Compiler Design

Fatemeh Deldar

Isfahan University of Technology

1402-1403

The Value-Number Method for Constructing DAGs

Algorithm 6.3: The value-number method for constructing the nodes of a DAG.

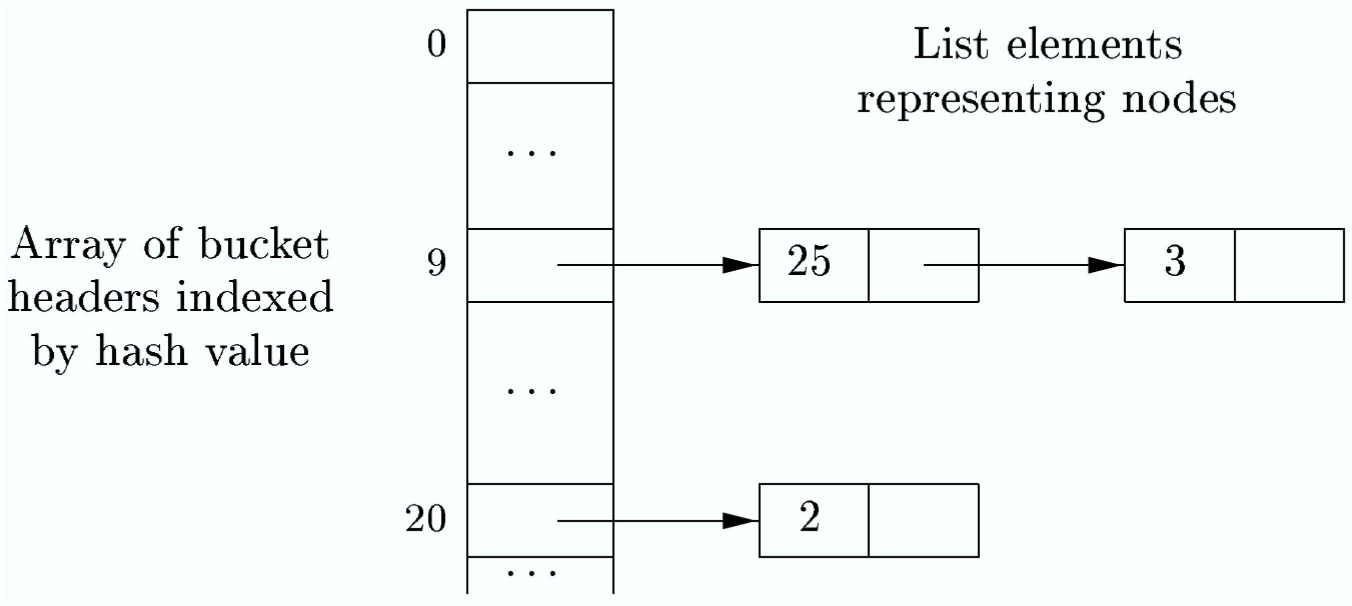
INPUT: Label op, node l, and node r.

OUTPUT: The value number of a node in the array with signature $\langle op, l, r \rangle$.

METHOD: 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 of 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 number. □

The Value-Number Method for Constructing DAGs

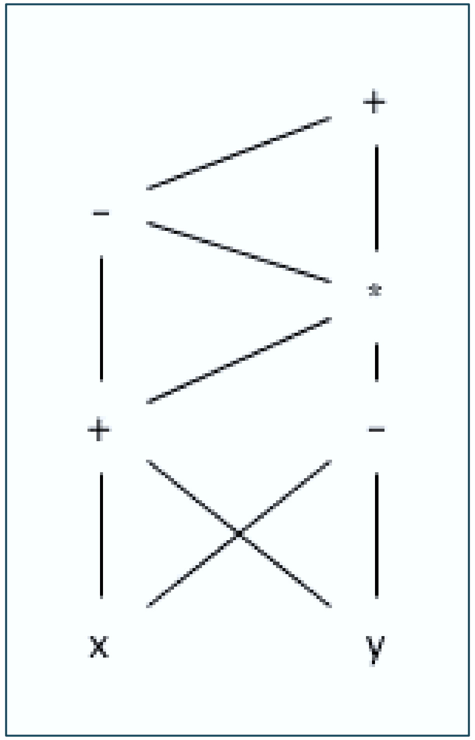
- **Searching the entire array every time we are asked to locate one node is expensive**
- A more efficient approach is to use a **hash table**, in which the nodes are put into buckets, each of which typically will have only a few node



Directed Acyclic Graph

Exercise 6.1.1 : Construct the DAG for the expression

$$((x + y) - ((x + y) * (x - y))) + ((x + y) * (x - y))$$



Directed Acyclic Graph

- **Example**

a+b+(a+b)

1	id	a	
2	id	b	
3	+	1	2
4	+	3	3

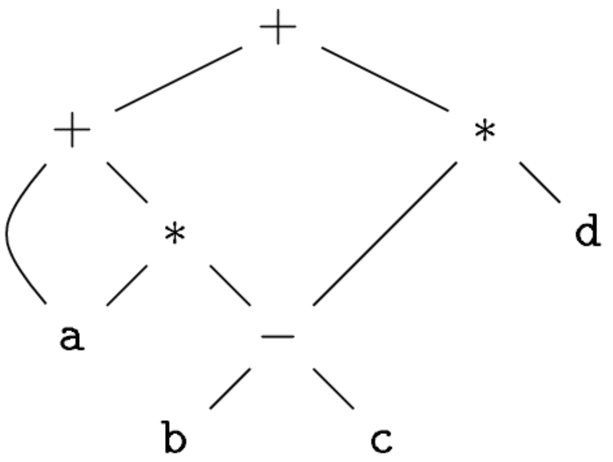
a+b+a+b

1	id	a	
2	id	b	
3	+	1	2
4	+	3	1
5	+	4	2

Three-Address Code

- In three-address code, there is at most one operator on the right side of an instruction
- **Example**
  - $x + y * z$

```
t1 = y * z
t2 = x + t1
```




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

Three-Address Code

- Three-address code is built from two concepts: **addresses** and **instructions**
- **An address can be one of the following**
  - ***Name***
    - We allow source-program names to appear as addresses in three-address code
  - ***Constant***
    - A compiler must deal with many different types of constants and variables
  - ***Compiler-generated temporary***
    - It is useful, especially in optimizing compilers, to create a distinct name each time a temporary is needed

Three-Address Code

- **A list of the common three-address instruction forms**
  1. Assignment instructions of the form **x = y op z**
  2. Assignments of the form **x = op y**, where op is a unary operation
  3. Copy instructions of the form **x = y**
  4. An unconditional **jump goto L**
  5. Conditional jumps of the form **if x goto L** and **if False x goto L**
  6. Conditional jumps such as **if x relop y goto L**
  7. Procedure (Function) calls and returns

     $$\boxed{\begin{array}{l} \texttt{param } x_1 \\ \texttt{param } x_2 \\ \cdots \\ \texttt{param } x_n \\ \texttt{call } p, n \end{array}}$$

  8. Indexed copy instructions of the form **x = y[i]** and **x[i]=y**
  9. Address and pointer assignments of the form **x = &y**, **x = *y**, and ***x = y**

Three-Address Code

- **Example**
  - $\boldsymbol{do\ i = i + 1;\ while\ (a[i] < v);}$

Symbolic Label

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

- **Three representations for three-address code are as follows**
  - Quadruples
  - Triples
  - Indirect triples

Three-Address Code

- **Quadruples**
  - A quadruple has four fields, which we call **op, arg1, arg2, and result**
  - **Some exceptions**
    - Instructions with unary operators like x = minus y or x = y do not use arg2
      - Note that for a copy statement like x = y, op is =, while for most other operations, the assignment operator is implied
    - Operators like param use neither arg2 nor result
    - Conditional and unconditional jumps put the target label in result
- **Example**
  - $\boldsymbol{a} \ = \ \boldsymbol{b} * -\boldsymbol{c} + \boldsymbol{b} * -\boldsymbol{c};$

```
t1 = minus c
t2 = b * t1
t3 = minus c
t4 = b * t3
t5 = t2 + t4
 a = t5
```

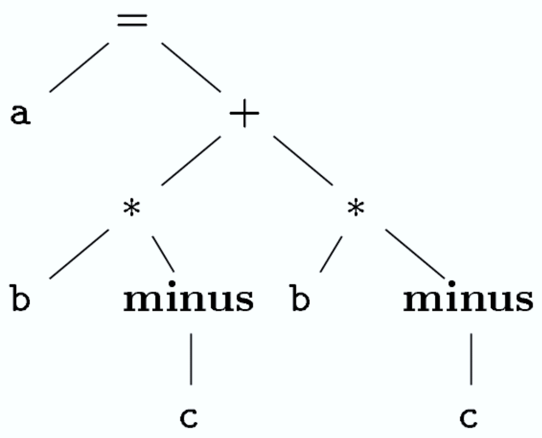
(a) Three-address code

	op	*arg*$_1$	*arg*$_2$	*result*
0	minus	c		t_1
1	*	b	t_1	t_2
2	minus	c		t_3
3	*	b	t_3	t_4
4	+	t_2	t_4	t_5
5	=	t_5		a
		. . .		

(b) Quadruples

Three-Address Code

- **Triples**
  - A triple has only three fields, which we call op, arg1, and arg2
  - We refer to the result of an operation x op y by its position, rather than by an explicit temporary name
- **Example**
  - $\boldsymbol{a} = \boldsymbol{b} * -\boldsymbol{c} + \boldsymbol{b} * -\boldsymbol{c};$



(a) Syntax tree

	op	arg_1	arg_2
0	minus	c	
1	*	b	(0)
2	minus	c	
3	*	b	(2)
4	+	(1)	(3)
5	=	a	(4)
		. . .	

(b) Triples

Three-Address Code

- **Indirect triples**
  - A benefit of quadruples over triples can be seen in an optimizing compiler, where instructions are often moved around
    - With triples, moving an instruction may require us to change all references to that result
  - **Indirect triples** consist of a listing of pointers to triples, rather than a listing of triples themselves
    - With indirect triples, an optimizing compiler can move an instruction by reordering the instruction list

- **Example**
  - $a = b * -c + b * -c;$

	instruction
35	(0)
36	(1)
37	(2)
38	(3)
39	(4)
40	(5)
	...

	op	arg_1	arg_2
0	minus	c	
1	*	b	(0)
2	minus	c	
3	*	b	(2)
4	+	(1)	(3)
5	=	a	(4)
	...		

Three-Address Code

- **Example**
  - $a + -(b + c)$
  - **Quadruple**

	on	arg1	arg2	result
0	+	b	c	t1
1	minus	t1		t2
2	+	a	t2	t3

  - **Triple**

	on	arg1	arg2
0	+	b	c
1	minus	(0)	
2	+	a	(1)

Indirect triples

	on	arg1	arg2
0	+	b	c
1	minus	(0)	
2	+	a	(1)

	instruction
0	(0)
1	(1)
2	(2)

Three-Address Code

- **Example**
  - $a = b[i] + c[j]$
  - **Quadruple**

```
0) =[]   b    i    t1
1) =[]   c    j    t2
2) +     t1   t2   t3
3) =     t3        a
```

  - **Triple**

```
0) =[]   b    i
1) =[]   c    j
2) +     (0)  (1)
3) =     a    (2)
```

Three-Address Code

- **Example**
  - $a[i] = b * c - b * d$
  - **Quadruple**

```
0) *    b    c    t1
1) *    b    d    t2
2) -    t1   t2   t3
3) []=  a    i    t4
4) =    t3        t4
```

  - **Triple**

```
0) *    b    c
1) *    b    d
2) -    (0)  (1)
3) []=  a    i
4) =    (3)  (2)
```

Three-Address Code

- **Example**
  - $x = f(y + 1) + 2$
  - **Quadruple**

```
0) +        y     1     t1
1) param    t1
2) call     f     1     t2
3) +        t2     2    t3
4) =        t3          x
```

  - **Triple**

```
0) +        y      1
1) param    (0)
2) call     f      1
3) +        (2)    2
4) =        x      (3)
```