Implementing Classes and Assignments

Logistics

–Quiz 5 has been posted

–PA5 Overview today, will be posted Monday night, Due May 4th

–HW4 will also be posted Monday night, Due April 27th

–Will talk about monad implementation during last week, until then check out paper "Imperative functional programming" by Simon L. Peyton Jones and Philip Wadler if you are curious.

–For each identifier: type, scope (includes lifetime), visibility, and run-time location

Implementing Classes and Assignments

(1) Building the Symbol Table, today overview, next week Haskell example

(2) Type Checking

(3) Code Generation

PA5 Overview

Goals

- Code generation for objects, assignment statements, and arrays
- Create a symbol table that is used by later passes/visitors in the compiler
- Perform semantic analysis to find ALL redeclared variables and to report the first type error.

New pieces of grammar

- Variable declarations
- Assignment statements
- Object creation
- Member variables
- Array creation and usage

Symbol Table

Information maintained in a symbol table, which is a kind of environment

- –For each identifier: type, scope (includes lifetime), visibility, and run-time location
- –For named scopes, the set of identifiers it contains.
- –While processing the program, maintain a stack of scopes.

Example scopes

- –global scope
- –file scope
- –named space
- –package
- –unnamed scopes

Scoping in MeggyJava

Implementing type checking for PA5 MeggyJava

Visitor over AST will check for type errors at each AST node

Syntax *AST node*

`id = Exp ;` `AssignStatement(id, Exp)`

[LINENUM,POSNUM] Undeclared variable VARNAME

[LINENUM,POSNUM] Invalid expression type assigned to variable VARNAME

`public Type name(...) {...return Exp; }` `MethodDecl(name, Stms, Exp)`

[LINENUM,POSNUM] Invalid type returned from method METHODNAME

Exp . name (Args) CallExp(name, Args)

[LINENUM,POSNUM] Receiver of method call must be a class type

[LINENUM,POSNUM] Method METHODNAME does not exist

[LINENUM,POSNUM] Method METHODNAME requires exactly NUM arguments

[LINENUM,POSNUM] Invalid argument type for method METHODNAME

Error message for symbols redeclared within same scope

```
Class ID …                      ClassDecl
public Type ID …                MethodDecl
Type ID;                        VarDecl
```

[LINENUM,POSNUM] Redefined symbol VARNAME

// different in that ALL of these must be printed

Code Gen for Classes and Local variables

Method activation records on run-time stack

- Parameters will still have locations in the activation record.
- Local variables will also have locations in the activation record.

Member variables will be stored in object instances

- The new expression should cause a call to malloc.
- Member variables will have offsets within an object instance.
- The "this" variable will contain a pointer to the object instance.

Exercise: draw a memory map (RTS and heap)

```
class PA5obj {
    public static void main(String[] whatever) {
        new C().setP((byte)3,(byte)7,Meggy.Color.BLUE);  } }
class C {
    Ind oy;
    public void setP(byte x, byte y, Meggy.Color c) {
       Ind ox; ox = new Ind(); ox.put(x);
       oy = new Ind(); oy.put(y); /* Here 3 */  }       }
class Ind{
    byte _i;
    public void put(byte i){ _i = i;    /* Here 1,2 */   }
    public byte get(){ return _i; }       }
```

1: just after **ox.put()** has executed (but not returned)

2: just after **oy.put()** has executed (but not returned)

3: just after **oy.put()** has returned

BuildSymTable for varDecl

VarDecl(node)

create varSTE given node name

if it is a member variable

make the base "Z"

make the offset the current class offset

increment the class offset/size with the size of the variable

else if it is a local

make the base "Y"

make the offset the current method offset

increment the method offset/size with the size of the variable

else scream

Code Generation for method call and this

CallExp

1) Using the mapping of expression nodes to types in the symbol table, look up the ClassSTE from the receiver type. Then lookup the MethodSTE from the ClassSTE scope.

2) Generate code that pops parameters off the stack and into the appropriate registers from right to left.

Receiver reference is the first parameter (this).

3) Generate code that calls the mangled method name.

4) Generate code that pushes the return value back on the stack.

ThisExp

1) push the value of the "this" parameter onto the run-time stack

load “this” into r31:30 and then push it

Code Generation for IdExp and assignStmt

IdExp

1) Lookup id in symbol table to get VarSTE
2) If the VarSTE is a member variable
   2a) Look up VarSTE for "this" and generate code
       that loads the value of "this" into registers r31:r30.
3) load variable into a register(s) using the base+offset from VarSTE.
4) Push the variable value on the stack.

AssignStatement

1) Lookup id in symbol table to get VarSTE
2) If the VarSTE is a member variable
   2a) Look up VarSTE for "this" and generate code
       that loads the value of "this" into registers r31:r30.
3) store value of expression on top of run-time stack into base+offset
   from VarSTE