Discussion 2: Project 1 – Phase 2
04/12/2013
04/15/2013
Overview

- Constant Folding
- Aliases (Typedefs/Structdefs)
- Arrays
**CONSTANT FOLDING**

- All ConstSTO objects will need to have a value actually stored in the STO

- The steps for constant folding are the following:
  1. Verify semantic validity
     (no errors present in the expression)
  2. 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 in VarDecl to associate a Type with the STO.
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
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”. Note that you do not 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.

- You do this by using some kind of STO for Typedefs: the TypedefSTO
  - Since it extends your base STO, it will have the Name and Type fields you need to store your information: the alias name and the base type.
So, How Do We Implement It?

- Notice that the only thing that makes an alias type different from its base type is the name that either one is identified by.

- So you can use the regular child classes of the parent Type class, and just set the type’s name to be the alias name (instead of “int”, “float”, etc.).

- This approach will require just a little more thought for dealing with typedefs of typedefs.
MY_PARSER_CODE_FOR_TYPEDEF

```java
void DoTypedefDecl (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);
}
```
MORE TYPE CHECKING

- Remember:
  - All types use structural equivalence (except structs)
  - Structdefs use strict name equivalence
  - Typedefs use loose 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
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;
}
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
STRUCT DECLARATION

- Syntax:

  ```
  structdef MYSTRUCTALIAS {
    int foo;
    float baz;
    MYSTRUCTALIAS* nextPtr;
    bool foo;
    function : float getBaz() { return this.baz; }
  }
  // This is the struct definition (similar to a typedef)
  ```

- What to check in the declaration:
  - Check for duplicate fields (field mySVar.foo in this case)
  - If a field is duplicated multiple times, an error is reported for each duplicate instance
**Struct Usage**

- **Syntax:**
  
  `myStruct.myField`

- **What to check:**
  
  - `myStruct` must be of type `StructType`
  
  - `myStruct` must contain the field `myField`, in this case
  
  - After checking, your result will be of the type of `myField`. 
FURTHER ANALYSIS OF STRUCTS

structdef REC {
    float a, b;
};
REC myRec1, myRec2;

function : void main() {
    myRec2 = myRec1;     // OK, myRec2 mod l-val
    myRec2.b = 3.6;      // assign 3.6 into b field
    myRec2.a = myRec2.b; // assign b field into a field
}
ARRAY DECLARATION

- Syntax:
  ```
  type[index] varName;         // (int[4] myArray;)
  ```

- What to check in the declaration:
  - Index is an int
  - Index > 0 (known at compile time → ConstSTO)

- Note that the only way to have an array of array is by using a typedef alias for the inner array (you can only use one [ ] type modifier per variable/type declaration)
Array Usage

Syntax:

myArray[index];
myArray[index][index];

- What to check:
  - The designator before the [] must be of Array or Pointer type
  - Index must be equivalent to int
  - If the index is a constant, you must check the bounds (this only applies when the designator before the [] is an Array type)
How can one encapsulate the information from the array declaration for later use?

Remember the Type Hierarchy:
- One possibility is to store information such as elementType, dimensionSize.
- In order to do this, you have to add these fields into the ArrayType definition and provide ways to set and read the information from them.
int[20] myArray;
int myInt;
int * myIptr;
const int c = 5;

function : void main() {
    myArray[5+c] = myArray[6-c]; // bounds check
    myArray[myInt] = 15; // no bounds check here
    myArray = 10; // error, non-modifiable L-val
    myIptr = myArray; // OK, since array id is ptr
    myIptr[c] = 100; // no bounds check since ptr
}
Arrays as Parameters

- An array parameter is **passed-by-reference**

  ```
  int sumArray( int[5] & intArray) { ... }
  ```

- A pointer parameter is **passed-by-value**

  ```
  int sumPtr( int* intPtr ) { ... }
  ```
Array Initialization

OptInit ::=   T_ASSIGN Expr
             |   T_ASSIGN T_LBRACE ArrElemsList T_RBRACE
             |   /* empty */
             ;

ArrElemsList ::=   Expr
                  |   ArrElemsList T_COMMA Expr
                  ;
Array Initialization

- Different from array assignment!!
- Things to check:
  - Elements should not exceed declared size
  - Elements must be constant expressions
  - Elements must be assignable to the base type

```c
int[10] iArray = {1,2,3,4,}; // OK

int x;
int[2] arr1 = {x, 1}; // FAIL x is not a constant STO
```
Array Initialization

- Different from array declaration!
- Things to check:
  - Elements should not exceed declared size
  - Elements must be constant expressions
  - Elements must be assignable to the base type

```java
int[10] iArray = {1,2,3,4,}; // OK
```

```java
int x;
int[2] arr1 = {x, 1};  // FAIL x is not a constant STO
```
WHAT TO DO NEXT!

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