Compilerconstructie

najaar 2012

http://www.liacs.nl/home/rvvliet/coco/

Rudy van Vliet
kamer 124 Snellius, tel. 071-527 5777
rvvliet(at)liacs.nl

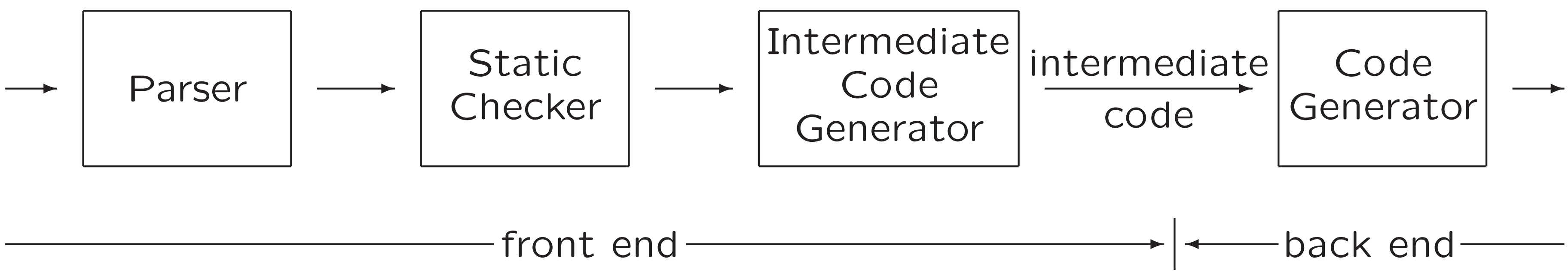
college 6, dinsdag 23 oktober 2012

Intermediate Code Generation

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++

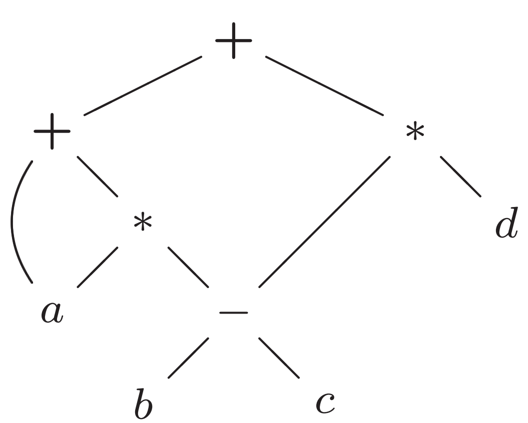
Source Program → High Level Intermediate Representation → $\cdots$ → Low Level Intermediate Representation → Target Code

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
```

Addresses

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	$x = \&y$, $x = *y$, $*x = y$

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

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
		$\ldots$		

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
	$\cdots$			

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	t_5
		. . .	

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	*ararg*1	*ararg*2
0	minus	c	
1	$*$	b	(0)
2	minus	c	
3	$*$	b	(2)
4	$+$	(1)	(3)
5	$=$	a	t_5
		$\cdots$	

Equivalent to DAG

Special case: $x[i] = y$ or $x = y[i]$

Pro: temporaries are implicit
Con: difficult to rearrange code

Implementation of Three-Address Instructions

Indirect triples: pointers to triples

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
```

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

	op	*vararg1*	*vararg2*
0	`minus`	c	
1	$*$	b	(0)
2	`minus`	c	
3	$*$	b	(2)
4	$+$	(1)	(3)
5	$=$	a	(4)
	...		

6.3.3 Declarations

- Three-address code is simplistic
  It assumes that names of variables can be easily resolved by the back end in global or local variables
- We need symbol tables to record global and local declarations in procedures, blocks, and structs to resolve names
- Symbol table contains type and relative adress of names

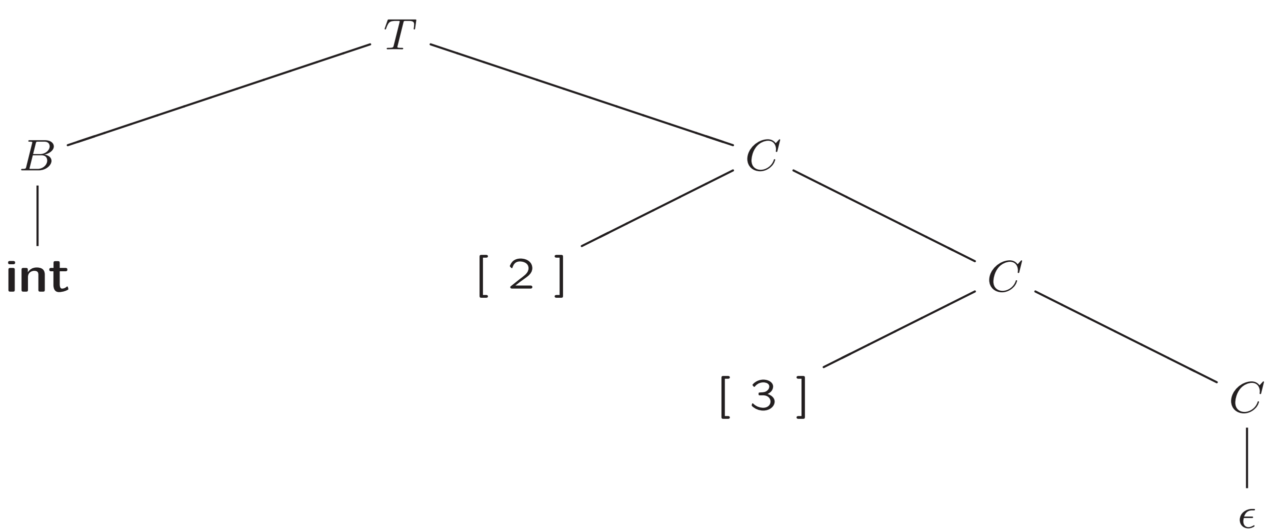
Example:

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

Structure of Types (Example)

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

`int[2][3]`



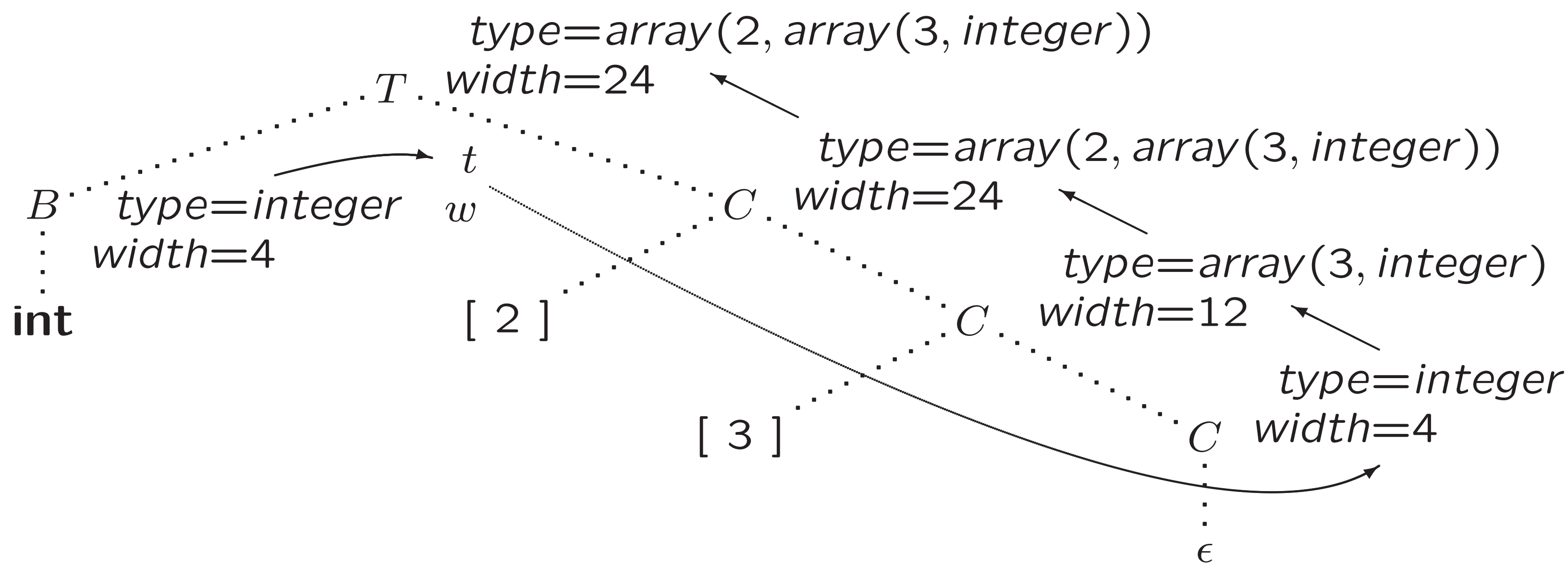
Storage Layout at Compile Time

- Storage comes in blocks of contiguous bytes
- Width of type is number of bytes needed

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

Types and Their Widths (Example)

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



Sequences of Declarations

$$D \;\rightarrow\; T\ \mathbf{id}; D \;\mid\; \epsilon$$

Use *offset* as next available address

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

Fields in Records and Classes

Example

```
float x;
record { float x; float y; } p;
record { int tag; float x; float y; } q;
x = p.x + q.x;
```

$$
\begin{array}{lcl}
D & \rightarrow & T\ \textbf{id};D \ \mid\ \epsilon \\
T & \rightarrow & \textbf{record}\ '\{'\ D\ '\}'
\end{array}
$$

- Fields are specified by sequence of declarations
  - Field names within record must be distinct
  - Relative address for field is relative to data area for that record

Fields in Records and Classes

Stored in separate symbol table t

Record type has form *record*(t)

$$
\begin{array}{lllll}
T & \rightarrow & \textbf{record}\ '\{' & \{ & \textit{Env.push(top)};\ \textit{top} = \textbf{new}\ \textit{Env}(); \\
& & & & \textit{Stack.push(offset)};\ \textit{offset} = 0; \} \\
& & D\ '\}' & \{ & T.\textit{type} = \textit{record(top)};\ T.\textit{width} = \textit{offset}; \\
& & & & \textit{top} = \textit{Env.pop}();\ \textit{offset} = \textit{Stack.pop}(); \}
\end{array}
$$

6.4 Translation of Expressions

- Temporary names are created
  $E \to E_1 + E_2$ 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

Syntax-Directed Definition

To produce three-address code for assignments

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

Translation scheme

To incrementally produce three-address code for assignments

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

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 $\mathit{base} + i \times w$

- 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 $\mathit{base} + i \times w_1 + j \times w_2$

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

Translation of Array References

L generates array name followed by sequence of index expressions

$$L \to L[E] \ \mid \ \textbf{id}[E]$$

Three synthesized attributes

- $L.\mathit{addr}$: temporary used to compute location in array
- $L.\mathit{array}$: pointer to symbol-table entry for array name
  - $L.\mathit{array}.\mathit{base}$: base address of array
- $L.\mathit{type}$: type of subarray generated by L
  - For type t: $t.\mathit{width}$
  - For array type t: $t.\mathit{elem}$

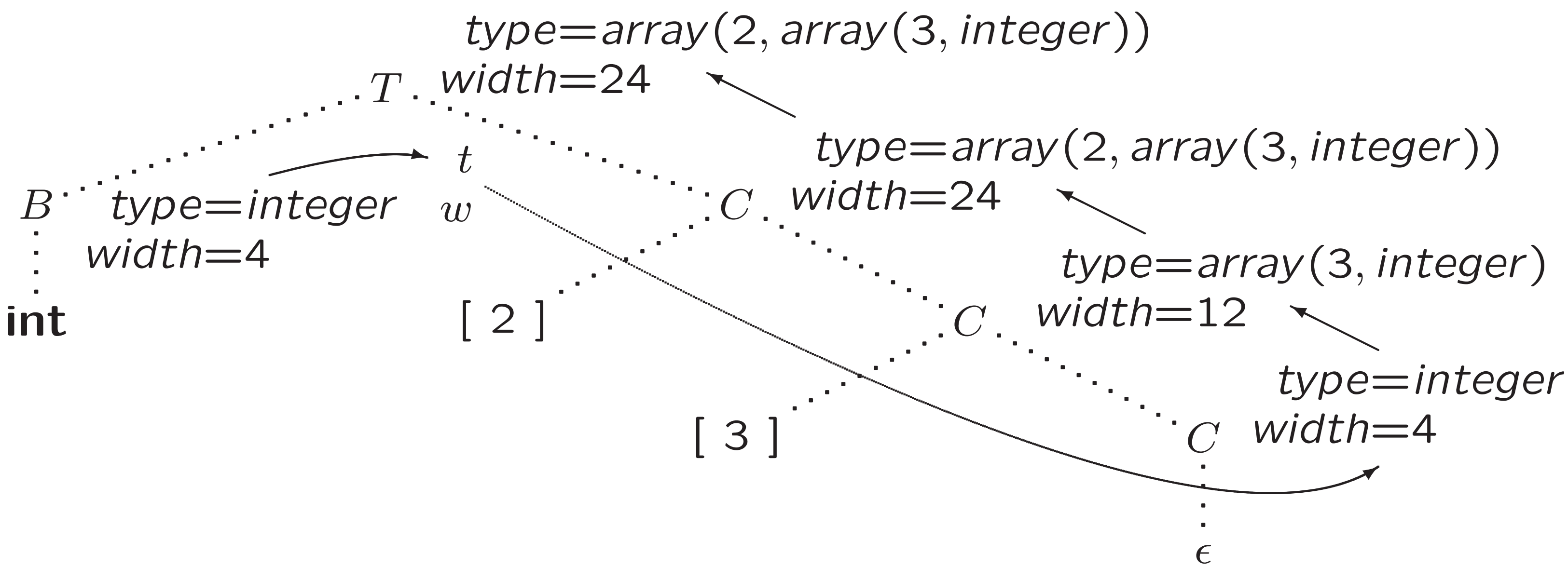
Translation of Array References

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

Translation of Array References

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

Types and Their Widths (Example)



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]`

6.6 Control Flow

- Boolean expressions used to

  1. Alter flow of control: **if** $(E)\ S$

  2. Compute logical values, cf. arithmetic expressions

- Generated by

  $B \rightarrow B || B \quad | \quad B \&\& B \quad | \quad !B \quad | \quad (B) \quad | \quad E$ **rel** $E \quad | \quad$ **true** $\quad | \quad$ **false**

- In $B_1 || B_2$, if B_1 is true, then expression is true
  In $B_1 \&\& B_2$, if . . .

Short-Circuit Code

or jumping code

Boolean operators ||, && and ! translate into jumps

Example

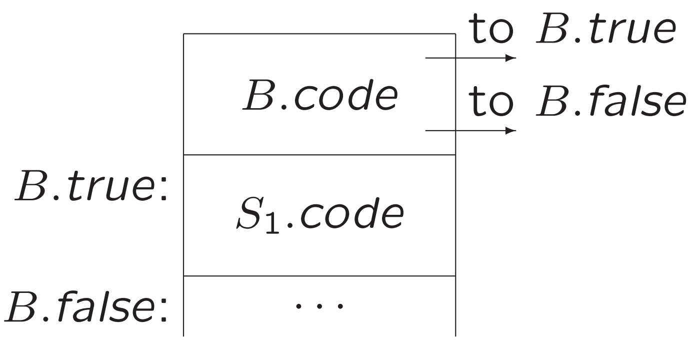
```
if ( x < 100 || x > 200 && x!=y ) x = 0;
```

Precedence: || $<$ && $<$!

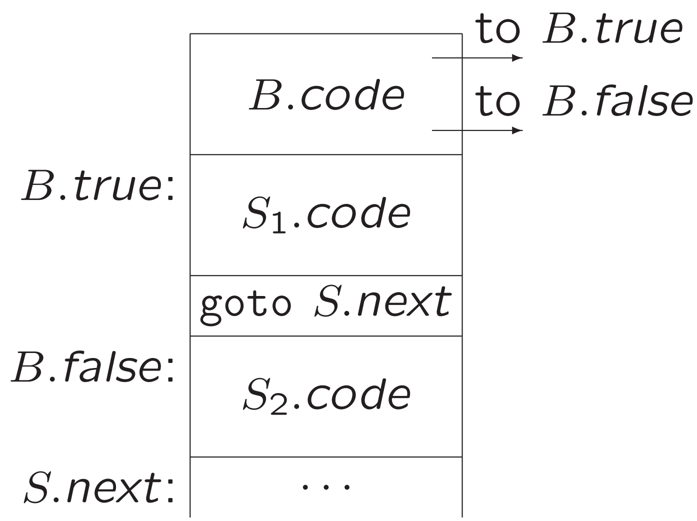
```
      if x < 100 goto L2
      ifFalse x > 200 goto L1
      ifFalse x != y goto L1
L2:   x = 0
L1:
```

Flow-of-Control Statements

$$S \rightarrow \textbf{if } (B) \; S_1$$
$$S \rightarrow \textbf{if } (B) \; S_1 \textbf{ else } S_2$$
$$S \rightarrow \textbf{while } (B) \; S_1$$



if



if-else

Translation using

- synthesized attributes *B.code* and *S.code*
- inherited attributes (labels) *B.true*, *B.false* and *S.next*

Syntax-Directed Definition

Production	Semantic Rules
$P \rightarrow S$	$S.next = newlabel()$ $P.code = S.code \ \|\| \ label(S.next)$
$S \rightarrow$ **if** $(B)\ S_1$	$B.true = newlabel()$ $B.false = S_1.next = S.next$ $S.code = B.code \ \|\| \ label(B.true) \ \|\| \ S_1.code$
$B \rightarrow B_1 \|\| B_2$	$B_1.true = B.true$ $B_1.false = newlabel()$ $B_2.true = B.true$ $B_2.false = B.false$ $B.code = B_1.code \ \|\| \ label(B_1.false) \ \|\| \ B_2.code$
$B_1 \rightarrow E_1$ **rel** E_2	$B_1.code = E_1.code \ \|\| \ E_2.code$ $\|\| \ gen($'`if`' $E_1.addr$ **rel**$.op$ $E_2.addr$ '`goto`' $B_1.true)$ $\|\| \ gen($'`goto`' $B_1.false)$
$B_2 \rightarrow B_3 \&\& B_4$	$B_3.true = newlabel()$ $B_3.false = B_2.false$ $B_4.true = B_2.true$ $B_4.false = B_2.false$ $B_2.code = B_3.code \ \|\| \ label(B_3.true) \ \|\| \ B_4.code$

Avoiding Redundant Gotos

```
      if x < 100 goto L2
      goto L3
L3:   if x > 200 goto L4
      goto L1
L4:   if x != y goto L2
      goto L1
L2:   x = 0
L1:
```

Versus

```
      if x < 100 goto L2
      ifFalse x > 200 goto L1
      ifFalse x != y goto L1
L2:   x = 0
L1:
```

6.7 Backpatching

- Code generation problem:
  - Labels (addresses) that control must go to may not be known at the time that jump statements are generated

- One solution:
  - Separate pass to bind labels to addresses

- Other solution: backpatching
  - Generate jump statements with empty target
  - Add such statements to a list
  - Fill in labels when proper label is determined

Backpatching

- Synthesized attributes $B.\mathit{truelist}$, $B.\mathit{falselist}$, $S.\mathit{nextlist}$ containing lists of jumps

- Three functions

  1. *makelist*(i) creates new list containing index i
  2. *merge*(p_1, p_2) concatenates lists pointed to by p_1 and p_2
  3. *backpatch*(p, i) inserts i as target label for each instruction on list pointed to by p

Grammars for Backpatching

- Grammar for boolean expressions:

$$\begin{array}{rcl} B & \rightarrow & B_1 || M B_2 \;\mid\; B_1 \&\& M B_2 \;\mid\; !B_1 \;\mid\; (B_1) \\ & & \mid\; E_1 \textbf{ rel } E_2 \;\mid\; \textbf{true} \;\mid\; \textbf{false} \\ M & \rightarrow & \epsilon \end{array}$$

M is marker nonterminal

- Grammar for flow-of-control statements

$$\begin{array}{rcl} S & \rightarrow & \textbf{if } (B)\ S_1 \;\mid\; \textbf{if } (B)\ S_1 \textbf{ else } S_2 \\ & \mid & \textbf{while } (B)\ S_1 \;\mid\; \{L\} \;\mid\; A; \\ N & \rightarrow & \epsilon \\ L & \rightarrow & L_1 S \;\mid\; S \end{array}$$

Example: `if (x < 100 || x > 200 && x != y) x = 0;`

Translation Scheme for Backpatching

$B \to E_1$ **rel** E_2	$\{$ B.*truelist* $=$ *makelist*(*nextinstr*); B.*falselist* $=$ *makelist*(*nextinstr* $+ 1$); *gen*('`if`' E_1.*addr* **rel**.*op* E_2.*addr* '`goto _`'); *gen*('`goto _`'); $\}$
$M \to \epsilon$	$\{$ M.*instr* $=$ *nextinstr*; $\}$
$B_2 \to B_3 \&\& M B_4$	$\{$ *backpatch*(B_3.*truelist*, M.*instr*); B_2.*truelist* $=$ B_4.*truelist*; B_2.*falselist* $=$ *merge*(B_3.*falselist*, B_4.*falselist*); $\}$
$B \to B_1 \vert\vert M B_2$	$\{$ *backpatch*(B_1.*falselist*, M.*instr*); B.*truelist* $=$ *merge*(B_1.*truelist*, B_2.*truelist*); $\}$ B.*falselist* $=$ B_2.*falselist*;
$S \to A$	$\{$ S.*nextlist* $=$ **null**; $\}$
$S \to$ **if** (B) $M S_1$	$\{$ *backpatch*(B.*truelist*, M.*instr*); S.*nextlist* $=$ *merge*(B.*falselist*, S_1.*nextlist*); $\}$

6.8 Switch-Statements

switch (E)
{ **case** V_1: S_1
case V_2: S_2
...
case V_{n-1}: S_{n-1}
default S_n
}

Translation:

1. Evaluate expression E
2. Find value V_j in list of cases that matches value of E
3. Execute statement S_j

Translation of Switch-Statement

```
      code to evaluate E into t
      goto test
L1:   code for S1
      goto next
L2:   code for S2
      goto next
      ...
L_{n-1}:  code for S_(n-1)
          goto next
L_{n}:    code for S_n
          goto next
test: if t = V1 goto L1
      if t = V2 goto L2
      ...
      if t = V_{n-1} goto L_{n-1}
      goto L_{n}
next:
```

Volgende week

- Maandag 29 oktober: inleveren opdracht 2
- Practicum over opdracht 3
- Eerst naar 403, daarna naar 302/304
- Inleveren 19 november

Compiler constructie

college 6
Intermediate Code Generation

Chapters for reading:
6.intro, 6.2–6.2.3, 6.3.3–6.3.6, 6.4, 6.6–6.8