The Tiger Compiler `tigerc`
version 0.1

Yoann Padioleau
yoann.padioleau@gmail.com

with code from
Paul Govereau

August 17, 2026

Contents

Chapter 1

Introduction

The goal of this book is to present in full details the source code of a compiler.

1.1 Motivations

Why a compiler? Because we think you are a better programmer if you fully understand how things work under the hood.

1.2 The Tiger compiler

1.3 Other compilers

Here are other candidates that were considered but ultimately discarded:

- gcc
- clang

1.4 Getting started

1.5 Requirements

1.6 About this document

1.7 Copyright

1.8 Acknowledgments

Chapter 2

Overview

2.1 Compiler principles

2.2 `tigerc` command line interface

⟨*function Option.usage* 8a⟩≡ (62g)

```
let rec usage() =
  Arg.usage options "Usage:";
  exit 0;
```

2.3 `helloworld.tig`

⟨*demos/hello.tig* 8b⟩≡

```
/* hello world */
let
  var a := "Hello, world.\n"
in
  print(a)
end
```

2.4 Input Tiger language

2.5 Output `C--` Portable assembly language

2.6 Output ARM assembly language

2.7 Code organization

2.8 Software architecture

2.9 Book structure

Chapter 3

Core Data Structures

3.1 Abstract syntax tree

3.1.1 Positions

⟨*type Ast.pos* 9a⟩≡ (64)
```
type pos = int
```

3.1.2 Names

⟨*type Ast.name* 9b⟩≡ (64)
```
type name = Symbol.symbol
```

⟨*type Ast.typename* 9c⟩≡ (64)
```
type typename = Symbol.symbol
```

3.1.3 Definitions

⟨*type Ast.dec* 9d⟩≡ (64)
```
type dec =
| VarDec       of name * typename option * exp * pos
| TypeDec      of (typename * ty * pos) list
| FunctionDec  of (name * field list * typename option * exp * pos) list
⟨Ast.dec cases 47f⟩
```

⟨*type Ast.field* 9e⟩≡ (64)
```
and field = (name * typename * pos)
```

3.1.4 Expressions

⟨*type Ast.exp* 9f⟩≡ (64)
```
and exp =
| NilExp
| IntExp    of int
| StringExp of string * pos

| VarExp    of var

| OpExp     of exp * oper * exp * pos
| CallExp   of name * exp list * pos

| RecordExp of name * (name * exp * pos) list * pos
```

```
  | ArrayExp  of name * exp * exp * pos

  | LetExp    of dec list * exp * pos
```

⟨Ast.exp *statement cases* 10b⟩
⟨Ast.exp *cases* 47h⟩

⟨*type Ast.var* 10a⟩≡ (64)

```
and var =
    SimpleVar    of name * pos
  | FieldVar     of var * name * pos
  | SubscriptVar of var * exp * pos
```

3.1.5 Statements

⟨Ast.exp *statement cases* 10b⟩≡ (9f)

```
  | AssignExp of var * exp * pos
  | SeqExp    of exp list * pos
  | IfExp     of exp * exp * exp option * pos
  | WhileExp  of exp * exp * pos
  | ForExp    of name * exp * exp * exp * pos
  | BreakExp  of pos
```

3.1.6 Types

⟨*type Ast.ty* 10c⟩≡ (64)

```
and ty =
| NameTy   of typename * pos
| RecordTy of field list
| ArrayTy  of typename * pos
```

3.2 Type system

⟨*type Environment.vartype* 10d⟩≡ (67a 66a)

```
type vartype =
  (* basic types *)
  | UNIT
  | INT
  | STRING

  (* aggregate types *)
  | RECORD of (Ast.name * vartype) list
  | ARRAY  of vartype

  (* other types *)
  ⟨Environment.vartype cases 10e⟩
```

⟨Environment.vartype *cases* 10e⟩≡ (10d) 10f▷

```
  | NIL
```

⟨Environment.vartype *cases* 10f⟩+≡ (10d) ◁10e 47a▷

```
  | ANY
```

3.3 Symbol table

3.3.1 Unique symbols

⟨*type Symbol.symbol* 11a⟩≡ (63b)
```
type symbol = string * int
```

⟨*global Symbol.nextsym* 11b⟩≡ (63b)
```
let nextsym = ref 0
```

⟨*global Symbol.hashtable* 11c⟩≡ (63b)
```
let (hashtable: (string, int) Hashtbl.t) = Hashtbl.create 128
```

⟨*function Symbol.name* 11d⟩≡ (63b)
```
let name = fst
```

⟨*function Symbol.uid* 11e⟩≡ (63b)
```
let uid  = snd
```

⟨*function Symbol.symbol* 11f⟩≡ (63b)
```
let symbol name =
  try
    let uid = Hashtbl.find hashtable name in
    (name,uid)
  with Not_found ->
    incr nextsym;
    Hashtbl.add hashtable name !nextsym;
    (name, !nextsym)
```

3.3.2 Symbol table

⟨*type Symbol.table* 11g⟩≡ (63e)
```
type 'a table = {
    tbl    : (symbol, 'a) Hashtbl.t;
    ⟨Symbol.table other fields 12a⟩
  }
```

⟨*function Symbol.enter* 11h⟩≡ (63e)
```
let enter env sym v =
  if Hashtbl.mem env.tbl sym
  then failwith "Compiler error: symbol table duplicate entry"
  else Hashtbl.add env.tbl sym v
```

⟨*function Symbol.mem* 11i⟩≡ (63g)
```
let mem env sym =
  Hashtbl.mem env.tbl sym
```

⟨*constructor Symbol.create* 11j⟩≡ (63h)
```
let create l =
  let env = {
    tbl = Hashtbl.create 20;
    level = 0;
    parent = None
  } in
  l |> List.iter (fun (key, data) -> enter env (symbol key) data);
  env
```

3.3.3 Scope managment

⟨Symbol.table *other fields* 12a⟩≡ (11g) 12d▷
```
parent : 'a table option;
```

⟨*function Symbol.look* 12b⟩≡ (63f)
```
let rec look env sym =
  try Hashtbl.find env.tbl sym
  with Not_found ->
    (match env.parent with
    | None -> raise Not_found
    | Some e -> look e sym
    )
```

⟨*constructor* Symbol.new_scope 12c⟩≡ (63h)
```
let new_scope env = {
  tbl = Hashtbl.create 20;
  level = env.level + 1;
  parent = Some env
}
```

⟨Symbol.table *other fields* 12d⟩+≡ (11g) ◁12a
```
level  : int;
```

3.4 Intermediate code

3.4.1 Instructions and labels

⟨*type Tree.stm* 12e⟩≡ (69a 68d)
```
type stm =
  | EXP    of exp
  | MOVE   of exp * exp

  | LABEL  of label
  | JUMP   of label
  | CJUMP  of exp * label * label
  | RET    of exp
  ⟨Tree.stm cases 13c⟩
```

⟨*type Tree.label* 12f⟩≡ (69a 68d)
```
type label = Symbol.symbol
```

3.4.2 Basic expressions and temporaries

⟨*type Tree.exp* 12g⟩≡ (69a 68d)
```
and exp =
  | CONST of int
  | BINOP of binop * exp * exp
  | RELOP of relop * exp * exp

  | NAME  of label
  | TEMP  of temp * is_ptr
  | MEM   of exp * is_ptr (* dereference? *x ? *)

  | CALL  of exp * exp list * string option * label option * is_ptr
  ⟨Tree.exp cases 13d⟩
```

⟨*type Tree.temp* 13a⟩≡ (69a 68d)
```
and  temp  = Symbol.symbol
```

⟨*types Tree.xxxop* 13b⟩≡ (69a 68d)
```
and binop = PLUS | MINUS | MUL | DIV
and relop = EQ | NE   | LT | GT | LE | GE
```

⟨`Tree.stm` *cases* 13c⟩≡ (12e) 48k ▷
```
| SEQ    of stm * stm
```

⟨`Tree.exp` *cases* 13d⟩≡ (12g)
```
| ESEQ  of stm * exp
```

3.4.3 Pointers

⟨*type* `Tree.is_ptr` 13e⟩≡ (69a 68d)
```
type is_ptr = bool
```

⟨*function* `Semantics.is_ptr` 13f⟩≡ (68a)
```
let is_ptr = function
    INT | UNIT -> false
  | NIL | RECORD _ | STRING | ARRAY _ -> true
  | _ -> E.internal "non-base type for variable"
```

3.4.4 Stack frames

⟨*type Frame.frame* 13g⟩≡ (73g)
```
type frame = {
  mutable params : (Tree.label * Tree.is_ptr) list;
  mutable vars   : (Tree.label * Tree.is_ptr) list;
  mutable temps  : (Tree.label * Tree.is_ptr) list;
  ⟨Frame.frame other fields 13h⟩
}
```

⟨`Frame.frame` *other fields* 13h⟩≡ (13g) 37d ▷
```
name           : Tree.label;
```

Chapter 4

main()

⟨*function Main.main* 14a⟩≡ (89a)
```
let main () =
  try
    Option.parse_cmdline();
    compile !Option.inch
  with Error.Error ex ->
    Error.handle_exception ex
```

⟨*toplevel* Main._ 14b⟩≡ (89a)
```
let _ =
  main ()
```

⟨*signature global Option.inch* 14c⟩≡ (62d)
```
val inch      : in_channel ref
```

⟨*command line flags* 14d⟩≡ (62e) 14h▷
```
let inch   = ref stdin
```

4.1 Command line processing

⟨*signature function* Option.parse_cmdline 14e⟩≡ (62d)
```
val parse_cmdline : unit -> unit
```

⟨*function* Option.parse_cmdline 14f⟩≡ (62g)
```
let parse_cmdline() =
  Arg.parse options set_input "Usage:"
```

⟨*constant Option.options* 14g⟩≡ (62g)
```
and options = [
  ⟨command line options 49d⟩
  "-help",     Arg.Unit usage, "\tprint this message";
]
```

⟨*command line flags* 14h⟩+≡ (62e) ◁14d 49c▷
```
let file   = ref ""
```

⟨*function* Option.set_input 14i⟩≡ (62f)
```
let set_input s =
  try
    file := s;
    inch := open_in s
  with Sys_error err ->
    raise (Arg.Bad ("could not open file " ^ err))
```

4.2 Main.compile()

⟨*function Main.compile* 15a⟩≡ (89a)
```
let compile ch =
  (* parsing *)
  let lexbuf = Lexing.from_channel ch in
  let ast = Parser.program Lexer.token lexbuf in
  ⟨Main.compile() if dump AST option 56c⟩

  (* checking and compiling part1 *)
  let base_env = Environment.new_env base_tenv base_venv in
  let xs = Semantics.translate base_env ast in

  ⟨Main.compile() generate headers 44a⟩
  ⟨Main.compile() generate data before functions 44b⟩

  (* compiling part2 and generating *)
  xs |> List.iter emit_function
```

⟨*signature function Lexer.token* 15b⟩≡ (65c)
```
val token : Lexing.lexbuf -> Parser.token
```

⟨*signature function Parser.program* 15c⟩≡
```
val program :
  (Lexing.lexbuf  -> token) -> Lexing.lexbuf -> Ast.exp
```

⟨*signature function* Environment.new_env 15d⟩≡ (66a)
```
val new_env : (string * vartype) list ->
              (string * string option * vartype list * vartype) list -> t
```

⟨*signature function Semantics.translate* 15e⟩≡ (67k)
```
val translate : Environment.t -> Ast.exp -> (Frame.frame * Tree.exp) list
```

4.3 Main.emit_function()

⟨*function* Main.emit_function 15f⟩≡ (89a)
```
let emit_function (frm, ex) =
  (* compiling *)
  ⟨Main.emit_function() if dump expression tree 59b⟩
  let ltree = Canonical.linearize (Tree.EXP ex) in
  ⟨Main.emit_function() if dump linearized tree 59g⟩
  ⟨Main.emit_function() adjust frm with temporaries in ltree 42d⟩

  (* generating *)
  Frame.output_header frm;
  Codegen.emit ltree;
  Frame.output_footer frm
```

⟨*signature function Canonical.linearize* 15g⟩≡ (72c)
```
val linearize : Tree.stm -> Tree.stm list
```

⟨*signature function* Frame.output_header 15h⟩≡ (73e)
```
val output_header  : frame -> unit
```

⟨*signature function* Frame.output_footer 15i⟩≡ (73e)
```
val output_footer  : frame -> unit
```

⟨*signature Codegen.emit* 15j⟩≡ (75c)
```
val emit               : Tree.stm list -> unit
```

Chapter 5

Lexing

5.1 Overview

⟨lexer.mll 16a⟩≡
```
{
module E = Error
module P = Parser

⟨global Lexer.keyword_table 18a⟩
⟨function Lexer.escape 19f⟩
⟨Lexer globals 17b⟩
}

⟨Lexer aliases 17a⟩

⟨rule token 16b⟩
⟨rule comment 17f⟩
⟨rule string 19e⟩
```

⟨rule token 16b⟩≡ (16a)
```
rule token = parse
  ⟨Lexer.token cases 17c⟩
  | eof  { P.EOF }
  | _
      { raise (E.Error(E.Illegal_character (Lexing.lexeme_char lexbuf 0),
                       Lexing.lexeme_start lexbuf)) }
```

⟨token declarations 16c⟩≡ (20a) 47d▷
```
%token <string> ID
%token <int> INT
%token <string> STRING
%token IF THEN ELSE END  WHILE DO FOR TO BREAK
%token PLUS MINUS TIMES DIVIDE
%token AND OR
%token EQ NEQ  LT LE GT GE
%token FUNCTION VAR LET IN
%token TYPE OF ARRAY
%token ASSIGN COLON COMMA DOT SEMICOLON
%token LPAREN RPAREN  LBRACE RBRACE  LBRACK RBRACK
%token NIL
%token EOF
```

5.2 Spaces and newlines

⟨*Lexer aliases* 17a⟩≡ (16a) 17g▷
```
let nl = ['\010' '\013']
let blank = [' ' '\009' '\012']
```

⟨*Lexer globals* 17b⟩≡ (16a) 17e▷
```
let line_num = ref 0
```

⟨Lexer.token *cases* 17c⟩≡ (16b) 17d▷
```
| nl    { incr line_num;
          E.add_source_mapping (Lexing.lexeme_end lexbuf) !line_num;
          token lexbuf }
| blank + { token lexbuf }
```

5.3 Comments

⟨Lexer.token *cases* 17d⟩+≡ (16b) ◁17c 17h▷
```
| "/*" { comment lexbuf; token lexbuf }
```

⟨*Lexer globals* 17e⟩+≡ (16a) ◁17b 19d▷
```
let comment_pos = Stack.create()
```

⟨*rule comment* 17f⟩≡ (16a)
```
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 }
```

5.4 Identifiers and keywords

⟨*Lexer aliases* 17g⟩+≡ (16a) ◁17a 19a▷
```
let letter = ['A'-'Z' 'a'-'z']
let identchar = ['A'-'Z' 'a'-'z' '_' '0'-'9']
```

⟨Lexer.token *cases* 17h⟩+≡ (16b) ◁17d 18b▷
```
| letter identchar *
    { let s = Lexing.lexeme lexbuf in
      try
        Hashtbl.find keyword_table s
      with Not_found ->
        P.ID s
    }
```

⟨*global* `Lexer.keyword_table` 18a⟩≡ (16a)

```
(* The table of keywords *)
let keyword_table = Hashtbl.create 22;;
List.iter (fun (key, data) -> Hashtbl.add keyword_table key data)
  [
   "if",         P.IF;
   "then",       P.THEN;
   "else",       P.ELSE;
   "while",      P.WHILE;
   "do",         P.DO;
   "for",        P.FOR;
   "to",         P.TO;
   "end",        P.END;
   "break",      P.BREAK;

   "function",   P.FUNCTION;
   "var",        P.VAR;
   "let",        P.LET;
   "in",         P.IN;

   "type",       P.TYPE;
   "of",         P.OF;
   "array",      P.ARRAY;

   "and",        P.AND;
   "or",         P.OR;

   "nil",        P.NIL;

   ⟨Lexer.keyword_table entries 47e⟩
 ]
```

5.5 Operators

⟨`Lexer.token` *cases* 18b⟩+≡ (16b) ◁17h 18c▷

```
| "+"  { P.PLUS }
| "-"  { P.MINUS }
| "*"  { P.TIMES }
| "/"  { P.DIVIDE }

| "&"  { P.AND }
| "|"  { P.OR }

| "="  { P.EQ }
| "<>" { P.NEQ }

| ">"  { P.GT }
| "<"  { P.LT }
| ">=" { P.GE }
| "<=" { P.LE }
```

5.6 Punctuations

⟨`Lexer.token` *cases* 18c⟩+≡ (16b) ◁18b 19b▷

```
| ":=" { P.ASSIGN }
| ":"  { P.COLON }
```

```
  | ","  { P.COMMA }
  | "."  { P.DOT }
  | ";"  { P.SEMICOLON }

  | "{"  { P.LBRACE } | "}"  { P.RBRACE }
  | "["  { P.LBRACK } | "]"  { P.RBRACK }
  | "("  { P.LPAREN } | ")"  { P.RPAREN }
```

5.7 Numbers

⟨*Lexer aliases* 19a⟩+≡ (16a) ◁17g
```
  let number = ['0'-'9']
```

⟨Lexer.token *cases* 19b⟩+≡ (16b) ◁18c 19c▷
```
  | number +
      { P.INT (int_of_string(Lexing.lexeme lexbuf)) }
```

5.8 Strings

⟨Lexer.token *cases* 19c⟩+≡ (16b) ◁19b
```
  | "\""
      { string_start_pos := Lexing.lexeme_start lexbuf;
        Buffer.clear buffer;
        P.STRING (string lexbuf) }
```

⟨*Lexer globals* 19d⟩+≡ (16a) ◁17e
```
  let string_start_pos = ref 0
  let buffer = Buffer.create 30
```

⟨*rule string* 19e⟩≡ (16a)
```
  and string = parse
      '"'
        { Buffer.contents buffer }
    | '\\' ['\\' '\'' '"' '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)) }
```

⟨*function Lexer.escape* 19f⟩≡ (16a)
```
  (* To buffer string literals *)
  let escape c =
    match c with
    | 'n' -> '\n'
    | 'r' -> '\r'
    | 'b' -> '\b'
    | 't' -> '\t'
    | _ -> c
```

Chapter 6

Parsing

6.1 Overview

⟨parser.mly 20a⟩≡
```
%{
module E = Error
module A = Ast
module S = Symbol

⟨parser helper functions ??⟩
⟨AST mk wrappers ??⟩
%}

/* Tokens */
⟨token declarations 16c⟩

/* Precedences and associativities (from low to high) */
⟨token priorities 23a⟩

/* start symbols */
%start program
⟨rule type declarations 21a⟩

/* %expect 63 */

%%
⟨grammar 20b⟩
```

⟨grammar 20b⟩≡ (20a)
```
⟨rule program 21b⟩

/* Expressions */
⟨rule expr 21c⟩
⟨rule lvalue 22e⟩
⟨subrules for expr 22d⟩
⟨subrules for stmt 24d⟩

/* Declarations */
⟨rule decs 21d⟩
⟨subrules for decs 21e⟩

/* Types */
⟨rule ty 25c⟩

/* Names */
```

⟨*rule id* 21h⟩

⟨*rule type declarations* 21a⟩≡ (20a)
```
%type <Ast.exp> program
%type <Ast.exp> expr
%type <Ast.var> lvalue
%type <Ast.dec list> decs
```

⟨*rule program* 21b⟩≡ (20b)
```
program:
  expr EOF { $1 }
```

⟨*rule expr* 21c⟩≡ (20b) 22c ▷
```
expr:
  LET decs IN expr_list END { mkLetExp $2 (mkSeqExp $4) }
```

6.2 Definitions

⟨*rule decs* 21d⟩≡ (20b)
```
/* 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 }
```

⟨*subrules for decs* 21e⟩≡ (20b)
```
⟨rule dec 21f⟩
⟨rule var_dec 21g⟩
⟨rule type_decs 22a⟩
⟨rule fun_decs 22b⟩
⟨rule exn_dec 47g⟩
```

⟨*rule dec* 21f⟩≡ (21e)
```
dec:
   var_dec   { $1 }
 | type_decs { mkTypeDec $1 }
 | fun_decs  { mkFunctionDec $1 }
 | exn_dec   { $1 }
```

6.2.1 Variables

⟨*rule* var_dec 21g⟩≡ (21e)
```
var_dec:
   VAR id ASSIGN expr          { mkVarDec $2 None $4 }
 | VAR id COLON id ASSIGN expr { mkVarDec $2 (Some $4) $6 }
```

⟨*rule id* 21h⟩≡ (20b)
```
/* Symbols are created in this rule only */
id: ID { S.symbol $1 };
```

6.2.2 Types

⟨*rule* type_decs 22a⟩≡ (21e)

```
type_decs:
   type_dec           { $1 :: [] }
 | type_dec type_decs { $1 :: $2 }
;
type_dec:
   TYPE id EQ ty { mkTyDec $2 $4 }
;
```

6.2.3 Functions

⟨*rule* fun_decs 22b⟩≡ (21e)

```
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 }
```

6.3 Expressions

⟨*rule expr* 22c⟩+≡ (20b) ◁21c 24c▷

```
   | literal                { $1 }
   | lvalue                 { mkVarExp $1 }
   | function_call          { $1 }
   | arithmetic             { $1 }
   | comparison             { $1 }
   | boolean                { $1 }
   | construction           { $1 }
```

6.3.1 Literals

⟨*subrules for expr* 22d⟩≡ (20b) 23d▷

```
/* Literals */
literal:
   NIL    { A.NilExp }
 | INT    { mkIntExp $1 }
 | STRING { mkStringExp $1 }
```

6.3.2 Variable, array and field accesses

⟨*rule lvalue* 22e⟩≡ (20b)

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

 | lvalue DOT id            { mkFieldVar $1 $3 }
 | lvalue LBRACK expr RBRACK { mkSubscriptVar $1 $3 }
```

6.3.3 Arithmetic

⟨*token priorities* 23a⟩≡ (20a)
```
%nonassoc ASSIGN
%left AND OR
%nonassoc EQ NEQ GT LT GE LE
%left PLUS MINUS
%left TIMES DIVIDE
%nonassoc UMINUS
```

⟨*type Ast.oper* 23b⟩≡ (64)
```
and oper =
⟨Ast.oper cases 23c⟩
```

⟨Ast.oper *cases* 23c⟩≡ (23b) 23e▷
```
| PlusOp | MinusOp | TimesOp | DivideOp
```

⟨*subrules for expr* 23d⟩+≡ (20b) ◁22d 23f▷
```
/* 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 }
```

6.3.4 Comparison

⟨Ast.oper *cases* 23e⟩+≡ (23b) ◁23c
```
| EqOp | NeqOp
| LtOp | LeOp | GtOp | GeOp
```

⟨*subrules for expr* 23f⟩+≡ (20b) ◁23d 23g▷
```
/* 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  }
```

6.3.5 Boolean

⟨*subrules for expr* 23g⟩+≡ (20b) ◁23f 24a▷
```
/* Boolean operators */
boolean:
   expr AND expr { mkIfExp $1 $3 (Some(mkIntExp 0)) }
 | expr OR expr  { mkIfExp $1 (mkIntExp 1) (Some $3) }
```

6.3.6 Function calls

⟨*subrules for expr* 24a⟩+≡ (20b) ◁23g 24b▷

```
/* function call */
function_call:
   id LPAREN fun_args RPAREN { mkCallExp $1 $3 }

fun_args:
   /* empty */         { [] }
 | expr                { $1 :: [] }
 | expr COMMA fun_args { $1 :: $3 }
```

6.3.7 Constructions

⟨*subrules for expr* 24b⟩+≡ (20b) ◁24a

```
/* 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 }
```

6.3.8 Local entity definitions

6.4 Statements

⟨*rule expr* 24c⟩+≡ (20b) ◁22c 24f▷

```
 | sequence                { mkSeqExp $1 }
 | if_statement            { $1 }
 | loop_statement          { $1 }
```

6.4.1 Sequence

⟨*subrules for stmt* 24d⟩≡ (20b) 24e▷

```
/* Sequence expression */
sequence:
   LPAREN RPAREN          { [] }
 | LPAREN expr_list RPAREN { $2 }
```

⟨*subrules for stmt* 24e⟩+≡ (20b) ◁24d 25a▷

```
expr_list:
   expr                     { $1 :: [] }
 | expr SEMICOLON expr_list { $1 :: $3 }
```

6.4.2 Assignments

⟨*rule expr* 24f⟩+≡ (20b) ◁24c 47i▷

```
 | lvalue ASSIGN expr       { mkAssignExp $1 $3 }
```

6.4.3 Conditionals

⟨*subrules for stmt* 25a⟩+≡ (20b) ◁24e 25b▷

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

6.4.4 Loops

⟨*subrules for stmt* 25b⟩+≡ (20b) ◁25a 48a▷

```
/* 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 }
```

6.5 Types

⟨*rule ty* 25c⟩≡ (20b)

```
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 }
```

Chapter 7

Checking

⟨*function Semantics.translate* 26a⟩≡ (68a)
```
let translate env ast =
  let (mainex, mainty) = trexp env ast in

  if mainty <> INT && mainty <> UNIT
  then E.type_err 0 "tiger program must return INT or UNIT";

  ⟨Semantics.translate() return translated functions 26e⟩
```

⟨*functions Semantics.trxxx* 26b⟩≡ (68c)
```
⟨function Semantics.trans.trexp 26c⟩
⟨function Semantics.trans.trdec 29d⟩
⟨function Semantics.trans.trvar 31c⟩
```

⟨*function Semantics.trans.trexp* 26c⟩≡ (26b)
```
let rec trexp env = function
  ⟨Semantics.trans.trexp() cases 26d⟩
```

⟨Semantics.trans.trexp() *cases* 26d⟩≡ (26c) 31a ▷
```
  | A.LetExp(decls, body, _) ->
      let env' = Env.new_scope env in
      let decs = decls |> List.map (fun d -> trdec env' d) in
      let bex,bty = trexp env' body in
      (Trans.sequence (decs @ [bex]), bty)
```

⟨Semantics.translate() *return translated functions* 26e⟩≡ (26a)
```
add_function (Env.frame env) (mainex, mainty);
List.rev !functions
```

⟨*global Semantics.functions* 26f⟩≡ (68b)
```
let functions                 = ref []
```

⟨*function* Semantics.add_function 26g⟩≡ (68b)
```
let add_function frm (ex,typ) =
  functions := (frm, Trans.func ex (is_ptr typ)) :: !functions
```

⟨Environment.t *other fields* 26h⟩≡ (27a) 34b ▷
```
frame       : Frame.frame;
```

⟨Environment.new_env() *other field initializations* 26i⟩≡ (29a)
```
frame       = F.new_frame (S.symbol "tiger_main") F.base_frame;
```

7.1 The environment

⟨*type Environment.t* 27a⟩≡ (67a 66a)
```
type t = {
    (* type definitions *)
    tenv        : vartype Symbol.table;
    (* value definitions *)
    venv        : vartype enventry Symbol.table;
    ⟨Environment.t other fields 26h⟩
  }
```

⟨*type Environment.enventry* 27b⟩≡ (67a 66a)
```
type 'ty enventry =
    VarEntry of (Frame.access * 'ty)
  | FunEntry of (Tree.label * string option * Frame.frame * 'ty list * 'ty)
```

⟨*function* Environment.new_scope 27c⟩≡ (67c)
```
let new_scope env = { env with
                      tenv = S.new_scope env.tenv;
                      venv = S.new_scope env.venv }
```

7.1.1 Lookups

⟨*signature function* Environment.lookup_type 27d⟩≡ (66d)
```
val lookup_type  : t -> Ast.typename -> Ast.pos -> vartype
```

⟨*signature function* Environment.lookup_value 27e⟩≡ (66d)
```
val lookup_value : t -> Ast.name -> Ast.pos -> vartype enventry
```

⟨*functions* Environment.lookup_xxx 27f⟩≡ (67e) 48e▷
```
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
```

7.1.2 Enters

⟨*signature function* Environment.enter_type 27g⟩≡ (66e)
```
val enter_type  : t -> Ast.typename -> vartype -> unit
```

⟨*functions* Environment.enter_xxx 27h⟩≡ (67f) 28a▷
```
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
```

⟨*signature function* Environment.enter_fun 27i⟩≡ (66e)
```
val enter_fun   :
  t -> Ast.name -> string option -> vartype list -> vartype  -> t
```

⟨*functions* Environment.enter_xxx 28a⟩+≡ (67f) ◁27h 48g▷

```
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
```

⟨*function* Environment.new_frame 28b⟩≡ (67d)

```
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 }
```

⟨*function* Symbol.new_symbol 28c⟩≡ (63b)

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

⟨*signature function* Environment.enter_param 28d⟩≡ (66e)

```
val enter_param : t -> Ast.name -> vartype -> Tree.is_ptr -> unit
```

⟨*signature function* Environment.enter_local 28e⟩≡ (66e)

```
val enter_local : t -> Ast.name -> vartype -> Tree.is_ptr -> Frame.access
```

⟨*function* Environment.enter_param 28f⟩≡ (67g)

```
let enter_param env sym typ ptr =
  let access = F.alloc_param env.frame sym ptr in
  enter env.venv sym (VarEntry(access, typ))
```

⟨*function* Environment.enter_local 28g⟩≡ (67h)

```
let enter_local env sym typ ptr =
  let access = F.alloc_local env.frame sym ptr in
  enter env.venv sym (VarEntry(access, typ));
  access
```

7.2 Builtins

⟨*constant* Main.base_tenv 28h⟩≡ (89a)

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

⟨*constant* Main.base_venv 28i⟩≡ (89a)

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

⟨*function* Environment.new_env 29a⟩≡ (67b)
```
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);

       ⟨Environment.new_env() other field initializations 26i⟩
       xenv        = Symbol.create [];
       break_label = None;
       exn_label   = None
     }
```

7.3 Typechecking helpers

⟨*functions* Semantics.check_type_xxx 29b⟩≡ (68a)
```
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
```

⟨*function* Semantics.check_type_eq 29c⟩≡ (68a)
```
let check_type_eq pos msg t1 t2 =
  let are_equivalent =
    match (t1,t2) with
    | (RECORD _,NIL)
    | (NIL,RECORD _)
    | (ARRAY ANY, ARRAY _)
    | (ARRAY _, ARRAY ANY) -> true
    | _                    -> t1 = t2
  in
  if not are_equivalent
  then E.type_err pos (Printf.sprintf msg (type_name t1) (type_name t2))
```

7.4 Definitions

⟨*function Semantics.trans.trdec* 29d⟩≡ (26b)
```
and trdec env = function
  ⟨Semantics.trans.trdec() cases 29e⟩
```

7.4.1 Variables

⟨Semantics.trans.trdec() *cases* 29e⟩≡ (29d) 30a▷
```
| A.VarDec(name, typ, init, pos) ->
    let e,t = trexp env init in
    (match typ with
      Some x -> check_type_eq pos
          "Variable of type %s cannot be initialized with type %s"
          (Env.lookup_type env x pos) t
    | None -> ()
    );
    let access = Env.enter_local env name t (is_ptr t) in
    Trans.assign (Trans.simple_var (Env.frame env) access) e
```

7.4.2 Types

⟨Semantics.trans.trdec() *cases* 30a⟩+≡ (29d) ◁29e 30b▷

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

7.4.3 Functions

⟨Semantics.trans.trdec() *cases* 30b⟩+≡ (29d) ◁30a 48h▷

```
| A.FunctionDec functions ->
    ⟨local function Semantics.trans.trdec.mk_param (for FunctionDec case) 30e⟩
    ⟨local function Semantics.trans.trdec.mk_func_env (for FunctionDec case) 30d⟩
    ⟨local function Semantics.trans.trdec.trans_func (for FunctionDec case) 30c⟩
    let envs = (List.map mk_func_env functions) in
    List.iter2 trans_func   envs functions;
    Trans.nil
```

⟨local function Semantics.trans.trdec.trans_func (for FunctionDec case) 30c⟩≡ (30b)

```
let trans_func fenv (_, _, _, body, _) =
  let b = trexp fenv body in
  add_function (Env.frame fenv) b
in
```

⟨local function Semantics.trans.trdec.mk_func_env (for FunctionDec case) 30d⟩≡ (30b)

```
let mk_func_env (name, params, typ, _, pos) =
  let ret_type =
    match typ with
    | Some x -> lookup_base_type env x pos
    | None   -> UNIT
  in
  let params_types = params|> List.map (fun(_,t,p)->lookup_base_type env t p) in
  let fenv  = Env.enter_fun env name None(*not C func*) params_types ret_type in
  params |> List.iter (fun p -> mk_param fenv p);
  fenv
in
```

⟨local function Semantics.trans.trdec.mk_param (for FunctionDec case) 30e⟩≡ (30b)

```
let mk_param fenv (name, typ, pos) =
  let t = lookup_base_type env typ pos in
  Env.enter_param fenv name t (is_ptr t)
in
```

7.5 Expressions

7.5.1 Literals

⟨Semantics.trans.trexp() *cases* 31a⟩+≡ (26c) ◁26d 31b▷
```
    | A.NilExp         -> (Trans.nil,           NIL)
    | A.IntExp i       -> (Trans.int_literal i, INT)
    | A.StringExp(s,_) -> (Trans.str_literal s, STRING)
```

7.5.2 Variable, array and field accesses

⟨Semantics.trans.trexp() *cases* 31b⟩+≡ (26c) ◁31a 32a▷
```
    | A.VarExp v       -> trvar env v
```

⟨*function Semantics.trans.trvar* 31c⟩≡ (26b)
```
  and trvar env = function
    ⟨Semantics.trans.trvar() cases 31d⟩
```

⟨Semantics.trans.trvar() *cases* 31d⟩≡ (31c) 31e▷
```
      A.SimpleVar(sym, pos) ->
        (match Env.lookup_value env sym pos with
          Env.VarEntry(access, vt) ->
            (Trans.simple_var (Env.frame env) access, base_type env vt)
        | Env.FunEntry _ ->
            E.type_err pos "function used as value"
        )
```

⟨Semantics.trans.trvar() *cases* 31e⟩+≡ (31c) ◁31d 31f▷
```
    | A.FieldVar(var, sym, pos) ->
        let (exp, fields) =
          match trvar env 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
        (Trans.field_var exp !offset (is_ptr typ), typ)
```

⟨Semantics.trans.trvar() *cases* 31f⟩+≡ (31c) ◁31e
```
    | A.SubscriptVar(var, exp, pos) ->
        let e,t = trexp env exp in
        check_type_int pos "subscript variable" t;
        (match trvar env var with
          (exp, ARRAY vt) ->
            let typ = (base_type env vt) in
            (Trans.subscript_var exp e (is_ptr typ) pos, typ)
        | _ ->
            E.type_err pos "attempt to dereference a non-array type"
        )
```

7.5.3 Constructions

⟨Semantics.trans.trexp() *cases* 32a⟩+≡ (26c) ◁31b 32c▷

```
    | A.RecordExp(var, fields, pos) ->
        ⟨function Semantics.trans.trexp.chk_field 32b⟩
        (match Env.lookup_type env var pos with
          RECORD dec_fields ->
            begin try
              let field_vals = (List.map2 chk_field fields dec_fields) in
              (Trans.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"
        )
```

⟨function Semantics.trans.trexp.chk_field 32b⟩≡ (32a)

```
  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 env e in
    check_type_eq p "field type (%s) does not match declaration (%s)"
                    (base_type env vt) ty;
    (ex, is_ptr ty)
  in
```

⟨Semantics.trans.trexp() *cases* 32c⟩+≡ (26c) ◁32a 32d▷

```
    | A.ArrayExp(name, size, init, pos) ->
        (match Env.lookup_type env name pos with
          ARRAY vt ->
            let size,sizety = trexp env size in
            let init,initty = trexp env init in
            let 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;
            (Trans.new_array size init (is_ptr typ), ARRAY vt)
        | _ ->
            E.type_err pos "Attempt to use a non-array type as an array"
        )
```

7.5.4 Operators

⟨Semantics.trans.trexp() *cases* 32d⟩+≡ (26c) ◁32c 33a▷

```
    | A.OpExp(left, oper, right, pos) ->
        let lexp,lty = trexp env left in
        let rexp,rty = trexp env 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;    Trans.arithmetic
          | A.EqOp | A.NeqOp
          | A.LtOp | A.LeOp | A.GtOp | A.GeOp ->
              (match lty with
                INT | NIL                                  -> Trans.compare_int
              | STRING                                     -> Trans.compare_str
              | ARRAY  _ when oper=A.EqOp || oper=A.NeqOp -> Trans.compare_int
              | RECORD _ when oper=A.EqOp || oper=A.NeqOp -> Trans.compare_int
```

```
            | _ ->
                E.type_err pos "Incomparable types"
            )
        in (trans_fn oper lexp rexp, INT)
```

7.5.5 Function calls

⟨Semantics.trans.trexp() *cases* 33a⟩+≡ (26c) ◁32d 33b▷

```
    | A.CallExp(sym, arglist, pos) ->
        let chk_arg = check_type_eq pos
                      "Argument type (%s) does not match declaration (%s)"
        in
        (match Env.lookup_value env sym pos with
          Env.FunEntry(lbl, cc, frm, dec_args, return_type) ->
            let args,tys = List.split (List.map (trexp env) arglist) in
            begin try
              List.iter2 chk_arg tys dec_args;
              let rtyp = base_type env return_type in
              (Trans.call (Env.frame env) lbl cc frm args
                      (Env.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")
        )
```

7.5.6 Local entity definitions

7.6 Statements

7.6.1 Sequence

⟨Semantics.trans.trexp() *cases* 33b⟩+≡ (26c) ◁33a 33c▷

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

7.6.2 Assignments

⟨Semantics.trans.trexp() *cases* 33c⟩+≡ (26c) ◁33b 34a▷

```
    | A.AssignExp(var, exp, pos) ->
        let exp,ety = trexp env exp in
        let var,vty = trvar env var in
        check_type_eq pos "Cannot assign to type %s from type %s" vty ety;
        (Trans.assign var exp, UNIT)
```

7.6.3 Conditionals

⟨Semantics.trans.trexp() *cases* 34a⟩+≡ (26c) ◁33c 34c▷

```
    | A.IfExp(if', then', else', pos) ->
        let iex,ity = trexp env if' in
        let tex,tty = trexp env then' in
        let eex,ety =
           match else' with
           | None    -> (Trans.nil, UNIT)
           | Some ex -> trexp env 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
        (* claude: a UNIT-typed branch (e.g. a print() call) has no
         * meaningful value - force both branches to the canonical nil
         * instead of merging whatever garbage was left in the result
         * register. See Translate.discard. *)
        let tex, eex =
          if typ = UNIT then (Trans.discard tex, Trans.discard eex)
          else (tex, eex)
        in
        (Trans.ifexp iex tex eex (is_ptr typ), typ)
```

7.6.4 Loops

⟨Environment.t *other fields* 34b⟩+≡ (27a) ◁26h 48b▷

```
  break_label : Tree.label option;
```

⟨Semantics.trans.trexp() *cases* 34c⟩+≡ (26c) ◁34a 34d▷

```
    | A.WhileExp(test, body, pos) ->
        let tex,tty = trexp env test in
        let body_env = Env.new_break_label env in
        let bex,bty = trexp body_env body in
        check_type_int pos "while condition" tty;
        check_type_eq  pos "body of while has type %s, must be %s" bty UNIT;
        (Trans.loop tex bex (Env.break_label body_env), UNIT)
```

⟨Semantics.trans.trexp() *cases* 34d⟩+≡ (26c) ◁34c 34e▷

```
    | A.BreakExp pos ->
        begin
          try (Trans.break (Env.break_label env), UNIT)
          with Not_found -> raise(E.Error(E.Illegal_break, pos))
        end
```

⟨Semantics.trans.trexp() *cases* 34e⟩+≡ (26c) ◁34d 48i▷

```
    | A.ForExp(sym, lo, hi, body, pos) ->
        let _,loty = trexp env lo in
        let _,hity = trexp env hi in
        check_type_int pos "for lower bound" loty;
        check_type_int pos "for upper bound" hity;
      ⟨Semantics.trans.trexp() ForExp case, unsugaring for in while 40e⟩
```

Chapter 8

Intermediate Representation Generation

8.1 Function definitions

⟨*signature functions Translate.xxx* 35a⟩≡ (71c) 35h ▷
```
val func          : Tree.exp -> Tree.is_ptr -> Tree.exp
```

⟨*function Translate.func* 35b⟩≡ (71d)
```
let func body ptr =
  let tmp = temp ptr in
  eseq nil [body => tmp; T.RET tmp]
```

⟨*function Translate.temp* 35c⟩≡ (71d)
```
let temp ptr =
  T.TEMP (T.new_temp(), ptr)
```

⟨*function* `Tree.new_temp` 35d⟩≡ (69f)
```
let new_temp () =
  S.new_symbol "temp"
```

⟨*function Translate.eseq* 35e⟩≡ (71d)
```
let eseq exp stmts =
  T.ESEQ (seq stmts, exp)
```

⟨*function Translate.seq* 35f⟩≡ (71d)
```
let rec seq = function
    []        -> E.internal "nil passed to seq"
  | x :: []   -> x
  | x :: rest -> T.SEQ(x, seq rest)
```

⟨*function Translate.arrow* 35g⟩≡ (72b)
```
let ( =>) e v = T.MOVE (e, v)
```

8.2 Expressions

8.2.1 Literals

⟨*signature functions Translate.xxx* 35h⟩+≡ (71c) ◁35a 36c ▷
```
val nil           : Tree.exp
val int_literal   : int -> Tree.exp
val str_literal   : string -> Tree.exp
```

⟨*functions Translate.xxx literals* 35i⟩≡ (71d)
```
let nil           = T.CONST 0
let int_literal i = T.CONST i
let str_literal s = T.NAME (F.alloc_string s)
```

⟨*global Frame.strings* 36a⟩≡ (74c)
```
let strings = H.create 20
```

⟨*function* Frame.alloc_string 36b⟩≡ (74c)
```
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)
```

8.2.2 Field, array and offsetization

⟨*signature functions Translate.xxx* 36c⟩+≡ (71c) ◁35h 36h▷
```
val field_var     : Tree.exp -> int -> Tree.is_ptr -> Tree.exp
```

⟨*function* Translate.field_var 36d⟩≡ (71d)
```
let field_var ex i ptr = T.MEM(ex <+> T.CONST(i * ws), ptr)
```

⟨*function* Translate.angles_xxx 36e⟩≡ (72b 71d)
```
let (<+>) = simplify T.PLUS  ( + )
let (<->) = simplify T.MINUS ( - )
let (<*>) = simplify T.MUL   ( * )
```

⟨*function Translate.simplify* 36f⟩≡ (72b 71d)
```
let simplify tig_op op =
 fun x y ->
  match (x,y) with
    (T.CONST x, T.CONST y) -> T.CONST (op x y)
  | _                      -> T.BINOP (tig_op, x, y)
```

⟨*constant Translate.ws* 36g⟩≡ (71d)
```
(* claude: target is always 32-bit x86 (qc-- only emits i386), regardless of
 * the host OCaml runtime's word size, so this must be a fixed constant and
 * not derived from Sys.word_size. *)
let  ws    = 4
```

⟨*signature functions Translate.xxx* 36h⟩+≡ (71c) ◁36c 36j▷
```
val subscript_var : Tree.exp -> Tree.exp -> Tree.is_ptr -> int -> Tree.exp
```

⟨*function* Translate.subscript_var 36i⟩≡ (71d)
```
let subscript_var e1 e2 ptr pos =
  let check = ext_c_call "bounds_check"
                       [e1;e2;T.CONST(fst (Error.line_number pos))] in
  let offset = (e2 <+> T.CONST 1) <*> T.CONST ws in
  eseq (T.MEM(e1 <+> offset, ptr)) [T.EXP check]
```

8.2.3 Variable and frameization

⟨*signature functions Translate.xxx* 36j⟩+≡ (71c) ◁36h 37h▷
```
val simple_var    : Frame.frame -> Frame.access -> Tree.exp
```

⟨*function* Translate.simple_var 36k⟩≡ (71d)
```
let simple_var frm = function
  | F.Stack(var_frm, offset, ptr) ->
      T.MEM(getfp frm var_frm <+> T.CONST(offset * ws), ptr)
```

⟨*type Frame.access* 36l⟩≡ (73)
```
type access =
    Stack of frame * int * Tree.is_ptr
```

⟨*function* `Frame.alloc_local` 37a⟩≡ (74a)
```
let alloc_local frm name ptr =
  frm.vars <- (name,ptr) :: frm.vars;
  stack_alloc frm ptr
```

⟨*function* `Frame.alloc_param` 37b⟩≡ (74a)
```
let alloc_param frm name ptr =
  frm.params <- frm.params @ [(name,ptr)];
  stack_alloc frm ptr
```

⟨*function* `Frame.stack_alloc` 37c⟩≡ (74a)
```
let stack_alloc frm ptr =
  let v = Stack(frm, frm.size, ptr) in
  frm.size <- frm.size + 1;
  v
```

⟨`Frame.frame` *other fields* 37d⟩+≡ (13g) ◁13h 37e▷
```
mutable size   : int;
```

⟨`Frame.frame` *other fields* 37e⟩+≡ (13g) ◁37d
```
level          : int;
```

⟨*function Translate.getfp* 37f⟩≡ (72a 71d)
```
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
```

⟨*function Frame.fp* 37g⟩≡ (73h)
```
let fp    _frm =
  T.NAME (S.symbol "fp")
```

8.2.4 Constructions and allocations

⟨*signature functions Translate.xxx* 37h⟩+≡ (71c) ◁36j 38b▷
```
val new_record    : (Tree.exp * Tree.is_ptr) list -> Tree.exp
val new_array     : Tree.exp -> Tree.exp -> Tree.is_ptr -> Tree.exp
```

⟨*function* `Translate.new_record` 37i⟩≡ (71d)
```
let new_record init =
  let tmp  = temp true in
  let 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)
```

⟨*function* Translate.new_array 38a⟩≡ (71d)
```
let new_array sizeEx initEx ptr =
  let ary  = temp true in
  let i    = temp false in
  let lbeg = T.new_label "init_start" in
  let 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 ]
```

8.2.5 Operators

⟨*signature functions Translate.xxx* 38b⟩+≡ (71c) ◁37h 38f▷
```
val arithmetic    : Ast.oper -> Tree.exp -> Tree.exp -> Tree.exp
val compare_int   : Ast.oper -> Tree.exp -> Tree.exp -> Tree.exp
val compare_str   : Ast.oper -> Tree.exp -> Tree.exp -> Tree.exp
```

⟨*function Translate.arithmetic* 38c⟩≡ (71d)
```
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)
```

⟨*function* Translate.compare_int 38d⟩≡ (71d)
```
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)
```

⟨*function* Translate.compare_str 38e⟩≡ (71d)
```
let compare_str op ex1 ex2 =
  let result = ext_c_call "compare_str" [ex1;ex2] in
  compare_int op result (T.CONST 0)
```

8.2.6 Function calls

⟨*signature functions Translate.xxx* 38f⟩+≡ (71c) ◁38b 39d▷
```
val call          :
  Frame.frame -> Tree.label -> string option -> Frame.frame ->
  Tree.exp list -> Tree.label option -> Tree.is_ptr ->
  Tree.exp
```

⟨*function Translate.call* 39a⟩≡ (71d)
```
let call caller_frm   lbl cc callee_frm   args k ptr =
  let args =
    if F.level callee_frm == 0
    then args
    else
      let pfp =
        if (F.level callee_frm) > (F.level caller_frm)
        then F.fp callee_frm
        else T.MEM(getfp caller_frm callee_frm, true)
      in
      pfp  :: args
  in
  match cc with
    None        -> T.CALL((T.NAME lbl), args, cc, k, ptr)
  ⟨Translate.call() match cc other cases 39b⟩
```

8.2.7 Foreign calls

⟨Translate.call() *match cc other cases* 39b⟩≡ (39a)
```
| Some "gc"   -> T.CALL((T.NAME lbl), args, cc, k, ptr)
| Some _ ->
    let tmp1      = temp ptr in
    let 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) ]
```

⟨*functions* Translate.ext_xxx_call 39c⟩≡ (71d)
```
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
```

8.3 Statements and labelization

8.3.1 Sequence

⟨*signature functions Translate.xxx* 39d⟩+≡ (71c) ◁38f 39f▷
```
val sequence      : Tree.exp list -> Tree.exp
```

⟨*function Translate.sequence* 39e⟩≡ (71d)
```
let rec sequence = function
   []       -> nil
  | e :: [] -> e
  | e :: es -> T.ESEQ((T.EXP e), (sequence es))
```

8.3.2 Assignments

⟨*signature functions Translate.xxx* 39f⟩+≡ (71c) ◁39d 40b▷
```
val assign        : Tree.exp -> Tree.exp -> Tree.exp
val discard       : Tree.exp -> Tree.exp
```

⟨*function Translate.assign* 40a⟩≡ (71d)
```
let assign v e =
  eseq nil [e => v]
(* claude: a UNIT-typed exp (e.g. a call to a void C runtime function like
 * print) carries no meaningful value - the callee's calling convention makes
 * no promise about what ends up in its "result" register. Reading that
 * value anyway (as ifexp's branch-merge used to unconditionally do) leaks
 * whatever garbage was left over, which became visible as e.g. a Tiger
 * program's exit code being the leftover value of its last print() call
 * instead of 0. Evaluate for effect only, and always yield the same nil. *)
let discard e = eseq nil [T.EXP e]
```

8.3.3 Conditionals

⟨*signature functions Translate.xxx* 40b⟩+≡ (71c) ◁39f 40f▷
```
val ifexp         : Tree.exp -> Tree.exp -> Tree.exp -> Tree.is_ptr -> Tree.exp
```

⟨*function Translate.ifexp* 40c⟩≡ (71d)
```
let ifexp test thn els ptr =
  let tmp  = temp ptr in
  let tru  = T.new_label "ifTrue" in
  let fls  = T.new_label "ifFalse" in
  let 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']
```

⟨*function Translate.goto* 40d⟩≡ (72b)
```
let goto lbl  =
  T.JUMP lbl
```

8.3.4 Loops

⟨Semantics.trans.trexp() *ForExp case, unsugaring for in while* 40e⟩≡ (34e)
```
  let v            = A.SimpleVar(sym, pos) in
  let ve           = A.VarExp v in
  trexp env
    (A.LetExp(
         [(A.VarDec(sym,(Some(S.symbol "int")), lo, pos))],
         (A.WhileExp
            (A.OpExp(ve, A.LeOp, hi, pos),
             (A.SeqExp([body;
                        (A.AssignExp(v,
                           A.OpExp(ve, A.PlusOp, (A.IntExp 1), pos), pos))
                        ], pos)),
             pos)),
       pos))
```

⟨*signature functions Translate.xxx* 40f⟩+≡ (71c) ◁40b 41a▷
```
val loop          : Tree.exp -> Tree.exp -> Tree.label -> Tree.exp
```

⟨*function Translate.loop* 40g⟩≡ (71d)
```
let loop test body lend =
  let lbeg = T.new_label "loop_start" in
  let 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 ]
```

⟨*signature functions Translate.xxx* 41a⟩+≡ (71c) ◁40f 48l▷
```
val break         : Tree.label -> Tree.exp
```

⟨*function Translate.break* 41b⟩≡ (71d)
```
let break lbl = eseq nil [goto lbl]
```

8.4 Statements, expressions and linearization

⟨*function Canonical.linearize* 41c⟩≡ (72d)
```
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 l ->
        T.JUMP l
    | 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, [])
```

⟨*function Canonical.commute* 42a⟩≡ (72d)

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

⟨*function Canonical.percent* 42b⟩≡ (72d)

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

⟨*constant Canonical.nop* 42c⟩≡ (72d)

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

⟨`Main.emit_function()` *adjust frm with temporaries in ltree* 42d⟩≡ (15f)

```
  ltree |> Tree.find_temps |> List.iter (fun (x,p) ->
    Frame.alloc_temp frm x p
  );
```

⟨*signature function* `Tree.find_temps` 43⟩≡ (69d)

```
val find_temps : stm list -> (temp * is_ptr) list
```

Chapter 9

C-- Generation

9.1 File header

⟨`Main.compile()` *generate headers* 44a⟩≡ (15a)

```
Codegen.output_file_header imports;
```

9.2 Data

⟨`Main.compile()` *generate data before functions* 44b⟩≡ (15a)

```
Frame.output_strings();
```

9.3 Function header

9.4 Function body

9.5 Function footer

Chapter 10

Runtime

10.1 Standard library

10.2 Startup code

10.3 Memory allocation

⟨*function Translate.alloc* 45a⟩≡ (71d)

```
let alloc size =
  let size = (size <+> T.CONST 1) <*> T.CONST ws in
  let test = T.RELOP(T.GT, alloc_ptr <+> size, space_end) in
  let tmp  = temp true in
  let tru  = T.new_label "alc_gc" in
  let 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
           ]
```

⟨*constant* `Translate.space_end` 45b⟩≡ (72b)

```
let space_end = T.MEM  (T.NAME (S.symbol "space_end"), true)
```

⟨*constant* `Translate.alloc_ptr` 45c⟩≡ (72b)

```
let alloc_ptr = T.NAME (S.symbol "alloc_ptr")
```

10.4 Garbage collection

Chapter 11

Standard Library

Chapter 12

Advanced Features

12.1 Typedefs

⟨Environment.vartype *cases* 47a⟩+≡ (10d) ◁10f
```
  | NAME   of Ast.typename
```

⟨*function* Semantics.base_type 47b⟩≡ (68a)
```
  let rec base_type env = function
      NAME s ->
        (try base_type env (Env.lookup_type env s 0)
         with Not_found -> E.internal "NAME symbol not found"
        )
    | x -> x
```

⟨*function* Semantics.lookup_base_type 47c⟩≡ (68a)
```
  let lookup_base_type env sym pos =
    base_type env (Env.lookup_type env sym pos)
```

12.2 Exceptions

⟨*token declarations* 47d⟩+≡ (20a) ◁16c 49f▷
```
  %token EXCEPTION  TRY RAISE HANDLE
```

⟨Lexer.keyword_table *entries* 47e⟩≡ (18a) 49g▷
```
  "exception", P.EXCEPTION;

  "try",       P.TRY;
  "handle",    P.HANDLE;
  "raise",     P.RAISE;
```

⟨Ast.dec *cases* 47f⟩≡ (9d)
```
  | ExceptionDec of name * pos
```

⟨*rule* exn_dec 47g⟩≡ (21e)
```
  exn_dec:
     EXCEPTION id { mkException $2 }
```

⟨Ast.exp *cases* 47h⟩≡ (9f) 49e▷
```
  | TryExp    of exp * (name * exp * pos) list * pos
  | RaiseExp  of name * pos
```

⟨*rule expr* 47i⟩+≡ (20b) ◁24f 49h▷
```
   | TRY expr handlers         { mkTryExp $2 (List.rev $3) }
   | RAISE id                  { mkRaise $2 }
```

⟨*subrules for stmt* 48a⟩+≡ (20b) ◁25b
```
/* exception handlers */
handler:
   HANDLE id expr END         { mkHandler $2 $3 }

handlers:
   handler          { $1 :: [] }
 | handler handlers { $1 :: $2 }
```

⟨Environment.t *other fields* 48b⟩+≡ (27a) ◁34b 48c▷
```
exn_label   : Tree.label option;
```

⟨Environment.t *other fields* 48c⟩+≡ (27a) ◁48b
```
xenv        : int Symbol.table;
```

⟨*signature function* Environment.lookup_exn 48d⟩≡ (66d)
```
val lookup_exn   : t -> Ast.name -> Ast.pos -> int
```

⟨*functions* Environment.lookup_xxx 48e⟩+≡ (67e) ◁27f
```
let lookup_exn   env = lookup env.xenv
```

⟨*signature function* Environment.enter_exn 48f⟩≡ (66e)
```
val enter_exn   : t -> Ast.name -> unit
```

⟨*functions* Environment.enter_xxx 48g⟩+≡ (67f) ◁28a
```
let enter_exn env sym = enter env.xenv sym (S.uid sym)
```

⟨Semantics.trans.trdec() *cases* 48h⟩+≡ (29d) ◁30b
```
    | A.ExceptionDec(sym,_) -> Env.enter_exn env sym; Trans.nil
```

⟨Semantics.trans.trexp() *cases* 48i⟩+≡ (26c) ◁34e 48j▷
```
    | A.TryExp(expr, handlers, pos) ->
        let new_env         = Env.new_exn_label env in
        let tryex, tryty    = trexp new_env expr in
        let handler (s,h,p) =
          let ex,ty = trexp env h in
          check_type_unit p "handler" ty;
          (S.uid s, ex)
        in
        check_type_unit pos "try" tryty;
        begin match Env.exn_label new_env with
          None     -> E.internal "no exception label for try block"
        | Some lbl ->
            (Trans.try_block tryex lbl (List.map handler handlers), tryty)
        end
```

⟨Semantics.trans.trexp() *cases* 48j⟩+≡ (26c) ◁48i 50a▷
```
    | A.RaiseExp(sym, pos) ->
        let exn_id = Env.lookup_exn env sym pos in
        (Trans.raise_exn exn_id, UNIT)
```

⟨Tree.stm *cases* 48k⟩+≡ (12e) ◁13c
```
    | CONT   of label * label list
    | TRY    of label
    | TRYEND of label
```

⟨*signature functions Translate.xxx* 48l⟩+≡ (71c) ◁41a 50b▷
```
val try_block     : Tree.exp -> Tree.label -> (int *Tree.exp) list -> Tree.exp
val raise_exn     : int -> Tree.exp
```

⟨*function* `Translate.try_block` 49a⟩≡ (71d)
```
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
  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
  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.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 ]
```

⟨*function* `Translate.raise_exn` 49b⟩≡ (71d)
```
let raise_exn uid =
  let fn = if !Option.unwind then "unwind" else "raise" in
  ext_cmm_call fn [T.CONST uid]
```

⟨*command line flags* 49c⟩+≡ (62e) ◁14h 56a▷
```
let unwind = ref false
```

⟨*command line options* 49d⟩≡ (14g) 56b▷
```
"-unwind",   Arg.Set unwind, "\tuse unwind continuations for exceptions";
```

12.3 Spawn

⟨`Ast.exp` *cases* 49e⟩+≡ (9f) ◁47h
```
| SpawnExp  of name * pos
```

⟨*token declarations* 49f⟩+≡ (20a) ◁47d
```
%token SPAWN
```

⟨`Lexer.keyword_table` *entries* 49g⟩+≡ (18a) ◁47e
```
"spawn",     P.SPAWN;
```

⟨*rule expr* 49h⟩+≡ (20b) ◁47i
```
  | SPAWN id                  { mkSpawn $2 }
```

⟨Semantics.trans.trexp() *cases* 50a⟩+≡ (26c) ◁48j

```
    | A.SpawnExp(sym, pos) ->
        (match Env.lookup_value env sym pos with
          Env.FunEntry(lbl, _cc, _frm, dec_args, _return_type) ->
            if dec_args <> []
            then E.type_err pos "spawn function must take zero arguments."
            else (Trans.spawn lbl, INT)
        | _ ->
            E.type_err pos (S.name sym ^ " is not a function")
        )
```

⟨*signature functions Translate.xxx* 50b⟩+≡ (71c) ◁48l

```
  val spawn         : Tree.label -> Tree.exp
```

⟨*function Translate.spawn* 50c⟩≡ (71d)

```
  let spawn lbl = ext_cmm_call "spawn" [T.NAME lbl]
```

Chapter 13

Advanced Topics

Chapter 14

Conclusion

Appendix A

Error Managment

A.1 The errors

⟨*type Error.error* 53a⟩≡ (61)
```
type error =
    Internal_error of string

  (* lexical errors *)
  | Illegal_character of char
  | Illegal_escape of string
  | Unterminated_comment
  | Unterminated_string

  (* syntaxic errors *)
  | Syntax_error

  (* semantic errors *)
  | Type_error of string
  | Undefined_symbol of string
  | Duplicate_symbol of string
  | Illegal_break
```

⟨*type Error.ex* 53b⟩≡ (61)
```
type ex = error * int
```

⟨*exception Error.Error* 53c⟩≡ (61)
```
exception Error of ex
```

⟨*parser helper functions* 53d⟩+≡ (20a) ◁??
```
let parse_error _s =
  let pos = getpos() in
  raise (E.Error(E.Syntax_error, pos))
```

⟨*function* Error.type_err 53e⟩≡ (61b)
```
let type_err  pos msg =
  raise(Error(Type_error msg, pos))
```

⟨*function Error.undefined* 53f⟩≡ (61b)
```
let undefined pos msg =
  raise(Error(Undefined_symbol msg, pos))
```

⟨*function Error.internal* 53g⟩≡ (61b)
```
let internal      msg =
  raise(Error(Internal_error msg, 0))
```

⟨*function* Error.handle_exception 54a⟩≡ (61b)
```
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
```

⟨*function* Semantics.type_name 54b⟩≡ (68a)
```
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"
```

A.2 Line position

⟨*function* Error.err_msg 54c⟩≡ (61b)
```
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.file line col msg
  else
    Printf.fprintf stderr "%s: %s\n" prefix msg
```

⟨*function Error.warning* 54d⟩≡ (61b)
```
let warning = err_msg "Warning"
```

⟨*function* Error.line_number 54e⟩≡ (61b)
```
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
```

⟨*type Error.sm* 54f⟩≡ (61b)
```
type sm = { mutable sm: (int * int) list }
```

⟨*global* `Error.source_map` 55a⟩≡ (61b)

```
let source_map = { sm = [(0,0)] }
```

⟨*function* `Error.add_source_mapping` 55b⟩≡ (61b)

```
let add_source_mapping pos line =
  source_map.sm <- source_map.sm @ [(pos,line)]
```

Appendix B

Debugging

B.1 Dumpers

B.1.1 `tiger -dump_ast`

⟨*command line flags* 56a⟩+≡ (62e) ◁49c 58b▷
```
let dump_ast    = ref false
```

⟨*command line options* 56b⟩+≡ (14g) ◁49d 59a▷
```
"-dump_ast",      Arg.Set dump_ast,    "\tprint Abstract Syntax Tree";
```

⟨`Main.compile()` *if dump AST option* 56c⟩≡ (15a)
```
if !Option.dump_ast
then Ast.print_tree ast;
```

⟨*function* `Ast.print_tree` 56d⟩≡ (65b)
```
let print_tree expression =
  ⟨declaration printer 56f⟩
  ⟨type printer 57a⟩
  ⟨variable printer 57b⟩
  ⟨expression printer 57c⟩
in exp 0 expression
```

⟨*function iprintf* 56e⟩≡ (65a)
```
let iprintf d fmt =
  let rec indent = function
      0 -> ()
    | i -> (print_string "  "; indent(i-1))
  in (indent d; Printf.printf fmt)
```

⟨*declaration printer* 56f⟩≡ (56d)
```
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* 57a⟩≡ (56d)

```
  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* 57b⟩≡ (56d)

```
  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* 57c⟩≡ (56d)

```
  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)
```

⟨*function Ast.opname* 58a⟩≡ (65a)

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

B.1.2 tiger -dump_ext

⟨*command line flags* 58b⟩+≡ (62e) ◁56a 59e▷

```
let dump_ext    = ref false
```

⟨*command line options* 59a⟩+≡ (14g) ◁56b 59f▷
```
  "-dump_ext",      Arg.Set dump_ext,    "\tprint Expression Trees";
```

⟨Main.emit_function() *if dump expression tree* 59b⟩≡ (15f)
```
  if !Option.dump_ext
  then Tree.print_exp ex;
```

⟨*function* Tree.print_exp 59c⟩≡ (70e)
```
  let print_exp e = print_stm (EXP e)
```

⟨*function* Tree.print_stm 59d⟩≡ (70d)
```
  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 l       -> iprintf d "JUMP:%s\n"   (S.name l)
      | 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
```

B.1.3 tiger -dump_lext

⟨*command line flags* 59e⟩+≡ (62e) ◁58b
```
  let dump_lext   = ref false
```

⟨*command line options* 59f⟩+≡ (14g) ◁59a
```
  "-dump_lext",     Arg.Set dump_lext,   "\tprint Linearized Expression Trees";
```

⟨Main.emit_function() *if dump linearized tree* 59g⟩≡ (15f)
```
  if !Option.dump_lext
  then List.iter Tree.print_stm ltree;
```

Appendix C

Extra Code

Appendix D

parsing

D.1 parsing/error.nw

⟨parsing/error.mli 61a⟩≡
  ⟨*type Error.error* 53a⟩
  ⟨*type Error.ex* 53b⟩
  ⟨*exception Error.Error* 53c⟩

  ⟨*error.mli* 61c⟩

⟨parsing/error.ml 61b⟩≡
  ⟨*type Error.error* 53a⟩
  ⟨*type Error.ex* 53b⟩
  ⟨*exception Error.Error* 53c⟩

  ⟨*type Error.sm* 54f⟩
  ⟨*global* `Error.source_map` 55a⟩
  ⟨*function* `Error.add_source_mapping` 55b⟩

  ⟨*function* `Error.line_number` 54e⟩

  ⟨*function* `Error.err_msg` 54c⟩

  ⟨*function Error.warning* 54d⟩

  ⟨*function* `Error.handle_exception` 54a⟩

  ⟨*function* `Error.type_err` 53e⟩
  ⟨*function Error.undefined* 53f⟩
  ⟨*function Error.internal* 53g⟩

D.2 Error Handling

⟨error.mli 61c⟩≡ (61a) 61d▷
```
val handle_exception : ex -> unit
```

⟨error.mli 61d⟩+≡ (61a) ◁61c 62a▷
```
val warning   : int -> string -> unit

val type_err  : int -> string -> 'a
val undefined : int -> string -> 'a
val internal  :        string -> 'a
```

⟨error.mli 62a⟩+≡ (61a) ◁61d
```
val add_source_mapping : int -> int -> unit
val line_number : int -> int * int
```

D.2.1 Error Implementation

D.3 parsing/option.nw

⟨parsing/option.mli 62b⟩≡
⟨*option.mli* 62d⟩

⟨parsing/option.ml 62c⟩≡
⟨*option.ml* 62e⟩

D.4 Command Line Options

D.4.1 Options

⟨option.mli 62d⟩≡ (62b)
⟨*signature function* `Option.parse_cmdline` 14e⟩

```
val dump_ast     : bool ref
val dump_ext     : bool ref
val dump_lext    : bool ref

val unwind    : bool ref
val file      : string ref
```
⟨*signature global Option.inch* 14c⟩

D.4.2 Option Implementation

⟨option.ml 62e⟩≡ (62c) 62f▷
⟨*command line flags* 14d⟩

⟨option.ml 62f⟩+≡ (62c) ◁62e 62g▷
⟨*function* `Option.set_input` 14i⟩

⟨option.ml 62g⟩+≡ (62c) ◁62f
⟨*function Option.usage* 8a⟩
⟨*constant Option.options* 14g⟩
⟨*function* `Option.parse_cmdline` 14f⟩

D.5 parsing/symbol.nw

⟨parsing/symbol.mli 62h⟩≡
⟨*symbol.mli* 63a⟩

⟨parsing/symbol.ml 62i⟩≡
⟨*symbol.ml* 63b⟩

D.6 Symbols

D.6.1 Basic Symbols

⟨symbol.mli 63a⟩≡ (62h) 63c▷
```
type symbol

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

⟨symbol.ml 63b⟩≡ (62i) 63e▷
⟨*type Symbol.symbol* 11a⟩

⟨*global Symbol.nextsym* 11b⟩

⟨*global Symbol.hashtable* 11c⟩

⟨*function Symbol.name* 11d⟩
⟨*function Symbol.uid* 11e⟩

⟨*function Symbol.symbol* 11f⟩

⟨*function* `Symbol.new_symbol` 28c⟩

D.6.2 Symbol Tables

⟨symbol.mli 63c⟩+≡ (62h) ◁63a 63d▷
```
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
```

⟨symbol.mli 63d⟩+≡ (62h) ◁63c
```
val iter : (int -> symbol -> 'a -> unit) -> 'a table -> unit
val fold : (int -> symbol -> 'a -> 'b -> 'b) -> 'a table -> 'b -> 'b
```

⟨symbol.ml 63e⟩+≡ (62i) ◁63b 63f▷
⟨*type Symbol.table* 11g⟩

⟨*function Symbol.enter* 11h⟩

⟨symbol.ml 63f⟩+≡ (62i) ◁63e 63g▷
⟨*function Symbol.look* 12b⟩

⟨symbol.ml 63g⟩+≡ (62i) ◁63f 63h▷
⟨*function Symbol.mem* 11i⟩

⟨symbol.ml 63h⟩+≡ (62i) ◁63g 64a▷
⟨*constructor Symbol.create* 11j⟩

⟨*constructor* `Symbol.new_scope` 12c⟩

⟨symbol.ml 64a⟩+≡ (62i) ◁63h

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

D.7 parsing/ast.nw

⟨parsing/ast.mli 64b⟩≡

⟨*ast.mli* 64d⟩

⟨parsing/ast.ml 64c⟩≡

⟨*ast.ml* 64e⟩

D.8 Abstract Syntax

⟨ast.mli 64d⟩≡ (64b)

⟨*type Ast.pos* 9a⟩
⟨*type Ast.name* 9b⟩
⟨*type Ast.typename* 9c⟩

⟨*type Ast.dec* 9d⟩
⟨*type Ast.ty* 10c⟩
⟨*type Ast.field* 9e⟩
⟨*type Ast.var* 10a⟩
⟨*type Ast.exp* 9f⟩
⟨*type Ast.oper* 23b⟩

```
val print_tree : exp -> unit
```

⟨ast.ml 64e⟩≡ (64c)

⟨*type Ast.pos* 9a⟩
⟨*type Ast.name* 9b⟩
⟨*type Ast.typename* 9c⟩

⟨*type Ast.dec* 9d⟩
⟨*type Ast.ty* 10c⟩
⟨*type Ast.field* 9e⟩
⟨*type Ast.var* 10a⟩
⟨*type Ast.exp* 9f⟩
⟨*type Ast.oper* 23b⟩

⟨*tree printer* 65a⟩

D.8.1 Ast Types

D.8.2 Abstract Syntax Tree Printer

⟨*tree printer* 65a⟩≡ (64e) 65b▷
```
module S = Symbol
```

⟨*function iprintf* 56e⟩

⟨*function Ast.opname* 58a⟩

⟨*tree printer* 65b⟩+≡ (64e) ◁65a
⟨*function* `Ast.print_tree` 56d⟩

D.9 `parsing/parser.nw`

D.10 Scanner

⟨`lexer.mli` 65c⟩≡
⟨*signature function Lexer.token* 15b⟩

D.11 Parser

Appendix E

frontend

E.1 frontend/environment.nw

⟨frontend/environment.mli 66a⟩≡
  ⟨*type Environment.vartype* 10d⟩
  ⟨*type Environment.enventry* 27b⟩
  ⟨*type Environment.t* 27a⟩

  ⟨*signature function* Environment.new_env 15d⟩

  ⟨*environment.mli* 66c⟩

⟨frontend/environment.ml 66b⟩≡
  ⟨*environment.ml* 67a⟩

E.2 Environments

E.2.1 Interface to the Environment

⟨environment.mli 66c⟩≡ (66a) 66d▷
```
val new_scope : t -> t
val frame     : t -> Frame.frame
val new_frame : t -> Symbol.symbol -> t
```

⟨environment.mli 66d⟩+≡ (66a) ◁66c 66e▷
  ⟨*signature function* Environment.lookup_type 27d⟩
  ⟨*signature function* Environment.lookup_value 27e⟩
  ⟨*signature function* Environment.lookup_exn 48d⟩

⟨environment.mli 66e⟩+≡ (66a) ◁66d 66f▷
  ⟨*signature function* Environment.enter_type 27g⟩
  ⟨*signature function* Environment.enter_fun 27i⟩
  ⟨*signature function* Environment.enter_param 28d⟩
  ⟨*signature function* Environment.enter_local 28e⟩
  ⟨*signature function* Environment.enter_exn 48f⟩

⟨environment.mli 66f⟩+≡ (66a) ◁66e 66g▷
```
val break_label     : t -> Tree.label
val new_break_label : t -> t
```

⟨environment.mli 66g⟩+≡ (66a) ◁66f
```
val exn_label     : t -> Tree.label option
val new_exn_label : t -> t
```

E.2.2 Implementation of Environments

⟨environment.ml 67a⟩≡ (66b) 67b▷
```
module E = Error
module S = Symbol
module F = Frame
module T = Tree
```
⟨*type Environment.vartype* 10d⟩
⟨*type Environment.enventry* 27b⟩
⟨*type Environment.t* 27a⟩

⟨environment.ml 67b⟩+≡ (66b) ◁67a 67c▷
⟨*function* Environment.new_env 29a⟩

⟨environment.ml 67c⟩+≡ (66b) ◁67b 67d▷
⟨*function* Environment.new_scope 27c⟩

⟨environment.ml 67d⟩+≡ (66b) ◁67c 67e▷
```
let frame env = env.frame
```
⟨*function* Environment.new_frame 28b⟩

⟨environment.ml 67e⟩+≡ (66b) ◁67d 67f▷
⟨*functions* Environment.lookup_xxx 27f⟩

⟨environment.ml 67f⟩+≡ (66b) ◁67e 67g▷
⟨*functions* Environment.enter_xxx 27h⟩

⟨environment.ml 67g⟩+≡ (66b) ◁67f 67h▷
⟨*function* Environment.enter_param 28f⟩

⟨environment.ml 67h⟩+≡ (66b) ◁67g 67i▷
⟨*function* Environment.enter_local 28g⟩

⟨environment.ml 67i⟩+≡ (66b) ◁67h 67j▷
```
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") }
```

⟨environment.ml 67j⟩+≡ (66b) ◁67i
```
let exn_label     env = env.exn_label
let new_exn_label env =
  { env with exn_label = Some(T.new_label "exn") }
```

E.3 frontend/semantics.nw

⟨frontend/semantics.mli 67k⟩≡
⟨*signature function Semantics.translate* 15e⟩

⟨frontend/semantics.ml 67l⟩≡
⟨*semantics.ml* 68a⟩

E.4 Semantic Analysis

⟨semantics.ml 68a⟩≡ (671)

```
module E = Error
module A = Ast
module S = Symbol
module Env = Environment
module Trans = Translate

open Environment
```

⟨*function* Semantics.type_name 54b⟩
⟨*function* Semantics.base_type 47b⟩
⟨*function* Semantics.lookup_base_type 47c⟩

⟨*function* Semantics.is_ptr 13f⟩
⟨*functions* Semantics.check_type_xxx 29b⟩
⟨*function* Semantics.check_type_eq 29c⟩

⟨*translators* 68b⟩
⟨*function Semantics.translate* 26a⟩

E.4.1 Type System

E.4.2 The translate Entry Point

⟨*translators* 68b⟩≡ (68a) 68c▷

⟨*global Semantics.functions* 26f⟩
⟨*function* Semantics.add_function 26g⟩

⟨*translators* 68c⟩+≡ (68a) ◁68b

⟨*functions Semantics.trxxx* 26b⟩

E.4.3 Declarations

E.4.4 Variables

E.4.5 Expressions

E.5 frontend/tree.nw

⟨frontend/tree.mli 68d⟩≡

⟨*type Tree.label* 12f⟩
⟨*type Tree.temp* 13a⟩

⟨*type* Tree.is_ptr 13e⟩

⟨*type Tree.stm* 12e⟩
⟨*type Tree.exp* 12g⟩

⟨*types Tree.xxxop* 13b⟩

⟨*utility functions* 69b⟩

⟨frontend/tree.ml 69a⟩≡
```
module S = Symbol

⟨type Tree.label 12f⟩
⟨type Tree.temp 13a⟩

⟨type Tree.is_ptr 13e⟩

⟨type Tree.stm 12e⟩
⟨type Tree.exp 12g⟩

⟨types Tree.xxxop 13b⟩

⟨tree.ml 69f⟩
```

E.6 Intermediate Representation

E.6.1 Interface to the IR

⟨utility functions 69b⟩≡ (68d) 69c▷
```
val new_label : string -> label
val new_temp  : unit   -> temp
```

⟨utility functions 69c⟩+≡ (68d) ◁69b 69d▷
```
val relop_inverse  : relop -> relop
val cmm_binop      : binop -> string
val cmm_relop      : relop -> string
```

⟨utility functions 69d⟩+≡ (68d) ◁69c 69e▷
```
val is_ptr     : exp -> is_ptr
⟨signature function Tree.find_temps 43⟩
```

⟨utility functions 69e⟩+≡ (68d) ◁69d
```
val print_stm : stm -> unit
val print_exp : exp -> unit
```

E.6.2 Implementation of IR Utilities

⟨tree.ml 69f⟩≡ (69a) 69h▷
```
⟨function Tree.new_label 69g⟩
⟨function Tree.new_temp 35d⟩
```

⟨function Tree.new_label 69g⟩≡ (69f)
```
let new_label s =
  S.new_symbol ("L" ^ s)
```

⟨tree.ml 69h⟩+≡ (69a) ◁69f 70a▷
```
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"
```

⟨tree.ml 70a⟩+≡ (69a) ◁69h 70b▷

```
  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
```

⟨tree.ml 70b⟩+≡ (69a) ◁70a 70c▷

```
  module TempSet = Set.Make(
    struct
      type t = Symbol.symbol * bool
      let compare = Stdlib.compare
    end)
```

⟨tree.ml 70c⟩+≡ (69a) ◁70b 70d▷

```
  let find_temps stmts =
    let rec stm set = function
        SEQ(a,b)     -> stm (stm set a) b
      | LABEL _      -> set
      | CONT _       -> set
      | JUMP _       -> set
      | 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,_,_,_) -> List.fold_left exp set (e :: el)
    in
    TempSet.elements (List.fold_left stm TempSet.empty stmts)
```

⟨tree.ml 70d⟩+≡ (69a) ◁70c 70e▷

```
  ⟨function Tree.print_stm 59d⟩
```

⟨tree.ml 70e⟩+≡ (69a) ◁70d

```
  ⟨function Tree.print_exp 59c⟩
```

E.7 frontend/translate.nw

⟨frontend/translate.mli 71a⟩≡
  ⟨*translate.mli* 71c⟩

⟨frontend/translate.ml 71b⟩≡
  ⟨*translate.ml* 71d⟩

E.8 Translation to Intermediate Representation

⟨translate.mli 71c⟩≡ (71a)

  ⟨*signature functions Translate.xxx* 35a⟩

E.8.1 Translation Implementation

⟨translate.ml 71d⟩≡ (71b)

```
module E = Error
module A = Ast
module S = Symbol
module T = Tree
module F = Frame
```

  ⟨*constant Translate.ws* 36g⟩

  ⟨*function Translate.temp* 35c⟩

  ⟨*function Translate.seq* 35f⟩
  ⟨*function Translate.eseq* 35e⟩

  ⟨*function Translate.simplify* 36f⟩
  ⟨*function* `Translate.angles_xxx` 36e⟩

  ⟨*function Translate.getfp* 37f⟩

  ⟨*utilities* 72a⟩

  ⟨*functions Translate.xxx literals* 35i⟩
  ⟨*function Translate.call* 39a⟩

  ⟨*functions* `Translate.ext_xxx_call` 39c⟩

  ⟨*function* `Translate.field_var` 36d⟩
  ⟨*function* `Translate.subscript_var` 36i⟩
  ⟨*function* `Translate.simple_var` 36k⟩

  ⟨*function Translate.arithmetic* 38c⟩
  ⟨*function* `Translate.compare_int` 38d⟩
  ⟨*function* `Translate.compare_str` 38e⟩

  ⟨*function Translate.assign* 40a⟩
  ⟨*function Translate.ifexp* 40c⟩
  ⟨*function Translate.loop* 40g⟩
  ⟨*function Translate.break* 41b⟩

  ⟨*function Translate.alloc* 45a⟩

⟨*function* `Translate.new_record` 37i⟩
⟨*function* `Translate.new_array` 38a⟩

⟨*function Translate.sequence* 39e⟩

⟨*function Translate.func* 35b⟩

⟨*function* `Translate.try_block` 49a⟩
⟨*function* `Translate.raise_exn` 49b⟩

⟨*function Translate.spawn* 50c⟩

⟨*utilities* 72a⟩≡ (71d) 72b▷
⟨*function Translate.getfp* 37f⟩

⟨*utilities* 72b⟩+≡ (71d) ◁72a
⟨*constant* `Translate.space_end` 45b⟩
⟨*constant* `Translate.alloc_ptr` 45c⟩

⟨*function Translate.goto* 40d⟩
⟨*function Translate.arrow* 35g⟩
⟨*function Translate.simplify* 36f⟩
⟨*function* `Translate.angles_xxx` 36e⟩

E.9 `frontend/canonical.nw`

⟨`frontend/canonical.mli` 72c⟩≡
⟨*signature function Canonical.linearize* 15g⟩

⟨`frontend/canonical.ml` 72d⟩≡
```
module T = Tree
module S = Symbol
```

⟨*constant Canonical.nop* 42c⟩

⟨*function Canonical.percent* 42b⟩

⟨*function Canonical.commute* 42a⟩

⟨*function Canonical.linearize* 41c⟩

E.10 Canonical Trees

E.11 `frontend/frame.nw`

⟨`frontend/frame.mli` 72e⟩≡
⟨*frame.mli* 73a⟩

⟨`frontend/frame.ml` 72f⟩≡
⟨*frame.ml* 73f⟩

E.12 Stack Frames

⟨frame.mli 73a⟩≡ (72e) 73b▷
```
type frame
⟨type Frame.access 36l⟩
```

⟨frame.mli 73b⟩+≡ (72e) ◁73a 73c▷
```
val fp    : frame -> Tree.exp
val name  : frame -> Tree.label
val level : frame -> int
```

⟨frame.mli 73c⟩+≡ (72e) ◁73b 73d▷
```
val base_frame : frame
val new_frame  : Tree.label -> frame -> frame
```

⟨frame.mli 73d⟩+≡ (72e) ◁73c 73e▷
```
val alloc_param  : frame  -> Tree.label -> Tree.is_ptr -> access
val alloc_local  : frame  -> Tree.label -> Tree.is_ptr -> access

val alloc_temp   : frame  -> Tree.label -> Tree.is_ptr -> unit
val alloc_string : string -> Tree.label
```

⟨frame.mli 73e⟩+≡ (72e) ◁73d
```
⟨signature function Frame.output_header 15h⟩
⟨signature function Frame.output_footer 15i⟩

val output_strings : unit  -> unit
```

E.12.1 Stack Frame Implementation

⟨frame.ml 73f⟩≡ (72f) 73g▷
```
module S = Symbol
module T = Tree
module H = Hashtbl
```

⟨frame.ml 73g⟩+≡ (72f) ◁73f 73h▷
```
⟨type Frame.frame 13g⟩
⟨type Frame.access 36l⟩
```

⟨frame.ml 73h⟩+≡ (72f) ◁73g 73i▷
```
⟨function Frame.fp 37g⟩
let name  frm =
  frm.name
let level frm =
  frm.level
```

⟨frame.ml 73i⟩+≡ (72f) ◁73h 74a▷
```
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 }
```

⟨frame.ml 74a⟩+≡ (72f) ◁73i 74b▷
```
⟨function Frame.stack_alloc 37c⟩

⟨function Frame.alloc_param 37b⟩

⟨function Frame.alloc_local 37a⟩
```

⟨frame.ml 74b⟩+≡ (72f) ◁74a 74c▷
```
let alloc_temp frm name ptr =
  frm.temps <- (name,ptr) :: frm.temps
```

⟨frame.ml 74c⟩+≡ (72f) ◁74b 74d▷
```
⟨global Frame.strings 36a⟩
⟨function Frame.alloc_string 36b⟩
```

⟨frame.ml 74d⟩+≡ (72f) ◁74c 74e▷
```
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
```

⟨frame.ml 74e⟩+≡ (72f) ◁74d 74f▷
```
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
```

⟨frame.ml 74f⟩+≡ (72f) ◁74e 74g▷
```
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"
```

⟨frame.ml 74g⟩+≡ (72f) ◁74f
```
(* 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"
```

Appendix F

backend

F.1 backend/codegen.nw

⟨backend/codegen.mli 75a⟩≡
  ⟨*codegen.mli* 75c⟩

⟨backend/codegen.ml 75b⟩≡
  ⟨*codegen.ml* 75d⟩

F.2 Code Generation

⟨codegen.mli 75c⟩≡ (75a)
```
val output_file_header : string list   -> unit
```
⟨*signature Codegen.emit* 15j⟩

⟨codegen.ml 75d⟩≡ (75b) 75e▷
```
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)
```

⟨codegen.ml 75e⟩+≡ (75b) ◁75d 75f▷
```
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"
```

⟨codegen.ml 75f⟩+≡ (75b) ◁75e
```
let emit exl =
  ⟨statements 76a⟩
  ⟨value expressions 76b⟩
  ⟨boolean expresssions 76c⟩
  in
  let code = List.map stm exl in
  List.iter (fun x -> pf "  %s\n" x) code
```

⟨*statements* 76a⟩≡ (75f)

```
    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 l              -> spf "goto %s;" (S.name l)
      | 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"
```

⟨*value expressions* 76b⟩≡ (75f)

```
    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.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* 76c⟩≡ (75f)

```
    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)
```

Appendix G

runtime

G.1 runtime/gc.nw

G.2 Allocation and Garbage Collection

G.2.1 Memory Allocator

⟨*alloc.c*– 77a⟩≡
  ⟨*alloc.c*– 77b⟩

⟨*alloc.c*– 77b⟩≡ (77a) 77c▷
```
target byteorder little;
import bits32 space_end;
import bits32 tig_gc;
export tig_alloc;
export tig_call_gc;
```

⟨*alloc.c*– 77c⟩+≡ (77a) ◁77b 77d▷
```
bits32 alloc_ptr;
tig_call_gc() {
  alloc_ptr = foreign "C" tig_gc(k) also cuts to k;
  return;
continuation k():
  return;
}
```

⟨*alloc.c*– 77d⟩+≡ (77a) ◁77c
```
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);
}
```

G.2.2 Garbage Collector Implementation

⟨gc.h 78a⟩≡
```
void* gc_init(int heap_size);
void* tig_gc(Cmm_Cont*);
```

⟨gc.c 78b⟩≡
⟨*gc.c* 78c⟩

⟨*gc.c* 78c⟩≡ (78b) 79▷
```
#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);
}
```

G.2.3 Garbage Collector Main Loop

⟨*gc.c* 79⟩+≡ (78b) ◁78c

```
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;
 }
```

G.3 runtime/runtime.nw

G.4 Runtime Startup Code

⟨*runtime.c*– 80⟩≡

```
 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);
}
```

G.4.1 Interpreter Startup

⟨client.c 81⟩≡

```
#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:pointer");
  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;
}
```

Appendix H

stdlib

H.1 stdlib/stdlib.nw

H.2 Tiger Standard Library

⟨stdlib.h 84a⟩≡
```
#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);
```

H.2.1 Standard Library Implementation

⟨stdlib.c 84b⟩≡
```
#include <qc--runtime.h>
#include "stdlib.h"
#include "gc.h"
#include <string.h>
#include <assert.h>
⟨C functions 85a⟩
```

⟨*stdlibcmm.c*– 84c⟩≡
```
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 86d⟩
}
⟨C– functions 85b⟩
```

⟨C functions 85a⟩≡ (84b) 86a▷

```
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);}
```

⟨C– functions 85b⟩≡ (84c) 85c▷

```
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);
}
```

⟨C– functions 85c⟩+≡ (84c) ◁85b 85d▷

```
tig_chr(bits32 ch) {
  bits32 str_ptr;
  str_ptr = new_string(1);
  bits8[str_ptr+4] = %lobits8(ch);
  return(str_ptr);
}
```

⟨C– functions 85d⟩+≡ (84c) ◁85c 86e▷

```
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);
}
```

⟨*C functions* 86a⟩+≡ (84b) ◁85a 86b▷
```
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);
  }
}
```

⟨*C functions* 86b⟩+≡ (84b) ◁86a 86c▷
```
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];
}
```

⟨*C functions* 86c⟩+≡ (84b) ◁86b 88a▷
```
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(s->chars, t->chars, s->length);

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

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

⟨*C– data* 86d⟩≡ (84c) 87b▷
```
 substr_msg: bits8[] "substring: index (%d,%d) out of range of (0,%d)\n\000";
```

⟨*C– functions* 86e⟩+≡ (84c) ◁85d 87a▷
```
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);
}
```

⟨*C– functions* 87a⟩+≡ (84c) ◁86e 87c▷
```
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);
}
```

⟨*C– data* 87b⟩+≡ (84c) ◁86d 88b▷
```
 /* claude: without this directive, curr_exn's byte offset in the data
  * section is whatever falls out of substr_msg's length (a string
  * literal, not a multiple of 4 in general) - qc-- lays out
  * "section data" purely by walking the declarations in source order
  * and packing them back to back; it never inserts alignment padding
  * on its own before a scalar based on that scalar's own width, only
  * where an explicit "align N;" directive appears in the source (see
  * qc--'s codegen/ast2ir.ml "datum" function and ir/asm.ml's #align
  * method - the whole pipeline from parser through codegen treats
  * "align" as its own AST node, not an inferred property of a
  * declaration's type). x86 and ppc silently tolerate the resulting
  * misaligned bits32 load/store; SPARC does not - it traps on a
  * misaligned access, so tig_set_handler (the first thing every tiger
  * program's runtime.c-- calls, since it touches curr_exn on every
  * program's very first cut/raise setup) was taking a SIGBUS before
  * tiger_main even started, on every SPARC test except hello (which
  * happens to get curr_exn 4-byte-aligned by coincidence, given the
  * specific string lengths linked into that one binary - the bug was
  * latent for it too, just not triggered). Confirmed by comparing
  * `nm`'s address for curr_exn across different linked test binaries -
  * it moved depending on what else was linked in, and landed on a
  * non-multiple-of-4 offset for forloop/funcall/queens/sieve/etc. */
 align 4;
 curr_exn : bits32;
```

⟨*C– functions* 87c⟩+≡ (84c) ◁87a 87d▷
```
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;
}
```

⟨*C– functions* 87d⟩+≡ (84c) ◁87c 88c▷
```
tig_unwind(bits32 exn_id) {
  foreign "C" unwinder(k, exn_id) also aborts also cuts to k;
  return;
continuation k():
  return;
}
```

⟨*C functions* 88a⟩+≡ (84b) ◁86c

```
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* 88b⟩+≡ (84c) ◁87b

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

⟨*C– functions* 88c⟩+≡ (84c) ◁87d

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

Appendix I

.

I.1 main.nw

I.2 Compiler Driver

⟨main.ml 89a⟩≡
```
module S = Symbol
module F = Frame
module Env = Environment
module T = Environment

⟨constant Main.base_tenv 28h⟩
⟨constant Main.base_venv 28i⟩
⟨constant Main.imports 89b⟩

⟨function Main.emit_function 15f⟩

⟨function Main.compile 15a⟩

⟨function Main.main 14a⟩

⟨toplevel Main._ 14b⟩
```

⟨constant Main.imports 89b⟩≡ (89a)
```
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
```

Appendix J

Changelog

Appendix K

Glossary

Indexes

Bibliography

[Knu92] Donald E. Knuth. *Literate Programming*. Center for the Study of Language and Information, 1992. cited page(s)

[Pad09] Yoann Padioleau. Syncweb, literate programming meets unison, 2009. https://github.com/aryx/syncweb. cited page(s)

[Ram12] Norman Ramsey. Noweb, 2012. http://www.cs.tufts.edu/~nr/noweb/. cited page(s)