6.035 Infosession 2

Fall 2014

Project 1: Done

So far, we have 7 teams:

- Scala: 4 teams
- Java: 2 teams
- Haskell: 1 team

Team repositories have been created

Project 2: Semantic Checker

Input: Parse tree of the program produced by the parser from Project 1

Construct internal data structures:

- Construct intermediate code representation
- Construct symbol table

Run checking procedures

- Make sure that the program is legal
- Use the generated IR and symbol table

Output: Decide whether a program is legal

Intermediate Representation Summary

Makes the traversal and subsequent analysis/code generation easier

Alternative ways of construction:

- Build concrete parse tree in parser
  - Generate AST
  - Generate high-level IR
- Build abstract syntax tree in parser
  - Generate high-level IR
- Build IR directly in parser

Hierarchy for Expressions

Variant 1

- IntegerLiteral
- BooleanLiteral
- Location
- MethodCallExpr
- CalloutExpr
- ArithmeticOp
- BooleanOp
- RelationalOp

Variant 2

- Literal
  - IntLiteral
  - BoolLiteral
- Location
- CallExpr
  - MethodCallExpr
  - CalloutCallExpr
- BinaryOp

Hierarchy for Expressions

Variant 1

- IntegerLiteral
- BooleanLiteral
- Location
- MethodCallExpr
- CalloutExpr
- ArithmeticOp
- BooleanOp
- RelationalOp

Variant 2

- Literal
  - IntLiteral
  - BoolLiteral
- Location
- CallExpr
  - MethodCallExpr
  - CalloutCallExpr
- BinaryOp

```
{
   OPER_DESC operator;
   Expression lhs;
   Expression rhs;
}
```

Statements and Methods

Statements

- AssignmentStmt

```
{
    Location lhs;
    Expression rhs;
}
```

- PlusAssingmentStmt
- BreakStmt
- ContinueStmt
- IfStmt
- WhileStmt

Method & Field Declarations

- MethodDecl
- VarDecl
  - ArrayVarDecl
- Type (TyInt and TyBool)

Specialization

Store and load nodes: for parameters, arrays and scalars

- May specify the index of the array
- May specify storage (stack, heap)

Symbol Table Summary

- Data structure that holds meta-information about the program's elements
- Descriptors for
  - Scalar and array variables: type and length
  - Functions: name, type, symbol table for parameters, symbol table for its local variables
  - Intermediate expressions: type
  - Debug information: line numbers, column numbers
- Requires efficient lookup operation

Scope

Scope: unit of program with one of more variables (or functions) defined in it.

- Program, functions, blocks
- Nested scopes: inner and outer

```
void f(int p) {  /* … */ }

void main() {
   int i;
   if (true) {
      int j; /*…*/
   }
}
```

Visibility of Variables/Functions

Defines in which scope(s) one can access the variable/function

In case of Decaf:

- Use after declaration: a variable or a function is visible only after it has been declared
- A variable/function defined in the outer scope is visible in the inner scope
- Variable declaration from inner scope may shadow the declaration from outer scope with the same name

Use after Declaration

```
void f( ) {  /* … */ }

void g( ) {

   f();

   h() ;
}

void h () { /* … */ }
```

- We can call f from g, but not h

Shadowing

- In Decaf, comes from nested scopes:

```
void main ( ) {
     int f;
     f = 1;
     if (true) {
        bool f;
        f = true; // ok
     }
}
```

- Variable is visible until the end of the surrounding scope

Global and Local Scopes

```
void f ( ) {
}

void main ( ) {
      int f;
      f = 42;  // ok
      f();     // not ok
}
```

Semantic Analyses

- Variable Referencing
- Type checking
- Array use
- Range checking
- …

Variable Referencing

```
...
A[10] = 1
...
```

- Is A declared?
- Is A an array?
- Then check for types:
  - Is A an integer array
  - Is index (10) a non-negative integer

Type Checking

- Implicit type conversions are not allowed:

```
1 + false

1 == true

false || (2+1)
```

- All expressions above are illegal!

Arrays

- Array variables can be used only to get indexed location and to get array length:

```
int a[10];
int x;
void f() {
    x = a; // not ok
    a = x; // not ok
    x = @a; // ok
    x = @x; // not ok
    x[1] = a[1]; // not ok
}
```

Range Checking

- Ensure that an integer constant is within the required range:

```
int x;
int a[10];
void main ( ) {
      x = -18446744073709551617; // not ok
      a[-1] = x; //not ok
}
```

Traversing Program's IR

- Visitor Pattern (Java)

- Pattern Matching (Scala/Haskell)

Team Tasks

- Select Parser: Select one or combine multiple
- IR design: Define the common interface and the type hierarchy
- Symbol table: Define the common interface, ensure the lookup is efficient
- Analyses: Implement the checker functions
- Tests: more important than it looks like!

Project Result

- Report: document the design decisions and the implementation
- Implementation:
  - Make a branch in the git repository (for archival)
  - Send the archive with the source code to us by the deadline time
  - We will send the instructions for submitting the project

Compiler Output

- Run as

```
./run.sh --target=inter program.dcf
```

- Returns status code 0 (no error), or non-zero (error)

- Also: prints semantics errors and file/line/column where the error happened
- Debug mode: pretty print the IR of a program

Project Evaluation

- We will run the project on a set of public and hidden tests
  - We will use Athena as our execution environment
- Most of the tests will be pass/fail
  - Compiler returns zero or non-zero status
- We will have several tests that manually check for multiple error recovery and reports

Project Evaluation

- We will make hidden tests available after all teams submit the project (including late days)
- We will report on the results of running the compiler on hidden tests a few days after the submission deadline
- Points:
  - Total 9% of the grade
  - 20% Documentation
  - 80% Implementation (public tests 33%, hidden tests 67%)

Timeline

- Announced: Monday evening
- Public tests: Later today
- Due: Next Thursday (Oct 2 midnight)