Übungsblatt 9

Übungen nach Weihnachten

Obwohl es am 22.12. keine Vorlesung geben wird, finden in der ersten Woche nach den Weihnachtsferien trotzdem Übungen statt (x86-Assembler, Systemaufrufe, Assemblierer und Binder, Debugger, ...).

Benchmark-Wettbewerb

Wir werden gegen Ende des Semesters einen kleinen Benchmark-Wettbewerb mit geheimen Testprogrammen durchführen, bei dem eure Übersetzer gegeneneinander antreten. Eine gute Implementierung der Übersetzung nach Zwischencode sowie effiziente Code-Erzeugung samt Registervergabe lohnt sich also... ☺

Aufgabe 9.1: Zwischencode im e2-Übersetzer

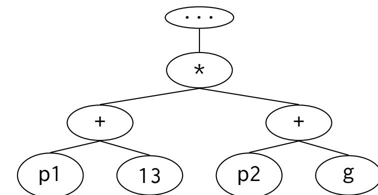
Auf der Internetseite zur Übung finden Sie eine knappe Übersicht über die *Zwischencodedarstellung*, die im e2-Übersetzer verwendet werden soll. Lesen Sie die Beschreibung und überlegen Sie sich die Semantik der Instruktionen und Operanden.

Aufgabe 9.2: Übersetzung in den e2-Zwischencode

Übersetzen Sie die folgenden Code-Fragmente in den e2-Zwischencode:

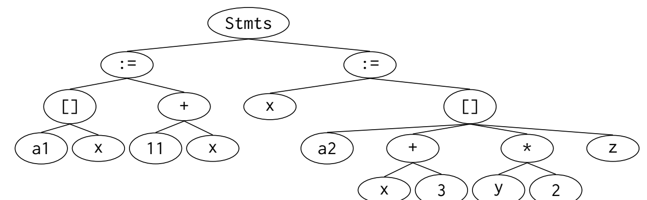
a)

```
var g : int;


func foo(p1 : int, p2 : int): int
  return (p1 + 13) * (p2 + g);
end
```



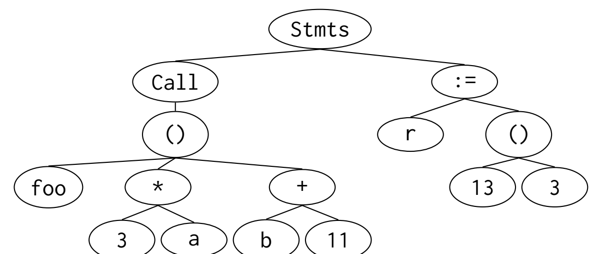
b)

```
# x, y, z seien lokale Variablen

var a1 : int[42];
var a2 : int[13][3][12];

a1[x] := 11 + x;
x := a2[x + 3][y * 2][z];
```



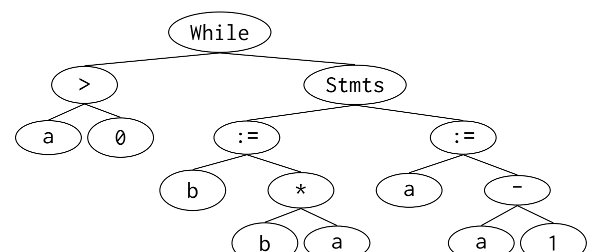
c)

```
# r, a, b seien lokale Variablen

foo(3 * a, b + 11);

r := foo(13, 3);
```



d)

```
# a, b seien lokale Variablen

while a > 0 do
  b := b * a;
  a := a - 1;
end
```



Aufgabe 9.3: Codierungsphase

Aus welchen Teilaufgaben besteht die *Codierungsphase* in einem Übersetzer?

Was sind Ein- und Ausgabe dieser Phase?

Aufgabe 9.4: Optimale Registerverwendung für Ausdrucksbäume

In der Vorlesung haben Sie ein Verfahren zur optimalen Registerverwendung für Ausdrucksbäume kennengelernt. Unter gewissen Voraussetzungen und für bestimmte Klassen von Zielarchitekturen garantiert dieses Verfahren, dass möglichst wenige Zwischenergebnisse in den Speicher geschrieben werden müssen (und dadurch eine möglichst kurze Instruktionssequenz zur Auswertung des Ausdrucks erzeugt wird).

Es sei eine Zielarchitektur mit den zwei Allzweckregistern `r[12]` sowie den folgenden Operationen gegeben (r_i bezeichne ein Register, M_x den Speicherplatz an der symbolischen Adresse `x`, `c` eine Konstante):

Name	Operation	Semantik
L	$r_i := M_x$	lade Wert von Speicherstelle M_x in Register r_i
S	$M_x := r_i$	speichere Wert von Register r_i an Speicherstelle M_x
M	$r_i := r_j$	kopiere Wert von Register r_j in Register r_i
RoR	$r_i := r_i \circ r_j$	verknüpfe die Werte in den Registern r_i und r_j mit dem arithmetischen Operator $\circ$ und speichere das Ergebnis in r_i
RoM	$r_i := r_i \circ M_x$	verknüpfe den Wert in Register r_i mit dem Wert an Speicherstelle M_x mit dem arithmetischen Operator $\circ$ und speichere das Ergebnis in r_i

Wenden Sie das Verfahren auf das folgende Code-Fragment in der `e2`-Zwischencodedarstellung an:

```
%v1 = add a, b
%v2 = sub c, %v1
%v3 = add a, b
%v4 = mul c, a
%v5 = add %v3, %v4
%v6 = mul %v2, %v5
d = mov %v6
```

Projektübung 5 (Abgabe bis 14.01.2018)

Im Rahmen dieser Projektübung erweitern Sie Ihren `e2`-Übersetzer um die Übersetzung des attributierten ASTs in semantisch äquivalenten Zwischencode.

a) Machen Sie sich zunächst mit den vorgegebenen Klassen für die Zwischencodedarstellung im Paket `e2c.middleend.ir` vertraut:

- Ein Programm wird durch eine Instanz der Klasse `IRProgram` repräsentiert. Diese Klasse stellt mit `createLabel()` auch eine Methode zur Erzeugung global eindeutiger Sprungziele zur Verfügung.
- Eine Funktion wird durch eine Instanz der Klasse `IRFunction` repräsentiert. Diese Klasse stellt Methoden zum Erzeugen von Parametern (`createParameter()`), lokalen Variablen (`createLocalVariable()`) und virtuellen Registern (`createVirtualRegister()`) zur Verfügung.
- Eine Instruktion wird durch eine Instanz einer Unterklasse von `IRInstruction` repräsentiert.
- Eine Instruktionsfolge wird durch eine Instanz der Klasse `IRInstructionList` repräsentiert, die Methoden zum Einfügen und Löschen von Instruktionen bereitstellt. Die eigentliche Verkettung erfolgt in den Instruktionen selbst.

- Ein Operand wird durch eine Instanz einer Unterklasse von `IROperand` repräsentiert. Ein Operand ist entweder eine Konstante (`IRConstant` bzw. Unterklassen davon) oder eine Variable (`IRVariable`). Eine Variable entspricht dabei entweder einem Parameter oder einer (globalen oder lokalen) Variable des Quellprogramms oder einem virtuellen Register. Für den Zugriff auf Variablen werden symbolische Namen verwendet (d.h. auch lokale Variablen werden im Zwischencode noch *nicht* relativ zu einem Framepointer o.ä. adressiert; dies soll erst bei der späteren Code-Erzeugung passieren).
- Wie der AST wird auch der Zwischencode mit Hilfe von Besuchern traversiert. Diese befinden sich im Paket `e2c.middleend.ir.visitors`.

b) Die Übersetzung des ASTs in semantisch äquivalenten Zwischencode soll mit Hilfe eines AST-Besuchers `e2c.frontend.ast.visitors.IRGenerator` geschehen. Sorgen Sie zunächst dafür, dass dieser eine Instanz von `IRProgram` für das Eingabeprogramm erzeugt.

c) Bevor Sie die eigentliche Übersetzung in Zwischencode implementieren, sollten Sie in der Klasse `E2Compiler` dafür sorgen, dass Ihr Übersetzer den Besucher zur Zwischencodeerzeugung geeignet aufruft. Wenn Sie Ihren Übersetzer nun mit der Kommandozeilenoption „`--dump-ir`“ starten, sollte eine textuelle Repräsentation des bei der Übersetzung generierten Zwischencodes in die angegebene Datei geschrieben werden. Diese Ausgabe ist zur Überprüfung der Implementierung äußerst hilfreich!

Hinweis: Die textuelle Repräsentation eines (korrekten) Zwischencodeprogramms kann mit Hilfe der Klasse `E2Assembler` bzw. des Skripts `e2as` auch wieder eingelesen werden.

d) **Optional**: Die Testfälle zum vierten Meilenstein (siehe unten) verwenden die vorgegebene Klasse `IRSanitizer`, die ein paar grundlegende Überprüfungen auf dem Zwischencode durchführt (beispielsweise, dass Variablennamen eindeutig sind oder dass die Typen der Operanden einer Instruktion zusammenpassen). Sie *können* nach der Übersetzung den `IRSanitizer` auf den generierten Zwischencode anwenden. Dies kann Ihnen u.U. beim Finden und Beheben von Fehlern in Ihrer Implementierung helfen.

e) Erweitern Sie den Besucher dahingehend, dass die globalen Variablen des Eingabeprogramms als Instanzen von `IRVariable` zum `IRProgram` hinzugefügt werden. Überlegen Sie sich in diesem Zusammenhang eine sinnvolle Möglichkeit, wie Sie später auf die zu den einzelnen Variablen gehörenden Instanzen von `IRVariable` zugreifen können.

f) Erweitern Sie den Besucher so, dass für jede Funktion des Eingabeprogramms eine Instanz von `IRFunction` erzeugt und dem `IRProgram` hinzugefügt wird. Sorgen Sie außerdem dafür, dass Instanzen von `IRVariable` für alle lokalen Variablen und Parameter erzeugt und der jeweiligen Funktion hinzugefügt werden.

Wichtig: Variablen müssen (zumindest innerhalb einer Funktion) einen **eindeutigen Namen** haben, damit das Zwischencodeprogramm serialisiert und deserialisiert werden kann. Falls eine Funktion also in unterschiedlichen Sichtbarkeitsblöcken mehrere Variablen mit demselben Namen deklariert, müssen diese in der Zwischencodedarstellung umbenannt werden. Verwenden Sie deshalb zum Erzeugen und Hinzufügen von lokalen Variablen und Parametern **unbedingt** die Methoden `createLocalVariable()` bzw. `createParameter()` der Klasse `IRFunction`, die diese Umbenennung durchführen.

g) Laut Sprachspezifikation geben Funktionen (je nach Rückgabetyp) einen Standardwert zurück, falls ihre Ausführung das Ende der Funktion erreicht, ohne vorher eine `return`-Anweisungen erreicht zu haben. Überlegen Sie sich, wie Sie dieses Verhalten bei der Übersetzung in Zwischencode umsetzen können.

Auch `void`-Funktionen sollen bei Terminierung eine explizite `RET`-Instruktion erreichen.

h) Implementieren Sie in Ihrem Besucher zunächst die Übersetzung arithmetischer Ausdrücke. Beachten Sie dabei die folgenden Hinweise:

- Für die Speicherung von Zwischenergebnissen müssen virtuelle Register erzeugt werden, die der jeweiligen Instanz von `IRFunction` genau wie lokale Variablen und Parameter hinzugefügt werden müssen. Verwenden Sie dazu die Methode `createVirtualRegister()` der Klasse `IRFunction`.

- Funktionsaufrufe werden auf `CALL`-Instruktionen abgebildet, die alle Argumente für die aufgerufene Funktion als Quelloperanden beinhalten. Beachten Sie, dass `void`-Funktionen keinen Wert zurückliefern, der in einer `IRVariable` gespeichert werden könnte...
- Die `LOAD`- und `STORE`-Instruktionen erlauben jeweils nur einen einzigen Indexoperanden. Zugriffe auf mehrdimensionale Arrays müssen deshalb bei der Zwischencodeerzeugung auf Zugriffe auf eindimensionale Arrays abgebildet werden. Die dazu notwendigen Schritte sind Ihnen aus der Vorlesung und der Übung bekannt.

  Hinweis: Der *Typ* einer Variable darf nach wie vor ein mehrdimensionales Array sein.

i) Implementieren Sie in Ihrem Besucher die Übersetzung von `return`-Anweisungen.

j) Implementieren Sie in Ihrem Besucher die Übersetzung von Zuweisungen. Bedenken Sie, dass auf der linken Seite einer Zuweisung ein Array-Ausdruck stehen kann.

k) Erweitern Sie Ihren Besucher um die Behandlung von logischen Ausdrücken in Bedingungen. Diese verwenden laut Sprachspezifikation die sog. *Kurzschlusssemantik*, d.h. der Ausdruck soll nur solange ausgewertet werden, bis sein Ergebnis eindeutig feststeht.

l) Implementieren Sie in Ihrem Besucher die Übersetzung von `if`- und `while`-Anweisungen.

Ihr Übersetzer sollte nun in der Lage sein, Zwischencode für beliebige Eingabeprogramme zu erzeugen. Zur Überprüfung der Implementierung stellt die Vorlage einen (sehr ineffizienten!) Interpreter zur Verfügung, der die Zwischencodeanweisungen ausführt. Verwenden Sie zur Ausführung des Interpreters die Kommandozeilenoption „`-x`“ beim Start des Übersetzers. Testen Sie Ihre Implementierung mit eigenen Testprogrammen bzw. mit den Testfällen zum vierten Meilenstein (siehe unten).

Meilenstein 4 (Abgabe bis 14.01.2018)

Stellen Sie sicher, dass Ihr Übersetzer alle zum vierten Meilenstein gehörenden Testfälle korrekt behandelt, indem Sie den Gradle-Task „`milestone4`“ ausführen und kontrollieren, dass am Ende die Ausgabe „`BUILD SUCCESSFUL`“ erscheint. Versehen Sie den finalen Stand mittels „`git tag`“ mit der Markierung „`milestone4`“ und pushen Sie Ihre Änderungen auf den `git`-Server des Lehrstuhls.