Python Compiler Internals

Thomas Lee <tom@vector-seven.com>

Abstract

In this paper we introduce a new language construct to demonstrate how one might go about modifying the Python language compiler. The Python language itself is a powerful, dynamically-typed language that runs on a variety of operating systems and platforms. The internals of the compiler itself follow a pattern common to many language implementations, with lexical analysis, parsing and code generation phases. This makes the Python source code a great place to learn how languages might be implemented in the more general sense. By the end of the paper, it is hoped the reader will see that contributing to the Python language core is not necessarily as difficult as it might seem.

1 Overview

It might surprise those who think of Python as a "scripting language" to learn that the core of the Python interpreter actually has the structure of a classic compiler. When you invoke the "python" command, your raw source code is scanned for tokens, these tokens are parsed into a tree representing the logical structure of the program, which is finally transformed into bytecode. Finally, this bytecode is executed by the virtual machine.

In order to demonstrate how the discrete components of the Python compiler fit together, we will use the Python 2.6 code base to walk through the addition of a new language construct: the "unless" statement, seen in Listing 1.

```
import sys

passwd = sys.stdin.readline().strip()
unless passwd == 'tehsecret':
    print 'Passwords do not match. Exiting!'
    sys.exit(1)
```

Listing 1: The syntax of the unless statement

All code in this paper was written and tested against Python 2.6 beta 3, for which the Subversion repository can be found here: http://svn.python.org/projects/python/tags/r26b3/. The only thing that should change in the final 2.6 release is the line numbers referenced in the source listings presented here.

2 Lexical Analysis

Rather than directly trying to parse a stream of text, it is often easier – and faster – to break the input text into a series of "tokens", which might include keywords, literals, operators and/or identifiers. These tokens can be inspected by the parser much more efficiently than a stream of raw text. The process of transforming the input text into tokens is known as "lexical analysis", "tokenizing" or simply "scanning".

The entry point to the lexical analyzer is the *PyTokenizer_Get* function in Parser/tokenizer.c. This function is called repeatedly from within the main parsing function, *parsetok* in Parser/parsetok.c, which is in turn used by one of several higher level parsing functions.

To implement our "unless" statement, we don't need to explicitly do anything to the tokenizer code – we need only modify the grammar description, as we will see in the next section.

3 Parsing

For the compiler to make any meaning of the source program, the token stream emitted by the lexical analyzer must be organized into some sort of structure. This is the job of Python's parser, which takes a token stream as input and – based on the rules declared in the Python grammar – produces an Abstract Syntax Tree (AST).

Python uses a custom parser generator to automatically generate its parser from a grammar description. The output of the parser is a parse tree, which can be thought of as a low level representation of the parsed program in the structure defined by the grammar description.

In Python 2.5, an additional step was added to the parse phase: the construction of an Abstract Syntax Tree (AST) from the parse tree. The AST, like the parse tree, is an in-memory representation of the program being parsed albeit at a somewhat higher level and thus easier to manipulate.

Now, to add our new "unless" construct to the Python language we must first modify the grammar file – Grammar/Grammar – to describe the syntax of this feature, as in Listing 2.

```
/* Grammar/Grammar:78 */
if_stmt: 'if' test ':' suite ('elif' test ':' suite)* ['else' ':' suite]
while_stmt: 'while' test ':' suite ['else' ':' suite]
unless_stmt: 'unless' test ':' suite
for_stmt: 'for' exprlist 'in' testlist ':' suite ['else' ':' suite]
```

Listing 2: Modifying the Python grammar to accommodate ``unless''

Since our "unless" statement is effectively an "if" statement with its test logic inverted, we *could* use the existing "If" and "Not" AST nodes to implement this feature. In fact, this is how the try..except..finally syntax was introduced in Python 2.5. However, for the purposes of this paper we will modify the AST structure so we can demonstrate how to generate bytecode for our new construct. To that end, we make a change to Parser/Python.asdl – which contains the AST definition – as in Listing 3.

```
/* Parser/Python.asdl:25 */
    | For(expr target, expr iter, stmt* body, stmt* orelse)
    | While(expr test, stmt* body, stmt* orelse)
    | Unless(expr test, stmt* body)
    | If(expr test, stmt* body, stmt* orelse)
```

Listing 3: Adding a new type of AST node

Finally, we need to add some code to handle the transformation from the grammar to the AST in Python/ast.c. Here we add *ast_for_unless_stmt*, as seen in Listing 4.

```
/* Python/ast.c:2805 */
static stmt_ty
ast_for_unless_stmt(struct compiling* c, const node *n)
{
    expr_ty test_expr;
    asdl_seq *suite_seq;

    /* unless_stmt: 'unless' test ':' suite */
    REQ(n, unless_stmt);

    test_expr = ast_for_expr(c, CHILD(n, 1));
    if (test_expr == NULL)
        return NULL;

    suite_seq = ast_for_suite(c, CHILD(n, 3));
    if (suite_seq == NULL)
        return NULL;

    return Unless(test_expr, suite_seq, LINENO(n), n->n_col_offset, c->c_arena);
}
/* ... <snip> ... */
/* Python/ast.c:3125 */
    case while_stmt:
        return ast_for_while_stmt(c, ch);
    case unless_stmt:
        return ast_for_unless_stmt(c, ch);
```

Listing 4: The transformation from unless_stmt to an Until AST node

The parse tree to AST transformation code follows the popular Visitor pattern. At line 3123, we add a hook to call the appropriate transformation code if we encounter an *unless_stmt* node in the parse tree. Once inside our transformation function, we construct an Unless node by recursively parsing the test and the body.

Note that after making these changes you may need to explicitly regenerate some files:

```
$ rm -f Python/graminit.c && make Python/graminit.c
$ rm -f Python/Python-ast.c && make Python/Python-ast.c
```

At this point, the compiler will be capable of scanning and parsing our new "unless" statement. All that remains is to add code to generate the bytecode for our new AST node.

4 Code Generation

The next phase of compilation – code generation – takes the AST constructed in the previous phase and produces a PyCodeObject as output. A PyCodeObject is an independent unit of executable

code, containing all the data and code necessary for independent execution by the Python bytecode interpreter.

Before we look at more source code, it is important to understand that Python's bytecode interpreter is a stack-based virtual machine. This means that the process of bytecode execution manipulates a data stack, with instructions adding, removing and operating upon the top couple of stack elements. With that fresh in our mind, let's look at how we generate bytecode from our new Unless AST node in Listing 5 (we will discuss what the bytecode actually *does* in the next section).

```
/* Python/compile.c:1623 */
static int
compiler_unless(struct compiler *c, stmt_ty s)
{
    basicblock *end;
    basicblock *next;

    assert(s->kind == Unless_kind);

    end = compiler_new_block(c);
    if (end == NULL)
        return 0;
    next = compiler_new_block(c);
    if (next == NULL)
        return 0;
    VISIT(c, expr, s->v.Unless.test);
    ADDOP_JREL(c, JUMP_IF_TRUE, next);
    ADDOP(c, POP_TOP);
    VISIT_SEQ(c, stmt, s->v.Unless.body);
    ADDOP_JREL(c, JUMP_FORWARD, end);
    compiler_use_next_block(c, next);
    ADDOP(c, POP_TOP);
    compiler_use_next_block(c, end);
    return 1;
}
/* ... <snip> ... */
/* Python/compile.c:2167 */
case If_kind:
    return compiler_if(c, s);
case Unless_kind:
    return compiler_unless(c, s);
```

Listing 5: Generating bytecode for the Unless AST node

At this point, our implementation of the "unless" statement is complete. Recompile Python and try it out:

```
$ ./configure && make clean && make
$ ./python <<EOF
> dv = False
> unless v:
>     print 'test'
> EOF
test
```

See Python/compile.c for the full details of Python's bytecode generator.

5 Code Execution

The execution of Python bytecode is handled by the bytecode interpreter. As mentioned in the previous section, the interpreter is a stack-based virtual machine that executes Python bytecode instructions which operate on a single data stack. We do not need to make any changes to the bytecode interpreter itself for our "unless" statement to work, but let's take a closer look at how it interprets the bytecode we generated in the previous section.

We first execute the test expression, the result of which will be pushed onto the stack. Next, the JUMP_IF_TRUE opcode will inspect the value on the top of the stack and determine whether or not it has a value representing truth. If it does, we skip the body entirely and continue execution at the end of the "unless" statement. If the expression evaluates to a *false* value, however, the compiler will execute the body of the "unless" statement.

You will also notice that the first instruction of whichever branch is executed is to pop the topmost element off the data stack (POP_TOP). This is because the JUMP_IF_TRUE expression leaves the tested expression on the stack. In our case, we no longer need the value of the test expression so it is simply discarded with the execution of the POP_TOP instruction.

If you are interested in the details of what the individual bytecodes do, documentation for all Python bytecodes can be found at `http://docs.python.org/lib/bytecodes.html`. Also, see *PyEval_CodeEvalEx* and *PyEval_EvalFrameEx* in Python/ceval.c if you are interested in looking at the bytecode interpreter itself in more detail.

6 Conclusion

It should now be clear that one doesn't have to be a rocket scientist to contribute to the core of the Python language. If you are at all interested in how the internals of a real world compiler work, Python is a great place to start. If you're unsure of where to go next, here are a couple of ideas that will be both achievable and educational for the entry-level Python hacker:

1. Rewrite the "unless" construct using only existing AST nodes
2. Add a new operator to the language by modifying the tokenizer
3. Investigate how builtin functions work under the hood
4. Investigate the symtable pass

There's much to learn from tinkering with the code, and more still by making an active contribution to the project by fixing bugs, contributing documentation or answering newbie questions on the user mailing list. Give it a go – you might just surprise yourself.