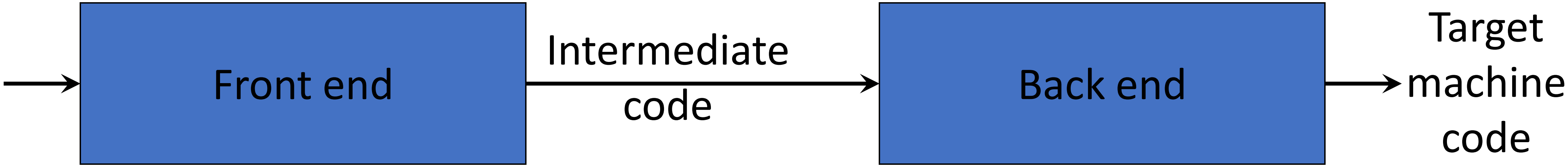
Intermediate Code Generation - Types of Three address code, Representation, Declarations

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

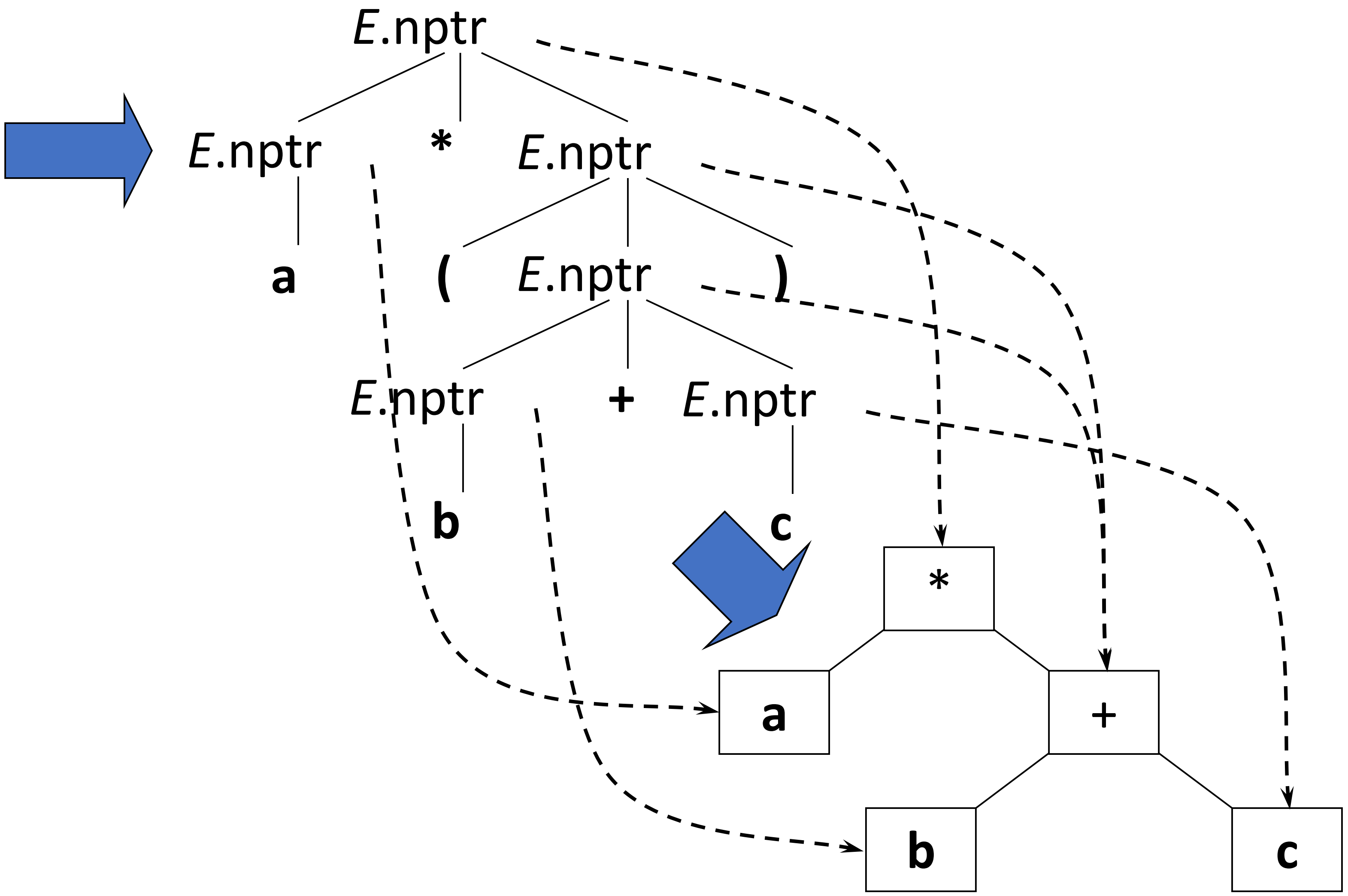
- Graphical representations
  - AST
- Postfix notation: operations on values stored on operand stack
  - JVM bytecode
- Three-address code: x := y op z
  - Variation of three address code - two-address code: x := op y

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)

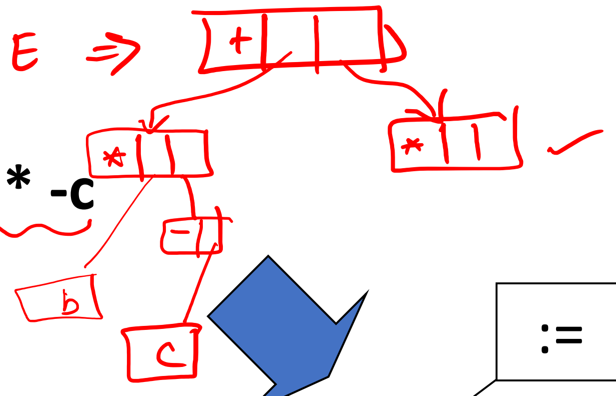
Abstract Syntax Trees

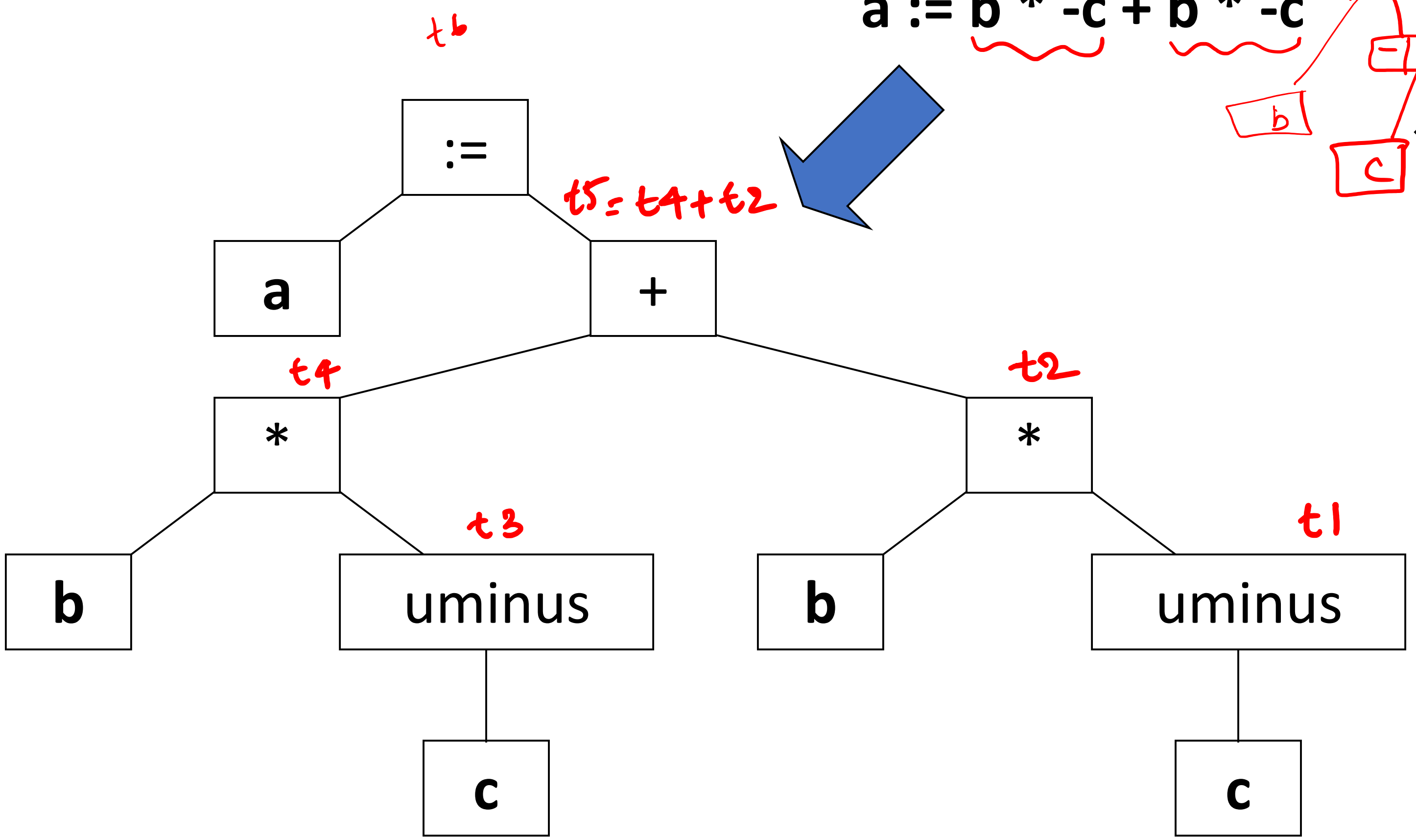
a * (b + c)



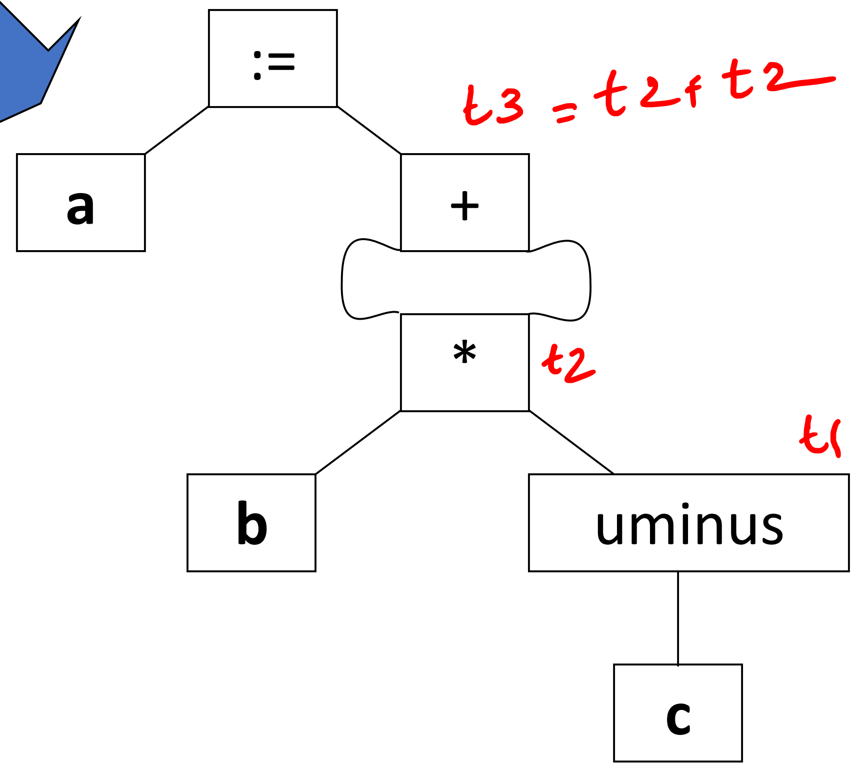
Abstract Syntax Trees versus DAGs

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





Tree



DAG

SDD for creating DAG's

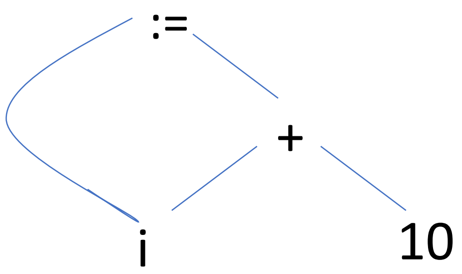
Production		Semantic Rules
1)	E -> E1+T	E.node= new Node('+', E1.node,T.node)
2)	E -> E1-T	E.node= new Node('-', E1.node,T.node)
3)	E -> T	E.node = T.node
4)	T -> (E)	T.node = E.node
5)	T -> id	T.node = new Leaf(id, id.entry)
6)	T -> num	T.node = new Leaf(num, num.val)

Example: **a + a * (b – c) + (b – c) * d**

1) p1=Leaf(id, entry-a)
2) P2=Leaf(id, entry-a)=p1
3) p3=Leaf(id, entry-b)
4) p4=Leaf(id, entry-c)
5) p5=Node('-',p3,p4)
6) p6=Node('*',p2,p5)
7) p7=Node('+',p1,p6)
8) p8=Leaf(id,entry-b)=p3
9) p9=Leaf(id,entry-c)=p4
10) p10=Node('-',p8,p9)=p5
11) p11=Leaf(id,entry-d)
12) p12=Node('*',p10,p11)
13) p13=Node('+',p7,p12)

Value-number method for constructing DAG's

$i = i + 10$



1	id	To entry of i	
2	num	10	
3	+	1	2
4	:=	1	3

- Algorithm
  - Search the array for a node M with label op, left child l and right child r
  - If there is such a node, return the value number M
  - If not create in the array a new node N with label op, left child l, and right child r and return its value
- We may use a hash table

Postfix Notation

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

a b c uminus * b c uminus * + assign

Postfix notation represents operations on a stack

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
```

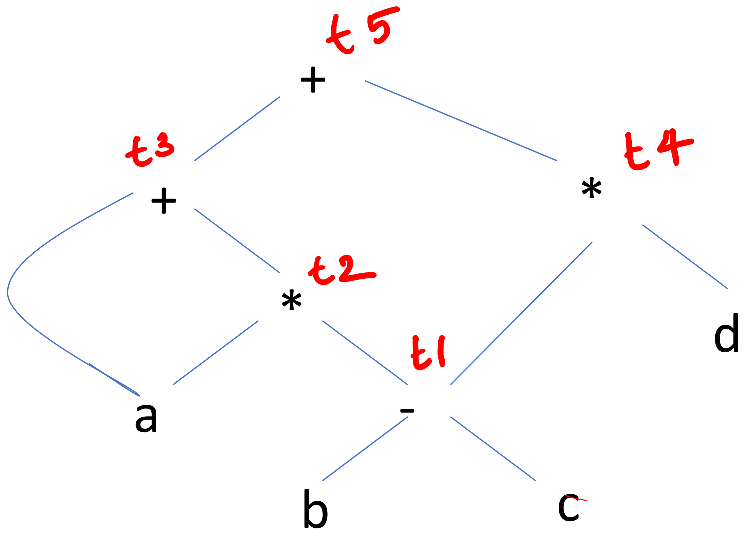
Linearized representation of a syntax tree

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

Linearized representation of a syntax DAG

Three address code

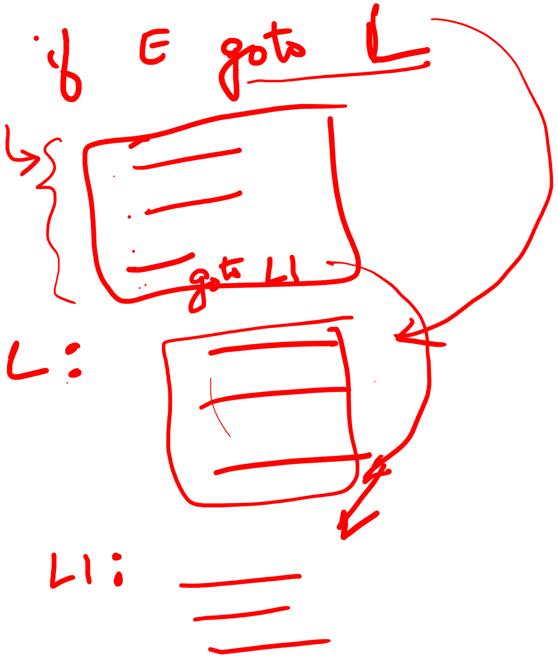
- In a three address code there is at most one operator at the right side of an instruction
- Example:



t1 = b – c
t2 = a * t1
t3 = a + t2
t4 = t1 * d
t5 = t3 + t4

Types of three address codes

- x = y op z
- x = op y
- x = y
- goto L
- if x goto L and if (false x) goto L1
- if x relop y goto L



Types of three address code

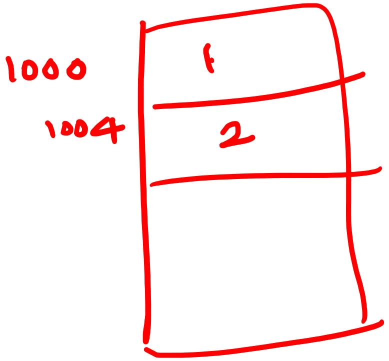
- Procedure calls using:
  - param x
  - call p,n
  - y = call p,n
- x = y[i] and x[i] = y
- x = &y
- x = *y and *x =y

param x_1

param x_2

param x_3

y = call $p,3$



$x := \underline{y[i]} + a$ ✓

$\overset{(1)}{x} := \overset{(2)}{y}[\overset{(3)}{i}]$

$t1 = y[i]$

$x := t1 + a$

Example

- do i = i+1; while (a[i] < v);

```
L:      t1 = i + 1
        i = t1
        t2 = i * 8
        t3 = a[t2]
        if t3 < v goto L
```

Symbolic labels

```
100:    t1 = i + 1
101:    i = t1
102:    t2 = i * 8
103:    t3 = a[t2]
104:    if t3 < v goto 100
```

Position numbers

Representing three address codes

- Quadruples
  - Has four fields: op, arg1, arg2 and result
- Example: a := b * - c + b * - c

Three address code

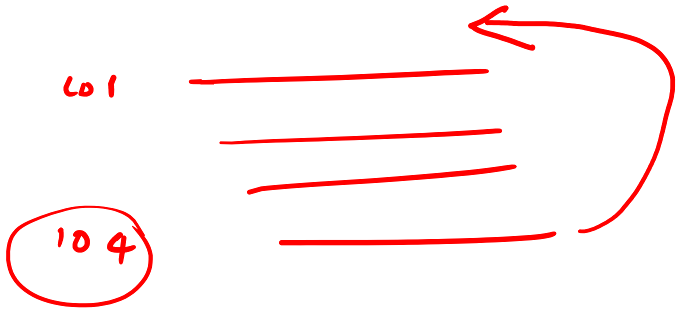
Example: a:= b * -c + b* -c

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

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

Representing three address codes

- Triples
  - Field corresponding to temporary results are not used and references to instructions are available



Example

	Op	Arg1	Arg2
(0)	uminus	c	
(1)	*	b	(0)
(2)	uminus	c	
(3)	*	b	(2)
(4)	+	(1)	(3)
(5)	:=	a	(4)

Triples for arrays

x [i] : = y

	op	arg1	arg2
(0)	[]=	x	i
(1)	assign	(0)	y

x : = y [i]

	op	arg1	arg2
(0)	=[]	y	I
(1)	assign	x	(0)

Representing three address codes

- Indirect triples
  - In addition to triples we use a list of pointers to triples

Example

	Statement
(0)	(10)
(1)	(11)
(2)	(12)
(3)	(13)
(4)	(14)
(5)	(15)

	Op	Arg1	Arg2
(10)	uminus	c	
(11)	*	b	(0)
(12)	uminus	c	
(13)	*	b	(2)
(14)	+	(1)	(3)
(15)	:=	a	(4)

SDT into Three address code

- Three address code is constructed based on the grammar construct
- Attributes
  - Code
  - Place - Address
  - Value

gen ⇒ writing 3address
emit ⇒ file.

Three address code for expression

Production	Semantic Rules
S → id := E;	S.code = E.code || gen (top.get(id.lexeme) '=' E.address emit
E → E1 + E2	E.addr = new Temp() E.code = E1.code || E2.code || gen (E.addr '=' E1.addr '+' E2.addr) emit
E → - E1	E.addr = new Temp() E.code = E1.code || gen (E.addr '=' 'uminus' E1.addr)
E → (E1)	E.addr = E1. addr E.code = E1.code

Three address code

Production	Semantic Rules
E → E1 * E2	E.addr = new Temp() E.code = E1.code || E2.code || gen (E.addr '=' E1.addr '*' E2.addr)
E → id	E.addr = top.get(id.lexeme) E.code = ' '

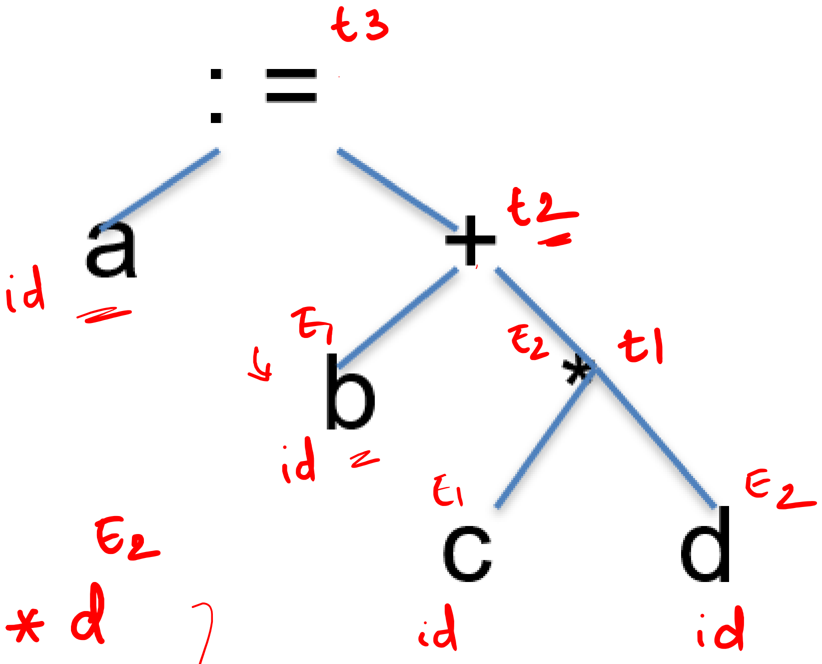
Example

- a:= b + c * d

S → id := E => E + E => E +E * E = > id + id * id

The corresponding syntax tree would be

Example



E1 E2

1. t1 = c * d

2. t2 = b + t1

3. a = t2

•Node – New temp

t1 = c * d

+ Node – New temp

t2 = b + t1

~~Root node – New temp~~

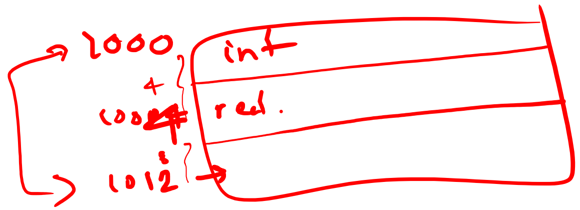
t3 = t2

Finally

a = t3

Declarations – Three address code

- Can be in a procedure – need to track scope of variable's and need symbol table
- Computing the address of variables and other is done by semantic rules related to three address code
  - Type, width, offset



Declarations

Production	Semantic rules
P → D	{offset = 0}
D → D ; D	
D → id ; T	{enter (id.name, T.type, offset); offset = offset + T.width;
T→ integer	T.type = integer; T.width = 4;
T → real	T.type = real; T.width = 8;
T→ array[num] of T1	T.type = array(num.val, T1.type) T.width = num *T1.width;

Declarations

Production	Semantic rules
T → ↑ T1	T.type = pointer (T1.type); T.width = 4;

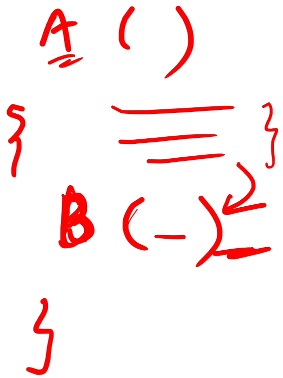
Declarations

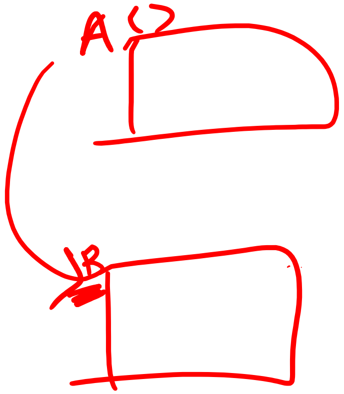
- P → D
- D → D ; D | id: T | proc id ; D ; S

A new symbol table is created when proc id; D;S is encountered

Symbol Table Functions

- 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
  - table – address of the current table, variable name, type and offset
- addwidth(table, width) accumulates the total width of all entries in table



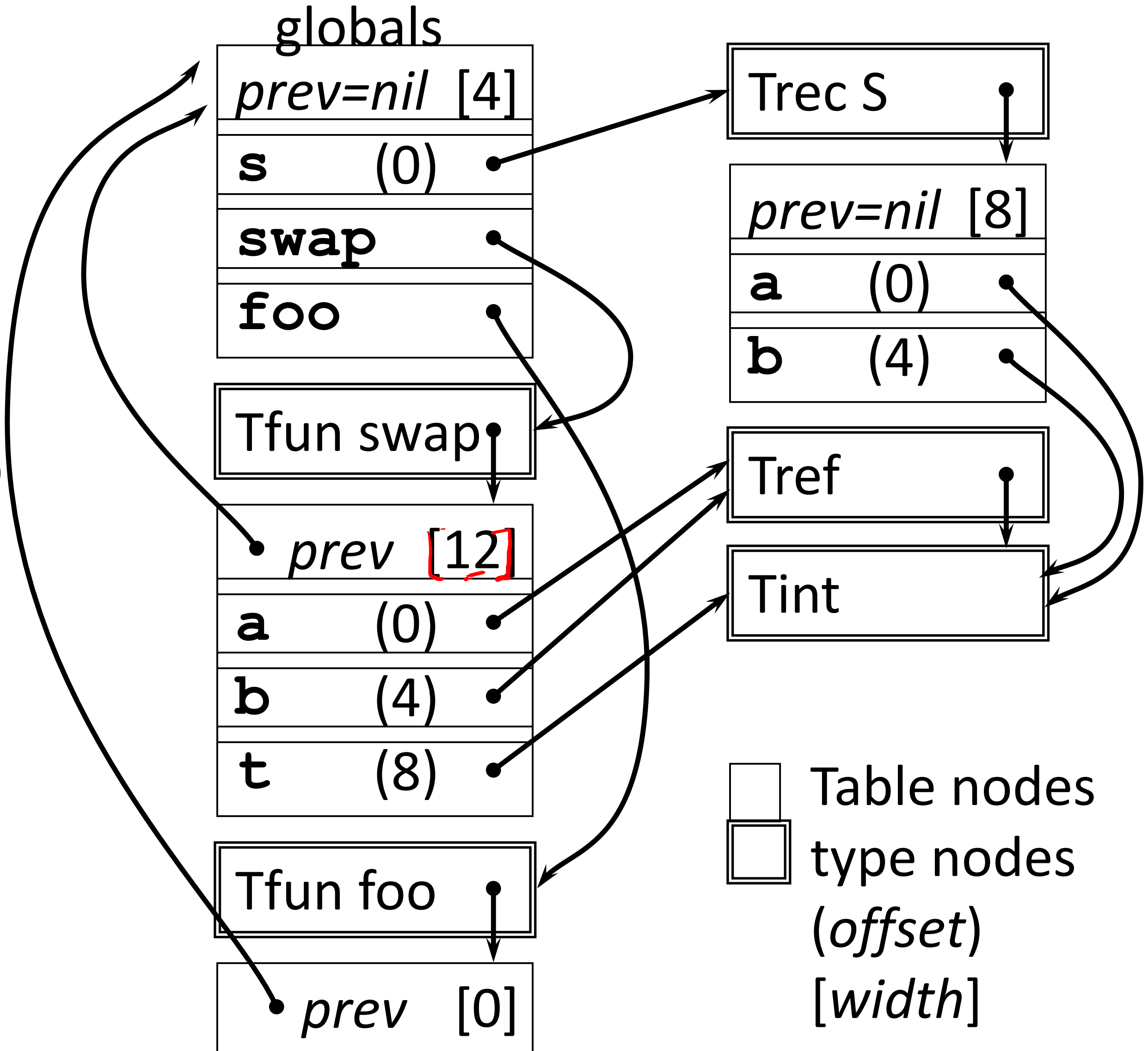


Symbol Table functions

- enterproc(table, name, newtable) creates a new entry in table for procedure with local scope newtable
  - table – existing table, name of the newtable, address of the new table
- lookup(table, name) returns a pointer to the entry in the table for name by following linked tables

Example

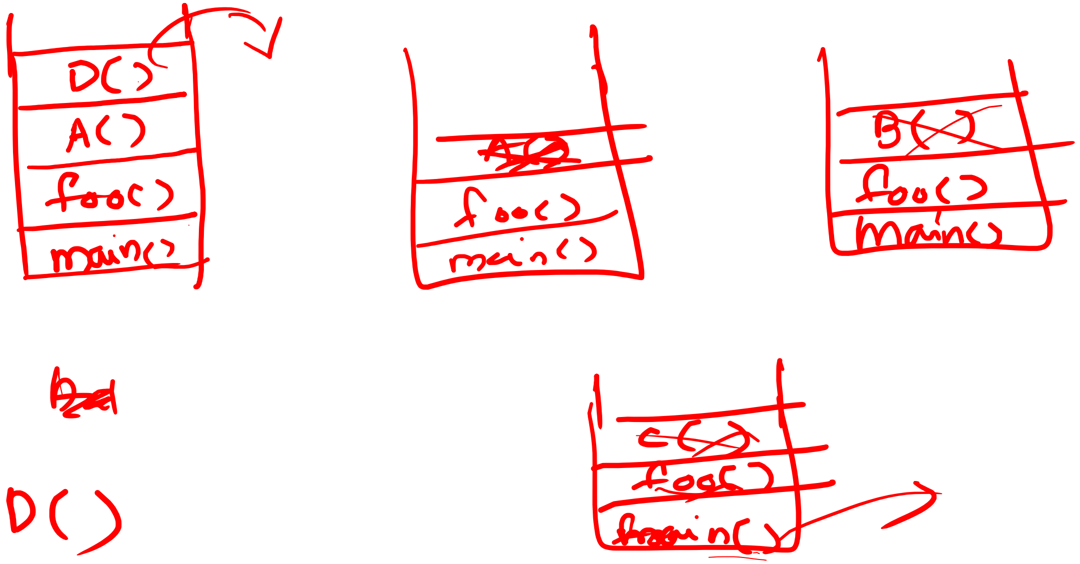
```
struct S
{ int a;
  int b;
} s;

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



Example

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

Example function call

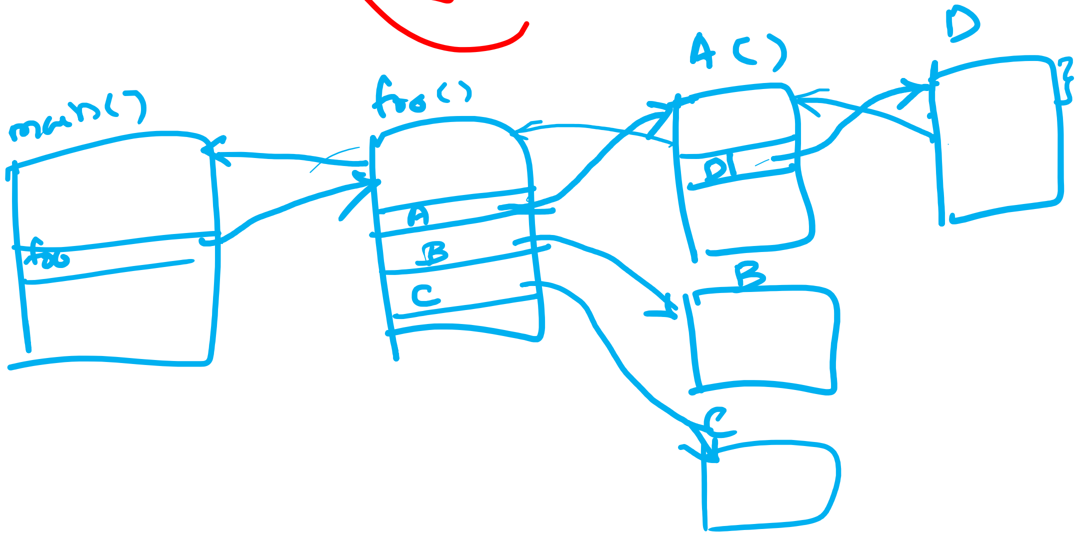



```
Void foo( )
{        call A()
         call B()
         call (C)
}
```

```
Void A()
{
              call D()
}
```

```
void D( )
{
}
```

```
main ( )
{
   foo( )
}
```



Calling stack

D
A
foo
Main

B
foo
Main

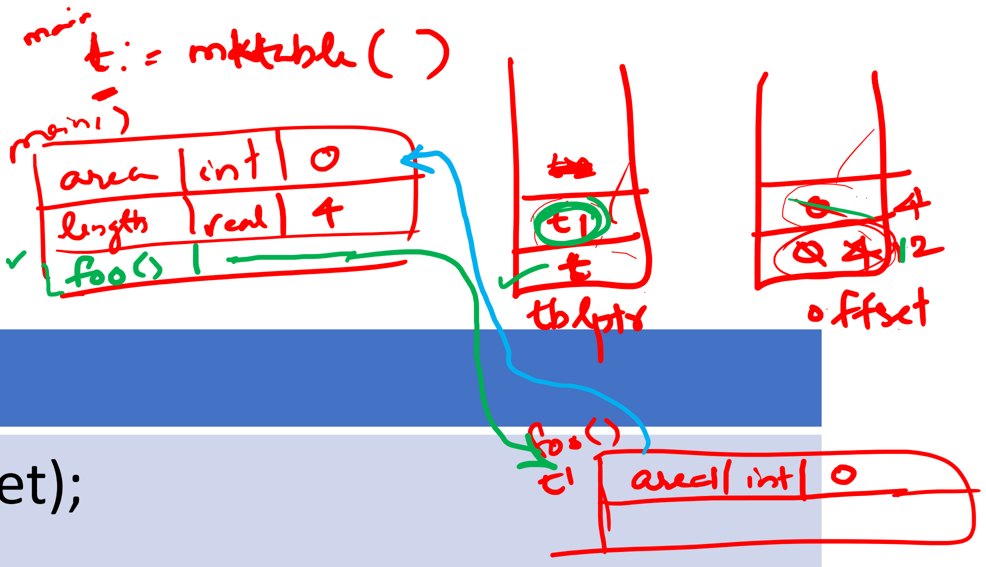
C
foo
Main

foo
Main

foo ( ) { area1 : int }

main( ) { area : int ; length : real ; }

Symbol Table semantic rules



Productions	Semantic Rule
P → M D ; S	{addwidth (top(tblptr), top(offset); pop(tblptr); pop(offset)}
M → ε	{ t := mktable(nil); push(t, tblptr); push(0, offset) }
D → **id** : T	{ enter(top(tblptr), **id**.name, T.type, top(offset)); top(offset) := top(offset) + T.width }
D → **proc id** ; N D_1 ; S	{ t := top(tblptr); addwidth(t, top(offset)); pop(tblptr); pop(offset); enterproc(top(tblptr), **id**.name, t) }

t_1 := mktable(t)

Symbol Table creation

Productions	Semantic Rule
N → ε	{ t_1:= mktable(top(tblptr)); push(t_1, tblptr); push(0, offset) }
D → D_1 **;** D_2	
T → **integer**	{ T.type := 'integer'; T.width := 4 }
T → **real**	{ T.type := 'real'; T.width := 8 }
T → **array [num] of** T_1	{ T.type := array(**num**.val, T_1.type); T.width := **num**.val * T_1.width }
T → **^** T_1	{ T.type := pointer(T_1.type); T.width := 4 }

Symbol table tracking

- Stack of tblptr is available to keep track of the available symbol table
- When a new procedure is called a symbol table is created and its pointer pushed to this stack with offset
- When the call terminates this tblptr is popped

Declarations and Records in Pascal

Production	Semantic rule
T $\rightarrow$ **record L** D **end**	{ T.type := record(top(tblptr)); T.width := top(offset); addwidth(top(tblptr), top(offset)); pop(tblptr); pop(offset) }
L $\rightarrow$ ε	{ t := mktable(nil); push(t, tblptr); push(0, offset) }

SDT's of Statements

- $S \rightarrow S$ **;** 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) }

Assignment statements

- Names in the symbol table
  - Variables referred to addresses
  - Lookup(id.name) identifies variable id in symbol table
    - Emit – used to emit three address statements to output file

Translation scheme

Production	**Semantic Rules**
S → id := E;	{p = lookup(id.name); If p ≠ nil then emit (p ':=' E.place) else error
E → E1 + E2	{E.place= newtemp(); emit (E.place ':=' E1.place '+' E2.place)}
E → E1*E2	{E.place= newtemp(); emit (E.place ':=' E1.place '*' E2.place)}
E → - E1	{E. place = newtemp(); emit(E.place ':=' 'uminus' E1.place)}
E → (E1)	{E.place = E1. place}
E → id	{p := lookp(id.name); if p ≠ nil then E.place :=p else error}

Reusing Temporary Names

$E_1 + E_2$ $\xrightarrow{\text{generate}}$

Evaluate E_1 into **`t1`**
Evaluate E_2 into **`t2`**
`t3 := t1 + t2` $\Rightarrow$ t1 := t1 + t2

If **`t1`** no longer used, can reuse **`t1`**
instead of using new temp **`t3`**

Modify *newtemp*() to use a “stack”:
Keep a counter *c*, initialized to 0
newtemp() increments *c* and returns temporary **\$***c*
Decrement counter on each use of a **\$***i* in a three-address statement

Reusing temporary name

x := a * b + c * d - e * f

Statement	*c*
	0
`$0 := a * b`	1
`$1 := c * d`	2
`$0 := $0 + $1`	1
`$1 := e * f`	2
`$0 := $0 - $1`	1
`x  := $0`	0