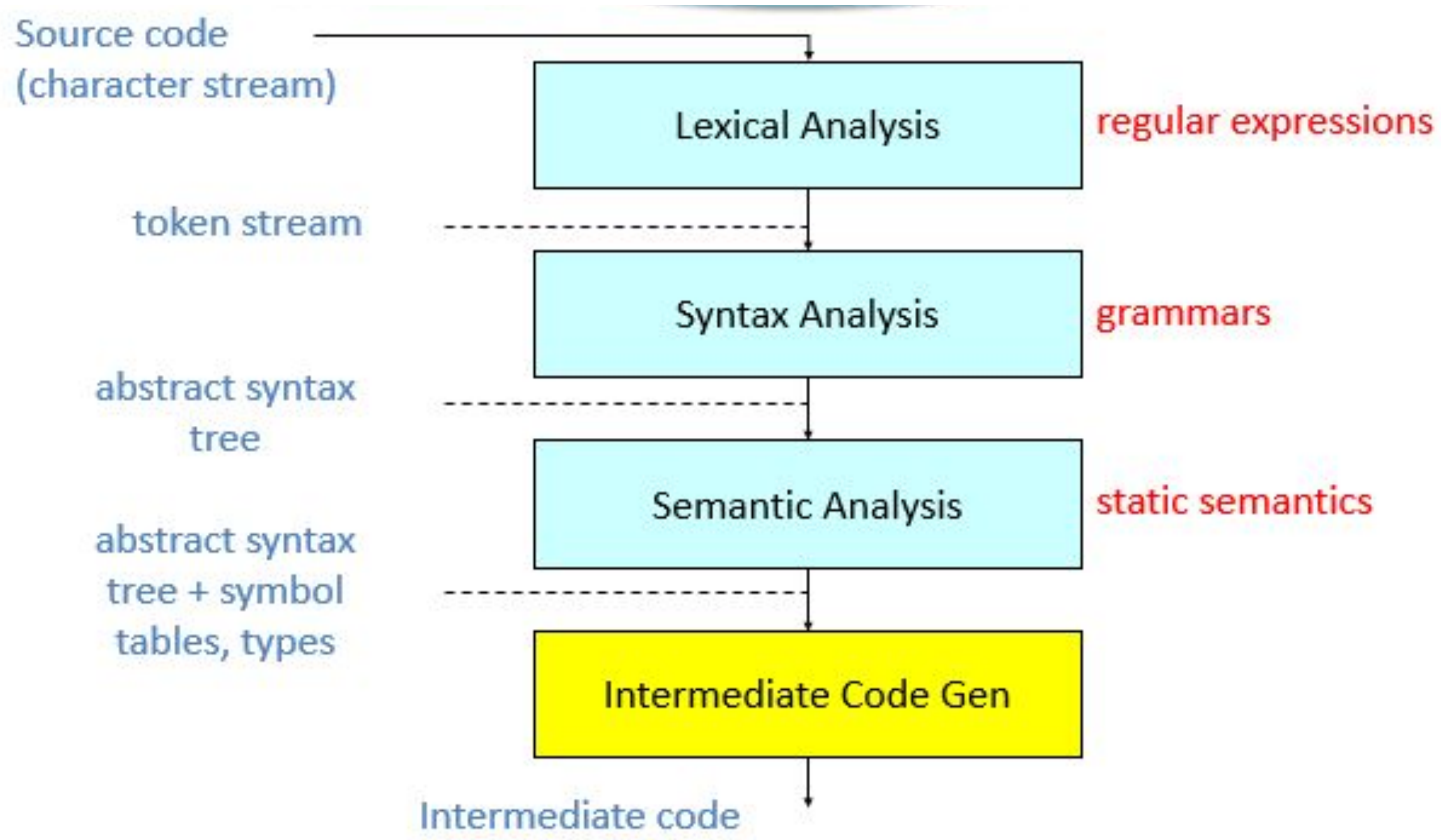
Intermediate Representation (IR)

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

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

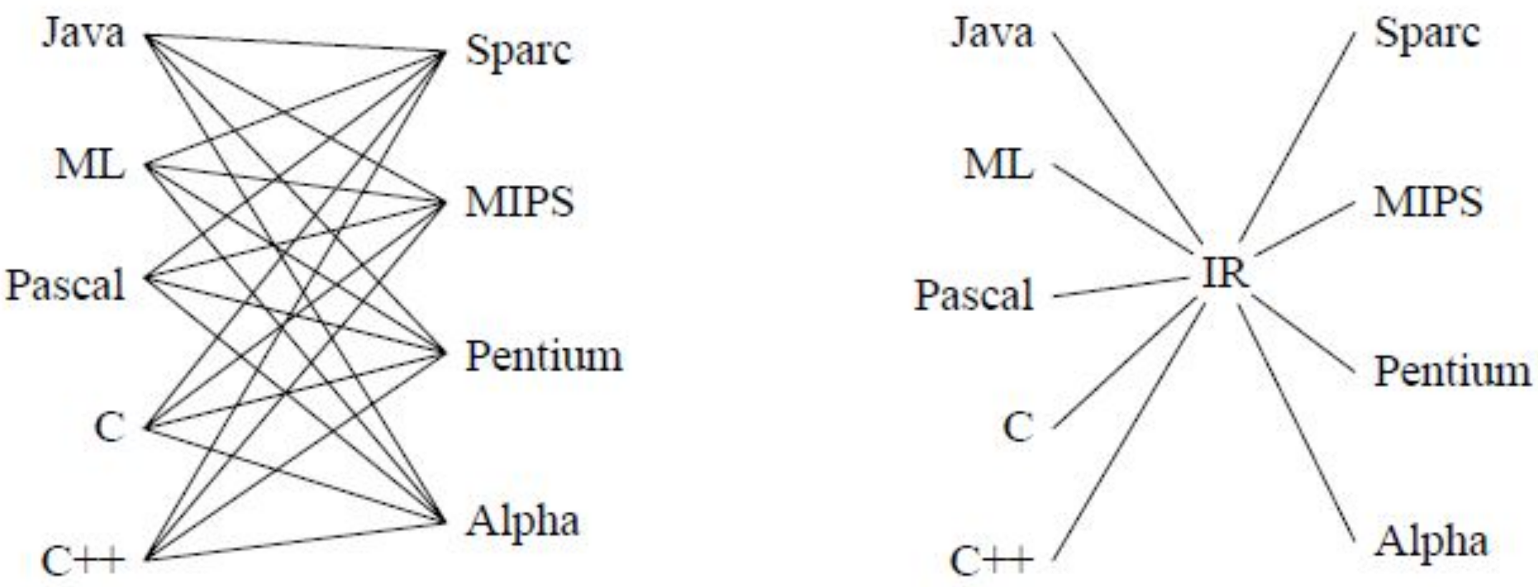
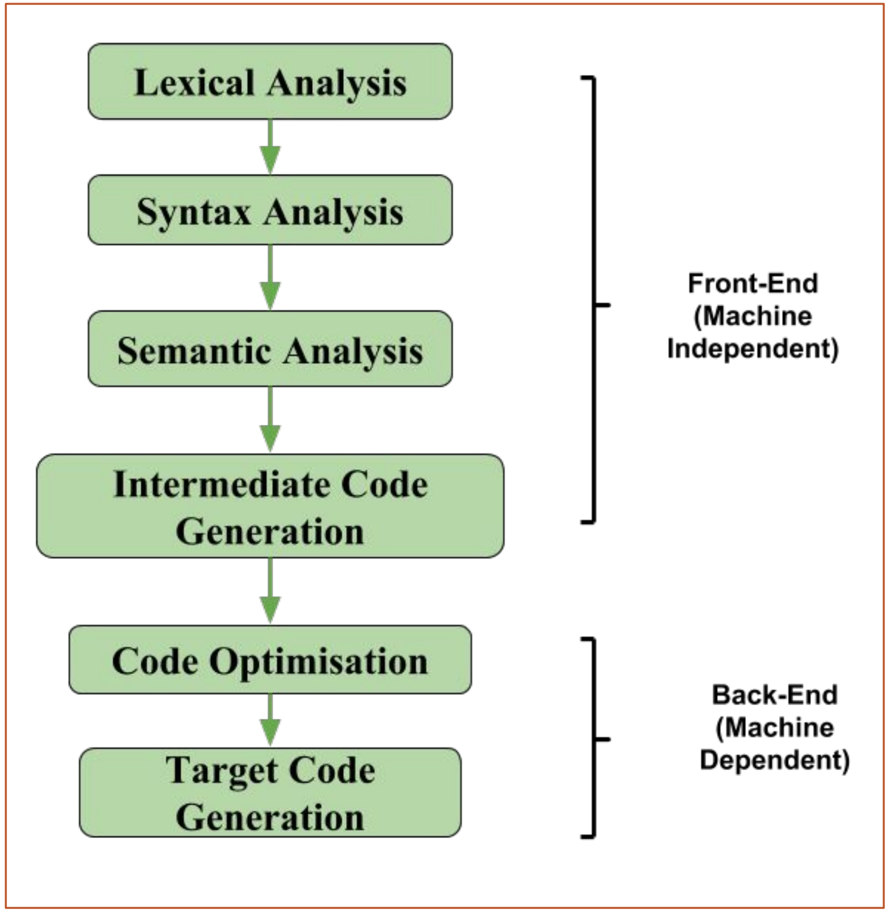


FIGURE 7.1. Compilers for five languages and four target machines: (left) without an IR, (right) with an IR.
From *Modern Compiler Implementation in ML,* Cambridge University Press, ©1998 Andrew W. Appel

Intermediate Representation

- ✔ If we generate machine code <u>directly from source code</u>, then for ***n target*** machine we will have ***n optimizers*** and ***n code generators*** but if we will have <u>a machine independent intermediate code</u>, 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 Code

- Intermediate language can be many different languages, and the designer of the compiler decides this intermediate language.
  - syntax trees can be used as an intermediate language.
  - postfix notation can be used as an intermediate language.
  - three-address code can be used as an intermediate language
- Intermediate language may have various levels.

Intermediate Languages Types

- Graphical Intermediate Representations:
  - Abstract Syntax trees
  - Directed Acyclic Graphs
  - Control Flow Graphs

- Linear Intermediate Representations :
  - Stack based (postfix)
  - Three address code (quadruples)

Graphical Intermediate Representations

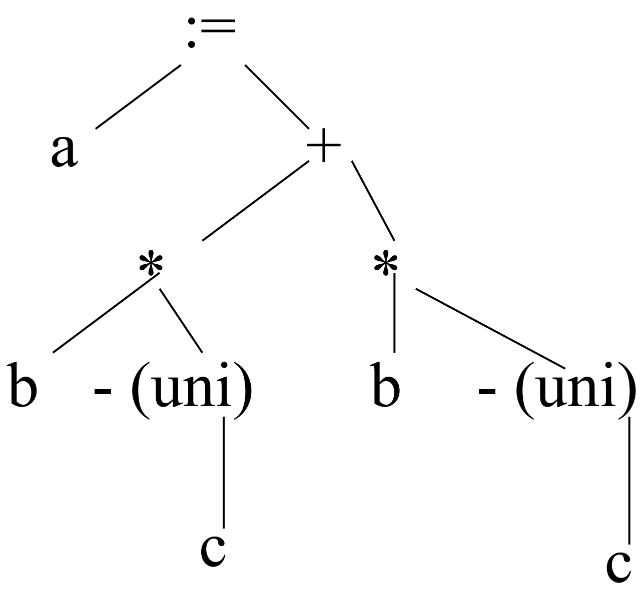
- Abstract Syntax Trees (AST) – retain essential structure of the parse tree, eliminating unneeded nodes.

- Directed Acyclic Graphs (DAG) – compacted AST to avoid duplication – smaller footprint as well

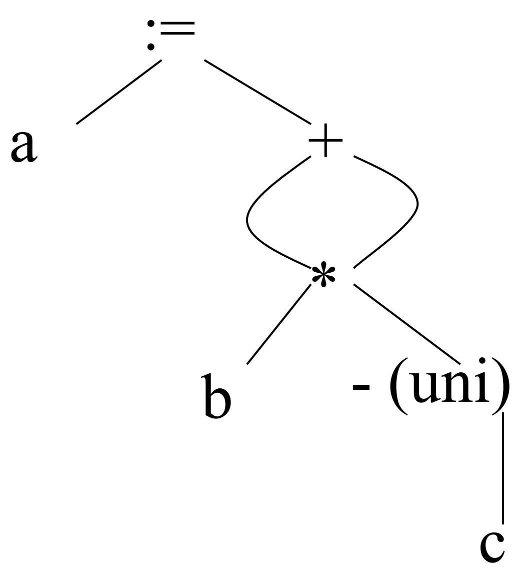
- Control flow graphs (CFG) – explicitly model control flow

Abstract Syntax Trees and Directed Acyclic Graphs:

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



AST



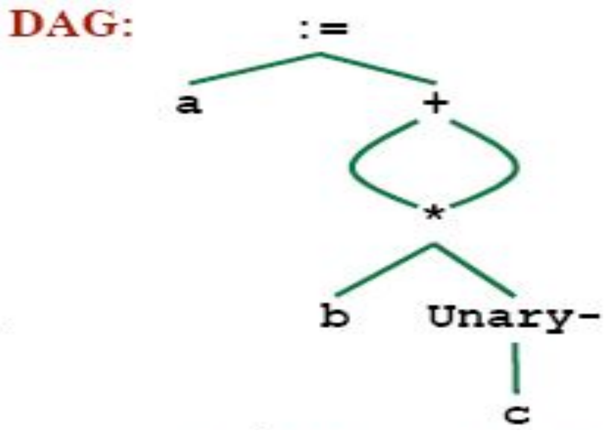
DAG

Linearized Representation of DAG

- Source Code
  - a = b * -c + b * -c
- Three address code

```
t1  := -c
t2  := b * t1
t5  := t2 + t2
a  := t5
```

- DAG Representation



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 –

```
a = b 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.

Quadruples

- A quadruple is a record structure with four fields: op, arg2, arg2, result.

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

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

	op	*arg1*	*arg2*	*result*
(0)	uminus	c		t_1
(1)	*	b	t_1	t_2
(2)	uminus	c		
(3)	*	b	t_3	t_4
(4)	+	t_2	t_4	t_5
(5)	:=	t_5		a

Triples

- Avoids entering temporary names into the symbol table.
- Temporary values are referred by the position of the statement that computes it.
- Requires three fields: op, arg1, arg2.

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

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

	op	*arg1*	*arg2*
(0)	**uminus**	**c**	
(1)	*****	**b**	**(0)**
(2)	**uminus**	**c**	
(3)	*****	**b**	**(2)**
(4)	**+**	**(1)**	**(3)**
(5)	**assign**	**a**	**(4)**

Intermediate Representation Formats

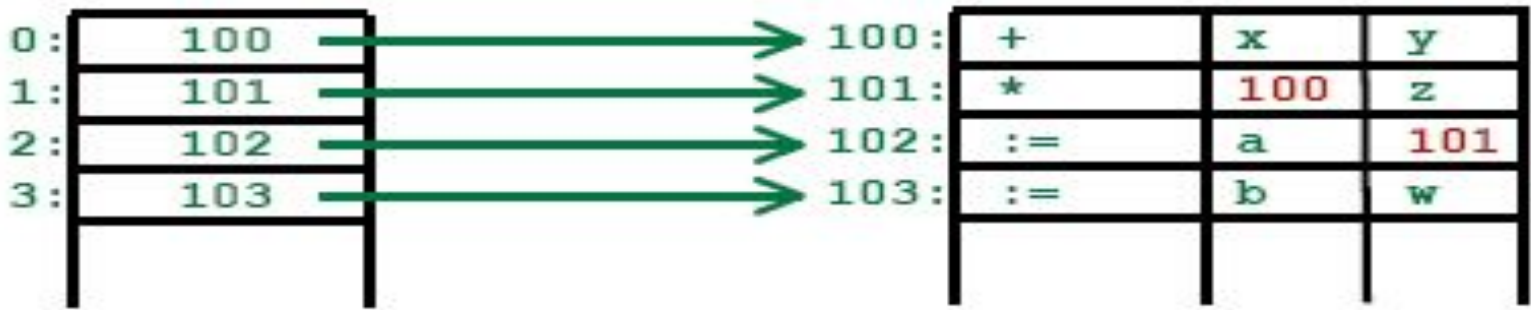
Implementation of Three-Address Code

<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.**

Indirect Triples

- Listing pointer to triplets.
- Solves the reordering problem.



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

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

	stmnt
(0)	**(14)**
(1)	**(15)**
(2)	**(16)**
(3)	**(17)**
(4)	**(18)**
(5)	**(19)**

	op	*arg1*	*arg2*
(14)	**uminus**	**c**	
(15)	*****	**b**	**(14)**
(16)	**uminus**	**c**	
(17)	*****	**b**	**(16)**
(18)	**+**	**(15)**	**(17)**
(19)	**assign**	**a**	**(18)**

Three Address Code, Quadruples, Triples, and Indirect Triples Example-2

Construct Three Address Code, Quadruples, Triples, and Indirect Triples for the expression

-(a + b) * (c + d) - (a + b + c)

Three Address Code

First of all this statement will be converted into Three Address Code as−

$t1 = a + b$

$t2 = -t1$

$t3 = c + d$

$t4 = t2 * t3$

$t5 = t1 + c$

$t6 = t4 - t5$

Quadruples

Location	Operator	arg 1	arg 2	Result
(0)	+	a	b	t1
(1)	−	t1		t2
(2)	+	c	d	t3
(3)	*	t2	t3	t4
(4)	+	t1	c	t5
(5)	−	t4	t5	t6

Triples

Location	**Operator**	**arg 1**	**arg 2**
(0)	+	a	b
(1)	−	(0)	
(2)	+	c	d
(3)	*	(1)	(2)
(4)	+	(0)	c
(5)	−	(3)	(4)

Indirect Triples

Indirect Triple

Statement	
(0)	(11)
(1)	(12)
(2)	(13)
(3)	(14)
(4)	(15)
(5)	(16)

Location	Operator	arg 1	arg 2
(11)	+	a	b
(12)	-	(11)	
(13)	+	c	d
(14)	*	(12)	(13)
(15)	+	(11)	c
(16)	-	(14)	(15)

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 –

$x = (a + b * c) / (a – b * c)$