Compiler Design

Lecture 10: Intermediate Code Generation

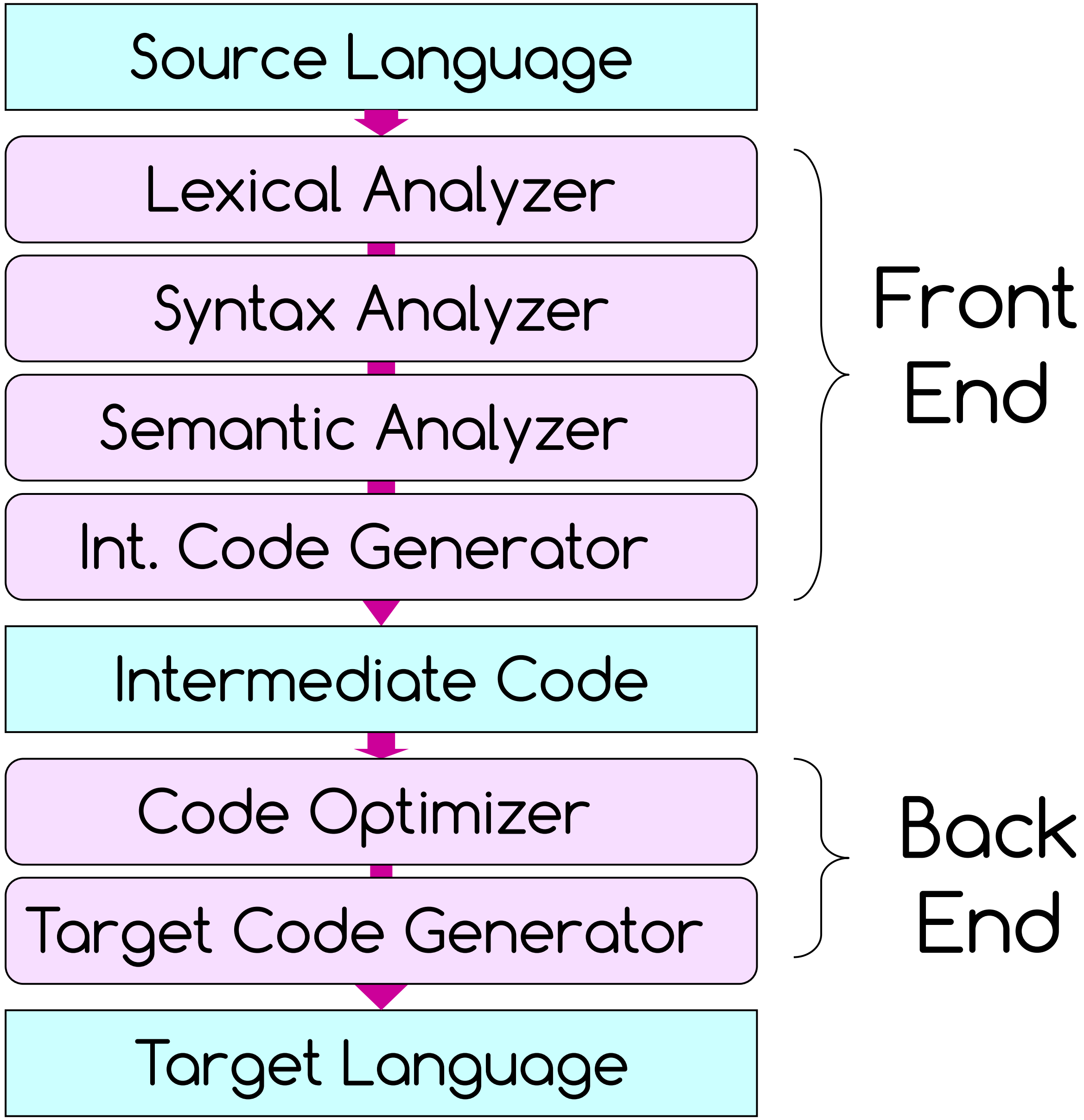
Sahar Selim

Agenda

- Intermediate Code Generation

- Intermediate Representations
  1) Postfix notation
  2) Three address code
  3) Syntax tree
     - Directed Acyclic Graph

Structure of a Compiler

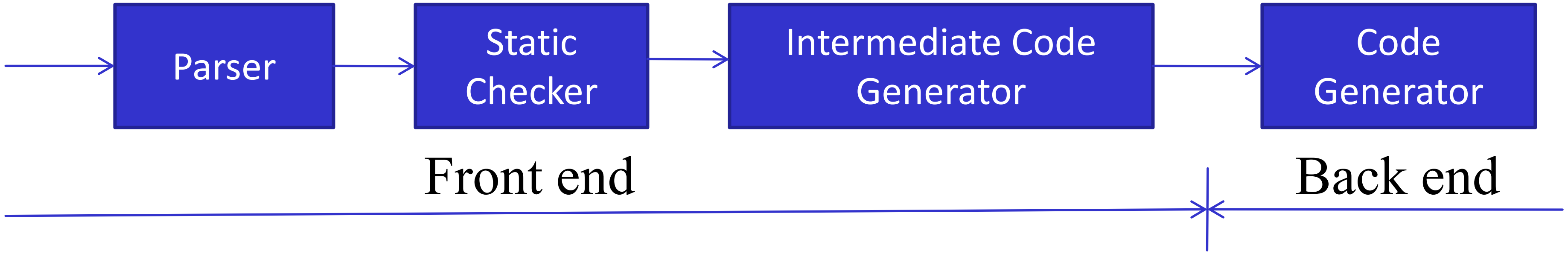


Intermediate Code Generation

- Intermediate codes are **machine independent** codes, but they are **close to machine instructions**.
- The given program in a source language is converted to an equivalent program in an intermediate language by the intermediate code generator.
- It ties the front and back ends together

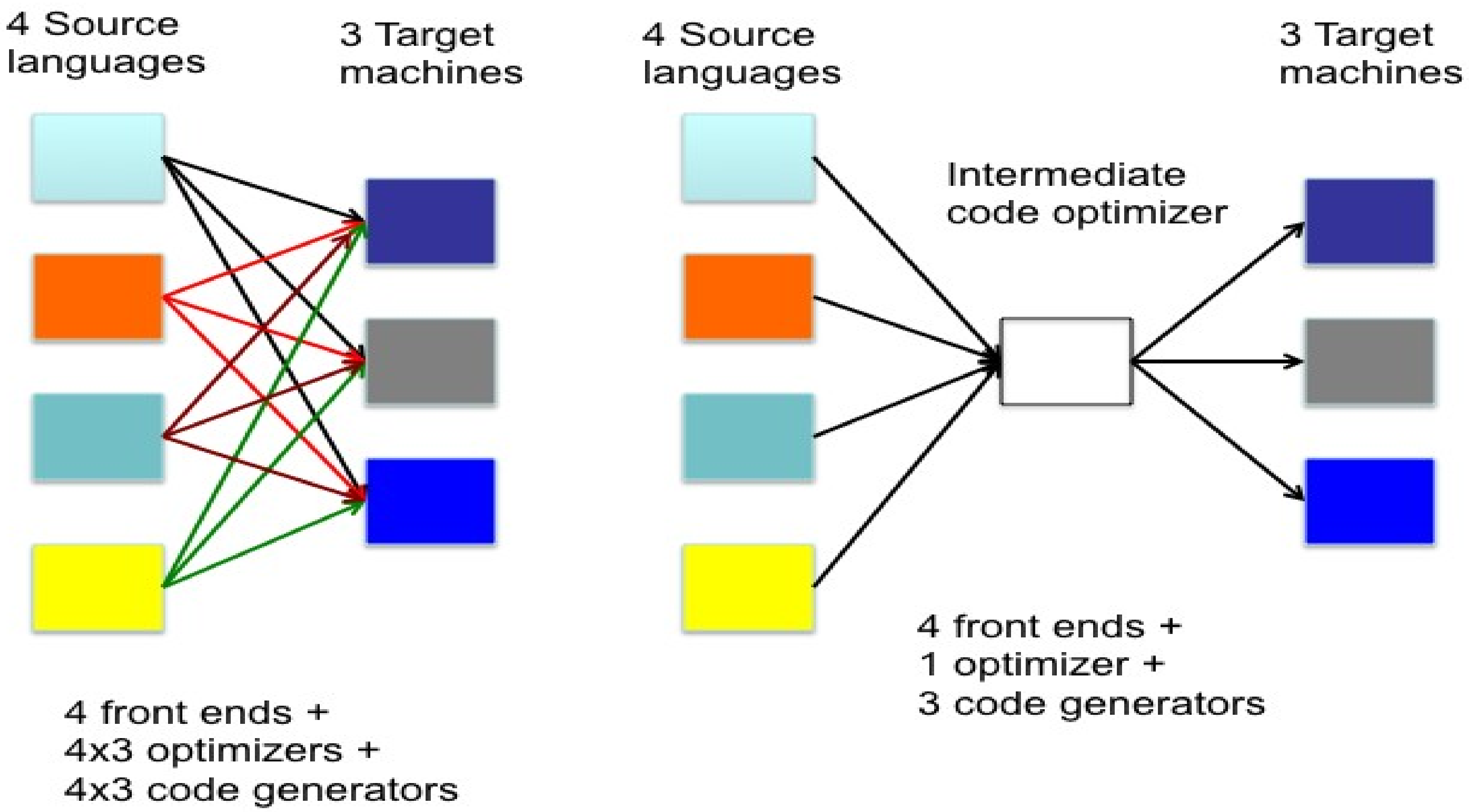
Intermediate Representation (IR)

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



- Why IR ?

Why Intermediate Code?



Why Intermediate Code?

- While generating machine code directly from source code is possible, it entails two problems
  - With m languages and n target machines, we need to write m front ends, $m \times n$ optimizers, and $m \times n$ code generators
  - The code optimizer which is one of the largest and very-difficult-to-write components of a compiler, cannot be reused
- By converting source code to an intermediate code, a machine-independent code optimizer may be written
- This means just m front ends, n code generators and 1 optimizer

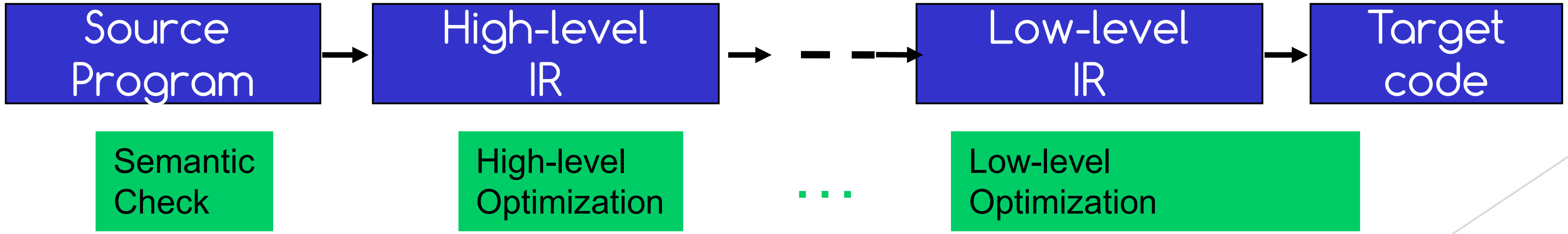
Advantages of Using an Intermediate Language

1. Retargeting
   - Build a compiler for a new machine by attaching a new code generator to an existing front-end.
2. 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.

Types of Intermediate Languages

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



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

1 Postfix Notation

Postfix Notation

- In postfix notation, the operator follows the operand.
- For example, in the expression

  (a-b)* (c +d) the postfix representation is:
- ab – cd +*

Example 2

String: a+b*c+d*e^f

Example 2

String: a+b*c+d*e^f

postfix: abc*+def^*+

a + b * c + d * e ↑ f	where,
a + T1 + d * e ↑ f	T1 = bc*
T2 + d * e ↑ f	T2 = aT1+
T2 + d * T3	T3 = ef↑
T2 + T4	T4 = dT3*
T5	T5 = T2T4+

backward Substitution the value of temporary variables →

T5
T2T4+
T2dT3*+
T2def↑*+
aT1+def↑*+
abc*+def↑*+

Postfix notation → abc*+def↑*+

Example 3

- Find the postfix notation of the following expressions:

Infix Notation	Postfix Notation
a = b * - c + b * - c	a b c uminus * b c uminus * + assign
(a + b) * c	ab + c*
(a – b) * (c + d) + (a – b)	ab – cd + *ab -+
a + b * c / (d - e)	a b c * d e - / +

2 Three-Address Code

Three Address Code.

- Three address code is a sequence of statement of the form x = y op z.
- Since a statement involves no more than three references, it is called a "three address statement", and a sequence of such statements is referred to as three address code.

Example

The three-address code for the expression

$$2 * a + (b - 3)$$

T1 = 2 * a

T2 = b - 3

T3 = T1 + T2

Three Address Code

- Sometimes a statement might contain less than three references; but it is still called a three-address statement.
- The following are the three address statements used to represent various programming language constructs:
- Used for representing arithmetic expressions:

X = Y op Z

X = op Y

X = Y

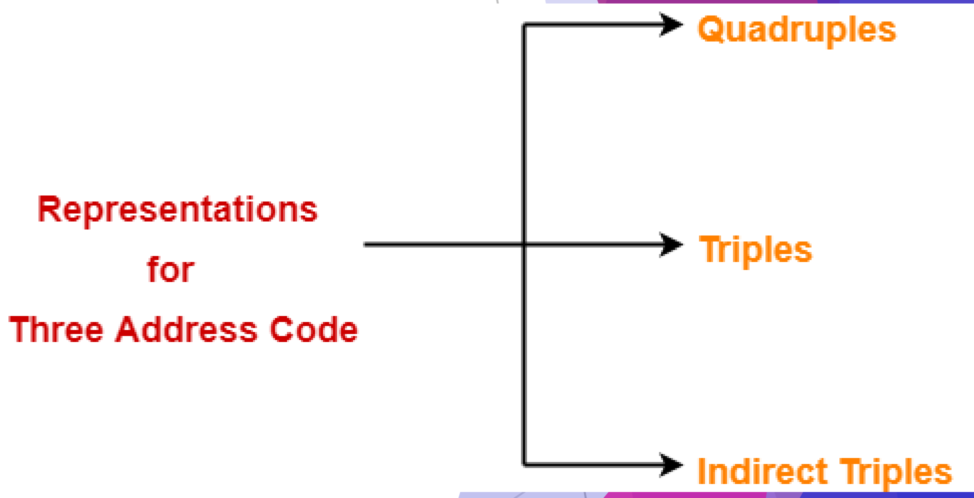
Three Address Code

- Used for representing Boolean expressions:

If A > B goto Z
goto Z

Data structures for three address codes

- Quadruples
  - Has four fields: op, arg1, arg2 and result
- Triples
  - Temporaries are not used and instead references to instructions are made
- Indirect triples
  - In addition to triples we use a list of pointers to triples



Quadruple Representation

- Using quadruple representation, the three-address statement **x = y op z** is represented by placing
  - op in the operator field
  - y in the operand1 field
  - z in the operand2 field
  - x in the result field

	Operator	Operand 1	Operand 2	Result
(1)	op	y	z	X
(2)	op	y		x
(3)				
(4)				
(5)				

Quadruple Representation

- The statement **x = op y**, where op is a unary operator, is represented by placing
  - op in the operator field
  - y in the operand1 field
  - x in the result field
  - the operand2 field is not used

Example 1

Define the quadruple representation of the three-address code for the statement

$$x = (a + b) * - c / d$$

Example1

x = ((a + b) * - c)/ d

	Operator	Operand 1	Operand 2	Result
(1)	+	a	b	t1
(2)	-	c		t2
(3)	*	t1	t2	t3
(4)	/	t3	d	t4
(5)	=	t4		x

Example 2

- Consider expression a = b * – c + b * – c.
- Define the three-address code with quadruples

Example 2

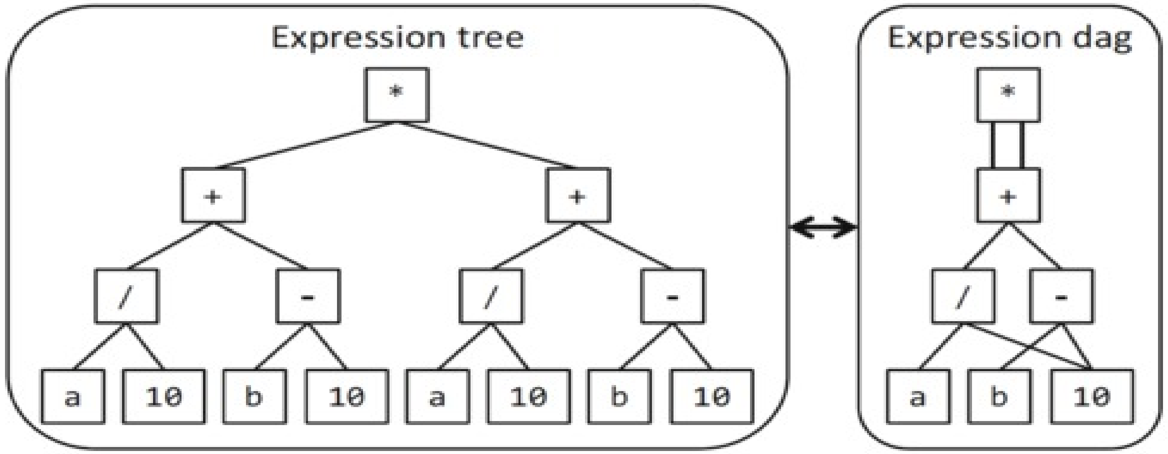
- Consider expression a = b * – c + b * – c.
- Define the three-address code with quadruples

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

#	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


Q
A



3 Syntax Trees

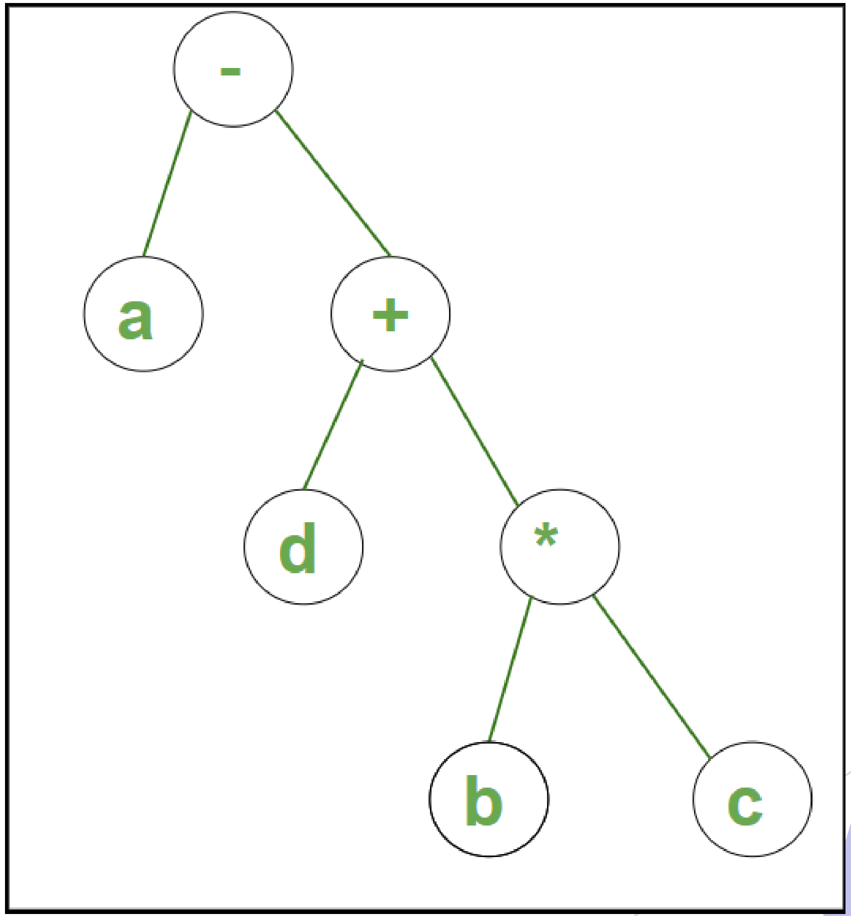
Directed Acyclic Graphs (DAG)

Syntax Tree

- A syntax tree is also known as **parse tree**.
- It represents the different categories of syntax used in the sentence.
- It enables one to understand the different **syntactical structure** of a sentence.

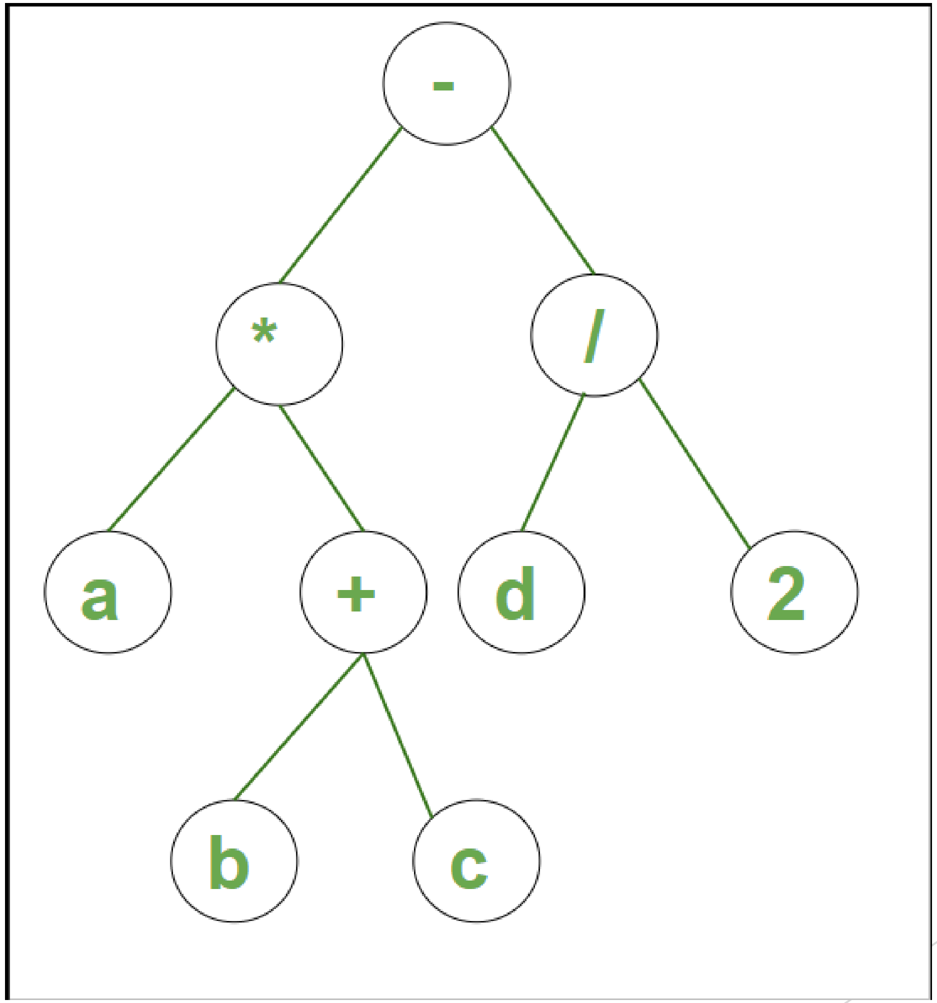
Syntax Tree: Example 1

- $a - b * c + d$



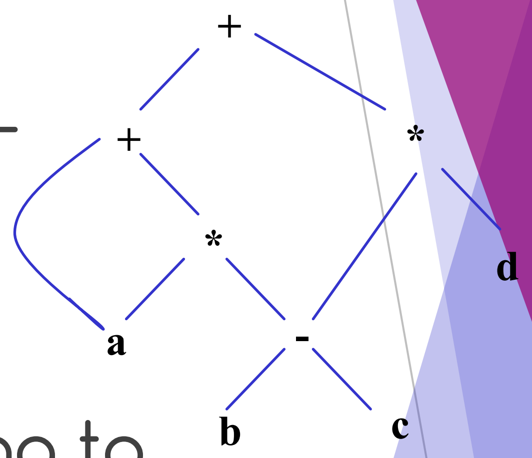
Syntax Tree: Example 2

- a * (b + c) – d /2



Variants of Syntax Trees

- It is sometimes beneficial to create a DAG instead of tree for Expressions.
- This way can easily show the common sub-expressions and then use that knowledge during code generation
- DAG has leaves corresponding to atomic operands and interior codes corresponding to operators.
- a **node N** in a **DAG has more than one parent** if N represents a common subexpression.



SDD for creating DAG's

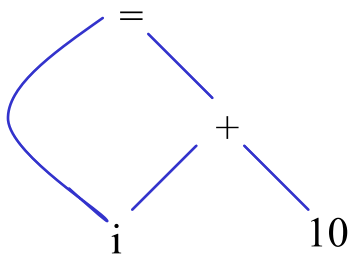
a+a*(b-c)+(b-c)*d

Production	Semantic Rules
1) E -> E1+T	E.node= new Node('+', E1.node,T.node)
2) E -> E1-T	E.node= new Node('-', E1.node,T.node)
3) E -> T	E.node = T.node
4) T -> (E)	T.node = E.node
5) T -> id	T.node = new Leaf(id, id.entry)
6) T -> num	T.node = new Leaf(num, num.val)

Example:

1) p1=Leaf(id, entry-a)
2) P2=Leaf(id, entry-a)=p1
3) p3=Leaf(id, entry-b)
4) p4=Leaf(id, entry-c)
5) p5=Node('-',p3,p4)
6) p6=Node('*',p1,p5)
7) p7=Node('+',p1,p6)
8) p8=Leaf(id,entry-b)=p3
9) p9=Leaf(id,entry-c)=p4
10) p10=Node('-',p3,p4)=p5
11) p11=Leaf(id,entry-d)
12) p12=Node('*',p5,p11)
13) p13=Node('+',p7,p12)

Value-number method for Constructing DAG's



id		
num	10	
+	1	2
3	1	3

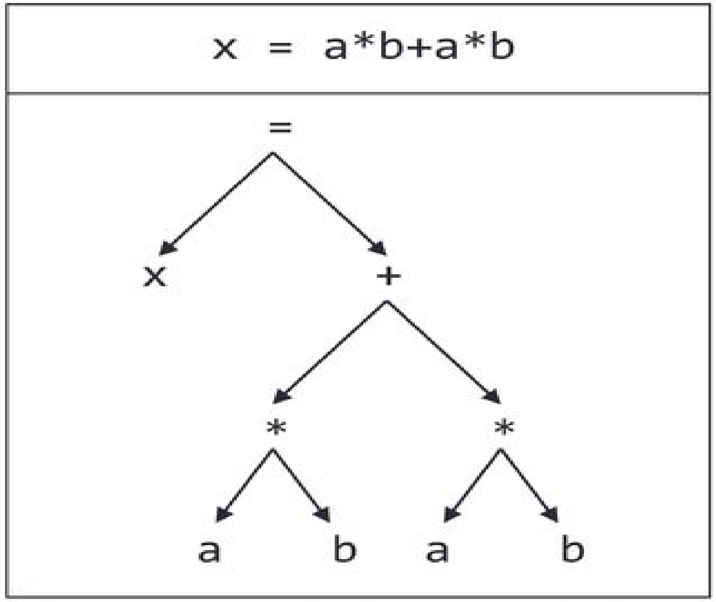
To entry for i

- Algorithm
  - Search the array for a node M with label op, left child l and right child r
  - If there is such a node, return the value number M
  - If not create in the array a new node N with label op, left child l, and right child r and return its value

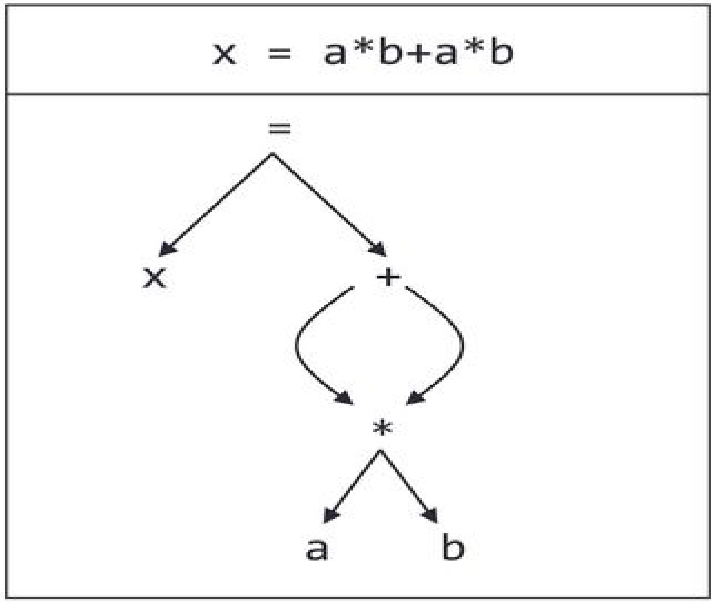
Abstract Syntax Tree vs DAG: Example 1

x = a * b + a * b

Abstract Syntax Tree

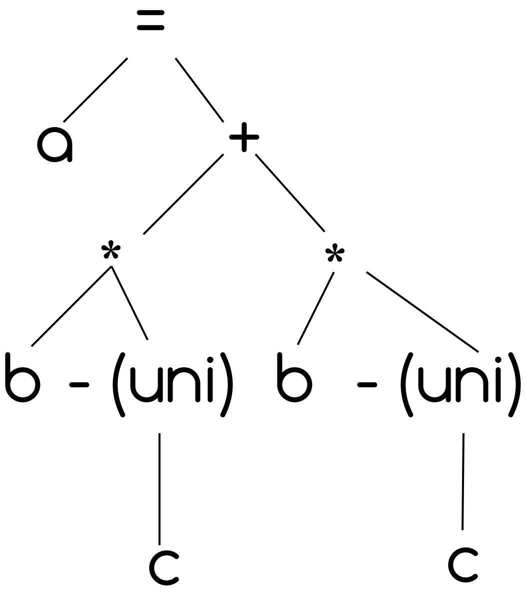


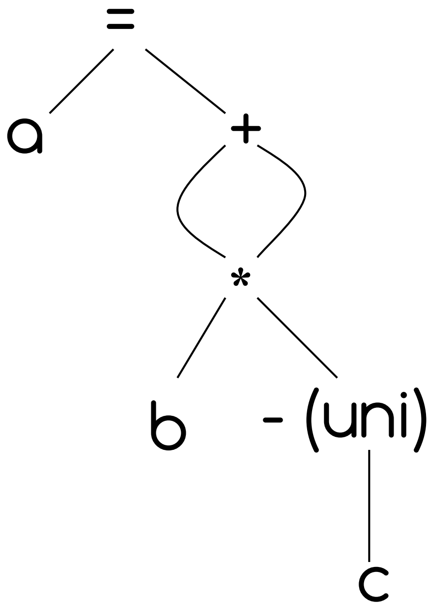
Directed Acyclic Graph



ASTs and DAGs: Example 2

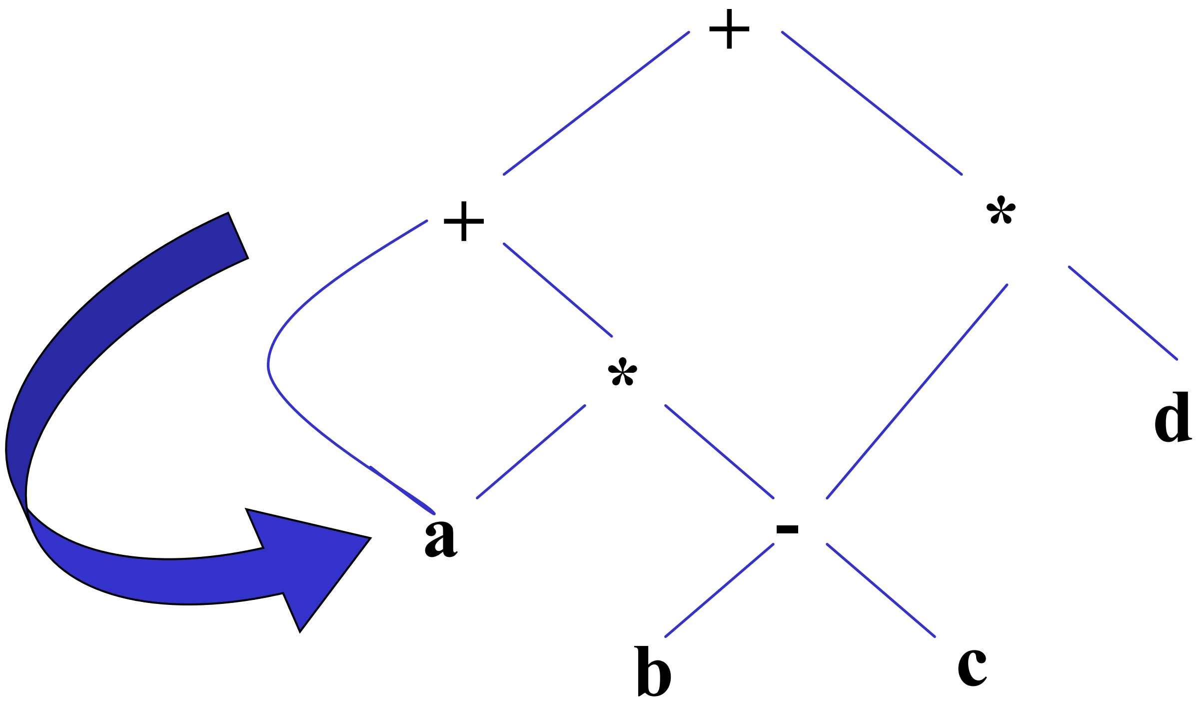
$$a = b * -c + b * -c$$





Directed Acyclic Graphs: Example

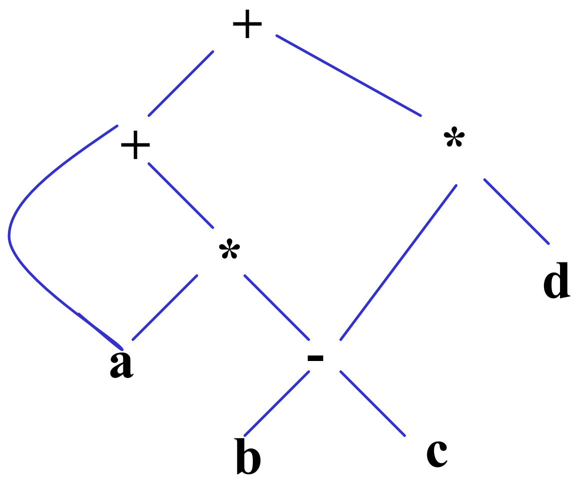
a+a*(b-c)+(b-c)*d



DAGs & Three address code

- In a three address code there is at most one operator at the right side of an instruction
- Example: a+a*(b-c)+(b-c)*d

DAG



Three address code

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

Summary

- 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.

Summary: Syntax Tree vs DAG

- A syntax tree is also known as parse tree.
  - It represents different categories of syntax used in the sentence.
  - It enables one to understand the different syntactical structure of a sentence.
- DAG is the Directed Acyclic Graph used in representing the basic block structure.
  - It is used in modeling probabilities, connectivity, and causality.
  - It uses unique node for each value.

Summary

- 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

Review Questions

Problem 1: $a+b*c-d/(b*c)$

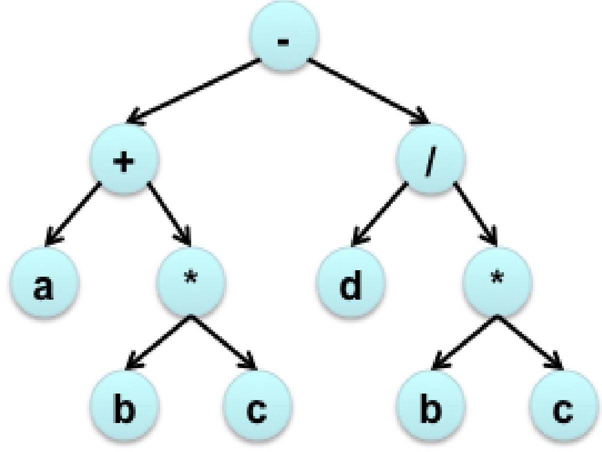
Translate this expression into the following representations:

- Abstract Syntax Tree
- Three-address code
- Postfix notation
- DAG

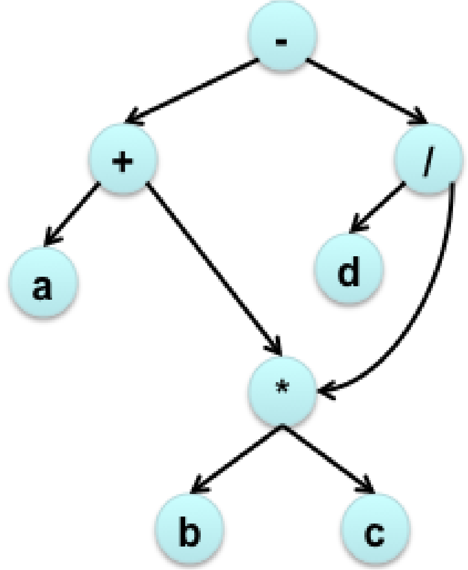
Problem 1: a+b*c-d/(b*c)

Translate this expression into the following representations:

- Abstract Syntax Tree
- Three-address code
- Postfix notation
- DAG



Syntax tree



DAG

Postfix notation: abc*+dbc*/-

Three Address code

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

Quadruples

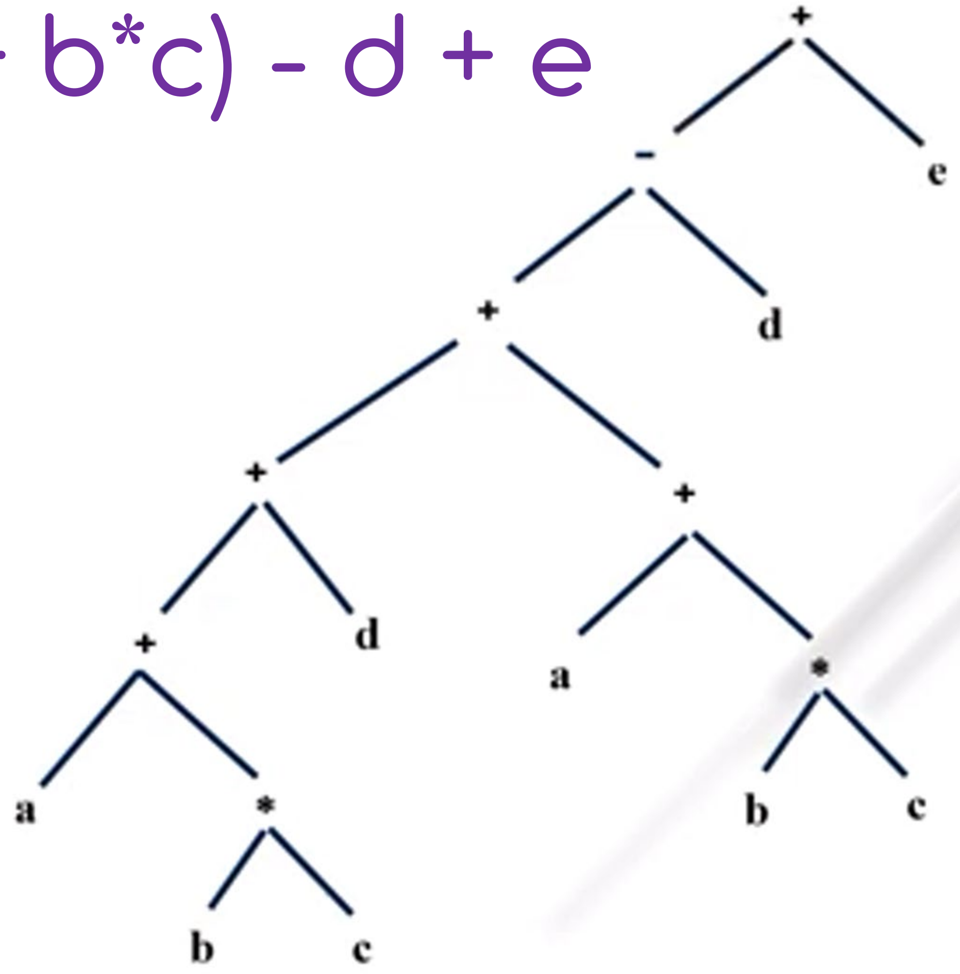
op	arg_1	arg_2	result
*	b	c	t1
+	a	t1	t2
*	b	c	t3
/	d	t3	t4
-	t2	t4	t5

Problem 2:
(a + b*c) + d + (a + b*c) - d + e

- Translate this expression into the following representations:
  - Abstract Syntax Tree
  - Three-address code
  - Postfix notation
  - DAG

Problem 2: Syntax Tree

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



Problem 2: Postfix Notation

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

(a + b * c) + d + (a + b * c) – d + e	where,
(a + T1) + d + (a + b * c) – d + e	T1 = bc*
T2 + d + (a + b * c) – d + e	T2 = aT1+
T3 + (a + b * c) – d + e	T3 = T2d+
T3 + (a + T4) – d + e	T4 = bc*
T3 + T5 – d + e	T5 = aT4+
T6 – d + e	T6 = T3T5+
T7 + e	T7 = T6d-
T8	T8 = T7e+

T8
T7 e +
T6 d - e +
T3 T5 + d – e +
T3 aT4 + + d – e +
T3 abc * + + d - e +
T2 d + abc * + + d – e +
aT1+ d + abc * + + d – e +
abc * + d + abc * + + d – e +

backward Substitution the value

of temporary variables

Postfix notatic

Problem 2: Three-Address Code

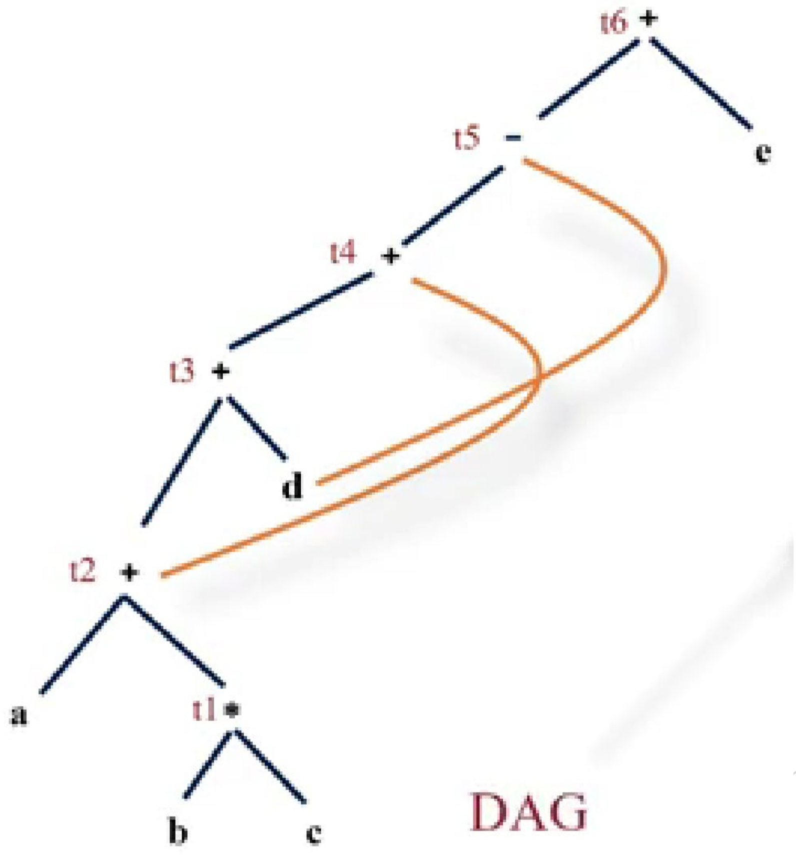
(a + b*c) + d + (a + b*c) - d + e

$t_1 = b * c$
$t_2 = a + t1$
$t_3 = t2 + d$
$t_4 = b * c$
$t_5 = a + t4$
$t_6 = t3 + t5$
$t_7 = t6 - d$
$t_8 = t7 + e$

Three - address Code

Problem 2: DAG

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



Problem 3:

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

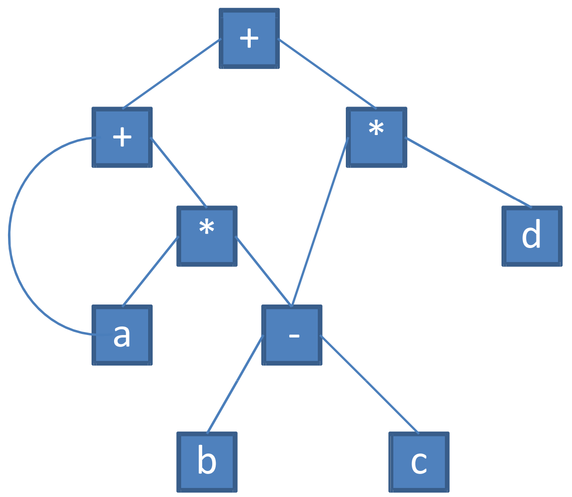
Translate this expression into:

- DAG
- Three Address Code - Quadraples

Problem 3: DAG & 3-address code

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

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



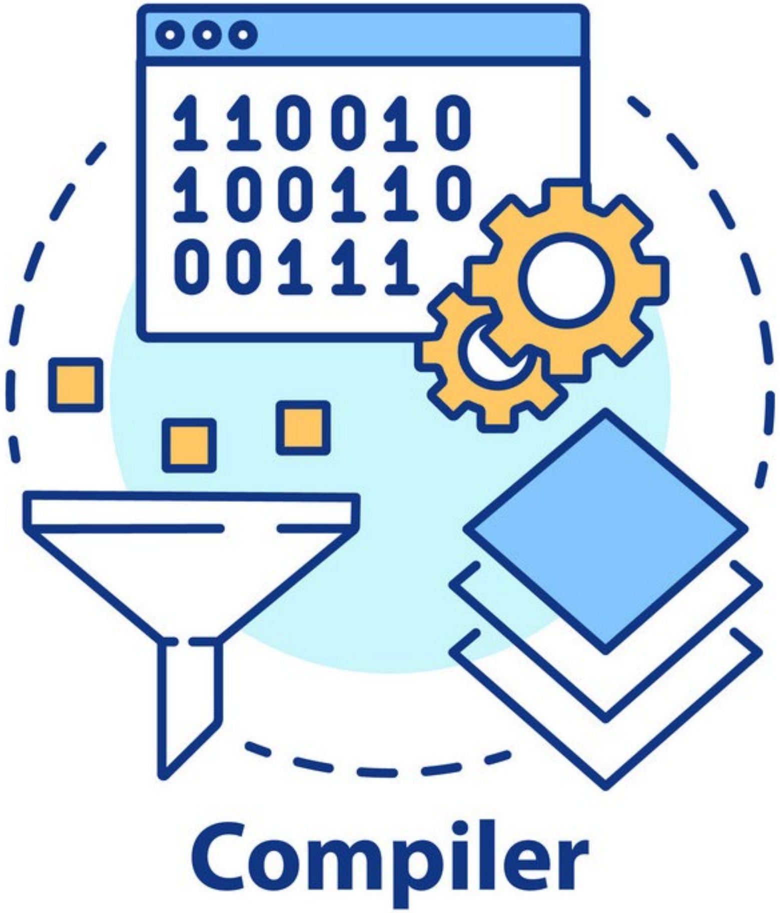
Problem 3: Quadruples

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

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

	Operator	Operand 1	Operand 2	Result
(1)	-	b	c	t1
(2)	*	a	t1	t2
(3)	+	a	t2	t3
(4)	*	t1	d	t4
(5)	+	t3	t4	t5


Q
A



Compilation Example

The Phases of a Compiler

Position := initial + rate * 60

↓

lexical analyzer

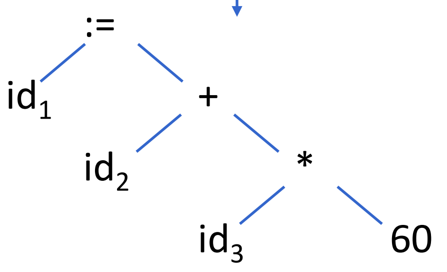
↓

$id_1 := id_2 + id_3 * 60$

↓

syntax analyzer

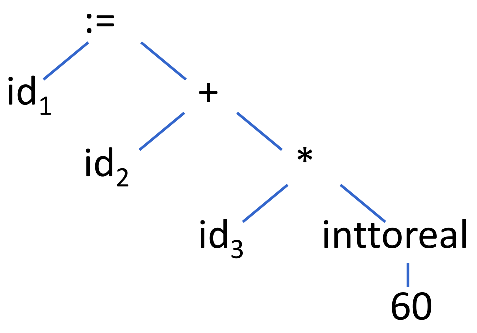
↓



↓

semantic analyzer

↓



↓

intermediate code generator

↓

temp1 := inttoreal (60)
temp2 := id_3 * temp1
temp3 := id_2 + temp2
id1 := temp3

↓

code optimizer

↓

temp1 := id_3 * 60.0
id1 := id_2 + temp1

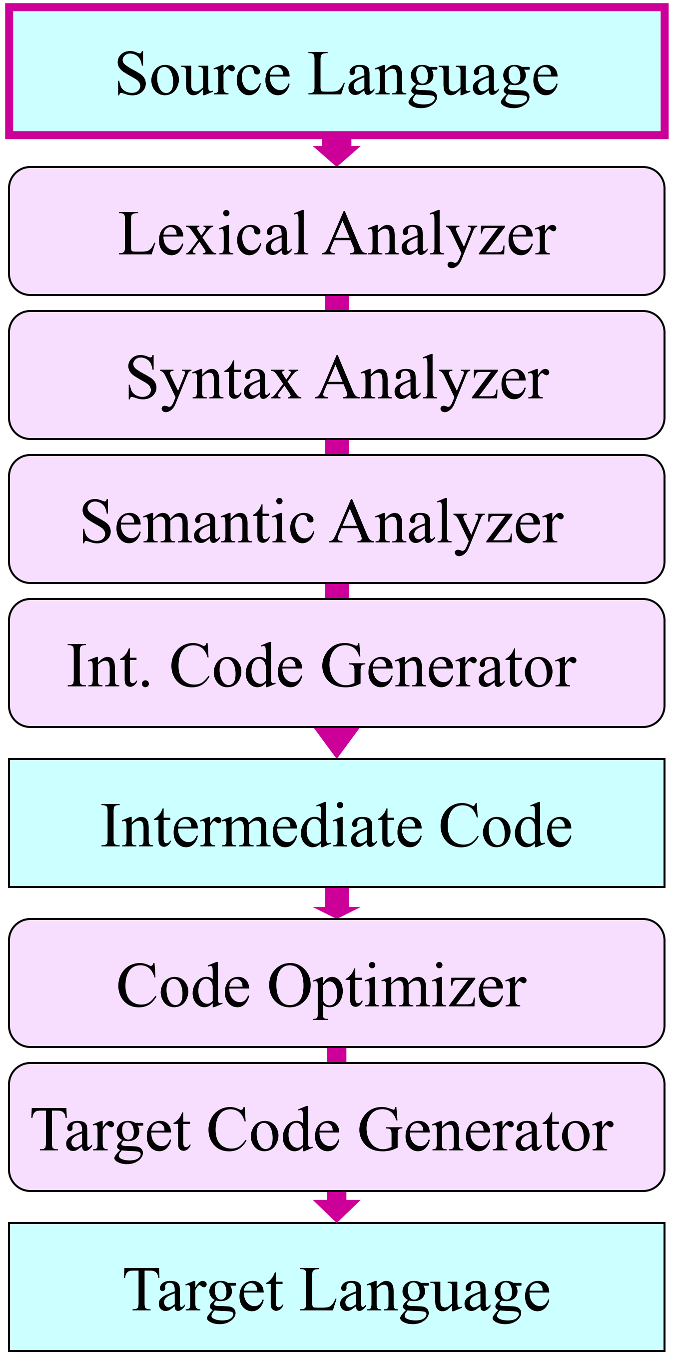
↓

code generator

↓

```
MOVF   id3,    R2
MULF   #60.0,  R2
MOVF   id2,    R1
ADDF   R2,     R1
MOVF   R1,     id1
```

Compilation Example



Source Code:

```
Position = Initial + Rate * 60
```

Compilation Example

Source Language
↓
Lexical Analyzer
↓
Syntax Analyzer
↓
Semantic Analyzer
↓
Int. Code Generator
↓
Intermediate Code
↓
Code Optimizer
↓
Target Code Generator
↓
Target Language

Source Code:

```
Position = Initial + Rate * 60
```

Lexical Analysis:

```
ID(1) ASSIGN ID(2) ADD ID(3) MULT INT(60)
```

Compilation Example

- Source Language
- Lexical Analyzer
- **Syntax Analyzer**
- Semantic Analyzer
- Int. Code Generator
- Intermediate Code
- Code Optimizer
- Target Code Generator
- Target Language

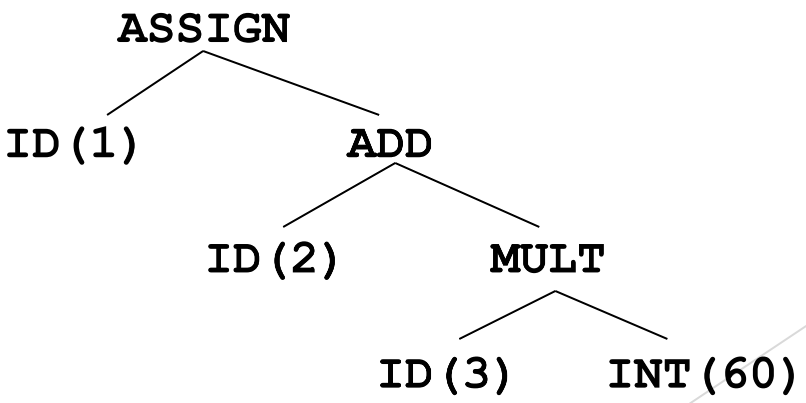
Source Code:

`Position = Initial + Rate * 60`

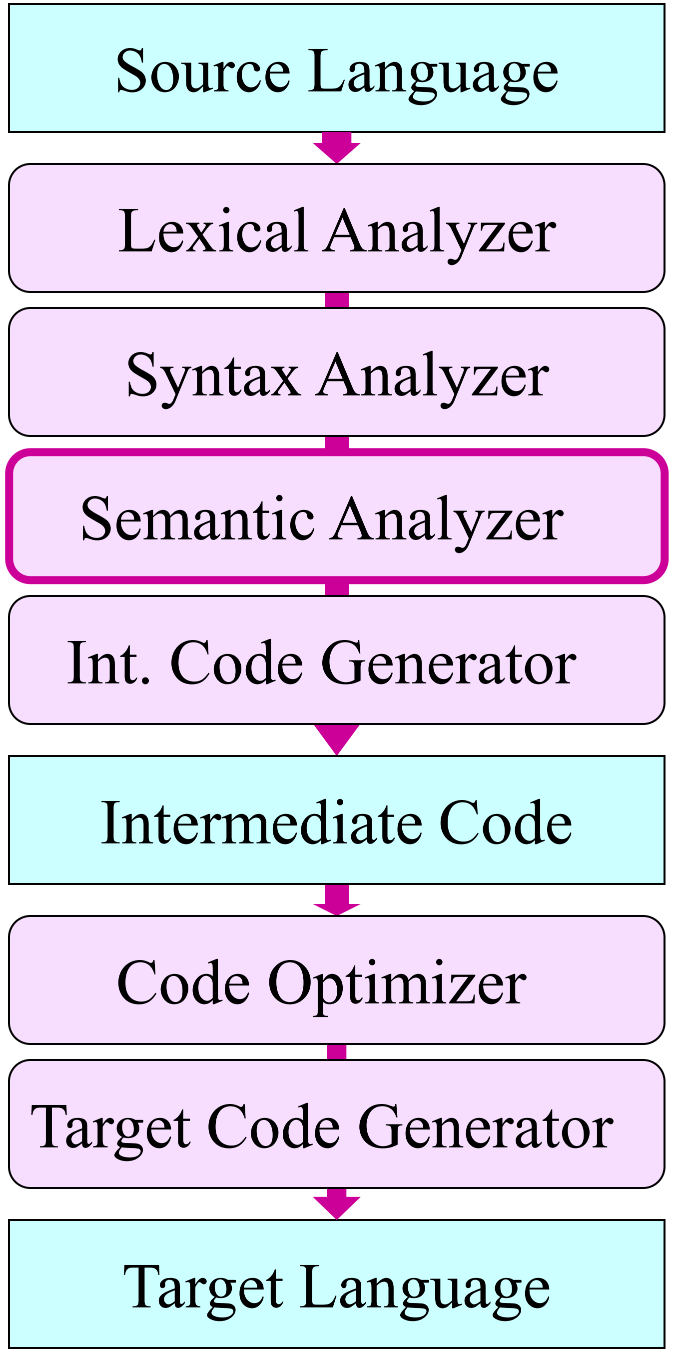
Lexical Analysis:

`ID(1) ASSIGN ID(2) ADD ID(3) MULT INT(60)`

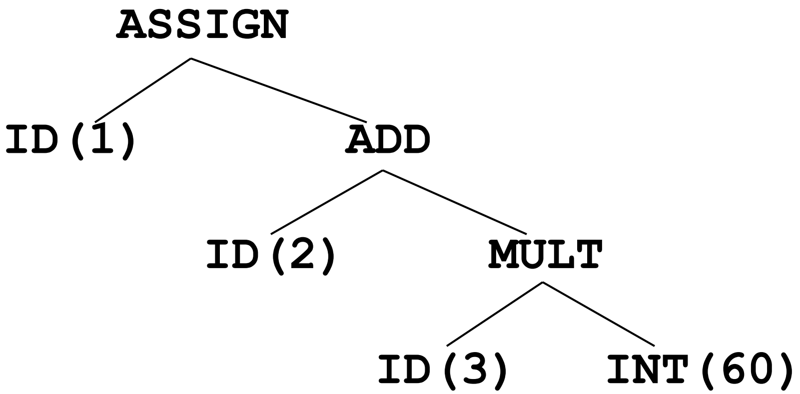
Syntax Analysis:



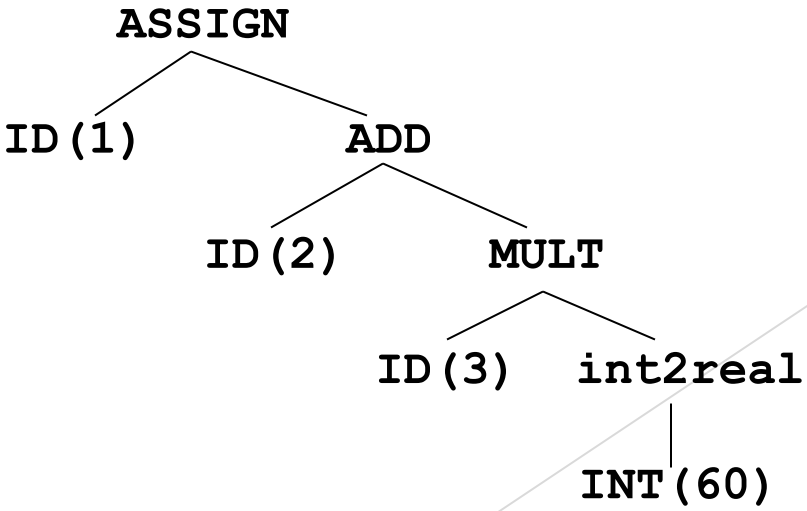
Compilation Example



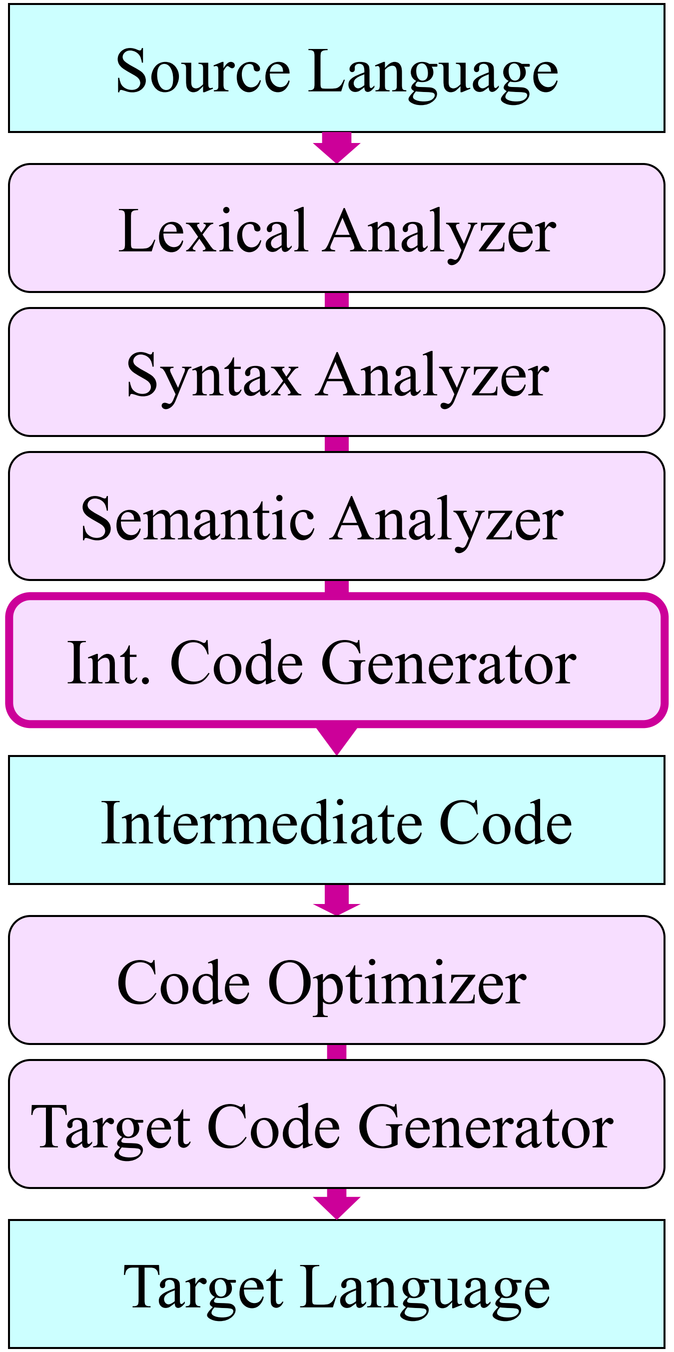
Syntax Analysis:



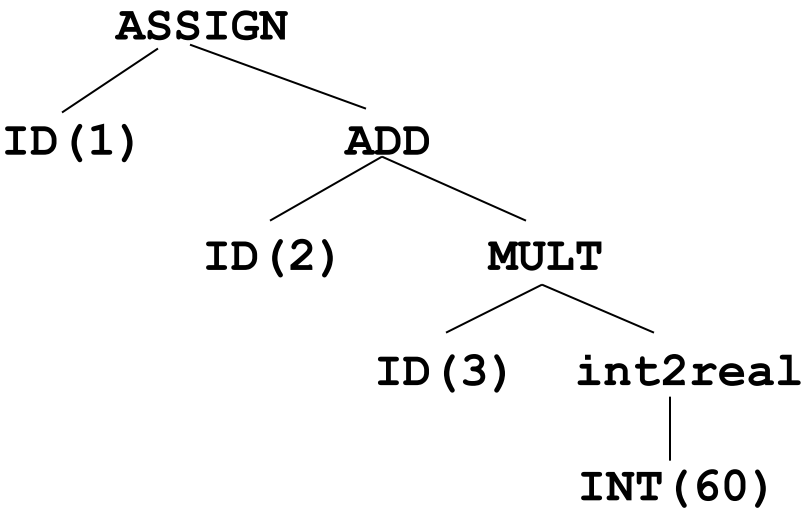
Sematic Analysis:



Compilation Example



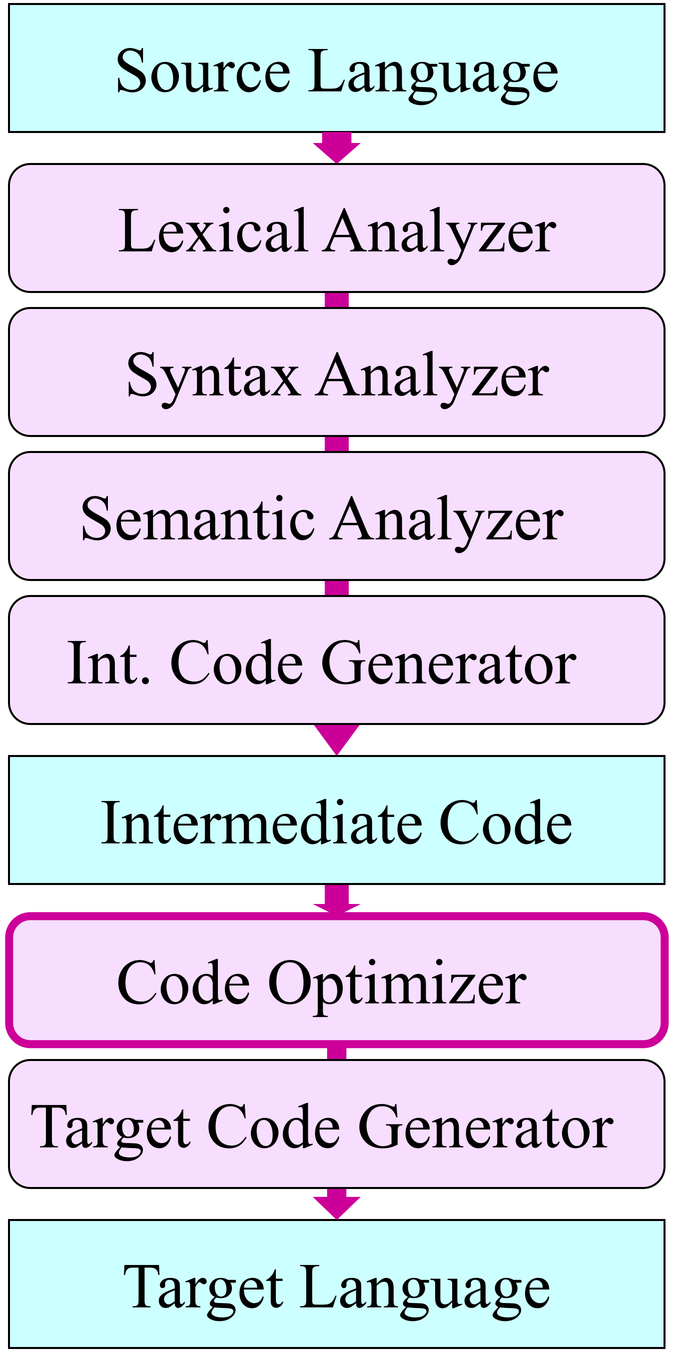
Sematic Analysis:



Intermediate Code:

```
temp1 = int2real(60)
temp2 = id3 * temp1
temp3 = id2 + temp2
id1 = temp3
```

Compilation Example



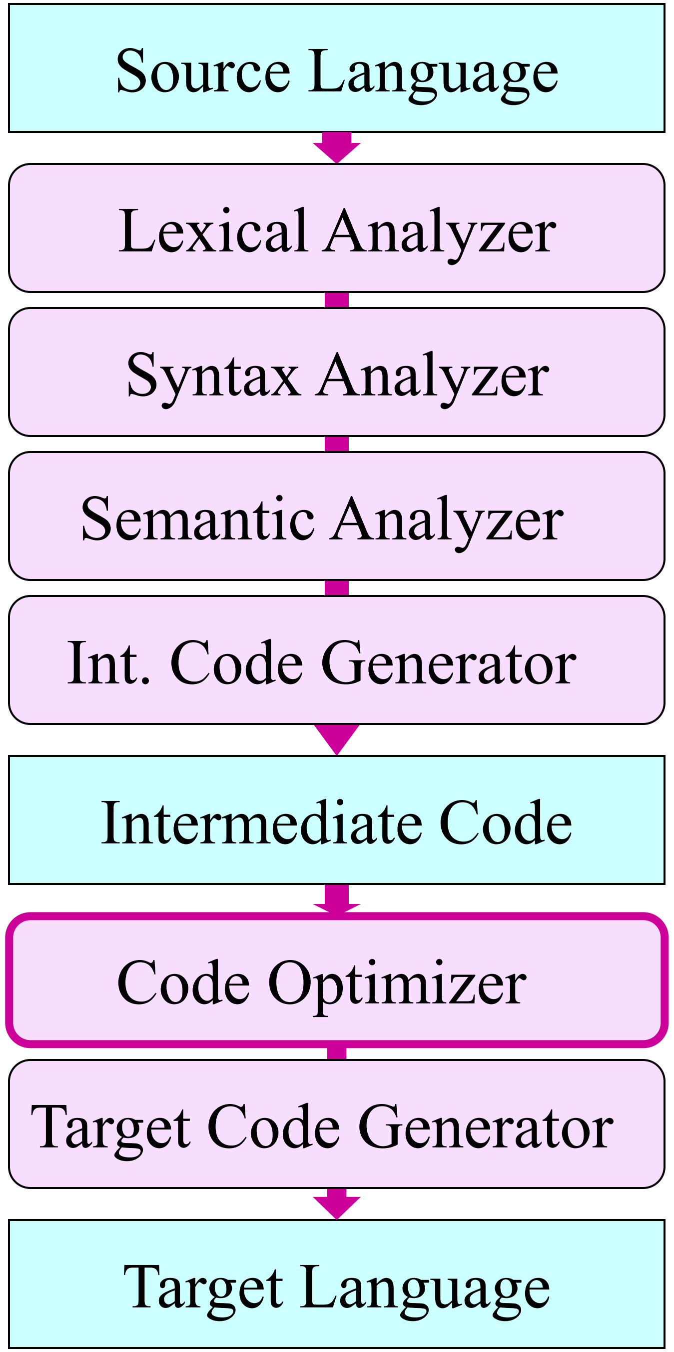
Intermediate Code:

```
temp1 = int2real(60)
temp2 = id3 * temp1
temp3 = id2 + temp2
id1 = temp3
```

Optimized Code (step 0):

```
temp1 = int2real(60)
temp2 = id3 * temp1
temp3 = id2 + temp2
id1 = temp3
```

Compilation Example



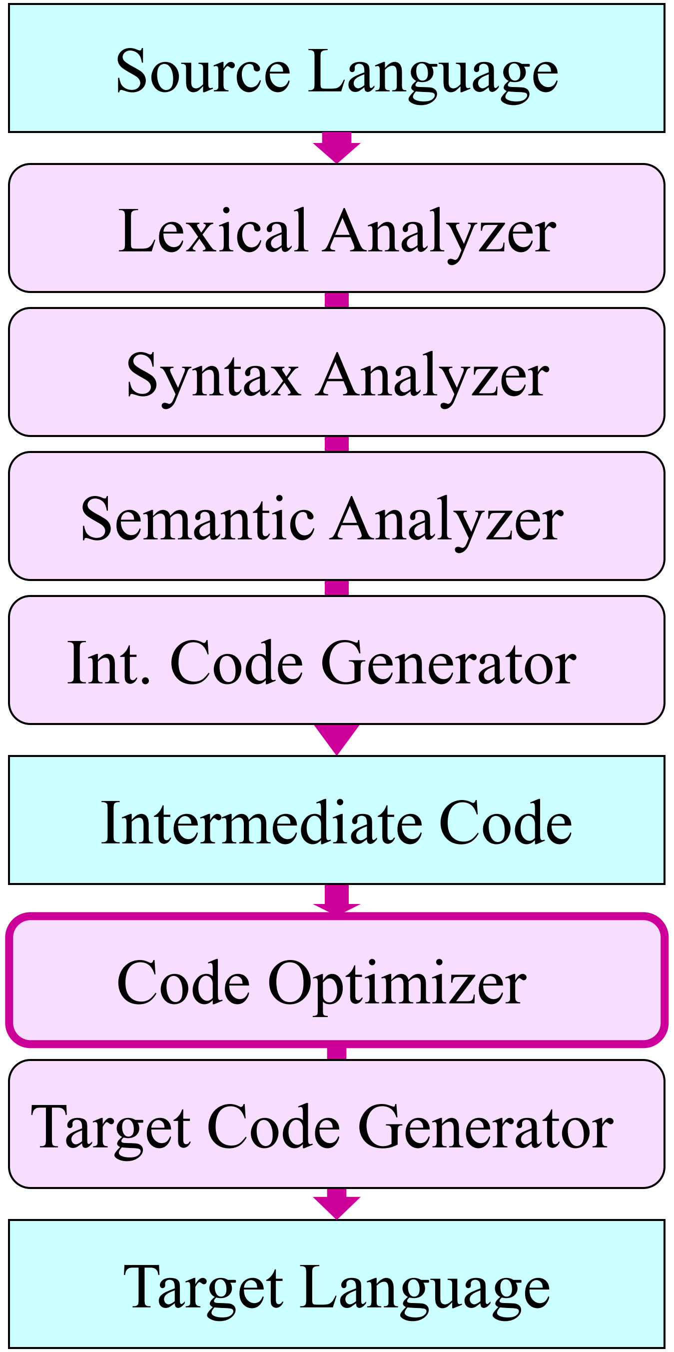
Intermediate Code:

```
temp1 = int2real(60)
temp2 = id3 * temp1
temp3 = id2 + temp2
id1 = temp3
```

Optimized Code (step 1):

```
temp1 = 60.0
temp2 = id3 * temp1
temp3 = id2 + temp2
id1 = temp3
```

Compilation Example



Intermediate Code:

```
temp1 = int2real(60)
temp2 = id3 * temp1
temp3 = id2 + temp2
id1 = temp3
```

Optimized Code (step 2):

```
temp2 = id3 * 60.0
temp3 = id2 + temp2
id1 = temp3
```

Compilation Example

Source Language
↓
Lexical Analyzer
↓
Syntax Analyzer
↓
Semantic Analyzer
↓
Int. Code Generator
↓
Intermediate Code
↓
Code Optimizer
↓
Target Code Generator
↓
Target Language

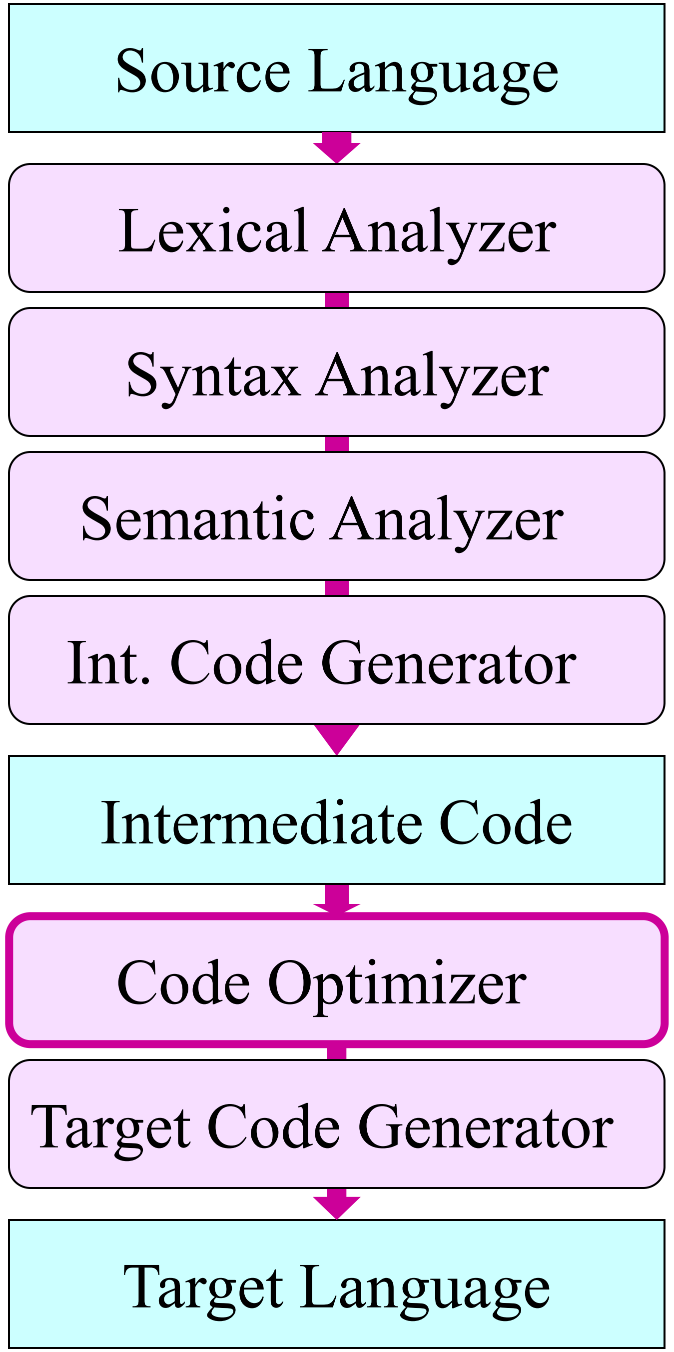
Intermediate Code:

```
temp1 = int2real(60)
temp2 = id3 * temp1
temp3 = id2 + temp2
id1 = temp3
```

Optimized Code (step 3):

```
temp2 = id3 * 60.0
id1 = id2 + temp2
```

Compilation Example



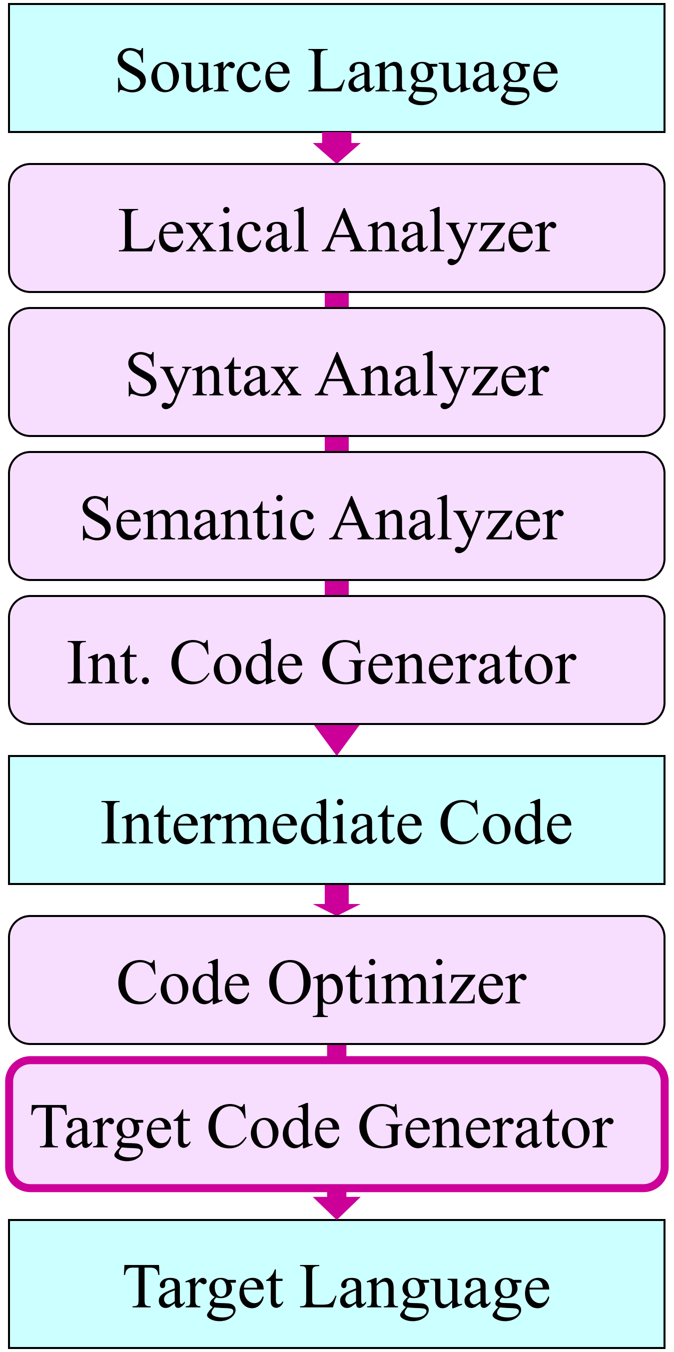
Intermediate Code:

```
temp1 = int2real(60)
temp2 = id3 * temp1
temp3 = id2 + temp2
id1 = temp3
```

Optimized Code:

```
temp1 = id3 * 60.0
id1 = id2 + temp1
```

Compilation Example



Intermediate Code:

```
temp1 = int2real(60)
temp2 = id3 * temp1
temp3 = id2 + temp2
id1 = temp3
```

Optimized Code:

```
temp1 = id3 * 60.0
id1 = id2 + temp1
```

Target Code:

```
MOVF id3, R2
MULF #60.0, R2
MOVF id2, R1
ADDF R2, R1
MOVF R1, id1
```

- End of Front-end
- End of Compilers Course

Useful links

Intermediate Code Generation

- https://www.youtube.com/watch?v=OFuJK7dBdo4
- https://www.youtube.com/watch?v=lqAG6xmvOUI
- https://www.youtube.com/watch?v=XeKnqo6UKw0&list=PLI22nRq-CejDnrs4rVANi6DXoFzqqUmhe&index=16