CSC4510 Programming Language Design and Implementation

Michael Oudshoorn

1

Using the ID Table

- The compiler starter kit provides you with the id_table class outline.
- Code needs to be added to provide functionality.
- An id_table_entry class is created to hold the information stored in the is_table.
- The id_table also manages the scope level.

2

Id_table

```
#include <iostream>
#include <string>

#include "token.h"
#include "error_handler.h"
#include "id_table_entry.h"
#include "lille_type.h"
#include "lille_kind.h"

using namespace std;

class id_table {
private:
    error_handler* error;

    bool debug_mode;
    int scope_level;
```

```
static const int max_depth = 1000;
// maximum depth of nesting permitted in source code.

struct node {
    node* left;
    node* right;
    id_table_entry* idt;
};

node* sym_table[max_depth];

node* search_tree(string s, node* p);
void add_table_entry(id_table_entry* it, node* ptr);
void dump_tree(node* ptr);
```

HIGH POINT UNIVERSITY

3

Id_table

```
public:
    id_table(error_handler* err);

    void enter_scope();
    void exit_scope();
    int scope();
    id_table_entry* lookup(string s);
    id_table_entry* lookup(token* tok);
    void trace_all(bool b);
    bool trace_all();

    void add_table_entry(id_table_entry* id);
```

```
id_table_entry* enter_id(token* id,
    lille_type typ = lille_type::type_unknown,
    lille_kind kind = lille_kind::unknown,
    int level = 0,
    int offset = 0,
    lille_type return_tipe =
            lille_type::type_unknown);

    void dump_id_table(bool dump_all = true);
};
```

HIGH POINT UNIVERSITY

4

Id_table

- Note that the id_table can generate errors such as attempts to add duplicate entries into the table, attempts to access a field that is inappropriate (i.e., return the integer field of a string variable).
- The pragma debug in *lille* is processed by the scanner (not the parser) and managed by the id_table. The parser generates PAL code to trace variables as determined by the flags in the id_table.
- The id_table is an array of nodes. Each node is a binary tree of id_table entries.
- There are public methods to add entries to the current scope level and to lookup id_table_entries matching a string or a token.

HIGH POINT UNIVERSITY

5

Id_table

- Look up searches from the current scope level down to scope level 0 to find a matching id_table_entry if possible.
- Scope level 0 is for predefined procedures and functions in lille.
  - These are added manually by the parser before he source code is processed.
  - They can be overridden at higher scope levels by procedures and functions with the same name.

HIGH POINT UNIVERSITY

6

Id_table

- For each entry in the id_table we need to track:
  - Whether we are tracing the variable (pragma trace).
  - The level and offset of the identifier (necessary for code generation).
  - The token that result the creation of the id_table_entry.
  - The kind of entity (variable, constant, function, procedure, ..)
  - The type of the entity (integer, real, ...)
  - The return type if it is a function
  - The actual vale if it is a constant.
  - The number of parameters in the case of a procedure or function.
  - A mechanism to link each parameter to a function or procedure.

HIGH POINT UNIVERSITY

7

Id_table

- The id_table is typically accessed by the parser:
  - Whenever you enter and exit a scope level (enter or exit a procedure or function).
  - Also increment and decrement scope level to generate code for a for loop (we treat them as though they were a procedure).
  - Whenever a declaration occurs, an entry is added to the symbol table at the the current scope level.
    - Constant
    - Variable
    - Procedure
    - Function
    - Parameter

HIGH POINT UNIVERSITY

8

Id_table support

- To support the *id_table* and *id_table_entry* classes, *lille_kind* helps manage the kind of object inserted into the table (variable, constant, etc) and *lille_type* helps manage the type system of *lille* and the types of objects inserted into the table (*integer*, *real*, *string* etc).
- Read the code provided before attempting to implement the *id_table* and *id_table_entry* classes.
- Remember – you are free to modify any of the starter code in any manner you wish.

HIGH POINT UNIVERSITY

9

Id_table_entry

```
#include <iostream>
#include <string>

#include "token.h"
#include "lille_type.h"
#include "lille_kind.h"
using namespace std;

class id_table_entry {
private:
    token* id_entry;
    int lev_entry;
    int offset_entry;
    lille_kind kind_entry;
    bool trace_entry;
    lille_type typ_entry;
    int i_val_entry;
    float r_val_entry;
    string s_val_entry;
    bool b_val_entry;
    id_table_entry* p_list_entry;
    int n_par_entry;
    lille_type r_ty_entry;
```

```
public:
    id_table_entry();
    id_table_entry(token* id,
        lille_type typ = lille_type::type_unknown,
        lille_kind kind = lille_kind::unknown,
        int level = 0,
        int offset = 0,
        lille_type return_tipe =
                    lille_type::type_unknown);
```

HIGH POINT UNIVERSITY

10

Id_table_entry

```
void trace_obj (bool trac);
bool trace();
int offset();
int level();
lille_kind kind();
lille_type tipe();
token* token_value();
string name();
int integer_value();
float real_value();
string string_value();
bool bool_value();
lille_type return_tipe();

void fix_const(int integer_value = 0,
               float real_value = 0,
               string string_value = "",
               bool bool_value = false);
void fix_return_type(lille_type ret_ty);
void add_param(id_table_entry* param_entry);
id_table_entry* nth_parameter(int n);
int number_of_params();
string to_string();
};
```

HIGH POINT UNIVERSITY

11

Handling predefined routines

- When we call the compiler, we begin a scope level 0.
- This level is reserved for predefined entities such as the functions:
  - Int2real
  - Real2int
  - Int2string
  - Real2string
- Each of these takes one parameter as an argument.
- Each of these can be overridden by the *lille* programmer who can write a function of the same name and mask (hide) the predefined version.

HIGH POINT UNIVERSITY

12

Handling predefined routines

- These are handled as soon as the parser is called for the first time, and before any tokens are read from the source file by the scanner.

```
token* predefined_func;
token* argument;
symbol* predefined_sym;
id_table_entry* func_id;
id_table_entry* param_id;

predefined_sym = new symbol(symbol::identifier);
predefined_func = new token(predefined_sym, 0, 0);
predefined_func->set_identifier_value("INT2REAL");
func_id = id_tab->enter_id(predefined_func, lille_type::type_func,
      lille_kind::unknown, 0, code->current_code_ptr(), lille_type::type_real);
```

HIGH POINT UNIVERSITY

13

Handling predefined routines

```
// Create a token for the parameter of the predefined function
argument = new token (predefined_sym, 0, 0);
argument->set_identifier_value("__int2real_arg__");
param_id = new id_table_entry(argument,
      lille_type::type_integer,
      lille_kind::value_param,
      0,
      0,
      lille_type::type_unknown);
// Associate parameter with the function.
func_id->add_param(param_id);
```

HIGH POINT UNIVERSITY

14

Entering a scope region

- Whenever a block is entered we increment the scope level:
  - `id_tab->enter_scope();`
- Whenever we exit a block, we decrement the scope level:
  - `id_tab->exit_scope();`
- Procedure and function names are at a different scope level to the arguments and local variables etc.

15

Entering a scope region

Enter scope level

Scope level: 0

```
program prog3 is
   number, newNumber: integer;
   procedure double(n1: value integer; n2: ref integer) is
   begin
      n2 := n1 + n1;
   end;
begin
   write ("Enter a number: ");
   read (number);
   double(number,newNumber);
   writeln ("Number doubled is " & int2string(newNumber));
end;
```

Scope level: 1

Scope level: 2

Exit scope level

16

Scope regions

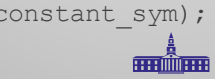
- The current scope region determines which level of the id_table new identifiers are inserted into.
- When searching for an identifier, start at the current scope level and search toward scope level 0 to find the version of the identifier that is in scope.
- An identifier can only be present once at any particular scope level.
- Identifiers with the same name mask identifiers at the lower scope levels.

HIGH POINT UNIVERSITY

17

Variable declarations

```
… if (this token is an identifier) { // variable or constant declaration
   do {
      if (this token is an identifier) {
         add ident token to list to add to ID-table when all info known.
         scan->get_token();
         count++;
      }
      comma_found = scan->have(symbol::comma_sym);
      if (comma_found)
         scan->must_be(symbol::comma_sym);
   } while (comma_found);

   scan->must_be(symbol::colon_sym);
   if (scan->have(symbol::constant_sym)) {
      const_decl = true;
      knd = lille_kind(lille_kind::constant);
      scan->must_be(symbol::constant_sym);
   }
   else
```



HIGH POINT UNIVERSITY

18

Variable declarations

```
knd = lille_kind(lille_kind::variable);
ty = tipe(fsys);
if (const_decl) {
   scan->must_be(symbol::becomes_sym);
   sym = scan->this_token()->get_symbol()->get_sym();
   switch (sym)
   {
   case symbol::real_num:
      r_const = scan->this_token()->get_real_value();
      if (!ty.is_type(lille_type::type_real))
         error->flag(scan->this_token(), 111);
         // Const expr does not match type declaration.
      break;
   case symbol::integer:
         …
   }
   scan->get_token();
}
```

HIGH POINT UNIVERSITY

19

Variable declarations

```
ptr = list;
while ((ptr != NULL) and (count >0)) {
   id = id_tab->enter_id(ptr->val, ty, knd, id_tab->scope(),
            current_offset, lille_type::type_unknown);

   if (const_decl)
      id->fix_const(i_const, r_const, s_const, b_const);
      // ****  fix constant
   ptr = ptr->next;
   count--;
}
}
```

HIGH POINT UNIVERSITY

20

Variable declarations

- Must create a list of tokens so that they can be inserted into the symbol table when all the information is known.
- After a constant is inserted into the symbol table, need to fix the entry to include the value of the constant.
- Note that we must tack the type of the entity, the kind of entity we are dealing with and perform appropriate type checking as defined by the sematic rules for *lille*.

HIGH POINT UNIVERSITY

21

Procedures and functions

```
… if (scan->have(symbol::function_sym)) {
    scan->must_be(symbol::function_sym);
    if (scan->have(symbol::identifier))
       id = id_tab->enter_id(scan->this_token(), lille_type::type_func,
          lille_kind::unknown, id_tab->scope(), code->current_code_ptr(),
          lille_type::type_unknown);
    // Need to fix the return_type later once it is known.
    block_name = scan->this_token();
    scan->must_be(symbol::identifier);
    id_tab->enter_scope();
    if (scan->have(symbol::left_paren_sym)) {
       scan->must_be(symbol::left_paren_sym);
       param_list(fsys + symbol::right_paren_sym, id);
       scan->must_be(symbol::right_paren_sym);
    }
```

HIGH POINT UNIVERSITY

22

Procedures and functions

```
scan->must_be(symbol::return_sym);
if (scan->have(symbol::integer_sym))
{
   id->fix_return_type(lille_type::type_integer);
   scan->must_be(symbol::integer_sym);
}
else if …
else
   error->flag(scan->this_token(), 108);    // type expected.

scan->must_be(symbol::is_sym);
block(fsys + symbol::semicolon_sym + symbol::identifier,
         block_name, return_count);
id_tab->exit_scope();
current_offset = old_offset;
```

HIGH POINT UNIVERSITY

23

Procedures and functions

```
if (return_count == 0)
   error->flag(scan->this_token(), 109);
   // Functions must have at least 1 return statement.
if (scan->have(symbol::identifier)) {
   if (block_name->get_identifier_value() !=
         scan->this_token()->get_identifier_value())
      error->flag(scan->this_token(), 107);
      // Identifier must match name of the block.

   scan->must_be(symbol::identifier);
}
}
```

HIGH POINT UNIVERSITY

24

Parameters

- Notice that the code for implementing procedures and functions contained:

```
if (scan->have(symbol::identifier))
   id = id_tab->enter_id(scan->this_token(), lille_type::type_func,
      lille_kind::unknown, id_tab->scope(), code->current_code_ptr(),
      lille_type::type_unknown);
block_name = scan->this_token();
scan->must_be(symbol::identifier);
id_tab->enter_scope();
if (scan->have(symbol::left_paren_sym)) {
   scan->must_be(symbol::left_paren_sym);
   param_list(fsys + symbol::right_paren_sym, id);
   scan->must_be(symbol::right_paren_sym);
}
```

- Param_list is responsible for entering the parameters of the function or procedure into the id_table.
- However we must also keep track of how many parameters a function or procedure has and what their types etc are.
  - Param_list must also handle this.

HIGH POINT UNIVERSITY

25

Parameters

- Inside param_list:

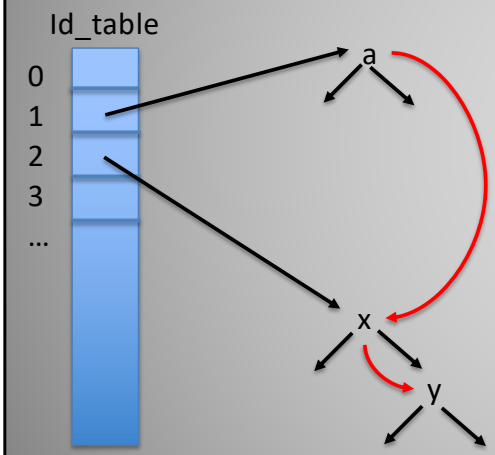
```
 Build a list of all the parameters encountered.
 Note the kind of parameter (value|ref)
 Note the type associated with the parameters
 Insert the parameters into it_table and associate them with
     the function of procedure (id was passed to the
     param_list subparser)

while ((ptr != NULL) and (count > 0)) {
   param_id = new id_table_entry(ptr->val, ty, knd, id_tab->scope(),
                    current_offset, lille_type::type_unknown);
   id_tab->add_table_entry(param_id);
   id->add_param(param_id);
   count--;
   if (ptr->next != NULL)
      ptr = ptr->next;
}
```

HIGH POINT UNIVERSITY

26

Parameters

- `procedure A (x, y: value integer) is …`



- The black lines show how each name is accessed through the id_table.
- The red lines show how the procedure a knows it has 2 parameters and that the first is x and the second if y.
- Type information etc is able through the id_table entries for x and y.

HIGH POINT UNIVERSITY

27

Assignments

```
case symbol::identifier:
   tok = scan->this_token();
   id_info = id_tab->lookup(tok);
   scan->must_be(symbol::identifier);
   if (scan->have(symbol::left_paren_sym)) {
      // Better be a procedure!
      scan->must_be(symbol::left_paren_sym);
      if (!id_info->tipe().is_type(lille_type::type_proc))
         error->flag(tok, 90);
         // Identifier must be a procedure name in this context.
      ident_list(fsys + symbol::right_paren_sym, id_info);
      scan->must_be(symbol::right_paren_sym);
   }
   else if (scan->have(symbol::becomes_sym)) {
      // Its an assignment…
```

HIGH POINT UNIVERSITY

28

Assignment

```
   if (!(id_info->kind().is_kind(lille_kind::variable)
         or id_info->kind().is_kind(lille_kind::ref_param)))
      error->flag(tok, 85);
      // Identifier is not assignable. Must be a variable
      // or reference parameter.
   scan->must_be(symbol::becomes_sym);
   if (expr_first_symbols.member(
         scan->this_token()->get_symbol()->get_sym())) {
      lille_type exp_ty = expr(fsys);
      // check LHS and RHS have matching types
      if (!id_info->tipe().is_equal(exp_ty)) {
         error->flag(scan->this_token(), 93);
         // LHS and RHS of assignment are not type compatible.
      }
```

HIGH POINT UNIVERSITY

29

Assignment

```
   if ((id_info->kind().is_kind(lille_kind::ref_param)) or
         (id_info->kind().is_kind(lille_kind::variable)))
      trace_write(id_info, true);
   else
      error->flag(scan->this_token(), 92);
      // String or expression expected.
}
else
   error->flag(tok, 91);   // Identifier illegal in this context.
break;
```

HIGH POINT UNIVERSITY

30

Assignment

- Notice that the code to implement assignment checks the type compatibility by having the call to the expression subparser return the determine and return the type of the expression handled.
- This is then compared to the known type of the LHS of the assignment statement to ensure type compatibility.

HIGH POINT UNIVERSITY

31

Handling an identifier in an expression

```
case symbol::identifier:
     tok = scan->this_token();
     id = id_tab->lookup(tok);
     return_type = id->tipe();;
     knd = id->kind();
     …
return return_type;
```

- The expression subparser deals with recovering the type from the id_table when an identifier is encountered and determines the type of the expression calculated.

HIGH POINT UNIVERSITY

32

Handling an identifier in an expression

- For example, many languages perform automatic type conversion:
  - `r := 3.0 + 4;`
  - The result of adding a real number to an integer is a real number.
  - The RHS evaluates to a real number.
  - The type associated with `r` should be a real.
- Read the sematic rules provided in the *lille* handout to determine what *lille* needs to implement.

HIGH POINT UNIVERSITY

33

Static semantic analysis

- Static semantic analysis refers to all the semantic rules that can be checked and enforced at compile time.
  - As opposed to dynamic semantic analysis which is the semantic analysis that can only be performed at run-time.
- All the types rules in *lille* can be enforced at compiler time.
  - The id_table is used heavily to enforce the static semantic rules of programming languages,

HIGH POINT UNIVERSITY

34


HIGH POINT UNIVERSITY
Extraordinary Education
Inspiring Environment
Caring People
MCMXXIV

35