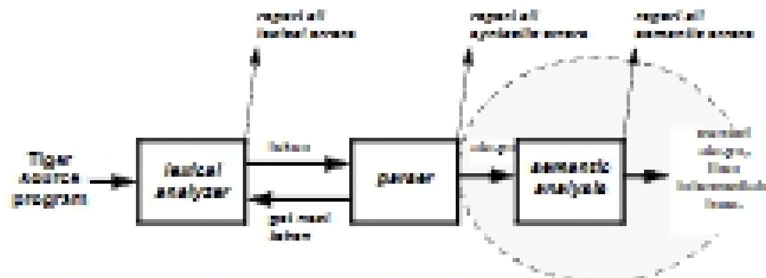
Tiger Semantic Analysis



- *construct variable definitions to their uses*
- *checks that each expression has a correct type*
- *translates the abstract syntax into a simpler intermediate representation suitable for generating machine code.*

Connecting Definition and Use ?

- *Make sure each variable is defined; Check the type consistency !*

```
.......
function f(v : int) =
 let var v := 6
     function g(x : int) =
          (print (x+v); print "\n")

     function h(v : int) =
          (print v; print "\n")

   in g v;
      let var v := 5 in print v end;
      h v;
  end
```

- *Solution: use a symbol table --- traverse the abstract syntax tree in certain order while maintaining a "(variable -> type)" symbol table.*

Symbol Tables

- *Conceptually, a symbol table (also called environment) is a set of "(name, attribute)" pairs.*
- *Typical Names:* *strings, e.g.,* `"foo"`, `"do_nothing1"`, ...
- *Typical Attributes (also called bindings):*

type identifier	type (e.g., `int`, `string`)
variable identifier	type ; access info. or value
function identifier	arg. & result type; access info. or ...

- ***Main Issues --- for a symbol table T***

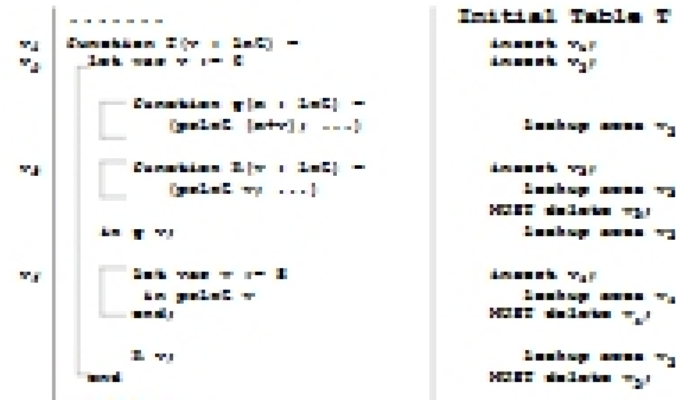
  Given an identifier name, how to look up its attribute in T ?

  How to insert or delete a pair of new "(id, *attr*)" into the table T?

  Efficiency is important !!!

Symbol Tables (cont'd)

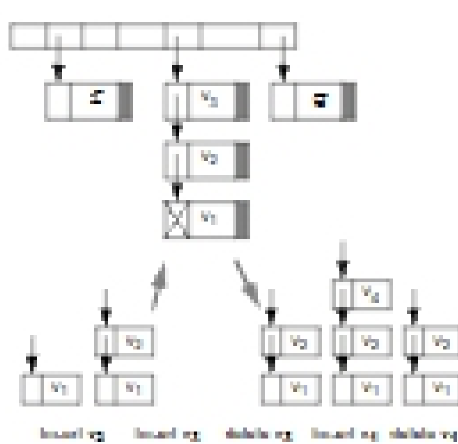
- *How to deal with visibility (i.e., lexical scoping under nested block structure) ?*



Symbol Table Impl.

- *Hash Table — efficient; but need explicit "delete" due to side-effects !*

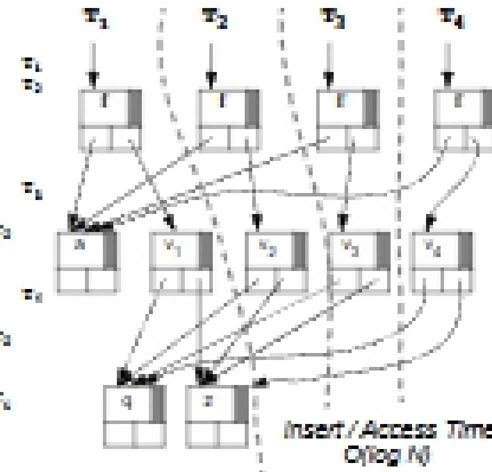
```
Initial Table T
    insert v1;
    insert v2;

       lookup some v2

    insert v3;
       lookup some v3
    MUST delete v3;
       lookup some v2

    insert v4;
       lookup some v4
    MUST delete v4;

       lookup some v2
    MUST delete v2;
```



Symbol Table Impl. (cont'd)

- *Balanced Binary-Tree — "persistent", "functional", yet "efficient"*

```
Initial Table T0

    insert v1;                 T1
    insert v2;                 T2

       lookup some v2

    insert v3;                 T3
       lookup some v3
(* delete v3;*)   use T2
       lookup some v2

    insert v4;                 T4
       lookup some v4
(* delete v4;*)   use T2

       lookup some v2
(* delete v2;*)   use T1
```



Insert / Access Time O(log N)

Summary: Symbol Table Impl.

- *Using hash-table is ok but explicit "delete" is a big headache !*
- *We prefer the functional approach — using persistent balanced binary tree — no need to explicit "delete"; access and insertion time O(log N)*
- *The Symbol signature (symbol table is an abstract datatype — used to hide the implementation details)*

```
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
```

No "delete" because we use "functional" approach!

String <==> Symbol

- *Using string as the search key is slow — involves a string comparison*
- *Associate each string with a integer — which is used as the key for all access to the symbol table (i.e., binary tree)*

```
type symbol = string * int

exception Symbol
val nextsym = ref 0

structure H = ... a HashTable from STRING to INTEGER ...

fun symbol name =
  case H.find hashtable name
   of SOME i => (name, i)
    | NONE => let val i = !nextsym
               in inc nextsym;
                  H.insert hashtable (name,i);
                  (name,i)
              end

fun name(s,n) = s
```

Summary: Symbol Table

- *A symbol is a pair of string and integer* `(s,n)` *where the string* `s` *is the identifier name, the integer* `n` *is its associated search key.*
- *The mapping from a string to its corresponding search key (a integer) is implemented using a hash table.*
- *The symbol table — from a symbol to its attributes — is implemented using IntBinaryMap — a persistent balanced binary tree.*

```
structure Symbol :> SYMBOL = (* see Appel page 110 *)
struct
  type symbol = string * int
  ......
  type 'a table = 'a IntBinaryMap.intmap (* in SML Library *)

  val empty = IntBinaryMap.empty
  fun enter(t, (s,n),a) = IntBinaryMap.insert(t,n,a)
  fun look(t, (s,n)) = IntBinaryMap.look(t,n)
end
```

Environments

- *Bindings — interesting attributes associated with type, variable, or function identifiers during compilations.*
- *Type bindings — internal representation of types*

```
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
```

- *Variable/Function Bindings — type + location&access information*

Environments (cont'd)

- *The signature for Environment*

```
signature Env =
struct
  type access
  type level
  type label
  type ty    (* = Type.ty *)

  datatype enventry
      = VARentry of {access : access, ty : ty}
      | FUNentry of {level : level, label : label,
                     formals : ty list, result : ty}

  val base_tenv : ty Symbol.table
  val base_env : enventry Symbol.table
end
```


Normally we build one environment for each name space !

`base_tenv` *is the initial type environment*
`base_env` *is the initial variable+function environment*

Tiger Absyn

```
datatype 'a option = NONE | SOME of 'a

datatype var = ...
and exp
  = ...
  | OpExp of {left: exp, oper: oper, right: exp,...
  | LetExp of {decs: dec list, body: exp, ...

and dec
  = FunctionDec of fundec list
  | TypeDec of tydec list
  | VarDec of vardec

withtype
    field = {name: symbol, typ: symbol, pos: pos}

and fundec = {name: symbol, params: field list,
              result : (symbol * pos) option,
              body: exp, pos: pos}
```