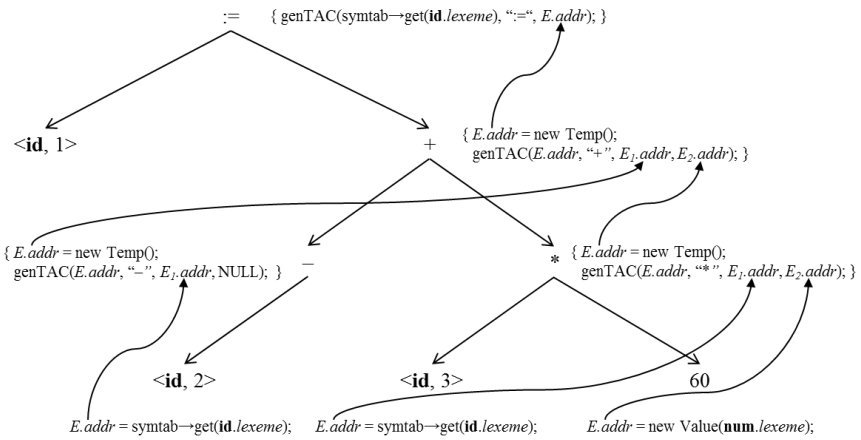
Intermediate-Code Generation

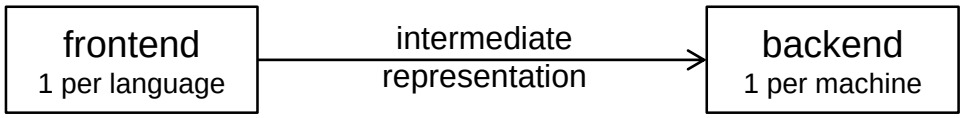


Intermediate Representations

CSE 컴퓨터공학부
Department of Computer Science & Engineering

Intermediate Representation (IR)

- Separation of the frontend (scanner, parser) from the backend (machine-dependent code generator)
  - IR: “glue” between the frontend and backend



  - allows to perform language-independent optimizations on the IR level
  - if the IR is designed well, any frontend producing the IR can be combined with any backend generating machine code from the IR
    - obtain NxM compilers (N languages, M machines) with just N frontends and M backends

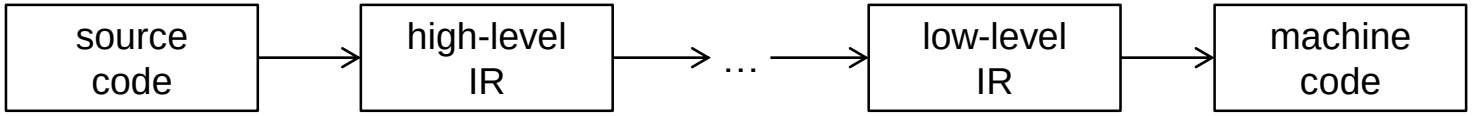
“In theory, theory and practice are the same. In practice, they are not.”

Albert Einstein

CSE 컴퓨터공학부
Department of Computer Science & Engineering

Intermediate Representation

- Often, there exists more than one IR in a compiler suite
  - from high-level to low-level

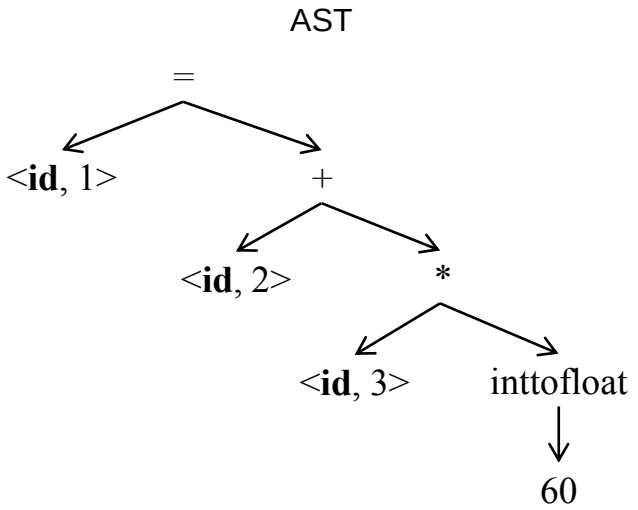


  - high-level: closer to source
    - structures preserved (variable names, control-flow structures)
    - easier to do high-level optimizations
  - low-level: closer to machine code
    - often in a ‘assembly-like’ form

CSE 컴퓨터공학부 Department of Computer Science & Engineering

Intermediate Representation

- Forms of intermediate representations
  - any form that is convenient to the compiler
    - not a stream of characters any more
    - not (yet) machine operations in binary form
  - typical forms:
    - trees
      - parse tree, abstract syntax tree
    - linear representations
      - three-address code




```
t1 = inttofloat(60)
t2 = id3 * t1
t3 = id2 + t2
id1 = t3
```

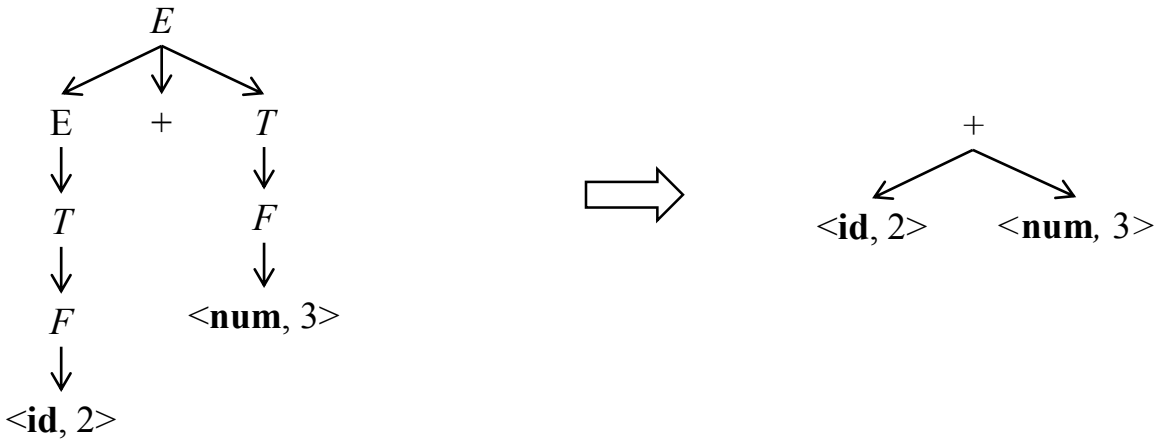
three-address code

- Higher-level IRs typically use some form of an AST (abstract syntax tree), whereas lower-level IRs are usually represented in linear form

CSE 컴퓨터공학부 Department of Computer Science & Engineering

IR: Abstract Syntax Trees (AST)

- Abstract Syntax Trees (AST)
  - each interior node represents an operator, the children of the node represent the operands for the operator

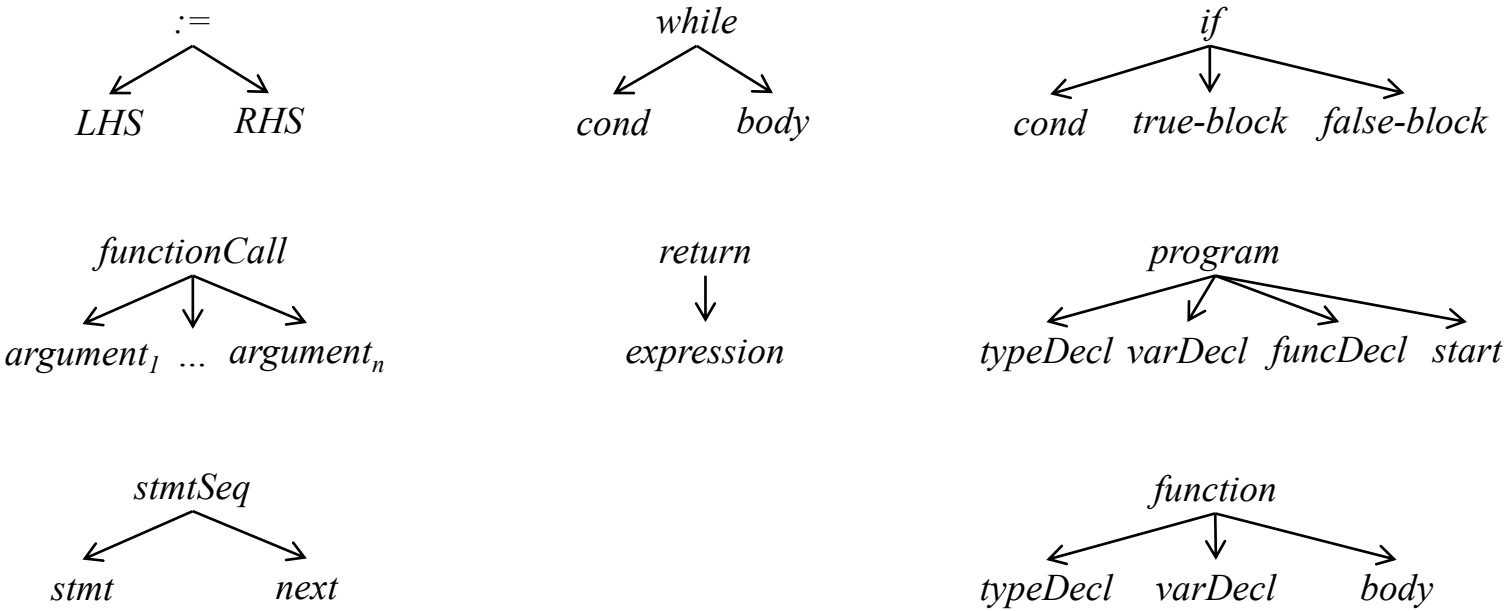


parse tree AST

CSE 컴퓨터공학부 Department of Computer Science & Engineering

IR: Abstract Syntax Trees (AST)

- Abstract Syntax Trees (AST)
  - any programming construct can be handled by making up an “operator” for the construct and adding “operands” to represent the semantically meaningful components of that construct

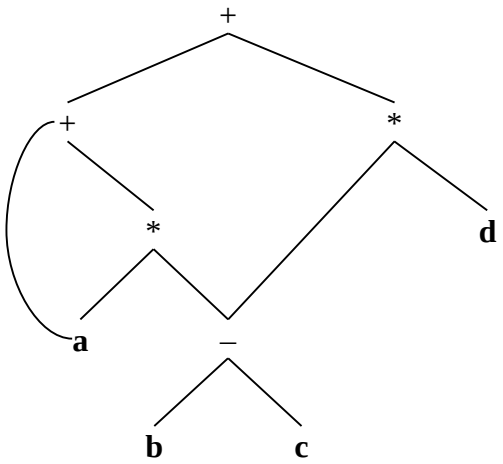


CSE 컴퓨터공학부 Department of Computer Science & Engineering

High-Level IR: Directed Acyclic Graphs

- Directed Acyclic Graph (DAG)
  - very close to syntax trees
  - node N has more than one parent if N represents a common subexpression

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



CSE 컴퓨터공학부
Department of Computer Science & Engineering

Identifying Common Subexpressions

- How do we identify common subexpressions?

- Idea: identify identical nodes and return a pointer to the existing node instead of creating a new one

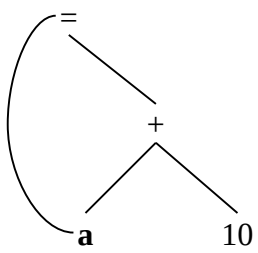
Production			***Semantic Rule***
E	→	$E_1 + T$	$E.node = GetNode(\text{`+'}, E_1.node, T.node)$
E	→	$E_1 - T$	$E.node = GetNode(\text{`–'}, E_1.node, T.node)$
E	→	T	$E.node = T.node$
T	→	**(** E **)**	$T.node = E.node$
T	→	**id**	$T.node = GetLeaf($**id**, **id**$.entry)$
T	→	**num**	$T.node = GetLeaf($**num**, **num**$.val)$

CSE 컴퓨터공학부 Department of Computer Science & Engineering

The Value-Number Method

- Traditionally, the nodes of a DAG were stored in an array.
- The index for a node into that array is called its *value number*.

a = a + 1



1	**id**		→ to entry of a in symbol table
2	**num**	1	
3	+	1	2
4	=	1	3
5	…		

CSE 컴퓨터공학부 Department of Computer Science & Engineering

The Value-Number Method

- Algorithm for constructing a DAG with common subexpressions

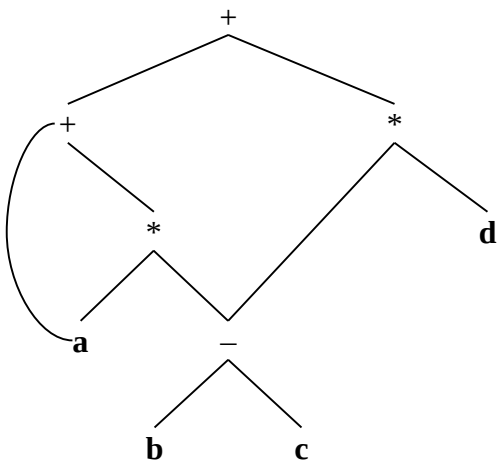
```
GetNode(Label op, Node l, Node r)
{
  if there is an entry in the DAG array that matches {op,l,r}
  then {
    return an index to that entry;
  } else {
    insert a new entry {op,l,r} into the DAG array;
    return the index of the new entry;
  }
}

GetLeaf(Type t, Value v)
{
  if there is an entry in the DAG array that matches {t, v}
  then {
    return an index to that entry;
  } else {
    insert a new entry {t, v} into the DAG array;
    return the index of the new entry;
  }
```


CSE 컴퓨터공학부 Department of Computer Science & Engineering

IR: Three-Address Code

- Three-address Code
  - list of simple instructions (think: “assembly instructions”)
  - at most one operator on the right-hand side of an instruction
  - linearized representation of a syntax tree or a DAG




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

DAG

three-address code

CSE 컴퓨터공학부 Department of Computer Science & Engineering

Three-Address Code

- Building blocks of three-address code
  - instructions
  - addresses
    - names (variables)
    - constants
    - compiler-generated temporary values
  - labels to represent control-flow
    - symbolic or index

```
   t1 = b - c
L1:
   t2 = a * t1
   t3 = a + t2
   t4 = t1 * d
   t5 = t3 + t4
   if t5 goto L1
```


CSE 컴퓨터공학부 Department of Computer Science & Engineering

Three-Address Code

- Common forms of three-address code instructions
  - assignments
    - binary `x = y op z`
    - unary `x = op y`
    - copy `x = y`
  - jumps
    - unconditional `goto L`
    - conditional `if x goto L / ifFalse x goto L`

      `if x relop y goto L`
  - procedure calls
    - parameters `param x`
    - procedure `call p, n`
    - function `y = call p, n`

CSE 컴퓨터공학부 Department of Computer Science & Engineering

Three-Address Code

- Common forms of three-address code instructions (cont'd)
  - indexed copies
    - read indexed `x = y[i]`
    - write indexed `x[i] = y`
  - address manipulation
    - location `x = &y`
    - pointer assignment `*x = y`

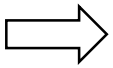
      `x = *y`

CSE 컴퓨터공학부 Department of Computer Science & Engineering

Three-Address Code

- Example

```
do {
  i = i+1;
} while (a[i] < v);
```



```
L:
    t1 = i + 1
    i = t1
    t2 = i*4
    t3 = a[t2]
    if t3 < v goto L
```


CSE 컴퓨터공학부 Department of Computer Science & Engineering

Storing Tree-Address Code

- Quads (aka quadruples)
  - each statement is represented by a record with up to 4 elements
    - exceptions: unary operators (no arg2), params (neither arg2 nor result), jumps with labels (target in result)

a = b*-c + b*-c

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

	op	arg1	arg2	result
1	minus	c		t1
2	*	b	t1	t2
3	minus	c		t3
4	*	b	t3	t4
5	+	t2	t4	t5
6	=	t5		a

CSE 컴퓨터공학부 Department of Computer Science & Engineering

Storing Tree-Address Code

- Triples and indirect triples
  - similar to quads, except that there is no explicit result
  - triples: refer to the result of an operation by its index
    - difficult to move instructions around

a = b*-c + b*-c

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

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

CSE 컴퓨터공학부 Department of Computer Science & Engineering

Storing Tree-Address Code

- Triples and indirect triples
  - indirect triples: triples still stored in array, but extra instruction list
    - can move around instructions in instruction list

a = b*-c + b*-c

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

instructions

103	(0)
104	(1)
105	(2)
106	(3)
107	(4)
108	(5)
109	(6)

triples

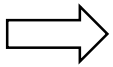
	op	arg1	arg2
1	minus	c	
2	*	b	(1)
3	minus	c	
4	*	b	(3)
5	+	(2)	(4)
6	=	a	(5)

CSE 컴퓨터공학부 Department of Computer Science & Engineering

Static Single-Assignment (SSA) Form

- IR that facilitates lots of code optimizations
  - used in most production compilers
- SSA is a special form of three-address code with two exceptions
  - all assignments are to variables with distinct names (versions)

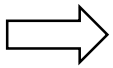
```
p = a + b
q = p - c
p = q * d
p = e - p
q = p + q
```



$p_1 = a + b$
$q_1 = p_1 - c$
$p_2 = q_1 * d$
$p_3 = e - p_2$
$q_2 = p_3 + q_1$

  - a special function "φ" to combine different versions of the same variable

if (flag) x = -1; else x = 1;
y = x * a;



if (flag) x_1 = -1; else x_2 = 1;
$x_3 = \varphi(x_1, x_2)$;
y = x_3 * a;

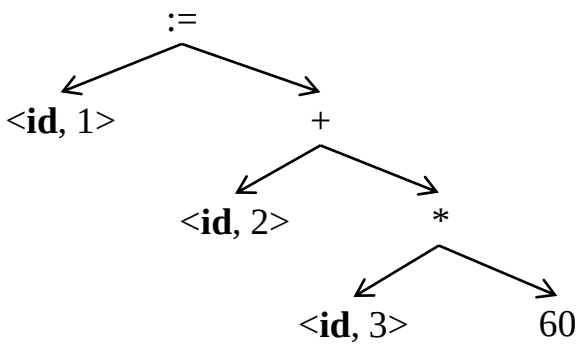
CSE 컴퓨터공학부 Department of Computer Science & Engineering

Intermediate-Code Generation

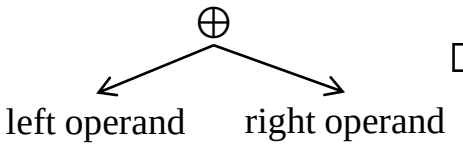
CSE 컴퓨터공학부 Department of Computer Science & Engineering

Translation of Expressions

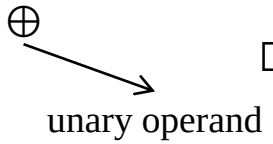
- Translation patterns



  - binary/unary operations



op	arg1	arg2	result
⊕	left operand	right operand	new Temp()

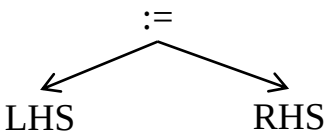


op	arg1	arg2	result
⊕	unary operand	NULL	new Temp()

CSE 컴퓨터공학부 Department of Computer Science & Engineering

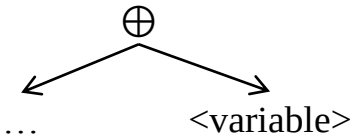
Translation of Expressions

- Translation patterns
  - assignments



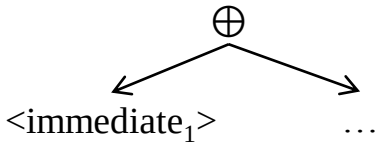
op	arg1	arg2	result
:=	LHS	RHS	NULL

  - operands
    - scalar variables



op	arg1	arg2	result
⊕	…	<variable>	new Temp()

    - immediate values



op	arg1	arg2	result
⊕	<immediate$_1$>	…	new Temp()

CSE 컴퓨터공학부 Department of Computer Science & Engineering

Translation of Expressions

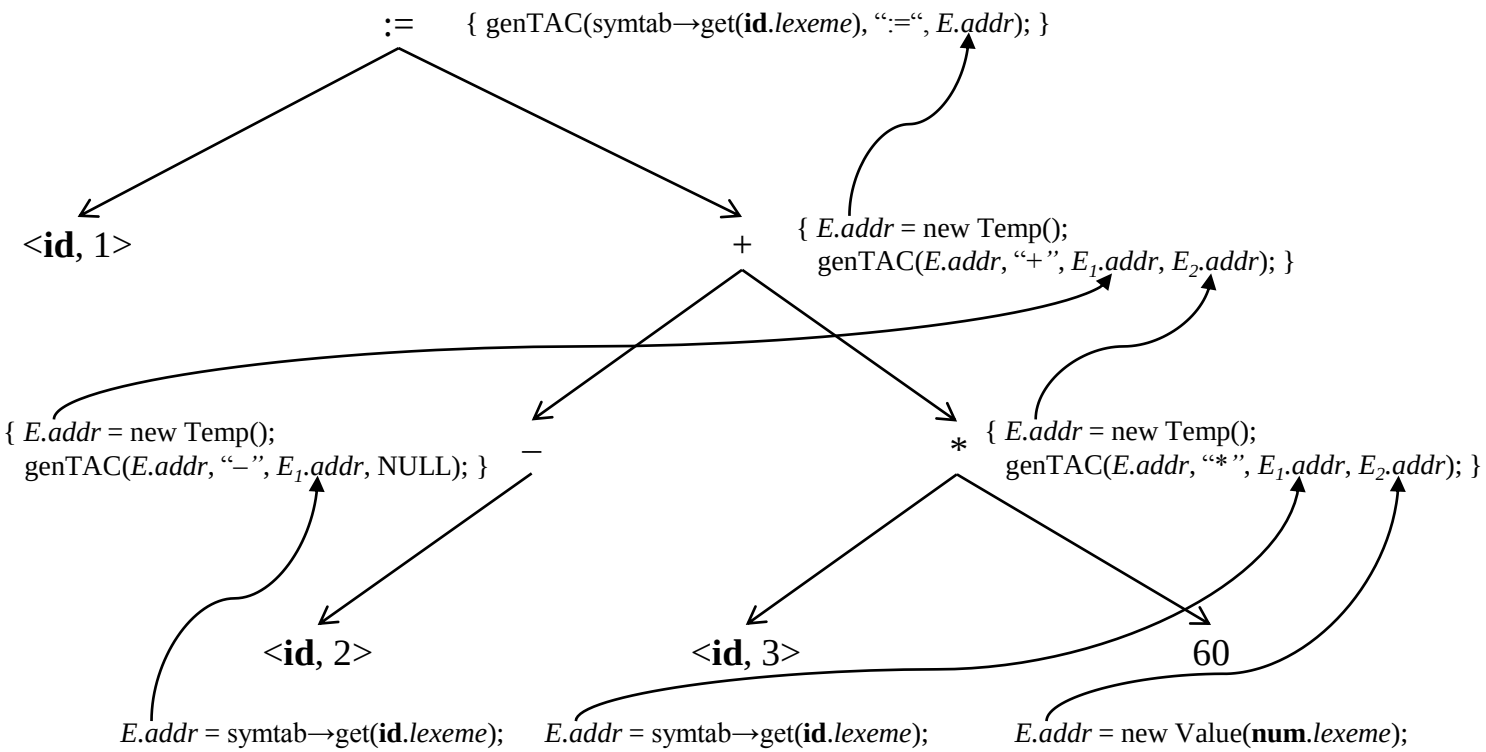
- Syntax-Directed Translation

Production			***Semantic Rule***
S	→	**id** := E ;	{ genTAC(symtab→get(**id**.*lexeme*), ":=", E.*addr*); }
E	→	$E_1 \oplus E_2$	{ E.*addr* = new Temp(); genTAC(E.*addr*, "$\oplus$", E_1.*addr*, E_2.*addr*); }
	|	$\oplus E_1$	{ E.*addr* = new Temp(); genTAC(E.*addr*, "$\oplus$", E_1.*addr*, NULL); }
	|	**(** E_1 **)**	{ E.*addr* = E_1.*addr*; }
	|	**id**	{ E.*addr* = symtab→get(**id**.*lexeme*); }
	|	**num**	{ E.*addr* = new Value(**num**.*lexeme*); }

CSE 컴퓨터공학부 Department of Computer Science & Engineering

Translation of Expressions

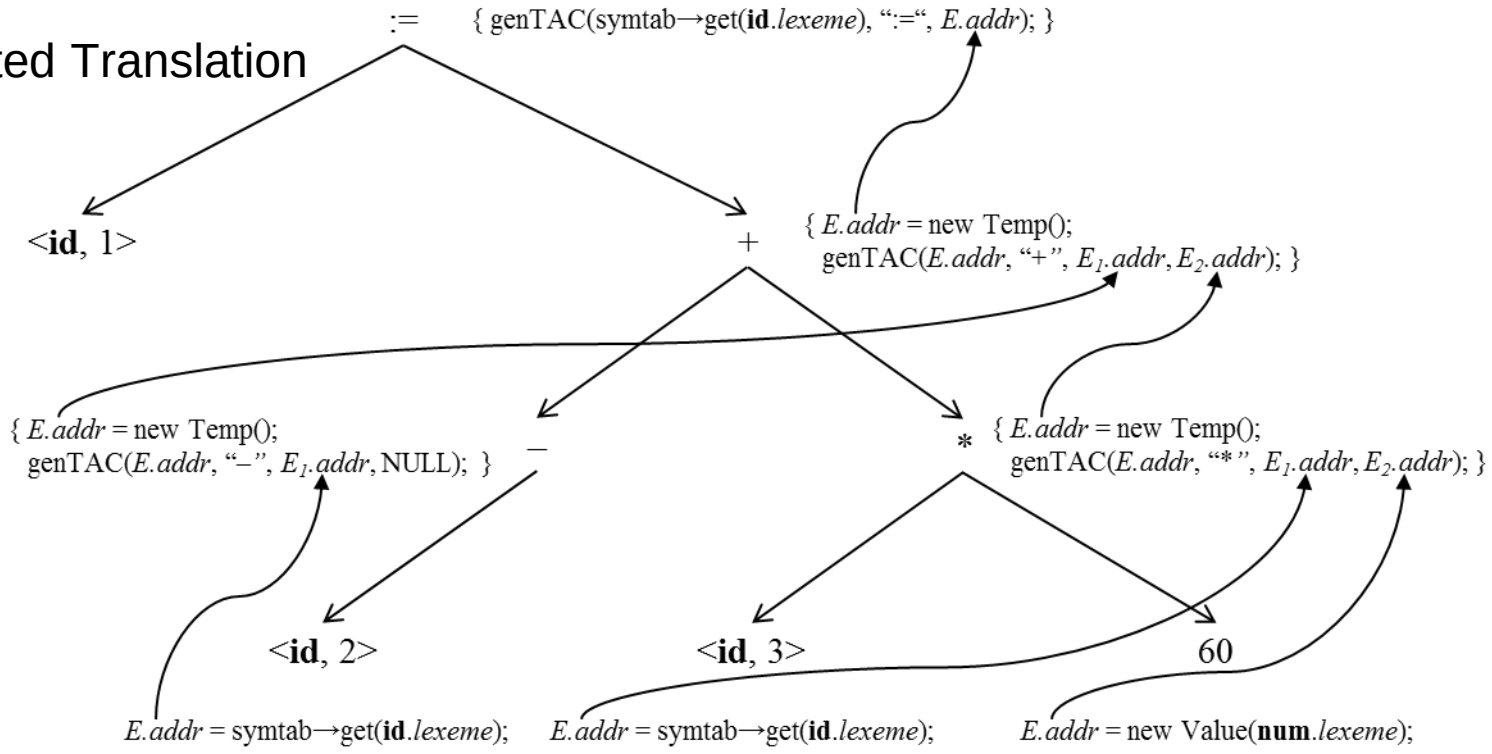
- Syntax-Directed Translation



CSE 컴퓨터공학부 Department of Computer Science & Engineering

Translation of Expressions

- Syntax-Directed Translation



	op	arg1	arg2	result
1	minus	<id, 2>		t1
2	*	<id, 3>	60	t2
3	+	t1	t2	t3
4	:=	t3	<id, 1>	

CSE 컴퓨터공학부
Department of Computer Science & Engineering

Translation of Array Elements

- Address calculation of array elements
  - assumption: row-major layout

```
int A[N][M][O];

A[i][k][l];
```

address computation

```
A + ((i*M + k)*O + l) * sizeof(int)
```


CSE 컴퓨터공학부 Department of Computer Science & Engineering

Translation of Array Elements

- A syntax-directed translation scheme to translate array references

Production			*Semantic Rule*
S	→	**id** := E ;	{ genTAC(symtab→get(**id**.*lexeme*), ":=", *E.addr*); }
	|	L := E ;	{ t = new Temp(); genTAC(t, "+", *L.base.addr*, *L.addr*); genTAC(t , ":=", *E.addr*); }
E	→	$E_1 \oplus E_2$	{ *E.addr* = new Temp(); genTAC(*E.addr*, "$\oplus$", E_1.*addr*, E_2.*addr*); }
	|	**id**	{ *E.addr* = symtab→get(**id**.*lexeme*); }
	|	L	{ *E.addr* = new Temp(); genTAC(*E.addr*, "+", *L.base.addr*, *L.addr*); }
L	→	**id** [E]	{ *L.array* = symtab→get(**id**.*lexeme*); *L.type* = *L.array.type.elem*; *L.addr* = new Temp(); genTAC(*L.addr*, "*", *E.addr*, *L.type.width*); }
	|	L_1 [E]	{ *L.array* = L_1.*array*; *L.type* = L_1.*type.elem*; t = new Temp(); *L.addr* = new Temp(); genTAC(t, "*", *E.addr*, *L.type.width*); genTAC(*L.addr*, "+", L_1.*addr*, t); }

CSE 컴퓨터공학부 Department of Computer Science & Engineering

Translation of Array Elements

- A syntax-directed translation scheme to translate array references

```
int A[2][3];

A[i][j] := c + A[i][j]
```

```
t1 = i * 12;
t2 = j * 4;
t3 = t1 + t2;
t4 = A + t3;
t5 = c + t4;
t6 = i * 12;
t7 = j * 4;
t8 = t6 + t7;
t9 = A + t8;
t9 = t5;
```


CSE 컴퓨터공학부 Department of Computer Science & Engineering

Translation of Boolean Expressions

- Boolean expressions assume two different roles
  - alter the control flow

```
if (<boolean expression>) then begin
  …
end else begin
  …
end
```

  - compute logical values

```
var b: boolean;
…
b := <boolean expression>
```

  - role determined by the syntactic context

CSE 컴퓨터공학부 Department of Computer Science & Engineering

Translation of Boolean Expressions

- Short-circuit code
  - boolean operators &&, ||, and ! are translated into jumps

```
if (x < 100 || x > 200 && x != y) x = 0;
```

can be translated into

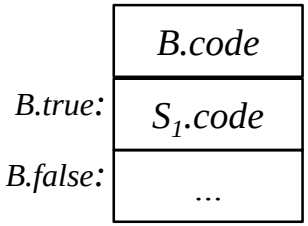
```
        if x < 100 goto L2
        ifNot x > 200 goto L1
        ifNot x != y goto L1
L2:     x = 0
L1:
```

→ value of boolean expression is represented by a position in the code

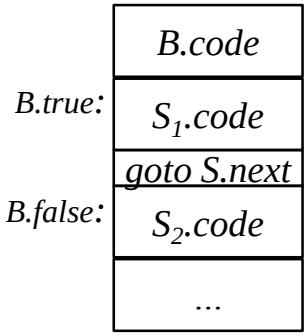
CSE 컴퓨터공학부 Department of Computer Science & Engineering

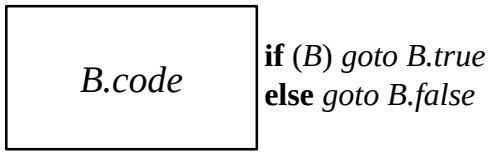
Translation of Boolean Expressions

- Boolean expression as part of control-flow statements
  - `if (B) S1`

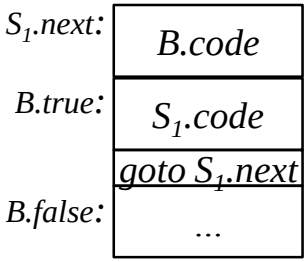


  - `if (B) S1 else S2`





  - `while (B) S1`



CSE 컴퓨터공학부 Department of Computer Science & Engineering

Translation of Boolean Expressions

- SDD for Translating Boolean Expressions – Control-Flow Statements

Production			***Semantic Rule***
P	→	S	$S.next$ = new Label() $P.code$ = $S.code$ || genTAC("label" $S.next$)
S	→	**assign**	$S.code$ = **assign**.$code$
S	→	**if (** B **)** S_1	$B.true$ = new Label() $B.false$ = $S_1.next$ = $S.next$ $S.code$ = $B.code$ || genTAC("label" $B.true$) || $S_1.code$
S	→	**if (** B **)** S_1 **else** S_2	$B.true$ = new Label() $B.false$ = new Label() $S_1.next$ = $S_2.next$ = $S.next$ $S.code$ = $B.code$ || genTAC("label" $B.true$) || $S_1.code$ || genTAC('goto' $S.next$) || genTAC("label" $B.false$) || $S_2.code$
S	→	**while (** B **)** S_1	$B.true$ = new Label() $B.false$ = $S.next$ $S_1.next$ = new Label() $S.code$ = genTAC("label" $S_1.next$) || $B.code$ || genTAC("label" $B.true$) || $S_1.code$ || genTAC('goto' $S_1.next$)
S	→	S_1 S_2	$S_1.next$ = new Label() $S_2.next$ = $S.next$ $S.code$ = $S_1.code$ || genTAC("label" $S_1.next$) || $S_2.code$

CSE 컴퓨터공학부 Department of Computer Science & Engineering

Translation of Boolean Expressions

- SDD for Translating Boolean Expressions – Boolean Expressions

Production			***Semantic Rule***
B	→	B_1 || B_2	$B_1.true$ = $B.true$ $B_1.false$ = new Label() $B_2.true$ = $B.true$ $B_2.false$ = $B.false$ $B.code$ = $B_1.code$ || genTAC("label" $B_1.false$) || $B_2.code$
B	→	B_1 **&&** B_2	$B_1.true$ = new Label() $B_1.false$ = $B.false$ $B_2.true$ = $B.true$ $B_2.false$ = $B.false$ $B.code$ = $B_1.code$ || genTAC("label" $B_1.true$) || $B_2.code$
B	→	**!** B_1	$B_1.true$ = $B.false$ $B_1.false$ = $B.true$ $B.code$ = $B_1.code$
B	→	E_1 **relOp** E_2	$B.code$ = $E_1.code$ || $E_2.code$ || genTAC("if" $E_1.addr$ **relOp**.*op* $E_2.addr$ "goto" $B.true$) || genTAC("goto" $B.false$)
B	→	**true**	$B.code$ = genTAC('goto' $B.true$)
B	→	**false**	$B.code$ = genTAC('goto' $B.false$)

CSE 컴퓨터공학부 Department of Computer Science & Engineering

Translation of Boolean Expressions

- Example

```
if (x < 100 || x > 200 && x != y) x = 0;
```

```
        if x < 100 goto L2
        goto L3
L3:     if x > 200 goto L4
        goto L1
L4:     if x != y goto L2:
        goto L1
L2:     x = 0
L1:
```

- Avoiding redundant gotos: see textbook 6.6.5

CSE 컴퓨터공학부 Department of Computer Science & Engineering

Translation of Boolean Expressions

- Evaluation of Boolean Values vs Jumping Code
  - two variants
    - two-pass
      generate complete AST for the input, then walk the tree
    - one-pass for statements, two passes for expressions
      use the SDD from before to generate code for expressions appearing in control flow constructs, but two passes (as above) for expressions
  - Example

Production

$S \rightarrow$ **id** := E | **if** (E) S | **while** (E) S | S ; S

$E \rightarrow$ E || E | E && E | E **relOp** E | E + E | (E) | **id** | **true** | **false**

    - use separate code-generation functions
      - jump(E.n) to generate jumping code, E.n representing the syntax tree
      - rvalue(E.n) to generate code valuating the expression into a temporary

CSE 컴퓨터공학부 Department of Computer Science & Engineering

Translation of Switch Statements

switch (E **) begin**
  case Const_1 **:** S_1
  case Const_2 **:** S_2
  …
  case Const_{n-1} **:** S_{n-1}
  default: S_n
end

- Switch translation steps
  1. evaluate E
  2. find matching $Const_i$ in the list of possible cases; use **default** if no $Const_i$ matches (and the **default** statement is provided)
  3. execute associated statement S_i (if any)

CSE 컴퓨터공학부 Department of Computer Science & Engineering

Translation of Switch Statements

- Translation of the n-way branch (step 2)
  - lookup table
    - n small: lookup table
    - n large (~ > 10 entries): hash table

    + compiler-generated lookup code

  - jump table
    - direct jump, very efficient
    - if $MAX(Case_i) - MIN(Case_i) / \# Cases$ is close to 1, i.e., there are not too many holes within the range of all possible cases
    - generate jump table jtbl (fill undefined cases with the location of **default**.*code* or *S.next*),a range check, and one indirect jump to

      jump jtbl[E.val – $MIN(Case_i)$]

CSE 컴퓨터공학부 Department of Computer Science & Engineering

Translation of Switch Statements

- Generate high-level code, delegate decision of multiway branch implementation to code generator

```
                    t = E.val
                    goto test
L_Const1:           S_1.code
                    goto next
L_Const2:           S_2.code
                    goto next
                    …
L_Constn-1:         S_n-1.code
                    goto next
L_default:          S_n.code
                    goto next
test:               if t = Const_1 goto L_Const1
                    if t = Const_2 goto L_Const2
                    …
                    if t = Const_n-1 goto L_Constn-1
                    goto L_default
next:               …
```


CSE 컴퓨터공학부 Department of Computer Science & Engineering

Translation of Procedure Calls

$$P(E_1, E_2, \ldots);$$
$$v := F(E_1, E_2, \ldots);$$
$$v := F(\ldots, F_1(\ldots), \ldots);$$

- Procedure/Function call translations
  - implemented through a sequence of call and optional param operations

$$P(1, E_1, 3);$$

$$\begin{aligned}
&\text{param } 1\\
&t_1 \leftarrow E_1\\
&\text{param } t_1\\
&\text{param } 3\\
&\text{call P}
\end{aligned}$$

  - may pose problems for nested calls if arguments are passed in fixed locations

CSE 컴퓨터공학부 Department of Computer Science & Engineering

Translation of Procedure Calls

- Procedure/Function call translations
  - assigning parameters to temporary variables and issuing the param statements immediately before the invocation solves the problem of nested procedure calls

foo(E_1, bar(E_2), E_3);

t1 ← E_1
t2 ← E_2
param t2
t3 ← call bar
t4 ← E_3
param t1
param t3
param t4
call foo

→ register allocation eliminates most of the temporaries and associated moves

CSE 컴퓨터공학부 Department of Computer Science & Engineering