KSP Aufgabe 7

In dieser Aufgabenstellung soll das Rechnen mit Komponenten von Records und Arrays implementiert werden. Bei uns sind diese beiden Datenstrukturen (im Gegensatz zu C) "richtige Objekte", d.h. alle Instanzen von Records und Arrays leben auf dem Heap und werden nur über Zeiger zugegriffen. Das ist in Ninja nicht anders als in Java, und deshalb heissen die Komponenten auch "Instanzvariable". Um Ihnen einen Eindruck zu vermitteln, welche Berechnungen nun möglich sind, gibt es hier drei Beispielprogramme:

- listrev.nj (Umdrehen einer Liste von Zahlen)
- twodim.nj (Belegen und Ausgeben eines zweidimensionalen Arrays)
- matinv.nj (Invertieren einer 2x2-Matrix mit Brüchen als Komponenten)

Vorgehen

1. Studieren Sie die neu hinzugekommenen Instruktionen des dann kompletten Befehlssatzes unserer VM, wobei Sie die Diskussion der Struktur von Objekten und die Erläuterungen zu Referenzvergleichen hinzuziehen sollten.
2. Überprüfen Sie Ihr Verständnis der neuen Befehle! Vorschlag: Sie lassen das o.g. Programm `matinv.nj` übersetzen und entnehmen dem entstandenen Assemblerprogramm die Funktionen newFraction , subFraction und newMatrix . Dann ergänzen Sie die Funktionen Zeile für Zeile um einen aussagekräftigen Kommentar, was da jeweils genau passiert. Sie sollten zu jedem Zeitpunkt den Stack der VM zeichnen können!
3. Implementieren Sie die neuen Instruktionen. Sie können mit der Umsetzung der Objektstruktur beginnen. Es gibt ja zwei Sorten von Objekten: primitive Objekte (die eine Anzahl Bytes speichern) und zusammengesetzte Objekte (die eine Anzahl Objektreferenzen speichern). Das höchste Bit des Zählers, der in jedem Objekt enthalten ist, bestimmt, ob der Zähler Bytes oder Instanzvariable zählt. Deswegen kann man als Interface zum Objektspeicher die beiden folgenden Funktionen vorsehen:
   a. `ObjRef newPrimitiveObject(int numBytes);`
   b. `ObjRef newCompoundObject(int numObjRefs);`
4. Dann realisieren Sie `new`, `getf` und `putf`. Denken Sie daran, alle Instanzvariablen mit `nil` zu initialisieren! Als nächstes kommen die Instruktionen für Arrays an die Reihe; bitte auch hier die Initialisierung nicht vergessen. Und wo wir schon dabei sind: die lokalen Variablen einer Methode müssen ebenfalls mit `nil` initialisiert werden, genauso wie die globalen Variablen. Noch ein Hinweis: Ihre VM muss erkennen, wenn auf ein Objekt zugegriffen werden soll, aber nur eine `nil`-Referenz vorliegt. Ebenso müssen Sie einen Zugriff auf ein Array mit unzulässigem Index abfangen. Testen Sie alle Befehle und Fehlerbedingungen ausführlich! Vergessen Sie nicht die Instruktion `getsz` zum Bestimmen der Größe eines Objektes, sowie die Instruktionen zu den Referenzvergleichen!
5. Passen Sie Ihren Debugger der veränderten Situation an. Insbesondere die Inspektion von Objekten wird wahrscheinlich auf das MSB des Zählers Rücksicht nehmen müssen.
6. Natürlich gibt's einen erweiterten Compiler , der mit Records und Arrays umgehen kann (dazu

die Grammatik , die Tokens , und die vordefinierten Bezeichner), sowie einen passenden Assembler , der die neuen Instruktionen assemblieren kann.

7. Hier wie immer die Referenzimplementierung: njvm

Vollständige Liste der Instruktionen der VM

Table 1. VM Instruktionen

Instruktion	Opcode	Stack Layout
halt	0	... -> ...
pushc <const>	1	... -> ... value
add	2	... n1 n2 -> ... n1+n2
sub	3	... n1 n2 -> ... n1-n2
mul	4	... n1 n2 -> ... n1*n2
div	5	... n1 n2 -> ... n1/n2
mod	6	... n1 n2 -> ... n1%n2
rdint	7	... -> ... value
wrint	8	... value -> ...
rdchr	9	... -> ... value
wrchr	10	... value -> ...
pushg <n>	11	... -> ... value
popg <n>	12	... value -> ...
asf <n>	13	
rsf	14	
pushl <n>	15	... -> ... value
popl <n>	16	... value -> ...
eq	17	... n1 n2 -> ... n1==n2
ne	18	... n1 n2 -> ... n1!=n2
lt	19	... n1 n2 -> ... n1<n2
le	20	... n1 n2 -> ... n1<=n2
gt	21	... n1 n2 -> ... n1>n2
ge	22	... n1 n2 -> ... n1>=n2
jmp <target>	23	... -> ...
brf <target>	24	... b -> ...
brt <target>	25	... b -> ...
call <target>	26	... -> ... ra
ret	27	... ra -> ...
drop <n>	28	... a0 a1...an-1 -> ...
pushr	29	... -> ... rv
popr	30	... rv -> ...

Instruktion	Opcode	Stack Layout
dup	31	... n -> ... n n
new <n>	32	... --> ... object
getf <n>	33	... object --> ... value
putf <n>	34	... object value --> ...
newa	35	... number_elements --> ... array
getfa	36	... array index --> ... value
putfa	37	... array index value --> ...
getsz	38	... object --> ... number_fields
pushn	39	... --> ... nil
refeq	40	... ref1 ref2 --> ... ref1==ref2
refne	41	... ref1 ref2 --> ... ref1!=ref2

Aufgaben für Tests

a. Entwerfen Sie eine Datenstruktur zum Speichern von Bäumen für arithmetische Ausdrücke in Ninja (ganz so, wie wir das in der Vorlesung für C gemacht haben). Stellen Sie die entsprechenden Konstruktoren bereit, sowie eine Methode, die einen Baum mit korrekter Einrückung ausgibt. Überprüfen Sie Ihr Programm mit dem Ausdruck `5 - (1 + 3) * (4 - 7)`: diese Konstruktoraufrufe sollten einen Baum erzeugen, der, wenn er ausgegeben wird, diese Ausgabe produzieren sollte. Nein, das ist kein Fehler in der Aufgabenstellung... ;-)

b. Ersetzen Sie die Ausgabe in a) durch eine Auswertefunktion ("Evaluator"). Kommt bei dem o.g. Beispiel wirklich 17 heraus?

c. Ersetzen Sie jetzt den Evaluator aus b) durch eine Prozedur zum Erzeugen von Ninja-Assembler. Wenn Sie den erzeugten Code für das o.g. Beispiel assemblieren und ausführen lassen, kommt dann wieder 17 heraus?