Lecture 11

Intermediate code generator

Three Address Code (TAC)

Hyosu Kim

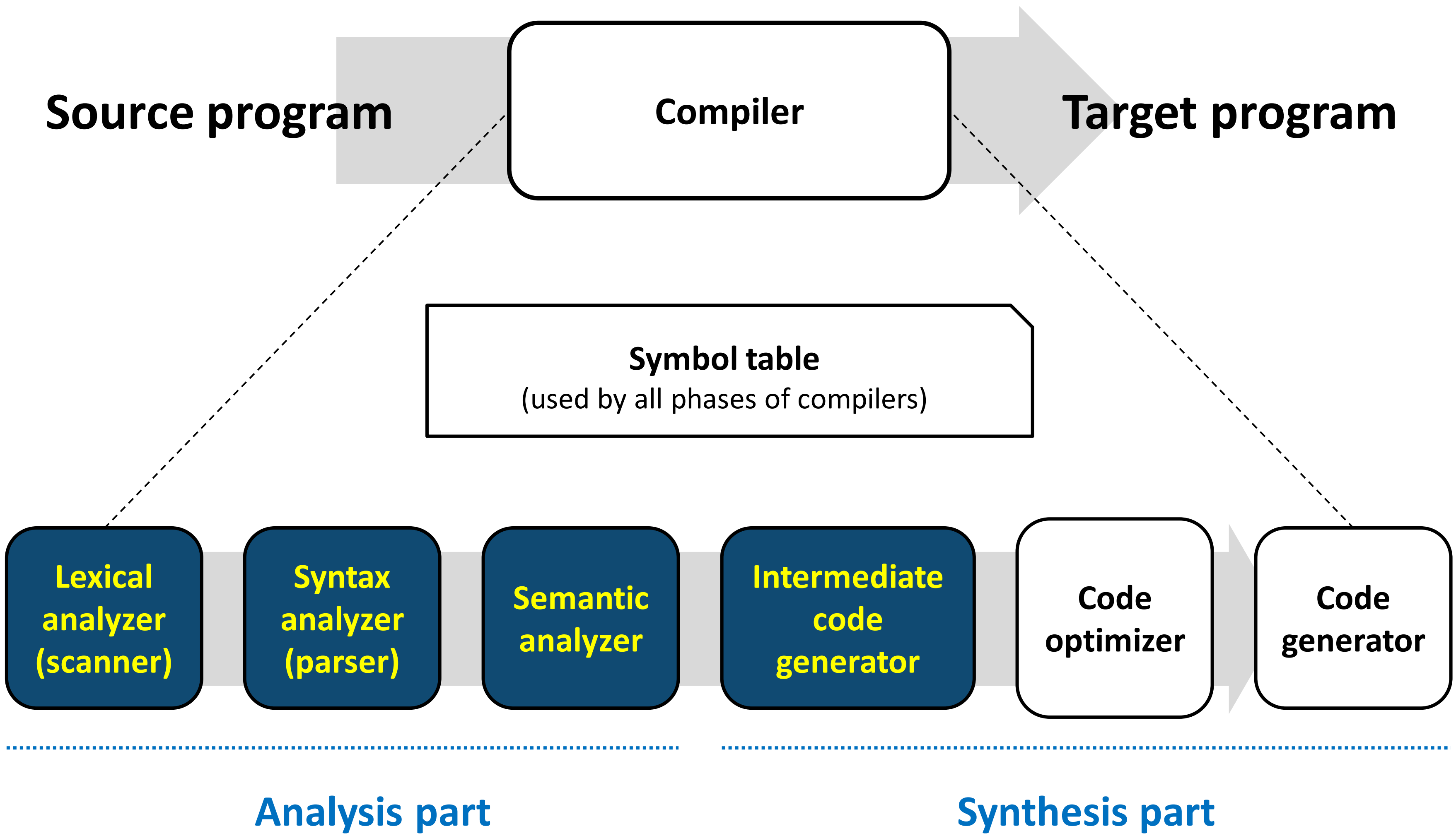
School of Computer Science and Engineering

Chung-Ang University, Seoul, Korea

https://sites.google.com/view/hyosukim

hskimhello@cau.ac.kr, hskim.hello@gmail.com

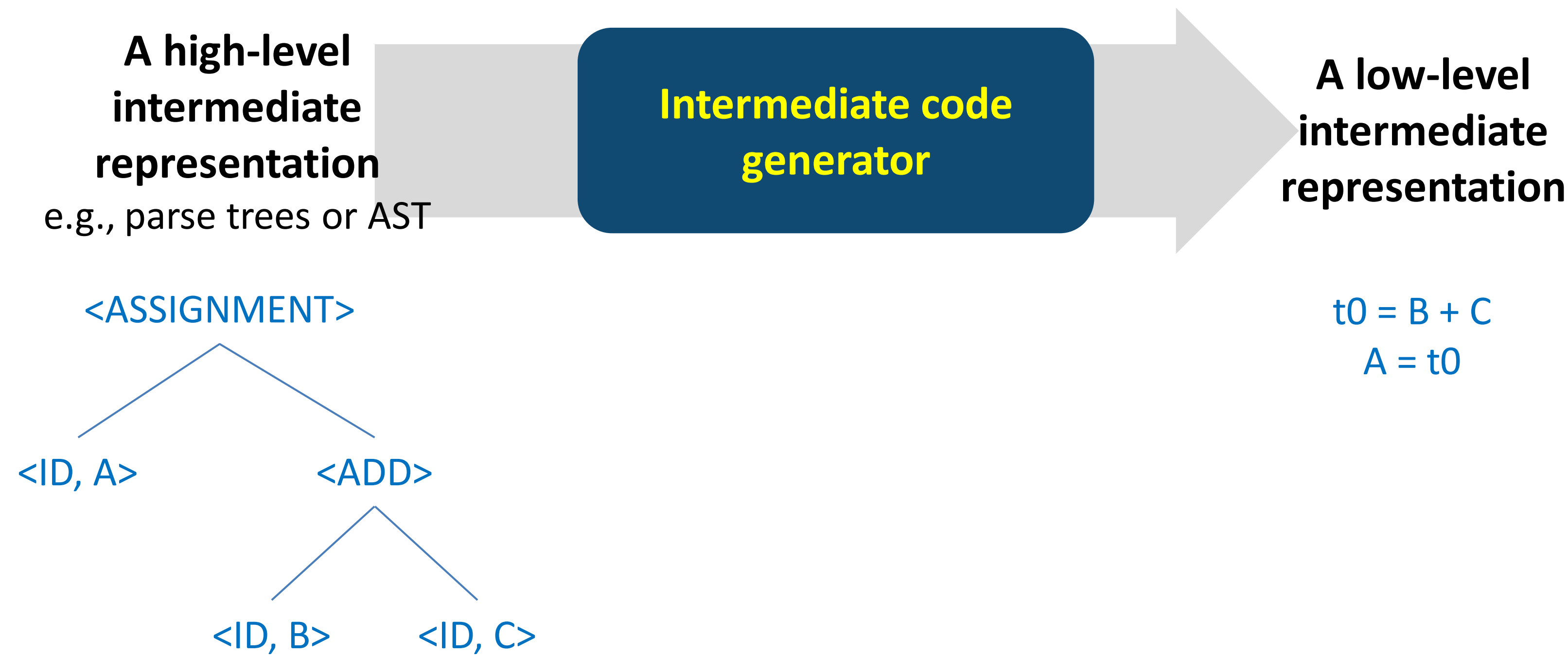
Overview


Source program
Compiler
Target program
Symbol table
(used by all phases of compilers)
Lexical analyzer (scanner)
Syntax analyzer (parser)
Semantic analyzer
Intermediate code generator
Code optimizer
Code generator
Analysis part
Synthesis part

Intermediate code generator

Translates a high-level intermediate representation (e.g., parse trees or AST)

into a low-level intermediate representation (e.g., three address code)



Intermediate code generator

Why do we use an intermediate representation?

Easy-to-understand/optimize

- Doing optimizations on an intermediate representation is much easier and clearer than that on a machine-level code
- A machine code has many constraints that inhibit optimizations

Easy-to-be-translated

- Compared to a high-level code, it looks much more like a machine-level code. Therefore, we can translate an intermediate representation to a machine-level code **with low cost**

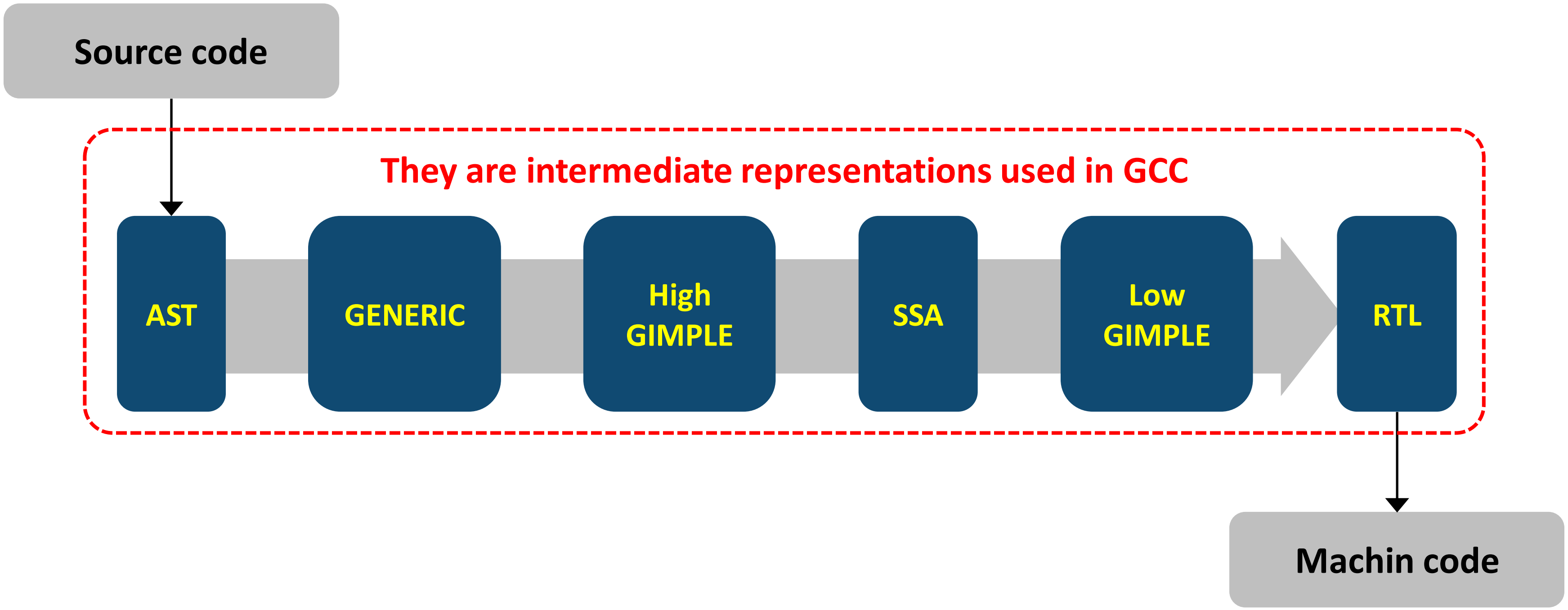
Intermediate code generator

How to design a good intermediate representation??

“Often a single compiler has multiple intermediate representations”

Different intermediate representations have different information/characteristics which can be used for optimizations

In GCC,

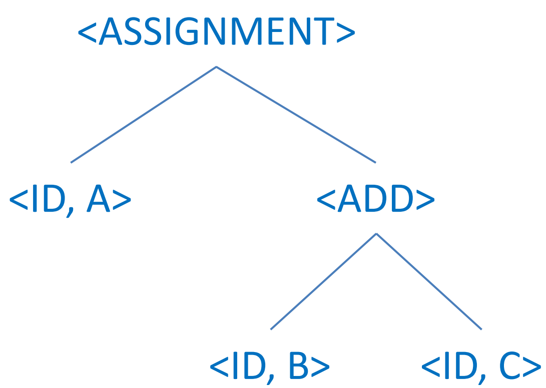


In this class,

We study two representative intermediate representations

- **AST (Abstract Syntax Tree)**

  A high-level intermediate representation (we’ve already studied)



- **TAC (Three Address Code)**

  A low-level intermediate representation (we’ll newly study)

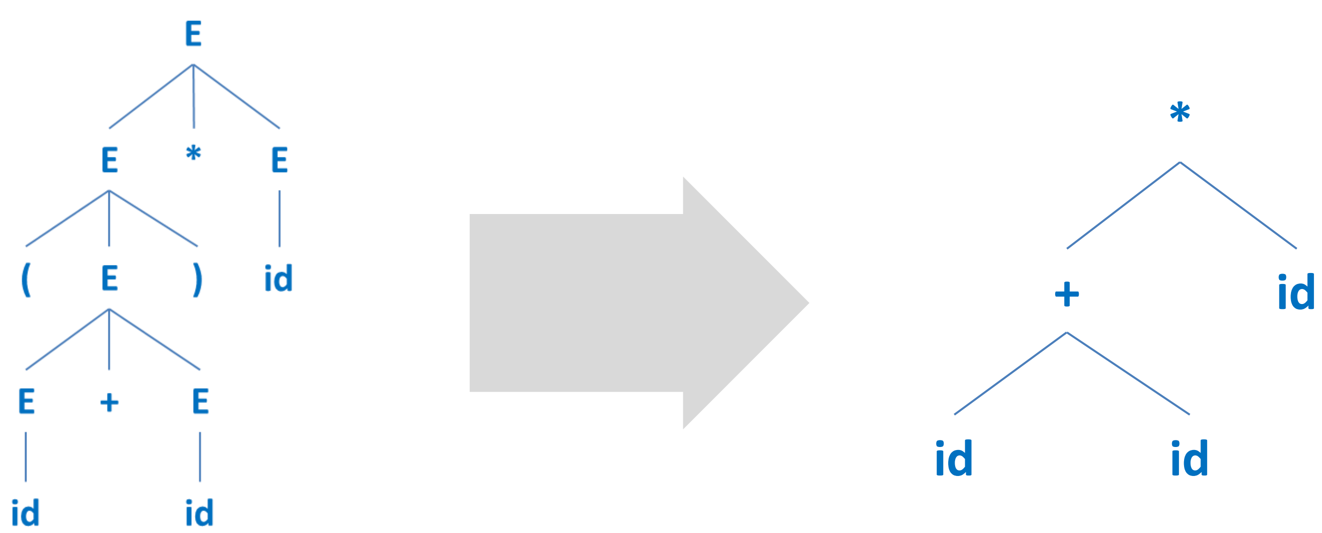
$$t_0 = B + C$$
$$A = t_0$$

AST

Abstract syntax trees look like parse trees, but without some parsing details

We can eliminate the following nodes in parse trees

1. Single-successor nodes
2. Symbols for describing syntactic details
3. Non-terminals with an operator and arguments as their child nodes



AST

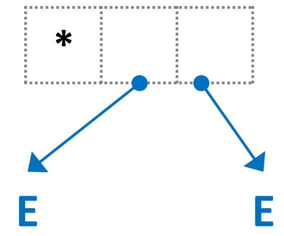
Abstract syntax trees look like parse trees, but without some parsing details

AST can be constructed by using semantic actions

- Example

  For a CFG G: $E \rightarrow E + E | E * E | (E) | id$

Production	Semantic action
$E \rightarrow E_1 + E_2$	$E.node = new\ Node('+', E_1.node, E_2.node)$
$E \rightarrow E_1 * E_2$	$E.node = new\ Node('*', E_1.node, E_2.node)$
$E \rightarrow (E_1)$	$E.node = E_1.node$
$E \rightarrow id$	$E.node = new\ Leaf(id, id.value)$



An example sequence of derivations for $id * id$

$E \Rightarrow_{lm} E * E$

AST

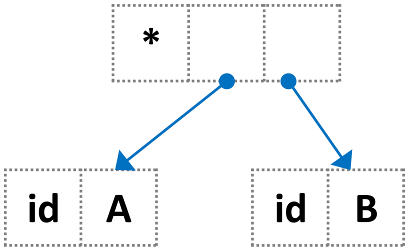
Abstract syntax trees look like parse trees, but without some parsing details

AST can be constructed by using semantic actions

- Example

  For a CFG G: $E \rightarrow E + E | E * E | (E) | id$

Production	Semantic action
$E \rightarrow E_1 + E_2$	$E.node = new\ Node(‘+’, E_1.node, E_2.node)$
$E \rightarrow E_1 * E_2$	$E.node = new\ Node(‘*’, E_1.node, E_2.node)$
$E \rightarrow (E_1)$	$E.node = E_1.node$
$E \rightarrow id$	$E.node = new\ Leaf(id, id.value)$



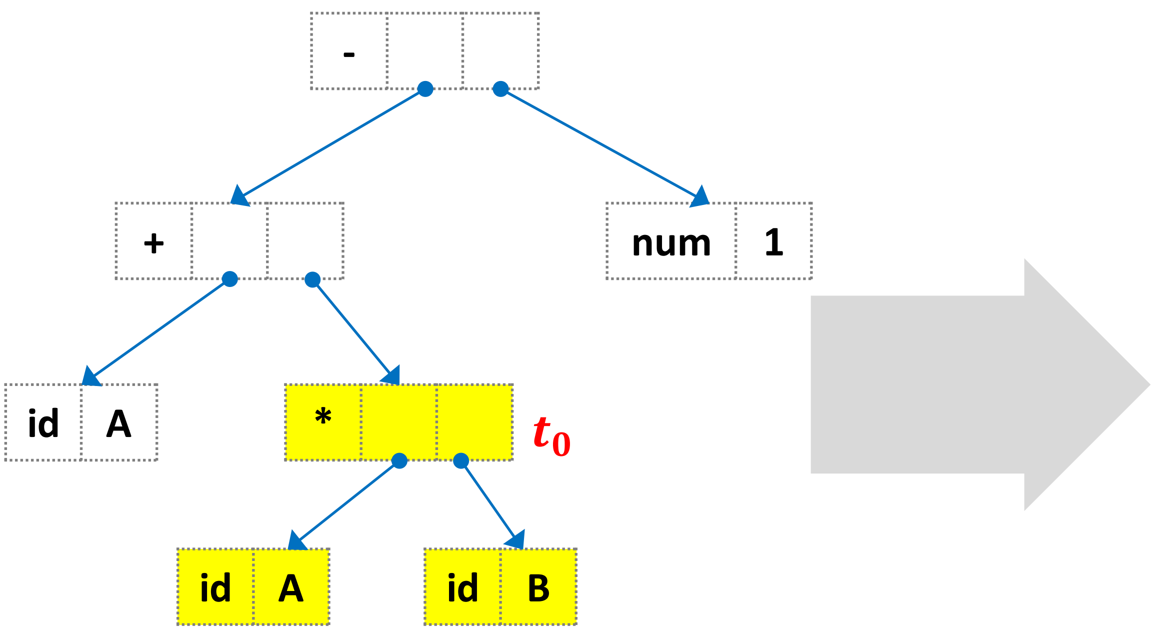
An example sequence of derivations for $id * id$

$E \Rightarrow_{lm} E * E \Rightarrow^{*}_{lm} id * id$

TAC (Three Address Code)

A high-level assembly where each operation has at most three operands

A linearized representation of AST

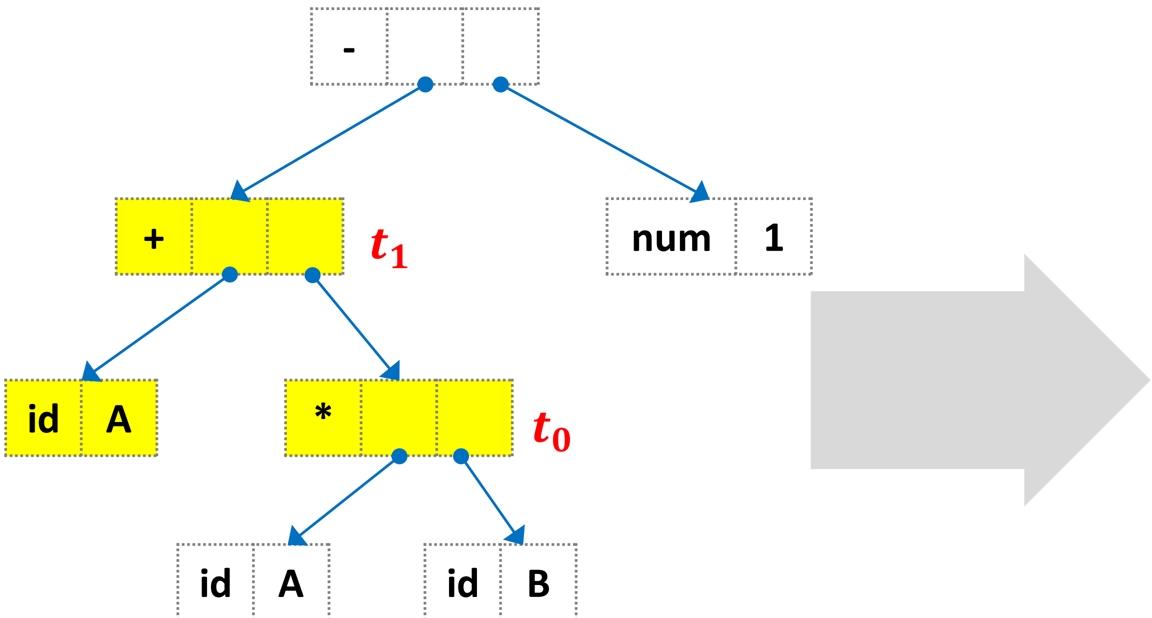


$$t_0 = A * B$$

TAC (Three Address Code)

A high-level assembly where each operation has at most three operands

A linearized representation of AST



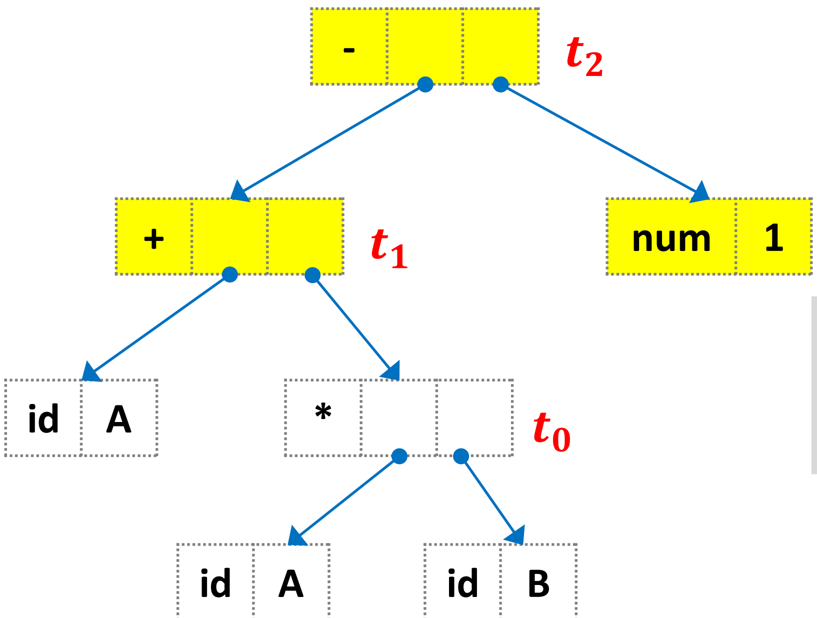
$$t_0 = A * B$$

$$t_1 = A + t_0$$

TAC (Three Address Code)

A high-level assembly where each operation has at most three operands

A linearized representation of AST



$$t_0 = A * B$$

$$t_1 = A + t_0$$

$$t_2 = t_1 - 1$$

TAC (Three Address Code)

A high-level assembly where each operation has at most three operands

A linearized representation of AST

Components of TAC

- Operands (= addresses)
  - Explicit variables: e.g., A, B, C
  - Constants: e.g., 1
  - Temporary variables: e.g., t_0, t_1, t_2

$$\boldsymbol{t_0 = A * B}$$

$$\boldsymbol{t_1 = A + t_0}$$

$$\boldsymbol{t_2 = t_1 - 1}$$

- Operators (= instructions)

  e.g., *, +, -, =

Types of TAC: variable assignment

Copy operation: explicit or temporary variable = any kind of operand

e.g., $A = 1,\ t_0 = 2,\ t_0 = A,\ A = t_0$

Binary operation

explicit or temporary variable = any kind of operand **binary operator** any kind of operand

- Binary operators
  - Arithmetic operators: +, -, *, /, %,
  - Boolean operators: &&, | |, …
  - Comparison operators: ==, !=, <, >, <=, >=, …

e.g., $A = B + C,\ A = 1 * 2,\ \ t_0 = A||1,\ A = t_0 \leq 0$

Unary operation

Explicit or temporary variable = **unary operator** any kind of operand

- Unary operators: -, !

e.g., $t_0 = -2,\ t_0 = !\,true;$

Types of TAC: control flow statements

Unconditional jump with label (e.g., L_n)

- $goto\ L_n$

  …

  L_n:

  …

Conditional branch

- $if\ x\ GOTO\ L_n$

- $ifnot\ x\ GOTO\ L_n$

  Where x is a Boolean variable that evaluates true or false

Types of TAC: more...

Call procedures

Make a procedure call with n parameters

$foo(x_1, x_2, \dots, x_n)$ ⇒

$param\ x_1$

$param\ x_2$

$\dots$

$param\ x_n$

$call\ foo, n$

$y = foo(x_1, x_2, \dots, x_n)$ ⇒

$param\ x_1$

$param\ x_2$

$\dots$

$param\ x_n$

$y = call\ foo, n$

Array operations

$A[i] = B$

$A = B[i]$

Return statements

$return\ x$

Types of TAC

Example #1

$if\ (x == y)\ \{$

$\quad x = x + 1;$

$\}\ \text{else}\ \{$

$\quad x = x + 2;$

$\}$

$x = x + 3;$

$return\ x;$

$t_0 = x == y$

$if\ t_0\ \ goto\ \ L_0$

$goto\ L_1$

L_0:

$\quad t_0 = x + 1$

$\quad x = t_0$

$\quad goto\ L_2$

L_1:

$\quad t_1 = x + 2$

$\quad x = t_1$

L_2:

$\quad t_2 = x + 3$

$\quad x = t_2$

$\quad return\ x$

Types of TAC

Example #2

$while\ (x < y)\ \{$

$\quad x = x + 1;$

$\}$

$\mathrm{foo}(x - 1);$

L_0:

$\quad t_0 = x < y$

$\quad ifnot\ t_0\ \ goto\ \ L_1$

$\quad t_1 = x + 1$

$\quad x = t_1$

$\quad goto\ L_0$

L_1:

$\quad t_2 = \ x - 1$

$\quad param\ t_2$

$\quad call\ foo, 1$

Types of TAC

Practice

$for\ (i = 0; i < 10; i++)\ \{$

$\quad b[i] = i + 1;$

$\}$

How to represent TAC: Quadruples

Quadruples have four fields (op, arg1, arg2, result) and they are stored in a linked list

op (e.g., +, *..)	arg1	arg2	result

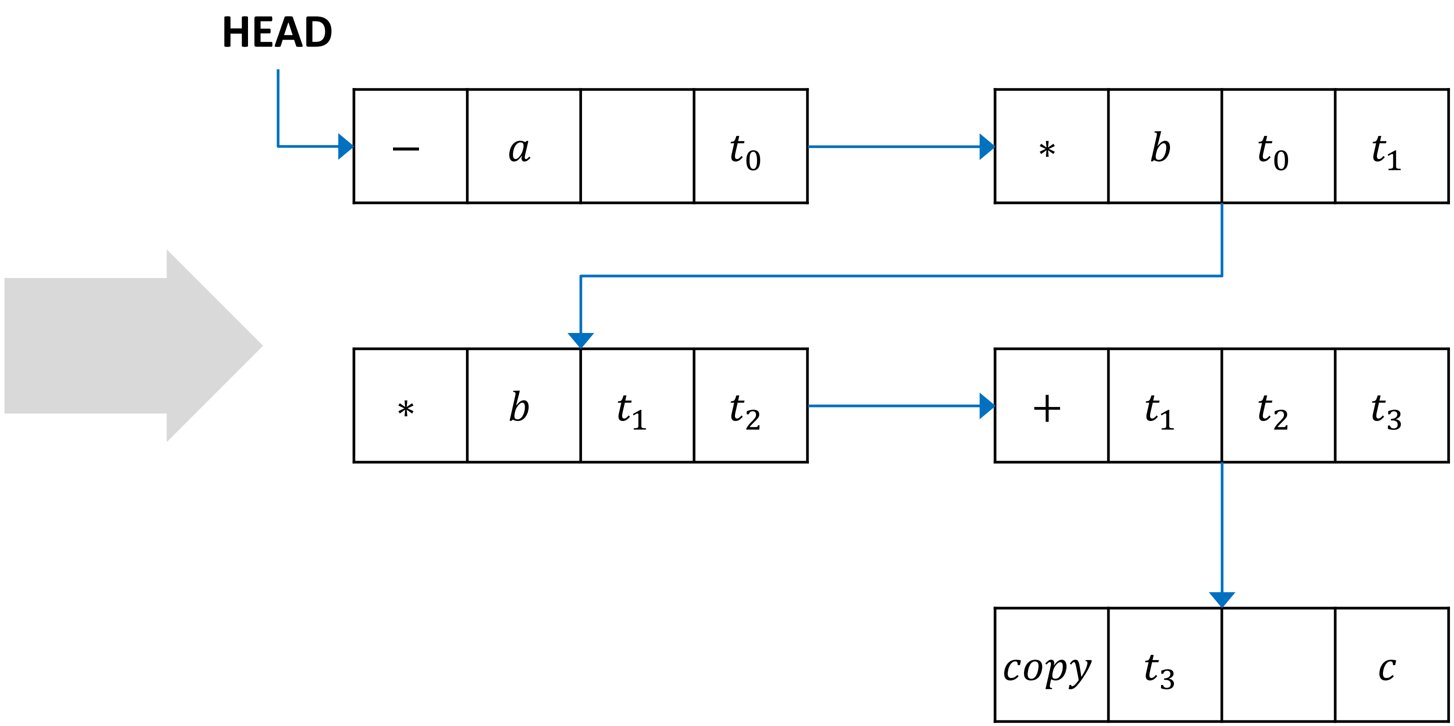
$$t_0 = -a$$

$$t_1 = b * t_0$$

$$t_2 = b * t1$$

$$t_3 = t_1 + t_2$$

$$c = t_3$$



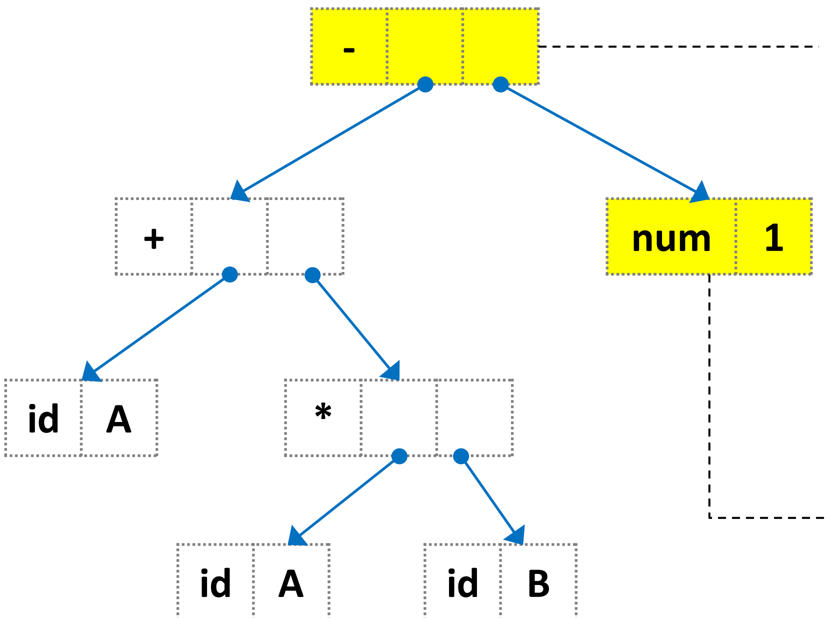
How to represent TAC: Quadruples

Examples

	op	arg1	arg2	result
Copy operation (a = b)	=	b		a
Binary operation (a = b - c)	-	b	c	a
Unary operation (a = -b)	-	b		a
L:	label	L		
goto L	goto	L		
if x goto L	if	x	L	
param x	param	x		
call f, n	call	f	n	
x = call f, n	call	f	n	x
return x	return	x		
x[i] = y	[]=	i	y	x
x = y[i]	=[]	y	i	x

AST to TAC

When we construct AST, we define the TAC construction rules for each node



TAC construction rules

- Create a new quadruple with $op = -$
- $arg1$ = the computation result of its left child
- $arg2$ = the computation result of its right child
- $result$ = a new temporary variable t_i
- Store the quadruple to the end of the linked list

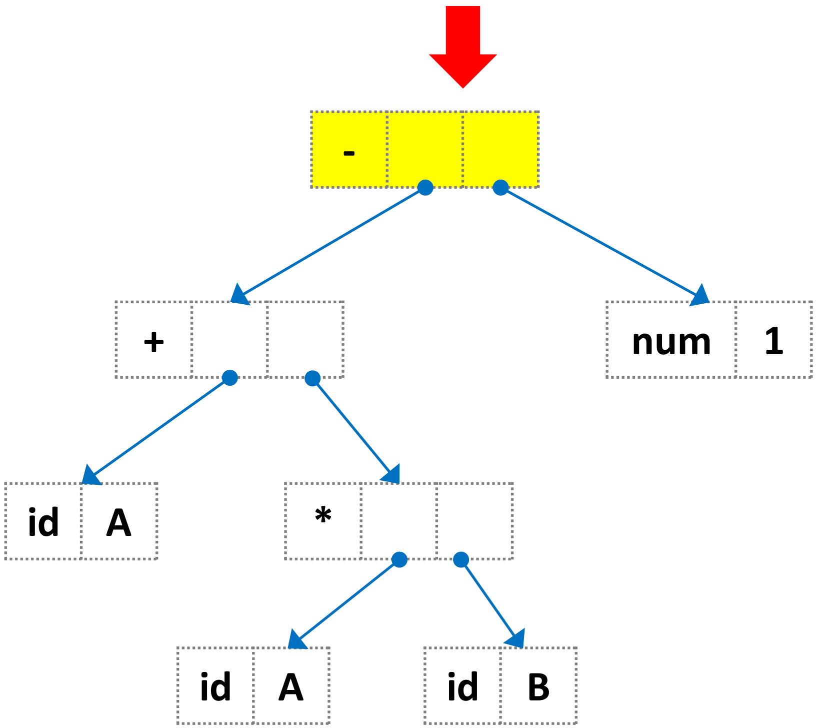
TAC construction rules

- Return its value

AST to TAC

When we construct AST, we define the TAC construction rules for each node

While traveling AST, construct TAC based on the rules



TAC construction rules

- Create a new quadruple with $op = -$

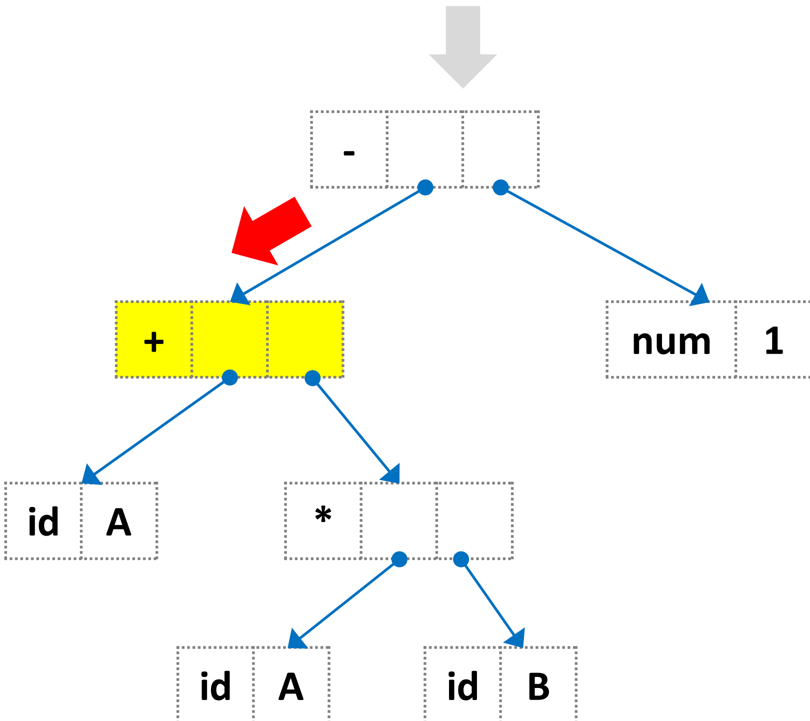
$-$	$arg1$	$arg2$	$result$

- $arg1$ = the computation result of its left child

AST to TAC

When we construct AST, we define the TAC construction rules for each node

While traveling AST, construct TAC based on the rules



TAC construction rules

- Create a new quadruple with $\boldsymbol{op} = +$

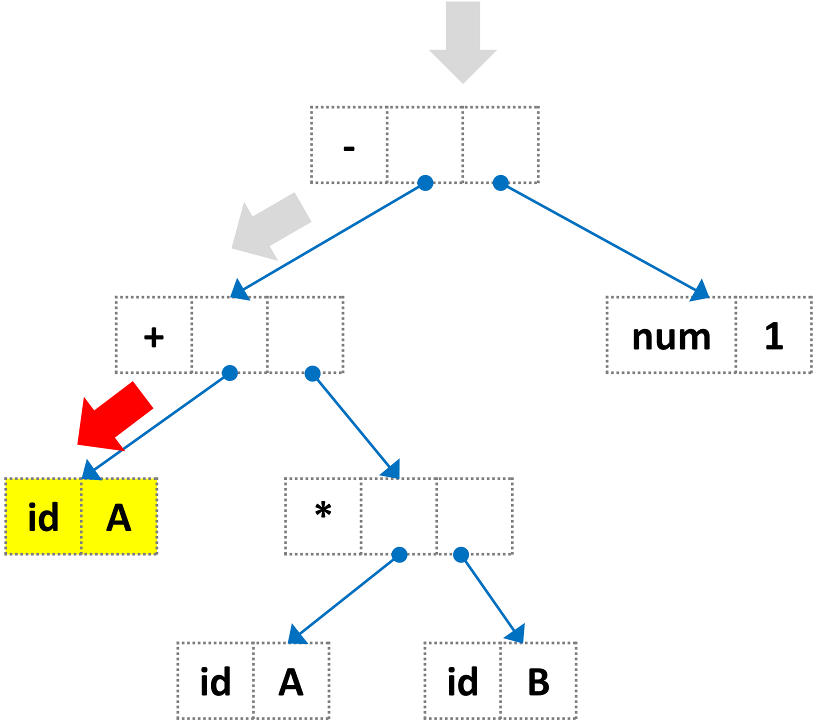
$+$	$arg1$	$arg2$	$result$

- $arg1$ = the computation result of its left child

AST to TAC

When we construct AST, we define the TAC construction rules for each node

While traveling AST, construct TAC based on the rules



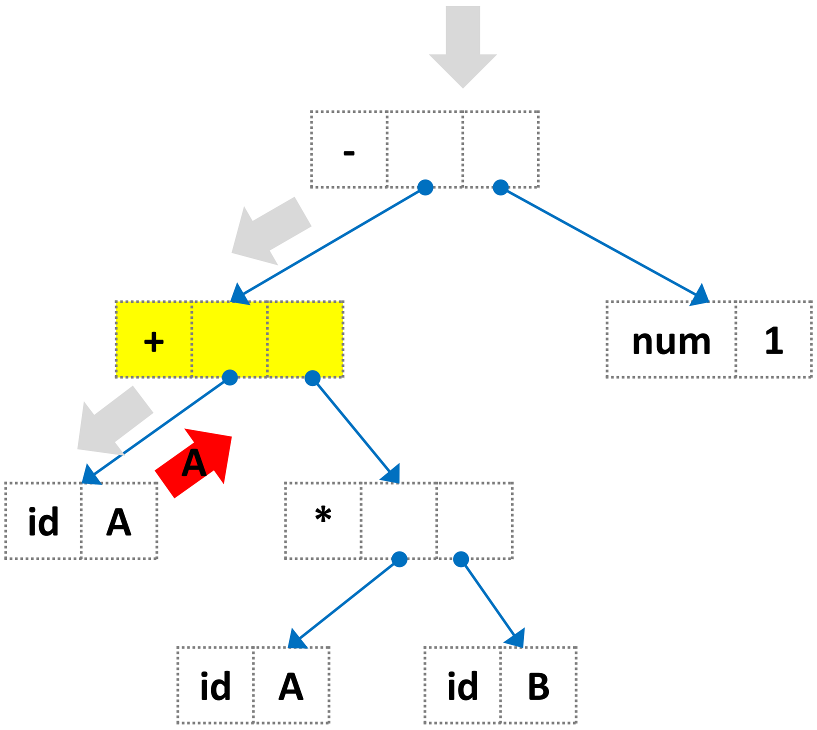
TAC construction rules

- Return its name

AST to TAC

When we construct AST, we define the TAC construction rules for each node

While traveling AST, construct TAC based on the rules



TAC construction rules

- Create a new quadruple with $op = +$

$+$	$arg1$	$arg2$	$result$

- $arg1$ = the computation result of its left child

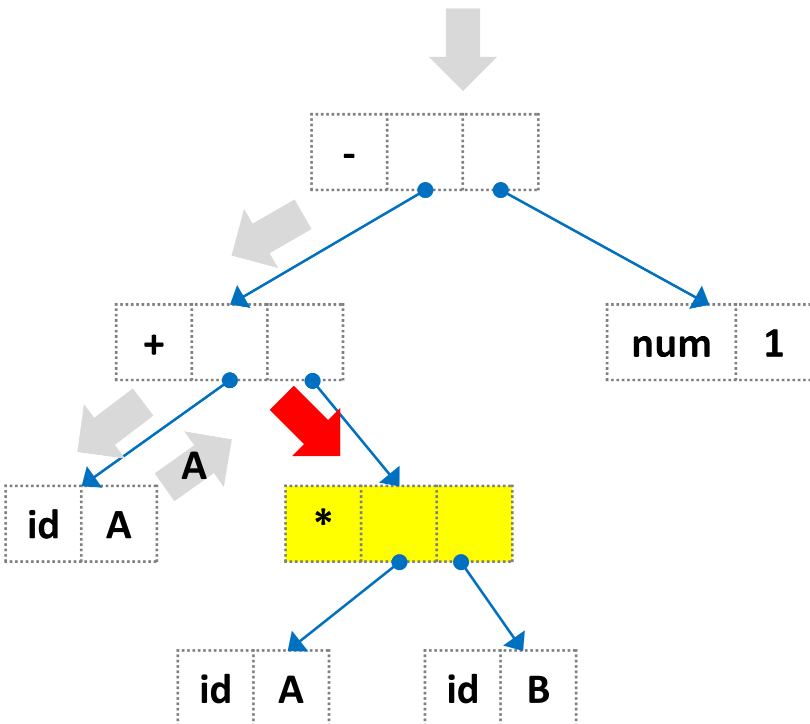
$+$	A	$arg2$	$result$

- $arg2$ = the computation result of its right child

AST to TAC

When we construct AST, we define the TAC construction rules for each node

While traveling AST, construct TAC based on the rules



TAC construction rules

- Create a new quadruple with $\boldsymbol{op} = *$

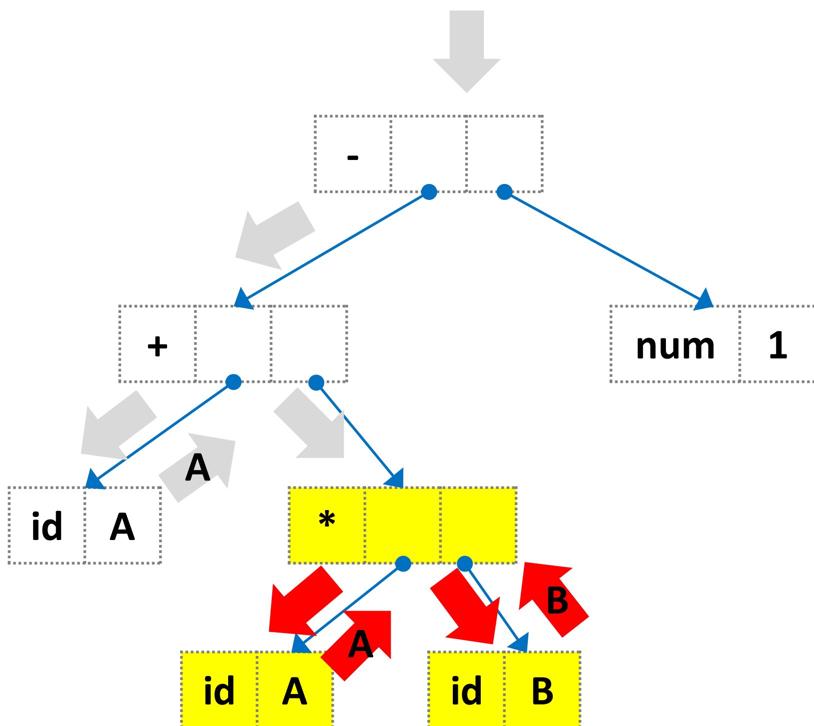
$*$	$arg1$	$arg2$	$result$

- $arg1$ = the computation result of its left child

AST to TAC

When we construct AST, we define the TAC construction rules for each node

While traveling AST, construct TAC based on the rules



TAC construction rules

- Create a new quadruple with $op = *$

$*$	$arg1$	$arg2$	$result$

- $arg1$ = the computation result of its left child

$*$	A	$arg2$	$result$

- $arg2$ = the computation result of its right child

$*$	A	B	$result$

- $result$ = a new temporary variable t_i

$*$	A	B	t_0

- Store the quadruple to the end of the linked list

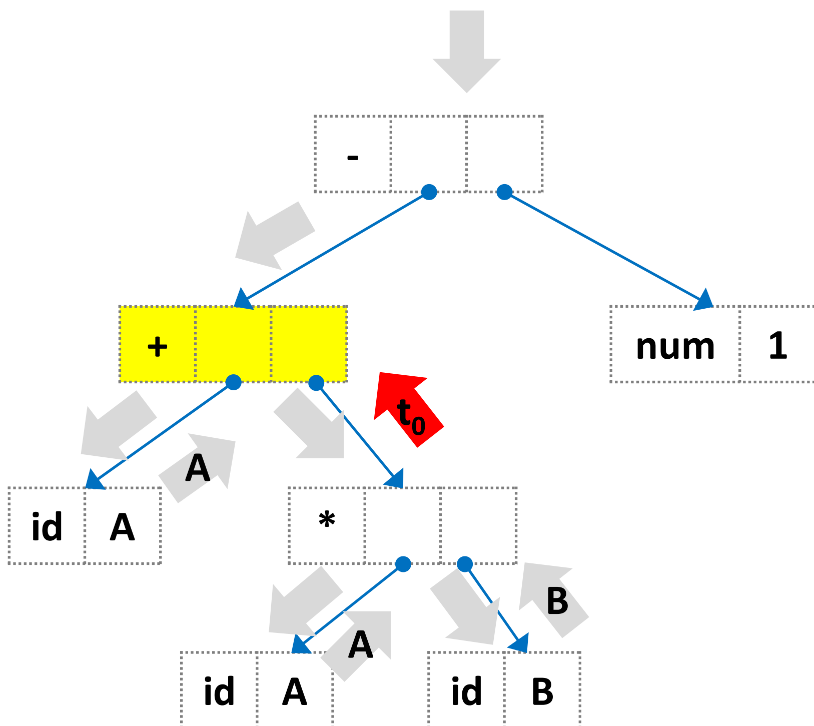
HEAD →

$*$	A	B	t_0

AST to TAC

When we construct AST, we define the TAC construction rules for each node

While traveling AST, construct TAC based on the rules



TAC construction rules

- Create a new quadruple with $op = +$

$+$	$arg1$	$arg2$	$result$

- $arg1$ = the computation result of its left child

$+$	A	$arg2$	$result$

- $arg2$ = the computation result of its right child

$+$	A	t_0	$result$

- $result$ = a new temporary variable t_i

$+$	A	t_0	t_1

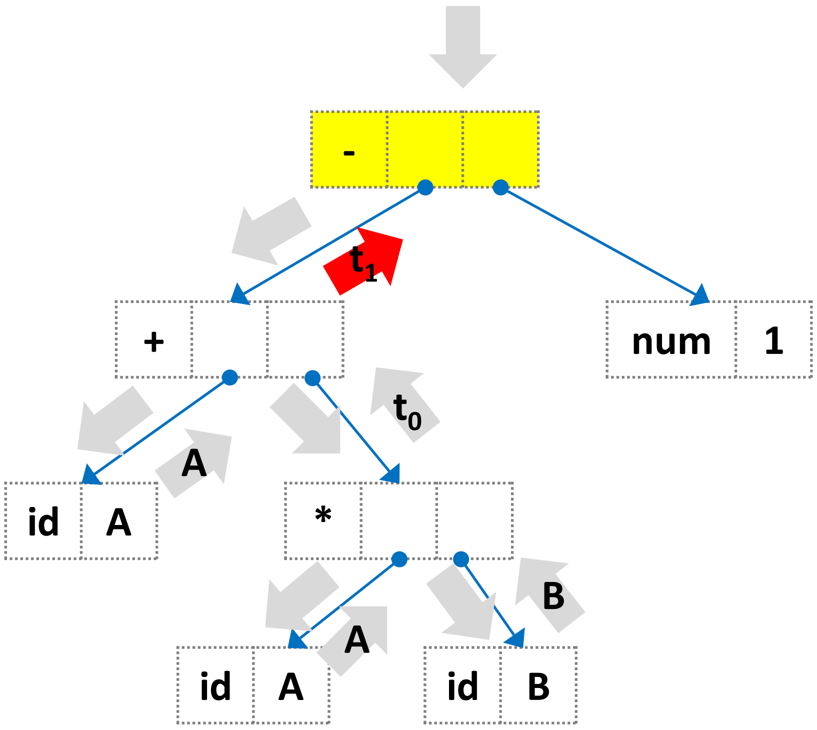
- Store the quadruple to the end of the linked list

HEAD → | * | A | B | t_0 | → | + | A | t_0 | t_1 |

AST to TAC

When we construct AST, we define the TAC construction rules for each node

While traveling AST, construct TAC based on the rules



TAC construction rules

- Create a new quadruple with $op = -$

$-$	$arg1$	$arg2$	$result$

- $arg1$ = the computation result of its left child

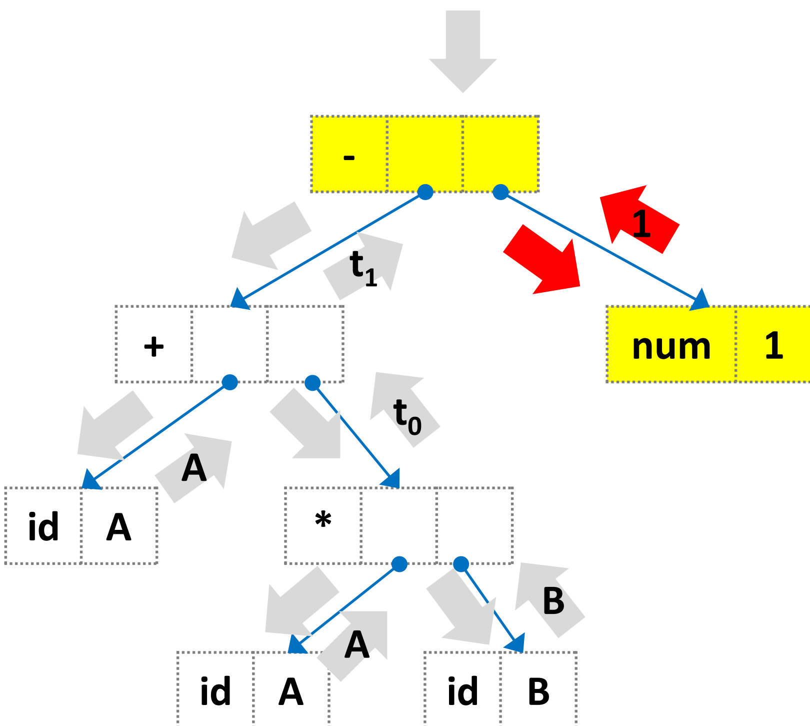
$-$	t_1	$arg2$	$result$

- $arg2$ = the computation result of its right child

AST to TAC

When we construct AST, we define the TAC construction rules for each node

While traveling AST, construct TAC based on the rules



TAC construction rules

- Create a new quadruple with $op = -$

$-$	$arg1$	$arg2$	$result$

- $arg1$ = the computation result of its left child

$-$	t_1	$arg2$	$result$

- $arg2$ = the computation result of its right child

$-$	t_1	1	$result$

- $result$ = a new temporary variable t_i

$-$	t_1	1	t_2

- Store the quadruple to the end of the linked list

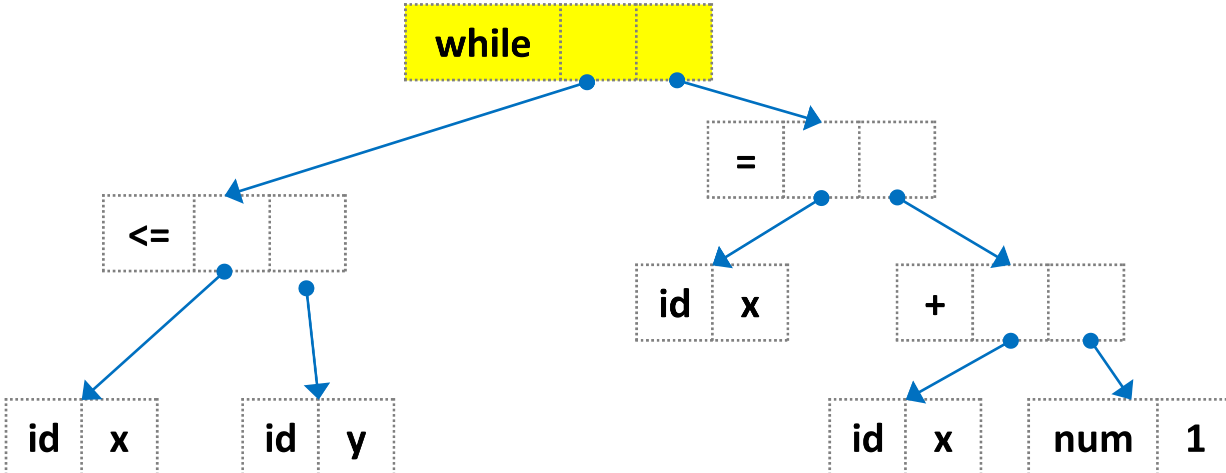
HEAD → [* | A | B | t_0] → [+ | A | t_0 | t_1] → [- | t_1 | 1 | t_2]

AST to TAC

Practice

TAC construction rules for while

- Create and store a new quadruple with $op = label, arg1 = a\ new\ label\ L_i$
- Create a new quadruple with $op = ifnot$
- $arg1$ = the computation result of its left child (compute condition)
- $arg2$ = another new label L_j
- Store the quadruple
- Compute the right child (compute the while statement's block)
- Create and store a new quadruple with $op = goto, arg1 = L_i$
- Create and store a new quadruple with $op = label, arg1 = L_j$



Summary: intermediate code generator

Translates a high-level intermediate representation (e.g., parse trees or AST)

into a low-level intermediate representation (e.g., three address code)