Compilers — Homework 3
P_0 compiler

Due: Friday, 5 Oct 2012, 18:00

For this project, work in pairs. Larger or smaller groups, or groups containing both Bachelor and Master's students are not allowed, except with the instructors permission.

1 P_0

In this project, you will build a compiler from P_0 to x86 assembly code. The source language is the same as the language in the previous assignment, but to be precise:

P_0 consists of integers, variables, arithmetic expressions, assignment statements, **print** statements, and calls to the `input` function. Read the sections of the Python Reference Manual (`http://docs.python.org/reference/`) that apply to P_0:

- Integers are *plain integers* (i.e., 32-bit integers) as defined in Section 3.2. You do not need to implement integer overflow. Plain integers are the only values in the program; that is, variables will only ever contain integers, **print** will only print integers, `input` will only return integers.
- Integer literals are as defined in Section 5.2.2 and 2.4.4. You do not need to implement long integer literals (because they are not 32 bit).
- The arithmetic operators are as defined in Sections 5.5–5.8. These include the usual unary and binary arithmetic and bitwise operators. You do not have to implement the power operator (`**`) or the division and remainder operators (`/`, `//`, `%`).
- Evaluation order is left-to-right (Section 5.14)
- The `input` function is in Section 2.1. The `input` function should read only integer constants, and return an integer.
- The **print** statement is as defined in Section 6.6, but accepts only a single integer expression as its argument, not a tuple. The extended "**print** chevron" statement should not be supported.
- Assignment statements as as defined in Section 6.2, but can assign to only one variable, not to a tuple.

Note that there are no control-flow statements or booleans in P_0.

2 Compiler

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_0 program. The script should compile that program into x86 assembly code, which it prints to the standard output (stdout).

If the source program is not a legal P_0 program, your compiler should print an error message and exit gracefully. It should not just crash. We will provide a C library that you can link to to implement `input` and **print**.

As an example, given a file `test1.py` with the following contents:

```
x = - input()
print x + input()
```

The compiler can be used as follows:

```
$ python compile.py test1.py > test1.s
$ gcc -m32 test1.s runtime.c -o test1
```

This will generate a program `test1` that when run should read two integers (e.g., 99 and 55, below) and subtract the first from the second.

```
$ ./test1
99
55
-44
```

Since P_0 is a subset of Python, the behavior of the generated program should be the same as if `python` were run on `test1.py`.

```
$ python test1.py
99
55
-44
```

You can use this fact to test your compiler.

3 Testing

When developing your compiler, you should use test-driven development. Write test cases—that is, P_0 input files—that exercise each feature of your compiler. You can reuse test cases from the interpreter assignment. 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.

4 Debugging

Inevitably, the assembly code you generate may not compile or may not run correctly. This can often be quite tricky to debug. You can use `gdb` to step through the program instruction by instruction, examining what's in each register. Often, it is faster, however, to reason about the generated code on paper, again stepping through it instruction by instruction. If something doesn't work, try to edit the assembly code by hand. Try to prune the code, removing code for that works until only the buggy code remains. Use small test cases so that there is not a lot of superfluous assembly code to reason about.

5 Submission

When submitting the assignment, include the names of both members of the pair in each file of your submission. Only one member of the pair needs to submit the code. If both submit, 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.

6 Approach

Read Chapter 1 of the lecture notes carefully.

To get the AST for the source program, you should use the `compiler` module just as in the previous assignment. Do not implement your own AST classes. From an AST you should generate a single x86 function named `_main`.

Your compiler should be structured as a series of passes. First, since assembly code is a flat sequence of instructions and P_0 code allows expressions to be nested, the compiler needs to *flatten* the ASTs so that it consists of a sequence of statements with only simple sub-expressions. The second phase is to traverse the flattened AST and to generate assembly instructions.

6.1 Flattening

The flattened AST should consist of a sequence of statements, each of which performs only one operation, assigning the reuslt (if any) to a variable. Thus, the following code from above:

```
x = - input()
print x + input()
```

can be flattened to:

```
t1 = input()
x = - t1
t2 = input()
t3 = x + t2
print t3
```

Here, the temporary variables, or *temporaries*, `t1`, `t2`, and `t3` are introduced to store the results of evaluating sub-expressions.

The flattening pass should be structured as a recursive function over ASTs (similar to the interpreter and to `num_nodes`). The function should take a non-flattened AST as an argument and return a flattened AST. Handling the temporaries is the trickiest part. When recursing on a subexpression whose value you need, you need to create a fresh temporary by generating a new name, flatten the subexpression and generate an assignment of the result of the subexpression into the fresh temporary. You can pass the name of the new temporary down to the recursive call on a child so it can generate the assignment when flattening the child tree.

6.2 Instruction selection

Once you have a flattened the tree, you need to traverse this tree, generating assembly code for each statement. Each variable or temporary needs to have a slot in the stack frame allocated for it. (In the next assignment, variables will be mapped to registers, which is more efficient.) You'll need a data structure (hint: dictionary) to keep track of the mapping of variable names to stack slots. Each statement should map to just a few instructions, often just one. Multiplication and calls to the C runtime library will take more care.

I would strongly suggest that rather than simply printing out strings of assembly code, you instead define a set of classes for assembly language instructions—essentially AST classes for assembly language—and then generate a list of instruction objects. Then, to output the assembly code, traverse this list and print. This extra effort now will pay off in the next project, in which you will implement register allocation and in which you will need to perform some analysis on the generated assembly. This approach will also make it simpler to generate the right function prologue and epilogue since you need to know how much space to allocate for the stack frame.

6.3 Odds and ends

You will also have to generate the appropriate boilerplate for the generated assembly (the `.section` directives, the function prologue and epilogue, etc). Look at the lecture notes for details.