P. Thiemann, S. Wehr Wintersemester 2006/2007

Compilerbaupraktikum

http://proglang.informatik.uni-freiburg.de/teaching/compilerpraktikum/2006ws/

Aufgabenblatt 3

Zwischensprache

22.12.2006

Ziel dieses Aufgabenblatts ist die Transformation von MiniJava in eine Zwischensprache `JIR`. Nach der Transformation sollen alle Methodenrümpfe in `JIR` vorliegen. Zur Vereinfachung können sie Initialisierungsausdrücke für Instanzvariablen ignorieren.

`JIR` abstrahiert dabei über Register `reg` (von denen es beliebig viele gibt), Speicheradressen `addr` sowie über Sprungmarken `label`. Folgende OCaml Datentypen definieren `JIR`:

```
type op   = Add | Sub | Mult | Div | And | Or | Lt | Leq | Gt | Geq | Eq | Neq
type uop  = UMinus | Not
type sys  = Write | Read | Alloc
type exp  = Const of int
          | Addr of addr
          | Reg of reg
          | Get of exp
          | Binop of exp * op * exp
          | Unop of uop * exp
type stmt = MoveMem of exp * exp
          | MoveReg of reg * exp
          | Call of reg * int * exp list
          | Syscall of reg * sys * exp list
          | Jump of exp * label
          | Label of label
          | Return of exp
```

Ausdrücke vom Typ `exp` haben keine Seiteneffekte. Die Bedeutung der einzelnen Ausdrucksformen ist wie folgt:

- `Const(`i`)`. Die Integerkonstante i.
- `Addr(`a`)`. Eine Speicheradresse a.
- `Reg(`r`)`. Der Inhalt des Registers r.
- `Get(`e`)`. Speicherzugriff. Auswerten des Ausdrucks e muss eine Adresse a liefern. Das Ergebnis des gesamten Ausdrucks ist dann der Inhalt des Speichers an Adresse a.
- `Binop(`e_1`, `o`, `e_2`)`. Ein binärer Operator o angewandt auf zwei Operanden e_1 und e_2.
- `Unop(`o`, `e`)`. Ein unärer Operator o angewandt auf den Operanden e.

Statements vom Typ `stmt` können Seiteneffekte haben. Die Bedeutung der einzelnen Statements ist wie folgt definiert:

- `MoveMem(`e_1`, `e_2`)`. Wertet den Ausdruck e_1 zu einer Adresse a aus. Der Wert von e_2 wird dann in a gespeichert.
- `MoveReg(`r`, `e`)`. Speichert den Wert des Ausdrucks e in Register r.
- `Call(`r`, `i`, [`e_0`; ...; `e_n`])`. Methodenaufruf. Der Ausdruck e_0 ist der Empfänger des Aufrufs; $e_1, \ldots, e_n$ sind die Argumente. Der Index i ist der Index der aufzurufenden Methoden in der zum Empfänger gehörenden Methodentabelle. (Methodentabellen sind z.B. im Buch von Andrew Appel, welches auf der Homepage zur Vorlesung aufgeführt ist, erklärt.) Im Register r wird schließlich das Ergebnis des Aufrufs abgelegt.
- `Syscall(`r`, `s`, [`e_1`; ...; `e_n`])`. Führt den Systemaufruf s aus und speichert das Ergebnis im Register r. Die Argument e_1 bis e_n und das Ergebnis sind abhängig vom Systemaufruf:

Systemaufruf	*Argumente*	*Ergebnis*
`Write`	e_1: Adresse des zu druckenden Strings	undefiniert
`Read`	keine	Adresse des gelesenen Strings
`Alloc`	e_1: Länge des zu allozierenden Speichers	Anfangsadresse des allozierten Speichers

- `Jump(`e`, `l`)`. Springt zum Label l falls e zu 0 auswertet.
- `Label(`l`)`. Setzt den Label l.
- `Return(`e`)`. Gibt e als Ergebnis zurück.

Fügen sie abschließend dem Treibermodule `Driver` eine neue Phase hinzu, welche die Übersetzung nach `JIR` aufruft. Vergessen sie nicht, Tests zu schreiben.

Deadline: 17.1.2007