Building Recognizers By Hand

This is the most important chapter in the book because it bridges the gap between the skills of the average programmer and basic language recognition technology. For a discussion of language recognition and translation to be accessible to the average programmer, it must begin with the basic tools common to all programmers: a programming language and a file input/output library.[1] If you understand Java syntax, are familiar with object-oriented principles, and know a few classes in the `java.io.*` package, you will be able to build small language recognizers by the end of the chapter. Beginning with a simple while-loop, you will learn:

- what it means to recognize structure and how to construct a simple recognizer
- when, why, and how recognizers use lookahead to distinguish between various input substructures
- that most input is best analyzed on two structure levels and how you can build two pipelined recognizers to solve a single recognition problem.

By the end of this chapter, you will not know everything about building recognizers by hand, but you will have the basis to grasp the next chapter, which explains the different types of languages, how you can use grammars to succinctly describe languages, and how you can use tools such as ANTLR to automate recognizer construction.

Recognizers and Structure

Imagine a file of random characters. To read it into memory, you could do the following:

```
BufferedReader f = new BufferedReader(new FileReader("random.txt"));
int c = f.read(); // get the first char
StringBuffer s = new StringBuffer();
while ( c != -1 ) {
  s.append((char)c);
  c = f.read();
}
```

You might say that the file has no structure or that the structure is simply "one or more characters." Now, consider an input file that contains a series of letters followed by a series of digits such as

```
acefbetqd392293
```

You could read in this slightly structured input with something like

```
BufferedReader f =
  new BufferedReader(new FileReader("lettersAndDigits.txt"));
int c = f.read(); // get the first char
StringBuffer letters = new StringBuffer();
StringBuffer digits = new StringBuffer();
while ( Character.isLetter((char)c) ) {
```

[1] Besides, as my graduate advisor Hank Dietz at Purdue told me in 1987, "it's no fair using parser generators like yacc if you don't know how to build parsers and parser generators."

```
  letters.append((char)c);
  c = f.read();
}
while ( Character.isDigit((char)c) ) {
  digits.append((char)c);
  c = f.read();
}
```

On the other hand, the previous simple loop would also read in the letters and digits. The difference lies in that the previous loop would not *recognize* the structure of the input. Recognizing structure involves comparing the input against a series of constraints as dictated by the structure. In this example, recognition implies that your program verifies that the input consists of letters followed by digits. The previous loop verifies no constraints. Another way to think about recognition is that, after recognizing some input, your program can provide the groups of characters associated with the various elements of the structure such as the letters and digits in this case. Simply munching up all the input does not partition the input in any way.

You can look at most structured input on two levels: the character level and the overall file structure level. You can see the obvious character structures contained in the previous input file: "a sequence of letters" (an identifier) and "a sequence of digits" (an integer). The overall file structure is then "an identifier followed by an integer."

The various character-level structures make up the vocabulary of your language and the sequence and grouping of vocabulary symbols represent the overall file structure. In the parsing world, you call either of these structures *languages*. A language is essentially just a set of valid input sequences (*sentences*) of vocabulary symbols. Character-level languages use individual characters as vocabulary symbols and overall-structure languages generally use groups of character sequences as vocabulary symbols. For example, in English, words in the dictionary and punctuation comprise the character-level language, the vocabulary. On the higher level, the English Language grammatical structure describes the meaningful sequences of vocabulary words, or valid sentences (except for those of American teenagers). Being able to consider the two structure levels independently makes it easier to build recognizers for them, as you will see later.

For the most part, you refer to the overall structure of your input as the language or *syntax* and the character-level structure as the vocabulary even though the vocabulary is itself a language (sometimes called the *lexical* language). The vocabulary symbols are most often called *tokens*. To *recognize* or *match* a character or token is to examine (looking for a mismatch) and *consume* that item, thus, removing it from further consideration by the recognizer.

To better illustrate the two levels, consider the structures associated with a sequence of variable assignments in a fictional language called AL:

```
width=200
height=100
```

At the overall structure level, you see a sequence of assignments where an assignment is an identifier followed by an equals sign followed by an integer followed by a newline. The four vocabulary symbols (the assignment operator, the identifier, the integer, and the newline) represent the character-level language.

How do you begin constructing a recognizer? First, assume that there exists some basic infrastructure such as an instance variable called c that always contains the next character to match. Next, begin writing methods to match the substructures of the language. Start at the most abstract level and work your way down to the character-level structures. The terms used to

specify a language usually guide the names of the methods and the substructures the methods recognize. Recognizers that begin the recognition process at the most abstract language level are called *top-down* recognizers and recognizers composed of mutually-recursive methods fall into a subcategory called *recursive-descent* recognizers.

To build a recursive-descent recognizer for AL, you know that an input file consists of one or more assignments:

```
void inputFile() { // overall structure method
  while ( c != -1 ) {
    assignment();
    consume(); // consume the newline
  }
}
```

Leave the recognition of assignments themselves to a routine of that name. You know that assignments are an identifier followed by '=' followed by an integer. Again staying as abstract as you can, leave recognition of the actual integers and identifiers to other methods; focus on the sequence inherent in the structure of an assignment.

```
void assignment() { // overall structure method
  id = identifier();
  consume(); // consume the '='
  i =  integer();
}
```

After finishing the overall file structure, continue downwards in abstraction to the character-level structure. An identifier is a series of letters and an integer is a sequence of digits:

```
String identifier() { // character-level structure method
  StringBuffer s = new StringBuffer();
  while ( Character.isLetter((char)c) ) {
    s.append((char)c);
    consume();
  }
  return s.toString(); // return text matched for identifier
}

String integer() { // character-level structure method
  StringBuffer s = new StringBuffer();
  while ( Character.isDigit((char)c) ) {
    s.append((char)c);
    consume();
  }
  return s.toString();
}
```

For completeness, character-level methods return the text matched for the associated structures.

Notice that the language recognizer asks for the recognition of vocabulary symbols "on demand". That is, when the language recognizer method assignment() decides it needs to match an identifier, it calls method identifier(), a part of the lexical language recognizer, to actually recognize that vocabulary structure on the character input stream. The tight coupling of the recognizer methods for the language and lexical levels is easy to understand but has serious limitations, as you will learn later.

Once you have the core methods of a recognizer ready, create a class definition whose constructor primes the recognizer by loading instance variable c with the first character from the input stream:

```
import java.io.*;
class ALAssignments {

private int c;
private Reader input;

public ALAssignments(Reader f) {
  input = f;
  consume(); // get first char of file
}

void consume() {
  try {
    c = input.read();
    if ( c=='\r' ) { // normalize \r\n to \n
      c = input.read();
    }
  }
  catch (IOException io) {
    System.err.println(io.getMessage());
    System.exit(1);
  }
}
void inputFile() {…}
void assignment() {…}
String identifier() {…}
String integer() {…}
}
```

Consistency is the name of the game when building recognizers by hand. Linefeed normalization is one example. The consume() method not only gets characters from the input stream, setting instance variable c, but also normalizes the PC carriage-return linefeed sequence to linefeed. Another important convention is that variable c always holds the next character to recognize upon return from a recognizer method. That way you never have to wonder what state the input stream is in and you can test c to guide your recognizer. For instance, method identifier() consumes until c is no longer a letter and returns with c as the first character beyond the identifier. On a smaller scale, each while-loop leaves variable c loaded with the next character.

To recognize a stream of AL input with class ALAssignments, open a stream, create a recognizer, pass the stream to the recognizer, and call the method matching the most abstract language level construct: inputFile().

```
public static void main(String args[]) throws IOException {
  BufferedReader f =
    new BufferedReader(new FileReader("ALAssignments.txt"));
  ALAssignments recog = new ALAssignments(f);
  recog.inputFile();
  f.close();
}
```

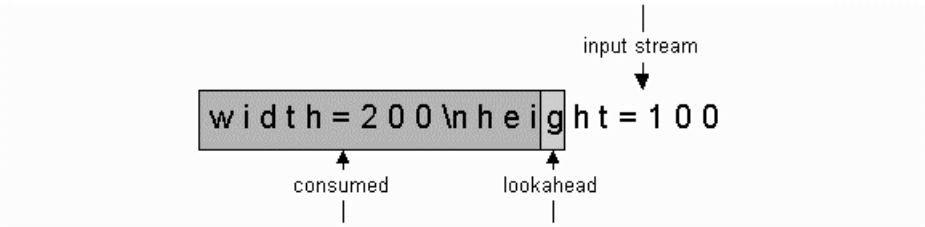
###tie this in better; lexer lookahead used to know when to terminate token recog…say that parser can do same.

Recognizer Lookahead

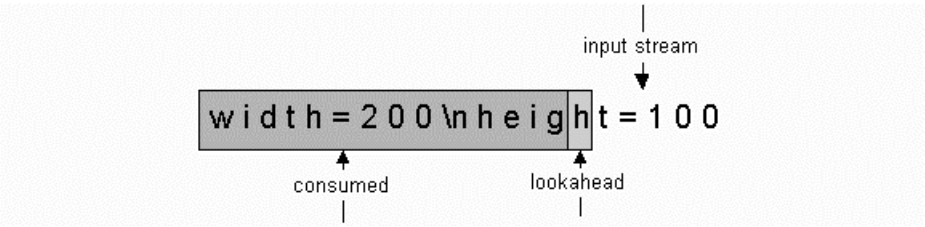
The lexical methods such as identifier() consume characters until variable c no longer holds a letter:

```
while ( Character.isLetter((char)c) ) {…}
```

The loop condition examines c, but it is not until the body of the loop that the recognizer consumes the character, effectively recognizing it. You might say that the recognizer is looking ahead to decide whether to continue grouping characters into an identifier. The following figure illustrates the state of the recognizer when c is 'g' at the while-loop condition.



The recognizer has read the character 'g' from the input stream, but has not yet recognized it. Upon entry into the while-loop body, the consume() method reads another character into c, removing the 'g' from further consideration. The following figure illustrates the state of the recognition process after calling consume().



A recognizer partitions the input into three parts:

1. The characters already recognized (consumed) by the recognizer.
2. The lookahead window. The recognizer has read this character from the input stream, but has not yet recognized it.
3. The characters still on the input stream that the recognizer has not read.

The lookahead window contains candidates for recognition. A recognizer uses them to predict what substructure is coming down the input stream. If the lookahead finds that the lookahead does not fit into a sentence of the language, a recognizer should report an error. A recognizer may test the same lookahead character many times as control-flow is routed to the appropriate recognition method. Lookahead operations have no side effects because they examine input without consuming.

Top-down recognizers are considered *predictive* because they attempt to predict which sentence structure is coming down the input stream by examining only a piece of the input: the character or characters in the lookahead window.

An A-maze-ingly Useful Metaphor…

To better understand the need for recognizer lookahead, consider having to navigate an utterly dark maze. If the maze had no forks (decision points), successfully traversing the maze would be easy, as you would have no choice about which path to follow. On the other hand, if the maze had many forks, you would probably have to do a lot of backtracking in the dark to successfully navigate the maze and reach the exit.

Suppose the maze had letters stamped on the floor and that each successful path through the maze spelled a different word by combining the letters along that path. Any invalid path would therefore compose an invalid word. Now, armed with a flashlight and one of the "passwords", you could find your way through the maze by comparing the letters illuminated on the floor by your flashlight with the sequence of letters in your password. Each decision point presents a choice of letters, one from the start of each path. The next letter in your password dictates which path to take.

What do you do when the same letter appears at the start of more than one path emanating from a fork? If you do not have a bigger flashlight with which to see more than a single letter down each path, you cannot make a decision.[2] A maze with many similar passwords requires a big flashlight.

The list of passwords represents a language (the set of valid symbol sequences). The maze and the adventurer together form the recognizer for that language, which continuously compares the input characters (the password) against the valid patterns. In object-oriented parlance, you can think of the maze as the code or behavior of the recognizer and the adventurer and its position as the state. Finally, the flashlight represents the lookahead of the recognizer.

As the similarity of a language's sentences increases, so does the complexity of a recognizer for those patterns. For example, in C, many declarations are very similar to expressions; "f()" could be the start of a function declaration or a function call. The more of a C phrase you can see and the more you know about C phrase structure, the more easily you can distinguish between declarations and expressions. The strategy of the recognizer and the amount of available lookahead prescribe the strength of the recognizer and, hence, the types of languages it can handle.

[2] Naturally, if you are willing to backtrack you do not need to see ahead—you simply try each path until you find the correct one.

In AL thus far, only the lexical methods need lookahead to recognize structure. Recognizing the structure of variable assignments does not require prediction on the recognizer's part. In other words, you know exactly what is followed by what because an assignment is a fixed sequence and there are only assignments in the language.

When your language has more than a single sentence structure that it can match, your recognizer must make decisions. Adding comments to AL, forces a decision in both the language and the vocabulary methods. Now, the language allows a sequence of either assignments or shell-style comments.

```
# sample input
width=200
height=100
```

The inputFile() method must accommodate the alternative substructures (assignment or comment) by making a decision based upon lookahead:

```
void inputFile() {
  while ( c != -1 ) {
    if ( c=='#' ) { // decision
      comment();
      consume(); // scarf newline
    }
    else {
      assignment();
      consume(); // scarf newline
    }
  }
}
```

As with identifier() and integer(), method comment() must use lookahead variable c to find the end of the structure (end of line):

```
String comment() {
  StringBuffer s = new StringBuffer();
  while ( c!='\n' ) {
    s.append((char)c);
    consume();
  }
  return s.toString();
}
```

Notice that the '#' character is always tested twice when it sits in the lookahead window—once at the language level in inputFile() and once at the lexical level. Upon seeing the '#' character, the recognizer decides that a comment will appear next and control-flow is routed to the comment() method.

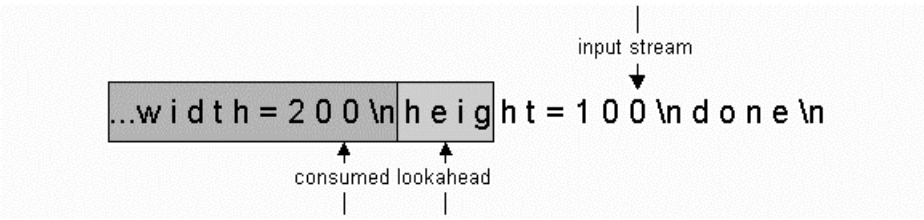
Using More Than One Lookhead Character

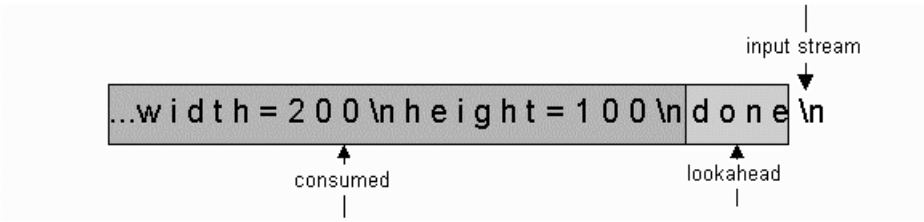
A recognizer may use more than a single symbol of lookahead, denoted k>1. Consider adding a keyword such as "done" to language AL:

```
# sample input
width=200
height=100
done
```

How will inputFile() decide whether to match the simple sentence "done" versus an assignment? Because "done" is lexically an identifier, the prediction mechanism cannot use the same

prediction expression to decide whether to call identifier() or a method such as keyword_done(). The prediction operation must look four characters ahead to see if the special case of an identifier, the keyword "done", is coming down the input stream. The following figures demonstrate the state of a recognizer with k=4 lookahead characters at the start of the second assignment and at the start of the done statement.





When the lookahead is "heig", the recognizer knows by the first character that the "done" statement is not approaching, but the identifier for an assignment could be "donate" or "donotread" (requiring four characters to uniquely identify the "done"). For the general case, the recognizer needs a decision of the form:

```
if ( LA(1)=='d' && LA(2)=='o' && LA(3)=='n' && LA(4)=='e' ) {
  keyword_done();
}
else {
  assignment();
}
```

where LA(i) is some method returning the ith character of lookahead. The logical AND operator behaves such that the condition LA(1)=='h' will efficiently prevent the other three comparisons from being computed.

How do you look four characters ahead without consuming the next three characters? You must buffer the characters. The following recognizer for AL uses class CharBuffer, which is explained below. The recognizer uses a lookahead depth of 4 and replaces references to variable c with LA(1) to obtain the current character of lookahead. Method consume() delegates character consumption to the CharBuffer with the added benefit that the recognizer is isolated from the source of characters. Additionally, the use of method match() makes it more obvious what character the recognizer is trying to consume. The overall code framework and strategy remain the same—only the manner of obtaining characters has changed.

```
import java.io.*;
class ALAssignmentsCommentsAndDone {
private CharBuffer input;
private static final int LOOKAHEAD_DEPTH = 4;

public ALAssignmentsCommentsAndDone(Reader f) {
  input = new CharBuffer(f, LOOKAHEAD_DEPTH);
  // no need to prime lookahead: char buf will handle that
}

// delegate lookahead and consume to the char buffer
private int LA(int i) { return input.LA(i); }
private void consume() { input.consume(); }

/** Match a character; just consume for now.  Don't do
 *  error checking yet.
 */
private void match(int c) { consume(); }

void inputFile() {
  while ( LA(1) != -1 ) {
    if ( LA(1)=='#' ) { // only need k=1 here
      comment();
      match('\n');
    }
    else if ( LA(1)=='d' && LA(2)=='o' && LA(3)=='n' && LA(4)=='e' ) {
      keyword_done();
      match('\n');
    }
    else {
      assignment();
      match('\n');
    }
  }
}

void assignment() {
  identifier();
  match('=');
  integer();
}

String identifier() {
  StringBuffer s = new StringBuffer();
  while ( Character.isLetter((char)LA(1)) ) {
    s.append((char)LA(1));
    consume();
  }
  return s.toString(); // return text matched for identifier
}

String integer() {
  StringBuffer s = new StringBuffer();
  while ( Character.isDigit((char)LA(1)) ) {
    s.append((char)LA(1));
    consume();
  }
```

```
  return s.toString();
}

String comment() {
  StringBuffer s = new StringBuffer();
  while ( LA(1)!='\n' ) {
    s.append((char)LA(1));
    consume();
  }
  return s.toString();
}

String keyword_done() {
  match('d');
  match('o');
  match('n');
  match('e');
  return "done";
}
}
```

If asked to characterize this recognizer, you would say that it uses a top-down recursive-descent strategy with a lookahead depth of 4.

Support For k>1 Lookahead Characters

The use of a single instance variable c for k=1 lookahead is the as using a queue of size one. For k>1 lookahead, a queue of size k works similarly. The following class implements an inefficient, but simple fixed-size buffering mechanism. The buffer is an integer array of size k and is always primed with the next k characters. Upon construction of a character buffer then, the first four characters on the stream are available via the LA() method. To consume a character, the buffer shifts left the 2..k lookahead characters to the 1..k-1 positions, throwing the previous LA(1) character away, and reads a new character into lookahead position k.

```
class CharBuffer {
  private int[] buf;
  private Reader f;
  private int k;
  public CharBuffer(Reader in, int k) {
    this.k = k;
    buf = new int[k];
    f = in;
    try {
      for (int i=0; i<k; i++) { // prime buffer
        buf[i]=f.read();
      }
    }
    catch (IOException io) {
      System.err.println(io.getMessage());
      System.exit(1);
    }
 }

  public int LA(int i) {
    if ( i>=1 && i<=k ) {
      return buf[i-1];
    }
    return 0;
  }

  void consume() {
    // shift out oldest char
    for (int i=0; i<k-1; i++) {
      buf[i]=buf[i+1];
    }
    // get new char into kth position
    try {
      buf[k-1] = f.read();
      if ( buf[k-1]=='\r' ) { // normalize \r\n to \n
        buf[k-1] = f.read();
      }
    }
    catch (IOException io) {
      System.err.println(io.getMessage());
      System.exit(1);
    }
  }
}
```

The Limitations of Combined Syntactic and Lexical Recognizers

The recognizers demonstrated so far handle both the language and vocabulary structures together in the same logical program unit. The language recognizer methods also explicitly invoke vocabulary methods for the character-level structures they expect to match. Unfortunately, this easy-to-grasp strategy quickly breaks down for all but the simplest languages. Adding method call syntax to AL demonstrates how the strategy breaks down:

```
# sample input
width=200
height=100
```

```
f(25)
done
```

Method inputFile() uses finite a lookahead depth of four to distinguish between assignments and the "done" statement. How much lookahead is required to see past an identifier to the '=' or the '(' that follows? Theoretically, an identifier can be infinitely long, implying that finite lookahead is not sufficient to distinguish between an assignment and a method call.

There are two solutions to this immediate problem of predicting assignment versus method call. One solution is to *left-factor* out the identifier from the front of those two alternative sentences; the identifier is considered an infinite *left-prefix*. You could modify the recognizer as follows:

```
void inputFile() {
  while ( LA(1) != -1 ) {
    if ( LA(1)=='#' ) { // decision
      comment();
      match('\n');
    }
    else if ( LA(1)=='d' && LA(2)=='o' && LA(3)=='n' && LA(4)=='e' ) {
      keyword_done();
      match('\n');
    }
    else {
      // left-factored identifer()
      identifier();
      if ( LA(1)=='=' ) {
        assignmentMinusIdentifier();
      }
      else {
        methodCallMinusIdentifier();
      }
      match('\n');
    }
  }
}

void assignmentMinusIdentifier() {
  match('=');
  integer();
}

void methodCallMinusIdentifier() {
  match('(');
  integer();
  match(')');
}
```

While the lookahead requirement is finite using this solution, it is not considered very "clean". The methods assignmentMinusIdentifier() and methodCallMinusIdentifier() are counter-intuitive since they are missing the code for the complete alternative substructure and exist solely because of a limitation in your prediction technology. Widespread left-factoring results in recognizers that are hard to implement, difficult to read, and difficult to augment with code to perform translations.

The second way to distinguish between assignments and method calls involves changing the way your recognizer views vocabulary symbols such as identifiers. The idea is to do what you do

when you read these English sentences. This text is a sequence of characters, but your brain applies grammatical structure to the words as if they were complete tokens like Chinese characters no matter how long the words are. For example, the word for the Hawaiian state fish is pretty long: "Humuhumunukunukuapua'a" but you read it as one symbol. Your brain somehow implicitly forms words from the characters and looks them up in a dictionary of sorts using a vocabulary recognizer to provide a vocabulary symbol to your grammar recognizer.

Reading Morse code makes this process even more obvious because it is very clear you are combining dots and dashes representing letters into words before reading the sentence. For example, what does the following say (using International not American Morse code)?

```
._ _. _ ._.. ._.  .. ...  _._. ___ ___ ._..
```

If you guessed "ANTLR is cool" you would be right and are either an obsequious ANTLR fan or a programmer for the NSA (No Such Agency) or the CIA (Christians In Action). A radio operator translates the series of dots and dashes into single characters, then words, then into English sentences. Here is the character by character translation:

```
._ _. _ ._.. ._.  .. ...  _._. ___ ___ ._..
A  N T   L  R  I  S    C  O  O   L
```

While amazing to us lay people, experienced operators can receive and decode Morse code without the help of pen and paper and can generate 20 or 30 words per minute!

Revisiting our fearless adventurer in the maze helps to understand multilevel language structures.

The previous maze had a single structure level. The adventurer matched a password against the letters stamped on the floor to successfully navigate the maze. To demonstrate a two-level structure, imagine a much bigger maze where the paths are so long that the sequence of letters form complete sentences. The adventurer must now match a "pass phrase" composed of **words** against a long sequence of **letters**.

Clearly, there are two problems: matching words and matching sentences. The adventurer must break up the sequence of letters into words. Only then can the adventurer compare the sequence of words on the floor against the passphrase. The difference between levels is starker if the letters on the floor form word puzzles like anagrams rather than English words. In this case, serious decoding/recognition work is required before the adventurer can compare the words against the passphrase.

The adventurer's job is much easier if a sidekick tags along to work on decoding the word puzzles, allowing the adventurer to concentrate on matching English words against the passphrase. This separation between recognizers for the two structure levels is very convenient because the sidekick frees the adventurer from the details of forming English words. For example, given a sidekick that can read a stream of Greek letters and generate a stream of equivalent English words, the English adventurer could navigate a Greek maze.

> ###add something about lookahead at the outer level structure too.
>
> ### can't do Greek sentence structure, which is implied..find something like reading old English words instead of Greek.

Recognizers with Separate Parsers and Lexers

To mimic with a computer the technique your brain uses to recognize sentences, separate the language level processing from the vocabulary level processing into two complete recognizers. The language level recognizer is usually called the *parser* and the vocabulary recognizer is usually called the *scanner, lexical analyzer*, or *lexer*. The only difference between the two recognizers is that the parser recognizes grammatical structure in a stream of tokens and the lexer recognizes structure in a stream of characters. Both perform essentially the same task and you can implement both using the same strategy. In effect, a separated parser/lexer differs from a combined recognizer in that the separated parser asks of the lexer "what token did you find?" rather than "go match this character structure".

The Advantages

The separation of the parser and lexer may seem like an unnecessary complication if you're used to building little recognizers by hand, however, the separation reduces what you have to worry about at each language level. A common engineering principle is to break a difficult task down into manageable subtasks. There are several implementation advantages as well:

- **The parser can treat arbitrarily long character sequences as single tokens**.
- **The parser sees a pipeline of tokens, which isolates it from the lexical language recognizer**. In fact, the parser could be fed a token stream from a previous run of the lexer that the lexer had stored in memory or in a file. In theory, you could even send a modified stream of tokens out of the parser into another parser, thus, pipelining a series of recognizers.
- **Having a separate lexer also simplifies the recognition of keywords** that are lexically identical to identifiers. The lexer can match both identifiers and keywords with the same method and then look up the identifier in a table of keywords returning different token types.
- **The lexer can filter the input, sending only tokens of interest to the parser**. This feature makes it easy to handle white space and other lexical structures that you may want to discard—the lexer simply never passes them to the parser. Further, a separate lexer lets you filter out lexical structures that can appear anywhere such as comments. If comments were passed to the parser, the parser would have to constantly check for comment tokens and filter them out. It is easier to have the lexer filter out as much as possible, leaving the parser to worry only about overall structure.

[### add picture of lexer accepting stream of characters, feeding stream of tokens to the parser.]

Communication between The Lexer and The Parser

A lexer sees a stream of characters that it converts to a stream of tokens for use by a parser. Because of this separation between parser and lexer, the parser no longer goes looking for a particular lexical structure just as the lexer does not look for a particular character. Both recognizers now ask their input streams for the next element, be it a character or a token. The

lexer views the character stream as a sequence of integers identifying the character codes, but the parser must see the result of a series of lexical pattern matches. Consequently, tokens are objects that contain minimally two pieces of information:

1. the text matched for the token by the lexer

2. an integer (set element) indicating which type of symbol was found such as INTEGER or IDENTIFIER (this is called the *token type*)

In Java, you would code this as:

```
class Token {
  private String text;
  private int type;

  public Token(int type, String text) {
    this.type = type;
    this.text = text;
  }

  public void setText(String t) { text=t; }
  public String getText() { return text; }
  public void setType(int t) { type=t; }
  public int getType() { return type; }
}
```

The token type definitions are best specified in an interface that the lexer and parser can implement to see the constant definitions (a generally useful way to specify and use constant pools)

```
interface ALTokenTypes {
  public static final int IDENTIFIER   = 1;
  public static final int INTEGER      = 2;
  public static final int KEYWORD_DONE = 3;
  public static final int COMMENT      = 4;
  public static final int EQUALS       = 5;
  public static final int NEWLINE      = 6;
  public static final int LPAREN       = 7;
  public static final int RPAREN       = 8;
  public static final int EOF          = 9999;
}
```

For input "width", a lexer for AL would create a token object containing:

["width", IDENTIFIER]

Similarly, a lexer for AL would convert the complete character stream

```
# sample input
width=200
height=100
f(25)
done
```

into the following token stream:

```
["# sample input", COMMENT]
["\n", NEWLINE]
["width", IDENTIFIER]
["=", EQUALS]
```

```
["200", INTEGER]
["\n", NEWLINE]
["height", IDENTIFIER]
["=", EQUALS]
["100", INTEGER]
["\n", NEWLINE]
["f", IDENTIFIER]
["(", LPAREN]
["25", INTEGER]
[")", RPAREN]
["\n", NEWLINE]
["done", KEYWORD_DONE]
["\n", NEWLINE]
```

The Effect Of Separate Lexers and Parsers On Lookahead Requirements

Separating the lexer and parser typically reduces the lookahead requirements for language level recognition because the parser sees lexical elements that are arbitrarily long, like identifiers, as single elements. Reconsider the problem of distinguishing assignments from method calls. The previous solution left-factored the call to identifier in method inputFile() like this:

```
identifier();
if ( LA(1)=='=' ) {
  assignmentMinusIdentifier();
}
else {
  methodCallMinusIdentifier();
}
match('\n');
```

No finite amount of lookahead will see past an arbitrarily long character sequence to the '=' or the '(' beyond the identifier. Left-factoring is a must in a combined lexer/parser. On the other hand, a separate parser sees a single token type, INTEGER, followed by EQUALS or LPAREN. A Lookahead depth of k=2 tokens neatly predicts assignment versus method call statements:

```
if ( LA(1)==IDENTIFIER && LA(2)==EQUALS ) {
  assignment();
}
else if ( LA(1)==IDENTIFIER && LA(2)==LPAREN ) {
  methodCall();
}
```

Separating the lexer and parser also tends to reduce the lookahead requirements for the lexical level. For example, a separate lexer can merge the recognition of lexically identical element such as keywords and identifiers and then send different token types to the parser such as IDENTIFIER and KEYWORD_DONE. Method identifier() in a separated lexer looks like:

```
Token identifier() {
  StringBuffer s = new StringBuffer();
  while ( Character.isLetter((char)LA(1)) ) {
    s.append((char)LA(1));
    consume();
  }
  String id = s.toString();
  if ( id.equals("done") ) {
    return new Token (KEYWORD_DONE, "done");
  }
  return new Token(IDENTIFIER, id);
}
```

A Parser For AL With k=2

The following parser is a reformulation of the previous conglomerate parser/lexer that operates on a stream of tokens.

```
void inputFile() {
  while ( LA(1) != EOF ) {
    if ( LA(1)==COMMENT ) {
      String cmt = LT(1).getText();
      match(COMMENT);
    }
    else if ( LA(1)==KEYWORD_DONE ) {
      match(KEYWORD_DONE);
}
    else if ( LA(1)==IDENTIFIER && LA(2)==EQUALS ) {
      assignment();
    }
    else if ( LA(1)==IDENTIFIER && LA(2)==LPAREN ){
      methodCall();
    }
    match(NEWLINE);
  }
}

void assignment() {
  String id = LT(1).getText();
  match(IDENTIFIER);
  match(EQUALS);
  match(INTEGER);
}

void methodCall() {
  String id = LT(1).getText();
  match(IDENTIFIER);
  match(LPAREN);
  match(INTEGER);
  match(RPAREN);
}
```

The infrastructure for the separated parser is:

```
import java.io.*;

class ALParser implements ALTokenTypes {
private TokenBuffer input;
private static final int LOOKAHEAD_DEPTH = 2;

public ALParser(Tokenizer lexer) {
      input = new TokenBuffer(lexer, LOOKAHEAD_DEPTH);
}

public ALParser(TokenBuffer buf) {
      input = buf;
}

private int LA(int i) { return input.LA(i).getType(); }
private Token LT(int i) { return input.LA(i); }
private void consume() { input.consume(); }
private void match(int c) { consume(); }

void inputFile() {…}
void assignment () {…}
void methodCall () {…}

}
```

A Lexer For AL With k=1

The rules now return a Token object instead of a string and the identifier routine also checks for the "done" keyword.

```
Token identifier() {
  StringBuffer s = new StringBuffer();
  while ( Character.isLetter((char)LA(1)) ) {
    s.append((char)LA(1));
    consume();
  }
  String id = s.toString();
  if ( id.equals("done") ) {
    return new Token (KEYWORD_DONE, "done");
  }
  return new Token(IDENTIFIER, id);
}

Token integer() {
  StringBuffer s = new StringBuffer();
  while ( Character.isDigit((char)LA(1)) ) {
    s.append((char)LA(1));
    consume();
  }
  return new Token(INTEGER, s.toString());
}

Token comment() {
  StringBuffer s = new StringBuffer();
  match('#');
  while ( LA(1)!='\n' ) {
    s.append((char)LA(1));
    consume();
  }
  return new Token(COMMENT, s.toString());
}
```

The analog of the adventurer's sidekick is the routine that asks "what is out there" instead of having the parser say "go get this lexical structure":

```
public Token getToken() {
  if ( Character.isLetter((char)LA(1)) ) {
    return identifier();
  }
  else if ( Character.isDigit((char)LA(1)) ) {
    return integer();
  }
  else if ( LA(1)=='#' ) {
    return comment();
  }
  else if ( LA(1)=='=' ) {
    consume();
    return new Token(EQUALS, "=");
  }
  else if ( LA(1)=='\n' ) {
    consume();
    return new Token(NEWLINE, "\n");
  }
  else if ( LA(1)=='(' ) {
    consume();
    return new Token(LPAREN, "(");
  }
  else if ( LA(1)==')' ) {
    consume();
    return new Token(RPAREN, ")");
  }
  else if ( LA(1)==-1 ) {
    return new Token(EOF, "<eof>");
  }
  return null;
}
```

The infrastructure is similar to before:

```
class ALLexer implements ALTokenTypes {
private CharBuffer input;
private static final int LOOKAHEAD_DEPTH = 1;

public ALLexer(Reader f) {
  input = new CharBuffer(f, LOOKAHEAD_DEPTH);
}

private int LA(int i) { return input.LA(i); }
private void consume() { input.consume(); }
private void match(int c) { consume(); }

public Token getToken() {…}
Token identifier() {…}
Token integer() {…}
Token comment() {…}

}
```

Filtering What the Parser Sees

The recognizers above do not allow whitespace characters like space and tab. For example, the following assignment with spaces around the equals is not valid.

```
width = 200
```

Because whitespace can occur anywhere, matching whitespace in the parser is a hassle. You would have to put whitespace recognition code before or after every match() method and so on.

A more general solution is to simply have the lexer filter out the whitespace, allowing the parser to focus on the stream of tokens rather than formatting details like whitespace. The following slight modification to the lexer's getToken() method to consume spaces and tabs is sufficient to allow whitespace anywhere in the input without modifying the parser.

```
public Token getToken() {
  for ( ; ; ) {
    if ( LA(1)==' ' || LA(1)=='\t' ) {
      consume(); // do not return; try again.
    }
    else if ( Character.isLetter((char)LA(1)) ) {
      return identifier();
    }
    else if ( Character.isDigit((char)LA(1)) ) {
      return integer();
    }
    else if ( LA(1)=='#' ) {
      return comment();
    }
    …
    else {
      return null;
    }
  }
}
```

Because Java does not support goto statements, a "forever" loop is placed around everything in getToken(). Valid token matches will perform a return, thus, avoiding the infinite loop. Lexical constructs you want to skip do not return and restart the process of looking for a token.

To also allow newline to appear anywhere instead of only at the end of every statement, do the following:

1. Remove the `NEWLINE` token type definition from interface `ALTokenTypes`; it is no longer needed because the lexer will never inform the parser that a newline was found.

2. Remove the code that matches the `NEWLINE` token in method `inputFile()`of the parser:

```
void inputFile() {
  while ( LA(1) != EOF ) {
    if ( LA(1)==COMMENT ) {
      String cmt = LT(1).getText();
      match(COMMENT);
    }
    else if ( LA(1)==KEYWORD_DONE ) {
      match(KEYWORD_DONE);
    }
    else if ( LA(1)==IDENTIFIER && LA(2)==EQUALS ) {
      assignment();
    }
    else if ( LA(1)==IDENTIFIER && LA(2)==LPAREN ){
      methodCall();
    }
    // Remove: match(NEWLINE);
  }
}
```

3. Modify getToken() in the lexer to discard newlines along with other whitespace.

```
public Token getToken() {
  for ( ; ; ) {
    if ( LA(1)==' ' || LA(1)=='\t' || LA(1)=='\n' ) { // scarf \n too
      consume(); // do not return; try again.
    }
    else if ( Character.isLetter((char)LA(1)) ) {
      return identifier();
    }
    else if ( Character.isDigit((char)LA(1)) ) {
      return integer();
    }
    else if ( LA(1)=='#' ) {
      return comment();
    }
    else if ( LA(1)=='=' ) {
      consume();
      return new Token(EQUALS, "=");
    }
    /*
    else if ( LA(1)=='\n' ) {
      consume();
      return new Token(NEWLINE, "\n");
    }
    */
    …
    else {
      return null;
    }
  }
}
```

4. Force the `comment()` method in the lexer to consume the ending newline as it is really part of the single-line comment.

```
Token comment() {
  StringBuffer s = new StringBuffer();
  match('#');
  while ( LA(1)!='\n' ) {
    s.append((char)LA(1));
    consume();
  }
  match('\n'); // consume trailing newline now
  return new Token(COMMENT, s.toString());
}
```

You can also allow comments anywhere by:

1. changing the "`return comment();`" statement in `getToken()` to call comment() without returning from `getToken()`
2. removing token type `COMMENT`
3. removing the code to match comment statements in the parser

Simple Error Handling

The recognizers in this chapter presented so far assume valid input. That is, they assume that all input sentences conform to the language recognized by the recognizer. In practice, of course, humans can provide invalid sentences. How do you deal with erroneous input?

There are two basic issues: reporting and recovery. Reporting involves catching errors and notifying the user. Recovery refers to the parser's attempt to resynchronize its state and possibly the state of the input token stream so that the parser may continue parsing the remaining input. See the chapter on error handling for a discussion of parser recovery. For the moment, assume that a recognizer terminates upon the first error.

Error Reporting Support

Bad sentences are exceptional conditions and Java exceptions are a great way to trap erroneous input. There are two fundamental kinds of errors and, hence, you need to define two exception types: `MismatchedTokenException` and `NoViableAlternativeException`. Define these exceptions as subclasses of a class called `ParserException` so that you can catch any syntax error from the parser by catching `ParserException` type exceptions. These classes are just error names and do nothing but reuse the functionality of a generic Java exception. Here are their definitions.

```
class ParserException extends Exception {
  public ParserException(String msg) {
    super(msg);
  }
}

class MismatchedTokenException extends ParserException {
  public MismatchedTokenException(String msg) {
    super(msg);
  }
}

class NoViableAltException extends ParserException {
```

```
  public NoViableAltException(String msg) {
    super(msg);
  }
}
```

Mismatched Tokens

A mismatched token occurs when the parser tries, for example, to match an integer via "`match(INTEGER)`" and the next input token is not an integer. An old compiler from Apple Computer used to say fabulous things like

```
A ')' came as a complete surprise to me at this point in your program
```

The goal is to catch missing or extra tokens as in the following erroneous input.

```
# sample input
width=200
height=100
f()
done
```

The argument to the method call is missing and you would like a message similar to:

```
token mismatch at ")"; expecting type 2
```

###add token type table.

To catch element mismatch errors in a recognizer, put an if-test and throw statement inside the match() method. For example, to catch errors in the parser, modify `ALParser.match()` as follows:

```
private void match(int t) throws MismatchedTokenException {
  if ( LA(1)!=t ) {
    throw new MismatchedTokenException(
      "token mismatch at \""+LT(1).getText()+
      "\"; expecting type "+t
    );
  }
  consume();
}
```

Upon a mismatch, `match()` will throw the exception forcing an exit from `match()` and any rules of the parser out to the `catch`-statement around the call to `inputFile()`.

No Viable Alternative Substructure

The other kind of error occurs when a recognizer must make a decision about which alternative substructure to match and the lookahead is inconsistent with the start of any of the possible alternatives. For example, statement "`(25)`" is missing a method name and the two lookahead tokens parenthesis and `25` do not predict any valid statement. Here, the goal is to catch erroneous input such as:

```
# sample input
width=200
height=100
(25)
done
```

A reasonable error message would be:

```
bad statement at "("
```

Throw `NoViableAltException` exceptions as the default case wherever you have a recognition decision. In the parser for AL, there is only one decision point: the if-then-else that routes control to the appropriate statement recognizer method. The following modification of inputFile() throws an exception if the lookahead does not predict a comment, "done", an assignment, or a method call.

```
void inputFile() throws ParserException {
  while ( LA(1) != EOF ) {
    if ( LA(1)==COMMENT ) {
      String cmt = LT(1).getText();
      match(COMMENT);
    }
    else if ( LA(1)==KEYWORD_DONE ) {
      match(KEYWORD_DONE);
    }
    else if ( LA(1)==IDENTIFIER && LA(2)==EQUALS ) {
      assignment();
    }
    else if ( LA(1)==IDENTIFIER && LA(2)==LPAREN ){
      methodCall();
    }
    else { // TRAP ERRORS. No statement predicted.
      throw new NoViableAltException(
        "bad statement at \""+LT(1).getText()+"\"");
    }
    match(NEWLINE);
  }
}
```

Modifications To Recognizer Methods

To handle either type of parser exception, each parser method must indicate that it can throw a `ParserException` of some kind just like method `inputFile()` does above. Class `ALParserTrapErrors` looks like the following.

```
class ALParserTrapErrors implements ALTokenTypes {
…
private void match(int t) throws MismatchedTokenException {…}

public static void main(String args[]) throws IOException {
  //BufferedReader f =
  // new BufferedReader(new FileReader("ALinput.badtoken.txt"));
  BufferedReader f =
    new BufferedReader(new FileReader("ALinput.badalt.txt"));
  ALLexer lexer = new ALLexer(f);
  ALParserTrapErrors parser = new ALParserTrapErrors(lexer);
  try {
    parser.inputFile();
  }
  catch (ParserException pe) {
    System.err.println(pe.getMessage());
  }
  f.close();
}

void inputFile()  throws ParserException {…}
```

```
void assignment() throws ParserException {…}
void methodCall() throws ParserException {…}
}
```

The main program differs from the previous one in that the call to the starting parser method, `inputFile()`, has a try-catch around it to catch parser exceptions thrown while trying to parse the input. Recall that the parser will terminate upon the first error.

Adding Actions

Recognizers are valuable in that today can answer questions about the syntactic conformity but input sentence. On the other hand, it is rare that you are only interested in checking the validity of a sentence. Most often you have a translation to perform. For example, after having read 300 MB of data, you normally want to perform some operation on that data. Further, you cannot even check the semantic validity of a sentence with a simple recognizer. You must add code to your recognizer to load the symbol table with variable definitions and type definitions and so on. The semantic rules of a language cannot be applied by checking mere syntax.

There are two basic kinds of actions. The first type of action with is independent of the actual content and is purely a function of the syntactic structure. The second type of action relies upon the incoming data such as the name of a variable as opposed to the fact that a variable was recognized. You can view either type of action as turning a recognizer into a translator. The result of a translator is not just an answer as to whether the sentence was valid grammatically, but also includes some form of output.

To demonstrate the first kind of action, consider adding some `println()`s to the recognizer for the AL language. After each type of statement, increment a count for the type of statement that was recognized by modifying `inputFile()` as follows.

```
void inputFile() {
  int ncomment=0, ndone=0, nassign=0, ncall=0;
  while ( LA(1) != EOF ) {
    if ( LA(1)==COMMENT ) {
      match(COMMENT);
      ncomment++;
    }
    else if ( LA(1)==KEYWORD_DONE ) {
      match(KEYWORD_DONE);
      ndone++;
    }
    else if ( LA(1)==IDENTIFIER && LA(2)==EQUALS ) {
      assignment();
      nassign++;
    }
    else if ( LA(1)==IDENTIFIER && LA(2)==LPAREN ){
      methodCall();
      ncall++;
    }
    match(NEWLINE);
  }
  System.out.println("found "+ncomment+" comment(s)");
  System.out.println("found "+ndone+" done(s)");
  System.out.println("found "+nassign+" assignment(s)");
  System.out.println("found "+ncall+" method call(s)");
}
```

For input:

```
# sample input
width=200
height=100
f(25)
done
```

you should get the following output:

```
found 1 comment(s)
found 1 done(s)
found 2 assignment(s)
found 1 method call(s)
```

While this output is useful in some circumstances, most often a translation is a function of the actual input symbols. When you ask the recognizer for the token value as opposed to the token type, you are asking for the so-called *attributes* of the input symbols, which end up being properties of the `Token` object.

To demonstrate the second type of action, modify method `inputFile()` so that each comment is printed back, each keyword done is printed back, the variable assigned to is printed back, and the name of each method invoked his printed back. Use method Token.LT() to get the ith token object and ask the token object for the text matched for it by the lexer. Make sure you ask for the text of a token before the token is consumed.

```
void inputFile() {
  while ( LA(1) != EOF ) {
    if ( LA(1)==COMMENT ) {
      String cmt = LT(1).getText(); // get attribute
      match(COMMENT);
      System.out.println("comment: "+cmt);
```

```
    }
    else if ( LA(1)==KEYWORD_DONE ) {
      match(KEYWORD_DONE);
      System.out.println("done");
    }
    else if ( LA(1)==IDENTIFIER && LA(2)==EQUALS ) {
      String id = LT(1).getText();
      System.out.println("assign: "+id);
      assignment();
    }
    else if ( LA(1)==IDENTIFIER && LA(2)==LPAREN ){
      methodCall();
    }
    match(NEWLINE);
  }
}

void assignment() {
  match(IDENTIFIER);
  match(EQUALS);
  match(INTEGER);
}

void methodCall() {
  String id = LT(1).getText();
  System.out.println("call: "+id);
  match(IDENTIFIER);
  match(LPAREN);
  match(INTEGER);
  match(RPAREN);
}
```

The code to print assignment variables is in `inputFile()` while the code to print method call identifiers is in `methodCall()` to illustrate that the call to `LT()` does not have to be right next to a `match()`.

For the same input as before, the output should now reveal partial input contents:

```
comment:  sample input
assign: width
assign: height
call: f
done
```

Summary

Conventional wisdom dictates diff impl strategies. Implications of parsing strength. Again and again say that lexer/parser are the same; recount lookahead strategies and "top-down" approach.

Same technology…ANTLR pretty much considers anything a parser, lexer, syntax recognizer, tree walker.