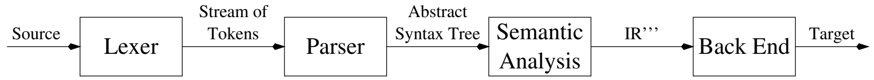
Today's Lecture

- Symbol Tables
- Type Checking

Symbol Tables



- Semantic Analysis Phase:
  - Type check AST to make sure each expression has correct type
  - Translate AST into IR trees
- Main data structure used by semantic analysis: *symbol table*
  - Contains entries mapping identifiers to their bindings (e.g. type)
  - As new type, variable, function declarations encountered, symbol table augmented with entries mapping identifiers to bindings.
  - When identifier subsequently used, symbol table consulted to find info about identifier.
  - When identifier goes out of scope, entries are removed.

Symbol Table Example

```
function f(b:int,
           c:int) =
  (print_int(b+c);
   let
     var j := b
     var a := "x"
   in
     print(a)
     print(j)
   end
   print_int(a)
  )
```

$\sigma_0 = \{a \mapsto int\}$

$\sigma_1 = \{b \mapsto int, c \mapsto int, a \mapsto int\}$

$\sigma_2 = \{j \mapsto int, b \mapsto int, c \mapsto int, a \mapsto int\}$

$\sigma_3 = \{a \mapsto string, j \mapsto int, b \mapsto int, c \mapsto int, a \mapsto int\}$

$\sigma_1 = \{b \mapsto int, c \mapsto int, a \mapsto int\}$

$\sigma_0 = \{a \mapsto int\}$

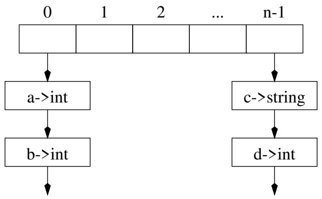
Symbol Table Implementation

- Imperative Style: (side effects)
  - Global symbol table
  - When beginning-of-scope entered, entries added to table using side-effects. (old table destroyed)
  - When end-of-scope reached, auxiliary info used to remove previous additions. (old table reconstructed)
- Functional Style: (no side effects)
  - When beginning-of-scope entered, *new* environment created by adding to old one, but old table remains intact.
  - When end-of-scope reached, retrieve old table.

Imperative Symbol Tables

Symbol tables must permit fast lookup of identifiers.

- *Hash Tables* - an array of *buckets*
- *Bucket* - linked list of entries (each entry maps identifier to binding)



- Suppose we with to lookup entry for id i in symbol table:
  1. Apply *hash function* to key i to get array element $j \in [0, n-1]$.
  2. Traverse bucket in table[j] in order to find binding b.
     (table[x]: all entries whose keys hash to x)

Imperative Symbol Tables

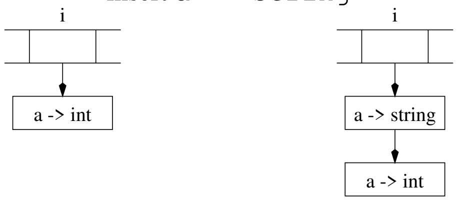
```
val size = 109    (* prime number *)
type binding = ...
type bucket = (string * binding) list
type table = bucket Array.array
val t:table = Array.array(SIZE, nil)
   (* assume:  fun hash(s:string) -> 0 <= j < SIZE *)

exception notFound
fun lookup(s:string) =
  let
    val i = hash(s)
    fun search((s’, b)::rest) = if s = s’ then b
                                else search rest
      | search([]) = raise notFound
  in
    search(Array.sub(t,i))
  end
```


Imperative Symbol Tables

```
fun insert(s:string, b:binding) =
  let
    val i = hash(s)
  in
    Array.update(t,i,(s,b)::Array.sub(t,i))
  end
```

Inserts new element at front of bucket.

insert a $\mapsto$ string



Imperative Symbol Tables

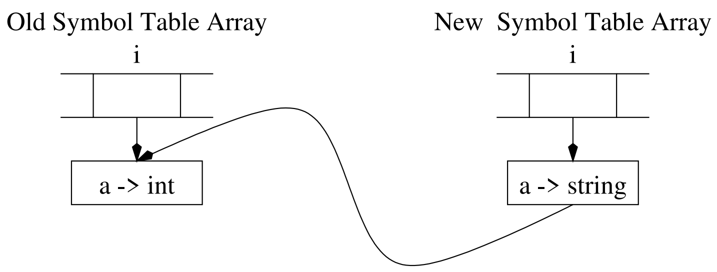
To restore hash table, pop items off items at front of bucket.

```
fun pop(s:string) =
  let
    val i = hash(s)
    val (s’, b)::rest = Array.sub(t,i)
  in
    assert(s = s’)
    Array.update(t,i,rest)
  end
```

Functional Symbol Tables

Hash tables not efficient for functional symbol tables.

Insert `a` $\mapsto$ `string` $\Rightarrow$ copy array, share buckets:

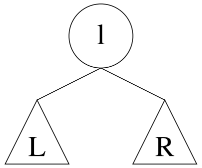


Not feasible to copy array each time entry added to table.

Functional Symbol Tables

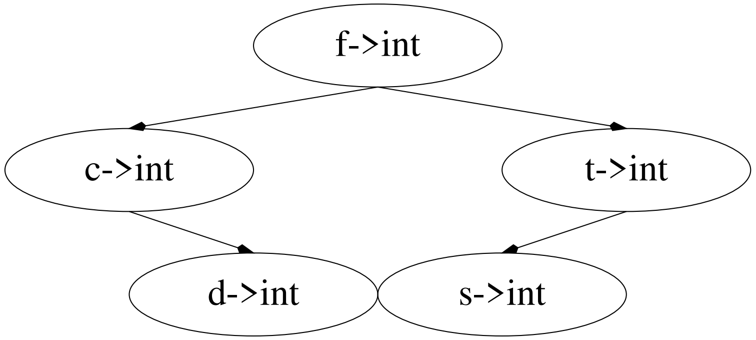
Better method: use *binary search trees (BSTs)*.

- Functional additions easy.
- Need "less than" ordering to build tree.
  - Each node contains mapping from identifier (key) to binding.
  - Use string comparison for "less than" ordering.
  - For all nodes $n \in L$, key(n) $<$ key(l)
    For all nodes $n \in R$, key(n) $>=$ key(l)



Functional Symbol Table Example

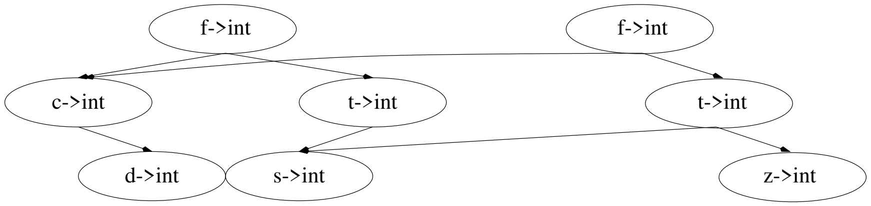
Lookup:



Functional Symbol Table Example

Insert:

insert `z` $\mapsto$ `int`, create node `z`, copy all ancestors of `z`:



Issues With Table Implementations

When key value = string
⇒ need to do expensive string compares when doing lookup operation.
Solution: use *symbol* data structure instead

- Each symbol object associated with integer value.
- All occurrences of same string map onto same symbol (2 different strings map onto different symbols)
- key value = symbol ⇒ do cheap integer comparisons during lookup

Issues With Table Implementations

```
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
```

- Implements symbol tables (function) using BSTs.
- Table is polymorphic: each entry maps symbol to binding of type 'a
  - Need type bindings for type symbols.
  - Need value bindings for variable and function symbols.

Issues With Table Implementations

```
structure Symbol:SYMBOL =
struct
  type symbol = string * int
  val nextsym = ref 0
  fun symbol(name:string) =
    case HashTable.find hashtable name of
      SOME(i) => (name, i)
    | NONE => let
                val i = !nextsym
              in
                nextsym := i + 1;
                HashTable.insert hashtable(name, i);
                (name, i)
              end
```


Issues With Table Implementations

```
  fun name((s,n)) = s
  type 'a table = 'a IntBinaryMap.map
  val empty = IntBinaryMap.empty
  fun enter(t:'a table, (s,n):symbol, a:'a) =
    IntBinaryMap.insert(t,n,a)
  fun look(t:'a table, (s,n):symbol) =
    IntBinaryMap.look(t,n)
end
```

Environments in Tiger Compiler

Two name spaces (types, variables/functions) ⇒ two environments

- *type environment*: maps types symbols to type that it stands for
- *value environment*:
  - Maps variable symbols to their types.
  - Maps function symbols to parameter and result types.

Type Environment

```
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
```

- In order to distinguish each record type, associate unit ref value with RECORD data constructor
  - Each ref is unique.
  - Can compare it with another unit ref for equality.
- NAME: used when processing mutually-recursive types, placeholder for types whose name is known, but whose definition has yet to be seen.

Value Environment

```
signature ENV = sig
  type access
  type 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
```

- `base_tenv` contains `''int''` $\mapsto$ `INT`, `''string''` $\mapsto$ `STRING`.
- `base_venv` contains predefined functions in appendix.

Type Checking

Symbol structure implements functional symbol tables using BSTs.

- `type 'a table` $\rightarrow$ environment contains mappings from symbol to 'a
- `val empty:'a table`
- `val enter:'a table * symbol * 'a -> 'a table`
- `val look:'a table * symbol -> 'a option`

Type Checking

Need 2 environments:

1. *Type environment*: maps type symbol to type that it stands for
   Types.ty - describes bindings for type environment
   ⇒ Types.ty Symbol.table
2. *Value environment*: maps variable symbol to its type
   Maps function symbol to parameter and result types
   Env.enventry - describes bindings for value environment

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

   ⇒ Env.enventry Symbol.table

Env structure contains predefined type and value environments:

- `base_tenv` contains `''int''` ↦ `INT`, `''string''` ↦ `STRING`.
- `base_venv` contains predefined functions in appendix.

Type Checking Expressions

Semant structure: performs type-checking of ASTs

```
type venv = Env.enventry Symbol.table
type tenv = Types.ty Symbol.table
type expty = {exp: Translate.exp, ty: Types.ty}
```

- Will be implementing four primary functions

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

- For now, not concerned with translation into IR code, so use `()` for every `Translate.exp` value:

```
structure Translate = struct
  type exp = unit
end
```


Type Checking Expressions

```
fun transProg(t) = let
  val {exp, ty} = transExp(Env.base_venv, Env.base_tenv) t
in
  exp
end

structure A = Absyn

fun checkInt({exp, ty}, pos) =
  (if ty = Types.INT then ()
  else ErrorMsg.error pos "int required";
  exp)
```


General Structure of `transExp`

```
fun transExp(venv, tenv) = let
    fun trexp(A.IntExp...) = ...
      | trexp(A.OpExp...) = ...
      | ...
  in
    trexp
  end
```

Suppose we want to type-check e1 + e2

- Both e1, e2 must be ints
- Type of expression is `INT`

```
fun transExp(venv, tenv) = let
    fun trexp(A.OpExp{left, oper = A.PlusOp, right, pos}) =
      (checkInt(trexp(left), pos);
       checkInt(trexp(right), pos);
       {exp = (), ty = Types.INT})
```

General Structure of `transExp`

Type-check 'while' expression:

```
| trexp(A.WhileExp{test, body, pos}) =
(checkInt(trexp(test), pos);
 checkUnit(trexp(body), pos);
 {exp = (), ty = Types.UNIT})
```


Type Checking Variables

```
| trexp(A.VarExp(v)) = trvar(v)
...
and trvar(A.SimpleVar(id, pos)) =
  (case Symbol.look(venv, id) of
      SOME(Env.VarEntry{ty}) => {exp=(), ty=actual_ty(ty)}
    | NONE => (ErrorMsg.error pos
                       ("undefined var") ^ Symbol.name(id));
      {exp=(), ty=Types.INT})
```

- Type in `VarEntry` may be `NAME` type, `NAME` used as placeholder when processing mutually recursive types.

Type Checking Variables

- Type returned by `trexp` must be *actual* type that is not a `NAME`
  `actual_ty(t)` skips past all `NAME`s in `t` until underlying type reached.
  - `Types.NAME(sym, ref SOME(t))` ≡ `t`
  - `Types.NAME(a,`
    `ref SOME(Types.NAME(b,`
    `ref SOME(Types.INT))))` ≡ `Types.INT`
    (OK for record types to have NAME components)

```
| trvar(A.SubscriptVar(var, exp, pos)) =
```

- Make sure `exp` is of type `T.INT` → (apply `trexp`)
- Make sure `var` is of type `T.ARRAY(t, u)` → (apply `trvar`)
- Result type = `t`

Type Checking Declarations

Declarations modify environments, appear only in `LET` expressions.

```
| trexp(A.LetExp{decs,body,pos}) =
  let
    val {venv=venv', tenv=tenv'} = transDecs(venv,tenv,decs)
  in
    transExp(venv', tenv') body
  end
```

Var Declarations

```
  var x := exp

fun transDec(venv, tenv,
             A.VarDec{name, escape, typ=NONE, init, pos}) =
  let
    val {exp,ty} = transExp(venv,tenv) init
  in
    if ty=Types.NIL then
      (ErrorMsg.error pos ``var type must be record'';
       {tenv=tenv,
        venv=Symbol.enter(venv,name,
                          Env.VarEntry{ty=Types.INT})})
    else
      {tenv=tenv,
       venv=Symbol.enter(venv,name,Env.VarEntry{ty=ty})}
  end
```


Type Declarations

Consider non-recursive type decs:

```
| transDec(venv, tenv, A.TypeDec[{name,ty,pos}]) =
    {venv=venv, tenv=Symbol.enter(tenv, name,
                                   transTy(tenv, ty))}
```

`transTy` translates `Absyn.ty` into `Types.ty`
(e.g. `Absyn.ArrayTy` → `Types.ARRAY`)

Function Declarations

`function f(a:int) = body`

1. Look up 'int' in `tenv` → `Types.INT`
2. `venv' = venv + f` ↦ `Env.FunEntry { formals = [Types.INT], result = Types.UNIT }`
3. `venv'' = venv' + a` ↦ `Env.VarEntry { ty = Types.INT }`
4. Type-check body in {`tenv, venv''`} using `transExp`
5. Return {`tenv, venv'`} for use in processing expressions which refer to `f`

Recursive Declarations

Consider type declaration: `type list = {first:int, rest:list}`
When `transTy` translates `A.RecordTy` corresponding to record, will encounter undefined type '`list`'.
Solution: use two passes:

1. Put all type "headers" into type environment, but ignore "bodies" (RHS) - use `Types.NAME` to represent headers.
2. Call `transTy` on body, give it new type environment
   Assign result into reference variable of `NAME`

Recursive Declaration Example

`type list = {first:int, rest:list}`

After pass 1:
`list` $\mapsto$ `Types.NAME(list, ref NONE)`

After pass 2:
`list` $\mapsto$ `Types.NAME(list, ref SOME)`
`Types.RECORD([(first, Types.INT),(rest, Types.NAME(list, ref SOME))], ref())`

- `transTy` must stop as soon as it finds a `NAME` type.
- If behavior like `actual_ty`, would encounter `NONE` when skipping past `NAME` types.

Recursive Function Declaration Example

Mutually-recursive function declarations handled similarly.

```
function f1(a:int):int = f2(a)
function f2(b:int):int = f1(b)
```

Pass 1:
Put all function headers into value environment, but ignore bodies.
`f1` $\mapsto$ `Env.FunEntry{formals=[Types.INT], result=Types.INT}`
`f2` $\mapsto$ `Env.FunEntry{formals=[Types.INT], result=Types.INT}`

Pass 2:
Process function bodies in new value environment.
For each body, enter `VarEntry`'s for each formal parameter.