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 pro
guage refers to the rules used to determine the structure of a prog
the language. *Syntax analysis* is the process of applying these rule
the structure of a program. A program is *syntactically correct* if
syntax rules defining the language. Every programming language h
(these rules differ from one programming language to another) that
programming language specification.

Before we can specify the syntax rules of a programming langu
specify the *lexical* (from a Greek word meaning "word") structure o
the symbols used to construct a program in the language. These sym
tokens. *Lexical analysis* is the process of applying these rules by re
input and isolating its tokens. Tokens comprise the "atomic structure

Lexical analysis is also called *scanning*. You can think of scannir
do when you "scan" a line of printed text on a page for the words
text. Programming language tokens normally consist of things su
("23" or "54.7"), identifiers ("foo" or "x"), reserved words ("for",
punctuation symbols (".", "["). Every programming language has r
the tokens in the language (these rules differ from one programmi
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 lexica
particular language. Such a program is also called a *scanner*, *toke*
The input to a scanner is a stream (sequence) of characters, and
stream of tokens. The behavior of a scanner for a language is define
specification of the language.

A *syntax analyzer* is a program or procedure that carries out syntax
particular language. Such a program is also called a *parser*. The in
is a stream of tokens (produced by a scanner), and its output is a *pa*
an abstract representation of the structure of the program. The beha
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 *le*
ample, when you read a word (token) from printed text on a page
collection of characters that make up the word is its lexeme. In this
first lexeme is "the" (ignoring case), consisting of the individual lett
'e'. This lexeme is an instance of an English part of speech called a
this case, "article" is the token and "the" is the instance. The other i
"article" token (in English) are "a" and "an". **A token is an abst**
lexeme is an instance of this abstraction.

Lexical Analysis, Syntax Analysis, and Semantic Analysis (conti

The *semantics* (from a Greek word meaning "meaning") of a pro
guage refers to the rules used to determine the meaning of a progran
language. Here "meaning" refers to (a) whether the program makes
what the program does when it is run. In languages such as English,
make sense (such as "the dog ate the bone") but it may not have a
terms of what it "does". Programs are expected to "do" something,
"do" is part of their semantics. *Semantic analysis* is the process of
rules to a program written in the language to determine its meaning.

A *semantic analyzer* is a program or procedure that carries out sen
The input to a semantic analyzer is a parse tree. The output is
execution of the resulting program (as defined by what the progr
it is run) or some intermediate form (such as machine code) that
some other time. Direct execution is called *interpretation*, wherea
intermediate form is called *compilation*. We will use the interpret
in these notes, though the techniques we describe apply as well tc
approach.

Lexical Analysis, Syntax Analysis, and Semantic Analysis (conti

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

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

This course is about programming language syntax and semantics, w
sis on semantics. Syntax doesn't matter if you don't understand sem

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

```
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 Pythor
comes to integer division?

Syntax and semantics

You can use a language *compiler* to tell you if a program you write correct, but it's much more difficult to determine if your program al the behavior you *want* – that is, if a program is semantically "corr two basic problems:

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

Of course, **a program is its own specification**: it behaves exactly structions say it should behave! But nobody knows exactly how to *ioral specification* that precisely and unambiguously describes wha part because what you want is often imprecise and ambiguous – late this behavioral specification into a program whose semantics *pr* according to the specification. Because of this, programming will a lematic. (Creating behavioral specifications is a topic of interest i and properly belongs in the discipline of software engineering.)

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

Syntax and semantics (continued)

This course is about programming languages, and particularly about gramming languages. A programming language *specification* is a 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 desc and syntax structure of a number of languages and how to implemen behaviors (semantics). We show how variables are bound to values functions, and how parameters are passed when functions are called

Because a program in a particular language must be syntactically co semantic behavior can be determined, part of this course is about syn final anaysis, semantics is paramount.

Tokens

Assume that we have a program written in some programming lan
of languages such as C, Java, Python, and so forth.) The first step i
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
typically ignored (for example, "whitespace", including spaces, tabs,
while some characters group together to form things that can be i
example) as integers, floats, and identifiers. Some specific characte
meaningful in the language, such as '`class`' and '`for`' in Java. S
characters are meaningful, such as parentheses, brackets, and the
while some pairs or characters are meaningful such as '`++`' and '`<`
term *token* to refer to such atoms.

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

Tokens (continued)

Lexical analysis takes character stream input and produces token stre
example, if a language knows only about integers and the dot symb
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 produ
token, with the following lexeme:

```
23.587
```

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

Tokens (continued)

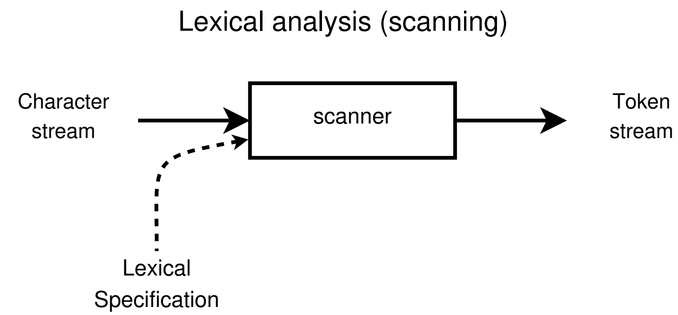
The purpose of lexical analysis is to take program input as a strear and to produce output consisting of a stream of tokens that conforr *specification* of the language.

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

By a *stream of characters* we mean a stream of items consistin characters in a character set such as ASCII or UTF-8. By *strean* mean a stream of items that are tokens defined by a token specificati analysis process provides a `hasNext()/next()` interface for stream of tokens, but it also provides a similar `cur()/adv()` inte in more detail below.

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

Tokens (continued)



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

A programming language specification in PLCC is a text file n `grammar`. Each language that we describe in this course has its file. A `grammar` file has three sections: the lexical specification, th fication, and the semantic specification. These sections appear in the in this order: first lexical, then syntax, and finally semantic. A line single '`%`' separate the sections.

We start by describing the structure of the lexical specification secti

Tokens (continued)

The lexical specification section of a `grammar` file uses *regular* specify language tokens. A regular expression is a formal descripti that can match a sequence of characters in a character stream. Fc regular expression '`d`' matches the letter `d`, the regular expressior any decimal digit, and the regular expression '`\d+`' matches one oı digits. **You should read the Java documentation for the** `Patter` **formation about how to write regular expressions.**

When specifying tokens, we must identify what input stream charac long to tokens and should be skipped. Typically, we skip whitespac and newlines. We identify these skipped characters in the `gramm` *skip specification* line like this :

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

The regular expression '`\s`' stands for "space" (the space charac newline), and the regular expression '`\s+`' stands for one or mo use the symbolic name "WHITESPACE" to identify this particula characters to be skipped. (Any symbolic name would suffice, but it use one that describes its purpose.) We also typically skip comment

Characters to be skipped during lexical analysis are not tokens.

Tokens (continued)

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

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

where <TOKEN NAME> identifies the token (it must be in ALL C is a regular expression that defines the structure of the token's lexem

We adopt one simplifying rule for *all* the languages we discuss in th *cannot cross line boundaries*. Be warned, however, that not all pro guages conform to this rule.

For example, we may use the following lines in our `grammar` fi *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 r they represent.

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

Tokens (continued)

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

```
procedure
```

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

If two or more token specifications match the same input (longest
ways *choose the first specification line in the* `grammar` *file that mat*
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 spe**
   appears in the `grammar` **file.**

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

Tokens (continued)

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

The `plcc.py` Python program and the `Std` subdirectory that cont
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` th
`plcc.py` program with input from a `grammar` language specific
mally you will run `plccmk` in a directory that contains this specific
`plcc.py` program produces a collection of Java programs in a sub
`Java`. The `plccmk` script then compiles these Java programs (usin
piler). When you are working on one of our Ubuntu lab systems,
run `plccmk` to process the various languages we will specify in thi

See the `HOWTO.html` file in `/usr/local/pub/plcc/tvf` fo
formation about how to set up your CS account environment so tha
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 th
   want to implement.
2. In this working directory, put your `grammar` language specifi
   we describe later, you may also put additional `grammar`-relat
   directory.
3. If you wish, create subdirectories containing test files that you c
   cise your language implementation.
4. Run `plccmk` in your working directory. This will create a `Jav`
   containing Java source files for an interpreter for your language.
5. If `plccmk` produces any PLCC or Java compile errors, edit your
   to fix these errors and repeat the above step.

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

Tokens (continued)

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

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

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

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 workin
the given contents. Then, in this directory, run the following comma

```
plccmk
```

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

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

along with some additional support files.

Examine the `Token.java` file in the `Java` subdirectoryto see ho
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 for

```
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 `toke`
the `skip` term). This means both

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

and

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

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

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

Tokens (continued)

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

A `Token` object has four public fields (also called *instance*
`enum Match` field named `match` that is the token's symb
`String str` field that is the token's lexeme derived from the inp
is returned by the `toString()` method in this class); an `int l`
the line number, starting at one, of the input stream where the toke
a `String line` field that is the source line of the input stream w
appears.

For the purposes of compatibility, the `Scan` class defines method
and `next()` that behave exactly like their counterparts in the Java `S`
The `boolean hasNext()` method returns `true` if and only if th
has additional tokens, in which case the `Token next()` method re
sumes) the next `Token` object from the input stream.

Tokens (continued)

For example, consider a `grammar` file (directory `IDNUM`) with the
cal 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 `Toke`
Java subdirectory having a public inner `enum` class named `Match` v
consist of the following identifiers and associated patterns:

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

Any Java file that needs to use the `enum` values `NUM` and `ID` can refe
bolically as `Token.Match.NUM` and `Token.Match.ID`.

Running the `Scan` program in the Java directory takes character str
standard input (typically your keyboard) and prints all of the resu
standard output (typically your screen), one token per line. Each pr
the line number where the token appears, the token name (`NUM` or `I`
ple), and the lexeme (printed in single quotes). Any input that does
of the skip or token specifications prints the representation of an `$E`

Tokens (continued)

The `printTokens()` method in the `Scan` class produces the ou
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.toS
        }
        System.out.println(String.format("%4d: %s",
    }
}
```

When invoked with no command-line arguments, the `main()` meth
class calls the `printTokens()` method on a `Scan` object constru
dard 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 na an uppercase letter, these “tokens” cannot be confused with language symbols.

The `$ERROR` “token” is produced when the scanner encounters acter that does not match the beginning of any skip or token spe `toString()` value of this token is of the form `!ERROR(...)`, w part displays the offending character. In the context of syntax anal cover later), such a character cannot be part of a syntactically corre syntax analysis will terminate with an error.

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