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:

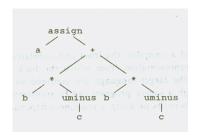
- Retargeting is facilitated;
- ► A machine independent code optimizer can be applied to the intermediate representation.

Intermediate Representation

- Syntax trees
- DAG
- ▶ Three Address Code

Example: Syntax Tree and DAG

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



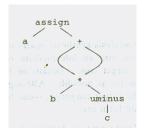
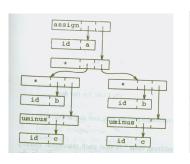


Table: SDD to produce Syntax Trees for assignment statements

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 o id	E.nptr = mkleaf(id, id.place)



0	id	b	1
i	id	C	1
2	uminus	1	
3	=: * 3	0	2
4	id	b	
5	:id	C	
6	uminus	5	The
7	**	4	6
8	+ 100	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
$t_1 = -c$	
$t_2 = b * t_1$	$ \begin{vmatrix} t_1 = -c \\ t_2 = b * t_1 \end{vmatrix} $
$t_3 = -c$	-
$t_4 = b * t_3$	$t_5=t_2+t_2$
$t_5=t_2+t_4$	$a=t_5$
$a=t_5$	

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
```

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, ..., xn)
param x1
param x2
param x3
...
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, ..., xn) param x1 param x2 param x3 ... 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, ..., xn) param x1 param x2 param x3 ... 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
$S \rightarrow id = E$	$\mathcal{S}_{.code} = \mathcal{E}_{.code} \parallel \mathit{gen}(\mathit{id}_{.\mathit{place},'=',\mathcal{E}_{.\mathit{place}}})$
$E \rightarrow E_1 + E_2$	$E_{.place} = newtemp$
	$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} $
	$gen(E_{.place}, '=', E_{1.place}, '*', E_{2.place})$
$E ightarrow -E_1$	$E_{.place} = newtemp$
	$E_{.code} = E_{1.code} gen(E_{.place}, '=', 'uminus', E_{1.place}) $
$E ightarrow (E_1)$	$E_{.place} = E_{1.place}$
	$E_{.code} = E_{1.code}$
E o id	$E_{.place} = id_{.place}$
	$E_{.code} = $

II represents concat E.code = E1.code II E2.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

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

	ор	arg1	arg2	result
(0)	uminus	С		t1
(1)	*	b	t1	t2
(2)	uminus	С		t3
(3)	*	b	t3	t4
(4)	+	t2	t4	t5
(5)	=	t5		а

Triples

It is a record structure with three fields (op, arg1, arg2)

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

	ор	arg1	arg2
(0)	uminus	С	
(1)	*	b	(0)
(2)	uminus	С	
(3)	*	b	(2)
(4)	+	(1)	(3)
(5)	=	а	(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)

	ор	arg1	arg2
(14)	uminus	С	
(15)	*	b	(14)
(16)	uminus	С	
(17)	*	b	(16)
(18)	+	(15)	(17)
(19)	=	а	(18)

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

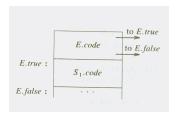
$$S \rightarrow \textit{if E then } S_1$$

| $\textit{if E then } S_1 \textit{ else } S_2$
| $\textit{while E do } S_1$

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.

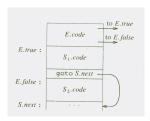
SDD for Flow-of-Control: *if* – *then*



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

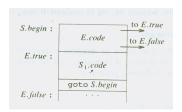
SDD for Flow-of-Control : if - then - else

Production	Semantic Rules
$S \rightarrow if {\color{red} E} then S_1 else S_2$	E _{.true} = newlabel;
	E _{.false} = newlabel;
	$S_{1.next} = S_{.next};$
	$\mathcal{S}_{2.\mathit{next}} = \mathcal{S}_{.\mathit{next}}$
	$S_{.code} = {\color{red} {m{\mathcal{E}}_{.code}}} \ $
	$gen(E_{.true},':') \parallel S_{1.code} \parallel$
	gen('goto', <i>S.next</i>) ∥
	$gen(E_{.false}, ':') \parallel S_{2.code}$



SDD for Flow-of-Control: while – do

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

$$a < b$$
 or $c < d$ and $e < f$

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

 $\begin{array}{lll} \text{if} & a < b \text{ goto Ltrue} \\ \text{goto L1} & & \\ \end{array}$

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	$E_{1.true} = E_{.true};$
	$E_{1.false} = newlabel;$
	$E_{2.true} = E_{.true};$
	$E_{2.false} = E_{.false};$
	$E_{.code} = E_{1.code} \parallel gen(E_{1.false}, ':') \parallel E_{2.code}$
$E \rightarrow E_1$ and E_2	$E_{1.true} = newlabel;$
	$E_{1.false} = E_{.false};$
	$E_{2.true} = E_{.true};$
	$E_{2.false} = E_{.false};$
	$E_{.code} = E_{1.code} \parallel gen(E_{1.true}, ':') \parallel E_{2.code}$
$E \rightarrow not E_1$	$E_{1.true} = E_{.false};$
	$E_{1.false} = E.true$
	$E_{.code} = E_{1.code}$

SDD for: Boolean expression

Table: default

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