Programming Language Concepts
CSCI344

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*. Reading a program to isolate its tokens is called *lexical analysis*.

Lexical Analysis, Syntax, and Semantics

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 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 and other examples can be run in the `JNotes/0Examples` directory.

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 tells 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* (a behavioral specification) 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. 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 in 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 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 describes:

1. the rules that dictate how to write a program in the language so that it is syntactically correct, and

2. the behavior of a program when it is run (whether or not its behavior is what you *want* is a different matter).

Along the way, we show how to implement programming language behaviors (semantics) with regard to specifics such as how variables are bound to values, how functions are defined, and how parameters are passed when functions are called.

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 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 determine its constituent parts: 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 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.

Tokens (continued)

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
```

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. This is similar to what we do with `hasNext()` and `next()` for Scanner objects in Java's `java.util.Scanner` library, where the tokens are `String`s.

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. Be warned, however, that our lexical analysis process doesn't work *exactly* like `hasNext()/next()`.

Lexical analysis is sometimes referred to as *scanning*, and a *scanner* is a program that carries out this process.

Tokens (continued)

For the languages we deal with in this course, we 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 we do is to specify what characters do *not* appear in a token. Typically, tokens do not have whitespace – spaces, tabs, and newlines – so these characters must be skipped. We express these skipped characters using a notation such as

```
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 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.

Tokens (continued)

To specify a particular token, we give the token a 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.

Tokens (continued)

Whenever we are faced with two possible token specifications (rules) that match a string upon input, we *always choose the longest possible match*. So if the input stream contained the string

```
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.

If two or more token specifications match the same input (longest match), we always *choose the first specification in our list that matches*.

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 in the list of rules.

Writing a scanner is somewhat involved, so we have provided you with a tool set that produces a Java scanner automatically from a file that specifies the tokens using regular expressions. This tool set, named `PLCC`, consists of a program written in Python 3 along with some support files. `PLCC` stands for a "Programming Languages Compiler Compiler". You should be able to use this tool set 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 Ubuntu lab systems in this directory:

```
/usr/local/pub/plcc/PLCC
```

This directory also contains a shell script called `plcc` that runs the `plcc.py` Python program, along with a script called `plccmk` that also compiles the Java programs created by PLCC.

If you are working on one of our Ubuntu lab systems, you can simply run `plcc` or `plccmk` to process the various languages we proceed to specify.

If you are working on some other system (like a Mac or Windows), copy the PLCC files and directories into a suitable directory structure on that system where you intend to do your work. If you do so successfully, you may want to share your experiences with others, but you are on your own.

When running `plcc`, you need to give the name of the language specification file on the command line. For example:

```
plcc grammar
```

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 filename `grammar` is typically used, but you can name your file anything you wish, so that both of the following examples are acceptable:

```
plcc mygrammar
plcc foo
```

The language specification file can contain comments starting with a '#' character and continuing to the end of the line. These comments are ignored by `plcc`.

The `plccmk` script runs `plcc` on a language specification file whose name must be `grammar`. In addition to creating a `Java` subdirectory and depositing the `Scan.java` program in that directory (along with a few other necessary Java support files), `plccmk` compiles the Java programs in that directory. Initially, our language specification file contains only token specification rules (as we are doing here). Later, we will use this file to define language syntax and semantics. For now, we concentrate only on token specifications.

Tokens (continued)

Here are some example specification files that you can try. 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, 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 expects input from standard input (your keyboard) and produces output lines that list the tokens as they are scanned, in the form

```
NAME: string
```

where `NAME` is replaced by the token name and `string` is replaced by the token's corresponding lexeme from the input file that produced this token.

Tokens (continued)

After running the `plccmk` command, a `Java` subdirectory is created, populated with the following Java source files

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

as well as a few other Java support files. 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 token specifications in your `grammar` file are translated into Java code that associates the token names with their corresponding token patterns, and similarly for skip patterns.

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.

Tokens (continued)

The `plccmk` script calls the `plcc.py` 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 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 `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 three public fields (also called *instance variables*): 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 the token's lexeme derived from the input stream (and is returned by the `toString()` method in this class); and an integer `lno` field that is the line number, starting at one, of the input stream where the token appears.

For the purposes of compatibility, the `Scan` class also defines methods `hasNext()` and `next()` that behave exactly like their counterparts in the Java `Scan` 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.

For example, consider a `grammar` file with the following lines:

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

In this case, a token that matches the `ID` pattern has token enum value named `ID`, and a token that matches the `NUM` pattern has 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 consist of the following identifiers and associated patterns:

```
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()` is designed to be *lazy*: If a token needs to be gotten from the input stream, calling `cur()` gets the token and returns it. If you call `cur()` again, it returns 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.

Tokens (continued)

Here is a simple loop that reads and prints all of the tokens from a `Scan` object named `scn` that has been constructed from some input stream:

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

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

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