Intermediate Code Generation

Sudakshina Dutta

IIT Goa

12th April, 2022

Intermediate code

Quadruple

- It has four fields, e.g., *op*, arg_1, arg_2 and *result*
- Three-address instruction $x = y + z$ is represented by placing $+$ in *op*, y in arg_1, z in arg_2, and x in *result*
- Some restriction
  1. Instruction with unary operator $x = minus\ y$ or $x = y$ do not use arg_2
  2. Operators like *param* use neither arg_2 nor *result*
  3. Conditional and unconditional jumps put the target label in *result*

Quadruple — example

$t_1 = minus\ c$
$t_2 = b * t_1$
$t_3 = minus\ c$
$t_4 = b * t_3$
$t_5 = t_2 + t_4$
$a = t_5$

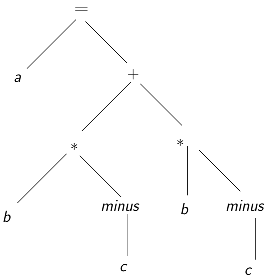
	op	arg_1	arg_2	*result*
0	*minus*	c		t_1
1	$*$	b	t_1	t_2
2	*minus*	c		t_3
3	$*$	b	t_3	t_4
4	$+$	t_2	t_4	t_5
5	$=$	t_5		a
		$\dots$		

Triple

- Three fields — *op*, arg_1, arg_2
- The field *result* in quadruple is only used to hold temporaries
- Instead of a temporary t_1, triple uses its position e.g., (*pos*)
- DAG and triple are equivalent; however, there are differences in expressing control-flow

Triple — example

Expression — $a = b * -c + b * -c$



	op	*arg*$_1$	*arg*$_2$
0	*minus*	*c*	
1	$*$	*b*	(0)
2	*minus*	*c*	
3	$*$	*b*	(2)
4	$+$	(1)	(3)
5	$=$	*a*	(4)
		. . .	

Static Single-Assignment

- All assignments in SSA are to variables with distinct names; hence, the term static single-assignment
- It facilitates certain code optimizations

$$
\begin{array}{ll}
p = a + b & p_1 = a + b \\
q = p - c & q_1 = p_1 - c \\
p = q * d & p_2 = q_1 * d \\
p = e - p & p_3 = e - p_2 \\
q = p + q & q_2 = p_3 + q_1
\end{array}
$$

Static Single-Assignment

- The same variable may be defined in two different control-flow paths in a program
- SSA uses a notational convention called the $\phi-$function to combine the two definitions of x

$if\ (flag)\ x = -1;\ else\ x = 1;$

$y = x * a;$

$if\ (flag) = 1\ x_1 = -1;\ else\ x_2 = 1;$

$x_3 = \phi(x_1, x_2);$

$y = x_3 * a;$

Translation of expression

Grammar

$$S \rightarrow id = E$$

$$E \rightarrow E + E | - E | (E) | id$$

- The attribute **code** is synthesized attribute
- Attributes $S.code$ and $E.code$ denote three-address code for S and E, respectively
- Attribute $E.addr$ denotes the address that will hold the value of E

Translation of expression

Three-address code for expression

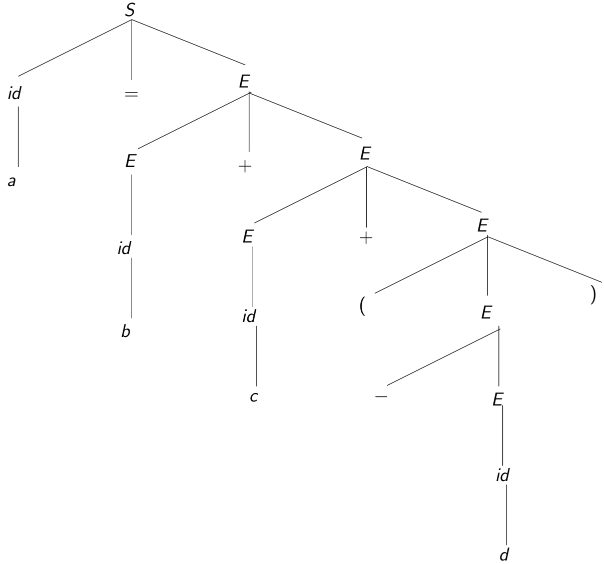
gen -> Means generating the 3 address code corresponding to argument passed.

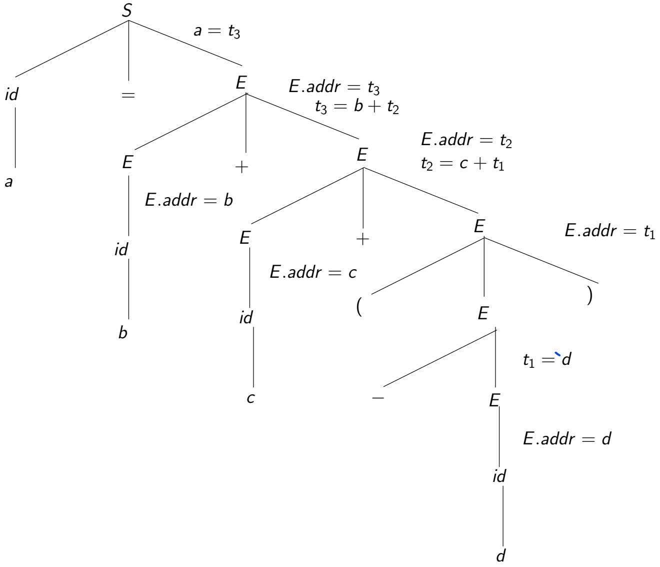
top -> Symbol Table. We are passing the name & getting the address.

PRODUCTION RULE	SEMANTIC RULES
$S \rightarrow id = E$	$\{gen(top.get(id.lexeme))\ '=' \ E.addr)\}$
$E \rightarrow E_1 + E_2$	$\{E.addr = new \ \ Temp();$
	$gen(E.addr' =' E_1.addr' +' E_2.addr); \}$
$\mid - E_1$	$\{E.addr = new \ \ Temp();$
	$gen(E.addr' =' 'minus' E_1.addr); \}$
$\mid (E_1)$	$\{E.addr = E_1.addr\}$
$\mid id$	$\{E.addr = top.get(id.lexeme)\}$

Consider the following code segment

$a = b + c + (-d)$





$t_1 = -d$
$t_2 = c + t_1$
$t_3 = b + t_2$
$a = t_3$

Intermediate code - Example 1

$int\ a[10],\ b[10],\ dot\_prod,\ i;$

$dot\_prod = 0;$

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

$$\begin{aligned}
& dot\_prod = 0; \\
& i = 0; \\
L_1 :\ & if(i >= 10)\ goto\ L2 \\
& T1 = addr(a) \\
& T2 = i * 4 \\
& T3 = T1[T2] \\
& T4 = addr(b) \\
& T5 = i * 4
\end{aligned}$$

$$\begin{aligned}
& T6 = T4[T5] \\
& T7 = T3 * T6 \\
& T8 = dot\_prod + T7 \\
& dot\_prod = T8 \\
& T9 = i + 1 \\
& i = T9 \\
& goto\ L1 \\
L2 :\ &
\end{aligned}$$

When converting manually to Intermediate Code,
if we want to increment i by 1,
tmp = i + 1
i = tmp