Compiler Construction

Lecture 23: Code Generation VIII (Generation of Machine Code)

Thomas Noll

Lehrstuhl für Informatik 2
(Software Modeling and Verification)

RWTH Aachen University

noll@cs.rwth-aachen.de

http://www-i2.informatik.rwth-aachen.de/i2/cc08/

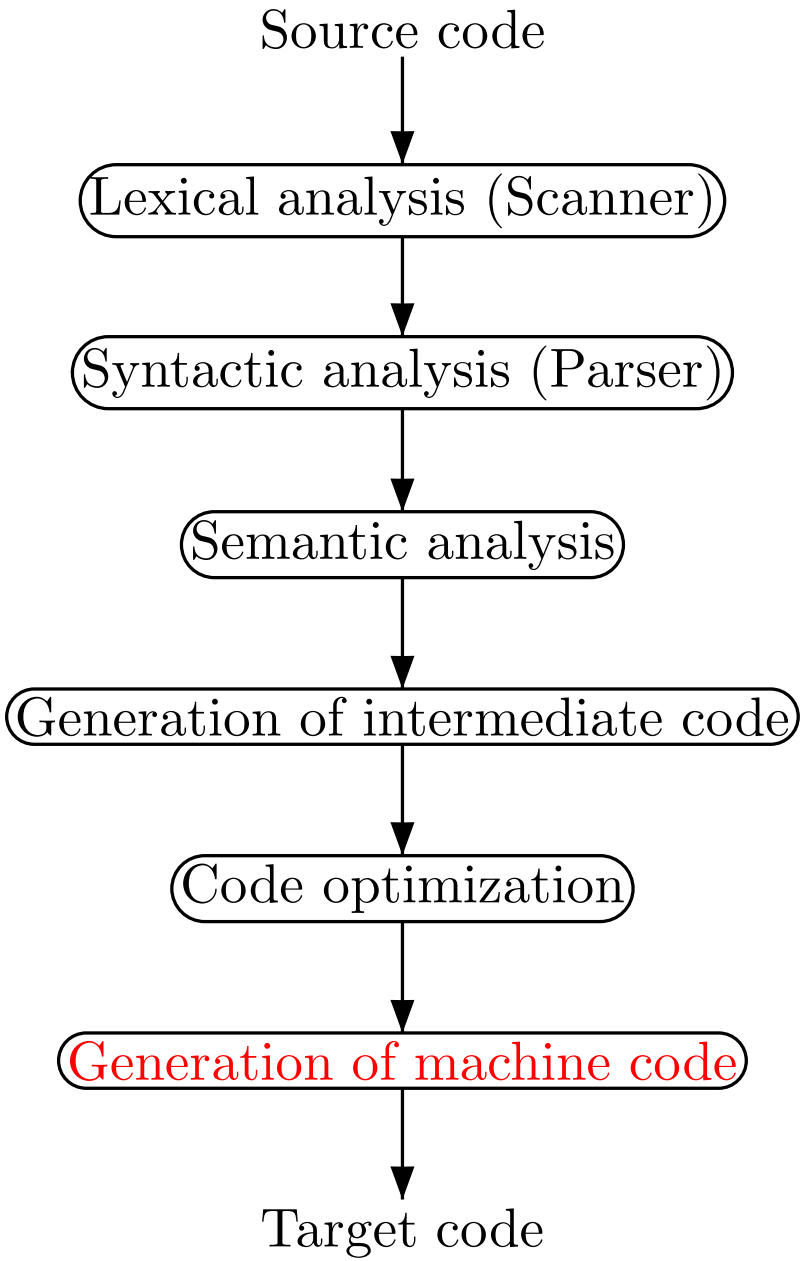
Summer semester 2008

Outline

1 Generation of Machine Code

2 Wrap-Up

Conceptual Structure of a Compiler



The Compiler Backend

Final step: translation of (optimized) abstract machine code into "real" machine code (possibly followed by assembling phase)

The Compiler Backend

Final step: translation of (optimized) abstract machine code into "real" machine code (possibly followed by assembling phase)

Goal: runtime and storage efficiency

- fast backend (optimization problems!)
- fast and compact code
- low memory requirements for data

The Compiler Backend

Final step: translation of (optimized) abstract machine code into "real" machine code (possibly followed by assembling phase)

Goal: runtime and storage efficiency

- fast backend (optimization problems!)
- fast and compact code
- low memory requirements for data

Memory hierarchy: decreasing speed & costs

- registers (program counter, data [universal/floating point/ address], frame pointer, index register, condition code, ...)
- cache ("fast" RAM)
- main memory ("slow" RAM)
- background storage (disks, sticks, ...)

The Compiler Backend

Final step: translation of (optimized) abstract machine code into "real" machine code (possibly followed by assembling phase)

Goal: runtime and storage efficiency

- fast backend (optimization problems!)
- fast and compact code
- low memory requirements for data

Memory hierarchy: decreasing speed & costs

- registers (program counter, data [universal/floating point/address], frame pointer, index register, condition code, ...)
- cache ("fast" RAM)
- main memory ("slow" RAM)
- background storage (disks, sticks, ...)

Principle: use fast memory whenever possible

- evaluation of expressions in registers (instead of data/runtime stack)
- code/procedure stack/heap in main memory

The Compiler Backend

Final step: translation of (optimized) abstract machine code into "real" machine code (possibly followed by assembling phase)

Goal: runtime and storage efficiency

- fast backend (optimization problems!)
- fast and compact code
- low memory requirements for data

Memory hierarchy: decreasing speed & costs

- registers (program counter, data [universal/floating point/address], frame pointer, index register, condition code, ...)
- cache ("fast" RAM)
- main memory ("slow" RAM)
- background storage (disks, sticks, ...)

Principle: use fast memory whenever possible

- evaluation of expressions in registers (instead of data/runtime stack)
- code/procedure stack/heap in main memory

Instruction set: depending on

- number of operands
- type of operands
- addressing modes

Code Generation Phases

1. Register allocation: registers used for
   - values of (frequently used) variables and intermediate results
   - computing memory addresses
   - passing parameters to procedures/functions
2. Instruction selection:
   - translation of abstract instructions into (sequences of) real instructions
   - employ special instructions for efficiency (e.g., `INC(x)` rather than `ADD(x,1)`)
3. Instruction placement: increase level of parallelism and/or pipelining by ordering instructions smartly

Code Generation Phases

1. Register allocation: registers used for
   - values of (frequently used) variables and intermediate results
   - computing memory addresses
   - passing parameters to procedures/functions
2. Instruction selection:
   - translation of abstract instructions into (sequences of) real instructions
   - employ special instructions for efficiency (e.g., `INC(x)` rather than `ADD(x,1)`)
3. Instruction placement: increase level of parallelism and/or pipelining by ordering instructions smartly

Example 23.1

Assignment:

```
z := (u+v)-(w-(x+y))
```

Example 23.1

Assignment:

```
z := (u+v)-(w-(x+y))
```

Target machine with
r registers $\mathtt{R}_0$, $\mathtt{R}_1$, ..., $\mathtt{R}_{r-1}$
and main memory `M`

Example 23.1

Assignment:

```
z := (u+v)-(w-(x+y))
```

Target machine with
r registers $\mathtt{R}_0$, $\mathtt{R}_1$, ..., $\mathtt{R}_{r-1}$
and main memory `M`

Instruction types:

$\mathtt{R}_i$:= `M[`a`]`
`M[`a`]` := $\mathtt{R}_i$
$\mathtt{R}_i$:= $\mathtt{R}_i$ *op* `M[`a`]`
$\mathtt{R}_i$:= $\mathtt{R}_i$ *op* $\mathtt{R}_j$
(with address a)

Example 23.1

Assignment:

```
z := (u+v)-(w-(x+y))
```

Target machine with r registers $\mathtt{R}_0$, $\mathtt{R}_1$, ..., $\mathtt{R}_{r-1}$ and main memory M

Instruction types:

$\mathtt{R}_i$:= M[a]
M[a] := $\mathtt{R}_i$
$\mathtt{R}_i$:= $\mathtt{R}_i$ *op* M[a]
$\mathtt{R}_i$:= $\mathtt{R}_i$ *op* $\mathtt{R}_j$
(with address a)

Instruction sequence for $r = 2$:

```
R0 := M[u]
R0 := R0+M[v]
R1 := M[x]
R1 := R1+M[y]
M[t] := R1
R1 := M[w]
R1 := R1-M[t]
R0 := R0-R1
M[z] := R0
```

Example 23.1

Assignment:

```
z := (u+v)-(w-(x+y))
```

Target machine with r registers R_0, R_1, ..., R_{r-1} and main memory M

Instruction types:

R_i := M[a]
M[a] := R_i
R_i := R_i *op* M[a]
R_i := R_i *op* R_j
(with address a)

Instruction sequence for $r = 2$:

```
R0 := M[u]
R0 := R0+M[v]
R1 := M[x]
R1 := R1+M[y]
M[t] := R1
R1 := M[w]
R1 := R1-M[t]
R0 := R0-R1
M[z] := R0
```

Shorter sequence:

```
R0 := M[w]
R1 := M[x]
R1 := R1+M[y]
R0 := R0-R1
R1 := M[u]
R1 := R1+M[v]
R1 := R1-R0
M[z] := R1
```

Example 23.1

Assignment:

```
z := (u+v)-(w-(x+y))
```

Target machine with r registers R_0, R_1, ..., R_{r-1} and main memory M

Instruction types:

R_i := M[a]
M[a] := R_i
R_i := R_i *op* M[a]
R_i := R_i *op* R_j
(with address a)

Instruction sequence for $r = 2$:

```
R0 := M[u]
R0 := R0+M[v]
R1 := M[x]
R1 := R1+M[y]
M[t] := R1
R1 := M[w]
R1 := R1-M[t]
R0 := R0-R1
M[z] := R0
```

Shorter sequence:

```
R0 := M[w]
R1 := M[x]
R1 := R1+M[y]
R0 := R0-R1
R1 := M[u]
R1 := R1+M[v]
R1 := R1-R0
M[z] := R1
```

- **Reason:** first variant requires intermediate storage for x+y

Example 23.1

Assignment:

```
z := (u+v)-(w-(x+y))
```

Target machine with r registers R_0, R_1, ..., R_{r-1} and main memory M

Instruction types:

R_i := M[a]
M[a] := R_i
R_i := R_i *op* M[a]
R_i := R_i *op* R_j
(with address a)

Instruction sequence for $r = 2$:

```
R0 := M[u]
R0 := R0+M[v]
R1 := M[x]
R1 := R1+M[y]
M[t] := R1
R1 := M[w]
R1 := R1-M[t]
R0 := R0-R1
M[z] := R0
```

Shorter sequence:

```
R0 := M[w]
R1 := M[x]
R1 := R1+M[y]
R0 := R0-R1
R1 := M[u]
R1 := R1+M[v]
R1 := R1-R0
M[z] := R1
```

- **Reason:** first variant requires intermediate storage for x+y
- How to compute systematically?

Register Allocation

Example 23.1

Assignment:

```
z := (u+v)-(w-(x+y))
```

Target machine with r registers R_0, R_1, ..., R_{r-1} and main memory M

Instruction types:

R_i := M[a]
M[a] := R_i
R_i := R_i *op* M[a]
R_i := R_i *op* R_j
(with address a)

Instruction sequence for $r = 2$:

```
R0 := M[u]
R0 := R0+M[v]
R1 := M[x]
R1 := R1+M[y]
M[t] := R1
R1 := M[w]
R1 := R1-M[t]
R0 := R0-R1
M[z] := R0
```

Shorter sequence:

```
R0 := M[w]
R1 := M[x]
R1 := R1+M[y]
R0 := R0-R1
R1 := M[u]
R1 := R1+M[v]
R1 := R1-R0
M[z] := R1
```

- **Reason:** first variant requires intermediate storage for x+y
- How to compute systematically?
- **Idea:** start with register-intensive subexpressions

Register Optimization

- Let $e = e_1 \; op \; e_2$.
- Assumption: e_i requires r_i registers for evaluation

Register Optimization

- Let $e = e_1 \; op \; e_2$.
- Assumption: e_i requires r_i registers for evaluation
- Evaluation of e:
  - if $r_1 < r_2 \leq r$, then e can be evaluated using r_2 registers:
    1. evaluate e_2 (using r_2 registers)
    2. keep result in 1 register
    3. evaluate e_1 (using $r_1 + 1 \leq r_2$ registers in total)
    4. combine results
  - if $r_2 < r_1 \leq r$, then e can be evaluated using r_1 registers
  - if $r_1 = r_2 < r$, then e can be evaluated using $r_1 + 1$ registers
  - if more than r registers required: use main memory as intermediate storage

Register Optimization

- Let $e = e_1 \; op \; e_2$.
- Assumption: e_i requires r_i registers for evaluation
- Evaluation of e:
  - if $r_1 < r_2 \leq r$, then e can be evaluated using r_2 registers:
    1. evaluate e_2 (using r_2 registers)
    2. keep result in 1 register
    3. evaluate e_1 (using $r_1 + 1 \leq r_2$ registers in total)
    4. combine results
  - if $r_2 < r_1 \leq r$, then e can be evaluated using r_1 registers
  - if $r_1 = r_2 < r$, then e can be evaluated using $r_1 + 1$ registers
  - if more than r registers required: use main memory as intermediate storage
- The corresponding optimization algorithm works in two phases:
  1. Marking phase (computes r_i values)
  2. Generation phase (produces actual code)

  (for details see Wilhelm/Maurer: *Übersetzerbau, 2. Auflage*, Springer, 1997, Sct. 11.4)

The Marking Phase

Algorithm 23.2 (Marking phase)

Input: *expression (with binary operators op and variables x)*

Procedure: *recursively compute*

$$r(x) := \begin{cases} 1 & \text{if } x \text{ is a “left leaf”} \\ 0 & \text{if } x \text{ is a “right leaf”} \\ 1 & \text{if } x \text{ is at the root} \end{cases}$$

$$r(e_1 \ op \ e_2) := \begin{cases} \max\{r(e_1), r(e_2)\} & \text{if } r(e_1) \neq r(e_2) \\ r(e_1) + 1 & \text{if } r(e_1) = r(e_2) \end{cases}$$

Output: *number of required registers r(e)*

The Marking Phase

Algorithm 23.2 (Marking phase)

Input: *expression (with binary operators op and variables x)*

Procedure: *recursively compute*

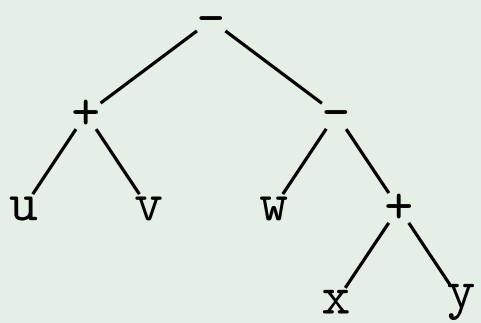
$$r(x) := \begin{cases} 1 & \text{if } x \text{ is a “left leaf”} \\ 0 & \text{if } x \text{ is a “right leaf”} \\ 1 & \text{if } x \text{ is at the root} \end{cases}$$

$$r(e_1 \ op \ e_2) := \begin{cases} \max\{r(e_1), r(e_2)\} & \text{if } r(e_1) \neq r(e_2) \\ r(e_1) + 1 & \text{if } r(e_1) = r(e_2) \end{cases}$$

Output: *number of required registers* $r(e)$

Example 23.3 (cf. Example 23.1)

$e =$ `(u+v)-(w-(x+y))`:



The Marking Phase

Algorithm 23.2 (Marking phase)

Input: *expression (with binary operators op and variables x)*

Procedure: *recursively compute*

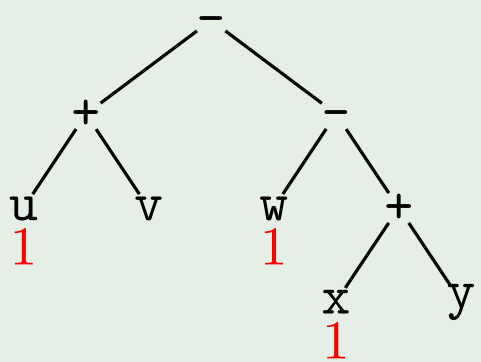
$$r(x) := \begin{cases} 1 & \textit{if } x \textit{ is a ``left leaf''} \\ 0 & \textit{if } x \textit{ is a ``right leaf''} \\ 1 & \textit{if } x \textit{ is at the root} \end{cases}$$

$$r(e_1 \; op \; e_2) := \begin{cases} \max\{r(e_1), r(e_2)\} & \textit{if } r(e_1) \neq r(e_2) \\ r(e_1) + 1 & \textit{if } r(e_1) = r(e_2) \end{cases}$$

Output: *number of required registers* $r(e)$

Example 23.3 (cf. Example 23.1)

$e =$ `(u+v)-(w-(x+y))`:



The Marking Phase

Algorithm 23.2 (Marking phase)

Input: *expression (with binary operators op and variables x)*

Procedure: *recursively compute*

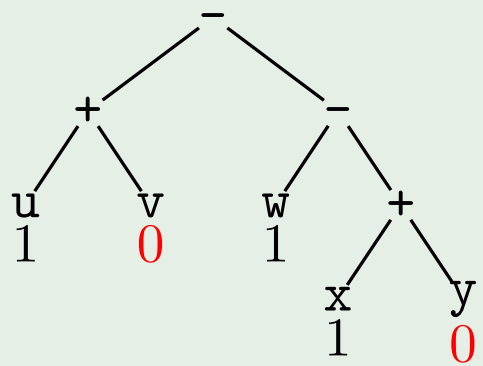
$$r(x) := \begin{cases} 1 & \text{if } x \text{ is a “left leaf”} \\ 0 & \text{if } x \text{ is a “right leaf”} \\ 1 & \text{if } x \text{ is at the root} \end{cases}$$

$$r(e_1 \ op \ e_2) := \begin{cases} \max\{r(e_1), r(e_2)\} & \text{if } r(e_1) \neq r(e_2) \\ r(e_1) + 1 & \text{if } r(e_1) = r(e_2) \end{cases}$$

Output: *number of required registers* $r(e)$

Example 23.3 (cf. Example 23.1)

$e =$ `(u+v)-(w-(x+y))`:



The Marking Phase

Algorithm 23.2 (Marking phase)

Input: *expression (with binary operators op and variables x)*

Procedure: *recursively compute*

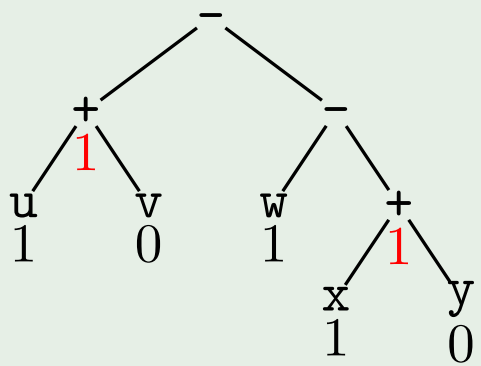
$$r(x) := \begin{cases} 1 & \text{if } x \text{ is a “left leaf”} \\ 0 & \text{if } x \text{ is a “right leaf”} \\ 1 & \text{if } x \text{ is at the root} \end{cases}$$

$$r(e_1 \ op \ e_2) := \begin{cases} \max\{r(e_1), r(e_2)\} & \text{if } r(e_1) \neq r(e_2) \\ r(e_1) + 1 & \text{if } r(e_1) = r(e_2) \end{cases}$$

Output: *number of required registers* $r(e)$

Example 23.3 (cf. Example 23.1)

$e =$ `(u+v)-(w-(x+y))`:



The Marking Phase

Algorithm 23.2 (Marking phase)

Input: *expression (with binary operators op and variables x)*

Procedure: *recursively compute*

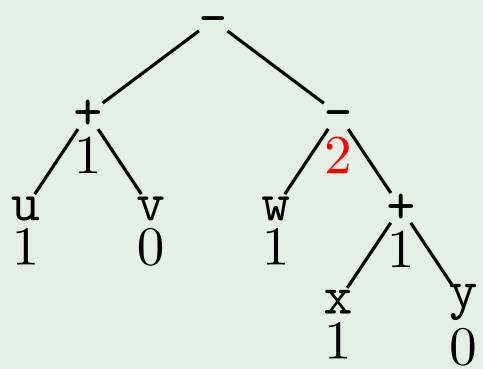
$$r(x) := \begin{cases} 1 & \text{if } x \text{ is a “left leaf”} \\ 0 & \text{if } x \text{ is a “right leaf”} \\ 1 & \text{if } x \text{ is at the root} \end{cases}$$

$$r(e_1 \ op \ e_2) := \begin{cases} \max\{r(e_1), r(e_2)\} & \text{if } r(e_1) \neq r(e_2) \\ r(e_1) + 1 & \text{if } r(e_1) = r(e_2) \end{cases}$$

Output: *number of required registers* $r(e)$

Example 23.3 (cf. Example 23.1)

$e =$ `(u+v)-(w-(x+y))`:



The Marking Phase

Algorithm 23.2 (Marking phase)

Input: *expression (with binary operators op and variables x)*

Procedure: *recursively compute*

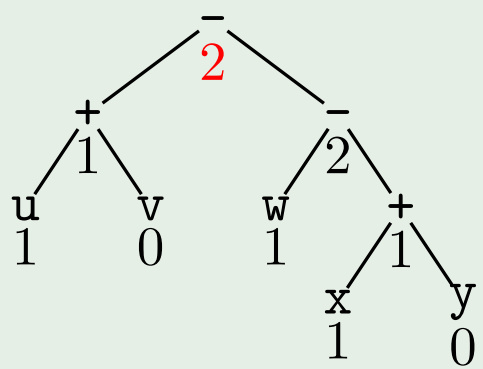
$$r(x) := \begin{cases} 1 & \text{if } x \text{ is a “left leaf”} \\ 0 & \text{if } x \text{ is a “right leaf”} \\ 1 & \text{if } x \text{ is at the root} \end{cases}$$

$$r(e_1 \ op \ e_2) := \begin{cases} \max\{r(e_1), r(e_2)\} & \text{if } r(e_1) \neq r(e_2) \\ r(e_1) + 1 & \text{if } r(e_1) = r(e_2) \end{cases}$$

Output: *number of required registers* $r(e)$

Example 23.3 (cf. Example 23.1)

$e =$ `(u+v)-(w-(x+y))`:



The Generation Phase I

- **Goal:** generate optimal (= shortest) code for evaluating expression e with register requirement $r(e)$

The Generation Phase I

- **Goal:** generate optimal (= shortest) code for evaluating expression e with register requirement $r(e)$
- Data structures used in Algorithm 23.4:

  RS: stack of available registers (initially: all registers; never empty)

  CS: stack of available main memory cells

The Generation Phase I

- **Goal:** generate optimal (= shortest) code for evaluating expression e with register requirement $r(e)$
- Data structures used in Algorithm 23.4:

  RS: stack of available registers (initially: all registers; never empty)

  CS: stack of available main memory cells

- Auxiliary procedures used in Algorithm 23.4:

  output: outputs the argument as code

  top: returns the topmost entry of a stack S (leaving S unchanged)

  pop: removes and returns the topmost entry of a stack

  push: puts an element onto a stack

  exchange: exchanges the two topmost elements of a stack

The Generation Phase II

Algorithm 23.4 (Generation phase)

Input: *expression e, annotated with register requirement $r(e)$*

Variables: *RS: stack of registers;*
CS: stack of memory cells;
R: register; C: memory cell;

Procedure: *recursive execution of procedure $code(e)$, defined by $code(e) :=$*

if $e = x$, $r(x) = 1$: % left leaf
  $output(top(RS)$:= M[x]$)$

if $e = e_1$ op y, $r(y) = 0$: % right leaf
  $code(e_1)$;
  $output(top(RS)$:=$top(RS)$ op M[y]$)$

if $e = e_1$ op e_2, $r(e_1) < r(e_2)$, $r(e_1) < r$:
  $exchange(RS)$;
  $code(e_2)$;
  $R := pop(RS)$;
  $code(e_1)$;
  $output(top(RS)$:=$top(RS)$ op $R)$;
  $push(RS, R)$;
  $exchange(RS)$

if $e = e_1$ op e_2, $r(e_1) \geq r(e_2)$, $r(e_2) < r$:
  $code(e_1)$;
  $R := pop(RS)$;
  $code(e_2)$;
  $output(R$:=R op $top(RS))$;
  $push(RS, R)$

if $e = e_1$ op e_2, $r(e_1) \geq r$, $r(e_2) \geq r$:
  $code(e_2)$;
  $C := pop(CS)$;
  $output($M[C]:=$top(RS))$;
  $code(e_1)$;
  $output(top(RS)$:=$top(RS)$ op M[C]$)$;
  $push(CS, C)$

Output: *optimal (= shortest) code for evaluating e*

The Generation Phase III

- Invariants of Algorithm 23.4:
  - after executing $code(e)$, both RS and CS have their original values
  - after executing the machine code produced by $code(e)$, the value of e is stored in the top register of RS

The Generation Phase III

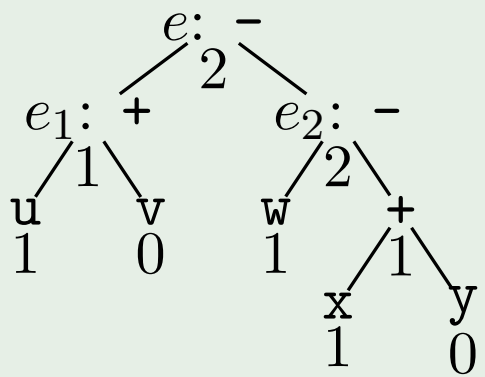
- Invariants of Algorithm 23.4:
  - after executing $code(e)$, both RS and CS have their original values
  - after executing the machine code produced by $code(e)$, the value of e is stored in the top register of RS
- Shortcoming of Algorithm 23.4: multiple evaluation of common subexpressions

  ($\Longrightarrow$ dynamic programming, graph coloring, ...)

The Generation Phase III

- Invariants of Algorithm 23.4:
  - after executing $code(e)$, both RS and CS have their original values
  - after executing the machine code produced by $code(e)$, the value of e is stored in the top register of RS
- Shortcoming of Algorithm 23.4: multiple evaluation of common subexpressions
  ($\Longrightarrow$ dynamic programming, graph coloring, ...)

Example 23.5 (cf. Example 23.3)



(on the board)

Outline

1 Generation of Machine Code

2 Wrap-Up

Omitted

- Code optimization
- Translation of higher-level constructs (modules, classes)
- Translation of non-procedural languages
  - object-oriented (polymorphism, dynamic dispatch)
  - functional (higher-order functions, typechecking)
  - logic (unification, backtracking)
- Bootstrapping

Outlook

Winter semester 2008/09:

- *Introduction to Model Checking* [Katoen; V4Ü2]
- *Semantics and Verification of Software* [Noll; V4Ü2]
- [Seminar *Timed Automata*]

Course Evaluation

Discussion:

- – More examples
- – Lecture sometimes too fast
- – Slides sometimes to full
- – Some handouts inappropriate for printing
- – Curtain in AH 2
- \+ Repetition in beginning of lecture