Keenan Thompson
15 March 2017
CS 536 - Professor Thomas Reps

Homework 7

Question 1

First, typedefs must allow typedefs and structs:

```
typedef -> TYPEDEF STRUCT ID ID SEMICOLON
typedef -> TYPEDEF ID ID SEMICOLON
```

Now for typedef variable usages, assuming they can be used as variable types, function parameters and function return values (the assignment did not say otherwise):

```
varDecl -> ID id SEMICOLON         // VarDeclNode of type TypedefNode
formalDecl -> ID id                // FormalDeclNode of type TypedefNode
fnDecl -> ID formals fnBody        // FnDeclNode of return type TypedefNode
```

Question 2

a. The key would be the typedef’s name. The value stored in the table should be a custom symbol structure for typedefs. It should contain a reference/pointer to the type or symbol with which they are associated.

   For example, “typedef struct Pair Point;”. The key would be “Point” and the symbol would be a typedef symbol containing a reference to the struct “Pair”.

b. T must be checked to ensure it is a valid type. If it is a primitive, it can be checked directly (throwing an error if it is void). Otherwise, perform a symbol lookup. If nothing is found, or something other than a struct or typedef is found, we have an error.

   xxx must be checked in the symbol table to ensure it is not already declared somewhere.

Finally, all conditions have passed. Thus, we should add “xxx” to the symbol table as a typedef holding a pointer to the type or symbol with which it is synonymous.

c. Assuming primitive types, structs and typedefs can be used in a variable, function or parameter declaration. Also assuming that T is not necessarily a typedef...

   T must be checked to ensure it is either a primitive, struct or typedef. Just like in (b), primitives can be checked directly, otherwise perform a global symbol lookup.

   xxx must also be checked to ensure it hasn’t already been declared.

   Finally, all conditions have passed. Add xxx to the symbol table, pointing it to the symbol found.

d. First check to see if xxx is in the symbol table. If not, we have an error because we don’t know what xxx is. Nothing else needs to be done (assuming the id is checked for multiple declarations) because we now can associate the id with the type found in the symbol table.

```
typedef int money;
money y; // money is now in the symbol table, so put y -> money
         // in the table
```

<u>Question 3</u>

```
class TypedefSym extends SemSym {
    private SemSym ref;  // Reference to this typedef’s synonym.
    …                    // It could be a primitive, a struct or a typedef
}
```

Symbol Table:

name	**value**
MonthDayYear	StructDeclSym
date	TypedefSym(ref=MonthDayYear)
today	SemSym(type=date)
dollars	TypedefSym(ref=int)
salary	SemSym(type=dollars)
moreDollars	TypedefSym(ref=dollars)
md	SemSym(type=moreDollars)
d	SemSym(type=int)

With this structure, we can find the original type of “md”, for example, by recursively checking it’s type.

```
md = moreDollars (typedef) -> dollars (typedef) -> int
thus, md is an int
```