Compilerconstructie

najaar 2019

http://www.liacs.leidenuniv.nl/~vlietrvan1/coco/

Rudy van Vliet
kamer 140 Snellius, tel. 071-527 2876
rvvliet(at)liacs(dot)nl

college 6, vrijdag 18 oktober 2019

Intermediate Code Generation 1

Today

- Types of three-address instructions

- Implementations of three-address instructions

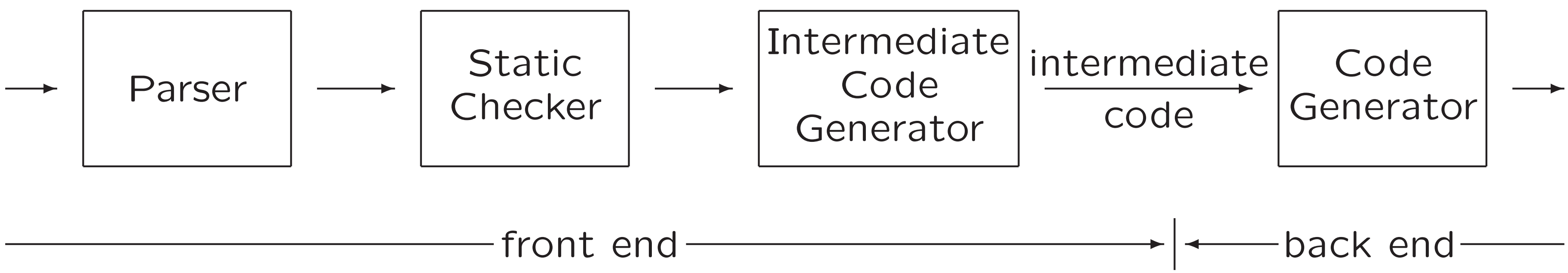
- Translation of expressions

- Translation of array references

6. Intermediate Code Generation

- Front end: generates intermediate representation

- Back end: generates target code



Intermediate Representation

- Facilitates efficient compiler suites: $m + n$ instead of $m * n$
- Different types, e.g.,
  - syntax trees
  - three-address code: $x = y \; op \; z$
- High-level vs. low-level
- C for C++

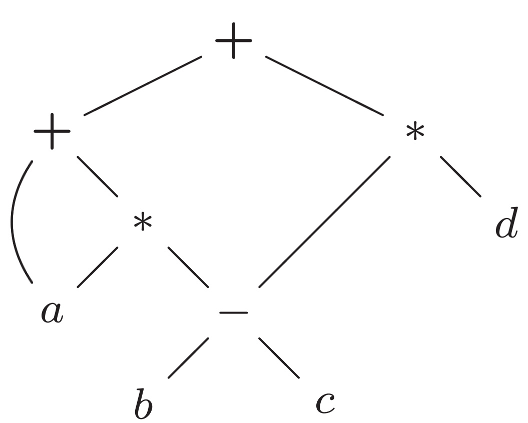
Source Program → High Level Intermediate Representation → · · · → Low Level Intermediate Representation → Target Code

6.2 Three-Address Code

- Linearized representation of syntax tree / syntax DAG
- Sequence of instructions: $x = y$ *op* z

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

Syntax DAG



Three-address code

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

6.2.1 Addresses and Instructions

At most three addresses per instruction

- Name: source program name / symbol-table entry
- Constant
- Compiler-generated temporary: distinct names

Three-Address Instructions

1.	Assignment instructions	$x = y$ *op* z
2.	Assignment instructions	$x =$ *op* y
3.	Copy instructions	$x = y$
4.	Unconditional jumps	`goto` L
5.	Conditional jumps	`if` x `goto` L / `ifFalse` x `goto` L
6.	Conditional jumps	`if` x *relop* y `goto` L / `ifFalse`...
7.	Procedure calls and returns	`param` x_1 `param` x_2 ... `param` x_n `call` p, n `return` y
8.	Indexed copy instructions	$x = y[i]$ / $x[i] = y$
9.	Address and pointer assignments	$x = \&y, \quad x = *y, \quad *x = y$

Symbolic label L represents index of instruction

Three-Address Instructions (Example)

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

Syntax tree. . .

Three-Address Instructions (Example)

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

Syntax tree. . .

Two examples of possible translations:

Symbolic labels

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

Position numbers

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

Implementation of Three-Address Instructions

Quadruples: records *op*, *vararg*1, *vararg*2, *result*

Example: `a = b * - c + b * - c`

Syntax tree. . .

Implementation of Three-Address Instructions

Quadruples: records *op*, *vararg*1, *vararg*2, *result*

Example: `a = b * - c + b * - c`

Syntax tree. . .

Three-address code

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

	op	*vararg*1	*vararg*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
	. . .			

Implementation of Three-Address Instructions

Three-address code

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

	op	*vararg*1	*vararg*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
	...			

Exceptions

1. minus, $=$
2. param
3. jumps

Field *result* mainly for temporaries...

Implementation of Three-Address Instructions

Triples: records *op*, *vararg*1, *vararg*2

Example: `a = b * - c + b * - c`

Syntax tree. . .

Three-address code

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

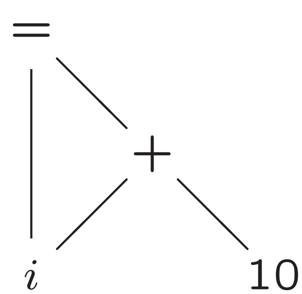
	op	*vararg*1	*vararg*2
0	`minus`	c	
1	$*$	b	(0)
2	`minus`	c	
3	$*$	b	(2)
4	$+$	(1)	(3)
5	$=$	a	(4)
		. . .	

A slide from lecture 5:

6.1.2 The Value-Number Method

An implementation of DAG

DAG for $i = i + 10$



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

- Search array for (existing) node
- Use hash table

Implementation of Three-Address Instructions

Three-address code

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

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

Equivalent to DAG

Pro: temporaries are implicit
Con: difficult to rearrange code
Indirect triples...

Three-Address Instructions in quadruples, triples...

1.	Assignment instructions	$x = y$ *op* z
2.	Assignment instructions	$x =$ *op* y
3.	Copy instructions	$x = y$
4.	Unconditional jumps	`goto` L
5.	Conditional jumps	`if` x `goto` L / `ifFalse` x `goto` L
6.	Conditional jumps	`if` x *relop* y `goto` L / `ifFalse`...
7.	Procedure calls and returns	`param` x_1 `param` x_2 ... `param` x_n `call` p, n `return` y
8.	Indexed copy instructions	$x = y[i]$ / $x[i] = y$
9.	Address and pointer assignments	$x = \&y$, $x = *y$, $*x = y$

Symbolic label L represents index of instruction

6.4 Translation of Expressions

- Temporary names are created
  $E \rightarrow E_1 + E_2$ yields $t = E_1 + E_2$, e.g.,

```
t5 = t2 + t4
 a = t5
```

- If expression is identifier, then no new temporary

- Nonterminal E has two attributes:
  - $E.addr$ – address that will hold value of E
  - $E.code$ – three-address code sequence

- Nonterminal S has one attribute:
  - $S.code$ – three-address code sequence

6.4.1 Operations Within Expressions

Syntax-directed definition
to produce three-address code for assignments

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

Example: $a = b + -c\ \ldots$

6.4.2 Incremental Translation

Translation scheme

to produce three-address code for assignments

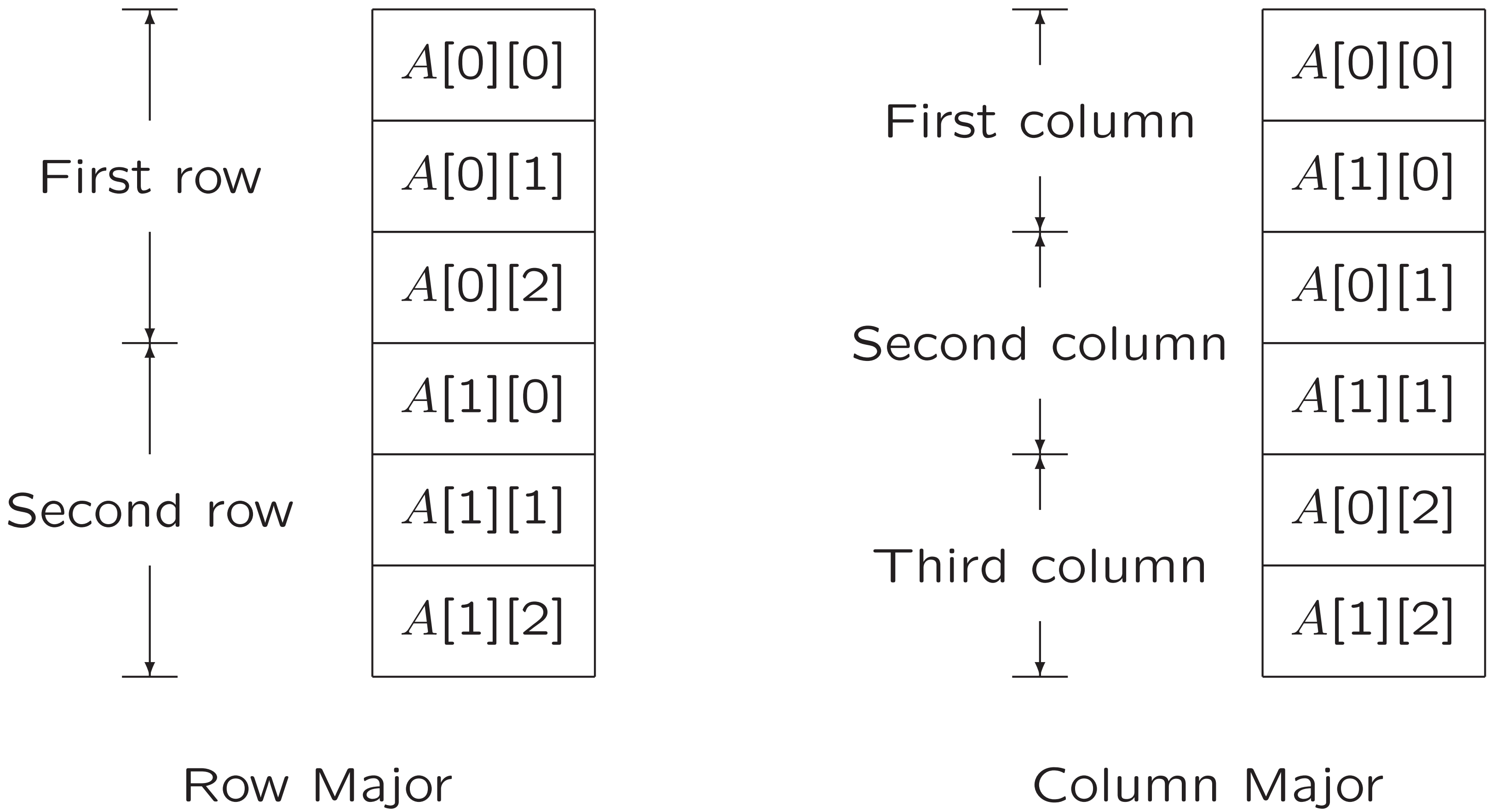
$$
\begin{array}{lcll}
S & \rightarrow & \textbf{id} = E; & \{ \ \textit{gen}(\textit{top.get}(\textbf{id}.\textit{lexeme})\ '='\ E.\textit{addr}); \} \\
E & \rightarrow & E_1 + E_2 & \{ \ E.\textit{addr} = \textbf{new}\ \ \textit{Temp}(); \\
 & & & \quad \textit{gen}(E.\textit{addr}\ '='\ E_1.\textit{addr}\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

6.4.3 Addressing Array Elements

- Array $A[n]$ with elements at positions $0, 1, \ldots, n-1$
- Let
  - w be width of array element
  - *base* be relative address of storage allocated for A $(= A[0])$

  Element $A[i]$ begins in location *base* $+ i \times w$
- In two dimensions. . .

Two-Dimensional Arrays



6.4.3 Addressing Array Elements

- In two dimensions, let
  - w_1 be width of row,
  - w_2 be width of element of row

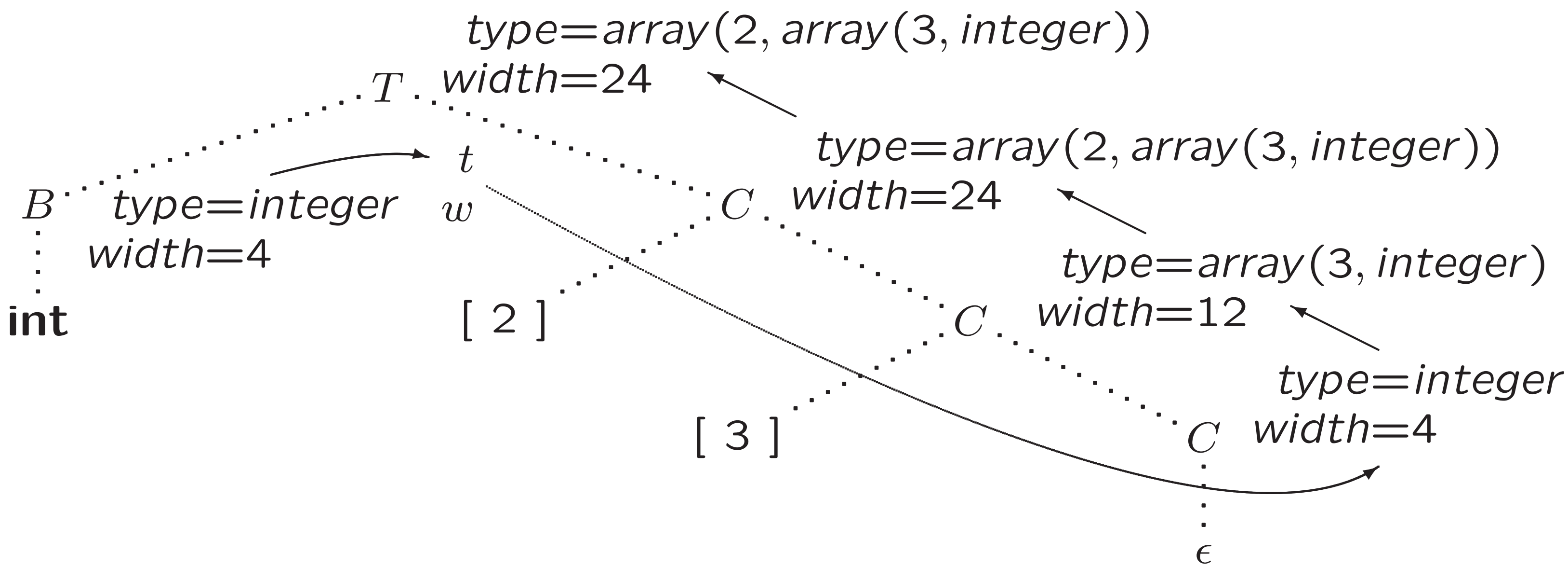
  Element $A[i][j]$ begins in location $\quad \textit{base} + i \times w_1 + j \times w_2$

- In k dimensions $\quad \textit{base} + i_1 * w_1 + i_2 * w_2 + \cdots + i_k * w_k$

A slide from lecture 5:

Types and Their Widths (Example)

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



Addressing Array Elements

More general: `int A[low..high];`

- $\mathit{base} + (i - \mathit{low}) \times w = i \times w + \underbrace{\mathit{base} - \mathit{low} \times w}_{c}$

- More dimensions. . .

- Precalculate c

- Dynamic arrays. . .

6.4.4 Translation of Array References

L generates array name followed by sequence of index expressions

$$\begin{array}{rcl} E & \rightarrow & E + E \mid \textbf{id} \mid L \\ L & \rightarrow & L[E] \mid \textbf{id}[E] \end{array}$$

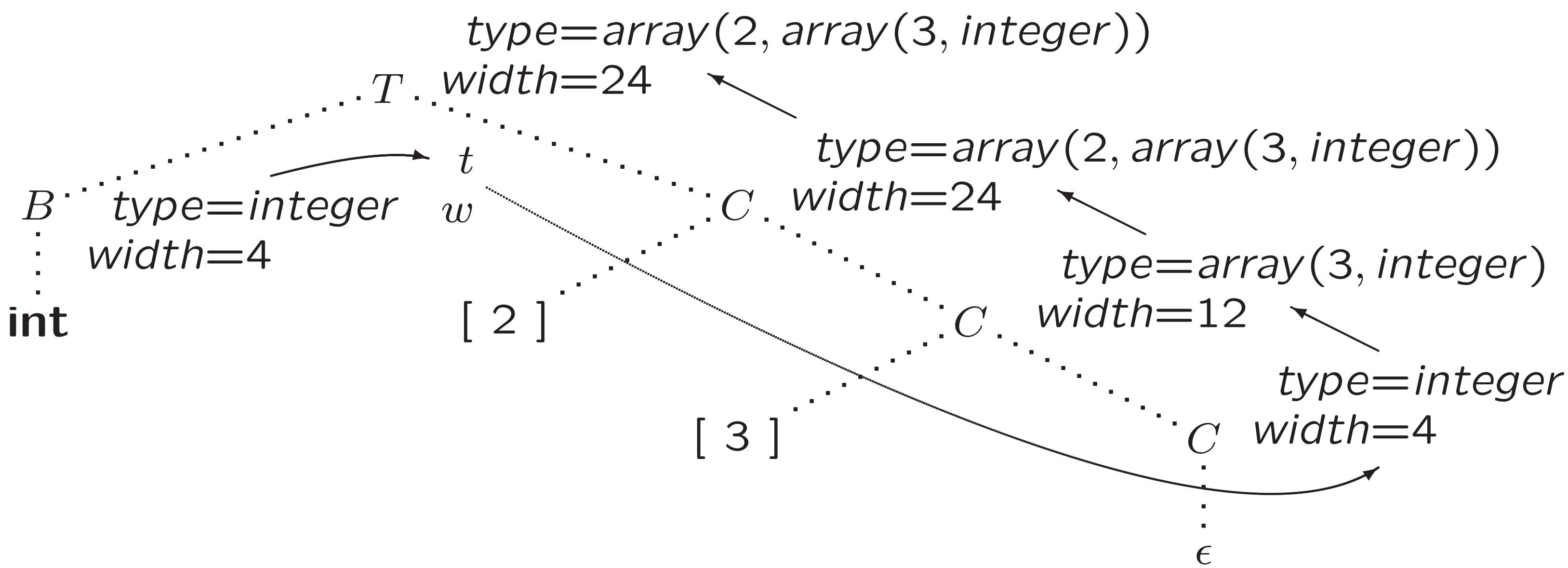
Parse tree for $c + a[i][j]\ldots$

Compare to 'syntax tree' for declaration type...

A slide from lecture 5:

Types and Their Widths (Example)

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



Translation of Array References

Three synthesized attributes

- L.*addr*: temporary used to compute location in array
- L.*array*: pointer to symbol-table entry for array name
  - L.*array*.*base*: base address of array
- L.*type*: type of subarray generated by L ('what must we multiply index by')
  - For type t: t.*width*
  - For array type t: t.*elem*

Translation of Array References

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

Translation of Array References

$$
\begin{array}{lcll}
S & \rightarrow & \textbf{id} = E; & \{ \ \textit{gen}(\textit{top.get}(\textbf{id}.\textit{lexeme})\ '='\ E.\textit{addr}); \} \\
S & \rightarrow & L = E; & \{ \ \textit{gen}(L.\textit{array.base}\ '['L.\textit{addr}\ ']'\ '='\ E.\textit{addr}); \} \\
E & \rightarrow & E_1 + E_2 & \{ \ E.\textit{addr} = \textbf{new}\ \textit{Temp}(); \\
 & & & \quad \textit{gen}(E.\textit{addr}\ '='\ E_1.\textit{addr}\ '+'\ E_2.\textit{addr}); \} \\
E & \rightarrow & \textbf{id} & \{ \ E.\textit{addr} = \textit{top.get}(\textbf{id}.\textit{lexeme}); \} \\
E_2 & \rightarrow & L & \{ \ E_2.\textit{addr} = \textbf{new}\ \textit{Temp}(); \\
 & & & \quad \textit{gen}(E_2.\textit{addr}\ '='\ L.\textit{array.base}\ '['L.\textit{addr}\ ']'); \} \\
L_1 & \rightarrow & \textbf{id}\ [E_4] & \{ \ L_1.\textit{array}\ = \textit{top.get}(\textbf{id}.\textit{lexeme}); \\
 & & & \quad L_1.\textit{type}\ = L_1.\textit{array.type.elem}; \\
 & & & \quad L_1.\textit{addr}\ = \textbf{new}\ \textit{Temp}(); \\
 & & & \quad \textit{gen}(L_1.\textit{addr}\ '='\ E_4.\textit{addr}\ '*'\ L_1.\textit{type.width}); \} \\
L & \rightarrow & L_1[E_3] & \{ \ L.\textit{array}\ = L_1.\textit{array}; \\
 & & & \quad L.\textit{type}\ = L_1.\textit{type.elem}; \\
 & & & \quad t = \textbf{new}\ \textit{Temp}(); \\
 & & & \quad L.\textit{addr}\ = \textbf{new}\ \textit{Temp}(); \\
 & & & \quad \textit{gen}(t\ '='\ E_3.\textit{addr}\ '*'\ L.\textit{type.width}); \\
 & & & \quad \textit{gen}(L.\textit{addr}\ '='\ L_1.\textit{addr}\ '+'\ t); \}
\end{array}
$$

Translation of Array References (Example)

- Let a be 2×3 array of integers
- Let c, i and j be integers
- Ad hoc three-address code for `c + a[i][j]` ...
- Annotated parse tree for expression `c + a[i][j]`

Exercise 1

Komende tijd

- Vanmiddag, 14.15-16.00: practicum opdracht 2
- Vrijdag 25 oktober, 14.15-16.00: practicum opdracht 2
- Donderdag 31 oktober: inleveren opdracht 2
- Vrijdag 1 november,
  11.15-13.00: hoorcollege
  14.15-. . . : introductie opdracht 3 + werkcollege
- Inleveren 21 november

Compilerconstructie

college 6
Intermediate Code Generation 1

Chapters for reading:
6.intro, 6.2–6.2.3 (except indirect triples), 6.4