



Compiler Construction

Lecture 19: Code Generation V (Compiler Backend)

Winter Semester 2018/19

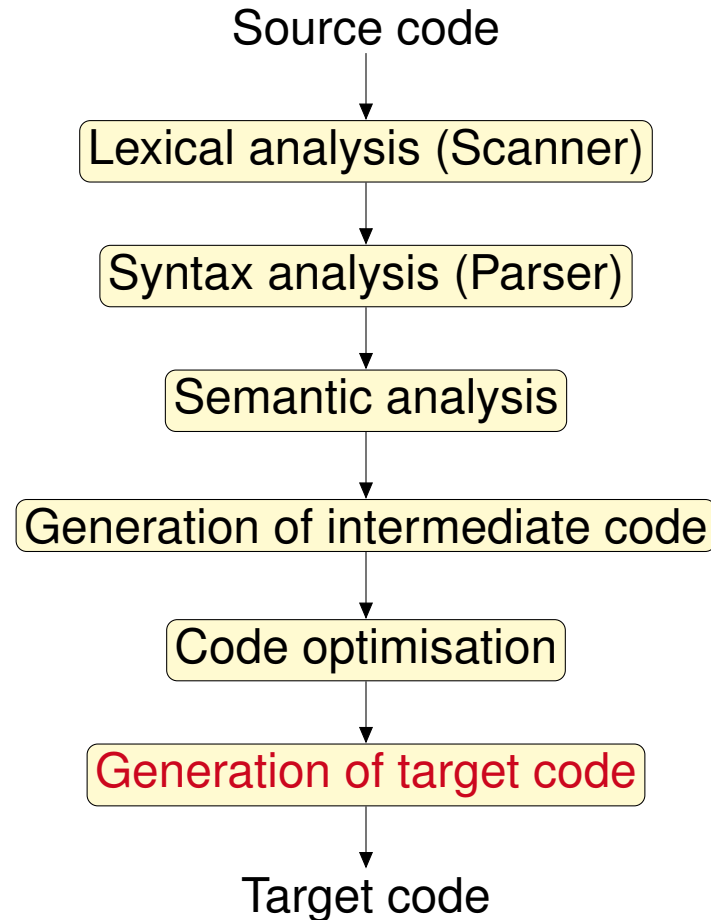
Thomas Noll

Software Modeling and Verification Group

RWTH Aachen University

<https://moves.rwth-aachen.de/teaching/ws-1819/cc/>

Conceptual Structure of a Compiler



The Compiler Backend

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

Register Allocation

Register Allocation

Example 19.1

Assignment:

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

Target machine with

r registers R_0, R_1, \dots, R_{r-1}
and main memory M

Instruction types:

$$R_j := M[a]$$
$$M[a] := R_j$$
$$R_j := R_j \text{ op } M[a]$$
$$R_j := R_j \text{ op } R_j$$

(with address a)

Instruction sequence ($r = 2$):

$$R_0 := M[u]$$
$$R_0 := R_0 + M[v]$$
$$R_1 := M[x]$$
$$R_1 := R_1 + M[y]$$
$$M[t] := R_1$$
$$R_1 := M[w]$$
$$R_1 := R_1 - M[t]$$
$$R_0 := R_0 - R_1$$
$$M[z] := R_0$$

Shorter sequence:

$$R_0 := M[w]$$
$$R_1 := M[x]$$
$$R_1 := R_1 + M[y]$$
$$R_0 := R_0 - R_1$$
$$R_1 := M[u]$$
$$R_1 := R_1 + M[v]$$
$$R_1 := R_1 - R_0$$
$$M[z] := R_1$$

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

Register Allocation

Register Optimisation

- Let $e = e_1 \text{ op } e_2$.
- 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)

Register Allocation

The Marking Phase

Algorithm 19.2 (Marking phase)

Input: expression

(with binary operators op and variables x)

Procedure: recursively compute

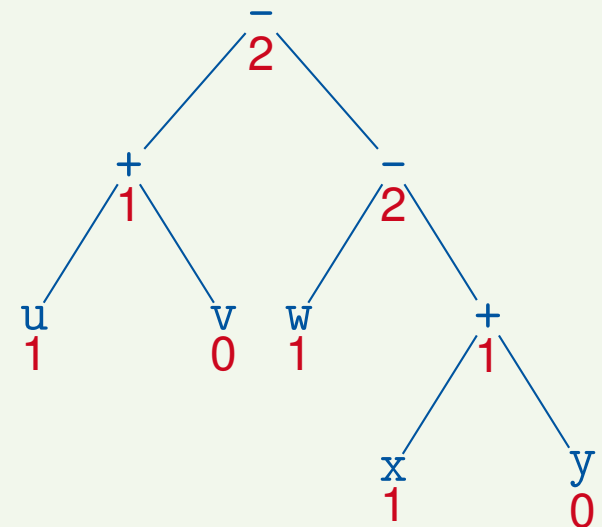
$$r(x) := \begin{cases} 1 & \text{if } x \text{ is a "left leaf"} \\ 0 & \text{if } x \text{ is a "right leaf"} \\ 1 & \text{if } x \text{ is at the root} \end{cases}$$

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

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

Example 19.3 (cf. Ex. 19.1)

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



Register Allocation

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)

Register Allocation

The Generation Phase II

Algorithm 19.4 (Generation phase)

Input: total register number 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
 $output(top(RS) := M[x])$
- (2) if $e = e_1 op y$, $r(y) = 0$: % right leaf
 $code(e_1)$;
 $output(top(RS) := top(RS) op M[y])$
- (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 $e = e_1 op e_2$, $r(e_1) \geq r$, $r(e_2) \geq r$:
 $code(e_2)$;
 $C := pop(CS)$;
 $output(M[C] := top(RS))$;
 $code(e_1)$;
 $output(top(RS) := top(RS) op M[C])$;
 $push(CS, C)$

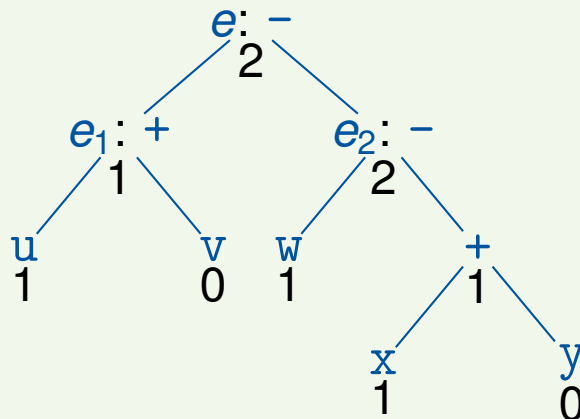
Output: optimal (= shortest) code for evaluating e

Register Allocation

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** (\implies dynamic programming [Wilhelm/Maurer])

Example 19.5 (cf. Example 19.3)



Application of Algorithm 19.4:
on the board

Register Allocation

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** if 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

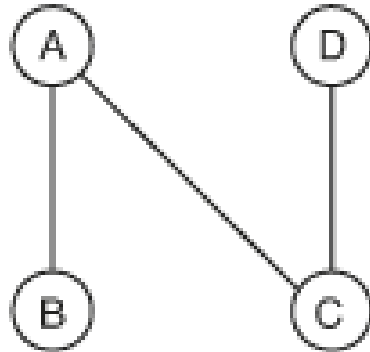
Register Allocation by Graph Colouring II

Example 19.6

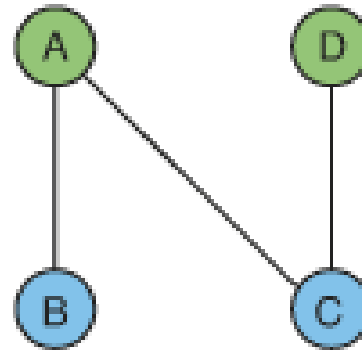
Code Sequence

```
A = ...  
B = ...  
... B ...  
C = ...  
... A ...  
D = ...  
... D ...  
... C ...
```

Interference Graph



Colored Graph



Final Allocation

```
R1 = ...  
R2 = ...  
... R2 ...  
R2 = ...  
... R1 ...  
R1 = ...  
... R1 ...  
... R2 ...
```

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** (detection & recovery)
- **Bootstrapping**
- Compiler **verification**
 - *Semantics and Verification of Software* in Summer 2019
- Code **optimisation** (on source/intermediate/machine code level)
 - *Static Program Analysis* in Winter 2019/20

Exams

Exams

1. Thursday, 21 February, 10:00–12:00, AH 1/4/5, Aula 2 (location subject to change)
2. Monday, 25 March, 10:00–12:00, AH 4/5 (location subject to change)

Courses & Seminars

Summer 2019: Course *Semantics and Verification of Software* [Noll]

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

Summer 2019: Seminar *Program Synthesis* [Noll et al.]

Goal: automatically finding a program (of the underlying programming language) that satisfies a user-given specification

- Enumerative search
- Constraint Solving
- Stochastic Search
- Programming by Examples
- Applications

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

Nicht so spezialles,
aber insgesamt gut :)

5.2 What did you **dislike** about the lecture?

Die Vorlesung ist in der PO als 3SW
gekennzeichnet, wurde aber verständlich gehalten.
Das finde ich vom Inhalt ok, aber
vielleicht wird mit dem Modul
"offiziell" auf 4 SW geändert

Das Beispiel für EPL hat
ausführlicher sein können. Insbeson-
dere wie die Stacks aussehen nach
Operationen

The explanation for
cycle checking was
quite as cryptic in the
handout.

the lecture is super notation-heavy. Maybe something as simple as a "notation cheat sheet" with some examples could help a lot.

- Manchmal eine lange, langweilige Abfolge von Notationen und Definitionen, bis man zu einem Ergebnis kommt

- Slides would be easier to understand with (additional) informal explanations

Aus irgend einem Grund,
den ich per se jetzt
nicht feststellen kann,
lässt meine Konzentration
in CC überproportional
schnell nach!

Sometimes too much focus
on detail

More

You can't see on the website when lecture
slides are updated

▷ The slides (especially definitions)
are sometimes not very clear
(For example: implicit notations)

- I would like to have more
examples in the slides

- slides are sometimes too 'mathematical'; some intuition / more examples on the slides would be nice

It is often difficult to get an ~~overview~~ overview about all definitions.

↳ sometimes not easy to understand

slides are sometimes hard
to understand when re-looking
at them without explanation

1. Don't provide pseudo code
for example for LR(0), LR(1),
SLR algorithms, or tell us
~~how~~ the implementation
idea of them (using stack...)
2. ~~by~~ by applying which data structure
we can implement symbol table
etc..
3. some symbols not uniquely defined.

~~Ente~~
some slides are
overloaded with content
(e.g. examples).

Es wird den Dozenten
nicht erlaubt seine
Vorlesung in Ruhe zu
beenden. Sobald
das Ende absehbar
ist, ~~stört~~^{ist} sehr viel
Geräusch hörbar.

Den Dozenten könnte
ab und zu um Ruhe
beten.