Principles of Programming Languages

http://www.di.unipi.it/~andrea/Didattica/PLP-15/

Prof. Andrea Corradini

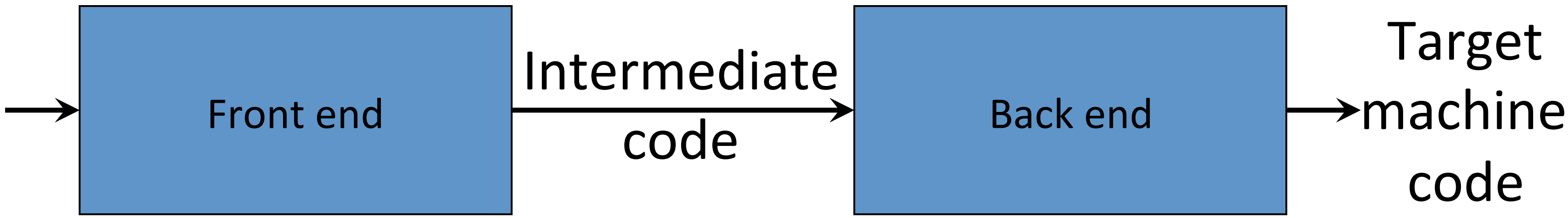
Department of Computer Science, Pisa

Lesson 19

- Intermediate-Code Generation
  - Intermediate representations
  - Syntax-directed translation to three address code

Intermediate Code Generation

- Facilitates *retargeting*: enables attaching a back end for the new machine to an existing front end



- Enables machine-independent code optimization

Summary

- Intermediate representations
- Three address statements and their implementations
- Syntax-directed translation to three address statements
  - Expressions and statements

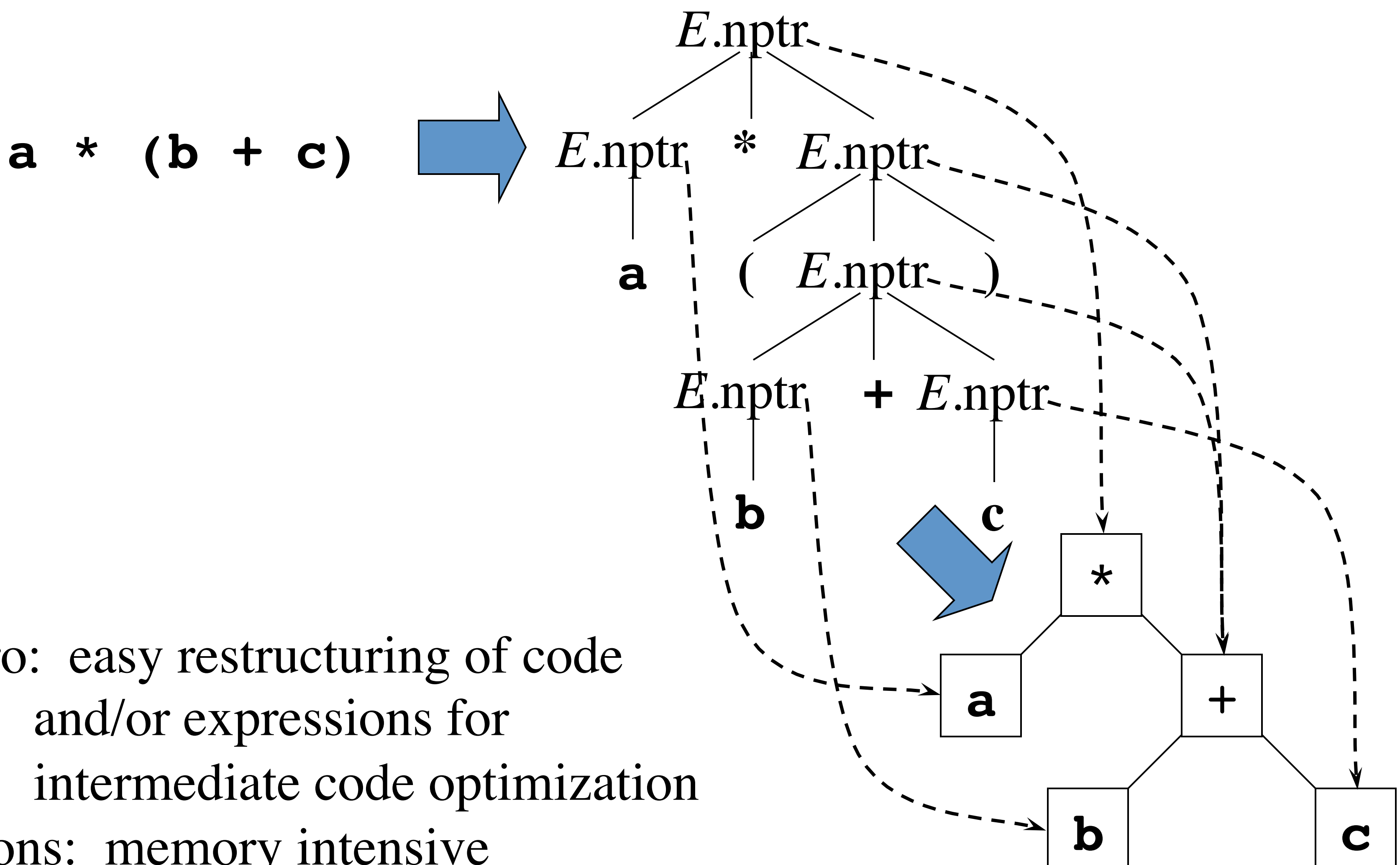
Intermediate Representations

- *Graphical representations* (e.g. AST and DAGs)
- *Postfix notation*: operations on values stored on operand stack (similar to JVM bytecode)
- *Three-address code*: (e.g. *triples* and *quads*)
  $x := y$ op z
- *Two-address code*:
  $x :=$ op y
  which is the same as $x := x$ op y

Syntax-Directed Translation of Abstract Syntax Trees

Production	Semantic Rule
$S \rightarrow$ **id :=** E	S.nptr := *mknode*(':=' , *mkleaf*(**id**, **id**.entry), E.nptr)
$E \rightarrow E_1$ **+** E_2	E.nptr := *mknode*('+' , E_1.nptr, E_2.nptr)
$E \rightarrow E_1$ ***** E_2	E.nptr := *mknode*('*' , E_1.nptr, E_2.nptr)
$E \rightarrow$ **-** E_1	E.nptr := *mknode*('uminus' , E_1.nptr)
$E \rightarrow$ **(** E_1 **)**	E.nptr := E_1.nptr
$E \rightarrow$ **id**	E.nptr := *mkleaf*(**id**, **id**.entry)

Abstract Syntax Trees

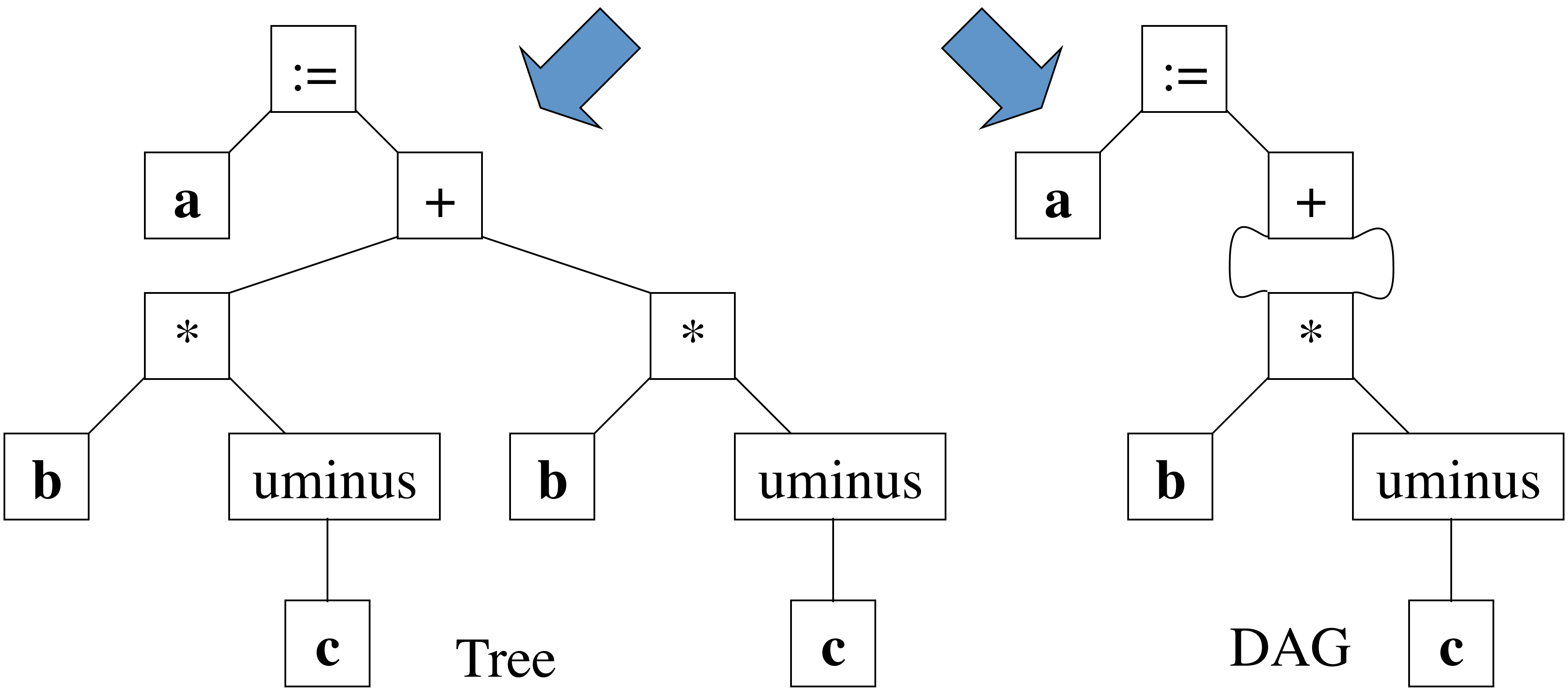


Pro: easy restructuring of code
and/or expressions for
intermediate code optimization

Cons: memory intensive

Abstract Syntax Trees versus DAGs

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



- Repeated subtrees are shared
- Implementation: *mkleaf* and *makenode are* redefined. They do not create a new node if it exists already.

Postfix Notation

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

a b c uminus * b c uminus * + assign

Postfix notation represents operations on a stack

Pro: easy to generate
Cons: stack operations are more difficult to optimize

Bytecode (for example)

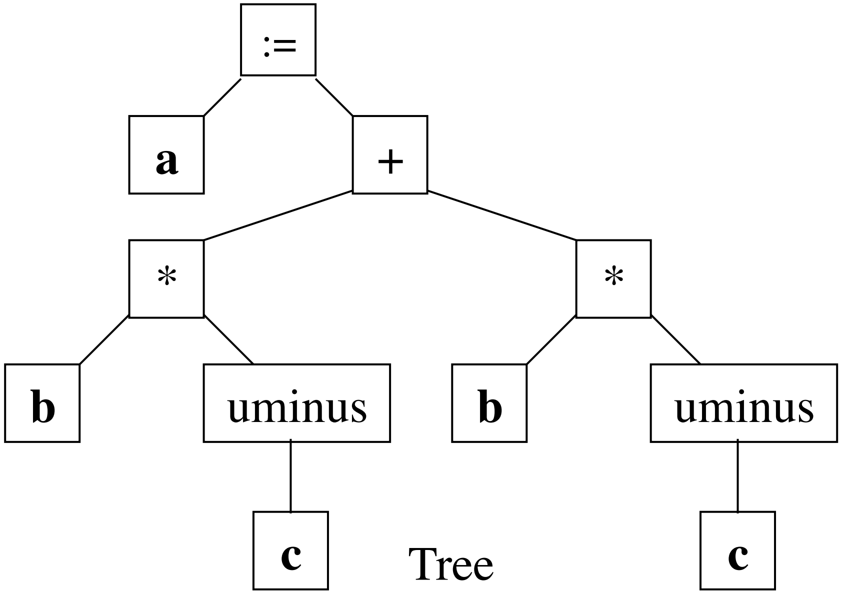
```
iload 2    // push b
iload 3    // push c
ineg       // uminus
imul       // *
iload 2    // push b
iload 3    // push c
ineg       // uminus
imul       // *
iadd       // +
istore 1   // store a
```

Three-Address Code (1)

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

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

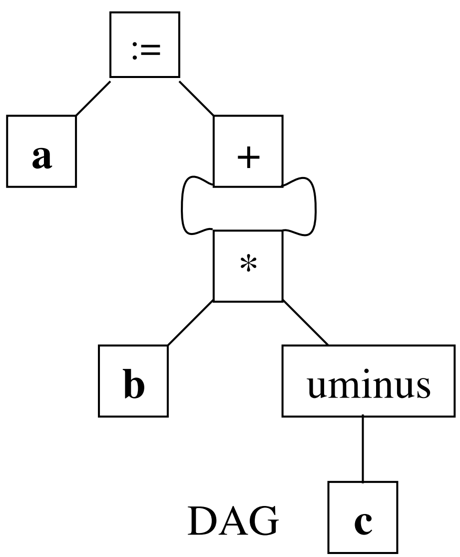
Linearized representation
of a syntax tree



Tree

Three-Address Code (2)

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




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

Linearized representation of a syntax DAG

Three-Address Statements

"Addresses" are *names*, *constants* or *temporaries*

- Assignment statements: *x* **:=** *y op z*, *x* **:=** *op y*
- Indexed assignments: *x* **:=** *y*[*i*], *x*[*i*] **:=** *y*
- Pointer assignments: *x* **:= &***y*, *x* **:= ****y*, ******x* **:=** *y*
- Copy statements: *x* **:=** *y*
- Unconditional jumps: **goto** *lab*
- Conditional jumps: **if** *x relop y* **goto** *lab*
- Function calls: **param** *x*… ; **call** *p*, *n*
  (or *y* = **call** *p*, *n*) ; **return** *y*

Implementation of Three-Address Statements: Quads

Sample expression

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

Three-address code

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

#	*Op*	*Arg1*	*Arg2*	*Res*
(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)

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

Implementation of Three-Address Statements: Triples

Sample expression

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

Three-address code

```
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)	:=	a	(4)

Triples

Pro: temporaries are implicit

Cons: difficult to rearrange code

Implementation of Three-Address Statements: Indirect Triples

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

Program

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

Triple container

Pro: temporaries are implicit & easier to rearrange code

Syntax-Directed Translation into Three-Address Code

Productions

$S \rightarrow$ **id :=** E
 | **while** E **do** S
 | S ; S
$E \rightarrow E$ **+** E
 | E ***** E
 | **-** E
 | **(** E **)**
 | **id**
 | **num**

Synthesized attributes:

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

gen(*E*.place ':=' E_1.place '+' E_2.place)

Code generation

```
t3 := t1 + t2
```

Syntax-Directed Translation into Three-Address Code (cont'd)

Productions	Semantic rules
$S \rightarrow$ **id** := E	S.code := E.code ‖ *gen*(**id**.place ':=' E.place); S.begin := S.after := nil
$S \rightarrow$ **while** E **do** S_1	(*see next slide*)
$E \rightarrow E_1$ **+** 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_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 := **id**.name E.code := ''
$E \rightarrow$ **num**	E.place := *newtemp*(); E.code := *gen*(E.place ':=' **num**.value)
$S \rightarrow S_1$; S_2	S.code := S_1.code ‖ S_2.code S.begin := S_1. begin S.after := S_2.after

Syntax-Directed Translation into Three-Address Code (cont’d)

Production

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

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.code
	`goto` S.begin
S.after:	…

Example (check it at home!)

```
i := 2 * n + k;
while i do
    i := i - k
```

```
     t1  := 2
     t2  := t1 * n
     t3  := t2 + k
     i   := t3
L1:  if i = 0 goto L2
     t4  := i - k
     i   := t4
     goto L1
L2:
```