

Copyright (C) 2021 Timothy Fossum

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.1 or any later version published by the Free Software Foundation. A copy of the license is included in the file "COPYING", entitled "GNU Free Documentation License".

Programming Language Concepts CSCI344

Course Objectives

- Lexical analysis
- Syntax
- Semantics
- Functional programming
- Variable lifetime and scoping
- Parameter passing
- Object-oriented programming
- Logic programming
- Continuations
- Exception handling and threading

Lexical Analysis, Syntax analysis, and Semantic analysis

The *syntax* (from a Greek word meaning “arrangement”) of a programming language refers to the rules used to determine the structure of a program in the language. *Syntax analysis* is the process of applying these rules to determine the structure of a program. A program is *syntactically correct* if it follows the syntax rules defining the language. Every programming language has a set of syntax rules (these rules differ from one programming language to another) that define its programming language specification.

Before we can specify the syntax rules of a programming language, we must first specify the *lexical* (from a Greek word meaning “word”) structure of the language, i.e., the symbols used to construct a program in the language. These symbols are called *tokens*. *Lexical analysis* is the process of applying these rules by reading the input and isolating its tokens. Tokens comprise the “atomic structure” of a program.

Lexical analysis is also called *scanning*. You can think of scanning as what you do when you “scan” a line of printed text on a page for the words in the text. Programming language tokens normally consist of things such as numbers (“23” or “54.7”), identifiers (“foo” or “x”), reserved words (“for”, “while”, etc.), punctuation symbols (“.”, “[”]. Every programming language has a set of rules that define the tokens in the language (these rules differ from one programming language to another) that are part of the programming language specification.

Lexical Analysis, Syntax Analysis, and Semantic Analysis

A *lexical analyzer* is a program or procedure that carries out lexical analysis for a particular language. Such a program is also called a *scanner*, *tokenizer*, or *lexer*. The input to a scanner is a stream (sequence) of characters, and its output is a stream of tokens. The behavior of a scanner for a language is defined by the lexical specification of the language.

A *syntax analyzer* is a program or procedure that carries out syntax analysis for a particular language. Such a program is also called a *parser*. The input to a parser is a stream of tokens (produced by a scanner), and its output is a *parse tree*, which is an abstract representation of the structure of the program. The behavior of a parser for a language is defined by the syntax specification of the language.

The string of input characters that makes up a token is called a *lexeme*. For example, when you read a word (token) from printed text on a page, the sequence of characters that make up the word is its lexeme. In this case, the first lexeme is “the” (ignoring case), consisting of the individual letters ‘t’, ‘h’, and ‘e’. This lexeme is an instance of an English part of speech called an *article*. In this case, “article” is the token and “the” is the instance. The other instances of the “article” token (in English) are “a” and “an”. **A token is an abstraction, and a lexeme is an instance of this abstraction.**

Lexical Analysis, Syntax Analysis, and Semantic Analysis (continued)

The *semantics* (from a Greek word meaning “meaning”) of a program language refers to the rules used to determine the meaning of a program in that language. Here “meaning” refers to (a) whether the program makes sense (such as “the dog ate the bone”) but it may not have a meaning in terms of what it “does”. Programs are expected to “do” something, and “do” is part of their semantics. *Semantic analysis* is the process of applying the rules to a program written in the language to determine its meaning.

A *semantic analyzer* is a program or procedure that carries out semantic analysis. The input to a semantic analyzer is a parse tree. The output is the execution of the resulting program (as defined by what the program does when it is run) or some intermediate form (such as machine code) that can be executed at some other time. Direct execution is called *interpretation*, whereas generating an intermediate form is called *compilation*. We will use the interpreter in these notes, though the techniques we describe apply as well to compilation.

Lexical Analysis, Syntax Analysis, and Semantic Analysis (continued)

When a program produces some output, for example, the language defines what specific behavior results from running the program. For example, the defined semantics of Java dictates that the following Java program prints the standard output stream, the decimal character 3 followed by a newline.

```
public class Div {  
    public static void main(String [] args) {  
        System.out.println(18/5);  
    }  
}
```

This course is about programming language syntax and semantics, with a focus on semantics. Syntax doesn’t matter if you don’t understand semantics.

Quick question: what output is produced by the following snippet of Java code?

```
print(18/5)
```

Try executing this using the following command:

```
python3 -c "print(18/5)"
```

Does this say anything about how the semantics of Java and Python differ when it comes to integer division?

Syntax and semantics

You can use a language *compiler* to tell you if a program you write is correct, but it's much more difficult to determine if your program actually does the behavior you *want* – that is, if a program is semantically “correct”. There are two basic problems:

1. how to specify formally the behavior you want, and
2. how to translate that specification into a program that actually behaves according to the specification.

Of course, **a program is its own specification**: it behaves exactly as the instructions say it should behave! But nobody knows exactly how to write a *behavioral specification* that precisely and unambiguously describes what you want. Part of the problem is because what you want is often imprecise and ambiguous – it's hard to translate this behavioral specification into a program whose semantics precisely match the specification. Because of this, programming will always be an inherently ambiguous activity. (Creating behavioral specifications is a topic of interest in software engineering, and properly belongs in the discipline of software engineering.)

In this course, we are interested in defining precisely and unambiguously what a program *behaves*: its semantics. After all, if *you* don't know how a program behaves, it's hopeless to put that program into a production environment where users expect it to behave in a certain way.

Syntax and semantics (continued)

This course is about programming languages, and particularly about the semantics of programming languages. A programming language *specification* is a document that defines:

1. the lexical structure of the language (its tokens);
2. the syntax of the language; and
3. the behavior of a program when it is run.

In particular, we give examples of language specifications that describe the semantics and syntax structure of a number of languages and how to implement them. We show how variables are bound to values, how functions are called, and how parameters are passed when functions are called.

Because a program in a particular language must be syntactically correct before its semantic behavior can be determined, part of this course is about syntax. In the final analysis, semantics is paramount.

Tokens

Assume that we have a program written in some programming language (of languages such as C, Java, Python, and so forth.) The first step in the structure of a program is to examine its lexical structure: the “atoms

A program is, at the lowest level, a stream of characters. But some characters are typically ignored (for example, “whitespace”, including spaces, tabs, and newlines), while some characters group together to form things that can be interpreted (for example) as integers, floats, and identifiers. Some specific characters are meaningful in the language, such as ‘class’ and ‘for’ in Java. Some characters are meaningful, such as parentheses, brackets, and the comma, while some pairs or characters are meaningful such as ‘++’ and ‘<’. The term *token* is used to refer to such atoms.

A *token* in a programming language is an abstraction that considers one or more characters in the character stream as having a particular meaning in the language – a meaning that is more than the individual characters in the string. The term *lexical analysis* refers to the process of taking the characters representing a program and converting it into a stream of tokens meaningful to the language.

Tokens (continued)

Lexical analysis takes character stream input and produces token stream output. For example, if a language knows only about integers and the dot symbol, a character stream consisting of characters

23.587

might produce three tokens as output, with the following lexemes:

23

.
587

while a language that knows about floats and doubles might produce one token, with the following lexeme:

23.587

In what follows, we will often use the term “token” (an abstraction). However, it may be more appropriate to use the term “lexeme” (an instance of the abstraction). The context should make our intent clear.

Tokens (continued)

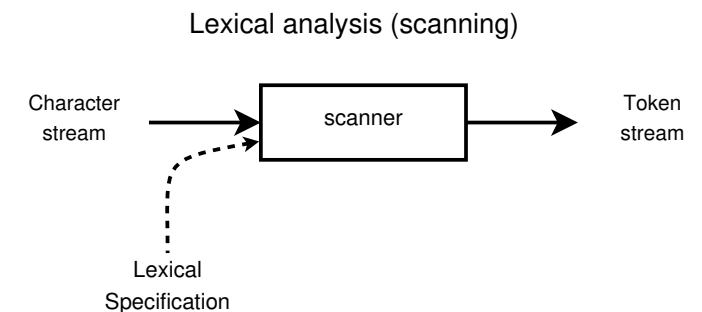
The purpose of lexical analysis is to take program input as a stream and to produce output consisting of a stream of tokens that conform to the *specification* of the language.

By a *stream*, we mean an object that allows us to examine the current item in the stream, to advance to the next item in the stream, and to determine whether there are more items in the stream. We accomplish something similar to the `hasNext()` and `next()` for Scanner objects in Java's `java.util` library.

By a *stream of characters* we mean a stream of items consisting of characters in a character set such as ASCII or UTF-8. By *stream of tokens* we mean a stream of items that are tokens defined by a token specification. The lexical analysis process provides a `hasNext()/next()` interface for a stream of tokens, but it also provides a similar `current()/advance()` interface in more detail below.

As we noted earlier, lexical analysis is referred to as *scanning*, and the program that carries out this process.

Tokens (continued)



In this course, we describe a tool set called PLCC, which stands for “Program Language Compiler Compiler”. The PLCC tool set takes a program language *specification* and “compiles” it into a set of Java programs. These programs implement the three phases of program interpretation: lexical, syntax, and semantic analysis.

A programming language specification in PLCC is a text file that describes the language grammar. Each language that we describe in this course has its own grammar file. A grammar file has three sections: the lexical specification, the syntax specification, and the semantic specification. These sections appear in the file in this order: first lexical, then syntax, and finally semantic. A line containing a single ‘%’ separate the sections.

We start by describing the structure of the lexical specification section.

Tokens (continued)

The lexical specification section of a grammar file uses *regular* expressions to specify language tokens. A regular expression is a formal description that can match a sequence of characters in a character stream. For example, the regular expression ‘d’ matches the letter d, the regular expression ‘\d’ matches any decimal digit, and the regular expression ‘\d+’ matches one or more decimal digits. **You should read the Java documentation for the Pattern class for more information about how to write regular expressions.**

When specifying tokens, we must identify what input stream characters are not part of tokens and should be skipped. Typically, we skip whitespace and newlines. We identify these skipped characters in the grammar file with a *skip specification* line like this :

```
skip WHITESPACE '\s+'
```

The regular expression ‘\s’ stands for “space” (the space character and newline), and the regular expression ‘\s+’ stands for one or more space characters. We use the symbolic name “WHITESPACE” to identify this particular set of characters to be skipped. (Any symbolic name would suffice, but it is customary to use one that describes its purpose.) We also typically skip comment characters.

Characters to be skipped during lexical analysis are not tokens.

Tokens (continued)

To specify tokens, we use *token specification* lines having the form

```
token <TOKEN NAME> <re>
```

where <TOKEN NAME> identifies the token (it must be in ALL CAPS), and <re> is a regular expression that defines the structure of the token’s lexeme.

We adopt one simplifying rule for *all* the languages we discuss in this book: *tokens cannot cross line boundaries*. Be warned, however, that not all programming languages conform to this rule.

For example, we may use the following lines in our grammar file to specify the *number*, the *reserved word* `proc`, and an *identifier*:

```
token NUM '\d+'
token PROC 'proc'
token ID '[A-Za-z]\w+'
```

We like to use symbolic names for our tokens that make it easy to remember what they represent.

You can find the documentation for regular expressions such as the `Pattern` class.

Tokens (continued)

Whenever we are faced with two token specifications whose regular expressions match a string of consecutive input characters, we *always choose the longest possible match*. For example, if the next characters in the input are

```
procedure
```

the above specifications would match an ID token with lexeme `procedure` as well as a PROC token with lexeme `proc`. Both of these specifications begin with the beginning ('`proc`') of input, but the ID match is longer.

If two or more token specifications match the same input (longest match), we *choose the first specification line in the grammar file that matches the token*.

In summary, for a given input string, we always

- 1. choose the token specification with the longest match, and**
- 2. among those with the longest match, choose the first token specification that appears in the grammar file.**

We use the phrase *first longest match* to describe these rules for choosing the token.

Tokens (continued)

Writing a scanner is somewhat involved, so our PLCC tool set provides a scanner `Scan.java` automatically from a grammar language specification. The PLCC tool set consists of the program `plcc.py` written in Python, along with a collection of Java support files. This tool set works with any system that supports Python 3 and Java.

The `plcc.py` Python program and the `Std` subdirectory that contains the support files are on the RIT Ubuntu lab systems in this directory:

```
/usr/local/pub/plcc/src
```

This directory also contains a shell script called `plccmk` that runs the `plcc.py` program with input from a grammar language specification. Typically you will run `plccmk` in a directory that contains this specification. The `plcc.py` program produces a collection of Java programs in a subdirectory. The `plccmk` script then compiles these Java programs (using `javac` compiler). When you are working on one of our Ubuntu lab systems, you can run `plccmk` to process the various languages we will specify in this document.

See the `HOWTO.html` file in `/usr/local/pub/plcc/tvf` for more information about how to set up your CS account environment so that you are able to access the required program files and library routines on CS workstations.

Tokens (continued)

To use the PLCC tool set, follow these steps:

1. Create a working directory. This directory will be specific to the language you want to implement.
2. In this working directory, put your `grammar` language specification file. As we describe later, you may also put additional `grammar-related` files in this directory.
3. If you wish, create subdirectories containing test files that you can use to exercise your language implementation.
4. Run `plccmk` in your working directory. This will create a Java program containing Java source files for an interpreter for your language.
5. If `plccmk` produces any PLCC or Java compile errors, edit your source files to fix these errors and repeat the above step.

The `grammar` file is a text file that defines the tokens of the language. In a language specification file, comments starting with a '#' character are ignored by the PLCC tool to the end of the line. These comments are ignored by the PLCC tool.

Tokens (continued)

Our first `grammar` file examples contain only token specifications. We use `grammar` files to define language syntax, and then semantics. We concentrate only on token specifications.

Each of these examples can be put in a file named `grammar` for the language. These examples define what input should be skipped and what should be treated as tokens. Comments in a language specification start with the # character and go to the end of a line.

- `# Every character in the file is a token, including the token CHAR '.'`
- `# Every line in the file is a token
token LINE '.*'`
- `# Tokens in the file are 'words' consisting of one or more
letters, digits or underscores -- skip everything else
skip NONWORD '\W+' # skip non-word characters
token WORD '\w+' # keep one or more word characters`
- `# Tokens in the file consist of one or more non-whitespace
characters, skipping all whitespace.
Gives the same output as Java's 'next()' Scanner
skip WHITESPACE '\s+' # skip whitespace characters
token NEXT '\S+' # keep one or more non-whitespace characters`

Tokens (continued)

To test these, follow the steps give on Slide 0.16. Create a separate tory for each test (we use the convention that such directories have written IN_ALL_CAPS), and create a `grammar` file in this working the given contents. Then, in this directory, run the following comma

```
plccmk
```

The `plccmk` command creates a `Java` subdirectory populated with Java source files

```
Token.java  
Scan.java
```

along with some additional support files.

Examine the `Token.java` file in the `Java` subdirectory to see how lates the token specifications in your `grammar` file into Java code the token and skip names with their corresponding regular expressio

Also examine the `Scan.java` file to see how the PLCC tool set ner for the language. You will generally never need to make ch Java source files – they are created automatically by the PLCC to grammar language specification file.

Tokens (continued)

In your working directory, run your scanner with the following com

```
java -cp Java Scan
```

and enter strings from your terminal to see what tokens are recogniz ner. The ‘`-cp_Java`’ command-line arguments sets the Java CLA ronment to the `Java` directory; this, in turn, tells the Java interprete the `Scan` program.

The `Scan` program expects input from standard input (your keyb duces output lines that list the tokens as they are scanned, in the form

```
lno: NAME 'string'
```

where `lno` is replaced by the input line number where the token i is replaced by the token’s symbolic name, and `string` is replaced corresponding lexeme from the input that matched the `NAME` token

Tokens (continued)

When specifying tokens in a `grammar` file, you can omit the `token` term (the `skip` term). This means that both

```
WORD '\S+'
```

and

```
token WORD '\S+'
```

are considered as equivalent. We follow this convention in all of our examples.

The important pieces of the `Scan` class are the constructor and `cur()` and `adv()`. The `Scan` constructor must be passed a `BufferedReader` which is the input stream of characters to be read by the scanner. A `BufferedReader` can be constructed from a `File` object, from a `InputStream`, or from a filename given in a `String`. The `Scan` program reads from this `BufferedReader` object line-by-line, extracts tokens from each line (skipping characters if necessary), and delivers the current token with the `cur()` method – `cur` stands for *current*.

Tokens (continued)

The `adv()` method advances the scanning process so that the token returned by the next call to `cur()` is the next token in the input. Notice that multiple calls to `cur()` without any intervening calls to `adv()` all return the same token.

A `Token` object has four public fields (also called *instance variables*): a `enum_Match` field named `match` that is the token's symbol; a `String_str` field that is the token's lexeme derived from the input stream (this is returned by the `toString()` method in this class); an `int_line` field that is the line number, starting at one, of the input stream where the token appears; and a `String_line` field that is the source line of the input stream where the token appears.

For the purposes of compatibility, the `Scan` class defines methods `hasNext()` and `next()` that behave exactly like their counterparts in the Java Scanner class. The `boolean_hasNext()` method returns `true` if and only if there are more tokens, in which case the `Token_next()` method returns the next `Token` object from the input stream.

Tokens (continued)

For example, consider a grammar file (directory **IDNUM**) with the following lexical specification:

```
skip WHITESPACE '\s+'
NUM '\d+'           # one or more decimal digits
ID '[A-Za-z]\w*'    # a letter followed by zero or more
```

When you run `plccmk` on this specification, it creates the file `Token.java` in the `Java` subdirectory having a public inner enum class named `Match` which consists of the following identifiers and associated patterns:

```
WHITESPACE ("\\s+", true) // the 'true' means it's a skip
NUM ("\\d+")
ID ("[A-Za-z]\\w*")
```

Any Java file that needs to use the enum values `NUM` and `ID` can refer to them symbolically as `Token.Match.NUM` and `Token.Match.ID`.

Running the `Scan` program in the `Java` directory takes character strings from standard input (typically your keyboard) and prints all of the results on standard output (typically your screen), one token per line. Each printout includes the line number where the token appears, the token name (`NUM` or `ID` for example), and the lexeme (printed in single quotes). Any input that does not match any of the skip or token specifications prints the representation of an `$E`

Tokens (continued)

The `printTokens()` method in the `Scan` class produces the output shown on the previous slide. Here is the code for this method:

```
public void printTokens() {
    while (hasNext()) {
        Token t = next();
        String s;
        switch(t.match) {
            case $ERROR:
                s = t.toString();
                break;
            default:
                s = String.format("%s '%s'", t.match.toString(), t.lexeme);
        }
        System.out.println(String.format("%4d: %s", line, s));
    }
}
```

When invoked with no command-line arguments, the `main()` method of the `Scan` class calls the `printTokens()` method on a `Scan` object constructed with standard input: `System.in`.

Tokens (continued)

Two special “tokens” are defined in the `Token.java` class:

```
$ERROR  
$EOF
```

Since the PLCC lexical specification requires that token symbol names start with an uppercase letter, these “tokens” cannot be confused with language symbols.

The `$ERROR` “token” is produced when the scanner encounters a character that does not match the beginning of any skip or token specification. The `toString()` value of this token is of the form `!ERROR (. . .)`, where the parentheses part displays the offending character. In the context of syntax analysis (covered later), such a character cannot be part of a syntactically correct program. If syntax analysis will terminate with an error.

The `$EOF` “token” is produced when the scanner encounters end-of-input. In the context of syntax analysis, encountering end-of-input means that there are no further input tokens to process, so syntax analysis terminates prematurely).