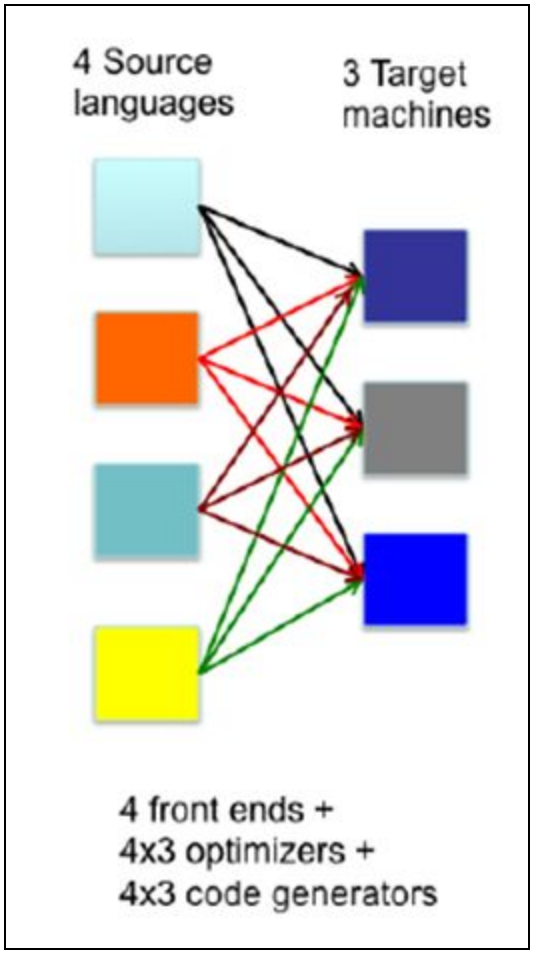
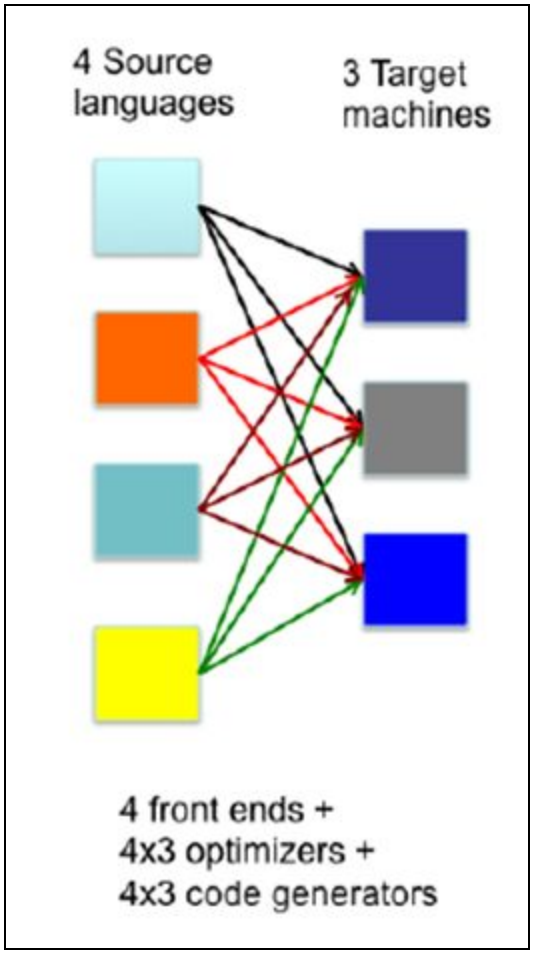
Intermediate Code Generation

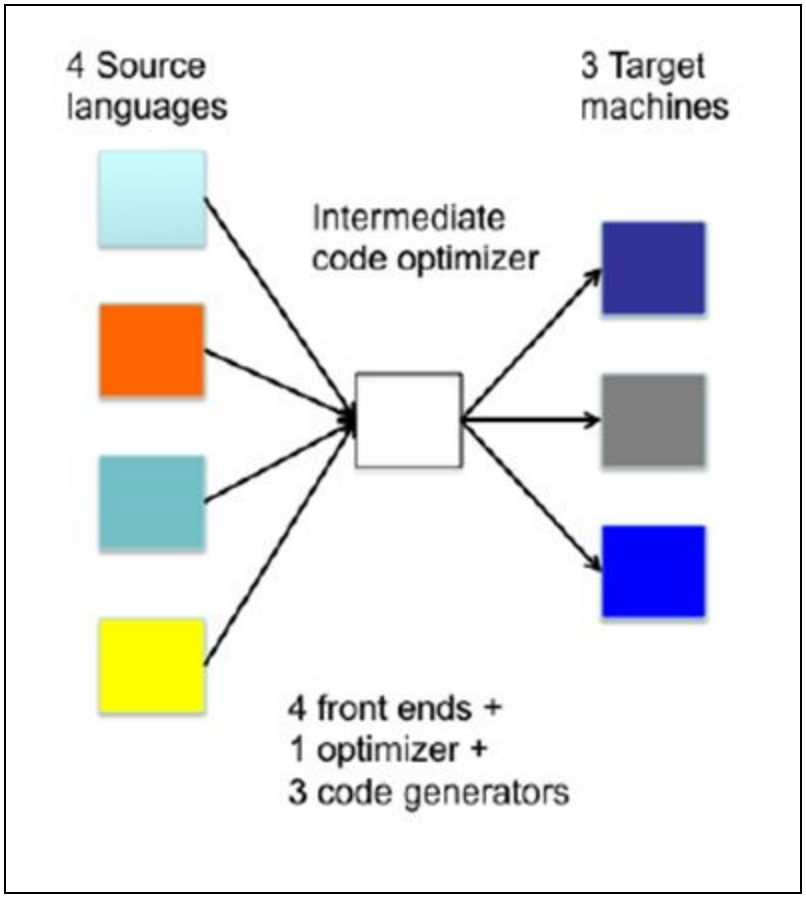
Why Intermediate Code?

Why Intermediate Code?



Why Intermediate Code?





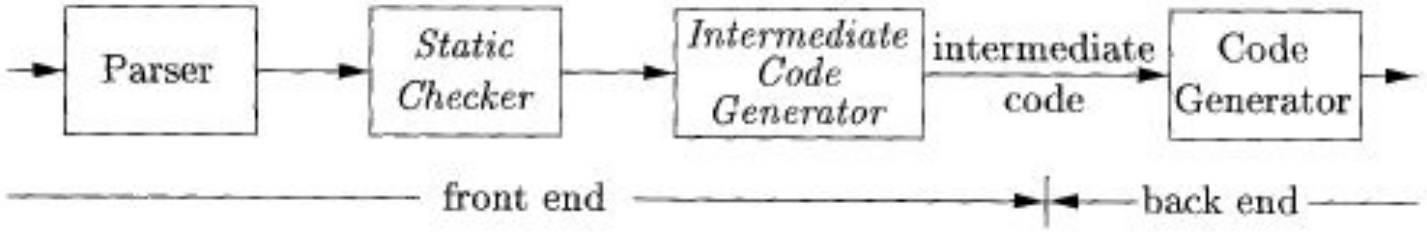


Figure 6.1: Logical structure of a compiler front end

- Instead of $m \times n$ compilers we can have m front ends and n backends.
  - Each front end gives the same intermediate representation and back end is going use this.
- C was used as an inter. representation for designing C++.
  - For many machines, C compilers are existing!

Other advantages …

- Machine independent code optimization.
  - A separate field of study.

Other advantages …

- Machine independent code optimization.
  - A separate field of study.
- We can assume that there is an virtual machine which runs the intermediate code.
  - Interpreters for various machines can be written.
  - Portability increased.
  - Java. JVM.

Different Types of Intermediate Code

- Intermediate code must be easy to produce and easy to translate to machine code
  - A sort of universal assembly language
  - Should not contain any machine-specific parameters (registers, addresses, etc.)
- The type of intermediate code deployed is based on the application
- Quadruples, triples, indirect triples, abstract syntax trees are the classical forms used for machine-independent optimizations and machine code generation
- Static Single Assignment form (SSA) is a recent form and enables more effective optimizations
  - Conditional constant propagation and global value numbering are more effective on SSA
- Program Dependence Graph (PDG) is useful in automatic parallelization, instruction scheduling, and software pipelining

Static Checking -- SDD

- Static checking and intermediate code generation can be done with the help of SDTs.
- Static checking
  - Type mismatch checking of operands of an operator.
  - Ensuring that break; stmt should occur within a loop or switch ---- anywhere else it is an error.
  - continue; should occur within a loop.

Various intermediate codes

Various intermediate codes

Figure 6.2: A compiler might use a sequence of intermediate representations

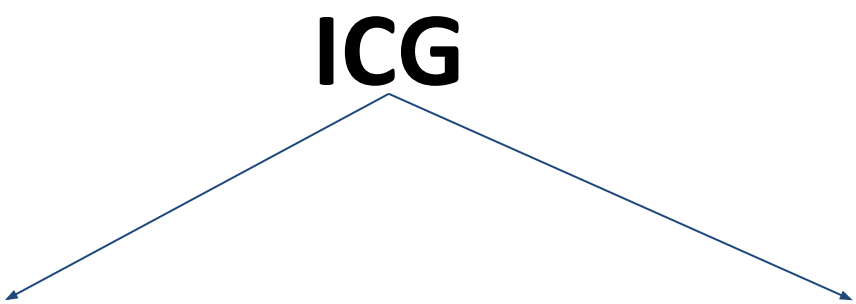
Various intermediate codes

Figure 6.2: A compiler might use a sequence of intermediate representations

- Higher level inter. Codes are closer to the HLL
- Low level inter. Codes are closer to the m/c.

- Originally, frontend for C++ simply translated into C code (intermediate language!). Then, backend is a C compiler.

Different Types



High Level

- Close to source language
- Easy to generate from i/p program
- Code optimization difficult

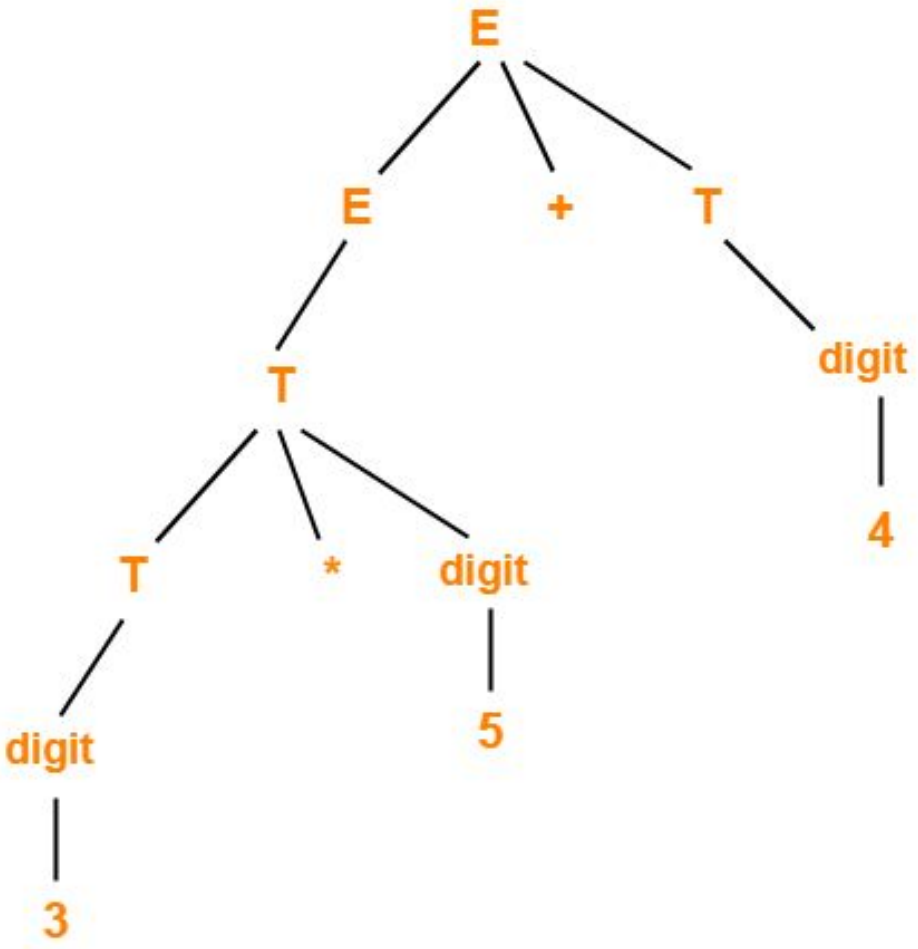
Low Level

- Close to target language
- Easy to generate final code
- Need more effort to generate

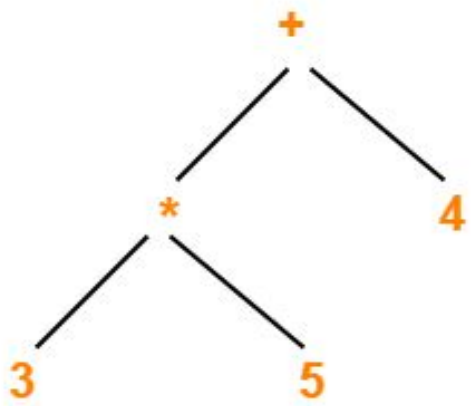
Different Types

- Abstract Syntax Tree
- Directed Acyclic Graph(DAG)
- Polish Notation
- Three Address Code

Abstract Syntax Tree



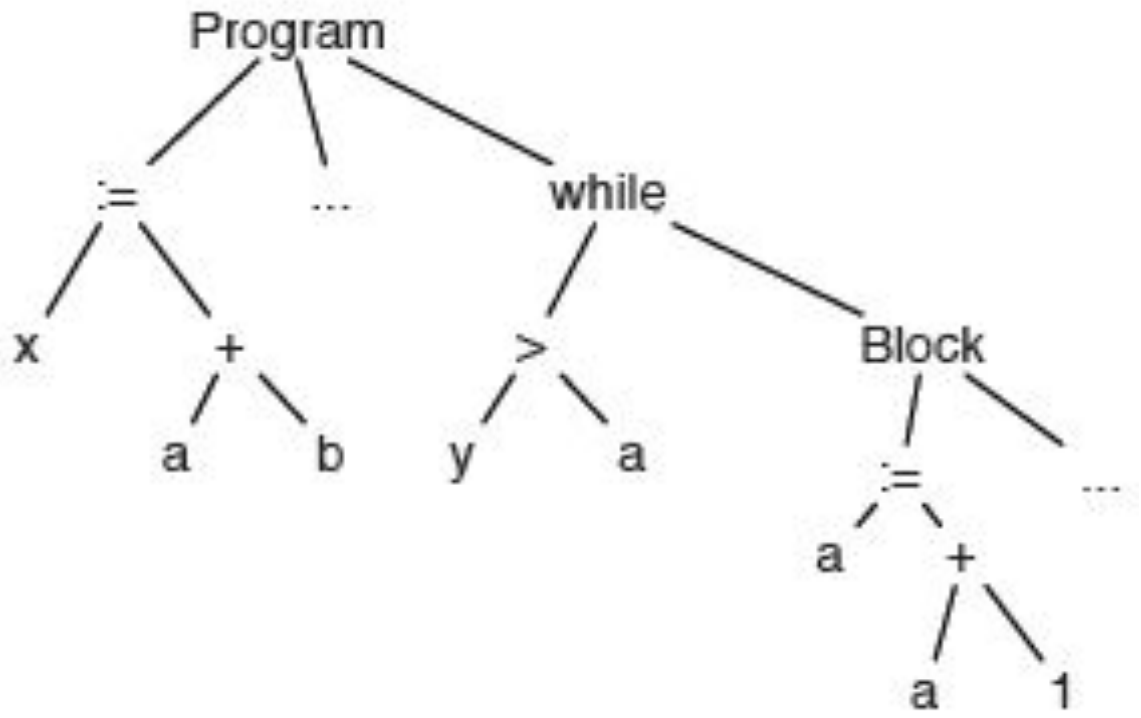
Parse Tree



Syntax Tree

Abstract Syntax Tree

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



6.1.1 Directed Acyclic Graphs for Expressions

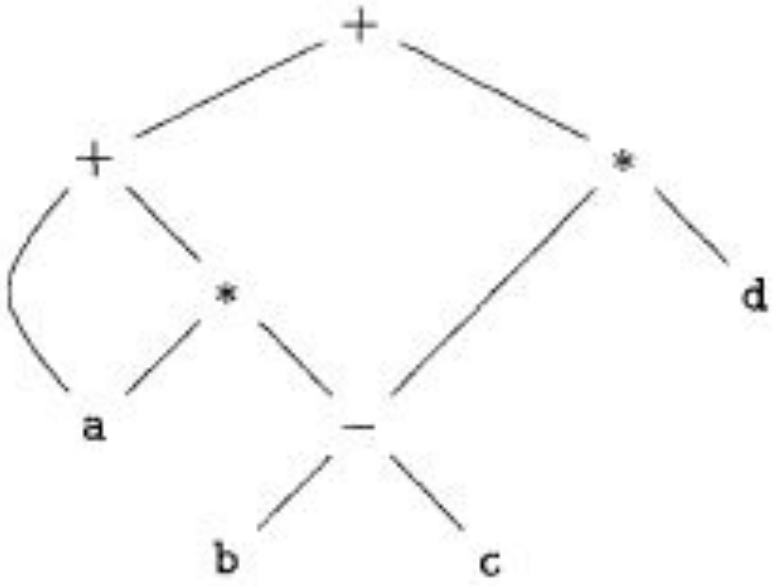


Figure 6.3: Dag for the expression `a + a * (b - c) + (b - c) * d`

- SDD can be used
- Modification we need is
  - Whenever you want to create a new node verify whether a node with identical information already exists … if so use that.

	PRODUCTION	SEMANTIC RULES
1)	$E \rightarrow E_1 + T$	$E.node =$ **new** $Node('+', E_1.node, T.node)$
2)	$E \rightarrow E_1 - T$	$E.node =$ **new** $Node('-', E_1.node, T.node)$
3)	$E \rightarrow T$	$E.node = T.node$
4)	$T \rightarrow (E)$	$T.node = E.node$
5)	$T \rightarrow$ **id**	$T.node =$ **new** $Leaf($**id**, **id**$.entry)$
6)	$T \rightarrow$ **num**	$T.node =$ **new** $Leaf($**num**, **num**$.val)$

Figure 6.4: Syntax-directed definition to produce syntax trees or DAG's

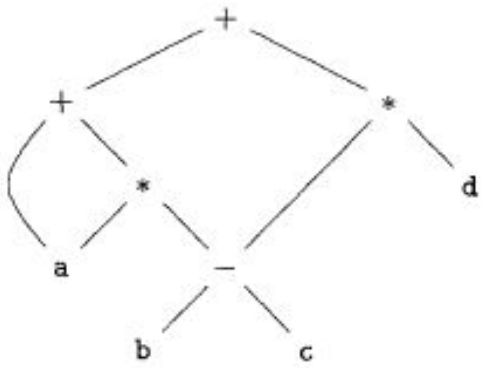
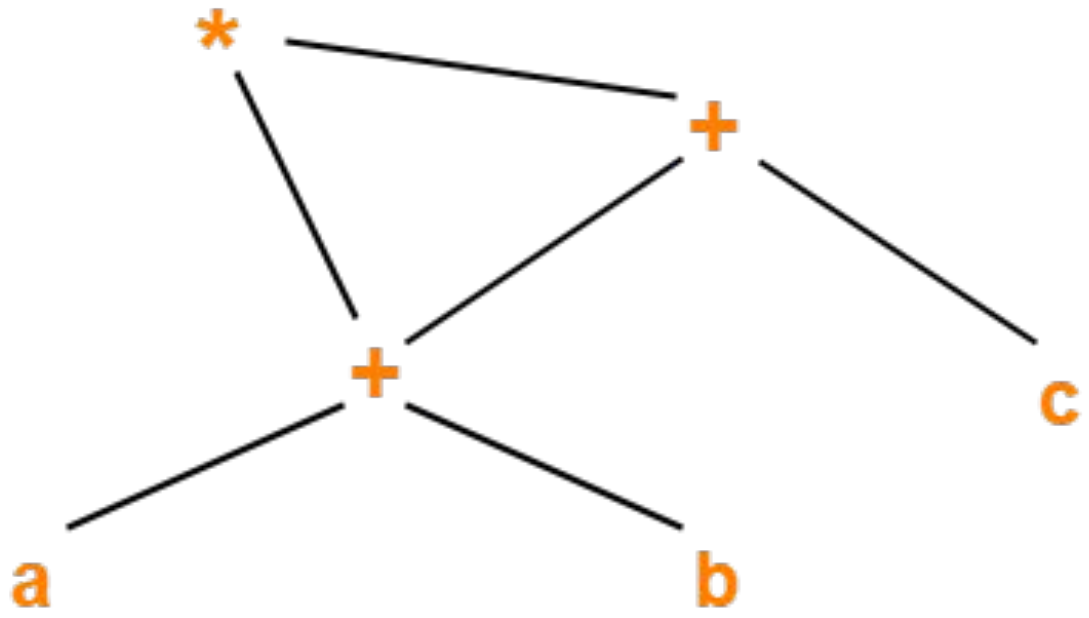


Figure 6.3: Dag for the expression `a + a * (b - c) + (b - c) * d`

Directed Acyclic Graph(DAG)

$$(a + b) \times (a + b + c)$$

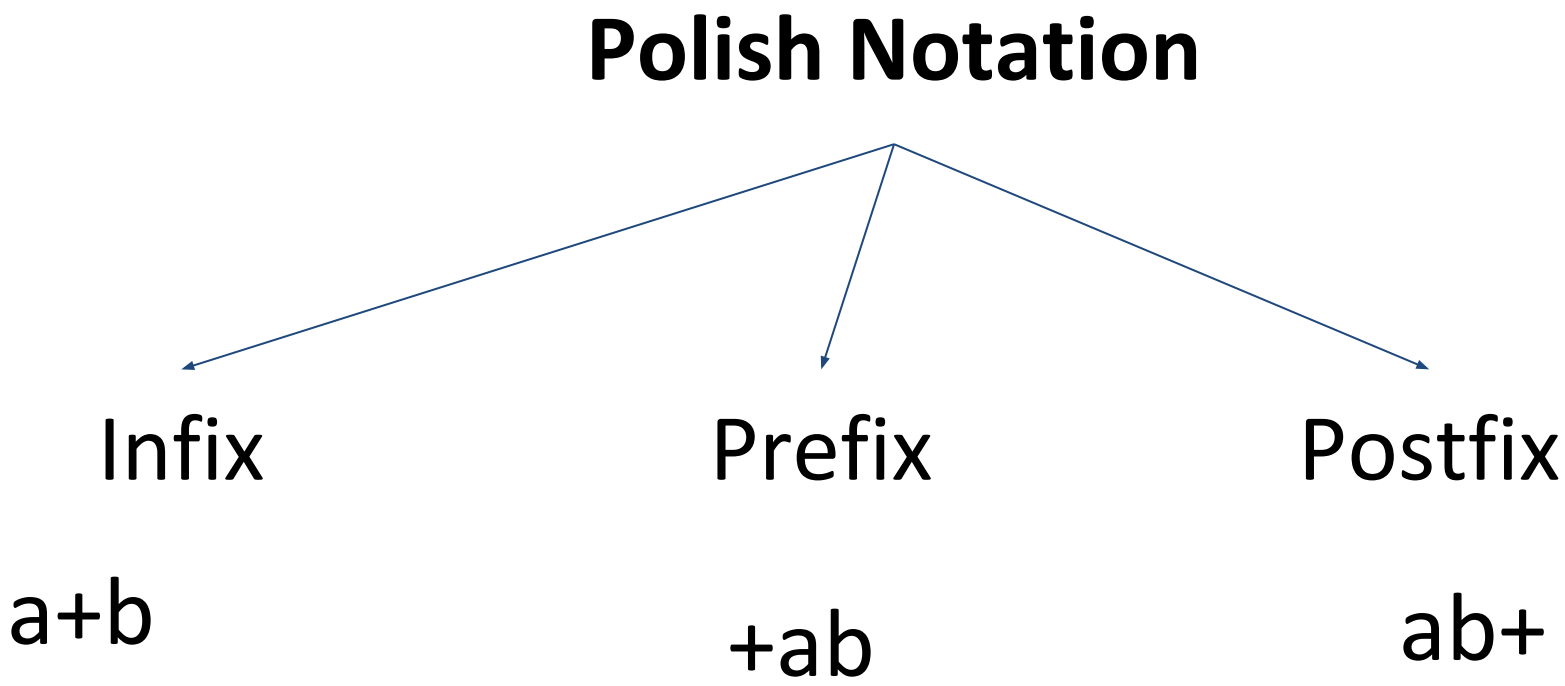
Directed Acyclic Graph(DAG)



Directed Acyclic Graph

(a + b) x (a + b + c)

Polish Notation



a+b

Polish Notation

(a – b) * (c + d) + (a – b)

Postfix ?

Polish Notation

(a – b) * (c + d) + (a – b)

Postfix ?

ab – cd + *ab -+

6.2 Three-Address Code

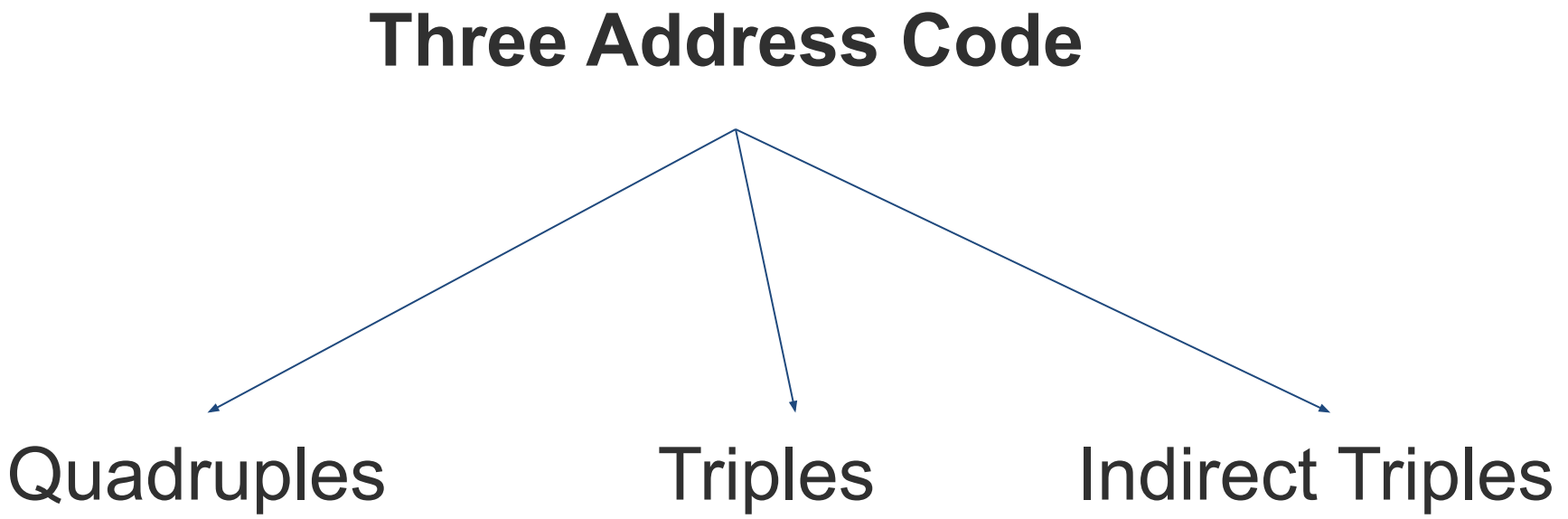
In three-address code, there is at most one operator on the right side of an instruction; that is, no built-up arithmetic expressions are permitted.

Thus a source-language expression like `x+y*z` might be translated into the sequence of three-address instructions

$$\begin{aligned} t_1 &= y * z \\ t_2 &= x + t_1 \end{aligned}$$

where t_1 and t_2 are compiler-generated temporary names.

Three Address Code



Quadruples

It consist of 4 fields:

- op: Operator
- arg1: Operand 1
- arg2: Operand 2
- result: store the result

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.

Quadruples

a = b * – c + b * – c

t1 = uminus c
t2 = b * t1
t3 = uminus c
t4 = b * t3
t5 = t2 + t4
a = t5

Quadruples

#	Op	Arg1	Arg2	Result
(0)	uminus	c		t1
(1)	*	t1	b	t2
(2)	uminus	c		t3
(3)	*	t3	b	t4
(4)	+	t2	t4	t5
(5)	=	t5		a

Quadruple representation

Triples

It consist of three fields:

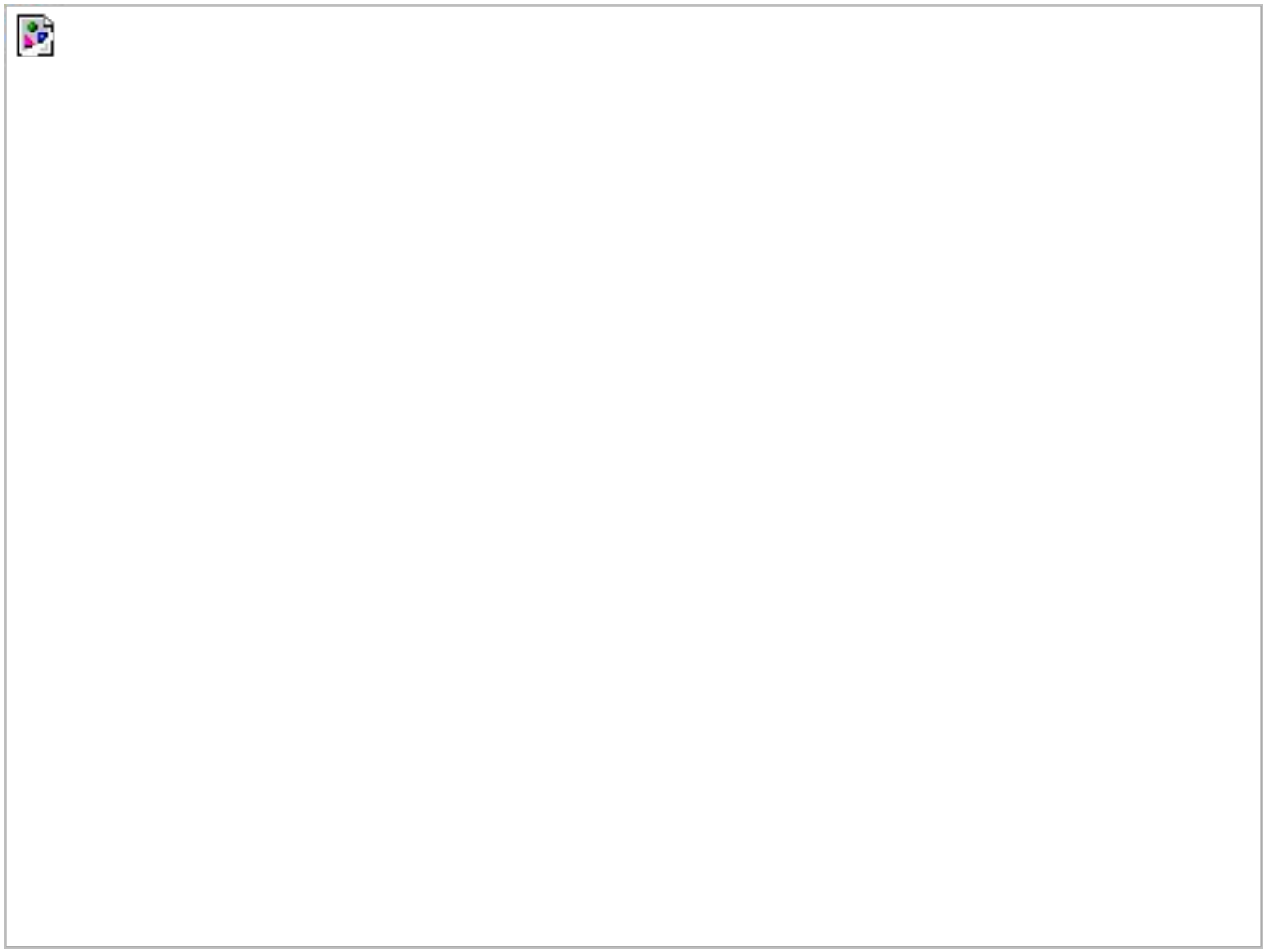
- op,
- arg1
- arg2.

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.

Triples

a = b * – c + b * – c

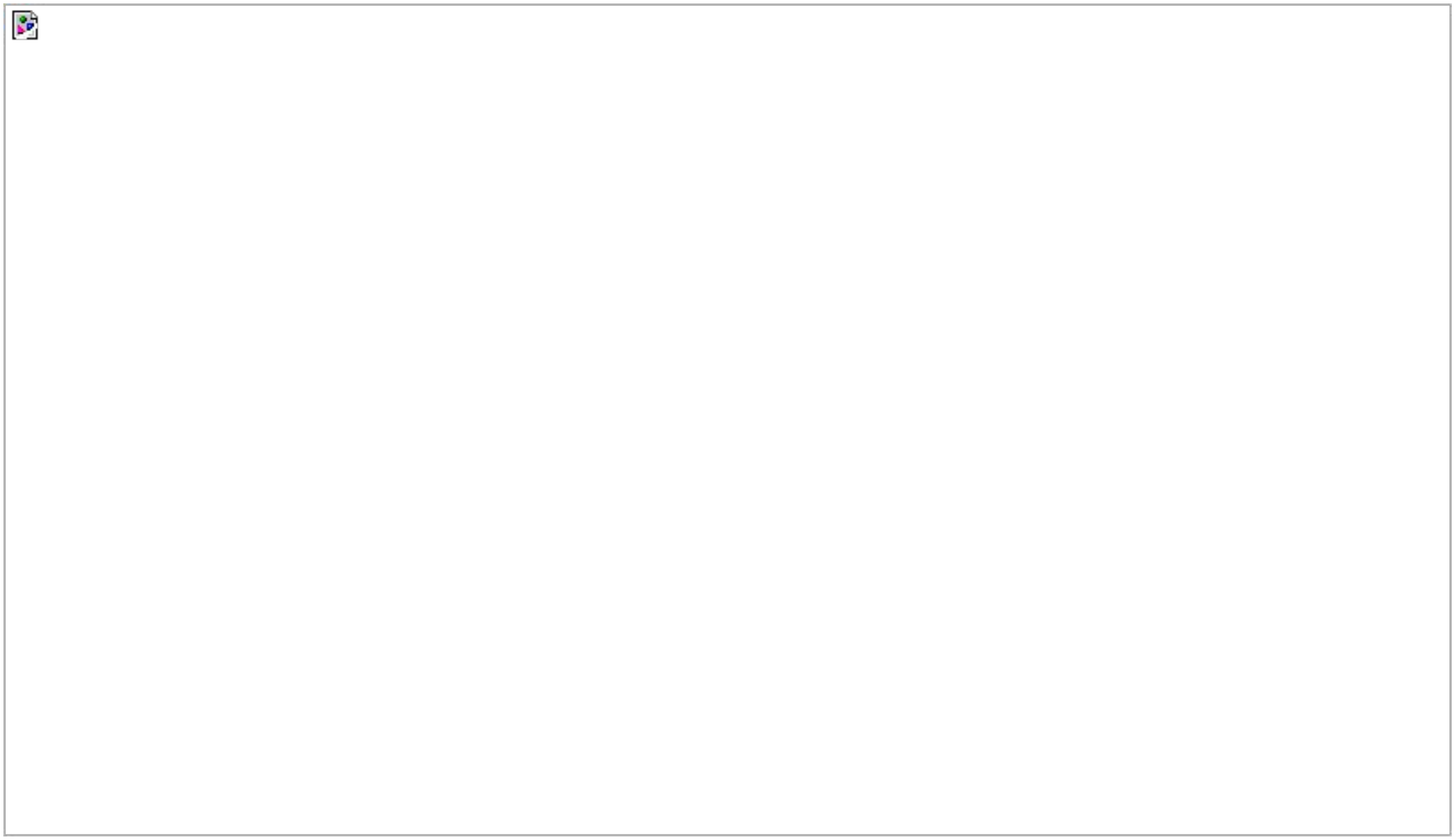


Indirect Triples

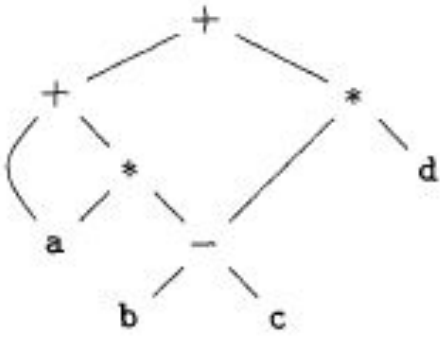
This representation makes use of pointer to the listing of all references to computations which is made separately and stored. Its similar in utility as compared to quadruple representation but requires less space than it. Temporaries are implicit and easier to rearrange code.

Indirect Triples

a = b * – c + b * – c



Example 6.4: 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.



(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

Figure 6.8: A DAG and its corresponding three-address code

Exercise 6.2.1: Translate the arithmetic expression $a + -(b + c)$ into:

a) A syntax tree.

b) Quadruples.

c) Triples.

d) Indirect triples.

6.2.1 Addresses and Instructions

An address can be one of the following:

- *A name.* For convenience, we allow source-program names to appear as addresses in three-address code.

  In an implementation, a source name is replaced by a pointer to its symbol-table entry, where all information about the name is kept.

- *A constant.*

- *A compiler-generated temporary.* It is useful, especially in optimizing compilers, to create a distinct name each time a temporary is needed.

Common Three address instructions

1. Assignment instructions of the form $x = y \ op \ z$, where op is a binary arithmetic or logical operation, and x, y, and z are addresses.

Common Three address instructions

1. Assignment instructions of the form $x = y \ op \ z$, where op is a binary arithmetic or logical operation, and x, y, and z are addresses.

2. Assignments of the form $x = op \ y$, where op is a unary operation.

Common Three address instructions

1. Assignment instructions of the form $x = y \ op \ z$, where *op* is a binary arithmetic or logical operation, and x, y, and z are addresses.

2. Assignments of the form $x = op \ y$, where *op* is a unary operation.

3. *Copy instructions* of the form $x = y$, where x is assigned the value of y.

Common Three address instructions

1. Assignment instructions of the form $x = y \ op \ z$, where *op* is a binary arithmetic or logical operation, and x, y, and z are addresses.

2. Assignments of the form $x = op \ y$, where *op* is a unary operation.

3. *Copy instructions* of the form $x = y$, where x is assigned the value of y.

4. An unconditional jump `goto` L. The three-address instruction with label L is the next to be executed.

Common Three address instructions

1. Assignment instructions of the form $x = y \ op \ z$, where *op* is a binary arithmetic or logical operation, and x, y, and z are addresses.

2. Assignments of the form $x = op \ y$, where *op* is a unary operation.

3. *Copy instructions* of the form $x = y$, where x is assigned the value of y.

4. An unconditional jump `goto` L. The three-address instruction with label L is the next to be executed.

5. Conditional jumps of the form `if` x `goto` L and `ifFalse` x `goto` L. These instructions execute the instruction with label L next if x is true and false, respectively.

6. Conditional jumps such as `if` x *relop* y `goto` L, which apply a relational operator (<, ==, >=, etc.) to x and y, and execute the instruction with label L next if x stands in relation *relop* to y.

6. Conditional jumps such as `if` x *relop* y `goto` L, which apply a relational operator (<, ==, >=, etc.) to x and y, and execute the instruction with label L next if x stands in relation *relop* to y.

7. Procedure calls and returns are implemented using the following instructions: `param` x for parameters; `call` p, n and y = `call` p, n for procedure and function calls, respectively; and `return` y, where y, representing a returned value, is optional. Their typical use is as the sequence of three-address instructions

```
param x1
param x2
...
param xn
call p, n
```

generated as part of a call of the procedure $p(x_1, x_2, \ldots, x_n)$.

8. Indexed copy instructions of the form $x = y[i]$ and $x[i] = y$.

9. Address and pointer assignments of the form $x = \&y$, $x = *y$, and $*x = y$.

Statement in Three Address Code

- Intermediate languages usually have the following types of statements
  - Assignment
  - Jumps
  - Address and Pointer Assignments
  - Procedure Call/Return
  - Miscellaneous

Assignment

- Three types of assignment statements
  - x = y op z, op being a binary operator
  - x = op y, op being a unary operator
  - x = y

Jump

- Both conditional and unconditional jumps are required
  - goto L, L being a label
  - if x relop y goto L

Arrays

- Only one-dimensional arrays need to be supported
- Arrays of higher dimensions are converted to one-dimensional arrays
- Statements to be supported
  - x = y[i]
  - x[i] = y

Address & Pointer

- Statements required are of following types
  - x = &y, address of y assigned to x
  - x = *y, content of location pointed to by y is assigned to x
  - x = y, simple pointer assignment, where x and y are pointer variables

Procedure Call

- A call to the procedure P(x1, x2, …, xn) is converted as

```
param x1
param x2
…
param xn
```

Static Single-Assignment Form

- An IR (Intermediate representation) form
- Useful in optimization
- Idea: Use various versions of a variable (each with a distinct name)
- When time comes to use the variable, apply the merge operation to get the exact version that should be used.

```
p = a + b
q = p - c
p = q * d
p = e - p
q = p + q
```

(a) Three-address code.

```
p1 = a + b
q1 = p1 - c
p2 = q1 * d
p3 = e - p2
q2 = p3 + q1
```

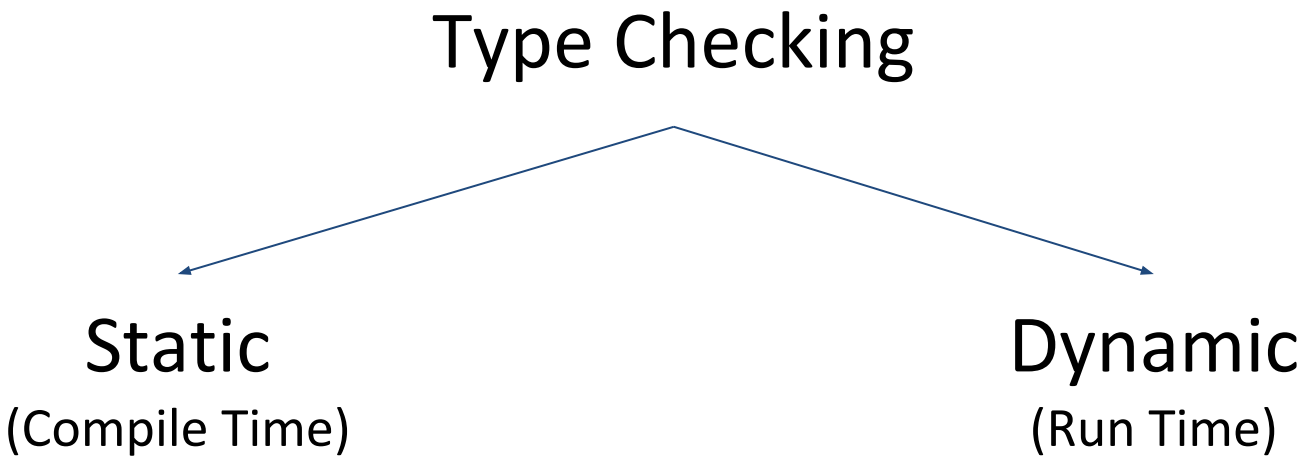
(b) Static single-assignment form.

```
if ( flag ) x = -1; else x = 1;
y = x * a;
```

```
if ( flag ) x1 = -1; else x2 = 1;
x3 = φ(x1, x2);
```

Data flow analysis can tell us about the merge function.

Type Checking



Type Checking

- Type checking uses logical rules to reason about the behavior of a program at run time.
  - Types of operands should match.
  - Relational operator requires Boolean operands.

Type Expressions

Type of a language construct is denoted by a type expressions.
It can be:

- A Basic Type
  - datatype such as int, char, etc
- A Type name
  - a name used for data expression

Type Expressions

- Type constructor is used to create complicated types
  - Classes
  - Structures
  - Pointers to arrays, array of pointers, etc

Type Expressions

- In C, `int [2][3]` is a type , it is an array of two elements, where each element is an array of 3 integers.
- The corresponding type expression is array(2,array(3,integer)). Second argument is a type, first argument is the number of elements.

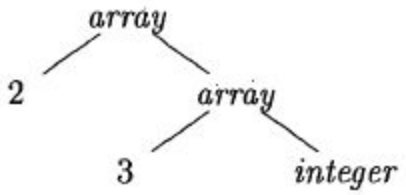
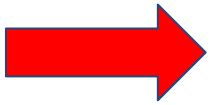


Figure 5.15: Type expression for **int**[2][3]

Type Expressions

- Basic type is a type expression.
- A type name is a type expression.
- array(number, type expression).
- record(type exp1, type exp2, …).

E → id  **{E.type := int}**

Type Equivalence

- Often, we want to verify whether two types are equivalent or not (error has to be given).
- Operations are defined on same/particular typed variables/values
  - So type checking is a must
- Ambiguity arise when names are given to type expressions, which in turn are used in creating new types.

Two types are structurally equivalent ..

- If and only if one of the following is true:

- They are the same basic type.
- They are formed by applying the same constructor to structurally equivalent types.
- One is a type name that denotes the other.

- First two conditions define *name equivalence* of type expressions.