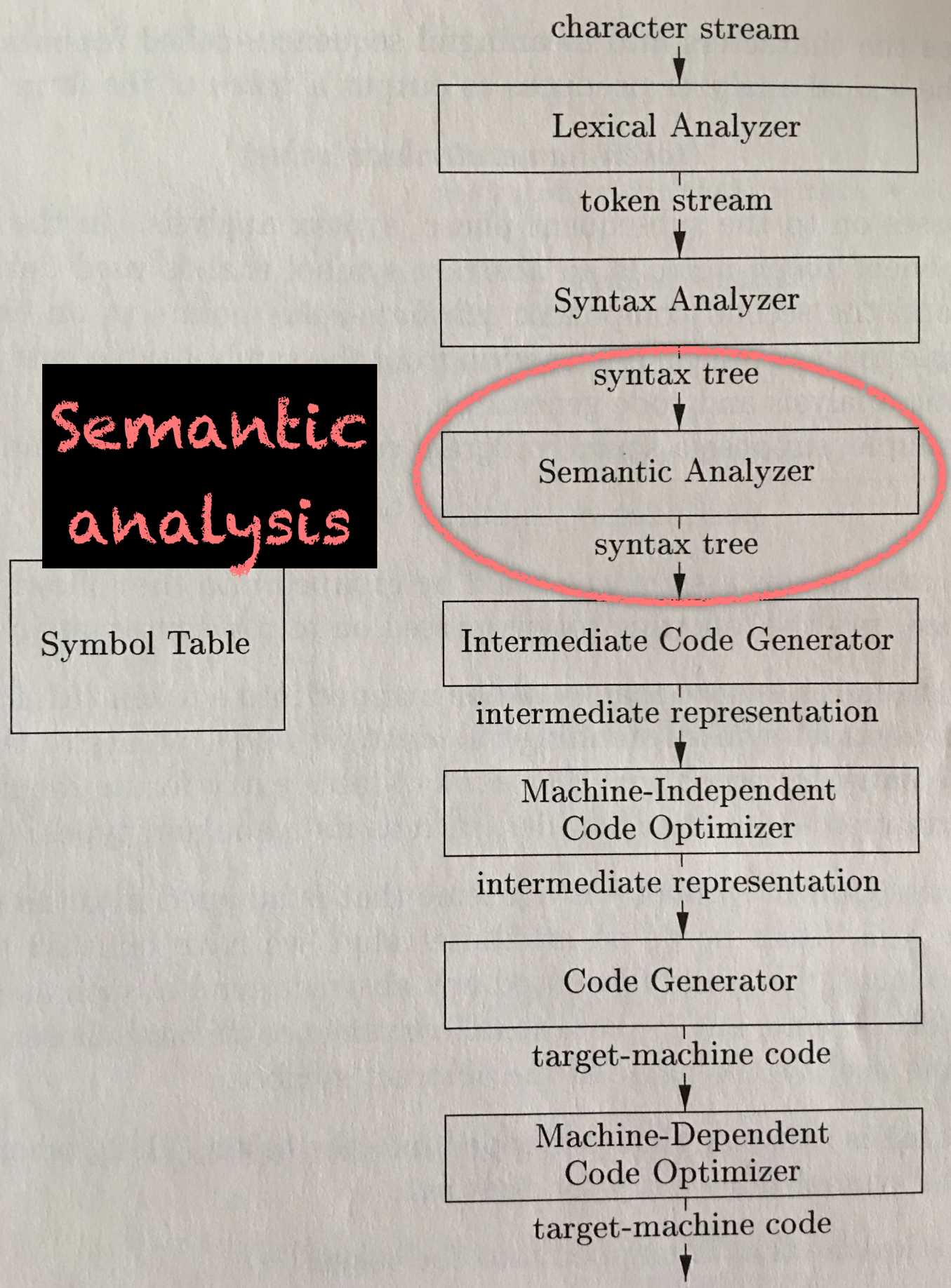
CSE443
Compilers

Dr. Carl Alphonce
alphonce@buffalo.edu
343 Davis Hall

Phases of a compiler

Figure 1.6, page 5 of text

Semantic analysis



Recursive records

A record type must allow a component to be of the same type as the type itself:

type Node: [integer datum ; Node rest]

Recursive records

A record type must allow a component to be of the same type as the type itself:

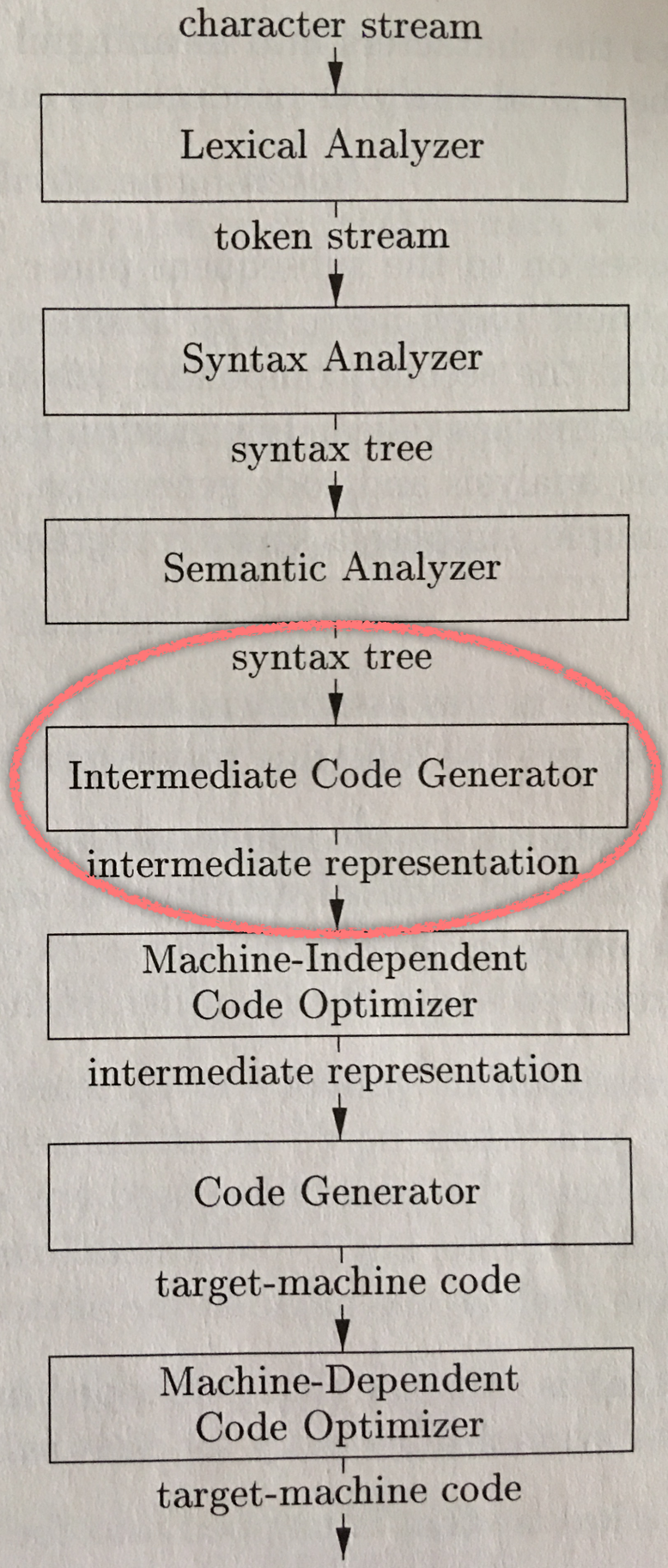
```
type Node: [ integer datum ; Node rest ]
```

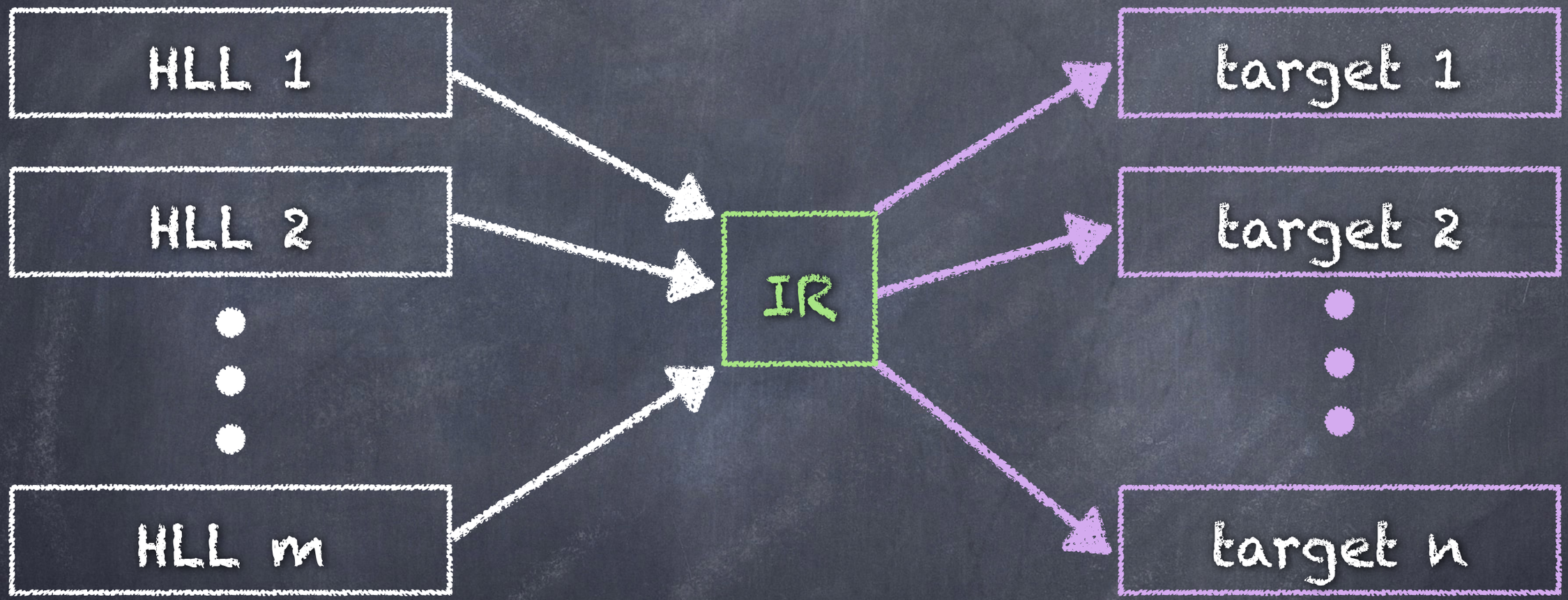
Be careful how you process declaration: you need to ensure that the second occurrence of Node does not trigger an undefined name

Phases of a compiler

Intermediate Representation (IR): specification and generation

Figure 1.6, page 5 of text




HLL 1
HLL 2
HLL m
IR
target 1
target 2
target n

machine independent optimizations

machine dependent optimizations

IR

IR

target 1

target 2

target n

HLL

Sprint 2 or 3

target 1

IR

machine
independent
optimizations

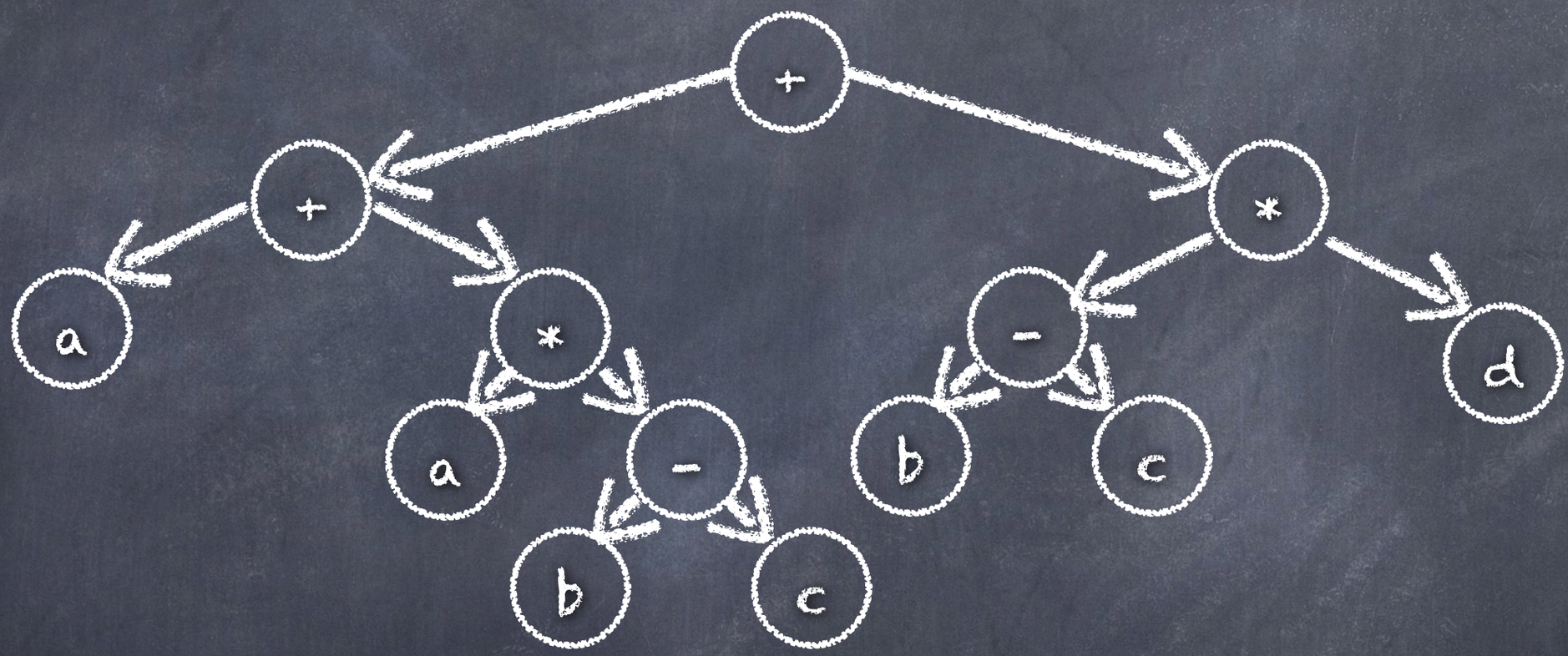
Sprint 2,
3, or 4

Milestone 4
(maybe some
simple ones)

Intermediate Representations

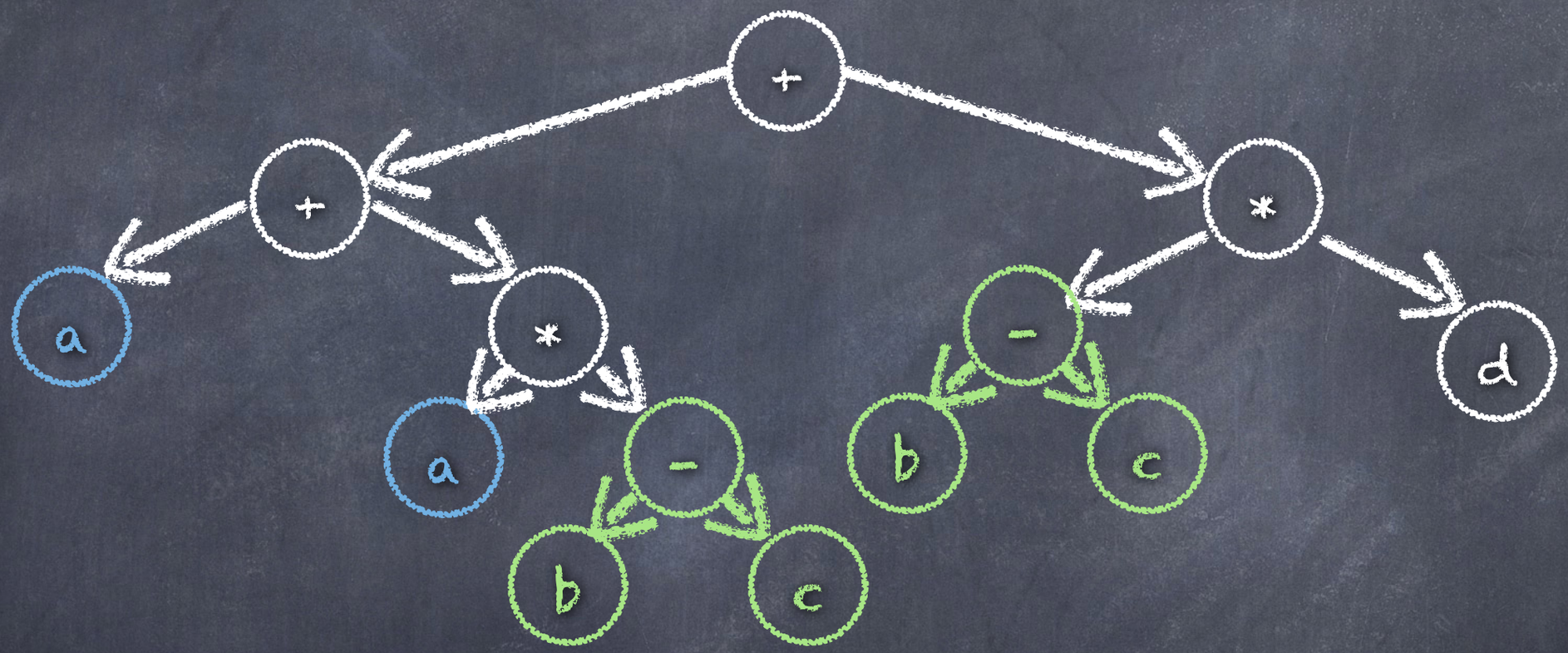
Ex. 6.1 [p 359]

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



Ex. 6.1 [p 359]

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

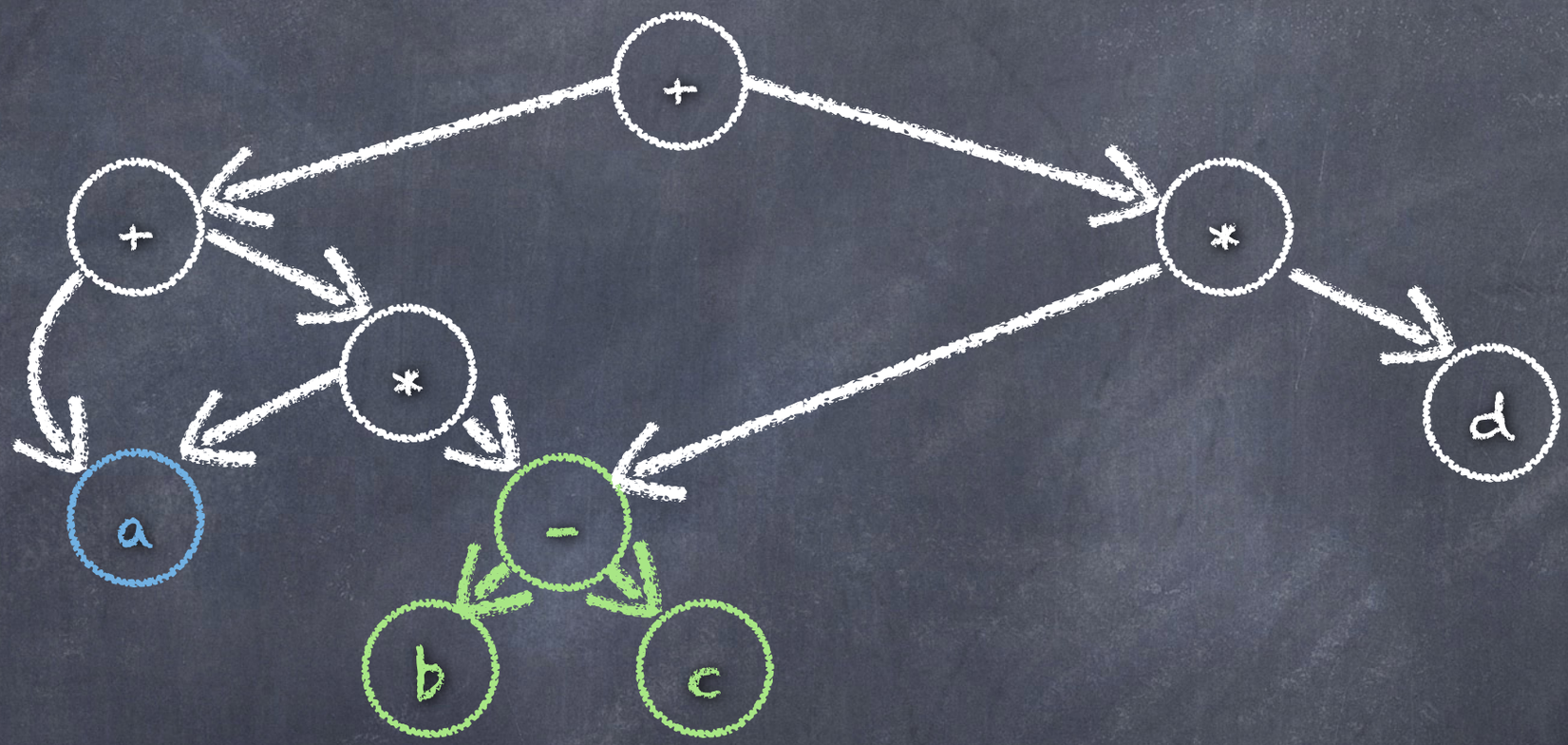


Directed Acyclic Graph (DAG)

- Similar to a syntax tree
- No repeated nodes: structure sharing

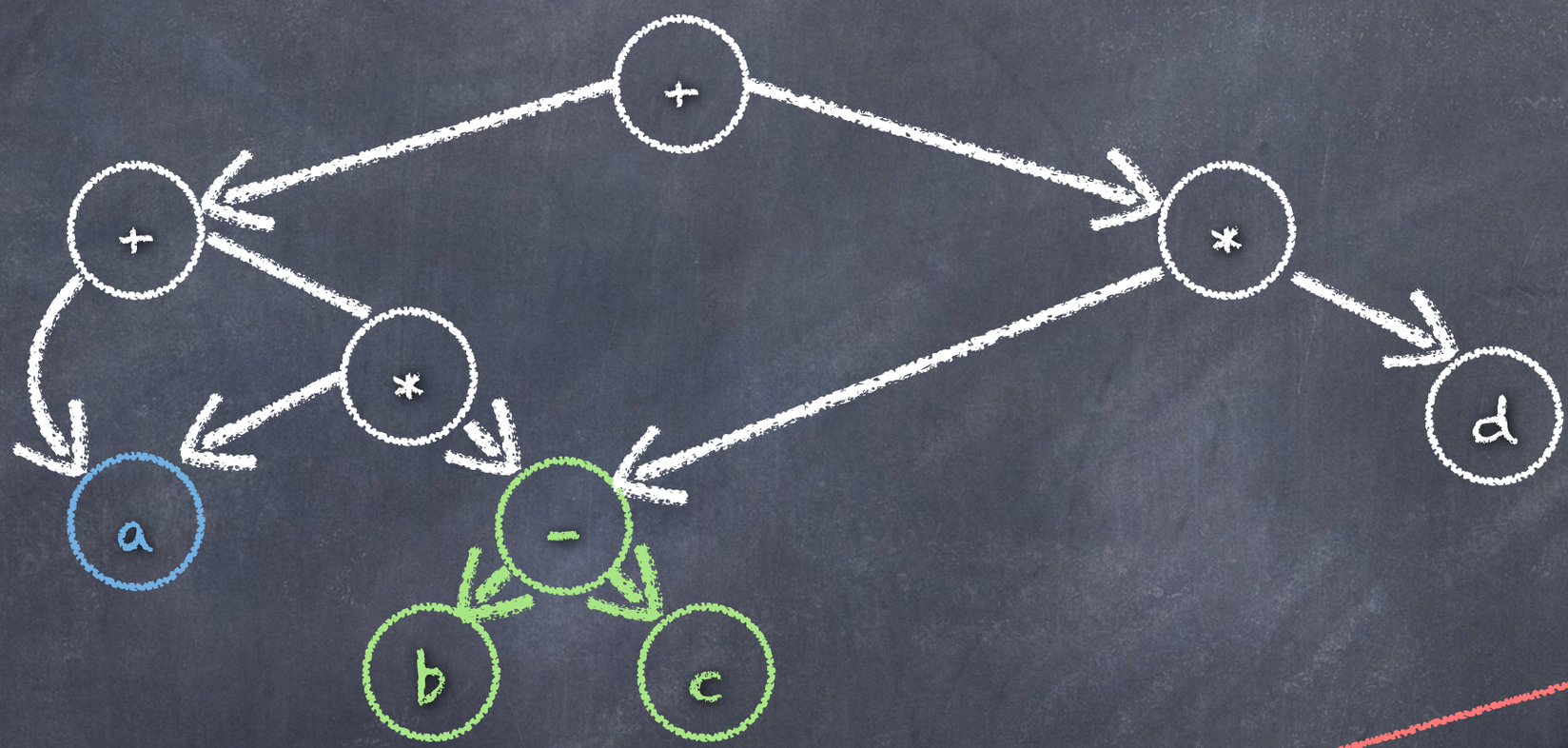
Ex. 6.1 [p 359]

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



Ex. 6.1 [p 359]

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



Things can be more complicated if expressions have side effects

SDT
Tree or DAG

	Production	Semantic Rule
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	$E \rightarrow E_1 * T$	E.node = new Node('*', E_1.node, T.node)
4	E -> T	E.node = T.node
5	T -> (E)	T.node = E.node
6	T -> id	T.node = new Leaf(id, id.entry)
7	T -> num	T.node = new Leaf(num, num.val)

Figure 6.4 in text (p. 360), corrected according to errata sheet.

SDT
Tree or DAG

- SDT produces a tree if each call to Node creates a new tree node.
- SDT produces a DAG if for each call to Node there is a check whether this node already exists, and if so it returns a reference to the existing node rather than returning a new node.

Example

p_1 =	Leaf(id, entry-a)
p_2 =	Leaf(id, entry-a) = p_1
p_3 =	Leaf(id, entry-b)
p_4 =	Leaf(id, entry-c)
p_5 =	Node('-',p_3,p_4)
p_6 =	Node('*',p_1,p_5)
p_7 =	Node('+',p_1,p_6)
p_8 =	Leaf(id, entry-b) = p_3
p_9 =	Leaf(id, entry-c) = p_4
p_{10} =	Node('-',p_3,p_4) = p_5
p_{11} =	Leaf(id, entry-d)
p_{12} =	Node('*',p_5,p_{11})
p_{13} =	Node('+',p_7,p_{12})

Value-number method
Algorithm 6.3 [p. 361]

- Input: label op, node l, node r
- Output: The value number of a node in the array with signature <op,l,r>
- Method: Search the array for a node M with signature <op,l,r>. If there is such a node, return the value number of M. If not, create in the array a new node N with signature <op,l,r> and return its value number.

Value-number method
Algorithm 6.3 [p. 361]

- Input: label op, node l, node r
- Output: The value number of a node in the array with signature <op,l,r>

Can use hash table for efficiency.

- Method: Search the array for a node M with signature <op,l,r>. If there is such a node, return the value number of M. If not, create in the array a new node N with signature <op,l,r> and return its value number.

Revisiting 6.1
see construction steps in figure 6.5 [p. 360]

1	id	-> to ST entry for a	
2	id	-> to ST entry for b	
3	id	-> to ST entry for c	
4	-	2	3
5	*	1	4
6	+	1	5
7	id	-> to ST entry for d	
8	*	4	7
9	+	6	8