

Compilerconstructie

najaar 2016

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

Rudy van Vliet

kamer 124 Snellius, tel. 071-527 5777

rvvliet(at)liacs(dot)nl

college 6, woensdag 26 oktober 2016

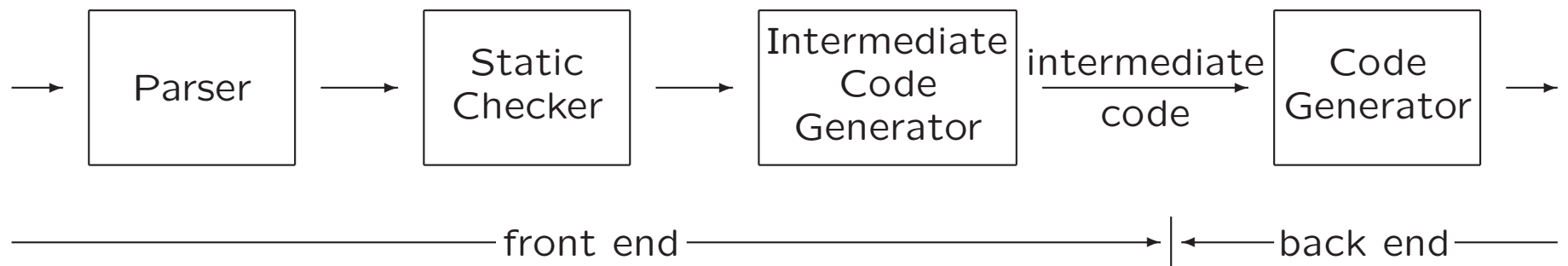
Intermediate Code Generation 1

Today

- Types of three-address instructions
- Implementations of three-address instructions
- Translation of expressions
- Translation of array references

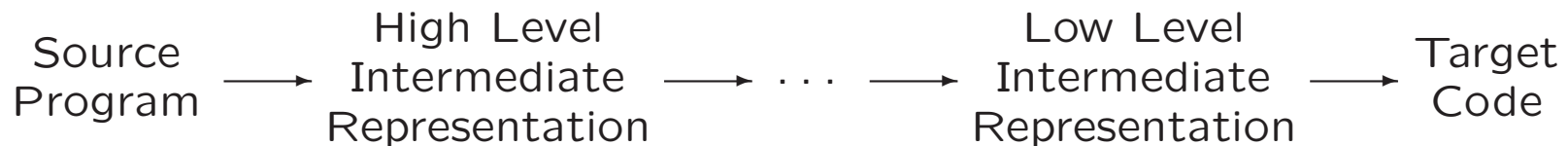
6. Intermediate Code Generation

- 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: $x = y \text{ op } z$
- High-level vs. low-level
- C for C++

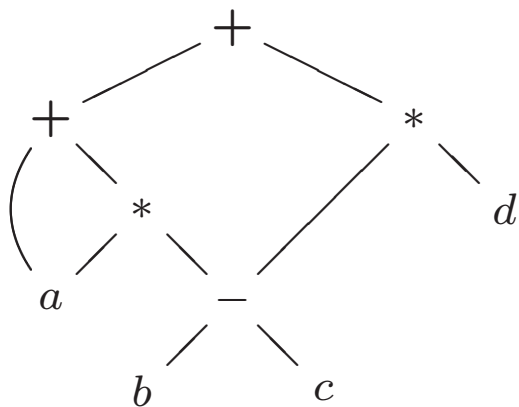


6.2 Three-Address Code

- Linearized representation of syntax tree / syntax DAG
- Sequence of instructions: $x = y \text{ op } z$

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 lable L represents index of instruction

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*, *vararg1*, *vararg2*, *result*

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

Syntax tree...

Implementation of Three-Address Instructions

Quadruples: records *op*, *vararg1*, *vararg2*, *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

	<i>op</i>	<i>vararg1</i>	<i>vararg2</i>	<i>result</i>
0	minus	<i>c</i>		<i>t</i> ₁
1	*	<i>b</i>	<i>t</i> ₁	<i>t</i> ₂
2	minus	<i>c</i>		<i>t</i> ₃
3	*	<i>b</i>	<i>t</i> ₃	<i>t</i> ₄
4	+	<i>t</i> ₂	<i>t</i> ₄	<i>t</i> ₅
5	=	<i>t</i> ₅		<i>a</i>
		...		

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
```

	<i>op</i>	<i>vararg1</i>	<i>vararg2</i>	<i>result</i>
0	minus	<i>c</i>		<i>t1</i>
1	*	<i>b</i>	<i>t1</i>	<i>t2</i>
2	minus	<i>c</i>		<i>t3</i>
3	*	<i>b</i>	<i>t3</i>	<i>t4</i>
4	+	<i>t2</i>	<i>t4</i>	<i>t5</i>
5	=	<i>t5</i>		<i>a</i>
			...	

Exceptions

1. minus, =
2. param
3. jumps

Field *result* mainly for temporaries...

Implementation of Three-Address Instructions

Triples: records *op*, *vararg1*, *vararg2*

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

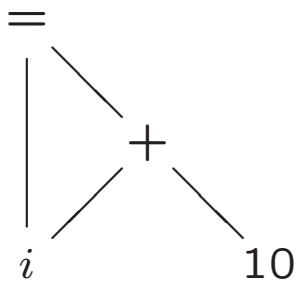
	<i>op</i>	<i>vararg1</i>	<i>vararg2</i>
0	minus	<i>c</i>	
1	*	<i>b</i>	(0)
2	minus	<i>c</i>	
3	*	<i>b</i>	(2)
4	+	(1)	(3)
5	=	<i>a</i>	(4)
		...	

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

	<i>op</i>	<i>vararg1</i>	<i>vararg2</i>
0	minus	<i>c</i>	
1	*	<i>b</i>	(0)
2	minus	<i>c</i>	
3	*	<i>b</i>	(2)
4	+	(1)	(3)
5	=	<i>a</i>	(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
$S \rightarrow \mathbf{id} = E;$	$S.code = E.code \parallel$ $gen(top.get(\mathbf{id}.lexeme) ' = ' E.addr)$
$E \rightarrow E_1 + E_2$	$E.addr = \mathbf{new Temp}()$ $E.code = E_1.code \parallel E_2.code \parallel$ $gen(E.addr ' = ' E_1.addr ' + ' E_2.addr)$
$-E_1$	$E.addr = \mathbf{new Temp}()$ $E.code = E_1.code \parallel$ $gen(E.addr ' = ' 'minus' E_1.addr)$
(E_1)	$E.addr = E_1.addr$ $E.code = E_1.code$
\mathbf{id}	$E.addr = top.get(\mathbf{id}.lexeme)$ $E.code = ''$

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

6.4.2 Incremental Translation

Translation scheme

to produce three-address code for assignments

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

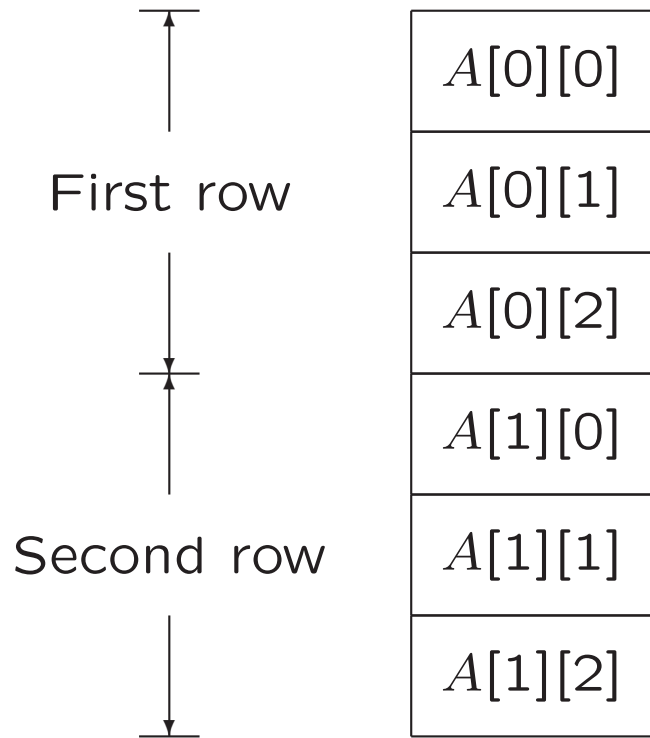
6.4.3 Addressing Array Elements

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

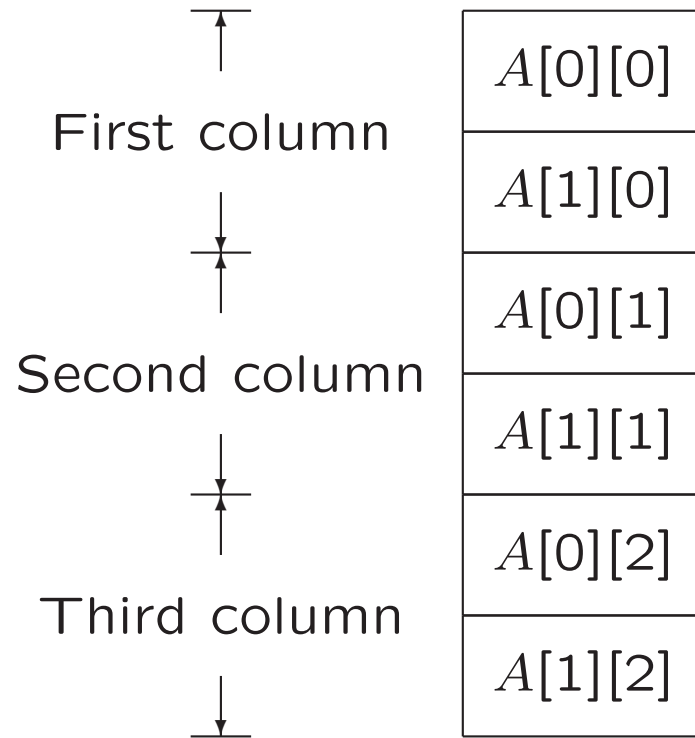
Element $A[i]$ begins in location $base + i \times w$

- In two dimensions. . .

Two-Dimensional Arrays



Row Major



Column Major

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 $base + i \times w_1 + j \times w_2$

- In k dimensions $base + i_1 * w_1 + i_2 * w_2 + \dots + i_k * w_k$

Addressing Array Elements

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

- $base + (i - low) \times w = i \times w + \underbrace{base - low \times w}_c$
- More dimensions. . .
- Precalculate c
- Dynamic arrays. . .

6.4.4 Translation of Array References

L generates array name followed by sequence of index expressions

$$E \rightarrow E + E \mid \mathbf{id} \mid L$$

$$L \rightarrow L[E] \mid \mathbf{id}[E]$$

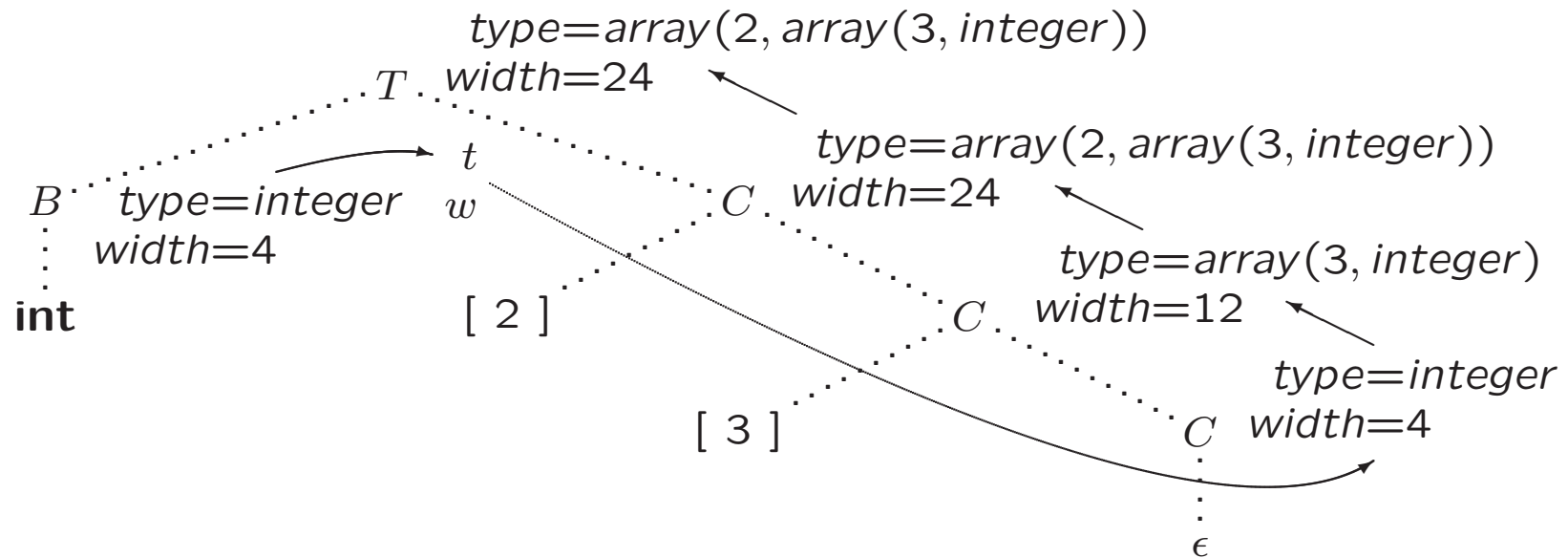
Parse tree for $c + a[i][j]. \dots$

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

A slide from lecture 5:

Types and Their Widths (Example)

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



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 **sub**array generated by *L* ('what must we multiply index by')
 - For type *t*: *t.width*
 - For array type *t*: *t.elem*

Translation of Array References

```
S → id = E;    { gen(top.get(id.lexeme) ' = ' E.addr); }
S → L = E;    { gen(L.array.base '[' L.addr ']' ' = ' E.addr); }
E → E1 + E2 { E.addr = new Temp();
                  gen(E.addr ' = ' E1.addr ' + ' E2.addr); }
E → id        { E.addr = top.get(id.lexeme); }
E → 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); }
L → 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); }
```

Translation of Array References

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

Translation of Array References (Example)

- Let a be 2×3 array of integers
- Let c , i and j be integers
- Annotated parse tree for expression $c + a[i][j]$

Exercise 1

Komende week

- Vrijdag 28 oktober, 11.15-13.00: practicum
- Dinsdag 1 november: inleveren opdracht 2
- Woensdag 2 november,
11.15-13.00: hoorcollege + introductie opdracht 3
13.45-...: werkcollege
- Inleveren 22 november

Compilerconstructie

college 6

Intermediate Code Generation 1

Chapters for reading:

6.intro, 6.2–6.2.3 (except indirect triples), 6.4