Discussion 2

CSE 131

overview

- constant folding
- aliases (typedefs/structdefs)
- arrays

constant folding

- all ConstSTOs must have their value stored inside them
- to constant fold
  a. verify semantic correctness (no errors)
  b. if all operands are ConstSTOs, the result is a ConstSTO and the 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

- typedefs provide a way to define another name for a type (alias)
- structdefs provide a way for users to define new types
- probably want to use TypedefSTO for both typedefs and structdefs

typedefs

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

T3 x;
float y;
```

- x has an alias name of T3 with an underlying type of float
- you do not need to remember the intermediate aliases

implementation

- what does a typedef need?
  - the name of the alias
  - the base type it aliases
- typedefs are referred to by their name
  - this means they eventually need to be on the symbol table

typedef code in MyParser

```
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);
}
```

a dilemma

- what if I perform an operation with an alias and its base type?
  - what do I print in the error message?
  - what do I use as the type of the result?

a dilemma

- for all operations
  - always use the alias name in the error message
  - always use the base type as the result
- this duality poses an interesting problem
  - you mean I’m going to have to check everywhere whether I have an alias or not and ask for the base type if it is?
  - so pain much annoy very ick wow

a dilemma

- what if there was a way to do the right thing™ without having to check whether we're dealing with an alias or not?
  - think object oriented

type checking

- all types use structural equivalence (except structs)
- structs use strict name equivalence
- typedefs use loose name equivalence to resolve down to the lowest-level type

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 equivalent to y? nope
// x equivalent to z? yep
// y equivalent to z? nope
```

struct declaration

```
structdef MYSTRUCT
{
     int foo;
     float baz;
     MYSTRUCT* nextPtr;
     bool foo;
};
```

- what do I check in the declaration?
  - duplicate fields
    - if fields are duplicated multiple times, report an error for each duplicate instance
  - not using the current struct type directly in the current struct declaration (pointers to current struct type ok)

struct usage

```
myStruct.someRandomField
```

- what do I check?
  - myStruct must be a StructType
  - myStruct must contain the field (someRandomField, in this case)
- resulting expression is the type of someRandomField

array declaration

```
type[index] varName;
typedef type[index] ALIASNAME
```

- what do I check?
  - index is an int
  - index > 0 (must be known at compile time)
- only way to have an array of an array type is by using a typedef alias for the inner array

array usage

```
y = myArray[index];
x = myArray[index][index];
```

- what do I check?
  - the designator before the [] must be an array or pointer type
  - index must be equivalent to int
  - if index is a constant and the designator is an array, you must check the bounds

implementation

- need to store the information from the array declaration to use later
- remember the type hierarchy

array examples

```
int[20] myArray
int myInt;
int *myPtr;
const int c = 5;

function : void main()
{
     myArray[5 + c] = myArray[6 - c];      // bounds check
     myArray[myInt] = 15;                  // no bounds check
     myPtr = myArray;                      // ok, since array id assignable to ptr
     myPtr[c] = 100;                       // no bounds check since ptr
}
```

arrays as parameters

- an array parameter is passed by reference
  - int sumArray(int[5] & array) { … }
- a pointer parameter is passed by value
  - int sumPointer(int * intPtr) { … }

next steps

- finish phase 2
- write test programs
- come to lab hours and ask questions