Intermediate Code Generation

Chapter 7

Outline

- Intermediate code generation
  - Intermediate Code for Code Generation
  - Basic Code Generation Techniques
  - Code Generation of Control Statements and Logical Expressions

2 Basic Code Generation Techniques

- Syntax-directed translation
  - Intermediate Code or Target Code as a Synthesized Attribute
  - Code generation can be viewed as an attribute computation.
  - Intermediate code can be generated by a post-order traversal of the syntax tree
  - Intermediate code can be generated during parsing

TAC generation for expressions/assignment statements

- Attribute grammar for generating three-address code
  - Attribute
    - tacode for three-address code
    - name for temporary name generated for intermediate results in expressions
  - Symbol for string concatenation
    - || is used for string concatenation with a newline
    - ++ is used for string concatenation with a space
  - Function
    - newtemp( ) :return a new temporary name

TAC generation for expressions/assignment statements

Example

Given the grammar of simple expressions and assignment statements, how code can be defined as a synthesized attribute

```
exp    -> id=exp | aexp
aexp  -> aexp+factor | factor
factor -> (exp) | num | id
```

- Tokens id and num are assumed to have a precomputed attribute strval that is the string value of the token

Grammar Rule	Semantic Rules

exp1 -> id=exp2	exp1.name=exp2.name
	exp1.tacode=exp2.tacode|| id.strval++“=”++exp2.name / gen(id.strval“=”exp2.name)

exp -> aexp	exp.name=aexp.name
	exp.tacode=aexp.tacode

aexp1-> aexp2+factor	aexp1.name=newtemp()
	aexp1.tacode=aexp2.tacode ||factor.tacode || aexp1.name ++ ”=” ++aexp2.name ++ ”+” ++factor.name

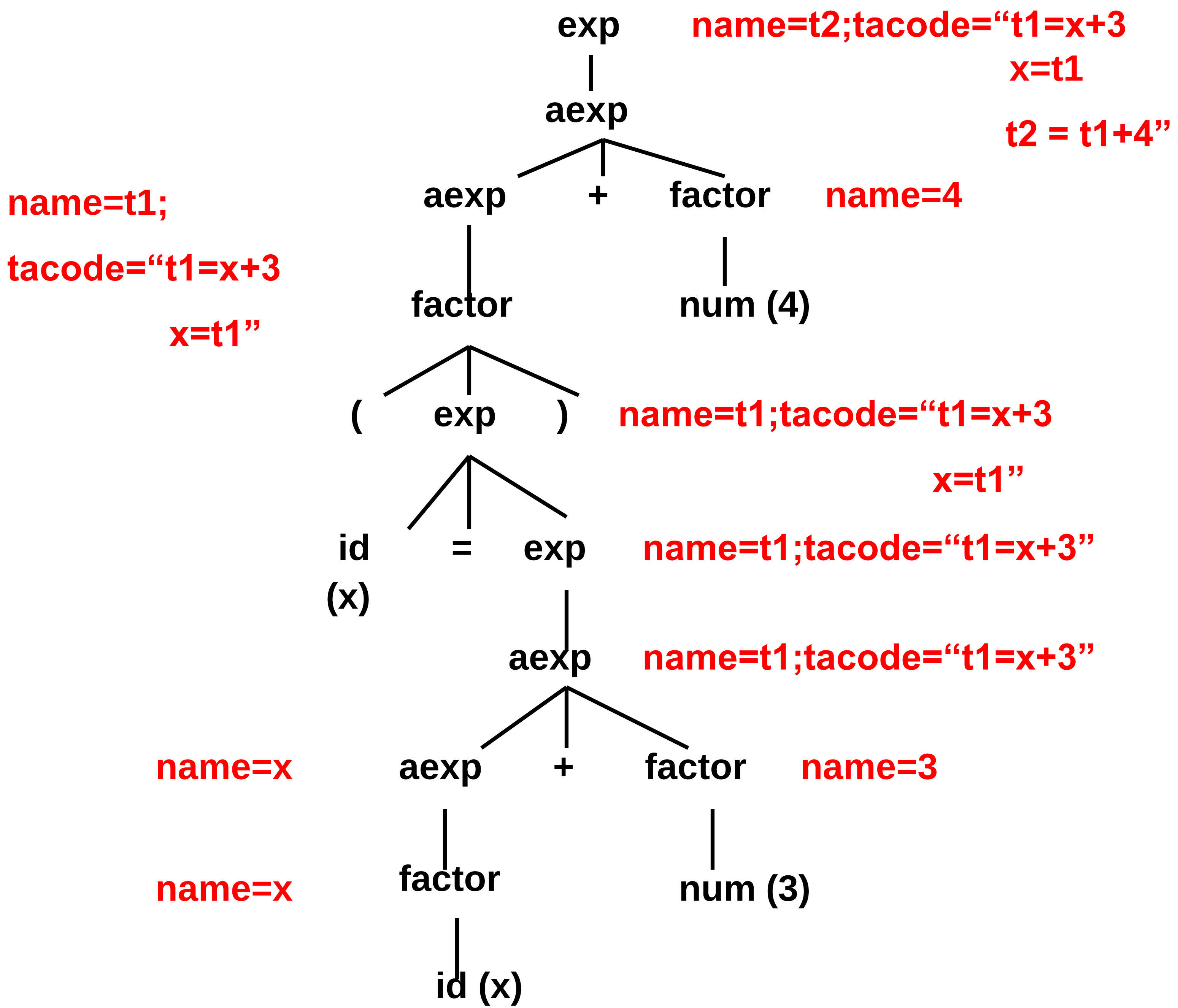
aexp -> factor	aexp.name=factor.name
	aexp.tacode=factor.tacode

factor -> (exp)	factor.name=exp.name
	factor.tacode=exp.tacode

factor -> num	factor.name=num.strval
	factor.tacode=“ ”

factor -> id	factor.name=id.strval
	factor.tacode=“ ”

tacode attribute of expression “(x=x+3)+4”



exp name=t2;tacode=“t1=x+3
x=t1

t2 = t1+4”

aexp

name=t1;
tacode=“t1=x+3
x=t1” aexp + factor name=4

factor num (4)

(exp) name=t1;tacode=“t1=x+3
x=t1”

id (x) = exp name=t1;tacode=“t1=x+3”

aexp name=t1;tacode=“t1=x+3”

name=x aexp + factor name=3

name=x factor num (3)

id (x)

Code generation for individual language constructs

- A program consists of declarations and statements
  - Declarations do not generate intermediate codes, for each declared name, we create a symbol-table entry
  - Basic code generation for assignment and simple arithmetic expressions including Array reference (2.)
  - Code generation for control statements and boolean expressions (3.)

TAC generation for declarations

- Declarations do not generate intermediate codes, for each declared name, we create a symbol-table entry

Grammar rule	Semantic Rules
decl→type varlist	varlist.type=type.type
type→int	type.type=integer
type→float	type.type=real

varlist1→id,varlist2	insert(id.name,varlist1.type)
	varlist2.type =varlist1.type

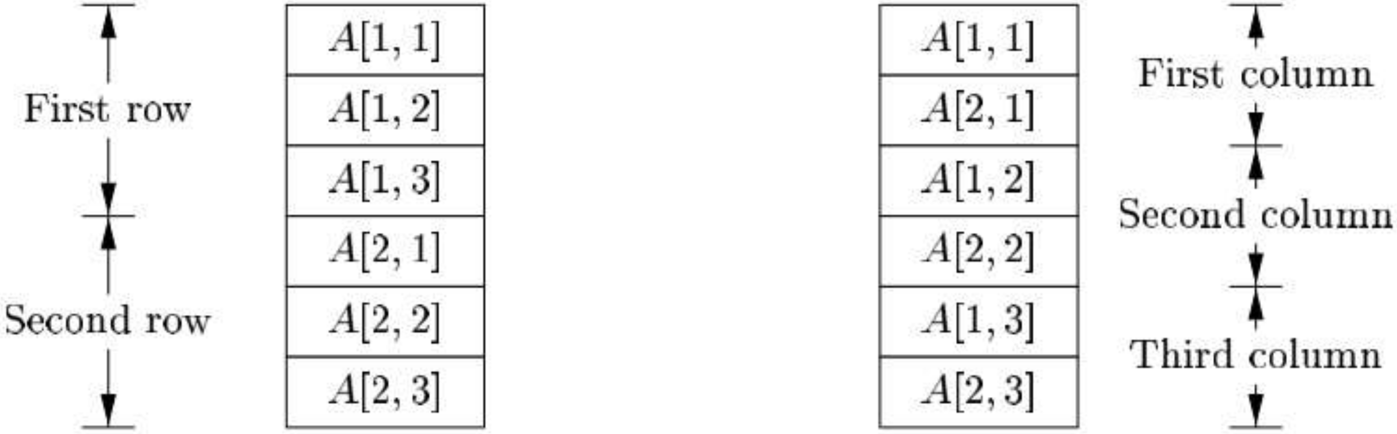
varlist→id	insert(id.name,varlist.type)

TAC generation for **Array reference** in expressions/assignment

- Array elements can be accessed quickly if they are stored in a block of consecutive locations.
- The chief problem in generating code for array references is to relate the address-calculation
- Address-calculation
  - Based on the relative address (***base***) of the storage allocated for the array, layout for the array, and the ***width*** of array elements

TAC generation for **Array reference** in expressions/assignment

- Layout for the array
  - row-major (row-by-row) vs. column-major (column-by-column)



(a) Row Major

(b) Column Major

TAC generation for **Array reference** in expressions/assignment

- TAC generation for Array reference
  - If ***base*** is the relative address of the storage allocated for the array, and the width of each array element is ***w*** with row-major layout, the address-calculation for array reference can be
    - The element of A[i] begins in location base + i * w
    - The element of A[i][j] in A[n][m] may begin in location base + (i*m+j)*w

Attribute Grammar of TAC generation for **Array reference** in expressions

Grammar Rule	Semantic Rules

E → E^1+E^2	E.name=newtemp();
	E.tacode=E^1.tacode ||E^2.tacode ||
	gen(E.name “=” E^1.name “+” E^2.name)

E → id	E.name= id.name E.tacode=“ ”

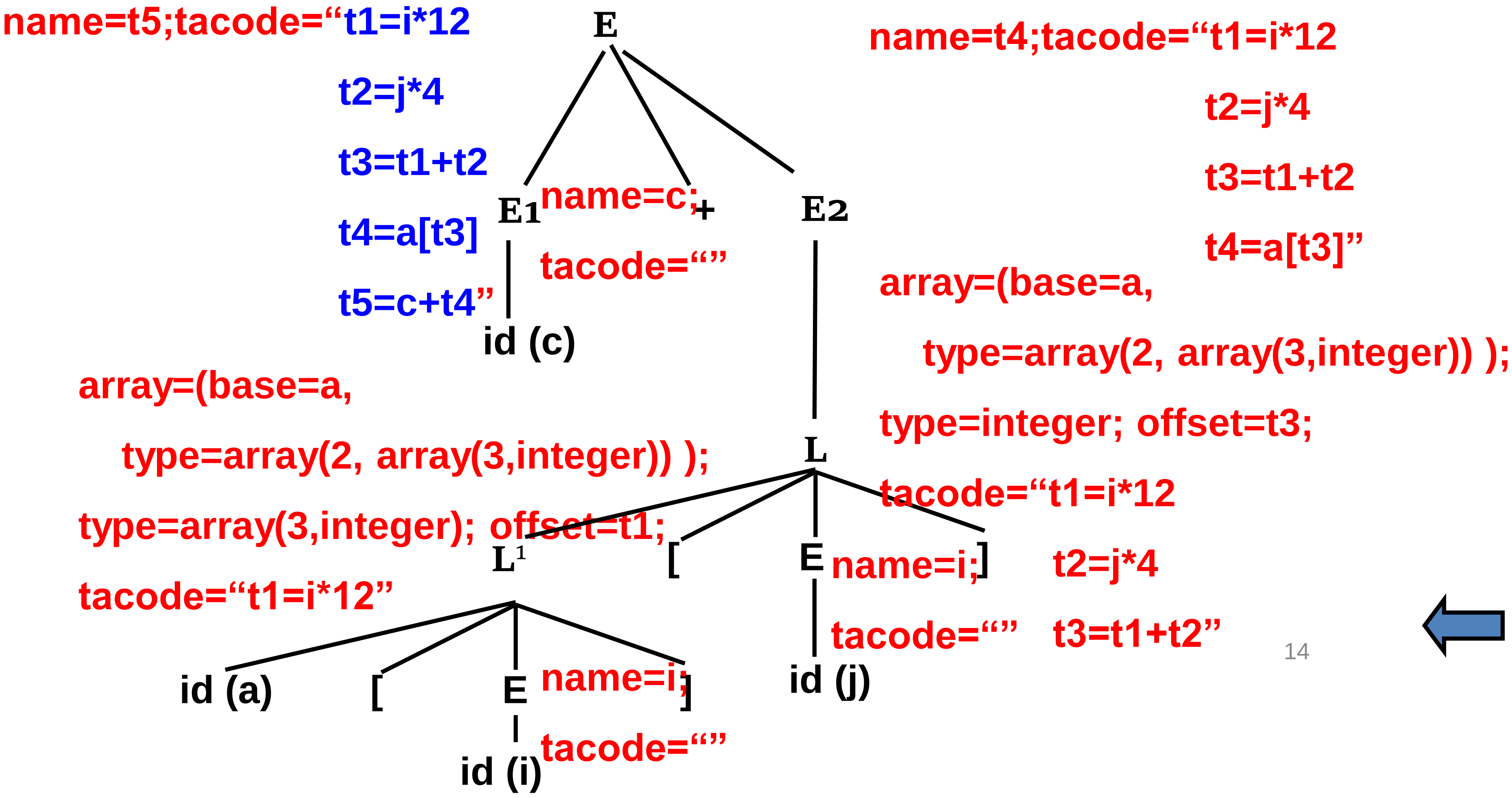
E → L	E.name = newtemp();
	E.tacode=L.tacode ||gen(E.name “=” L.array.base[L.offset])

L → id[E]	L.array = lookup(id.name); L.type = L.array.type.elem;
	L.offset = newtemp();
	L.tacode=E.tacode||gen(L.offset“=”E.name“*”L.type.width)

L → L^1[E]	L.array = L^1.array; L.type = L^1.type.elem;
	t = newTemp(); L.offset = newtemp ();
	L.tacode=L^1.tacode ||E.tacode ||gen(t “=” E.name“*”L.type.width)
	||gen(L.offset “=” L^1.offset “+” t);

TAC generation for **Array reference** in expressions

- Example: c + a[i][j], let c, i, j all denote integer variables, a denote a 2*3 array of integers



name=t5;tacode="t1=i*12
t2=j*4
t3=t1+t2
t4=a[t3]
t5=c+t4"

name=t4;tacode="t1=i*12
t2=j*4
t3=t1+t2
t4=a[t3]"

name=c;
tacode=""

array=(base=a,
type=array(2, array(3,integer)));
type=integer; offset=t3;
tacode="t1=i*12
t2=j*4
t3=t1+t2"

array=(base=a,
type=array(2, array(3,integer)));
type=array(3,integer); offset=t1;
tacode="t1=i*12"

name=i;
tacode=""

name=j;
tacode=""