



NEW YORK UNIVERSITY

G22.2130-001

Compiler Construction

Lecture 9:

Intermediate-Code Generation

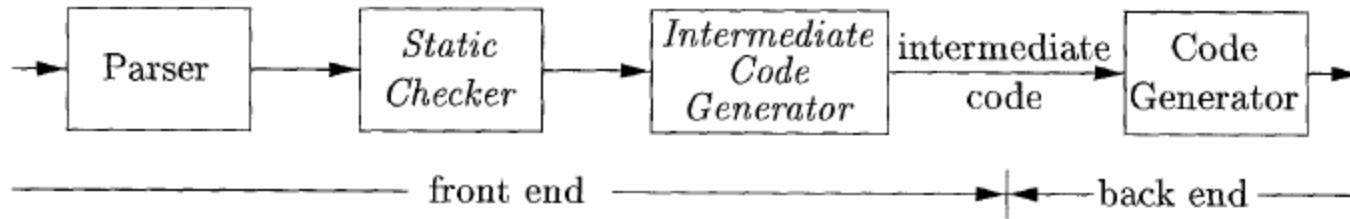
Mohamed Zahran (aka Z)

mzahran@cs.nyu.edu

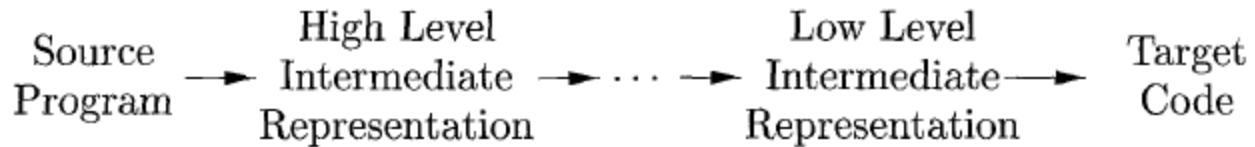
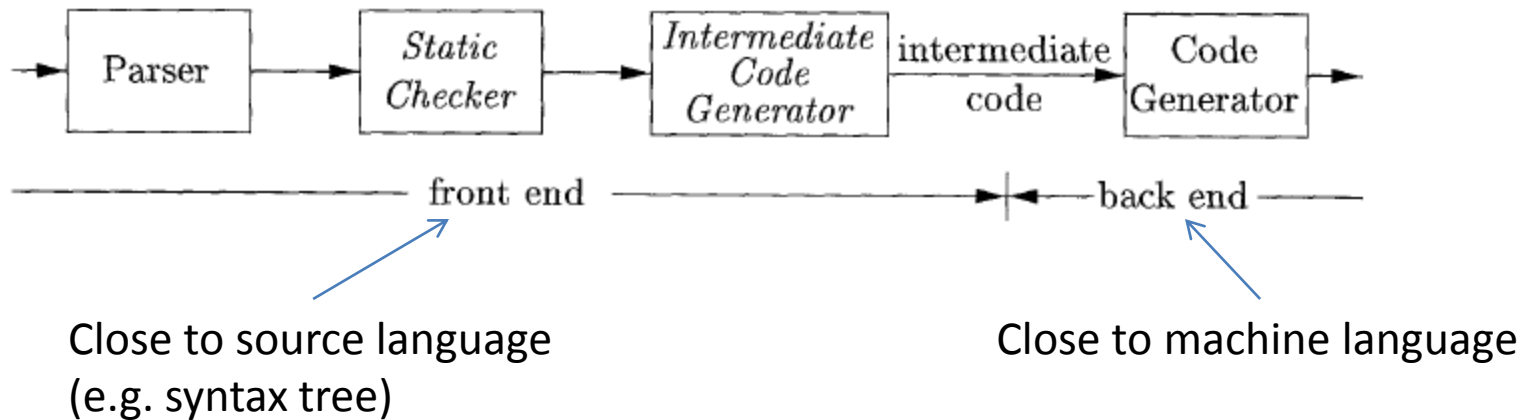


Copyright © Randy Glasbergen. www.glasbergen.com

Back-end and Front-end of A Compiler

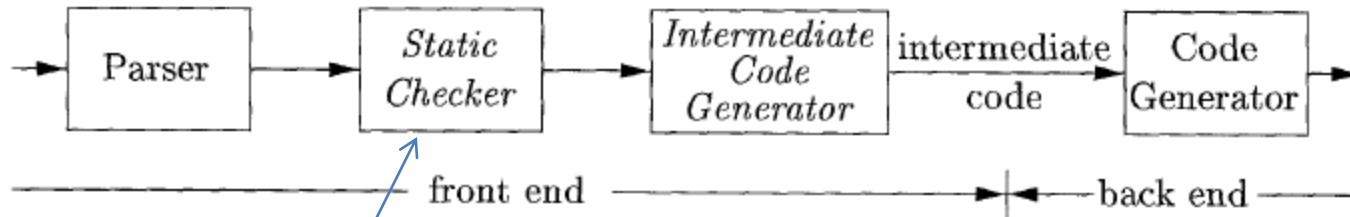


Back-end and Front-end of A Compiler



m x n compilers can be built by writing just m front ends and n back ends

Back-end and Front-end of A Compiler



Includes:

- Type checking
- Any syntactic checks that remain after parsing (e.g. ensure *break* statement is enclosed within while-, for-, or switch statements).

Syntax Tree

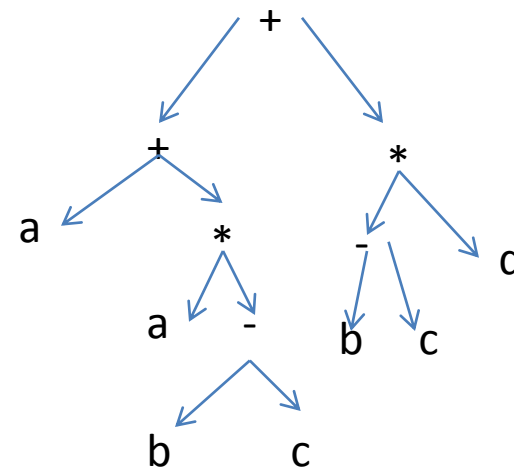
PRODUCTION	SEMANTIC RULES
1) $E \rightarrow E_1 + T$	$E.node = \text{new Node}('+', E_1.node, T.node)$
2) $E \rightarrow E_1 - T$	$E.node = \text{new Node}('-', E_1.node, T.node)$
3) $E \rightarrow T$	$E.node = T.node$
4) $T \rightarrow (E)$	$T.node = E.node$
5) $T \rightarrow \text{id}$	$T.node = \text{new Leaf}(\text{id}, \text{id.entry})$
6) $T \rightarrow \text{num}$	$T.node = \text{new Leaf}(\text{num}, \text{num.val})$

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

Syntax Tree

PRODUCTION	SEMANTIC RULES
1) $E \rightarrow E_1 + T$	$E.node = \text{new Node}('+', E_1.node, T.node)$
2) $E \rightarrow E_1 - T$	$E.node = \text{new Node}('-', E_1.node, T.node)$
3) $E \rightarrow T$	$E.node = T.node$
4) $T \rightarrow (E)$	$T.node = E.node$
5) $T \rightarrow \text{id}$	$T.node = \text{new Leaf}(\text{id}, \text{id.entry})$
6) $T \rightarrow \text{num}$	$T.node = \text{new Leaf}(\text{num}, \text{num.val})$

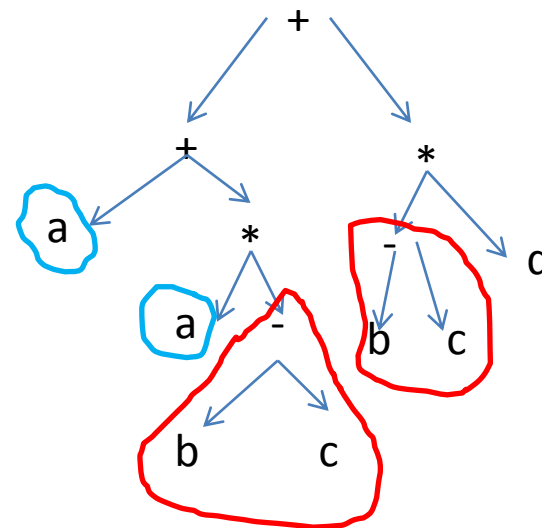
a + a * (b - c) + (b - c) * d



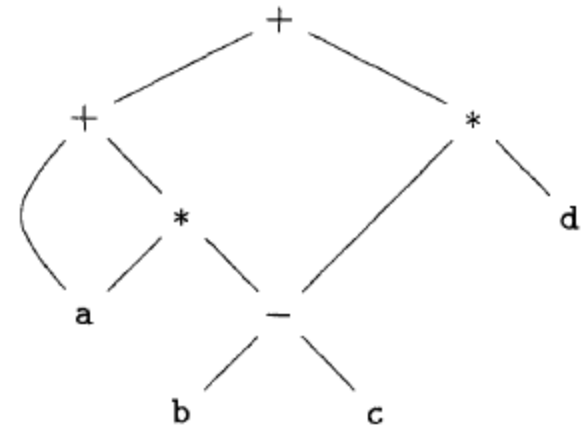
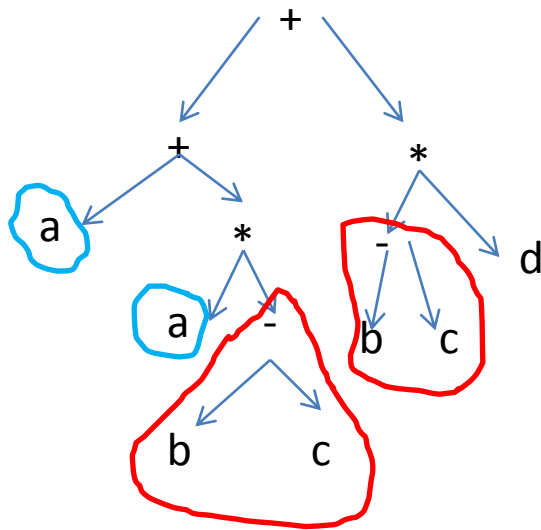
Syntax Tree

PRODUCTION	SEMANTIC RULES
1) $E \rightarrow E_1 + T$	$E.node = \text{new Node}('+', E_1.node, T.node)$
2) $E \rightarrow E_1 - T$	$E.node = \text{new Node}('-', E_1.node, T.node)$
3) $E \rightarrow T$	$E.node = T.node$
4) $T \rightarrow (E)$	$T.node = E.node$
5) $T \rightarrow \text{id}$	$T.node = \text{new Leaf}(\text{id}, \text{id.entry})$
6) $T \rightarrow \text{num}$	$T.node = \text{new Leaf}(\text{num}, \text{num.val})$

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



Syntax Tree



Node can have more than one parent

Directed Acyclic Graph (DAG):

- More compact representation
- Gives clues regarding generation of efficient code

Example

Construct the DAG for:

$$((x + y) - ((x + y) * (x - y))) + ((x + y) * (x - y))$$

How to Generate DAG from Syntax-Directed Definition?

PRODUCTION	SEMANTIC RULES
1) $E \rightarrow E_1 + T$	$E.node = \text{new Node}('+', E_1.node, T.node)$
2) $E \rightarrow E_1 - T$	$E.node = \text{new Node}('-', E_1.node, T.node)$
3) $E \rightarrow T$	$E.node = T.node$
4) $T \rightarrow (E)$	$T.node = E.node$
5) $T \rightarrow \text{id}$	$T.node = \text{new Leaf}(\text{id}, \text{id.entry})$
6) $T \rightarrow \text{num}$	$T.node = \text{new Leaf}(\text{num}, \text{num.val})$

All what is needed is that functions such as **Node** and **Leaf** above check whether a node already exists. If such a node exists, a pointer is returned to that node.

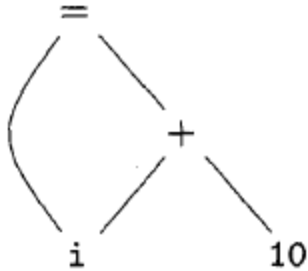
How to Generate DAG from Syntax-Directed Definition?

PRODUCTION	SEMANTIC RULES
1) $E \rightarrow E_1 + T$	$E.node = \mathbf{new Node}('+', E_1.node, T.node)$
2) $E \rightarrow E_1 - T$	$E.node = \mathbf{new Node}('-', E_1.node, T.node)$
3) $E \rightarrow T$	$E.node = T.node$
4) $T \rightarrow (E)$	$T.node = E.node$
5) $T \rightarrow \mathbf{id}$	$T.node = \mathbf{new Leaf}(\mathbf{id}, \mathbf{id.entry})$
6) $T \rightarrow \mathbf{num}$	$T.node = \mathbf{new Leaf}(\mathbf{num}, \mathbf{num.val})$

- 1) $p_1 = \mathit{Leaf}(\mathbf{id}, \mathit{entry-a})$
- 2) $p_2 = \mathit{Leaf}(\mathbf{id}, \mathit{entry-a}) = p_1$
- 3) $p_3 = \mathit{Leaf}(\mathbf{id}, \mathit{entry-b})$
- 4) $p_4 = \mathit{Leaf}(\mathbf{id}, \mathit{entry-c})$
- 5) $p_5 = \mathit{Node}('-', p_3, p_4)$
- 6) $p_6 = \mathit{Node}('*', p_1, p_5)$
- 7) $p_7 = \mathit{Node}('+', p_1, p_6)$
- 8) $p_8 = \mathit{Leaf}(\mathbf{id}, \mathit{entry-b}) = p_3$
- 9) $p_9 = \mathit{Leaf}(\mathbf{id}, \mathit{entry-c}) = p_4$
- 10) $p_{10} = \mathit{Node}('-', p_3, p_4) = p_5$
- 11) $p_{11} = \mathit{Leaf}(\mathbf{id}, \mathit{entry-d})$
- 12) $p_{12} = \mathit{Node}('*', p_5, p_{11})$
- 13) $p_{13} = \mathit{Node}('+', p_7, p_{12})$

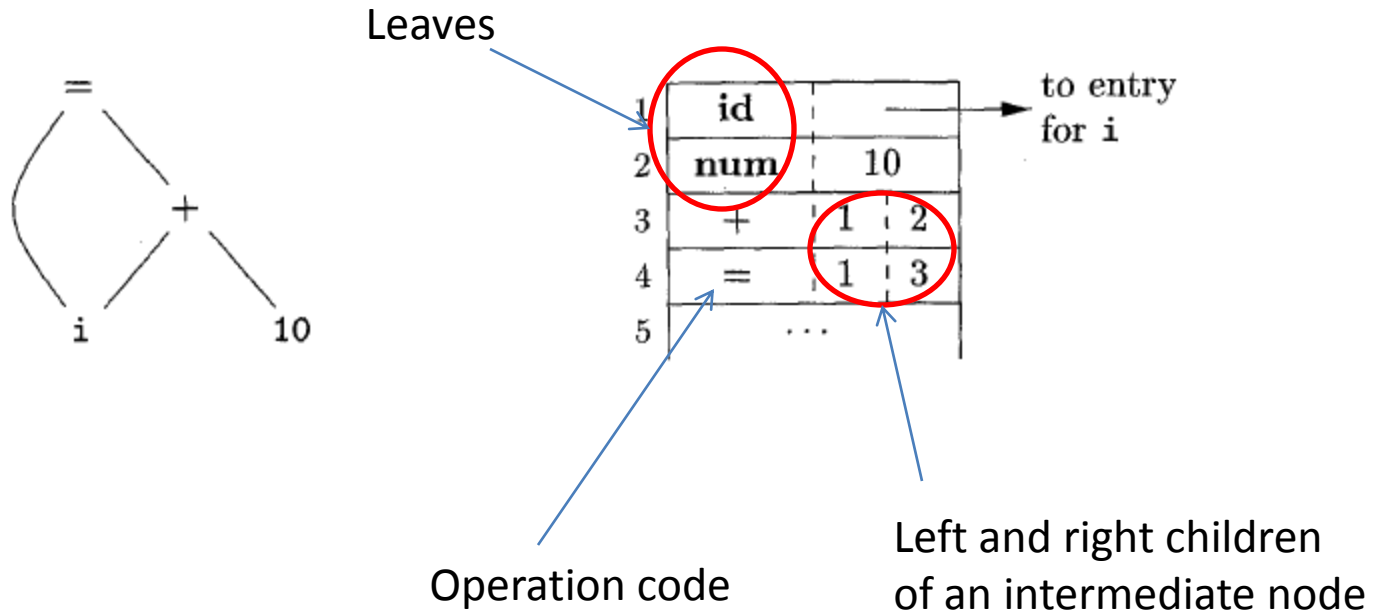
a + a * (b - c) + (b - c) * d

Data Structure: Array



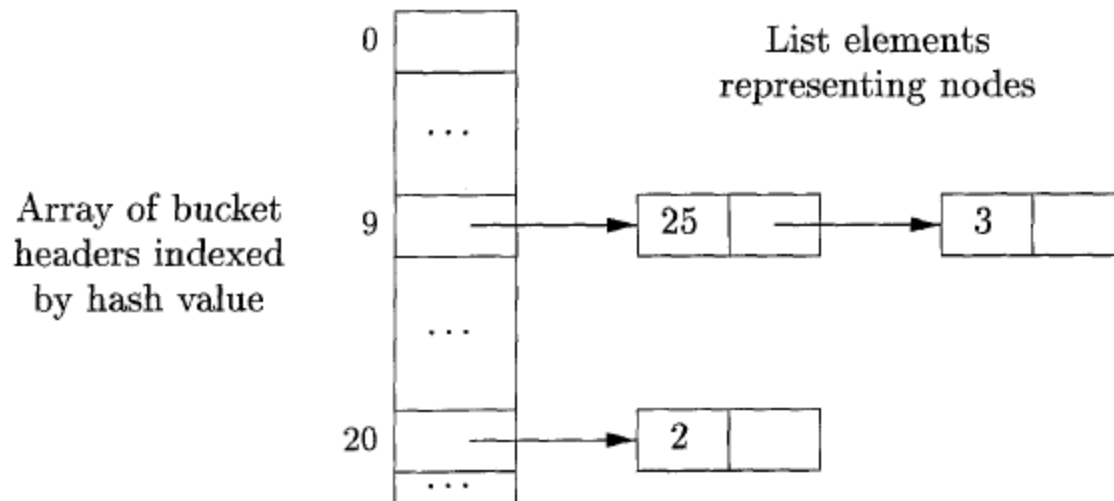
1	id			to entry for i
2	num	10		
3	+	1	2	
4	=	1	3	
5		...		

Data Structure: Array



Scanning the array each time a new node is needed, is not an efficient thing to do.

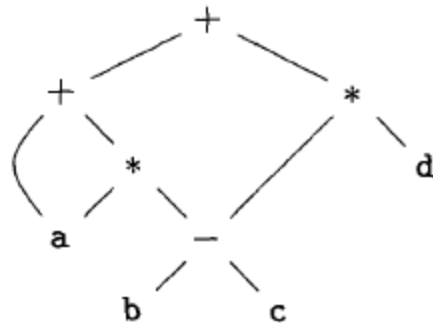
Data Structure: Hash Table



Hash function = $h(\text{op}, L, R)$

Three-Address Code

- Another option for intermediate presentation
- Built from two concepts:
 - addresses and instructions
- At most one operator



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

Address

Can be one of the following:

- A name: source program name
- A constant
- Compiler-generated temporary

Instructions

Assignment instructions of the form $x = y \text{ op } z$

Assignments of the form $x = \text{op } y$

Copy instructions of the form $x = y$

An unconditional jump `goto L`

Conditional jumps of the form `if x goto L` and `ifFalse x goto L`

Conditional jumps such as `if x relop y goto L`

Procedure call such as `p(x1, x2, ..., xn)` is implemented as:

```
param x1
param x2
...
param xn
call p, n
```

Indexed copy instructions of the form $x = y[i]$ and $x[i] = y$.

Address and pointer assignments of the form $x = \&y$, $x = *y$, and $*x = y$

Example

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



```
L:  t1 = i + 1  
    i = t1  
    t2 = i * 8  
    t3 = a [ t2 ]  
    if t3 < v goto L
```

OR

```
100: t1 = i + 1  
101: i = t1  
102: t2 = i * 8  
103: t3 = a [ t2 ]  
104: if t3 < v goto 100
```

Choice of Operator Set

- Rich enough to implement the operations of the source language
- Close enough to machine instructions to simplify code generation

Data Structure

How to present these instructions in a data structure?

- Quadruples
- Triples
- Indirect triples

Data Structure: Quadruples

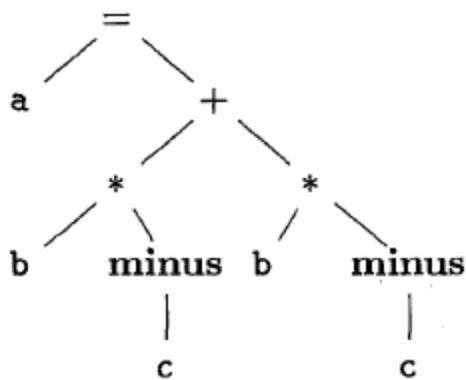
- Has four fields: op, arg1, arg2, result
- Exceptions:
 - Unary operators: no arg2
 - Operators like *param*: no arg2, no result
 - (Un)conditional jumps: target label is the result

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

	<i>op</i>	<i>arg₁</i>	<i>arg₂</i>	<i>result</i>
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
		...		

Data Structure: Triples

- Only three fields: no *result* field
- Results referred to by its position



(a) Syntax tree

	<i>op</i>	<i>arg₁</i>	<i>arg₂</i>
0	minus	c	
1	*	b	(0)
2	minus	c	
3	*	b	(2)
4	+	(1)	(3)
5	=	a	(4)
		...	

(b) Triples

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

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

Data Structure: Indirect Triples

- When instructions are moving around during optimizations: quadruples are better than triples.
- Indirect triples solve this problem

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

	<i>op</i>	<i>arg₁</i>	<i>arg₂</i>
0	minus	c	
1	*	b	(0)
2	minus	c	
3	*	b	(2)
4	+	(1)	(3)
5	=	a	(4)
		...	

List of pointers to triples



Optimizing compiler can reorder instruction list, instead of affecting the triples themselves

Single-Static-Assignment (SSA)

- Is an intermediate presentation
- Facilitates certain code optimizations
- All assignments are to variables with distinct names

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

```
p1 = a + b
q1 = p1 - c
p2 = q1 * d
p3 = e - p2
q2 = p3 + q1
```

(a) Three-address code. (b) Static single-assignment form.

Single-Static-Assignment (SSA)


Example:

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

If we use different names for X in true part and false part, then which name shall we use in the assignment of $y = x * a$?

The answer is: \emptyset -function

```
if ( flag ) x1 = -1; else x2 = 1;  
x3 =  $\phi(x_1, x_2)$ ;
```



Returns the value of its argument that corresponds to the control-flow path that was taken to get to the assignment statement containing the \emptyset -function

Example

Translate the arithmetic expression $a + -(b + c)$ into:

- a) A syntax tree.
- b) Quadruples.
- c) Triples.
- d) Indirect triples.

Types and Declarations

- Type checking: to ensure that types of operands match the type expected by operator
- Determine the storage needed
- Calculate the address of an array reference
- Insert explicit type conversion
- Choose the write version of an operator
- ...

Storage Layout

- From the type, we can determine amount of storage at run time.
- At compile time, we will use this amount to assign its name a relative address.
- Type and relative address are saved in the symbol table entry of the name.
- Data with length determined only at run time saves a pointer in the symbol table.

Storage Layout

- Multibyte objects are stored in consecutive bytes and given the address of the first byte
- Storage for aggregates (e.g. arrays and classes) is allocated in one contiguous block of bytes.

$$\begin{aligned} D &\rightarrow T \text{ id} ; D \mid \epsilon \\ T &\rightarrow B C \mid \text{record } \{ D \} \\ B &\rightarrow \text{int} \mid \text{float} \\ C &\rightarrow \epsilon \mid [\text{num}] C \end{aligned}$$

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

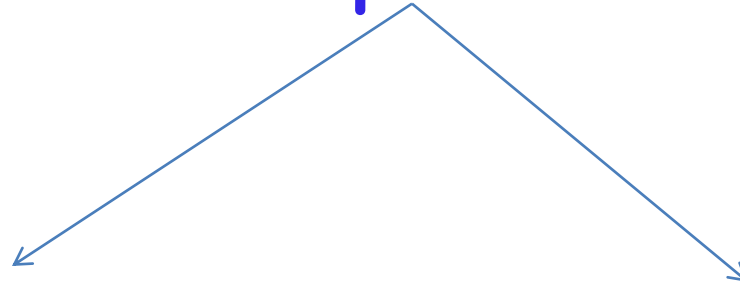
To keep track of the next available relative address

$P \rightarrow D \quad \{ \text{offset} = 0; \}$
 $D \rightarrow T \text{ id } ; \quad \{ \text{top.put}(\text{id.lexeme}, T.\text{type}, \text{offset});$
 $\quad \quad \quad \text{offset} = \text{offset} + T.\text{width}; \}$
 $D \rightarrow \epsilon$

The diagram shows three grammar rules. The first rule is $P \rightarrow D$ with a semantic action $\{ \text{offset} = 0; \}$. The second rule is $D \rightarrow T \text{ id } ;$ with a semantic action $\{ \text{top.put}(\text{id.lexeme}, T.\text{type}, \text{offset}); \text{offset} = \text{offset} + T.\text{width}; \}$. The third rule is $D \rightarrow \epsilon$. A blue arrow points from the text 'To keep track of the next available relative address' to the offset variable in the first rule. Another blue arrow points from the text 'Create a symbol table entry' to the top.put function call in the second rule.

Create a symbol table entry

Translations of Statements and Expressions



Syntax-Directed Definition
(SDD)

Syntax-Directed Translation
(SDT)

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

Address holding value of E
(e.g. tmp variable, name, constant)

Three-address code of E

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

Build an instruction

Get a temporary variable

Current symbol table

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

$a = b + - c$



$t_1 = \text{minus } c$
 $t_2 = b + t_1$
 $a = t_2$

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

Generating three-address code **incrementally** to avoid long strings manipulations



gen() does two things:

- generate three address instruction
- append it to the sequence of instructions generated so far

$S \rightarrow \text{id} = E ; \quad \{ gen(top.get(\text{id.lexeme}) \neq E.addr); \}$

$E \rightarrow E_1 + E_2 \quad \{ E.addr = \text{new Temp}();$
 $gen(E.addr \neq E_1.addr \neq E_2.addr); \}$

$| - E_1 \quad \{ E.addr = \text{new Temp}();$
 $gen(E.addr \neq \text{'minus'} E_1.addr); \}$

$| (E_1) \quad \{ E.addr = E_1.addr; \}$

$| \text{id} \quad \{ E.addr = top.get(\text{id.lexeme}); \}$

Arrays

- Elements of the same time
- Stored consecutively in memory
- In languages like C or Java elements are: $0, 1, \dots, n-1$
- In some other languages:
 $low, low+1, \dots, high$

Arrays

If elements start with 0, and element width is w , then $a[i]$ address is: $base + i \times w$
base is address of $A[0]$

Generalizing to two-dimensions $a[i_1][i_2]$:

w_1 is width of a row and

w_2 the width of an element or

$$base + i_1 \times w_1 + i_2 \times w_2$$

w is width of an element, n_2 is number of elements

$$base + (i_1 \times n_2 + i_2) \times w$$

Generalizing to k -dimensions: $base + i_1 \times w_1 + i_2 \times w_2 + \dots + i_k \times w_k$

or

$$base + ((\dots (i_1 \times n_2 + i_2) \times n_3 + i_3) \dots) \times n_k + i_k) \times w$$

```

S → id = E ; { gen( top.get(id.lexeme) != E.addr); }

| L = E ; { gen(L.addr.base '[' L.addr ']' != E.addr); }

E → E1 + E2 { E.addr = new Temp();
                 gen(E.addr != E1.addr '+' E2.addr); }

| id { E.addr = top.get(id.lexeme); }

| L { E.addr = new Temp();
      gen(E.addr != L.array.base '[' L.addr ']); }

L → id [ E ] { L.array = top.get(id.lexeme);
                L.type = L.array.type.elem;
                L.addr = new Temp(); ← temporary used
                gen(L.addr != E.addr '*' L.type.width); } while computing
                                                                    the offset

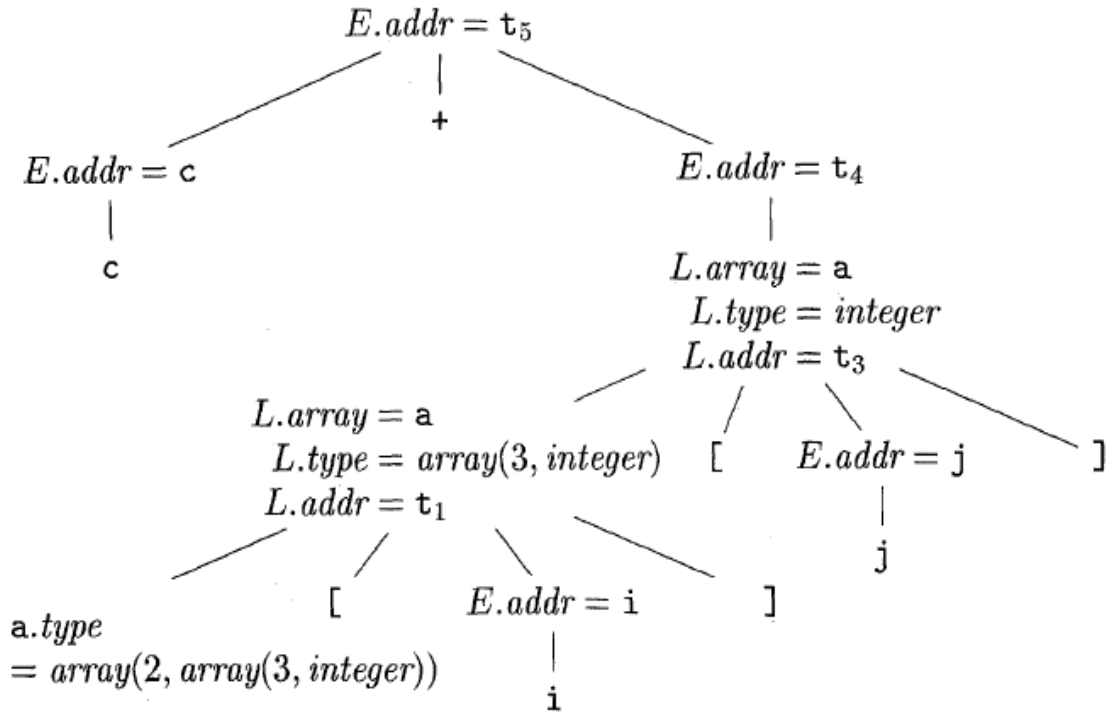
| L1 [ E ] { L.array = L1.array; ← pointer to symbol table entry
               L.type = L1.type.elem;
               t = new Temp();
               L.addr = new Temp();
               gen(t != E.addr '*' L.type.width); }
               gen(L.addr != L1.addr '+' t); }

```

```

S → id = E ; { gen( top.get(id.lexeme) != E.addr); }
  | L = E ; { gen(L.addr.base '[' L.addr ']' != E.adr
E → E1 + E2 { E.addr = new Temp();
                gen(E.addr != E1.addr '+' E2.addr); }
  | id      { E.addr = top.get(id.lexeme); }
  | L      { E.addr = new Temp();
            gen(E.addr != L.array.base '[' L.addr
L → id [ E ] { L.array = top.get(id.lexeme);
              L.type = L.array.type.elem;
              L.addr = new Temp();
              gen(L.addr != E.addr '*' L.type.width
  | L1 [ E ] { L.array = L1.array;
               L.type = L1.type.elem;
               t = new Temp();
               L.addr = new Temp();
               gen(t != E.addr '*' L.type.width); }
               gen(L.addr != L1.addr '+' t); }

```



Annotated parse tree for $c + a[i][j]$

A is a 2x3 array of integers

```

t1 = i * 12
t2 = j * 4
t3 = t1 + t2
t4 = a [ t3 ]
t5 = c + t4

```

So...

- Skim: 6.3.1, 6.3.2
- Read: Beginning of chapter 6 -> 6.4