Symbol Tables

- A symbol table (or environment) associates information with program identifiers — e.g., their types, their locations, ...
- Information is added when we process declarations; looked up when identifiers are used.

Implementation

- Obvious idea: maintain a hash table
  - Expected constant time lookup
  - Expected constant time insert.

- Complication: nesting of variable scopes
  - When we leave the scope of a variable we want to drop it from the current symbol table
  - But we must still have any information about "outer" variables of the same name.

Scoping

```c
int s = 0;
void f() {
    double s = 1.0;
    printf("%f", s);
    return;
}
void g() {
    int n = 3;
    printf("%d", s+n);
    return;
}
```

refers to local `s`

refers to global `s`
Alternative 1

• "Functional" data structure
  - E.g., balanced binary tree.
• The insert operation creates a new tree.
  - The old tree is unaffected and ready for use once we leave the scope of the newly-added variable.
• Advantage: easy (especially in ML)
• Disadvantage: lookup is now log-time.

Alternative 2

• Implement functions enter_scope and leave_scope.
  - Delimits the effect of imperative updates to the symbol table.
• Several possible implementations

Alternative 2a

• Stack of hash tables
  - enter_scope pushes a new table on the stack
  - leave_scope pops the top table off the stack
  - insert always inserts into the top hash table
  - lookup searches all the hash tables in the stack from newest to oldest.

Alternative 2b

• Hash table with external chaining
  - Each insert adds a new item to the hash table without deleting previous information about variables of the same name
    • and remembers that we did the insert
  - Lookup must get the most recent information about any variable name
  - enter_scope does nothing
  - must manually delete all entries when leaving scope
Alternative 2c

- Hash table with external chaining
  - Give every scope in the program a unique key (as we come to them).
    - enter_scope can allocate a fresh key
  - Maintain a current stack of scope identifiers
    - pushed on enter_scope, popped on leave_scope
  - Insert adds new information marked with scope key
  - Lookup searches for newest variable in scope
  - Items never deleted from the symbol table

Tiger Compiler

- Since we’re programming in SML, we’ll use a the tree representation.
  - But with a slight optimization.
  - To avoid string comparisons, each variable name will correspond to an integer.
    - Arbitrary, but fixed throughout compilation
    - Searches can then do integer comparisons

Symbols

```signature SYMBOL =
  sig
    eqtype symbol
    val symbol : string -> symbol
    val name : symbol -> string
  type 'a table
  val empty : 'a table
  val enter : 'a table * symbol * 'a -> 'a table
  val look : 'a table * symbol -> 'a option
  end```

The Tiger Type System

- Base types
  - Just int and string
- Record types
  - Written as in ML, `{x:int,y:int}`
  - Unlike ML, every time you write this it refers to a new record type.
- Array types
  - Written as array of int
  - Also defines a new array type every time.
Record Types

let type a = {x:int, y:int}
type c = a
var i : a := a{x=3,y=4}
var j : c := c{x=2,y=5}
in
i := j  /* this is ok */
end

Record Types

let type a = {x:int, y:int}
type c = {x:int, y:int}
var i : a := a{x=3,y=4}
var j : c := c{x=2,y=5}
in
i := j  /* this isn't */
end

Record Types

type intlist = {hd:int, tl: intlist}  /* ok */
type tree = {key:int, children: treelist}  /* ok */
type treelist = {hd:tree, tl: treelist}
type b = c  /* not ok */
type c = b

- Adjacent record types can be recursive
- But, recursion must go through a record type.

Nil

- The constant nil is a member of every record type.
- Can only be used where context determines its type.

var a : my_record := nil
a := nil
if a <> nil then ...
if nil <> a then ...
if a = nil then ...
function f(p:my_record) ... f(nil)

var a := nil
if nil <> nil then ...

ok
not ok
Array Types

```ml
let type a = array of int
type c = a
var i : a := a[3] of 0
in
  i := j /* this is ok */
end
```

Array Types

```ml
let type a = array of int
  type c = array of int
var i : a := a[3] of 0
in
  i := j /* this isn't */
end
```

Internal Type Representation

```ml
structure Types =
  struct
    type unique = unit ref

  datatype ty = INT
  | STRING
  | RECORD of (Symbol.symbol * ty) list * unique
  | ARRAY of ty * unique
  | NIL
  | UNIT
  | NAME of Symbol.symbol * ty option ref
end
```

Type Environments

```ml
signature ENV =
  sig
    type ty = Types.ty

    datatype enventry = VarEntry of {ty : ty}
    | FunEntry of {formals : ty list, result : ty}

    val base_tenv : ty Symbol.table
    val base_venv : enventry Symbol.table
  end
```
Typechecker Interface

```ocaml
type venv = Env.enventry Symbol.table
type tenv = Types.ty Symbol.table

val transVar : venv * tenv * Absyn.var -> Types.ty
val transExp : venv * tenv * Absyn.exp -> Types.ty
val transDec : venv * tenv * Absyn.dec -> {venv : venv, tenv : tenv}
val transTy : tenv * Absyn.ty -> Types.ty
```

Sample Code

```ocaml
  let val tyleft = transExp(venv,tenv,left)
      val tyright = transExp(venv,tenv,right)
  in
    case (tyleft,tyright) of
      (Types.INT,Types.INT) => Types.INT
    | _ => error pos "integer required"
  end
|...
```

Sample Code

```ocaml
fun transVar(venv, tenv, Absyn.SimpleVar(id, pos)) =
  (case Symbol.look(venv,id) of
    SOME(E.VarEntry{ty}) => ty
  | NONE => error pos "undefined variable")
|...
```

Sample Code

```ocaml
transExp(venv, tenv, Absyn.LetExp{decs,body,pos}) =
  let
    val {venv=venv', tenv=tenv'} =
      transDecs(venv,tenv,decs)
  in
    transExp(venv',tenv',body)
  end
|...
```
fun transDec(venv, tenv, Absyn.VarDec{name,type=NONE,init,...}) = 
  let 
  val ty = transExp(venv,tenv,init) 
  in 
  {tenv=tenv, 
   venv=Symbol.enter(venv,name, Env.VarEntry{ty=ty}} 
  end 
| ...

| transDec(venv, tenv, Absyn.TypeDec{name,ty,pos}) = 
  (venv = venv, 
   tenv = Symbol.enter(tenv,name, transTy(tenv,ty))) 
| ...

| transDec(venv, tenv, A.FunctionDec[{name,params,body,pos, 
  result=SOME{rt,pos}}]) = 
| ...

Sample Code