UNIVERSITY OF CALIFORNIA
Department of Electrical Engineering
and Computer Sciences
Computer Science Division

CS 164 **P. N. Hilfinger**
Spring 2005

CS 164: Programming Languages and Compilers
Class Notes #2: Lexical*

1 Introduction

The purpose of *syntactic analysis* is to analyze textual input so as to confirm that it is *syntactically well-formed*—that it obeys certain general structural rules dictated by the specification of the input language—and to convert it into a form that gives later parts of the compiler convenient access to this structure.

For example, we might say that in Java a conditional statement can have the form

if
(*Expression*) *Statement* **else** *Statement*

In later parts of the compiler, the programmer might reasonably want a data structure that represents "an **if** statement" and that provides operations that return "the **then** clause," "the **else** clause," and "the test" from this statement. These operations would be awkward to implement if the data structure used were simply a copy of the original text of the statement (a string). Instead, a tree-like form is a better representation. This requires analyzing the original text into its constituent grammatical parts.

This task is traditionally broken down into *lexical analysis*— which breaks the text down into the smallest useful atomic units, known as *tokens*, while throwing away (or at least, putting to one side) extraneous information, such as white space and comments—and *parsing*—which operates on tokens and groups them into useful grammatical structures. There is no sharp distinction between these two activities—I am happy to classify both under "syntactic analysis." A single monolithic subprogram could handle both simultaneously, as was done in very early compilers. To a certain extent, we divide the tasks as we do to accommodate certain techniques and certain automatic or semi-automatic tools.

We're going to start with lexical analysis. The part of a compiler that performs this task is called a *lexical analyzer, tokenizer,* or *scanner*. In brief outline,

*Parts of these notes are adapted from material by Alex Aiken, George Necula, and Ras Bodik

- *Regular expressions* can describe a variety of languages (sets of strings), including the set of atomic symbols of a typical programming language.
- *Finite-state automata* (FSAs) are abstract machines that also recognize languages.
- *Deterministic finite-state automata* (DFAs) are a subset of finite-state automata that are easily converted into programs.
- There exists a translation from regular expressions into FSAs.
- There exists a translation from FSAs that happen to be nondeterministic into DFAs (and hence into programs).
- The total process of conversion from regular expression to program is automatable. In fact, we'll be using a couple of handy programs: FLEX (for producing scanners written in C or C++) and JFLEX (for producing scanners written in Java). These programs are really compilers themselves, translating succinct descriptions of programming-language syntax (a piece of it, anyway) into programs that "execute" these descriptions to extract tokens from the input.

2 Tokens and tokenizing

In the context of programming-language translation we use the term *token* to refer to an abstraction for the smallest unit of program text that it is convenient when describing the syntax of a language. You don't want them to be too small. The parsing techniques we'll use in this class are designed to decide on what to do next on the basis of the next token of input. If tokens are single characters, they won't generally contain enough information to make this decision. For example, suppose a program has seen the characters '`x+y`' and the next character is a blank. This is insufficient information to determine whether '`x+y`' is to be treated as a subexpression, since if the next non-blank character is '`*`', then `y` should be grouped with whatever is after the asterisk. The lexer, on the other hand, can first eliminate whitespace, making the decision easier. Another example is '`x+y`' followed by a '`+`'. Here, the decision depends on whether the character immediately after the '`+`' is another '`+`'. If the lexer has previously grouped all '`++`'s into single tokens, the decision is easily made, with no *ad hoc* scanning ahead in special cases. Tokenizing is thus the process of bridging the gap between the input (made of characters—too small) and tokens.

As an example, a Java program (a Java *source file*) might contain the phrase

```
if(i== j)
    z = 0;  /* No work needed */
else
    z= 1;
```

which a translating program sees as a sequence of characters:

```
\tif(i== j)\n\t\tz = 0;  /* No work needed */\n\telse\n\t\tz= 1;
```

The job of the scanner is to convert this to a sequence of values such as this:

```
IF, LPAR, ID("i"), EQUALS, ID("j"), RPAR, ID("z"), ASSIGN,
INTLIT(""), SEMI, ELSE, ID("z"), ASSIGN, INTLIT(""), SEMI
```

Here, the upper-case symbols denote *syntactic categories* (often internally represented as integers in an actual compiler). The syntactic categories are consumed by the next stage of the compiler—the parser. When determining the structure of a program, it is not particularly important *which* identifier or integer literal appears at some point; the important point is that *some* identifier appears there. Hence, scanners typically separate the syntactic category from what I'll call the *lexical value* of the token (shown in parentheses), which gives information that the parser doesn't need, but later stages of the compiler will. In our example, the lexical value of an identifier happens to be the *lexeme* itself—the character string constituting the token.

As you can see from the example, information unnecessary to the rest of the compiler is filtered out entirely. All the blanks, tabs, newlines, and comments are gone, so that the little discrepancies in spacing around the operators are removed. This is a typical pattern in the translation process: each stage makes the job of its successors easier by removing "noise" and guaranteeing that certain errors are filtered out.

Actually, real scanners don't entirely do away with whitespace. If it is going to produce useful error messages, a compiler must keep track of where each token appears so that it can "point" at the offending part of the program in the original source file. Therefore, tokens often contain positional information—but just like the semantic value, it is separated from other parts of the token so that it can be referenced only when needed.

If we were building an actual scanner in Java, our tokens might be represented by objects with fields like this:

```
class Token {
   enum SyntacticCategory { IF, LPAR, ID, EQUALS, RPAR, ASSIGN, ... };
   SyntacticCategory syntax;
   Object value;
   Location sourcePosition;
   ...
}
```

3 Implementation

The rest of these notes come from the Fall 2004 edition of the course, courtesy of Alex Aiken, Ras Bodik, Richard Fateman, and George Necula.