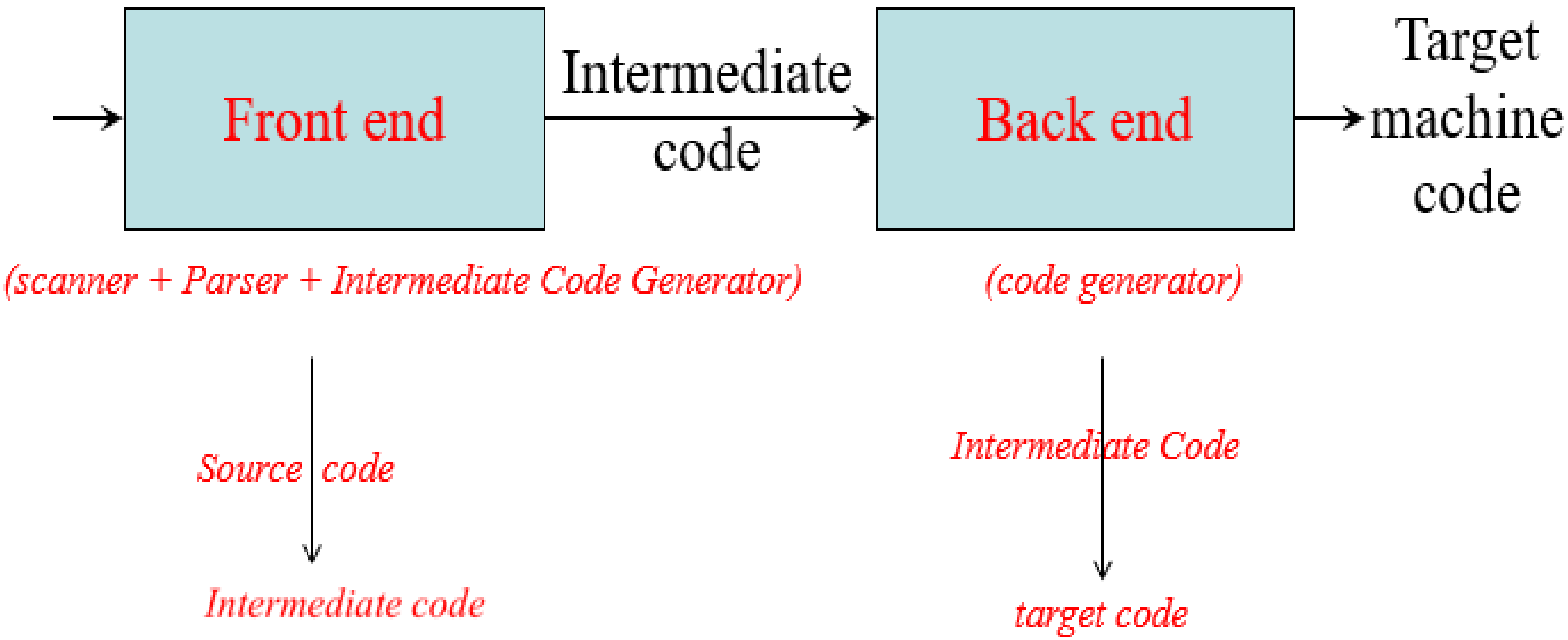
Intermediate Code Generation [Chapter 6]

Lecture 15

Edited by Dr. Ismail Hababeh
German Jordanian University
Adapted from slides by Dr. Robert A. Engelen

Intermediate Code Generation



Intermediate Code Generator

- Intermediate code generator receives input from its previous phase (semantic analyzer) in the form of Annotated Syntax Tree.
- Annotated Syntax Tree converted into a linear representation, e.g., postfix notation.
- Intermediate code tends to be machine independent code.

Intermediate Code Generation

- We build one front-end for the language and then we create a back-end for each machine.
- Simplifies *retargeting* enables attaching back-end for the new machine to an existing front-end.
- Enables machine-independent code optimization

Intermediate Code Representations

- *Postfix notation*: operations on values stored on operand stack (similar to JVM bytecode)
- *Three-address code*: (a code having at most three addresses in a line)

  $x = y$ op z
- *Two-address code*:

  x op= y

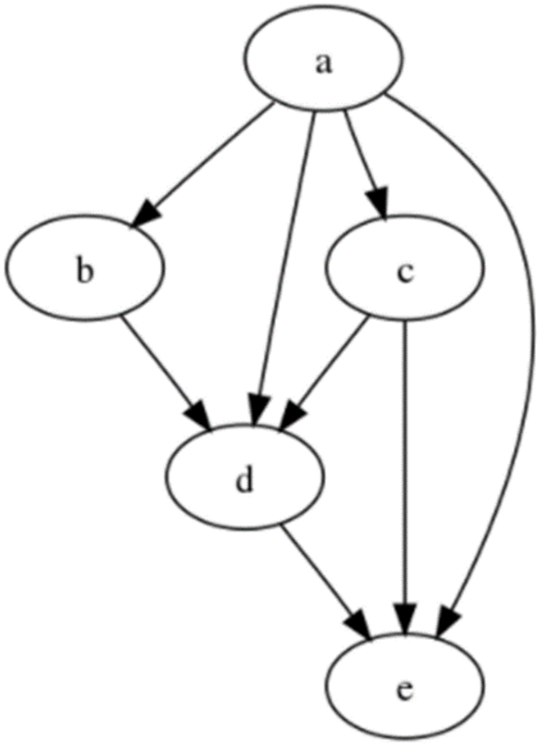
  which is the same as $x = x$ op y

Example: $x{+}{=}1$

Intermediate Code Representations

- *Graphical representations*

*A directed acyclic graph (DAG) is a directed graph with no directed cycles. That is, it consists of nodes and edges, where each edge directed from one node to another, such that those directions will never form a closed loop (deadlock).



Dependency graph

*Wikimedia

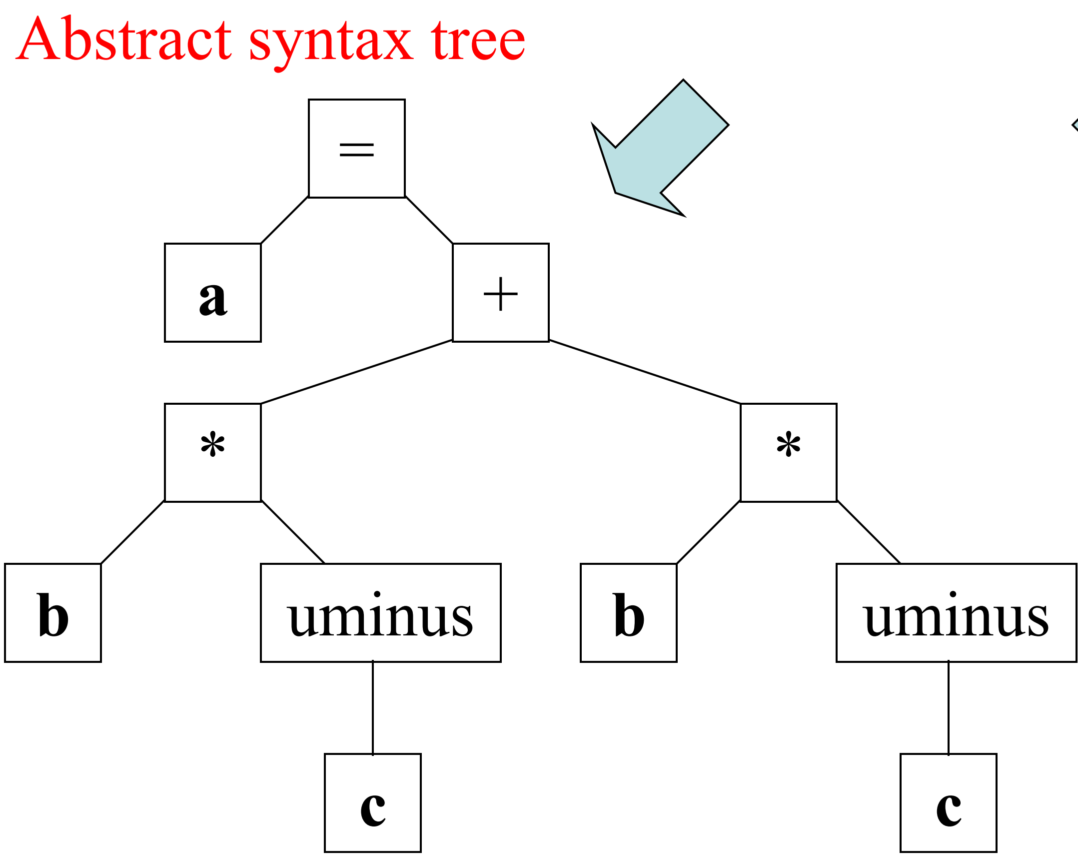
Abstract Syntax Trees vs. DAGs

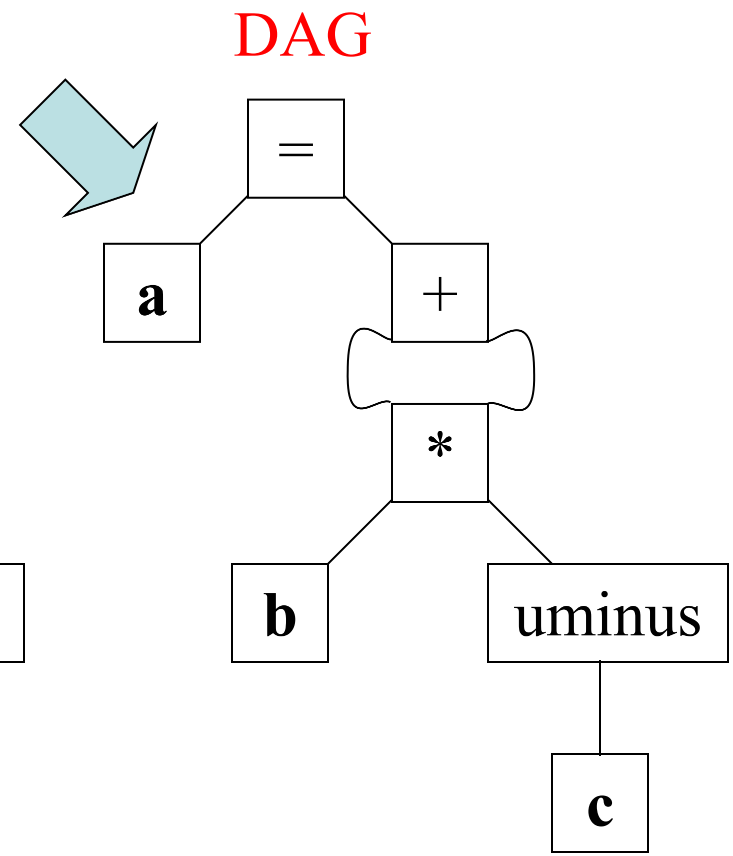
- A Directed Acyclic Graph (DAG) is similar to a parse tree
- The DAG enables efficient code generation
- A node N in DAG has more than one parent if N represents a common sub-expression.
- The Abstract Syntax Tree for the common sub-expression would be replicated as many times as the sub-expression appears in the original expression.

Abstract Syntax Trees vs. DAGs - Example

Show different graph representation for the following expression

a = b * -c + b * -c





Postfix Notation Translation - Example

a = b * -c + b * -c

a b c uminus * b c uminus * + assign

Postfix notation represents operations on a stack

Pros: easy to generate

Cons: stack operations are more difficult to optimize

Bytecode (where a:1, b:2 , c:3)

```
Instruction     Meaning
-----------     -------
iload 2         // push b
iload 3         // push c
ineg            // uminus
imul            // *
iload 2         // push b
iload 3         // push c
ineg            // uminus
imul            // *
iadd            // +
istore 1        // store a
```

Three-Address Code - Example

Write the three-address code of the following expression:

Note: In three-address code there is at most one operator on the right side of an instruction

a = b * -c + b * -c

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

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

Linearized representation of abstract syntax tree

Linearized representation of a syntax DAG

Three-Address Code Statements - Examples

- Assignment statements: $x = y \ op \ z$, $x = op \ y$
- Indexed assignments: $x = y[i]$, $x[i] = y$
- Address assignments: $x = \&y$ (the r-value of x is made equal to the content of (location) y.
- Pointer assignments: $x = *y$ (the r-value of x is made equal to the content of the location pointed to by y).
- Pointer assignments: $*x = y$ (sets the r-value of the object pointed to by x to the r-value of y)

Three-Address Code Statements - Examples

- Copy statements: *x* **=** *y*
- Unconditional jumps: **`goto`** *lab* (the instruction labeled *lab* is the next to be executed)
- Conditional jumps: **`if`** *x relop y* **`goto`** *lab* (apply a relational operator (==, <, >, …etc.) to x and y, and execute the instruction with label *lab* next if true.

Three-Address Code Statements - Examples

- Assume a function with n parameters:

  `param` *x1*

  `param` *x2*

  `param` *x3*

  .....

  `param` *xn*

- Function call:

  `call` *f, n*
  `return` *y*

  Generates a call of the function f with parameters (x1, x2, x3, …, xn) and returns the function call value y.

Syntax-Directed Translation into Three-Address Code – Example1

Productions:

$S \rightarrow$ **id** $=$ E

 | **while** E **do** S

$E \rightarrow E_1 + E_2$

 | E_1 * E_2

 | - E_1

 | (E_1)

 | **id**

 | **num**

1. Synthesized attributes:

S.code	three-address code for S
S.begin	label to start of S or null
S.after	label to end of S or null
E.code	three-address code for E
E.place	a name holding the value of E

$$gen(E.\text{place} \text{ ‘=’ } E_1.\text{place} \text{ ‘+’ } E_2.\text{place})$$

Three Address Code generation ⟶ `t3 = t1 + t2`

Syntax-Directed Translation into Three-Address Code – Example1

Productions:	**2. Semantic rules:**
$S \rightarrow$ **id** $= E$	S.code = E.code || *gen*(**id**.place '=' E.place); S.begin = S.after = null
$S \rightarrow$ **while** E **do** S_1	→ (*see next slide*)

$E \rightarrow E_1 + E_2$	E.place = *newtemp*();//generates a new temporary name to hold the value of E
	E.code = E_1.code || E_2.code || *gen*(E.place '=' E_1.place '+' E_2.place)

$E \rightarrow E_1 * E_2$	E.place = *newtemp*(); E.code = E_1.code || E_2.code || *gen*(E.place '=' E_1.place '*' E_2.place)
$E \rightarrow$ **-** E_1	E.place = *newtemp*(); E.code = E_1.code || *gen*(E.place '=' 'uminus' E_1.place)
$E \rightarrow$ **(** E_1 **)**	E.place = E_1.place E.code = E_1.code
$E \rightarrow$ **id**	E.place = *newtemp*(); E.code = *gen*(E.place '=' **id**.name)
$E \rightarrow$ **num**	E.place = *newtemp*(); E.code = *gen*(E.place '=' **num**.value)

Syntax-Directed Translation into Three-Address Code – Example1

Production

$S \rightarrow$ **while** E **do** S_1

3. Semantic rule:

S.begin = *newlabel*()
S.after = *newlabel*()
S.code = *gen*(S.begin ':')
|| E.code
|| *gen*('if' E.place '=' '0' 'goto' S.after)
|| S_1.code
|| *gen*('goto' S.begin)
|| *gen*(S.after ':')

S.begin:	E.code
	`if` E.place `= 0` `goto` S.after
	S_1.code
	`goto` S.begin
S.after:	…

Three-Address Code – Example2

Write the three-address code of the following expressions:

i = 2 * n + k
while i do //True
i = i - k

```
      t1 = 2
      t2 = t1 * n
      t3 = t2 + k
      i  = t3
L1:   if i = 0 goto L2   //False case
      t4 = -k
      t5 = i + t4
      i  = t5
      goto L1    // continue
L2:
```

Implementation of Three-Address Code Statements: Quads

a = b * -c + b * -c

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

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

Quads (quadruples)

Pros: easy to rearrange code for global optimization
Cons: lots of temporaries

Implementation of Three-Address Code Statements: Triples

a = b * -c + b * -c

```
t1 = minus c
t2 = b * t1
t3 = minus 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)	=	a	(4)

Save the result in (1)

Save the result in (3)

Triples

Pros: temporaries are implicit

Cons: difficult to rearrange code

Implementation of Three-Address Code Statements: Pointers

#	*Op*	*Arg1*	*Arg2*
(14)	uminus	c	
(15)	*	b	(14)
(16)	uminus	c	
(17)	*	b	(16)
(18)	+	(15)	(17)
(19)	=	a	(18)

List of pointers to table

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

Enhancement over triples representation. It uses an additional instruction array to list the pointers to the triples in the desired order. Thus, instead of position, pointers are used to store the results.

Pros: temporaries are implicit & easier to rearrange code