CSE 131 – Compiler Construction

Discussion 2: Project 1 – Phase 1 & 2

4/10/2009

4/13/2009

Overview

- Functions
- Constant Folding
- Aliases (Typedefs/Structdefs)
- Topics/Questions you may have

Functions!

- First, some fundamental points:
  - We are writing a static translator, not an interpreter.
  - Once we finish the function declaration, including the body, we're done. We don't need to remember the code in the body, since we'll just be spitting out assembly code (Project 2).
  - A function call will basically boil down to an assembly "call foo" type instruction, once all type checking is complete.

Functions

- For Project 1, we will need to:
  - Check the function call against the function declaration to ensure argument symmetry.
  - Check the body of the function, type checking the same way we check statements in main.
  - Check the return logic of the function.
  - Allow function overloading (Extra Credit)

FuncSTO

- In order to do most of these checks, we will need to store some information about the function. FuncSTO is designed for this!
- Things that should be stored:
  - The Return Type.
  - All parameter information:
    - Total number of parameters
    - Type of each parameter, including whether pass-by-reference or not

Function Declaration

```
FunctionDecl ::=  T_FUNCTION T_COLON ReturnType T_ID:_2
                {:
                    ((MyParser) parser).SaveLineNum ();
                    ((MyParser) parser).DoFuncDecl_1 (_2);
                :}
                T_LPAREN OptParamList:_3 T_RPAREN
                {:
                    ((MyParser) parser).DoFormalParams (_3);
                :}
                CodeBlock
                {:
                    ((MyParser) parser).DoFuncDecl_2();
                :}
    ;
```

Function Declaration

```
void DoFuncDecl_1 (String id) {
        if (m_symtab.accessLocal (id) != null) {
            m_nNumErrors++;
            m_errors.print (Formatter.toString(ErrorMsg.redeclared_id, id));
        }

        FuncSTO     sto = new FuncSTO (id);
        m_symtab.insert (sto);            // Inserted into current scope

        m_symtab.openScope ();            // New scope opened
        m_symtab.setFunc (sto);           // Current Func we're in is set
}
```

Function Declaration

```
void DoFuncDecl_2 () {
        m_symtab.closeScope ();       // Close scope (pops top scope off)
        m_symtab.setFunc (null);      // Says we're back in outer scope
    }
```

Function Declaration

```
function : bool foo (float a, float b, float &c) {
  bool x;   // local variable
  x = a > b;
  x = (a + c) <= 2;
  return x;
}
```

- In this example, we want to create a FuncSTO with the name "foo", set its return type to boolean, set its param count to 3, and remember the parameters are: value float, value float, and reference float).
- Furthermore, we want to insert the VarSTO for a, b, and c into the SymTable so they are available for the body of the procedure.

Function Call

- Now that we have a FuncSTO in the SymTable, and type-checked the function declaration, we are ready to call it.

  foo(1, 2, 3.3);
- Given the above call, there would be an error since '3.3' is not addressable and cannot be sent to a reference parameter.

Function Call

- So, when we call a function, we want to get the FuncSTO from the SymTable and check its parameters with the arguments.
- Consider making a Vector of some new object (you create it how you want) that holds the parameters. When you call the function, compare the two vectors.
  - Important design choices!!!
  - Remember function overloading for extra credit – how to uniquely encode the name with the parameters and know which one to get from the Symbol Table.

Functions Return Types

- The type of the return expression needs to be checked against the type declared as the function's return type
- When a function call is used in an expression, it behaves like any other ExprSTO. You need to use the function's return type for semantic checking
- Functions are also special in that they are the only object that can have a "void" type
  - The "void" type is not equivalent or assignable to anything (including itself), so any use of an object of "void" type in an expression should result in an error

Constant Folding

- All ConstSTO objects will need to have a value actually stored in the STO
- The steps for constant folding are the following:
  - Verify semantic validity (no errors present in the expression)
  - If all operands are ConstSTOs, the resulting object is a ConstSTO, and its value is the result of the operation

Constant Folding

```
int x = 1, y = 9;
const int a = 3;
const float b = 7;

x + y  → ExprSTO [type: int]
x + a → ExprSTO [type: int]
a + b → ConstSTO [type: float; value: 10.00]
a == b → ConstSTO  [type: bool; value: false]
```

Typedefs/Structdefs

- We suggest you use the TypedefSTO class provided for typedefs and structdefs.

- TypedefDecl needs to have same changes as just described in VarDecl to associate a Type with the STO.

Typedefs

```
typedef float T1;  // note: must be all uppercase
typedef T1 T2;
typedef T2 T3;

T3 x;
float y;
```

Typedefs/Structdefs

- Typedefs provide a way for users to define another name for a type (an alias).

- Structdefs provide a way for users to define new types.

- So, what needs to be done?
  - We mainly need to store 2 things:
    - The name of the alias
    - The base type it represents

Typedefs

- So, let’s look at this example more closely:

```
typedef float T1;
typedef T1 T2;
typedef T2 T3;

T3 x;
float y;
```

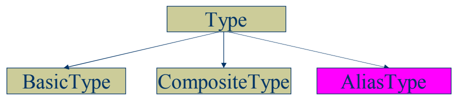
- In this case, “x” has a alias name of “T3”, and an underlying type of “float”. Notice we don’t need to remember the intermediate aliases!

So, How Do We Implement It?

- Since typedefs are referred to by an identifier (their name), they will eventually need to get on the Symbol Table.
- So, one way is to have some kind of STO for Typedefs → TypedefSTO
  - Since it extends your base STO, it will have the Name and Type fields you need to store your information.

Implementation Idea #2

- Another possible implementation can be to add an AliasType to your Type hierarchy:



Trade-offs

- The first method (using TypedefSTO) fits nicely into the given code, relying on the SymbolTable and STO hierarchy:

```
void DoTypeDecl (String id, Type baseType)
{
        if (m_symtab.accessLocal (id) != null)
        {
                m_nNumErrors++;
                m_errors.print (Formatter.toString(ErrorMsg.redeclared_id, id));
        }

        TypedefSTO      sto = new TypedefSTO (id, baseType);
        m_symtab.insert (sto);
}
```

Trade-offs

- The second method (using AliasType), will be more consistent with the Type hierarchy, but will still rely on some STO to store the AliasType onto the Symbol Table.
- Will result in the following lookup, which has an extraneous level of indirection:
  - STO → AliasType → Base Type

A Third Possibility

- Modify the Symbol Table in order to allow an AliasType to be stored and accessed similar to the current STO approach.
- One can overload the “access” and “insert” methods to also take an AliasType, and automatically wrap that into an STO and call the corresponding method as normal.
  - This just hides the indirection

Typedefs

- As you can see, there are numerous ways to handle the typedefs. Think about what you are most comfortable implementing and what can be done to make your program work before the deadline.

More Type Checking

- Remember:
  - All types use structural equivalence (except structs)
  - All typedefs/structdefs use name equivalence to resolve down to the lowest-level type
  - Structs-level operations (e.g. assignment, equality, and inequality) use name equivalence. All structs are defined with structdef

Alias Equivalence

```
typedef int FOO;
typedef FOO BAR;
typedef BAR BAZ;
int x;
BAZ y;
function : int main() {
  x = 5; // OK
  x = y; // OK by name equivalence
  return 0;
}
```

Alias/Struct Equivalence

```
structdef R1 { float a, b; };
structdef R2 { float a, b; };
typedef R1 R3;
R1 x;
R2 y;
R3 z;

x ⇔ y?  NO
x ⇔ z?  YES
y ⇔ z?  NO
```

What to do Next!

1. Finish up Phase 1.
2. Write more test programs.
3. Start Phase 2.
4. Come to lab hours and ask questions.

Topics/Questions you may have

- Anything else you would like me to go over now?
- Anything in particular you would like to see next week?