NEW YORK UNIVERSITY

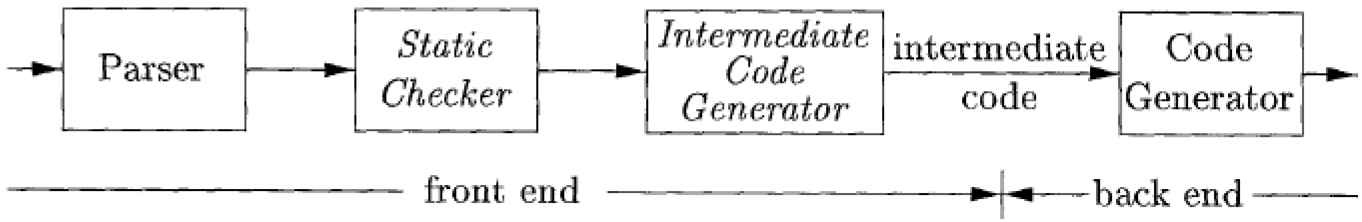
CSCI-GA.2130-001

Compiler Construction

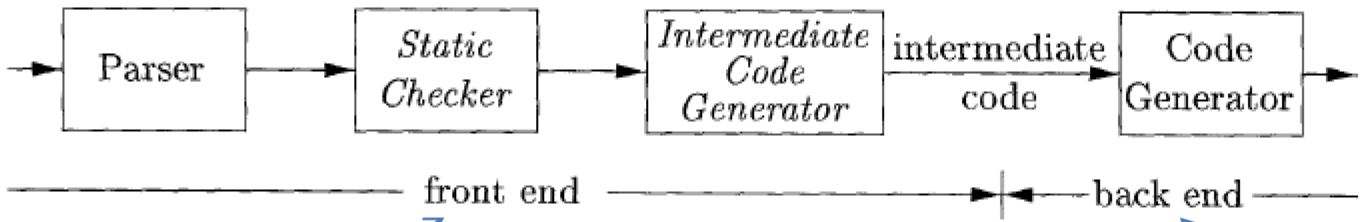
Lecture 9: Intermediate-Code Generation I

Mohamed Zahran (aka Z)
mzahran@cs.nyu.edu

Back-end and Front-end of A Compiler

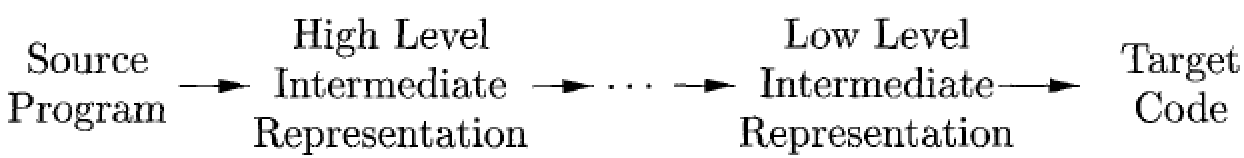


Back-end and Front-end of A Compiler



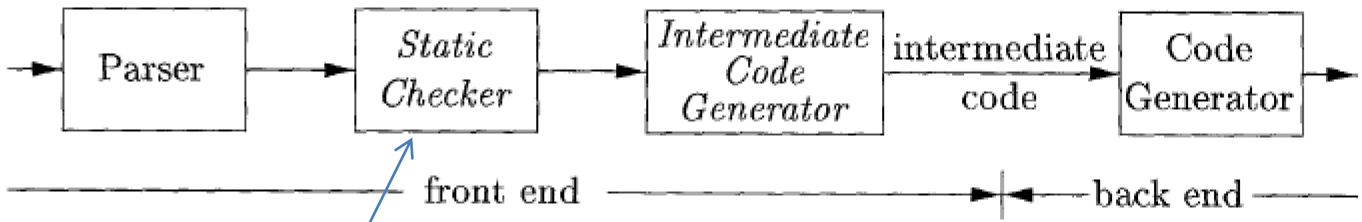
Close to source language (e.g. syntax tree)

Close to machine language



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

T -> E1 * T **T.node = new Node('*'. E1.node, T.node)**

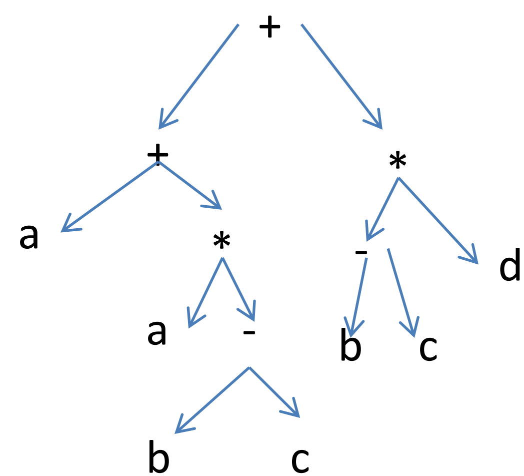
`a + a * (b - c) + (b - c) * d`

Syntax Tree

	Production	Semantic Rules
1)	$E \to E_1 + T$	$E.node = \textbf{new } Node('+', E_1.node, T.node)$
2)	$E \to E_1 - T$	$E.node = \textbf{new } Node('-', E_1.node, T.node)$
3)	$E \to T$	$E.node = T.node$
4)	$T \to (E)$	$T.node = E.node$
5)	$T \to \textbf{id}$	$T.node = \textbf{new } Leaf(\textbf{id}, \textbf{id}.entry)$
6)	$T \to \textbf{num}$	$T.node = \textbf{new } Leaf(\textbf{num}, \textbf{num}.val)$

T -> E1 * T **T.node = new Node('*'. E1.node, T.node)**

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

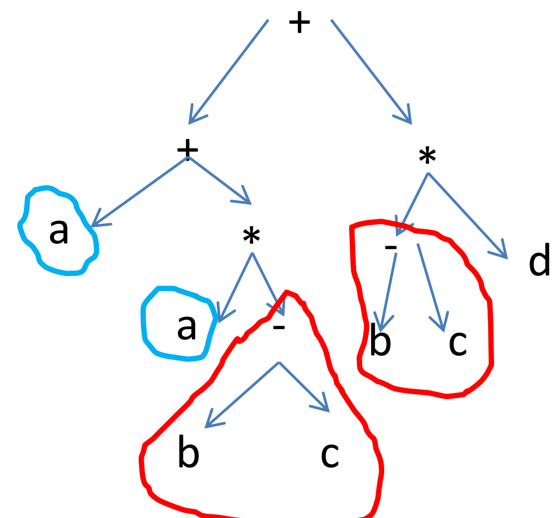


Syntax Tree

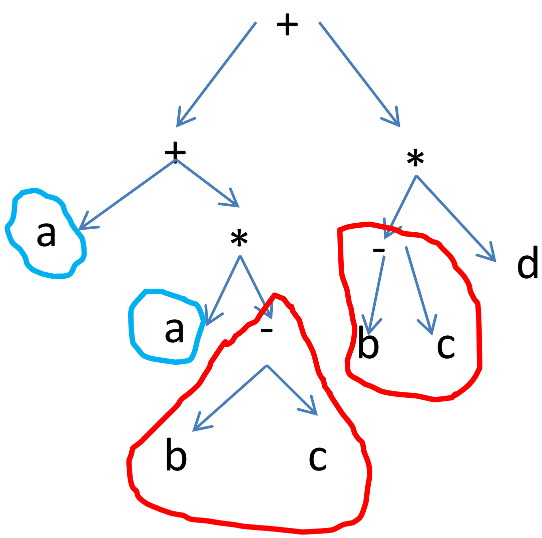
	PRODUCTION	SEMANTIC RULES
1)	$E \to E_1 + T$	$E.node = \textbf{new}\ Node('+', E_1.node, T.node)$
2)	$E \to E_1 - T$	$E.node = \textbf{new}\ Node('-', E_1.node, T.node)$
3)	$E \to T$	$E.node = T.node$
4)	$T \to (\ E\ )$	$T.node = E.node$
5)	$T \to \textbf{id}$	$T.node = \textbf{new}\ Leaf(\textbf{id},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

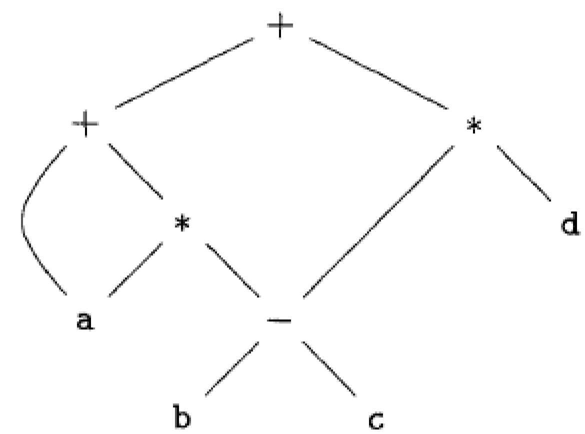
T -> E1 * T   T.node = new Node('*'. E1.node, T.node)

```
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)	$E \rightarrow E_1 + T$	$E.node = \mathbf{new}\ Node('+', E_1.node, T.node)$
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)	$T \rightarrow \mathbf{num}$	$T.node = \mathbf{new}\ Leaf(\mathbf{num}, \mathbf{num}.val)$
	T -> E1 * T	**T.node = new Node('*'. E1.node, T.node)**

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.

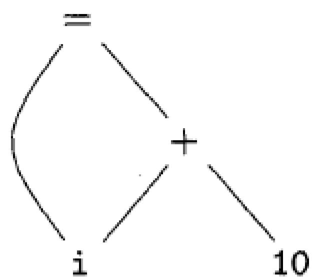
How to Generate DAG from Syntax-Directed Definition?

	Production	Semantic Rules
1)	$E \rightarrow E_1 + T$	$E.node = \textbf{new}\ Node('+', E_1.node, T.node)$
2)	$E \rightarrow E_1 - T$	$E.node = \textbf{new}\ Node('-', E_1.node, T.node)$
3)	$E \rightarrow T$	$E.node = T.node$
4)	$T \rightarrow (\ E\ )$	$T.node = E.node$
5)	$T \rightarrow \textbf{id}$	$T.node = \textbf{new}\ Leaf(\textbf{id}, \textbf{id}.entry)$
6)	$T \rightarrow \textbf{num}$	$T.node = \textbf{new}\ Leaf(\textbf{num}, \textbf{num}.val)$
	T -> E1 * T	**T.node = new Node('*'. E1.node, T.node)**

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

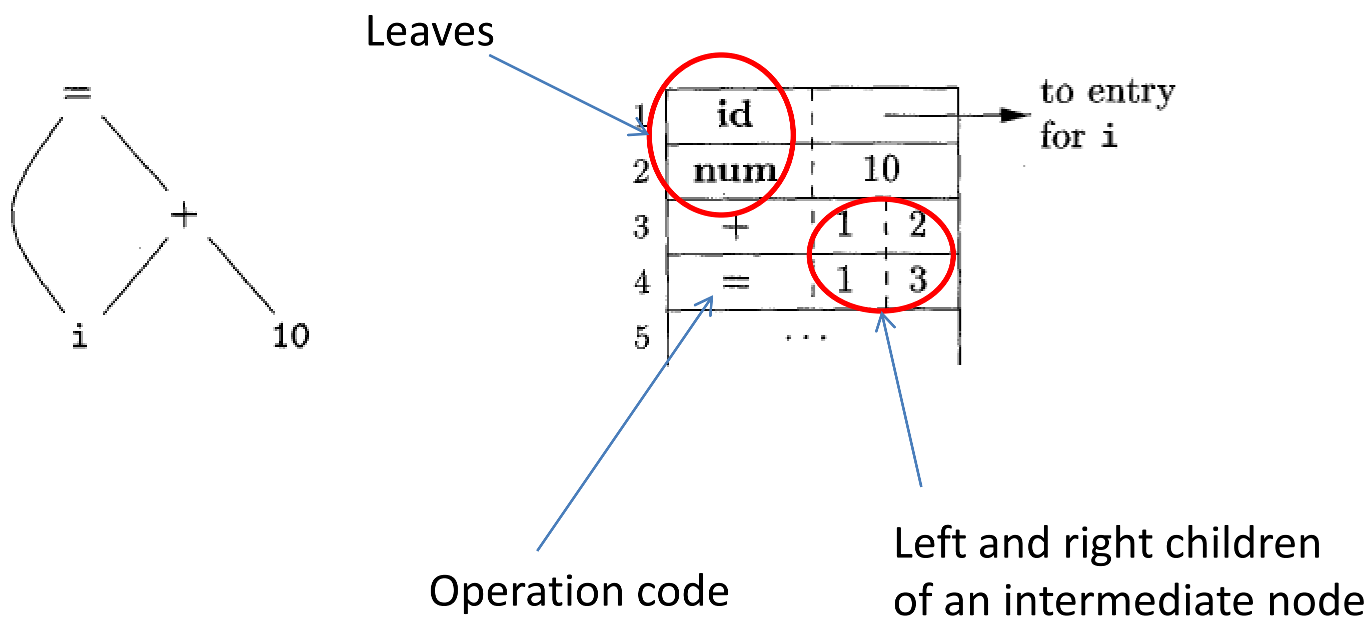
```
a + a * (b - c) + (b - c) * d
```

Data Structure: Array



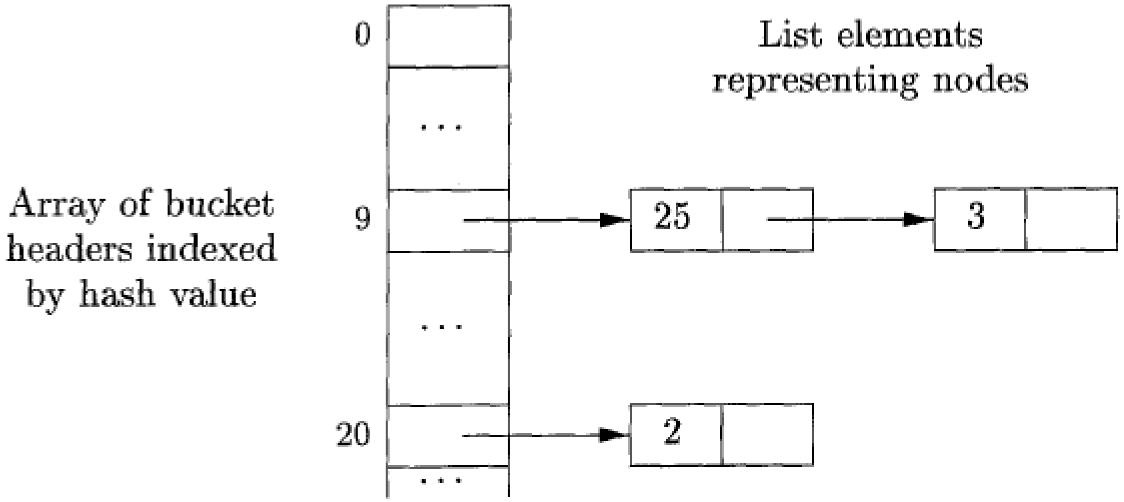
1	**id**	to entry for i	
2	**num**	10	
3	+	1	2
4	=	1	3
5	. . .		

Data Structure: Array



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

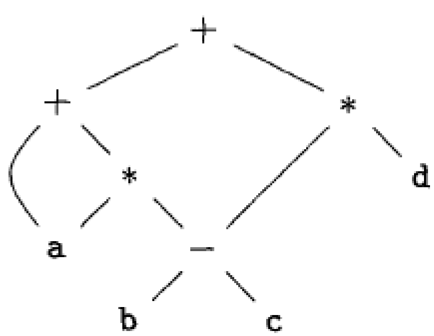
Data Structure: Hash Table



Hash function = h(op, L, R)

Three-Address Code

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



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

Address

Can be one of the following:

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

Instructions

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

Assignments of the form $x = op \ y$

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 x_1
param x_2
...
param x_n
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

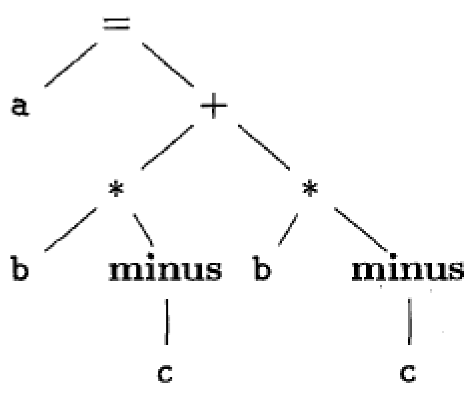
- Has four fields: op, arg1, arg2, result
- Exceptions:
  - Unary operators: no arg2
  - Operators like *param*: no arg2, no result
  - (Un)conditional jumps: target label is the result

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

	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
	...			

Data Structure: Triples

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



(a) Syntax tree

	op	*arg_1*	*arg_2*	
0	minus	c		t_1 = minus c
1	*	b	(0)	t_2 = b * t_1
2	minus	c		t_3 = minus c
3	*	b	(2)	t_4 = b * t_3
4	+	(1)	(3)	t_5 = t_2 + t_4
5	=	a	(4)	a = t_5
		. . .		

(b) Triples

Representation of **a = b * -c + b * -c**

Data Structure: Indirect Triples

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

	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)
		...	

List of pointers to triples

Optimizing complier 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

(a) Three-address code.	(b) Static single-assignment form.
$p = a + b$	$p_1 = a + b$
$q = p - c$	$q_1 = p_1 - c$
$p = q * d$	$p_2 = q_1 * d$
$p = e - p$	$p_3 = e - p_2$
$q = p + q$	$q_2 = p_3 + q_1$

(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: Ø-function

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

Returns the value of its argument that corresponds to the control-flow path that was taken to get to the assignment statement containing the Ø-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 type of operands matches the type expected by operator
- Determine the storage needed
- Calculate the address of an array reference
- Insert explicit type conversion
- Choose the right 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.

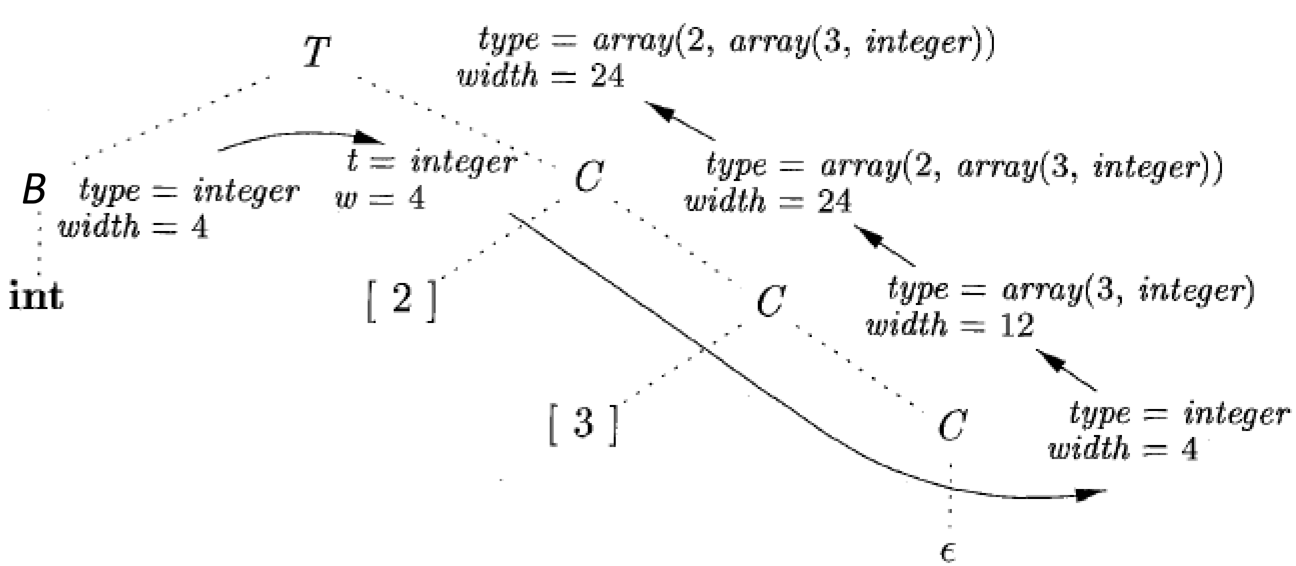
Storage Layout

- 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.

$$
\begin{array}{lcl}
D & \rightarrow & T\ \mathbf{id}\ ;\ D \mid \epsilon \\
T & \rightarrow & B\ C \mid \mathbf{record}\ '\{'\ D\ '\}' \\
B & \rightarrow & \mathbf{int} \mid \mathbf{float} \\
C & \rightarrow & \epsilon \mid [\ \mathbf{num}\ ]\ C
\end{array}
$$

T	$\rightarrow$	B	$\{\ t = B.type;\ w = B.width;\ \}$
		C	
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	$\{\ array(\mathbf{num}.value,\ C_1.type);$ $C.width = \mathbf{num}.value \times C_1.width;\ \}$

`int[2][3]`



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

To keep track of the next available relative address

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

Create a symbol table entry

Translations of Statements and Expressions

Syntax-Directed Definition (SDD)

Syntax-Directed Translation (SDT)

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

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

Three-address code of E

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

Build an instruction

Get a temporary variable

Current symbol table

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

a = b + - c

t_1 = minus c
t_2 = b + t_1
a = t_2

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

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

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

Arrays

- Elements of the same type
- Stored consecutively in memory
- In languages like C or Java elements are: 0, 1, …, n-1
- In some other languages:

  low, low+1, …, 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[i1][i2]:
w1 is width of a row and
w2 the width of an element

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

or

w is width of an element, n2 is number of elements per row

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

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

or

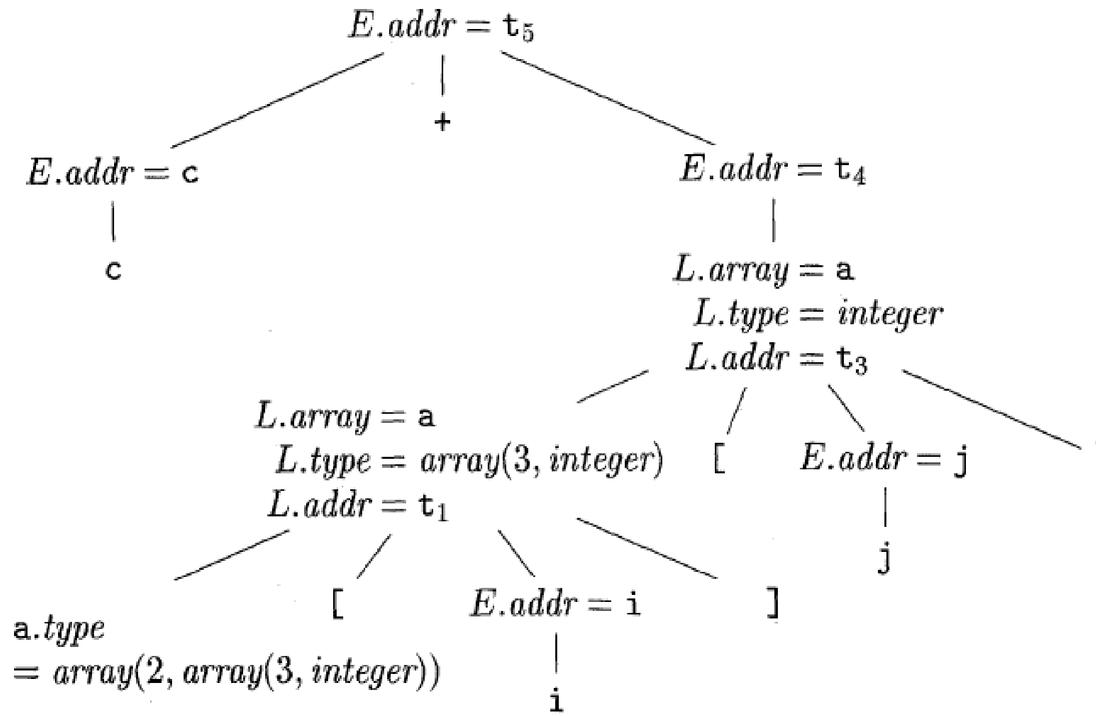
$$base + ((\cdots(i_1 \times n_2 + i_2) \times n_3 + i_3) \cdots) \times n_k + i_k) \times w$$

$$
\begin{array}{lcll}
S & \rightarrow & \textbf{id} = E \;; & \{\; gen(\; top.get(\textbf{id}.lexeme)\; '=' \; E.addr); \;\} \\
& | & L = E \;; & \{\; gen(L.addr.base \; '[' \; L.addr \; ']' \; '=' \; E.addr); \;\} \\
E & \rightarrow & E_1 + E_2 & \{\; E.addr = \textbf{new}\; Temp\,(); \\
& & & \quad gen(E.addr \; '=' \; E_1.addr \; '+' \; E_2.addr); \;\} \\
& | & \textbf{id} & \{\; E.addr = top.get(\textbf{id}.lexeme); \;\} \\
& | & L & \{\; E.addr = \textbf{new}\; Temp\,(); \\
& & & \quad gen(E.addr \; '=' \; L.array.base \; '[' \; L.addr \; ']'); \;\} \\
L & \rightarrow & \textbf{id} \; [\; E \;] & \{\; L.array = top.get(\textbf{id}.lexeme); \\
& & & \quad L.type = L.array.type.elem; \\
& & & \quad L.addr = \textbf{new}\; Temp\,(); \\
& & & \quad gen(L.addr \; '=' \; E.addr \; '*' \; L.type.width); \;\} \\
& | & L_1 \; [\; E \;] & \{\; L.array = L_1.array; \\
& & & \quad L.type = L_1.type.elem; \\
& & & \quad t = \textbf{new}\; Temp\,(); \\
& & & \quad L.addr = \textbf{new}\; Temp\,(); \\
& & & \quad gen(t \; '=' \; E.addr \; '*' \; L.type.width); \;\} \\
& & & \quad gen(L.addr \; '=' \; L_1.addr \; '+' \; t); \;\}
\end{array}
$$

temporary used while computing the offset

pointer to symbol table entry

$S \rightarrow$	**id** = E ;	{ *gen*(*top.get*(**id**.*lexeme*) '=' *E.addr*); }
$\mid$	L = E ;	{ *gen*(*L.addr.base* '[' *L.addr* ']' '=' *E.ad*
$E \rightarrow$	E_1 + E_2	{ *E.addr* = **new** *Temp* (); *gen*(*E.addr* '=' E_1.*addr* '+' E_2.*addr*);
$\mid$	**id**	{ *E.addr* = *top.get*(**id**.*lexeme*); }
$\mid$	L	{ *E.addr* = **new** *Temp* (); *gen*(*E.addr* '=' *L.array.base* '[' *L.addr*
$L \rightarrow$	**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*
$\mid$	L_1 [E]	{ *L.array* = L_1.*array*; *L.type* = L_1.*type.elem*; *t* = **new** *Temp* (); *L.addr* = **new** *Temp* (); *gen*(*t* '=' *E.addr* '*' *L.type.width*); } *gen*(*L.addr* '=' L_1.*addr* '+' *t*); }



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

A is a 2x3 array of integers

t_1 = i * 12
t_2 = j * 4
t_3 = t_1 + t_2
t_4 = a [t_3]
t_5 = c + t_4

So…

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