Intermediate Code Generation

Part II

Translating Expression

Idea: Use Syntax-directed translations

For each expression, we'll synthesize two attributes:

E.code

This is the code we will generate for expression E.

(It is a sequence of all the IR instructions in the translation.)

When executed (at runtime), these instructions will compute the value of the expression and place the value into some variable.

E.place

The name of the variable (often a temporary variable) into which this code will move the final result value when executed.

For each statement, we will synthesize one attribute:

S.code

The IR code for this source statement.

Call "NewTemp" to create a new temporary variable

```
t = NewTemp();
```

Call "IR" to create a new 3-address instruction

```
IR (t, ":=", x, "+", y)  ⇒  "t := x + y"
```

Goal

Take a source statement and produce a sequence of IR quads:

Example:

```
x := y + z;
```

IR Quads:

```
t1 := y + z
x := t1
```

Example:

```
x := (y + z) * (u + v);
```

IR Quads:

```
t1 := y + z
t2 := u + v
t3 := t1 * t2
x := t3
```

Synthesized Code and Place Attributes

Consider

$E_0 \rightarrow E_1 + E_2$

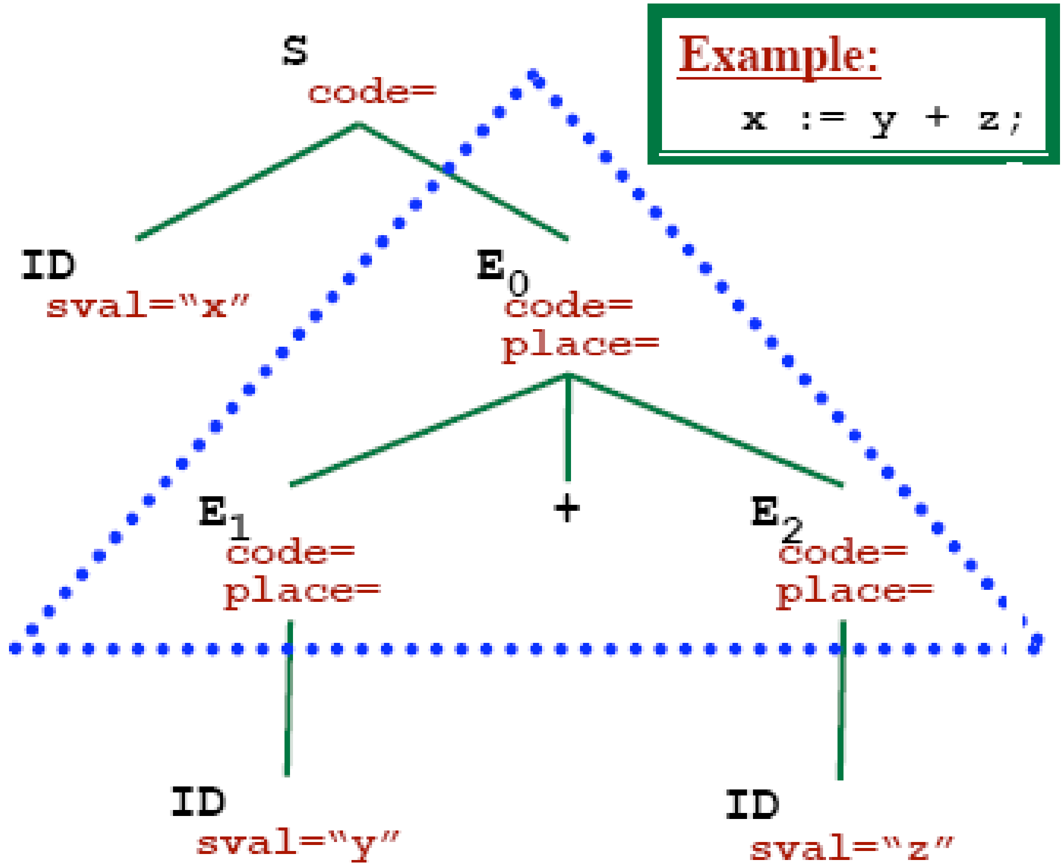
Work bottom-up.

Assume we already have

E_1.place = "y"
E_1.code = " "
E_2.place = "z"
E_2.code = " "

Then, use the rules to compute

E_0.place
E_0.code



Evaluating the attributes for code generation

$E_0 \rightarrow E_1 + E_2$

$E_0.place := NewTemp\ ()$
$E_0.code := E_1.code\ \|\ E_2.code\ \|$
$\quad IR\ (E_0.place,\ \text{':='},\ E_1.place,\ \text{'+'},\ E_2.place)$

Assume we already have

$E_1.place = \text{"y"}$
$E_1.code = \text{" "}$
$E_2.place = \text{"z"}$
$E_2.code = \text{" "}$

Then, use the rules to compute

$E_0.place = \text{"t1"}$
$E_0.code = \text{"t1 := y + z"}$

Three-address code for expression

$E_0 \rightarrow E_1 + E_2$	E_0.place := NewTemp () E_0.code := E_1.code ‖ E_2.code ‖ IR (E_0.place, ':=', E_1.place, '+', E_2.place)
$E_0 \rightarrow E_1 * E_2$	E_0.place := NewTemp () E_0.code := E_1.code ‖ E_2.code ‖ IR (E_0.place, ':=', E_1.place, '*', E_2.place)
$E_0 \rightarrow$ **ID**	E_0.place := ID.svalue E_0.code := " "
$E_0 \rightarrow$ - E_1	E_0.place := NewTemp () E_0.code := E_1.code ‖ IR (E_0.place, ':=', '-', E_1.place)
$E_0 \rightarrow$ (E_1)	E_0.place := E_1.place E_0.code := E_1.code
S $\rightarrow$ **ID** := E ;	S.code := E.code ‖ IR (ID.svalue, ':=', E.place)

Incremental Translation

- ‘code’ attribute can be long string
- Instead of building up ‘E.code’
  - We arrange to generate new three-address instructions
  - ‘code’ attribute is not used
  - ‘gen’ method is used instead of ‘IR’
    - ‘gen’ constructs a three address instruction and appends it to the sequence of instructions generated so far

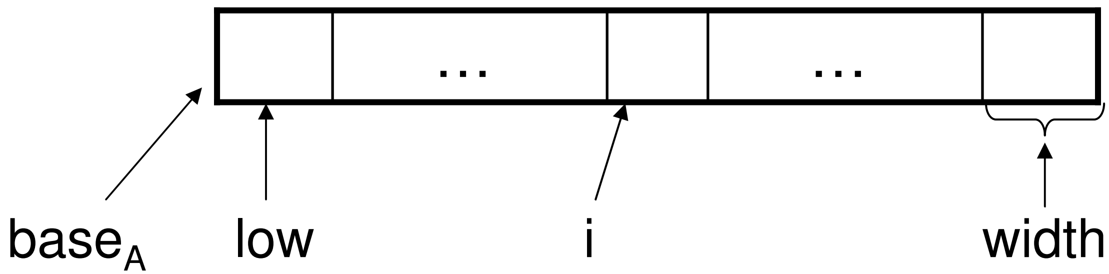
Syntax-Directed Translation into Three-Address Code

$S \rightarrow$ **id** := E	{gen(ID.svalue ':=' E.place)}
$E \rightarrow E_1 + E_2$	{E.place := NewTemp();
	gen(E.place ':=' E_1.place '+' E_2.place)}
$E \rightarrow E_1 * E_2$	{E.place = NewTemp();
	gen(E.place ':=' E_1.place '*' E_2.place ',')}
$E \rightarrow - E_1$	{E.place = NewTemp();
	gen(E.place ':=' '**minus**' E_1.place)}
$E \rightarrow (E_1)$	{E.place = E_1.place;}
$E \rightarrow$ **id**	{E.place = **ID**.svalue;}

Addressing Array Elements

- Elements of arrays can be accessed quickly if the elements are stored in a block of consecutive locations.

A one-dimensional array **A**:



$base_A$ is the address of the first location of the array A,

width is the width of each array element.

low is the index of the first array element

location of A[i] ➔ $base_A$+(i-low)*width

Addressing Array Elements (cont.)

$\texttt{base}_A\texttt{+(i-low)*width}$

can be re-written as

$$\underbrace{\texttt{i*width}}_{\text{should be computed at run-time}} + \underbrace{\texttt{(base}_A\texttt{-low*width)}}_{\text{can be computed at compile-time}}$$

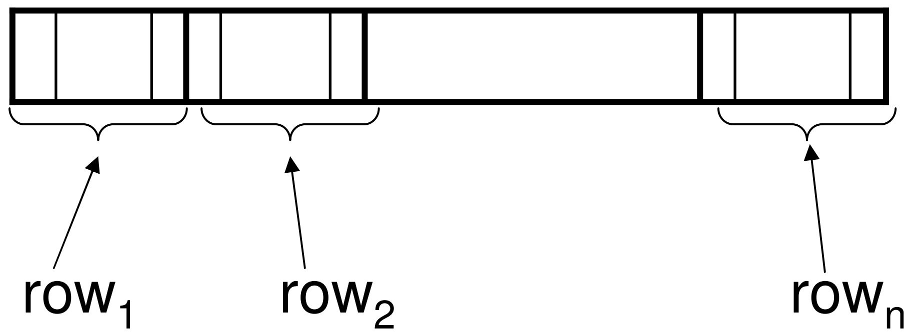
- So, the location of A[i] can be computed at the run-time by evaluating the formula i*width+c where c is ($base_A$-low*width) which is evaluated at compile-time.

- Intermediate code generator should produce the code to evaluate this formula i*width+c (one multiplication and one addition operation).

Two-Dimensional Arrays

- A two-dimensional array can be stored in
  - either **row-major** (*row-by-row*) or
  - **column-major** (*column-by-column*).
- Most of the programming languages use **row-major** method.

- Row-major representation of a two-dimensional array:



Two-Dimensional Arrays (cont.)

- The location of $A[i_1,i_2]$ is

$$base_A + ((i_1 - low_1)*n_2 + i_2 - low_2)*width$$

base$_A$ is the location of the array A.
low$_1$ is the index of the first row
low$_2$ is the index of the first column
n$_2$ is the number of elements in each row
width is the width of each array element

- Again, this formula can be re-written as

$$\overbrace{((i_1*n_2)+i_2)*width} + \overbrace{(base_A-((low_1*n_1)+low_2)*width)}$$

should be computed at run-time

can be computed at compile-time

Multi-Dimensional Arrays

- In general, the location of $A[i_1,i_2,...,i_k]$ is

  $((\ ...\ ((i_1*n_2)+i_2)\ ...)*n_k+i_k)*width\ +\ (base_A-((...((low_1*n_1)+low_2)...)*n_k+low_k)*width)$

- So, the intermediate code generator should produce the codes to evaluate the following formula (to find the location of $A[i_1,i_2,...,i_k]$) :

  $((\ ...\ ((i_1*n_2)+i_2)\ ...)*n_k+i_k)*width\ +\ c$

- To evaluate the $((\ ...\ ((i_1*n_2)+i_2)\ ...)*n_k+i_k$ portion of this formula, we can use the recurrence equation:

  $e_1 = i_1$

  $e_m = e_{m-1} * n_m + i_m$

Translation of Array Elements

- One dimensional

  $$\text{base} + i \times w$$

  w: width of each array element

- Two Dimensional

  $$\text{base} + i_1 \times w_1 + i_2 \times w_2$$

  w_1: width of a row

  w_2: width of an element in a row

- k dimensional (generalized)

  $$\text{base} + i_1 \times w_1 + i_2 \times w_2 + \ldots + i_k \times w_k$$

Translation of Array References

- Need to relate the address calculation formulas to a grammar for array references
- Consider the non-terminal L to generate an array

$$L \rightarrow L\ [E]\ |\ id\ [E]$$

- Nonterminal L has three synthesized attributes
  - L.addr denotes a temporary used to compute the offset for array reference $i_j \, x \, w_j$
  - L.array is a pointer to the symbol-table entry
    - L.array.base is used to determine the actual l-value of it
  - L.type is the type of sub-array generated by L
    - L.type.width gives the width of the type

Syntax-Directed Translation into Three-Address Code

$S \rightarrow$ **id** := E;	{gen(**ID**.svalue ‘:=’ E.place)}
| L := E;	{gen(L.array.base ‘[’ L.addr ‘]’ ‘:=’ E.place)}
$E \rightarrow E_1 + E_2$	{E.place := NewTemp();
	gen(E.place ‘:=’ E_1.place ‘+’ E_2.place)}
| id	{E.place = **ID**.svalue;}
| L	{E.place = NewTemp();
	gen(E.place ‘:=’ L.array.base ‘[’ L.addr ‘]’);}
$L \rightarrow$ id [E]	{L.array = ID.svalue;
	L.type = L.array.type.elem;
	L.addr = NewTemp();
	gen(L.addr ‘:=’ E.place ‘*’ L.type.width)}
| L_1 [E]	{L.array = L_1.array;
	L.type = L_1.type.elem;
	t = NewTemp();
	L.addr = NewTemp();
	gen(t ‘:=’ E.place ‘*’ L.type.width);
	gen(L.addr ‘:=’ L_1.addr ‘+’ t)}