Tiger frontend compiler for `C--`

Paul Govereau

January 29, 2007

Contents

Chapter 1

Introduction

This document describes the source code for a Tiger language front end to the Quick `C--` compiler[1]. The Tiger language is described in "Modern Compiler Implementation in ML"[**?**]. To quote from Appendix A of this book:

> The Tiger language is a small language with nested functions, record values with implicit pointers, arrays, integer and string variables, and a few simple structured control constructs.

In addition, we have added exceptions to the basic Tiger language. Full source code and documentation can be found at the `C--` website, or in the Quick `C--` CVS repository under `frontends/tiger`.

1.1 Guide to the Source Code

The sources are dived into several modules, each of which is described below. Figure **??** shows the dependencies between the major modules in the compiler.

Abstract Syntax The abstract syntax for Tiger programs is described in the `Ast` module. Symbols and tables of symbols are implemented in a companion module `Symbol`. As usual, Tiger source programs are converted into abstract syntax by the parser. The abstract syntax is then used by the rest of the compiler as the representation of programs.

symbol.nw	Symbols and symbol tables.
ast.nw	Abstract Syntax Trees.

Analysis The analysis phase verifies that the abstract syntax tree represents a semanticly correct program. This process includes type checking and verifying

[1] See URL `http://www.cminusminus.org`.

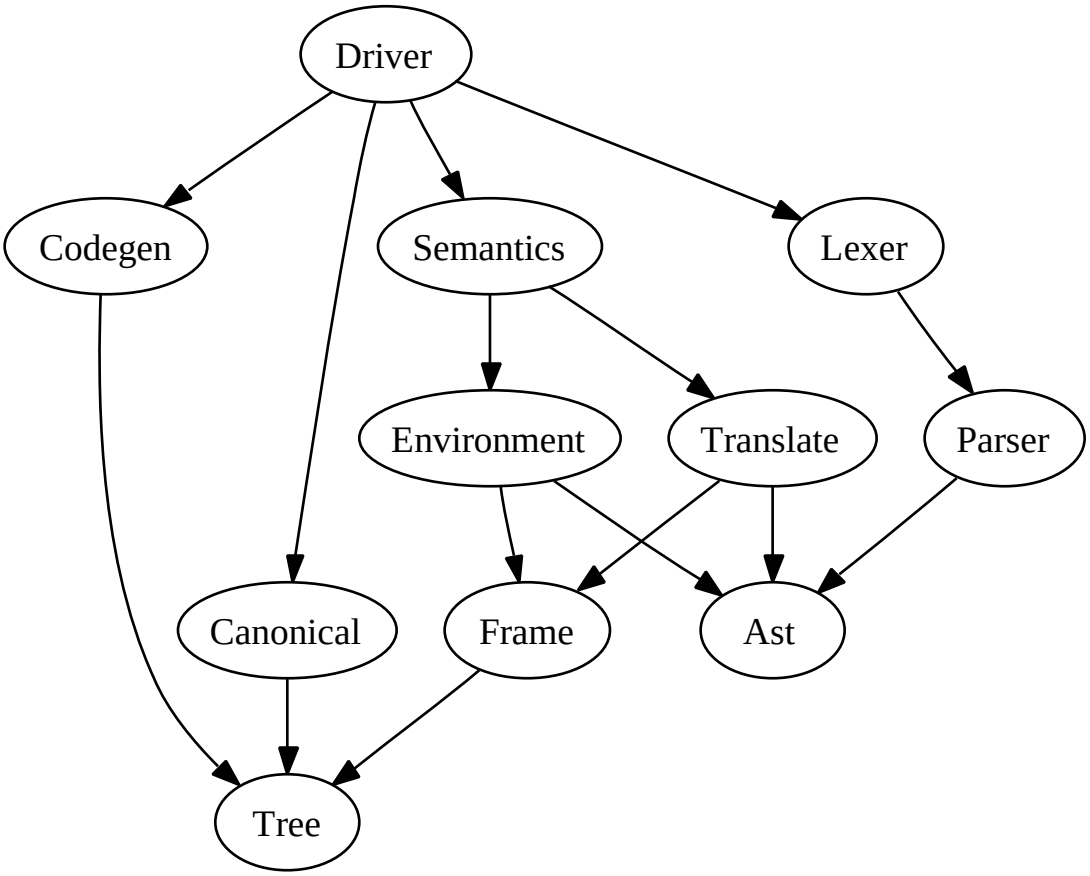

Figure 1.1: Module Dependencies

that expressions are used in a proper context (e.g. a `break` can only occur within a loop). The semantic analysis uses an environment to keep track of type definitions, variable bindings and other information. The environment is implemented in the `Environment` module, and the analysis is implemented in the `Semantics` module.

environment.nw	Environments.
semantics.nw	Semantic analysis.

Translation The `Translate` module translates an abstract syntax tree into the intermediate representation used by the rest of the compiler. The intermediate representation is defined in the `Tree` module.

tree.nw	The intermediate representation.
translate.nw	Translates ASTs to intermediate representation.

Code Generation The code generator translates the intermediate representation into a `C--` program. Each Tiger function is translated into a `C--` function by the Tiger compiler. The stack frame for each function is managed by the `Frame` module. The `Codegen` module outputs `C--` programs from the intermediate representation and information contained in the `Frame` module.

frame.nw	Stack frames.
codegen.nw	Generates `C--` code.

Tiger Runtime System Compiled tiger programs must be linked with the tiger runtime system to create a complete program. The runtime system contains the startup code, a simple copying garbage collector, and a library of standard functions.

gc.nw	Garbage collector.
stdlib.nw	Tiger standard library functions.
runtime.nw	Startup code.

Other Modules The Tiger compiler also contains several support modules. These modules are listed below—the complete sources can be found in the appendices.

driver.nw	The compiler driver.
error.nw	Error reporting.
option.nw	Command line options.
parser.nw	Parser and scanner.
canonical.nw	Linearizes expression trees.

Chapter 2

Abstract Syntax of Tiger

2.1 Symbols

The `Symbol` module provides the representation of symbols and an implementation of symbol tables.

2.1.1 Basic Symbols

Symbols are created from strings found in the source program. The interface to symbols is given below. A new symbol can be created from a string. The `new_symbol` function ensures that the newly created symbol is unique.

⟨*symbol.mli*⟩≡

```
type symbol

val uid        : symbol -> int
val name       : symbol -> string
val symbol     : string -> symbol
val new_symbol : string -> symbol
```

Symbols are implemented as stamped strings—a pair containing a string and an int. The integer is used to identify the symbol. All symbols are held in a hash-table implemented by the standard library `Hashtbl` module.

⟨*symbol.ml*⟩≡
```
type symbol = string * int

let nextsym = ref 0
let hashtable = Hashtbl.create 128

let name = fst
let uid  = snd

let symbol name =
  try let s = Hashtbl.find hashtable name in (name,s)
  with Not_found ->
    incr nextsym; Hashtbl.add hashtable name !nextsym;
    (name, !nextsym)

let new_symbol s = symbol (Printf.sprintf "%s_%d" s !nextsym)
```

2.1.2 Symbol Tables

A symbol table is a mapping from symbols to values. In addition, symbol tables support "scoping" in that each table has a parent table. If a symbol is not found in the a table, then the parent table will be consulted—each child table hide the definitions of its parent.

A symbol table is created with a list of key-value pairs for the initial table. Symbols can be inserted and queried up with the `enter` and `look` functions, and a nested scope can be created with the `new_scope` function.

⟨*symbol.mli*⟩+≡
```
type 'a table

val enter  : 'a table -> symbol -> 'a -> unit
val look   : 'a table -> symbol -> 'a
val mem    : 'a table -> symbol -> bool
val create : (string * 'a) list -> 'a table

val new_scope : 'a table -> 'a table
```

The `iter` and `fold` functions perform the usual iteration and folding over a symbol table and its parents. The first parameter to the supplied functions gives to nesting level in which the current symbol-value pair lives. The outermost level is 0.

⟨*symbol.mli*⟩+≡
```
val iter : (int -> symbol -> 'a -> unit) -> 'a table -> unit
val fold : (int -> symbol -> 'a -> 'b -> 'b) -> 'a table -> 'b -> 'b
```

Symbol tables are implemented using the standard library `Hashtbl` module. Duplicate entries in the same table are disallowed.

⟨*symbol.ml*⟩+≡
```
type 'a table = {
    level  : int;
    tbl    : (symbol, 'a) Hashtbl.t;
    parent : 'a table option
  }

let enter env s v =
  if Hashtbl.mem env.tbl s
  then failwith "Compiler error: symbol table duplicate entry"
  else Hashtbl.add env.tbl s v
```

If a symbol is not found in a given symbol table, then its parent table is checked. If there is no parent table, then `Not_found` is raised.

⟨*symbol.ml*⟩+≡
```
let rec look env s =
  try Hashtbl.find env.tbl s
  with Not_found -> match env.parent with
    None -> raise Not_found
  | Some e -> look e s
```

The `mem` function can be used to determine if a symbol is defined in the current nesting level.

⟨*symbol.ml*⟩+≡
```
let mem env = Hashtbl.mem env.tbl
```

Symbol tables are created from a list of initial mappings. All symbol tables are created with an initial nesting level of zero. As new nested scopes are created the level increases by one for each level.

⟨*symbol.ml*⟩+≡
```
let create l =
  let env = {
    level = 0;
    tbl = Hashtbl.create 20;
    parent = None
  } in
  List.iter (fun (key, data) -> enter env (symbol key) data) l;
  env

let new_scope env = { level = env.level + 1;
                      tbl = Hashtbl.create 20;
                      parent = Some env }
```

Both `iter` and `fold` are implemented by lifting the corresponding `Hashtbl` functions up to our representation of symbol tables.

⟨*symbol.ml*⟩+≡
```
let rec iter f env =
  Hashtbl.iter (f env.level) env.tbl;
  match env.parent with
    None -> ()
  | Some e -> iter f e

let rec fold f env init =
  let result = Hashtbl.fold (f env.level) env.tbl init in
  match env.parent with
    None -> result
  | Some e -> fold f e result
```

2.2 Abstract Syntax

The `Ast` module describes the types used to represent the abstract syntax of a Tiger program. The `print_tree` function will print a string representation of an abstract syntax tree for debugging purposes.

⟨*ast.mli*⟩≡
```
⟨types⟩
val print_tree : exp -> unit
```

⟨*ast.ml*⟩≡
```
⟨types⟩
⟨tree printer⟩
```

2.2.1 Ast Types

Symbols are defined in the `Symbol` module (Section **??**), and positions within the source file are integers representing a byte offset.

⟨*types*⟩≡
```
type pos    = int
and  symbol = Symbol.symbol
```

The Tiger language allows declarations of functions, types, and variables. A function declaration may have a return type, but it is optional. Variable declaration also have an optional type.

⟨*types*⟩+≡
```
type dec =
    FunctionDec  of (symbol * field list * symbol option * exp * pos) list
  | VarDec       of symbol * symbol option * exp * pos
  | TypeDec      of (symbol * ty * pos) list
  | ExceptionDec of symbol * pos
```

A type declaration may be either an alias (reference to a previously declared type), a record declaration, or an array declaration.

⟨*types*⟩+≡
```
  and ty =
      NameTy   of symbol * pos
    | RecordTy of field list
    | ArrayTy  of symbol * pos
```

Record and function definitions both take parameters of the form `name : type` where both `name` and `type` are symbols. These `name-type` pairs are represented by the Ocaml type `field`.

⟨*types*⟩+≡
```
  and field = (symbol * symbol * pos)
```

A variable expressions is either a simple variable name, a field variable for referencing members of a record type, or a subscript variable for accessing the elements of an array.

⟨*types*⟩+≡
```
  and var =
      SimpleVar    of symbol * pos
    | FieldVar     of var * symbol * pos
    | SubscriptVar of var * exp * pos
```

Expressions in Tiger are described by the `exp` type below. The `ArrayExp` and `RecordExp` types are used for anonymous arrays and records respectively. The first element of both the `ArrayExp` and `RecordExp` types is a reference to an already existing type.

⟨*types*⟩+≡
```
  and exp =
      NilExp
    | VarExp    of var
    | IntExp    of int
    | StringExp of string * pos
    | RecordExp of symbol * (symbol * exp * pos) list * pos
    | ArrayExp  of symbol * exp * exp * pos
    | AssignExp of var * exp * pos
    | OpExp     of exp * oper * exp * pos
    | CallExp   of symbol * exp list * pos
    | IfExp     of exp * exp * exp option * pos
    | WhileExp  of exp * exp * pos
    | ForExp    of symbol * exp * exp * exp * pos
    | BreakExp  of pos
    | SeqExp    of exp list * pos
    | LetExp    of dec list * exp * pos
    | TryExp    of exp * (symbol * exp * pos) list * pos
    | RaiseExp  of symbol * pos
    | SpawnExp  of symbol * pos
```

Finally, the valid operators in the Tiger language are `+,-,*,/,=,<>,<,<=,>,` and `>=`.

⟨*types*⟩+≡
```
and oper = PlusOp | MinusOp | TimesOp | DivideOp
         | EqOp | NeqOp | LtOp | LeOp | GtOp | GeOp
```

2.2.2 Abstract Syntax Tree Printer

The `Ast` module provides a tree printing function for debugging purposes. The abstract syntax tree can be printed by specifying the appropriate command line arguments (see the `Driver` module in Section **??**).

The code for the AST printer is shown below. The `iprintf` function is an indenting printf which will indent a number of spaces before calling `printf`, and the `opname` function returns a printable string representation for a given operator type.

⟨*tree printer*⟩≡
```
module S = Symbol

let iprintf d fmt =
  let rec indent = function
      0 -> ()
    | i -> (print_string "  "; indent(i-1))
  in (indent d; Printf.printf fmt)

let opname = function
    PlusOp   -> "PlusOp"
  | MinusOp  -> "MinusOp"
  | TimesOp  -> "TimesOp"
  | DivideOp -> "DivideOp"
  | EqOp     -> "EqOp"
  | NeqOp    -> "NeqOp"
  | LtOp     -> "LtOp"
  | LeOp     -> "LeOp"
  | GtOp     -> "GtOp"
  | GeOp     -> "GeOp"
```

The tree printer itself consists of four functions for printing declarations, types, variables, and expressions. Each Tiger program is an expression, and an outer function starts the process by calling the expression printer with an initial indenting level of 0.

⟨*tree printer*⟩+≡
```
let print_tree expression =
  ⟨declaration printer⟩
  ⟨type printer⟩
  ⟨variable printer⟩
  ⟨expression printer⟩
in exp 0 expression
```

⟨*declaration printer*⟩≡
```
let rec dec d =
  let print_opt_sym d = function
      None   -> iprintf (d+1) ": NONE\n"
    | Some s -> iprintf (d+1) ": SOME(%s)\n" (S.name s)
  in
  function
    FunctionDec functions ->
      let prfield d (n,t,_) =
        iprintf d "%s:%s\n" (S.name n) (S.name t)
      in
      let prfun d (name, params, type', body, _) =
        iprintf d "%s:\n" (S.name name);
        List.iter (prfield (d+1)) params;
        print_opt_sym (d+1) type';
        exp (d+2) body
      in
      iprintf d "FunctionDec:\n";
      List.iter (prfun (d+1)) functions
  | VarDec(name, type', init,_) ->
      iprintf d "VarDec: %s\n" (S.name name);
      print_opt_sym (d+1) type';
      exp (d+1) init
  | TypeDec types ->
      let prtdec d (name, type',_) =
        iprintf d "%s:\n" (S.name name); ty (d+1) type'
      in
      iprintf d "TypeDec:\n";
      List.iter (prtdec (d+1)) types
  | ExceptionDec(s,_) ->
      iprintf d "ExceptionDec:%s\n" (S.name s)
```

⟨*type printer*⟩≡

```
and ty d = function
    NameTy(s,_)     -> iprintf d "NameTy : %s\n" (S.name s)
  | ArrayTy(s,_)    -> iprintf d "ArrayTy: %s\n" (S.name s)
  | RecordTy fields ->
      let f d (n,t,_) =
        iprintf d "%s:%s\n" (S.name n) (S.name t)
      in
      iprintf d "RecordTy:\n";
      List.iter (f (d+1)) fields
```

⟨*variable printer*⟩≡

```
and var d = function
    SimpleVar(s,_)      -> iprintf d "SimpleVar: %s\n" (S.name s)
  | FieldVar(v,s,_)     -> iprintf d "FieldVar:\n";
                           var (d+1) v;
                           iprintf (d+1) "%s\n" (S.name s)
  | SubscriptVar(v,e,_) -> iprintf d "SubscriptVar:\n";
                           var (d+1) v;
                           exp (d+1) e
```

⟨*expression printer*⟩≡

```
and exp d = function
    VarExp v       -> var d v
  | NilExp         -> iprintf d "NilExp\n"
  | IntExp i       -> iprintf d "IntExp: %d\n" i
  | StringExp(s,_) -> iprintf d "StringExp:%s\n" (String.escaped s)
  | RecordExp(name, fields, _) ->
      let f d (n,e,_) =
        (iprintf d "%s:\n" (S.name n); exp (d+1) e)
      in
      iprintf d "RecordExp: %s\n" (S.name name);
      List.iter (f (d+1)) fields
  | ArrayExp(v, size, init, p) ->
      iprintf d "ArrayExp: %s\n" (S.name v);
      exp (d+1) size;
      exp (d+1) init
  | AssignExp(v, e, _) ->
      iprintf d "AssignExp:\n";
      var (d+1) v;
      exp (d+1) e
  | OpExp(left, oper, right, _) ->
      iprintf d "OpExp:%s\n" (opname oper);
      exp (d+1) left;
      exp (d+1) right
  | CallExp(name, args, _) ->
      iprintf d "CallExp: %s\n" (S.name name);
      List.iter (exp (d+1)) args
  | IfExp(if', then', else', _) ->
      iprintf d "IfExp:\n";
      exp (d+1) if';
      exp (d+1) then';
      begin match else' with
        None   -> ()
      | Some a -> exp (d+1) a
      end
  | WhileExp(test, body, _) ->
      iprintf d "WhileExp:\n";
      exp (d+1) test;
      exp (d+1) body
  | ForExp(var, lo, hi, body, _) ->
      iprintf d "ForExp: %s\n" (S.name var);
      exp (d+1) lo;
      exp (d+1) hi;
      exp (d+1) body
  | BreakExp _ ->
      iprintf d "BreakExp\n"
```

```
| SeqExp(l, _) ->
    iprintf d "SeqExp:\n"; List.iter (exp (d+1)) l
| LetExp(decs, body, _) ->
    iprintf d "LetExp:\n";
    List.iter (dec (d+1)) decs;
    iprintf d "IN:\n";
    exp (d+2) body
| TryExp(expr, handlers, _) ->
    iprintf d "TryExp:\n";
    exp (d+1) expr;
    List.iter
      (fun (n,h,_) -> iprintf (d+2) "%s:\n" (S.name n); exp (d+2) h)
      handlers
| RaiseExp(name,_) ->
    iprintf d "RaiseExp %s\n" (S.name name)
| SpawnExp(name,_) ->
    iprintf d "SpawnExp %s\n" (S.name name)
```

Chapter 3

Analysis

3.1 Environments

The `Environment` module provides the environment structure that is used during type checking (Section **??**). For each Tiger function, there is an associated environment containing variable definitions, function definitions, and other information. An environment is made up of: a table of type definitions, a table of variable names and their defined types, the list of defined exceptions, the current procedure stack frame, the current break label, and the current exception handler label.

⟨*envtype*⟩≡
```
type 'a t = {
    tenv        : 'a Symbol.table;
    venv        : 'a enventry Symbol.table;
    xenv        : int Symbol.table;
    frame       : Frame.frame;
    break_label : Tree.label option;
    exn_label   : Tree.label option
  }
```

The environment type `'a t` is polymorphic—any Ocaml type can be used to represent Tiger types. The type variable `'a` is instantiated with the representation of Tiger types defined in Section **??**.

A name can be bound to either a variable or a function in the environment. For variables, we keep information about the location in the current stack frame and the type. For functions, we keep the function name, an optional calling convention, the stack frame, the parameter types, and the return type.

⟨*enventry*⟩≡
```
type 'a enventry =
    VarEntry of (Frame.access * 'a)
  | FunEntry of (Symbol.symbol * string option * Frame.frame * 'a list * 'a)
```

3.1.1 Interface to the Environment

The environment is a mostly functional data structure. However, the symbol tables representing type definitions and variable types are mutated in place. Therefore, most operations return a new environment, but adding new type definitions and variable bindings do not.

The environment types are exposed in the interface.

⟨*environment.mli*⟩≡
  ⟨*enventry*⟩
  ⟨*envtype*⟩

An initial environment can be generated using the `new_env` function. The new environment will be populated with an initial set of type definitions and functions. All new environments are automatically given a new stack frame.

⟨*environment.mli*⟩+≡
```
val new_env : (string * 'a) list ->
              (string * string option * 'a list * 'a) list -> 'a t
```

There are three functions for manipulating the variable scope and the current stack frame. Creating a new stack frame automatically creates a new variable scope.

⟨*environment.mli*⟩+≡
```
val new_scope : 'a t -> 'a t
val frame     : 'a t -> Frame.frame
val new_frame : 'a t -> Symbol.symbol -> 'a t
```

The environment provides functions for looking up types, values and exception identifiers. Calling a lookup function for a symbol that does not exist indicates that the original source has an error (use of undefined symbol). Each lookup function takes an `Ast.pos` argument that is used to report errors.

⟨*environment.mli*⟩+≡
```
val lookup_type  : 'a t -> Symbol.symbol -> Ast.pos -> 'a
val lookup_value : 'a t -> Symbol.symbol -> Ast.pos -> 'a enventry
val lookup_exn   : 'a t -> Symbol.symbol -> Ast.pos -> int
```

New types, values, and exceptions can be added to an environment using the enter functions. There are five functions for entering type definitions, exceptions, function definitions, formal parameters, and local variables. Both `enter_param` and `enter_local` take a boolean parameter that is true if the new local or parameter will hold a pointer value.

⟨*environment.mli*⟩+≡
```
val enter_type  : 'a t -> Symbol.symbol -> 'a -> unit
val enter_exn   : 'a t -> Symbol.symbol -> unit
val enter_fun   : 'a t -> Symbol.symbol -> string option ->
                  'a list -> 'a  -> 'a t
val enter_param : 'a t -> Symbol.symbol -> 'a -> bool -> unit
val enter_local : 'a t -> Symbol.symbol -> 'a -> bool -> Frame.access
```

The environment is used to track the current break label for exiting from loops. The `break_label` function will raise `Not_Found` if there is no current break label.

⟨*environment.mli*⟩+≡
```
val break_label     : 'a t -> Tree.label
val new_break_label : 'a t -> 'a t
```

Similar to the break label, the environment also tracks the current exception label. The exception label can be `None` indicating that there is no current exception handler.

⟨*environment.mli*⟩+≡
```
val exn_label     : 'a t -> Tree.label option
val new_exn_label : 'a t -> 'a t
```

3.1.2 Implementation of Environments

⟨*environment.ml*⟩≡
```
module E = Error
module S = Symbol
module F = Frame
module T = Tree
⟨enventry⟩
⟨envtype⟩
```

To create a new environment, we construct a record that contains the initial types and functions, and a base frame.

⟨*environment.ml*⟩+≡
```
let new_env types funs =
  let mkfe (n,cc,a,r) = (n, FunEntry(S.symbol n,cc,F.base_frame,a,r))
  in { tenv        = Symbol.create types;
       venv        = Symbol.create (List.map mkfe funs);
       xenv        = Symbol.create [];
       frame       = F.new_frame (S.symbol "tiger_main") F.base_frame;
       break_label = None;
       exn_label   = None }
```

To create a new variable scope, we update the symbol tables.

⟨*environment.ml*⟩+≡
```
let new_scope env = { env with
                      tenv = S.new_scope env.tenv;
                      venv = S.new_scope env.venv }
```

Creating a new frame requires updating the frame and removing the current exception label in addition to updating the symbol tables.

⟨*environment.ml*⟩+≡
```
let frame env = env.frame
let new_frame env sym = { env with
                          tenv = S.new_scope env.tenv;
                          venv = S.new_scope env.venv;
                          frame = Frame.new_frame sym env.frame;
                          exn_label = None }
```

The lookup functions use the symbol table (Section **??**) implementation to lookup values in the various tables. If a symbol is not found, then we raise an undefined symbol error.

⟨*environment.ml*⟩+≡
```
let lookup env sym pos =
  try S.look env sym
  with Not_found ->
    raise(E.Error(E.Undefined_symbol (S.name sym), pos))

let lookup_type  env = lookup env.tenv
let lookup_value env = lookup env.venv
let lookup_exn   env = lookup env.xenv
```

Entering new types, exceptions, and values into the environment is also implemented with the symbol table functions. In this case, we may raise a duplicate symbol error if a name is defined twice in a source file.

⟨*environment.ml*⟩+≡
```
let enter tbl sym v =
  if S.mem tbl sym
  then raise(E.Error(E.Duplicate_symbol (S.name sym), 0))
  else S.enter tbl sym v

let enter_type env = enter env.tenv
```

For exceptions, we need to generate a unique identifier for each exception. For this, we just use the unique identifier associated with the symbol.

⟨*environment.ml*⟩+≡
```
let enter_exn env sym = enter env.xenv sym (S.uid sym)
```

For function definitions, we generate a fresh label based on the function name, and a new environment for the function parameters and local variables. After entering the function definition in the environment, we return the new function environment to the caller.

⟨*environment.ml*⟩+≡
```
let enter_fun env sym cc args result =
  let lbl = S.new_symbol (S.name sym) in
  let fenv = new_frame env lbl in
  let fe = FunEntry (lbl, cc, fenv.frame, args, result) in
  enter env.venv sym fe;
  fenv
```

When a formal function parameter is entered in to the environment, a new slot is allocated in the current frame to hold the parameter.

⟨*environment.ml*⟩+≡
```
let enter_param env sym typ ptr =
  let acc = F.alloc_param env.frame sym ptr in
  enter env.venv sym (VarEntry(acc,typ))
```

When a new local variable is entered into the environment, we first allocate space in the current frame to hold a new temporary and then enter it into the value table.

⟨*environment.ml*⟩+≡
```
let enter_local env sym typ ptr =
  let acc = F.alloc_local env.frame sym ptr in
  enter env.venv sym (VarEntry(acc, typ));
  acc
```

To create a new break label, we generate a new label and store it in the environment. When the break label is requested, the `break_label` function will raise `Not_Found` if there is no current label.

⟨*environment.ml*⟩+≡
```
let break_label env =
  match env.break_label with
    None      -> raise Not_found
  | Some(lbl) -> lbl

let new_break_label env =
  { env with break_label = Some(T.new_label "loop_end") }
```

The implementation for exception labels is similar. However, the `exn_label` function will not raise an exception if there is no current exception label. Instead, this function returns an option type.

⟨*environment.ml*⟩+≡
```
let exn_label     env = env.exn_label
let new_exn_label env =
  { env with exn_label = Some(T.new_label "exn") }
```

3.2 Semantic Analysis

The Tiger compiler performs type-checking and translation to intermediate representation in a single pass. The `Semantics` module type checks each Tiger expressions and the passes each one to the `Translate` module (Section **??**) where it is translated to the intermediate representation.

The interface to the `Semantic` module consist of a single function, `translate`. The `translate` function takes an initial environment and an abstract syntax tree as input, and produces a list of Tiger functions. Each Tiger function consists of a stack frame (Section **??**), and an expression tree (Section **??**)—the intermediate representation of the function body.

⟨*semantics.mli*⟩≡
```
⟨variable types⟩
val translate : vartype Environment.t -> Ast.exp ->
                (Frame.frame * Translate.exp) list
```

The `Semantics` module implementation consists of a definition of the type system used internally in the compiler, the `translate` function entry point, and a private translator function for each type of `Ast` node.

⟨*semantics.ml*⟩≡
```
module E = Error
module A = Ast
module S = Symbol
module V = Environment
module T = Translate
⟨variable types⟩
⟨type system⟩
⟨translators⟩
⟨entry point⟩
```

3.2.1 Type System

The set of types used internally by the compiler is given below. The `NAME` type is an alias to another defined type. The `ANY` type can be used as a type variable when constructing polymorphic types. The `ANY` type is only used to define library functions.

⟨*variable types*⟩≡
```
type vartype =
    UNIT
  | NIL
  | INT
  | STRING
  | ARRAY  of vartype
  | RECORD of (Symbol.symbol * vartype) list
  | NAME   of Symbol.symbol
  | ANY
```

The `type_name` function will return a string representation of a type. This function is used by to report type errors.

⟨*type system*⟩≡
```
let rec type_name = function
    RECORD l -> (List.fold_left (fun x y -> x ^ (type_name (snd y)))
                "record {" l) ^ "}"
  | NIL      -> "nil"
  | INT      -> "int"
  | STRING   -> "string"
  | ARRAY vt -> "array of " ^ (type_name vt)
  | NAME(s)  -> "named type " ^ (S.name s)
  | UNIT     -> "unit"
  | ANY      -> "any"
```

The internal type system allows type aliases by way of the `NAME` type. We often want to get the concrete base type for a type alias. The `base_type` function will lookup the definition of a `NAME` type and return the concrete base type. The return value is guaranteed to be a base type (not a `NAME` type).

To compute a base type, we keep looking up the definitions of `NAME` types until we reach a base type. If a `NAME` type references a type that is not defined, then there is something wrong with the compiler and we give up.

⟨*type system*⟩+≡
```
let rec base_type env = function
    NAME s ->
      begin
        try base_type env (V.lookup_type env s 0)
        with Not_found -> E.internal "NAME symbol not found"
      end
  | x -> x

let lookup_base_type env sym pos =
  base_type env (V.lookup_type env sym pos)
```

For code generation we will be interested in partitioning the internal types into two classes: those that are represented by a pointer to a data structure (e.g. arrays), and those that are represented by a single machine word (e.g. integers). The `is_ptr` function returns true of the supplied type is represented as a pointer and false otherwise.

The types `INT` and `UNIT` are represented as machine words, all others are pointers to data structures. The compiler should not call the `is_ptr` function with a `NAME` or and `ANY` type.

⟨*type system*⟩+≡
```
let is_ptr = function
    INT | UNIT -> false
  | NIL | RECORD _ | STRING | ARRAY _ -> true
  | _ -> E.internal "non-base type for variable"
```

During type-checking, we commonly want to assert that an expression either has type `int` or type `unit`.

⟨*type system*⟩+≡
```
let check_type_t ty pos msg typ =
  if typ <> ty
  then E.type_err pos (msg ^ " must be of type " ^ type_name ty)
let check_type_int  = check_type_t INT
let check_type_unit = check_type_t UNIT
```

Similarly, we often want to assert that two types are equivalent. In Tiger, `nil` is the type of an uninitialized record. Therefore, the nil and record types are equivalent for the purposes of type-checking.

The `ANY` type can only be used in a few restricted ways. The `ANY` type is completely internal to the compiler, and it only has as much power as we need to define the initial basis.

⟨*type system*⟩+≡
```
let check_type_eq pos msg t1 t2 =
  let check = match (t1,t2) with
                (RECORD _,NIL)
              | (NIL,RECORD _)
              | (ARRAY ANY, ARRAY _)
              | (ARRAY _, ARRAY ANY) -> false
              | _                    -> true
  in if check && t1 <> t2 then
    E.type_err pos (Printf.sprintf msg (type_name t1) (type_name t2))
```

3.2.2 The `translate` Entry Point

The `translate` function returns a list of function bodies and their associated frames. The functions are held in a mutable reference to a list. When a new expression is added to the list, we first call the `Translate` module (Section **??**) to convert the expression into a function body.

⟨*translators*⟩≡
```
let functions                = ref []
let get_functions ()         = List.rev !functions
let add_function frm (ex,typ) =
  functions := (frm, T.func frm ex (is_ptr typ)) :: !functions
```

There are three private translation functions for handling declarations, variables, and expressions. All of these are enclosed in the function `trans`. The `trans` function can only be called with an expression or declaration.

⟨*translators*⟩+≡
```
type ast_node = DEC of Ast.dec | EXP of Ast.exp
let rec trans (env : vartype V.t) (node : ast_node) =
  ⟨declaration translator⟩
  ⟨variable translator⟩
  ⟨expression translator⟩
in match node with
  DEC d -> (trdec d, NIL)
| EXP e -> trexp e
```

The entry point for the `Semantics` module is the `translate` function shown below.

⟨*entry point*⟩≡
```
let translate env ast =
  begin
    let (mainex,mainty) = trans env (EXP ast) in
    if mainty <> INT && mainty <> UNIT then
      E.type_err 0 "tiger program must return INT or UNIT"
    else ();
    add_function (V.frame env) (mainex,mainty);
    get_functions()
  end
```

3.2.3 Declarations

The `trdec` function translates declarations of functions, variables, and types.

⟨*declaration translator*⟩≡
```
let rec trdec = function
  ⟨function declarations⟩
  ⟨variable declarations⟩
  ⟨type declarations⟩
  ⟨exception declarations⟩
```

Function Declarations Function declarations in tiger can be mutually recursive if the functions are declared together in the same block. To translate a list of function declarations, we first add each function signature to the current environment, and then translate the function bodies. The translated function bodies are then added to the function list.

The `mk_func_env` function enters a function signature into the variable environment, and creates a new nested environment for evaluating the function body.

⟨*function declarations*⟩≡

```
A.FunctionDec functions ->
  let mk_param fenv (name, typ, pos) =
    let t = lookup_base_type env typ pos in
    V.enter_param fenv name t (is_ptr t)
  in
  let mk_func_env (name, params, typ, _, pos) =
    let ret_type = match typ with
      Some x -> lookup_base_type env x pos
    | None   -> UNIT
    and types = List.map (fun(_,t,p)-> lookup_base_type env t p) params in
    let fenv  = V.enter_fun env name None types ret_type in
    List.iter (mk_param fenv) params;
    fenv
  in
```

After all of the function signatures have been added to the environment, the bodies of the functions can be evaluated. `trans_func` evaluates a function body in the given environment and adds the translated body to the function list.

⟨*function declarations*⟩+≡

```
  let trans_func fenv (_, _, _, body, _) =
    let b = trans fenv (EXP body) in
    add_function (V.frame fenv) b
  in
```

For each function, we must call `mk_func_env` to create a new environment, and then `trans_func` to translate the function body. As mentioned above, we need enter all of the function signatures into the new environment before processing the function bodies.

⟨*function declarations*⟩+≡

```
  let envs = (List.map mk_func_env functions) in
  List.iter2 trans_func envs functions;
  T.nil
```

Variable Declarations To translate a variable declaration, first we evaluate the initializing expression in the current environment. Then, we check that the type of the initial expression is compatible with the declared type of the variable. If the types are compatible, we enter the variable definition into the environment, allocate a new local on the current frame, and initialize it.

⟨*variable declarations*⟩≡

```
| A.VarDec(name, typ, init, pos) ->
    let e,t = trexp init in
    begin match typ with
      Some x -> check_type_eq pos
          "Variable of type %s cannot be initialized with type %s"
          (V.lookup_type env x pos) t
    | None -> ()
    end;
    let acc = V.enter_local env name t (is_ptr t) in
    T.assign (T.simple_var (V.frame env) acc) e
```

Type Declarations Similar to functions, types may have mutually recursive definitions. To translate a list of type declarations, first a new scope is created that will later be discarded when all of the types have been checked. Each type name is entered into the new environment as an alias to itself. Technically, this is an invalid type definition. However, as long as we do not try to get the base type, it is a good placeholder for the real type. After all of the types are checked in the new environment, we enter the real type definitions into the current environment and discard the new environment.

⟨*type declarations*⟩≡

```
| A.TypeDec types ->
    let penv = V.new_scope env in
    let real_type (name, typ, _) = (name, match typ with
      A.NameTy(name, pos) -> V.lookup_type penv name pos
    | A.RecordTy(fields) ->
        let chkfld(name,ty,p) = (name,(V.lookup_type penv ty p))
        in RECORD (List.map chkfld fields)
    | A.ArrayTy(name, pos) -> ARRAY (V.lookup_type penv name pos))
    in
    List.iter (fun(n,_,_) -> V.enter_type penv n (NAME n)) types;
    let real_types = (List.map real_type types) in
    List.iter (fun (n,t) -> V.enter_type env n t) real_types;
    T.nil
```

Exception Declarations Exception declarations in Tiger are very simple—an exception is just an identifier. When an exception is declared, we store the identifier in the environment.

⟨*exception declarations*⟩≡
```
    | A.ExceptionDec(sym,_) -> V.enter_exn env sym; T.nil
```

3.2.4 Variables

The variable translator checks and translates simple variables, field variables which refer to record elements, and subscript variables which refer to array elements.

⟨*variable translator*⟩≡
```
  and trvar = function
```
⟨*simple vars*⟩
⟨*field vars*⟩
⟨*subscript vars*⟩

Simple Variables A simple variable can be used as an expression in Tiger as long as the variable is defined. To translate a simple variable, we look up its definition in the environment and use it to create a variable access.

⟨*simple vars*⟩≡
```
      A.SimpleVar(sym, pos) ->
        begin match V.lookup_value env sym pos with
          V.VarEntry(acc, vt) ->
            (T.simple_var (V.frame env) acc, base_type env vt)
        | V.FunEntry _ ->
            E.type_err pos "function used as value"
        end
```

Field Variables Since field variables are references to record elements, first we must check that the base variable is of type `RECORD`. Then, the field offset is calculated by searching through the list of declared fields. The `Translate` module (Section **??**) then converts the base variable and offset into a field access.

⟨*field vars*⟩≡

```
| A.FieldVar(var, sym, pos) ->
    let (exp, fields) = match (trvar var) with
      (x, RECORD y) -> (x,y)
    | _ -> E.type_err pos "attempt to dereference non-record type"
    in
    let offset  = ref (-1) in
    let (_,fld) =
      try List.find (fun (s,v) -> incr offset; s = sym) fields
      with Not_found -> E.undefined pos (S.name sym)
    in
    let typ = base_type env fld in
    (T.field_var exp !offset (is_ptr typ), typ)
```

Subscript Variables Subscript variables are computed in the same manner as field variables. However, instead of computing the offset, we use an expression which must be of type `int`.

⟨*subscript vars*⟩≡

```
| A.SubscriptVar(var, exp, pos) ->
    let e,t = (trexp exp) in
    check_type_int pos "subscript variable" t;
    begin match (trvar var) with
      (exp, ARRAY vt) ->
        let typ = (base_type env vt) in
        (T.subscript_var exp e (is_ptr typ) pos, typ)
    | _ ->
        E.type_err pos "attempt to dereference a non-array type"
    end
```

3.2.5 Expressions

Expressions are translated by the `trexp` function.

⟨*expression translator*⟩≡
```
and trexp = function
  ⟨simple expressions⟩
  ⟨records⟩
  ⟨arrays⟩
  ⟨assignment⟩
  ⟨operator expressions⟩
  ⟨function calls⟩
  ⟨conditionals⟩
  ⟨loops⟩
  ⟨sequences⟩
  ⟨let expressions⟩
  ⟨exceptions⟩
  ⟨threads⟩
```

Simple Expressions The simple expressions include variable access, `nil`, and string and integer literals. Variables are handled by the `trvar` function. Literals are translated and assigned the appropriate type.

⟨*simple expressions*⟩≡
```
    A.VarExp v       -> trvar v
  | A.NilExp         -> (T.nil,           NIL)
  | A.IntExp i       -> (T.int_literal i, INT)
  | A.StringExp(s,_) -> (T.str_literal s, STRING)
```

Records When creating a new record instance, the new instance must match the declared type of the record. A record instance matches a record declaration if they both have the same number of fields, with the same names and types, in the same order. The `chk_field` function compares a field label and expression with a declared field label and type. To check a record instance, we call the `chk_field` function on each field expression paired with the corresponding field declaration.

⟨*records*⟩≡

```
| A.RecordExp(var, fields, pos) ->
    let chk_field (s1,e,p) (s2,vt) =
      if (s1 <> s2) then E.type_err p "field names do not match";
      let ex,ty = trexp e in
      check_type_eq p "field type (%s) does not match declaration (%s)"
                      (base_type env vt) ty;
      (ex, is_ptr ty)
    in
    begin match V.lookup_type env var pos with
      RECORD dec_fields ->
        begin try
          let field_vals = (List.map2 chk_field fields dec_fields) in
          (T.new_record field_vals, RECORD dec_fields)
        with Invalid_argument s ->
          E.type_err pos "Record instance does not match declared type"
        end
    | _ ->
        E.type_err pos "Attempt to use non-record type as record"
    end
```

Arrays When creating a new array instance, the initializing expression must match the declared type of the array elements.

⟨*arrays*⟩≡

```
| A.ArrayExp(name, size, init, pos) ->
    begin match V.lookup_type env name pos with
      ARRAY vt ->
        let size,sizety = trexp size
        and init,initty = trexp init
        and typ         = base_type env vt in
        check_type_int pos "array size" sizety;
        check_type_eq  pos "array type(%s) does not type(%s)" typ initty;
        (T.new_array size init (is_ptr typ), ARRAY vt)
    | _ ->
        E.type_err pos "Attempt to use a non-array type as an array"
    end
```

Assignment For assignment expressions, the type of the variable being assigned to on the left-hand side, must match the type of the expression on the right-hand side. In Tiger, assignments are only executed for side-effect; the type of an assignment expression is `UNIT`.

⟨*assignment*⟩≡
```
    | A.AssignExp(var, exp, pos) ->
        let exp,ety = trexp exp
        and var,vty = trvar var in
        check_type_eq pos "Cannot assign to type %s from type %s" vty ety;
        (T.assign var exp, UNIT)
```

Operator Expressions Operators come in two flavors: arithmetic and comparison. The arithmetic operators can only be applied to expressions of type `int`. All of the comparison operators can be applied to integers, `nil`, and strings. In addition, the equality testing operators can also be applied to records and arrays.

The `Translate` module provides three functions for each of arithmetic, integer comparison, and string comparison. To translate an operator expression, we first check that the types of the left-hand and right-hand side expressions are compatible. Then, we select the correct translation function, and apply it to the two expressions.

⟨*operator expressions*⟩≡
```
    | A.OpExp(left, oper, right, pos) ->
        let lexp,lty = trexp left
        and rexp,rty = trexp right in
        check_type_eq pos "Incompatible types %s,%s" lty rty;
        let trans_fn =
          match oper with
            A.PlusOp | A.MinusOp | A.TimesOp | A.DivideOp ->
              check_type_int pos "operator argument" lty;    T.arithmetic
          | A.EqOp | A.NeqOp
          | A.LtOp | A.LeOp | A.GtOp | A.GeOp ->
              begin match lty with
                INT | NIL                                    -> T.compare_int
              | STRING                                       -> T.compare_str
              | ARRAY  _ when oper=A.EqOp or oper=A.NeqOp -> T.compare_int
              | RECORD _ when oper=A.EqOp or oper=A.NeqOp -> T.compare_int
              | _ ->
                  E.type_err pos "Incomparable types"
              end
        in (trans_fn oper lexp rexp, INT)
```

Function Calls When a function is called, the supplied arguments must have the correct types. To translate a function call, first the argument types are checked against the function declaration. If the types match, then the return type of the function is used as the result type for the call expression.

⟨*function calls*⟩≡

```
| A.CallExp(sym, arglist, pos) ->
    let chk_arg = check_type_eq pos
                  "Argument type (%s) does not match declaration (%s)"
    in begin match V.lookup_value env sym pos with
      V.FunEntry(lbl, cc, frm, dec_args, return_type) ->
        let args,tys = List.split (List.map trexp arglist) in
        begin try
          List.iter2 chk_arg tys dec_args;
          let rtyp = base_type env return_type in
          (T.call (V.frame env) lbl cc frm args
                  (V.exn_label env) (is_ptr rtyp), rtyp)
        with Invalid_argument x ->
          E.type_err pos "function arguments do not match declaration"
        end
    | _ ->
        E.type_err pos (S.name sym ^ " is not a function")
    end
```

Conditionals Conditional expressions return a value if both the then clause and the else clause have the same type. Otherwise, the if statement does not return a value, and has type `UNIT`. The `Translate` module provides two translators for conditionals—one for if expressions that return a value and one for if statements that do not.

⟨*conditionals*⟩≡

```
| A.IfExp(if', then', else', pos) ->
    let iex,ity = trexp if'
    and tex,tty = trexp then'
    and eex,ety = match else' with
                    None    -> (T.nil, UNIT)
                  | Some ex -> trexp ex
    in
    check_type_int pos "if condition" ity;
    check_type_eq  pos
      "type of then expression (%s) does not match else (%s)" tty ety;
    let typ = base_type env tty in
    (T.ifexp iex tex eex (is_ptr typ), typ)
```

Loops Tiger defines two looping constructs: a while loop and a for loop. The while loop is composed of a test expression of type `INT`, and a body of type `UNIT`. Before the body is translated, a break label is added to the environment.

⟨*loops*⟩≡
```
| A.WhileExp(test, body, pos) ->
    let body_env = V.new_break_label env in
    let tex,tty = trexp test
    and bex,bty = trans body_env (EXP body) in
    check_type_int pos "while condition" tty;
    check_type_eq  pos "body of while has type %s, must be %s" bty UNIT;
    (T.loop tex bex (V.break_label body_env), UNIT)
```

For loops are translated into equivalent `let` and `while` expressions. First, a looping variable is created and initialized to the low value. Then a test expression is created that compares the looping variable to the high value. Finally, the body of the for loop is modified to include an assignment to the looping variable, and is then used as the body of the new while loop.

⟨*loops*⟩+≡
```
| A.ForExp(sym, lo, hi, body, pos) ->
    let _,loty = trexp lo
    and _,hity = trexp hi in
    check_type_int pos "for lower bound" loty;
    check_type_int pos "for upper bound" hity;
    let v            = A.SimpleVar(sym, pos) in
    let ve           = A.VarExp v in
    let v_less_eq_hi = A.OpExp(ve, A.LeOp, hi, pos)
    and v_plus_1     = A.OpExp(ve, A.PlusOp, (A.IntExp 1), pos) in
    trexp (A.LetExp(
           [(A.VarDec(sym,(Some(S.symbol "int")), lo, pos))],
           (A.WhileExp
              (v_less_eq_hi,
               (A.SeqExp([body;
                          (A.AssignExp(v, v_plus_1, pos))], pos)),
               pos)),
           pos))
```

Break expressions are handled by generating a jump to the nearest break label, which is stored in the current environment.

⟨*loops*⟩+≡
```
| A.BreakExp pos ->
    begin
      try (T.break (V.break_label env), UNIT)
      with Not_found -> raise(E.Error(E.Illegal_break, pos))
    end
```

Sequences A sequence expression evaluates a list of tiger expressions and returns the value of the last expression. The empty sequence is a valid expression, and returns a value of type `UNIT`.

⟨*sequences*⟩≡

```
| A.SeqExp ([],_) -> (T.nil, UNIT)
| A.SeqExp (el,_) ->
    let exprs = List.rev_map trexp el in
    let _,typ = List.hd exprs in
    let exprs = List.rev_map fst exprs in
    (T.sequence exprs, typ)
```

Let Expressions The let expression consists of a sequence of declarations followed by a sequence of expressions. To process the let expression, first a new environment is created. Then each declaration is processed and added to the environment using the `trdec` function. Finally, the let body is processed inside the new environment, and a sequence is created consisting of the declarations followed by the body.

⟨*let expressions*⟩≡

```
| A.LetExp(decls, body, _) ->
    let trns    = trans (V.new_scope env) in
    let decs    = List.map (fun d -> fst (trns (DEC d))) decls
    and bex,bty = trns (EXP body) in
    (T.sequence (decs @ [bex]), bty)
```

Exceptions A try block consists of an expression and one or more exception handlers. For simplicity, a try block is required to have type `unit`. Therefore, both the expression and all of the handlers must have type `unit`. To translate a try block, we create a new environment with a fresh exception label and translate the expression in the new environment. Then, we translate each of the exception handlers in our current environment. Finally, we pass the expression and the handlers to the `Translate` module.

⟨*exceptions*⟩≡
```
| A.TryExp(expr, handlers, pos) ->
    let new_env         = V.new_exn_label env in
    let tryex, tryty    = trans new_env (EXP expr) in
    let handler (s,h,p) =
      let ex,ty = trexp h in
      check_type_unit p "handler" ty;
      (S.uid s, ex)
    in
    check_type_unit pos "try" tryty;
    begin match V.exn_label new_env with
      None     -> E.internal "no exception label for try block"
    | Some lbl ->
        (T.try_block tryex lbl (List.map handler handlers), tryty)
    end
```

When an exception is raised, we lookup the exception id in the environment and call the `Translate` module.

⟨*exceptions*⟩+≡
```
| A.RaiseExp(sym, pos) ->
    let exn_id = V.lookup_exn env sym pos in
    (T.raise_exn exn_id, UNIT)
```

The spawn keyword is given a function name. The function must take zero arguments; the return value is ignored.

⟨*threads*⟩≡
```
| A.SpawnExp(sym, pos) ->
    begin match V.lookup_value env sym pos with
      V.FunEntry(lbl, cc, frm, dec_args, return_type) ->
        if dec_args <> []
        then E.type_err pos "spawn function must take zero arguments."
        else (T.spawn lbl,INT)
    | _ ->
        E.type_err pos (S.name sym ^ " is not a function")
    end
```

Chapter 4

Translation

4.1 Intermediate Representation

The intermediate representation used by the Tiger compiler is a variant of the "Tree Language" described in [**?**]. The abstract syntax for the intermediate representation "trees" is show below.

Both code labels and temporary variables are represented by symbols.

⟨*types*⟩≡
```
type label = Symbol.symbol
and  temp  = Symbol.symbol
```

Statements are represented by the `stm` type. Sequences of statements can be created using the `SEQ` constructor. In addition, any expression can be used as a statement by way of the `EXP` constructor. The `TRY` and `TRYEND` constructors mark the start and end of try blocks. Both of the try constructors take the label of the continuation for the exception handlers.

⟨*types*⟩+≡
```
type stm =
    SEQ    of stm * stm
  | LABEL  of label
  | CONT   of label * label list
  | JUMP   of exp
  | CJUMP  of exp * label * label
  | MOVE   of exp * exp
  | EXP    of exp
  | TRY    of label
  | TRYEND of label
  | RET    of exp
```

The abstract syntax of expressions is represented by the `exp` type. Many expressions carry a boolean flag that is true if the expression results in a pointer value. The `CALL` constructor also takes an optional string that indicates the calling convention of the function being called, and an optional label indicating a continuation that the function may cut to or unwind to.

⟨*types*⟩+≡
```
and exp =
    BINOP of binop * exp * exp
  | RELOP of relop * exp * exp
  | MEM   of exp * bool
  | TEMP  of temp * bool
  | ESEQ  of stm * exp
  | NAME  of label
  | CONST of int
  | CALL  of exp * exp list * string option * label option * bool
```

Finally, a handful of binary operators are supported. The binary operators are divided into relational and arithmetic types.

⟨*types*⟩+≡
```
and binop = PLUS | MINUS | MUL | DIV
and relop = EQ | NE | LT | GT | LE | GE
```

4.1.1 Interface to the IR

In addition to defining the abstract syntax of the IR, the `Tree` module also defines a number of utility functions for manipulating the IR.

⟨*tree.mli*⟩≡
```
⟨types⟩
⟨utility functions⟩
```

New code labels and temporary variables can be created using the `new_label` and `new_temp` functions. The `new_label` function generates a new unique code label, and the `new_temp` function generates a new temporary variable.

⟨*utility functions*⟩≡
```
val new_label : string -> label
val new_temp  : unit   -> temp
```

There are three functions related to binary operators. The `relop_inverse` function reverses the meaning of a relational operator. The other two functions return a string representing the `C--` operator associated with an given tree language operator.

⟨*utility functions*⟩+≡
```
val relop_inverse  : relop -> relop
val cmm_binop      : binop -> string
val cmm_relop      : relop -> string
```

In Tiger, each expression results in a value that is either a machine word, or a pointer to a data structure. The `is_ptr` function returns true if a given expression will return a pointer. A related function, `find_temps` returns a list of all the temporary variables in a statement. With each temporary in the list is a boolean flag that is true only if the temporary variable holds a pointer.

⟨*utility functions*⟩+≡
```
val is_ptr     : exp -> bool
val find_temps : stm list -> (temp * bool) list
```

Finally, for debugging purposes, there are two functions for printing out string representations of expressions and statements.

⟨*utility functions*⟩+≡
```
val print_stm : stm -> unit
val print_exp : exp -> unit
```

4.1.2 Implementation of IR Utilities

⟨*tree.ml*⟩≡
```
module S = Symbol
```
⟨*types*⟩

New labels and temporaries are generated using the `new_symbol` function from the `Symbol` module.

⟨*tree.ml*⟩+≡
```
let new_label s = S.new_symbol ("L" ^ s)
let new_temp () = S.new_symbol "temp"
```

The functions for binary operators are simple mappings.

⟨*tree.ml*⟩+≡
```
let relop_inverse = function
    EQ  -> NE
  | NE  -> EQ
  | LT  -> GE
  | GT  -> LE
  | LE  -> GT
  | GE  -> LT
let cmm_binop = function
    PLUS  -> "add"
  | MINUS -> "sub"
  | MUL   -> "mul"
  | DIV   -> "quot"
let cmm_relop = function
    EQ -> "eq"
  | NE -> "ne"
  | LT -> "lt"
  | GT -> "gt"
  | LE -> "le"
  | GE -> "ge"
```

The `is_ptr` function is implemented by returning the appropriate field from each of the expression types. Expression types that do not have a pointer field are known to *not* be pointers.

⟨*tree.ml*⟩+≡
```
let rec is_ptr = function
    BINOP _         -> false
  | RELOP _         -> false
  | MEM(_,p)        -> p
  | TEMP(_,p)       -> p
  | ESEQ(_,e)       -> is_ptr e
  | NAME _          -> false
  | CONST _         -> false
  | CALL(_,_,_,_,p) -> p
```

In order to implement the `find_temps` function, we build a set of temporary variables and then convert the set into a list. We use the standard library set implementation to build a unique set of temporaries.

⟨*tree.ml*⟩+≡
```
module TempSet = Set.Make(
  struct
    type t = Symbol.symbol * bool
    let compare = Pervasives.compare
  end)
```

Using the `TempSet` module, the implementation of `find_temps` is relatively easy.

⟨*tree.ml*⟩+≡

```
let find_temps stmts =
  let foldl = List.fold_left in
  let rec stm set = function
      SEQ(a,b)     -> stm (stm set a) b
    | LABEL _      -> set
    | CONT _       -> set
    | JUMP e       -> exp set e
    | CJUMP(e,_,_) -> exp set e
    | MOVE(a,b)    -> exp (exp set a) b
    | EXP e        -> exp set e
    | TRY _        -> set
    | TRYEND _     -> set
    | RET e        -> exp set e
  and exp set = function
      BINOP(_,a,b)     -> exp (exp set a) b
    | RELOP(_,a,b)     -> exp (exp set a) b
    | MEM(e,_)         -> exp set e
    | TEMP(t,ptr)      -> TempSet.add (t,ptr) set
    | ESEQ(s,e)        -> exp (stm set s) e
    | NAME _           -> set
    | CONST _          -> set
    | CALL(e,el,_,_,_) -> foldl exp set (e :: el)
  in
  TempSet.elements (foldl stm TempSet.empty stmts)
```

Now, we give the implementation of the two printing functions for statements and expressions. We will define the function for printing statements first.

⟨*tree.ml*⟩+≡

```
let print_stm =
  let rec iprintf = function
      0 -> Printf.printf
    | i -> (print_string "  "; iprintf (i-1))
  in
  let rec prstm d = function
    | LABEL l      -> iprintf d "LABEL:%s\n " (S.name l)
    | CONT(l,ls)   -> iprintf d "CONT:%s\n "  (S.name l)
    | TRY l        -> iprintf d "TRY:%s\n"    (S.name l)
    | TRYEND l     -> iprintf d "TRYEND:%s\n" (S.name l)
    | SEQ(a,b)     -> iprintf d "SEQ:\n";     prstm(d+1) a; prstm(d+1) b
    | MOVE(a,b)    -> iprintf d "MOVE:\n";    prexp(d+1) a; prexp(d+1) b
    | JUMP e       -> iprintf d "JUMP:\n";    prexp(d+1) e
    | EXP e        -> iprintf d "EXP:\n";     prexp(d+1) e
    | RET e        -> iprintf d "RET:\n";     prexp(d+1) e
    | CJUMP(a,t,f) -> iprintf d "CJUMP:\n";   prexp(d+1) a;
                      iprintf (d+1) "true  label: %s\n" (S.name t);
                      iprintf (d+1) "false label: %s\n" (S.name f)
  and prexp d = function
      BINOP(p,a,b)     -> iprintf d "BINOP:%s\n" (cmm_binop p);
                          prexp (d+1) a; prexp (d+1) b
    | RELOP(p,a,b)     -> iprintf d "RELOP:%s\n" (cmm_relop p);
                          prexp (d+1) a; prexp (d+1) b
    | MEM(e,_)         -> iprintf d "MEM:\n"; prexp (d+1) e
    | TEMP(t,_)        -> iprintf d "TEMP %s\n" (S.name t)
    | ESEQ(s,e)        -> iprintf d "ESEQ:\n";
                          prstm (d+1) s; prexp (d+1) e
    | NAME lab         -> iprintf d "NAME %s\n" (S.name lab)
    | CONST i          -> iprintf d "CONST %d\n" i
    | CALL(e,el,_,_,_) -> iprintf d "CALL:\n";
                          prexp (d+1) e; List.iter (prexp (d+2)) el
  in prstm 0
```

The expression printer just calls the statement printer after converting the expression into a statement.

⟨*tree.ml*⟩+≡

```
let print_exp e = print_stm (EXP e)
```

4.2 Translation to Intermediate Representation

The `Translate` module transforms an abstract syntax tree (Section **??**) into the compiler's intermediate representation—a "tree language" expression (Section **??**). The `Translate` module is called by the `Semantics` module (Section **??**) to transform expressions and statements after they have been type-checked. A translation function is defined for each of the different types of abstract syntax fragments that need to be translated.

⟨*translate.mli*⟩≡

```
type exp   = Tree.exp
type label = Tree.label

val nil           : exp
val int_literal   : int -> exp
val str_literal   : string -> exp
val simple_var    : Frame.frame -> Frame.access -> exp
val field_var     : exp -> int -> bool -> exp
val subscript_var : exp -> exp -> bool -> int -> exp
val assign        : exp -> exp -> exp
val call          : Frame.frame -> label -> string option ->
                    Frame.frame -> exp list -> label option -> bool -> exp
val arithmetic    : Ast.oper -> exp -> exp -> exp
val compare_int   : Ast.oper -> exp -> exp -> exp
val compare_str   : Ast.oper -> exp -> exp -> exp
val ifexp         : exp -> exp -> exp -> bool -> exp
val loop          : exp -> exp -> label -> exp
val break         : label -> exp
val new_record    : (exp * bool) list -> exp
val new_array     : exp -> exp -> bool -> exp
val sequence      : exp list -> exp
val func          : Frame.frame -> exp -> bool -> exp

val try_block     : exp -> label -> (int *exp) list -> exp
val raise_exn     : int -> exp
val spawn         : label -> exp
```

4.2.1 Translation Implementation

⟨*translate.ml*⟩≡
```
module E = Error
module A = Ast
module S = Symbol
module T = Tree
module F = Frame
type exp   = T.exp
type label = T.label
let  ws    = Sys.word_size / 8
```
⟨*utilities*⟩
⟨*literals*⟩
⟨*function calls*⟩
⟨*variables*⟩
⟨*operator expressions*⟩
⟨*conditionals*⟩
⟨*loops*⟩
⟨*records and arrays*⟩
⟨*sequences*⟩
⟨*functions*⟩
⟨*exceptions*⟩
⟨*threads*⟩

Utilities A few small utility functions are defined to make the `Translate` module more readable. The `seq` function converts a list of statements into a tree of `SEQ` nodes.

⟨*utilities*⟩≡
```
let rec seq = function
    []        -> E.internal "nil passed to seq"
  | x :: []   -> x
  | x :: rest -> T.SEQ(x, seq rest)
```

The `eseq` function produces an expression from and expression and a list of statements. The list of statement is executed first, then the expression is evaluated.

⟨*utilities*⟩+≡
```
let eseq exp stmts = T.ESEQ (seq stmts, exp)
```

During translation we often need to allocate new temporary variables. The `temp` function produces a fresh temporary variable.

⟨*utilities*⟩+≡
```
let temp ptr = T.TEMP (T.new_temp(), ptr)
```

The Tiger compiler uses static links to manage nested functions. Each generated function has a frame pointer that points to the top of the function's stack frame. The first word in each stack frame is the frame pointer of the calling function. Thus, we can fetch our caller's frame pointer by dereferencing the first location in our own frame pointer. The `getfp` function generates an expression that fetches the frame pointer of one frame relative to another.

⟨*utilities*⟩+≡
```
let getfp frm parent_frm =
  let diff = F.level frm  - F.level parent_frm in
  let rec deref = function
      0 -> F.fp frm
    | x -> T.MEM (deref (x-1), true)
  in assert (diff >= 0); deref diff
```

Finally, we define a few functions for constructing tree expressions and statements. For the binary operators, we simplify expressions when possible.

⟨*utilities*⟩+≡
```
let alloc_ptr = T.NAME (S.symbol "alloc_ptr")
let space_end = T.MEM  (T.NAME (S.symbol "space_end"), true)
let goto lbl  = T.JUMP (T.NAME lbl)
let ( =>) e v = T.MOVE (e, v)

let simplify tig_op op x y =
  match (x,y) with
    (T.CONST x, T.CONST y) -> T.CONST (op x y)
  | _                      -> T.BINOP (tig_op, x, y)
let (<+>) = simplify T.PLUS  ( + )
let (<->) = simplify T.MINUS ( - )
let (<*>) = simplify T.MUL   ( * )
```

Literals In Tiger, there are three types of literal value: nil, integers, and strings. For strings literals, we must output the string as read-only data and refer to it by a label. Most of the implementation of string literals is handled by the `Frame` module (Section **??**). This module gets a label from the `Frame` module and generates a reference to the label using a `NAME` expression.

⟨*literals*⟩≡
```
let nil           = T.CONST 0
let int_literal i = T.CONST i
let str_literal s = T.NAME (F.alloc_string s)
```

Function Calls Each Tiger function will eventually be translated into a C-- function. In addition, there are functions defined in the standard library and in the runtime system. We call these two types of functions "internal" and "external", respectively. All internal functions take as a first parameter the frame pointer of the enclosing function. To generate a call to an internal function, we must compute the correct frame pointer to pass as the first argument. If we are calling a nested function, then we pass the current frame pointer. Otherwise, we compute the frame pointer of the enclosing function using `getfp`.

⟨*function calls*⟩≡
```
let call myfrm lbl cc frm args k ptr =
  let args =
    if F.level frm == 0 then args
    else let pfp =
      if (F.level frm) > (F.level myfrm) then F.fp frm
      else T.MEM(getfp myfrm frm, true)
    in pfp  :: args
  in
```

If we are calling a foreign function (a function not written in C--), then we need to save and restore the global register variables. In Tiger, we only have one global register variable—the allocation pointer.

⟨*function calls*⟩+≡
```
  match cc with
    None        -> T.CALL((T.NAME lbl), args, cc, k, ptr)
  | Some "gc"   -> T.CALL((T.NAME lbl), args, cc, k, ptr)
  | Some _ ->
      let tmp1      = temp ptr
      and tmp2      = temp ptr
      in eseq tmp2 [ T.MOVE(alloc_ptr, tmp1);
                     T.MOVE(T.CALL((T.NAME lbl), args, cc, k, ptr), tmp2);
                     T.MOVE(tmp1, alloc_ptr) ]
```

For internal use, we define a few abbreviations for calling both C and C-- functions from the standard library.

⟨*function calls*⟩+≡
```
let ext_call cc name args =
  call F.base_frame (S.symbol name) cc
       F.base_frame args None false

let ext_c_call   = ext_call (Some "C")
let ext_gc_call  = ext_call  None (* (Some "gc") *)
let ext_cmm_call = ext_call  None
```

Variables There are three types of variable access expressions: simple variables, record fields, and array indices. Simple variables can live on the stack, or in C-- local variables. We generate a memory access for stack variables, and for local variables we just output their name.

⟨*variables*⟩≡
```
let simple_var frm = function
    F.Temp lbl                    -> T.NAME lbl
  | F.Stack(var_frm, offset, ptr) ->
      T.MEM(getfp frm var_frm <+> T.CONST(offset * ws), ptr)
```

Record fields always live in a memory location. For record field variables, we generate a memory access at an offset from the beginning of the record.

⟨*variables*⟩+≡
```
let field_var ex i ptr = T.MEM(ex <+> T.CONST(i * ws), ptr)
```

Array subscripts also always live in a memory location. However, before accessing an array element, we first do an array bounds check. There is a small optimization when the array index expression is a constant, in that we can compute the offset into memory at compile time.

⟨*variables*⟩+≡
```
let subscript_var e1 e2 ptr pos =
  let check = ext_c_call "bounds_check"
                         [e1;e2;T.CONST(fst (Error.line_number pos))]
  and offset = (e2 <+> T.CONST 1) <*> T.CONST ws
  in
  eseq (T.MEM(e1 <+> offset, ptr)) [T.EXP check]
```

Finally, any variable expression can be updated with a MOVE node.

⟨*variables*⟩+≡
```
let assign v e = eseq nil [e => v]
```

Operator Expressions There are three types of operator expressions: arithmetic, integer comparison, and string comparison. Arithmetic operations are translated into `BINOP` nodes, and integer comparisons are translated into `RELOP` nodes.

⟨*operator expressions*⟩≡
```
let arithmetic op ex1 ex2 =
  let oper = match op with
    A.PlusOp   -> T.PLUS
  | A.MinusOp  -> T.MINUS
  | A.TimesOp  -> T.MUL
  | A.DivideOp -> T.DIV
  | _          -> E.internal "relop used as binop"
  in T.BINOP(oper, ex1, ex2)

let compare_int op ex1 ex2 =
  let oper = match op with
    A.EqOp  -> T.EQ
  | A.NeqOp -> T.NE
  | A.LtOp  -> T.LT
  | A.LeOp  -> T.LE
  | A.GtOp  -> T.GT
  | A.GeOp  -> T.GE
  | _       -> E.internal "binop used as relop"
  in T.RELOP(oper, ex1, ex2)
```

String comparisons are translated into integer comparisons by first calling the standard library function `compare_str`, which returns -1, 0, or 1. The result of this function is then compared to zero using an integer comparison.

⟨*operator expressions*⟩+≡
```
let compare_str op ex1 ex2 =
  let result = ext_c_call "compare_str" [ex1;ex2] in
  compare_int op result (T.CONST 0)
```

Conditionals To translate an if expression, we create a new temporary variable which becomes the result. The true and false expressions are converted into move statements that store values into the temporary.

⟨*conditionals*⟩≡
```
let ifexp test thn els ptr =
  let tmp  = temp ptr
  and tru  = T.new_label "ifTrue"
  and fls  = T.new_label "ifFalse"
  and end' = T.new_label "ifEnd" in
  eseq tmp [ T.CJUMP(test, tru, fls);
             T.LABEL tru; thn => tmp; goto end';
             T.LABEL fls; els => tmp;
             T.LABEL end']
```

Loops For each loop statement, the `Semantics` module will call the `loop` function with translated test and body expressions, and a break label. The break label is produced by the `Semantics` module and stored in the environment.

⟨*loops*⟩≡
```
let loop test body lend =
  let lbeg = T.new_label "loop_start"
  and lbdy = T.new_label "loop_body" in
  eseq nil [ T.LABEL lbeg;
             T.CJUMP(test, lbdy, lend);
             T.LABEL lbdy; T.EXP body; goto lbeg;
             T.LABEL lend ]
```

A `break` statement is translated as a jump to the current break label. The current break label is fetched from the environment and provided by the `Semantics` module.

⟨*loops*⟩+≡
```
let break lbl = eseq nil [goto lbl]
```

Records and Arrays When new records and arrays are created, memory must be allocated on the heap for their storage. Memory is allocated by calling the `alloc` function defined below. This function generates code that checks for heap exhaustion, calls the garbage collector, and updates the allocation pointer.

⟨*records and arrays*⟩≡
```
let alloc size =
  let size = (size <+> T.CONST 1) <*> T.CONST ws in
  let test = T.RELOP(T.GT, alloc_ptr <+> size, space_end)
  and tmp  = temp true
  and tru  = T.new_label "alc_gc"
  and fls  = T.new_label "alc"
  in eseq tmp [ T.CJUMP(test, tru, fls);
                T.LABEL tru; T.EXP (ext_gc_call "call_gc" []);
                T.LABEL fls;
                size => T.MEM(alloc_ptr, true);
                (alloc_ptr <+> T.CONST ws) => tmp;
                (alloc_ptr <+> size) => alloc_ptr
                (* ; T.EXP (ext_gc_call "call_gc" []) *)
              ]
```

To initialize a new record, we must copy the results of the initializing expressions into the the record fields. The private function `initialize` creates a list of assignments to the record fields. The result of calling `initialize` is placed in sequence after an allocation statement, and a new temporary variable is returned that contains a pointer to the new record.

⟨*records and arrays*⟩+≡
```
let new_record init =
  let tmp  = temp true
  and size = T.CONST (List.length init) in
  let rec initialize offset = function
      []             -> []
    | (ex,ptr)::rest -> (ex => field_var tmp offset ptr)
                        :: initialize (offset+1) rest
  in
  eseq tmp ((alloc size => tmp) :: initialize 0 init)
```

To initialize a new array, we must copy the result of the initializing expression into each array element. First, we allocate memory for the array and store the array size into the first element of the array. Then, we construct a small loop that initializes each element of the array.

⟨*records and arrays*⟩+≡
```
let new_array sizeEx initEx ptr =
  let ary  = temp true
  and i    = temp false
  and lbeg = T.new_label "init_start"
  and lend = T.new_label "init_end" in
  eseq ary
    [ alloc (sizeEx <+> T.CONST 1) => ary;
      sizeEx => T.MEM(ary, false);
      T.CONST 1 => i;
      T.LABEL lbeg;
      initEx => T.MEM (ary <+> (i <*> T.CONST ws), ptr);
      i <+> T.CONST 1 => i;
      T.CJUMP(T.RELOP(T.LE, i, sizeEx <+> T.CONST 1), lbeg, lend);
      T.LABEL lend ]
```

Sequences To translate a sequence of n expressions, we convert the first $n-1$ expressions into statements and execute them. Then, we evaluate the last expression and the result is used as the result of the whole sequence.

⟨*sequences*⟩≡
```
let rec sequence = function
   []       -> nil
  | e :: [] -> e
  | e :: es -> T.ESEQ((T.EXP e), (sequence es))
```

Functions Once a code fragment has been produced, the `Semantics` module will call `func` to transform it into a function. To produce a function, we allocate a temporary to store the result of evaluating the body. Then, we add a return node that returns the temporary.

⟨*functions*⟩≡
```
let func frm ex ptr =
  let tmp = temp ptr in
  eseq nil [ex => tmp; T.RET tmp]
```

Exceptions When a try block is encountered, first the expression within the block and all of the handlers will be translated. Then, the expression inside the try block and the expressions corresponding to each handler will be passed to the `try_block` function. When translating a try block, we place all of the handlers under a new continuation that takes a single parameter—the exception identifier. The private `cont` function constructs a continuation with the label `l` that accepts one parameter in the temporary `t`.

⟨*exceptions*⟩≡
```
let try_block exp exn_lbl hs =
  let cont l = function
      T.TEMP(t,_) -> T.CONT(l, [t])
    | _           -> E.internal "non temp in continuation node"
  in
```

Each handler must test the continuation parameter, and then either handle the exception or pass control down to the next handler. The `handler` function constructs this sequence of statements for a single handler.

⟨*exceptions*⟩+≡
```
  let try_endl         = T.new_label "try_end"
  and tmp              = temp false in
  let handler (uid,ex) =
    let hl = T.new_label "handle"
    and sl = T.new_label "skip" in
    [ T.CJUMP(T.RELOP(T.EQ, tmp, T.CONST uid), hl, sl);
      T.LABEL hl; T.EXP ex; goto try_endl;
      T.LABEL sl ]
  in
```

The complete try block consists of the expression within the try block followed by a new continuation and the handlers for the block. The scope of the try block is marked by `TRY` and `TRYEND`. At the beginning and end of the try block we must set and reset the current exception handler. However, if we are generating code for the unwinding implementation, then we can omit these calls to `set_handler`.

⟨*exceptions*⟩+≡
```
    let old            = temp false in
    let set_handler    = ext_cmm_call "set_handler" [T.NAME exn_lbl] => old
    and reset_handler  = T.EXP (ext_cmm_call "set_handler" [old])
    and not_unwind stm = if Option.use_unwind() then T.EXP nil else stm
    in
    eseq tmp [ T.TRY exn_lbl;
               not_unwind set_handler;
               exp => tmp;
               not_unwind reset_handler;
               T.TRYEND exn_lbl;
               goto try_endl;
               cont exn_lbl tmp;
               not_unwind reset_handler;
               seq (List.flatten (List.map handler hs));
               T.LABEL try_endl ]
```

When raising an exception, we will either call the `raise` function or the `unwind` function depending on how the compiler was called.

⟨*exceptions*⟩+≡
```
  let raise_exn uid =
    let fn = if Option.use_unwind() then "unwind" else "raise" in
    ext_cmm_call fn [T.CONST uid]
```

Threads All of the thread magic is in the runtime system. Here, we just make a call to the runtime system function "tig$_s pawn''passingitthelabelofafunction.$
⟨*threads*⟩≡
```
  let spawn lbl = ext_cmm_call "spawn" [T.NAME lbl]
```

Chapter 5

Code Generation

```
% -*- mode: Noweb; noweb-code-mode: caml-mode -*-% ===> this file was generated automatically b
% $Id: frame.nw,v 1.1.1.1 2007-01-29 16:31:59 govereau Exp $
% ------------------------------------------------------------------------
```

5.1 Stack Frames

```
% ------------------------------------------------------------------------
The Frame module manages the stack frames of generated functions.
Since we are generating C-- code, we do not have access to the
real stack frame, but we still need to track function parameters,
local variables, and temporaries. This module provides the compiler
with a mechanism for allocating new variables and temporaries within a
function, and is responsible for generating the corresponding C--
code. In addition, it is also convenient to track string literals
using this module.

A stack frame is represented by an abstract type frame. Each
parameter, local, or temporary allocated results in a variable access
specification. The variable access specification is represented by the
type access.
```

⟨*frame.mli*⟩≡

```
type frame
type access =
    Temp  of Tree.label
  | Stack of frame * int * bool
```

This module provides functions for reading elements of a stack frame. There are functions for reading the frame pointer, the name of the function associated with the frame, and the nesting level.

⟨*frame.mli*⟩+≡
```
val fp    : frame -> Tree.exp
val name  : frame -> Tree.label
val level : frame -> int
```

There are two functions for creating new stack frames. The base_frame function will create a stack frame for a top-level function, and new_frame will create a stack frame for a nested function. In Tiger, all functions are nested inside of the tiger_main function, so the compiler only uses new_frame for generated functions; base_frame is used for external library functions.

⟨*frame.mli*⟩+≡
```
val base_frame : frame
val new_frame  : Tree.label -> frame -> frame
```

There are four functions for allocating new parameters, local variables, temporary variables, and string literals.

⟨*frame.mli*⟩+≡
```
val alloc_param  : frame  -> Tree.label -> bool -> access
val alloc_local  : frame  -> Tree.label -> bool -> access
val alloc_temp   : frame  -> Tree.label -> bool -> access
val alloc_string : string -> Tree.label
```

During code generation, this module will output a C-- header and footer for each function that initializes the stack frame. In addition, this module will also output the string literals as initialized data.

⟨*frame.mli*⟩+≡
```
val output_header  : frame -> unit
val output_footer  : frame -> unit
val output_strings : unit  -> unit
```

```
% ------------------------------------------------------------------------
```

5.1.1 Stack Frame Implementation

```
% ------------------------------------------------------------------------
```

⟨*frame.ml*⟩≡

```
module S = Symbol
module T = Tree
module H = Hashtbl
```

A stack frame is a record containing a name, and nesting level, the size of the stack, and a list of variables. There are three types of variables: function parameters, stack allocated variables, and temporaries. For each variable we store the name and a boolean that is true if the variable holds a pointer.

⟨*frame.ml*⟩+≡

```
type frame = { name           : T.label
             ; level          : int
             ; mutable size   : int
             ; mutable params : (T.label * bool) list
             ; mutable vars   : (T.label * bool) list
             ; mutable temps  : (T.label * bool) list
             }
type access =
    Temp  of T.label
  | Stack of frame * int * bool
```

The accessor functions for frames just read the appropriate elements from the record. In the case of the frame pointer, we return the name of the variable that holds the frame pointer. The name of the frame pointer variable is always fp.

⟨*frame.ml*⟩+≡
```
let fp    frm = T.NAME (S.symbol "fp")
let name  frm = frm.name
let level frm = frm.level
```

The base frame is a record with defaults for each field. Every function takes at least one parameter---its parent's frame pointer pfp. To create a new nested frame, we need to get the nesting level of the parent frame. Otherwise, a nested frame starts out with the same defaults as the base frame.

⟨*frame.ml*⟩+≡
```
let base_frame = { name   = (S.symbol "frame0")
                 ; level  = 0
                 ; params = [(S.symbol "pfp", true)]
                 ; size   = 1
                 ; vars   = []
                 ; temps  = []
                 }
let new_frame lbl parent = { base_frame with
                             name  = lbl;
                             level = parent.level + 1 }
```

Allocated parameters and local variables both end up on the stack. In theory, a local variable could be allocated as a temporary variable as long as it is not referenced by a nested function. However, this optimization would require an escape analysis that this compiler does not currently do.

To allocate parameters and locals, we update the appropriate list, increment the stack size and return a stack access specification. Function parameters must be kept in order they are allocated.

⟨*frame.ml*⟩+≡
```
let stack_alloc frm ptr =
  let v = Stack(frm, frm.size, ptr) in
  frm.size <- frm.size + 1; v

let alloc_param frm name ptr =
  frm.params <- frm.params @ [(name,ptr)];
  stack_alloc frm ptr

let alloc_local frm name ptr =
  frm.vars <- (name,ptr) :: frm.vars;
  stack_alloc frm ptr
```

Allocating temporaries is similar, but we return an access specification for a temporary variable. A temporary variable is just a C-- local variable, and we only need to remember the name of the variable.

⟨*frame.ml*⟩+≡
```
let alloc_temp frm name ptr =
  frm.temps <- (name,ptr) :: frm.temps;
  Temp name
```

When a new string literal is allocated, we first check to see if the same literal has already been allocated, and if so, we return a reference to the previously allocated string. Otherwise, we add the new string to the set, and return a new label. The implementation uses the standard library Hashtbl module to store the set of allocated strings.

⟨*frame.ml*⟩+≡
```
let strings = H.create 20
let alloc_string s =
  try H.find strings s
  with Not_found ->
    let lbl = T.new_label "gbl" in
    (H.add strings s lbl; lbl)
```

The output functions are a bit tedious to write and even more tedious to read. However, we can at least attempt to control the ensuing insanity with a few nice abbreviations and utility functions.

⟨*frame.ml*⟩+≡
```
let pf           = Printf.printf
let spf          = Printf.sprintf
let join_map f l = String.concat "," (List.map f l)
let iter_ndx f   = let n   = ref(-1) in
                   let g x = incr n; f !n x in
                   List.iter g
```

For each function header, we output the function name and parameters. The function body is enclosed in a span that references the pointer data for the garbage collector. The body of each function begins by initializing the stack data locations that correspond to the formal parameters, and declaring of all temporaries.

⟨*frame.ml*⟩+≡

```
let output_header frm =
  let param  (p,_) = spf "bits32 %s" (S.name p)
  and init n (p,_) = pf "  bits32[fp+%d] = %s;\n" (4*n) (S.name p)
  and temp   (t,_) = pf "  bits32 %s;\n" (S.name t)
  and name         = (S.name frm.name) in
  pf "%s(%s) {\n" name (join_map param frm.params);
  pf " span 1 %s_gc_data {\n" name;
  pf "  stackdata { align 4; fp : bits32[%d]; }\n" frm.size;
  iter_ndx  init frm.params;
  List.iter temp frm.temps
```

Each function is followed by the pointer information for the stack data and the C-- variables. The stack data information is output first as an array of bits32. A second array follows with the pointer information for the C-- variables.

⟨*frame.ml*⟩+≡

```
let output_footer frm =
  let var_data vl =
    let int_of_var (_,p) = if p then 1 else 0 in
    let data = List.length vl :: List.map int_of_var vl in
    join_map string_of_int data
  in
  pf "}}\n";
  pf "section \"data\" {\n";
  pf " %s_gc_data:\n" (S.name frm.name);
  pf "   bits32[] { %s };\n" (var_data (frm.params @ List.rev frm.vars));
  pf "   bits32[] { %s };\n" (var_data (frm.params @ frm.temps));
  pf "}\n\n"
```

```
For the string literals, we output a data section containing a label
and initialized data for each string.
```

⟨*frame.ml*⟩+≡

```
(* output *)
let output_strings() =
  let print_string str lbl =
    let len = String.length str
    and str = String.escaped str in
    pf " %s: bits32 { %d }; bits8[] \"%s\\000\";\n"
       (S.name lbl) len str
  in
  pf "section \"data\" { align 4;\n";
  H.iter print_string strings;
  pf "}\n\n"
```

```
% -*- mode: Noweb; noweb-code-mode: caml-mode -*-% ===> this file was generated autor
% $Id: codegen.nw,v 1.1.1.1 2007-01-29 16:31:59 govereau Exp $
% ----------------------------------------------------------------------------
```

5.2 Code Generation

```
% ----------------------------------------------------------------------------
This module has two interface functions: one for generating the
C-- file header, and one for generating the bodies of C--
functions. The compiler diver will call the first function to generate
the file header and pass in a list of imports. The imports correspond
to the external functions that may be called by Tiger programs.
```

⟨*codegen.mli*⟩≡

```
val output_file_header : string list   -> unit
val emit               : Tree.stm list -> unit
```

⟨*codegen.ml*⟩≡
```
module E = Error
module S = Symbol
module T = Tree
let pf           = Printf.printf
let spf          = Printf.sprintf
let join_map f l = String.concat "," (List.map f l)
```

In the file header we declare our C-- program to be little endian, import all of the external functions, and export the tiger_main function. In addition, we must declare the alloc_ptr global to be consistent with the other C-- sources in the standard library and garbage collector. We also import the space_end pointer from the garbage collector to support inlined allocations.

⟨*codegen.ml*⟩+≡
```
let output_file_header imports =
  let pr_import x = pf "import bits32 \"tig_%s\" as %s;\n" x x in
  pf "target byteorder little;\n";
  List.iter pr_import imports;
  pf "export tiger_main;\n\n";
  pf "bits32 alloc_ptr;\n";
  pf "import space_end;\n\n"
```

The heart of the code generator is the emit function that converts a list of Tree statements (Section~??) into C--.
The emit function uses three private functions for converting statements, and expressions either in a value context or a boolean context.

⟨*codegen.ml*⟩+≡
```
let emit exl =
  ⟨statements⟩
  ⟨value expressions⟩
  ⟨boolean expresssions⟩
  in
  let code = List.map stm exl in
  List.iter (fun x -> pf "  %s\n" x) code
```

Generating C-- code from our intermediate representation is relatively easy. However, when generating code for Tree statements, there is one small subtlety; we always wrap try blocks inside of a span with a key value of 2, and a data pointer of 1. This ‘‘pointer’’ marks the activation as having an exception handler that can be unwound to. The span data for exceptions is not examined, but we need something to put into the span, and it cannot be zero, because then we could not distinguish it from an error result from GetDescriptor in the runtime system.

⟨*statements*⟩≡

```
let rec stm = function
    T.LABEL l            -> spf "%s:" (S.name l)
  | T.CONT(l,ls)         -> spf "continuation %s(%s):"
                            (S.name l) (join_map S.name ls)
  | T.JUMP e             -> spf "goto %s;" (valexp e)
  | T.CJUMP(ex, l1, l2)  -> spf "if(%s) {goto %s;} else {goto %s;}"
                            (boolexp ex) (S.name l1) (S.name l2)
  | T.MOVE(e1, e2)       -> spf "%s = %s;" (valexp e2) (valexp e1)
  | T.EXP(T.CALL _ as e) -> spf "%s;" (valexp e)
  | T.EXP e              -> spf "/* eliminated: %s */" (valexp e)
  | T.TRY l              -> spf "span 2 1 { /* %s */" (S.name l)
  | T.TRYEND l           -> spf "} /* end %s */" (S.name l)
  | T.RET e              -> spf "return(%s);" (valexp e)
  | T.SEQ _              -> E.internal "SEQ node found in code gen"
```

```
  Generating C-- code for Tree expressions is also quite simple.
  The CALL nodes require a little bit of work to get all of the
  annotations correct, but there are not any surprises.
```

⟨*value expressions*⟩≡

```
    and valexp = function
        T.BINOP(bop, e1, e2)  -> spf "%%%s(%s, %s)"
                                   (T.cmm_binop bop) (valexp e1) (valexp e2)
      | T.RELOP _ as e        -> spf "%%sx32(%%bit(%s))" (boolexp e)
      | T.MEM(e,ptr)          -> spf "bits32[%s]" (valexp e)
      | T.TEMP(t,ptr)         -> spf "%s" (S.name t)
      | T.NAME l              -> (S.name l)
      | T.CONST i             -> string_of_int i
      | T.CALL(l,el,cc,k,ptr) ->
          let cc = match cc with
                     None   -> ""
                   | Some s -> spf "foreign \"%s\" " s
          and k  = match k with
                     None   -> ""
                   | Some l -> spf "also %s to %s"
                      (if Option.use_unwind() then "unwinds" else "cuts")
                      (S.name l)
          in
          spf "%s %s(%s) also aborts %s"
              cc (valexp l) (join_map valexp el) k
      | T.ESEQ _ ->
          E.internal "ESEQ node found in code gen"
```

⟨*boolean expresssions*⟩≡

```
    and boolexp = function
      | T.RELOP(rop, e1, e2) -> spf "%%%s(%s, %s)"
                                  (T.cmm_relop rop) (valexp e1) (valexp e2)
      | e                    -> spf "%%ne(%s, 0)" (valexp e)
```

Chapter 6

Tiger Runtime System

```
% -*- mode: Noweb; noweb-code-mode: c-mode -*-% ===> this file was generated automati
% $Id: gc.nw,v 1.1.1.1 2007-01-29 16:31:59 govereau Exp $
% --------------------------------------------------------------------------
```

6.1 Allocation and Garbage Collection

```
% --------------------------------------------------------------------------

% --------------------------------------------------------------------------
```

6.1.1 Memory Allocator

```
% --------------------------------------------------------------------------
The memory allocator is written in C--, however  this
allocator is not called by compiled Tiger code. The Tiger compiler
generate in-line allocations, which may call the call_gc function,
but not the alloc function. The alloc function is only
provided as an example, and as a convenience for library functions
that need to allocate memory.
```

⟨*alloc.c–*⟩≡

```
target byteorder little;
import bits32 space_end;
```

```
import bits32 tig_gc;
export tig_alloc;
export tig_call_gc;
```

The call_gc function is a small C-- proxy for the garbage collector which is written in C. The call_gc function is called with the gc callling convention. Be sure to recompile this code whenever the gc convention changes. Otherwise compiled code and this code will make different assumptions about the convention which will result in hard-to-find bugs.

When the garbage collector is called, the allocation pointer will be overwritten. However, the garbage collector returns a new allocation pointer, so we do not need to save alloc_ptr before calling into the C code.

⟨*alloc.c–*⟩+≡

```
bits32 alloc_ptr;
tig_call_gc() {
  alloc_ptr = foreign "C" tig_gc(k) also cuts to k;
  return;
continuation k():
  return;
}
```

```
This is an example allocator---it is  not called by compiled
code. The Tiger compiler in-lines allocations which are roughly
equivalent to a call to this function, but they are more efficient.
When doing an allocation, we add an extra word at the beginning for
the garbage collector data, and then round up to the nearest machine
word.
```

⟨*alloc.c–*⟩+≡

```
tig_alloc(bits32 size) {
  bits32 p;
  p = 0;
  size = (size + 7) & 0xFFFFFFFC;
  if (alloc_ptr + size > bits32[space_end]) {
    tig_call_gc();
  }

  bits32[alloc_ptr] = size;
  p = alloc_ptr + 4;
  alloc_ptr = alloc_ptr + size;
  return(p);
}
```

```
% ------------------------------------------------------------------------
```

6.1.2 Garbage Collector Implementation

```
% ------------------------------------------------------------------------
The garbage collector exports two functions for initializing the
collector, and performing garbage collection. Both functions return
the value of the allocation pointer.
```

⟨*gc.h*⟩≡

```
void* gc_init(int heap_size);
void* tig_gc(Cmm_Cont*);
```

The Tiger garbage collector is a very simple copying collector. Most of the code is generic in that it is not specific to either Tiger or C--.

⟨*gc.c*⟩≡

```
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <string.h>
#include <qc--runtime.h>

/* global private state of GC */
static unsigned  heap_size  = 0;
static unsigned* heap       = NULL;
static unsigned* from_space = NULL;
static unsigned* to_space   = NULL;
static unsigned* alloc_ptr  = NULL;

/* This one is visible externally */
unsigned* space_end = NULL;

#define FORWARDED 0x80000000
#define SIZE_MASK 0x7FFFFFFF

#define gc_bits(x)         (*(unsigned*)(((unsigned)x) - sizeof(unsigned)))
#define forwarded(x)       (gc_bits(x) & FORWARDED)
#define forward_address(x) (*(unsigned*)(x))
#define size(x)            (gc_bits(x) & SIZE_MASK)

void set_forward_address(void* p, void* fp) {
  gc_bits(p) |= FORWARDED;
  *(unsigned*)p = (unsigned)fp;
}

/* flip also works for initialization */
void flip() {
  if (from_space == heap) {
    to_space = heap;
    from_space = (unsigned*)((unsigned)heap + heap_size);
  } else {
    from_space = heap;
    to_space = (unsigned*)((unsigned)heap + heap_size);
  }
  space_end = (unsigned*)((unsigned)from_space + heap_size);
}
```

```
void* gc_init(int size) {
  heap_size = size;
  heap = malloc(heap_size * 2);
  if (heap == NULL) {
    perror("could not create heap");
    exit(1);
  }
  bzero(heap, heap_size * 2);
  flip();
  alloc_ptr = from_space;
  return alloc_ptr;
}

void* internal_alloc(int size) {
  void *p = alloc_ptr + 1;
  assert(size > 0);
  size = (size + 7) & 0xFFFFFFFC;
  assert(size % 4 == 0);
  (*(unsigned*)alloc_ptr) = size & SIZE_MASK;
  alloc_ptr = (unsigned*)((unsigned)alloc_ptr + size);
  return p;
}

int is_pointer(unsigned p) {
  if (p < (unsigned)from_space ||
      p > (unsigned)from_space + heap_size) {
    return 0;
  }
  return 1;
}

unsigned* gc_forward(unsigned *p, int ptr) {
  void* addr;
  if (ptr == 0 || !is_pointer((unsigned)p)) return p;
  if (forwarded(p)) return (void*)forward_address(p);

  addr = internal_alloc(size(p) - 4);
  memcpy(addr, p, size(p) - 4);
  set_forward_address(p, addr);
  return addr;
}

void gc_copy(void) {
  unsigned* scan;
  for (scan = to_space; scan < alloc_ptr; scan++)
```

```
    *scan = (unsigned)gc_forward((unsigned*)*scan, -1);
}
```

```
% ------------------------------------------------------------------------
```

6.1.3 Garbage Collector Main Loop

```
% ------------------------------------------------------------------------
The Tiger compiler outputs GC data for each procedure in the form of
user spans. The span data indicates the type of each variable in the
procedure; pointer or not-pointer. There is a list of types for stack
parameters, followed by a list for temporaries. The first parameter of
a function is always the parent frame pointer, and we can safely
ignore it. We also know that the GC is only called from call_gc,
and we can safely ignore the first activation record.
```

⟨*gc.c*⟩+≡

```
void* tig_gc(Cmm_Cont* k) {
  Cmm_Activation a;
  alloc_ptr = to_space;
  space_end = (unsigned*)((unsigned)to_space + heap_size);

  a = Cmm_YoungestActivation(k);   // ignore call_gc activation
  while (Cmm_ChangeActivation(&a))
  {
    int i;
    unsigned  var_count = Cmm_LocalVarCount(&a);
    unsigned* gc_data   = Cmm_GetDescriptor(&a, 1);

    assert(!gc_data || gc_data[gc_data[0]+1] == var_count);

    /* If we have gc_data and stack vars, then we are in a proper
       tiger function. The first stack var will be the pfp and we can
       safely skip it. The assertion checks that the first stack var is a
       pointer -- it should be the parent frame pointer.
     */
    if (gc_data && gc_data[0] > 0) {
      unsigned* tig_fp          = Cmm_FindStackLabel(&a, 0);
      unsigned  stack_var_count = gc_data[0];

      assert(tig_fp);
      assert(gc_data[1] == 1);
      for (i = 1; i < stack_var_count; ++i)
        tig_fp[i] = (unsigned)gc_forward((void*)tig_fp[i], gc_data[i+1]);
    }
```

```
    /* The first local will be the pfp in a tiger procedure, but the
       forward function will ignore it. For stdlib functions we may
       need to collect the first argument.
     */
    for (i = 0; i < var_count; ++i) {
      int ptr_flg;
      unsigned** rootp = (unsigned **) Cmm_FindLocalVar(&a, i);
      if (rootp != NULL) {
        if (gc_data) ptr_flg = gc_data[gc_data[0] + 2 + i];
        else         ptr_flg = -1;
        *rootp = gc_forward(*rootp, ptr_flg);
      }
    }
  }
  gc_copy();
  bzero(from_space, heap_size);
  flip();
  return alloc_ptr;
}
```

```
% -*- mode: Noweb; noweb-code-mode: c-mode -*-% ===> this file was generated automati
% $Id: stdlib.nw,v 1.1.1.1 2007-01-29 16:31:59 govereau Exp $
% ------------------------------------------------------------------------
```

6.2 Tiger Standard Library

```
% ------------------------------------------------------------------------
```

⟨*stdlib.h*⟩≡

```
#include <stdio.h>
#include <stdlib.h>

/* Internal representation of strings */
typedef struct _string {
  unsigned length;
  unsigned char chars[1];
} string;

/* standard library funcitons */
void     tig_print(string *s);
void     tig_printi(int n);
void     tig_flush(void);
string*  tig_getchar(void);
int      tig_ord(string *s);
string*  tig_chr(unsigned i);
unsigned tig_size(string* s);
unsigned tig_sizea(void* array);
string*  tig_substring(string*, unsigned first, unsigned n);
string*  tig_concat(string *a, string *b);
int      tig_not(int i);
void     tig_exit(int status);
```

```
% ------------------------------------------------------------------------
```

6.2.1 Standard Library Implementation

```
% ------------------------------------------------------------------------
The standard library is implemented in C and C--. Whenever we cross
over to C we must save and restore the allocation pointer because
it is held in a C-- global variable which may be a register. Any
library function that needs to allocate memory is written in C-- so
that we can call the tig_alloc function without crossing over to
C and back. In addition, during allocation we may need to call the
garbage collector, and we want to ensure that local variables will be
found by the runtime system when walking the stack. Therefore, we do
not want any C frames on the stack if we can avoid it.
```

⟨*stdlib.c*⟩≡

```
#include <qc--runtime.h>
#include "stdlib.h"
#include "gc.h"
#include <string.h>
#include <assert.h>
⟨C functions⟩
```

⟨*stdlibcmm.c–*⟩≡

```
target byteorder little memsize 8 wordsize 32 pointersize 32;
import bits32 tig_alloc;
import bits32 unwinder;
import bits32 printf;
import bits32 exit;
import bits32 getchar;
import bits32 bcopy;

export tig_substring;
export tig_concat;
export tig_chr;
export tig_getchar;
export tig_set_handler;
export tig_raise;
export tig_unwind;
export tig_spawn;

bits32 alloc_ptr;
section "data" { ⟨C– data⟩ }
⟨C– functions⟩
```

Library Functions

To start off, we will define the easy one line C functions.

⟨*C functions*⟩≡

```
unsigned tig_sizea(void* array) { return *((int*)array); }
unsigned tig_size(string *s)    { return s->length;      }
int      tig_not(int i)         { return !i;             }
void     tig_exit(int status)   { exit(status);          }
void     tig_flush()            { fflush(stdout);        }
void     tig_printi(int n)      { printf("%d", n);       }
void     tig_print(string *s)   { printf("%s", s->chars);}
```

From within the standard library, strings are the only thing that we need to allocate memory for. This small function calls the allocator and initializes the memory according to the string structure definition.

⟨*C– functions*⟩≡
```
new_string(bits32 size) {
  bits32 str_ptr;
  str_ptr = tig_alloc(size + 4 + 1);
  bits32[str_ptr] = size;
  bits8[str_ptr + 4 + size] = 0 :: bits8;
  return(str_ptr);
}
```

The chr and getchar functions both allocate a string of length 1. In Tiger, EOF is equivalent to the empty string.

⟨*C– functions*⟩+≡
```
tig_chr(bits32 ch) {
  bits32 str_ptr;
  str_ptr = new_string(1);
  bits8[str_ptr+4] = %lobits8(ch);
  return(str_ptr);
}
```

⟨*C– functions*⟩+≡
```
tig_getchar() {
  bits32 ch;
  bits32 p;
  p = alloc_ptr;
  ch = foreign "C" getchar();
  alloc_ptr = p;
  p = tig_chr(ch);
  if (ch == 0xFFFFFFFF) {
    bits32[p] = 0;
  }
  return(p);
}
```

The bounds check function is called before all array accesses to provide dynamic array bounds checking.

⟨*C functions*⟩+≡
```
void tig_bounds_check(void *array, int index, int line) {
  int size = tig_sizea(array);
  if (index < 0 || index >= size) {
    fprintf(stderr, "Runtime Error line(%d): Attempt to access "
            "array index %d for array of size %d\n",
            line, index, size);
    exit(1);
  }
}
```

The ord function can only be called on strings of length one. In this case, the character is converted to an integer and returned.

⟨*C functions*⟩+≡
```
int tig_ord(string *s) {
  if (s->length != 1) {
    fprintf(stderr, "Tiger program took ord of string of length %d\n",
            s->length);
    exit(1);
  }
  return s->chars[0];
}
```

String comparison should return -1, 0, or 1 if the first string is less than, equal to, or greater that the second string.

⟨*C functions*⟩+≡

```
int tig_compare_str(string *s, string *t) {
  int i;
  assert(t);
  assert(s);
  if (s == t) return 0;

  if (s->length == t->length)
    return strncmp((const char*)s->chars,
                   (const char*)t->chars, s->length);

  i = strncmp((const char*)s->chars,
              (const char*)t->chars,
              (s->length < t->length ? s->length : t->length));

  if (i != 0) return i;
  if (s->length < t->length) return -1;
  return 1;
}
```

The implementation of substring must allocate memory to hold the new string. In addition, this function will have a live heap pointer (the input string) during the allocation, and we want to be able to locate this pointer in the case of a garbage collection. Therefore, we implement this function in C--.

⟨*C– data*⟩≡

```
substr_msg: bits8[] "substring: index (%d,%d) out of range of (0,%d)\n\000";
```

⟨*C– functions*⟩+≡

```
tig_substring("address" bits32 str_ptr, bits32 first, bits32 length) {
  bits32 new_str_ptr;
  bits32 ap;
  if (first < 0)                          { goto Lerror; }
  if (first + length > bits32[str_ptr]) { goto Lerror; }

  new_str_ptr = new_string(length);
  ap = alloc_ptr;
  foreign "C" bcopy(str_ptr+first+4, new_str_ptr+4, length);
  alloc_ptr = ap;
  return(new_str_ptr);

Lerror:
  foreign "C" printf(substr_msg, first, length, bits32[str_ptr]);
  foreign "C" exit(1) never returns;
  return(0);
}
```

String concatenation has a similar constraint as substring, and it is also implemented in C--.

⟨*C– functions*⟩+≡

```
tig_concat("address" bits32 str_a, "address" bits32 str_b) {
  bits32 new_str;
  bits32 ap;
  if (bits32[str_a] == 0) { return(str_b); }
  if (bits32[str_b] == 0) { return(str_a); }

  new_str = new_string(bits32[str_a] + bits32[str_b]);
  ap = alloc_ptr;
  foreign "C" bcopy(str_a+4, new_str+4,               bits32[str_a]);
  foreign "C" bcopy(str_b+4, new_str+4+bits32[str_a], bits32[str_b]);
  alloc_ptr = ap;
  return(new_str);
}
```

Exceptions

There are two implementations of exceptions in the Tiger compiler. The first implementation maintains the current closest exception handler dynamically using the tig_set_handler function. Raising an exception can then be done with a C-- cut to statement.

⟨*C– data*⟩+≡

```
curr_exn : bits32;
```

⟨*C– functions*⟩+≡

```
tig_set_handler(bits32 exn) {
  bits32 old_exn;
  old_exn = bits32[curr_exn];
  bits32[curr_exn] = exn;
  return(old_exn);
}
tig_raise(bits32 exn_id) {
  cut to bits32[curr_exn](exn_id);
  return;
}
```

The second implementation unwinds the stack using the C-- runtime interface.

⟨*C– functions*⟩+≡

```
tig_unwind(bits32 exn_id) {
  foreign "C" unwinder(k, exn_id) also aborts also cuts to k;
  return;
continuation k():
  return;
}
```

⟨*C functions*⟩+≡

```
void unwinder(Cmm_Cont* k, unsigned exn_id) {
  Cmm_Activation a = Cmm_YoungestActivation(k);
  do {
    if ((unsigned)Cmm_GetDescriptor(&a, 2) == 1) {
      Cmm_Cont* exn = Cmm_MakeUnwindCont(&a, 0, exn_id);
      Cmm_CutTo(exn);
      return;
    }
  } while(Cmm_ChangeActivation(&a));
  assert(0);
}
```

⟨*C– data*⟩+≡

```
spawn_msg : bits8[] "spawning to %X\n\000";
```

⟨*C– functions*⟩+≡

```
tig_spawn(bits32 lbl) {
  foreign "C" printf(spawn_msg, lbl);
  return(0);
}
```

```
% -*- mode: Noweb; noweb-code-mode: c-mode -*-% ===> this file was generated automatically by n
% $Id: runtime.nw,v 1.1.1.1 2007-01-29 16:31:59 govereau Exp $
% ----------------------------------------------------------------------------
```

6.3 Runtime Startup Code

```
The initial size of the heap is taken from the environment variable
TIGER_HEAPSIZE if present; otherwise a default is used.

% ----------------------------------------------------------------------------
```

⟨*runtime.c–*⟩≡

```
target byteorder little;
import tig_set_handler;
import gc_init;
import tiger_main;
import printf;
import getenv;
import atoi;
export main;

const HEAP_SIZE = 8192;

bits32 alloc_ptr;

section "data" {
  exn_msg           : bits8[] "unhandled exception %d\n\000";
  tiger_heapsize    : bits8[] "TIGER_HEAPSIZE\0";
  heapsize_msg      : bits8[] "heapsize %d\n\0";
}

foreign "C"
main(bits32 argc, "address" bits32 argv)
{
  bits32 rv;
  bits32 heapsize;
  bits32 env;
```

```
  env = foreign "C" getenv("address" tiger_heapsize);
  if (env != 0) {
    heapsize = foreign "C" atoi("address" env);
  } else {
    heapsize = HEAP_SIZE;
  }
  // foreign "C" printf("address" heapsize_msg, heapsize);
  alloc_ptr = foreign "C" gc_init(heapsize);

  tig_set_handler(k);
  rv = tiger_main(0) also cuts to k;
  foreign "C" return (rv);

continuation k(rv):
  foreign "C" printf(exn_msg, rv);
  foreign "C" return(-1);
}
```

```
% ------------------------------------------------------------------------
```

6.3.1 Interpreter Startup

```
% ------------------------------------------------------------------------
```

⟨*client.c*⟩≡

```
#include <assert.h>
#include <qc--interp.h>

#include "stdlib.h"
#include "gc.h"

#define BUFF_SIZE     256
#define VALSTACK_SIZE 256
#define ARGSPACE_SIZE 256
#define HEAP_SIZE     4096

typedef struct {
  Cmm_Cont cont;
  void     *stack_space;
  unsigned  stack_space_size;
  void     *limit_cookie;
} Cmm_TCB;

extern int      verbosity;
static unsigned stack_size = 65536;

static void* globals_backup = NULL;

Cmm_TCB *TCB_new(void) {
  Cmm_Dataptr  data;
  Cmm_TCB     *tcb = (Cmm_TCB *) malloc(sizeof(*tcb));
  assert(tcb != NULL);

  tcb->stack_space      = (Cmm_Dataptr) malloc(stack_size * sizeof(*data));
  mem_assert(tcb->stack_space);
  tcb->stack_space_size = stack_size;

  tcb->limit_cookie     = NULL;

  return tcb;
```

```
}

void TCB_free(Cmm_TCB *tcb) {
  free(tcb->stack_space);

  /* FIX make sure that 'cont' is freed? */
  free(tcb);
}

//extern void gc_set_thread(Cmm_Cont* t);

int main(int argc, char *argv[])
{
  verbosity = 0;

  if (Cmm_open(VALSTACK_SIZE, ARGSPACE_SIZE) != 0) {
    exit(1);
  }

  /* standard library functions */
  register_c_func("tig_print",     (void*)tig_print,     "pointer:void");
  register_c_func("tig_printi",    (void*)tig_printi,    "int:void");
  register_c_func("tig_flush",     (void*)tig_flush,     "void:void");
  register_c_func("tig_getchar",   (void*)tig_getchar,   "void:pointer");
  register_c_func("tig_ord",       (void*)tig_ord,       "pointer:int");
  register_c_func("tig_chr" ,      (void*)tig_chr,       "unsigned:pointer");
  register_c_func("tig_size" ,     (void*)tig_size,      "pointer:unsigned");
  register_c_func("tig_sizea" ,    (void*)tig_sizea,     "pointer:unsigned");
  register_c_func("tig_substring", (void*)tig_substring, "pointer,unsigned,unsigned:p
  register_c_func("tig_concat",    (void*)tig_concat,    "pointer,pointer:pointer");
  register_c_func("tig_not",       (void*)tig_not,       "int:int");
  register_c_func("tig_exit",      (void*)tig_exit,      "int:void");

  /* GC functions  */
  register_c_func("tig_gc",    (void*)tig_gc,    "void:void");
  register_c_func("tig_alloc", (void*)tig_alloc, "unsigned:pointer");
  register_c_func("tig_compare_str", (void*)tig_compare_str,
                                  "pointer,pointer:int");
  register_c_func("tig_bounds_check", (void*)tig_bounds_check,
                                  "pointer,int,int:void");

  if (!load_assembly_unit(argv[1],SRC_FILE))
  {
    Cmm_Codeptr loc = cmm_find_export("tiger_main");
    if (loc == NULL) {
      fprintf(stderr, "error: cannot find procedure \"tiger_main\"\n");
```

```
    } else {
      Cmm_TCB     *tcb = TCB_new();
      globals_backup = malloc(Cmm_GlobalSize());
      assert(globals_backup);
      tcb->cont = Cmm_CreateThread(loc,
                                   (void*)&globals_backup,
                                   tcb->stack_space,
                                   tcb->stack_space_size,
                                   &(tcb->limit_cookie));
      //gc_set_thread(&(tcb->cont));
      gc_init(atoi(getenv("TIGER_HEAPSIZE")) || HEAP_SIZE);
      tcb->cont = Cmm_RunThread(&(tcb->cont));
      gc_finish();
      free(globals_backup);
      TCB_free(tcb);
    }
  }
  Cmm_close();
  return 0;
}
```

Bibliography

[1]

Andrew~W. Appel.

Modern compiler implementation in ML: basic techniques.

Cambridge University Press, 1998.

ISBN 0-521-58775-1.

[2]

Norman Ramsey and Simon~L. Peyton~Jones.

A single intermediate language that supports multiple implementations of exceptions.

Proceedings of the ACM SIGPLAN~'00 Conference on Programming Language Design and Implementation, in *SIGPLAN Notices*, 35 (5): 285--298, May 2000.

Appendix A

Driver and Utilities

```
% -*- mode: Noweb; noweb-code-mode: caml-mode -*-% ===> this file was generated automatically b
% $Id: driver.nw,v 1.1.1.1 2007-01-29 16:31:59 govereau Exp $
% ------------------------------------------------------------------------
```

A.1 Compiler Driver

```
% ------------------------------------------------------------------------
The compiler driver has two jobs: to define the standard basis, and to
call the compiler modules in the correct order.
```

⟨*driver.ml*⟩≡

```
module S = Symbol
module F = Frame
module V = Environment
module M = Semantics
⟨standard basis⟩
⟨compiler driver⟩
```

Standard Basis

The standard basis consist of a set of initial type definitions, a set of initial functions, and a set of imports. There are only two initial type definitions in the standard basis: integers and strings.

⟨*standard basis*⟩≡

```
let base_tenv =
(* name     type *)
[ "int",    M.INT
; "string", M.STRING
]
```

The set of initial functions includes all of the functions in the Tiger standard library. For each function, we give the function name, the calling convention, and the argument and return types.

⟨*standard basis*⟩+≡

```
let base_venv =
(* name        cc        args                    return *)
[ "print",     Some "C", [M.STRING],             M.UNIT
; "printi",    Some "C", [M.INT],                M.UNIT
; "flush",     Some "C", [],                     M.UNIT
; "getchar",   None,     [],                     M.STRING
; "ord",       Some "C", [M.STRING],             M.INT
; "chr",       None,     [M.INT],                M.STRING
; "size",      Some "C", [M.STRING],             M.INT
; "sizea",     Some "C", [M.ARRAY M.ANY],        M.INT
; "substring", None,     [M.STRING;M.INT;M.INT], M.STRING
; "concat",    None,     [M.STRING;M.STRING],    M.STRING
; "not",       Some "C", [M.INT],                M.INT
; "exit",      Some "C", [M.INT],                M.UNIT
]
```

The set of imports includes all of the functions that may be called from compiled code. This includes the standard library functions as well as other internal functions such as the garbage collector and functions related to exceptions.

⟨*standard basis*⟩+≡

```
let imports =
  let internal = [ "alloc"
                 ; "call_gc"
                 ; "compare_str"
                 ; "bounds_check"
                 ; "set_handler"
                 ; "raise"
                 ; "unwind"
                 ; "spawn"
                 ]
  in
  List.map (fun(n,_,_,_) -> n) base_venv @ internal
```

Compiler Driver

The compiler driver is relatively simple. First, the command line options are parsed. Then, the input program is parsed, type checked, and converted to the intermediate representation. Finally, we iterate over the functions and output each one by calling emit_function.

⟨*compiler driver*⟩≡

```
let emit_function (frm,ex) =
  if Option.print_ext()  then Tree.print_exp ex;
  let ltree = Canonical.linearize (Tree.EXP ex) in
  if Option.print_lext() then List.iter Tree.print_stm ltree;
  List.iter (fun (x,p) -> ignore(Frame.alloc_temp frm x p))
            (Tree.find_temps ltree);
  Frame.output_header frm;
  Codegen.emit ltree;
  Frame.output_footer frm

let compile ch =
  let base_env = V.new_env base_tenv base_venv in
  let lexbuf = Lexing.from_channel ch in
  let ast = Parser.program Lexer.token lexbuf in
  let exl = Semantics.translate base_env ast in
  if Option.print_ast() then Ast.print_tree ast;
  Codegen.output_file_header imports;
  Frame.output_strings();
  List.iter emit_function exl

let _ =
  try
    Option.parse_cmdline();
    compile (Option.channel())
  with Error.Error ex ->
    Error.handle_exception ex
```

```
% -*- mode: Noweb; noweb-code-mode: caml-mode -*-% ===> this file was generated automatically b
% $Id: error.nw,v 1.1.1.1 2007-01-29 16:31:59 govereau Exp $
% ----------------------------------------------------------------------------
```

A.2 Error Handling

```
% ----------------------------------------------------------------------------
The Error module handles error reporting for the tiger compiler.
Other modules can raise exceptions defined in this module to signal
unrecoverable errors. The set of errors that can be raised is listed
below.
```

⟨*types*⟩≡

```
type error =
    Internal_error of string
  | Illegal_character of char
  | Illegal_escape of string
  | Unterminated_comment
  | Unterminated_string
  | Syntax_error
  | Type_error of string
  | Undefined_symbol of string
  | Duplicate_symbol of string
  | Illegal_break
type ex = error * int
exception Error of ex
```

```
When an exception is raised, the compiler driver will use this module
to report the error. The handle_exception function will print a
source line number and an error message, and then exit.
```

⟨*error.mli*⟩≡

```
⟨types⟩
val handle_exception : ex -> unit
```

A few functions are defined for reporting warnings and raising common errors.

⟨*error.mli*⟩+≡
```
val warning   : int -> string -> unit
val type_err  : int -> string -> 'a
val undefined : int -> string -> 'a
val internal  :        string -> 'a
```

The Error module maintains a mapping of source line numbers in order to provide more informative error messages. The scanner (Section~??) populates this mapping during the parsing phase.

⟨*error.mli*⟩+≡
```
val add_source_mapping : int -> int -> unit
val line_number : int -> int * int
```

```
% ----------------------------------------------------------------------
```

A.2.1 Error Implementation

```
% ----------------------------------------------------------------------
```

⟨*error.ml*⟩≡
⟨*types*⟩

The source map is a list of pairs of integers. The first integer is a source position, and the second is the line number that ends at that position. In effect, the scanner records the position and line number of each newline character.

⟨*error.ml*⟩+≡
```
type sm = { mutable sm: (int * int) list }
let source_map = { sm = [(0,0)] }
let add_source_mapping pos line =
  source_map.sm <- source_map.sm @ [(pos,line)]
```

To compute the line number of a source position, we do a linear search through the pairs of integers.

⟨*error.ml*⟩+≡
```
let line_number pos =
  let rec line ln last_p = function
      (p,l) :: rest ->
        if p > pos then (l, pos - last_p)
        else line l p rest
    | [] -> (ln + 1, pos - last_p)
  in line 0 0 source_map.sm
```

Errors are printed with a prefix that indicates the type of error. If the source line number can be computed, then it is also printed along with the error message.

⟨*error.ml*⟩+≡
```
let err_msg prefix pos msg =
  let (line,col) = line_number pos in
  if line > 0 then
    Printf.fprintf stderr
      "%s:%d,%d: %s\n" (Option.filename()) line col msg
  else
    Printf.fprintf stderr "%s: %s\n" prefix msg

let warning = err_msg "Warning"
```

The exception handler is used by the compiler driver (Section~??) to print error messages and exit in the case of errors.

⟨*error.ml*⟩+≡

```
let handle_exception (ex,pos) =
  let msg = match ex with
    Internal_error s     -> "Compiler bug: " ^ s
  | Illegal_character ch -> Printf.sprintf "illegal character '%c'" ch
  | Illegal_escape str   -> Printf.sprintf "illegal escape %s" str
  | Unterminated_comment -> "unterminated comment"
  | Unterminated_string  -> "unterminated string"
  | Syntax_error         -> "syntax error"
  | Type_error str       -> str
  | Undefined_symbol str -> "undefined symbol: " ^ str
  | Duplicate_symbol str -> "duplcate definition of: " ^ str
  | Illegal_break        -> "Illegal break statement"
  in
  err_msg "Error" pos msg;
  exit 1
```

Finally, here is the definition of the error raising functions.

⟨*error.ml*⟩+≡

```
let type_err  pos msg = raise(Error(Type_error msg, pos))
let undefined pos msg = raise(Error(Undefined_symbol msg, pos))
let internal      msg = raise(Error(Internal_error msg, 0))
```

```
% -*- mode: Noweb; noweb-code-mode: caml-mode -*-% ===> this file was generated automatically b
% $Id: option.nw,v 1.1.1.1 2007-01-29 16:31:59 govereau Exp $
```

A.3 Command Line Options

```
% ----------------------------------------------------------------------------
```

A.3.1 Options

```
% ----------------------------------------------------------------------------
The Option module provides command line parsing and access to
option settings. On startup, the parse_cmdline function parses the
command line and stores the state of all user settable options. Other
modules may access the value of options through the accessor
functions.
```

⟨*option.mli*⟩≡

```
val parse_cmdline : unit -> unit
val print_ast     : unit -> bool
val print_ext     : unit -> bool
val print_lext    : unit -> bool
val use_unwind    : unit -> bool
val filename      : unit -> string
val channel       : unit -> in_channel
```

```
% ------------------------------------------------------------------------
```

A.3.2 Option Implementation

```
% ------------------------------------------------------------------------
Each parameter is kept in a private variable, and the accessor
functions simply return the values of these variables.
```

⟨*option.ml*⟩≡

```
let ast    = ref false
let ext    = ref false
let lext   = ref false
let unwind = ref false
let file   = ref ""
let inch   = ref stdin

let print_ast()  = !ast
let print_ext()  = !ext
let print_lext() = !lext
let use_unwind() = !unwind
let filename()   = !file
let channel()    = !inch
```

When an input file is specified by the user, the set_input function is called.

⟨*option.ml*⟩+≡

```
let set_input s =
  try file := s; inch := open_in s
  with Sys_error err ->
    raise (Arg.Bad ("could not open file " ^ err))
```

```
The command line is processed using the Arg module from the Ocaml
standard library. First we define a set of option specs, and then call
the Arg module. After processing, we check the options for
consistency.
```

⟨*option.ml*⟩+≡

```
let rec usage() = Arg.usage options "Usage:";exit 0;
and options = [
  "-ast",      Arg.Set ast,    "\t\tprint Abstract Syntax Tree";
  "-ext",      Arg.Set ext,    "\t\tprint Expression Trees";
  "-lext",     Arg.Set lext,   "\tprint Linearized Expression Trees";
  "-unwind",   Arg.Set unwind, "\tuse unwind continuations for exceptions";
  "-help",     Arg.Unit usage, "\tprint this message";
]
let parse_cmdline() = Arg.parse options set_input "Usage:"
```

Appendix B

Scanner and Parser

```
% -*- mode: Noweb; noweb-code-mode: caml-mode -*-% ===> this file was generated autor
% $Id: parser.nw,v 1.1.1.1 2007-01-29 16:31:59 govereau Exp $
% ------------------------------------------------------------------------
```

B.1 Scanner

```
% ------------------------------------------------------------------------
```

⟨*lexer.mli*⟩≡

```
val token : Lexing.lexbuf -> Parser.token
```

⟨*lexer.mll*⟩≡

```
{
module E = Error
module P = Parser

(* The table of keywords *)
let keyword_table = Hashtbl.create 22;;
List.iter (fun (key, data) -> Hashtbl.add keyword_table key data)
  [
   "and",       P.AND;
   "array",     P.ARRAY;
   "break",     P.BREAK;
   "do",        P.DO;
   "else",      P.ELSE;
   "end",       P.END;
   "exception", P.EXCEPTION;
   "for",       P.FOR;
   "function",  P.FUNCTION;
   "handle",    P.HANDLE;
   "if",        P.IF;
   "in",        P.IN;
   "let",       P.LET;
   "nil",       P.NIL;
   "of",        P.OF;
   "or",        P.OR;
   "raise",     P.RAISE;
   "spawn",     P.SPAWN;
   "then",      P.THEN;
   "to",        P.TO;
   "try",       P.TRY;
   "type",      P.TYPE;
   "var",       P.VAR;
   "while",     P.WHILE;
 ];;

(* To buffer string literals *)

let escape c =
  match c with
  | 'n' -> '\n'
  | 'r' -> '\r'
  | 'b' -> '\b'
  | 't' -> '\t'
  | _ -> c
```

```
let line_num = ref 0
let string_start_pos = ref 0
let buffer = Buffer.create 30
let comment_pos = Stack.create()
}

let nl = ['\010' '\013']
let blank = [' ' '\009' '\012']
let letter = ['A'-'Z' 'a'-'z']
let number = ['0'-'9']
let identchar = ['A'-'Z' 'a'-'z' '_' '0'-'9']

rule token = parse
    nl   { incr line_num;
           E.add_source_mapping (Lexing.lexeme_end lexbuf) !line_num;
           token lexbuf }
  | blank + { token lexbuf }
  | letter identchar *
      { let s = Lexing.lexeme lexbuf in
        try
          Hashtbl.find keyword_table s
        with Not_found ->
          P.ID s }
  | number +
      { P.INT (int_of_string(Lexing.lexeme lexbuf)) }
  | "\""
      { string_start_pos := Lexing.lexeme_start lexbuf;
        P.STRING (string lexbuf) }
  | "/*" { comment lexbuf; token lexbuf }
  | "&"  { P.AND }
  | ":=" { P.ASSIGN }
  | ":"  { P.COLON }
  | ","  { P.COMMA }
  | "/"  { P.DIVIDE }
  | "."  { P.DOT }
  | "="  { P.EQ }
  | ">=" { P.GE }
  | ">"  { P.GT }
  | "{"  { P.LBRACE }
  | "["  { P.LBRACK }
  | "<=" { P.LE }
  | "("  { P.LPAREN }
  | "<"  { P.LT }
  | "-"  { P.MINUS }
  | "<>" { P.NEQ }
```

```
  | "|"  { P.OR }
  | "+"  { P.PLUS }
  | "}"  { P.RBRACE }
  | "]"  { P.RBRACK }
  | ")"  { P.RPAREN }
  | ";"  { P.SEMICOLON }
  | "*"  { P.TIMES }
  | eof  { P.EOF }
  | _
      { raise (E.Error(E.Illegal_character (Lexing.lexeme_char lexbuf 0),
                       Lexing.lexeme_start lexbuf)) }

and comment = parse
    "/*" { Stack.push (Lexing.lexeme_start lexbuf) comment_pos;
           comment lexbuf; }
  | "*/" { try (ignore(Stack.pop comment_pos); comment lexbuf)
           with Stack.Empty -> () }
  | nl   { incr line_num;
           E.add_source_mapping (Lexing.lexeme_end lexbuf) !line_num;
           comment lexbuf }
  | eof  { let st = Stack.top comment_pos in
           raise (E.Error(E.Unterminated_comment, st)) }
  | _    { comment lexbuf }

and string = parse
    '"'
      { let s = Buffer.contents buffer in
        (Buffer.clear buffer; s) }
  | '\\' ['\\' '\'' '"' 'n' 't' 'b' 'r']
      { Buffer.add_char buffer (escape (Lexing.lexeme_char lexbuf 1));
        string lexbuf }
  | [^ '"' '\\'] +
      { Buffer.add_string buffer (Lexing.lexeme lexbuf);
        string lexbuf }
  | eof
      { raise (E.Error(E.Unterminated_string, !string_start_pos)) }
```

```
% ------------------------------------------------------------------------
```

B.2 Parser

```
% ------------------------------------------------------------------------
```

⟨*parser.mly*⟩≡

```
%{
module E = Error
module A = Ast
module S = Symbol

let getpos = Parsing.symbol_start
let parse_error s =
  let pos = getpos() in
  raise (E.Error(E.Syntax_error, pos))

(* record creation functions *)
let mkField n t           = (n, t, getpos())

let mkSimpleVar v         = A.SimpleVar(v, getpos())
let mkFieldVar v t        = A.FieldVar(v, t, getpos())
let mkSubscriptVar v e    = A.SubscriptVar(v, e, getpos())

let mkNilExp              = A.NilExp
let mkVarExp v            = A.VarExp(v)
let mkIntExp i            = A.IntExp(i)
let mkStringExp s         = A.StringExp(s, getpos())
let mkCallExp f a         = A.CallExp(f, a, getpos())
let mkOpExp l r op        = A.OpExp(l, op, r, getpos())
let mkRecFld n e          = (n, e, getpos())
let mkRecExp n f          = A.RecordExp(n, f, getpos())
let mkSeqExp el           = A.SeqExp(el, getpos())
let mkAssignExp v e       = A.AssignExp(v, e, getpos())
let mkIfExp tst t e       = A.IfExp(tst, t, e, getpos())
let mkWhileExp tst b      = A.WhileExp(tst, b, getpos())
let mkForExp v lo hi body = A.ForExp(v, lo, hi, body, getpos())
let mkBreakExp            = A.BreakExp(getpos())
let mkLetExp decs body    = A.LetExp(decs, body, getpos())
let mkArrayExp v s init   = A.ArrayExp(v, s, init, getpos())

let mkHandler name exp    = (name, exp, getpos())
let mkTryExp exp handlers = A.TryExp(exp, handlers, getpos())
```

```
let mkRaise id            = A.RaiseExp(id, getpos())

let mkSpawn id            = A.SpawnExp(id, getpos())

let mkFunDec n f t b      = (n, f, t, b, getpos())
let mkFunctionDec l       = A.FunctionDec(l)
let mkVarDec n t i        = A.VarDec(n, t, i, getpos())
let mkTyDec n t           = (n, t, getpos())
let mkTypeDec l           = A.TypeDec(l)
let mkNameTy n            = A.NameTy(n, getpos())
let mkRecordTy l          = A.RecordTy(l)
let mkArrayTy n           = A.ArrayTy(n, getpos())
let mkException s         = A.ExceptionDec(s, getpos())
%}

/* Tokens */

%token AND ARRAY ASSIGN BREAK COLON COMMA DIVIDE DO DOT
%token ELSE END EOF EQ EXCEPTION FOR FUNCTION GE GT HANDLE IF IN
%token LBRACE LBRACK LE LET LPAREN LT MINUS NEQ NIL
%token OF OR PLUS RAISE RBRACE RBRACK RPAREN SEMICOLON SPAWN
%token TIMES THEN TO TRY TYPE VAR WHILE
%token <int> INT
%token <string> ID STRING

/* Precedences and associativities.
   The precedences must be listed from low to high. */

%nonassoc ASSIGN
%left AND OR
%nonassoc EQ NEQ GT LT GE LE
%left PLUS MINUS
%left TIMES DIVIDE
%nonassoc UMINUS

/* start symbols */

%start program
%type <Ast.exp> program
%type <Ast.exp> expr
%type <Ast.var> lvalue
%type <Ast.dec list> decs

/* %expect 63 */

%%
```

```
program:
  expr EOF { $1 }
;
expr:
   lvalue                    { mkVarExp $1 }
 | sequence                  { mkSeqExp $1 }
 | literal                   { $1 }
 | function_call             { $1 }
 | arithmetic                { $1 }
 | comparison                { $1 }
 | boolean                   { $1 }
 | construction              { $1 }
 | lvalue ASSIGN expr        { mkAssignExp $1 $3 }
 | if_statement              { $1 }
 | loop_statement            { $1 }
 | LET decs IN expr_list END { mkLetExp $2 (mkSeqExp $4) }
 | TRY expr handlers         { mkTryExp $2 (List.rev $3) }
 | RAISE id                  { mkRaise $2 }
 | SPAWN id                  { mkSpawn $2 }
;

/* Symbols are created in this rule only */
id: ID { S.symbol $1 };

/* Variables (L-values)
   This rule is overly explicit to avoid conflicts with
   the construction rule below */
lvalue:
 | id                        { mkSimpleVar $1 }
 | id DOT id                 { mkFieldVar (mkSimpleVar $1) $3 }
 | id LBRACK expr RBRACK     { mkSubscriptVar (mkSimpleVar $1) $3 }
 | lvalue DOT id             { mkFieldVar $1 $3 }
 | lvalue LBRACK expr RBRACK { mkSubscriptVar $1 $3 }
;

/* Sequence expression */
sequence:
   LPAREN RPAREN           { [] }
 | LPAREN expr_list RPAREN { $2 }
;
expr_list:
   expr                     { $1 :: [] }
 | expr SEMICOLON expr_list { $1 :: $3 }
;
```

```
/* Literals */
literal:
   NIL    { mkNilExp }
 | INT    { mkIntExp $1 }
 | STRING { mkStringExp $1 }
;

/* function call */
function_call:
   id LPAREN fun_args RPAREN { mkCallExp $1 $3 }
;
fun_args:
   /* empty */         { [] }
 | expr                { $1 :: [] }
 | expr COMMA fun_args { $1 :: $3 }
;

/* Simple Arithmetic */
arithmetic:
   MINUS expr %prec UMINUS { mkOpExp (mkIntExp 0) $2 A.MinusOp }
 | expr PLUS expr          { mkOpExp $1 $3 A.PlusOp    }
 | expr MINUS expr         { mkOpExp $1 $3 A.MinusOp  }
 | expr TIMES expr         { mkOpExp $1 $3 A.TimesOp  }
 | expr DIVIDE expr        { mkOpExp $1 $3 A.DivideOp }
;

/* Comparison */
comparison:
   expr EQ expr  { mkOpExp $1 $3 A.EqOp  }
 | expr NEQ expr { mkOpExp $1 $3 A.NeqOp }
 | expr GT expr  { mkOpExp $1 $3 A.GtOp  }
 | expr LT expr  { mkOpExp $1 $3 A.LtOp  }
 | expr GE expr  { mkOpExp $1 $3 A.GeOp  }
 | expr LE expr  { mkOpExp $1 $3 A.LeOp  }
;

/* Boolean operators */
boolean:
   expr AND expr { mkIfExp $1 $3 (Some(mkIntExp 0)) }
 | expr OR expr  { mkIfExp $1 (mkIntExp 1) (Some $3) }
;

/* Record and array construction */
construction:
   id LBRACE ctor_list RBRACE    { mkRecExp $1 $3 }
 | id LBRACK expr RBRACK OF expr { mkArrayExp $1 $3 $6 }
```

```
;
ctor_list:
   id EQ expr                 { (mkRecFld $1 $3) :: [] }
 | id EQ expr COMMA ctor_list { (mkRecFld $1 $3) :: $5 }
;

/* If statements */
if_statement:
   IF expr THEN expr           { mkIfExp $2 $4 None }
 | IF expr THEN expr ELSE expr { mkIfExp $2 $4 (Some $6) }
;

/* Loop statements */
loop_statement:
   WHILE expr DO expr                 { mkWhileExp $2 $4 }
 | FOR id ASSIGN expr TO expr DO expr { mkForExp $2 $4 $6 $8 }
 | BREAK                              { mkBreakExp }
;

/* exception handlers */
handler:
   HANDLE id expr END         { mkHandler $2 $3 }
;
handlers:
   handler          { $1 :: [] }
 | handler handlers { $1 :: $2 }
;

/* Declarations */

/* recursive declarations in tiger.
   My interpretation of the tiger language spec is that
   mutally recursive types and functions are valid if they
   are declared together in a sequence. That is:
   type a = {b:int c:d} type d = a
   is valid where as
   type a = {b:int c:d} var x := 1 type d = a
   is not.
*/
decs:
   dec { $1 :: [] }
 | dec decs { $1 :: $2 }
;
dec:
   var_dec   { $1 }
 | type_decs { mkTypeDec $1 }
```

```
 | fun_decs  { mkFunctionDec $1 }
 | exn_dec   { $1 }
;
var_dec:
   VAR id ASSIGN expr          { mkVarDec $2 None $4 }
 | VAR id COLON id ASSIGN expr { mkVarDec $2 (Some $4) $6 }
;
type_decs:
   type_dec           { $1 :: [] }
 | type_dec type_decs { $1 :: $2 }
;
type_dec:
   TYPE id EQ ty { mkTyDec $2 $4 }
;
ty:
   id                       { mkNameTy $1 }
 | LBRACE ty_fields RBRACE { mkRecordTy $2 }
 | ARRAY OF id             { mkArrayTy $3 }
;
ty_fields:
   /* empty */                 { [] }
 | id COLON id                 { (mkField $1 $3) :: [] }
 | id COLON id COMMA ty_fields { (mkField $1 $3) :: $5 }
;
fun_decs:
   fun_dec          { $1 :: [] }
 | fun_dec fun_decs { $1 :: $2 }
;
fun_dec:
   FUNCTION id LPAREN ty_fields RPAREN EQ expr
     { mkFunDec $2 $4 None $7 }
 | FUNCTION id LPAREN ty_fields RPAREN COLON id EQ expr
     { mkFunDec $2 $4 (Some $7) $9 }
;
exn_dec:
   EXCEPTION id { mkException $2 }
;
```

Appendix C

Linearize Algorithm

```
% -*- mode: Noweb; noweb-code-mode: caml-mode -*-% ===> this file was generated autor
% $Id: canonical.nw,v 1.1.1.1 2007-01-29 16:31:59 govereau Exp $
% ----------------------------------------------------------------------------
```

C.1 Canonical Trees

```
% ----------------------------------------------------------------------------
This module reduces a Tree to canonical form. Currently there are
only two requirements for a tree in canonical form:
```

- The tree contains no SEQ or ESEQ nodes.
- The parent of every CALL node is either an EXP or a MOVE(...,TEMP _).

```
The algorithm implemented here is almost identical to the algorithm
found in Chapter~8 of [?].
```

⟨*canonical.mli*⟩≡

```
val linearize : Tree.stm -> Tree.stm list
```

⟨*canonical.ml*⟩≡

```
module T = Tree
module S = Symbol

let nop = T.EXP(T.CONST 0)

let ( % ) x y =
  match (x,y) with
    (T.EXP(T.CONST _), _) -> y
  | (_, T.EXP(T.CONST _)) -> x
  | _ -> T.SEQ(x,y)

let commute = function
    (T.EXP(T.CONST _), _) -> true
  | (_, T.NAME _) -> true
  | (_, T.CONST _) -> true
  | _ -> false


let linearize stm0 =
  let rec reorder = function
      T.CALL(_,_,_,_,ptr) as call :: rest ->
        let t = T.TEMP(T.new_temp(), ptr)
        in reorder(T.ESEQ(T.MOVE(call, t), t) :: rest)
    | a :: rest ->
        let (stms,e) = do_exp a
        and (stms',el) = reorder rest in
        if commute(stms',e)
        then (stms % stms',e::el)
        else let t = T.TEMP(T.new_temp(), T.is_ptr e) in
        (stms % T.MOVE(e, t) % stms', t :: el)
    | [] -> (nop,[])

  and reorder_exp(el,build) =
    let (stms,el') = reorder el
    in (stms, build el')

  and reorder_stm(el,build) =
    let (stms,el') = reorder el
    in stms % (build el')

  and do_stm = function
      T.SEQ(a,b) ->
        do_stm a % do_stm b
```

```
  | T.JUMP e ->
      let f l = T.JUMP (List.hd l)
      in reorder_stm([e], f)
  | T.CJUMP(e,t,f) ->
      let f l = T.CJUMP(List.hd l, t, f)
      in reorder_stm([e], f)
  | T.MOVE(T.CALL(e,el,ext,k,ptr),T.TEMP(t,_)) ->
      let f l = T.MOVE(T.CALL(List.hd l, List.tl l, ext, k, ptr), T.TEMP(t,ptr))
      in reorder_stm(e :: el, f)
  | T.MOVE(b,T.TEMP(t,ptr)) ->
      let f l = T.MOVE(List.hd l, T.TEMP(t,ptr))
      in reorder_stm([b], f)
  | T.MOVE(b,T.NAME n) ->
      let f l = T.MOVE(List.hd l, T.NAME n)
      in reorder_stm([b], f)
  | T.MOVE(T.ESEQ(s,e), b) ->
      do_stm(T.SEQ(s,T.MOVE(e,b)))
  | T.MOVE(b,T.MEM(e,ptr)) ->
      let f l = T.MOVE(List.hd l, T.MEM(List.nth l 1, ptr))
      in reorder_stm([b ; e], f)
  | T.MOVE(b,T.ESEQ(s,e)) ->
      do_stm(T.SEQ(s,T.MOVE(b,e)))
  | T.EXP(T.CALL(e,el,ext,k,ptr)) ->
      let f l = T.EXP(T.CALL(List.hd l, List.tl l, ext, k, ptr))
      in reorder_stm(e :: el, f)
  | T.EXP e ->
      let f l = T.EXP (List.hd l)
      in reorder_stm([e], f)
  | s ->
      let f l = s
      in reorder_stm([], f)

and do_exp = function
    T.BINOP(p,a,b) ->
      let f l = T.BINOP(p, List.hd l, List.nth l 1)
      in reorder_exp([a ; b], f)
  | T.RELOP(p,a,b) ->
      let f l = T.RELOP(p, List.hd l, List.nth l 1)
      in reorder_exp([a ; b], f)
  | T.MEM(a,ptr) ->
      let f l = T.MEM (List.hd l,ptr)
      in reorder_exp([a], f)
  | T.ESEQ(s,e) ->
      let stms = do_stm s
      and (stms',e) = do_exp e
      in (stms%stms',e)
```

```
    | T.CALL(e,el,ext,k,ptr) ->
        let f l = T.CALL(List.hd l, List.tl l, ext, k, ptr)
        in reorder_exp(e :: el, f)
    | e ->
        let f l = e
        in reorder_exp([], f)

  (* linear gets rid of the top-level SEQ's, producing a list *)
  and linear = function
      (T.SEQ(a,b),l) -> linear(a,linear(b,l))
    | (s,l) -> s :: l

in (* body of linearize *)
   linear(do_stm stm0, [])
```