Compilerconstructie

najaar 2014

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

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

college 6, dinsdag 21 oktober 2014

Intermediate Code Generation

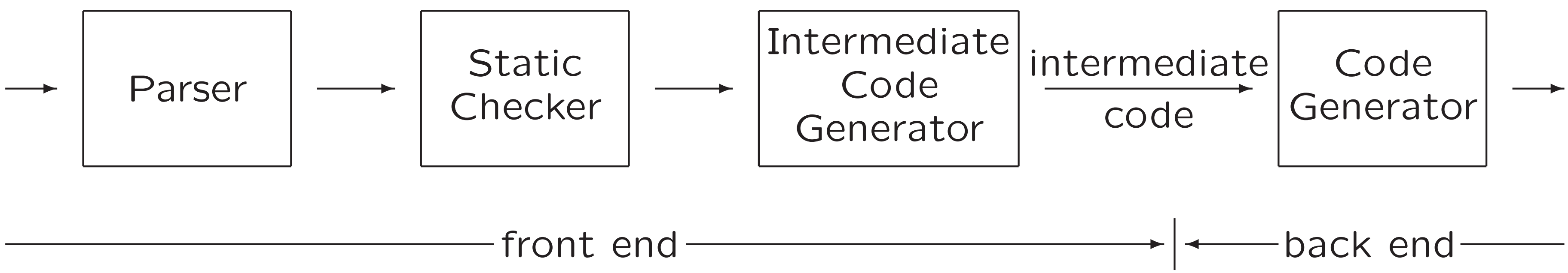
Today

- Types of three-address instructions
- Implementations of three-address instructions
- Translation of expressions
- Translation of array references
- Translation of control flow
  - Top-down passing of labels (inherited attributes)
  - Backpatching (synthesized attributes)
- Translation of switch-statements

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

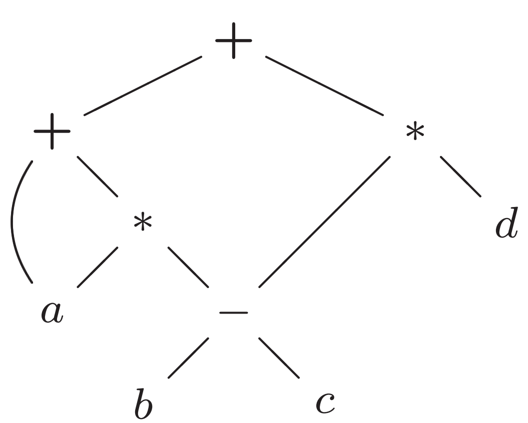
6.2 Three-Address Code

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

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

```
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
		$\ldots$		

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	*vararg1*	*vararg2*	*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
```

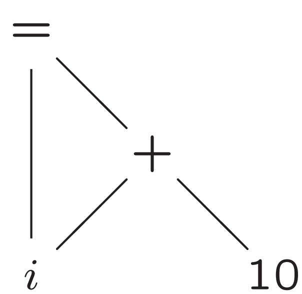
	op	*vararg*1	*vararg*2
0	`minus`	c	
1	$*$	b	(0)
2	`minus`	c	
3	$*$	b	(2)
4	$+$	(1)	(3)
5	$=$	a	(4)
	...		

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	*vararg1*	*vararg2*
0	minus	c	
1	$*$	b	(0)
2	minus	c	
3	$*$	b	(2)
4	$+$	(1)	(3)
5	$=$	a	(4)
		$\ldots$	

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)
	...		

6.4 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: $a = b + -c$ . . .

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

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

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

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

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

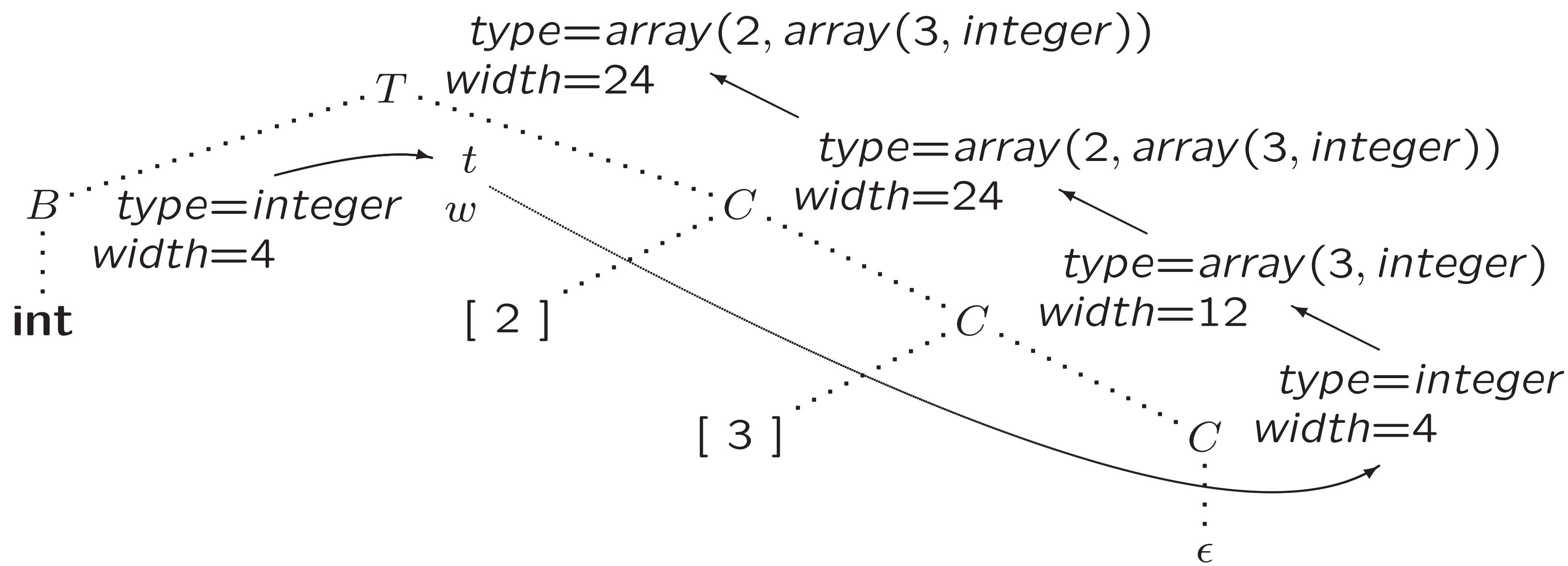
Parse tree for $c + a[i][j]\ldots$

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

A slide from lecture 5:

Types and Their Widths (Example)

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	$\{\ C.\mathit{type} = \mathit{array}(\textbf{num}.\mathit{value},\ C_1.\mathit{type});$ $C.\mathit{width} = \textbf{num}.\mathit{value} \times C_1.\mathit{width};\ \}$



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 subarray generated by L
  - For type t: t.*width*
  - For array type t: t.*elem*

Translation of Array References

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

Translation of Array References

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

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

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

  $B \to B || B \quad | \quad B \&\& B \quad | \quad !B \quad | \quad (B) \quad | \quad E$ **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 . . .

6.6.2 Short-Circuit Code

or jumping code

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

Example

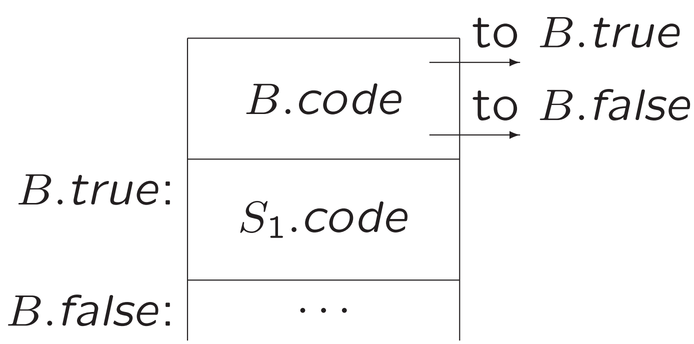
```
if ( x < 100 || x > 200 && x!=y ) x = 0;
```

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

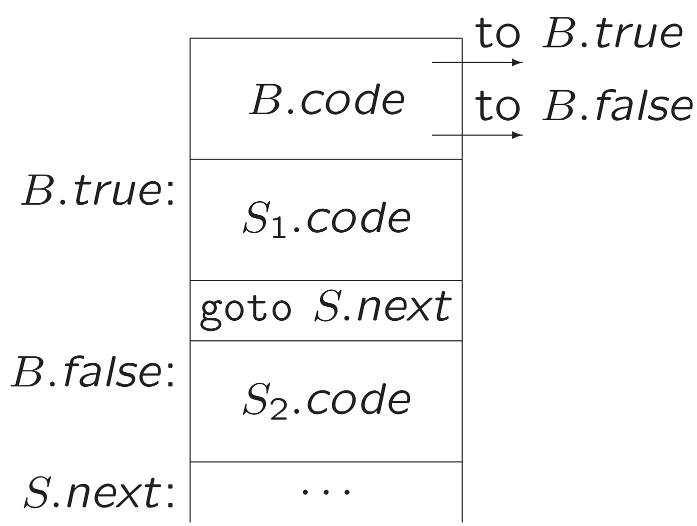
```
      if x < 100 goto L2
      ifFalse x > 200 goto L1
      ifFalse x != y goto L1
L2:   x = 0
L1:
```

6.6.3 Flow-of-Control Statements

$$
\begin{aligned}
S &\rightarrow \textbf{if } (B)\ S_1 \\
S &\rightarrow \textbf{if } (B)\ S_1 \textbf{ else } S_2 \\
S &\rightarrow \textbf{while } (B)\ S_1
\end{aligned}
$$



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
$P \rightarrow S$	$S.\mathit{next} = \mathit{newlabel}()$
	$P.\mathit{code} = S.\mathit{code} \Vert \mathit{label}(S.\mathit{next})$
$S \rightarrow$ **if** $(B)\ S_1$	$B.\mathit{true} = \mathit{newlabel}()$
	$B.\mathit{false} = S_1.\mathit{next} = S.\mathit{next}$
	$S.\mathit{code} = B.\mathit{code} \Vert \mathit{label}(B.\mathit{true}) \Vert S_1.\mathit{code}$
$B \rightarrow B_1 \vert\vert B_2$	$B_1.\mathit{true} = B.\mathit{true}$
	$B_1.\mathit{false} = \mathit{newlabel}()$
	$B_2.\mathit{true} = B.\mathit{true}$
	$B_2.\mathit{false} = B.\mathit{false}$
	$B.\mathit{code} = B_1.\mathit{code} \Vert \mathit{label}(B_1.\mathit{false}) \Vert B_2.\mathit{code}$
$B_1 \rightarrow E_1$ **rel** E_2	$B_1.\mathit{code} = E_1.\mathit{code} \Vert E_2.\mathit{code}$
	$\Vert\ \mathit{gen}(\texttt{'if'}\ E_1.\mathit{addr}\ \mathbf{rel}.\mathit{op}\ E_2.\mathit{addr}\ \texttt{'goto'}\ B_1.\mathit{true})$
	$\Vert\ \mathit{gen}(\texttt{'goto'}\ B_1.\mathit{false})$
$B_2 \rightarrow B_3 \&\& B_4$	$B_3.\mathit{true} = \mathit{newlabel}()$
	$B_3.\mathit{false} = B_2.\mathit{false}$
	$B_4.\mathit{true} = B_2.\mathit{true}$
	$B_4.\mathit{false} = B_2.\mathit{false}$
	$B_2.\mathit{code} = B_3.\mathit{code} \Vert \mathit{label}(B_3.\mathit{true}) \Vert B_4.\mathit{code}$

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

6.6.5 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

6.7.1 One-Pass Code Generation Using Backpatching

- Synthesized attributes B.*truelist*, B.*falselist*, S.*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
  (marker nonterminals omitted for readability)

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

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

Translation Scheme for Backpatching

$B \rightarrow B_1 \| \| M B_2$	{ *backpatch*(B_1.*falselist*, M.*instr*); B.*truelist* = *merge*(B_1.*truelist*, B_2.*truelist*); B.*falselist* = B_2.*falselist*; }
$B \rightarrow B_1 \&\& M B_2$	{ *backpatch*(B_1.*truelist*, M.*instr*); B.*truelist* = B_2.*truelist*; B.*falselist* = *merge*(B_1.*falselist*, B_2.*falselist*); }
$B \rightarrow E_1$ **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 _`'); }
$M \rightarrow \epsilon$	{ M.*instr* = *nextinstr*; }
$S \rightarrow$ **if** (B) $M S_1$	{ *backpatch*(B.*truelist*, M.*instr*); S.*nextlist* = *merge*(B.*falselist*, S_1.*nextlist*); }
$S \rightarrow$ **id** = **num**;	{ S.*nextlist* = **null**; *gen*(**id**.*addr* '=' **num**.*val*); }

Translation Scheme for Backpatching

$B \rightarrow B_1 \| \| M B_2$	{ *backpatch*(B_1.*falselist*, M.*instr*); B.*truelist* = *merge*(B_1.*truelist*, B_2.*truelist*); B.*falselist* = B_2.*falselist*; }
$B_2 \rightarrow B_3 \&\& M B_4$	{ *backpatch*(B_3.*truelist*, M.*instr*); B_2.*truelist* = B_4.*truelist*; B_2.*falselist* = *merge*(B_3.*falselist*, B_4.*falselist*); }
$B \rightarrow E_1$ **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 _`'); }
$M \rightarrow \epsilon$	{ M.*instr* = *nextinstr*; }
$S \rightarrow$ **if** (B) $M S_1$	{ *backpatch*(B.*truelist*, M.*instr*); S.*nextlist* = *merge*(B.*falselist*, S_1.*nextlist*); }
$S \rightarrow$ **id** = **num**;	{ S.*nextlist* = **null**; *gen*(**id**.*addr* '=' **num**.*val*); }

Exercises 2 and 3

Translation Scheme for Backpatching

For Exercise 2

(Boolean Expressions)

$B \rightarrow B_1 \&\& M_1 B_2$	$\{$ *backpatch*$(B_1$.*truelist*$, M_1$.*instr*$)$; B.*truelist* $=$ B_2.*truelist*; B.*falselist* $=$ *merge*$(B_1$.*falselist*$, B_2$.*falselist*$); \}$
$B_2 \rightarrow (\ B_3\ )$	$\{$ B_2.*truelist* $=$ B_3.*truelist*; B_2.*falselist* $=$ B_3.*falselist*$; \}$
$B_3 \rightarrow B_4 \| \| M_2 B_5$	$\{$ *backpatch*$(B_4$.*falselist*$, M_2$.*instr*$)$; B_3.*truelist* $=$ *merge*$(B_4$.*truelist*$, B_5$.*truelist*$)$; B_3.*falselist* $=$ B_5.*falselist*$; \}$
$B \rightarrow E_1$ **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 _`'); $\}$
$M \rightarrow \epsilon$	$\{$ M.*instr* $=$ *nextinstr*; $\}$

Translation Scheme for Backpatching

For Exercise 3

(Flow-of-Control Statements)

$S \rightarrow \{L\}$	$\{$ *S.nextlist* $=$ *L.nextlist*; $\}$
$L \rightarrow L_1 M_3 S_1$	$\{$ *backpatch*(L_1.*nextlist*, M_3.*instr*); *L.nextlist* $=$ S_1.*nextlist*; $\}$
$L_1 \rightarrow S_2$	$\{$ L_1.*nextlist* $=$ S_2.*nextlist*; $\}$
$S_2 \rightarrow$ **if** (B) $M_4 S_3$	$\{$ *backpatch*(*B.truelist*, M_4.*instr*); S_2.*nextlist* $=$ *merge*(*B.falselist*, S_3.*nextlist*); $\}$
$S_3 \rightarrow$ **id** $=$ **num**;	$\{$ *S.nextlist* $=$ **null**; *gen*(**id**.*addr* $'='$ **num**.*val*); $\}$
$M \rightarrow \epsilon$	$\{$ *M.instr* $=$ *nextinstr*; $\}$

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 27 oktober: inleveren opdracht 2
- Dinsdag 28 oktober: practicum over opdracht 3
- Eerst naar 402, daarna naar 302/304
- Inleveren 17 november

Compilerconstructie

college 6
Intermediate Code Generation

Chapters for reading:
6.intro, 6.2–6.2.3, 6.4,
6.6–top-of-page-406,
6.7–6.7.3, 6.8