Principles of Programming Languages

http://www.di.unipi.it/~andrea/Didattica/PLP-14/

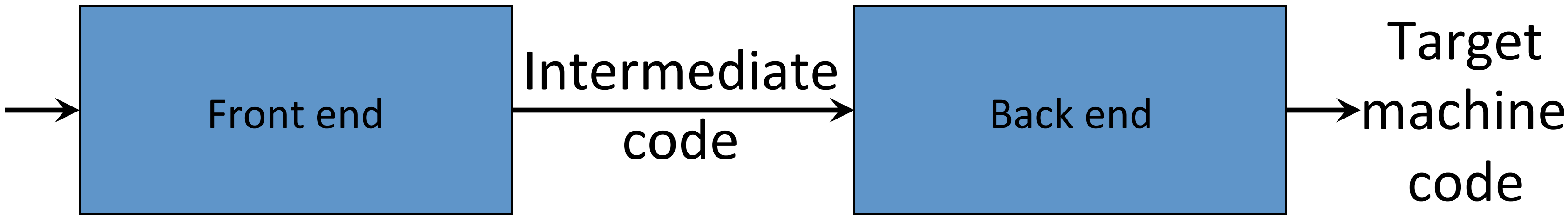
Prof. Andrea Corradini
Department of Computer Science, Pisa

Lesson 12

- Intermediate-Code Generation
  – 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
- Handling local names and scopes with symbol tables
- Syntax-directed translation of
  - Declarations in scope
  - Expressions in scope
  - Statements in scope

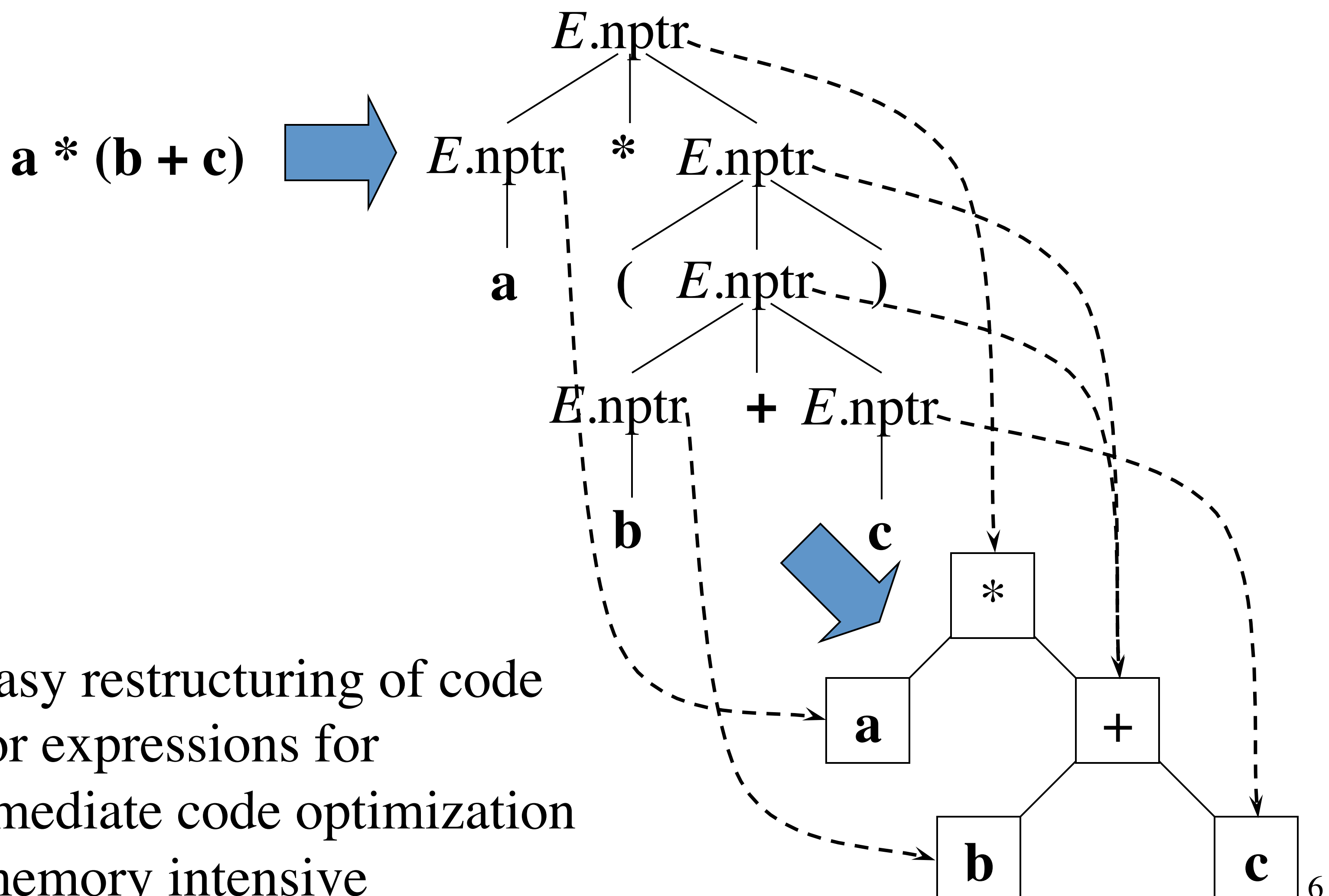
Intermediate Representations

- *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
$S \rightarrow$ **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

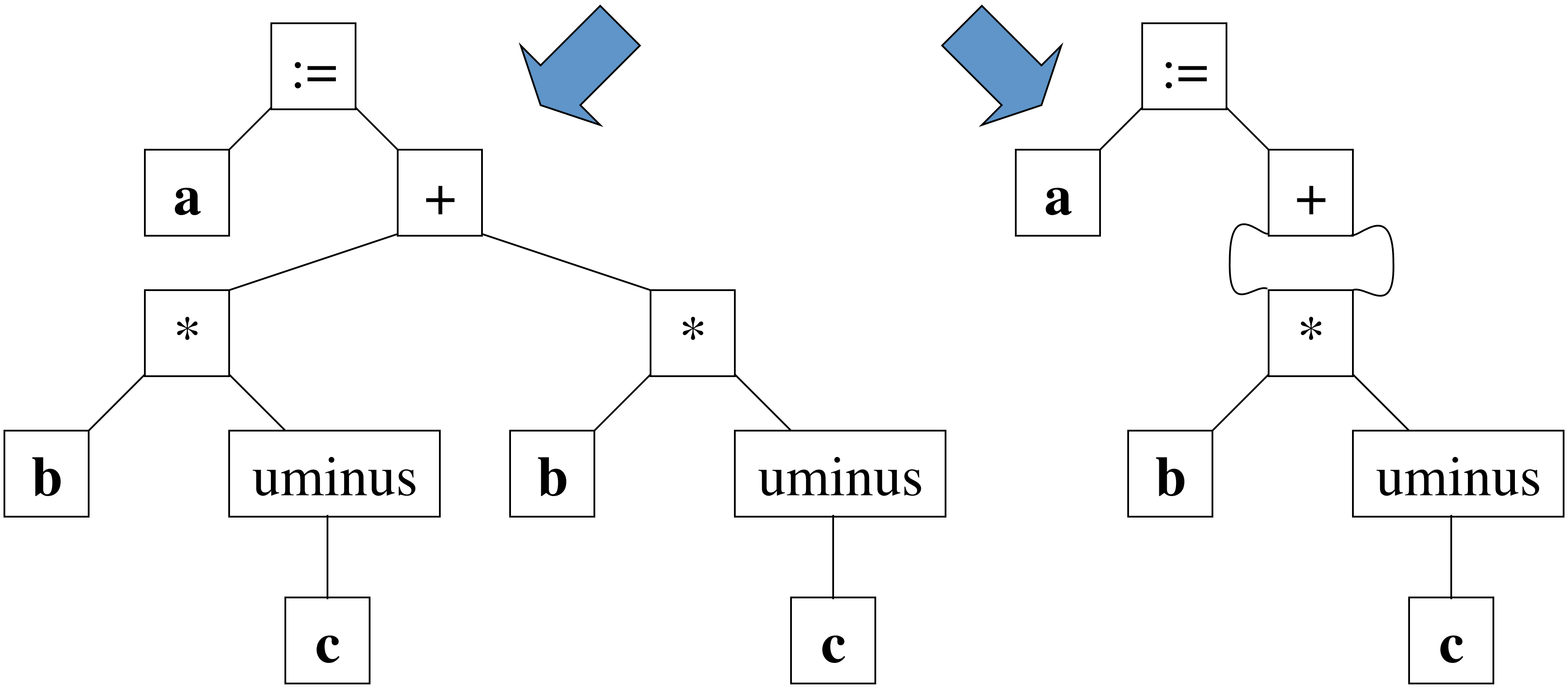


Pro: easy restructuring of code and/or expressions for intermediate code optimization

Cons: memory intensive

Abstract Syntax Trees versus DAGs

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



Tree

DAG

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

"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

$E \rightarrow E$ **+** 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)

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

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:
```

Names and Scopes

- The three-address code generated by the syntax-directed definitions shown is simplistic
- 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;

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

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

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

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

Subroutine frame

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

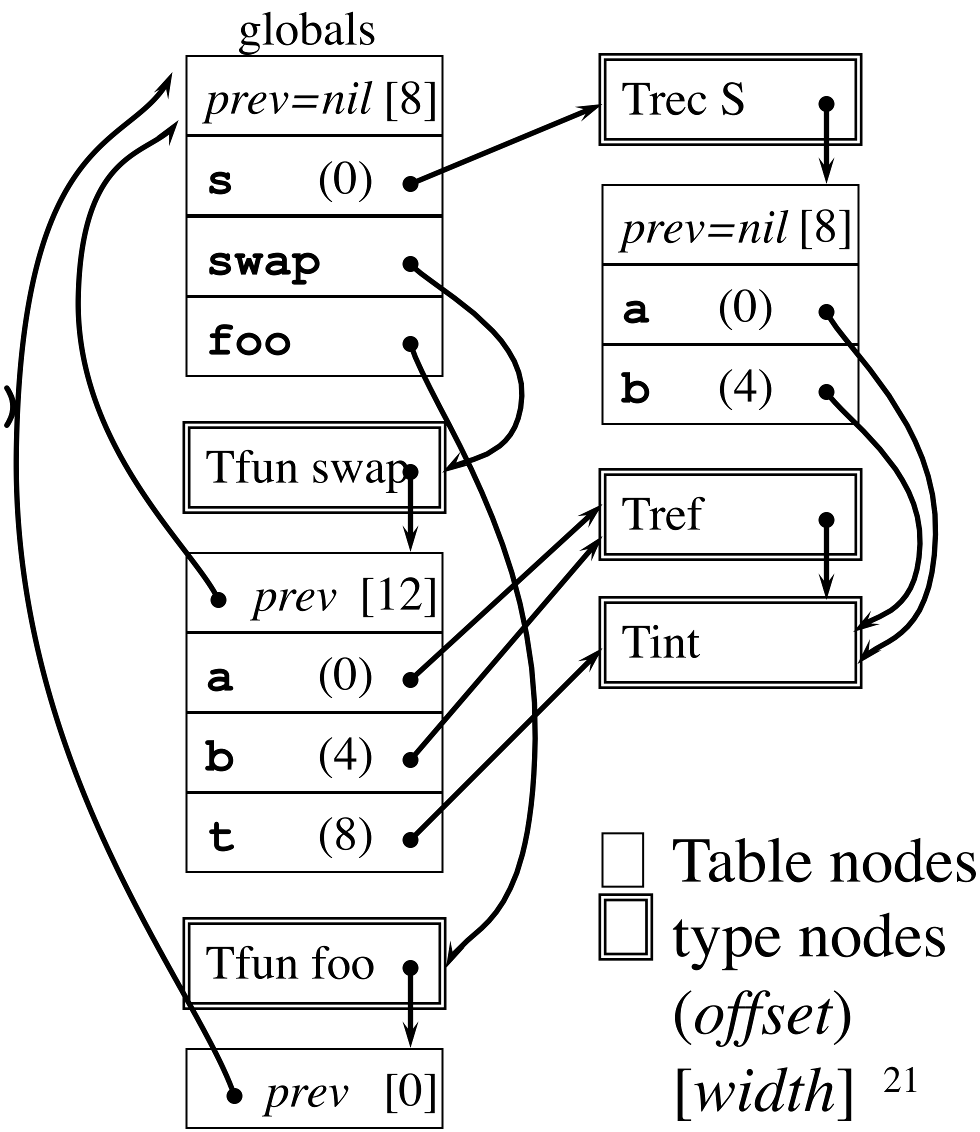
The *width* of the frame is 12

Symbol Tables for Scoping

```
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 (empty) 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
    | **id :** T
    | **proc id ;** D **;** S
$T \rightarrow$ **integer**
    | **real**
    | **array [num] of** T
    | **^** T
    | **record** D **end**
$S \rightarrow S$ **;** S
    | **id :=** E
    | **call id (** A **)**

Productions *(cont'd)*

$E \rightarrow E$ **+** E
    | E ***** E
    | **-** E
    | **(** E **)**
    | **id**
    | E **^**
    | **&** E
    | E **. id**
$A \rightarrow A$ **,** E
    | E

Synthesized attributes:

T.type	pointer to type
T.width	storage width of type (bytes)
E.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*(nil); *push*(*t*, *tblptr*); *push*(0, *offset*) }
D **;** *S*

$D \rightarrow$ **id :** *T*
{ *enter*(*top*(*tblptr*), **id**.name, *T*.type, *top*(*offset*));
top(*offset*) := *top*(*offset*) + *T*.width }

$D \rightarrow$ **proc id ;**
{ *t* := *mktable*(*top*(*tblptr*)); *push*(*t*, *tblptr*); *push*(0, *offset*) }
D_1 **;** *S*
{ *t* := *top*(*tblptr*); *addwidth*(*t*, *top*(*offset*));
pop(*tblptr*); *pop*(*offset*);
enterproc(*top*(*tblptr*), **id**.name, *t*) }

$D \rightarrow D_1$ **;** D_2

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

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

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

$T \rightarrow$ **array [num] of** T_1

{ T.type := *array*(**num**.val, T_1.type);

T.width := **num**.val * T_1.width }

$T \rightarrow$ **^** T_1

{ T.type := *pointer*(T_1.type); T.width := 4 }

$T \rightarrow$ **record**

{ *t* := *mktable*(nil); *push*(*t*, *tblptr*); *push*(0, *offset*) }

D **end**

{ T.type := *record*(*top*(*tblptr*)); T.width := *top*(*offset*);

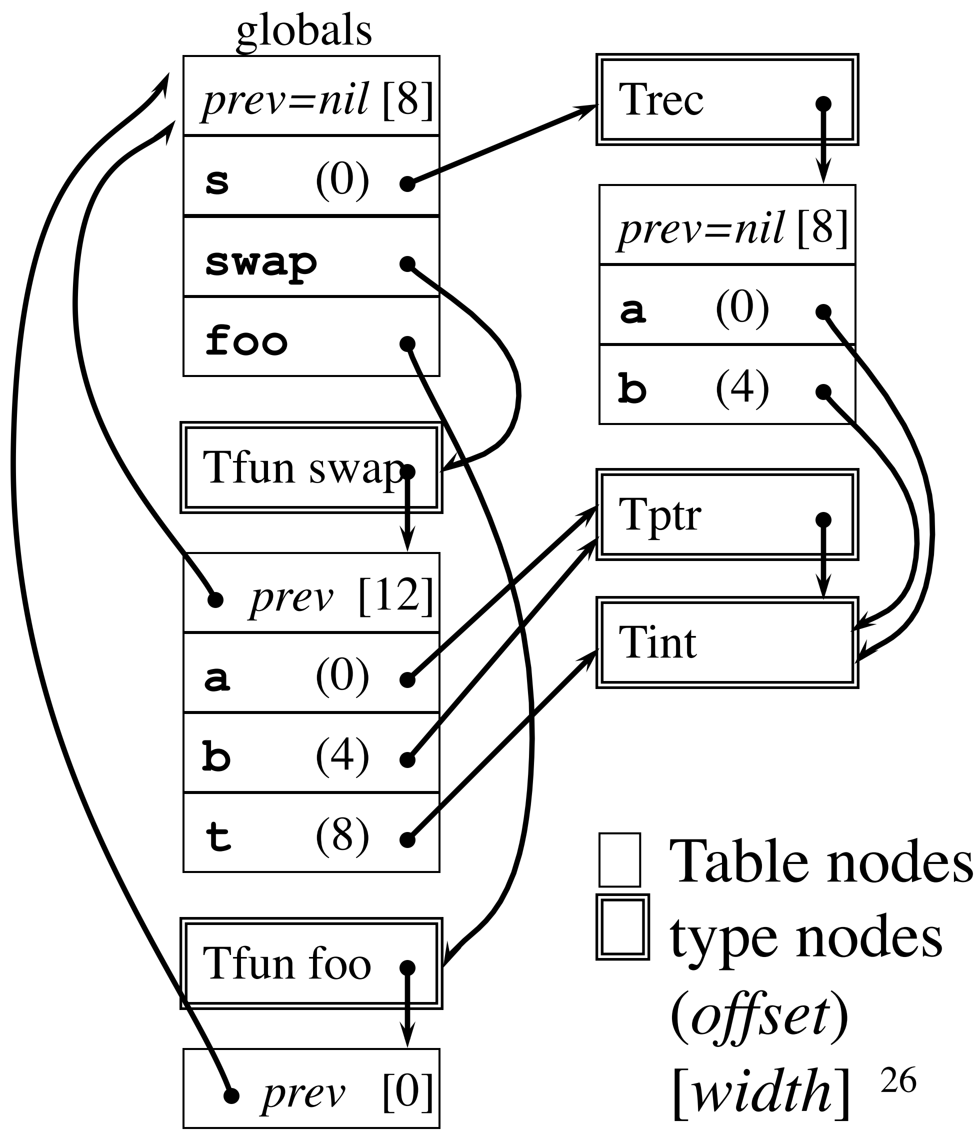
addwidth(*top*(*tblptr*), *top*(*offset*)); *pop*(*tblptr*); *pop*(*offset*) }

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

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

{ *p* := *lookup*(*top*(*tblptr*), **id**.name);

if *p* = nil **then**

error()

else if *p*.level = 0 **then** *// global variable*

emit(**id**.place ':=' *E*.place)

else *// local variable in subroutine frame*

emit(fp[*p*.offset] ':=' *E*.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 ':=' 'uminus' E_1.place) }
$E \rightarrow$ **(** E_1 **)** { E.place := E_1.place }
$E \rightarrow$ **id** { *p* := *lookup*(*top*(*tblptr*), **id**.name);
if *p* = nil **then** *error*()
else if *p*.level = 0 **then** *// global variable*
emit(E.place ':=' **id**.place)
else *// local variable in frame*
emit(E.place ':=' fp[*p*.offset]) }

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

$E \rightarrow E_1$ **^** { E.place := *newtemp*();
emit(E.place ':=' '*' E_1.place) }
$E \rightarrow$ **&** E_1 { E.place := *newtemp*();
emit(E.place ':=' '&' E_1.place) }
$E \rightarrow$ $\mathbf{id}_1$ **.** $\mathbf{id}_2$ { p := *lookup*(*top*(*tblptr*), $\mathbf{id}_1$.name);
if p = nil **or** p.type != Trec **then** *error*()
else
q := *lookup*(p.type.table, $\mathbf{id}_2$.name);
if q = nil **then** error()
else if p.level = 0 **then** // *global variable*
emit(E.place ':=' $\mathbf{id}_1$.place[q.offset])
else // *local variable in frame*
emit(E.place ':=' fp[p.offset+q.offset])}