Compilers — Homework 8 (Project Part 3) $P_{2.5}$: booleans and objects

Due: Wed, 12 Dec 2012, 20:00

For this project, work in groups of two or three people. Groups containing both Bachelor and Master's students are not allowed, except with the instructors permission. Every member of the group is responsible for understanding the project implementation. In this project, you will extend the compiler from Homework 5 with objects and control-flow statements. You are responsible for fixing any bugs in earlier versions of your compiler.

1 $P_{2.5}$

The language you will compile is called $P_{2.5}$. I use this name rather than to distinguish it from the languages described in the lecture notes. $P_{2.5}$ supports all features of P_0, plus:

- The boolean literals `True` and `False`.
- The comparison operators ==, `!=`, <, >, <=, and >=.
- The logical operators **`and`**, **`or`**, and **`not`**.
- **`if`** statements, including **`elif`** and **`else`** clauses.
- **`while`** statements.
- **`class`** definitions. A class must have of a *single* superclass, possibly `object`. A class must have at most one constructor and zero or more methods. Class definitions may appear only at the top-level of the file; they may not be nested in other classes or in other functions.
- Methods. A class may have one or more methods. Methods must take at least one argument. The first argument is the method receiver. Methods may appear only within a class declaration. There are no top-level functions in $P_{2.5}$. You do not need to support variadic arguments or keyword arguments. You do not need to support first-class functions. Methods can only access their parameters (including `self`) and their local variables. They cannot access variables in enclosing scopes. Methods may be recursive.
- Method invocations. A call to a method should evaluate the receiver and all other arguments, lookup the address of the method's code and call the method, passing in the receiver and the other arguments. `super` calls need not be supported. Only instance methods (not class methods) need to be supported.
- Constructors. A constructor is declared as a method named `__init__` and must take at least one argument. The first argument is the new object.
- Object allocation. A call to a function whose name is the name of a class should create a new object and invoke the class's `__init__` method—if it exists—on the new object, passing in any additional arguments.
- Fields. An object may have one or more fields. Fields are created in an object when they are first assigned. It is a run-time error to read a field that has not been initialized. The assignment statement can have a single local variable or a single field as its left-hand-side.
- **`return`** statements. Each **`return`** statement returns a single value, never a tuple. A method can have multiple return statements. It is illegal for a **`return`** statement to appear outside a method or constructor body.

- Polymorphism. Variables, including local variables, method parameters, and fields, may contain either integers, booleans, or references to objects. The `input()` function should still return integers, and the the **`print`** statement should only print integers (as a bonus, you can generalize **`print`** to handle other types).

For example, in $P_{2.5}$, you should be able to write code like the following:

```
class Point(object):
  def __init__(self, x, y):
    self.x = x
    self.y = y
  def move(self, dx, dy):
    self.x = self.x + dx
    self.y = self.y + dy

p = Point(1, 4)
while p.x < p.y:
  p.move(2, 1)
print p.x + p.y
```

This program should print `14`.

2 Task

Create a Python script named `compile.py` that takes the name of a file as a command-line argument. The file should contain the text of a $P_{2.5}$ program. The script should compile that program into x86 assembly code, which it prints to the standard output (stdout).

Use the `compiler` module to parse the new constructs. You will need to figure out the ASTs generated by the extended language. You should reject illegal programs by reporting an error at compile time.

You will need to extend the flattening and instruction selection passes to handle the new constructs. Small changes to other passes may also be required.

The body of each function, method, or constructor should be compiled into a separate x86 function. Top-level code outside a function body should be compiled into a `main` (or `_main` on Mac OS X) function. You will need to mangle function or method names in the generated code to avoid conflicts between identically named functions in different scopes and with the generated `main` function.

3 Booleans

All the passes in your compiler need to be extended to support booleans.

- Flattening should be extended to flatten the bodies of **`if`** and **`while`** statements. You might introduce a `goto` statement into the intermediate (i.e., flattened) language to make flattening easier.
- (Pseudo-)assembly code generation should be extended to support branch instructions, comparisons, and labels.
- You can choose either to disable register allocation, allocating variables on the stack, or to extend it to handle control-flow statements (for bonus points). If you do implement register allocation, virtual regsiters should be used for function parameters. The function prologue should copy parameters from the stack into the virtual registers. Liveness analysis should be extended to handle branches, using a dataflow analysis. Interference graph construction and coloring should not require any changes except those needed to substitute registers for virtual registers in the new instructions. You should use all available registers, both caller-saves and callee-saves. Callee-saves registers should be saved on entry to the function, if the function uses them. Caller-saves registers should be saved around calls (the spilling algorithm should do this automatically).
- Function return should be implemented by copying the return value into `eax` and jumping to the function epilogue.

4 Objects

A new version of `runtime.c` provides functions for creating objects, accessing fields, and dispatching methods. You are free to optimize these functions or to implement your own. You can extend and modify this file as needed to implement the required functionality.

Unlike in the lecture notes, you are not required to implement top-level functions. Therefore the code for calling functions is simpler than in the notes. If the function name is the name of a class, your compiler should generate code to allocate an instance of the class and to invoke the class's constructor (`__init__`) method. Unlike Python, only instance methods (not class methods—like Java's `static` methods) need be supported.

5 Polymorphism and tagging

$P_{2.5}$ supports polymorphic variables. For instance, the following code is legal:

```
if input():
  p = Point(1, 2)
else:
  p = 3
```

At the end of this code, `p` may contain a `Point` or it may contain an `int`. Performing an incorrect operation on a value (e.g., trying to add an object and an `int` using `+` or trying to invoke a method on an `int`) should result in a run-time error when the operation is attempted.

Since variables can store values of multiple types, we need a way to perform run-time type-checking to check if an operation is allowed. One way to do this is to *box* all primitive values (i.e., integers and booleans) into objects, then fall back on the method dispatching mechanism. Functions are provided in `runtime.c` to create and manipulate boxed integers and booleans. A more efficient implementation is to tag primitive values so that they are distinguished from pointers. You need to implement one of these methods.

6 Testing

When developing your compiler, you should use test-driven development. Write test cases—that is, $P_{2.5}$ input files—that exercise each feature of your compiler. You can reuse test cases from earlier assignments. You are encouraged to share test cases with your colleagues. Writing thorough tests before you write the compiler itself will help ensure that you cover all the cases correctly. Be sure to include both legal programs and illegal programs.

When submitting test cases, each test should consist of three files: a $P_{2.5}$ source file *test*`.py`, an input file *test*`.in`, and an output file *test*`.out`.

We will post the script we will use for running your tests. You are free to develop your code anywhere, but we will test your code on `atelier.inf.usi.ch`.

When submitting tests, each test should consist of three files: a $P_{2.5}$ source file *testname*`.py`, an input file *testname*`.in`, and an output file *testname*`.out`.

7 Deadlines

The assignment has several deadlines.

- 30 Nov: submit a set of test cases on which you will test your code. Tests submitted by *all* groups, plus tests written by the course staff, will be used to test your final compiler. Each test should consist of three files: a $P_{2.5}$ source file *test*`.py`, an input file *test*`.in`, and an output file *test*`.out`.
- 7 Dec and 12 Dec: give a 20-minute walkthrough of your code to the course staff. We will provide feedback and run tests.
- 14 Dec: submit your code, including any additional test cases.
- 19 Dec: end-of-semester presentations in the Aula Magna

When submitting the assignment, include the names of all members of the group in each file of your submission. Only one member of the group needs to submit the code. If more than one submits, the later submission will be graded. Submit both your source code and test cases. Code that does not parse or that fails because it calls undefined functions or tries to read undefined variables will be given a 0.

Both your code and tests will be graded. Be sure they are readable and well-documented.

8 Grading

This assignment counts as two regular homework assignments. The assignment will be graded as follows. The grade is out of 100 points.

- [10 pts] Test cases submitted on 30 Nov
- [10 pts] Code walkthroughs
- [80 pts] Functional correctness of language features:
  - [20 pts] Booleans, conditionals, and loops.
  - [15 pts] Class definitions, constructors, and object allocation.
  - [15 pts] Field accesses and assignments.
  - [15 pts] Method calls and return.
  - [15 pts] P_0 language features

There are also plenty of opportunities to earn more than 100 points.

- Optimizations:
  - [10 pts] Register allocation extended to handle loops.
  - [10 pts] Implement polymorphism using tagging rather than boxing.
  - [10 pts] Runtime system optimizations.
  - [20 pts] Dataflow-based optimizations.
- Additional language features:
  - [10 pts] Strings
  - [10 pts] Floats
  - [10 pts] Lists or tuples
  - [10 pts] Top-level functions
  - [20 pts] Lambdas
  - [20 pts] Type inference and type checking
  - [?? pts] ...