Program 6

Symbol Tables for Type Checking

Kim Buckner

University of Wyoming

Nov 24, 2020

Current starting point

- Your programs should be building a syntax tree for the input code.
- And it should be building symbol tables.
- You may or may not have done some type checking.
- Now we finish it up.

The symbol tables.

- There are as many ways to this as there programmers but some are easier than other.
- As discussed earlier, we need to be able to determine scope.
- For this language we have essentially two levels
  - block scope similar to C++ and Java and
  - "global" scope.

Global Scope

- This is really only the primitive data types and the user-defined data types (classes).
- Whenever the programmer adds a class, that class name becomes a type.
- When we type check then, we must know that some type exists.
- Which means those new type names must be stored someplace accessible at all times.

(more . . .)

- You all know that I really dislike true global variables in your programs.
- That is because you
  - overuse them,
  - misuse them, and
  - have real problems debugging after you do both of the above.

(more . . .)

- That being said if there is a single reference to the root (type?) symbol table, that would not be a totally evil thing.
- Suppose that I said you could not have ANY global variables, what would you do then?

Types

- There are two issues in type checking
  - Attributes of "basic" types: storage size and allowed operations
  - Attributes of user-defined types: storage size, allowed operations, characteristics, components.
- This all leads one to thinking that "types" are different from "symbols".

FORTRAN attributes

- These are "modifications" to the basic type or user-defined type.
- The attributes are applied to the variables but modify the variable's type.
- Consider the POINTER attribute.
- This means that the variable can "point-to" a TARGET or be allocated.

Other languages

- Take C, and the declaration `int *p`
- This means that **p** is "type pointer-to-integer".
- Or C++ and `B &a` in a function parameter list.
- This means **a** is a "reference to class B object".

The question

- How do we store these attributes?
- We can store them as separate elements of the symbol table entries.
- Or we can create/modify the type name to represent them.
- For instance `int*` can be *int_pointer*

(more . . .)

- Likewise, `int[][]` can be *int_array_2* where 2 is the number of dimensions.
- The size of this array is only significant to the variable.
- And only needed when the code to allocate memory for the array is emitted.

One or many

- That is the question usually asked for this project.
- If we only have one symbol table, how do we differentiate between names?
- That translates to how do we determine scope?
- Any ideas?

(more . . .)

- If we choose not to develop/borrow a name mangling scheme we can
  - leave the names alone,
  - store the attributes as separate elements, and
  - create multiple tables.
- Or maybe
  - leave the names alone,
  - mangle the types, and
  - create multiple tables.
- Or . . .

Many tables

- Then we create a tree of tables that mirrors the syntax tree.
- For each element that defines a scope, create a new table.
- Link each such table with its "parent" or enclosing scope.
- In the *type* table there should be a link from each class type to its scope table.

The tables

- All the symbol tables could be the same.
- But do we really want that?
- Consider a **class** table, one for a **method**, and one for a **block**.

A block table

- Here I mean "block of code."
- Regardless of whether it is part of a loop or the code for a method, blocks have two parts:
  - variable declarations and
  - statements.

A method table

- Methods have three parts:
  - parameters,
  - variable declarations, and
  - statements
- I assume here that the return type is part of an entry in a class table.

A class table

- Classes have three parts:
  - variable declarations,
  - constructors, and
  - methods.
- Do the methods require something different from the constructors?

(more . . .)

- You should have completed all of this.
- And fixed all the problems from other iterations.
- Will get Program 5 graded asap.
- Program 6 will be available on the 25^{th}.

Program 6

- Due: December 11 at midnight.
- Cannot be late, no exceptions, no problems, no dogs or aliens eating the homework.

main()

- One and only class will have a method "main()."
- This will be the first check you will make after the code is processed, if no main(), do not continue. Give a simple error message.
- It cannot have parameters.
- It must have an *int* or *void* return type.

this

- It must be the FIRST in a dotted name.

```
this.a.b.val
```

- It CANNOT be

```
a.b.this.c
```

- "this" always refers to the current class, that is its type.

(more . . .)

- Means that

  ```
  b = this;
  ```

  is only true if “b” is of the same type as the current class.

Operators

- Arithmetic and comparison operators require INT operands.
- Boolean expressions return INTs.
- So conditional operators && and || only take INT operands.
- == can compare a reference type to another reference type OR to **null**. Otherwise only INT operands.

new

- Returns a reference to some type.
- If allocating an array, the number of brackets must match.
- Only the rightmost [] can be empty.

(more . . .)

This is valid.

```
int [][] a;

a = new int [5][];
a[3] = new int[10];
```

null

- This is a reference.
- It can be used anyplace where a reference type is expected as an *r-value*.
- It cannot be compared to an INT.

read and print

- *read()* is an operator that returns an integer.
- *print()* is an operator that takes 0 or more integer arguments.
- The *print()* return type is **void**.

Constructors

- Constructors cannot be called directly.
- Only as part of a **new** statement.

```
foo a;


a = new foo(x,y); // good
a(x,y); // not allowed
foo(x,y); // not allowed
```

return

- No constructor or method is required to have a return statement.
- Not even if the method has a non-void return type.
- BUT the optional part MUST match the return type specified when the method was declared.