(* File ListC/Comp.fs A compiler from list-C, a sublanguage of the C language extended with lists, to an abstract machine with garbage collection.
Direct (forwards) compilation without optimization of jumps to jumps, tail-calls etc. sestoft@itu.dk * 2009-10-27

TODO: Update description below:

A value is an integer; it may represent an integer or a pointer, where a pointer is just an address in the store (of a variable or pointer or the base address of an array).

The compile-time environment maps a global variable to a fixed store address, and maps a local variable to an offset into the current stack frame, relative to its bottom. The run-time store maps a location to an integer. This freely permits pointer arithmetics, as in real C. A compile-time function environment maps a function name to a code label. In the generated code, labels are replaced by absolute code addresses.

Expressions can have side effects. A function takes a list of typed arguments and may optionally return a result.

Arrays can be one-dimensional and constant-size only. For simplicity, we represent an array as a variable which holds the address of the first array element. This is consistent with the way array-type parameters are handled in C, but not with the way array-type variables are handled. Actually, this was how B (the predecessor of C) represented array variables.

The store behaves as a stack, so all data except global variables are stack allocated: variables, function parameters and arrays.
*)

module Comp

open System.IO open Absyn open Machine

(* ———————————————————————————————————————- *)

(* Simple environment operations *)

type 'data env = (string * 'data) list

let rec lookup env x = match env with | [] -> failwith (x + " not found") | (y, v)::yr -> if x=y then v else lookup yr x

(* A global variable has an absolute address, a local one has an offset: *)

type var = | Glovar of int (* absolute address in stack *) | Locvar of int (* address relative to bottom of frame *)

(* The variable environment keeps track of global and local variables, and keeps track of next available offset for local variables *)

type varEnv = (var * typ) env * int

(* The function environment maps function name to label and parameter decs *)

type paramdecs = (typ * string) list type funEnv = (label * typ option * paramdecs) env

(* Bind declared variable in env and generate code to allocate it: *)

let allocate (kind : int -> var) (typ, x) (varEnv : varEnv) : varEnv * instr list = let (env, fdepth) = varEnv match typ with | TypA (TypA ,) -> raise (Failure "allocate: array of arrays not permitted") | TypA (t, Some i) -> let newEnv = ((x, (kind (fdepth+i), typ)) :: env, fdepth+i+1) let code = [INCSP i; GETSP; CSTI (i-1); SUB] (newEnv, code) | _ -> let newEnv = ((x, (kind (fdepth), typ)) :: env, fdepth+1) let code = if typ=TypD then [NIL] else [CSTI 0] (newEnv, code)

(* Bind declared parameters in env: *)

let bindParam (env, fdepth) (typ, x) : varEnv = ((x, (Locvar fdepth, typ)) :: env , fdepth+1)

let bindParams paras ((env, fdepth) : varEnv) : varEnv = List.fold bindParam (env, fdepth) paras;

(* --- *)

(* Build environments for global variables and functions *)

let makeGlobalEnvs (topdecs : topdec list) : varEnv * funEnv * instr list = let rec addv decs varEnv funEnv = match decs with | [] -> (varEnv, funEnv, []) | dec::decr -> match dec with | Vardec (typ, var) -> let (varEnv1, code1) = allocate Glovar (typ, var) varEnv let (varEnvr, funEnvr, coder) = addv decr varEnv1 funEnv (varEnvr, funEnvr, code1 @ coder) | Fundec (tyOpt, f, xs, body) -> addv decr varEnv ((f, (newLabel(), tyOpt, xs)) :: funEnv) addv topdecs ([], 0) []

(* --- *)

(* Compiling micro-C statements:

- stmt is the statement to compile
- varenv is the local and global variable environment
- funEnv is the global function environment *)

let rec cStmt stmt (varEnv : varEnv) (funEnv : funEnv) : instr list = match stmt with | If(e, stmt1, stmt2) -> let labelse = newLabel() let labend = newLabel() cExpr e varEnv funEnv @ [IFZERO labelse] @ cStmt stmt1 varEnv funEnv @ [GOTO labend] @ [Label labelse] @ cStmt stmt2 varEnv funEnv @ [Label labend]
| While(e, body) -> let labbegin = newLabel() let labtest = newLabel() [GOTO labtest; Label labbegin] @ cStmt body varEnv funEnv @ [Label labtest] @ cExpr e varEnv funEnv @ [IFNZRO labbegin] | Expr e -> cExpr e varEnv funEnv @ [INCSP -1] | Block stmts -> let rec loop stmts varEnv = match stmts with | [] ->

(snd varEnv, []) | s1::sr -> let varEnv1, code1 = cStmtOrDec s1 varEnv funEnv let fdepthr, coder = loop sr varEnv1 (fdepthr, code1 @ coder) let (fdepthend, code) = loop stmts varEnv code @ [INCSP(snd varEnv - fdepthend)] | Return None -> [RET (snd varEnv - 1)] | Return (Some e) -> cExpr e varEnv funEnv @ [RET (snd varEnv)]

and cStmtOrDec stmtOrDec (varEnv : varEnv) (funEnv : funEnv) : varEnv * instr list = match stmtOrDec with | Stmt stmt -> (varEnv, cStmt stmt varEnv funEnv) | Dec (typ, x) -> allocate Locvar (typ, x) varEnv

(* Compiling micro-C expressions:

- e is the expression to compile
- varEnv is the local and gloval variable environment
- funEnv is the global function environment

Net effect principle: if the compilation (cExpr e varEnv funEnv) of expression e returns the instruction sequence instrs, then the execution of instrs will leave the rvalue of expression e on the stack top (and thus extend the current stack frame with one element).
*)

and cExpr (e : expr) (varEnv : varEnv) (funEnv : funEnv) : instr list = match e with | Access acc -> cAccess acc varEnv funEnv @ [LDI] | Assign(acc, e) -> cAccess acc varEnv funEnv @ cExpr e varEnv funEnv @ [STI] | CstI i -> [CSTI i] | CstN -> [NIL] | Addr acc -> cAccess acc varEnv funEnv | Prim1(ope, e1) -> cExpr e1 varEnv funEnv @ (match ope with | "!" -> [NOT] | "printi" -> [PRINTI] | "printc" -> [PRINTC] | "car" -> [CAR] | "cdr" -> [CDR] | "createStack" -> [CREATESTACK] | "popStack" -> [POPSTACK] | "printStack" -> [PRINTSTACK] | _ -> raise (Failure "unknown primitive 1")) | Prim2(ope, e1, e2) -> cExpr e1 varEnv funEnv @ cExpr e2 varEnv funEnv @ (match ope with | "*" -> [MUL] | "+" -> [ADD] | "-" -> [SUB] | "/" -> [DIV] | "%" -> [MOD] | "==" -> [EQ] | "!=" -> [EQ; NOT] | "<" -> [LT] | ">=" -> [LT; NOT] | ">" -> [SWAP; LT] | "<=" -> [SWAP; LT; NOT] | "cons" -> [CONS] | "setcar" -> [SETCAR] | "setcdr" -> [SETCDR] | "pushStack" -> [PUSHSTACK] | _ -> raise (Failure "unknown primitive 2")) | Andalso(e1, e2) -> let labend = newLabel() let labfalse = newLabel() cExpr e1 varEnv funEnv @ [IFZERO labfalse] @ cExpr e2 varEnv funEnv @ [GOTO labend; Label labfalse; CSTI 0; Label labend]
| Orelse(e1, e2) -> let labend = newLabel() let labtrue = newLabel() cExpr e1 varEnv funEnv @ [IFNZRO labtrue] @ cExpr e2 varEnv funEnv @ [GOTO labend; Label labtrue; CSTI 1; Label labend] | Call(f, es) -> callfun f es varEnv funEnv

(* Generate code to access variable, dereference pointer or index array. The effect of the compiled code is to leave an lvalue on the stack. *)

and cAccess access varEnv funEnv : instr list = match access with | AccVar x -> match lookup (fst varEnv) x with | Glovar addr, _ -> [CSTI addr] | Locvar

addr, _ -> [GETBP; CSTI addr; ADD] | AccDeref e -> cExpr e varEnv funEnv | AccIndex(acc, idx) -> cAccess acc varEnv funEnv @ [LDI] @ cExpr idx varEnv funEnv @ [ADD]

(* Generate code to evaluate a list es of expressions: *)

and cExprs es varEnv funEnv : instr list = List.concat(List.map (fun e -> cExpr e varEnv funEnv) es)

(* Generate code to evaluate arguments es and then call function f: *)

and callfun f es varEnv funEnv : instr list = let (labf, tyOpt, paramdecs) = lookup funEnv f let argc = List.length es if argc = List.length paramdecs then cExprs es varEnv funEnv @ [CALL(argc, labf)] else raise (Failure (f + ": parameter/argument mismatch"))

(* Compile a complete micro-C program: globals, call to main, functions *)

let cProgram (Prog topdecs) : instr list = let _ = resetLabels () let ((globalVarEnv, *), funEnv, globalInit) = makeGlobalEnvs topdecs let compilefun (tyOpt, f, xs, body) = let (labf, _, paras) = lookup funEnv f let (envf, fdepthf) = bindParams paras (globalVarEnv, 0) let code = cStmt body (envf, fdepthf) funEnv [Label labf] @ code @ [RET (List.length paras-1)] let functions = List.choose (function | Fundec (rTy, name, argTy, body) -> Some (compilefun (rTy, name, argTy, body)) | Vardec _ -> None) topdecs let (mainlab, _, mainparams) = lookup funEnv "main" let argc = List.length mainparams globalInit @ [LDARGS; CALL(argc, mainlab); STOP] @ List.concat functions

(* Compile a complete micro-C and write the resulting instruction list to file fname; also, return the program as a list of instructions. *)

let intsToFile (inss : int list) (fname : string) = File.WriteAllText(fname, String.concat " " (List.map string inss))

let compileToFile program fname = let instrs = cProgram program let bytecode = code2ints instrs (intsToFile bytecode fname; instrs)

(* Example programs are found in the files ex1.c, ex2.c, etc *)