PHP

Als Stack- und Registerinterpreter

Alexander Mandl, 1429186

June 26, 2018

Aufgabenstellung

- Interpreter für eine (sehr minimierte) Version von PHP
- Stack-Version und Register-Version
- Vergleich der Interpreter
- Analyse der Ergebnisse
- Ergebnis: 4 Versionen um die Verbesserungen zu vergleichen

Testfälle

fib.php:

- Rekursive Berechnung der ersten 40 Fibonacci-Zahlen
- viele Funktionsaufrufe durch Rekursion

prime.php/prime100000.php

- Berechnung der ersten 10.000 bzw. 100.000 Primzahlen
- Prüfung für jede Zahl x: Gibt es eine Zahl zwischen 2 und $\sqrt{x}$, die x teilt.

V1: Design-Entscheidungen (und Fehler)

Compiler

- Geschrieben in GO
- Verwendet goyacc [1] und nex [2]

Interpreter

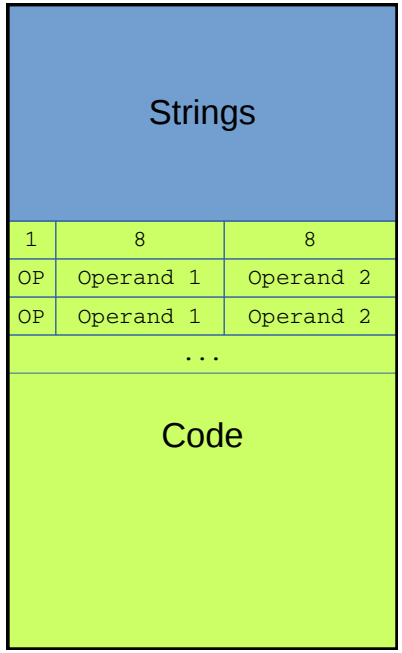
- Geschrieben in GO
- Instructions sind Funktionen in einem Array

[1] https://godoc.org/golang.org/x/tools/cmd/goyacc
[2] http://www-cs-students.stanford.edu/~blynn/nex/

V1: Design-Entscheidungen (und Fehler)

Stackinterpreter

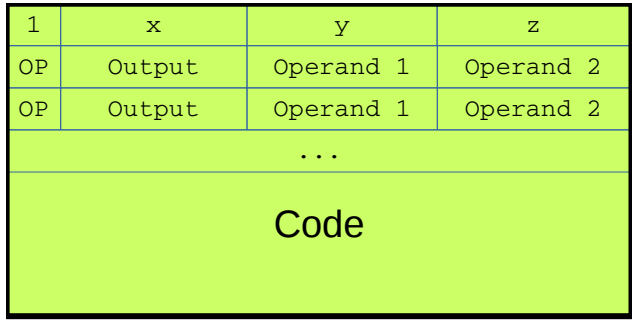
- Stack und Heap als Arrays in globalen Variablen
- 17 byte Instruktionen (fixe Größe)
- Strings und Arrays werden im Heap gespeichert, Literale am Anfang des Heaps kopiert
- Variablen werden am Stack gespeichert
- Der aktuelle Block im Code ({}) wird immer als Stackframe behandelt
- Funktionen liegen dort wo sie im Input stehen (und werden mit Jumps übersprungen)
- relative Sprünge



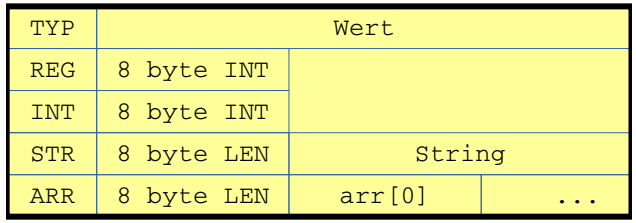
V1: Design-Entscheidungen (und Fehler)

Registerinterpreter

- Stack und Heap als Arrays in globalen Variablen
- Instruktionen haben Variable Größe
- Strings werden als Teil der Instruktion gespeichert
- Anzahl an Registern ist unlimitiert; Register sind Teil des Stacks
- Jede Variable hat ein Register
- Nur die aktuelle Funktion wird als Stackframe behandelt
- Funktionen liegen am Ende der Code-Sektion
- absolute Sprünge



Operand:



Beispielprogramm

```
...
if($v > 4444){
        return $v;
} else {
        return $v+1;
}
...
```

Beispielprogramm

Stackinterpreter

```
                                    retrieve(0, 0)
                                    push(4444, 0)
                                    gt(0, 0)
                                    jfalse(8, 0)
if($v > 4444){                      blockentry(0, 0)

                                    retrieve(0, 1)
                                    store(1, 1)
                                    blockreturn(1, 0)
        return $v;                  j(11, 0)

                                    blockexit(0, 0)
                                    j(9, 0)
} else {                            blockentry(0, 0)

                                    retrieve(0, 1)
                                    push(1, 0)
                                    add(0, 0)
                                    store(1, 1)
                                    blockreturn(1, 0)
        return $v+1;                j(2, 0)

}                                   blockexit(0, 0)
```

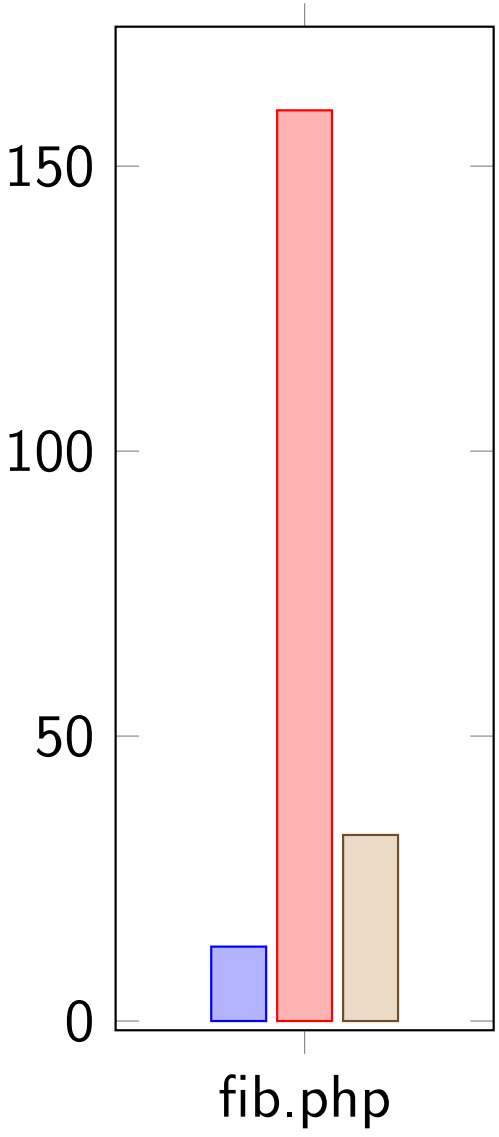
Beispielprogramm

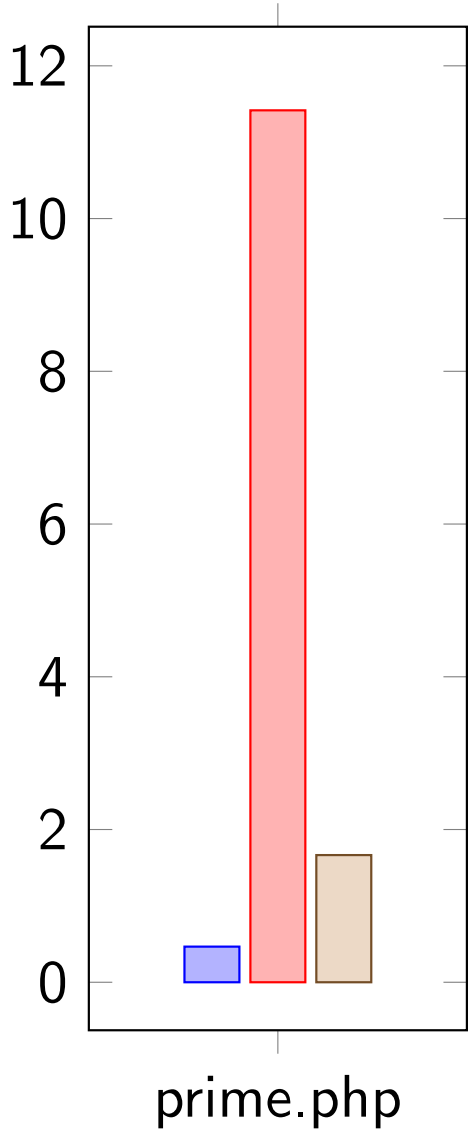
Registerinterpreter

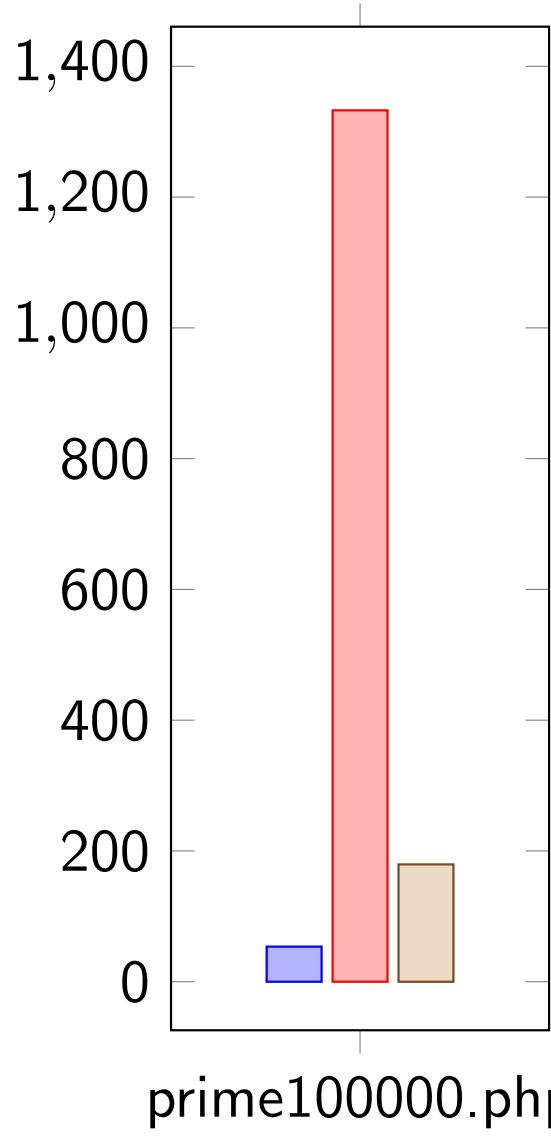
```
                                OpStore r1, 4444, _
                                OpGt r2, r0, r1
if($v > 4444){                  OpJFalse _, r2, 11

                                OpReturn _, r0, _
  return $v;                    OpJump _, _, 14

} else {                        OpStore r3, 1, _
                                OpAdd r4, r0, r3
  return $v+1;                  OpReturn _, r4, _
}
```

Tests

von links nach rechts: PHP 7.0, Stackinterpreter, Registerinterpreter







Tests

Analyse (Beispiel fib.php)

	MPHP Stack	**MPHP Register**
Anzahl Instruktionen	9.109.085.761	4.822.457.196
Durschnittszeit pro Instruktion (ns)	67,2	59,5
Branch misses (in % laut `perf`)	0,69	0,38

V2: Optimierungen

Beispiel fib.php

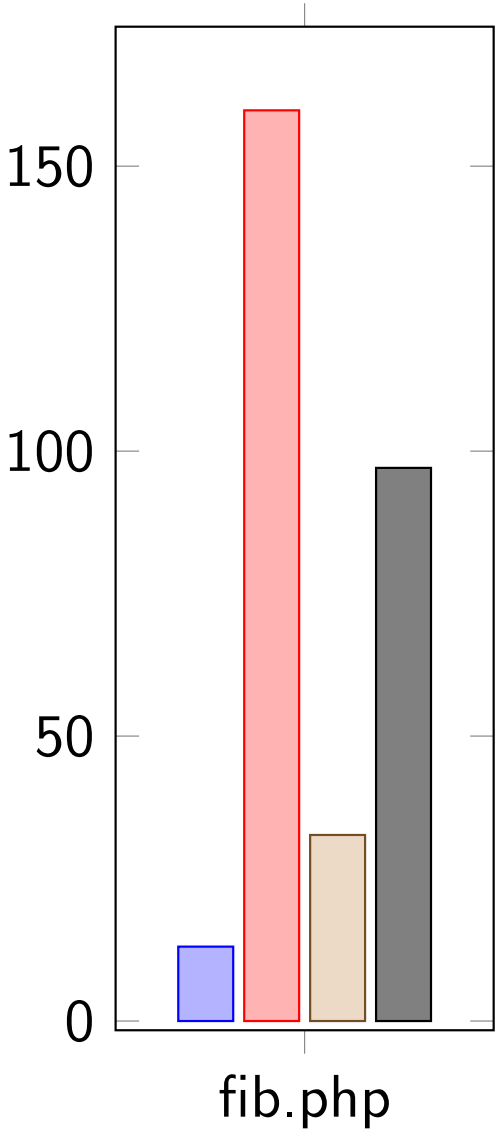
- *blockentry*, *blockreturn*, *blockexit* machen 11% der Instruktionen aus, werden aber eigentlich nicht benötigt.
- Zugriff auf lokale Variablen ist viel zu umständlich.
- Return ist als `jump` auf das Ende der Funktion umgesetzt. Die `jump` befehle können eingespart werden.
- Für Funktionsaufruf muss explizit Platz am Stack reserviert werden (2 überflüssige push Befehle).

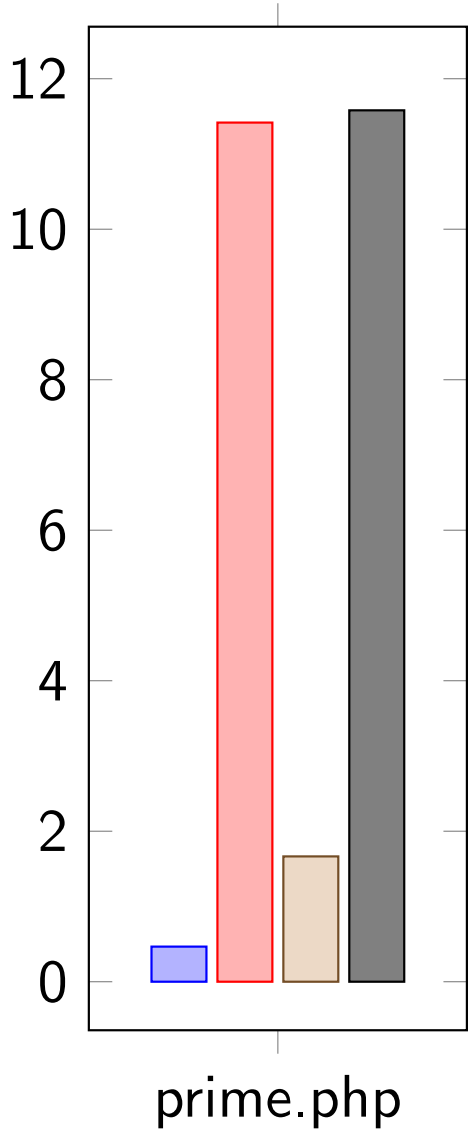
Optimierungen:

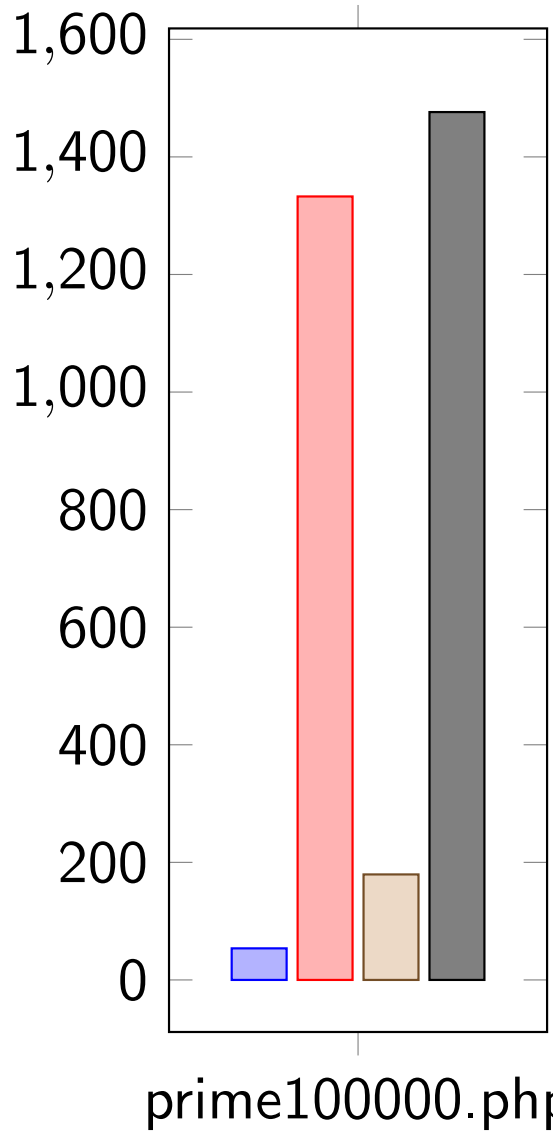
- Blöcke werden im Zwischencode nicht mehr abgebildet (der Compiler allein prüft die Sichtbarkeit).
- Es gibt nur mehr globale und lokale Sichtbarkeit von Variablen.
- Funktionsaufruf und Return werden analog zum Registerinterpreter implementiert.

Tests

von links nach rechts: PHP 7.0, Stackinterpreter, Registerinterpreter, Stackinterpreter V2







Tests

Analyse V2

- Verbesserung bei großer Anzahl von Funktionsaurufen (*fib.php*)
- Verschlechterung bei *prime.php*

Problem: Overhead durch Holen und Starten der Funktionen für die Instruktionen.

Einschränkungen durch GO:

- Bis auf das `switch` statement gibt es keine Möglichkeit Effizient Instruktionen aufzurufen.
- Die Möglichkeit C oder Maschinencode einzubinden beschränkt sich auf Funktionsaufrufe

Lösung: Stackinterpreter V3 in C

Stackinterpreter V3

- Befehlssatz von V2
- Direct threaded code
- Compiler bleibt gleich: Instruction codes werden vor dem Ausführen in Adressen umgewandelt

Dispatch

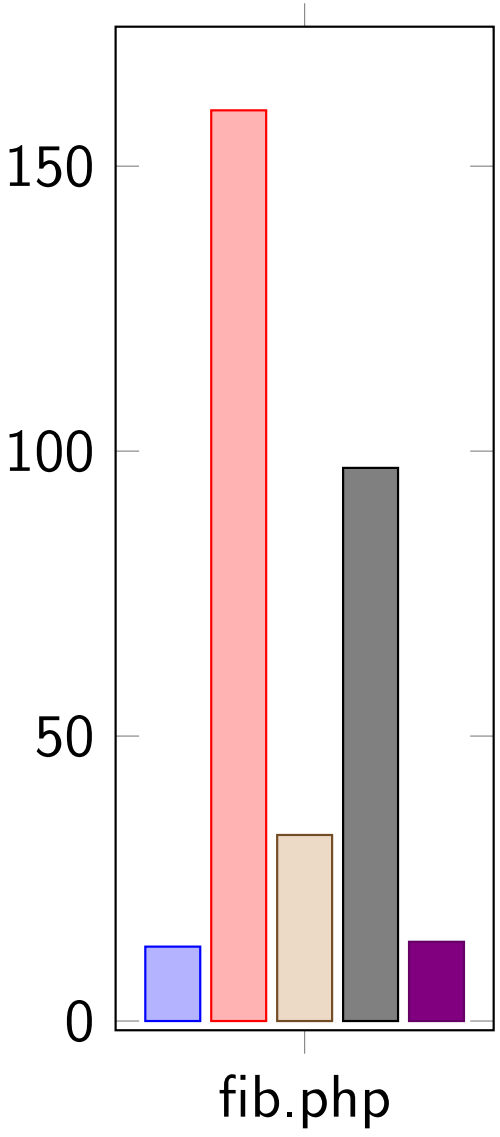
```
void **pc = code_addresses;
```

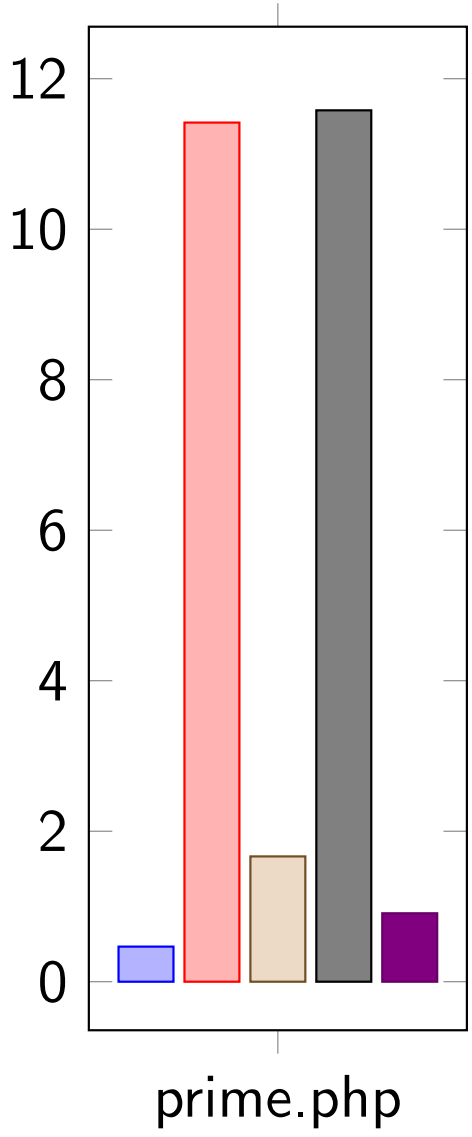
```
...
pc++;
goto **pc;
```

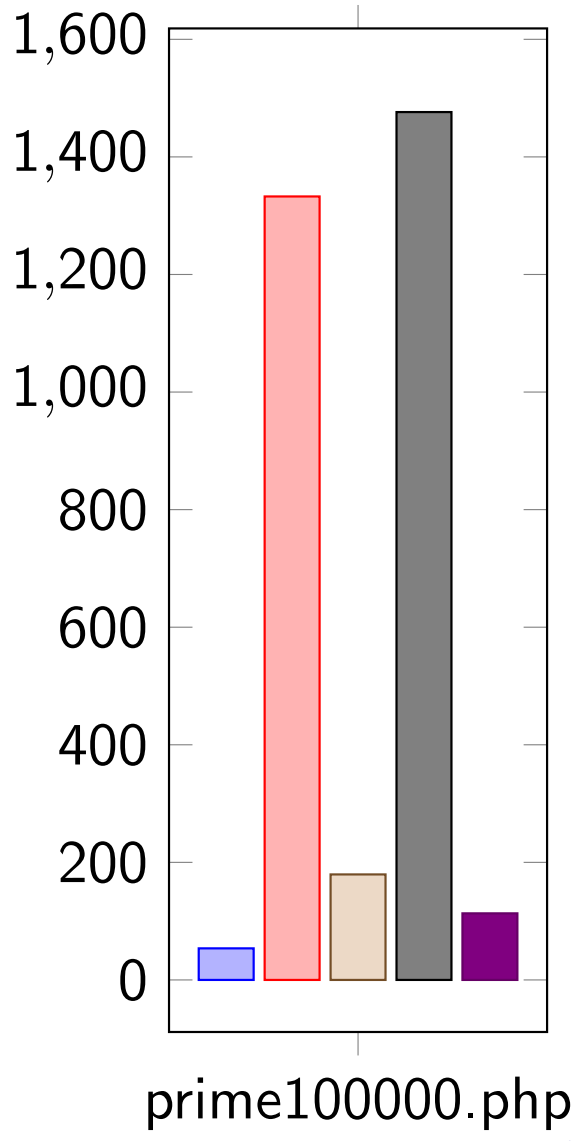
```
add     $0x8,%rbx
...
mov     (%rbx),%rax
jmpq    *%rax
```

Tests

von links nach rechts: PHP 7.0, Stackinterpreter, Registerinterpreter, Stackinterpreter V2, Stackinterpreter V3







Fazit

- Größte Verbesserung durch Dispatch
  - Betrifft jeden Instruktionsaufruf
  - In GO nicht umsetzbar
- Ausnutzen von Sprachfeatures (Labels as Values) und Compileroptimierungen
- Stack-Interpreter von langsamen Dispatch viel stärker betroffen
- Abhängig von Einsatzgebiet: Auswirkung von Intitialisierungsaufwand vernachlässigbar
- Bei beiden Interpretern noch viele Optimierungen möglich
  - Zusammenfassen von Operationen
  - Eigene *increment* Befehle
  - Zugriff auf parameter verbessern
  - Registerinterpreter mit direct threaded code
  - Verwalten von Registern und Pointern in lokalen Variablen
  - Zusammenfassen von Array-Initialisierung
  - Optimieren von Operationen auf Literale