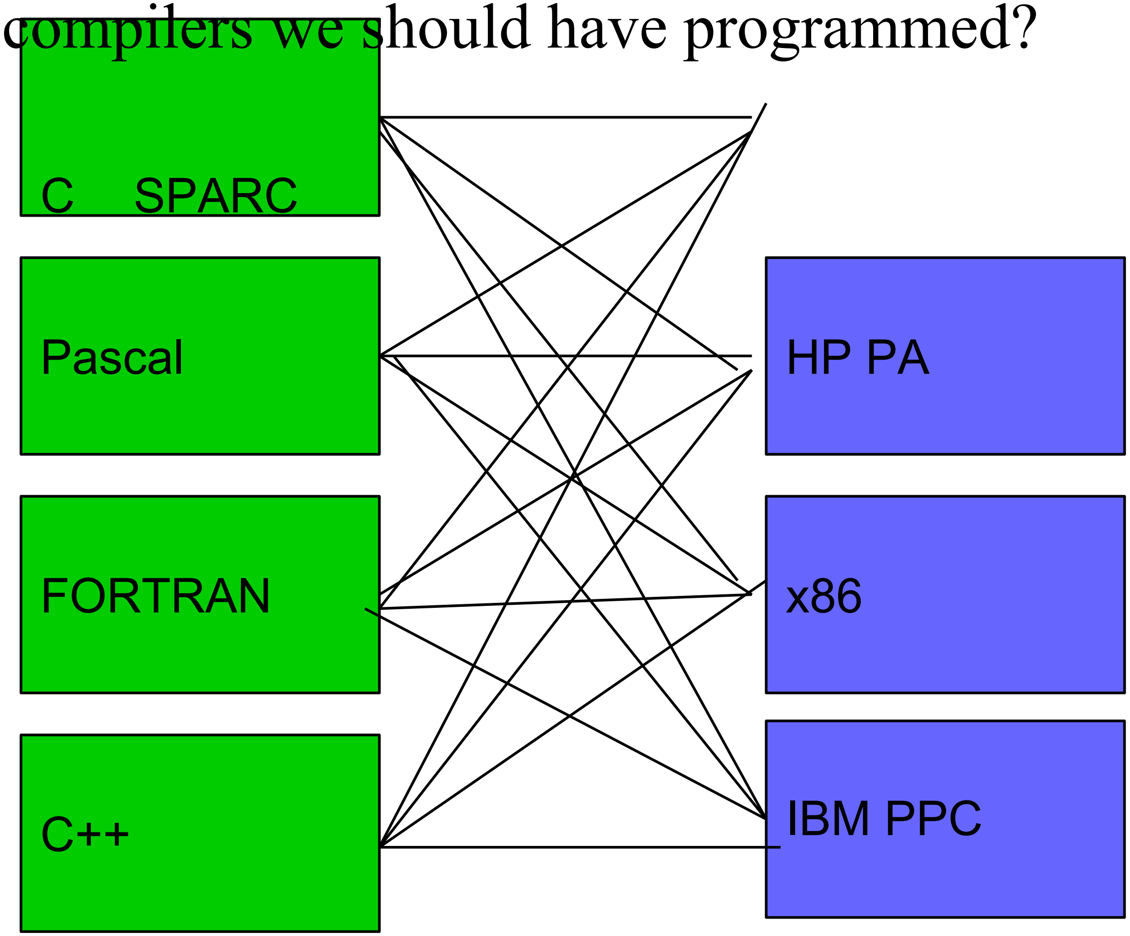
Intermediate Representations

Overview

- Need for Intermediate Representation
- Types of Intermediate Representations
  - Abstract syntax tree
  - Directed acyclic graph
  - Three Address code
- Logical structure of compiler front end

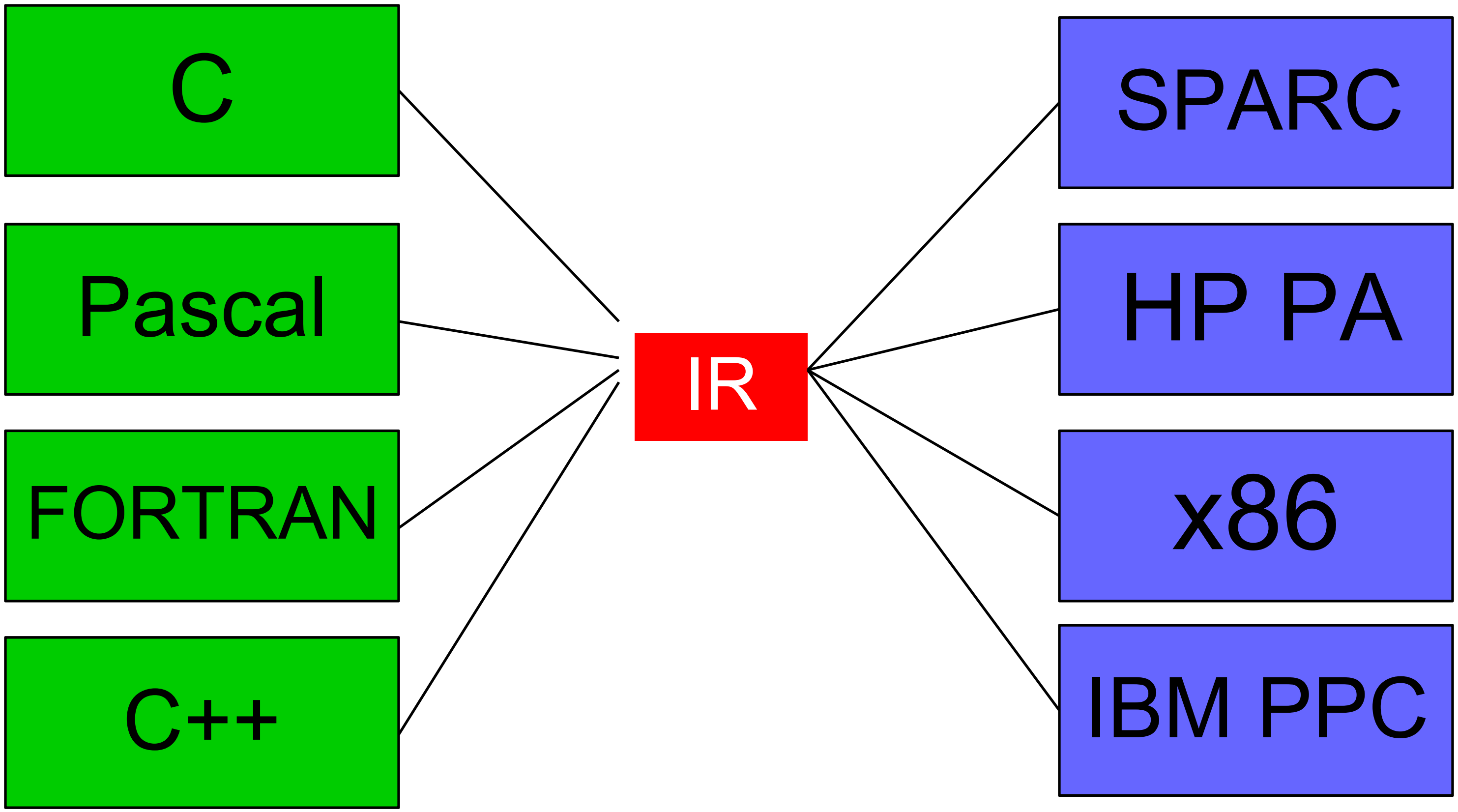
WHY INTERMEDIATE REPRESENTATION?

- For k languages and n target architectures how many compilers we should have programmed?



k*n

Why Intermediate Representation?



k+n

Why Intermediate Representation?

- Intermediate Representation
  - Machine and Language independent version of original source code.
- Use of intermediate representation provides
  - Increased abstraction
  - Cleaner separation between the front end and back end
  - Adds possibilities for re-targeting/cross-compilation.
  - Supports compiler optimizations.

Types of Intermediate Representations

- Two kinds of intermediate representations are
  - Trees, includes parse trees and (abstract) syntax trees
  - Linear representation, three-address code
- Intermediate representations are categorized according to where they fall between a high-level language and machine code.
  - IRs close to high-level language are called high-level IRs
  - IRs close to assembly are called low-level IRs

Original
```
float a[10][20];
a[i][j+2];
```

High IR
```
t1 = a[i, j+2]
```

Mid IR
```
t1 = j + 2
t2 = i * 20
t3 = t1 + t2
t4 = 4 * t3
t5 = addr a
t6 = t5 + t4
t7 = *t6
```

Low IR
```
r1 = [fp - 4]
r2 = [r1 + 2]
r3 = [fp - 8]
r4 = r3 * 20
r5 = r4 + r2
r6 = 4 * r5
r7 = fp - 216
f1 = [r7 + r6]
```

TYPES OF INTERMEDIATE REPRESENTATIONS

- High-level IRs usually preserve information like
  - Loop-structure, array subscripts, if-then-else statements etc.
  - They tend to reflect the source language they are compiling more than low-level IRs.
- Medium-level IRs are independent of both the source language and the target machine.
- Low-level IRs
  - Converts the array subscripts into explicit addresses and offsets.
  - Tend to reflect the target architecture very closely, often machine-dependent.
- Different optimizations are possible at different IR levels.

ABSTRACT SYNTAX TREE (AST)

- An **abstract syntax tree** is a tree representation of the source code used as an IR where
  - Intermediate nodes may be collapsed
  - Grouping units can be dispensed with etc,
- Each node represents a piece of the program structure and the node will have references to its children subtrees.
- Grouping is done is such a way that the structure obtained drives the semantic processing and code generation.

EXAMPLE

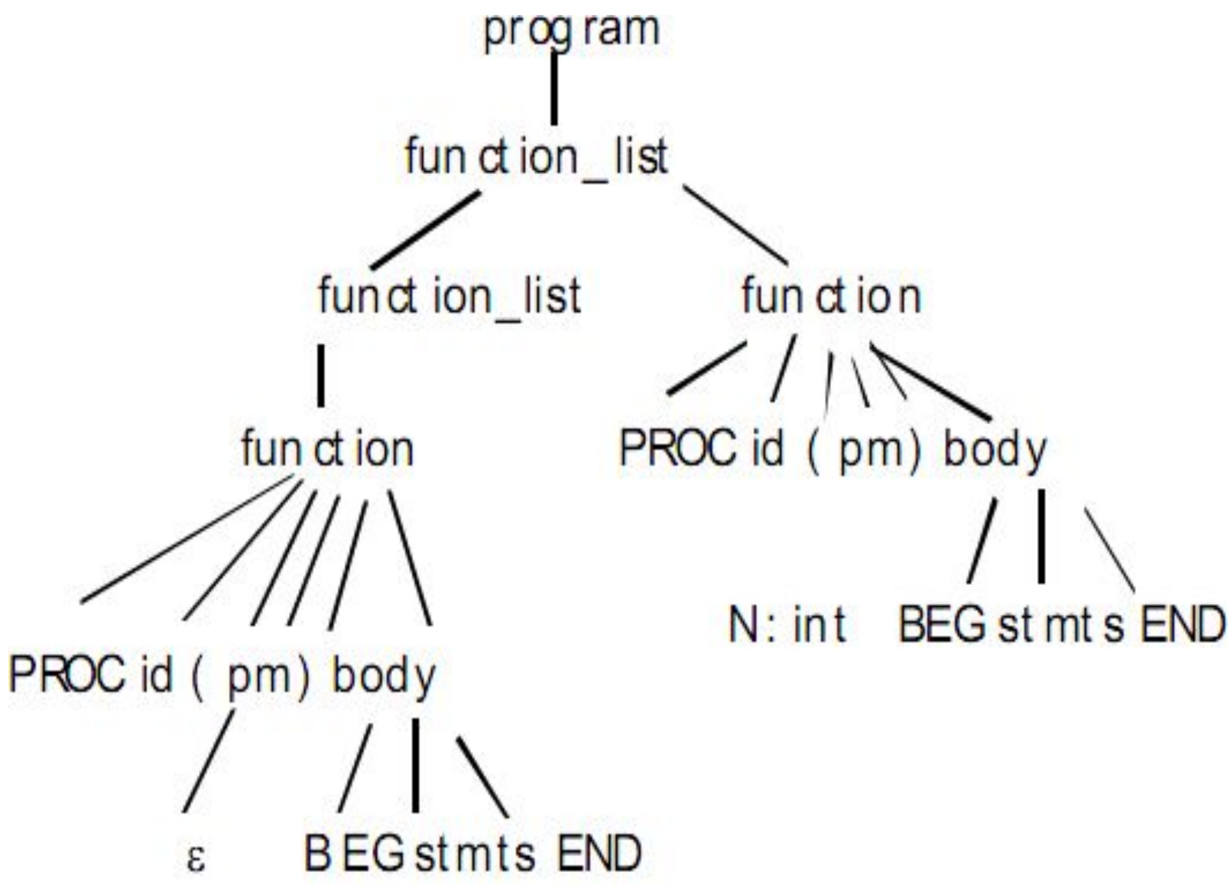
Consider the following excerpt of a programming language grammar:

```
program          ->   function_list
function_list    ->   function_list function | function
function         ->   PROCEDURE ident ( params ) body
params           ->   ...
```

A sample program for this language:

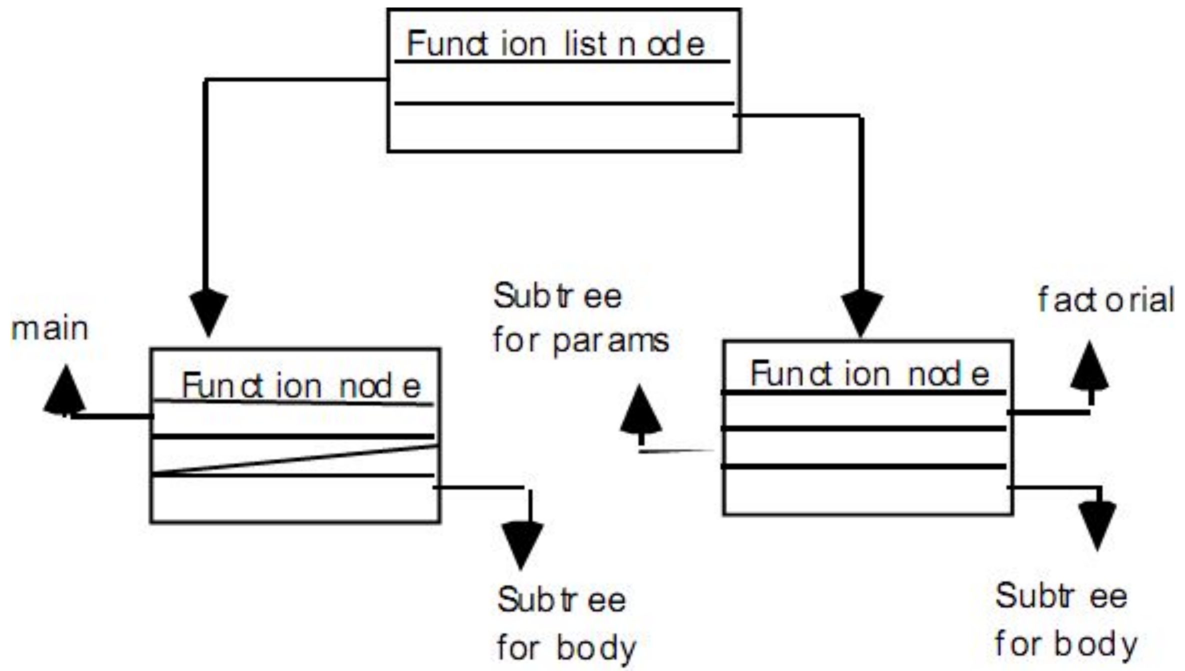
```
PROCEDURE main()
BEGIN
   statement...
END

PROCEDURE factorial(n:INTEGER)
BEGIN
   statement...
END
```

EXAMPLE



parse tree

EXAMPLE



abstract syntax tree

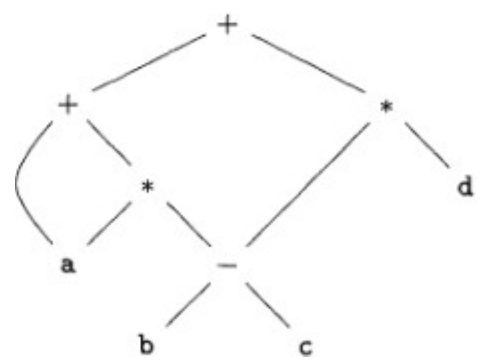
Directed Acyclic Graph

- In a syntax tree, there is only one path from root to each leaf of a tree.
  - When using trees as intermediate representations, it is often the case that some subtrees are duplicated.
  - A logical optimization is to share the common sub-tree.
- A syntax tree with more than one path from start symbol to terminals is called **Directed Acyclic Graph (DAG)**.
  - They are hard to construct internally, but provide an obvious savings in space.
  - They also highlight equivalent sections of code which will be used in optimizations like computing the needed result once and saving it, rather than re-generating it several times.

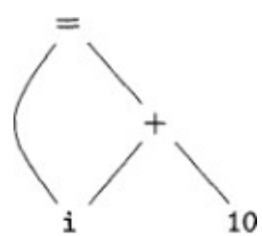
EXAMPLE

- The DAG for the expression

  a + a * (b – c) + (b-c) * d is:



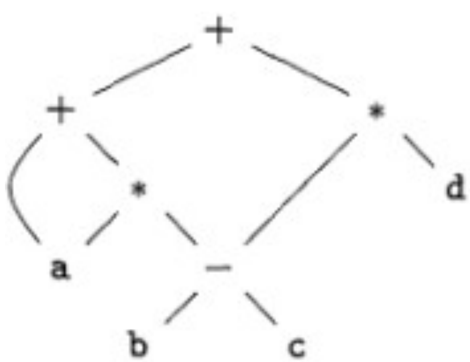
- The DAG for the expression i = i + 10 is:



Three Address Code (TAC)

- A three address code is a linearized representation of a syntax tree or a DAG in which explicit names correspond to the interior nodes of the graph.
- An address can be one of the following:
  - A name
  - A constant
  - A compiler-generated temporary
- An expression like X + Y * Z in three address instructions is
  - t1 = y * z
  - t2 = x + t1

EXAMPLE



(a) DAG

$t_1 = b - c$

$t_2 = a * t_1$

$t_3 = a + t_2$

$t_4 = t_1 * d$

$t_5 = t_3 + t_4$

(b) Three-address code

A DAG and its corresponding three-address code

Three-Address Codes

Instructions	Three Address Code
Assignment instruction	*x := y op z*
Assignment instruction	*x := op y*
Copy instruction	*x := y*
Unconditional jump	*goto L*
Conditional jump	*if x relop y goto L*
Procedural call	*param x1* *param x2* ..... *param xn* *call p, n*
Indexed Instruction	*x := y[i]* *x[i] := y*
Address and pointer Instruction	*x := &y* *x := *y* **x := y*

Example 1

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

⇩

t1 := -c
t2 := b * t1
t3 := -d
t4 := b * t3
t5 := t2 + t4
a := t5

Example 2

```
if (c == 0) {
  while (c < 20) {
  c = c + 2;
  }
}
else
  c = n * n + 2;
```

```
(1)   if c == 0 goto 3
(2)   goto 7
(3)   if c < 20 goto 5
(4)   goto 9
(5) c = c + 2
(6)   goto 3
(7)t2 = n * n
(8) c = t2 + 2
(9)
```

```
(1)   if c == 0 goto 5
(2)   t2 = n * n
(3) c = t2 + 2
(4)   goto 8
(5)   if c >= 20 goto 8
(6) c = c + 2
(7) goto 5
(8)
```

EXAMPLE 3

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

```
(1) t1 = i + 1
(2) i = t1
(3) t2 = i * 8
(4) t3 = a[t2]
(5) If t3 < v goto 1
(6)
```

Implementation of Three Address Code

- TAC specifies the components of each type of instruction.
- These instructions are implemented in a data structure as objects with fields for their operators and operands.
- Commonly used data structures are
  - Quadruples
  - Triples
  - Indirect Triples

QUADRUPLES

- A quadruple has four fields, *op*, *arg1*, *arg2* and *result*.
- The three address instruction $x := y + z$ is represented by placing
  - \+ in *op*
  - **y** in *arg1*
  - **z** in *arg2*
  - **x** in *result*

Type of operation	Three address code	Quadruple	Example
Binary Operation	result := y op z	op y, z, result	**add a,b,c**
Unary Operation	result := op y	op y, , result	**uminus a, , c** **not a, ,c** **inttoreal a, , c**
Assignment Operation	result := y	mov y, , result	**mov a, , c**
Unconditional Jump	goto L	jmp , , L	**jmp , , L1**
Conditional Jump	if y ***relop*** z goto L	jmp***relop*** y, z, L	**jmpgt y, z, L1**
Procedure call	param x1 …. param xn call p, n	param x1, , …. param xn, , call p, n,	**// calls f(x+1,y)** **add x,1,t** **param t1,,** **param y,,** **call f,2,**
Indexed Operation	x := y[i] y[i] := x	move y[i], , x move x, ,y[i]	
Address and Pointer Operation	x := &y x := *y	moveaddr y, , x movecont y, , x	

EXAMPLE

	Operator	Arg1	Arg2	Result	Code
0	uminus	c		t1	t1=-c
1	sub	b	t1	t2	t2=b*t1
2	uminus	c		t3	t3=-c
3	mul	b	t3	t4	t4=b*t3
4	add	t2	t4	t5	t5=t2+t4
5	mov	t5		a	a=t5

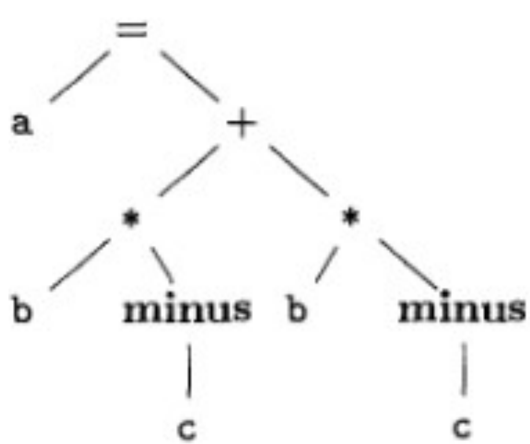
EXAMPLE

```
x:=1;
y:=x+10;
while (x<y) {
    x:=x+1;
    if (x%2==1) then
        y:=y+1;
    else y:=y-2;
}
```

01:	mov	1,,x	09:	eq	t4,1,t5
02:	add	x,10,t1	10:	jmpf	t5,,14
03:	mov	t1,,y	11:	add	y,1,t6
04:	lt	x,y,t2	12:	mov	t6,,y
05:	jmpf	t2,,17	13:	jmp	,,16
06:	add	x,1,t3	14:	sub	y,2,t7
07:	mov	t3,,x	15:	mov	t7,,y
08:	mod	x,2,t4	16:	jmp	,,4
			17:		

TRIPLES

- A triple has only three fields, *op*, *arg1* and *arg2.*
  - Instead of a result field we can use pointers to the triple structure itself
- The DAG and triple representations are equivalent.



(a) Syntax tree

	op	arg_1	arg_2
0	minus	c	
1	*	b	(0)
2	minus	c	
3	*	b	(2)
4	+	(1)	(3)
5	=	a	(4)
		...	

(b) Triples

Representations of $a + a * (b - c) + (b - c) * d$

INDIRECT TRIPLES

- Indirect triples consist of a listing of pointers to triples, rather than listing of triples themselves.
- Useful for an optimizing compiler
  - Can move instructions by reordering the *instruction* list, without affecting the triples themselves.

	instruction
35	(0)
36	(1)
37	(2)
38	(3)
39	(4)
40	(5)
	. . .

	op	arg_1	arg_2
0	minus	c	
1	*	b	(0)
2	minus	c	
3	*	b	(2)
4	+	(1)	(3)
5	=	a	(4)
		. . .	

Indirect triples representation of three-address code

COMPARISON

- Quadruples
  - direct access of the location for temporaries
  - easier for optimization
- Triples
  - space efficiency
- Indirect Triples
  - easier for optimization
  - space efficiency

LOGICAL STRUCTURE OF COMPILER FRONT END

- Front end analyzes the source code and creates an intermediate representation.
  - Details of the source language are confined to front end.
- Static checker does *type checking*
  - Ensures that operators are applied to compatible operands.
- Back end synthesizes target code.
  - Details of target machine are confined to back end.