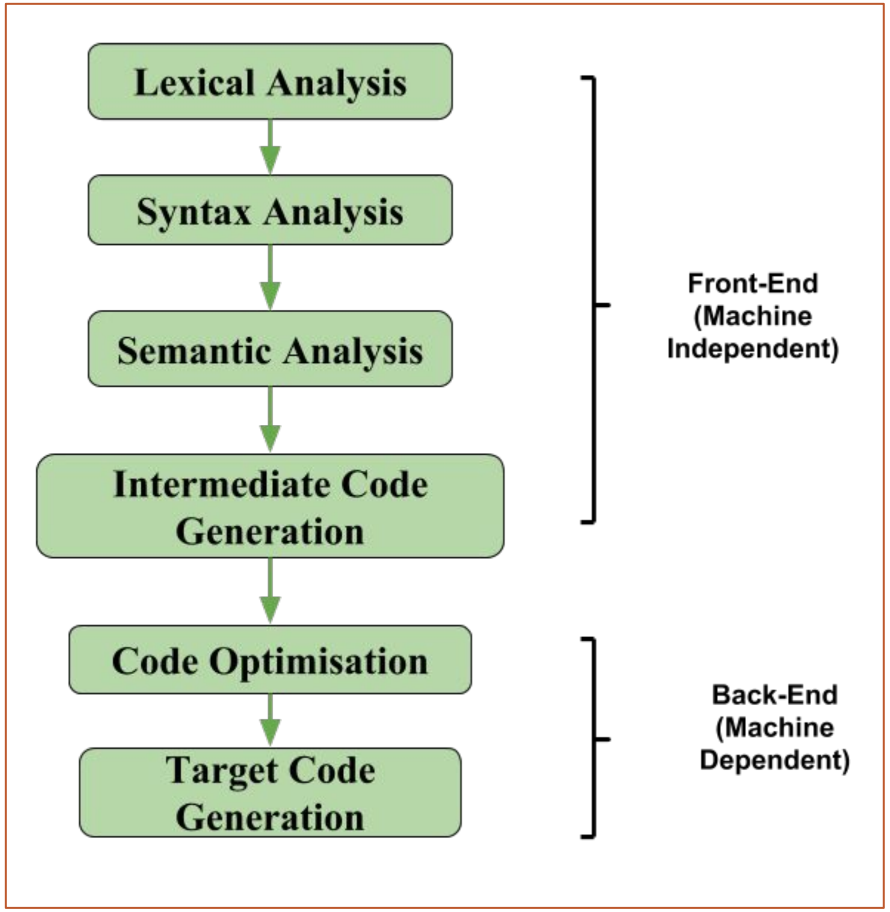
**Low Level IR**

- This one is close to the target machine, which makes it suitable for register and memory allocation, instruction set selection, etc. **It is good for machine-dependent optimizations**.

# Intermediate Representation

- ✔ If we generate machine code directly from source code, then for ***n target*** machine we will have ***n optimizers*** and ***n code generators*** but if we will have a machine independent intermediate code, we will have **only one optimizer**.

- ✔ **Intermediate code can be either language specific (e.g., Byte Code for Java) or language independent (three-address code).**



# Intermediate Representation Formats

## (1) Postfix Notation

- The ordinary (infix) way of writing the sum of a and b is with operator in the middle : a + b. The postfix notation for the same expression places the operator at the right end as **ab +**.
- In general, if e1 and e2 are any postfix expressions, and + is any binary operator, the result of applying + to the values denoted by e1 and e2 is postfix notation by **e1e2 +.**
- No parentheses are needed in postfix notation because the position and arity (number of arguments) of the operators permit only one way to decode a postfix expression. In postfix notation the operator follows the operand.
- **Example –** The postfix representation of the expression (a – b) * (c + d) + (a – b) is : ab – cd + *ab -+.

# Intermediate Representation Formats

## (2) Three-Address Code

- A type of intermediate code which is easy to generate and can be easily converted to machine code.
- It makes use of **at most three addresses and one operator** to represent an expression and the value computed at each instruction is stored in temporary variable generated by compiler.
- The compiler decides the order of operation given by three address code.

**General representation –** $\boxed{a = b \text{ op } c}$

Where a, b or c represents **operands like names, constants or compiler generated temporaries** and **op represents the operator**

# Intermediate Representation Formats

## (2) Three-Address Code

Example:

- *a = b + c * d*
- The intermediate code generator will try to **divide this expression into sub-expressions** and then generate the corresponding code

  *t1 = c * d;*

  *t2 = b + t1;*

  *a = t2*

  t1,t2 are temporary variables

✔ A three-address code has **at most three address locations to calculate the expression**.

✔ A three-address code can be represented in two forms : quadruples and triples.

# Intermediate Representation Formats

## Implementation of Three-Address Code

### Quadruples

Each instruction in quadruples presentation is divided into four fields: operator, arg1, arg2, and result. The above example is represented below in quadruples format:

**Advantage –**
Easy to rearrange code for global optimization.
One can quickly access value of temporary variables using symbol table.
**Disadvantage –**
Contain lot of temporaries.
Temporary variable creation increases time and space complexity.

| Op | $arg_1$ | $arg_2$ | result |
|---|---|---|---|
| * | c | d | r1 |
| + | b | r1 | r2 |
| + | r2 | r1 | r3 |
| = | r3 | | a |

# Intermediate Representation Formats

## Implementation of Three-Address Code

**Triples**

Each instruction in triples presentation has three fields : op, arg1, and arg2. The results of respective sub-expressions are denoted by the position of expression. Triples represent similarity with DAG and syntax tree. They are equivalent to DAG while representing expressions.

| Op | $arg_1$ | $arg_2$ |
|---|---|---|
| * | c | d |
| + | b | (0) |
| + | (1) | (0) |
| = | (2) | |

# Intermediate Representation Formats

## Implementation of Three-Address Code

**<u>Triples</u>**

**Advantage –**

This representation doesn't make use of extra temporary variable to represent a single operation instead when a reference to another triple's value is needed, a pointer to that triple is used

**Disadvantage –**

- Temporaries are implicit and difficult to rearrange code.
- It is difficult to optimize because optimization involves moving intermediate code. When a triple is moved, any other triple referring to it must be updated also. With help of pointer one can directly access symbol table entry.

# Intermediate Representation Formats

**Implementation of Three-Address Code**

**<u>Indirect Triples</u>**

- This representation is an enhancement over triples representation.
- It uses **pointers instead of position** to store results.
- **This enables the optimizers to freely re-position the sub-expression to produce an optimized code.**

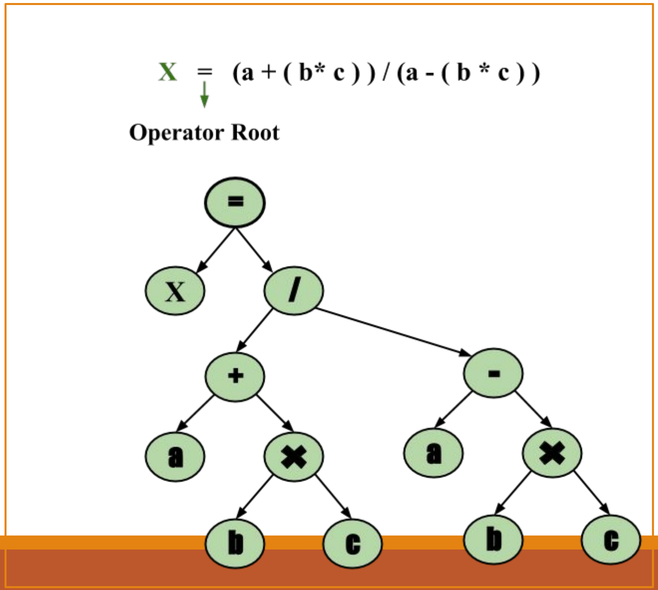
# Intermediate Representation Formats

## (3) Syntax Tree –

- Syntax tree is nothing more than condensed form of a parse tree.
- **The operator and keyword nodes of the parse tree are moved to their parents** and a chain of single productions is replaced by single link in syntax tree. The **internal nodes are operators and child nodes are operands.**
- To form syntax tree put parentheses in the expression, this way it's easy to recognize which operand should come first.

## Example –

x = (a + b * c) / (a – b * c)



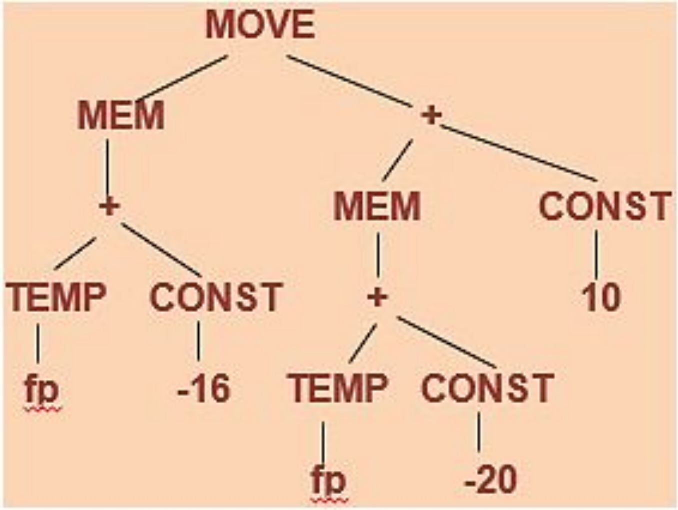
# Good IR properties

A good intermediate representation . . .

- must be convenient to produce from an AST,
- must be convenient to translate into real, machine code (for all desired targets)
- and must have clear and simple meaning, so that optimizations (which rewrite the IR) are easy to specify and implement.

For the IR in our course, we will use a simple ***tree language***.

# IR Expression Tree




**Represents the IR:**

MOVE(MEM(+(TEMP(fp),CONST(-16))),

+(MEM(+(TEMP(fp),CONST(-20))), CONST(10)))

**which evaluates the program:**

- **M[fp-16] := M[fp-20]+10**

# IR Expressions

*CONST(i):* the integer constant i

***MEM(e):* if e is an expression that calculates a memory address, then this is the contents of the memory at address e (one word)

*NAME(n):* the address that corresponds to the label n

- eg. MEM(NAME(x)) returns the value stored at the location X

*TEMP(t):* if t is a temporary register, return the value of the register,

- eg. MEM(BINOP(PLUS,TEMP(fp),CONST(24)))

fetches a word from the stack located 24 bytes above the frame pointer

** Note that when MEM is used as the left child of a MOVE, it means “store”, but anywhere it means else it means “fetch”

# IR Expressions

*BINOP(op,e1,e2):* evaluate e1, evaluate e2, and perform the binary operation op over the results of the evaluations of e1 and e2

- op can be PLUS, AND, etc
- we abbreviate BINOP(PLUS,e1,e2) by +(e1,e2)

CALL(f,[e1,e2,...,en]): evaluate the expressions e1, e2, etc (in that order), and at the end call the function f over these n parameters

- eg. CALL(NAME(g),ExpList(MEM(NAME(a)),ExpList(CONST(1),NULL)))

  represents the function call g(a,1)

ESEQ(s,e): execute statement s and then evaluate and return the value of the expression e

# IR STATEMENTS

**MOVE(TEMP(t),e)**: store the value of the expression e into the register t

**MOVE(MEM(e1),e2)**: evaluate e1 to get an address, then evaluate e2, and then store the value of e2 in the address calculated from e1

- eg, MOVE(MEM(+(NAME(x),CONST(16))),CONST(1))

  computes x[4] := 1 (since 4*4 bytes = 16 bytes).

**EXP(e)**: evaluate e and discard the result

**JUMP(L)**: Jump to the address L

- L must be defined in the program by some LABEL(L)

# IR STATEMENTS

*CJUMP(o,e1,e2,t,f):* evaluate e1 & e2. If the values of e1 and e2 are related by o, then jump to the address calculated by t, else jump the one for f

- the binary relational operator *o* must be EQ, NE, LT etc

*SEQ(s1,s2,...,sn):* perform statement s1, s2, ... sn is sequence

LABEL(n): define the name n to be the address of this statement

- you can retrieve this address using NAME(n)

EQ=Equal
NE = Not Equal
LT=Less Than
GT=Greater Than

# ARRAY ACCESSES

Given array variable a,

```
&(a[0]) = a
&(a[1]) = a + w, where w is the word-size of machine
&(a[2]) = a + (2 * w)
...
```

Contents of array **a** at $i^{th}$ index = **MEM(+(NAME(a), CONST(i*w)))**

**MEM(BINOP(PLUS, NAME(a), CONST(i*w)))**

# RECORDS

For records, we need to know the byte offset of each field (record attribute) in the base record

Since every value is 4 bytes long, the ith field of a structure *a* can be retrieved using **MEM(+(A,CONST(i*4)))**, where A is the address of a

- here i is always a constant since we know the field name

# Example

**for i := 0 to 9 do V[i] := 5**

Assume, every value in the array V requires 4 bytes

```
SEQ(MOVE(i, CONST(0)),
    CJUMP(GT, i, CONST(9), done, loop),
      LABEL(loop),
      MOVE(MEM(+(Name(V), CONST(i*4))),const(5)),
      MOVE(i,+(i,CONST(1))),
      CJUMP(GT, i, CONST(9), done, loop),
      LABEL(done))
```

# Example

If( i < 10) then y := 0 else i:=i-1

```
SEQ( CJUMP(LT, i, CONST(10), trueL,
falseL),
    LABEL(trueL),
    MOVE(y, CONST(0)),
    JUMP(exit),
    LABEL(falseL),
    MOVE(i, - (i,CONST(1))),
    JUMP(exit),
    LABEL(exit))
```

# The End