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!

```plaintext
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 := 8 in print v end;
    h v;
end
```

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 (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)?

```plaintext
Initial Table T

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>v1</td>
<td>function f(v : int) =</td>
<td></td>
</tr>
<tr>
<td>v2</td>
<td>let var v := 6</td>
<td></td>
</tr>
<tr>
<td></td>
<td>function g(x : int) =</td>
<td></td>
</tr>
</tbody>
</table>
|    |    (print (x+v); ...
| v3 | function h(v : int) = |
|    |    (print v); ...
|    | in g v;
| v4 |    let var v := 8 |
|    |    in print v |
|    |    end;
|    | h v;
|    | end
```

```plaintext
Insert v1:
insert v1;
lookup sees v2:
insert v2;
lookup sees v3:
MUST delete v2;
lookup sees v2:
insert v3;
lookup sees v4:
MUST delete v4;
lookup sees v2:
MUST delete v2;
```
Symbol Table Impl.

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

  **Initial Table T**
  - insert v1
  - insert v2
  - lookup sees v2
  - insert v3
  - lookup sees v3
  - MUST delete v3
  - lookup sees v2
  - MUST delete v2
  - lookup sees v2
  - insert v4
  - lookup sees v4
  - MUST delete v4
  - lookup sees v2
  - insert v3
  - insert v3

  delete v3
  insert v4
  delete v4

Symbol Table Impl. (cont’d)

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

  **Initial Table T₀**
  - insert v1
  - insert v2
  - lookup sees v2
  - insert v3
  - lookup sees v3
  - (* delete v3;*) use T₂
  - lookup sees v2
  - insert v4
  - lookup sees v4
  - (* delete v4;*) use T₂
  - lookup sees 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

String \(\leftrightarrow\) Symbol

- Using string as the search key is slow --- involves a string comparison

- Associate each string with an 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

  No “delete” because we use “functional” approach!
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.

```plaintext
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

```plaintext
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
```

Environments (cont’d)

- The signature for Environment

```plaintext
signature Env =
struct
  type access
  type level
  type label
  type ty (* = Type.ty *)
  datatype enventry
    = VARentry of {access : access, ty : ty} (* VARentry 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!
```

Tiger Absyn

```plaintext
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}
```
Type-Checking Expressions

```ml
(* transexp : env * tenv -> exp -> ty *)
fun transexp (env,tenv) =
  let fun g(OpExp{left,oper=A.plusOp,right,pos}) =
    (checkInt(g left, pos);
     checkInt(g right, pos);
     Types.INT)
  | g(LetExp{decs, body, pos}) =
    let val (env',tenv') = transdecs (env,tenv) decs
    in transexp (env',tenv') body
    end
  in g
end
```

Type-Checking Declarations

```ml
(* transdec : env * tenv -> dec -> env * tenv *)
fun transdec (env,tenv) =
  let fun g(VarDec{var,typ=NONE,init}) =
    let val ty = transexp (env,tenv) init
    val b = VARentry{access=(),ty=ty}
    in (enter(env,var,b), tenv)
    end
  | g(FunctionDec[{name,params,body,pos,result=_}])= 
    let val b = FUNentry{...}
    val env' = enter(env,name,b)
    val env'' = enterparams(params,env')
    in transexp (env'',tenv) body;
    (env', tenv)
    end
  | ...
  in g
end
```

Type-Checking

- **The type** of an expression tells us the values it can denote and the operations that can be applied to it.
- **Type system** — definition of well-formed types + a set of **typing rules** that define what type-consistency means.
- **Type-checking** ensures that the operations in a program are applied properly. A program that executes without type errors is said to be **type safe**.
- **Static** Type-checking: type are checked at **compile time**, (once and for all)

  ![Type-checking Diagram](https://via.placeholder.com/150)

- **Dynamic** Type-checking: types are checked at **run time**, (inside the code)

Type Safety

- Modern programming languages are always equipped with a **strong type system** — meaning a program will either run successfully, or the compiler & the runtime system will report the type error.
  - strongly-typed languages: Modula-3, Scheme, ML, Haskell
  - weakly-typed languages: C, C++
- **Safety** — a language feature is **unsafe** if its misuse can corrupt the runtime system so that further execution of the program is not faithful to the language semantics. (e.g., no array bounds checking, ...)
- **A statically-typed language** (e.g., ML, Haskell) does most of its type-checking at **compile time** (except array-bounds checking).
- **A dynamically-typed language** (e.g., Scheme, Lisp) does most of its type-checking at **run time**.
Main Issues

• What are valid type expressions?
  e.g., int, string, unit, nil, array of int, record {...

• How to define two types are equivalent?
  name equivalence or structure equivalence

• What are the typing rules?

• How much type info should be specified in the source program?
  implicitly-typed lang., e.g., ML ----- uses type inference
  explicitly-typed lang. e.g., Tiger, Modula-3 ----- must specify the type of each newly-introduced variables.

Types in Tiger

Tiger types are

\[ \text{ty} \rightarrow \text{type-id} \mid \text{array of type-id} \mid \{} \mid \{\text{id : type-id}, \text{id : type-id}\} \]

\text{type-id} \text{ is defined by type declarations:}

\[ \text{tydec} \rightarrow \text{type type-id = ty} \]

Typechecker must translate all source-level type specification (in absyn) into the following internal type representation:

\[ \text{structure Types = struct type unique = unit ref}\]
\[ \text{datatype ty = RECORD of (Symbol.symbol * ty) list * unique}\]
\[ \text{NIL}\]
\[ \text{INT}\]
\[ \text{STRING}\]
\[ \text{ARRAY of ty * unique}\]
\[ \text{NAME of Symbol.symbol * ty option ref}\]
\[ \text{UNIT}\]

implementing Name Equivalence

for recursive type declarations

Type Equivalence

When are two type expressions equivalent?

• Name equivalence (NE): \( T_1 \) and \( T_2 \) are equivalent iff \( T_1 \) and \( T_2 \) are identical type names defined by the exact same type declaration.

• Structure equivalence (SE): \( T_1 \) and \( T_2 \) are equivalent iff \( T_1 \) and \( T_2 \) are composed of the same constructors applied in the same order.

Here \( \text{point} \) and \( \text{ptr} \) are equivalent under SE but not equivalent under NE

\[ \text{point} = \{x : \text{int}, y : \text{int}\} \]
\[ \text{type ptr} = \{x : \text{int}, y : \text{int}\} \]
\[ \text{function f(x : point) = a} \]

Here the redeclaration of \( \text{point} \) defines a new type under NE; thus it is a type error

when function \( f \) is applied to \( p \)

\[ \text{type point} = \{x : \text{int}, y : \text{int}\} \]
\[ \text{var p : point = point} \{x=3, y=5\} \]
\[ \text{var q : point = f(p)} \]

Typing Rules in Tiger

• Tiger uses name equivalence; type constraints must be a \text{type-id} (used on variable declarations, function parameters and results, array elements, and record fields)

• The expression \text{nil} has the special type \text{NIL}. \text{NIL} belongs to every record type -- it is equivalent to any record type. \text{nil} must be used in a context where its type can be determined.

\[ \text{var p : point := nil} \quad \text{OK} \]
\[ \text{if p <> nil then ...} \quad \text{OK} \]
\[ \text{var a := nil} \quad \text{Illegal} \]

• For variable declaration: \[ \text{var id : type-id := exp} \]
  the type of expression \text{exp} must be equivalent to type \text{type-id}.

• Assignment expression \[ \text{id := exp} \]
  \text{id} & \text{exp} have equivalent type.
Typing Rules in Tiger (cont’d)

- Function call: the types of formal parameters must be equivalent to the types of actual arguments.
- Array subscript must have integer type.
- Array creation `type-id [exp] of exp1` has type int, `exp2` must have type equivalent to that of the element of `type-id`
- Record creation `type-id {id = exp1,...}` the type of each field (`exp1`) must have type equivalent to that defined in `type-id`
- If-expression `if exp then exp2 else exp3` the type of `exp` must be integer, the type of `exp2` and `exp3` should be equivalent.
- For-expression `for id := exp1 to exp2 do exp3` the type of `exp1` and `exp2` must be integer. `exp3` should produce no value ...
- For more info, read Appendix in Appel.

Recursive Type Declarations

- How to convert the following declaration into the internal type representations?

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

Problem: when we do the conversion of the r.h.s., “list” is not defined in the `tenv` yet.

Solution: use the special `Name` type

```
datatype ty = NAME of Symbol.symbol * ty option ref |
```

First, enter a “header” type for `list`
```
val tenv' = enter(tenv, name, NAME(name, ref NONE))
```

Then, we process the body (i.e., r.h.s.) of the type declarations, and assign the result into the reference cell in the `NAME` type

Recursive Function Declarations

- Problem: when we process the right hand side of function declarations, we may encounter symbols that are not defined in the `env` yet

```
function do_nothing1(a: int, b: string) = do_nothing2(a+1)
function do_nothing2(d: int) = do_nothing1(d, "str")
```

Solution: first put all function names (on the l.h.s.) with their header information (e.g., parameter list, function name, type, etc., all can be figured out easily) into the `env` then process each function’s body in this augmented `env`.

Other Semantic Check

- Many other things can be done in the type-checking phase:
  - resolve overloading operators
  - type inference
  - check if all identifiers are defined
  - check correct nesting of `break` statements.

Comming soon ---

Assignment 5 is to write the type-checker.