Technische Universität Braunschweig

Programmiersprachen und Übersetzer

Prof. Dr. Christan Dietrich

(Name)	(Vorname)	(Matrikel-Nr.)
(HBK Matrikel-Nr.)	(Studiengang)	(Semester)

Durch meine Unterschrift bestätige ich

- den Empfang der vollständigen Klausur (20 Seiten inklusive Deckblatt),
- die Kenntnisnahme der Hinweise auf Seite 2.

Braunschweig, 14.02.2025

(Unterschrift)

Hinweise

Bitte lesen Sie die folgenden Informationen aufmerksam!

- Die Bearbeitungszeit beträgt 90 Minuten. Antworten können in deutscher oder englischer Sprache verfasst werden.
- Als Hilfsmittel sind **ausschließlich** Lineal, Stift und ein nicht-programmierbarer Taschenrechner zugelassen.
- Die Lösung einer Aufgabe soll auf das Aufgabenblatt in den dafür vorgesehenen Raum geschrieben werden. Beachten Sie bitte, dass der freigelassene Platz großzügig bemessen ist und nicht unbedingt der erwarteten Antwortlänge entspricht. Sollte der Platz nicht ausreichen, können Sie die Rückseiten der Aufgabenblätter mitverwenden. Kennzeichnen Sie dabei die Zugehörigkeit Ihrer Lösung zu einer Aufgabe.
- Bei Bedarf können zusätzliche, besonders markierte, Lösungsblätter (weiß) ausgeteilt werden. Vermerken Sie vor deren Verwendung unbedingt den Klausurcode darauf!
- Die Lösungen müssen dokumentenecht in blau oder schwarz geschrieben werden. Als falsch Erkanntes muss deutlich durchgestrichen werden. Tintenkiller und andere Korrekturstifte dürfen nicht verwendet werden. Keinen Bleistift verwenden!
- Schmierpapier (farbige Blätter) darf **nicht** abgegeben werden. Bei Bedarf ist von der Aufsicht weiteres Schmierpapier (farbig) erhältlich.
- Es dürfen **keine nummerierten** Seiten herausgetrennt werden. Die letzte Seite enthält den Klausurcode und darf herausgetrennt werden

☐ **Aktion**: Tragen Sie Ihren Namen und Vornamen, Ihre Matrikelnummer, Studiengang und Fachsemesterzahl auf dem Deckblatt der Klausur ein.

☐ **Aktion**: Überprüfen Sie die Anzahl der Blätter Ihrer Klausur und unterschreiben Sie die Erklärung auf dem Deckblatt.

☐ **Aktion**: Bitte legen Sie Ihren gültigen Studierenden- und Lichtbildausweis zur Kontrolle bereit.

- Um Unruhe und die Störung Ihrer Mitstudierenden zu vermeiden, ist die vorzeitige Abgabe der Klausur ausgeschlossen. Bleiben Sie an Ihrem Platz sitzen, bis die Aufsicht das Zeichen zum Gehen gibt.

Viel Erfolg!

Korrektur – Zusammenfassung

	erreichbare Punkte	erhaltene Punkte					
Aufgabe 1	15	**1**	**2**	**3**	**4**	**5**	
Aufgabe 2	17	**1**	**2**				
Aufgabe 3	15	**1**	**2**				
Aufgabe 4	8	**1**					
Aufgabe 5	12	**1**					
Aufgabe 6	16	**1**	**2**				
Aufgabe 7	7	**1**					
Summe	90						

Aufgabe 1: Ankreuzfragen (15 Punkte)

In dieser Aufgabe sind jeweils m Aussagen angegeben. Davon sind n ($0 \leq n \leq m$) Aussagen richtig. Kreuzen Sie jeweils an, ob die entsprechende Aussage richtig oder falsch ist.
Jede korrekte Antwort gibt 0,5 Punkte, jede falsche Antwort 0,5 Punkte Abzug. Nicht beantwortete Aussagen gehen neutral in die Bewertung ein. Eine Teilaufgabe wird minimal mit 0 Punkten gewertet, falsche Antworten wirken sich nicht auf andere Teilaufgaben aus.

Wollen Sie eine falsch angekreuzte Antwort korrigieren, streichen Sie bitte das Kreuz mit drei waagerechten Strichen durch (☒).

Lesen Sie die Frage genau, bevor Sie antworten.

Aufgabe 1.1: Syntaktische Analyse

3 Punkte-

Richtig	Falsch	
☐	☐	Die Lexikalische Analyse prüft auf Rechtschreibfehler.
☐	☐	Der Rust-Compiler ist ein Prozessor der Rust Sprache.
☐	☐	LL(1)-Parser sind nur für manche Programme deterministisch.
☐	☐	Der Abstrakte Syntaxbaum beschreibt die Grammatik einer Sprache.
☐	☐	Nach der semantischen Analyse wissen wir, ob ein gültiges Programm vorliegt.
☐	☐	Der Parser stellt fest, ob ein arithmetischer Ausdruck korrekt geklammert ist.

Aufgabe 1.2: Semantische Analyse

3 Punkte-

Richtig	Falsch	
☐	☐	Eine Cast-Operation in C ist eine Form des Inklusionspolymorphismus.
☐	☐	Die vollständige Addressberechnung ist erst nach der Bindezeit möglich.
☐	☐	Zwei Typen sind strukturell äquivalent, wenn sie den gleichen Namen tragen.
☐	☐	Vom Wurzelnamensraum sind alle deklarierten Namen sichtbar.
☐	☐	Record- und Arraytypen sind skalare Typen.
☐	☐	Eine unifizierende Ersetzung gleicht zwei Terme mit freien Variablen an.

Aufgabe 1.3: Codeerzeugung

3 Punkte-

Richtig	Falsch	
☐	☐	Statische Überladung kann ohne Laufzeitkosten umgesetzt werden.
☐	☐	Ein Selektions-Statement provoziert eine Verzweigung im Kontrollfluss.
☐	☐	Zur Berechnung des L-Wert einer Variable wird ihr Inhalt ausgelesen.
☐	☐	Das Wertemodell für Variablen verbietet Zeigertypen auf Sprachebene.
☐	☐	Die Allokation eines Objekts muss seiner Initialisierung vorausgehen.
☐	☐	Entfernt ein Mark-and-Sweep Garbage Collector ein Objekt, so gab es vorher keine einzige Referenz mehr, die auf das Objekt zeigt.

Aufgabe 1.4: Optimierung

3 Punkte-

Richtig	Falsch	
☐	☐	Das Alias-Problem tritt auf, wenn Typen ähnlich benannt sind.
☐	☐	Nach der Optimierung ist der C Ausdruck `4*13+4` ist ebenso effzient wie `56`.
☐	☐	Ein optimierender Übersetzer verändert das beobachtbare Verhalten eines Programms.
☐	☐	Eine gute Registerzuweisung reduziert Speicherzugriffe des Programms erheblich.
☐	☐	Der Optimierer lässt die Anzahl der Basisblöcke gleich.
☐	☐	Jede Optimierung hat zum Ziel, die Ausführung eines Programms schneller zu machen.

Aufgabe 1.5: Maschinencode

3 Punkte-

Richtig	Falsch	
☐	☐	Für Prozessoren eines Typs kann es nur eine Aufrufkonvention geben.
☐	☐	Der Programmlader erzeugt ELF-Dateien.
☐	☐	Symbole sind benannte Offsets innerhalb einer ELF Section
☐	☐	Bei der Relokation wird der ausführbare Binärcode verändert.
☐	☐	Beim Laden entsteht das BSS Segment durch eine Diskrepanz zwischen `FileSiz` und `MemSize` beim Datensegment.
☐	☐	Die Link-View einer ELF-Datei wird zur Laufzeit benötigt.

Aufgabe 2: Syntaktische Analyse (17 Punkte)

Aufgabe 2.1: Umformungen in LL(1)

5 Punkte-

Die Grammatik hat die Terminale {**Ident**, **Eq**, **Int**, **Add**, **Sub**, **LPar**, **RPar**, ⊣}, und die Nichtterminale {*S*, *stmt*, *funCall*, *expr*}. Das Startsymbol ist *S* und es gelten die folgenden Regeln mit den dazugehörigen PREDICT-Mengen.

Begründen Sie, warum die gegebene Grammatik keine LL(1)-Grammatik ist und formen Sie diese in eine um! Für die Umformung ist es ausreichend, die geänderte(n) Regel(n) anzugeben.

$A \rightarrow \alpha$	PREDICT
S→*stmt* ⊣	{Ident}
stmt→**Ident Eq** *expr*	{Ident}
stmt→*funCall*	{Ident}
funCall→**Ident LPar** *expr* **RPar**	{Ident}
expr→*expr* **Add Int**	{Int}
expr→*expr* **Sub Int**	{Int}
expr→**Int**	{Int}

12 Punkte-

Aufgabe 2.2: PREDICT-Mengen

Eine gegebene Grammatik hat die Terminale **Ret**, **If**, **Else**, **End**, **Now**, $\dashv$ und die Nichtterminale *stmt*, *ifStmt*, *mod*, *S*. Das Startsymbol ist *S* und die untenstehende Tablle enthält die Produktionsregeln der Grammatik.

Bestimmen Sie die PREDICT-Mengen für die gegebene Grammatik!

Hinweis: Anhand von EPS können Sie entscheiden, ob Sie nur die FIRST- oder auch die FOLLOW-Menge berechnen müssen.

$A \to \alpha$	EPS(α)	FIRST(α)	FOLLOW(A)	PREDICT
S→*stmt* $\dashv$				
stmt→*ifStmt*				
stmt→*mod* **Ret**				
ifStmt→**If** *stmt* **Else** *stmt* **End**				
ifStmt→ϵ				
mod→ϵ				
mod→**Now**				

Aufgabe 3: Semantische Analyse (15 Punkte)

Aufgabe 3.1: Namensauflösung (Let-Sprache)

10 Punkte-

Gegeben sei folgendes Programm in der let-Sprache, die Sie in der Vorlesung kennengelernt haben. Namen werden mittels **let** deklariert und mittels **ref** referenziert. Standardmäßig sind alle Namen intern sichtbar, extern sichtbare Namen sind gesondert mit **[E]** gekennzeichnet. Importierte Namen werden **nicht** re-exportiert.

(a) Lösen Sie alle Namensreferenzen auf, indem Sie den **vollständig-qualifizierten Namen** der Deklaration oder **NOT FOUND** angeben (linker Kasten). Zusätzlich sollen Sie im jeweils rechten Kasten angeben, ob die Namensauflösung erfolgreich (OK) oder fehlerhaft (Fehler) war, falls bspw. ein Namen mit interner Sichtbarkeit referenziert wurde.

```
1  let A
2  ref A ----------------------->  ::A (Zeile 1)  |  OK
3  let[E] lib {
4    let[E] A
5    let[E] lib = {
6      let lib
7      let[E] PI
8      let[E] A
9      ref lib::A ---------------->  ::  |
10   }
11   ref A ---------------------->  ::  |
12   ref lib::A ----------------->  ::  |
13   ref PI ---------------------->  ::  |
14 }
15 let[E] math {
16     import lib::lib::*
17     let[E] PI
18     let baz
19     ref lib ------------------->  ::  |
20     ref A --------------------->  ::  |
21 }
22 let app = { import ::lib::*  ; import ::math::* ; }
```

(b) **Vervollständigen Sie** folgende Symboltabelle für den Namensraum `app`, die nur direkt referenzierbare Namen (ohne Nutzung von `::`) enthält. **Diesmal werden importierte Namen re-exportiert!**

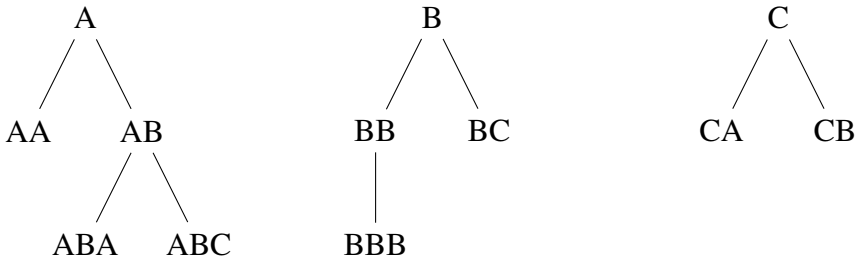
Name	Referenzierte Deklaration
foo	X::Y::foo (Zeile 4)

Name	Referenzierte Deklaration

5 Punkte-

Aufgabe 3.2: Aufruf einer überladenen Funktion

Gegeben sei folgende Vererbungshierarchie, bei der A, B und C unabhängige Basisklassen sind.



Ihre Aufgabe ist es den überladenen Funktionsaufruf in Zeile 4 aufzulösen.
Dabei soll ein **dynamischer Dispatch** im zweiten Argument durchgeführt werden.

```
1  AB arg1   = new ABA();
2  BB arg2   = new BBB();
3  C  arg3   = new CA();
4  BB retval = foo(arg1, arg2, arg3);
```

Berechnen Sie zunächst die Argument-Signatur, welche Sie im nächsten Schritt für die Auflösung der Überlagung verwenden. Die grau hinterlegten Felder bieten Platz für Ihre Antworten:

Um die Funktion mit der höchsten Spezifität zu finden, berechnen Sie für jede der folgenden Deklarationen den Malus. Sollte dies für eine Signatur nicht möglich sein, begründen Sie dies kurz.

Nr.	Parameter-Signatur	Malus	Problem bei der Malusberechnung
1.	`foo(AB, BB, C)`		
2.	`foo(ABC, B, C)`		
3.	`foo(A, BB, CA)`		
4.	`fox(AB, BB, C)`		
5.	`foo(AB, B, C)`		
6.	`foo(A, BB, C)`		

Geben Sie die Nummer (Nr.) der Funktion, die ausgewählt wird. falls die Überladung nicht eindeutig auflösbar ist, geben Sie `-1` an:

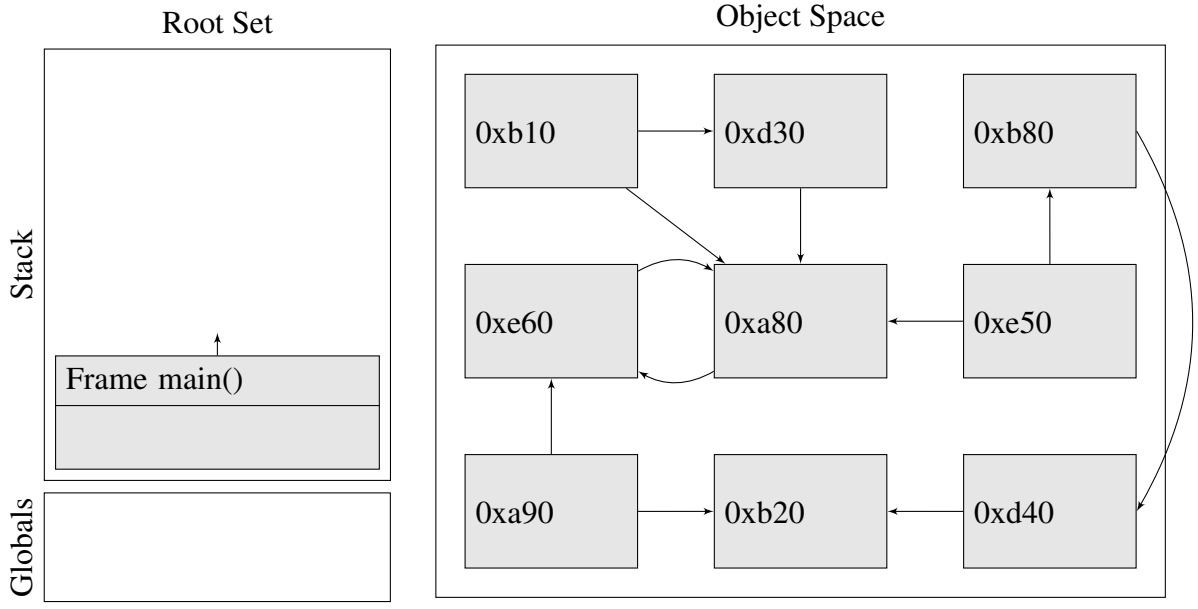
Aufgabe 4: Garbage Collection (8 Punkte)

8 Punkte-

Gegeben sei folgendes Programm (in C-Syntax), welches zusammen mit einem Mark-and-Sweep Garbage Collector ausgeführt wird. Das Programm befindet sich aktuell in Ausführung und die unten stehende Grafik zeigt den Object Space, in den das Programm Referenzen durch C-Casts erzeugt. Nehmen Sie an, dass das Programm zum durch den Kommentar gekennzeichneten Zeitpunkt durch den Garbage Collector unterbrochen wird.

Bestimmen Sie zunächst das Root Set, inklusive der Verbindung in den Object Space, und führen Sie dann die Mark-Phase der Garbage Collection durch. Markieren Sie in der Grafik zu löschende Objekte mit einem Kreuz(×) und Objekte, die nicht gelöscht werden dürfen, mit einem Haken (✓). Die Prozessorregister können vernachlässigt werden.

```
1  void *V = (void *) 0xb20;
2
3  void *object() {
4      return (void *) 0xb10;
5  }
6
7  void run(void *q, int c) {
8      void *b = (void *) 0xe50;
9      b = object();
10     void *r = (void *) 0xa80;
11
12     // <-- Startzeitpunkt des Garbage Collectors!
13
14     void *l = (void *) 0xd30;
15 }
16
17 void main() {
18     void *q = (void *) 0xe60;
19     int b = 0xd40;
20     run(0xe60, 0xe60);
21 }
```



Aufgabe 5: Zwischencodeerzeugung (12 Punkte)

12 Punkte-

Übersetzen Sie das gegebene Programm in der Sprache L0 manuell, **ohne** Umweg über einen abstrakten Syntaxbaum, zu IR-Code. Verwenden Sie dazu die 3-Addresscode -Notation, die Sie in der Vorlesung kennengelernt haben. Geben Sie für die Funktion `foo` ebenfalls die Menge der Parameter und die Menge der lokalen Variablen an. Basisblöcke dürfen, falls möglich, direkt zusammengefasst werden.

Als IR-Befehlssatz stehen Ihnen folgende Instruktionen zur Verfügung:

- Ref, Load, Store
- StackAlloc, HeapAlloc, HeapFree
- Goto, IfGoto, Call, Return
- Add, Sub, Mul, Div und LessEqual

Hinweis: Es lohnt sich, wenn Sie sich zuerst den Kontrollflussgraphen skizzieren!

```
func foo(limit : int, ptr : &int) : int {
  var ret : int; var i : int;
  ret := 0;      i := 0;
  while (i <= limit) {
        i   := i + 1;
        if ((i / 2) * 2 == i) {
           continue;
        }
        ret := ret + i * (*ptr) + 3;
  }
  return ret;
}
```

Schreiben Sie Ihre Lösung auf die **nächste Seite** ⟶

```
func:foo {
// Parameter:
// Lokale Variablen:
BB0: // Entry Block
}
```

1:

Platz für weitere Notizen.

Aufgabe 6: Optimierung (16 Punkte)

Im Folgenden sollen Sie zwei unterschiedliche Optimierungen durchführen, die Berechnungen und Zuweisungen entfernen, deren Ergebnis nicht benötigt wird. Bitte beachten Sie, dass Sie bisher nur die erste Variante exakt so aus der Vorlesung kennen!

Aufgabe 6.1: Fluss-INsensitive Dead Variable Elimination

7 Punkte-

Zunächst sollen Sie eine flussinsensitive Dead Variable Elimination auf dem folgenden Programm durchführen. Dabei dürfen Sie nur Instruktionen entfernen, deren Ergebnis niemals direkt oder indirekt von einer anderen Instruktion gelesen wird. Der Operand von `Return` gilt immer als gelesen und Funktionen können Seiteneffekte haben.

Führen Sie die Optimierung als Fixpunktiteration aus und markieren Sie in der jeweiligen Iteration welcher Befehl entfernt wurde. Jede Iteration verläuft in zwei Phasen: (1) Erkennung der ungenutzten Variablen und (2) Entfernung der unnötigen/toten Zuweisungen. Im folgenden Beispiel wurde der Befehl in der 3. Iteration als eine unnötige Zuweisung erkannt und entfernt:

	Iteration							
IR-Befehl	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
~~`a := Add b, c`~~			×	–	–	–	–	–

Hinweis: Bei flussinsensitiven Optimierung spielt der Kontrollfluss keine Rolle. Daher sind die Befehle des folgenden Programms von uns in zufälliger Reihenfolge notiert worden!

	Iteration							
IR-Befehl	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
`b := Add b, 1`								
`k := Call fn, ptr`								
`g := Sub d, b`								
`d := Mul b, 3`								
`i := Add i, i`								
`c := Add a, 2`								
`h := Add e, g`								
`ptr := Ref i`								
`a := Assign k`								
`j := Sub i, 4`								
`e := Div c, d`								
`f := LessEqual a, 3`								
`Return d`								

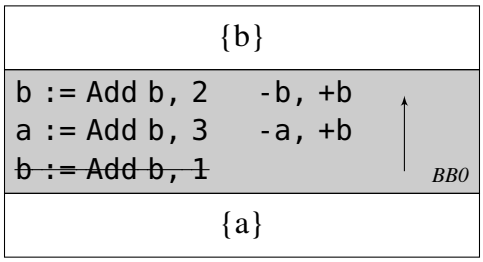
Platz für weitere Notizen.

Aufgabe 6.2: Fluss-Sensitive Dead Store Elimination

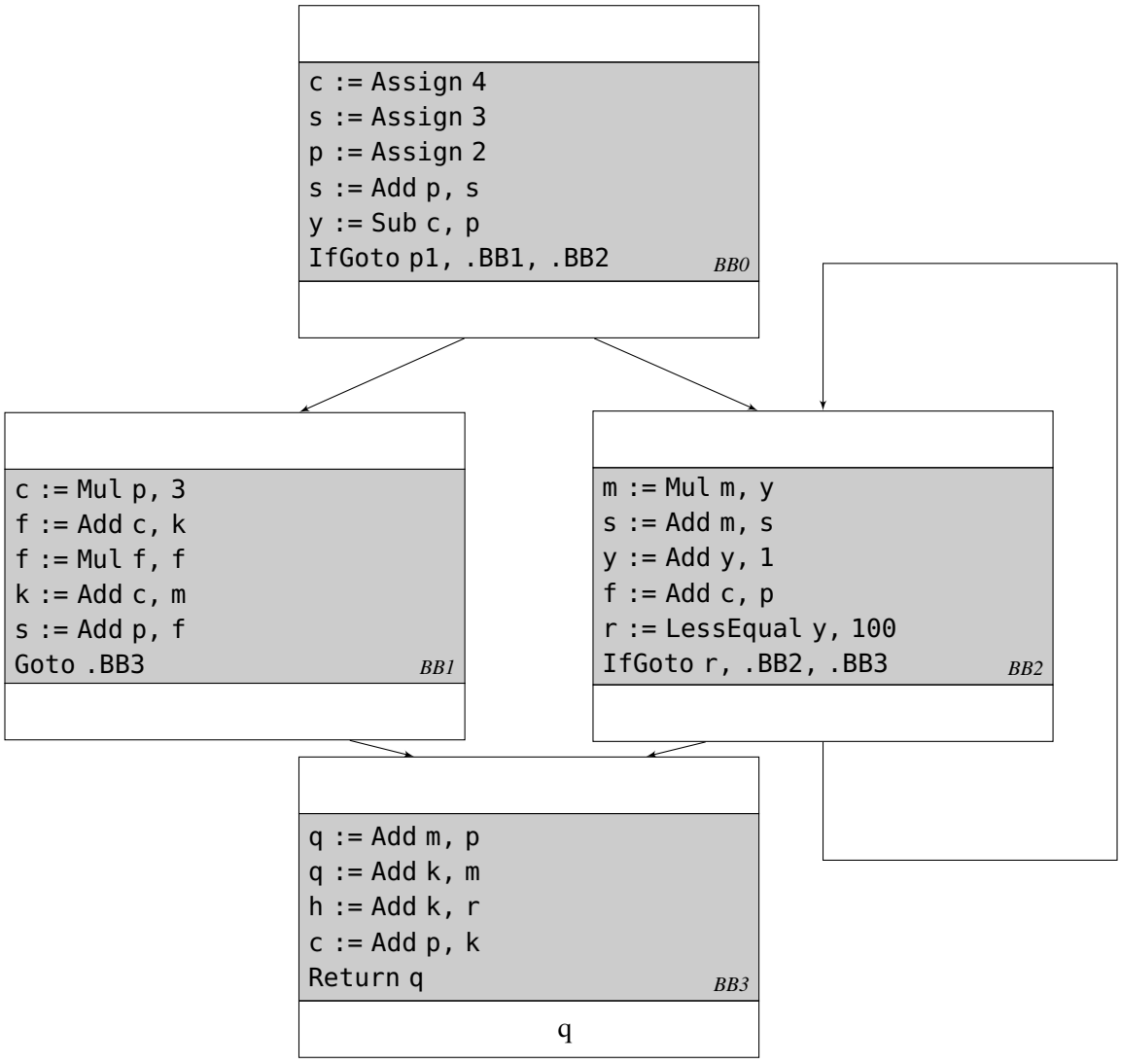
9 Punkte-

Im Folgenden, sollen Sie eine flusssensitive Dead Store Elimination als rückwärtsgerichtete Datenflussanalyse im Fixpunktverfahren durchführen. Dabei propagieren Sie die Menge der verwendeten Variablen “von hinten nach vorne” (im Kontrollflussgraph) und von “unten nach oben” (im Basisblock) und entfernen überflüssige Instruktionen.

Hinweis: Eine Variable wird zur Menge hinzugefügt wenn sie als Operand einer Instruktion gelesen wird und entfernt, wenn sie das Ergebnis einer Instruktution aufnimmt. Weiterhin gilt eine Variable am Ende eines Blockes als benutzt, wenn sie zu Beginn eines Nachfolgerblocks in der Menge ist. Die Return Instruktion darf nicht gelöscht werden. Das folgende Beispiel zeigt das Verfahren beispielhaft:



Hinweis: Beginnen Sie mit ihrer Optimierung bei BB3 und arbeiten Sie von unten nach oben.



Platz für weitere Notizen.

Aufgabe 7: Registervergabe (7 Punkte)

Im Folgenden sollen Sie eine Registervergabe „mit Gedächtnis“ durchführen. Vervollständigen Sie dazu die Assemblerausgabe, indem Sie die Operanden der Makroexpansion und eventuell nötige Spill- und Ladebefehle ergänzen! Geben Sie weiterhin den internen Zustand des Registerallokators nach der letzten Assemblerinstruktion eines IR-Befehls aus! Schreiben Sie Ihre Lösung in die grau hinterlegten Felder. Die übrigen Felder können Sie für Notizen benutzen.

Randbedingungen der Registervergabe:

- **Aufrufkonvention**: Der erste Parameter eines Funktionsaufrufs muss nach `%eax`; der Rückgabewert ist nach der Rückkehr in `%eax`; der Aufgerufene sichert keinen Maschinenzustand.
- Nehmen Sie an, dass zu jeder Variable eine **Referenz** erzeugt wurde!
- Verwenden Sie **symbolische Variablennamen** anstatt `%ebp`-relativer Slotoffsets.

Dokumentation zu einigen Assemblerbefehlen:

`mov %eax, %ebx`	Kopiert den Inhalt des Registers `%eax` ins Register `%ebx`.
`mov x, %eax`	Kopiert den Inhalt des Variablenslots `x` ins Register `%eax`
`add %eax, %ebx`	Addiert beide Register; das Ergebnis landet in `%ebx`.
`mov (%eax), %ebx`	Dereferenziert den Zeiger in `%eax`; das Ergebnis landet `%ebx`.
`mov %eax, (%ebx)`	Kopiert den Inhalt von `%eax` an die Adresse in `%ebx`.
`xchg %eax, %ecx`	Tauscht die Inhalte der Register `%eax` und `%ecx`.

Aufgabe 7.1: Basisblock BB0

7 Punkte-

IR-Code	Assemblerausgabe	Zustand nach Instruktion %eax	%ebx	%ecx
`b := Assign a`	`mov a, %eax`	`a, clean`	`---`	`---`
	`mov` [`%eax`], [`%ebx`]	`a, clean`	`b, dirty`	`---`
`a := Call fn, d`				
	`call fn`			
`c := Load *b`				
	`mov (` [ ] `),` [ ]			
`a := Add b, c`				
	`add` [ ], [ ]			
`*d = Store a`				
	`mov (` [ ] `),` [ ]			

Platz für weitere Notizen.

Entfernen Sie dieses Blatt vor der Bearbeitung der Klausur und bewahren Sie es auf! Mit Hilfe des Codes können Sie Ihr persönliches Ergebnis online abfragen.

Klausur **Programmiersprachen und Übersetzer**, 2025-02-14.

Ergebnisse online: https://studip.tu-braunschweig.de