Course Objectives

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

The syntax of a programming language refers to the rules governing the structure of a program written in the language. A program is syntactically correct if it follows the syntax rules defining the language. Every programming language has syntax rules, and these rules are part of the programming language specification.

Before the syntax of a programming language can be given, the language specification must define the tokens of the language – its “atomic structure”. Programming language tokens consist of things such as numbers (“23” or “54.7”), identifiers (“foo” or “x”), reserved words (“for”, “while”), and punctuation symbols (“.”, “[]”). A language’s specification always starts with defining its tokens.

The string of input characters that constitutes the token is called a lexeme – scanning a program to isolate its tokens is called lexical analysis.

The semantics of a programming language refers to the behavior of a program written in the language when the program is executed. When a program produces some output, for example, the specific output that is produced is defined by the language semantics. For example, the defined semantics of Java dictates that the following Java program produces output of 3:

```java
public class Three {
    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.
Syntax and semantics (continued)

A compiler for a language will 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 because nobody knows exactly how to translate what a person wants into a program that provably behaves the way the person wants, programming will always be problematic.

Behavioral specification is a topic of interest in its own right and properly belongs in the disciplines of programming languages and software engineering, but it is not the focus of this course. (There are some specification languages in use – Z (pronounced “Zed”) and CASL are two of them – but none have proven to be the magic bullet.) In this course, we are interested precisely in how a program behaves – its semantics. After all, if you don’t know how a program will behave, 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. A programming language documents how to write a program in the language that is syntactically correct and how that program behaves when it is run. Whether or not its behavior is what you want is a different matter. Along the way, we will show how different implementations of things such as variable bindings, functions definitions, and parameter passing can lead to different behaviors (semantics).

Because a program in a language must be syntactically correct before its semantic behavior can be determined, part of this course is about syntax. But in the final analysis, 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 determine its constituent parts: the “atoms”.

A program is, at the lowest level, a stream of characters. But some characters are typically ignored (“whitespace”, including spaces and tabs), while some characters group together to form things such as integers, floats, and identifiers. Some specific character sequences are meaningful, 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 of a programming language is a string of one or more characters that has 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.
Lexical analysis takes character stream input and produces token stream output. For example, if a language knows only about integers, the input stream consisting of characters

```
23.587
```

might produce three tokens as output:

```
23 . 587
```

while a language that knows about floats and doubles might produce just a single token:

```
23.587
```

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 *stream of tokens* we mean that we can examine the current token in the stream and advance to the next token in the stream, much as we do with `hasNext()` and `next()` for Scanner objects in Java. Be warned, however, that lexical analysis doesn’t always work *exactly* like `hasNext()`/`next()`. 

**Tokens** (continued)
The process of taking a stream of characters as input and producing a stream of tokens as output is called *scanning*: a *scanner* is a program that does this.

For the languages we will deal with in this course, we will specify the structure of tokens by means of *regular expressions*. (Other approaches use *deterministic finite automata*, also called *finite state machines*, to do the same thing.) 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 specify patterns.

When specifying tokens, one of the first things is to specify what characters should be skipped over because they cannot appear in a token. Typically, tokens do not have whitespace – spaces, tabs, newlines, so these characters must be skipped. When we define our tokens, we can express these skipped characters using a notation such as

```plaintext
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 will adopt one simplifying rule for the languages we discuss in this class: *tokens cannot cross line boundaries*. Be warned, however, that not all programming languages conform to this rule.
To specify a particular token, we name and define the structure of the token using a regular expression. For example, the following lines might specify a *number*:

```
  token NUM '\d+
```

– or the *reserved word* *proc*:

```
  token PROC 'proc'
```

– or an *identifier*:

```
  token ID '[A-Za-z]\w*
```

You can find the documentation for regular expressions such as these in the Java Pattern class.

One important rule for analyzing tokens is that whenever we are faced with two possible matches of a token upon input, we always *choose the longest possible match*. So if the input stream contained the characters

```
  procedure
```

the above specifications would produce an *ID* token with string value *procedure* instead of a *PROC* token with string value *proc*: both of these patterns match the beginning of input, but the *ID* match is longer.

The other important rule is that if two or more patterns match the same input (longest match), we always *choose the first pattern in our specification that matches*. 
Tokens (continued)

Writing a scanner is somewhat involved, so we have provided you with a tool that can produce a Java scanner program automatically from a file that specifies the tokens using regular expressions. This tool, named plcc, is written in Perl. You should be able to use this tool with any system that supports Perl and Java. You can find the plcc program and the template files that are in the Std subdirectory here:

/home/student/Classes/443/PLCC

This directory also contains a shell script called Make that further automates almost all of the activities necessary to create a scanner, given a token specification for a language.

If you are working on our lab systems, you can simply run plcc or plccmk (which links to the Make script) to process the various languages we will be specifying.

If you are working on some other system, you will need to copy the following files and directories into a suitable directory structure on that system where you will do your work. You may need to tweak the Make script to accommodate your particular directory structure: In the following list, plcc and Make are executable files and Std is a directory,

/home/student/Classes/443/PLCC/plcc
/home/student/Classes/443/PLCC/Std
/home/student/Classes/443/PLCC/Make

When running plcc, you need to give the name of the grammar file on the command line:

plcc grammar

When running plccmk, the file name grammar is assumed.
Here are some example grammar specification files that you can try. Each of these examples defines what input should be skipped and what input should be treated as tokens. Comments in the grammar specification 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
  # skip 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

To test these, create a separate directory for each test, and create a grammar file in this directory with the given contents. Then, in this directory, run the following commands:

```
plccmk
cd Java
java Scan
```

The Scan program will expect input from the keyboard and will produce output lines that list the tokens as they are scanned.
After running the `plccmk` command, a Java subdirectory will be created, populated with the following Java source files:

Token.java
Scan.java

The `plccmk` command also compiles these source files, so when you change to the Java subdirectory, you can run the Scan program directly to test your scanner: enter strings from your terminal and see what tokens are recognized by the scanner.

Examine the `Token.java` file to see how the grammar specification gets translated into Java values that specify the token names and token patterns.

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

```
WORD \S+
```

and

```
token WORD \S+
```

are considered as equivalent. We will be following this convention in all of our subsequent examples.
The `plccmk` script calls the `plcc` translator on the `grammar` specification file. The `plcc` translator takes the `Token.template` file in the `Std` directory and modifies it using the grammar specification, creating a Java source file `Token.java` in your Java subdirectory. It also copies the `Scan.java` file in the `Std` directory into your Java subdirectory. The `plccmk` script then compiles these two Java programs. If you have made any mistakes in your `grammar` file, these mistakes may show up during translation (with the `plcc` program) or during compilation of the Java source files.

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 source of characters to be read by the scanning process. A `BufferedReader` can be constructed from a `File` object, from `System.in`, or from a `String`. The `Scan` program reads characters from this `BufferedReader` object line-by-line, extracts tokens from these lines, and delivers the current token with the `cur()` method – `cur` stands for `current`.

A `Token` object has three public fields: an `enum Val` field named `val` that is the token enum value, one for each of the token specifiers (except for the skip tokens); a string `str` field that is returned by the `toString()` method; and an integer `lno` field that is the line number, starting at one, of the input file where the token appears.

The `adv()` method advances the scanning process so that the token returned by the next call to `cur()` will be the next token in the input. Notice that multiple calls to `cur()` without any intervening calls to `adv()` will all return the same token.
For example, consider a grammar file with the following lines:

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

In this case, a token that matches the ID pattern will have token enum value named ID, and a token that matches the NUM pattern will have token enum value named NUM. Once the plcc program is run on this specification, the Token.java file in the Java directory will have a public inner enum class named Val whose elements will consist of the following identifiers and associated patterns:

```plaintext
NUM ("\\d+")
ID ("[A-Za-z]\\w*")
```

Any Java file that needs to use either of these enum values can refer to them symbolically as Token.Val.ID and Token.Val.NUM.

The Scan class method `cur()` returns a Token object. This method is designed to be lazy: If a token needs to be gotten from the input stream, calling `cur()` will get the token and return it. If you call `cur()` again, it will return the same token. The `adv()` method tells the Scan object to force the next `cur()` call to get and return the next token instead of returning the same token. The `cur()` call returns null if there are no more tokens left in the input stream.
Here is a simple loop that will read and print all of the tokens from a `Scan` object named `scn` that has been constructed from some input stream:

```java
while (true) {
    Token t = scn.cur();
    if (t == null)
        break;
    System.out.println(t.toString());
    scn.adv();
}
```

Without the `adv()` call, this loop would perpetually print the first token in the input stream. This loop is similar to the `printTokens()` method in the `Scan` class.

One final note: If the scanner encounters input characters that cannot be matched by any of the `skip` or `token` specifications, it will throw an exception.