Compilers and Interpreters

Disclaimers

While programs like compilers and interpreters are, in their simplest forms, conceptually straightforward, many people over the decades have come up with, well, let's call it variations on a theme. This document doesn't attempt to be a history of all the improvements made. So if I say, "It works *this* way", I'm often simplifying and there may well be many other approaches. So take some of this with a grain (or twelve!) of salt. But hopefully this should get you started understanding the difference between a compiler and an interpreter. Fingers crossed!

Another disclaimer – I had originally planned to write a compiler for the same language I defined for my interpreter. Somehow it never got done. Apologies. Maybe someday I'll get back to this project. Meanwhile, the sections on this document that talk about compilers are unfinished. Hopefully I'll get back to this at some point and finish them.

Yet another disclaimer – I'm usually pretty good at documenting things, but for whatever reasons, this document didn't really come together very well. Apologies.

Finally, I basically threw this program together, then started adding extra features (e.g. breakpoints). Perhaps the code got away with me a bit, and doesn't show best practices in coding. Still, the code isn't totally horrendous.

Overview

Suppose you were given sentences in a language you don't understand (let's pretend it's French), and you have to read it aloud to someone, translating it into English while you do so. Here's the beginning of the story:

Il était une nuit sombre et orageuse. Soudain, un tir a sonné.

Your job is to speak the words "It was a dark and stormy night. Suddenly a shot rang out."

So how would you do this?

Well, assuming you don't just pay someone 10 bucks to translate it, you have at least two choices.

First, you can examine the text one word at a time, look each word up in a French-to-English dictionary, take into account things like word order (e.g. French puts adjectives after a noun; English puts them before) and idioms. And then you speak the corresponding English words.

OK, that would work.

But the next day you're asked to read them for someone else. Assuming you hadn't absorbed any French the first time around, you'd do the same thing again, looking up words and so forth.

Alternatively, you could go to the effort, before reading it aloud the first time, of doing the translation and writing it down on paper. Then you could just read the English.

And to read them aloud the next day, you'd ignore the original French version of the book and just read the English version.

That's the difference between an interpreter (the first approach) and a compiler (the second). They both get the job done, and some of the processing (e.g. vocabulary lookup) is essentially the same, but they approach it in different ways.

Things Compilers and Interpreters Have in Common

In both cases they have to "understand" the intent of the programmer, expressed in whatever language he/she's used and to ultimately perform the desired calculations[1]. Traditionally this is often approached as a series of logical steps:

- Scan each source statement and parse it into its component pieces. For example, "OUTPUT = Sum + Amount" contains 5 obvious pieces: OUTPUT, Sum and Amount are (presumably) variable names. The two special characters, '=' and '+' are operators.
    - And technically, blanks may be pieces of the puzzle. They're mostly ignored in most languages, but are significant in others.
- You normally want to keep track of things, especially variable names. These go into what's called a Symbol Table.
- Now make sense of the statement(s). This varies with the language you're processing. In most languages the statement above is a simple addition of Sum and Amount followed by saving the calculation in the variable OUTPUT. But in at least one language I've used (SNOBOL), OUTPUT is a keyword and assigning to it will print the value. As another example, in FORTRAN "A ** B" means "A to the power of B". But in other languages there is no ** operator and "A ** B" would be a syntax error.
    - Note that even '+' may have different meanings in a single programming language. For example, if A and B are simple integers, in many languages A+B will add them. But if they're strings, they may be concatenated.
- Now you have to make use of everything you've figured out. Suppose we're working on the assignment "A = B + C".
    - In an interpreter you'd know that you'd have to add B and C, and you'd do so. Then you'd put the result back into memory at location A.
    - Whereas a compiler would normally output machine language to a file (e.g. in Windows a .exe file). Once this file has been created, you don't need the original source code any more (any more than you'd need the original French text).

Interpreters

A simple interpreter[2] will parse each line in turn, figure out what to do, and do it. In our example above, it has to add two fields and store the result. It will then go on to the next statement. If the program loops, the same parsing and evaluation will occur over and over again, slowing down execution.

[1] Not just calculations – file access, screen updates, whatever…
[2] Like this one.

Compilers – Machine Langauge

While there have been CPUs that can execute certain languages[3] directly[4], the vast majority of computers

TODO:

Compilers – Machine Language

Well, first we have to understand machine language. Let's set it in terms of, well, I guess for lack of a better term I'll call it a "game". Imagine you have the following:

- A basic calculator that can add/subtract/multiply/divide.
- 1000 pigeonholes, numbered 000 to 999. Each pigeonhole has a piece of paper in it.
- A pencil with an eraser

You receive a message in code. It's a probably very long string of numbers. Here's the start of an example:

010000200108256…

Your job in the game is to make sense of this number and get the result out.

So here's the coded message.

Stop reading now if you're masochistic enough to want to try to figure these out yourself.

The Solution to the Game

As we'll see below, you'll need to know these "operation codes", also called opcodes.

01 – Enter the xxx value into the calculator

02 – Enter the contents of pigeonhole xxx into the calculator

03 – Press Add. xxx is unused

04 – Press Subtract. xxx is unused

05 – Press Multiply. xxx is unused

06 – Press Divide. xxx is unused

07 – Press = on the calculator. xxx is unused

08 – Erase the value in pigeonhole xxx and replace it with what the calculator displays.

09 – Compare the calculator display with xxx. Set the condition code (see below).

[3] The ones I can think of include APL, Java, and possibly LISP.
[4] Not the ASCII text directly but an internal representation of the program.

10 – Compare the calculator display with the value in pigeonhole xxx. Set the condition code.

11 – Continue executing from location xxx if the condition code is less than

12 – Continue executing from location xxx if the condition code is less than or equal

13 – Continue executing from location xxx if the condition code is equal

14 – Continue executing from location xxx if the condition code is not equal

15 – Continue executing from location xxx if the condition code is greater than or equal

16 – Continue executing from location xxx if the condition code is greater than

17 – Print the value in pigeonhole xxx

99 – Exit the program. xxx is unused

Each group of 5 digits is divided into two sections: a 2-digit opcode, and a 3-digit value. The opcode tells you what operation to perform and the 3 digit value (referred to as *xxx*) gives any additional information the opcode requires to perform its duties.

So our first job is to split these code up into groups of 5. This would be

TODO: Insert code here

So the first instruction says to TODO: Insert explanation here and a few more instructions.

In very early computer days (we're talking the 1940's here), this is how you'd program a computer, all in numbers. However someone figured out that he could write a program (all in numbers) that let you use text mnemonics for the opcodes. And you could refer to memory locations (i.e. the pigeonholes) symbolically. So instead of, say,

01123, you could write LDA 123, where LDA is short for LoaD into the Accumulator (which is what we're referring to as the calculator)

TODO: Write more here

For example, instead of writing "01123" you could write "LDA X" (Load

TODO:

C# Code Comments

I've written a simple[5] interpreter in C#. The code will be rather familiar to someone who's coded in C++ (or even C), and even other languages. But let me point out a couple of features of C# that might not be immediately obvious.

[5] Really, really simple. Verging on brain dead!

C# allows fields to be *null*. Similarly C/C++ has *nullptr* and Python has *None*. This is roughly akin to stating that something is undefined[6]. But how can this apply to something like an integer? C# has a special syntax: *int? foo;* This means that *foo* isn't really an *int*, but is a data structure with two fields in it – a normal *int* and a *bool* that says whether it's had a value assigned to it or not. Assigning *foo = null* results in the compiler simply setting the *bool* to *false*. Attempting to get the value of *foo* (via *foo.Value*) in that case will throw an exception.

Note that this syntax of appending a question mark to the data type applies only to basic value types (e.g. *int, short, float, double, bool* etc).

The Language Implemented by MyDumbInterpreter

In the description below, *var* represents a variable name. *num* represents a non-negative number. *varnum* represents either a *var* or a *num*. Finally, a *relop* or relational operator compares two values. The *relop*s are =, !=, <, <=, >, >=.

As mentioned in the next section, all variable names and numbers are limited to single characters. Thus

```
foo = goo + 27
```

would be invalid – no multi-character names, blanks, or multi-digit constants.

Also note that except for the **"** opcode (see below), blanks are <u>not</u> ignored and are invalid.

The valid statements in my dumb little language are either an assignment statement (defined as one that has an equal sign in the second position) or a command (identified by a specific character (known as an operation code or *opcode* in the first position).

- Assignment syntax: *var*=*varnum***op***varnum* where **op** is one of +, -, *, /.
  - For example, N=5
  - To get multi-digit values (e.g. N=10) you have to build up the value
    - N=5
    - N=N*2
  - Complex expressions are <u>not</u> supported – N=5*2+3 is invalid
- Commands:
  - * -- comment. Such lines are ignored.
  - . – breakpoint. Execution stops, but can be continued from where it left off.
  - : – label. Defines the target of a GoTo opcode, for looping
  - " – print. Displays text. A backtick (`) before a symbol displays the value of the symbol
  - > – goto. Compares two values and conditionally branches to a specified label

I'm going to play a dirty trick on you. I'm *not* going to further describe the format of the commands. Between comments in the source code, and the implementation of the commands, you get to figure

[6] A common use of *null*, especially in databases, is to signify not so much that a field is undefined, but that we just don't know what the value (if any) is. For example, a Person database table might have a MiddleInitial field. An empty string ("") could mean that the person has no middle initial. But a *null* value could mean that we don't know, one way or the other, if the person has a middle initial.

them out. Also, the sample code in my awfully obscure language might help. But reading the code and figuring out my language is meant to be a kind of homework for you.

The Structure of MyDumbInterpreter

I had a number of goals in designing my interpreter. These included:

- Simplify as much as possible so as not to get sidetracked on anything other than the concept of an interpreter. For example, to avoid having to scan for variable-length variable names, all names must be exactly one letter long (i.e. A-Z, a-z). Similarly, all numbers must be exactly one digit long (i.e. 0-9).
- But interpreters usually have nice debugging features, such as detecting undefined variables at runtime, the ability to set breakpoints, etc. We'll make a rough stab at supporting some of that.
- We support only integer values (i.e. no floating point, no strings, etc)
- I pay lip service to detecting and reporting errors. But not all errors may be checked for and any error messages produced may not be the best.

While there are many ways to approach the job of implementing an interpreter, a modern[7] interpreter us likely to have many components. One of the primary components is a *tokenizer* that scans the source code and does preliminary parsing. For example, it might take the statement

```
int NewValue = OldValue + 1
```

and break it into the 6 tokens:

- Int
- NewValue
- =
- OldValue
- +
- 1

The interpreter could then work on this[8] internal representation. At least this way, the overhead of recognizing variable names, functions, constants, etc is done only once, even if the statement was executed multiple times in a loop.

Note: MyDumpInterpreter doesn't do this. I didn't want to complicate the code by having to scan, for example, "OldValue+ 1" and realize that the name "OldValue" ended when you found a blank or a special character like a plus sign or an open or closed parenthesis or a bracket for subscripting, etc. This routine wouldn't be all that difficult to write (exercise: modify the code to allow multi-character names and multi-digit values), but it wouldn't really clarify the basic idea of an interpreter. And then, of course, we'd need data structures to represent things like variables, operators, etc.

[7] As opposed to earlier interpreters that had to accomplish their goal in a very small amount of storage. There were interpreters that ran in 3K of storage!

[8] Or possibly a more sophisticated data structure representing the input language.

The Internal Logic of the Interpreter

There is a small mainline (CompilerInterpreter.cs) that defines sample code and creates the Interpreter class and invokes its Run method, passing the sample code. There is also support for breakpoints to support Edit and Continue.

In some respects, the main routine in the interpreter is *DoArith*. That's where the actual calculations are done.

But overall, the idea is to get each line in turn (subject to looping), determine what it's trying to do (either an assignment, a numeric expression to be printed, or a command), and do what's required.

Ideas for Extending This Program

- Multi-character variable names (e.g. Total, not just T)
- Multi-character constants (e.g. 10, not just 0-9). If you're feeling ambitious, support unary minus (e.g. -10). Careful with unusual input such as A = B --5, which is, of course equivalent to A=B+5. And depending on the language you're defining/implementing, is a space (or tab!) between the minus sign and the 5 valid?
- Not just integers, but floating point
    - Hint: Put a DataType field in class Symbol for Integer and Float. You'll need two numeric fields, one for the integer and one for the floating point value.
    - The arithmetic routines must now be made more complicated. For example, to add two numbers you have to take into account whether both arguments are integer, whether the first is float and the second is an integer (and thus has to be converted to float before the addition can be done), and so on
- More complex expressions – A = (B + C) * D – 5.4
    - Good luck with this
- For the truly ambitious, include function calls in expressions.
- Vectors. And if you implement floating point as well as integers, don't forget to allow arrays of ints and floats.
- Add a "watch window" that shows the current values of all variables. Especially useful when a breakpoint has been reached.
- Add a "single step" button. It would execute one instruction, then stop. Also very useful with the watch window.
- Install a timer that would single-step every, say, 1 second, updating the watch window. See your program execute.
- Add a small window that would let you type in a subset of commands (assignment, expressions and Print) and see the result.
- Add a Load/Save feature for *.mdi (My Dumb Interpreter) source files.