Chapter 6

Intermediate Code Generation

Intermediate Code Generation

- In a compiler, the front end translates a source program into an intermediate representation, and the back end generates the target code from this **intermediate representation.**
- The use of a machine **independent** intermediate code (IC) is:
  - *retargeting* to another ***machine*** is facilitated
  - the *optimization* can be ***done on the machine*** independent ***code***
- Intermediate codes are machine *independent* codes, but they are close to machine instructions.
- The given program in a source language is converted to an equivalent program in an intermediate language by the **intermediate code generator**.

Intermediate Languages

- Intermediate languages can be many different languages, and the **designer** of the ***compiler decides*** this **intermediate language**.
  - syntax trees can be used as an intermediate language.
  - postfix notation can be used as an intermediate language.
  - three-address code (Quadraples) can be used as an ***intermediate language***
    - we will use quadraples to discuss intermediate code generation
    - quadraples are close to machine instructions, but they are not actual machine instructions.
- some **programming languages** have well defined **intermediate languages**.
  - java – java virtual machine
  - prolog – warren abstract machine
  - In fact, there are byte-code emulators to execute instructions in these intermediate languages.

Three-Address Code (Quadraples)

- A quadraple is:

  `x := y op z`

  where x, y and z are names, constants or compiler-generated temporaries; **op** is any operator.

- But we may also use the following notation for quadraples (much better notation because it looks like a machine code instruction)

  `op y,z,x`

  apply operator `op` to `y` and `z`, and store the result in `x`.

- We use the term "three-address code" because each statement usually contains three addresses (two for operands, one for the result).

Three-Address Statements

Binary Operator: `op y,z,result` or `result := y op z`

where `op` is a binary arithmetic or logical operator. This binary operator is applied to `y` and `z`, and the result of the operation is stored in `result`.

Ex:

```
add  a,b,c
gt   a,b,c
addr a,b,c
addi a,b,c
```

Unary Operator: `op y,,result` or `result := op y`

where `op` is a unary arithmetic or logical operator. This unary operator is applied to `y`, and the result of the operation is stored in `result`.

Ex:

```
uminus    a,,c
not       a,,c
inttoreal a,,c
```

Three-Address Statements (cont.)

Move Operator: `mov y,,result` or `result := y`

where the content of `y` is copied into `result`.

Ex:
```
mov   a,,c
movi  a,,c
movr  a,,c
```

Unconditional Jumps: `jmp ,,L` or `goto L`

We will jump to the three-address code with the label `L`, and the execution continues from that statement.

Ex:
```
jmp   ,,L1     // jump to L1
jmp   ,,7      // jump to the statement 7
```

Three-Address Statements (cont.)

Conditional Jumps: `jmp`***`relop`*** `y,z,L` or `if y` ***`relop`*** `z goto L`

We will jump to the **three-address code** with the label `L` if the result of `y` **`relop`** `z` is true, and the execution continues from that statement. If the result is false, the execution continues from the statement following this **conditional** jump statement.

Ex:

```
jmpgt   y,z,L1    // jump to L1 if y>z
jmpgte  y,z,L1    // jump to L1 if y>=z
jmpe    y,z,L1    // jump to L1 if y==z
jmpne   y,z,L1    // jump to L1 if y!=z
```

Our relational operator can also be a unary operator.

```
jmpnz   y,,L1     // jump to L1 if y is not zero
jmpz    y,,L1     // jump to L1 if y is zero
jmpt    y,,L1     // jump to L1 if y is true
jmpf    y,,L1     // jump to L1 if y is false
```

Three-Address Statements (cont.)

Procedure Parameters: `param x,,` or `param x`

Procedure Calls: `call p ,n,` or `call p , n`

where `x` is an **actual parameter**, we invoke the procedure `p` with `n` parameters.

Ex: `param` x_1`,,`

`param` x_2`,,`

➔ `p(`x_1`,...,`x_n`)`

`param` x_n`,,`

`call p,n,`

```
f(x+1,y)  ➔   add   x,1,t1
              param t1,,
              param y,,
              call  f,2,
```

Three-Address Statements (cont.)

Indexed Assignments:

```
move y[i],,x   or   x := y[i]
move x,,y[i]   or   y[i] := x
```

Address and Pointer Assignments:

```
moveaddr y,,x   or   x := &y
movecont y,,x   or   x := *y
```

Syntax-Directed Translation into Three-Address Code

- Syntax directed **translation** can be used to generate the **three-address code**
- Generally, either the three-address code is generated as an attribute of the attributed parse tree or the *semantic actions* have side effects that write the three-address code statements in a file
- When the *three-address code* is generated, it is often necessary to use *temporary variables* and *temporary names*.

Contd…

- The following functions are used:
  - **newtemp** - each time this function is called, it gives distinct names that can be used for temporary variables
  - **newlabel** - each time this function is called, it gives distinct names that can be used for label names
  - In addition, for convenience, we use the notation **gen** to create a **three-address code** from a **number of strings.**
  - **gen** will produce a three-address code after concatenating all the parameters
  - For example, if id1.lexeme = x, id2.lexeme =y and id3.lexeme = z: **gen (id1.lexeme, ':=', id2.lexeme, '+', id3.lexeme)** will produce the three-address code: x := y + z
- *Note:-variables and attribute values are evaluated by gen before being concatenated with the other parameters*

Ex: TAC for assignment Statement and Expression

$S \rightarrow \mathbf{id} := E$	S.code = E.code || gen('mov' E.place ',,' id.place)
$E \rightarrow E_1 + E_2$	E.place = newtemp();
	E.code = E_1.code || E_2.code || gen('add' E_1.place ',' E_2.place ',' E.place)
$E \rightarrow E_1 * E_2$	E.place = newtemp();
	E.code = E_1.code || E_2.code || gen('mult' E_1.place ',' E_2.place ',' E.place)
$E \rightarrow - E_1$	E.place = newtemp();
	E.code = E_1.code || gen('uminus' E_1.place ',,' E.place)
$E \rightarrow (E_1)$	E.place = E_1.place;
	E.code = E_1.code
$E \rightarrow \mathbf{id}$	E.place = **id**.place;
	E.code = null

- the attribute *place* will hold the value of the grammar symbol
- the attribute *code* will hold the sequence of three-address statements evaluating the grammar symbol
- The function newtemp returns a sequence of distinct names t1, t2, . . . in response to successive calls

Example: TAC for While Statement

$S \rightarrow$ while E do S_1 S.begin = newlabel();

S.after = newlabel();

S.code = gen(S.begin ":") || E.code ||

gen('jmpf' E.place ',,' S.after) || S_1.code ||

gen('jmp' ',,' S.begin) ||

gen(S.after ':")

The above semantic action will create the three-address code of the following form:

```
L1 :
    E.code
    if E.place = 0 goto L2
        S1.code
    goto L1
L2:
```

Declarations

- The declaration is used by the compiler as a source of type-information that it will store in the symbol table
- While processing the declaration, the ***compiler reserves memory area*** for the variables and stores the relative address of each variable in the symbol table
- The relative address consists of an address from the static data area
- We use in this section a number of variables, attributes and procedure that help the processing of the declaration
- The compiler maintains a global *offset* variable that indicates the first address not yet **allocated.**
- Initially, offset is assigned 0. Each time an address is allocated to a variable, the offset is incremented by the *width* of the data object denoted by the name
- The procedure ***enter(name, type, address)*** creates a symbol table entry for *name*, give it the type ***type*** and the relative address ***address***
- The synthesized attributes ***name*** and ***width*** for **non-terminal T** are also used to indicate the type and number of memory units taken by objects of that type

Declarations

We consider that an Integer and a pointer occupy 4 bytes and a real number occupies 8 bytes of memory

$P \rightarrow M\ D$

$M \rightarrow €$ { offset=0 }

$D \rightarrow D\ ;\ D$

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

$T \rightarrow$ int { T.type=int; 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 }

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

where *enter* crates a symbol table entry with given values.

Contd…

- Using the **symbol table**, it is possible to generate the three-address code statements corresponding to assignments
- In an earlier example, we have generated three-address code statements where variables are represented by their names.
- However, it is more common and practical for the implementation to *represent the variables by their symbol table entries*
- The function *lookup(lexeme)* checks if there is an entry for this occurrence of the name in the symbol table, and if so a pointer to the entry is returned; otherwise *nil* is returned.
- The *newtemp* function will generate temporary variables and reserve a memory area for the variables by modifying the offset and putting in the symbol table the reserved memories' addresses.

Ex: TAC for assignment Statement and Expression

$S \rightarrow$ **id** := E { p= lookup(id.name);
if (p is not nil) then emit('mov' E.place ',,' p)
else error("undefined-variable") }

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

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

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

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

$E \rightarrow$ **id** { p= lookup(id.name);
if (p is not nil) then E.place = **id**.place
else error("undefined-variable") }

Three Address Codes - Example

```
x:=1;
y:=x+10;
while (x<y) {
    x:=x+1;
    if (x%2==1) then y:=y+1;
    else y:=y-2;
}
```

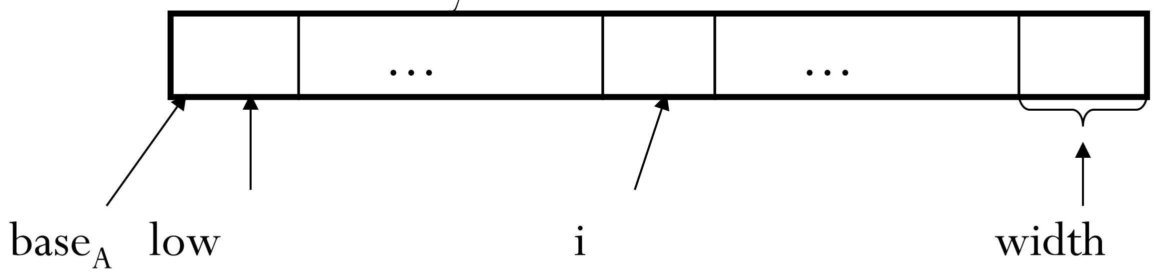
➔

```
01: mov   1,,x
02: add   x,10,t1
03: mov   t1,,y
04: lt    x,y,t2
05: jmpf  t2,,17
06: add   x,1,t3
07: mov   t3,,x
08: mod   x,2,t4
09: eq    t4,1,t5
10: jmpf  t5,,14
11: add   y,1,t6
12: mov   t6,,y
13: jmp   ,,16
14: sub   y,2,t7
15: mov   t7,,y
16: jmp   ,,4
17:
```

Addressing Arrays 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

- For example for an array declared as *A : array [5..10] of integer*; if it is stored at the address 100, A[7] = 100 + (7 – 5) * 4

Arrays (cont.)

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

can be re-written as $\underbrace{\texttt{i*width}} + \underbrace{(\texttt{base}_A\texttt{-low*width})}$

should be computed at run-time   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).

Translation Scheme for Arrays (cont.)

S→L := E	if L.offset = nil then /* L is a simple id */
	S.code := L.code || E.code || gen (L.place, ':=', E.place);
	else
	S.code := L.code || E.code || gen (L.place, '[', L.offset, '] :=', E.place)
E→E1 + E2	E.place := newtemp;
	E.code := E1.code || E2.code || gen (E.place, ':=', E1.place, '+', E2.place)
E→E1 * E2	E.place := newtemp;
	E.code := E1.code || E2.code || gen (E.place, ':=', E1.place, '*', E2.place)
E→- E1	E.place := newtemp;
	E.code := E1.code || gen (E.place, ':= uminus ', E1.place)
E→(E1)	E.place := newtemp;
	E.code := E1.code

Contd…

```
E→L        if L.offset = nil then /* L is simple */
             begin
                E.place := L.place;
                E.code := L.code
             end
           else
             begin
                E.place := newtemp;
                E.code := L.code || gen (E.place, ‘ :=’, L.place, ‘[’ , L.offset, ‘]’)
             end
L→id [E]     L.place := newtemp;
             L.offset := newtemp;
             L.code := E.code || gen (L.place, ‘:=’, base (id.lexeme) – width (id.lexeme) *
             low(id.lexeme)) || gen (L.offset, ‘:=’, E.place, ‘*’, width (id.lexeme));
L→id         p := lookup (id.lexeme);
             if p <> nil then L.place = p.lexeme else error;
             L.offset := nil; /* for simple identifier */
             E.code := ‘’ /* empty code */
```

Translation Scheme for Arrays – Example1

- A one-dimensional `double` array `A : 5..100`
  ➔ n_1=95 width=8 (double) low_1=5

- Intermediate codes corresponding to `x := A[y]`

```
mov     c,,t1            // where c=baseA-(5)*8
mult    y,8,t2
mov     t1[t2],,t3
mov     t3,,x
```

End of ch6…

Thank You

?