Massachusetts Institute of Technology
Department of Electrical Engineering and Computer Science

6.035, Fall 2000 | Segment III: Semantic Analysis — Homework Solutions | Monday, October 2

---

1. Draw the symbol tables and descriptors for the following class hierarchy. You should draw out the symbol tables and the class and method descriptors fully. You can represent each field, local, `this`, and parameter descriptor as a single node labeled with what it represents. For example, you would represent the `batteryLife` descriptor in the following class hierarchy with a single node labeled `field descriptor for batteryLife`. You can also represent the code for each method with a single node labeled with what it represents. For example, you would represent the `upgrade` code in the following class hierarchy with a single node labeled `code for upgrade`.

```
class car {
  int horsepower;
  void upgrade(int h) {
    horsepower = h;
  }
}
class Prius extends car {
  int batteryLife;
  void upgrade(int h) {
    horsepower = h;
    batteryLife = 0;
  }
  void charge(int c) {
    batteryLife = c;
  }
}
```

   Solution: Please see attached graphic.

2. One important aspect of semantic checking in strongly typed languages such as Espresso is to ensure that every program has a defined result. If the program violates a semantic rule, the result is a *compile time error*, and the compiler produces a message to this effect when the program is compiled. If the program violates a run time check, the result is *run time error*, and the program should print a message to this effect if it violates a run time check.

   If the compiler writer does not implement semantic checks correctly, or the semantic checks were not designed with enough care, it is possible for the compiler to generate code that generates an incorrect result or fails in some other way. For the purposes of this problem set, we say that the meaning of such a program is *chaos*, i.e., that ANY result is possible when the program runs. For the purposes of this problem set, assume that the meaning of a program is chaos if

(a) The program executes a statement that accesses a location of the form ⟨simple_expr⟩.⟨id⟩, but simple_expr does not have a field named id.

(b) The program executes a statement of the form ⟨simple_expr⟩.[⟨expr⟩], but simple_expr is not an array.

(c) The program executes a statement of the form ⟨simple_expr⟩.⟨id⟩(simple_expr$_1$, ..., simple_expr$_n$), but id is not one of simple_expr's methods.

Here are several examples of how changes in the semantic checks can cause the compiler to compile programs whose meaning is chaos.

(a) If rule 11 on page 9 of Handout 7 (Espresso Language Definition) is omitted from the list of semantic checks, the meaning of the following program is chaos.

```
class Program {
  void main() {
    badFieldName = 0;
  }
}
```

(b) If rule 10 on page 9 of Handout 7 (Espresso Language Definition) is changed so that it does not require the type of simple_expr to be an array, the meaning of the following program is chaos.

```
class Program {
  int i;
  void main() {
    i[5] = 0;
  }
}
```

(c) If rule 8 on page 8 of Handout 7 (Espresso Language Definition) is changed so that it does not require that for statements of the form ⟨simple_expr⟩.⟨id⟩(), ⟨id⟩ name one of simple_expr's methods, the meaning of the following program is chaos.

```
class C {}
class Program {
  void main() {
    C voidObject = new C();
    voidObject.foo();
  }
}
```

Each example program given as an answer to a problem must contain no more than four class declarations and four method declarations, one of the class declarations must be the Program class, and one of the method declarations must be the main method inside the program class. If no such program exists, your answer should be "No such program exists".

- **Problem A**: Assume we delete rule 18 on page 9 of Handout 7 .
  Provide an example program whose meaning is legal without the change, but compile time error after the change.

Provide an example program whose meaning is compile time error without the change, but chaos after the change.

Solution: There are two possible ways to interpret this problem. The original wording says that all fields are the equivalent of Java's "protected" fields. You can interpret removing rule 18 as either making them "public" or making them "private". We will accept both interpretations.

If you interpret it as making fields "public", i.e.: removing rule 18 means any other class can reference a class's fields, then there are no such programs to cover either case.

If you interpret it as making fields "private", i.e.: removing rule 18 means no other classes, not even subclasses, can reference a class's fields, then there are no such programs for the 2nd case (compile time error without change but chaos after change). For the 1st case (legal without the chage, but compile time error after change), the following is one possible solution:

```
class Foo {
  int x;
}
class Program extends Foo {
  int main() {
    Foo f;
    f = new Foo();
    return f.x;
  }
}
```

- **Problem B**: Assume we change rule 20 on page 10 of Handout 7 to read as follows:

  > For any class C1 which is compatible with class C2, the same name may be used for a field in C1 and C2 if and only if the type of the field in C1 is compatible with the type of the field in C2 (When such an overridden field is accessed, the first declaration up the inheritance hierarchy from the dynamic type of the object is used).

  Provide an example program whose meaning is compile time error without the change, but chaos after the change.

  Solution: No such program.

- **Problem C**: Assume we change rule 20 on page 10 of Handout 7 to read as follows:

  > For any class C1 which is compatible with class C2, the same name may be used for a field in C1 and C2 if and only if the type of the field in C2 is compatible with the type of the field in C1 (When such an overridden field is accessed, the first declaration up the inheritance hierarchy from the dynamic type of the object is used).

  Provide an example program whose meaning is compile time error without the change, but chaos after the change.

  Solution: Here is one possible solution:

```
class Foo {
}
class Bar extends Foo {
```

```
  Bar x;
  void init() {
    x = new Bar();
  }
  void bar() {
  }
}
class Quux extends Bar {
  Foo x;
  void init() {
    x = new Foo();
  }
}
class Program extends Quux {
  void main() {
    Bar zowie;
    zowie = new Quux();
    zowie.init();
    zowie.x.bar();
  }
}
```

3. The standard way to code symbol table lookups is to write a routine that starts at the lowest symbol table where the symbol could be found, then traverse up the symbol table hierarchy, returning the appropriate descriptor from the first symbol table containing the descriptor. Unfortunately, Ben Bitdiddle, intrepid 6.035 student, has a slight anomaly in his symbol table lookup routine. Instead of returning the descriptor from the *first* encountered symbol table that defines the symbol, the routine returns the descriptor from the *last* encountered symbol table that defines the symbol. During the compiler derby at the end of semester, this bug is found.

   For each of the next four problems (**A-D**) answer the questions with one of the following options: compile time error, run time error, infinite loop, chaos, or halts with variable result=n, where n is some integer (specify the value!). Also, provide a short explanation of your answers, clearly stating your assumptions.

   - **Problem A**: What is the result of compiling and running the following program using Ben's compiler?

     What is the result of compiling and running the following program using the reference compiler, which is always correct?

     ```
     class foo {
       int i;
       void bar() {
         i = 2;
         {
           int i;
           i = 1;
     ```

```
    }
  }
  int ret() {
    return i;
  }
}
class Program {
  int result;
  int main() {
    foo f;
    f = new foo();
    f.bar();
    result = f.ret();
  }
}
```

Solution: Runtime error for both compilers because `main()` never returns a value. Had it been declared `void main()` instead, the reference compiler would get 2, and Ben's compiler would get 1. Since this was a typo, full credit will be given for either answer.

- **Problem B**: What is the result of compiling and running the following program using Ben's compiler?

  What is the result of compiling and running the following program using the reference compiler, which is always correct?

```
class foo {
  int i[5];
  int bar() {
    int i;
    i[3] = 2;
    return(i[3]);
  }
}
class Program {
  int result;
  void main() {

    foo f;
    f = new foo();
    result = f.bar();
  }
}
```

Solution: Reference compiler gives compile time error, and Ben's compiler gives 2.

- **Problem C**: What is the result of compiling and running the following program using Ben's compiler?

  What is the result of compiling and running the following program using the reference compiler, which is always correct?

```
class foo {
```

```
  int i;
  void xyzzy() {
    i = 2;
  }
  void plugh() {
    return(i);
  }
}
class baz extends foo {
  int i;
  void blorple() {
    i = 1;
  }
  void plugh() {
    return(i);
  }
}
class Program {
  int result;
  void main() {
    baz f;
    f = new baz();
    f.xyzzy();
    f.blorple();
    result = f.plugh();
  }
}
```

Solution: Compile time error due to rule 20 (same-name fields are not allowed in related classes), and also because `plugh()` should return `int` rather than `void`. Were same-name fields allowed in related classes and the `plugh()`s declared as returning `int`, both compilers would give 1. The rule 20 issue was intended (as a trick question) but the `plugh()` issue was not, so we'll be giving full credit regardless of whether either issue was noticed.