Parsing Text Input CS 005
Jimmy Ho
Computer Science Department
Brown University
October 13, 1998

This document is aimed at increasing your understanding of parsing text input, as well as informing you of some Java tools that will be extremely helpful.

How does Java read your source file?

Overview:

There are five phases that a Java compiler will go through to process a .java file. Essentially, what a compiler does is *translate* a file written the Java language into a language that the Java Virtual Machine (JVM) can understand. Here is a quick summary of these five phases.

lexical analysis: In the beginning, the input file is simply a stream of characters. Lexical analysis will group the characters in logical units called *tokens*. This is analogous to breaking an English sentence into words and punctuation.

syntax analysis: The syntax analysis phase will then take these tokens and group them into a meaningful structure. At the same time, the compiler checks for syntactic errors. This meaningful structure is usually called an *abstract syntax tree*, or AST. Again, this is like organizing the English words into paragraphs, sentences, noun/verb constructs, etc.

semantic analysis: This phase checks for semantic correctness. For instance, type checking and argument checking are taken care of here. In English, "Turn off the stone." is *syntactically* correct, but does not make sense, because stone does not have an on/off function.

code optimization: This is where the code is optimized to run faster and more efficiently.

code generation: This phase takes the final version of the AST and generates the java byte codes to the .class file. Here is where the English document is translated to French, for example.

Thus, the compiler the breaks the file into words (tokens), organizes it into some sort of structure (AST), checks it for correctness, and then uses that to translate into the target language. This scheme is used by compilers for most programming languages today, including C, C++, Fortran, and Java. In fact, any time there is a clearly defined *source language*, a clearly defined *target language*, and a decent mapping between the two languages, then these methods can be used.

For instance, Intel Corporation uses different software tools from different companies to design, develop, and manufacture their Pentium chips. The problem is that these software packages use different file formats to store the circuit design or layout. Consequently, there are programmers at Intel who write translation software that transfers a design from one application to the other by translating the file formats. The exact same methodology used to translate a Java program into JVM language is used to do this translation. The same could be said of translating file formats for different word processor programs.

Each phase will be discussed in detail below.

Lexical Analysis:

Lexical analysis breaks up a string of characters into distinct units, called tokens. For example, a simple lexical analyzer can take a string of characters, and treat blanks as delimiters and all other printable characters as parts of tokens, as the following example illustrates:

> input:
> `A bunny's life's for me.`
>
> tokens:
> | `A` | `bunny's` | `life's` | `for` | `me.` |

But suppose it were useful to separate the punctuation from the words. Then the tokens can be divided into two classes, one for punctuation, and another for "word" tokens, which exclude punctuation characters.

> input:
> `A bunny's life's for me.`
>
> tokens:
> | `A` | `bunny` | `'` | `s` | `life` | `'` | `s` | `for` | `me` | `.` |

For more complicated lexical schemes (like in Java), there are *identifiers* which consists of upper and lower case letters plus the underscore ('_'). There are *key words* like `class` or `int`, *operators* which include *, (,), %, /, +, −, etc. etc. etc.

Thus, a input stream like this:

```
foobar=(foo + 2.34e6);
```

might be separated into these tokens:

ident	*operator*	*left-paren*	*ident*	*operator*	*floating-point-literal*	*right-paren*	*stmt-separator*
`foobar`	`=`	`(`	`foo`	`+`	`2.34e6`	`)`	`;`

where *ident* stands for *identifier* and *stmt-separator* stands for *statement-separator.*

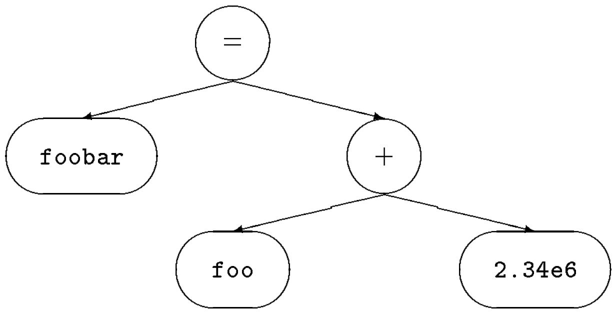
There are tools that assist in doing lexical analysis. Java has `StringTokenizer`, which does very simple lexical analysis. For more power and flexibility, one can use `StreamTokenizer`, `lex`, or `JLex`. When `lex` is given an input file with the appropriate parameters, it will output a piece of C code that will do all the lexical analysis one needs. `JLex` is basically the same as `lex`, except it is written in Java and outputs Java code. I talk about these packages so that the reader can remember them when go into the real world, but it would probably be overkill, and not worth the effort, to use this on the Bunny World project.

Syntactic Analysis

Syntactic analysis is the process of grouping the tokens into a coherent hierarchy. Thus, our tokens for the mathematical expression above can be converted into the following form:

input:
`foobar=(foo + 2.34e6);`

abstract syntax tree:



Notice that extraneous elements like the parenthesis or semi-colons are stripped. This is because the structure of the tree itself provides that information.

For Java, one can imagine a tree who has a root node of *file*, which branches off to all the `package` and `import` statements, as well as all the `class` declarations. Then the class declarations will, in turn, have child nodes representing the methods and instance variables. Instance variables will have child nodes representing the access modifier, variable type, variable name, and the initial value, if any. And so on.

Semantic Analysis

Semantic analysis usually involves "walking" the abstract syntax tree constructed in the parsing phase. Basically, the compiler traverses the tree making sure that the program is *semantically* correct. Some examples of semantic *in*correctness include:

- Naming conflicts. That is, declaring the same variable name twice in the same function.
- Type checking. For instance, assigning an integer to an Object reference, or making sure both sides of a boolean operator evaluate to a boolean value (i.e. `true` or `false`).
- Argument checking. Making sure the arguments passed in to a function are of the correct type.
- Non-existent variables. Making sure something is declared before being used.

This is harder than one might think, especially with object-oriented programming languages. One example is that a variable name in a function has to be checked against the function's local variables, that class' instance variables, the class' parents' instance variables, etc. before an error message can be issued. This gets worse in languages like C++, where there is multiple inheritence. Semantic analysis is typically the hardest part of writing a compiler (except maybe code optimization).

Code Optimization and Code Generation

Code optimization is process of making the code run faster, execute in less instructions (thus, smaller .class files), and use less memory. Code generation consists of taking the abstract syntax tree and producing the Java bytecodes that make up a .class file.

This part of compilers isn't really relevant to CS 005 and is beyond the scope of this course. More detailed information on this can be discovered by taking CS 126, the introductory compiler course.

Java tools for File I/O and Parsing

FileLineReader:

`FileLineReader` is a class that is *not* provided by Java. It is provided by CS 5's thoughtful, loving TAs. It is initialized with the pathname of the input file. Calling `readLine()` will result in String object returned containing a line of the input file. Here are the core methods one should be aware of:

Constructor function:
`public FileLineReader(String path);`
where path could be something like `"/u/jkh/foobar/inputfile.bunny"`.

To read in a line:
`public String readLine() throws InvalidFileException;`

To check for the end of the file:
`public boolean isEndOfFile();`

StringTokenizer:

Now that we can read a line from a file, what do we do with it? The answer lies in the, `StringTokenizer` class, which does a limited form of lexical analysis on any string given to it. It is extremely useful for processing text input for small scale applications (i.e. Bunny World). Here is the full declaration for its constructor function:

```
public StringTokenizer(String inputString, String delimiterCharacters, boolean returnTokens)
```

`inputString` is the string that is to be parsed. `delimiterCharacters` is a String containing all the characters that are desired to be delimiters. `returnTokens` will cause the object to return the delimiters as tokens as well. Usually, the delimiters will not be necessary, so this is set to false. In fact, Java provides another function declaration:

```
public StringTokenizer(String inputString, String delimiterCharacters)
```

which is equivalent to `StringTokenizer(str, delim, false)`, so delimiters are skipped, not returned. If all that is desired for delimiters is whitespace, then the following function is the best way to go:

```
public StringTokenizer(String inputString)
```

which is equivalent to `StringTokenizer(str, '' \n\t\r'', false)`. The escape sequences stand for newline, tab, and carriage return, respectively.

Here are some other functions one should be aware of. There functionality is self-explanatory:

```
public boolean hasMoreTokens(a);
public String nextToken();
```

Further documentation can be found in the usual Java reference texts. There is a short program that can be copied from `/home/jkh/pub/StringTokenizerTest.java` which allows one to experiment with StringTokenizer with different delimiters and a variety of inputs. Instructions on how to run it and how to vary its parameters are contained within the file.

Sample Usage:

Here is an example of using the FileLineReader and StringTokenizer classes together:

```
public void readFile(String filename) {

  FileLineReader lineReader = new FileLineReader(filename);
  StringTokenizer tokenizer = null;
  String line = null;

  while (lineReader.isEndOfFile() == false) {

    try {
      line = lineReader.readLine();
    } catch (InvalidFileException e) {
      // do something constructive.
    }

    tokenizer = new StringTokenizer(line);
    processTokens(tokenizer); // delegate to another subroutine
  }
}
```

Parsing Command Line Input

Overview:

Command line input is where the user can type one line instructions to the application or system. Parsing command line input usually consists of parsing one line of text at a time. Moreover, the instructions are executed as each line is read in. Thus, there is an immediate response to input from the command line. Examples of command line interfaces include some query interpreters for databases (SQL) or the UNIX shell (csh, tcsh, ksh).

Designing the Command Language:

The command language is the overall lexical and syntactic structure of the interface. Sometimes there is a trade-off between how user-friendly the interface is, and how difficult it is to program (or how much time is allotted to do it.). Finding a good balance depends on the application. That is, what it's used for, by whom, how often, will it be modified or extended later, etc. etc. However, there are principles that can make an interface more friendly and easier to program at the same time. The two most important ones are *structure* and *simplicity*.

Simplicity of design is a principle that should be applied to all aspects of software engineering. It makes things easier to understand, easier to code, easier to debug, easier to document, easier to modify, produces more robust code, makes concepts easier to to explain to Dilbert-like managers, requires less time to do, and is very elegant, even sexy. Enough said about simplicity.

Consistency of design is also important to designing the command language. If there is a discernible structure to the commands, then users are able to learn and remember commands more easily. Having an interface that is too complex will increase the learning curve for novices and give even the experienced users difficulty in recalling complex syntaxes and obscure commands.

One major issue is the naming of commands. Just because one is programming in the UNIX environment, he/she should not feel the urge to have cryptic UNIX-style abbreviations for commands (i.e. grep, awk, sed, ln). In fact, abbreviations aren't even necessary. Better yet, if the design is sufficiently modular, one can easily have more than one way of invoking a command (i.e. removefile = rm). Thus, the frequent users are satisfied because they don't have to type as much, and the novices appreciate the longer, more intuitive command names. If abbreviations are used, they should have a consistent abbreviation scheme (i.e. always take the first three letters, or subtract out all the vowels).

The grammatical structure of commands is important as well. Suppose there are two commands: GO and KILL. It would be confusing to the user to have:

```
>> GO direction
>> bunny KILL
```

A more consistent structure would be:

```
>> GO direction
>> KILL bunny
```

Notice also that these commands are all capitalized. It doesn't really matter whether the commands are in all caps or all lower-case, as long it is *consistent*. Generalizing to all commands, one can have a structure that looks like this:

```
>> command-name arguments
```

This is how most UNIX shells work.

Other examples of structure include the specification of arguments. For instance, here are several different ways to specify a file to print:

```
>> PRINT filename
>> PRINT -Ffilename
>> PRINT -F filename
>> PRINT FILE=filename
>> PRINT F:filename
```

Again, the decision to as to which scheme to use is often more a matter of taste than anything else. But be sure to *use the same structure for all commands.* By the way, the first `PRINT` command is the most simplistic, but the latter ones may be better when there are multiple, optional arguments (for instance, which printer to send the print job to, how many copies, multiple files, etc.).

Implementation:

Implemention is closely tied to the design of the command language. The more complex the language is, the more complex the implementation will be. Again, one of the most important principles to apply here is *simplicity.*

Although one can apply all of compiler theory in its full glory to this problem, it is better to keep it as simple as possible. Thus, if, for lexical analysis, one can choose between `lex` (very complex), `StreamTokenizer` (somewhat complex), and `StringTokenizer` (simple), look at the most simplistic tool first. Use the more complex ones only if the current tool lacks the functionality that is needed.

Syntactic analysis can be as simple as: "The first token is always the command name. The format of the rest of the tokens is determined by what command has been invoked."

$$\textit{command} \longrightarrow \textit{command-name argument1 argument2} \ldots$$

In fact, despite the awesome power and functionality of UNIX shells, they pretty much have the above structure, except that they add some rules involving semi-colons, pipes, and redirection characters.

Similarly, semantic analysis could just be: "If the command doesn't exist, then it's an error. Whether the arguments to a command make sense depend on the command."

In summary, the compiler concepts discussed at the beginning of this document can be used to organize the problem conceptually, but don't let those concepts make the problem harder than it actually is. All the usual rules of good design and coding practices apply here, including modularity, delegation, abstraction, information hiding, etc. Simple command line interfaces for applications like the Bunny World are a lot easier to do than most people believe.