

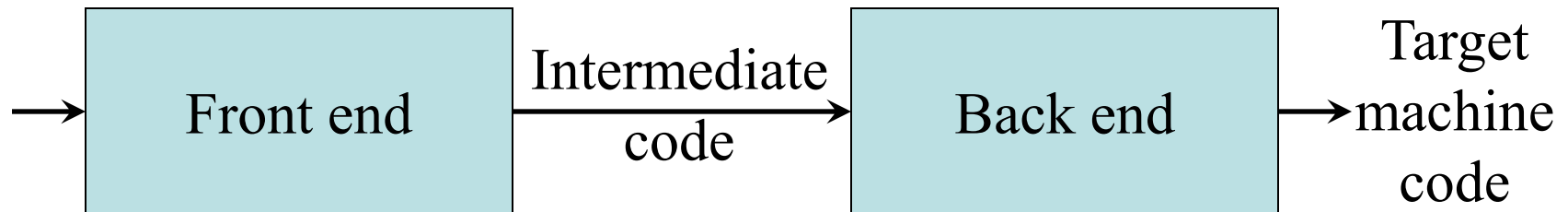
Intermediate Code Generation

Part I

Chapter 8

Intermediate Code Generation

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



- Enables machine-independent code optimization

Intermediate Representations

- *Graphical representations* (e.g. AST)
- *Postfix notation*: operations on values stored on operand stack (similar to JVM bytecode)
- *Three-address code*: (e.g. *triples* and *quads*)

$x := y \text{ op } z$

- *Two-address code*:

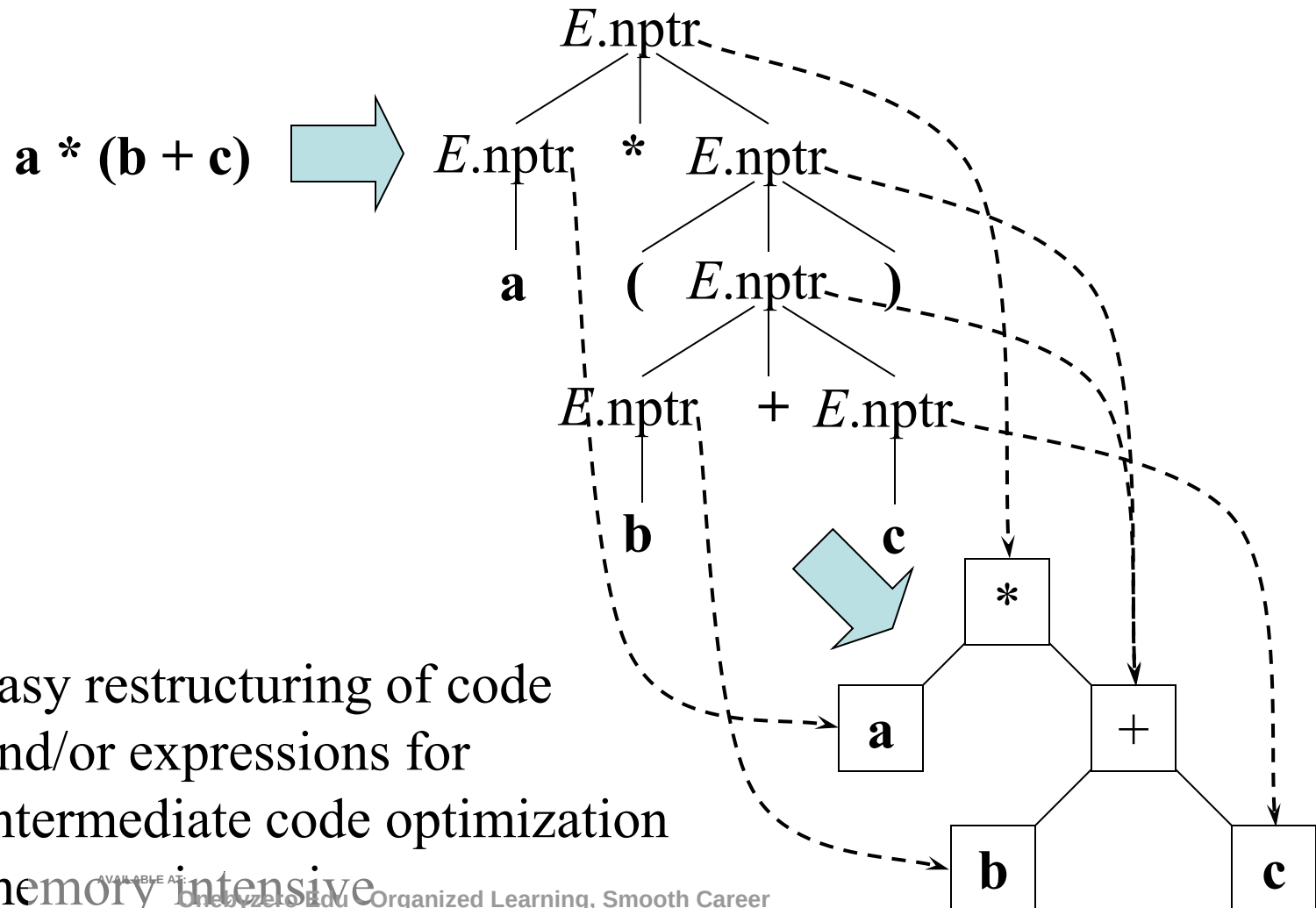
$x := \text{op } y$

which is the same as $x := x \text{ op } y$

Syntax-Directed Translation of Abstract Syntax Trees

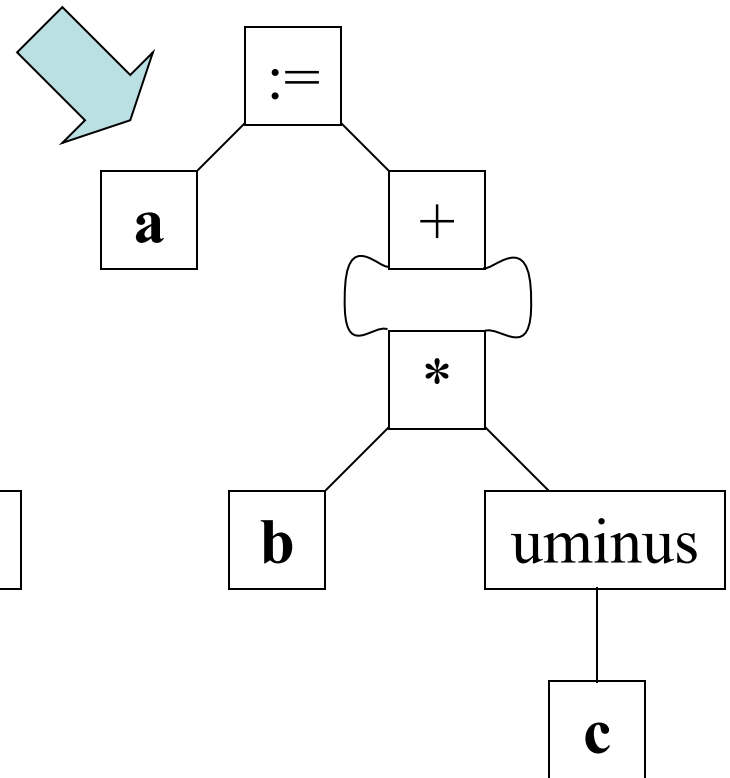
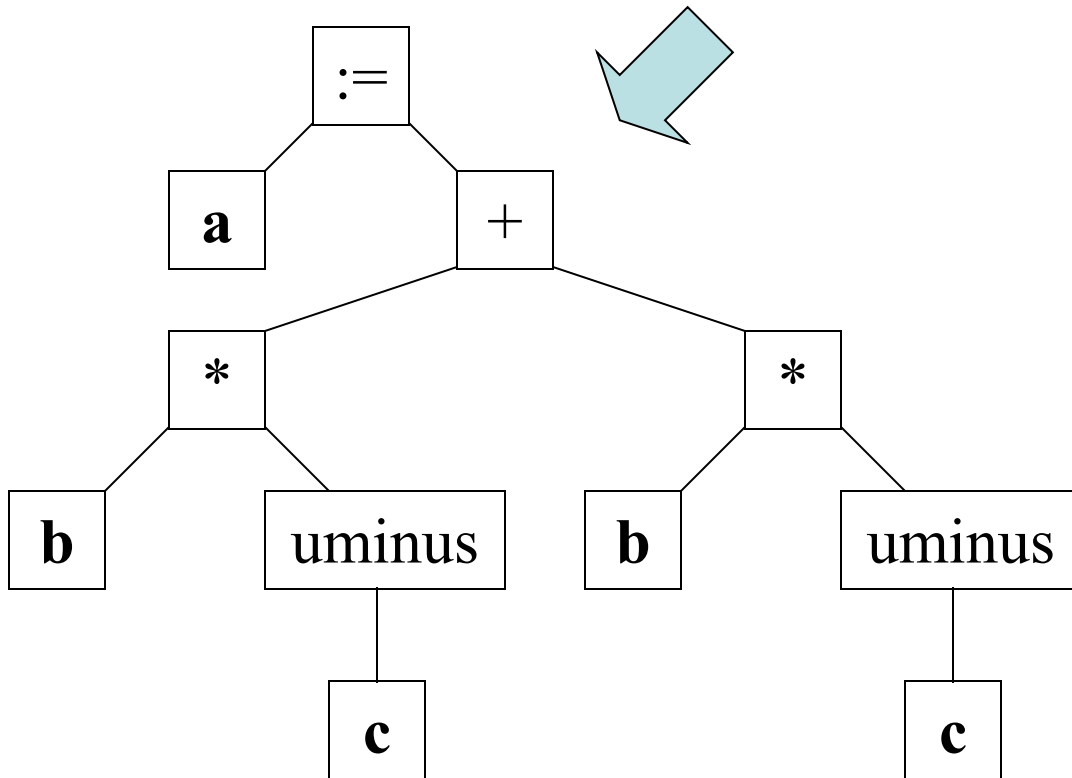
Production	Semantic Rule
$S \rightarrow \mathbf{id} := E$	$S.\text{nptr} := \text{mknode}(':=', \text{mkleaf}(\mathbf{id}, \mathbf{id}.\text{entry}), E.\text{nptr})$
$E \rightarrow E_1 + E_2$	$E.\text{nptr} := \text{mknode}('+', E_1.\text{nptr}, E_2.\text{nptr})$
$E \rightarrow E_1 * E_2$	$E.\text{nptr}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

Abstract Syntax Trees



Abstract Syntax Trees versus DAGs

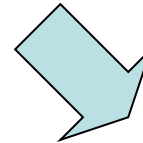
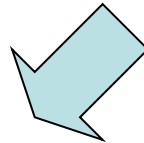
$a := b * -c + b * -c$



AVAILABLE AT:

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

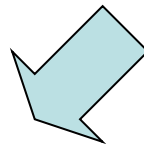
Cons: stack operations are more
difficult to optimize

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

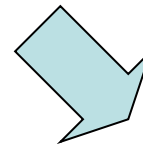
Three-Address Code

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



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

Linearized representation
of a syntax tree



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

Linearized representation
of a syntax DAG

Three-Address Statements

- Assignment statements: $x := y \text{ op } z$, $x := \text{op } y$
- 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** $x \text{ relop } y$ **goto** *lab*
- Function calls: **param** $x \dots$ **call** p, n
return y

Syntax-Directed Translation into Three-Address Code

Productions

$S \rightarrow \text{id} := E$
 $\quad | \text{ while } E \text{ do } S$
 $E \rightarrow E + E$
 $\quad | E * E$
 $\quad | - E$
 $\quad | (E)$
 $\quad | \text{id}$
 $\quad | \text{num}$

Synthesized attributes:

$S.\text{code}$	three-address code for S
$S.\text{begin}$	label to start of S or nil
$S.\text{after}$	label to end of S or nil
$E.\text{code}$	three-address code for E
$E.\text{place}$	a name holding the value of E

$\text{gen}(E.\text{place} \text{ ':=' } E_1.\text{place} \text{ '+' } E_2.\text{place})$

 Code generation $\rightarrow$ $t3 := t1 + t2$

AVAILABLE AT:

Onebyzero Edu - Organized Learning, Smooth Career

The Comprehensive Academic Study Platform for University Students in Bangladesh (www.onebyzeroedu.com)

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

Productions	Semantic rules
$S \rightarrow \mathbf{id} := E$	$S.code := E.code \parallel gen(\mathbf{id.place} := E.place); S.begin := S.after := nil$
$S \rightarrow \mathbf{while} E$ $\mathbf{do} S_1$	(see next slide)
$E \rightarrow E_1 + E_2$	$E.place := newtemp();$ $E.code := E_1.code \parallel E_2.code \parallel gen(E.place := E_1.place + E_2.place)$
$E \rightarrow E_1 * E_2$	$E.place := newtemp();$ $E.code := E_1.code \parallel E_2.code \parallel gen(E.place := E_1.place * E_2.place)$
$E \rightarrow - E_1$	$E.place := newtemp();$ $E.code := E_1.code \parallel gen(E.place := 'uminus' E_1.place)$
$E \rightarrow (E_1)$	$E.place := E_1.place$ $E.code := E_1.code$
$E \rightarrow \mathbf{id}$	$E.place := \mathbf{id.name}$ $E.code := ''$
$E \rightarrow \mathbf{num}$	$E.place := newtemp();$ $E.code := gen(E.place = num.value)$

AVAILABLE AT:

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

Production

$S \rightarrow \mathbf{while} \ E \ \mathbf{do} \ S_1$

Semantic rule

$S.\mathbf{begin} := \mathit{newlabel}()$

$S.\mathbf{after} := \mathit{newlabel}()$

$S.\mathbf{code} := \mathit{gen}(S.\mathbf{begin} \text{ ':' }) \parallel$
 $\quad E.\mathbf{code} \parallel$
 $\quad \mathit{gen}(\text{'if' } E.\mathbf{place} \text{ '=' '0' 'goto' } S.\mathbf{after}) \parallel$
 $\quad S_1.\mathbf{code} \parallel$
 $\quad \mathit{gen}(\text{'goto' } S.\mathbf{begin}) \parallel$
 $\quad \mathit{gen}(S.\mathbf{after} \text{ ':' })$

$S.\mathbf{begin}:$

$E.\mathbf{code}$

if $E.\mathbf{place} = 0$ **goto** $S.\mathbf{after}$

$S.\mathbf{code}$

goto $S.\mathbf{begin}$

$S.\mathbf{after}:$

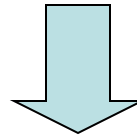
...

Example

```

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:

```

AVAILABLE AT:

Implementation of Three-Address Statements: Quads

#	Op	Arg1	Arg2	Res
(0)	umin us	c		t1
(1)	*	b	t1	t2
(2)	umin us	c		t3
(3)	*	b	t3	t4
(4)	+	t2	t4	t5

Pro: easy to rearrange code for global optimization

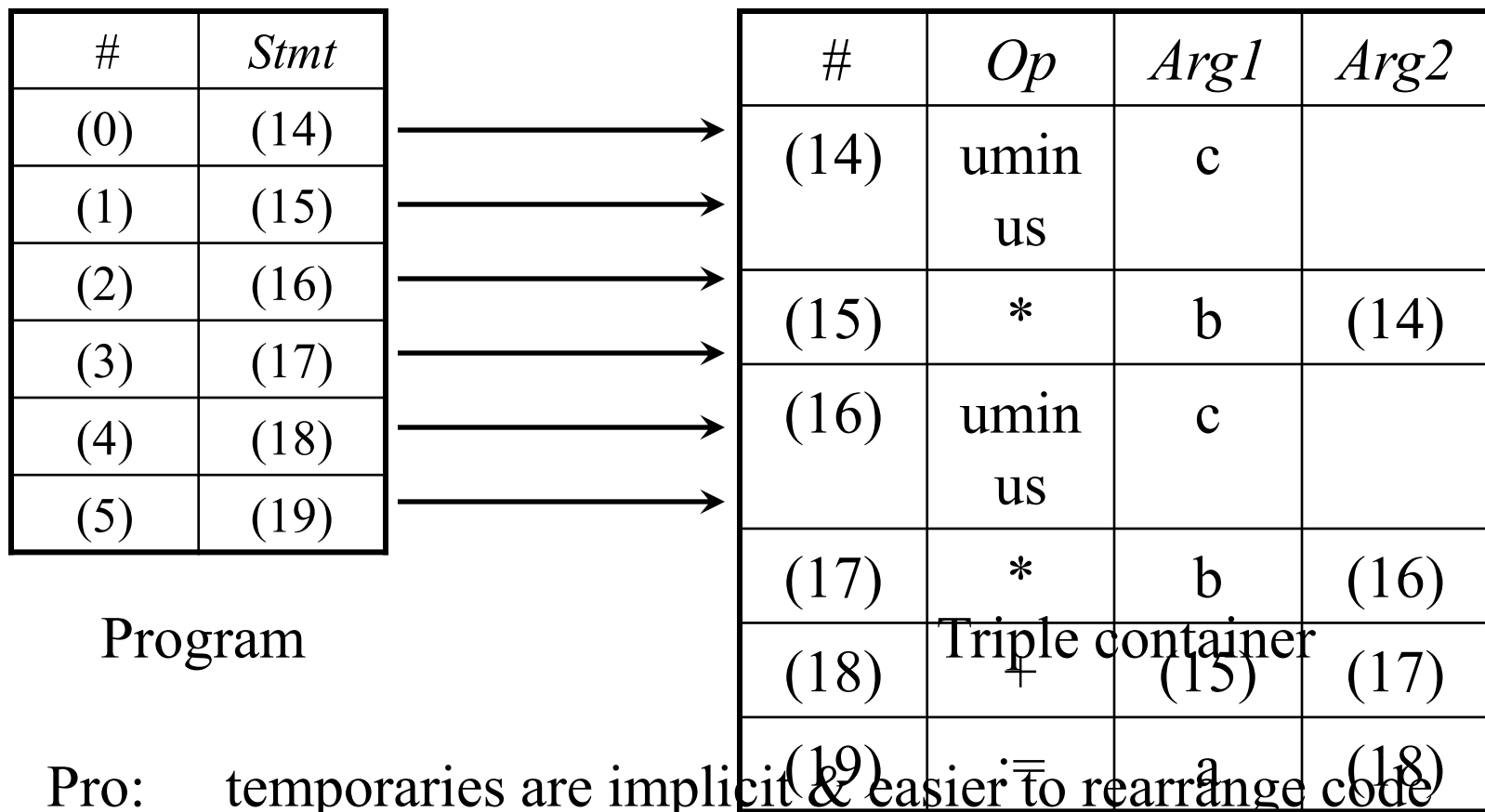
Cons: lots of temporaries

Implementation of Three-Address Statements: Triples

#	Op	Arg1	Arg2
(0)	umin us	c	
(1)	*	b	(0)
(2)	umin us	c	
(3)	*	b	(2)
(4)	+	(1)	(3)
(5)	.	a	(4)

Triples
 Pros: temporaries are implicit
 Cons: difficult to rearrange code

Implementation of Three-Address Stmts: Indirect Triples



AVAILABLE AT:

Onebyzero Edu - Organized Learning, Smooth Career

The Comprehensive Academic Study Platform for University Students in Bangladesh (www.onebyzeroedu.com)

Names and Scopes

- The three-address code generated by the syntax-directed definitions shown on the previous slides is somewhat simplistic, because it assumes that the names of variables can be easily resolved by the back end in global or local variables
- We need local symbol tables to record global declarations as well as local declarations in procedures, blocks, and structs to resolve names

Symbol Tables for Scoping

```
struct S
{ int a;
  int b;
} s;
```

We need a symbol table
for the *fields* of struct S

```
void swap(int& a, int& b)
{ int t;
  t = a;
  a = b;
  b = t;
}
```

Need symbol table
for *global* variables
and functions

Need symbol table for *arguments*
and *locals* for each function

```
void somefunc()
{ ...
  swap(s.a, s.b);
  ...
}
```

Check: **s** is global and has fields **a** and **b**
Using symbol tables we can generate
code to access **s** and its fields

Offset and Width for Runtime Allocation

```
struct S
{ int a;
  int b;
} s;
```

The fields **a** and **b** of struct **S** are located at *offsets* 0 and 4 from the start of **S**

```
void swap(int& a, int& b)
{ int t;
  t = a;
  a = b;
  b = t;
}
```

The *width* of **S** is 8

a	(0)
b	(4)

Subroutine frame holds arguments **a** and **b** and local **t** at *offsets* 0, 4, and 8

```
void somefunc()
{ ...
  swap(s.a, s.b);
  ...
}
```

Subroutine frame

fp[0]=	a	(0)
fp[4]=	b	(4)
fp[8]=	t	(8)

The *width* of the frame is 12

AVAILABLE AT:

Onebyzero Edu - Organized Learning, Smooth Career

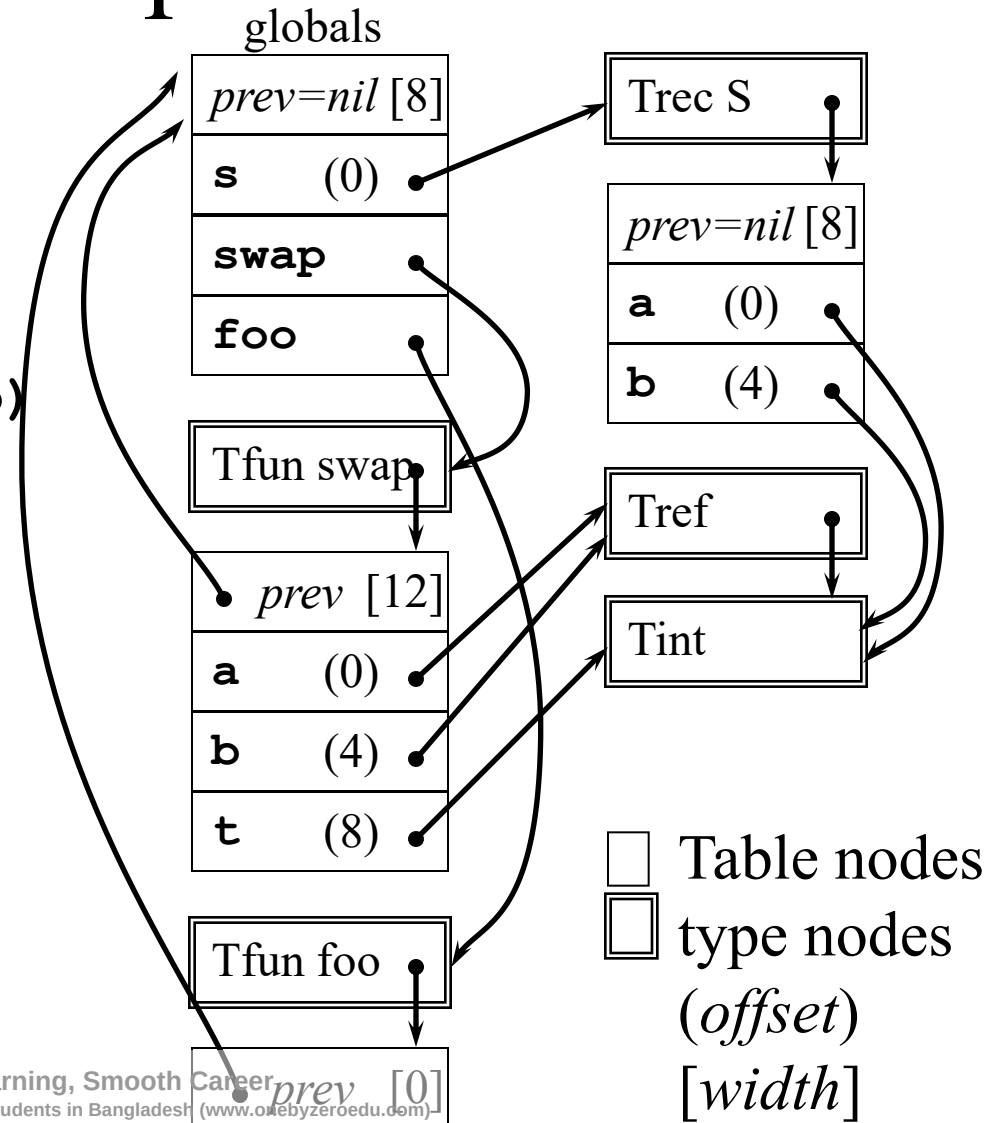
The Comprehensive Academic Study Platform for University Students in Bangladesh (www.onebyzeroedu.com)

Example

```
struct S
{ int a;
  int b;
} s;
```

```
void swap(int& a, int& b)
{ int t;
  t = a;
  a = b;
  b = t;
}
```

```
void foo()
{ ...
  swap(s.a, s.b);
  ...
}
```



Hierarchical Symbol Table Operations

- *mktable(previous)* returns a pointer to a new table that is linked to a previous table in the outer scope
- *enter(table, name, type, offset)* creates a new entry in *table*
- *addwidth(table, width)* accumulates the total width of all entries in *table*
- *enterproc(table, name, newtable)* creates a new entry in *table* for procedure with local scope *newtable*
- *lookup(table, name)* returns a pointer to the entry in the table for *name* by following linked tables

Syntax-Directed Translation of Declarations in Scope

Productions

$$P \rightarrow D ; S$$

$$D \rightarrow D ; D$$

$$| \text{ id } : T$$

$$| \text{ proc id } ; D ; S$$

$$T \rightarrow \text{ integer}$$

$$| \text{ real}$$

$$| \text{ array [num] of } T$$

$$| ^ T$$

$$| \text{ record } D \text{ end}$$

$$S \rightarrow S ; S$$

$$| \text{ id } := E$$

$$| \text{ call id } (A)$$

Productions (*cont'd*)

$$E \rightarrow E + E$$

$$| E * E$$

$$| - E$$

$$| (E)$$

$$| \text{ id}$$

$$| E ^$$

$$| \& E$$

$$| E . \text{ id}$$

$$A \rightarrow A , E$$

$$| E$$

Synthesized attributes:

$T.\text{type}$ pointer to type

$T.\text{width}$ storage width of type (bytes)

$E.\text{place}$ name of temp holding value of E

Global data to implement scoping:

tblptr stack of pointers to tables

offset stack of offset values

Syntax-Directed Translation of Declarations in Scope (cont'd)

$$P \rightarrow \{ t := mktable(\text{nil}); \text{push}(t, \text{tblptr}); \text{push}(0, \text{offset}) \}$$

$$D ; S$$

$$D \rightarrow \mathbf{id} : T$$

$$\{ \text{enter}(\text{top}(\text{tblptr}), \mathbf{id}.\text{name}, T.\text{type}, \text{top}(\text{offset}));$$

$$\text{top}(\text{offset}) := \text{top}(\text{offset}) + T.\text{width} \}$$

$$D \rightarrow \mathbf{proc id} ;$$

$$\{ t := mktable(\text{top}(\text{tblptr})); \text{push}(t, \text{tblptr}); \text{push}(0, \text{offset}) \}$$

$$D_1 ; S$$

$$\{ t := \text{top}(\text{tblptr}); \text{addwidth}(t, \text{top}(\text{offset}));$$

$$\text{pop}(\text{tblptr}); \text{pop}(\text{offset});$$

$$\text{enterproc}(\text{top}(\text{tblptr}), \mathbf{id}.\text{name}, t) \}$$

$$D \rightarrow D_1 ; D_2$$

Syntax-Directed Translation of Declarations in Scope (cont'd)

$T \rightarrow \mathbf{integer} \quad \{ T.type := 'integer'; T.width := 4 \}$

$T \rightarrow \mathbf{real} \quad \{ T.type := 'real'; T.width := 8 \}$

$T \rightarrow \mathbf{array} [\mathbf{num}] \mathbf{of} T_1$
 $\{ T.type := array(\mathbf{num.val}, T_1.type);$
 $T.width := \mathbf{num.val} * T_1.width \}$

$T \rightarrow ^ T_1$
 $\{ T.type := pointer(T_1.type); T.width := 4 \}$

$T \rightarrow \mathbf{record}$
 $\{ t := mktable(\mathbf{nil}); push(t, tblptr); push(0, offset) \}$

D end

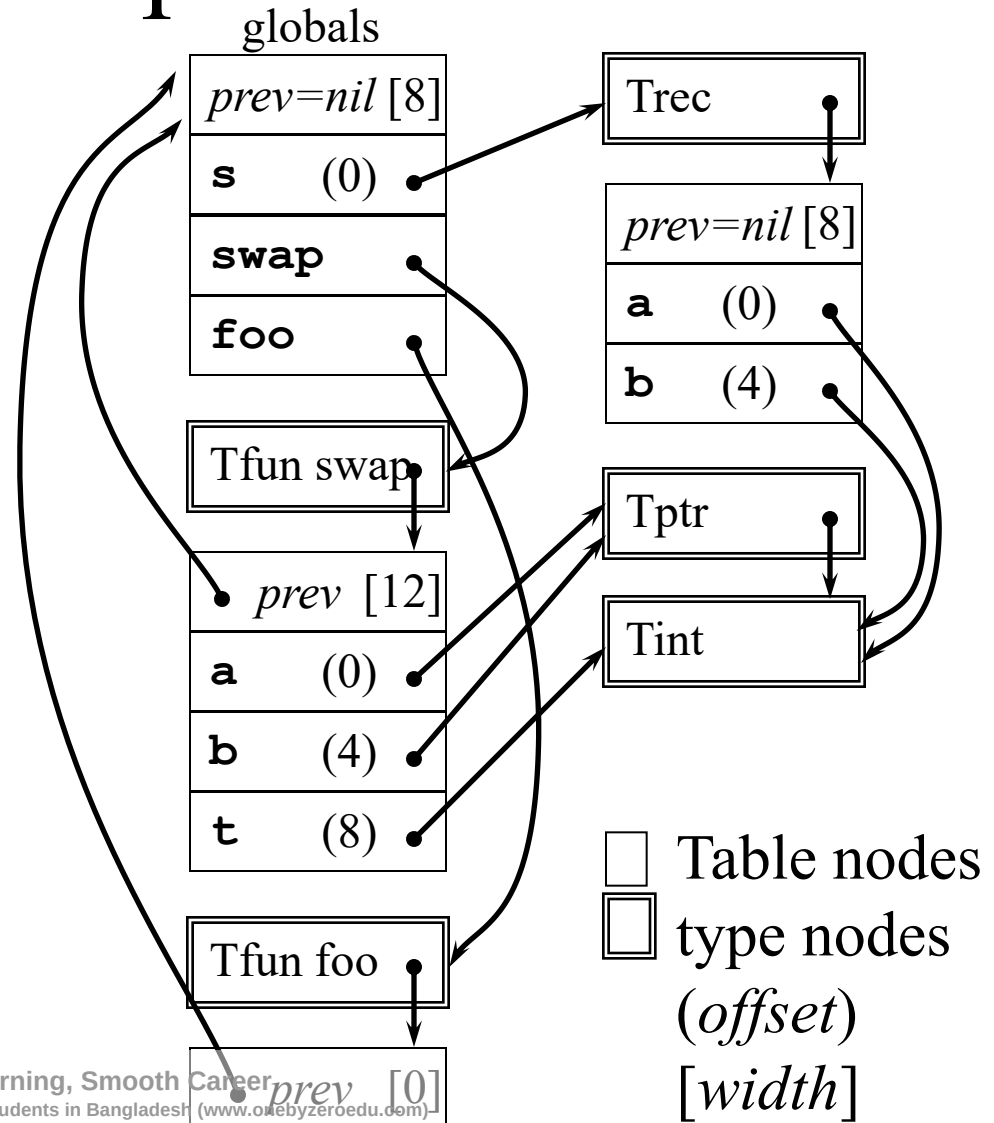
$\{ T.type := record(top(tblptr)); T.width := top(offset);$
 $addwidth(top(tblptr), top(offset)); pop(tblptr); pop(offset) \}$

Example

```
s: record
  a: integer;
  b: integer;
end;
```

```
proc swap;
  a: ^integer;
  b: ^integer;
  t: integer;
  t := a^;
  a^ := b^;
  b^ := t;
```

```
proc foo;
  call swap(&s.a, &s.b);
```



Syntax-Directed Translation of Statements in Scope

$S \rightarrow S ; S$

$S \rightarrow \mathbf{id} := E$

$\{ p := \text{lookup}(\text{top}(\text{tblptr}), \mathbf{id.name});$

$\mathbf{if} \ p = \text{nil} \ \mathbf{then}$

$\text{error}()$

$\mathbf{else if} \ p.\text{level} = 0 \ \mathbf{then} \ // \ \text{global variable}$

$\text{emit}(\mathbf{id.place} \ ' := ' \ E.\text{place})$

$\mathbf{else} \ // \ \text{local variable in subroutine frame}$

$\text{emit}(\text{fp}[p.\text{offset}] \ ' := ' \ E.\text{place}) \}$

Globals

s	(0)
x	(8)
y	(12)

Subroutine
frame

fp[0]=	a	(0)
fp[4]=	b	(4)
fp[8]=	t	(8)

...

Syntax-Directed Translation of Expressions in Scope

$E \rightarrow E_1 + E_2$ { $E.place := newtemp()$;
 $emit(E.place := E_1.place + E_2.place)$ }

$E \rightarrow E_1 * E_2$ { $E.place := newtemp()$;
 $emit(E.place := E_1.place * E_2.place)$ }

$E \rightarrow - E_1$ { $E.place := newtemp()$;
 $emit(E.place := \text{'uminus'} E_1.place)$ }

$E \rightarrow (E_1)$ { $E.place := E_1.place$ }

$E \rightarrow \mathbf{id}$ { $p := lookup(top(tblptr), \mathbf{id}.name)$;
 if $p = \mathbf{nil}$ **then** $error()$
 else if $p.level = 0$ **then** // *global variable*
 $E.place := \mathbf{id}.place$
 else // *local variable in frame*

Syntax-Directed Translation of Expressions in Scope (cont'd)

$$E \rightarrow E_1 \wedge \quad \{ E.place := newtemp(); \\ emit(E.place ':=' '*' E_1.place) \}$$

$$E \rightarrow \& E_1 \quad \{ E.place := newtemp(); \\ emit(E.place ':=' '\&' E_1.place) \}$$

$$E \rightarrow id_1 . id_2 \quad \{ p := lookup(top(tblptr), id_1.name); \\ \textbf{if } p = nil \textbf{ or } p.type \neq Trec \textbf{ then } error() \\ \textbf{else} \\ q := lookup(p.type.table, id_2.name); \\ \textbf{if } q = nil \textbf{ then } error() \\ \textbf{else if } p.level = 0 \textbf{ then } // global variable \\ E.place := id_1.place[q.offset] \\ \textbf{else } // local variable in frame \\ E.place := fp[p.offset+q.offset] \}$$