CS536: Homework 8

Sahit Mandala

November 17, 2015

Problem 1

In this case, we need to add rules to allow the grammer to accept structs and other typedefs as possible "types" in the typedef definition. So we should add the following rules:

typedef $\rightarrow$ TYPEDEF STRUCT ID ID SEMICOLON //accepts structs in typedefs declarations

typedef $\rightarrow$ TYPEDEF ID ID SEMICOLON //accepts typedefs in typedef declarations

Problem 2

a)

Each symbol table entry will be indexed on name. As for the values within a specific entry, the table will store (either as a field, boolean, or tailored symbol structure) whether the name is associated with a variable, primitive type, a struct, or a typedef. If a primitive type, we just store which type (e.g. int, boolean). For structs, we can store a internal symbol table with all the associated fields within the struct, mapping field names to symbol entries. Typedefs should store a pointer to the symbol they are associated with (e.g. for typedef struct Pair Point, we have a pointer to the Pair's symbol object)

b)

First, T should be checked to verify that it is either a valid primitive type, an already defined struct type, or an already defined typedef. Primitives can be checked directly, but the structs and typedefs can be checked in the symbol table on a query; also, T must be formated as "struct ID" when typedefing a struct type (so typedef Pair Point is not acceptable).

xxx should be checked globally in the symbol table to make sure it is not already associated with any other symbols (variables, functions, structs, typedefs, etc) to avoid duplicate declarations.

If these conditions are met, we should create a new symbol table entry on xxx with stores the fact that xxx is a typedef as well as a pointer to what it extends (e.g. Point should point to the struct Pair's symbol table entry)

c)

First, T should be checked to verify that it is either a valid primitive type, an already defined struct type, or an already defined typedef; Primitives can be checked directly, but the structs and typedefs can be checked globally in the symbol table on a query. Also, T must be formated as "struct ID" when typedefing a struct type (so typedef Pair Point is not acceptable).

xxx should be checked globally in the symbol table to make sure it is not already associated with any other symbols (variables, functions, structs, typedefs, etc) to avoid duplicate declarations.

If these conditions are met, we should create a new symbol table entry on xxx with stores the fact that xxx is a variable as well as a field/ptr to its associated type T symbol (e.g. int, Point, struct Pair, etc).

d)

The xxx can directly be looked up in the symbol table globally to verify that xxx has been declared/defined previously. If the use is within a typedef statement (e.g. typedef money dollars is a use of typedef money), then we should verify that money is a typedef using the symbol table. Similarly, if xx is used as a struct in a declaration, as we mentioned in part b/c, we should verify it exists and is a struct. More relevently, if xxx is in any other usage, we need to verify that it is an actual variable (again, checking its associated symbol in the symbol table).

Note: the notation "type: ID" is pseudocode for a ptr/structure for the type's sym object; so "type: moreDollars" means we store a reference to moreDollar's sym object TypedefSym{ type: dollars})

```
MonthDayYear: StructSym{ fields:{
          month:VarSym{type: int},
          day:VarSym{type: int},
          year:VarSym{type: int}
}}
date : TypedefSym{type: MonthDayYear}
today: VarSym{type: date}
dollars: TypedefSym{ type: int}
salary: VarSym{type: dollars}
moreDollars: TypedefSym{ type: dollars}
md: VarSym{type: moreDollars}
d: VarSym{type: int}
```