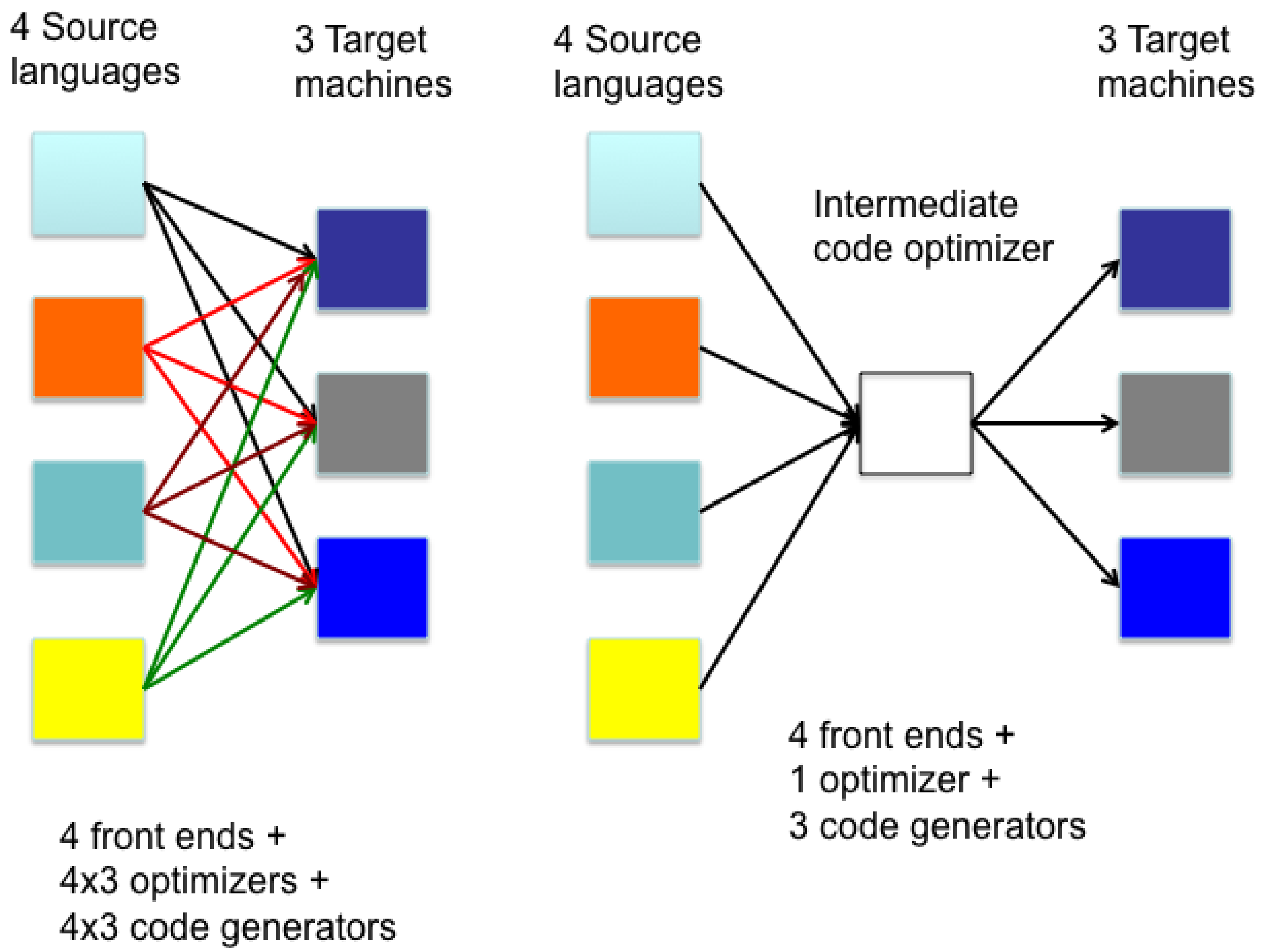
Intermediate Code Generation

Why Intermediate Code?



Without Intermediate Code Optimizer

Without intermediate code generator -

- –For **m** languages and **n** target machines, we need to write **m** front-ends, **mxn** optimizers, and **mxn** code generators.
- –Reuse is not possible.

With intermediate code generator -

- –A machine independent code optimizer can be written
- –**m** front ends, **n** code generators and 1 optimizer

Three Address Code

- Simple instructions
- LHS is the target and RHS has at most two operands and one operator
- RHS operands can be either variables or constants

Example:
a+b*c-d/(b*c)

3-address code

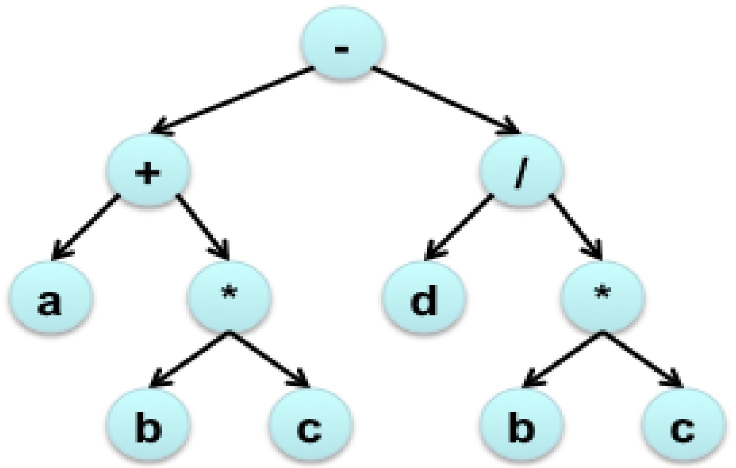
```
1  t1 = b*c
2  t2 = a+t1
3  t3 = b*c
4  t4 = d/t3
5  t5 = t2-t4
```

Quadruples

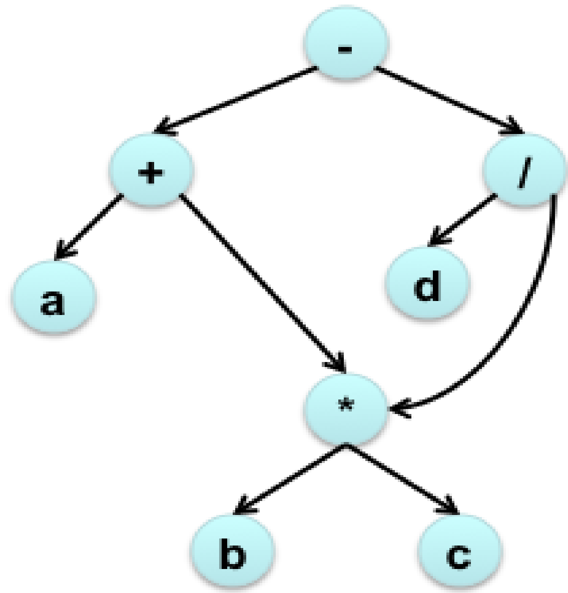
op	arg_1	arg_2	result
*	b	c	t1
+	a	t1	t2
*	b	c	t3
/	d	t3	t4
-	t2	t4	t5

Triples

	op	arg_1	arg_2
0	*	b	c
1	+	a	(0)
2	*	b	c
3	/	d	(2)
4	-	(1)	(3)



Syntax tree



DAG

Three Address Code

- **Assignment instructions:**

a = b biop c

a = uop b

  - biop is any binary arithmetic, logical, or relational operator
  - uop is any unary arithmetic (-, shift, conversion) or logical operator (~)

- **Copy instruction:**

a = b

- **Jump instructions:**

goto L

if t goto L

- L is the label of the next three-address instruction to be executed, t is a boolean variable, a and b are either variables or constants

Three Address Code

- **Functions:**

func begin <name> //beginning of the function

func end //end of a function

param p //place a value parameter p on stack

refparam p //place a reference parameter p on stack

call f, n //call a function f with n parameters

return //return from a function

return a //return from a function with a value a

Three Address Code

- <u>Indexed copy instructions:</u>

a = b[i] //a is set to contents(contents(b)+contents(i))

a[i] = b //i th location of array a is set to b

- <u>Pointer assignments:</u>

a = &b //a is set to the address of b, i.e., a points to b

*a = b //contents(contents(a)) is set to contents(b)

a = *b //a is set to contents(contents(b))

Three Address Code

- C-Program

```
int a[10], b[10], dot_prod, i;
dot_prod = 0;
for (i=0; i<10; i++)
        dot_prod += a[i]*b[i];
```

```
    dot_prod = 0;
    i = 0;
L1: T1 = i>10
    T2 =  i= =10
    T3 = T1 && T2
    if T3 goto L2
    T4 = addr(a)
    T5 = i*4
    T6 = T4[T5]
    T7 = addr(b)
    T8 = i*4
    T9 = T7[T8]
    T10 = T6*T9
    T11 = dot_prod+T10
    dot_prod = T11
    T12 = i+1
    i = T12
    goto L1
L2:
```

Single-Static-Assignment (SSA)

- Another intermediate representation
- Facilitates certain code optimizations
- All assignments are to variables with distinct names

(a) Three-address code.	(b) Static single-assignment form.
$p = a + b$	$p_1 = a + b$
$q = p - c$	$q_1 = p_1 - c$
$p = q * d$	$p_2 = q_1 * d$
$p = e - p$	$p_3 = e - p_2$
$q = p + q$	$q_2 = p_3 + q_1$

Translations of Statements and Expressions

- Syntax-Directed Definition (SDD)
- Syntax-Directed Translation (SDT)

Three Address Code for Expressions

PRODUCTION	SEMANTIC RULES
$S \rightarrow \mathbf{id} = E\ ;$	$S.code = E.code \;\|\|$ $gen(top.get(\mathbf{id}.lexeme)\ '='\ E.addr)$
$E \rightarrow E_1 + E_2$	$E.addr = \mathbf{new}\ Temp()$ $E.code = E_1.code \;\|\|\; E_2.code \;\|\|$ $gen(E.addr\ '='\ E_1.addr\ '+'\ E_2.addr)$
$\mid\ - E_1$	$E.addr = \mathbf{new}\ Temp()$ $E.code = E_1.code \;\|\|$ $gen(E.addr\ '='\ '\mathbf{minus}'\ E_1.addr)$
$\mid\ (\ E_1\ )$	$E.addr = E_1.addr$ $E.code = E_1.code$
$\mid\ \mathbf{id}$	$E.addr = top.get(\mathbf{id}.lexeme)$ $E.code = ''$

Stores address of E
(e.g. temp variable etc.)

Stores three-address code
For E

PRODUCTION	SEMANTIC RULES
$S \rightarrow \textbf{id} = E \; ;$	$S.code = E.code \;\|\|$ $\quad gen(top.get(\textbf{id}.lexeme) \; '=' \; E.addr)$
$E \rightarrow E_1 + E_2$	$E.addr = \textbf{new}\; Temp\,()$ $E.code = E_1.code \;\|\|\; E_2.code \;\|\|$ $\quad gen(E.addr \; '=' \; E_1.addr \; '+' \; E_2.addr)$
$\quad \mid \; - E_1$	$E.addr = \textbf{new}\; Temp\,()$ $E.code = E_1.code \;\|\|$ $\quad gen(E.addr \; '=' \; '\textbf{minus}' \; E_1.addr)$
$\quad \mid \; (\, E_1 \,)$	$E.addr = E_1.addr$ $E.code = E_1.code$
$\quad \mid \; \textbf{id}$	$E.addr = top.get(\textbf{id}.lexeme)$ $E.code = ''$

Builds an instruction

Returns a temporary variable

Current Symbol Table

An example: a = b + c + (d)

Top of stack

Stack	Production Rule	Semantic Rules	Action	'gen' Output
id = id	*E -> id*	$E.addr = top.get(\mathbf{id}.lexeme)$ $E.code = ''$	*E.addr = b*	
id = E_1 *+ id*	*E -> id*	$E.addr = top.get(\mathbf{id}.lexeme)$ $E.code = ''$	*E.addr = c*	
id = $E_1 + E_2$	$E \to E_1 + E_2$	$E.addr = \mathbf{new}\ Temp()$ $E.code = E_1.code \parallel E_2.code \parallel gen(E.addr\ '='\ E_1.addr\ '+'\ E_2.addr)$	*E.addr = t1*	*t1 = b + c*
id = E + (id	*E -> id*	$E.addr = top.get(\mathbf{id}.lexeme)$ $E.code = ''$	*E.addr = d*	
id = E + (E_1 *)*	*E -> (* E_1 *)*	$E.addr = E_1.addr$ $E.code = E_1.code$	*E.addr = d*	*t1 = b + c*

An example: a = b + c + (d)

Top of stack

Stack	Production Rule	Semantic Rules	Action	'gen' Output
id = id	E -> id	$E.addr = top.get(\mathbf{id}.lexeme)$ $E.code = ''$	E.addr = b	
id = E_1 + id b	E -> id	$E.addr = top.get(\mathbf{id}.lexeme)$ $E.code = ''$	E.addr = c	
id = E_1 + E_2 c	E -> E_1 + E_2	$E.addr = \mathbf{new}\ Temp()$ $E.code = E_1.code \parallel E_2.code \parallel$ $gen(E.addr\ '='\ E_1.addr\ '+'\ E_2.addr)$	E.addr = t1	t1 = b + c
id = E + (id t1	E -> id	$E.addr = top.get(\mathbf{id}.lexeme)$ $E.code = ''$	E.addr = d	
id = E + (E_1) t1 d	E -> (E_1)	$E.addr = E_1.addr$ $E.code = E_1.code$	E.addr = d	t1 = b + c

An example: a = b + c + (d)

Top of stack

Stack	*Production Rule*	*Semantic Rules*	*Action*	*'gen' Output*
$id = E_1 + E_2$	$E \rightarrow E_1 + E_2$	$E.addr = \textbf{new}\ Temp()$ $E.code = E_1.code \mid\mid E_2.code \mid\mid$ $gen(E.addr\ '='\ E_1.addr\ '+'\ E_2.addr)$	*E.addr = t2*	*t2 = t1 + d*
$id = E$	$S \rightarrow id = E$	$S.code = E.code \mid\mid$ $gen(top.get(\textbf{id}.lexeme)\ '='\ E.addr)$	*top.get returns a*	*a = t2*

Final Code

t1 = b + c
t2 = t1 + d
a = t2

Incremental Translation

$$
\begin{array}{llll}
S & \rightarrow & \textbf{id} = E\ ; & \{\ gen(\ top.get(\textbf{id}.lexeme)\ '='\ E.addr);\ \} \\
E & \rightarrow & E_1 + E_2 & \{\ E.addr = \textbf{new}\ Temp\,(); \\
 & & & \quad gen(E.addr\ '='\ E_1.addr\ '+'\ E_2.addr);\ \} \\
 & | & -\ E_1 & \{\ E.addr = \textbf{new}\ Temp\,(); \\
 & & & \quad gen(E.addr\ '='\ '\textbf{minus}'\ E_1.addr);\ \} \\
 & | & (\ E_1\ ) & \{\ E.addr = E_1.addr;\ \} \\
 & | & \textbf{id} & \{\ E.addr = top.get(\textbf{id}.lexeme);\ \}
\end{array}
$$

gen() does two things:

1. generate three address instruction
2. append it to the sequence of instructions generated so far

Three Address Code Generation

- Exercise:

1. Change the semantic rules to generate three-address codes for arithmetic expressions (binary operators + and -) involving one-dimensional array variables on the right hand side.
    - Note that no array element will appear on the left hand side of an expression.
    - All variables used are integers of 4 byte length.

Write a program in any of your preferred language to generate the three-address code.