

# Lexical Analysis and Parsing for Business Applications

---

## *Basic Examples*



## Table of Contents

---

1	Our Goal .....	2
2	General Overview .....	6
3	Finite State Machine Design.....	8
4	Getting Started .....	10
4.1	The First States .....	10
4.2	Get_Char(some text, pointer) .....	11
4.2.1	Get_Char Example.....	12
4.2.2	Get_Char and a State Machine .....	14
4.3	Errors.....	15
4.4	A one-state Machine with error control .....	16
4.5	Sample FSM Table – Reading Signed Integers .....	18
4.6	Sample Finite State Machine—Reading Multiple Values—“3X” .....	20
4.6.1	Interpreting the multiple value “3X” .....	21
4.6.2	Grammar for the multiple value “3X” .....	22
4.7	Delimiters and Control Flow .....	23
4.8	Recognizing a simple construction .....	24
4.9	Code Example: Simple COBOL Statement .....	28
5	Defining a Grammar .....	31
5.1	Simple State Tables.....	32
6	Describing Lexical Inputs.....	33
6.1	VBA Representation of State Tables .....	34
6.2	State Machine Object Model .....	34
7	Appendices .....	35
7.1	Recognizers.....	35
7.1.1	Recognizer for < /> .....	36
7.1.2	Recognizer for <identification>.....	36
7.1.3	Recognizer for <number>.....	36
7.1.4	Recognizer for <number,2>.....	36
7.1.5	How the recognizers work.....	36
7.2	Simple COBOL Data Definition.....	37
7.2.1	Slightly Less Simple Picture Statement.....	46
7.2.2	Use of a <picture> Recognizer .....	47
7.2.3	Grammar for COBOL variable definitions.....	47
7.2.4	Grammar for COBOL variable definitions with default values .....	48
7.2.5	More General Syntax.....	50
7.2.6	Defined Expressions .....	50

# 1 Our Goal

Our goal is to develop a fast, simple, powerful mechanism for handling lexical analysis and parsing tasks. Lexical analysis is the foundation on which rests all higher functions of semantic analysis.

Previous systems we've developed have used a basic, but powerful method for Lexing text.

First, we used generalized phrase dictionaries to **assign metatokens to the text**. These phrase dictionaries, while generalized, were relatively "concrete" recognizers for tokens in text streams. Our phrase dictionary contained bits of text we might encounter, for example:

- Money,
- Percents,
- Medical phrases, and
- Services

... and assigned metatokens to these bits of text. So, our phrase dictionary might contain data like the following:

```
George Washington    <president>
John Adams           <president>
Thomas Jefferson     <president>...
```

The phrase dictionary might additionally represent ordinal numbers:

```
First      <ordinal_number>
Second     <ordinal_number>
1st      <ordinal_number>
2nd      <ordinal_number>
3rd      <ordinal_number>...
```

This would give our parser a way to represent this sentence:

```
George Washington and John Adams were the first and second
presidents.
```

As:

```
<president> and <president> were the <ordinal_number> and
<ordinal_number> presidents.
```

Why "tokenize" the text in hand? Tokenizing the text provided more power to the next step in the process

The next step in our process was to **assign patterns** to the *tokenized* text.

The first time we read the line “George Washington and John Adams...” we looked at it, decided how to represent it in a table, and created a pattern for it. The pattern specified a set of metatokens, and gave a structure for their output.

(Putting the data in a table structured in a particular way allowed us to map our text data to a given scheme. Depending on the use you want to put your text to, your table might have a different structure.)

Whenever the parser encounters this string of metatokens, it puts the data in a table that we have designed to represent the data.

Later on, we might encounter the following sentence:

```
Thomas Jefferson and James Madison were the 3rd and 4th
presidents.
```

The tokens this sentence would produce would be the same as the token for the sentence about Washington and Adams:

```
<president> and <president> were the <ordinal_number> and
<ordinal_number> presidents.
```

We’ve not explicitly made provisions in our system for dealing with the third and fourth presidents. The ordinal numbers are represented differently in the two bits of text above. The program generates the same metatokens for the second sentence as it did for the first. It does so automatically, without any need for a human to analyze or even look at the second line of text, because it is using the phrasebook to tokenize the text and the pattern book to apply a pattern to the text. Our parser can then recognize the strings of text and take action accordingly.

Often, a single pattern can be used to automatically process massive amounts of text, particularly in domains where the language is fairly standard.

This basic method works well – but we can expand it to be much more flexible and powerful. We decided what kind of output our parser should generate – what sort of table the data should go into – by looking at our data and thinking about what our natural language processor was ultimately trying to produce. These sorts of considerations guided our judgments about what sort of tokens to create, what sorts of patterns to create, and what the output should ultimately look like. We had definite goals for our processor, and wanted the (specifically domain-limited) data to be leveraged and transformed to a particular, clear, limited, distinct framework.

We also sometimes coded very specific recognizers as tokenization patterns. For example, the phrasebook did not have a list of basic numbers – we had a code routine that looked at the text and decided if it was numeric or alphabetic.

We can create something more powerful, flexible, and easy to operate.

Our first step will be to separate the phrase book and tokenizing mechanisms from the actual semantic language analysis system. We want our program to take any phrase dictionary at all and tokenize text; it should then take any pattern book at all, and use it to put the text into any output format. And a phrase dictionary – or grammar – should be able to do more than tokenize text: it should be able to transform text, do mathematical operations, or replace some strings with others. It ought to be able to do a bit of basic logic (if the grammar gives it logical rules) and make associations between bits of data.

There's no universal way of deciding what data needs to be tokenized, or what a particular piece of data ultimately *means*. That really depends on who is going to use the data, and what it's going to be used *for*. There's no universal way of deciding what sorts of patterns need to be created, or how the output should be put into tables and used. It might be that the data is going to be mapped to another, already existing database – or it might be that you want to create an information base for strategic decisions. Actually, much of the theoretical framework for building some of this code comes from the process of building compilers – computer programs that translate computer code from so much near-gibberish to a something the computer can understand and execute. Your data might be financial data, or it might be a set of contracts or rules. Our basic method can handle all of these different sorts of data and uses.

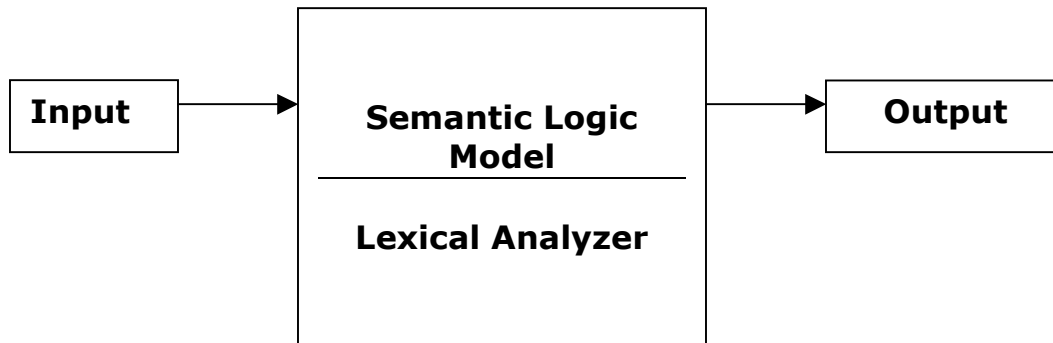
We can do so through flexible a system that will take a grammar and apply it to the text. Later on we can create patterns, apply them to the text, and generate whatever output is needed. The first step in generalizing our algorithms so they can be used anywhere, anytime, for anything, is giving it a way to do some of its most basic lexical analysis.

We can create more than just a phrasebook that attaches tokens to particular strings of text. Our new grammar model can assign “actions” to strings of text – so, for example, it can define bits of text in terms of complex criteria or perform transformations on particular bits of text. We can also do some basic semantic text tokenization based on the grammar rules of the language we're working with – in this case, English. A basic English grammar (for example, one that has a rule that words followed by the article “the” are nouns or noun phrases) can help in assigning patterns and in creating a grammar for a particular domain or application. Such a grammar can then be included (or not included, as the situation dictates) along with a subject-specific or goal-specific grammar when parsing for a particular project. Both can be used to tokenize the text and included in the patterns that ultimately create new ways of organizing and understanding your information.

Starting from the bottom up – from basic tokenization to pattern definition – our first step will be to generate a machine that can read text using whatever grammar you give it.

## 2 General Overview

The Lexical Analyzer is a key subsystem of the Semantic Logic Model (SLM). It is the primary mechanism for the SLM to bring in text input, and transform that to fact-representation output.



**Figure 1 – The Lexical Analyzer Supports the Semantic Logic Model**

The input can be any grammatically correct English prose. News articles, reports, are some examples of valid input for the lexical analyzer. The output ranges from fully “understood” concise text representation of the input to unparsable input that generates an error. The error case occurs when the input files are corrupt and/or in the wrong format. In this case, the Lexical Analyzer does not attempt to work on them. If the incoming file is readable, the reader might not be able to “grasp” the meaning. In this case we tag the text as unknown after some processing. In all other cases the output is a text representation of the meaning of the incoming text. For example, if the input is the following news:

“One year from now, on May 11,2008, states are required by federal law to begin overhauling the technology that manages driver’s licensure.”

... the output in the simplest possible case will be:

“One year from now <comma> on May <digit> <comma> <digit> <comma>  
states are required by federal law to begin overhauling the technology that  
manages driver’s licensure.”

Another example might be:

“Born out of recommendations made by the 9/11 commission, an estimated \$11 billion may be spent on the national Real ID initiative.”

and the corresponding output in the simplest case will look like:

“Born out of recommendations made by the <special\_date> commission <comma> an estimated <dollars> may be spent on the national Real ID initiative.”

### 3 Finite State Machine Design

Finite state machines for describing the computers theoretically were introduced by Alan Turing around the middle of the last century<sup>1</sup>. Since then they have become a widely used tool not only in theoretical computer science, but also in a variety of practical applications ranging from recognizers and acceptors to generators.

We are redesigning the Lexical Analysis System to be an easier-to-understand finite state machine.

The section FSM Code provides a first simple characterization of the design.

The machine has a number of states. In each state it can accept or reject a character that it reads from the input stream.

At the conclusion of the state transitions, the machine will have compiled a concrete token, and assigned it a processing type, e.g. <number>, <word>, <president>... whatever is contained in the grammar.

The general approach involves building a tool for generating a Finite State Machine. The Finite State Machine carries out the lexical analysis. The user need not code the logic to accommodate a specific area of analysis. Nor is coding necessary to change the logic whenever the domain of application changes.

---

<sup>1</sup> [Alan Turing](#) (1936), "On Computable Numbers, With an Application to the Entscheidungsproblem", *Proceedings of the London Mathematical Society*, Series 2, Volume 42 (1936). [Eprint](#). Reprinted in *The Undecidable* pp.115-154.

# How it Works

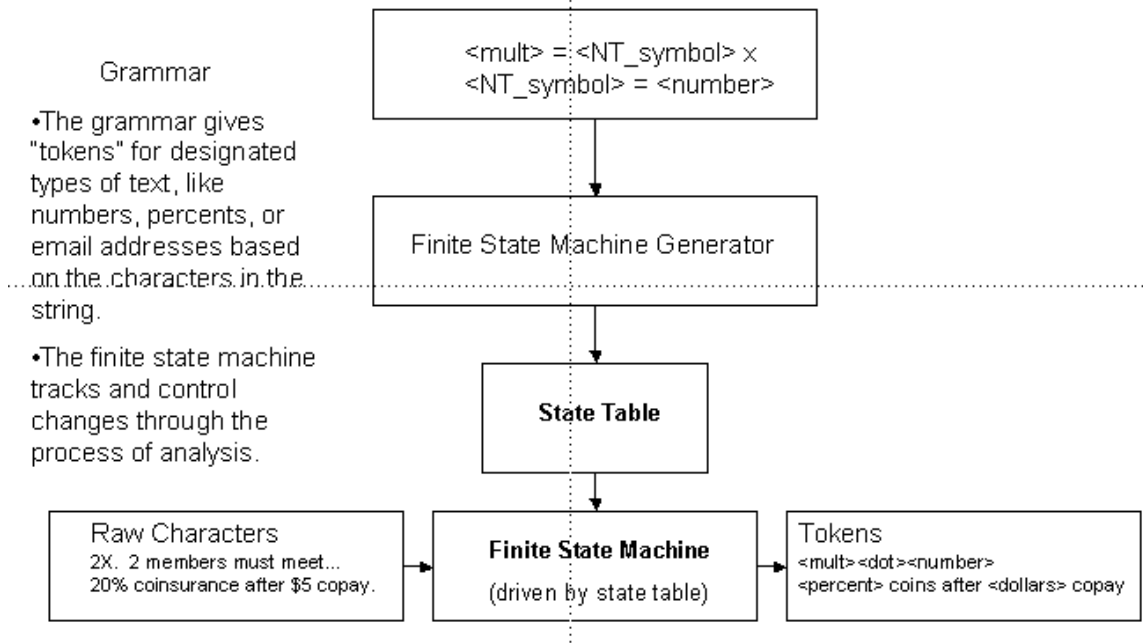


Figure 2 – The FSM Generators Produces State Tables for the FSM

## 4 Getting Started

Let us build a simple finite state machine that can carry out the kinds of recognition tasks we now see before us in the general FSM problem domain on natural language.

### 4.1 The First States

In the beginning there is just a single initial state. Let's call this single state by the name "START".

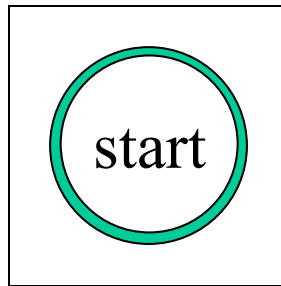


Figure 3 – Single START State

However, to do anything meaningful, we'll need at least one more state, which we will call "END". Now our simplest machine can start, and stop.

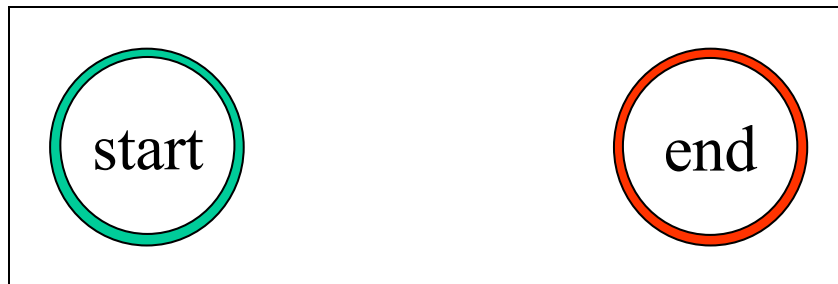
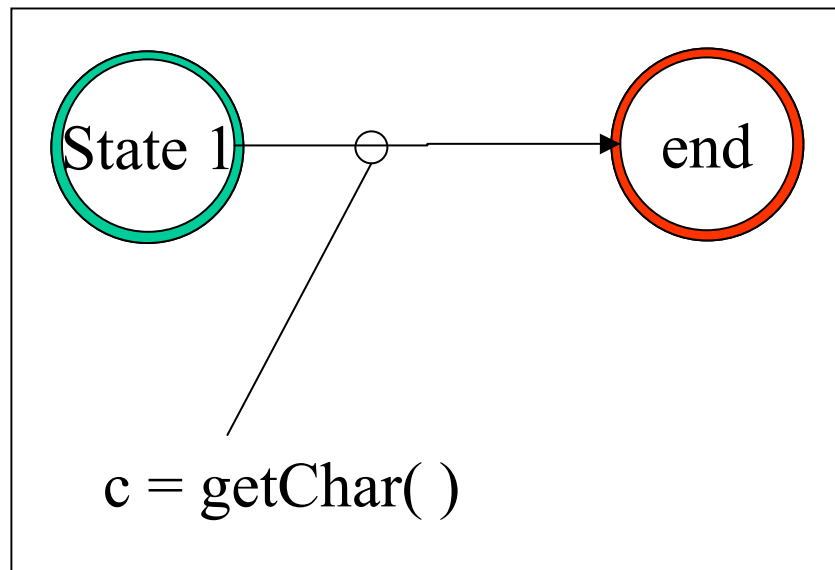


Figure 4 – A START State and an END State

We diagrammatically describe this action by drawing an arrow from “State 1” (which is where we start) to the “END” state.



**Figure 5 – get\_char Routine Triggers the State Transition**

The arrow means that the machine does something in going from one state to another (otherwise, what’s the point?). The something that it does is to read a character from the input stream. That’s pretty simple indeed, but everything that we need to do so to enable us to recognize the most complex structures is contained herein.

#### **4.2 Get\_Char(some text, pointer)**

get\_Char is the computational routine that knows how to collect characters from the input stream and make sense out of them.

The call:

```
c = get_Char(text, pointer)
```

is the code that uses the getChar routine to fetch the next character into the variable “c”. (When this routine is called, it takes two variables: the line of text that contains the character, and a pointer that tells it which particular character in the line it should return. This isn’t too complicated when there’s only one character, but get\_Char is a fundamental routine and has many uses.)

Reading this character “c” is what makes the machine transition from the start state to the end state.

### 4.2.1 Get\_Char Example

Let's look at an example of how Get\_Char works. For example, imagine that the machine has just received this line of text as input:

Third

Using Get\_Char, the machine reads the input character by character.

When we call Get\_Char, we pass in two bits of information:

- The text string in question (in this case, "Third")
- The "pointer" – a number that tells the machine which character in the string to return.

The first time we call get\_char, we feed it the number "1". It will pick out the first character in the string, and give us a "T".

T	h	i	r	d
---	---	---	---	---



linePointer = 1

We get the letter "T", we know we've got a real string of input – we want the next letter.

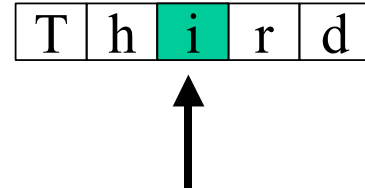
We'll move the pointer to "2", and ask get\_char to tell us what is in the next position

T	h	i	r	d
---	---	---	---	---



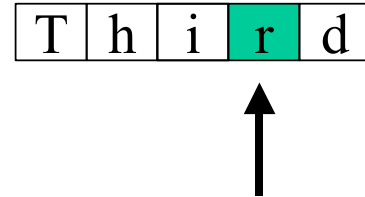
linePointer = 2

We've got a valid input with the last character in our string – the letter “h” – so we move on and see what's next.



linePointer = 3

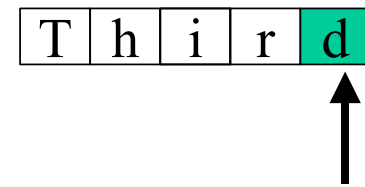
Each time get\_char reads a character, the pointer value is increased by one.



linePointer = 4

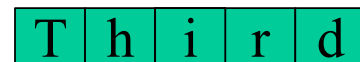
We keep going, taking the next character from the string.

Now the pointer is at “5” – which is the length of our bit of text. We aren't going to look for any more characters.



linePointer = 5

At the end of get\_char, we've taken in every single character in the string one by one.



There's a reason for taking a piece of text, reading it character by character, and then reassembling it.

We want to examine each character and compare it against criteria. Each time a character is produced by `Get_char`, we can compare it against whatever criteria we have established. Our machine analyzes the text input character by character, using `Get_Char`.

Note that at the end of the process, we have one token. We can then classify a token as – for example, the metatoken “<ordinal\_number>”. Having that sort of metatoken allows the parser, later on, to apply a pattern to the text and make sense of it.

#### 4.2.2 `Get_Char` and a State Machine

This machine, as did the last, goes through a few states using `get_char`. In fact, it is no different (in terms of first principles) than any other state.

To simplify our notation (and make it easier to handle automatically generated finite state machines), we change our notation a little and call the START state STATE 1.

To get to it, the machine starts in the initial state and reads a character out of the input stream using the call `c = get_Char(text, pointer)`.

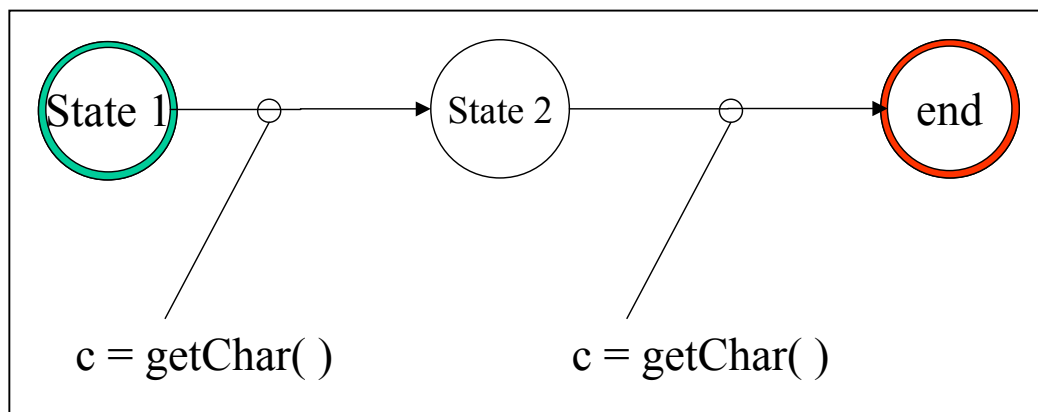


Figure 6 – `get_char` Reads Two Characters

And once in state “STATE 2”, the machine can read another character and go to the end state.

This machine will successfully read all strings that have two and only two characters.

### 4.3 Errors

BUT! What happens if we feed this machine an input stream with just a SINGLE character? Because of the way `getChar()` works, the machine will run right through just as if it had a “legal” two character stream.

Not good!

We want the machine to flag such problems in the input.

To do this we add another state which we call by the name “ERROR!”.

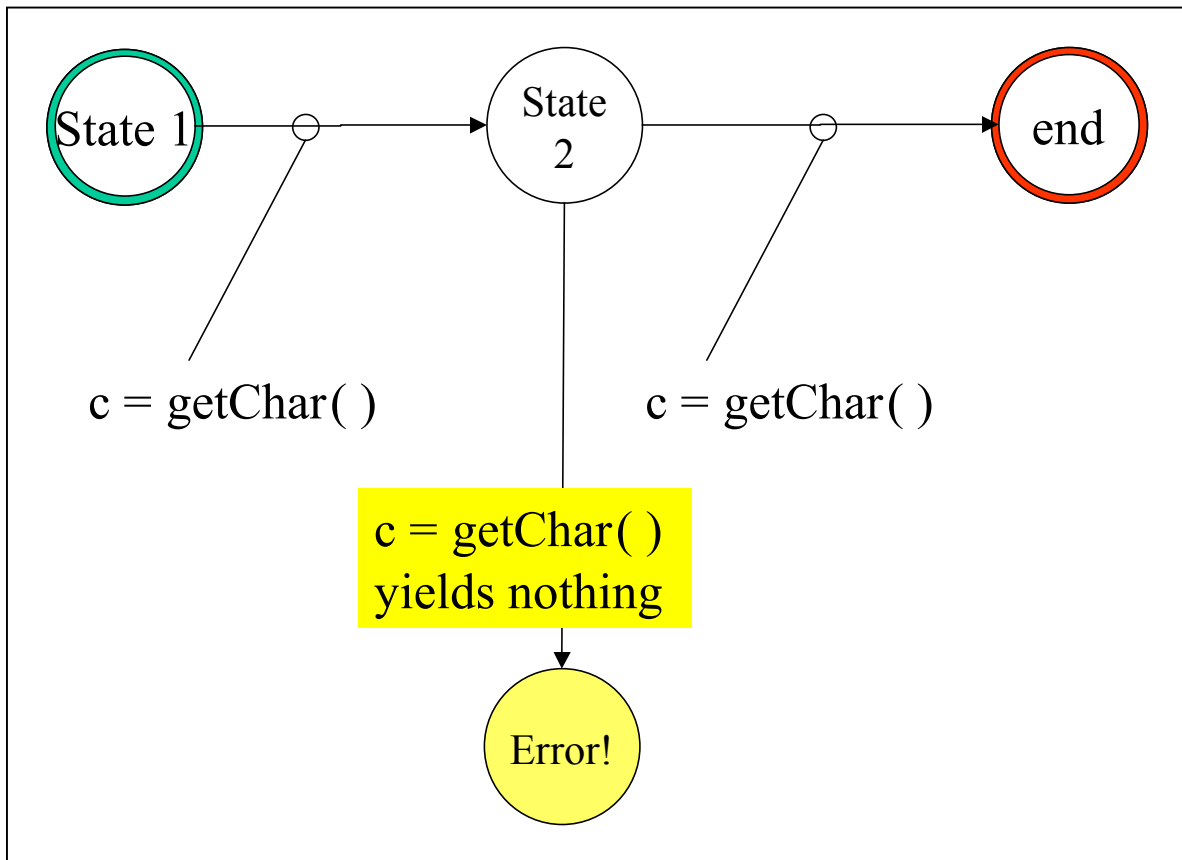


Figure 7 – Handling Read Error: Not Enough Characters to Read

Now, there is an arrow that leads from “STATE 1” to “ERROR!” whenever we try to grab a character from the input stream and come up empty handed.

In the situation where someone feeds this machine a 1-character stream, instead of a 2-character stream, it will complain as soon as it reads the first character and discovers that no more input remains.

#### 4.4 A one-state Machine with error control

Below, we show a lexical analysis machine with one state. We don't count the `err_state` nor the `end_state` – these are part of every machine we create.

It scans exactly one character and if that character is a `<digit>` it accepts the character and ends.

If it scans a character that is not a `<digit>` it rejects the character and goes to the error state.

The symbol `<else>` means any character that hasn't been yet mentioned in the state. The symbol `<nothing>` means that the recognizer does NOT read from the input stream.

We've built a simple table that defines the input, character by character, in terms of states.

STATE	READ	ACCEPT	NEXT STATE
1	<code>&lt;digit&gt;</code>	Ok	<code>end_state</code>
1	<code>&lt;else&gt;</code>	Error	<code>err_state</code>
<code>end_state</code>	<code>&lt;nothing&gt;</code>		
<code>err_state</code>	<code>&lt;nothing&gt;</code>	Reset	

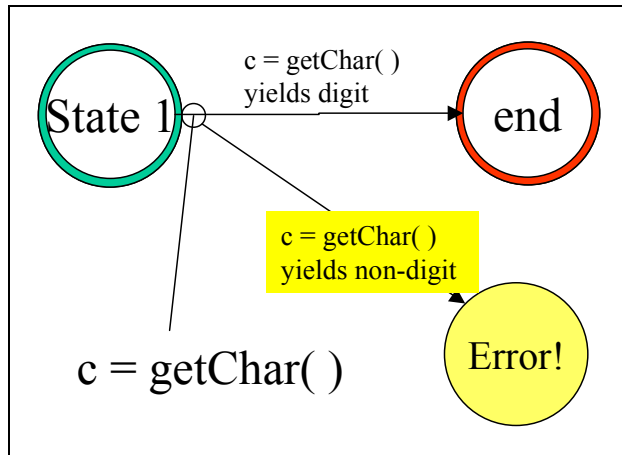
Figure 8 - Machine 1

For us, `<digit>` is a limited lexical recognizer function that examines a single character and returns true if it is in the set:

`{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}`

and false otherwise.

The table gives two criteria for the input: digit, or “else” – meaning anything other than a digit. Since the input in this case is “else” – a non-digit input – the machine goes to the error state.



**Figure 9 – Handling Errors: Machine Reads a non-Digit**

## 4.5 Sample FSM Table – Reading Signed Integers

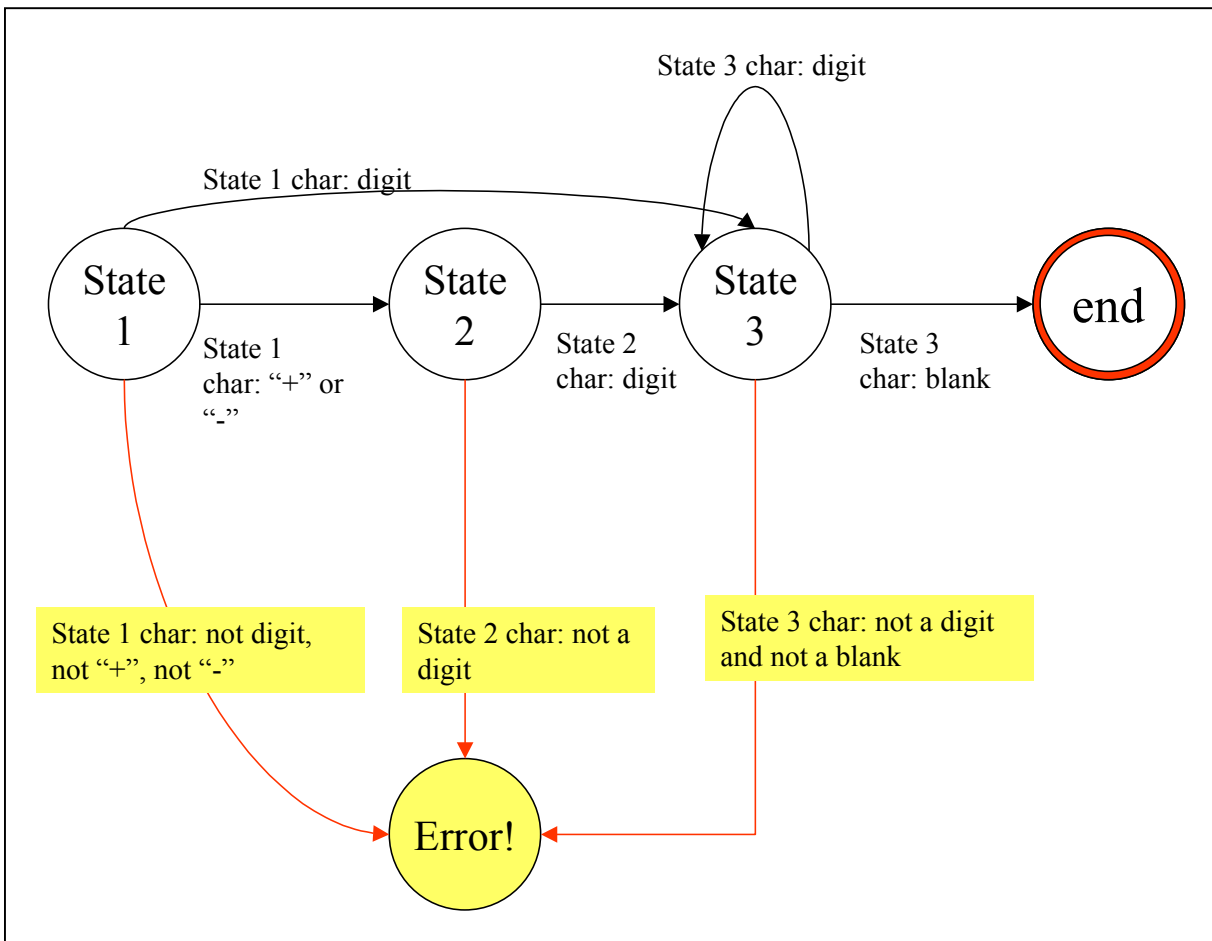
This simple FSM will lex strings of characters that have the following form:

```
+1
-1
1234
-1234
+1234
etc.
```

STATE	INPUT	ACCEPT	NEXT STATE
1	+	ok	2
1	-	ok	2
1	<digit>	ok	3
1	<else>	error	err state
2	<digit>	ok	3
2	<else>	error	err state
3	<digit>	ok	3
3	<blank>	ok	end state
3	<else>	error	err state
end state	<nothing>		
err state	<nothing>	Reset	

**Figure 10 – Machine to Read Signed Integers**

The table can be expressed in a diagram as follows:



**Figure 11 – State Chart for Machine to Read Signed Integers**

Fairly simple tables can create and support quite complex control flows.

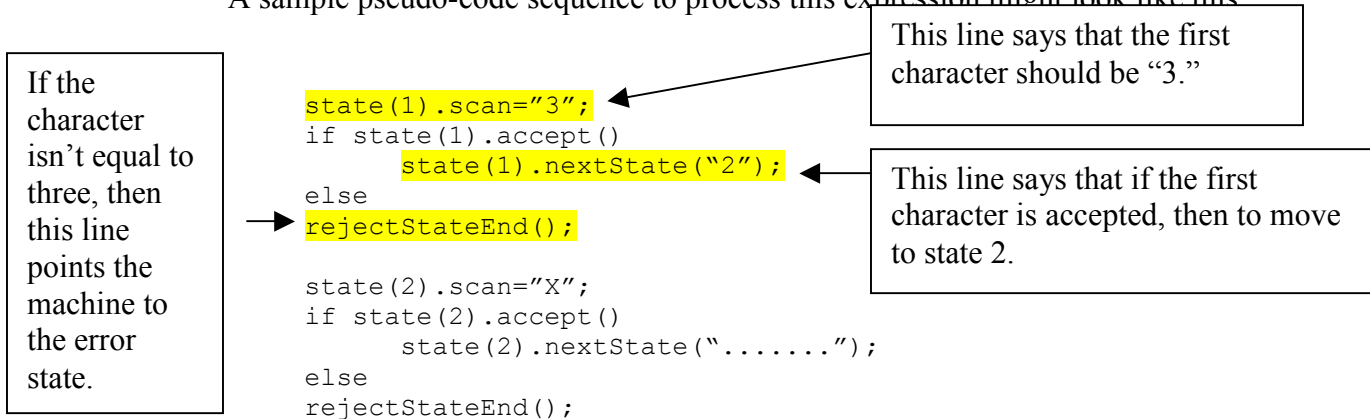
## 4.6 Sample Finite State Machine—Reading Multiple Values— “3X”

We can write out a set of states in code, and send a sample bit of text through the code. Using the code, we define the states however we like.

Expressions like “3X” have a special significance: they mean something quite specific. They say that something is three times the value of something else – they’re mathematical expressions, and ultimately what they point to is some kind of number, like a dollar amount that is equal to three times some other dollar amount.

We can create a fairly simple bit of code to represent this specially significant token type.

A sample pseudo-code sequence to process this expression might look like this:



With this finite state machine each individual character represents a state, so there are as many states as there are terminals.

The “reject” state is a dead end. Once the machine goes to the reject state, it stops doing anything.

Once there is no input to process and the machine hasn't visited the reject state, it enters the “done” state, which indicates that the processing was successful.

A diagram illustrates the process for the case of “3X”. States are in bold; state transitions are illustrated with arrows/slashes. The processed literal is between the states. Notice that with this type of machine we have “movement” only from left to right (or reject). There are no “jumps” and/or transitions between/among states. In other words, this simple case looks like:

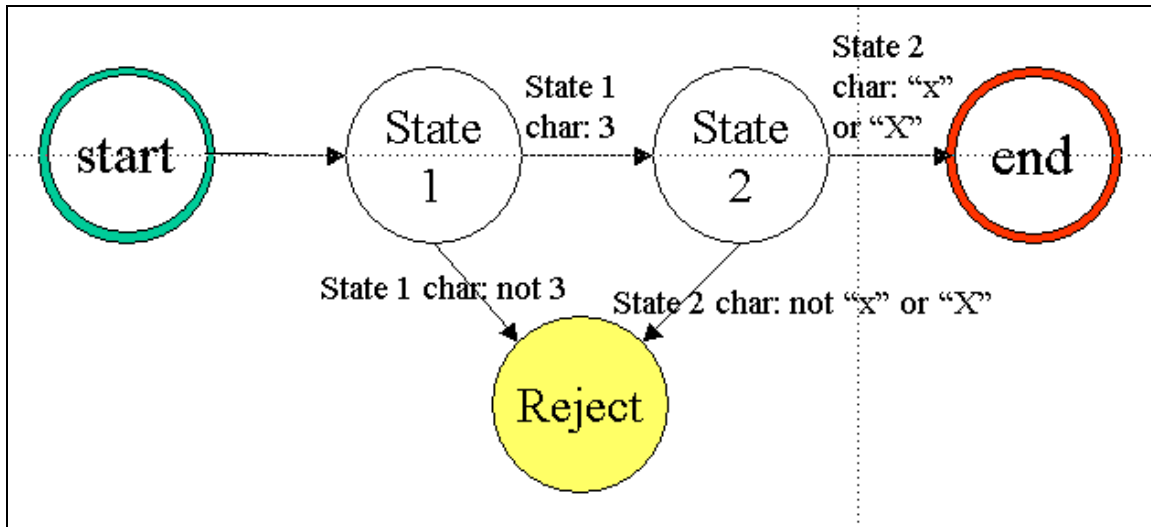


Figure 12 – State Chart of Machine to Read '3X'

#### 4.6.1 Interpreting the multiple value "3X"

The point of going through the process of checking these bits of text character by character is to compare the strings to the criteria we specify, apply patterns to the strings of text, and make sense of them.

We can see how checking to see that a string is equal to "3X" might be helpful. Once we've established that this string actually is "3X", we can break the string down a bit. For example, we can say that "3X" is a multiple, and that the value of the multiple is "3". So, this bit of text means three times something.

(Our next step would be to see if we could determine what it is a multiple of. In the semantic layer, we could make that connection – and possibly, depending on the information available, calculate the value. If the text string were "3X copay amount," for example, and we knew the copay amount, we could ultimately calculate what dollar value "3X copay" meant. This sort of interpretation, however, would happen in another layer of the program.)

In the case of "3X", on this very low level on the program, we'd just codify the information that this "3X" is a multiple, and that its value is "3". We would then have output like `symbol.name=multiple` and `symbol.value=3`. "Symbol.name" and "symbol.value" are pre-defined classes/structures/arrays/database fields that hold the parsed information.

The word "multiple" reflects the meaning of the specific business rule involved – for example, "3X" as a family deductible limit might mean that the family deductible limit equals three times the individual deductible limit.

## 4.6.2 Grammar for the multiple value “3X”

In the above example, we showed how we could simply create a bit of code that read the string “3X”. But that bit of code only handled two characters – and even a relatively small bit of text has a lot more characters than that. (Imagine using code above to read this document. And then imagine changing the code whenever the document changed!)

Having a more general grammar with a set of rules for interpreting text is more manageable and flexible.

A grammar snippet that might handle the specific case of “3X” could be quite simple:

Terminal literals:

```
<digit> ::= 0-9
<alphanumeric-character> ::= x-X
```

Factors:

```
<character> ::= <digit> | <alphanumeric-character>
```

Notice that these rules will handle both “3X” and “3x”. Since the <digit> category includes the numerals from 0-9, expressions like 4x and 7X can be parsed too.

In case the input does not match any of the literals the finite state machine rejects it as unrecognizable for these particular specs. Otherwise it accepts until there are no more characters to process or there is unknown terminus encountered after the first one is processed. It is clear that the initial state of the finite state machine will coincide with the very first literal of the input; if there is no input the machine just does nothing.

The special meaning of expressions like “3X” is context specific and business policy dependant. Therefore, it is not recommended to hard code it, but rather let the end users dynamically specify it before actually parsing at run time.

Giving the users the capability to dynamically specify business specific rules greatly increases the flexibility and usability of the final product as well as making it suitable for parsing diverse corpora<sup>2</sup> -- not necessarily and not only business ones. The protocol as well as its features will be discussed later in this document. For now we will loosely define it as the way companies do business internally as well as externally with other entities.

We'll discuss grammars in more detail later on.

---

<sup>2</sup> Corpora, Plural form for the Latin word for ‘body’

## 4.7 *Delimiters and Control Flow*

“3X Family policy options are to be paid so that the entire family is considered covered.”

We are tackling a composite surface that consists of business specific characters that have business relevant meaning as well as common English surfaces. Using the chapter above we can process “3X” but how do we deal with the rest of the proposition? For that purpose we need to introduce two types of characters:

A delimiter—a character that separates a word surface from the rest of the word surfaces in a proposition. For example, in the sentence “Johnny has a little dog.” the delimiter is

“ ” (white space). White space is what separates the word surfaces from looking like “Johnnyhasalittledog.” which is not very readable. In other words delimiters are the characters that delineate and determine the end and start of individual word surfaces.

A token is any character surface that is in the proposition and that is not a delimiter. If we continue with the previous example the word surface “has” is just one example of a token. It works similarly for the rest of the words in the sentence.

While delimiters separate, tokens carry the lexical and syntactic meaning of the proposition. A C-like escape mechanism can be used to handle the case of a word surface being both a token and a delimiter although in machine pre-formatted files this is not the case.

## 4.8 Recognizing a simple construction

We can design a simple parsing program around the structure of the input stream. For example, the following machine is predicated on our parsing program built for an input stream structured as follows:

“(COBOL,2)”

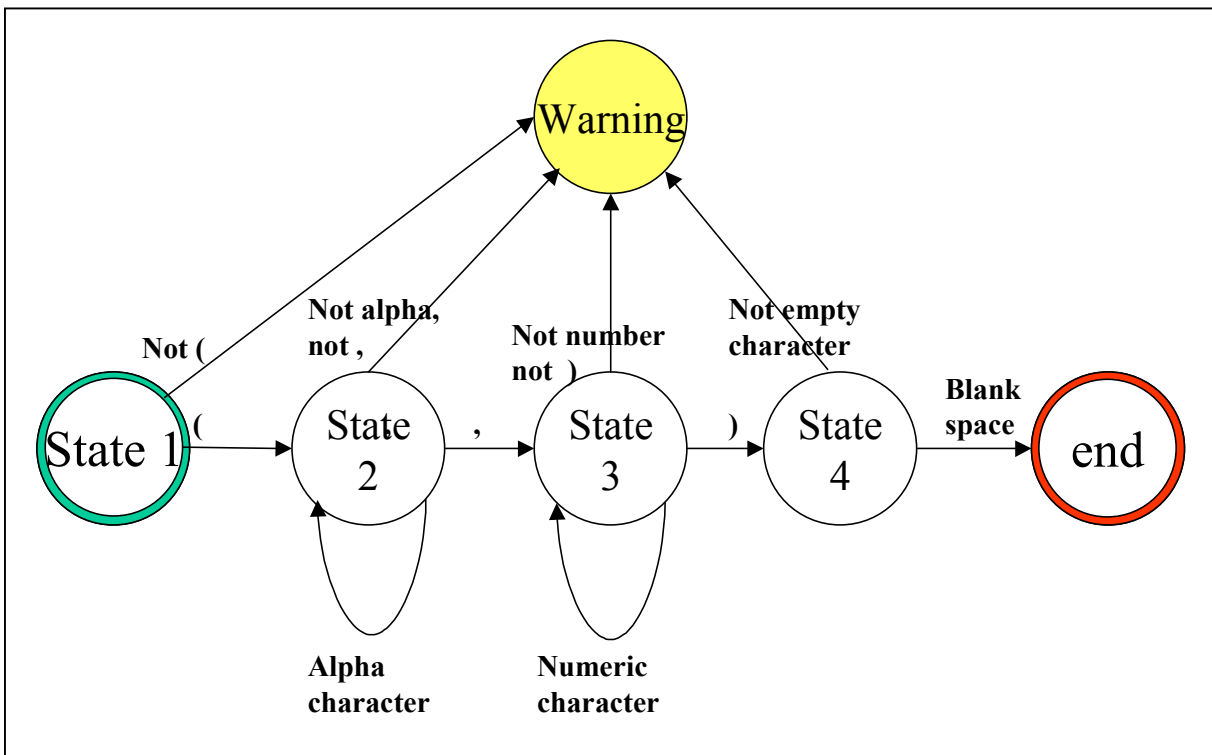


Figure 13 – State Chart for “(COBOL,2)”

This new machine needs five states to accomplish its goal.

The machine initializes in a state = “0” and it reads a character out of the input stream using the call `c = get_Char()`.

If the first character it reads is a left parenthesis, then the state is changed to State 1.

Again using the call `c = get_Char()`, if the next character is not an alpha character, the machine will output a warning that the second character did not match its set of rules.

Here is another example.

```
05 abc pic x(10).
```

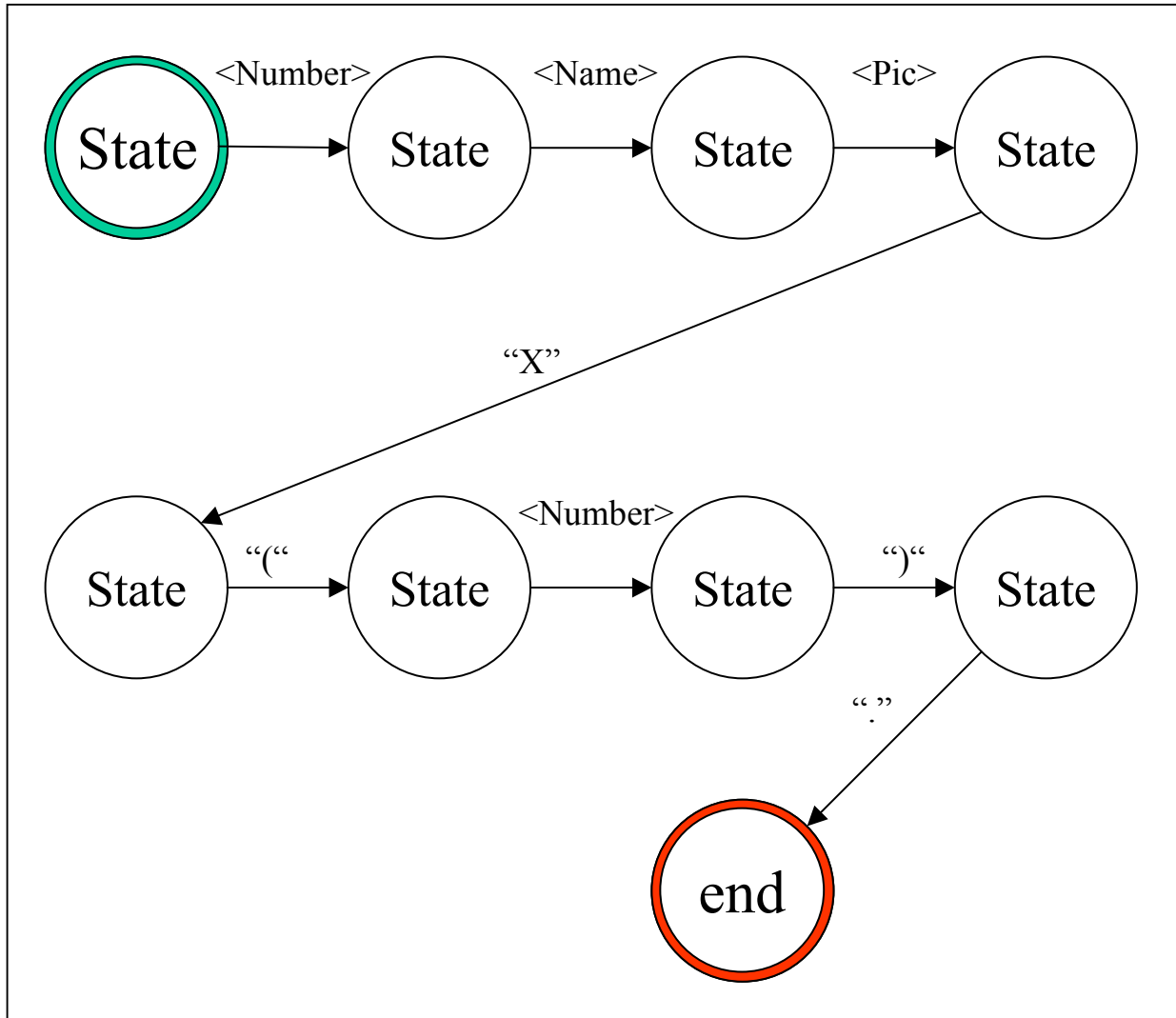


Figure 14 – State Chart for Reading a COBOL Picture Statement

For now, we can see how the FSM can be used to generate a table that shows how exactly our grammar is working with the text.

Getting under the hood of the program – without having to read arcane computer codes – can help in constructing and evaluating grammars, troubleshooting, and maintenance. It makes the system more flexible, and broadens its potential for use by subject matter experts rather than engineers.

## Code for the Simple Construction

```
'-----  
Sub littleLexer()  
'-----  
    Dim c As String  
    Dim read As Boolean  
    s = "(COBOL,2)"  
    p = 0  
    state = "start"  
    c = getChar()  
    Do While c <> "end" And state <> "done"  
        Select Case state  
            Case "start"  
                If c = "(" Then state = "1"  
                    read = True  
            Case "1"  
                If isAlpha(c) Then  
                    state = "1"  
                    read = True  
                Else  
                    state = "2"  
                    read = False  
                End If  
            Case "2"  
                If c = "," Then  
                    state = "3"  
                    read = True  
                End If  
            Case "3"  
                If IsNumeric(c) Then  
                    state = "3"  
                    read = True  
                Else  
                    state = "4"  
                    read = False  
                End If  
            Case "4"  
                If c = ")" Then  
                    state = "end"  
                    read = False  
                Else  
                    compileError ("missing right parenthesis")  
                End If  
            Case "end"  
                MsgBox "successful end"  
                Exit Do  
        End Select  
        Debug.Print state; read, c  
        If read Then c = getChar()  
    Loop  
End Sub
```

## 4.9 Code Example: Simple COBOL Statement

Hard-coding the data to go into a state table can be fairly simple.

```

Option Compare Database
Option Explicit
Dim char As String
Dim p As Integer
Dim l As Integer
Dim pgm As String
Dim read As Boolean
Dim s As String
Dim state As String
Dim tc As String
'-----
Sub littleCOBOL()
'-----
  Dim c As String, t As String
  s = "identification division."
  p = 0
  state = "start"
  read = True
  t = getToken()
  Do While t <> "end" And state <> "end"
    Select Case state
      Case "start"
        If t = "identification" Then state = "1"
      Case "1"
        If t = " " Then
          state = "2"
        End If
      Case "2"
        If t = "division" Then
          state = "3"
        End If
      Case "3"
        If t = "." Then
          state = "end"
        End If
      Case "end"
        MsgBox "successful end"
        Exit Do
    End Select
    Debug.Print state; read, c
    t = getToken()
  Loop

End Sub

```

This line sets "t" equal to the results of getToken (below) which uses get\_char to retrieve each token.

This is the string we are reading.

Here is our initial state.

Each "case" represents a state. When the "case" is "start", then if (and only if) the token is "identification" the machine goes to the next state.

Notice that this machine has no error control.

```

Function getToken()
  Dim token As String
  token = ""
  If read Then tc = getChar()
  If breakChar(tc) Then
    token = tc
    read = True
    GoTo doneToken
  End If
  Do While Not breakChar(tc)
token = token + tc
    tc = getChar()
  Loop
  read = False      'you stopped on a break character
                   'so return the token; and leave the
                   'break character to be read next call

doneToken:
  getToken = token
End Function

Function breakChar(b)
  Dim result As Boolean
  result = True
  If isAlpha(b) Or IsNumeric(b) Then
    result = False
  End If
  breakChar = result
End Function

Function getmatch(x)
  Dim k As Long, token As String
  k = InStr(p, s, x)
  If k = p Then
    'it matched
    token = x
    p = p + Len(x)
  Else
    'it didn't match
    token = ""
    'p doesn't change.  it keeps original value
  End If
  getmatch = token
End Function

Function isAlpha(x)
  If "a" <= LCase(x) And LCase(x) <= "z" Then
    isAlpha = True
  Else
    isAlpha = False
  End If
End Function

Sub mainTestRoutine()
  Dim c As String

```

These functions get the token, using other functions that check each character, check for spaces, and perform other basic texts. Notice that quite a lot of code is needed to create this sort of state machine.

```
s = "(COBOL,2)"
p = 0
Do While True
    c = getChar
    If c = "end" Then Exit Do
    Debug.Print "mainTestRoutine:"; p, c
Loop

End Sub

Function getChar()
    Dim c As String
    If p < Len(s) Then
        p = p + 1
        char = Mid(s, p, 1)
    Else
        p = p + 1
        char = "end"
    End If
    getChar = char
    Debug.Print p, char
End Function

Sub compileError(x)
    Debug.Print x
End Sub

Sub makeTestCase()
    Dim t As String
    Dim NL As String
    NL = Chr(10)
    pgm = "identification division. "

End Sub
```

## 5 Defining a Grammar

We introduced you to how the program would use grammars when we discussed the “3X” processing example.

The state machine above used quite a lot of code to check a few simple words. It’s difficult to maintain, too: small changes could require serious reworking of the whole thing.

We want a lot more flexibility, so that the tests can be created, changed, updated, combined, and managed much more easily.

We are creating a machine that should be able to take in any “grammar” – any set of rules about whether to accept different sorts of input and whether or how to tokenize or transform the input.

For the simple state machine we used to process the statement “(COBOL,2)”, part of a simple grammar might look like:

```
<program> = <IdentificationDivision><EnvDivision><DataDivision><ProcedureDivision>
```

That is, a program is a collection of statements organized into four divisions, where each division is a collection of statements.

In turn we might describe a simplified Identification Division as just having a single statement:

```
<IdentificationDivision> =  
<'identification'><'division'><'.'>
```

That is, for our simplest finite state machine, the construction <IdentificationDivision> consists of the string ‘identification’ followed by the string ‘division’ followed by the string ‘.’ (a single period).

## 5.1 Simple State Tables

When the parser generator reads this grammar it will output a series of arrays that designate the finite state machine. The state table is dynamic: when you change the grammar, the machine changes.

The compact form of the FSM is the matrix below.

STATE	READ	ACCEPT	NEXT STATE
1	Identification	-	2
2	Division	-	3
3	.	-	-

Figure 15 – State Table for Program Division Recognition

For the time being we are representing these matrices as several arrays.

First, we aren't concerned yet with ACTIONS (other than just scanning past the symbol in the input stream). But ultimately, the finite state machine will be able to take specified actions when it encounters specified input.

(For example, sometimes data has been converted from one format to another, and there are strange characters in the middle of the text – like “\” or “\0”. We could specify an action when these characters are encountered – like replacing these characters with spaces, or sending out a warning that the data has these sorts of characters.)

Second, we keep track of the number of symbols allowable for each state with an array of counters, NUMTOKEN().

In this simple example, State 1 has just a single allowable symbol: “identification”. So numTokens(1) is equal to 1:

Likewise State 2 and State 3 each has a single allowable token, the values of numTokens(2) and numTokens(3) are each just 1.

Then for any state  $j$ , the array symbol can have up to numTokens( $j$ ) entries. In this simple case:

```
symbol(1,1) = 'identification'
symbol(2,1) = "division"      and
symbol(3,1) = "."             (that is, a period)
```

## 6 Describing Lexical Inputs

We need a way of talking about the lexical input that is easy and natural to implement in computer terms.

Rather than code the Finite State Machine states by hand, we want to use our shorthand, and have SLM automatically generate the FSM. This will give us the flexibility we need and can't get from complex, hand-coded finite state machines like the ones above. Being able to dynamically create these machines will allow us to have different grammars for different text analysis purposes.

For example, to represent Machine 1, we would rather just say:

```
<digit>
```

For Machine 2, we would rather say:

```
[+, -] <digit>(1 to 10)
```

That is, machine 2 recognizes a string that has an optional sign, followed by 1 to 10 digits.

So we need to develop a small language for representing what the machine can understand (once we generate it).

Ideally, we would like to say something like:

“single digit” instead of <digit>

or;

“a string of 1 to 10 digits” instead of <digit>(1 to 10)

but for the moment we will use a more compressed shorthand.

The FSM controls and documents the process we use when we take different grammars and use them for different texts. It's flexible enough that we can create different states and different actions when we encounter different types of text in different grammars. Using this generator, we can easily and dynamically define lexical patterns, assess our patterns, and control the whole parsing process.

## 6.1 VBA Representation of State Tables

In our Visual Basic for Application module we define these values in a routine call `setStates`.

```

Sub setStates
    '.....
    numTokens(1) = 1
    symbol(1, 1) = "identification"
    nextState(1, 1) = 2
    '.....
    numTokens(2) = 1
    symbol(2, 1) = "division"
    nextState(2, 1) = 3
    '.....
    numTokens(3) = 1
    symbol(3, 1) = "."
    nextState(3, 1) = 0
    '.....
end sub

```

We call `setStates` at initialize time to set up the Finite State Machine.

## 6.2 State Machine Object Model

The examples above show very simple ways of moving from one state to another. In each case, the computer code specifically takes a bit of criteria, applies it to the input stream, and moves on to the next state.

While this shows the basic method, the preliminary implementation of the state machine actually uses an “object.” It

- takes a set of criteria from a file
- organizes the criteria into a set of variables that belong to it
- compares the input stream to the criteria
- and generates an error file

Let’s take an example of a very simple ruleset we’ve put into a text file. It is:

```
<plus_sign> = +
```

There’s only one rule in this set. The only input the machine should accept is a single character – “+”.

When we run the lexer, we set it to read this rule.

When it reads the rule, it puts it into an object:

```
Public state_num As Long (=1; there's only one rule here)
Public state_size As Long (=1; the state is just a plus sign (+))
Public read(10) As String (=+; the item to read is a plus sign)
Public accept(10) As String ("OK"; variable holds response to a
match)
Public next_state(10) (=0; this is the last state; only one rule
was read from the file)
Public action(10) As String (empty, for rules like this one that
don't stipulate a particular action to be taken.)
```

(We have an array of these objects: each line in the text file is put into a different node of the array. But the process is the same for each rule.)

We can take any instance of this object and set up a state table using it. Then we can run text through the system and get an error table output.

Let's say I'd like to change the text file content to something else. I don't want to check to see if the input is a plus sign; I want a plus sign and a number. I add a line to my text file, so that now it looks like this:

```
<plus_sign> = +
<minus sign> = -
```

Now the machine will recognize input that looks like this: "+-".

## 7 Appendices

### 7.1 Recognizers

Some recognizers handle hard, definite characters or sequences of characters. These hard sequences we call terminals.

Other recognizers can handle a range of patterns, that is, they will handle a range of character sequences. These are what we call NONterminals.

Familiar nonterminals are grammar definition symbols like <expr> that represents an expression such as

$$(x + (y+2/z) * a) .$$

### 7.1.1 Recognizer for <(>

This **terminal** recognizer will look for a left parenthesis and nothing else. If it finds a left parenthesis, it returns true, otherwise it returns false. This is an important recognizer because our metatokens are delimited by these characters.

### 7.1.2 Recognizer for <identification>

This is a recognizer that scans for the string “identification” returning true if it finds, it, false otherwise.

### 7.1.3 Recognizer for <number>

This is a meta-recognizer for a nonterminal. It looks for a string of characters that correspond to a numeric string. What it does is pick up the next “sensible” token out of the input stream, and then tests to see if that token is a number or not.

If the token IS a number, then it returns true, and if it isn't a number, it returns false.

### 7.1.4 Recognizer for <number,2>

This is a recognizer for a two-digit integer.

### 7.1.5 How the recognizers work

While a recognizer is working, it runs the linePointer through the input stream buffer character by character until it can decide what it has.

So – the first thing the recognizer should do is to push the linePointer onto a stack, and hold it there for safekeeping.

If the recognizer actually finds what it is looking for, it can pop the saved value off the stack and reposition the linePointer for real.

If the recognizer does NOT find what it is looking for, it pop the saved value and reset the current linePointer to this value. In this case, the next scan will start exactly where it should, and it will be as if the recognizer had never touched the buffer.

## 7.2 Simple COBOL Data Definition

`<SimpleDef> = <number> <name> 'pic' 'x' '(' <number> ')'`.

STATE	SYMBOL	ACTION	NEXT STATE
1	<number>	-	2
2	<name>	-	3
3	pic	-	4
4	x	-	5
5	(	-	6
6	<number>	-	7
7	)	-	8
8	.	-	done

Figure 16 – State Table for Cobol Picture Statement

Let's look into how to implement this state machine.  
 We will use a lineBuffer and a linePointer.  
 The line Buffer fetches a line and saves it into a buffer.

The whole line is saved into the buffer

0	5		a	b	c		p	i	c		x	(	1	0	)	.
---	---	--	---	---	---	--	---	---	---	--	---	---	---	---	---	---

Figure 17 - Buffer for Cobol Picture Statement

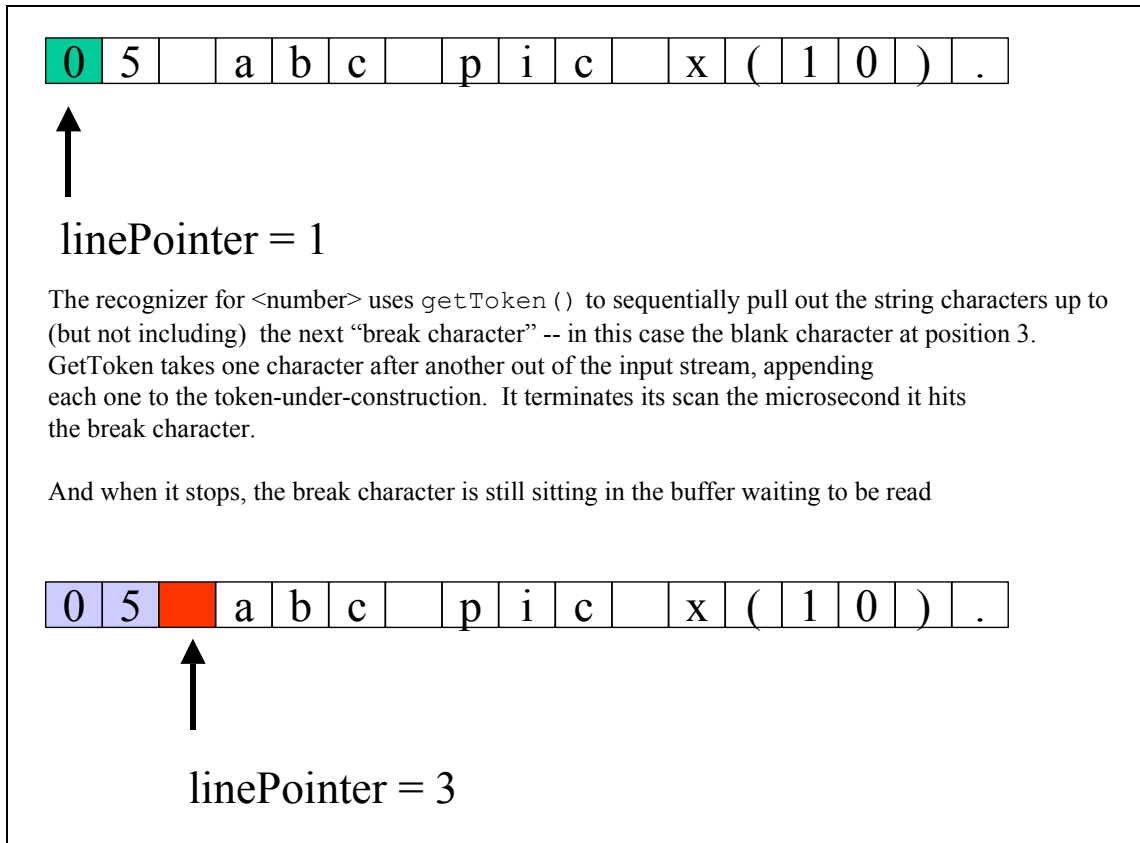
Once the line is saved in the buffer, we will use the state machine to read a character at a time.

We will scan the buffer starting from the beginning, and we need to keep track of where we are as we read the buffer that is why a line pointer is needed.

We initialize the line pointer to 0. The first time getChar reads a character, it will increment the linePointer to 1.

Then the linePointer will point to the first letter in the buffer.

Let's follow this along.



**Figure 18 - Buffer Pointer for Cobol Statement 1**

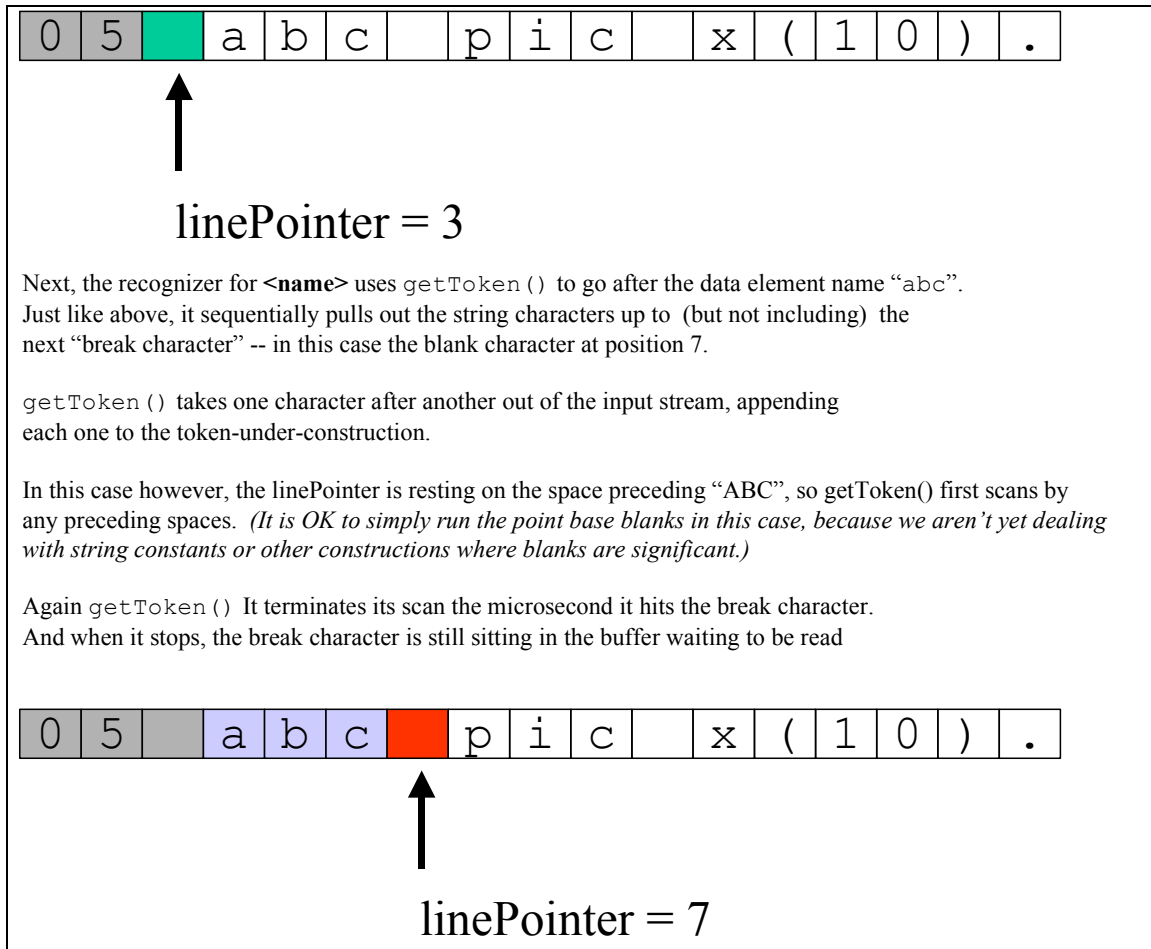


Figure 19- Buffer Pointer for Cobal Statement 2

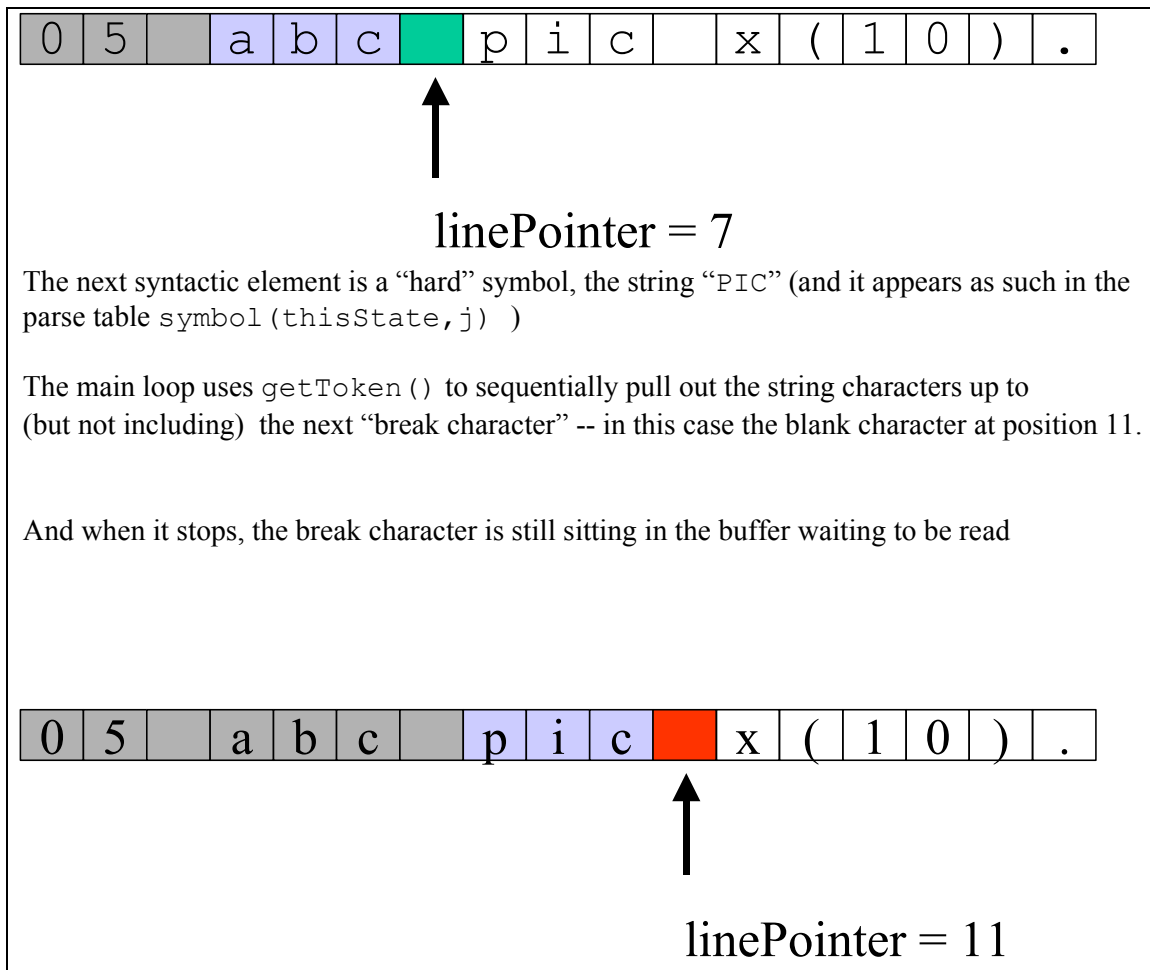


Figure 20- Buffer Pointer for Cobal Statement 3

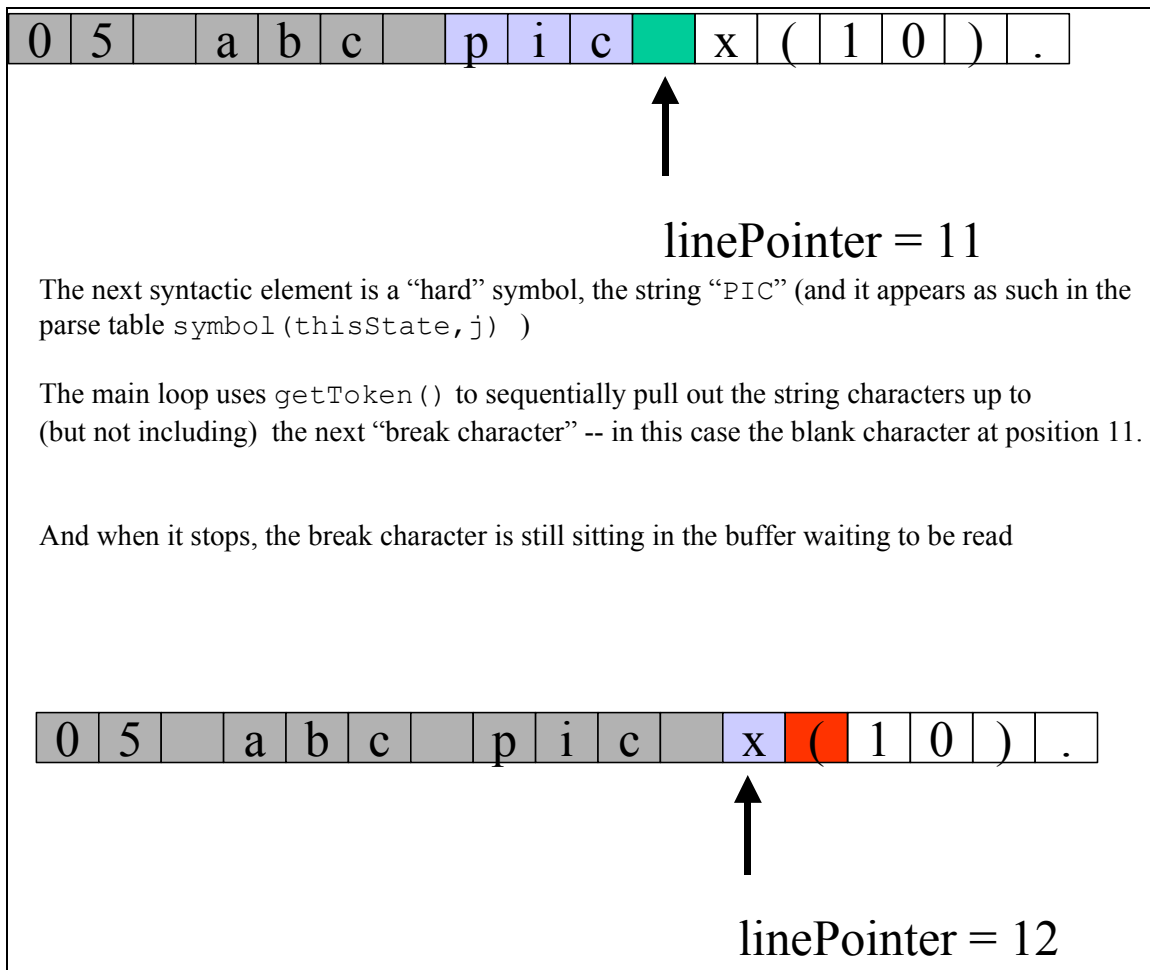


Figure 21- Buffer Pointer for Cobal Statement 4

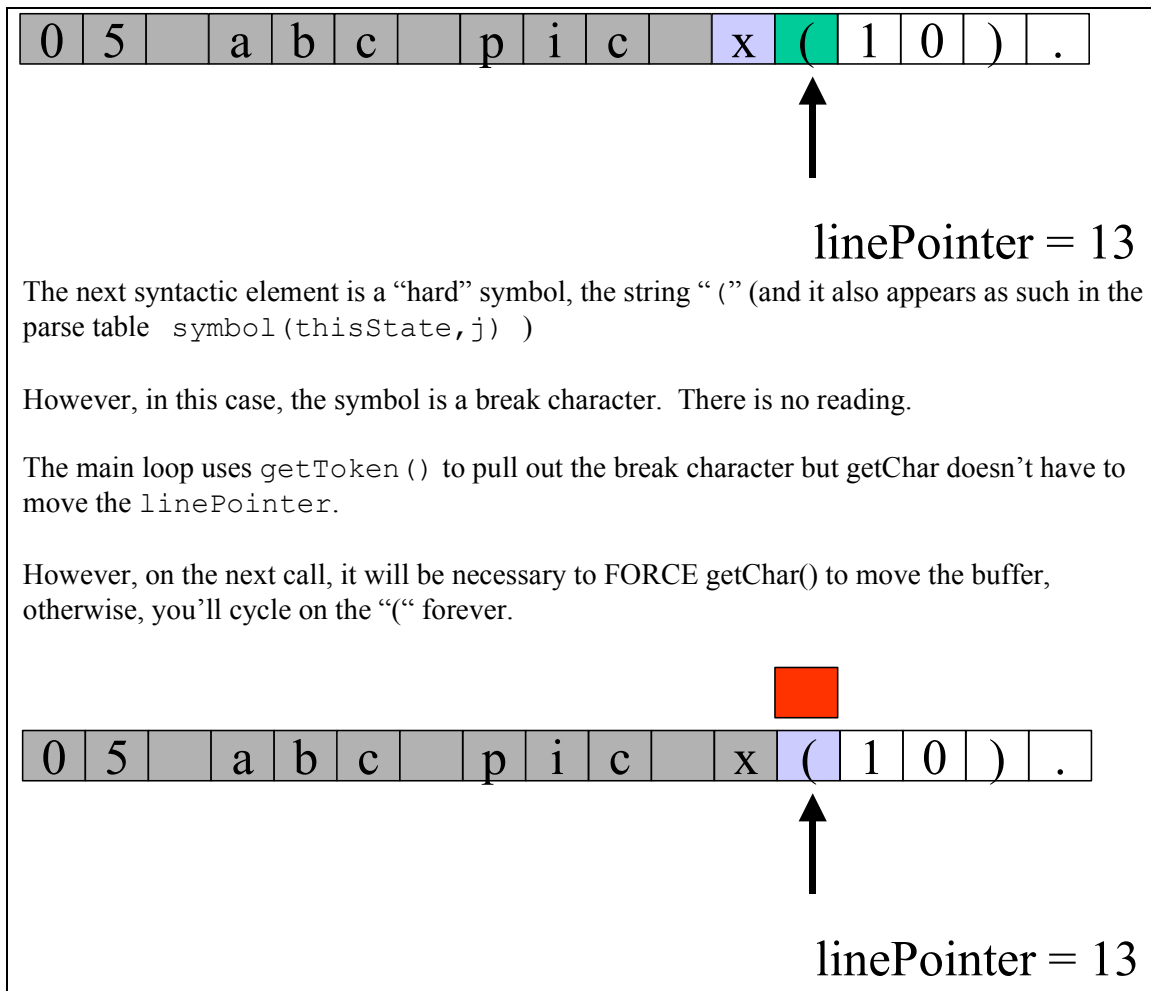


Figure 22- Buffer Pointer for Cobal Statement 5

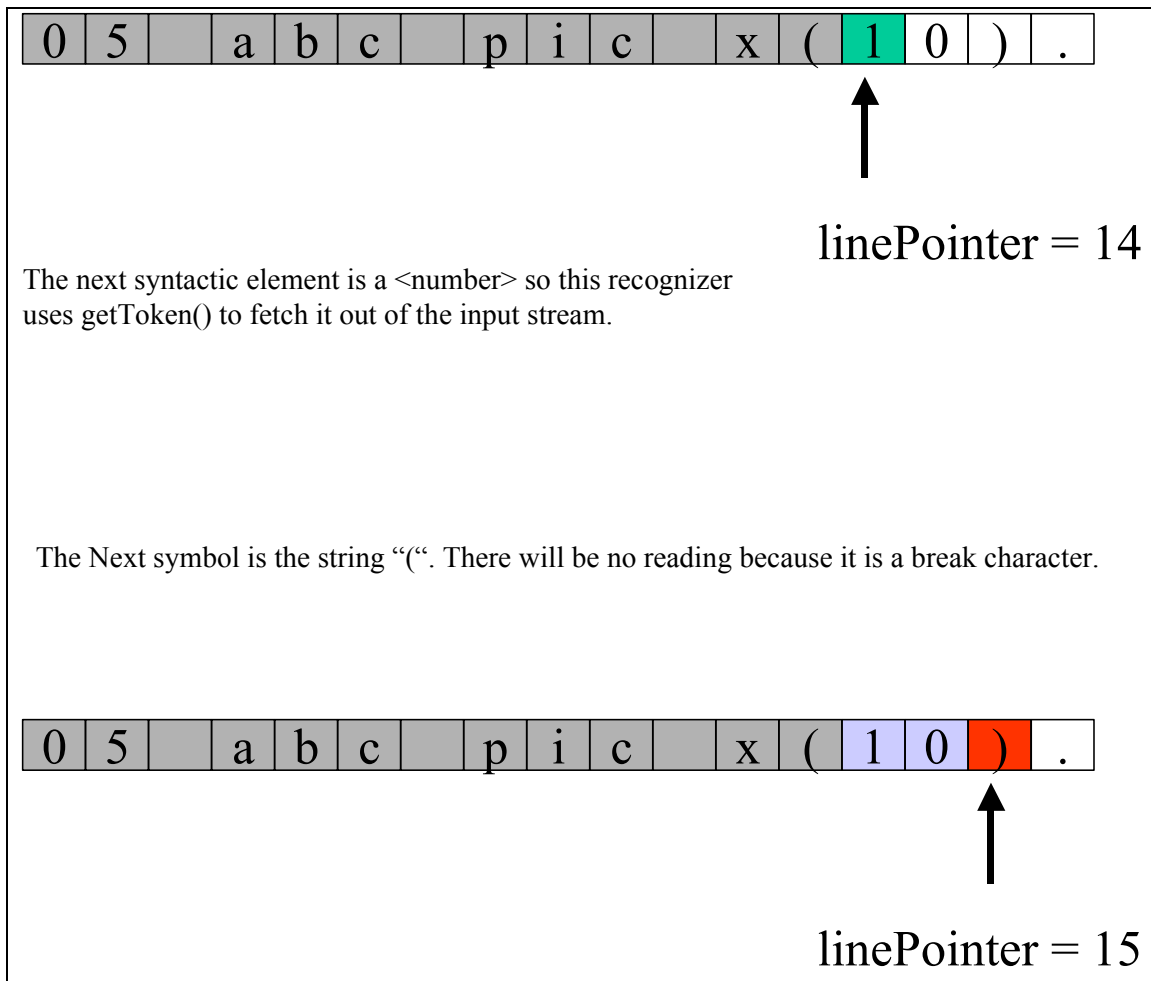


Figure 23- Buffer Pointer for Cobal Statement 6

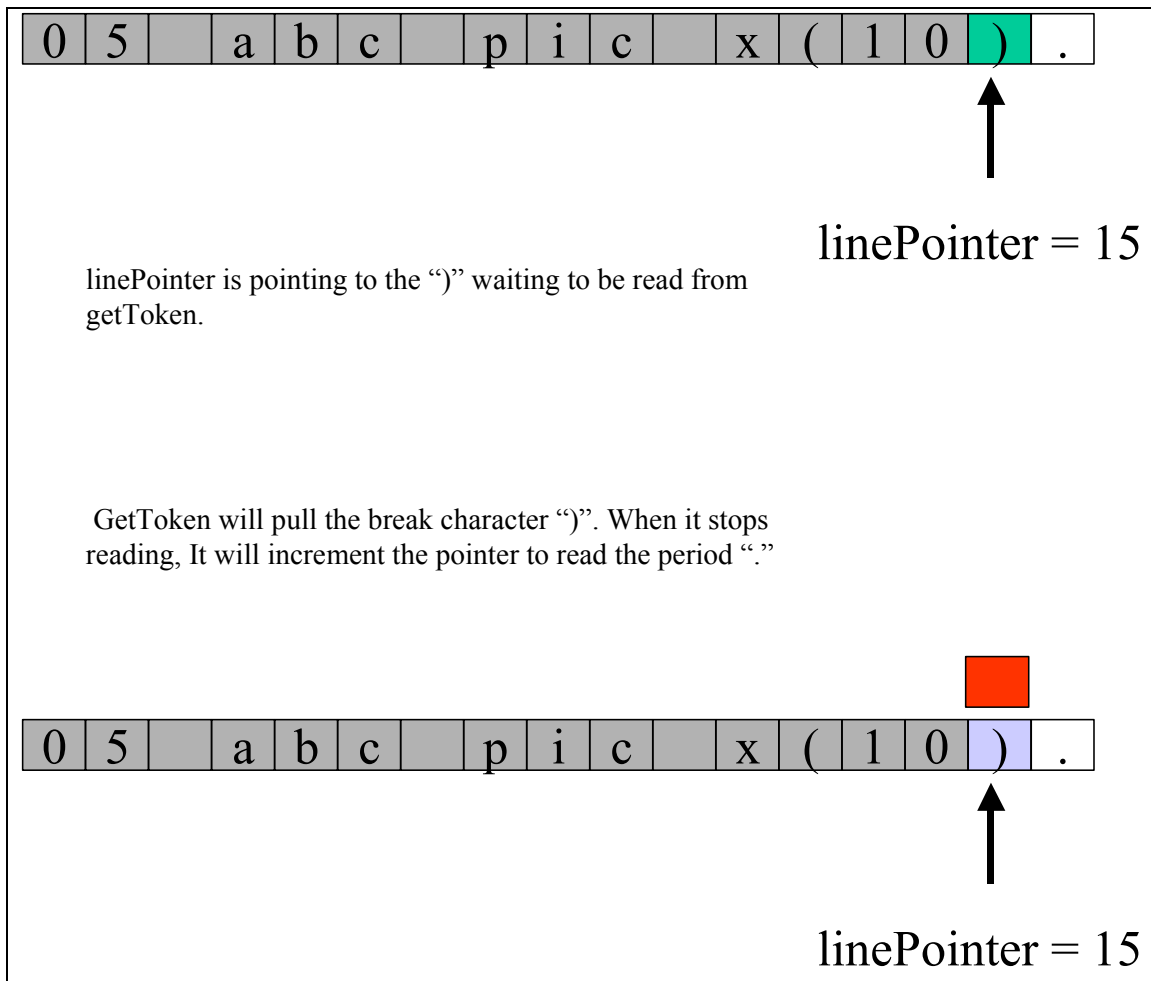


Figure 24- Buffer Pointer for Cobal Statement 7

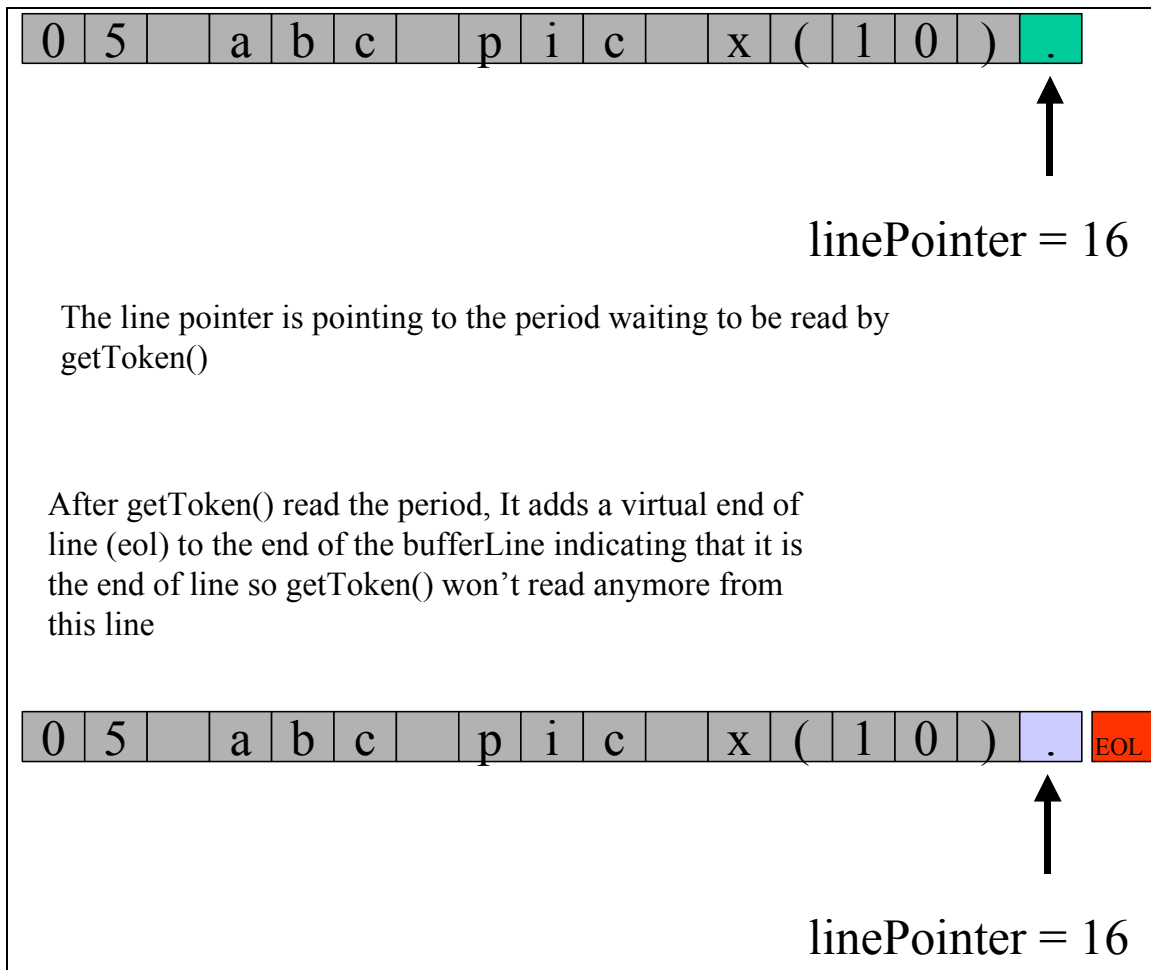


Figure 25- Buffer Pointer for Cobal Statement 8

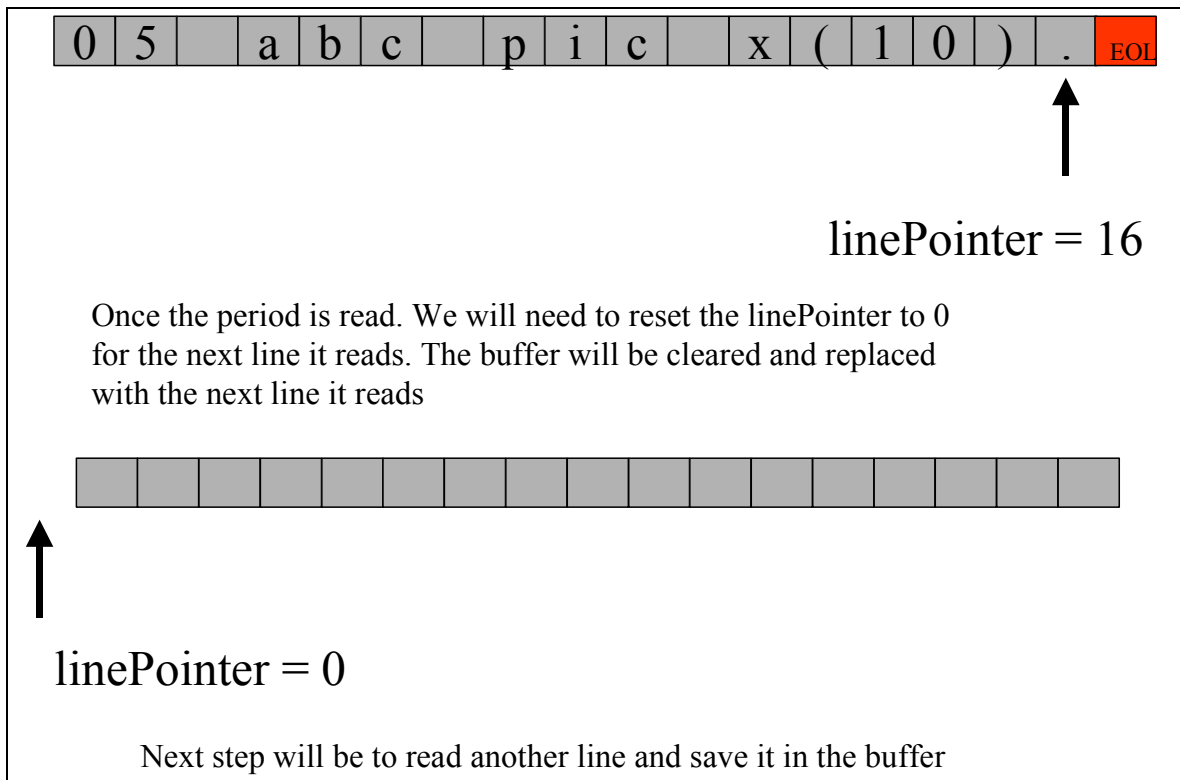


Figure 26- Buffer Pointer for Cobol Statement 9

### 7.2.1 Slightly Less Simple Picture Statement

<lessSimpleDef> = <number> <name> [pic | picture] X ( <number> ).

STATE	READ	ACTION	NEXT STATE
1	<number>	-	2
2	<name>	-	3
3	picture	-	4
3	pic	-	4
4	x	-	5
5	(	-	6
6	<number>	-	7
7	)	-	8
8	.	-	done

Figure 27 - State Table for Less Simple Picture Statement

## 7.2.2 Use of a <picture> Recognizer

STATE	READ	ACTION	NEXT STATE
1	<number>	-	2
2	<name>	-	3
3	<picture>	-	4
4	x	-	5
5	(	-	6
6	<number>	-	7
7	)	-	8
8	.	-	done
9	picture	-	done
9	pic	-	done

Figure 28 - State Table for Picture Recognizer

## 7.2.3 Grammar for COBOL variable definitions

These are all possible data definitions in COBOL

^ X ^ describes a string of X's of any length

```

<picDef> = <number> <name> .
<picDef> = <number> <name> [PICTURE IS | PIC] X ( <number> ).
<picDef> = <number> <name> [PICTURE IS | PIC] ^ X ^ .
<picDef> = <number> <name> [PICTURE IS | PIC] 9 ( <number> ).
<picDef> = <number> <name> [PICTURE IS | PIC] ^ 9 ^ .
<picDef> = <number> <name> [PICTURE IS | PIC] S9 ( <number> ).

```

STATE	READ	ACTION	NEXT STATE
1	<number>	-	2
2	<name>	-	3
3	.	-	Done
3	PICTURE IS	-	4
3	PIC	-	4
4	X	-	10
4	9	-	10
4	S	-	11
10	(	-	17
10	X	-	11
10	9	-	11

10	.	-	Done
11	X	-	11
11	9	-	11
11	.	-	Done
12	9	-	12
13	9	-	13
13	(	-	17
13	.	-	Done
16	(	-	17
17	<number>	-	18
18	)	-	19
19	.	-	Done

Figure 29 - State Table for Variable Definitions

Examples of variable definitions that are valid within this grammar:

```

20 EFF-DATE .
25 VALID-ADMIN-CODE-IND          PIC X
01  ERROR-SUB          PICTURE IS  S9(3)
05  DEPENDENT-ONLY          PIC XX
10  ERROR-0038          PICTURE IS  9(4)
05  VALUE-ONE          PIC S9
05  ERROR-CODE1-REDEF          PIC 9999.

```

NOTE:

<name> =

[A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, T, U, V, W, X, Y, Z, -]

<number> = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

## 7.2.4 Grammar for COBOL variable definitions with default values

These are all possible data definitions in COBOL

^ X^ describes a string of X's of any length

```

<picDef> = <number> <name> VALUES "<alphanum>".
<picDef> = <number> <name> [PICTURE IS | PIC] X ( <number> ).
<picDef> = <number> <name> [PICTURE IS | PIC] ^X^ .
<picDef> = <number> <name> [PICTURE IS | PIC] 9 ( <number> ).
<picDef> = <number> <name> [PICTURE IS | PIC] ^9^ .
<picDef> = <number> <name> [PICTURE IS | PIC] S9 ( <number> ).
<picDef> = <number> <name> [PICTURE IS | PIC] S^9^ .

```

STATE	READ	ACTION	NEXT STATE
1	<number>	-	2

2	<name>	-	3
3	.	-	Done
3	PICTURE IS	-	4
3	PIC	-	4
4	X	-	10
4	9	-	10
4	S	-	11
10	(	-	17
10	X	-	11
10	9	-	11
10	.	-	Done
11	X	-	11
11	9	-	11
11	.	-	Done
12	9	-	12
13	9	-	13
13	(	-	17
13	.	-	Done
16	(	-	17
17	<number>	-	18
18	)	-	19
19	.	-	Done

**Figure 30 - State Table for Variable Definitions with Default Values**

Examples of variable definitions that are valid within this grammar:

```

20 EFF-DATE.
25 VALID-ADMIN-CODE-IND PIC X.
01 ERROR-SUB          PICTURE IS  S9(3) .
05 DEPENDENT-ONLY    PIC XX.
10 ERROR-0038        PICTURE IS  9(4) .
05 VALUE-ONE         PIC S9.
05 ERROR-CODE1-REDEF PIC 9999.

```

NOTE:

<name> = [A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P,Q,R,S,T,U,V,W,X,Y,Z,-]

<number> = [0,1,2,3,4,5,6,7,8,9]

## 7.2.5 More General Syntax

### 7.2.5.1 Optional uses “[ ” and “]”

A symbol enclosed in square brackets means “this symbol is optional”.

### 7.2.5.2 Occurrence Counts

<digit> is a general lexical element.

You can qualify any element with a “Occurs” expression.

```
<digit> (“occurrence count”)
```

For example, we can say:

```
<digit>(1)      to mean exactly 1 digit
```

or

```
<digit>(10)     to mean exactly 10 digits.
```

We can also say:

```
<digit>(0 to 10)  to mean from 0 digits up to 10 digits
```

In essence, an occurrence count with a minimum of 0 means that the lexical element is optional.

## 7.2.6 Defined Expressions

We can define expressions, and then use them later to represent the definition value.

For example:

```
sign = [+ , -]
```

This tells us that the lexical element `sign` is an optional , but if it occurs, it must either be a + or -.

We can then define:

```
sign<digit>(1 to 10)
```

This is the same as having written:

```
[+,-]<digit>(1 to 10)
```

We could also write:

```
<whole_number> = <digit>(1 to 10)
```

Then we could write:

```
sign <whole_number>
```

and this would be the same as all of the following:

```
[+,-]<digit>(1 to 10)
```

```
sign <digit>(1 to 10)
```

```
[+,-]<whole_number>
```