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 written 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 syntax rules (these rules differ from one programming language to another) that are part of the programming language specification.

Before we can specify the syntax rules of a programming language, we must specify the *lexical* (from a Greek word meaning “word”) structure of the language: 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 program 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 (tokens) 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”), and punctuation symbols (“.”, “[”). Every programming language has 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* that 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 particular collection of characters that make up the word is its lexeme. In this paragraph, 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 programming language refers to the rules used to determine the meaning of a program written in the language. Here “meaning” refers to (a) whether the program makes sense, and (b) what the program does when it is run. In languages such as English, a sentence can make sense (such as “the dog ate the bone”) but it may not have any meaning in terms of what it “does”. Programs are expected to “do” something, and what they “do” is part of their semantics. *Semantic analysis* is the process of applying these 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 either a direct 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 run at some other time. Direct execution is called *interpretation*, whereas producing an intermediate form is called *compilation*. We will use the interpretation approach in these notes, though the techniques we describe apply as well to a compilation approach.

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

When a program produces some output, for example, the language semantics defines what specific behavior results from running the program. For example, the defined semantics of Java dictates that the following Java program sends, to 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 an emphasis on semantics. Syntax doesn't matter if you don't understand semantics.

Quick question: what output is produced by the following snippet of python3 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 syntactically correct, but it's much more difficult to determine if your program always produces 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 the way its instructions say it should behave! But nobody knows exactly how to create a *behavioral specification* that precisely and unambiguously describes what you *want* – in part because what you want is often imprecise and ambiguous – and then translate this behavioral specification into a program whose semantics *provably behaves* according to the specification. Because of this, programming will always be problematic. (Creating behavioral specifications is a topic of interest in its own right and properly belongs in the discipline of software engineering.)

In this course, we are interested in defining precisely and unambiguously how 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 other users expect it to behave in a certain way.

Syntax and semantics (continued)

This course is about programming languages, and particularly about *specifying* 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 lexical and syntax structure of a number of languages and how to implement their run-time behaviors (semantics). We show how variables are bound to values, how to define functions, 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. But in the final anaysis, semantics is paramount.

Tokens

Assume that we have a program written in some programming language. (Think of languages such as C, Java, Python, and so forth.) The first step in analyzing 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 character sequences are meaningful in the language, such as ‘`class`’ and ‘`for`’ in Java. Some individual characters are meaningful, such as parentheses, brackets, and the equals symbol, while some pairs or characters are meaningful such as ‘++’ and ‘<=’. We use the term *token* to refer to such atoms.

A *token* in a programming language is an abstraction that considers a string of 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 that make up the string. The term *lexical analysis* refers to the process of taking a stream of characters representing a program and converting it into a stream of *tokens* that are 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 '`.`', an input 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 just a single token, with the following lexeme:

```
23.587
```

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

Tokens (continued)

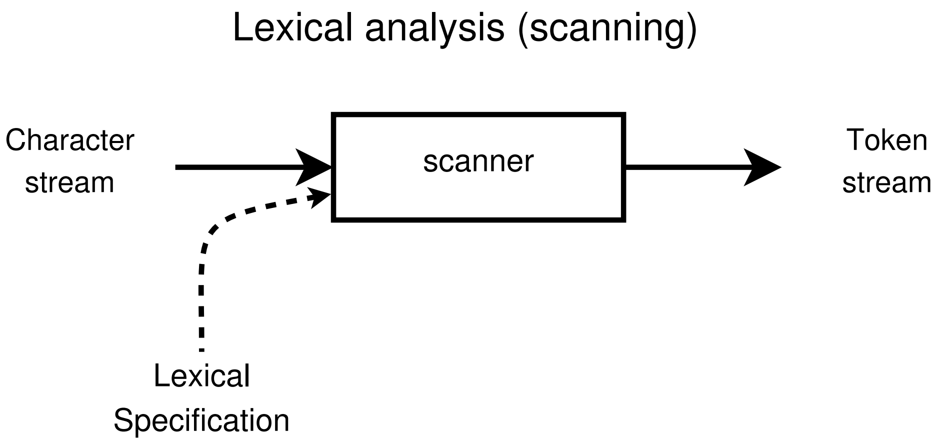
The purpose of lexical analysis is to take program input as a stream of characters and to produce output consisting of a stream of tokens that conform to the *lexical specification* of the language.

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 if there are no more items in the stream. We accomplish something similar to this in Java with `hasNext()` and `next()` for Scanner objects in Java's `java.util.Scanner` library.

By a *stream of characters* we mean a stream of items consisting of individual 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. Our lexical analysis process provides a `hasNext()/next()` interface for accessing the stream of tokens, but it also provides a similar `cur()/adv()` interface described in more detail below.

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

Tokens (continued)



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

A programming language specification in PLCC is a text file normally named `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 `grammar` file in this order: first lexical, then syntax, and finally semantic. A line consisting of 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 of a pattern 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 information about how to write regular expressions.**

When specifying tokens, we must identify what input stream characters do *not* belong to tokens and should be skipped. Typically, we skip whitespace: spaces, tabs, and newlines. We identify these skipped characters in the `grammar` file using a *skip specification* line like this :

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

The regular expression '`\s`' stands for "space" (the space character, a tab, or a newline), and the regular expression '`\s+`' stands for one or more spaces. We use the symbolic name "WHITESPACE" to identify this particular collection of characters to be skipped. (Any symbolic name would suffice, but it makes sense to use one that describes its purpose.) We also typically skip comments.

Characters to be skipped during lexical analysis are not tokens.

Tokens (continued)

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

```
   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 class: *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 a *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 these in the Java `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 one with longest possible match*. For example, if the next characters in the input stream 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 match the beginning (‘`proc`’) of input, but the `ID` match is longer.

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

Tokens (continued)

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

The `plcc.py` Python program and the `Std` subdirectory that contains its 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 invokes the `plcc.py` program with input from a `grammar` language specification file. Normally you will run `plccmk` in a directory that contains this specification file. The `plcc.py` program produces a collection of Java programs in a subirectory named `Java`. The `plccmk` script then compiles these Java programs (using the Java compiler). When you are working on one of our Ubuntu lab systems, you can simply run `plccmk` to process the various languages we will specify in this course.

See the `HOWTO.html` file in `/usr/local/pub/plcc/tvf` for *important information* about how to set up your CS account environment so that PLCC will be able to access the required program files and library routines on CS servers and lab 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` subdirectory containing Java source files for an interpreter for your language.
5. If `plccmk` produces any PLCC or Java compile errors, edit your `grammar` file to fix these errors and repeat the above step.

The `grammar` file is a text file that defines the tokens of the language using skip specifications and token specifications as we have illustrated earlier. The language specification file can contain comments starting with a '`#`' character and continuing to the end of the line. These comments are ignored by the PLCC tool set.

Tokens (continued)

Our first `grammar` file examples contain only token specifications. Later, we will use `grammar` files to define language syntax, and then semantics. For now, we concentrate only on token specifications.

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

-
  ```
  # Every character in the file is a token, including whitespace
  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 method.
  skip WHITESPACE ’\s+’  # skip whitespace characters
  token NEXT ’\S+’       # keep one or more non-space characters
  ```

Tokens (continued)

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

```
plccmk
```

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

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

along with some additional support files.

Examine the `Token.java` file in the `Java` subdirectoryto see how PLCC translates the token specifications in your `grammar` file into Java code that associates the token and skip names with their corresponding regular expression patterns.

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

Tokens (continued)

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

```
java -cp Java Scan
```

and enter strings from your terminal to see what tokens are recognized by the scanner. The '`-cp Java`' command-line arguments sets the Java `CLASSPATH` environment to the `Java` directory; this, in turn, tells the Java interpreter where to find the `Scan` program.

The `Scan` program expects input from standard input (your keyboard) and produces 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 is found, `NAME` is replaced by the token's symbolic name, and `string` is replaced by the token's corresponding lexeme from the input that matched the `NAME` token specification.

Tokens (continued)

When specifying tokens in a `grammar` file, you can omit the `token` term (but not 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 subsequent examples.

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

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*): an enum Match field named match that is the token's symbolic name; a String str field that is the token's lexeme derived from the input stream (and is returned by the toString() method in this class); an int lno 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 the input stream has additional tokens, in which case the Token next() method returns (and consumes) 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 "word" chars
```

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` whose elements consist of the following identifiers and associated patterns:

```
WHITESPACE ("\\s+", true) // the ’true’ means it’s a skip spec.
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 stream input from standard input (typically your keyboard) and prints all of the resulting tokens to standard output (typically your screen), one token per line. Each printed line gives the line number where the token appears, the token name (`NUM` or `ID` in this example), and the lexeme (printed in single quotes). Any input that does not match one of the skip or token specifications prints the representation of an `$ERROR` token.

Tokens (continued)

The `printTokens()` method in the `Scan` class produces the output described 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.str);
        }
        System.out.println(String.format("%4d: %s", lno, s));
    }
}
```

When invoked with no command-line arguments, the `main()` method in the `Scan` class calls the `printTokens()` method on a `Scan` object constructed from 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 begin with an uppercase letter, these “tokens” cannot be confused with language-specific token symbols.

The `$ERROR` “token” is produced when the scanner encounters an input 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 ‘`...`’ part displays the offending character. In the context of syntax analysis (which we cover later), such a character cannot be part of a syntactically correct program, so syntax analysis will terminate with an error.

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