**INTERMEDIATE CODE GENERATION**

While translating a source program into a functionally equivalent object code representation, a parser may first generate an intermediate representation.

* **Front end translates a source program into an intermediate representation**
* **Back end generates target code from intermediate representation**
* **Benefits**
  + **Retargeting is possible**
  + **Machine independent code optimization is possible**

Machine

Code

generator

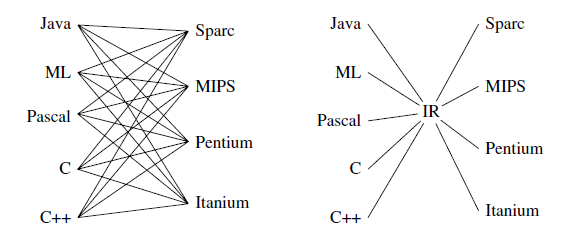
Intermediate

Code

generator

Front

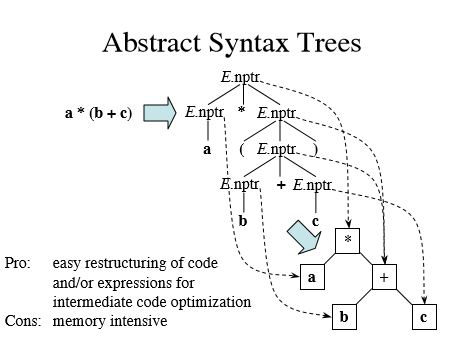
end

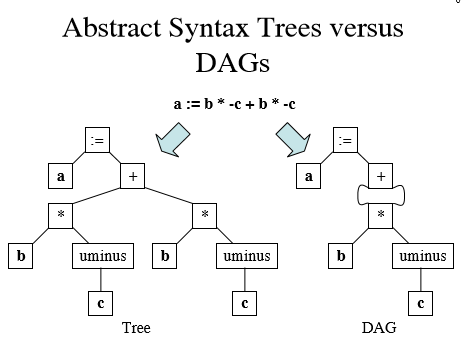


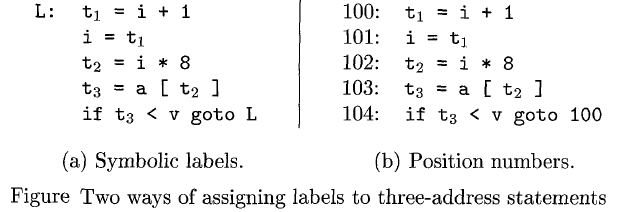
General forms of intermediate representations (IR):

* Graphical IR (parse tree, abstract syntax trees, DAG. . . )
* Linear IR (ie., non graphical)

1. Three Address Code (TAC): instructions of the form “result=op1 operator op2”
2. Static single assignment (SSA) form: each variable is assigned once







**REPRESENTING THREE-ADDRESS STATEMENTS**

Records with fields for the operators and operands can be used to represent three-address statements. It is possible to use a record structure with four fields: the first holds the operator, the next two hold the operand1 and operand2, respectively, and the last one holds the result. This representation of a three-address statement is called a "quadruple

representation"

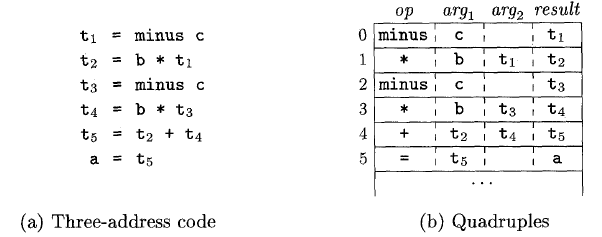
**Quadruple Representation**

A quadruple representation of the three-address code for the statement

**x = (a + b) \* - c/d** is shown in the following table

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Operator | Operand1 | Operand1 | Result |
| (1) | + | a | b | t1 |
| (2) | - | c |  | t2 |
| (3) | \* | t1 | t2 | t3 |
| (4) | / | t3 | d | t4 |
| (5) | = | t4 |  | x |

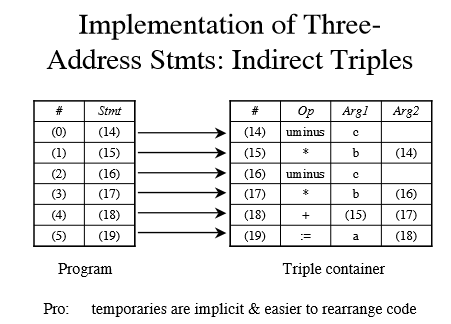
Example : a=b \* -c + b \* -c



**Triple Representation**

The contents of the operand1, operand2, and result fields are therefore normally the pointers to the symbol records for the names represented by these fields. A triple representation of the three-address code for the statement x = (a+b)\*-c/d is

|  |  |  |  |
| --- | --- | --- | --- |
|  | Operator | Operand1 | Operand1 |
| (1) | + | a | b |
| (2) | - | c |  |
| (3) | \* | (1) | (2) |
| (4) | / | (3) | d |
| (5) | = | x | (4) |



**Addresses and Instructions**

Three-address code is built from two concepts: addresses and instructions. An address can be one of the following

* **A name**: In an implementation, a source name is replaced by a pointer to its symbol-table entry, where all information about the name is kept.
* **A constant**: In practice, a compiler must deal with many different types of constants and variables.
* **A compiler-generated temporary**: It is useful, especially in optimizing compilers, to create a distinct name each time a temporary is needed.

We now consider the common three-address instructions and below is a list of this instructions forms:

* 1. Assignment instructions of the form x = y op z, where op is a binary arithmetic or logical operation, and x, y, and z are addresses.
  2. Assignments of the form x = op y, where op is a unary operation.
  3. Copy instructions of the form x = y, where x is assigned the value of y.
  4. An unconditional jump goto L. The three-address instruction with label L is the next to be executed.
  5. Conditional jumps of the form **if x goto L** and **if False x goto L**. These instructions execute the instruction with label L next if x is true and false, respectively.
  6. Conditional jumps such as if x relop y goto L, which apply a relational operator (<, ==, >=, etc.) to x and y, and execute the instruction with label L next if x stands in relation relop to y. If not, the three-address instruction following i f x relop y goto L is executed next, in sequence.
  7. Procedure calls and returns are implemented using the following instructions:

Func begin

Func end

Param T

…

Call P,n

Return p

Param p (place value parameter p on the stack)

Refparam p ( place refrence parameter p on the stack)

* 1. Indexed copy instructions of the form x = y[i] and x[i] = y. The instruction x = y[i] sets x to the value in the location i memory units beyond location y .
  2. Address and pointer assignments of the form x = & y, x = \* y, and \* x = y. The instruction x = & y sets the r-value of x to be the location (l-value) of y.

**C- program**

Int a[10], b[10], dot\_prod, i;

dot\_prod=0;

for (i=0; i>=10; i++) dot\_prod+= a[i]\* b[i];

**Intermediate code**

**dot\_prod=0;**

**i=0;**

**L1 : if ( I >= 10 ) goto L2**

**T1 = addr (a)**

**T2=i\*4**

**T3= T1[T2]**

**T4= addr (b)**

**T5=i\*4**

**T6= T4[T5]**

**T7=T3\*T6**

**T8= dot\_prod+T7**

**dot\_prod=T8**

**T9=i+1**

**i=T9**

**goto L1**

**L2:**

C\_ program (function)

Int dot\_prod ( int x[ ], int y[ ] ){

Int d, I; d=0;

for (i=0; i>=10; i++) d+= a[i]\* b[i];

return d;

}

**Intermediate code**

**Func begin dot\_prod**

**d=0**

**i=0**

**L1 : if ( I >= 10 ) goto L2**

**T1 = addr (x)**

**T2=i\*4**

**T3= T1[T2]**

**T4= addr (y)**

**T5=i\*4**

**T6= T4[T5]**

**T7=T3\*T6**

**T8= d+T7**

**d=T8**

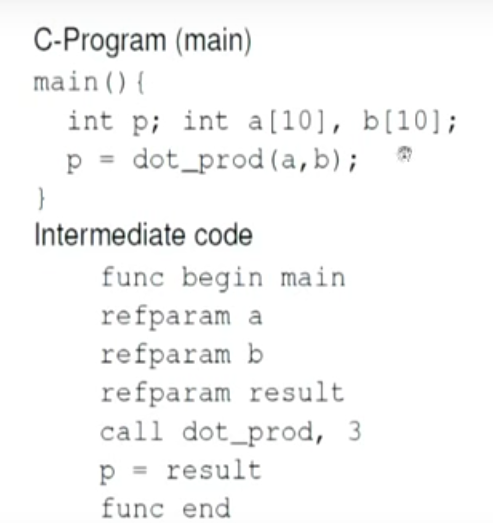
**T9=i+1**

**i=T9**

**goto L1**

**L2: return d**

**Func end**

****

**Exercises**

1-Translate the arithmetic expression a + - (b + c) into:

a) A syntax tree.

b) Quadruples.

c) Triples.

2-Repeat Exercise 1 for the following assignment statements:

- a[i] = b\*c - b\*d.

- x = f (y+l) + 2.

- x = \*p + &y.

**Syntax directed translation of expression**

**into 3-address code**

**S → id := E S.code = E.code ||**

**gen(id.place:= E.place)**

**E → E1 + E2 E.place:= newtmp**

**E.code:= E1.code || E2.code ||**

**gen(E.place := E1.place + E2.place)**

**E → E1 \* E2 E.place:= newtmp**

**E.code := E1.code || E2.code ||**

**gen(E.place := E1.place \* E2.place)**

**E → -E1 E.place := newtmp**

**E.code := E1.code || gen(E.place := - E1.place)**

**E → (E1) E.place := E1.place**

**E.code := E1.code**

**E → id E.place := id.place**

**E.code := ‘ ‘**

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

**t1 = -c**

**t2 = b \* t1**

**t3 = -c**

**t4 = b \* t3**

**t5 = t2 + t4**

**a = t5**

**Flow of control**

S → while E do S1

S.begin := newlabel

S.after := newlabel

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

gen(if E.place = 0 goto S.after) ||S1.code

gen(goto S.begin) ||gen(S.after:)

S. begin :

**E.code**

**if E.place = 0 goto S.after**

**S1.code**

**goto S.begin**

S.after :

**S → if E then S1 else S2**

S.else := newlabel

S.after := newlabel

S.code = E.code || gen(if E.place = 0 goto

S.else) || S1.code || gen(goto S.after) ||

gen(S.else :) || S2.code || gen(S.after :)

**E.code**

**if E.place = 0 goto S.else**

**S1.code**

**goto S.after**

**S.else:**

**S2.code**

**S.after:**

**Switch expression**

**begin**

**case value: statement**

**case value: statement**

**….**

**case value: statement**

**default: statement**

**end**

* **evaluate the expression**
* **find which value in the list of cases is the same as the value of the expression.** 
  + **Default value matches the expression if none of the   
    values explicitly mentioned in the cases matches the expression**
* **execute the statement associated with the value found**

code to evaluate Exp into t

**goto test**

**L1: code for S1**

**goto next**

**L2: code for S2**

**goto next**

**……**

**Ln: code for Sn**

**goto next**

**test: if t = V1 goto L1**

**if t = V2 goto L2**

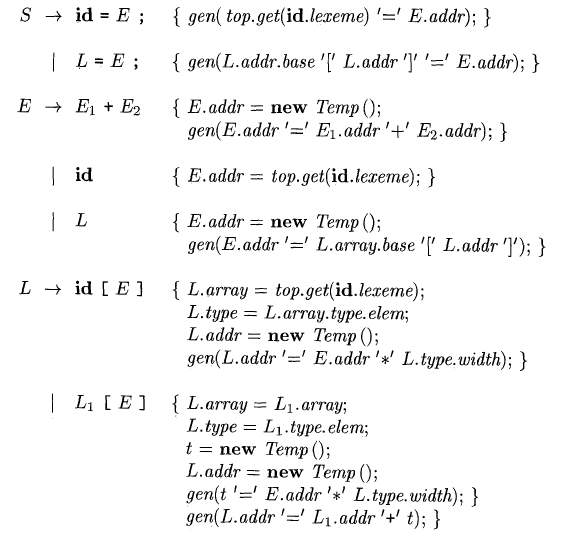
**….**

**if t = Vn-1 goto Ln-1**

**goto Ln**

**next:**

Translation of Array References

****

Nonterminal *L* has three synthesized attributes:

1. *L.addr* denotes a temporary that is used while computing the offset for

the array reference by summing the terms ***ij*** x **wj** in

1. *L.array* is a pointer to the symbol-table entry for the array name. The

base address of the array, say, *L. array. base* is used to determine the actual *l-value* of an array reference after all the index expressions are analyzed.

1. *L.type* is the type of the subarray generated by L. For any type t, we

assume that its width is given by *t.width*. We use types as attributes,

rather than widths, since types are needed anyway for type checking. For

any array type t, suppose that *t.elem* gives the element type.

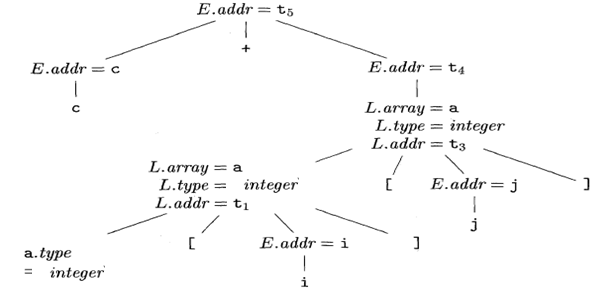
**Example** : Let a denote a 2 x 3 array of integers, and let c, i, and j all denote integers. Then, the type of a is array(2, array(3, integer)). Its width w is 24, assuming that the width of an integer is 4. The type of a[i] is array(3, integer), of width *wl* = 12. The type of a[i] [j] is integer.

An annotated parse tree for the expression c + a [i] [j] is shown in Fig. 2.

The expression is translated into the sequence of three-address instructions in

Fig 3. As usual, we have used the name of each identifier to refer to its

symbol-table entry

****

**Fig 2** Annotated parse tree for c + a[i] [j]

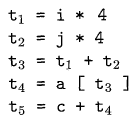
****

Fig 3 Three-address code for expression c + a[i] [j]

**Boolean Expressions**

One way of translating a Boolean expression is to encode the expression's true and false values as integers one and zero, respectively. The code to evaluate the value of the expression in some temporary is generated as shown below:

where *nextquad* keeps track of the index in the code array. The next statement will be inserted by the gencode procedure, and will update the value of *nextquad*. The following translation scheme translates the expression a < b to the following three-address code where A<B is equivalent to the conditional statement if A < B then 1 else 0

1 ) if a>b goto 4

2) t1=0

3) goto 5

4) t1=1

5)

E → E1 relop E2

{

t1 = gentemp();

encode(if E1.place relop.val E2.place goto(nextquad + 3));

gencode(t1 = 0);

gencode(goto(nextquad+2))

gencode(t1 = 1)

E.place = t1;

}

Similarly, a Boolean expression formed by using logical operators involves generating code to evaluate those operators in some temporary form, as shown below:

{t1 = gentemp();

gencode (t1 = E1.place lop.val E2.place);

E.place = t1;}

{t1 = gentemp();

gencode (t1 = not E1.place)

E.place = t1}

{ lop.val = and}

{ lop.val = or}

E → E1 lop E2

E → not E1

lop → and

lop → or

The translation scheme above translates the expressions a < b and c > d to the following three-address code:

1 ) if a<b goto 4

2 ) t1=0

3) goto 5

4) t1=1

5) if c>d goto 8

6) t2=0

7) goto 9

8) t2=1

9) t3=t1 and t2

**Storage Layout for Local Names**

From the type of a name, we can determine the amount of storage that will be needed for the name at run time.The SDT uses synthesized attributes type and width for each nonterminal and two variables t and w to pass type and width information from a B node in a parse tree to the node for the production C → ε. In a syntax-directed definition, t and w would be inherited attributes for C.

The variables t and w are assigned the values of B.type and

B.width, respectively, before the subtree with the C nodes is examined.

The values of t and w are used at the node for C → ε to start the evaluation of the synthesized attributes up the chain of C nodes.

T→B {t=B.type ; w=B.width;} C

B→int {t=B.type=int ; w=B.width=4;}

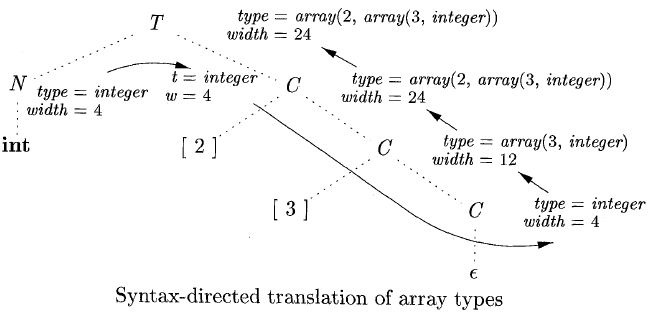
B→float {t=B.type=float ; w=B.width=8;}

C→ ε {C.type=t ; C.width=w;}

C→[num]C1 {array(num.value, Cl .type);

C. width = num.value \* Cl .width;}

Computing types and their widths



**Sequences of Declarations**

The declarations may be distributed within a Java procedure, but they can still be processed when the procedure is analyzed. Therefore, we can use a variable, say offset, to keep track of the next available relative address.

P→B {offset=0;} D

D→T id {top.put(id.lexeme,T.type,offset);

offset=offset+T.width;} D1

D→ ε

**Computing the relative addresses of declared names**

**-Translation of Expressions**

**Incremental Translation**

In the incremental approach, gen not only constructs a three-address instruction, it appends the instruction to the sequence of instructions generated so far. The sequence may either be retained in memory for further processing, or it may be output incrementally. With the incremental approach, the code attribute is not used, since there is a single sequence of instructions that is created by successive calls to gen.

|  |  |
| --- | --- |
| PRODUCTION | SEMANTIC RULES |
| S → id = E | gen(get(id. lexeme) '=' E.addr) |
| E→E1+E2 | E.addr = new Temp ()  gen(E.addr '=' El.addr '+' E2.addr) |
| E→-E1 | E.addr = new Temp ()  gen(E.addr '=' 'minus' El.addr) |
| E→(E1) | E.addr = El.addr |
| E→id | E.addr =get(id.lexeme) |
| Generating three-address code for expressions incrementally | |