COMP 520 - Compilers

Lecture 15 (Thu Mar 28)

miniJava code generation and runtime organization

• Reading
  • skim PLPJ Chapter 8 on interpretation
  • study mJAM miniJava Abstract Machine
  • study example from class today

• Project
  • PA4 assignment and all related materials are online
The PA4 checkpoint

• your pa4 directory should have
  – miniJava package
    • Compiler.java
    • SyntacticAnalyzer
    • AbstractSyntaxTrees
    • ContextualAnalyzer
    • CodeGenerator (new subpackage)

  – mJAM package (supplied on our web page)
    • Interpreter.java
    • Disassembler.java
    • Instruction.java
    • Machine.java
    • ObjectFile.java

• mJAM is needed to check the generated code gives the right result
  – pa4 testing will not copy your mJAM, it uses the mJAM as distributed

• pa4 readiness check will be available:  ../../check/pa4.pl
Compiling and running miniJava programs (Unix)

- **Compiling test.java**
  - `java miniJava/Compiler test.java`
  - use mJAM.ObjectFile to write `test.mJAM` (note spelling!), be sure that it is written in the same directory as `test.java`
  - do not run the generated program as part of compilation!

- **Disassembling test.mJAM**
  - `java mJAM/Disassembler test.mJAM`
  - should write `test.asm` in same directory as `test.mJAM`

- **Running test.mJAM**
  - `java mJAM/Interpreter test.mJAM`
  - `System.out.println` results from `test.java` will appear on stdout prefixed by “>>> “

- **Debugging test.mJAM**
  - `java mJAM/Interpreter test.mJAM test.asm`
  - Show machine data store and state, show code, set/remove breakpoints, single instruction execution
  - Type “?” for help
Check results

• To compare miniJava and java semantics of program  foo.java

1. Run as miniJava program
   ```
   java miniJava/Compiler foo.java
   java mJAM/Interpreter foo.mJAM
   ```

2. Run as java program
   ```
   javac foo.java
   java  foo.class
   ```

• Note that mJAM println prefixes output with “>>> “
mJAM: runtime support for simple classes

- **mJAM code sequences**

  ```java
  A a = new A();  // (object creation)
  a.x;           // (qualified reference)
  a.p();         // (method invocation)
  x = x + 3;     // (field upd within p() )
  ```

  ```java
  LOADL -1  LOADL d_a[LB]  LOADL d_x  CALLI d_p[CB]  LOADL 3  CALL ADD  STORE d_x[OB]
  ```

  ```java
  LOADL S_a  LOADL d_a[LB]  CALL newobj  CALL fieldref  STORE d_a[LB]
  ```

  ```java
  CALL fieldref  STORE d_a[LB]
  ```

  ```java
  LOAD d_a[LB]  CALLI d_p[CB]  LOAD d_x[OB]  CALL ADD  STORE d_x[OB]
  ```

**Diagram:**
- Activation record on stack
- Object instance in heap
- Call instance

**Notes:**
- `LOADL -1` loads the activation record.
- `LOADL S_a` loads the object instance in heap.
- `LOADL d_a[LB]` loads the activation record field.
- `LOADL d_x` loads the instance field.
- `CALLI d_p[CB]` invokes the method within the instance field.
- `LOADL 3` loads the constant value.
- `CALL ADD` performs the addition.
- `STORE d_x[OB]` updates the object instance field.
- `LOAD d_a[LB]` loads the activation record.
- `CALL fieldref` refers to the instance field.
- `LOAD d_a[LB]` loads the activation record again.
- `CALLI d_p[CB]` invokes the method within the instance field again.
- `LOAD d_x[OB]` loads the instance field.
- `CALL ADD` performs the addition.
- `STORE d_x[OB]` updates the object instance field.

**Diagram Details:**
- `“this”` indicates the current object instance.
- `d_x` points to the instance field within `p()`.
- `x` is updated by `x = x + 3;` within `p()`.
This package is available on our web page
  - produces code for the Counter.java example
    • illustrates the Machine interface to generate mJAM instructions
  - .. then executes the generated code using mJAM
    • the Interpreter is started in debug mode so you can trace execution of the example code
    • to simplify the testing of your code generator you can install a similar shortcut to automatically execute generated code (e.g. in your compiler driver)
      - If you do so, be sure to restore standard functionality when submitting PA4