Compiler Construction

Lecture 19: Code Generation V (Compiler Backend)

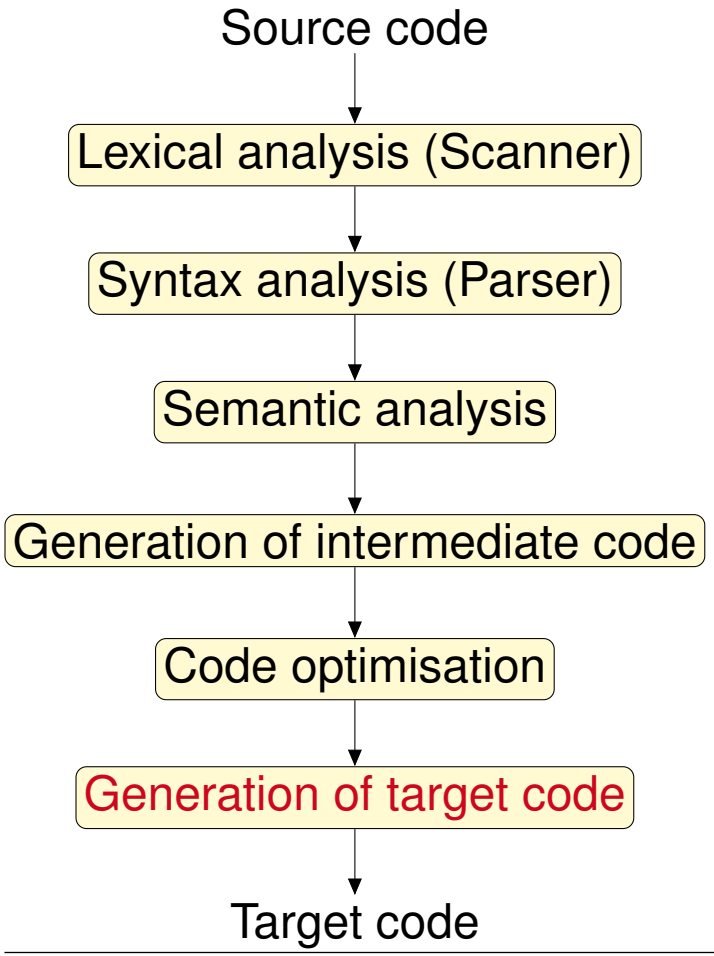
Summer Semester 2017

Thomas Noll
Software Modeling and Verification Group
RWTH Aachen University

https://moves.rwth-aachen.de/teaching/ss-17/cc/

Outline of Lecture 19

Conceptual Structure of a Compiler



The Compiler Backend

Final step: translation of (optimised) abstract machine code into “real” machine code (possibly followed by assembling phase)

The Compiler Backend

Final step: translation of (optimised) abstract machine code into “real” machine code
(possibly followed by assembling phase)

Goal: runtime and storage efficiency

- fast backend
- fast and compact code
- low memory requirements for and efficient access to data

The Compiler Backend

Final step: translation of (optimised) abstract machine code into “real” machine code (possibly followed by assembling phase)

Goal: runtime and storage efficiency

- fast backend
- fast and compact code
- low memory requirements for and efficient access to 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 (optimised) abstract machine code into “real” machine code (possibly followed by assembling phase)

Goal: runtime and storage efficiency

- fast backend
- fast and compact code
- low memory requirements for and efficient access to 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 (optimised) abstract machine code into “real” machine code (possibly followed by assembling phase)

Goal: runtime and storage efficiency

- fast backend
- fast and compact code
- low memory requirements for and efficient access to 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

Instructions: select adequately (number/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 (array indexing, ...)
   - 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 scheduling (placement): increase level of parallelism and/or pipelining by smart ordering of instructions

Code Generation Phases

1. Register allocation: registers used for
   - values of (frequently used) variables and intermediate results
   - computing memory addresses (array indexing, ...)
   - 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 scheduling (placement): increase level of parallelism and/or pipelining by smart ordering of instructions

Outline of Lecture 19

Register Allocation

Example 19.1

Assignment:

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

Register Allocation

Example 19.1

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 $\mathtt{M}$

Register Allocation

Example 19.1

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 $\mathtt{M}$

Instruction types:

$\mathtt{R}_i := \mathtt{M}[a]$

$\mathtt{M}[a] := \mathtt{R}_i$

$\mathtt{R}_i := \mathtt{R}_i \ \mathit{op} \ \mathtt{M}[a]$

$\mathtt{R}_i := \mathtt{R}_i \ \mathit{op} \ \mathtt{R}_j$

(with address a)

Register Allocation

Example 19.1

Assignment:

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

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

Instruction types:

$\mathtt{R}_i := \mathtt{M}[a]$

$\mathtt{M}[a] := \mathtt{R}_i$

$\mathtt{R}_i := \mathtt{R}_i \ op \ \mathtt{M}[a]$

$\mathtt{R}_i := \mathtt{R}_i \ op \ \mathtt{R}_j$

(with address a)

Instruction sequence ($r = 2$):

$\mathtt{R}_0 := \mathtt{M}[\mathtt{u}]$

$\mathtt{R}_0 := \mathtt{R}_0 + \mathtt{M}[\mathtt{v}]$

$\mathtt{R}_1 := \mathtt{M}[\mathtt{x}]$

$\mathtt{R}_1 := \mathtt{R}_1 + \mathtt{M}[\mathtt{y}]$

$\mathtt{M}[\mathtt{t}] := \mathtt{R}_1$

$\mathtt{R}_1 := \mathtt{M}[\mathtt{w}]$

$\mathtt{R}_1 := \mathtt{R}_1 - \mathtt{M}[\mathtt{t}]$

$\mathtt{R}_0 := \mathtt{R}_0 - \mathtt{R}_1$

$\mathtt{M}[\mathtt{z}] := \mathtt{R}_0$

Register Allocation

Example 19.1

Assignment:

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

Target machine with r registers $R_0, R_1, \ldots, R_{r-1}$ 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 ($r = 2$):

```
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
```

Register Allocation

Example 19.1

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 ($r = 2$):

```
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:** 2nd variant avoids intermediate storage `t` for `x+y`

Register Allocation

Example 19.1

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 ($r = 2$):

```
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:** 2nd variant avoids intermediate storage `t` for `x+y`
- How to compute systematically?

Register Allocation

Example 19.1

Assignment:

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

Target machine with r registers $R_0, R_1, \ldots, R_{r-1}$ 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 ($r = 2$):

```
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:** 2nd variant avoids intermediate storage `t` for `x+y`
- How to compute systematically?
- **Idea:** start with register-intensive subexpressions

Register Optimisation

- Let $e = e_1 \; op \; e_2$.
- Assumption: e_i requires r_i registers for evaluation, r available in total

Register Optimisation

- Let $e = e_1 \; op \; e_2$.
- Assumption: e_i requires r_i registers for evaluation, r available in total
- 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 (symmetrically)
  - 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 Optimisation

- Let $e = e_1 \; op \; e_2$.
- Assumption: e_i requires r_i registers for evaluation, r available in total
- 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 (symmetrically)
  - 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 optimisation algorithm works in two phases:
  1. Marking phase (computes r_i values)
  2. Generation phase (produces actual code)

  (cf. Wilhelm/Maurer: *Übersetzerbau, 2. Auflage*, Springer, 1997, Sct. 12.4)

The Marking Phase

Algorithm 19.2 (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 19.2 (Marking phase)

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

Procedure: *recursively compute*

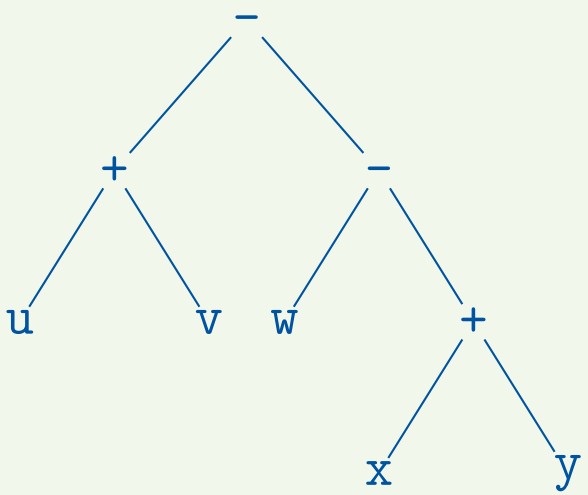
$$r(x) := \begin{cases} 1 & \text{if } x \text{ is a “left leaf”} \\ 0 & \text{if } x \text{ is a “right leaf”} \\ 1 & \text{if } x \text{ is at the root} \end{cases}$$

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

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

Example 19.3 (cf. Ex. 19.1)

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



The Marking Phase

Algorithm 19.2 (Marking phase)

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

Procedure: *recursively compute*

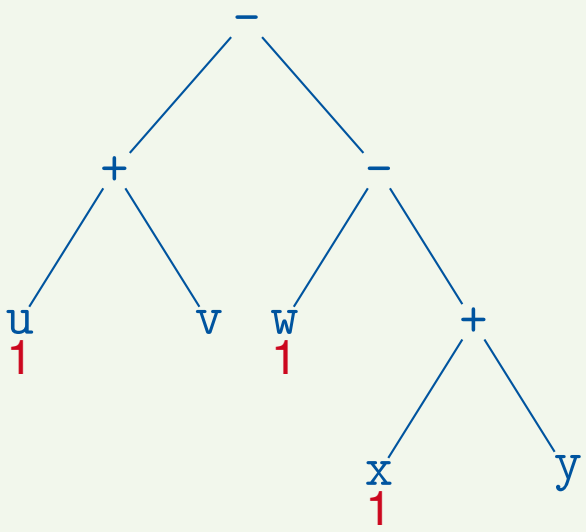
$$r(x) := \begin{cases} 1 & \text{if } x \text{ is a “left leaf”} \\ 0 & \text{if } x \text{ is a “right leaf”} \\ 1 & \text{if } x \text{ is at the root} \end{cases}$$

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

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

Example 19.3 (cf. Ex. 19.1)

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



The Marking Phase

Algorithm 19.2 (Marking phase)

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

Procedure: *recursively compute*

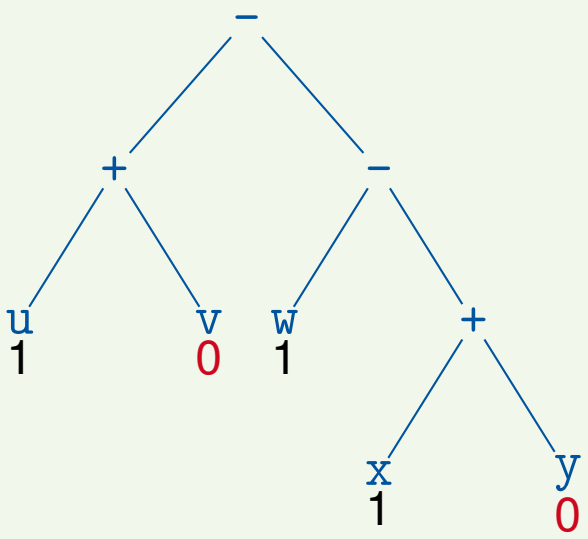
$$r(x) := \begin{cases} 1 & \text{if } x \text{ is a “left leaf”} \\ 0 & \text{if } x \text{ is a “right leaf”} \\ 1 & \text{if } x \text{ is at the root} \end{cases}$$

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

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

Example 19.3 (cf. Ex. 19.1)

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



The Marking Phase

Algorithm 19.2 (Marking phase)

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

Procedure: *recursively compute*

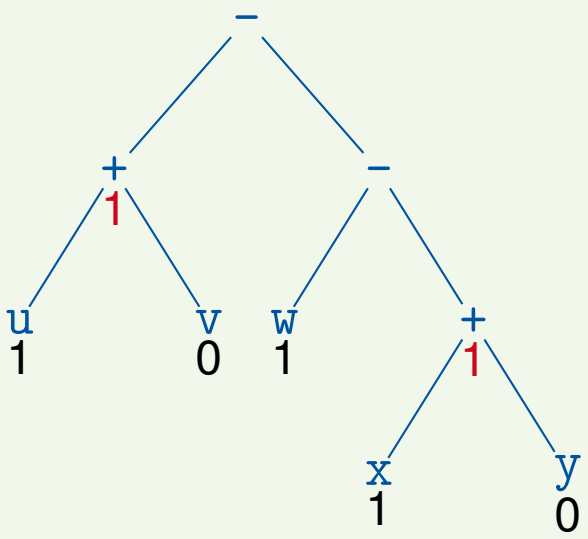
$$r(x) := \begin{cases} 1 & \text{if } x \text{ is a “left leaf”} \\ 0 & \text{if } x \text{ is a “right leaf”} \\ 1 & \text{if } x \text{ is at the root} \end{cases}$$

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

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

Example 19.3 (cf. Ex. 19.1)

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



The Marking Phase

Algorithm 19.2 (Marking phase)

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

Procedure: *recursively compute*

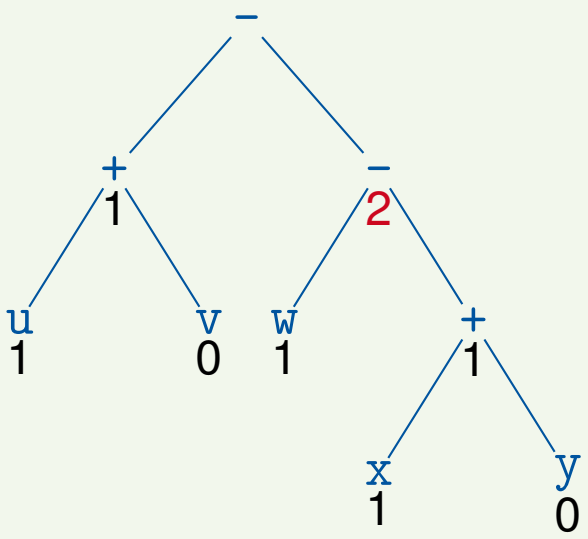
$$r(x) := \begin{cases} 1 & \text{if } x \text{ is a “left leaf”} \\ 0 & \text{if } x \text{ is a “right leaf”} \\ 1 & \text{if } x \text{ is at the root} \end{cases}$$

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

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

Example 19.3 (cf. Ex. 19.1)

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



The Marking Phase

Algorithm 19.2 (Marking phase)

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

Procedure: *recursively compute*

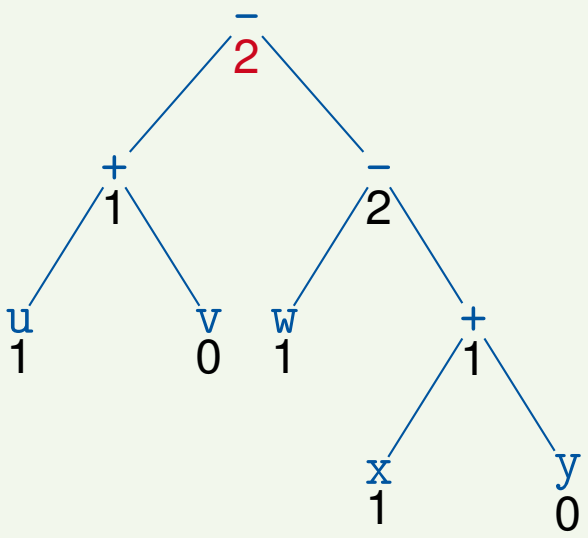
$$r(x) := \begin{cases} 1 & \text{if } x \text{ is a “left leaf”} \\ 0 & \text{if } x \text{ is a “right leaf”} \\ 1 & \text{if } x \text{ is at the root} \end{cases}$$

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

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

Example 19.3 (cf. Ex. 19.1)

$e =$ `(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 19.4:

  RS: stack of available registers (initially: all r 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 19.4:
  - RS: stack of available registers (initially: all r registers; never empty)
  - CS: stack of available main memory cells
- Auxiliary procedures used in Algorithm 19.4:
  - *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 (for preserving argument order in binary operations)

The Generation Phase II

Algorithm 19.4 (Generation phase)

Input: total register numer r; 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 $code(e) :=$

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

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

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

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

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

Output: optimal (= shortest) code for evaluating e

The Generation Phase III

- Invariants of Algorithm 19.4:
  - after executing $code(e)$, both RS and CS have their original values
  - after executing $code(e)$, value of e is stored in topmost register of RS

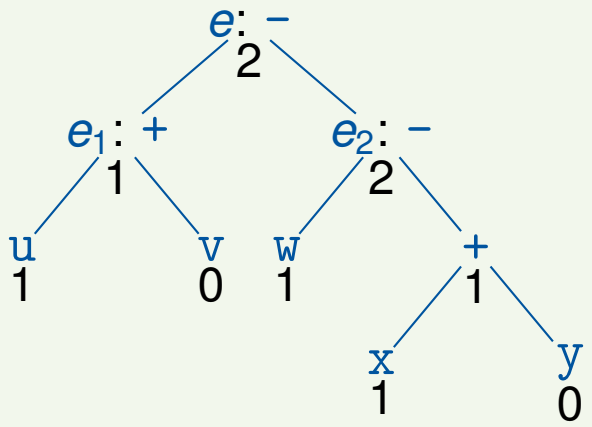
The Generation Phase III

- Invariants of Algorithm 19.4:
  - after executing $code(e)$, both RS and CS have their original values
  - after executing $code(e)$, value of e is stored in topmost register of RS
- Shortcoming of Algorithm 19.4: multiple evaluation of common subexpressions
  ($\Longrightarrow$ dynamic programming [Wilhelm/Maurer])

The Generation Phase III

- Invariants of Algorithm 19.4:
  - after executing $code(e)$, both RS and CS have their original values
  - after executing $code(e)$, value of e is stored in topmost register of RS
- Shortcoming of Algorithm 19.4: multiple evaluation of common subexpressions
  ($\Longrightarrow$ dynamic programming [Wilhelm/Maurer])

Example 19.5 (cf. Example 19.3)



Application of Algorithm 19.4:
on the board

Register Allocation by Graph Colouring I

- Algorithm 19.4: register allocation for single expressions
- Required: global allocation within program/procedure body
- Approach: graph colouring

Register Allocation by Graph Colouring I

- Algorithm 19.4: register allocation for single expressions
- Required: global allocation within program/procedure body
- Approach: graph colouring

Register allocation by graph colouring

1. Use unbounded number of symbolic registers for storing intermediate values

Register Allocation by Graph Colouring I

- Algorithm 19.4: register allocation for single expressions
- Required: global allocation within program/procedure body
- Approach: graph colouring

Register allocation by graph colouring

1. Use unbounded number of symbolic registers for storing intermediate values
2. Consider life span of symbolic registers: *r* is live at program point *p* if
   - there is a path to *p* on which *r* is set and
   - there is a path from *p* on which *r* is read before being set

Register Allocation by Graph Colouring I

- Algorithm 19.4: register allocation for single expressions
- Required: global allocation within program/procedure body
- Approach: graph colouring

Register allocation by graph colouring

1. Use unbounded number of symbolic registers for storing intermediate values
2. Consider life span of symbolic registers: r is live at program point p if
   - there is a path to p on which r is set and
   - there is a path from p on which r is read before being set
3. Life span of r = program points where r is live

Register Allocation by Graph Colouring I

- Algorithm 19.4: register allocation for single expressions
- Required: global allocation within program/procedure body
- Approach: graph colouring

Register allocation by graph colouring

1. Use unbounded number of symbolic registers for storing intermediate values
2. Consider life span of symbolic registers: r is live at program point p if
   - there is a path to p on which r is set and
   - there is a path from p on which r is read before being set
3. Life span of r = program points where r is live
4. Two registers are in interference their life spans intersect

Register Allocation by Graph Colouring I

- Algorithm 19.4: register allocation for single expressions
- Required: global allocation within program/procedure body
- Approach: graph colouring

Register allocation by graph colouring

1. Use unbounded number of symbolic registers for storing intermediate values
2. Consider life span of symbolic registers: r is live at program point p if
   - there is a path to p on which r is set and
   - there is a path from p on which r is read before being set
3. Life span of r = program points where r is live
4. Two registers are in interference their life spans intersect
5. Yields register interference graph (nodes = life spans, edges = interferences)

Register Allocation by Graph Colouring I

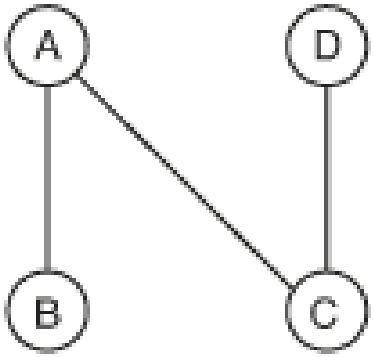
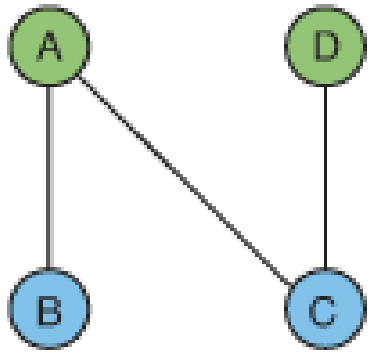
- Algorithm 19.4: register allocation for single expressions
- Required: global allocation within program/procedure body
- Approach: graph colouring

Register allocation by graph colouring

1. Use unbounded number of symbolic registers for storing intermediate values
2. Consider life span of symbolic registers: r is live at program point p if
   - there is a path to p on which r is set and
   - there is a path from p on which r is read before being set
3. Life span of r = program points where r is live
4. Two registers are in interference their life spans intersect
5. Yields register interference graph (nodes = life spans, edges = interferences)
6. Program executable with k real registers iff interference graph k-colourable

Register Allocation by Graph Colouring II

Example 19.6

Code Sequence	Interference Graph	Colored Graph	Final Allocation
A = … B = … … B … C = … … A … D = … … D … … C …			R1 = … R2 = … … R2 … R2 = … … R1 … R1 = … … R1 … … R2 …

Outline of Lecture 19

Outlook

Further Topics in Compiler Construction

- Translation of higher-level constructs (modules, classes, ...)
- Translation of non-procedural languages
  - object-oriented (inheritance, polymorphism, dynamic dispatch, ..)
  - functional (higher-order functions, type checking/inference, lazy evaluation, ...)
  - logical (unification, resolution, backtracking, ...)
- Symbol-table handling
- Error handling
- Bootstrapping
- Code optimisation (on source/intermediate/machine code level)
  - *Static Program Analysis* in SS 2018
- Compiler verification
  - *Semantics and Verification of Software* in WS 2017/18

Outlook

Exams & Seminars

Exams

1. Friday, 4 August, 14:00-16:00, AH 1/2/3 (location subject to change)
2. Tuesday, 19 September, 11:00-13:00, AH 2/3 (location subject to change)

Outlook

Exams & Seminars

Exams

1. Friday, 4 August, 14:00-16:00, AH 1/2/3 (location subject to change)
2. Tuesday, 19 September, 11:00-13:00, AH 2/3 (location subject to change)

WS 2017/18: *Programming Language Design and Implementation* [Noll et al.]

- Advanced parsing techniques
- Static program analysis (pointers, security, performance bugs, ...)
- Automation and verification of compilation

Companion seminar: *Foundations of Probabilistic Programming* [Katoen et al.]

Outlook

Lectures

WS 2017/18: *Semantics and Verification of Software* [Noll]

- Operational semantics
- Denotational semantics
- Axiomatic semantics
- Semantic equivalence
- Compiler correctness

Outlook

Lectures

WS 2017/18: *Semantics and Verification of Software* [Noll]

- Operational semantics
- Denotational semantics
- Axiomatic semantics
- Semantic equivalence
- Compiler correctness

WS 2017/18: *Concurrency Theory* [Katoen, Noll]

- Process algebras (CCS, π-Calculus)
- Hennessy-Milner logic
- Strong and weak bisimulation
- Petri nets

Outline of Lecture 19

Profillinie

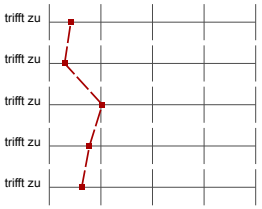
Teilbereich: Informatik
Name der/des Lehrenden: apl. Prof. Dr.rer.nat. Thomas Noll
Titel der Lehrveranstaltung: (Name der Umfrage) Compilerbau (17ss-38290)

Verwendete Werte in der Profillinie: Mittelwert

Allgemein

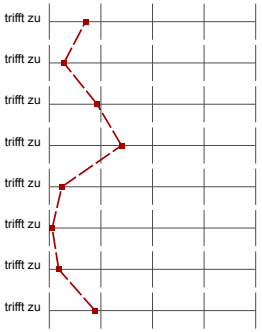
	trifft zu	trifft nicht zu	n	mw	md	s
1.6 Die Veranstaltung interessiert mich.			n=34	mw=2,1	md=2,0	s=0,9

Konzept der Vorlesung



	trifft zu	trifft nicht zu	n	mw	md	s
2.1 Die Lernziele der Vorlesung sind definiert.			n=36	mw=1,4	md=1,0	s=0,6
2.2 Die Vorlesung hat eine klar erkennbare Struktur.			n=37	mw=1,3	md=1,0	s=0,5
2.3 Die zur Verfügung gestellten Materialien sind hilfreich.			n=37	mw=2,0	md=2,0	s=1,1
2.4 Die ausgewählten Beispiele sind hilfreich.			n=36	mw=1,8	md=2,0	s=0,8
2.5 Es werden Zusammenfassungen an sinnvollen Stellen gemacht.			n=35	mw=1,6	md=1,0	s=1,0

Vermittlung und Verhalten



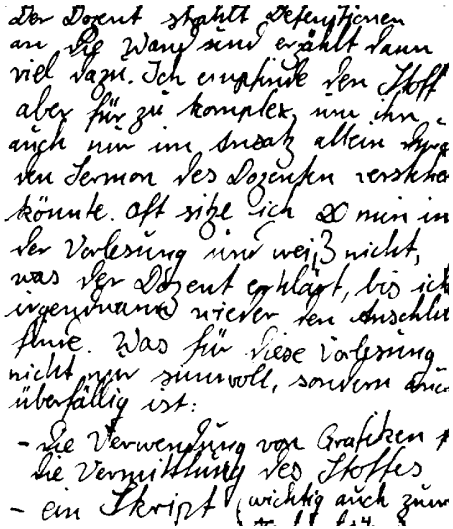
	trifft zu	trifft nicht zu	n	mw	md	s
3.1 ... erklärt den Stoff verständlich.			n=37	mw=1,7	md=1,0	s=1,0
3.2. ... geht auf Verständnisfragen ein.			n=32	mw=1,3	md=1,0	s=0,7
3.3 ... berücksichtigt unterschiedliche Kenntnisstände der Studierenden.			n=27	mw=1,9	md=2,0	s=0,9
3.4 ... schafft es, mich für den Vorlesungsstoff zu begeistern.			n=35	mw=2,4	md=2,0	s=0,9
3.5 ... spricht angemessen laut und deutlich.			n=37	mw=1,2	md=1,0	s=0,5
3.6 ... ist gut vorbereitet.			n=36	mw=1,1	md=1,0	s=0,2
3.7 ... ist außerhalb der Vorlesung ansprechbar.			n=11	mw=1,2	md=1,0	s=0,4
3.8 ... setzt Medien ein, die zum Verständnis beitragen.			n=35	mw=1,9	md=2,0	s=1,0

Rahmenbedingungen

	trifft zu	trifft nicht zu	n	mw	md	s
4.1 Der zeitliche Rahmen der Vorlesung wird eingehalten.			n=37	mw=1,1	md=1,0	s=0,2

Zusammenfassung am
Anfang der Stunde

5.2 Was hat Ihnen an der Vorlesung **nicht** gefallen?


Der Dozent spricht [illegible] an die Wand und erzählt dann viel dazu. Ich empfinde den Stoff aber für zu komplex, um ihn auch nur im Ansatz allein durch den Lernen des Dozenten verstehen könnte. Oft sitze ich da nun in der Vorlesung und weiß nicht, was der Dozent erklärt, bis ich irgendwann wieder den Anschluss finde. Was für diese Vorlesung nicht gut zum voll, sondern auch überfällig ist:
- die Verwendung von Grafiken [illegible] die Vermittlung des Stoffes
- ein Skript (wichtig auch zum

Der Professor neigt dazu, sehr viele Klammern aufzumachen beim sprechen. Daher ist es manchmal schwierig den ursprünglichen Gedanken im Blick zu behalten.

Folien teilweise überfüllt (Optimierung)

- Folien teils unübersichtlich
- Folien im Handout teils kaputt

-Folien sehr voll

Ich interessiere mich mehr
für Optimierung / Generierung
als für etwas Kurz

- Für meinen Geschmack etwas sehr formell :)

Folien (zu viel Text,
komplexe Notation)

Folien etwas überladen,
häufig aber vermutlich nicht gut
zu vermeiden