

# Assignment 1: The Scanner

This is due on Friday, February 12.

Create a Token class. A token is one of the basic lexical symbols of your program. Tokens can be

- **Identifiers:** These start with a letter and consist of letters, digits, and underscores. You can extend this for your implementation if you wish.
- **Numbers:** These are non-negative integers (so we regard “-23” as two tokens, a minus-sign and the number 23. If you want to allow negative numbers as tokens, you may.
- **Keywords.** These are listed on the last page of the BPL reference manual: **int void string if else while return write writeln read**
- Special symbols and punctuation marks:  **; , [ ] { } ( ) < <= == != > > + - \* / % &**
- The **end-of-file token** that indicates the end of the input file has been reached.

A Token object should have instance variables that hold its string value, an integer that describes the kind of token it is, and an integer for the line number of the line of the source file where the token was found. For example, if line 26 of our source file is

```
if (num < 23)
```

you should find 6 tokens for this line. The data for these are:

```
Kind: T_IF   Value: "if"   Line: 26
Kind: T_LPAREN Value: "("   Line: 26
Kind: T_ID   Value: "num"   Line: 26
Kind: T_LESS Value: "<"     Line: 26
Kind: T_NUM  Value: "23"    Line: 26
Kind: T_RPAREN Value: ")"   Line: 26
```

The constants that describe the tokens can be given any names and values you choose; these should all be defined in your token class.

White space has no meaning in BPL (this isn't Python) and your scanner should skip over it. The two lines

```
f(23)
f ( 23 )
```

have exactly the same token streams, as does

```
f
(
23
)
```

Comments, which in BPL are delimited by `/*` and `*/` are similarly ignored.

The heart of your scanner is a method **getNextToken()** that assigns to a public Token variable **nextToken**. Note that `getNextToken()` doesn't return anything; it assigns to a variable. You can

examine this variable as often as you wish; it doesn't change until the next call to getNextToken(). In your parser module you will call getNextToken( ) each time you have handled the current token and are ready for the next. For example, if you are parsing the function call f(5+x), you start with nextToken representing the identifier f. After saving this token, you call getNextToken( ) and see that your next token is a left parenthesis. This says you are parsing a function call. Another call to getNextToken( ) should "consume" that parenthesis and put you at the start of the argument for the function call. When you have parsed the entire argument, you should expect nextToken to be the right parenthesis. And so forth.

**Printing.** Both for debugging purposes and for error handling in future modules, you should provide some way to print token information, including the kind, the string value, and the line number on which the token was found.

**Error handling.** Your compiler should not crash, regardless of what input it receives. It should handle errors gracefully, but not necessarily persistently. If you wish to have your compiler exit the first time it finds an error, that is perfectly acceptable and in keeping with current compiler trends. In Java this is easy to accomplish by having your compiler modules, such as the scanner, throw an exception when it finds an error. If you are implementing in a different language find your own solution. Regardless of how you treat this, your compiler should never, ever crash.

**Scanner output.** The job of the scanner is to produce a token stream, which it generally does silently. For this assignment we need to be sure that your scanner is finding tokens correctly, so you should give some output identifying tokens and their line numbers. **You should turn in a program that asks the user for the name of a BPL file, opens this file, and repeatedly calls getNextToken(), printing each token found until it gets to the end of the file.** Here is such a program in Java that works for my Scanner:

```
public static void main(String[] args) {
    String inputFileName;
    Scan myScanner;

    inputFileName = args[0];
    myScanner = new Scan( inputFileName );
    while (myScanner.next Token.kind != Token.T_EOF) {
        try {
            myScanner.getNextToken();
            System.out.println( myScanner.next Token );
        }
        catch (ScannerException e) {
            System.out.println( e );
        }
    }
}
```

If I run this on the file

```
/* A program to compute factorials */

int fact( int n) {
    if (n <= 1)
        return 1;
    else
        return n*fact(n-1);
}

void main(void) {
    int x;
    x = 1;
    while (x <= 10) {
        write(x);
        write(fact(x));
        writeln();
        x = x + 1;
    }
}
```

I get the following token stream:

```
Token 100, string int, line number 3
Token 132, string fact, line number 3
Token 128, string (, line number 3
Token 100, string int, line number 3
Token 132, string n, line number 3
Token 129, string ), line number 3
Token 114, string {, line number 3
Token 103, string if, line number 4
Token 128, string (, line number 4
Token 132, string n, line number 4
Token 123, string <=, line number 4
Token 130, string 1, line number 4
Token 129, string ), line number 4
Token 106, string return, line number 5
Token 130, string 1, line number 5
Token 110, string ;, line number 5
Token 104, string else, line number 6
Token 106, string return, line number 7
Token 132, string n, line number 7
Token 118, string *, line number 7
Token 132, string fact, line number 7
Token 128, string (, line number 7
Token 132, string n, line number 7
Token 117, string -, line number 7
Token 130, string 1, line number 7
```

and so forth. Your token kinds, being constants, are likely to be different from mine, but your string values and line numbers should be the same.