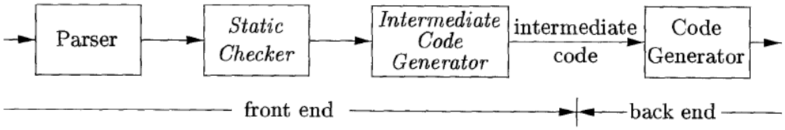
Intermediate-Code Generation

Intermediate-Code Generation



Three-Address Code

- In three-address code, there is at most **one operator** on the **right side of an instruction**
- Thus a **source-language expression like x+y*z** might be translated into the sequence of three-address instructions

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

Common three-address instructions

1. Assignment instructions of the form $x = y \; op \; z$, where op is a binary arithmetic or logical operation, and x, y, and z are addresses.
2. Assignments of the form $x = op \; y$, where op is a unary operation. Essential unary operations include unary minus, logical negation, shift operators, and conversion operators that, for example, convert an integer to a floating-point number.
3. *Copy instructions* of the form $x = y$, where x is assigned the value of y.
4. An unconditional jump `goto` L. The three-address instruction with label L is the next to be executed.
5. Conditional jumps of the form `if` x `goto` L and `ifFalse` x `goto` L. These instructions execute the instruction with label L next if x is true and false, respectively. Otherwise, the following three-address instruction in sequence is executed next, as usual.

6. Conditional jumps such as `if` x *relop* y `goto` L, which apply a relational operator (`<`, `==`, `>=`, etc.) to x and y, and execute the instruction with label L next if x stands in relation *relop* to y. If not, the three-address instruction following `if` x *relop* y `goto` L is executed next, in sequence.

7. Procedure calls and returns are implemented using the following instructions: `param` x for parameters; `call` p, n and y = `call` p, n for procedure and function calls, respectively; and `return` y, where y, representing a returned value, is optional. Their typical use is as the sequence of three-address instructions

$$
\begin{array}{l}
\texttt{param } x_1 \\
\texttt{param } x_2 \\
\dots \\
\texttt{param } x_n \\
\texttt{call } p, n
\end{array}
$$

generated as part of a call of the procedure $p(x_1, x_2, \dots, x_n)$. The integer n, indicating the number of actual parameters in "`call` p, n,"

8. Indexed copy instructions of the form $x = y[i]$ and $x[i] = y$. The instruction $x = y[i]$ sets x to the value in the location i memory units beyond location y. The instruction $x[i] = y$ sets the contents of the location i units beyond x to the value of y.

Common three-address instructions

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

$$
\begin{array}{ll}
L: & t_1 = i + 1 \\
 & i = t_1 \\
 & t_2 = i * 8 \\
 & t_3 = a\,[\,t_2\,] \\
 & \text{if } t_3 < v \text{ goto } L
\end{array}
$$

Data structures for TAC

Quadruples

A *quadruple* (or just "*quad*") has four fields, which we call *op*, arg_1, arg_2, and *result*. The *op* field contains an internal code for the operator. For instance, the three-address instruction $x = y + z$ is represented by placing + in *op*, y in arg_1, z in arg_2, and x in *result*. The following are some exceptions to this rule:

1. Instructions with unary operators like $x = \texttt{minus}\, y$ or $x = y$ do not use arg_2. Note that for a copy statement like $x = y$, *op* is =, while for most other operations, the assignment operator is implied.

2. Operators like `param` use neither arg_2 nor *result*.

3. Conditional and unconditional jumps put the target label in *result*.

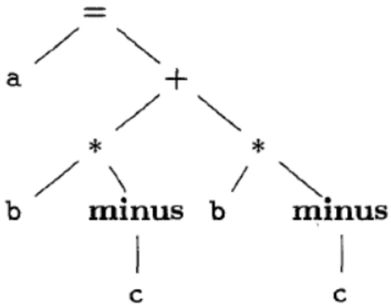
Data structures for TAC

Quadruples

Three-address code for the assignment `a = b*-c+b*-c;`

$$
\begin{aligned}
t_1 &= \texttt{minus } c \\
t_2 &= b * t_1 \\
t_3 &= \texttt{minus } c \\
t_4 &= b * t_3 \\
t_5 &= t_2 + t_4 \\
a &= t_5
\end{aligned}
$$

(a) Three-address code



(a) Syntax tree

Data structures for TAC

Quadruples

Three-address code for the assignment `a = b*-c+b*-c;`

$$
\begin{aligned}
t_1 &= \texttt{minus } c \\
t_2 &= b * t_1 \\
t_3 &= \texttt{minus } c \\
t_4 &= b * t_3 \\
t_5 &= t_2 + t_4 \\
a &= t_5
\end{aligned}
$$

(a) Three-address code

	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
	...			

(b) Quadruples

Data structures for TAC
Triples

- A triple has **only three fields**, which we call **op, arg1, and arg2**
- Note that the **result field in Quad** is used primarily for temporary names.
- Using **triples**, we refer to the **result of an operation** x op y **by its position**, rather than by an explicit temporary name.
- Thus, instead of the **temporary t**, a triple representation would refer to **position (0)**.
- **Parenthesized numbers** represent pointers into the **triple structure** itself.

Data structures for TAC
Triples

Three-address code for the assignment `a = b*-c+b*-c;`

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

(a) Three-address code

	op	arg_1	arg_2
0	minus	c	
1	*	b	(0)
2	minus	c	
3	*	b	(2)
4	+	(1)	(3)
5	=	a	(4)
		...	

(b) Triples

Benefit of Quad over Triples

- A benefit of **quadruples** over triples can be seen in an **optimizing compiler**, where **instructions are often moved around**.
    - With quadruples, **if we move an instruction** that **computes a temporary t**, then the instructions that **use t** require **no change**.

- With **triples**, the **result** of an operation is referred to by **its position**

- So **moving an instruction** may require us to **change all references** to that result.

Indirect triples

- Consist of a **listing of pointers** to triples,
    - Rather than a listing of triples themselves.
- For example, use an **array *instruction*** to list **pointers to triples** in the desired order.

With indirect triples, an **optimizing compiler** can move an instruction by **reordering the instruction list**, without affecting the triples themselves.

	instruction
35	(0)
36	(1)
37	(2)
38	(3)
39	(4)
40	(5)
	...

	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 Form

Two distinctive aspects distinguish SSA from three-address code.
(a) The first is that **all assignments in SSA** are to **variables with distinct names**; hence the term static single-assignment.

$$\begin{array}{l} p = a + b \\ q = p - c \\ p = q * d \\ p = e - p \\ q = p + q \end{array}$$

(a) Three-address code.

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

(b) Static single-assignment form.

Static Single-Assignment Form

Two distinctive aspects distinguish SSA from three-address code.
(a) The first is that **all assignments in SSA** are to **variables with distinct names**; hence the term static single-assignment.

(b)

The same variable may be defined in two different control-flow paths in a program. For example, the source program

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

has two control-flow paths in which the variable x gets defined. If we use different names for x in the true part and the false part of the conditional statement, then which name should we use in the assignment y = x * a? Here is where the second distinctive aspect of SSA comes into play. SSA uses a notational convention called the ϕ-function to combine the two definitions of x:

$$\texttt{if (flag) } x_1 = -1\texttt{; else } x_2 = 1;$$
$$x_3 = \phi(x_1, x_2);$$

Static Single-Assignment Form

Here, $\phi(\mathbf{x}_1, \mathbf{x}_2)$ has the value $\mathbf{x}_1$ if the control flow passes through the true part of the conditional and the value $\mathbf{x}_2$ if the control flow passes through the false part. That is to say, the ϕ-function returns the value of its argument that corresponds to the control-flow path that was taken to get to the assignment-statement containing the ϕ-function.

Major translation classes of Three address code generation

(a) Declaration statements (+ handling data type and storage)

(b) Expressions and statements

(a) Control flow statements

Declaration statement

Representing data types: **Type Expressions**

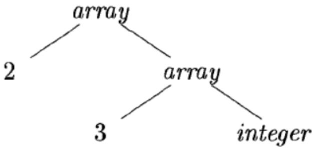
Types have structure, which we shall represent using type expressions.

- A **type expression is either a basic type** (*boolean, char, integer, float,* and *void*)

  or
- is formed by **applying an operator** called a **type constructor** to a type expression.
- A **type expression** can be formed by applying the **array type constructor** to a **number and a type expression**.

Declaration statement

- The array type int [2] [3] can be read as "array of 2 arrays of 3 integers each"
- Can be represented as a **type expression** array(2, array(3, integer)).
- This type is represented by the tree.



- The **operator array** takes **two parameters**, a number and a type.
    - Here the **type expression** can be formed by applying the **array type constructor** to a number and a type expression.

Declaration statement Example SDD

PRODUCTION	SEMANTIC RULES
$T \rightarrow B\ C$	$T.t = C.t$ $C.b = B.t$
$B \rightarrow \textbf{int}$	$B.t = integer$
$B \rightarrow \textbf{float}$	$B.t = float$
$C \rightarrow [\ \textbf{num}\ ]\ C_1$	$C.t = array(\textbf{num}.val,\ C_1.t)$ $C_1.b = C.b$
$C \rightarrow \epsilon$	$C.t = C.b$

Type Expressions

- Nonterminal T generates either a **basic type** or an **array type**.
- Nonterminal B generates one of the basic types int and float.
- T generates a basic type when C derives €.
- Otherwise, C generates array components consisting of a sequence of integers, each integer surrounded by brackets.

Declaration statement Example SDD

PRODUCTION	SEMANTIC RULES
$T \rightarrow B\ C$	$T.t = C.t$ $C.b = B.t$
$B \rightarrow \textbf{int}$	$B.t = integer$
$B \rightarrow \textbf{float}$	$B.t = float$
$C \rightarrow [\ \textbf{num}\ ]\ C_1$	$C.t = array(\textbf{num}.val,\ C_1.t)$ $C_1.b = C.b$
$C \rightarrow \epsilon$	$C.t = C.b$

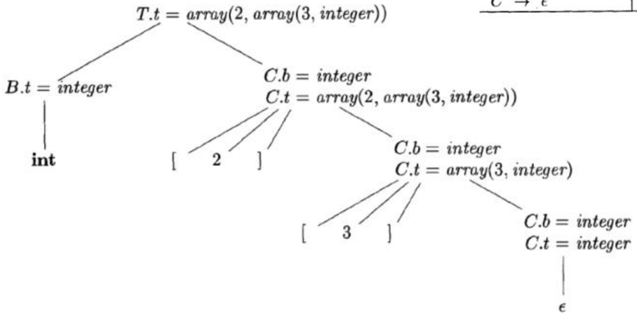
Type Expressions

- The nonterminals *B* and *T* have a synthesized attribute *t* representing a type.
- The nonterminal *C* has two attributes: an inherited attribute *b* and a synthesized attribute *t.*

Declaration statement Example SDD

input string int [2][3]

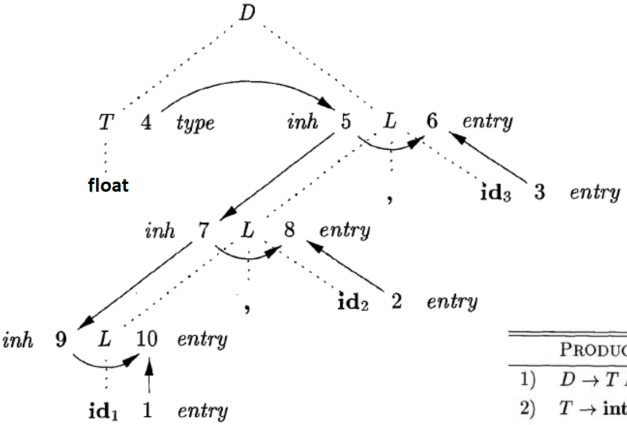
PRODUCTION	SEMANTIC RULES
$T \rightarrow B\ C$	$T.t = C.t$ $C.b = B.t$
$B \rightarrow$ **int**	$B.t = integer$
$B \rightarrow$ **float**	$B.t = float$
$C \rightarrow [$ **num** $]\ C_1$	$C.t = array(\textbf{num}.val,\ C_1.t)$ $C_1.b = C.b$
$C \rightarrow \epsilon$	$C.t = C.b$



- The nonterminal *C* has two attributes: an inherited attribute *b* and a synthesized attribute *t.*
- The inherited *b* attributes pass a basic type down the tree, and the synthesized *t* attributes accumulate the result.

Declaration statement: Example SDD

float $\mathbf{id}_1$, $\mathbf{id}_2$, $\mathbf{id}_3$



	PRODUCTION	SEMANTIC RULES
1)	$D \to T\ L$	$L.inh = T.type$
2)	$T \to$ **int**	$T.type =$ integer
3)	$T \to$ **float**	$T.type =$ float
4)	$L \to L_1$, **id**	$L_1.inh = L.inh$ $addType(\mathbf{id}.entry, L.inh)$
5)	$L \to$ **id**	$addType(\mathbf{id}.entry, L.inh)$

Symbol table

ST(global) *This is the Symbol Table for global symbols*

Name	Type	Initial Value	Size	Offset	Nested Table
d	float	2.3	8	0	null
i	int	null	4	8	null
w	*array*(10, int)	null	40	12	null
a	int	4	4	52	null
p	*ptr*(int)	null	4	56	null
b	int	null	4	60	null
func	*function*	null	0	64	ptr-to-ST(func)
c	char	null	1	64	null

Find the storage for each variable

More on Declaration statement

Data type + Storage layout

$$
\begin{array}{lcl}
D & \rightarrow & T\ \mathbf{id}\ ;\ D \mid \epsilon \\
T & \rightarrow & B\ C \\
B & \rightarrow & \mathbf{int} \mid \mathbf{float} \\
C & \rightarrow & \epsilon \mid [\ \mathbf{num}\ ]\ C
\end{array}
$$

- Simplified grammar that declares just **one name at a time**;
- We already explored the declarations with lists of names

Storage layout:

- **Relative address** of all the variables
- From the **type of a name**, we can determine the **amount of storage** that will be needed for the name at run time.
- At **compile time**, we can use these amounts to **assign each name a relative address**.
- The **type** and **relative address** are saved in the **symbol-table entry** for the name.

More on Declaration statement

Data type + Storage layout

- The **width** of a **type** is the number of **storage units** needed for objects of that type **(offset)**.
- A **basic type**, such as a character, integer, or float, requires an integral number of bytes.
- **Arrays** allocated in one **contiguous block of bytes**

$$
\begin{array}{lll}
T & \rightarrow & B \quad \{\ t = B.type;\ w = B.width;\ \} \\
  &   & C \\
B & \rightarrow & \textbf{int} \quad \{\ B.type = integer;\ B.width = 4;\ \} \\
B & \rightarrow & \textbf{float} \quad \{\ B.type = float;\ B.width = 8;\ \} \\
C & \rightarrow & \epsilon \quad \{\ C.type = t;\ C.width = w;\ \} \\
C & \rightarrow & [\ \textbf{num}\ ]\ C_1 \quad \{\ array(\textbf{num}.value,\ C_1.type); \\
  &   & \qquad C.width = \textbf{num}.value \times C_1.width;\ \}
\end{array}
$$

Computes **data types** and their **widths** for basic and array types

Type Expressions

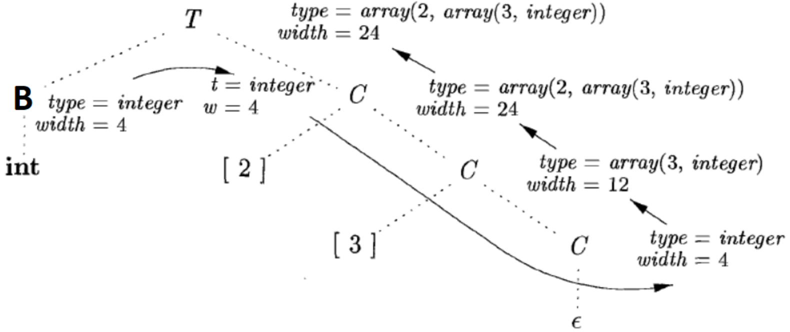
The **width of an array** is obtained by **multiplying** the **width of an element** by the **number of elements** in the array.

More on Declaration statement

Data type + Storage layout

`int[2][3]`

T	$\rightarrow$	B C	$\{ t = B.type;\ w = B.width; \}$
B	$\rightarrow$	**int**	$\{ B.type = integer;\ B.width = 4; \}$
B	$\rightarrow$	**float**	$\{ B.type = float;\ B.width = 8; \}$
C	$\rightarrow$	ϵ	$\{ C.type = t;\ C.width = w; \}$
C	$\rightarrow$	[**num**] C_1	$\{ array(\textbf{num}.value,\ C_1.type);$ $C.width = \textbf{num}.value \times C_1.width; \}$



Relative address

Name	Data type	Relative address
d	float	0
i	int	8
w	*array*(10, int)	12

Sequences of Declarations

```
int x;
float y;
```

- Relative address: **offset**
- Keeps track of the **next available relative address**

$$
\begin{array}{lll}
P & \rightarrow & \{ \ offset = 0; \ \} \quad D \\
D & \rightarrow & T \ \textbf{id} \ ; \quad \{ \ top.put(\textbf{id}.lexeme, \ T.type, \ offset); \\
 & & \qquad\qquad offset \ = \ offset + T.width; \ \} \\
 & & D_1 \\
D & \rightarrow & \epsilon
\end{array}
$$

- The translation scheme deals with a **sequence of declarations** of the form T id, where T generates a data type
- Before the first declaration is considered, **offset is set to 0**.
- As **each new name x** is seen, x is entered into the **symbol table** with its **relative address = current value of offset**,
  - which is **then incremented** by the width of the type of x.

Sequences of Declarations

- Relative address: offset
- Keeps track of the next available relative address

$$
\begin{array}{lll}
P & \rightarrow & \{\ \mathit{offset} = 0;\ \}\ \ D \\
D & \rightarrow & T\ \mathbf{id}\ ;\quad \{\ \mathit{top.put}(\mathbf{id}.\mathit{lexeme},\ T.\mathit{type},\ \mathit{offset}); \\
 & & \qquad\qquad\ \ \mathit{offset}\ =\ \mathit{offset} + T.\mathit{width};\ \} \\
 & & D_1 \\
D & \rightarrow & \epsilon
\end{array}
$$

The semantic action within the production $D \rightarrow T\ \mathbf{id}\ ;\ D_1$ creates a symbol-table entry by executing $\mathit{top.put}(\mathbf{id}.\mathit{lexeme},\ T.\mathit{type},\ \mathit{offset})$. Here *top* denotes the current symbol table. The method *top.put* creates a symbol-table entry for $\mathbf{id}.\mathit{lexeme}$, with type $T.\mathit{type}$ and relative address *offset* in its data area.

Major translation classes of Three address code generation

(a) Declaration statements (+ handling data type and storage)

(b) Expressions and statements

(a) Control flow statements

Translation of Expressions

statement a = b + - c

Three-address code for an assignment statement

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

Attribute **code** for S

attributes **addr and code** for an expression E.

Attributes **S.code and E.code denote the three-address code** for S and E, respectively.

Attribute **E.addr denotes the address** that will hold the **value of E.**

Translation of Expressions

Three-address code for an assignment statement

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

When an expression is a **single identifier**, say x, then **x itself holds the value of the expression**.

The semantic rules for this production define **E.addr to point to the symbol-table entry**

Translation of Expressions

Three-address code for an assignment statement

PRODUCTION	SEMANTIC RULES
$S \rightarrow \mathbf{id} = E\ ;$	$S.code = E.code \; \Vert$ $gen(top.get(\mathbf{id}.lexeme)\ '='\ E.addr)$
$E \rightarrow E_1 + E_2$	$E.addr = \mathbf{new}\ Temp\,()$ $E.code = E_1.code \; \Vert \; E_2.code \; \Vert$ $gen(E.addr\ '='\ E_1.addr\ '+'\ E_2.addr)$

The semantic rules for $E \rightarrow E_1 + E_2$, generate code to compute the value of E from the values of E_1 and E_2. Values are computed into newly generated temporary names. If E_1 is computed into $E_1.addr$ and E_2 into $E_2.addr$, then $E_1 + E_2$ translates into $t = E_1.addr + E_2.addr$, where t is a new temporary name. $E.addr$ is set to t. A sequence of distinct temporary names $t_1, t_2, \ldots$ is created by successively executing $\mathbf{new}\ Temp()$.

Translation of Expressions

Three-address code for an assignment statement

PRODUCTION	SEMANTIC RULES
$S \rightarrow \mathbf{id} = E\ ;$	$S.code = E.code \parallel gen(top.get(\mathbf{id}.lexeme)\ '='\ E.addr)$
$E \rightarrow E_1 + E_2$	$E.addr = \mathbf{new}\ Temp()$ $E.code = E_1.code \parallel E_2.code \parallel gen(E.addr\ '='\ E_1.addr\ '+'\ E_2.addr)$

Finally, the production $S \rightarrow \mathbf{id} = E;$ generates instructions that assign the value of expression E to the identifier **id**. The semantic rule for this production uses function *top.get* to determine the address of the identifier represented by **id**, as in the rules for $E \rightarrow \mathbf{id}$. $S.code$ consists of the instructions to compute the value of E into an address given by $E.addr$, followed by an assignment to the address $top.get(\mathbf{id}.lexeme)$ for this instance of **id**.

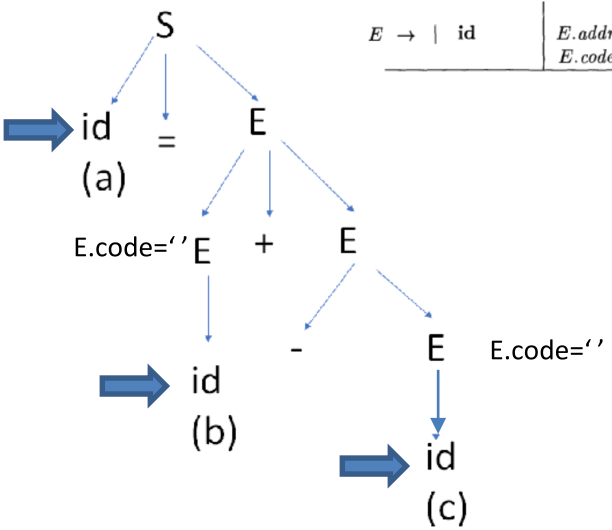
Translation of Expressions

Three-address code for an assignment statement

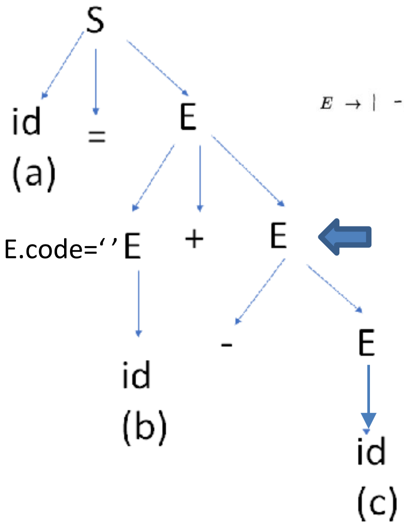
PRODUCTION	SEMANTIC RULES
$S \rightarrow \mathbf{id} = E\ ;$	$S.code = E.code \ \|\|$ $gen(top.get(\mathbf{id}.lexeme)\ '='\ E.addr)$
$E \rightarrow E_1 + E_2$	$E.addr = \mathbf{new}\ Temp\,()$ $E.code = E_1.code \ \|\|\ E_2.code \ \|\|$ $gen(E.addr\ '='\ E_1.addr\ '+'\ E_2.addr)$
$\quad \mid\ - E_1$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

statement $a = b + - c$

$E \rightarrow$	$\mid$ **id**	$E.addr = top.get(\mathbf{id}.lexeme)$ $E.code = ''$



statement a = b + − c

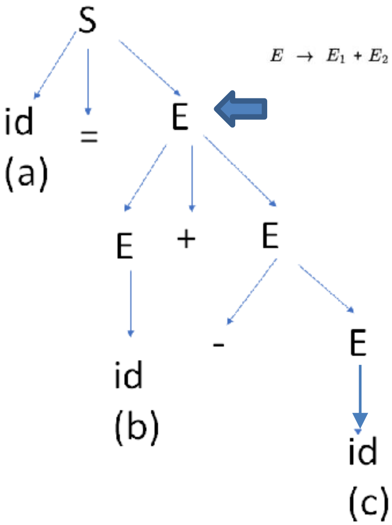


$$E \to - E_1 \quad \begin{array}{|l} E.addr = \textbf{new } Temp\,() \\ E.code = E_1.code \;\|\; \\ \qquad gen(E.addr\ '='\ '\textbf{minus}'\ E_1.addr) \end{array}$$

E.code=

t_1 = minus c

statement a = b + − c

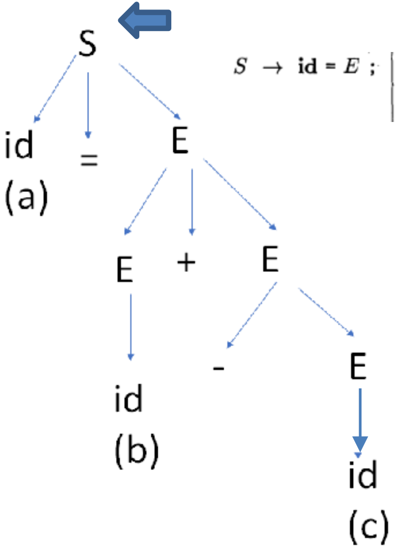


$E \rightarrow E_1 + E_2$	$E.addr = \textbf{new}\ Temp()$ $E.code = E_1.code \parallel E_2.code \parallel$ $gen(E.addr\ '='\ E_1.addr\ '+'\ E_2.addr)$

E.code=

```
t1 = minus c
t2 = b + t1
```

statement $a = b + - c$



$$S \rightarrow \mathbf{id} = E\ ; \quad \Big| \quad S.code = E.code \;\|\; gen(top.get(\mathbf{id}.lexeme)\ '\!=\!'\ E.addr)$$

S.code=

t_1 = minus c

t_2 = b + t_1

a = t_2

statement a = b + − c

Translation of Expressions

Three-address code for an assignment statement

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

statement a = b + − c

```
t1 = minus c
t2 = b + t1
a = t2
```

Incremental Translation

- So far, ***E.Code*** attributes were long strings
    - Generated incrementally
- In incremental translation, generate only the **new three-address instructions**
- **Past sequence** may either be **retained in memory** for further processing, or it may be **output incrementally**.
- In the incremental approach, gen() not only constructs a three-address instruction,
    - it **appends** the instruction to the sequence of instructions generated so far.

Incremental Translation

$$
\begin{array}{llll}
S & \rightarrow & \textbf{id} = E \; ; & \{ \; gen(\; top.get(\textbf{id}.lexeme) \; '=' \; E.addr); \; \} \\
E & \rightarrow & E_1 + E_2 & \{ \; E.addr = \textbf{new} \; Temp(); \\
 & & & \quad gen(E.addr \; '=' \; E_1.addr \; '+' \; E_2.addr); \; \} \\
 & | & - E_1 & \{ \; E.addr = \textbf{new} \; Temp(); \\
 & & & \quad gen(E.addr \; '=' \; '\textbf{minus}' \; E_1.addr); \; \} \\
 & | & (\; E_1 \;) & \{ \; E.addr = E_1.addr; \; \} \\
 & | & \textbf{id} & \{ \; E.addr = top.get(\textbf{id}.lexeme); \; \}
\end{array}
$$

- This **translation scheme generates the same code** as the previous syntax directed definition.
- With the incremental approach, the **E.code attribute is not used,**
    - Since there is a **single sequence of instructions** that is created by **successive calls to gen()**.