Compiler Design

Samit Biswas

samit@cs.iiests.ac.in

Department of Computer Science and Technology,
Indian Institute of Engineering Science and Technology, Shibpur

October 4, 2018

Intermediate Code Generation

Benefits of of using a machine-independent intermediate form are:

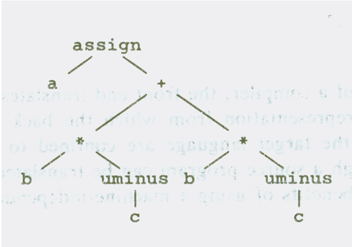
- Retargeting is facilitated;
- A machine independent code optimizer can be applied to the intermediate representation.

Intermediate Representation

- Syntax trees
- DAG
- Three Address Code

Example: Syntax Tree and DAG

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



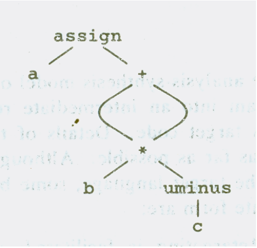
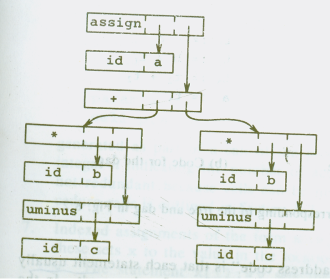
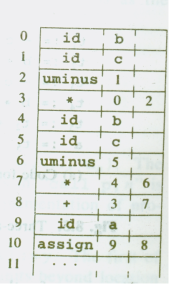


Table: SDD to produce Syntax Trees for assignment statements

Productions	Semantic Rules
$S \rightarrow id = E$	$S._{nptr}$ = mknode('assign', mkleaf(id, id.place), $E._{nptr}$)
$E \rightarrow E_1 + E_2$	$E._{nptr} = mknode('+', E_{1.nptr}, E_{2.nptr})$
$E \rightarrow E_1 * E_2$	$E._{nptr} = mknode('*', E_{1.nptr}, E_{2.nptr})$
$E \rightarrow -E_1$	$E._{nptr} = mkunode('uminus', E_{1.nptr})$
$E \rightarrow (E_1)$	$E._{nptr} = E_{1.nptr}$
$E \rightarrow id$	$E.nptr = mkleaf(id, id.place)$





0	id	b	
1	id	c	
2	uminus	1	
3	*	0	2
4	id	b	
5	id	c	
6	uminus	5	
7	*	4	6
8	+	3	7
9	id	a	
10	assign	9	8
11	. . .		

Three Address Code

Three address code is a sequence of statements of the general form

$$x = y \; op \; z$$

where x, y and z are names, constants or compiler-generated temporaries; op stands for operator.

The Source language expressions like $x + y * z$ might be translated into the following sequences:

$$t_1 = y * z$$
$$t_2 = x + t_1$$

Three address code is a linearised representation of Syntax tree or DAG.
Example: $a = b * -c + b * -c$

Code for the Syntax Tree	Code for the DAG
$t_1 = -c$	
$t_2 = b * t_1$	$t_1 = -c$
$t_3 = -c$	$t_2 = b * t_1$
$t_4 = b * t_3$	$t_5 = t_2 + t_2$
$t_5 = t_2 + t_4$	$a = t_5$
$a = t_5$	

Types of Three Address Statements

- **Assignment statements**

$$x = y \ op \ z$$

Here *op* is a binary arithmetic or logical operation.

Types of Three Address Statements

- **Assignment statements**

$$x = y\ op\ z$$

Here *op* is a binary arithmetic or logical operation.

- **Assignment statements**

$$x = op\ z$$

here op is a unary *operation*

Types of Three Address Statements

- **Assignment statements**

$$x = y \; op \; z$$

Here *op* is a binary arithmetic or logical operation.

- **Assignment statements**

$$x = op \; z$$

here op is a unary *operation*

- **Copy statements**

$$x = y$$

The value of y is assigned to x.

- **Unconditional Jump**

  $$goto\ L$$

  Three address statement with label L is the next to be executed.

- **Unconditional Jump**

$$goto\ L$$

Three address statement with label L is the next to be executed.
- **Conditional Jump**

if x relop y goto L

Example: **if** $a < b$ **then 1 else 0**

- **Unconditional Jump**

  goto L

  Three address statement with label L is the next to be executed.
- **Conditional Jump**

  if x relop y goto L

  Example: **if** $a < b$ **then 1 else 0** .

  .
  100: if $a < b$ goto 103
  101: $t = 0$
  102: goto 104
  103: t =1
  104:
  .
  .

```
while a < b do
    if c < d then
        x = y + z
    else
        x = y - z
```

```
while a < b do
    if c < d then
        x = y + z
    else
        x = y - z
```

Three address code :

```
L1:   if a< b goto L2
        goto Lnext
L2:   if c < d goto L3
        goto L4
L3:   t1 = y + z
      x = t1
      goto L1
L4:   t2 = y - z
      x =t2
      goto L1
Lnext:
```

- **Statement for procedure calls**
  - Param x, set a parameter for a procedure call
  - Call p, n call procedure p with n parameters
  - Return y return from a procedure with return value y (optional)

Example: procedure call: $p(x1, x2, x3, \ldots, xn)$
param x1
param x2
param x3
$\ldots$
param xn
call p, n

- **Statement for procedure calls**
  - Param x, set a parameter for a procedure call
  - Call p, n call procedure p with n parameters
  - Return y return from a procedure with return value y (optional)

  Example: procedure call: $p(x1, x2, x3, \ldots, xn)$

  param x1

  param x2

  param x3

  $\ldots$

  param xn

  call p, n
- **Indexed Assignments**
  - x = y[i] and x[i] = y

- **Statement for procedure calls**
  - Param x, set a parameter for a procedure call
  - Call p, n call procedure p with n parameters
  - Return y return from a procedure with return value y (optional)

  Example: procedure call: $p(x1, x2, x3, \ldots, xn)$

  param x1

  param x2

  param x3

  $\ldots$

  param xn

  call p, n
- **Indexed Assignments**
  - x = y[i] and x[i] = y
- **Address and Pointer Assignments**
  - x =&y, x=*y

Syntax Directed Translation into Three Address Code

Production	Semantic Rules
$S \rightarrow id = E$	$S_{.code} = E_{.code} \parallel gen(id_{.place,'=',E_{.place}})$
$E \rightarrow E_1 + E_2$	$E_{.place} = newtemp$ $E_{.code} = E_{1.code} \| \| E_{2.code} \| \|$ $gen(E_{.place},' =', E_{1.place},' +', E_{2.place})$
$E \rightarrow E_1 * E_2$	$E_{.place} = newtemp$ $E_{.code} = E_{1.code} \| \| E_{2.code} \| \|$ $gen(E_{.place},' =', E_{1.place},' *', E_{2.place})$
$E \rightarrow -E_1$	$E_{.place} = newtemp$ $E_{.code} = E_{1.code} \| \| gen(E_{.place},' =',' uminus', E_{1.place})$
$E \rightarrow (E_1)$	$E_{.place} = E_{1.place}$ $E_{.code} = E_{1.code}$
$E \rightarrow id$	$E_{.place} = id_{.place}$ $E_{.code} = ' '$

Three address Code : Assignment Statement

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

Three Address Code:

$t_1 = -c$

$t_2 = b * t_1$

$t_3 = -c$

$t_4 = b * t_3$

$t_5 = t_2 + t_4$

$a = t_5$

Implementations of Three-Address Statements

A Three address code is an abstract form of Intermediate code. This can be implemented in the form of records with fields for the **operator and the operands.** Three such representations are as follows:

- Quadruples
- Triples
- indirect Triples

Quadruples

It is a record structure with four fields (op, $arg1$, $arg2$, $result$)

- x = y op z
  representing by placing **y in arg1**, **z in arg2** and
  x in result.
- $x = -y$ or $x = y$
  we do not use arg2.
- The fields *arg1* or *arg2* and *result* are pointers to the symbol table.

Quadruples for the assignment
$a = b * -c + b * -c$

	op	arg1	arg2	result
(0)	uminus	c		t1
(1)	*	b	t1	t2
(2)	uminus	c		t3
(3)	*	b	t3	t4
(4)	+	t2	t4	t5
(5)	=	t5		a

Triples

It is a record structure with three fields (*op*, *arg*1, *arg*2)

- The fields arg1 or arg2 are either pointers to the symbol table entry or pointer into Triple structure.

	op	arg1	arg2
(0)	uminus	c	
(1)	*	b	(0)
(2)	uminus	c	
(3)	*	b	(2)
(4)	+	(1)	(3)
(5)	=	a	(4)

Indirect Triples

Listing of Pointers to Triples is maintained by a separate structure.

	Statement
(0)	(14)
(1)	(15)
(2)	(16)
(3)	(17)
(4)	(18)
(5)	(19)

	op	arg1	arg2
(14)	uminus	c	
(15)	*	b	(14)
(16)	uminus	c	
(17)	*	b	(16)
(18)	+	(15)	(17)
(19)	=	a	(18)

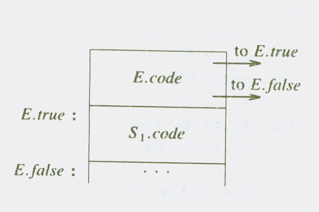
Semantic rules generating **three address code** for a **flow of control statements** statement:

$$
\begin{aligned}
S &\rightarrow \textit{if } E \textit{ then } S_1 \\
&\mid \textit{if } E \textit{ then } S_1 \textit{ else } S_2 \\
&\mid \textit{while } E \textit{ do } S_1
\end{aligned}
$$

we assume that a three address statement can be symbolically labelled and the function *newlabel* returns a new symbolic label each time called. We associate two labels:

- E.true : The label to which control flows if E is true.
- E.false: The label to which control flows if E is false.

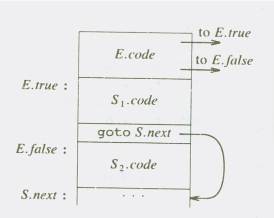
SDD for Flow-of-Control : *if – then*



Production	Semantic Rules
$S \rightarrow if\ E\ then\ S_1$	$E_{.true} = newlabel;$ $E_{.false} = S_{.next};$ $S_{1.next} = S.next;$ $S_{.code} = E_{.code} \parallel gen(E.true,':') \parallel S_{1.code}$

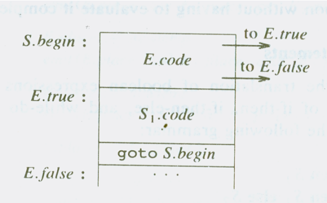
SDD for Flow-of-Control : *if – then – else*

Production	Semantic Rules
$S \rightarrow if\ E\ then\ S_1\ else\ S_2$	$E_{.true}$= *newlabel*; $E_{.false}$= *newlabel*; $S_{1.next} = S_{.next}$; $S_{2.next} = S_{.next}$ $S_{.code} = E_{.code} \|\|$ $gen(E_{.true},' :') \|\| S_{1.code} \|\|$ gen('goto', $S.next$) $\|\|$ gen($E_{.false}$, ':') $\|\| S_{2.code}$



SDD for Flow-of-Control : *while – do*

Production	Semantic Rules
$S \rightarrow while\ E\ do\ S_1$	$S_{.begin} = newlabel;$ $E_{.true} = newlabel$ $E_{.false} = S_{.next};$ $S_{1.next} = S_{.begin};$ $S_{.code} = gen(S_{.begin}, ':') \parallel E_{.code} \parallel$ $gen(E_{.true}, ':') \parallel S_{1.code} \parallel$ gen('goto', $S.begin$)



Semantic rules generating TAC for a **while** statement:

```
while a < b do
    if c < d then
        x = y + z
    else
        x = y - z
```

Three address code :

```
L1:    if a< b goto L2
         goto Lnext
L2:    if c < d goto L3
         goto L4
L3:    t1 = y + z
       x = t1
       goto L1
L4:    t2 = y - z
       x =t2
       goto L1
Lnext:
```

SDD for : Boolean expression
Let us Consider the following Expression:

$$a < b \text{ or } c < d \text{ and } e < f$$

Suppose that **true** and **false** exists for the entire expression have been set to *Ltrue* and *Lfalse*

```
    if a < b goto Ltrue
    goto L1
L1: if c < d goto L2
    goto Lfalse
L2: if e < f goto Ltrue
    goto Lfalse
```

SDD for : Boolean expression

Production	Semantic Rules
$E \rightarrow E_1$ *or* E_2	$E_{1.true} = E_{.true}$; $E_{1.false} = newlabel$; $E_{2.true} = E_{.true}$; $E_{2.false} = E_{.false}$; $E_{.code} = E_{1.code} \| gen(E_{1.false}, ':') \| E_{2.code}$
$E \rightarrow E_1$ *and* E_2	$E_{1.true} = newlabel$; $E_{1.false} = E_{.false}$; $E_{2.true} = E_{.true}$; $E_{2.false} = E_{.false}$; $E_{.code} = E_{1.code} \| gen(E_{1.true}, ':') \| E_{2.code}$
$E \rightarrow$ *not* E_1	$E_{1.true} = E_{.false}$; $E_{1.false} = E.true$ $E_{.code} = E_{1.code}$

SDD for : Boolean expression

Table: default

Production	Semantic Rules
$E \rightarrow (E_1)$	$E_{1.true} = E_{.true}$; $E_{1.false} = E_{false}$; $E.code = E_{1.code}$;
$E \rightarrow id_1 \ relop \ id_2$	$E.code = gen('if', id_{1.place}, relop_{op}, id_{2.place}$ $'goto', E_{true}) \| gen('goto', E_{false})$
$E \rightarrow true$	$E_{code} = gen('goto', E_{true})$
$E \rightarrow false$	$E_{code} = gen('goto', E_{false})$