Pattern Matching Chapter 16

The last chapter covered character strings and various operations on those strings. A very typical program reads a sequence of strings from the user and compares the strings to see if they match. For example, DOS' COMMAND.COM program reads command lines from the user and compares the strings the user types to fixed strings like "COPY", "DEL", "RENAME", and so on. Such commands are easy to *parse* because the set of allowable commands is finite and fixed. Sometimes, however, the strings you want to test for are not fixed; instead, they belong to a (possibly infinite) set of different strings. For example, if you execute the DOS command "DEL *.BAK", MS-DOS does not attempt to delete a file named "*.BAK". Instead, it deletes all files which match the *generic pattern* "*.BAK". This, of course, is any file which contains four or more characters and ends with ".BAK". In the MS-DOS world, a string containing characters like "*" and "?" are called *wildcards*; wildcard characters simply provide a way to specify different names via patterns. DOS' wildcard characters are very limited forms of what are known as *regular expressions*; regular expressions are very limited forms of patterns in general. This chapter describes how to create patterns that match a variety of character strings and write pattern matching routines to see if a particular string *matches* a given pattern.

16.1 An Introduction to Formal Language (Automata) Theory

Pattern matching, despite its low-key coverage, is a very important topic in computer science. Indeed, pattern matching is the main programming paradigm in several programming languages like Prolog, SNOBOL4, and Icon. Several programs you use all the time employ pattern matching as a major part of their work. MASM, for example, uses pattern matching to determine if symbols are correctly formed, expressions are proper, and so on. Compilers for high level languages like Pascal and C also make heavy use of pattern matching to parse source files to determine if they are syntactically correct. Surprisingly enough, an important statement known as *Church's Hypothesis* suggests that any computable function can be programmed as a pattern matching problem[1]. Of course, there is no guarantee that the solution would be efficient (they usually are not), but you could arrive at a correct solution. You probably wouldn't need to know about Turing machines (the subject of Church's hypothesis) if you're interested in writing, say, an accounts receivable package. However, there many situations where you may want to introduce the ability to match some generic patterns; so understanding some of the theory of pattern matching is important. This area of computer science goes by the stuffy names of *formal language theory* and *automata theory*. Courses in these subjects are often less than popular because they involve a lot of proofs, mathematics, and, well, theory. However, the concepts behind the proofs are quite simple and very useful. In this chapter we will not bother trying to prove everything about pattern matching. Instead, we will accept the fact that this stuff really works and just apply it. Nonetheless, we do have to discuss some of the results from automata theory, so without further ado…

16.1.1 Machines vs. Languages

You will find references to the term "machine" throughout automata theory literature. This term does not refer to some particular computer on which a program executes. Instead, this is usually some function that reads a string of symbols as input and produces one of two outputs: match or failure. A typical machine (or *automaton*) divides all possible strings into two sets – those strings that it *accepts* (or matches) and those string that it rejects. The *language* accepted by this machine is the set of all strings that the machine

1. Actually, Church's Hypothesis claims that any computable function can be computed on a Turing machine. However, the Turing machine is the ultimate pattern machine computer.

accepts. Note that this language could be infinite, finite, or the empty set (i.e., the machine rejects all input strings). Note that an infinite language does not suggest that the machine accepts all strings. It is quite possible for the machine to accept an infinite number of strings and reject an even greater number of strings. For example, it would be very easy to design a function which accepts all strings whose length is an even multiple of three. This function accepts an infinite number of strings (since there are an infinite number of strings whose length is a multiple of three) yet it rejects twice as many strings as it accepts. This is a very easy function to write. Consider the following 80x86 program that accepts all strings of length three (we'll assume that the carriage return character terminates a string):

```
MatchLen3       proc    near
                getc                    ;Get character #1.
                cmp     al, cr          ;Zero chars if EOLN.
                je      Accept
                getc                    ;Get character #2.
                cmp     al, cr
                je      Failure
                getc                    ;Get character #3.
                cmp     al, cr
                jne     MatchLen3
Failure:        mov     ax, 0           ;Return zero to denote failure.
                ret

Accept:         mov     ax, 1           ;Return one to denote success.
                ret
MatchLen3       endp
```

By tracing through this code, you should be able to easily convince yourself that it returns one in ax if it succeeds (reads a string whose length is a multiple of three) and zero otherwise.

Machines are inherently *recognizers*. The machine itself is the embodiment of a *pattern*. It recognizes any input string which matches the built-in pattern. Therefore, a codification of these automatons is the basic job of the programmer who wants tomatch some patterns.

There are many different classes of machines and the languages they recognize. From simple to complex, the major classifications are *deterministic finite state automata* (which are equivalent to *nondeterministic finite state automata*), *deterministic push down automata, nondeterministic push down automata,* and *Turing machines*. Each successive machine in this list provides a superset of the capabilities of the machines appearing before it. The only reason we don't use Turing machines for everything is because they are more complex to program than, say, a deterministic finite state automaton. If you can match the pattern you want using a deterministic finite state automaton, you'll probably want to code it that way rather than as a Turing machine.

Each class of machine has a class of languages associated with it. Deterministic and nondeterministic finite state automata recognize the *regular* languages. Nondeterministic push down automata recognize the *context free* languages[2]. Turing machines can recognize all recognizable languages. We will discuss each of these sets of languages, and their properties, in turn.

16.1.2 Regular Languages

The regular languages are the least complex of the languages described in the previous section. That does not mean they are less useful; in fact, patterns based on regular expression are probably more common than any other.

2. Deterministic push down automata recognize only a subset of the context free languages.

16.1.2.1 Regular Expressions

The most compact way to specify the strings that belong to a regular language is with a *regular expression*. We shall define, recursively, a regular expression with the following rules:

- $\varnothing$ (the empty set) is a regular language and denotes the empty set.
- ε is a regular expression[3]. It denotes the set of languages containing only the empty string: $\{\varepsilon\}$.
- Any single symbol, *a*, is a regular expression (we will use lower case characters to denote arbitrary symbols). This single symbol matches exactly one character in the input string, that character must be equal to the single symbol in the regular expression. For example, the pattern "m" matches a single "m" character in the input string.

Note that $\varnothing$ and ε are not the same. The empty set is a regular language that does not accept *any* strings, including strings of length zero. If a regular language is denoted by $\{\varepsilon\}$, then it accepts exactly one string, the string of length zero. This latter regular language accepts something, the former does not.

The three rules above provide our *basis* for a recursive definition. Now we will define regular expressions recursively. In the following definitions, assume that *r*, *s*, and *t* are any valid regular expressions.

- Concatenation. If *r* and *s* are regular expressions, so is *rs*. The regular expression *rs* matches any string that begins with a string matched by *r* and ends with a string matched by *s*.
- Alternation/Union. If *r* and *s* are regular expressions, so is $r \mid s$ (read this as *r* **or** *s*) This is equivalent to $r \cup s$, (read as *r* union *s*). This regular expression matches any string that *r* or *s* matches.
- Intersection. If *r* and *s* are regular expressions, so is $r \cap s$. This is the set of all strings that both *r* **and** *s* match.
- Kleene Star. If *r* is a regular expression, so is r^*. This regular expression matches zero or more occurrences of *r*. That is, it matches ε, *r*, *rr*, *rrr*, *rrrr*, ...
- Difference. If *r* and *s* are regular expressions, so is *r-s*. This denotes the set of strings matched by *r* that are not also matched by *s.*
- Precedence. If *r* is a regular expression, so is (*r*). This matches any string matched by *r* alone. The normal algebraic associative and distributive laws apply here, so $(r \mid s)\, t$ is equivalent to $rt \mid st$.

These operators following the normal associative and distributive laws and exhibit the following precedences:

```
Highest:        (r)
                Kleene Star
                Concatentation
                Intersection
                Difference
Lowest:         Alternation/Union
```

Examples:

```
(r | s) t = rt | st
rs* = r(s*)
r ∪ t - s = r ∪ (t - s)
r ∩ t - s = (r ∩ t) - s
```

Generally, we'll use parenthesis to avoid any ambiguity

Although this definition is sufficient for an automata theory class, there are some practical aspects to this definition that leave a little to be desired. For example, to define a

3. The empty string is the string of length zero, containing no symbols.

regular expression that matches a single alphabetic character, you would need to create something like ($a \mid b \mid c \mid \ldots \mid y \mid z$). Quite a lot of typing for such a trivial character set. Therefore, we shall add some notation to make it easier to specify regular expressions.

- Character Sets. Any set of characters surrounded by brackets, e.g., [abcdefg] is a regular expression and matches a single character from that set. You can specify ranges of characters using a dash, i.e., "[a-z]" denotes the set of lower case characters and this regular expression matches a single lower case character.
- Kleene Plus. If r is a regular expression, so is r^+. This regular expression matches one or more occurrences of r. That is, it matches $r, rr, rrr, rrrr, \ldots$ The precedence of the Kleene Plus is the same as for the Kleene Star. Note that $r^+ = rr^*$.
- Σ represents any single character from the allowable character set. Σ^* represents the set of all possible strings. The regular expression Σ^*-r is the *complement* of r – that is, the set of all strings that r does not match.

With the notational baggage out of the way, it's time to discuss how to actually use regular expressions as pattern matching specifications. The following examples should give a suitable introduction.

Identifiers: Most programming languages like Pascal or C/C++ specify legal forms for identifiers using a regular expression. Expressed in English terms, the specification is something like "An identifier must begin with an alphabetic character and is followed by zero or more alphanumeric or underscore characters." Using the regular expression (RE) syntax described in this section, an identifier is

[a-zA-Z][a-zA-Z0-9_]*

Integer Consts: A regular expression for integer constants is relatively easy to design. An integer constant consists of an optional plus or minus followed by one or more digits. The RE is

(+ | - | ε) [0-9]$^+$

Note the use of the empty string (ε) to make the plus or minus optional.

Real Consts: Real constants are a bit more complex, but still easy to specify using REs. Our definition matches that for a real constant appearing in a Pascal program – an optional plus or minus, following by one or more digits; optionally followed by a decimal point and zero or more digits; optionally followed by an "e" or an "E" with an optional sign and one or more digits:

(+ | - | ε) [0-9]$^+$ ("." [0-9]* | ε) (((e | E) (+ | - | ε) [0-9]$^+$) | ε)

Since this RE is relatively complex, we should dissect it piece by piece. The first parenthetical term gives us the optional sign. One or more digits are mandatory before the decimal point, the second term provides this. The third term allows an optional decimal point followed by zero or more digits. The last term provides for an optional exponent consisting of "e" or "E" followed by an optional sign and one or more digits.

Reserved Words: It is very easy to provide a regular expression that matches a set of reserved words. For example, if you want to create a regular expression that matches MASM's reserved words, you could use an RE similar to the following:

(mov | add | and | … | mul)

Even: The regular expression (ΣΣ)* matches all strings whose length is a multiple of two.

Sentences: The regular expression:

(Σ* " "*)* run (" "$^+$ (Σ* " "$^+$ | ε)) fast (" " Σ*)*

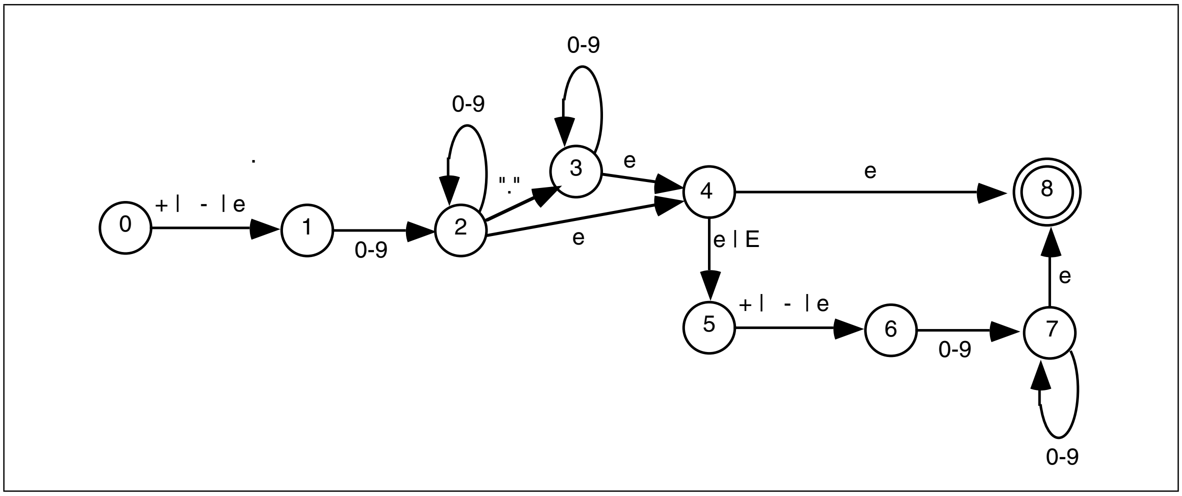


Figure 16.1 NFA for Regular Expression (+ | - | e) [0-9]+ ("." [0-9]* | e) (((e | E) (+ | - | e) [0-9]+) | e)

matches all strings that contain the separate words "run" followed by "fast" somewhere on the line. This matches strings like "I want to run very fast" and "run as fast as you can" as well as "run fast."

While REs are convenient for specifying the pattern you want to recognize, they are not particularly useful for creating programs (i.e., "machines") that actually recognize such patterns. Instead, you should first convert an RE to a *nondeterministic finite state automaton*, or NFA. It is very easy to convert an NFA into an 80x86 assembly language program; however, such programs are rarely efficient as they might be. If efficiency is a big concern, you can convert the NFA into a *deterministic finite state automaton* (DFA) that is also easy to convert to 80x86 assembly code, but the conversion is usually far more efficient.

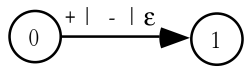
16.1.2.2 Nondeterministic Finite State Automata (NFAs)

An NFA is a directed graph with *state numbers* associated with each node and *characters or character strings* associated with each edge of the graph. A distinguished state, the *starting state*, determines where the machine begins attempting to match an input string. With the machine in the starting state, it compares input characters against the characters or strings on each edge of the graph. If a set of input characters matches one of the edges, the machine can change states from the node at the start of the edge (the tail) to the state at the end of the edge (the head).

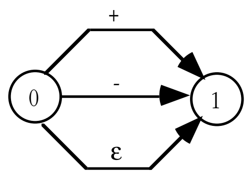
Certain other states, known as *final* or *accepting* states, are usually present as well. If a machine winds up in a final state after exhausting all the input characters, then that machine *accepts* or *matches* that string. If the machine exhausts the input and winds up in a state that is not a final state, then that machine *rejects* the string. Figure 16.1 shows an example NFA for the floating point RE presented earlier.

By convention, we'll always assume that the starting state is state zero. We will denote final states (there may be more than one) by using a double circle for the state (state eight is the final state above).

An NFA always begins with an input string in the starting state (state zero). On each edge coming out of a state there is either ε, a single character, or a character string. To help unclutter the NFA diagrams, we will allow expressions of the form " xxx | yyy | zzz | …" where xxx, yyy, and zzz are ε, a single character, or a character string. This corresponds to multiple edges from one state to the other with a single item on each edge. In the example above,



is equivalent to



Likewise, we will allow *sets* of characters, specified by a string of the form x-y, to denote the expression x | x+1 | x+2 | … | y.

Note that an NFA accepts a string if there is *some* path from the starting state to an accepting state that exhausts the input string. There may be multiple paths from the starting state to various final states. Furthermore, there may be some particular path from the starting state to a non-accepting state that exhausts the input string. This does not necessarily mean the NFA rejects that string; if there is some other path from the starting state to an accepting state, then the NFA accepts the string. An NFA rejects a string only if there are *no* paths from the starting state to an accepting state that exhaust the string.

Passing through an accepting state does not cause the NFA to accept a string. You must wind up in a final state *and* exhaust the input string.

To process an input string with an NFA, begin at the starting state. The edges leading out of the starting state will have a character, a string, or ε associated with them. If you choose to move from one state to another along an edge with a single character, then remove that character from the input string and move to the new state along the edge traversed by that character. Likewise, if you choose to move along an edge with a character string, remove that character string from the input string and switch to the new state. If there is an edge with the empty string, ε, then you may elect to move to the new state given by that edge without removing any characters from the input string.

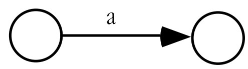
Consider the string "1.25e2" and the NFA in Figure 16.1. From the starting state we can move to state one using the ε string (there is no leading plus or minus, so ε is our only option). From state one we can move to state two by matching the "1" in our input string with the set 0-9; this eats the "1" in our input string leaving ".25e2". In state two we move to state three and eat the period from the input string, leaving "25e2". State three loops on itself with numeric input characters, so we eat the "2" and "5" characters at the beginning of our input string and wind up back in state three with a new input string of "e2". The next input character is "e", but there is no edge coming out of state three with an "e" on it; there is, however, an ε-edge, so we can use that to move to state four. This move does not change the input string. In state four we can move to state five on an "e" character. This eats the "e" and leaves us with an input string of "2". Since this is not a plus or minus character, we have to move from state five to state six on the ε edge. Movement from state six to state seven eats the last character in our string. Since the string is empty (and, in particular, it does not contain any digits), state seven cannot loop back on itself. We are currently in state seven (which is not a final state) and our input string is exhausted. However, we can move to state eight (the accepting state) since the transition between states seven and eight is an ε edge. Since we are in a final state and we've exhausted the input string, This NFA accepts the input string.

16.1.2.3 Converting Regular Expressions to NFAs

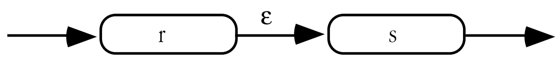
If you have a regular expression and you want to build a machine that recognizes strings in the regular language specified by that expression, you will need to convert the

RE to and NFA. It turns out to be very easy to convert a regular expression to an NFA. To do so, just apply the following rules:

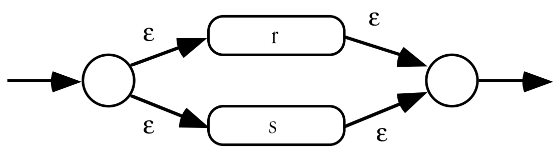
- The NFA representing regular language denoted by the regular expression ∅ (the empty set) is a single, non-accepting state.
- If a regular expression contains an ε, a single character, or a string, create two states and draw an arc between them with ε, the single character, or the string as the label. For example, the RE “a” is converted to an NFA as



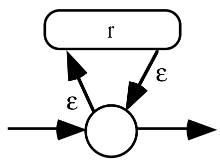
- Let the symbol  denote an NFA which recognizes some regular language specified by some regular expression *r*, *s*, or *t*. If a regular expression takes the form *rs* then the corresponding NFA is



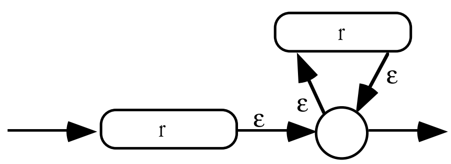
- If a regular expression takes the form *r* | *s*, then the corresponding NFA is



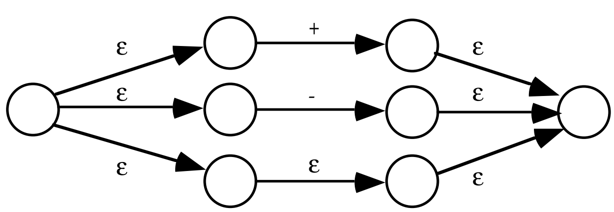
- If a regular expression takes the form *r** then the corresponding NFA is



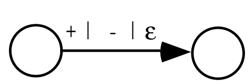
All of the other forms of regular expressions are easily synthesized from these, therefore, converting those other forms of regular expressions to NFAs is a simple two-step process, convert the RE to one of these forms, and then convert this form to the NFA. For example, to convert r^+ to an NFA, you would first convert r^+ to rr^*. This produces the NFA:



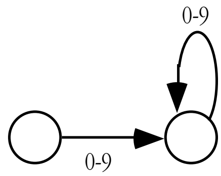
The following example converts the regular expression for an integer constant to an NFA. The first step is to create an NFA for the regular expression (+ | - | ε). The complete construction becomes



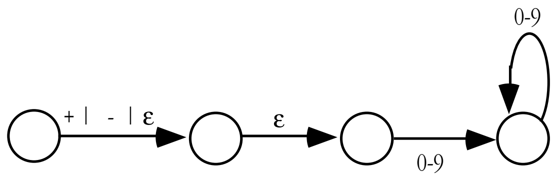
Although we can obviously optimize this to



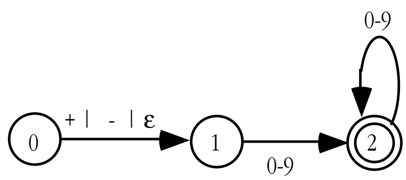
The next step is to handle the [0-9]$^+$ regular expression; after some minor optimization, this becomes the NFA



Now we simply concatenate the results to produce:



All we need now are starting and final states. The starting state is always the first state of the NFA created by the conversion of the leftmost item in the regular expression. The final state is always the last state of the NFA created by the conversion of the rightmost item in the regular expression. Therefore, the complete regular expression for integer constants (after optimizing out the middle edge above, which serves no purpose) is

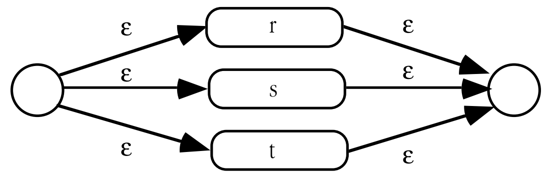


16.1.2.4 Converting an NFA to Assembly Language

There is only one major problem with converting an NFA to an appropriate matching function – NFAs are *nondeterministic*. If you're in some state and you've got some input character, say "a", there is no guarantee that the NFA will tell you what to do next. For example, there is no requirement that edges coming out of a state have unique labels. You could have two or more edges coming out of a state, all leading to different states on the single character "a". If an NFA accepts a string, it only guarantees that there is some path that leads to an accepting state, there is no guarantee that this path will be easy to find.

The primary technique you will use to resolve the nondeterministic behavior of an NFA is *backtracking.* A function that attempts to match a pattern using an NFA begins in the starting state and tries to match the first character(s) of the input string against the edges leaving the starting state. If there is only one match, the code must follow that edge. However, if there are two possible edges to follow, then the code must arbitrarily choose one of them *and remember the others as well as the current point in the input string*. Later, if it turns out the algorithm guessed an incorrect edge to follow, it can return back and try one of the other alternatives (i.e., it *backtracks* and tries a different path). If the algorithm exhausts all alternatives without winding up in a final state (with an empty input string), then the NFA does not accept the string.

Probably the easiest way to implement backtracking is via procedure calls. Let us assume that a matching procedure returns the carry flag set if it succeeds (i.e., accepts a string) and returns the carry flag clear if it fails (i.e., rejects a string). If an NFA offers multiple choices, you could implement that portion of the NFA as follows:




```
AltRST          proc    near
                push    ax          ;The purpose of these two instructions
                mov     ax, di      ; is to preserve di in case of failure.
                call    r
                jc      Success
                mov     di, ax      ;Restore di (it may be modified by r).
                call    s
                jc      Success
                mov     di, ax      ;Restore di (it may be modified by s).
                call    t
Success:        pop     ax          ;Restore ax.
                ret
AltRST          endp
```

If the r matching procedure succeeds, there is no need to try s and t. On the other hand, if r fails, then we need to try s. Likewise, if r and s both fail, we need to try t. AltRST will fail only if r, s, and t all fail. This code assumes that es:di points at the input string to match. On return, es:di points at the next available character in the string after a match *or it points at some arbitrary point if the match fails.* This code assumes that r, s, and t all preserve the ax register, so it preserves a pointer to the current point in the input string in ax in the event r or s fail.

To handle the individual NFA associated with simple regular expressions (i.e., matching ε or a single character) is not hard at all. Suppose the matching function r matches the regular expression (+ | - | ε). The complete procedure for r is

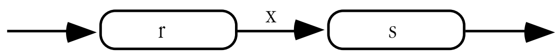
```
r               proc    near
                cmp     byte ptr es:[di], '+'
                je      r_matched
                cmp     byte ptr es:[di], '-'
                jne     r_nomatch
r_matched:      inc     di
r_nomatch:      stc
                ret
r               endp
```

Note that there is no explicit test for ε. If ε is one of the alternatives, the function attempts to match one of the other alternatives first. If none of the other alternatives succeed, then the matching function will succeed anyway, although it does not consume any input characters (which is why the above code skips over the inc di instruction if it does not match "+" or "-"). Therefore, any matching function that has ε as an alternative will always succeed.

Of course, not all matching functions succeed in every case. Suppose the s matching function accepts a single decimal digit. the code for s might be the following:

```
s               proc    near
                cmp     byte ptr es:[di], '0'
                jb      s_fails
                cmp     byte ptr es:[di], '9'
                ja      s_fails
                inc     di
                stc
                ret

s_fails:        clc
                ret
s               endp
```

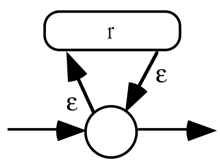
If an NFA takes the form:



Where x is any arbitrary character or string or ε, the corresponding assembly code for this procedure would be

```
ConcatRxS       proc    near
                call    r
                jnc     CRxS_Fail               ;If no r, we won't succeed

; Note, if x=ε then simply delete the following three statements.
; If x is a string rather than a single character, put the the additional
; code to match all the characters in the string.

                cmp     byte ptr es:[di], 'x'
                jne     CRxS_Fail
                inc     di

                call    s
                jnc     CRxS_Fail
                stc                     ;Success!
                ret

CRxS_Fail:      clc
                ret
ConcatRxS       endp
```

If the regular expression is of the form r* and the corresponding NFA is of the form



Then the corresponding 80x86 assembly code can look something like the following:

```
RStar           proc    near
                call    r
                jc      RStar
                stc
                ret
RStar           endp
```

Regular expressions based on the Kleene star always succeed since they allow zero or more occurrences. That is why this code always returns with the carry flag set.

The Kleene Plus operation is only slightly more complex, the corresponding (slightly optimized) assembly code is

```
RPlus           proc    near
                call    r
                jnc     RPlus_Fail
RPlusLp:        call    r
                jc      RPlusLp
                stc
                ret

RPlus_Fail:     clc
                ret
RPlus           endp
```

Note how this routine fails if there isn't at least one occurrence of r.

A major problem with backtracking is that it is potentially inefficient. It is very easy to create a regular expression that, when converted to an NFA and assembly code, generates considerable backtracking on certain input strings. This is further exacerbated by the fact that matching routines, if written as described above, are generally very short; so short, in fact, that the procedure calls and returns make up a significant portion of the execution time. Therefore, pattern matching in this fashion, although easy, can be slower than it has to be.

This is just a taste of how you would convert REs to NFAs to assembly language. We will not go into further detail in this chapter; not because this stuff isn't interesting to know, but because you will rarely use these techniques in a real program. If you need high performance pattern matching you would not use nondeterministic techniques like these. If you want the ease of programming offered by the conversion of an NFA to assembly language, you still would not use this technique. Instead, the UCR Standard Library provides very powerful pattern matching facilities (which exceed the capabilities of NFAs), so you would use those instead; but more on that a little later.

16.1.2.5 Deterministic Finite State Automata (DFAs)

Nondeterministic finite state automata, when converted to actual program code, may suffer from performance problems because of the backtracking that occurs when matching a string. Deterministic finite state automata solve this problem by comparing different strings *in parallel*. Whereas, in the worst case, an NFA may require n comparisons, where n is the sum of the lengths of all the strings the NFA recognizes, a DFA requires only m comparisons (worst case), where m is the length of the longest string the DFA recognizes.

For example, suppose you have an NFA that matches the following regular expression (the set of 80x86 real-mode mnemonics that begin with an "A"):

```
( AAA | AAD | AAM | AAS | ADC | ADD | AND )
```

A typical implementation as an NFA might look like the following:

```
MatchAMnem      proc    near
                strcmpl
                byte    "AAA",0
                je      matched
                strcmpl
                byte    "AAD",0
                je      matched
                strcmpl
                byte    "AAM",0
                je      matched
                strcmpl
                byte    "AAS",0
                je      matched
                strcmpl
                byte    "ADC",0
                je      matched
                strcmpl
                byte    "ADD",0
                je      matched
                strcmpl
                byte    "AND",0
                je      matched
                clc
                ret

matched:        add     di, 3
                stc
                ret
MatchAMnem      endp
```

If you pass this NFA a string that it doesn't match, e.g., "AAND", it must perform seven string comparisons, which works out to about 18 character comparisons (plus all the overhead of calling strcmpl). In fact, a DFA can determine that it does not match this character string by comparing only three characters.

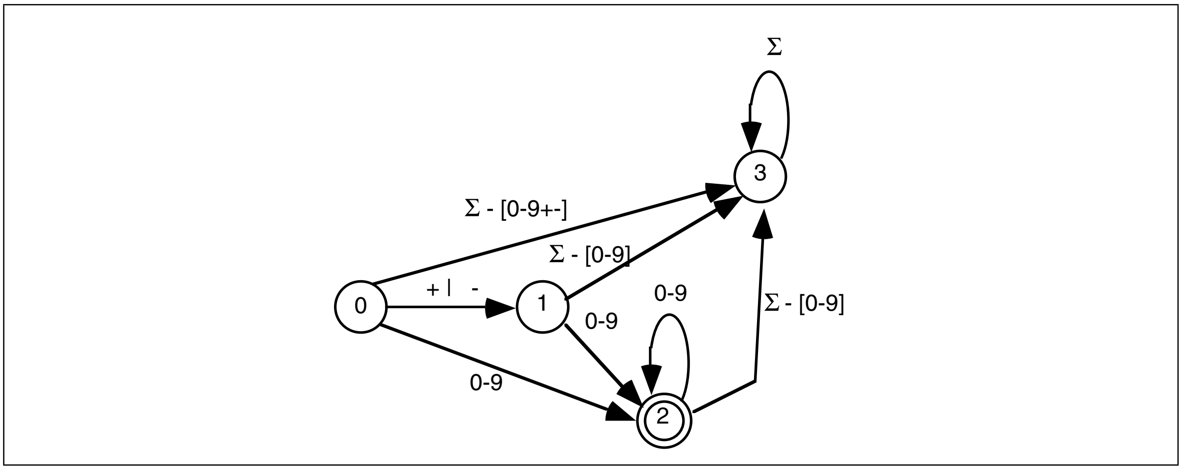


Figure 16.2 DFA for Regular Expression (+ | - | ε) $[0\text{-}9]^+$

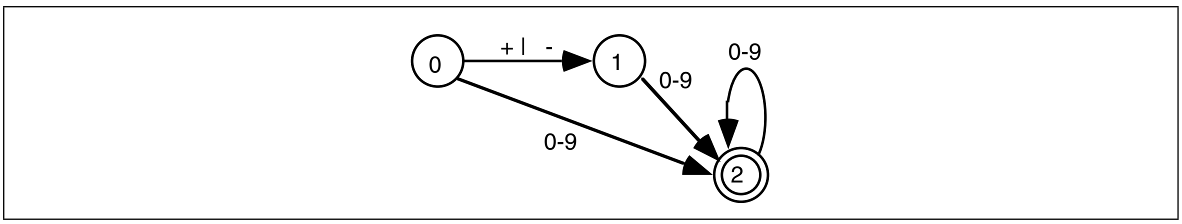


Figure 16.3 Simplified DFA for Regular Expression (+ | - | ε) $[0\text{-}9]^+$

A DFA is a special form of an NFA with two restrictions. First, there must be *exactly* one edge coming out of each node for each of the possible input characters; this implies that there must be one edge for each possible input symbol *and* you may not have two edges with the same input symbol. Second, you cannot move from one state to another on the empty string, ε. A DFA is deterministic because at each state the next input symbol determines the next state you will enter. Since each input symbol has an edge associated with it, there is never a case where a DFA "jams" because you cannot leave the state on that input symbol. Similarly, the new state you enter is never ambiguous because there is only one edge leaving any particular state with the current input symbol on it. Figure 16.2 shows the DFA that handles integer constants described by the regular expression

(+ | - | ε) $[0\text{-}9]^+$

Note than an expression of the form "Σ - [0-9]" means *any character except a digit*; that is, the *complement* of the set [0-9].

State three is a *failure state.* It is not an accepting state and once the DFA enters a failure state, it is stuck there (i.e., it will consume all additional characters in the input string without leaving the failure state). Once you enter a failure state, the DFA has already rejected the input string. Of course, this is not the only way to reject a string; the DFA above, for example, rejects the empty string (since that leaves you in state zero) and it rejects a string containing only a "+" or a "-" character.

DFAs generally contain more states than a comparable NFA. To help keep the size of a DFA under control, we will allow a few shortcuts that, in no way, affect the operation of a DFA. First, we will remove the restriction that there be an edge associated with each possible input symbol leaving every state. Most of the edges leaving a particular state lead to the failure state. Therefore, our first simplification will be to allow DFAs to drop the edges that lead to a failure state. If a input symbol is not represented on an outgoing edge from some state, we will assume that it leads to a failure state. The above DFA with this simplification appears in Figure 16.2.

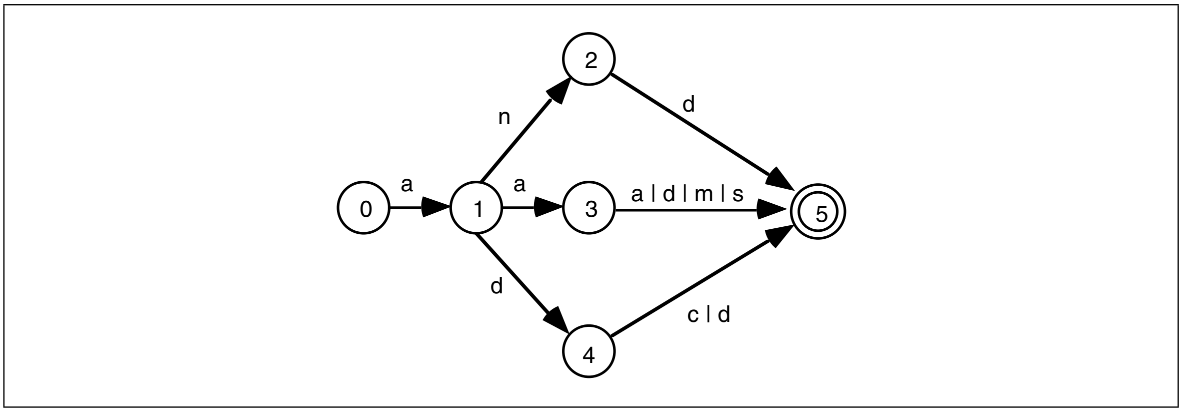
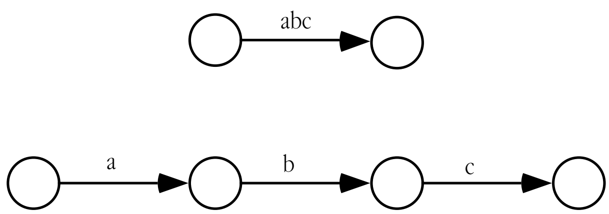


Figure 16.4 DFA that Recognizes AND, AAA, AAD, AAM, AAS, ADD, and ADC

A second shortcut, that is actually present in the two examples above, is to allow sets of characters (or the alternation symbol, "|") to associate several characters with a single edge. Finally, we will also allow strings attached to an edge. This is a shorthand notation for a list of states which recognize each successive character, i.e., the following two DFAs are equivalent:



Returning to the regular expression that recognizes 80x86 real-mode mnemonics beginning with an "A", we can construct a DFA that recognizes such strings as shown in Figure 16.4.

If you trace through this DFA by hand on several accepting and rejecting strings, you will discover than it requires no more than six character comparisons to determine whether the DFA should accept or reject an input string.

Although we are not going to discuss the specifics here, it turns out that regular expressions, NFAs, and DFAs are all equivalent. That is, you can convert anyone of these to the others. In particular, you can always convert an NFA to a DFA. Although the conversion isn't totally trivial, especially if you want an *optimized* DFA, it is always possible to do so. Converting between all these forms is beginning to leave the scope of this text. If you are interested in the details, *any* text on formal languages or automata theory will fill you in.

16.1.2.6 Converting a DFA to Assembly Language

It is relatively straightforward to convert a DFA to a sequence of assembly instructions. For example, the assembly code for the DFA that accepts the A-mnemonics in the previous section is

```
DFA_A_Mnem      proc     near
                cmp      byte ptr es:[di], 'A'
                jne      Fail
                cmp      byte ptr es:[di+1], 'A'
                je       DoAA
                cmp      byte ptr es:[di+1], 'D'
                je       DoAD
                cmp      byte ptr es:[di+1], 'N'
                je       DoAN
```

```
Fail:           clc
                ret

DoAN:           cmp     byte ptr es:[di+2], 'D'
                jne     Fail
Succeed:        add     di, 3
                stc
                ret

DoAD:           cmp     byte ptr es:[di+2], 'D'
                je      Succeed
                cmp     byte ptr es:[di+2], 'C'
                je      Succeed
                clc                                     ;Return Failure
                ret

DoAA:           cmp     byte ptr es:[di+2], 'A'
                je      Succeed
                cmp     byte ptr es:[di+2], 'D'
                je      Succeed
                cmp     byte ptr es:[di+2], 'M'
                je      Succeed
                cmp     byte ptr es:[di+2], 'S'
                je      Succeed
                clc
                ret
DFA_A_Mnem      endp
```

Although this scheme works and is considerably more efficient than the coding scheme for NFAs, writing this code can be tedious, especially when converting a large DFA to assembly code. There is a technique that makes converting DFAs to assembly code almost trivial, although it can consume quite a bit of space – to use state machines. A simple state machine is a two dimensional array. The columns are indexed by the possible characters in the input string and the rows are indexed by state number (i.e., the states in the DFA). Each element of the array is a new state number. The algorithm to match a given string using a state machine is trivial, it is

```
state := 0;
while (another input character ) do begin

       ch := next input character ;
       state := StateTable [state][ch];

end;
if (state in FinalStates) then accept
else reject;
```

FinalStates is a set of accepting states. If the current state number is in this set after the algorithm exhausts the characters in the string, then the state machine accepts the string, otherwise it rejects the string.

The following state table corresponds to the DFA for the "A" mnemonics appearing in the previous section:

Table 62: State Machine for 80x86 “A” Instructions DFA

State	A	C	D	M	N	S	Else
0	1	F	F	F	F	F	F
1	3	F	4	F	2	F	F
2	F	F	5	F	F	F	F
3	5	F	5	5	F	5	F
4	F	5	5	F	F	F	F
5	F	F	F	F	F	F	F
F	F	F	F	F	F	F	F

State five is the only accepting state.

There is one major drawback to using this table driven scheme – the table will be quite large. This is not apparent in the table above because the column labelled “Else” hides considerable detail. In a true state table, you will need one column for each possible input character. since there are 256 possible input characters (or at least 128 if you’re willing to stick to seven bit ASCII), the table above will have 256 columns. With only one byte per element, this works out to about 2K for this small state machine. Larger state machines could generate very large tables.

One way to reduce the size of the table at a (very) slight loss in execution speed is to classify the characters before using them as an index into a state table. By using a single 256-byte lookup table, it is easy to reduce the state machine to the table above. Consider the 256 byte lookup table that contains:

- A one at positions *Base+"a"* and *Base+"A"*,
- A two at locations *Base+"c"* and *Base+"C"*,
- A three at locations *Base+"d"* and *Base+"D"*,
- A four at locations *Base+"m"* and *Base+"M"*,
- A five at locations *Base+"n"* and *Base+"N"*,
- A six at locations *Base+"s"* and *Base+"S"*, and
- A zero everywhere else.

Now we can modify the above table to produce:

Table 63: Classified State Machine Table for 80x86 “A” Instructions DFA

State	0	1	2	3	4	5	6	7
0	6	1	6	6	6	6	6	6
1	6	3	6	4	6	2	6	6
2	6	6	6	5	6	6	6	6
3	6	5	6	5	5	6	5	6
4	6	6	5	5	6	6	6	6
5	6	6	6	6	6	6	6	6
6	6	6	6	6	6	6	6	6

The table above contains an extra column, “7”, that we will not use. The reason for adding the extra column is to make it easy to index into this two dimensional array (since the extra column lets us multiply the state number by eight rather than seven).

Assuming Classify is the name of the lookup table, the following 80386 code recognizes the strings specified by this DFA:

```
DFA2_A_Mnem     proc
                push    ebx                         ;Ptr to Classify.
                push    eax                         ;Current character.
                push    ecx                         ;Current state.
                xor     eax, eax                    ;EAX := 0
                mov     ebx, eax                    ;EBX := 0
                mov     ecx, eax                    ;ECX (state) := 0
                lea     bx, Classify
WhileNotEOS:    mov     al, es:[di]                 ;Get next input char.
                cmp     al, 0                       ;At end of string?
                je      AtEOS
                xlat                                ;Classify character.
                mov     cl, State_Tbl[eax+ecx*8]    ;Get new state #.
                inc     di                          ;Move on to next char.
                jmp     WhileNotEOS

AtEOS:          cmp     cl, 5                       ;In accepting state?
                stc                                 ;Assume acceptance.
                je      Accept
                clc
Accept:         pop     ecx
                pop     eax
                pop     ebx
                ret
DFA2_A_Mnem     endp
```

The nice thing about this DFA (the DFA is the combination of the classification table, the state table, and the above code) is that it is very easy to modify. To handle any other state machine (with eight or fewer character classifications) you need only modify the Classification array, the State_Tbl array, the lea bx, Classify statement and the statements at label AtEOS that determine if the machine is in a final state. The assembly code does not get more complex as the DFA grows in size. The State_Tbl array will get larger as you add more states, but this does not affect the assembly code.

Of course, the assembly code above *does* assume there are exactly eight columns in the matrix. It is easy to generalize this code by inserting an appropriate imul instruction to multiply by the size of the array. For example, had we gone with seven columns rather than eight, the code above would be

```
DFA2_A_Mnem     proc
                push    ebx                         ;Ptr to Classify.
                push    eax                         ;Current character.
                push    ecx                         ;Current state.
                xor     eax, eax                    ;EAX := 0
                mov     ebx, eax                    ;EBX := 0
                mov     ecx, eax                    ;ECX (state) := 0
                lea     bx, Classify
WhileNotEOS:    mov     al, es:[di]                 ;Get next input char.
                cmp     al, 0                       ;At end of string?
                je      AtEOS
                xlat                                ;Classify character.
                imul    cx, 7
                movzx   ecx, State_Tbl[eax+ecx]     ;Get new state #.
                inc     di                          ;Move on to next char.
                jmp     WhileNotEOS

AtEOS:          cmp     cl, 5                       ;In accepting state?
                stc                                 ;Assume acceptance.
                je      Accept
                clc
Accept:         pop     ecx
                pop     eax
                pop     ebx
                ret
DFA2_A_Mnem     endp
```

Although using a state table in this manner simplifies the assembly coding, it does suffer from two drawbacks. First, as mentioned earlier, it is slower. This technique has to execute all the statements in the while loop for each character it matches; and those instructions are not particularly fast ones, either. The second drawback is that you've got to create the state table for the state machine; that process is tedious and error prone.

If you need the absolute highest performance, you can use the state machine techniques described in (see “State Machines and Indirect Jumps” on page 529). The trick here is to represent each state with a short segment of code and its own one dimensional state table. Each entry in the table is the target address of the segment of code representing the next state. The following is an example of our “A Mnemonic” state machine written in this fashion. The only difference is that the zero byte is classified to value seven (zero marks the end of the string, we will use this to determine when we encounter the end of the string). The corresponding state table would be:

Table 64: Another State Machine Table for 80x86 “A” Instructions DFA

State	0	1	2	3	4	5	6	7
0	6	1	6	6	6	6	6	6
1	6	3	6	4	6	2	6	6
2	6	6	6	5	6	6	6	6
3	6	5	6	5	5	6	5	6
4	6	6	5	5	6	6	6	6
5	6	6	6	6	6	6	6	5
6	6	6	6	6	6	6	6	6

The 80x86 code is

```
DFA3_A_Mnem     proc
                push    ebx
                push    eax
                push    ecx
                xor     eax, eax

                lea     ebx, Classify
State0:         mov     al, es:[di]
                xlat
                inc     di
                jmp     cseg:State0Tbl[eax*2]

State0Tbl       word    State6, State1, State6, State6
                word    State6, State6, State6, State6

State1:         mov     al, es:[di]
                xlat
                inc     di
                jmp     cseg:State1Tbl[eax*2]

State1Tbl       word    State6, State3, State6, State4
                word    State6, State2, State6, State6

State2:         mov     al, es:[di]
                xlat
                inc     di
                jmp     cseg:State2Tbl[eax*2]

State2Tbl       word    State6, State6, State6, State5
                word    State6, State6, State6, State6

State3:         mov     al, es:[di]
                xlat
                inc     di
                jmp     cseg:State3Tbl[eax*2]
```

```
State3Tbl         word     State6, State5, State6, State5
                  word     State5, State6, State5, State6

State4:           mov      al, es:[di]
                  xlat
                  inc      di
                  jmp      cseg:State4Tbl[eax*2]

State4Tbl         word     State6, State6, State5, State5
                  word     State6, State6, State6, State6

State5:           mov      al, es:[di]
                  cmp      al, 0
                  jne      State6
                  stc
                  pop      ecx
                  pop      eax
                  pop      ebx
                  ret

State6:           clc
                  pop      ecx
                  pop      eax
                  pop      ebx
                  ret
```

There are two important features you should note about this code. First, it only executes four instructions per character comparison (fewer, on the average, than the other techniques). Second, the instant the DFA detects failure it stops processing the input characters. The other table driven DFA techniques blindly process the entire string, even after it is obvious that the machine is locked in a failure state.

Also note that this code treats the accepting and failure states a little differently than the generic state table code. This code recognizes the fact that once we're in state five it will either succeed (if EOS is the next character) or fail. Likewise, in state six this code knows better than to try searching any farther.

Of course, this technique is not as easy to modify for different DFAs as a simple state table version, but it is quite a bit faster. If you're looking for speed, this is a good way to code a DFA.

16.1.3 Context Free Languages

Context free languages provide a superset of the regular languages – if you can specify a class of patterns with a regular expression, you can express the same language using a *context free grammar*. In addition, you can specify many languages that are not regular using context free grammars (CFGs).

Examples of languages that are context free, but not regular, include the set of all strings representing common arithmetic expressions, legal Pascal or C source files[4], and MASM macros. Context free languages are characterized by *balance* and *nesting*. For example, arithmetic expression have balanced sets of parenthesis. High level language statements like repeat…until allow nesting and are always balanced (e.g., for every repeat there is a corresponding until statement later in the source file).

There is only a slight extension to the regular languages to handle context free languages – function calls. In a regular expression, we only allow the objects we want to match and the specific RE operators like "|", "*", concatenation, and so on. To extend regular languages to context free languages, we need only add recursive function calls to regular expressions. Although it would be simple to create a syntax allowing function calls

4. Actually, C and Pascal are *not* context free languages, but Computer Scientists like to treat them as though they were.

within a regular expression, computer scientists use a different notation altogether for context free languages – a context free grammar.

A context free grammar contains two types of symbols: *terminal symbols* and *nonterminal symbols*. Terminal symbols are the individual characters and strings that the context free grammar matches plus the empty string, ε. Context free grammars use nonterminal symbols for function calls and definitions. In our context free grammars we will use italic characters to denote nonterminal symbols and standard characters to denote terminal symbols.

A context free grammar consists of a set of function definitions known as *productions*. A production takes the following form:

Function_Name → «list of terminal and nonterminal symbols»

The function name to the left hand side of the arrow is called the *left hand side* of the production. The function body, which is the list of terminals and nonterminal symbols, is called the *right hand side* of the production. The following is a grammar for simple arithmetic expressions:

```
expression → expression + factor
expression → expression - factor
expression → factor
factor → factor * term
factor → factor / term
factor → term
term → IntegerConstant
term → ( expression )
IntegerConstant → digit
IntegerConstant → digit IntegerConstant
digit → 0
digit → 1
digit → 2
digit → 3
digit → 4
digit → 5
digit → 6
digit → 7
digit → 8
digit → 9
```

Note that you may have multiple definitions for the same function. Context-free grammars behave in a non-deterministic fashion, just like NFAs. When attempting to match a string using a context free grammar, a string matches if there exists some matching function which matches the current input string. Since it is very common to have multiple productions with identical left hand sides, we will use the alternation symbol from the regular expressions to reduce the number of lines in the grammar. The following two subgrammars are identical:

```
expression → expression + factor
expression → expression - factor
expression → factor
```

The above is equivalent to:

```
expression → expression + factor  |  expression - factor  |  factor
```

The full arithmetic grammar, using this shorthand notation, is

```
expression → expression + factor  |  expression - factor  |  factor
factor → factor * term  |  factor / term  |  term
term → IntegerConstant  |  ( expression )
```

IntegerConstant → *digit* | *digit IntegerConstant*
digit → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

One of the nonterminal symbols, usually the first production in the grammar, is the *starting symbol*. This is roughly equivalent to the starting state in a finite state automaton. The starting symbol is the first matching function you call when you want to test some input string to see if it is a member of a context free language. In the example above, *expression* is the starting symbol.

Much like the NFAs and DFAs recognize strings in a regular language specified by a regular expression, *nondeterministic pushdown automata* and *deterministic pushdown automata* recognize strings belonging to a context free language specified by a context free grammar. We will not go into the details of these pushdown automata (or *PDAs*) here, just be aware of their existence. We can match strings directly with a grammar. For example, consider the string

7+5*(2+1)

To match this string, we begin by calling the starting symbol function, *expression*, using the function *expression* → *expression* + *factor*. The first plus sign suggests that the *expression* term must match "7" and the *factor* term must match "5*(2+1)". Now we need to match our input string with the pattern *expression* + *factor*. To do this, we call the *expression* function once again, this time using the *expression* → *factor* production. This give us the *reduction*:

expression ⇒ *expression* + *factor* ⇒ *factor* + *factor*

The ⇒ symbol denotes the application of a nonterminal function call (a reduction).

Next, we call the factor function, using the production *factor* → *term* to yield the reduction:

expression ⇒ *expression* + *factor* ⇒ *factor* + *factor* ⇒ *term* + *factor*

Continuing, we call the *term* function to produce the reduction:

expression ⇒ *expression* + *factor* ⇒ *factor* + *factor* ⇒ *term* + *factor* ⇒ *IntegerConstant* + *factor*

Next, we call the *IntegerConstant* function to yield:

expression ⇒ *expression* + *factor* ⇒ *factor* + *factor* ⇒ *term* + *factor* ⇒ *IntegerConstant* + *factor* ⇒ 7 + *factor*

At this point, the first two symbols of our generated string match the first two characters of the input string, so we can remove them from the input and concentrate on the items that follow. In succession, we call the *factor* function to produce the reduction 7 + *factor* * *term* and then we call *factor, term*, and *IntegerConstant* to yield 7 + 5 * *term*. In a similar fashion, we can reduce the term to "(*expression*)" and reduce expression to "2+1". The complete *derivation* for this string is

```
expression  ⇒ expression + factor
            ⇒ factor + factor
            ⇒ term + factor
            ⇒ IntegerConstant + factor
            ⇒ 7 + factor
            ⇒ 7 + factor * term
            ⇒ 7 + term * term
            ⇒ 7 + IntegerConstant * term
            ⇒ 7 + 5 * term
            ⇒ 7 + 5 * ( expression )
            ⇒ 7 + 5 * ( expression + factor )
            ⇒ 7 + 5 * ( factor + factor )
            ⇒ 7 + 5 * ( IntegerConstant + factor )
            ⇒ 7 + 5 * ( 2 + factor )
            ⇒ 7 + 5 * ( 2 + term )
            ⇒ 7 + 5 * ( 2 + IntegerConstant )
            ⇒ 7 + 5 * ( 2 + 1 )
```

The final reduction completes the derivation of our input string, so the string 7+5*(2+1) is in the language specified by the context free grammar.

16.1.4 Eliminating Left Recursion and Left Factoring CFGs

In the next section we will discuss how to convert a CFG to an assembly language program. However, the technique we are going to use to do this conversion will require that we modify certain grammars before converting them. The arithmetic expression grammar in the previous section is a good example of such a grammar – one that is *left recursive*.

Left recursive grammars pose a problem for us because the way we will typically convert a production to assembly code is to call a function corresponding to a nonterminal and compare against the terminal symbols. However, we will run into trouble if we attempt to convert a production like the following using this technique:

expression → *expression* + *factor*

Such a conversion would yield some assembly code that looks roughly like the following:

```
expression      proc    near
                call    expression
                jnc     fail
                cmp     byte ptr es:[di], '+'
                jne     fail
                inc     di
                call    factor
                jnc     fail
                stc
                ret
Fail:           clc
                ret
expression      endp
```

The obvious problem with this code is that it will generate an infinite loop. Upon entering the expression function this code immediately calls expression recursively, which immediately calls expression recursively, which immediately calls expression recursively, ... Clearly, we need to resolve this problem if we are going to write any real code to match this production.

The trick to resolving left recursion is to note that if there is a production that suffers from left recursion, there must be *some* production with the same left hand side that is not left recursive[5]. All we need do is rewrite the left recursive call in terms of the production that does not have any left recursion. This sound like a difficult task, but it's actually quite easy.

To see how to eliminate left recursion, let X_i and Y_j represent any set of terminal symbols or nonterminal symbols that do not have a right hand side beginning with the nonterminal A. If you have some productions of the form:

$$A \rightarrow AX_1 \mid AX_2 \mid \ldots \mid AX_n \mid Y_1 \mid Y_2 \mid \ldots \mid Y_m$$

You will be able to translate this to an equivalent grammar without left recursion by replacing each term of the form $A \rightarrow Y_i$ by $A \rightarrow Y_i A$ and each term of the form $A \rightarrow AX_i$ by $A' \rightarrow X_i A' \mid \varepsilon$. For example, consider three of the productions from the arithmetic grammar:

expression → *expression* + *factor*
expression → *expression* - *factor*
expression → *factor*

In this example A corresponds to *expression*, X_1 corresponds to "+ *factor* ", X_2 corresponds to "- *factor* ", and Y_1 corresponds to "*factor* ". The equivalent grammar without left recursion is

expression → *factor E'*
E' → - *factor E'*
E' → + *factor E'*
E' → ε

The complete arithmetic grammar, with left recursion removed, is

expression → *factor E'*
E' → + *factor E'* | - *factor E'* | ε
factor → *term F'*
F' → * *term F'* | / *term F'* | ε
term → *IntegerConstant* | (*expression*)
IntegerConstant → *digit* | *digit IntegerConstant*
digit → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Another useful transformation on a grammar is to left factor the grammar. This can reduce the need for backtracking, improving the performance of your pattern matching code. Consider the following CFG fragment:

stmt → if *expression* then *stmt* endif
stmt → if *expression* then *stmt* else *stmt* endif

These two productions begin with the same set of symbols. Either production will match all the characters in an if statement up to the point the matching algorithm encounters the first else or endif. If the matching algorithm processes the first statement up to the point of the endif terminal symbol and encounters the else terminal symbol instead, it must backtrack all the way to the if symbol and start over. This can be terribly inefficient because of the recursive call to *stmt* (imagine a 10,000 line program that has a single if statement around the entire 10,000 lines, a compiler using this pattern matching technique would have to recompile the entire program from scratch if it used backtracking in this fashion). However, by left factoring the grammar before converting it to program code, you can eliminate the need for backtracking.

To left factor a grammar, you collect all productions that have the same left hand side and begin with the same symbols on the right hand side. In the two productions above, the common symbols are "if *expression* then *stmt* ". You combine the common strings into a single production and then append a new nonterminal symbol to the end of this new production, e.g.,

5. If this is not the case, the grammar does not match any finite length strings.

stmt → if *expression* then *stmt NewNonTerm*

Finally, you create a new set of productions using this new nonterminal for each of the suffixes to the common production:

NewNonTerm → endif | else *stmt* endif

This eliminates backtracking because the matching algorithm can process the if, the *expression*, the then, and the stmt before it has to choose between endif and else.

16.1.5 Converting REs to CFGs

Since the context free languages are a superset of the regular languages, it should come as no surprise that it is possible to convert regular expressions to context free grammars. Indeed, this is a very easy process involving only a few intuitive rules.

1) If a regular expression simply consists of a sequence of characters, xyz, you can easily create a production for this regular expression of the form $P \rightarrow$ xyz. This applies equally to the empty string, ε.
2) If r and s are two regular expression that you've converted to CFG productions R and S, and you have a regular expression rs that you want to convert to a production, simply create a new production of the form $T \rightarrow R\ S$.
3) If r and s are two regular expression that you've converted to CFG productions R and S, and you have a regular expression $r \mid s$ that you want to convert to a production, simply create a new production of the form $T \rightarrow R \mid S$.
4) If r is a regular expression that you've converted to a production, R, and you want to create a production for r^*, simply use the production $RStar \rightarrow R\ RStar \mid \varepsilon$.
5) If r is a regular expression that you've converted to a production, R, and you want to create a production for r^+, simply use the production $RPlus \rightarrow R\ RPlus \mid R$.
6) For regular expressions there are operations with various precedences. Regular expressions also allow parenthesis to override the default precedence. This notion of precedence does not carry over into CFGs. Instead, you must encode the precedence directly into the grammar. For example, to encode $R\ S^*$ you would probably use productions of the form:

   $T \rightarrow R\ SStar$
   $SStar \rightarrow S\ SStar \mid \varepsilon$

   Likewise, to handle a grammar of the form $(RS)^*$ you could use productions of the form:

   $T \rightarrow RS\ T \mid \varepsilon$
   $RS \rightarrow R\ S$

16.1.6 Converting CFGs to Assembly Language

If you have removed left recursion and you've left factored a grammar, it is very easy to convert such a grammar to an assembly language program that recognizes strings in the context free language.

The first convention we will adopt is that es:di always points at the start of the string we want to match. The second convention we will adopt is to create a function for each nonterminal. This function returns success (carry set) if it matches an associated subpattern, it returns failure (carry clear) otherwise. If it succeeds, it leaves di pointing at the next character is the staring *after* the matched pattern; if it fails, it preserves the value in di across the function call.

To convert a set of productions to their corresponding assembly code, we need to be able to handle four things: terminal symbols, nonterminal symbols, alternation, and the empty string. First, we will consider simple functions (nonterminals) which do not have multiple productions (i.e., alternation).

If a production takes the form $T \rightarrow \varepsilon$ and there are no other productions associated with *T*, then this production always succeeds. The corresponding assembly code is simply:

```
T               proc    near
                stc
                ret
T               endp
```

Of course, there is no real need to ever call *T* and test the returned result since we know it will always succeed. On the other hand, if *T* is a *stub* that you intend to fill in later, you should call *T*.

If a production takes the form $T \rightarrow$ xyz, where xyz is a string of one or more terminal symbols, then the function returns success if the next several input characters match xyz, it returns failure otherwise. Remember, if the prefix of the input string matches xyz, then the matching function must advance di beyond these characters. If the first characters of the input string does not match xyz, it must preserve di. The following routines demonstrate two cases, where xyz is a single character and where xyz is a string of characters:

```
T1              proc    near
                cmp     byte ptr es:[di], 'x'           ;Single char.
                je      Success
                clc                                     ;Return Failure.
                ret

Success:        inc     di                              ;Skip matched char.
                stc                                     ;Return success.
                ret
T1              endp


T2              proc    near
                call    MatchPrefix
                byte    'xyz',0
                ret
T2              endp
```

MatchPrefix is a routine that matches the prefix of the string pointed at by es:di against the string following the call in the code stream. It returns the carry set and adjusts di if the string in the code stream is a prefix of the input string, it returns the carry flag clear and preserves di if the literal string is not a prefix of the input. The MatchPrefix code follows:

```
MatchPrefix     proc    far             ;Must be far!
                push    bp
                mov     bp, sp
                push    ax
                push    ds
                push    si
                push    di

                lds     si, 2[bp]       ;Get the return address.
CmpLoop:        mov     al, ds:[si]     ;Get string to match.
                cmp     al, 0           ;If at end of prefix,
                je      Success         ; we succeed.
                cmp     al, es:[di]     ;See if it matches prefix,
                jne     Failure         ; if not, immediately fail.
                inc     si
                inc     di
                jmp     CmpLoop

Success:        add     sp, 2           ;Don't restore di.
                inc     si              ;Skip zero terminating byte.
                mov     2[bp], si       ;Save as return address.
                pop     si
                pop     ds
                pop     ax
```

```
                pop     bp
                stc                     ;Return success.
                ret

Failure:        inc     si                      ;Need to skip to zero byte.
                cmp     byte ptr ds:[si], 0
                jne     Failure
                inc     si
                mov     2[bp], si               ;Save as return address.

                pop     di
                pop     si
                pop     ds
                pop     ax
                pop     bp
                clc                             ;Return failure.
                ret
MatchPrefix     endp
```

If a production takes the form $T \rightarrow R$, where R is a nonterminal, then the T function calls R and returns whatever status R returns, e.g.,

```
T               proc    near
                call    R
                ret
T               endp
```

If the right hand side of a production contains a string of terminal and nonterminal symbols, the corresponding assembly code checks each item in turn. If any check fails, then the function returns failure. If all items succeed, then the function returns success. For example, if you have a production of the form $T \rightarrow R$ abc S you could implement this in assembly language as

```
T               proc    near
                push    di                      ;If we fail, must preserve
di.
                call    R
                jnc     Failure
                call    MatchPrefix
                byte    "abc",0
                jnc     Failure
                call    S
                jnc     Failure
                add     sp, 2                   ;Don't preserve di if we
succeed.
                stc
                ret

Failure:        pop     di
                clc
                ret
T               endp
```

Note how this code preserves di if it fails, but does not preserve di if it succeeds.

If you have multiple productions with the same left hand side (i.e., alternation), then writing an appropriate matching function for the productions is only slightly more complex than the single production case. If you have multiple productions associated with a single nonterminal on the left hand side, then create a sequence of code to match each of the individual productions. To combine them into a single matching function, simply write the function so that it succeeds if any one of these code sequences succeeds. If one of the productions is of the form $T \rightarrow e$, then test the other conditions first. If none of them could be selected, the function succeeds. For example, consider the productions:

$$E' \rightarrow + \; factor \; E' \mid - \; factor \; E' \mid \varepsilon$$

This translates to the following assembly code:

```
EPrime          proc        near
                push        di
                cmp         byte ptr es:[di], '+'
                jne         TryMinus
                inc         di
                call        factor
                jnc         EP_Failed
                call        EPrime
                jnc         EP_Failed
Success:        add         sp, 2
                stc
                ret

TryMinus:       cmp         byte ptr es:[di], '-'
                jne         EP_Failed
                inc         di
                call        factor
                jnc         EP_Failed
                call        EPrime
                jnc         EP_Failed
                add         sp, 2
                stc
                ret

EP_Failed:      pop         di
                stc                         ;Succeed because of E' -> ε
                ret
EPrime          endp
```

This routine always succeeds because it has the production $E' \rightarrow \varepsilon$. This is why the stc instruction appears after the EP_Failed label.

To invoke a pattern matching function, simply load es:di with the address of the string you want to test and call the pattern matching function. On return, the carry flag will contain one if the pattern matches the string up to the point returned in di. If you want to see if the entire string matches the pattern, simply check to see if es:di is pointing at a zero byte when you get back from the function call. If you want to see if a string belongs to a context free language, you should call the function associated with the starting symbol for the given context free grammar.

The following program implements the arithmetic grammar we've been using as examples throughout the past several sections. The complete implementation is

```
; ARITH.ASM
;
; A simple recursive descent parser for arithmetic strings.

                  .xlist
                  include   stdlib.a
                  includelibstdlib.lib
                  .list


dseg              segment   para public 'data'

; Grammar for simple arithmetic grammar (supports +, -, *, /):
;
; E -> FE'
; E' -> + F E' | - F E' | <empty string>
; F -> TF'
; F' -> * T F' | / T F' | <empty string>
; T -> G | (E)
; G -> H | H G
; H -> 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
;


InputLine         byte      128 dup (0)

dseg              ends
```

```
cseg            segment  para public 'code'
                assume   cs:cseg, ds:dseg

; Matching functions for the grammar.
; These functions return the carry flag set if they match their
; respective item. They return the carry flag clear if they fail.
; If they fail, they preserve di. If they succeed, di points to
; the first character after the match.


; E -> FE'

E               proc     near
                push     di
                call     F            ;See if F, then E', succeeds.
                jnc      E_Failed
                call     EPrime
                jnc      E_Failed
                add      sp, 2        ;Success, don't restore di.
                stc
                ret

E_Failed:       pop      di           ;Failure, must restore di.
                clc
                ret
E               endp



; E' -> + F E' | - F E' | ε

EPrime          proc     near
                push     di

; Try + F E' here

                cmp      byte ptr es:[di], '+'
                jne      TryMinus
                inc      di
                call     F
                jnc      EP_Failed
                call     EPrime
                jnc      EP_Failed
Success:        add      sp, 2
                stc
                ret

; Try  - F E' here.

TryMinus:       cmp      byte ptr es:[di], '-'
                jne      Success
                inc      di
                call     F
                jnc      EP_Failed
                call     EPrime
                jnc      EP_Failed
                add      sp, 2
                stc
                ret

; If none of the above succeed, return success anyway because we have
; a production of the form E' -> ε.

EP_Failed:      pop      di
                stc
                ret
EPrime          endp
```

```
; F -> TF'

F               proc    near
                push    di
                call    T
                jnc     F_Failed
                call    FPrime
                jnc     F_Failed
                add     sp, 2           ;Success, don't restore di.
                stc
                ret

F_Failed:       pop     di
                clc
                ret
F               endp



; F -> * T F' | / T F' | ε

FPrime          proc    near
                push    di
                cmp     byte ptr es:[di], '*'           ;Start with "*"?
                jne     TryDiv
                inc     di                              ;Skip the "*".
                call    T
                jnc     FP_Failed
                call    FPrime
                jnc     FP_Failed
Success:        add     sp, 2
                stc
                ret

; Try F -> / T F' here

TryDiv:         cmp     byte ptr es:[di], '/'           ;Start with "/"?
                jne     Success                         ;Succeed anyway.
                inc     di                              ;Skip the "/".
                call    T
                jnc     FP_Failed
                call    FPrime
                jnc     FP_Failed
                add     sp, 2
                stc
                ret

; If the above both fail, return success anyway because we've got
; a production of the form F -> ε

FP_Failed:      pop     di
                stc
                ret
FPrime          endp


; T -> G | (E)

T               proc    near

; Try T -> G here.

                call    G
                jnc     TryParens
                ret

; Try T -> (E) here.
```

```
TryParens:      push    di                          ;Preserve if we fail.
                cmp     byte ptr es:[di], '('       ;Start with "("?
                jne     T_Failed                    ;Fail if no.
                inc     di                          ;Skip "(" char.
                call    E
                jnc     T_Failed
                cmp     byte ptr es:[di], ')'       ;End with ")"?
                jne     T_Failed                    ;Fail if no.
                inc     di                          ;Skip ")"
                add     sp, 2                       ;Don't restore di,
                stc                                 ; we've succeeded.
                ret

T_Failed:       pop     di
                clc
                ret
T               endp


; The following is a free-form translation of
;
; G -> H | H G
; H -> 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
;
; This routine checks to see if there is at least one digit. It fails if there
; isn't at least one digit; it succeeds and skips over all digits if there are
; one or more digits.

G               proc    near
                cmp     byte ptr es:[di], '0'       ;Check for at least
                jb      G_Failed                    ; one digit.
                cmp     byte ptr es:[di], '9'
                ja      G_Failed

DigitLoop:      inc     di                          ;Skip any remaining
                cmp     byte ptr es:[di], '0'       ; digits found.
                jb      G_Succeeds
                cmp     byte ptr es:[di], '9'
                jbe     DigitLoop
G_Succeeds:     stc
                ret

G_Failed:       clc                                 ;Fail if no digits
                ret                                 ; at all.
G               endp


; This main program tests the matching functions above and demonstrates
; how to call the matching functions.

Main            proc
                mov     ax, seg dseg ;Set up the segment registers
                mov     ds, ax
                mov     es, ax

                printf
                byte    "Enter an arithmetic expression: ",0
                lesi    InputLine
                gets
                call    E
                jnc     BadExp

; Good so far, but are we at the end of the string?

                cmp     byte ptr es:[di], 0
                jne     BadExp

; Okay, it truly is a good expression at this point.

                printf
```

```
                byte     "'%s' is a valid expression",cr,lf,0
                dword    InputLine
                jmp      Quit

BadExp:         printf
                byte     "'%s' is an invalid arithmetic expression",cr,lf,0
                dword    InputLine

Quit:           ExitPgm
Main            endp

cseg            ends

sseg            segment  para stack 'stack'
stk             byte     1024 dup ("stack ")
sseg            ends

zzzzzzseg       segment  para public 'zzzzzz'
LastBytes       byte     16 dup (?)
zzzzzzseg       ends
                end      Main
```

16.1.7 Some Final Comments on CFGs

The techniques presented in this chapter for converting CFGs to assembly code do not work for all CFGs. They only work for a (large) subset of the CFGs known as LL(1) grammars. The code that these techniques produce is a *recursive descent predictive parser*[6]. Although the set of context free languages recognizable by an LL(1) grammar is a subset of the context free languages, it is a very large subset and you shouldn't run into too many difficulties using this technique.

One important feature of predictive parsers is that they do not require any backtracking. If you are willing to live with the inefficiencies associated with backtracking, it is easy to extended a recursive descent parser to handle any CFG. Note that when you use backtracking, the *predictive* adjective goes away, you wind up with a nondeterministic system rather than a deterministic system (predictive and deterministic are very close in meaning in this case).

There are other CFG systems as well as LL(1). The so-called operator precedence and LR(k) CFGs are two examples. For more information about parsing and grammars, consult a good text on formal language theory or compiler construction (see the bibliography).

16.1.8 Beyond Context Free Languages

Although most patterns you will probably want to process will be regular or context free, there may be times when you need to recognize certain types of patterns that are beyond these two (e.g., *context sensitive* languages). As it turns out, the finite state automata are the simplest machines; the pushdown automata (that recognize context free languages) are the next step up. After pushdown automata, the next step up in power is the *Turing machine*. However, Turing machines are equivalent in power to the 80x86[7], so matching patterns recognized by Turing machines is no different than writing a normal program.

The key to writing functions that recognize patterns that are not context free is to maintain information in variables and use the variables to decide which of several productions you want to use at any one given time. This technique introduces *context sensitiv-*

6. A *parser* is a function that determines whether a pattern belongs to a language.
7. Actually, they are more powerful, in theory, because they have an infinite amount of memory available.

ity. Such techniques are very useful in artificial intelligence programs (like natural language processing) where ambiguity resolution depends on past knowledge or the current context of a pattern matching operation. However, the uses for such types of pattern matching quickly go beyond the scope of a text on assembly language programming, so we will let some other text continue this discussion.

16.2 The UCR Standard Library Pattern Matching Routines

The UCR Standard Library provides a very sophisticated set of pattern matching routines. They are patterned after the pattern matching facilities of SNOBOL4, support CFGs, and provide fully automatic backtracking, as necessary. Furthermore, by writing only *five* assembly language statements, you can match simple or complex patterns.

There is very little assembly language code to worry about when using the Standard Library's pattern matching routines because most of the work occurs in the data segment. To use the pattern matching routines, you first construct a pattern data structure in the data segment. You then pass the address of this pattern and the string you wish to test to the Standard Library match routine. The match routine returns failure or success depending on the state of the comparison. This isn't quite as easy as it sounds, though; learning how to construct the pattern data structure is almost like learning a new programming language. Fortunately, if you've followed the discussion on context free languages, learning this new "language" is a breeze.

The Standard Library *pattern* data structure takes the following form:

```
Pattern          struct
MatchFunction    dword     ?
MatchParm        dword     ?
MatchAlt         dword     ?
NextPattern      dword     ?
EndPattern       word      ?
StartPattern     word      ?
StrSeg           word      ?
Pattern          ends
```

The MatchFunction field contains the address of a routine to call to perform some sort of comparison. The success or failure of this function determines whether the pattern matches the input string. For example, the UCR Standard Library provides a MatchStr function that compares the next *n* characters of the input string against some other character string.

The MatchParm field contains the address or value of a parameter (if appropriate) for the MatchFunction routine. For example, if the MatchFunction routine is MatchStr, then the MatchParm field contains the address of the string to compare the input characters against. Likewise, the MatchChar routine compares the next input character in the string against the L.O. byte of the MatchParm field. Some matching functions do not require any parameters, they will ignore any value you assign to MatchParm field. By convention, most programmers store a zero in unused fields of the Pattern structure.

The MatchAlt field contains either zero (NULL) or the address of some other pattern data structure. If the current pattern matches the input characters, the pattern matching routines ignore this field. However, if the current pattern fails to match the input string, then the pattern matching routines will attempt to match the pattern whose address appears in this field. If this alternate pattern returns success, then the pattern matching routine returns success to the caller, otherwise it returns failure. If the MatchAlt field contains NULL, then the pattern matching routine immediately fails if the main pattern does not match.

The Pattern data structure only matches one item. For example, it might match a single character, a single string, or a character from a set of characters. A real world pattern will probably contain several small patterns concatenated together, e.g., the pattern for a Pascal identifier consists of a single character from the set of alphabetic characters followed by one or more characters from the set [a-zA-Z0-9_]. The NextPattern field lets you create a composite pattern as the concatenation of two individual patterns. For such a composite pattern to return success, the current pattern must match and then the pattern specified by the NextPattern field must also match. Note that you can chain as many patterns together as you please using this field.

The last three fields, EndPattern, StartPattern, and StrSeg are for the internal use of the pattern matching routine. You should not modify or examine these fields.

Once you create a pattern, it is very easy to test a string to see if it matches that pattern. The calling sequence for the UCR Standard Library match routine is

```
lesi     « Input string to match »
ldxi     « Pattern to match string against »
mov      cx, 0
match
jc       Success
```

The Standard Library match routine expects a pointer to the input string in the es:di registers; it expects a pointer to the pattern you want to match in the dx:si register pair. The cx register should contain the length of the string you want to test. If cx contains zero, the match routine will test the entire input string. If cx contains a nonzero value, the match routine will only test the first cx characters in the string. Note that the end of the string (the zero terminating byte) must not appear in the string before the position specified in cx. For most applications, loading cx with zero before calling match is the most appropriate operation.

On return from the match routine, the carry flag denotes success or failure. If the carry flag is set, the pattern matches the string; if the carry flag is clear, the pattern does not match the string. Unlike the examples given in earlier sections, the match routine does not modify the di register, even if the match succeeds. Instead, it returns the failure/success position in the ax register. The is the position of the first character after the match if match succeeds, it is the position of the first unmatched character if match fails.

16.3 The Standard Library Pattern Matching Functions

The UCR Standard Library provides about 20 built-in pattern matching functions. These functions are based on the pattern matching facilities provided by the SNOBOL4 programming language, so they are very powerful indeed! You will probably discover that these routines solve all your pattern matching need, although it is easy to write your own pattern matching routines (see "Designing Your Own Pattern Matching Routines" on page 922) if an appropriate one is not available. The following subsections describe each of these pattern matching routines in detail.

There are two things you should note if you're using the Standard Library's SHELL.ASM file when creating programs that use pattern matching and character sets. First, there is a line at the very beginning of the SHELL.ASM file that contains the statement "matchfuncs". This line is currently a comment because it contains a semicolon in column one. If you are going to be using the pattern matching facilities of the UCR Standard Library, you need to uncomment this line by deleting the semicolon in column one. If you are going to be using the character set facilities of the UCR Standard Library (very common when using the pattern matching facilities), you may want to uncomment the line containing "include stdsets.a" in the data segment. The "stdsets.a" file includes several common character sets, including alphabetics, digits, alphanumerics, whitespace, and so on.

16.3.1 Spancset

The spancset routine skips over all characters belonging to a character set. This routine will match zero or more characters in the specified set and, therefore, *always* succeeds.

The MatchParm field of the pattern data structure must point at a UCR Standard Library character set variable (see "The Character Set Routines in the UCR Standard Library" on page 856).

Example:

```
SkipAlphas        pattern  {spancset, alpha}
                    .
                    .
                    .
                  lesi     StringWAlphas
                  ldxi     SkipAlphas
                  xor      cx, cx
                  match
```

16.3.2 Brkcset

Brkcset is the *dual* to spancset – it matches zero or more characters in the input string which are *not* members of a specified character set. Another way of viewing brkcset is that it will match all characters in the input string *up to* a character in the specified character set (or to the end of the string). The matchparm field contains the address of the character set to match.

Example:

```
DoDigits          pattern  {brkcset, digits, 0, DoDigits2}
DoDigits2         pattern  {spancset, digits}
                    .
                    .
                    .
                  lesi     StringWDigits
                  ldxi     DoDigits
                  xor      cx, cx
                  match
                  jnc      NoDigits
```

The code above matches any string that contains a string of one or more digits somewhere in the string.

16.3.3 Anycset

Anycset matches a single character in the input string from a set of characters. The matchparm field contains the address of a character set variable. If the next character in the input string is a member of this set, anycset set accepts the string and skips over than character. If the next input character is not a member of that set, anycset returns failure.

Example:

```
DoID              pattern  {anycset, alpha, 0, DoID2}
DoID2             pattern  {spancset, alphanum}
                    .
                    .
                    .
                  lesi     StringWID
                  ldxi     DoID
                  xor      cx, cx
                  match
                  jnc      NoID
```

This code segment checks the string StringWID to see if it begins with an identifier specified by the regular expression [a-zA-Z][a-zA-Z0-9]*. The first subpattern with anycset makes sure there is an alphabetic character at the beginning of the string (alpha is the stdsets.a set variable that has all the alphabetic characters as members). If the string does not begin with an alphabetic, the DoID pattern fails. The second subpattern, DoID2, skips over any following alphanumeric characters using the spancset matching function. Note that spancset always succeeds.

The above code does *not* simply match a string that is an identifier; it matches strings that *begin* with a valid identifier. For example, it would match "ThisIsAnID" as well as "ThisIsAnID+SoIsThis - 5". If you only want to match a single identifier and nothing else, you must explicitly check for the end of string in your pattern. For more details on how to do this, see "EOS" on page 919.

16.3.4 Notanycset

Notanycset provides the complement to anycset – it matches a single character in the input string that is *not* a member of a character set. The matchparm field, as usual, contains the address of the character set whose members must not appear as the next character in the input string. If notanycset successfully matches a character (that is, the next input character is not in the designated character set), the function skips the character and returns success; otherwise it returns failure.

Example:

```
DoSpecial         pattern   {notanycset, digits, 0, DoSpecial2}
DoSpecial2        pattern   {spancset, alphanum}
                    .
                    .
                    .
                  lesi      StringWSpecial
                  ldxi      DoSpecial
                  xor       cx, cx
                  match
                  jnc       NoSpecial
```

This code is similar to the DoID pattern in the previous example. It matches a string containing any character except a digit and then matches a string of alphanumeric characters.

16.3.5 MatchStr

Matchstr compares the next set of input characters against a character string. The matchparm field contains the address of a zero terminated string to compare against. If matchstr succeeds, it returns the carry set and skips over the characters it matched; if it fails, it tries the alternate matching function or returns failure if there is no alternate.

Example:

```
DoString          pattern   {matchstr, MyStr}
MyStr             byte      "Match this!",0
                    .
                    .
                    .
                  lesi      String
                  ldxi      DoString
                  xor       cx, cx
                  match
                  jnc       NotMatchThis
```

This sample code matches any string that begins with the characters "Match This!"

16.3.6 MatchiStr

Matchistr is like matchstr insofar as it compares the next several characters against a zero terminated string value. However, matchistr does a *case insensitive* comparison. During the comparison it converts the characters in the input string to upper case before comparing them to the characters that the matchparm field points at. Therefore, *the string pointed at by the matchparm field must contain uppercase wherever alphabetics appear.* If the matchparm string contains any lower case characters, the matchistr function will always fail.

Example:

```
DoString        pattern   {matchistr, MyStr}
MyStr           byte      "MATCH THIS!",0
                  .
                  .
                  .
                lesi      String
                ldxi      DoString
                xor       cx, cx
                match
                jnc       NotMatchThis
```

This example is identical to the one in the previous section except it will match the characters "match this!" using any combination of upper and lower case characters.

16.3.7 MatchToStr

Matchtostr matches all characters in an input string up to and including the characters specified by the matchparm parameter. This routine succeeds if the specified string appears somewhere in the input string, it fails if the string does not appear in the input string. This pattern function is quite useful for locating a substring and ignoring everything that came before the substring.

Example:

```
DoString        pattern   {matchtostr, MyStr}
MyStr           byte      "Match this!",0
                  .
                  .
                  .
                lesi      String
                ldxi      DoString
                xor       cx, cx
                match
                jnc       NotMatchThis
```

Like the previous two examples, this code segment matches the string "Match this!" However, it does not require that the input string (String) begin with "Match this!" Instead, it only requires that "Match this!" appear somewhere in the string.

16.3.8 MatchChar

The matchchar function matches a single character. The matchparm field's L.O. byte contains the character you want to match. If the next character in the input string is that character, then this function succeeds, otherwise it fails.

Example:

```
DoSpace         pattern   {matchchar, ' '}
                  .
                  .
                  .
                lesi      String
                ldxi      DoSpace
                xor       cx, cx
                match
                jnc       NoSpace
```

This code segment matches any string that begins with a space. Keep in mind that the match routine only checks the prefix of a string. If you wanted to see if the string contained only a space (rather than a string that begins with a space), you would need to explicitly check for an end of string after the space. Of course, it would be far more efficient to use strcmp (see "Strcmp, Strcmpl, Stricmp, Stricmpl" on page 848) rather than match for this purpose!

Note that unlike matchstr, you encode the character you want to match directly into the matchparm field. This lets you specify the character you want to test directly in the pattern definition.

16.3.9 MatchToChar

Like matchtostr, matchtochar matches all characters up to and including a character you specify. This is similar to brkcset except you don't have to create a character set containing a single member and brkcset skips up to *but not including* the specified character(s). Matchtochar fails if it cannot find the specified character in the input string.

Example:

```
DoToSpace         pattern  {matchtochar, ' '}
                    .
                    .
                    .
                  lesi     String
                  ldxi     DoSpace
                  xor      cx, cx
                  match
                  jnc      NoSpace
```

This call to match will fail if there are no spaces left in the input string. If there are, the call to matchtochar will skip over all characters up to, and including, the first space. This is a useful pattern for skipping over words in a string.

16.3.10 MatchChars

Matchchars skips zero or more occurrences of a singe character in an input string. It is similar to spancset except you can specify a single character rather than an entire character set with a single member. Like matchchar, matchchars expects a single character in the L.O. byte of the matchparm field. Since this routine matches zero or more occurrences of that character, it always succeeds.

Example:

```
Skip2NextWord     pattern  {matchtochar, ' ', 0, SkipSpcs}
SkipSpcs          pattern  {matchchars, ' '}
                    .
                    .
                    .
                  lesi     String
                  ldxi     Skip2NextWord
                  xor      cx, cx
                  match
                  jnc      NoWord
```

The code segment skips to the beginning of the next word in a string. It fails if there are no additional words in the string (i.e., the string contains no spaces).

16.3.11 MatchToPat

Matchtopat matches all characters in a string up to and including the substring matched by some other pattern. This is one of the two facilities the UCR Standard Library pattern matching routines provide to allow the implementation of nonterminal function calls (also see "SL_Match2" on page 922). This matching function succeeds if it finds a string matching the specified pattern somewhere on the line. If it succeeds, it skips the characters through the last character matched by the pattern parameter. As you would expect, the matchparm field contains the address of the pattern to match.

Example:

```
; Assume there is a pattern "expression" that matches arithmetic
; expressions. The following pattern determines if there is such an
; expression on the line followed by a semicolon.

FindExp         pattern  {matchtopat, expression, 0, MatchSemi}
MatchSemi       pattern  {matchchar, ';'}
                  .
                  .
                  .
                lesi     String
                ldxi     FindExp
                xor      cx, cx
                match
                jnc      NoExp
```

16.3.12 EOS

The EOS pattern matches the end of a string. This pattern, which must obviously appear at the end of a pattern list if it appears at all, checks for the zero terminating byte. Since the Standard Library routines only match prefixes, you should stick this pattern at the end of a list if you want to ensure that a pattern exactly matches a string with no left over characters at the end. EOS succeeds if it matches the zero terminating byte, it fails otherwise.

Example:

```
SkipNumber      pattern  {anycset, digits, 0, SkipDigits}
SkipDigits      pattern  {spancset, digits, 0, EOSPat}
EOSPat          pattern  {EOS}
                  .
                  .
                  .
                lesi     String
                ldxi     SkipNumber
                xor      cx, cx
                match
                jnc      NoNumber
```

The SkipNumber pattern matches strings that contain only decimal digits (from the start of the match to the end of the string). Note that EOS requires no parameters, not even a matchparm parameter.

16.3.13 ARB

ARB matches any number of arbitrary characters. This pattern matching function is equivalent to Σ^*. Note that ARB is a very inefficient routine to use. It works by assuming it can match all remaining characters in the string and then tries to match the pattern specified by the nextpattern field[8]. If the nextpattern item fails, ARB backs up one character and tries matching nextpattern again. This continues until the pattern specified by nextpattern succeeds or ARB backs up to its initial starting position. ARB succeeds if the pattern specified by nextpattern succeeds, it fails if it backs up to its initial starting position.

Given the enormous amount of backtracking that can occur with ARB (especially on long strings), you should try to avoid using this pattern if at all possible. The matchtostr, matchtochar, and matchtopat functions accomplish much of what ARB accomplishes, but they work forward rather than backward in the source string and may be more efficient. ARB is useful mainly if you're sure the following pattern appears late in the string you're matching or if the string you want to match occurs several times and you want to match the *last* occurrence (matchtostr, matchtochar, and matchtopat always match the first occurrence they find).

8. Since the match routine only matches prefixes, it does not make sense to apply ARB to the end of a pattern list, the same pattern would match with or without the final ARB. Therefore, ARB usually has a nextpattern field.

Example:

```
SkipNumber       pattern  {ARB,0,0,SkipDigit}
SkipDigit        pattern  {anycset, digits, 0, SkipDigits}
SkipDigits       pattern  {spancset, digits}
                   .
                   .
                   .
                 lesi     String
                 ldxi     SkipNumber
                 xor      cx, cx
                 match
                 jnc      NoNumber
```

This code example matches the *last* number that appears on an input line. Note that ARB does not use the matchparm field, so you should set it to zero by default.

16.3.14 ARBNUM

ARBNUM matches an arbitrary number (zero or more) of patterns that occur in the input string. If R represents some nonterminal number (pattern matching function), then ARBNUM(R) is equivalent to the production $ARBNUM \rightarrow R\ ARBNUM \mid \varepsilon$.

The matchparm field contains the address of the pattern that ARBNUM attempts to match.

Example:

```
SkipNumbers      pattern  {ARBNUM, SkipNumber}
SkipNumber       pattern  {anycset, digits, 0, SkipDigits}
SkipDigits       pattern  {spancset, digits, 0, EndDigits}
EndDigits        pattern  {matchchars, ‘ ‘, EndString}
EndString        pattern  {EOS}
                   .
                   .
                   .
                 lesi     String
                 ldxi     SkipNumbers
                 xor      cx, cx
                 match
                 jnc      IllegalNumbers
```

This code accepts the input string if it consists of a sequence of zero or more numbers separated by spaces and terminated with the EOS pattern. Note the use of the matchalt field in the EndDigits pattern to select EOS rather than a space for the last number in the string.

16.3.15 Skip

Skip matches n arbitrary characters in the input string. The matchparm field is an integer value containing the number of characters to skip. Although the matchparm field is a double word, this routine limits the number of characters you can skip to 16 bits (65,535 characters); that is, n is the L.O. word of the matchparm field. This should prove sufficient for most needs.

Skip succeeds if there are at least n characters left in the input string; it fails if there are fewer than n characters left in the input string.

Example:

```
Skip1st6         pattern  {skip, 6, 0, SkipNumber}
SkipNumber       pattern  {anycset, digits, 0, SkipDigits}
SkipDigits       pattern  {spancset, digits, 0, EndDigits}
EndDigits        pattern  {EOS}
                   .
                   .
                   .
                 lesi     String
                 ldxi     Skip1st6
                 xor      cx, cx
```

```
                   match
                   jnc      IllegalItem
```

This example matches a string containing six arbitrary characters followed by one or more decimal digits and a zero terminating byte.

16.3.16 Pos

Pos succeeds if the matching functions are currently at the nth character in the string, where *n* is the value in the L.O. word of the matchparm field. Pos fails if the matching functions are not currently at position *n* in the string. Unlike the pattern matching functions you've seen so far, pos does not consume any input characters. Note that the string starts out at position zero. So when you use the pos function, it succeeds if you've matched *n* characters at that point.

Example:

```
SkipNumber         pattern  {anycset, digits, 0, SkipDigits}
SkipDigits         pattern  {spancset, digits, 0, EndDigits}
EndDigits          pattern  {pos, 4}
                     .
                     .
                     .
                   lesi     String
                   ldxi     SkipNumber
                   xor      cx, cx
                   match
                   jnc      IllegalItem
```

This code matches a string that begins with exactly 4 decimal digits.

16.3.17 RPos

Rpos works quite a bit like the pos function except it succeeds if the current position is *n* character positions from the *end* of the string. Like pos, *n* is the L.O. 16 bits of the matchparm field. Also like pos, rpos does not consume any input characters.

Example:

```
SkipNumber         pattern  {anycset, digits, 0, SkipDigits}
SkipDigits         pattern  {spancset, digits, 0, EndDigits}
EndDigits          pattern  {rpos, 4}
                     .
                     .
                     .
                   lesi     String
                   ldxi     SkipNumber
                   xor      cx, cx
                   match
                   jnc      IllegalItem
```

This code matches any string that is all decimal digits except for the last four characters of the string. The string must be at least five characters long for the above pattern match to succeed.

16.3.18 GotoPos

Gotopos skips over any characters in the string until it reaches character position *n* in the string. This function fails if the pattern is already beyond position *n* in the string. The L.O. word of the matchparm field contains the value for *n*.

Example:

```
SkipNumber         pattern  {gotopos, 10, 0, MatchNmbr}
MatchNmbr          pattern  {anycset, digits, 0, SkipDigits}
```

```
SkipDigits       pattern  {spancset, digits, 0, EndDigits}
EndDigits        pattern  {rpos, 4}
                   .
                   .
                   .
                 lesi     String
                 ldxi     SkipNumber
                 xor      cx, cx
                 match
                 jnc      IllegalItem
```

This example code skips to position 10 in the string and attempts to match a string of digits starting with the 11th character. This pattern succeeds if the there are four characters remaining in the string after processing all the digits.

16.3.19 RGotoPos

Rgotopos works like gotopos except it goes to the position specified from the end of the string. Rgotopos fails if the matching routines are already beyond position *n* from the end of the string. As with gotopos, the L.O. word of the matchparm field contains the value for *n*.

Example:

```
SkipNumber       pattern  {rgotopos, 10, 0, MatchNmbr}
MatchNmbr        pattern  {anycset, digits, 0, SkipDigits}
SkipDigits       pattern  {spancset, digits}
                   .
                   .
                   .
                 lesi     String
                 ldxi     SkipNumber
                 xor      cx, cx
                 match
                 jnc      IllegalItem
```

This example skips to ten characters from the end of the string and then attempts to match one or digits starting at that point. It fails if there aren't at least 11 characters in the string or the last 10 characters don't begin with a string of one or more digits.

16.3.20 SL_Match2

The sl_match2 routine is nothing more than a recursive call to match. The matchparm field contains the address of pattern to match. This is quite useful for simulating parenthesis around a pattern in a pattern expression. As far as matching strings are concerned, pattern1 and pattern2, below, are equivalent:

```
Pattern2         pattern  {sl_match2, Pattern1}
Pattern1         pattern  {matchchar, 'a'}
```

The only difference between invoking a pattern directly and invoking it with sl_match2 is that sl_match2 tweaks some internal variables to keep track of matching positions within the input string. Later, you can extract the character string matched by sl_match2 using the patgrab routine (see "Extracting Substrings from Matched Patterns" on page 925).

16.4 Designing Your Own Pattern Matching Routines

Although the UCR Standard Library provides a wide variety of matching functions, there is no way to anticipate the needs of all applications. Therefore, you will probably discover that the library does not support some particular pattern matching function you need. Fortunately, it is very easy for you to create your own pattern matching functions to augment those available in the UCR Standard Library. When you specify a matching function name in the pattern data structure, the match routine calls the specified address using a far call and passing the following parameters:

es:di- Points at the next character in the input string. You should not look at any characters before this address. Furthermore, you should never look beyond the end of the string (see cx below).

ds:si- Contains the four byte parameter found in the matchparm field.

cx- Contains the last position, plus one, in the input string you're allowed to look at. Note that your pattern matching routine should not look beyond location es:cx or the zero terminating byte; whichever comes first in the input string.

On return from the function, ax must contain the offset into the string (di's value) of the last character matched *plus one*, if your matching function is successful. It must also set the carry flag to denote success. After your pattern matches, the match routine might call another matching function (the one specified by the next pattern field) and that function begins matching at location es:ax.

If the pattern match fails, then you must return the original di value in the ax register and return with the carry flag clear. Note that your matching function must preserve all other registers.

There is one very important detail you must never forget with writing your own pattern matching routines – ds does not point at your data segment, it contains the H.O. word of the matchparm parameter. Therefore, if you are going to access global variables in your data segment you will need to push ds, load it with the address of dseg, and pop ds before leaving. Several examples throughout this chapter demonstrate how to do this.

There are some obvious omissions from (the current version of) the UCR Standard Library's repertoire. For example, there should probably be matchtoistr, matchichar, and matchtoichar pattern functions. The following example code demonstrates how to add a matchtoistr (match up to a string, doing a case insensitive comparison) routine.

```
                .xlist

                include     stdlib.a
                includelib  stdlib.lib
                matchfuncs
                .list

dseg            segment  para public 'data'

TestString      byte     "This is the string 'xyz' in it",cr,lf,0

TestPat         pattern  {matchtoistr,xyz}
xyz             byte     "XYZ",0

dseg            ends


cseg            segment  para public 'code'
                assume   cs:cseg, ds:dseg

; MatchToiStr-  Matches all characters in a string up to, and including, the
;               specified parameter string. The parameter string must be
;               all upper case characters. This guy matches string using
;               a case insensitive comparison.
;
; inputs:
;               es:di-   Source string
;               ds:si-   String to match
;               cx-      Maximum match position
;
; outputs:
;               ax-      Points at first character beyond the end of the
;                        matched string if success, contains the initial DI
;                        value if failure occurs.
;               carry-   0 if failure, 1 if success.
```

```
MatchToiStr     proc    far
                pushf
                push    di
                push    si
                cld

; Check to see if we’re already past the point were we’re allowed
; to scan in the input string.

                cmp     di, cx
                jae     MTiSFailure

; If the pattern string is the empty string, always match.

                cmp     byte ptr ds:[si], 0
                je      MTSsuccess

; The following loop scans through the input string looking for
; the first character in the pattern string.

ScanLoop:       push    si
                lodsb                   ;Get first char of string

                dec     di
FindFirst:      inc     di              ;Move on to next (or 1st) char.
                cmp     di, cx          ;If at cx, then we’ve got to
                jae CantFind1st; fail.

                mov     ah, es:[di]     ;Get input character.
                cmp     ah, ‘a’         ;Convert input character to
                jb      DoCmp           ; upper case if it’s a lower
                cmp     ah, ‘z’         ; case character.
                ja      DoCmp
                and     ah, 5fh
DoCmp:          cmp     al, ah          ;Compare input character against
                jne     FindFirst       ; pattern string.


; At this point, we’ve located the first character in the input string
; that matches the first character of the pattern string. See if the
; strings are equal.

                push    di              ;Save restart point.

CmpLoop:        cmp     di, cx          ;See if we’ve gone beyond the
                jae     StrNotThere;    last position allowable.
                lodsb                   ;Get next input character.
                cmp     al, 0           ;At the end of the parameter
                je      MTSsuccess2;    string? If so, succeed.

                inc     di
                mov     ah, es:[di]     ;Get the next input character.
                cmp     ah, ‘a’         ;Convert input character to
                jb      DoCmp2          ; upper case if it’s a lower
                cmp     ah, ‘z’         ; case character.
                ja      DoCmp2
                and     ah, 5fh
DoCmp2:         cmp     al, ah          ;Compare input character against
                je      CmpLoop
                pop     di
                pop     si
                jmp     ScanLoop

StrNotThere:    add     sp, 2           ;Remove di from stack.
CantFind1st:    add     sp, 2           ;Remove si from stack.
MTiSFailure:    pop     si
                pop     di
                mov     ax, di          ;Return failure position in AX.
                popf
```

```
                clc                     ;Return failure.
                ret

MTSSuccess2:    add     sp, 2           ;Remove DI value from stack.
MTSSuccess:     add     sp, 2           ;Remove SI value from stack.
                mov     ax, di          ;Return next position in AX.
                pop     si
                pop     di
                popf
                stc                     ;Return success.
                ret
MatchToiStr     endp

Main            proc
                mov     ax, dseg
                mov     ds, ax
                mov     es, ax
                meminit

                lesi    TestString
                ldxi    TestPat
                xor     cx, cx
                match
                jnc     NoMatch
                print
                byte    "Matched",cr,lf,0
                jmp     Quit

NoMatch:        print
                byte    "Did not match",cr,lf,0

Quit:           ExitPgm
Main            endp

cseg            ends

sseg            segment para stack 'stack'
stk             db      1024 dup ("stack ")
sseg            ends

zzzzzzseg       segment para public 'zzzzzz'
LastBytes       db      16 dup (?)
zzzzzzseg       ends
                end     Main
```

16.5 Extracting Substrings from Matched Patterns

Often, simply determining that a string matches a given pattern is insufficient. You may want to perform various operations that depend upon the actual information in that string. However, the pattern matching facilities described thus far do not provide a mechanism for testing individual components of the input string. In this section, you will see how to extract portions of a pattern for further processing.

Perhaps an example may help clarify the need to extract portions of a string. Suppose you are writing a stock buy/sell program and you want it to process commands described by the following regular expression:

(buy | sell) $[0-9]^+$ shares of (ibm | apple | hp | dec)

While it is easy to devise a Standard Library pattern that recognizes strings of this form, calling the match routine would only tell you that you have a legal buy or sell command. It does not tell you if you are to buy or sell, *who* to buy or sell, or how many shares to buy or sell. Of course, you could take the cross product of (buy | sell) with (ibm | apple | hp | dec) and generate eight different regular expressions that uniquely determine whether you're buying or selling and whose stock you're trading, but you can't process the integer values this way (unless you willing to have *millions* of regular expressions). A better solution would be to extract substrings from the legal pattern and process these substrings after you verify that you have a legal buy or sell command. For example, you could extract buy or sell into one string, the digits into another, and the company name into a third. After verifying the syntax of the command, you could process the individual strings you've extracted. The UCR Standard Library patgrab routine provides this capability for you.

You normally call patgrab *after* calling match and verifying that it matches the input string. Patgrab expects a single parameter – a pointer to a pattern recently processed by match. Patgrab creates a string on the heap consisting of the characters matched by the given pattern and returns a pointer to this string in es:di. Note that patgrab only returns a string associated with a single pattern data structure, not a chain of pattern data structures. Consider the following pattern:

```
PatToGrab         pattern  {matchstr, str1, 0, Pat2}
Pat2              pattern  {matchstr, str2}
str1              byte     "Hello",0
str2              byte     " there",0
```

Calling match on PatToGrab will match the string "Hello there". However, if after calling match you call patgrab and pass it the address of PatToGrab, patgrab will return a pointer to the string "Hello".

Of course, you might want to collect a string that is the concatenation of several strings matched within your pattern (i.e., a portion of the pattern list). This is where calling the sl_match2 pattern matching function comes in handy. Consider the following pattern:

```
Numbers           pattern  {sl_match2, FirstNumber}
FirstNumber       pattern  {anycset, digits, 0, OtherDigs}
OtherDigs         pattern  {spancset, digits}
```

This pattern matches the same strings as

```
Numbers           pattern  {anycset, digits, 0, OtherDigs}
OtherDigs         pattern  {spancset, digits}
```

So why bother with the extra pattern that calls sl_match2? Well, as it turns out the sl_match2 matching function lets you create *parenthetical patterns*. A parenthetical pattern is a pattern list that the pattern matching routines (especially patgrab) treat as a single pattern. Although the match routine will match the same strings regardless of which version of Numbers you use, patgrab will produce two entirely different strings depending upon your choice of the above patterns. If you use the latter version, patgrab will only return the first digit of the number. If you use the former version (with the call to sl_match2), then patgrab returns the entire string matched by sl_match2, and that turns out to be the entire string of digits.

The following sample program demonstrates how to use parenthetical patterns to extract the pertinent information from the stock command presented earlier. It uses parenthetical patterns for the buy/sell command, the number of shares, and the company name.

```
                  .xlist
                  include    stdlib.a
                  includelib stdlib.lib
                  matchfuncs
                  .list

dseg              segment  para public 'data'

; Variables used to hold the number of shares bought/sold, a pointer to
; a string containing the buy/sell command, and a pointer to a string
; containing the company name.

Count             word     0
CmdPtr            dword    ?
CompPtr           dword    ?
```

```
; Some test strings to try out:

Cmd1            byte     "Buy 25 shares of apple stock",0
Cmd2            byte     "Sell 50 shares of hp stock",0
Cmd3            byte     "Buy 123 shares of dec stock",0
Cmd4            byte     "Sell 15 shares of ibm stock",0
BadCmd0         byte     "This is not a buy/sell command",0

; Patterns for the stock buy/sell command:
;
; StkCmd matches buy or sell and creates a parenthetical pattern
; that contains the string "buy" or "sell".

StkCmd          pattern  {sl_match2, buyPat, 0, skipspcs1}

buyPat          pattern  {matchistr,buystr,sellpat}
buystr          byte     "BUY",0

sellpat         pattern  {matchistr,sellstr}
sellstr         byte     "SELL",0

; Skip zero or more white space characters after the buy command.

skipspcs1       pattern  {spancset, whitespace, 0, CountPat}

; CountPat is a parenthetical pattern that matches one or more
; digits.

CountPat        pattern  {sl_match2, Numbers, 0, skipspcs2}
Numbers         pattern  {anycset, digits, 0, RestOfNum}
RestOfNum       pattern  {spancset, digits}

; The following patterns match " shares of " allowing any amount
; of white space between the words.

skipspcs2       pattern  {spancset, whitespace, 0, sharesPat}

sharesPat       pattern  {matchistr, sharesStr, 0, skipspcs3}
sharesStr       byte     "SHARES",0

skipspcs3       pattern  {spancset, whitespace, 0, ofPat}

ofPat           pattern  {matchistr, ofStr, 0, skipspcs4}
ofStr           byte     "OF",0

skipspcs4       pattern  {spancset, whitespace, 0, CompanyPat}

; The following parenthetical pattern matches a company name.
; The patgrab-available string will contain the corporate name.

CompanyPat      pattern  {sl_match2, ibmpat}

ibmpat          pattern  {matchistr, ibm, applePat}
ibm             byte     "IBM",0

applePat        pattern  {matchistr, apple, hpPat}
apple           byte     "APPLE",0

hpPat           pattern  {matchistr, hp, decPat}
hp              byte     "HP",0

decPat          pattern  {matchistr, decstr}
decstr          byte     "DEC",0

                include  stdsets.a
dseg            ends

cseg            segment  para public 'code'
                assume   cs:cseg, ds:dseg
```

```
; DoBuySell-     This routine processes a stock buy/sell command.
;                After matching the command, it grabs the components
;                of the command and outputs them as appropriate.
;                This routine demonstrates how to use patgrab to
;                extract substrings from a pattern string.
;
;                On entry, es:di must point at the buy/sell command
;                you want to process.

DoBuySell        proc     near
                 ldxi StkCmd
                 xor      cx, cx
                 match
                 jnc      NoMatch

                 lesi     StkCmd
                 patgrab
                 mov      word ptr CmdPtr, di
                 mov      word ptr CmdPtr+2, es

                 lesi     CountPat
                 patgrab
                 atoi                  ;Convert digits to integer
                 mov      Count, ax
                 free                  ;Return storage to heap.

                 lesi     CompanyPat
                 patgrab
                 mov      word ptr CompPtr, di
                 mov      word ptr CompPtr+2, es

                 printf
                 byte     "Stock command: %^s\n"
                 byte     "Number of shares: %d\n"
                 byte     "Company to trade: %^s\n\n",0
                 dword    CmdPtr, Count, CompPtr

                 les      di, CmdPtr
                 free
                 les      di, CompPtr
                 free
                 ret

NoMatch:         print
                 byte     "Illegal buy/sell command",cr,lf,0
                 ret
DoBuySell        endp


Main             proc
                 mov      ax, dseg
                 mov      ds, ax
                 mov      es, ax

                 meminit

                 lesi     Cmd1
                 call     DoBuySell
                 lesi     Cmd2
                 call     DoBuySell
                 lesi     Cmd3
                 call     DoBuySell
                 lesi     Cmd4
                 call     DoBuySell
                 lesi     BadCmd0
                 call     DoBuySell

Quit:            ExitPgm
Main             endp
```

```
cseg            ends

sseg            segment  para stack 'stack'
stk             db       1024 dup ("stack ")
sseg            ends

zzzzzzseg       segment  para public 'zzzzzz'
LastBytes       db       16 dup (?)
zzzzzzseg       ends
                end      Main
```

Sample program output:

```
Stock command: Buy
Number of shares: 25
Company to trade: apple

Stock command: Sell
Number of shares: 50
Company to trade: hp

Stock command: Buy
Number of shares: 123
Company to trade: dec

Stock command: Sell
Number of shares: 15
Company to trade: ibm

Illegal buy/sell command
```

16.6 Semantic Rules and Actions

Automata theory is mainly concerned with whether or not a string matches a given pattern. Like many theoretical sciences, practitioners of automata theory are only concerned if something is possible, the practical applications are not as important. For real programs, however, we would like to perform certain operations if we match a string or perform one from a set of operations depending on *how* we match the string.

A *semantic rule* or *semantic action* is an operation you perform based upon the type of pattern you match. This is, it is the piece of code you execute when you are satisfied with some pattern matching behavior. For example, the call to patgrab in the previous section is an example of a semantic action.

Normally, you execute the code associated with a semantic rule *after* returning from the call to match. Certainly when processing regular expressions, there is no need to process a semantic action in the *middle* of pattern matching operation. However, this isn't the case for a context free grammar. Context free grammars often involve recursion or may use the same pattern several times when matching a single string (that is, you may reference the same nonterminal several times while matching the pattern). The pattern matching data structure only maintains pointers (EndPattern, StartPattern, and StrSeg) to the last substring matched by a given pattern. Therefore, if you reuse a subpattern while matching a string and you need to execute a semantic rule associated with that subpattern, you will need to execute that semantic rule in the middle of the pattern matching operation, before you reference that subpattern again.

It turns out to be very easy to insert semantic rules in the middle of a pattern matching operation. All you need to do is write a pattern matching function that always succeeds (i.e., it returns with the carry flag clear). Within the body of your pattern matching routine you can choose to ignore the string the matching code is testing and perform any other actions you desire.

Your semantic action routine, on return, must set the carry flag and it must copy the original contents of di into ax. It must preserve all other registers. Your semantic action must *not* call the match routine (call sl_match2 instead). Match does not allow recursion (it is not *reentrant*) and calling match within a semantic action routine will mess up the pattern match in progress.

The following example provides several examples of semantic action routines within a program. This program converts arithmetic expressions in infix (algebraic) form to reverse polish notation (RPN) form.

```
; INFIX.ASM
;
; A simple program which demonstrates the pattern matching routines in the
; UCR library. This program accepts an arithmetic expression on the command
; line (no interleaving spaces in the expression is allowed, that is, there
; must be only one command line parameter) and converts it from infix notation
; to postfix (rpn) notation.

                 .xlist
                 include    stdlib.a
                 includelib stdlib.lib
                 matchfuncs
                 .list


dseg             segment  para public 'data'

; Grammar for simple infix -> postfix translation operation
; (the semantic actions are enclosed in braces}:
;
; E -> FE'
; E' -> +F {output '+'} E' | -F {output '-'} E' | <empty string>
; F -> TF'
; F -> *T {output '*'} F' | /T {output '/'} F' | <empty string>
; T -> -T {output 'neg'} | S
; S -> <constant> {output constant} | (E)
;
; UCR Standard Library Pattern which handles the grammar above:

; An expression consists of an "E" item followed by the end of the string:

infix2rpn        pattern  {sl_Match2,E,,EndOfString}
EndOfString      pattern  {EOS}

; An "E" item consists of an "F" item optionally followed by "+" or "-"
; and another "E" item:

E                pattern  {sl_Match2, F,,Eprime}
Eprime           pattern  {MatchChar, '+', Eprime2, epf}
epf              pattern  {sl_Match2, F,,epPlus}
epPlus           pattern  {OutputPlus,,,Eprime}         ;Semantic rule

Eprime2          pattern  {MatchChar, '-', Succeed, emf}
emf              pattern  {sl_Match2, F,,epMinus}
epMinus          pattern  {OutputMinus,,,Eprime}        ;Semantic rule

; An "F" item consists of a "T" item optionally followed by "*" or "/"
; followed by another "T" item:

F                pattern  {sl_Match2, T,,Fprime}
Fprime           pattern  {MatchChar, '*', Fprime2, fmf}
fmf              pattern  {sl_Match2, T, 0, pMul}
pMul             pattern  {OutputMul,,,Fprime}          ;Semantic rule

Fprime2          pattern  {MatchChar, '/', Succeed, fdf}
fdf              pattern  {sl_Match2, T, 0, pDiv}
pDiv             pattern  {OutputDiv, 0, 0,Fprime}      ;Semantic rule
```

```
; T item consists of an "S" item or a "-" followed by another "T" item:

T               pattern  {MatchChar, '-', S, TT}
TT              pattern  {sl_Match2, T, 0,tpn}
tpn             pattern  {OutputNeg}                    ;Semantic rule

; An "S" item is either a string of one or more digits or "(" followed by
; and "E" item followed by ")":

Const           pattern  {sl_Match2, DoDigits, 0, spd}
spd             pattern  {OutputDigits}                 ;Semantic rule

DoDigits        pattern  {Anycset, Digits, 0, SpanDigits}
SpanDigits      pattern  {Spancset, Digits}

S               pattern  {MatchChar, '(', Const, IntE}
IntE            pattern  {sl_Match2, E, 0, CloseParen}
CloseParen      pattern  {MatchChar, ')'}


Succeed         pattern  {DoSucceed}


                include  stdsets.a

dseg            ends



cseg            segment  para public 'code'
                assume   cs:cseg, ds:dseg

; DoSucceed matches the empty string. In other words, it matches anything
; and always returns success without eating any characters from the input
; string.

DoSucceed       proc     far
                mov      ax, di
                stc
                ret
DoSucceed       endp


; OutputPlus is a semantic rule which outputs the "+" operator after the
; parser sees a valid addition operator in the infix string.

OutputPlus      proc     far
                print
                byte     " +",0
                mov      ax, di                  ;Required by sl_Match
                stc
                ret
OutputPlus      endp


; OutputMinus is a semantic rule which outputs the "-" operator after the
; parser sees a valid subtraction operator in the infix string.

OutputMinus     proc     far
                print
                byte     " -",0
                mov      ax, di                  ;Required by sl_Match
                stc
                ret
OutputMinus     endp


; OutputMul is a semantic rule which outputs the "*" operator after the
; parser sees a valid multiplication operator in the infix string.
```

```
OutputMul         proc     far
                  print
                  byte     " *",0
                  mov      ax, di                 ;Required by sl_Match
                  stc
                  ret
OutputMul         endp


; OutputDiv is a semantic rule which outputs the "/" operator after the
; parser sees a valid division operator in the infix string.

OutputDiv         proc     far
                  print
                  byte     " /",0
                  mov      ax, di                 ;Required by sl_Match
                  stc
                  ret
OutputDiv         endp


; OutputNeg is a semantic rule which outputs the unary "-" operator after the
; parser sees a valid negation operator in the infix string.

OutputNeg         proc     far
                  print
                  byte     " neg",0
                  mov      ax, di                 ;Required by sl_Match
                  stc
                  ret
OutputNeg         endp


; OutputDigits outputs the numeric value when it encounters a legal integer
; value in the input string.

OutputDigits      proc     far
                  push     es
                  push     di
                  mov      al, ' '
                  putc
                  lesi     const
                  patgrab
                  puts
                  free
                  stc
                  pop      di
                  mov      ax, di
                  pop      es
                  ret
OutputDigits      endp



; Okay, here's the main program which fetches the command line parameter
; and parses it.

Main              proc
                  mov      ax, dseg
                  mov      ds, ax
                  mov      es, ax

                  meminit                         ; memory to the heap.


                  print
                  byte     "Enter an arithmetic expression: ",0
                  getsm
                  print
                  byte     "Expression in postfix form: ",0
```

```
                 ldxi     infix2rpn
                 xor      cx, cx
                 match
                 jc       Succeeded

                 print
                 byte     "Syntax error",0

Succeeded:       putcr

Quit:            ExitPgm
Main             endp

cseg             ends



; Allocate a reasonable amount of space for the stack (8k).

sseg             segment  para stack 'stack'
stk              db       1024 dup ("stack ")
sseg             ends


; zzzzzzseg must be the last segment that gets loaded into memory!

zzzzzzseg        segment  para public 'zzzzzz'
LastBytes        db       16 dup (?)
zzzzzzseg        ends
                 end      Main
```

16.7 Constructing Patterns for the MATCH Routine

A major issue we have yet to discuss is how to convert regular expressions and context free grammars into patterns suitable for the UCR Standard Library pattern matching routines. Most of the examples appearing up to this point have used an ad hoc translation scheme; now it is time to provide an algorithm to accomplish this.

The following algorithm converts a context free grammar to a UCR Standard Library pattern data structure. If you want to convert a regular expression to a pattern, first convert the regular expression to a context free grammar (see "Converting REs to CFGs" on page 905). Of course, it is easy to convert many regular expression forms directly to a pattern, when such conversions are obvious you can bypass the following algorithm; for example, it should be obvious that you can use spancset to match a regular expression like [0-9]*.

The first step you must always take is to eliminate left recursion from the grammar. You will generate an infinite loop (and crash the machine) if you attempt to code a grammar containing left recursion into a pattern data structure. For information on eliminating left recursion, see "Eliminating Left Recursion and Left Factoring CFGs" on page 903. You might also want to left factor the grammar while you are eliminating left recursion. The Standard Library routines fully support backtracking, so left factoring is not strictly necessary, however, the matching routine will execute faster if it does not need to backtrack.

If a grammar production takes the form $A \rightarrow B\ C$ where A, B, and C are nonterminal symbols, you would create the following pattern:

```
A                pattern  {sl_match2,B,0,C}
```

This pattern description for A checks for an occurrence of a B pattern followed by a C pattern.

If B is a relatively simple production (that is, you can convert it to a single pattern data structure), you can optimize this to:

```
A               pattern  {B's Matching Function, B's parameter, 0, C}
```

The remaining examples will always call sl_match2, just to be consistent. However, as long as the nonterminals you invoke are simple, you can fold them into A''s pattern.

If a grammar production takes the form $A \rightarrow B \mid C$ where A, B, and C are nonterminal symbols, you would create the following pattern:

```
A               pattern  {sl_match2, B, C}
```

This pattern tries to match B. If it succeeds, A succeeds; if it fails, it tries to match C. At this point, A''s success or failure is the success or failure of C.

Handling terminal symbols is the next thing to consider. These are quite easy – all you need to do is use the appropriate matching function provided by the Standard Library, e.g., matchstr or matchchar. For example, if you have a production of the form $A \rightarrow$ abc | y you would convert this to the following pattern:

```
A               pattern  {matchstr,abc,ypat}
abc             byte     "abc",0
ypat            pattern  {matchchar,'y'}
```

The only remaining detail to consider is the empty string. If you have a production of the form $A \rightarrow \varepsilon$ then you need to write a pattern matching function that always succeed. The elegant way to do this is to write a custom pattern matching function. This function is

```
succeed         proc     far
                mov      ax, di            ;Required by sl_match
                stc                        ;Always succeed.
                ret
succeed         endp
```

Another, sneaky, way to force success is to use matchstr and pass it the empty string to match, e.g.,

```
success         pattern  {matchstr, emptystr}
emptystr        byte     0
```

The empty string always matches the input string, no matter what the input string contains.

If you have a production with several alternatives and ε is one of them, you must process ε last. For example, if you have the productions $A \rightarrow$ abc | y | BC | ε you would use the following pattern:

```
A               pattern  {matchstr,abc, tryY}
abc             byte     "abc",0
tryY            pattern  {matchchar, 'y', tryBC}
tryBC           pattern  {sl_match2, B, DoSuccess, C}
DoSuccess       pattern  {succeed}
```

While the technique described above will let you convert *any* CFG to a pattern that the Standard Library can process, it certainly does not take advantage of the Standard Library facilities, nor will it produce particularly efficient patterns. For example, consider the production:

$Digits \rightarrow$ 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Converting this to a pattern using the techniques described above will yield the pattern:

```
Digits          pattern  {matchchar, '0', try1}
try1            pattern  {matchchar, '1', try2}
try2            pattern  {matchchar, '2', try3}
try3            pattern  {matchchar, '3', try4}
try4            pattern  {matchchar, '4', try5}
try5            pattern  {matchchar, '5', try6}
try6            pattern  {matchchar, '6', try7}
```

```
try7            pattern     {matchchar, '7', try8}
try8            pattern     {matchchar, '8', try9}
try9            pattern     {matchchar, '9'}
```

Obviously this isn't a very good solution because we can match this same pattern with the single statement:

```
Digits          pattern  {anycset, digits}
```

If your pattern is easy to specify using a regular expression, you should try to encode it using the built-in pattern matching functions and fall back on the above algorithm once you've handled the low level patterns as best you can. With experience, you will be able to choose an appropriate balance between the algorithm in this section and ad hoc methods you develop on your own.

16.8 Some Sample Pattern Matching Applications

The best way to learn how to convert a pattern matching problem to the respective pattern matching algorithms is by example. The following sections provide several examples of some small pattern matching problems and their solutions.

16.8.1 Converting Written Numbers to Integers

One interesting pattern matching problem is to convert written (English) numbers to their integer equivalents. For example, take the string "one hundred ninety-two" and convert it to the integer 192. Although written numbers represent a pattern quite a bit more complex than the ones we've seen thus far, a little study will show that it is easy to decompose such strings.

The first thing we will need to do is enumerate the English words we will need to process written numbers. This includes the following words:

> zero, one, two, three, four, five, six, seven, eight, nine, ten, eleven twelve, thirteen, fourteen, fifteen, sixteen, seventeen, eighteen, nineteen, twenty, thirty, forty, fifty sixty, seventy, eighty, ninety, hundred, *and* thousand.

With this set of words we can build all the values between zero and 65,535 (the values we can represent in a 16 bit integer.

Next, we've got to decide how to put these words together to form all the values between zero and 65,535. The first thing to note is that zero only occurs by itself, it is never part of another number. So our first production takes the form:

Number → `zero` | *NonZero*

The next thing to note is that certain values *may* occur in pairs, denoting addition. For example, eighty-five denotes the sum of eighty plus five. Also note that certain other pairs denote multiplication. If you have a statement like "two hundred" or "fifteen hundred" the "hundred" word says *multiply the preceding value by 100*. The multiplicative words, "hundred" and "thousand" , are also additive. Any value following these terms is added in to the total[9]; e.g., "one hundred five" means 1*100+5. By combining the appropriate rules, we obtain the following grammar

NonZero →	*Thousands Maybe100s* | *Hundreds*
Thousands →	*Under100* `thousand`
Maybe100s →	*Hundreds* | ε
Hundreds →	*Under100* `hundred` *After100* | *Under100*
After100 →	*Under100* | ε

9. We will ignore special multiplicative forms like "one thousand thousand" (one million) because these forms are all too large to fit into 16 bits. .

```
Under100 →     Tens Maybe1s| Teens | ones
Maybe1s  →     Ones | ε
ones →   one | two | three | four | five | six | seven | eight | nine
teens →  ten | eleven | twelve | thirteen | fourteen | fifteen | sixteen |
         seventeen | eighteen | nineteen
tens →   twenty | thirty | forty | fifty | sixty | seventy | eighty | ninety
```

The final step is to add semantic actions to actually convert the strings matched by this grammar to integer values. The basic idea is to initialize an accumulator value to zero. Whenever you encounter one of the strings that *ones, teens,* or *tens* matches, you add the corresponding value to the accumulator. If you encounter the hundred or thousand strings, you multiply the accumulator by the appropriate factor. The complete program to do the conversion follows:

```
; Numbers.asm
;
; This program converts written English numbers in the range “zero”
; to “sixty five thousand five hundred thirty five” to the corresponding
; integer value.

                 .xlist
                 include    stdlib.a
                 includelib stdlib.lib
                 matchfuncs
                 .list


dseg             segment  para public ‘data’

Value            word     0                       ;Store results here.
HundredsVal      word     0
ThousandsVal     word     0


Str0             byte     “twenty one”,0
Str1             byte     “nineteen hundred thirty-five”,0
Str2             byte     “thirty three thousand two hundred nineteen”,0
Str3             byte     “three”,0
Str4             byte     “fourteen”,0
Str5             byte     “fifty two”,0
Str6             byte     “seven hundred”,0
Str7             byte     “two thousand seven”,0
Str8             byte     “four thousand ninety six”,0
Str9             byte     “five hundred twelve”,0
Str10            byte     “twenty three thousand two hundred ninety-five”,0
Str11            byte     “seventy-five hundred”,0
Str12            byte     “sixty-five thousand”,0
Str13            byte     “one thousand”,0


; The following grammar is what we use to process the numbers.
; Semantic actions appear in the braces.
;
; Note: begin by initializing Value, HundredsVal, and ThousandsVal to zero.
;
; N              -> separators zero
;                | N4
;
; N4             -> do1000s maybe100s
;                | do100s
;
; Maybe100s      -> do100s
;                | <empty string>
;
; do1000s        -> Under100 “THOUSAND” separators
;                            {ThousandsVal := Value*1000}
;
; do100s         -> Under100 “HUNDRED”
```

```
;                              {HundredsVal := Value*100} After100
;                 | Under100
;
; After100        -> {Value := 0} Under100
;                 | {Value := 0} <empty string>
;
; Under100        -> {Value := 0} try20 try1s
;                 | {Value := 0} doTeens
;                 | {Value := 0} do1s
;
; try1s           -> do1s | <empty string>
;
; try20           -> "TWENTY" {Value := Value + 20}
;                 | "THIRTY" {Value := Value + 30}
;                 | ...
;                 | "NINETY" {Value := Value + 90}
;
; doTeens         -> "TEN"  {Value := Value + 10}
;                 | "ELEVEN" {Value := Value + 11}
;                 | ...
;                 | "NINETEEN" {Value := Value + 19}
;
; do1s            -> "ONE"  {Value := Value + 1}
;                 | "TWO"   {Value := Value + 2}
;                 | ...
;                 | "NINE"  {Value := Value + 9}


separators        pattern   {anycset, delimiters, 0, delim2}
delim2            pattern   {spancset, delimiters}
doSuccess         pattern   {succeed}
AtLast            pattern   {sl_match2, separators, AtEOS, AtEOS}
AtEOS             pattern   {EOS}


N                 pattern   {sl_match2, separators, N2, N2}
N2                pattern   {matchistr, zero, N3, AtLast}
zero              byte      "ZERO",0

N3                pattern   {sl_match2, N4, 0, AtLast}
N4                pattern   {sl_match2, do1000s, do100s, Maybe100s}
Maybe100s         pattern   {sl_match2, do100s, AtLast, AtLast}

do1000s           pattern   {sl_match2, Under100, 0, do1000s2}
do1000s2          pattern   {matchistr, str1000, 0, do1000s3}
do1000s3          pattern   {sl_match2, separators, do1000s4, do1000s5}
do1000s4          pattern   {EOS, 0, 0, do1000s5}
do1000s5          pattern   {Get1000s}
str1000           byte      "THOUSAND",0

do100s            pattern   {sl_match2, do100s1, Under100, After100}
do100s1           pattern   {sl_match2, Under100, 0, do100s2}
do100s2           pattern   {matchistr, str100, 0, do100s3}
do100s3           pattern   {sl_match2, separators, do100s4, do100s5}
do100s4           pattern   {EOS, 0, 0, do100s5}
do100s5           pattern   {Get100s}
str100            byte      "HUNDRED",0

After100          pattern   {SetVal, 0, 0, After100a}
After100a         pattern   {sl_match2, Under100, doSuccess}

Under100          pattern   {SetVal, 0, 0, Under100a}
Under100a         pattern   {sl_match2, try20, Under100b, Do1orE}
Under100b         pattern   {sl_match2, doTeens, do1s}

Do1orE            pattern   {sl_match2, do1s, doSuccess, 0}



NumPat            macro     lbl, next, Constant, string
```

```
                local    try, SkipSpcs, val, str, tryEOS
lbl             pattern  {sl_match2, try, next}
try             pattern  {matchistr, str, 0, SkipSpcs}
SkipSpcs        pattern  {sl_match2, separators, tryEOS, val}
tryEOS          pattern  {EOS, 0, 0, val}
val             pattern  {AddVal, Constant}
str             byte     string
                byte     0
                endm

                NumPat   doTeens, try11, 10, "TEN"
                NumPat   try11, try12, 11, "ELEVEN"
                NumPat   try12, try13, 12, "TWELVE"
                NumPat   try13, try14, 13, "THIRTEEN"
                NumPat   try14, try15, 14, "FOURTEEN"
                NumPat   try15, try16, 15, "FIFTEEN"
                NumPat   try16, try17, 16, "SIXTEEN"
                NumPat   try17, try18, 17, "SEVENTEEN"
                NumPat   try18, try19, 18, "EIGHTEEN"
                NumPat   try19, 0, 19, "NINETEEN"

                NumPat   do1s, try2, 1, "ONE"
                NumPat   try2, try3, 2, "TWO"
                NumPat   try3, try4, 3, "THREE"
                NumPat   try4, try5, 4, "FOUR"
                NumPat   try5, try6, 5, "FIVE"
                NumPat   try6, try7, 6, "SIX"
                NumPat   try7, try8, 7, "SEVEN"
                NumPat   try8, try9, 8, "EIGHT"
                NumPat   try9, 0, 9, "NINE"

                NumPat   try20, try30, 20, "TWENTY"
                NumPat   try30, try40, 30, "THIRTY"
                NumPat   try40, try50, 40, "FORTY"
                NumPat   try50, try60, 50, "FIFTY"
                NumPat   try60, try70, 60, "SIXTY"
                NumPat   try70, try80, 70, "SEVENTY"
                NumPat   try80, try90, 80, "EIGHTY"
                NumPat   try90, 0, 90, "NINETY"

                include  stdsets.a

dseg            ends



cseg            segment  para public 'code'
                assume   cs:cseg, ds:dseg


; Semantic actions for our grammar:
;
;
;
; Get1000s-     We've just processed the value one..nine, grab it from
;               the value variable, multiply it by 1000, and store it
;               into thousandsval.

Get1000s        proc     far
                push     ds
                push     dx
                mov      ax, dseg
                mov      ds, ax

                mov      ax, 1000
                mul      Value
                mov      ThousandsVal, ax
                mov      Value, 0

                pop      dx
```

```
                mov     ax, di              ;Required by sl_match.
                pop     ds
                stc                         ;Always return success.
                ret
Get1000s        endp


; Get100s-      We've just processed the value one..nine, grab it from
;               the value variable, multiply it by 100, and store it
;               into hundredsval.

Get100s         proc    far
                push    ds
                push    dx
                mov     ax, dseg
                mov     ds, ax

                mov     ax, 100
                mul     Value
                mov     HundredsVal, ax
                mov     Value, 0

                pop     dx
                mov     ax, di              ;Required by sl_match.
                pop     ds
                stc                         ;Always return success.
                ret
Get100s         endp


; SetVal-       This routine sets Value to whatever is in si

SetVal          proc    far
                push    ds
                mov     ax, dseg
                mov     ds, ax
                mov     Value, si
                mov     ax, di
                pop     ds
                stc
                ret
SetVal          endp

; AddVal-       This routine sets adds whatever is in si to Value

AddVal          proc    far
                push    ds
                mov     ax, dseg
                mov     ds, ax
                add     Value, si
                mov     ax, di
                pop     ds
                stc
                ret
AddVal          endp


; Succeed matches the empty string. In other words, it matches anything
; and always returns success without eating any characters from the input
; string.

Succeed         proc    far
                mov     ax, di
                stc
                ret
Succeed         endp


; This subroutine expects a pointer to a string containing the English
; version of an integer number. It converts this to an integer and
```

```
; prints the result.

ConvertNumber    proc     near
                 mov      value, 0
                 mov      HundredsVal, 0
                 mov      ThousandsVal, 0

                 ldxi     N
                 xor      cx, cx
                 match
                 jnc      NoMatch
                 mov      al, "'"
                 putc
                 puts
                 print
                 byte     "' = ", 0
                 mov      ax, ThousandsVal
                 add      ax, HundredsVal
                 add      ax, Value
                 putu
                 putcr
                 jmp      Done

NoMatch:         print
                 byte     "Illegal number",cr,lf,0

Done:            ret
ConvertNumber    endp



Main             proc
                 mov      ax, dseg
                 mov      ds, ax
                 mov      es, ax

                 meminit                         ;Init memory manager.

; Union in a "-" to the delimiters set because numbers can have
; dashes in them.

                 lesi     delimiters
                 mov      al, '-'
                 addchar

; Some calls to test the ConvertNumber routine and the conversion process.

                 lesi     Str0
                 call     ConvertNumber
                 lesi     Str1
                 call     ConvertNumber
                 lesi     Str2
                 call     ConvertNumber
                 lesi     Str3
                 call     ConvertNumber
                 lesi     Str4
                 call     ConvertNumber
                 lesi     Str5
                 call     ConvertNumber
                 lesi     Str6
                 call     ConvertNumber
                 lesi     Str7
                 call     ConvertNumber
                 lesi     Str8
                 call     ConvertNumber
                 lesi     Str9
                 call     ConvertNumber
                 lesi     Str10
                 call     ConvertNumber
                 lesi     Str11
```

```
                call     ConvertNumber
                lesi     Str12
                call     ConvertNumber
                lesi     Str13
                call     ConvertNumber


Quit:           ExitPgm
Main            endp

cseg            ends

sseg            segment  para stack 'stack'
stk             db       1024 dup ("stack ")
sseg            ends

zzzzzzseg       segment  para public 'zzzzzz'
LastBytes       db       16 dup (?)
zzzzzzseg       ends
                end      Main
```

Sample output:

```
'twenty one' = 21
'nineteen hundred thirty-five' = 1935
'thirty three thousand two hundred nineteen' = 33219
'three' = 3
'fourteen' = 14
'fifty two' = 52
'seven hundred' = 700
'two thousand seven' = 2007
'four thousand ninety six' = 4096
'five hundred twelve' = 512
'twenty three thousand two hundred ninety-five' = 23295
'seventy-five hundred' = 7500
'sixty-five thousand' = 65000
'one thousand' = 1000
```

16.8.2 Processing Dates

Another useful program that converts English text to numeric form is a date processor. A date processor takes strings like "Jan 23, 1997" and converts it to three integer values representing the month, day, and year. Of course, while we're at it, it's easy enough to modify the grammar for date strings to allow the input string to take any of the following common date formats:

```
Jan 23, 1997
January 23, 1997
23 Jan, 1997
23 January, 1997
1/23/97
1-23-97
1/23/1997
1-23-1997
```

In each of these cases the date processing routines should store one into the variable month, 23 into the variable day, and 1997 into the year variable (we will assume all years are in the range 1900-1999 if the string supplies only two digits for the year). Of course, we could also allow dates like "January twenty-third, nineteen hundred and ninety seven" by using an number processing parser similar to the one presented in the previous section. However, that is an exercise left to the reader.

The grammar to process dates is

```
Date →          EngMon Integer Integer |
                Integer EngMon Integer |
```

```
                 Integer / Integer / Integer |
                 Integer - Integer - Integer

EngMon →         JAN | JANUARY | FEB | FEBRUARY | … | DEC | DECEMBER
Integer →        digit Integer | digit
digit  →         0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

We will use some semantic rules to place some restrictions on these strings. For example, the grammar above allows integers of any size; however, months must fall in the range 1-12 and days must fall in the range 1-28, 1-29, 1-30, or 1-31 depending on the year and month. Years must fall in the range 0-99 or 1900-1999.

Here is the 80x86 code for this grammar:

```
; datepat.asm
;
; This program converts dates of various formats to a three integer
; component value- month, day, and year.

                 .xlist
                 .286
                 include    stdlib.a
                 includelib stdlib.lib
                 matchfuncs
                 .list
                 .lall


dseg             segment  para public 'data'

; The following three variables hold the result of the conversion.

month            word     0
day              word     0
year             word     0

; StrPtr is a double word value that points at the string under test.
; The output routines use this variable. It is declared as two word
; values so it is easier to store es:di into it.

strptr           word     0,0

; Value is a generic variable the ConvertInt routine uses

value            word     0



; Number of valid days in each month (Feb is handled specially)

DaysInMonth      byte     31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31



; Some sample strings to test the date conversion routines.

Str0             byte     "Feb 4, 1956",0
Str1             byte     "July 20, 1960",0
Str2             byte     "Jul 8, 1964",0
Str3             byte     "1/1/97",0
Str4             byte     "1-1-1997",0
Str5             byte     "12-25-74",0
Str6             byte     "3/28/1981",0
Str7             byte     "January 1, 1999",0
Str8             byte     "Feb 29, 1996",0
Str9             byte     "30 June, 1990",0
Str10            byte     "August 7, 1945",0
Str11            byte     "30 September, 1992",0
Str12            byte     "Feb 29, 1990",0
Str13            byte     "29 Feb, 1992",0
```

```
; The following grammar is what we use to process the dates
;
; Date ->         EngMon Integer Integer
;      |          Integer EngMon Integer
;      |          Integer "/" Integer "/" Integer
;      |          Integer "-" Integer "-" Integer
;
; EngMon->        Jan | January | Feb | February | ... | Dec | December
; Integer->       digit integer | digit
; digit->         0 | 1 | ... | 9
;
; Some semantic rules this code has to check:
;
; If the year is in the range 0-99, this code has to add 1900 to it.
; If the year is not in the range 0-99 or 1900-1999 then return an error.
; The month must be in the range 1-12, else return an error.
; The day must be between one and 28, 29, 30, or 31. The exact maximum
; day depends on the month.


separators       pattern  {spancset, delimiters}


; DatePat processes dates of the form "MonInEnglish Day Year"

DatePat          pattern  {sl_match2, EngMon, DatePat2, DayYear}
DayYear          pattern  {sl_match2, DayInteger, 0, YearPat}
YearPat          pattern  {sl_match2, YearInteger}

; DatePat2 processes dates of the form "Day MonInEng Year"

DatePat2         pattern  {sl_match2, DayInteger, DatePat3, MonthYear}
MonthYear        pattern  {sl_match2, EngMon, 0, YearPat}

; DatePat3 processes dates of the form "mm-dd-yy"

DatePat3         pattern  {sl_match2, MonInteger, DatePat4, DatePat3a}
DatePat3a        pattern  {sl_match2, separators, DatePat3b, DatePat3b}
DatePat3b        pattern  {matchchar, '-', 0, DatePat3c}
DatePat3c        pattern  {sl_match2, DayInteger, 0, DatePat3d}
DatePat3d        pattern  {sl_match2, separators, DatePat3e, DatePat3e}
DatePat3e        pattern  {matchchar, '-', 0, DatePat3f}
DatePat3f        pattern  {sl_match2, YearInteger}

; DatePat4 processes dates of the form "mm/dd/yy"

DatePat4         pattern  {sl_match2, MonInteger, 0, DatePat4a}
DatePat4a        pattern  {sl_match2, separators, DatePat4b, DatePat4b}
DatePat4b        pattern  {matchchar, '/', 0, DatePat4c}
DatePat4c        pattern  {sl_match2, DayInteger, 0, DatePat4d}
DatePat4d        pattern  {sl_match2, separators, DatePat4e, DatePat4e}
DatePat4e        pattern  {matchchar, '/', 0, DatePat4f}
DatePat4f        pattern  {sl_match2, YearInteger}


; DayInteger matches an decimal string, converts it to an integer, and
; stores the result away in the Day variable.

DayInteger       pattern  {sl_match2, Integer, 0, SetDayPat}
SetDayPat        pattern  {SetDay}

; MonInteger matches an decimal string, converts it to an integer, and
; stores the result away in the Month variable.

MonInteger       pattern  {sl_match2, Integer, 0, SetMonPat}
SetMonPat        pattern  {SetMon}
```

```
; YearInteger matches an decimal string, converts it to an integer, and
; stores the result away in the Year variable.


YearInteger      pattern  {sl_match2, Integer, 0, SetYearPat}
SetYearPat       pattern  {SetYear}


; Integer skips any leading delimiter characters and then matches a
; decimal string. The Integer0 pattern matches exactly the decimal
; characters; the code does a patgrab on Integer0 when converting
; this string to an integer.

Integer          pattern  {sl_match2, separators, 0, Integer0}
Integer0         pattern  {sl_match2, number, 0, Convert2Int}
number           pattern  {anycset, digits, 0, number2}
number2          pattern  {spancset, digits}
Convert2Int      pattern  {ConvertInt}




; A macro to make it easy to declare each of the 24 English month
; patterns (24 because we allow the full month name and an
; abbreviation).

MoPat            macro    name, next, str, str2, value
                 local SetMo, string, full, short, string2, doMon

name             pattern  {sl_match2, short, next}
short            pattern  {matchistr, string2, full, SetMo}
full             pattern  {matchistr, string, 0, SetMo}

string           byte str
                 byte     0

string2          byte     str2
                 byte     0

SetMo            pattern  {MonthVal, value}
                 endm


; EngMon is a chain of patterns that match one of the strings
; JAN, JANUARY, FEB, FEBRUARY, etc. The last parameter to the
; MoPat macro is the month number.

EngMon           pattern {sl_match2, separators, jan, jan}
                 MoPat    jan, feb, “JAN”, “JANUARY”, 1
                 MoPat    feb, mar, “FEB”, “FEBRUARY”, 2
                 MoPat    mar, apr, “MAR”, “MARCH”, 3
                 MoPat    apr, may, “APR”, “APRIL”, 4
                 MoPat    may, jun, “MAY”, “MAY”, 5
                 MoPat    jun, jul, “JUN”, “JUNE”, 6
                 MoPat    jul, aug, “JUL”, “JULY”, 7
                 MoPat    aug, sep, “AUG”, “AUGUST”, 8
                 MoPat    sep, oct, “SEP”, “SEPTEMBER”, 9
                 MoPat    oct, nov, “OCT”, “OCTOBER”, 10
                 MoPat    nov, decem, “NOV”, “NOVEMBER”, 11
                 MoPat    decem, 0, “DEC”, “DECEMBER”, 12




; We use the “digits” and “delimiters” sets from the standard library.

                 include  stdsets.a

dseg             ends
```

```
cseg                segment   para public 'code'
                    assume    cs:cseg, ds:dseg


; ConvertInt-       Matches a sequence of digits and converts them to an integer.

ConvertInt          proc      far
                    push      ds
                    push      es
                    push      di
                    mov       ax, dseg
                    mov       ds, ax

                    lesi Integer0         ;Integer0 contains the decimal
                    patgrab               ; string we matched, grab that
                    atou                  ; string and convert it to an
                    mov       Value, ax   ; integer and save the result.
                    free                  ;Free mem allocated by patgrab.

                    pop       di
                    mov       ax, di      ;Required by sl_match.
                    pop       es
                    pop       ds
                    stc                   ;Always succeed.
                    ret

ConvertInt          endp


; SetDay, SetMon, and SetYear simply copy value to the appropriate
; variable.

SetDay              proc      far
                    push      ds
                    mov       ax, dseg
                    mov       ds, ax
                    mov       ax, value
                    mov       day, ax
                    mov       ax, di
                    pop       ds
                    stc
                    ret
SetDay              endp


SetMon              proc      far
                    push      ds
                    mov       ax, dseg
                    mov       ds, ax
                    mov       ax, value
                    mov       Month, ax
                    mov       ax, di
                    pop       ds
                    stc
                    ret
SetMon              endp


SetYear             proc      far
                    push      ds
                    mov       ax, dseg
                    mov       ds, ax
                    mov       ax, value
                    mov       Year, ax
                    mov       ax, di
                    pop       ds
                    stc
                    ret
```

```
SetYear         endp


; MonthVal is a pattern used by the English month patterns.
; This pattern function simply copies the matchparm field to
; the month variable (the matchparm field is passed in si).

MonthVal        proc    far
                push    ds
                mov     ax, dseg
                mov     ds, ax
                mov     Month, si
                mov     ax, di
                pop     ds
                stc
                ret
MonthVal        endp



; ChkDate-      Checks a date to see if it is valid. Returns with the
;               carry flag set if it is, clear if not.

ChkDate         proc    far
                push    ds
                push    ax
                push    bx

                mov     ax, dseg
                mov     ds, ax

; If the year is in the range 0-99, add 1900 to it.
; Then check to see if it's in the range 1900-1999.

                cmp     Year, 100
                ja      Notb100
                add     Year, 1900
Notb100:        cmp     Year, 2000
                jae     BadDate
                cmp     Year, 1900
                jb      BadDate

; Okay, make sure the month is in the range 1-12

                cmp     Month, 12
                ja      BadDate
                cmp     Month, 1
                jb      BadDate

; See if the number of days is correct for all months except Feb:

                mov     bx, Month
                mov     ax, Day                 ;Make sure Day <> 0.
                test    ax, ax
                je      BadDate
                cmp     ah, 0                   ;Make sure Day < 256.
                jne     BadDate

                cmp     bx, 2                   ;Handle Feb elsewhere.
                je      DoFeb
                cmp     al, DaysInMonth[bx-1]   ;Check against max val.
                ja      BadDate
                jmp     GoodDate

; Kludge to handle leap years. Note that 1900 is *not* a leap year.

DoFeb:          cmp     ax, 29                  ;Only applies if day is
                jb      GoodDate                ; equal to 29.
                ja      BadDate                 ;Error if Day > 29.
                mov     bx, Year                ;1900 is not a leap year
```

```
                cmp     bx, 1900                ; so handle that here.
                je      BadDate
                and     bx, 11b                 ;Else, Year mod 4 is a
                jne     BadDate                 ; leap year.

GoodDate:       pop     bx
                pop     ax
                pop     ds
                stc
                ret

BadDate:        pop     bx
                pop     ax
                pop     ds
                clc
                ret
ChkDate         endp


; ConvertDate-  ES:DI contains a pointer to a string containing a valid
;               date. This routine converts that date to the three
;               integer values found in the Month, Day, and Year
;               variables. Then it prints them to verify the pattern
;               matching routine.

ConvertDate     proc    near

                ldxi    DatePat
                xor     cx, cx
                match
                jnc     NoMatch

                mov     strptr, di              ;Save string pointer for
                mov     strptr+2, es            ; use by printf

                call    ChkDate                 ;Validate the date.
                jnc     NoMatch

                printf
                byte    "%-20^s = Month: %2d Day: %2d Year: %4d\n",0
                dword   strptr, Month, Day, Year
                jmp     Done

NoMatch:        printf
                byte    "Illegal date ('%^s')",cr,lf,0
                dword   strptr

Done:           ret
ConvertDate     endp



Main            proc
                mov     ax, dseg
                mov     ds, ax
                mov     es, ax

                meminit                         ;Init memory manager.

; Call ConvertDate to test several different date strings.

                lesi    Str0
                call    ConvertDate
                lesi    Str1
                call    ConvertDate
                lesi    Str2
                call    ConvertDate
                lesi    Str3
                call    ConvertDate
```

```
                  lesi      Str4
                  call      ConvertDate
                  lesi      Str5
                  call      ConvertDate
                  lesi      Str6
                  call      ConvertDate
                  lesi      Str7
                  call      ConvertDate
                  lesi      Str8
                  call      ConvertDate
                  lesi      Str9
                  call      ConvertDate
                  lesi      Str10
                  call      ConvertDate
                  lesi      Str11
                  call      ConvertDate
                  lesi      Str12
                  call      ConvertDate
                  lesi      Str13
                  call      ConvertDate


Quit:             ExitPgm
Main              endp

cseg              ends

sseg              segment   para stack 'stack'
stk               db        1024 dup ("stack ")
sseg              ends

zzzzzzseg         segment   para public 'zzzzzz'
LastBytes         db        16 dup (?)
zzzzzzseg         ends
                  end       Main
```

Sample Output:

```
Feb 4, 1956              = Month:  2 Day:  4 Year: 1956
July 20, 1960            = Month:  7 Day: 20 Year: 1960
Jul 8, 1964              = Month:  7 Day:  8 Year: 1964
1/1/97                   = Month:  1 Day:  1 Year: 1997
1-1-1997                 = Month:  1 Day:  1 Year: 1997
12-25-74                 = Month: 12 Day: 25 Year: 1974
3/28/1981                = Month:  3 Day: 28 Year: 1981
January 1, 1999          = Month:  1 Day:  1 Year: 1999
Feb 29, 1996             = Month:  2 Day: 29 Year: 1996
30 June, 1990            = Month:  6 Day: 30 Year: 1990
August 7, 1945           = Month:  8 Day:  7 Year: 1945
30 September, 1992       = Month:  9 Day: 30 Year: 1992
Illegal date ('Feb 29, 1990')
29 Feb, 1992             = Month:  2 Day: 29 Year: 1992
```

16.8.3 Evaluating Arithmetic Expressions

Many programs (e.g., spreadsheets, interpreters, compilers, and assemblers) need to process arithmetic expressions. The following example provides a simple calculator that operates on floating point numbers. This particular program uses the 80x87 FPU chip, although it would not be too difficult to modify it so that it uses the floating point routines in the UCR Standard Library.

```
; ARITH2.ASM
;
; A simple floating point calculator that demonstrates the use of the
; UCR Standard Library pattern matching routines. Note that this
```

```
; program requires an FPU.

                .xlist
                .386
                .387
                option    segment:use16
                include    stdlib.a
                includelib stdlib.lib
                matchfuncs
                .list

dseg            segment   para public 'data'

; The following is a temporary used when converting a floating point
; string to a 64 bit real value.

CurValue        real8     0.0

; Some sample strings containing expressions to try out:

Str1            byte      "5+2*(3-1)",0
Str2            byte      "(5+2)*(7-10)",0
Str3            byte      "5",0
Str4            byte      "(6+2)/(5+1)-7e5*2/1.3e2+1.5",0
Str5            byte      "2.5*(2-(3+1)/4+1)",0
Str6            byte      "6+(-5*2)",0
Str7            byte      "6*-1",0
Str8            byte      "1.2e5/2.1e5",0
Str9            byte      "0.9999999999999999+1e-15",0
str10           byte      "2.1-1.1",0

; Grammar for simple infix -> postfix translation operation:
; Semantic rules appear in braces.
;
; E -> FE' {print result}
; E' -> +F {fadd} E' | -F {fsub} E' | <empty string>
; F -> TF'
; F -> *T {fmul} F' | /T {fdiv} F' | <empty string>
; T -> -T {fchs} | S
; S -> <constant> {fld constant} | (E)
;
;
;
; UCR Standard Library Pattern which handles the grammar above:

; An expression consists of an "E" item followed by the end of the string:

Expression      pattern   {sl_Match2,E,,EndOfString}
EndOfString     pattern   {EOS}

; An "E" item consists of an "F" item optionally followed by "+" or "-"
; and another "E" item:

E               pattern   {sl_Match2, F,,Eprime}
Eprime          pattern   {MatchChar, '+', Eprime2, epf}
epf             pattern   {sl_Match2, F,,epPlus}
epPlus          pattern   {DoFadd,,,Eprime}

Eprime2         pattern   {MatchChar, '-', Succeed, emf}
emf             pattern   {sl_Match2, F,,epMinus}
epMinus         pattern   {DoFsub,,,Eprime}

; An "F" item consists of a "T" item optionally followed by "*" or "/"
; followed by another "T" item:

F               pattern   {sl_Match2, T,,Fprime}
Fprime          pattern   {MatchChar, '*', Fprime2, fmf}
fmf             pattern   {sl_Match2, T, 0, pMul}
pMul            pattern   {DoFmul,,,Fprime}
```

```
Fprime2          pattern  {MatchChar, '/', Succeed, fdf}
fdf              pattern  {sl_Match2, T, 0, pDiv}
pDiv             pattern  {DoFdiv, 0, 0,Fprime}

; T item consists of an "S" item or a "-" followed by another "T" item:

T                pattern  {MatchChar, '-', S, TT}
TT               pattern  {sl_Match2, T, 0,tpn}
tpn              pattern  {DoFchs}

; An "S" item is either a floating point constant or "(" followed by
; and "E" item followed by ")".
;
; The regular expression for a floating point constant is
;
;      [0-9]+ ( "." [0-9]* | ) ( ((e|E) (+|-| ) [0-9]+) | )
;
; Note: the pattern "Const" matches exactly the characters specified
;      by the above regular expression. It is the pattern the calc-
;      ulator grabs when converting a string to a floating point number.

Const            pattern  {sl_match2, ConstStr, 0, FLDConst}
ConstStr         pattern  {sl_match2, DoDigits, 0, Const2}
Const2           pattern  {matchchar, '.', Const4, Const3}
Const3           pattern  {sl_match2, DoDigits, Const4, Const4}
Const4           pattern  {matchchar, 'e', const5, const6}
Const5           pattern  {matchchar, 'E', Succeed, const6}
Const6           pattern  {matchchar, '+', const7, const8}
Const7           pattern  {matchchar, '-', const8, const8}
Const8           pattern  {sl_match2, DoDigits}

FldConst         pattern  {PushValue}

; DoDigits handles the regular expression [0-9]+

DoDigits         pattern  {Anycset, Digits, 0, SpanDigits}
SpanDigits       pattern  {Spancset, Digits}

; The S production handles constants or an expression in parentheses.

S                pattern  {MatchChar, '(', Const, IntE}
IntE             pattern  {sl_Match2, E, 0, CloseParen}
CloseParen       pattern  {MatchChar, ')'}

; The Succeed pattern always succeeds.

Succeed          pattern  {DoSucceed}


; We use digits from the UCR Standard Library cset standard sets.

                 include  stdsets.a

dseg             ends



cseg             segment  para public 'code'
                 assume   cs:cseg, ds:dseg

; DoSucceed matches the empty string. In other words, it matches anything
; and always returns success without eating any characters from the input
; string.

DoSucceed        proc     far
                 mov      ax, di
                 stc
                 ret
DoSucceed        endp
```

```
; DoFadd - Adds the two items on the top of the FPU stack.

DoFadd          proc    far
                faddp   st(1), st
                mov     ax, di                  ;Required by sl_Match
                stc                             ;Always succeed.
                ret
DoFadd          endp

; DoFsub - Subtracts the two values on the top of the FPU stack.

DoFsub          proc    far
                fsubp   st(1), st
                mov     ax, di                  ;Required by sl_Match
                stc
                ret
DoFsub          endp

; DoFmul- Multiplies the two values on the FPU stack.

DoFmul          proc    far
                fmulp   st(1), st
                mov     ax, di                  ;Required by sl_Match
                stc
                ret
DoFmul          endp

; DoFdiv- Divides the two values on the FPU stack.

DoFDiv          proc    far
                fdivp   st(1), st
                mov     ax, di                  ;Required by sl_Match
                stc
                ret
DoFDiv          endp

; DoFchs- Negates the value on the top of the FPU stack.

DoFchs          proc    far
                fchs
                mov     ax, di                  ;Required by sl_Match
                stc
                ret
DoFchs          endp


; PushValue-    We've just matched a string that corresponds to a
;               floating point constant. Convert it to a floating
;               point value and push that value onto the FPU stack.

PushValue       proc    far
                push    ds
                push    es
                pusha
                mov     ax, dseg
                mov     ds, ax

                lesi    Const           ;FP val matched by this pat.
                patgrab                 ;Get a copy of the string.
                atof                    ;Convert to real.
                free                    ;Return mem used by patgrab.
                lesi    CurValue        ;Copy floating point accumulator
                sdfpa                   ; to a local variable and then
                fld     CurValue        ; copy that value to the FPU stk.

                popa
                mov     ax, di
                pop     es
                pop     ds
```

```
                stc
                ret
PushValue       endp

; DoExp-        This routine expects a pointer to a string containing
;               an arithmetic expression in ES:DI. It evaluates the
;               given expression and prints the result.

DoExp           proc     near
                finit               ;Be sure to do this!
                fwait

                puts                ;Print the expression

                ldxi     Expression
                xor      cx, cx
                match
                jc       GoodVal
                printff
                byte     " is an illegal expression",cr,lf,0
                ret

GoodVal:        fstp     CurValue
                printff
                byte     " = %12.6ge\n",0
                dword    CurValue
                ret
DoExp           endp


; The main program tests the expression evaluator.

Main            proc
                mov      ax, dseg
                mov      ds, ax
                mov      es, ax
                meminit

                lesi     Str1
                call     DoExp
                lesi     Str2
                call     DoExp
                lesi     Str3
                call     DoExp
                lesi     Str4
                call     DoExp
                lesi     Str5
                call     DoExp
                lesi     Str6
                call     DoExp
                lesi     Str7
                call     DoExp
                lesi     Str8
                call     DoExp
                lesi     Str9
                call     DoExp
                lesi     Str10
                call     DoExp

Quit:           ExitPgm
Main            endp

cseg            ends

sseg            segment  para stack 'stack'
stk             db       1024 dup ("stack ")
sseg            ends

zzzzzzseg       segment  para public 'zzzzzz'
LastBytes       db       16 dup (?)
```

```
zzzzzzseg       ends
                end     Main
```

Sample Output:

```
5+2*(3-1) = 9.000E+0000
(5+2)*(7-10) = -2.100E+0001
5 = 5.000E+0000
(6+2)/(5+1)-7e5*2/1.3e2+1.5 = -1.077E+0004
2.5*(2-(3+1)/4+1) = 5.000E+0000
6+(-5*2) = -4.000E+0000
6*-1 = -6.000E+0000
1.2e5/2.1e5 = 5.714E-0001
0.9999999999999999+1e-15 = 1.000E+0000
2.1-1.1 = 1.000E+0000
```

16.8.4 A Tiny Assembler

Although the UCR Standard Library pattern matching routines would probably not be appropriate for writing a full lexical analyzer or compiler, they are useful for writing small compilers/assemblers or programs where speed of compilation/assembly is of little concern. One good example is the simple nonsymbolic assembler appearing in the SIM886[10] simulator for an earlier version of the x86 processors[11]. This "mini-assembler" accepts an x86 assembly language statement and immediately assembles it into memory. This allows SIM886 users to create simple assembly language programs within the SIM886 monitor/debugger[12]. Using the Standard Library pattern matching routines makes it very easy to implement such an assembler.

The grammar for this miniassembler is

```
Stmt →          Grp1 reg "," operand |
                Grp2 reg "," reg "," constant |
                Grp3 operand |
                goto operand |
                halt

Grp1 →          load | store | add | sub
Grp2 →          ifeq | iflt | ifgt
Grp3 →          get | put

reg →           ax | bx | cx | dx

operand →       reg | constant | [bx] | constant [bx]

constant →      hexdigit constant | hexdigit

hexdigit →      0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | a | b |
                c | d | e | f
```

There are some minor semantic details that the program handles (such as disallowing stores into immediate operands). The assembly code for the miniassembler follows:

```
; ASM.ASM
;

                .xlist
                include     stdlib.a
                matchfuncs
                includelib  stdlib.lib
                .list
```

10. SIM886 is an earlier version of SIMx86. It is also available on the Companion CD-ROM.
11. The current x86 system is written with Borland's Delphi, using a pattern matching library written for Pascal that is very similar to the Standard Library's pattern matching code.
12. See the lab manual for more details on SIM886.

```
dseg            segment  para public 'data'

; Some sample statements to assemble:

Str1            byte     "load ax, 0",0
Str2            byte     "load ax, bx",0
Str3            byte     "load ax, ax",0
Str4            byte     "add ax, 15",0
Str5            byte     "sub ax, [bx]",0
Str6            byte     "store bx, [1000]",0
Str7            byte     "load bx, 2000[bx]",0
Str8            byte     "goto 3000",0
Str9            byte     "iflt ax, bx, 100",0
Str10           byte     "halt",0
Str11           byte     "This is illegal",0
Str12           byte     "load ax, store",0
Str13           byte     "store ax, 1000",0
Str14           byte     "ifeq ax, 0, 0",0

; Variables used by the assembler.

AsmConst        word     0
AsmOpcode       byte     0
AsmOprnd1       byte     0
AsmOprnd2       byte     0

                include  stdsets.a    ;Bring in the standard char sets.

; Patterns for the assembler:

; Pattern is (
;        (load|store|add|sub) reg "," operand |
;        (ifeq|iflt|ifgt) reg1 "," reg2 "," const |
;        (get|put) operand |
;        goto operand |
;        halt
;        )
;
; With a few semantic additions (e.g., cannot store to a const).

InstrPat        pattern  {spancset, WhiteSpace,Grp1,Grp1}

Grp1            pattern  {sl_Match2,Grp1Strs, Grp2 ,Grp1Oprnds}
Grp1Strs        pattern  {TryLoad,,Grp1Store}
Grp1Store       pattern  {TryStore,,Grp1Add}
Grp1Add         pattern  {TryAdd,,Grp1Sub}
Grp1Sub         pattern  {TrySub}

; Patterns for the LOAD, STORE, ADD, and SUB instructions.

LoadPat         pattern  {MatchStr,LoadInstr2}
LoadInstr2      byte     "LOAD",0

StorePat        pattern  {MatchStr,StoreInstr2}
StoreInstr2     byte     "STORE",0

AddPat          pattern  {MatchStr,AddInstr2}
AddInstr2       byte     "ADD",0

SubPat          pattern  {MatchStr,SubInstr2}
SubInstr2       byte     "SUB",0

; Patterns for the group one (LOAD/STORE/ADD/SUB) instruction operands:

Grp1Oprnds      pattern  {spancset,WhiteSpace,Grp1reg,Grp1reg}
Grp1Reg         pattern  {MatchReg,AsmOprnd1,,Grp1ws2}
Grp1ws2         pattern  {spancset,WhiteSpace,Grp1Comma,Grp1Comma}
Grp1Comma       pattern  {MatchChar,',',0,Grp1ws3}
Grp1ws3         pattern  {spancset,WhiteSpace,Grp1Op2,Grp1Op2}
```

```
Grp1Op2         pattern  {MatchGen,,,EndOfLine}
EndOfLine       pattern  {spancset,WhiteSpace,NullChar,NullChar}
NullChar        pattern  {EOS}

Grp1Op2Reg      pattern  {MatchReg,AsmOprnd2}

; Patterns for the group two instructions (IFEQ, IFLT, IFGT):

Grp2            pattern  {sl_Match2,Grp2Strs, Grp3 ,Grp2Oprnds}
Grp2Strs        pattern  {TryIFEQ,,Grp2IFLT}
Grp2IFLT        pattern  {TryIFLT,,Grp2IFGT}
Grp2IFGT        pattern  {TryIFGT}

Grp2Oprnds      pattern  {spancset,WhiteSpace,Grp2reg,Grp2reg}
Grp2Reg         pattern  {MatchReg,AsmOprnd1,,Grp2ws2}
Grp2ws2         pattern  {spancset,WhiteSpace,Grp2Comma,Grp2Comma}
Grp2Comma       pattern  {MatchChar,',',0,Grp2ws3}
Grp2ws3         pattern  {spancset,WhiteSpace,Grp2Reg2,Grp2Reg2}
Grp2Reg2        pattern  {MatchReg,AsmOprnd2,,Grp2ws4}
Grp2ws4         pattern  {spancset,WhiteSpace,Grp2Comma2,Grp2Comma2}
Grp2Comma2      pattern  {MatchChar,',',0,Grp2ws5}
Grp2ws5         pattern  {spancset,WhiteSpace,Grp2Op3,Grp2Op3}
Grp2Op3         pattern  {ConstPat,,,EndOfLine}

; Patterns for the IFEQ, IFLT, and IFGT instructions.

IFEQPat         pattern  {MatchStr,IFEQInstr2}
IFEQInstr2      byte     "IFEQ",0

IFLTPat         pattern  {MatchStr,IFLTInstr2}
IFLTInstr2      byte     "IFLT",0

IFGTPat         pattern  {MatchStr,IFGTInstr2}
IFGTInstr2      byte     "IFGT",0

; Grp3 Patterns:

Grp3            pattern  {sl_Match2,Grp3Strs, Grp4 ,Grp3Oprnds}
Grp3Strs        pattern  {TryGet,,Grp3Put}
Grp3Put         pattern  {TryPut,,Grp3GOTO}
Grp3Goto        pattern  {TryGOTO}

; Patterns for the GET and PUT instructions.

GetPat          pattern  {MatchStr,GetInstr2}
GetInstr2       byte     "GET",0

PutPat          pattern  {MatchStr,PutInstr2}
PutInstr2       byte     "PUT",0

GOTOPat         pattern  {MatchStr,GOTOInstr2}
GOTOInstr2      byte     "GOTO",0

; Patterns for the group three (PUT/GET/GOTO) instruction operands:

Grp3Oprnds      pattern  {spancset,WhiteSpace,Grp3Op,Grp3Op}
Grp3Op          pattern  {MatchGen,,,EndOfLine}

; Patterns for the group four instruction (HALT).

Grp4            pattern  {TryHalt,,,EndOfLine}

HaltPat         pattern  {MatchStr,HaltInstr2}
HaltInstr2      byte     "HALT",0

; Patterns to match the four non-register addressing modes:

BXIndrctPat     pattern  {MatchStr,BXIndrctStr}
BXIndrctStr     byte     "[BX]",0
```

```
BXIndexedPat     pattern  {ConstPat,,,BXIndrctPat}

DirectPat        pattern  {MatchChar,'[`,,DP2}
DP2              pattern  {ConstPat,,,DP3}
DP3              pattern  {MatchChar,']'}

ImmediatePat     pattern  {ConstPat}

; Pattern to match a hex constant:

HexConstPat      pattern  {Spancset, xdigits}

dseg             ends

cseg             segment  para public 'code'
                 assume   cs:cseg, ds:dseg

; The store macro tweaks the DS register and stores into the
; specified variable in DSEG.

store            macro    Where, What
                 push     ds
                 push     ax
                 mov      ax, seg Where
                 mov      ds, ax
                 mov      Where, What
                 pop      ax
                 pop      ds
                 endm

; Pattern matching routines for the assembler.
; Each mnemonic has its own corresponding matching function that
; attempts to match the mnemonic. If it does, it initializes the
; AsmOpcode variable with the base opcode of the instruction.

; Compare against the "LOAD" string.

TryLoad          proc     far
                 push     dx
                 push     si
                 ldxi     LoadPat
                 match2
                 jnc      NoTLMatch

                 store    AsmOpcode, 0            ;Initialize base opcode.

NoTLMatch:       pop      si
                 pop      dx
                 ret
TryLoad          endp

; Compare against the "STORE" string.

TryStore         proc     far
                 push     dx
                 push     si
                 ldxi     StorePat
                 match2
                 jnc      NoTSMatch
                 store    AsmOpcode, 1            ;Initialize base opcode.

NoTSMatch:       pop      si
                 pop      dx
                 ret
TryStore         endp

; Compare against the "ADD" string.

TryAdd           proc     far
                 push     dx
```

```
                push    si
                ldxi    AddPat
                match2
                jnc     NoTAMatch
                store   AsmOpcode, 2            ;Initialize ADD opcode.

NoTAMatch:      pop     si
                pop     dx
                ret
TryAdd          endp

; Compare against the "SUB" string.

TrySub          proc    far
                push    dx
                push    si
                ldxi    SubPat
                match2
                jnc     NoTMMatch
                store   AsmOpcode, 3            ;Initialize SUB opcode.

NoTMMatch:      pop     si
                pop     dx
                ret
TrySub          endp

; Compare against the "IFEQ" string.

TryIFEQ         proc    far
                push    dx
                push    si
                ldxi    IFEQPat
                match2
                jnc     NoIEMatch
                store   AsmOpcode, 4            ;Initialize IFEQ opcode.

NoIEMatch:      pop     si
                pop     dx
                ret
TryIFEQ         endp

; Compare against the "IFLT" string.

TryIFLT         proc    far
                push    dx
                push    si
                ldxi    IFLTPat
                match2
                jnc     NoILMatch
                store   AsmOpcode, 5            ;Initialize IFLT opcode.

NoILMatch:      pop     si
                pop     dx
                ret
TryIFLT         endp

; Compare against the "IFGT" string.

TryIFGT         proc    far
                push    dx
                push    si
                ldxi    IFGTPat
                match2
                jnc     NoIGMatch
                store   AsmOpcode, 6            ;Initialize IFGT opcode.

NoIGMatch:      pop     si
                pop     dx
                ret
TryIFGT         endp
```

```
; Compare against the “GET” string.

TryGET          proc    far
                push    dx
                push    si
                ldxi    GetPat
                match2
                jnc     NoGMatch
                store   AsmOpcode, 7            ;Initialize Special opcode.
                store   AsmOprnd1, 2            ;GET’s Special opcode.

NoGMatch:       pop     si
                pop     dx
                ret
TryGET          endp

; Compare against the “PUT” string.

TryPut          proc    far
                push    dx
                push    si
                ldxi    PutPat
                match2
                jnc     NoPMatch
                store   AsmOpcode, 7            ;Initialize Special opcode.
                store   AsmOprnd1, 3            ;PUT’s Special opcode.

NoPMatch:       pop     si
                pop     dx
                ret
TryPUT          endp

; Compare against the “GOTO” string.

TryGOTO         proc    far
                push    dx
                push    si
                ldxi    GOTOPat
                match2
                jnc     NoGMatch
                store   AsmOpcode, 7            ;Initialize Special opcode.
                store   AsmOprnd1, 1            ;PUT’s Special opcode.

NoGMatch:       pop     si
                pop     dx
                ret
TryGOTO         endp

; Compare against the “HALT” string.

TryHalt         proc    far
                push    dx
                push    si
                ldxi    HaltPat
                match2
                jnc     NoHMatch
                store   AsmOpcode, 7            ;Initialize Special opcode.
                store   AsmOprnd1, 0            ;Halt’s special opcode.
                store   AsmOprnd2, 0

NoHMatch:       pop     si
                pop     dx
                ret
TryHALT         endp

; MatchReg checks to see if we’ve got a valid register value. On entry,
; DS:SI points at the location to store the byte opcode (0, 1, 2, or 3) for
; a reasonable register (AX, BX, CX, or DX); ES:DI points at the string
; containing (hopefully) the register operand, and CX points at the last
```

```
; location plus one we can check in the string.
;
; On return, Carry=1 for success, 0 for failure. ES:AX must point beyond
; the characters which make up the register if we have a match.

MatchReg        proc    far

; ES:DI Points at two characters which should be AX/BX/CX/DX. Anything
; else is an error.

                cmp     byte ptr es:1[di], 'X'        ;Everyone needs this
                jne     BadReg
                xor     ax, ax                        ;886 "AX" reg code.
                cmp     byte ptr es:[di], 'A'         ;AX?
                je      GoodReg
                inc     ax
                cmp     byte ptr es:[di], 'B'         ;BX?
                je      GoodReg
                inc     ax
                cmp     byte ptr es:[di], 'C'         ;CX?
                je      GoodReg
                inc     ax
                cmp     byte ptr es:[di], 'D'         ;DX?
                je      GoodReg
BadReg:         clc
                mov     ax, di
                ret

GoodReg:
                mov     ds:[si], al             ;Save register opcode.
                lea     ax, 2[di]               ;Skip past register.
                cmp     ax, cx                  ;Be sure we didn't go
                ja      BadReg                  ; too far.
                stc
                ret
MatchReg        endp

; MatchGen-     Matches a general addressing mode. Stuffs the appropriate
;               addressing mode code into AsmOprnd2. If a 16-bit constant
;               is required by this addressing mode, this code shoves that
;               into the AsmConst variable.

MatchGen        proc    far
                push    dx
                push    si

; Try a register operand.

                ldxi    Grp1Op2Reg
                match2
                jc      MGDone

; Try "[bx]".

                ldxi    BXIndrctPat
                match2
                jnc     TryBXIndexed
                store   AsmOprnd2, 4
                jmp     MGDone

; Look for an operand of the form "xxxx[bx]".

TryBXIndexed:
                ldxi    BXIndexedPat
                match2
                jnc     TryDirect
                store   AsmOprnd2, 5
                jmp     MGDone

; Try a direct address operand "[xxxx]".
```

```
TryDirect:
                ldxi    DirectPat
                match2
                jnc     TryImmediate
                store   AsmOprnd2, 6
                jmp     MGDone

; Look for an immediate operand “xxxx”.

TryImmediate:
                ldxi    ImmediatePat
                match2
                jnc     MGDone
                store   AsmOprnd2, 7

MGDone:
                pop     si
                pop     dx
                ret
MatchGen        endp

; ConstPat-     Matches a 16-bit hex constant. If it matches, it converts
;               the string to an integer and stores it into AsmConst.

ConstPat        proc    far
                push    dx
                push    si
                ldxi    HexConstPat
                match2
                jnc     CPDone

                push    ds
                push    ax
                mov     ax, seg AsmConst
                mov     ds, ax
                atoh
                mov     AsmConst, ax
                pop     ax
                pop     ds
                stc

CPDone:         pop     si
                pop     dx
                ret
ConstPat        endp

; Assemble-     This code assembles the instruction that ES:DI points
;               at and displays the hex opcode(s) for that instruction.

Assemble        proc    near

; Print out the instruction we’re about to assemble.

                print
                byte    “Assembling: “,0
                strupr
                puts
                putcr

; Assemble the instruction:

                ldxi    InstrPat
                xor     cx, cx
                match
                jnc     SyntaxError

; Quick check for illegal instructions:

                cmp     AsmOpcode, 7                    ;Special/Get instr.
```

```
                jne     TryStoreInstr
                cmp     AsmOprnd1, 2                    ;GET opcode
                je      SeeIfImm
                cmp     AsmOprnd1, 1                    ;Goto opcode
                je      IsGOTO

TryStoreInstr:  cmp     AsmOpcode, 1                    ;Store Instruction
                jne     InstrOkay

SeeIfImm:       cmp     AsmOprnd2, 7                    ;Immediate Adrs Mode
                jne     InstrOkay
                print
                db      "Syntax error: store/get immediate not allowed."
                db      " Try Again",cr,lf,0
                jmp     ASMDone

IsGOTO:         cmp     AsmOprnd2, 7            ;Immediate mode for GOTO
                je      InstrOkay
                print
                db      "Syntax error: GOTO only allows immediate "
                byte    "mode.",cr,lf
                db      0
                jmp     ASMDone

; Merge the opcode and operand fields together in the instruction byte,
; then output the opcode byte.

InstrOkay:      mov     al, AsmOpcode
                shl     al, 1
                shl     al, 1
                or      al, AsmOprnd1
                shl     al, 1
                shl     al, 1
                shl     al, 1
                or      al, AsmOprnd2
                puth
                cmp     AsmOpcode, 4                    ;IFEQ instruction
                jb      SimpleInstr
                cmp     AsmOpcode, 6                    ;IFGT instruction
                jbe     PutConstant

SimpleInstr:    cmp     AsmOprnd2, 5
                jb ASMDone

; If this instruction has a 16 bit operand, output it here.

PutConstant:    mov     al, ' '
                putc
                mov     ax, ASMConst
                puth
                mov     al, ' '
                putc
                xchg    al, ah
                puth
                jmp     ASMDone

SyntaxError:    print
                db      "Syntax error in instruction."
                db      cr,lf,0

ASMDone:        putcr
                ret
Assemble        endp

; Main program that tests the assembler.

Main            proc
                mov     ax, seg dseg ;Set up the segment registers
                mov     ds, ax
                mov     es, ax
```

```
                    meminit

                    lesi     Str1
                    call     Assemble
                    lesi     Str2
                    call     Assemble
                    lesi     Str3
                    call     Assemble
                    lesi     Str4
                    call     Assemble
                    lesi     Str5
                    call     Assemble
                    lesi     Str6
                    call     Assemble
                    lesi     Str7
                    call     Assemble
                    lesi     Str8
                    call     Assemble
                    lesi     Str9
                    call     Assemble
                    lesi     Str10
                    call     Assemble
                    lesi     Str11
                    call     Assemble
                    lesi     Str12
                    call     Assemble
                    lesi     Str13
                    call     Assemble
                    lesi     Str14
                    call     Assemble

Quit:               ExitPgm
Main                endp
cseg                ends

sseg                segment  para stack 'stack'
stk                 db       256 dup ("stack ")
sseg                ends

zzzzzzseg           segment  para public 'zzzzzz'
LastBytes           db       16 dup (?)
zzzzzzseg           ends
                    end      Main
```

Sample Output:

```
Assembling: LOAD AX, 0
07 00 00
Assembling: LOAD AX, BX
01
Assembling: LOAD AX, AX
00
Assembling: ADD AX, 15
47 15 00
Assembling: SUB AX, [BX]
64
Assembling: STORE BX, [1000]
2E 00 10
Assembling: LOAD BX, 2000[BX]
0D 00 20
Assembling: GOTO 3000
EF 00 30
Assembling: IFLT AX, BX, 100
A1 00 01
Assembling: HALT
E0
Assembling: THIS IS ILLEGAL
Syntax error in instruction.
```

```
Assembling: LOAD AX, STORE
Syntax error in instruction.

Assembling: STORE AX, 1000
Syntax error: store/get immediate not allowed. Try Again

Assembling: IFEQ AX, 0, 0
Syntax error in instruction.
```

16.8.5 The "MADVENTURE" Game

Computer games are a perfect example of programs that often use pattern matching. One class of computer games in general, the *adventure* game[13], is a perfect example of games that use pattern matching. An adventure style game excepts English-like commands from the user, parses these commands, and acts upon them. In this section we will develop an adventure game *shell*. That is, it will be a reasonably functional adventure style game, capable of accepting and processing user commands. All you need do is supply a story line and a few additional details to develop a fully functioning adventure class game.

An adventure game usually consists of some sort of *maze* through which the player moves. The program processes commands like *go north* or *go right* to move the player through the maze. Each move can deposit the player in a new room of the game. Generally, each room or area contains objects the player can interact with. This could be reward objects such as items of value or it could be an antagonistic object like a monster or enemy player.

Usually, an adventure game is a *puzzle* of some sort. The player finds clues and picks up useful object in one part of the maze to solve problems in other parts of the maze. For example, a player could pick up a key in one room that opens a chest in another; then the player could find an object in the chest that is useful elsewhere in the maze. The purpose of the game is to solve all the interlocking puzzles and maximize one's score (however that is done). This text will not dwell upon the subtleties of game design; that is a subject for a different text. Instead, we'll look at the tools and data structures required to implement the game design.

The Madventure game's use of pattern matching is quite different from the previous examples appearing in this chapter. In the examples up to this point, the matching routines specifically checked the validity of an input string; Madventure does not do this. Instead, it uses the pattern matching routines to simply determine if certain key words appear on a line input by the user. The program handles the actual parsing (determining if the command is syntactically correct). To understand how the Madventure game does this, it would help if we took a look at how to play the Madventure game[14].

The Madventure prompts the user to enter a command. Unlike the original adventure game that required commands like "GO NORTH" (with no other characters other than spaces as part of the command), Madventure allows you to write whole sentences and then it attempts to pick out the key words from those sentences. For example, Madventure accepts the "GO NORTH" command; however, it also accepts commands like "North is the direction I want to go" and "I want to go in the north direction." Madventure doesn't really care as long as it can find "GO" and "NORTH" *somewhere* on the command line. This is a little more flexible that the original Adventure game structure. Of course, this scheme isn't infallible, it will treat commands like "I absolutely, positively, do *NOT* want to go anywhere near the north direction" as a "GO NORTH" command. Oh well, the user almost always types just "GO NORTH" anyway.

13. These are called adventure games because the original program of the genre was called "Adventure."
14. One word of caution, no one is going to claim that Madventure is a great game. If it were, it would be sold, it wouldn't appear in this text! So don't expect too much from the design of the game itself.

A Madventure command usually consists of a *noun* keyword and a *verb* keyword. The Madventure recognizes six verbs and fourteen nouns[15]. The verbs are

```
verbs →           go | get | drop | inventory | quit | help
```

The nouns are

```
nouns →           north | south | east | west | lime | beer | card |
                  sign | program | homework | money | form | coupon
```

Obviously, Madventure does not allow all combinations of verbs and nouns. Indeed, the following patterns are the only legal ones:

```
LegalCmds →       go direction | get item | drop item | inventory |
                  quit | help

direction →       north | south | east | west

item →            lime | beer | card | sign | program | homework |
                  money | form | coupon
```

However, the pattern does not enforce this grammar. It just locates a noun and a verb on the line and, if found, sets the noun and verb variables to appropriate values to denote the keywords it finds. By letting the main program handle the parsing, the program is somewhat more flexible.

There are two main patterns in the Madventure program: NounPat and VerbPat. These patterns match words (nouns or verbs) using a regular expression like the following:

```
(ARB* ' ' | ε) word (' ' | EOS)
```

This regular expression matches a word that appears at the beginning of a sentence, at the end of a sentence, anywhere in the middle of a sentence, or a sentence consisting of a single word. Madventure uses a macro (MatchNoun or MatchVerb) to create an expression for each noun and verb in the above expression.

To get an idea of how Madvent processes words, consider the following VerbPat pattern:

```
VerbPat         pattern    {sl_match2, MatchGo}
                MatchVerb  MatchGO, MatchGet, "GO", 1
                MatchVerb  MatchGet, MatchDrop, "GET", 2
                MatchVerb  MatchDrop, MatchInv, "DROP", 3
                MatchVerb  MatchInv, MatchQuit, "INVENTORY", 4
                MatchVerb  MatchQuit, MatchHelp, "QUIT", 5
                MatchVerb  MatchHelp, 0, "HELP", 6
```

The MatchVerb macro expects four parameters. The first is an arbitrary pattern name; the second is a link to the next pattern in the list; the third is the string to match, and the fourth is a number that the matching routines will store into the verb variable if that string matches (by default, the verb variable contains zero). It is very easy to add new verbs to this list. For example, if you wanted to allow "run" and "walk" as synonyms for the "go" verb, you would just add two patterns to this list:

```
VerbPat         pattern       {sl_match2, MatchGo}
                MatchVerb  MatchGO, MatchGet, "GO", 1
                MatchVerb  MatchGet, MatchDrop, "GET", 2
                MatchVerb  MatchDrop, MatchInv, "DROP", 3
                MatchVerb  MatchInv, MatchQuit, "INVENTORY", 4
                MatchVerb  MatchQuit, MatchHelp, "QUIT", 5
                MatchVerb  MatchHelp, MatchRun, "HELP", 6
                MatchVerb  MatchRun, MatchWalk, "RUN", 1
                MatchVerb  MatchWalk, 0, "WALK", 1
```

There are only two things to consider when adding new verbs: first, don't forget that the next field of the last verb should contain zero; second, the current version of Madventure

15. However, one beautiful thing about Madventure is that it is *very* easy to extend and add more nouns and verbs.

only allows up to seven verbs. If you want to add more you will need to make a slight modification to the main program (more on that, later). Of course, if you only want to create synonyms, as we've done here, you simply reuse existing verb values so there is no need to modify the main program.

When you call the match routine and pass it the address of the VerbPat pattern, it scans through the input string looking for the first verb. If it finds that verb ("GO") it sets the verb variable to the corresponding verb value at the end of the pattern. If match cannot find the first verb, it tries the second. If that fails, it tries the third, and so on. If match cannot find *any* of the verbs in the input string, it does not modify the verb variable (which contains zero). If there are *two* or more of the above verbs on the input line, match will locate the first verb in the verb list above. *This may not be the first verb appearing on the line.* For example, if you say "Let's get the money and go north" the match routine will match the "go" verb, not the "get" verb. By the same token, the NounPat pattern would match the north noun, not the money noun. So this command would be identical to "GO NORTH."

The MatchNoun is almost identical to the MatchVerb macro; there is, however, one difference – the MatchNoun macro has an extra parameter which is the name of the data structure representing the given object (if there is one). Basically, all the nouns (in this version of Madventure) except NORTH, SOUTH, EAST, and WEST have some sort of data structure associated with them.

The maze in Madventure consists of nine rooms defined by the data structure:

```
Room            struct
north           word     ?
south           word     ?
west            word     ?
east            word     ?
ItemList        word     MaxWeight dup (?)
Description     word     ?
Room            ends
```

The north, south, west, and east fields contain near pointers to other rooms. The program uses the CurRoom variable to keep track of the player's current position in the maze. When the player issues a "GO" command of some sort, Madventure copies the appropriate value from the north, south, west, or east field to the CurRoom variable, effectively changing the room the user is in. If one of these pointers is NULL, then the user cannot move in that direction.

The direction pointers are independent of one another. If you issue the command "GO NORTH" and then issue the command "GO SOUTH" upon arriving in the new room, there is no guarantee that you will wind up in the original room. The south field of the second room may not point at the room that led you there. Indeed, there are several cases in the Madventure game where this occurs.

The ItemList array contains a list of near pointers to objects that could be in the room. In the current version of this game, the objects are all the nouns except *north, south, east*, and *west*. The player can carry these objects from room to room (indeed, that is the major purpose of this game). Up to MaxWeight objects can appear in the room (MaxWeight is an assembly time constant that is currently four; so there are a maximum of four items in any one given room). If an entry in the ItemList is non-NULL, then it is a pointer to an Item object. There may be zero to MaxWeight objects in a room.

The Description field contains a pointer to a zero terminated string that describes the room. The program prints this string each time through the command loop to keep the player oriented.

The second major data type in Madventure is the Item structure. This structure takes the form:

```
Item            struct
Value           word     ?
Weight          word     ?
Key             word     ?
ShortDesc       word     ?
LongDesc        word     ?
WinDesc         word     ?
Item            ends
```

The Value field contains an integer value awarded to the player when the player drops this object in the appropriate room. This is how the user scores points.

The Weight field usually contains one or two and determines how much this object "weighs." The user can only carry around MaxWeight units of weight at any one given time. Each time the user picks up an object, the weight of that object is added to the user's total weight. When the user drops an object, Madventure subtracts the object's weight from the total.

The Key field contains a pointer to a room associated with the object. When the user drops the object in the Key room, the user is awarded the points in the Value field and the object disappears from the game. If the user drops the object in some other room, the object stays in that room until the user picks it up again.

The ShortDesc, LongDesc, and WinDesc fields contain pointers to zero terminated strings. Madventure prints the ShortDesc string in response to an INVENTORY command. It prints the LongDesc string when describing a room's contents. It prints the WinDesc string when the user drops the object in its Key room and the object disappears from the game.

The Madventure main program is deceptively simple. Most of the logic is hidden in the pattern matching routines and in the parsing routine. We've already discussed the pattern matching code; the only important thing to remember is that it initializes the noun and verb variables with a value uniquely identifying each noun and verb. The main program's logic uses these two values as an index into a two dimensional table that takes the following form:

Table 65: Madventure Noun/Verb Table

	No Verb	GO	GET	DROP	Inventory	Quit	Help
No Noun					Inventory	Quit	Help
North		Do North					
South		Do South					
East		Do East					
West		Do West					
Lime			Get Item	Drop Item			
Beer			Get Item	Drop Item			
Card			Get Item	Drop Item			
Sign			Get Item	Drop Item			
Program			Get Item	Drop Item			

Table 65: Madventure Noun/Verb Table

	No Verb	GO	GET	DROP	Inven-tory	Quit	Help
Home-work			Get Item	Drop Item			
Money			Get Item	Drop Item			
Form			Get Item	Drop Item			
Coupon			Get Item	Drop Item			

The empty entries in this table correspond to illegal commands. The other entries are addresses of code within the main program that handles the given command.

To add more nouns (objects) to the game, you need only extend the NounPat pattern and add additional rows to the table (of course, you may need to add code to handle the new objects if they are not easily handled by the routines above). To add new verbs you need only extended the VerbPat pattern and add new columns to this table[16].

Other than the goodies mentioned above, the rest of the program utilizes techniques appearing throughout this and previous chapters. The only real surprising thing about this program is that you can implement a fairly complex program with so few lines of code. But such is the advantage of using pattern matching techniques in your assembly language programs.

```
; MADVENT.ASM
;
; This is a "shell" of an adventure game that you can use to create
; your own adventure style games.

                 .xlist
                 .286
                 include    stdlib.a
                 includelib stdlib.lib
                 matchfuncs
                 .list

dseg             segment    para public 'data'

; Equates:

NULL             equ      0
MaxWeight        equ      4            ;Max weight user can carry at one time.


; The "ROOM" data structure defines a room, or area, where a player can
; go. The NORTH, SOUTH, EAST, and WEST fields contain the address of
; the rooms to the north, south, east, and west of the room. The game
; transfers control to the room whose address appears in these fields
; when the player supplies a GO NORTH, GO SOUTH, etc., command.
;
; The ITEMLIST field contains a list of pointers to objects appearing
; in this room. In this game, the user can pick up and drop these
; objects (if there are any present).
;
; The DESCRIPTION field contains a (near) address of a short description
; of the current room/area.
```

16. Currently, the Madventure program computes the index into this table (a 14x8) table by shifting to the left three bits rather than multiplying by eight. You will need to modify this code if you add more columns to the table.

```
Room            struct
north           word     ? ;Near pointers to other structures where
south           word     ? ; we will wind up on the GO NORTH, GO SOUTH,
west            word     ? ; etc., commands.
east            word     ?

ItemList        word     MaxWeight dup (?)

Description     word     ? ;Description of room.
Room            ends


; The ITEM data structure describes the objects that may appear
; within a room (in the ITEMLIST above). The VALUE field contains
; the number of points this object is worth if the user drops it
; off in the proper room (i.e, solves the puzzle). The WEIGHT
; field provides the weight of this object. The user can only
; carry four units of weight at a time. This field is usually
; one, but may be more for larger objects. The KEY field is the
; address of the room where this object must be dropped to solve
; the problem. The SHORTDESC field is a pointer to a string that
; the program prints when the user executes an INVENTORY command.
; LONGDESC is a pointer to a string the program prints when des-
; cribing the contents of a room. The WINDESC field is a pointer
; to a string that the program prints when the user solves the
; appropriate puzzle.

Item            struct
Value           word     ?
Weight          word     ?
Key             word     ?
ShortDesc       word     ?
LongDesc        word     ?
WinDesc         word     ?
Item            ends


; State variables for the player:

CurRoom         word       Room1                  ;Room the player is in.
ItemsOnHand     word       MaxWeight dup (?)      ;Items the player carries.
CurWeight       word       0                      ;Weight of items carried.
CurScore        word       15                     ;Player's current score.
TotalCounter    word       9                      ;Items left to place.
Noun            word       0                      ;Current noun value.
Verb            word       0                      ;Current verb value.
NounPtr         word       0                      ;Ptr to current noun item.


; Input buffer for commands

InputLine       byte     128 dup (?)
; The following macros generate a pattern which will match a single word
; which appears anywhere on a line. In particular, they match a word
; at the beginning of a line, somewhere in the middle of the line, or
; at the end of a line. This program defines a word as any sequence
; of character surrounded by spaces or the beginning or end of a line.
;
; MatchNoun/Verb matches lines defined by the regular expression:
;
;       (ARB* ' ' | ε) string (' ' | EOS)

MatchNoun       macro      Name, next, WordString, ItemVal, ItemPtr
                local      WS1, WS2, WS3, WS4
                local      WS5, WS6, WordStr

Name            Pattern    {sl_match2, WS1, next}
WS1             Pattern    {MatchStr, WordStr, WS2, WS5}
WS2             Pattern    {arb,0,0,WS3}
WS3             Pattern    {Matchchar, ' ',0, WS4}
```

```
WS4             Pattern    {MatchStr, WordStr, 0, WS5}
WS5             Pattern    {SetNoun,ItemVal,0,WS6}
WS6             Pattern    {SetPtr, ItemPtr,0,MatchEOS}
WordStr         byte       WordString
                byte       0
                endm


MatchVerb       macro      Name, next, WordString, ItemVal
                local      WS1, WS2, WS3, WS4
                local      WS5, WordStr

Name            Pattern    {sl_match2, WS1, next}
WS1             Pattern    {MatchStr, WordStr, WS2, WS5}
WS2             Pattern    {arb,0,0,WS3}
WS3             Pattern    {Matchchar, ‘ ‘,0, WS4}
WS4             Pattern    {MatchStr, WordStr, 0, WS5}
WS5             Pattern    {SetVerb,ItemVal,0,MatchEOS}
WordStr         byte       WordString
                byte       0
                endm



; Generic patterns which most of the patterns use:

MatchEOS        Pattern    {EOS,0,MatchSpc}
MatchSpc        Pattern    {MatchChar,’ ‘}


; Here are the list of nouns allowed in this program.

NounPat         pattern    {sl_match2, MatchNorth}

                MatchNoun  MatchNorth, MatchSouth, “NORTH”, 1, 0
                MatchNoun  MatchSouth, MatchEast, “SOUTH”, 2, 0
                MatchNoun  MatchEast, MatchWest, “EAST”, 3, 0
                MatchNoun  MatchWest, MatchLime, “WEST”, 4, 0
                MatchNoun  MatchLime, MatchBeer, “LIME”, 5, Item3
                MatchNoun  MatchBeer, MatchCard, “BEER”, 6, Item9
                MatchNoun  MatchCard, MatchSign, “CARD”, 7, Item2
                MatchNoun  MatchSign, MatchPgm, “SIGN”, 8, Item1
                MatchNoun  MatchPgm, MatchHW, “PROGRAM”, 9, Item7
                MatchNoun  MatchHW, MatchMoney, “HOMEWORK”, 10, Item4
                MatchNoun  MatchMoney, MatchForm, “MONEY”, 11, Item5
                MatchNoun  MatchForm, MatchCoupon, “FORM”, 12, Item6
                MatchNoun  MatchCoupon, 0, “COUPON”, 13, Item8


; Here is the list of allowable verbs.

VerbPat         pattern    {sl_match2, MatchGo}

                MatchVerb  MatchGO, MatchGet, “GO”, 1
                MatchVerb  MatchGet, MatchDrop, “GET”, 2
                MatchVerb  MatchDrop, MatchInv, “DROP”, 3
                MatchVerb  MatchInv, MatchQuit, “INVENTORY”, 4
                MatchVerb  MatchQuit, MatchHelp, “QUIT”, 5
                MatchVerb  MatchHelp, 0, “HELP”, 6

; Data structures for the “maze”.

Room1           room       {Room1, Room5, Room4, Room2,
                            {Item1,0,0,0},
                            Room1Desc}

Room1Desc       byte       “at the Commons”,0

Item1           item       {10,2,Room3,GS1,GS2,GS3}
```

```
GS1             byte        "a big sign",0
GS2             byte        "a big sign made of styrofoam with funny "
                byte        "letters on it.",0
GS3             byte        "The ETA PI Fraternity thanks you for return"
                byte        "ing their sign, they",cr,lf
                byte        "make you an honorary life member, as long as "
                byte        "you continue to pay",cr,lf
                byte        "your $30 monthly dues, that is.",0

Room2           room        {NULL, Room5, Room1, Room3,
                             {Item2,0,0,0},
                             Room2Desc}

Room2Desc       byte        'at the "C" on the hill above campus',0

Item2           item        {10,1,Room1,LC1,LC2,LC3}
LC1             byte        "a lunch card",0
LC2             byte        "a lunch card which someone must have "
                byte        "accidentally dropped here.", 0
LC3             byte        "You get a big meal at the Commons cafeteria"
                byte        cr,lf
                byte        "It would be a good idea to go visit the "
                byte        "student health center",cr,lf
                byte        "at this time.",0

Room3           room        {NULL, Room6, Room2, Room2,
                             {Item3,0,0,0},
                             Room3Desc}

Room3Desc       byte        "at ETA PI Frat House",0

Item3           item        {10,2,Room2,BL1,BL2,BL3}
BL1             byte        "a bag of lime",0
BL2             byte        "a bag of baseball field lime which someone "
                byte        "is obviously saving for",cr,lf
                byte        "a special occasion.",0
BL3             byte        "You spread the lime out forming a big '++' "
                byte        "after the 'C'",cr,lf
                byte        "Your friends in Computer Science hold you "
                byte        "in total awe.",0

Room4           room        {Room1, Room7, Room7, Room5,
                             {Item4,0,0,0},
                             Room4Desc}

Room4Desc       byte        "in Dr. John Smith's Office",0

Item4           item        {10,1,Room7,HW1,HW2,HW3}
HW1             byte        "a homework assignment",0
HW2             byte        "a homework assignment which appears to "
                byte        "to contain assembly language",0
HW3             byte        "The grader notes that your homework "
                byte        "assignment looks quite",cr,lf
                byte        "similar to someone else's assignment "
                byte        "in the class and reports you",cr,lf
                byte        "to the instructor.",0

Room5           room        {Room1, Room9, Room7, Room2,
                             {Item5,0,0,0},
                             Room5Desc}

Room5Desc       byte         "in the computer lab",0

Item5           item        {10,1,Room9,M1,M2,M3}
M1              byte        "some money",0
M2              byte        "several dollars in an envelope in the "
                byte        "trashcan",0
M3              byte        "The waitress thanks you for your "
                byte        "generous tip and gets you",cr,lf
                byte        "another pitcher of beer. "
```

```
                byte        "Then she asks for your ID.",cr,lf
                byte        "You are at least 21 aren't you?",0

Room6           room        {Room3, Room9, Room5, NULL,
                             {Item6,0,0,0},
                             Room6Desc}

Room6Desc       byte        "at the campus book store",0

Item6           item        {10,1,Room8,AD1,AD2,AD3}
AD1             byte        "an add/drop/change form",0
AD2             byte        "an add/drop/change form filled out for "
                byte        "assembly to get a letter grade",0
AD3             byte        "You got the form in just in time. "
                byte        "It would have been a shame to",cr,lf
                byte        "have had to retake assembly because "
                byte        "you didn't realize you needed to ",cr,lf
                byte        "get a letter grade in the course.",0

Room7           room        {Room1, Room7, Room4, Room8,
                             {Item7,0,0,0},
                             Room7Desc}

Room7Desc       byte         "in the assembly lecture",0

Item7           item        {10,1,Room5,AP1,AP2,AP3}
AP1             byte        "an assembly language program",0
AP2             byte        "an assembly language program due in "
                byte        "the assemblylanguage class.",0
AP3             byte        "The sample program the instructor gave "
                byte        "you provided all the information",cr,lf
                byte        "you needed to complete your assignment. "
                byte        "You finish your work and",cr,lf
                byte        "head to the local pub to celebrate."
                byte        cr,lf,0

Room8           room        {Room5, Room6, Room7, Room9,
                             {Item8,0,0,0},
                             Room8Desc}

Room8Desc       byte         "at the Registrar's office",0

Item8           item        {10,1,Room6,C1,C2,C3}
C1              byte        "a coupon",0
C2              byte        "a coupon good for a free text book",0
C3              byte        'You get a free copy of "Cliff Notes for '
                byte        'The Art of Assembly',cr,lf
                byte        'Language Programming" Alas, it does not '
                byte        "provide all the",cr,lf
                byte        "information you need for the class, so you "
                byte        "sell it back during",cr,lf
                byte        "the book buy-back period.",0



Room9           room        {Room6, Room9, Room8, Room3,
                             {Item9,0,0,0},
                             Room9Desc}

Room9Desc       byte        "at The Pub",0
Item9           item        {10,2,Room4,B1,B2,B3}
B1              byte        "a pitcher of beer",0
B2              byte        "an ice cold pitcher of imported beer",0
B3              byte        "Dr. Smith thanks you profusely for your "
                byte        "good taste in brews.",cr,lf
                byte        "He then invites you to the pub for a "
                byte        "round of pool and",cr,lf
                byte        "some heavy duty hob-nobbing, "
                byte        "CS Department style.",0
```

```
dseg            ends


cseg            segment    para public 'code'
                assume     ds:dseg


; SetNoun-      Copies the value in SI (the matchparm parameter) to the
;               NOUN variable.

SetNoun         proc     far
                push     ds
                mov      ax, dseg
                mov      ds, ax
                mov      Noun, si
                mov      ax, di
                stc
                pop      ds
                ret
SetNoun         endp


; SetVerb-      Copies the value in SI (the matchparm parameter) to the
;               VERB variable.

SetVerb         proc     far
                push     ds
                mov      ax, dseg
                mov      ds, ax
                mov      Verb, si
                mov      ax, di
                stc
                pop      ds
                ret
SetVerb         endp

; SetPtr-       Copies the value in SI (the matchparm parameter) to the
;               NOUNPTR variable.

SetPtr          proc     far
                push     ds
                mov      ax, dseg
                mov      ds, ax
                mov      NounPtr, si
                mov      ax, di
                stc
                pop      ds
                ret
SetPtr          endp

; CheckPresence-
;               BX points at an item. DI points at an item list. This
;               routine checks to see if that item is present in the
;               item list. Returns Carry set if item was found,
;               clear if not found.

CheckPresence   proc

; MaxWeight is an assembly-time adjustable constant that determines
; how many objects the user can carry, or can be in a room, at one
; time. The following repeat macro emits "MaxWeight" compare and
; branch sequences to test each item pointed at by DS:DI.

ItemCnt         =        0
                repeat   MaxWeight
                cmp      bx, [di+ItemCnt]
                je       GotIt

ItemCnt         =        ItemCnt+2
                endm
```

```
                clc
                ret

GotIt:          stc
                ret
CheckPresence   endp

; RemoveItem-   BX contains a pointer to an item. DI contains a pointer
;               to an item list which contains that item. This routine
;               searches the item list and removes that item from the
;               list. To remove an item from the list, we need only
;               store a zero (NULL) over the top of its pointer entry
;               in the list.

RemoveItem      proc

; Once again, we use the repeat macro to automatically generate a chain
; of compare, branch, and remove code sequences for each possible item
; in the list.

ItemCnt         =       0
                repeat  MaxWeight
                local   NotThisOne
                cmp     bx, [di+ItemCnt]
                jne     NotThisOne
                mov     word ptr [di+ItemCnt], NULL
                ret
NotThisOne:
ItemCnt         =       ItemCnt+2
                endm

                ret
RemoveItem      endp


; InsertItem-   BX contains a pointer to an item, DI contains a pointer to
;               and item list. This routine searches through the list for
;               the first empty spot and copies the value in BX to that point.
;               It returns the carry set if it succeeds. It returns the
;               carry clear if there are no empty spots available.

InsertItem      proc

ItemCnt         =       0
                repeat  MaxWeight
                local   NotThisOne
                cmp     word ptr [di+ItemCnt], 0
                jne     NotThisOne
                mov     [di+ItemCnt], bx
                stc
                ret
NotThisOne:
ItemCnt         =       ItemCnt+2
                endm

                clc
                ret
InsertItem      endp

; LongDesc- Long description of an item.
; DI points at an item - print the long description of it.

LongDesc        proc
                push    di
                test    di, di
                jz      NoDescription
                mov     di, [di].item.LongDesc
                puts
                putcr
```

```
NoDescription:   pop      di
                 ret
LongDesc         endp


; ShortDesc- Print the short description of an object.
; DI points at an item (possibly NULL). Print the short description for it.

ShortDesc        proc
                 push     di
                 test     di, di
                 jz       NoDescription
                 mov      di, [di].item.ShortDesc
                 puts
                 putcr
NoDescription:   pop      di
                 ret
ShortDesc        endp

; Describe:      "CurRoom" points at the current room. Describe it and its
;                contents.

Describe         proc
                 push     es
                 push     bx
                 push     di
                 mov      di, ds
                 mov      es, di

                 mov      bx, CurRoom
                 mov      di, [bx].room.Description
                 print
                 byte     "You are currently ",0
                 puts
                 putcr
                 print
                 byte     "Here you find the following:",cr,lf,0

; For each possible item in the room, print out the long description
; of that item. The repeat macro generates a code sequence for each
; possible item that could be in this room.

ItemCnt          =        0
                 repeat   MaxWeight
                 mov      di, [bx].room.ItemList[ItemCnt]
                 call     LongDesc

ItemCnt          =        ItemCnt+2
                 endm


                 pop      di
                 pop      bx
                 pop      es
                 ret
Describe         endp


; Here is the main program, that actually plays the game.

Main             proc
                 mov      ax, dseg
                 mov      ds, ax
                 mov      es, ax
                 meminit

                 print
                 byte     cr,lf,lf,lf,lf,lf
                 byte     "Welcome to ",'"MADVENTURE"',cr,lf
                 byte     'If you need help, type the command "HELP"'
```

```
                byte    cr,lf,0

RoomLoop:       dec     CurScore        ;One point for each move.
                jnz     NotOverYet

; If they made too many moves without dropping anything properly, boot them
; out of the game.

                print
                byte    "WHOA! You lost! You get to join the legions of "
                byte    "the totally lame",cr,lf
                byte    'who have failed at "MADVENTURE"',cr,lf,0
                jmp     Quit

; Okay, tell 'em where they are and get a new command from them.

NotOverYet:     putcr
                call    Describe
                print
                byte    cr,lf
                byte    "Command: ",0
                lesi    InputLine
                gets
                strupr                  ;Ignore case by converting to U.C.

; Okay, process the command. Note that we don't actually check to see
; if there is a properly formed sentence. Instead, we just look to see
; if any important keywords are on the line. If they are, the pattern
; matching routines load the appropriate values into the noun and verb
; variables (nouns: north=1, south=2, east=3, west=4, lime=5, beer=6,
; card=7, sign=8, program=9, homework=10, money=11, form=12, coupon=13;
; verbs: go=1, get=2, drop=3, inventory=4, quit=5, help=6).
;
; This code uses the noun and verb variables as indexes into a two
; dimensional array whose elements contain the address of the code
; to process the given command. If a given command does not make
; any sense (e.g., "go coupon") the entry in the table points at the
; bad command code.

                mov     Noun, 0
                mov     Verb, 0
                mov     NounPtr, 0

                ldxi    VerbPat
                xor     cx, cx
                match

                lesi    InputLine
                ldxi    NounPat
                xor     cx, cx
                match

; Okay, index into the command table and jump to the appropriate
; handler. Note that we will cheat and use a 14x8 array. There
; are really only seven verbs, not eight. But using eight makes
; things easier since it is easier to multiply by eight than seven.

                mov     si, CurRoom;The commands expect this here.

                mov     bx, Noun
                shl     bx, 3           ;Multiply by eight.
                add     bx, Verb
                shl     bx, 1           ;Multiply by two - word table.
                jmp     cseg:jmptbl[bx]

; The following table contains the noun x verb cross product.
; The verb values (in each row) are the following:
;
;     NONE      GO       GET DROP    INVNTRY   QUIT   HELP   unused
;      0         1        2   3        4         5      6      7
```

```
;
; There is one row for each noun (plus row zero, corresponding to no
; noun found on line).

jmptbl          word     Bad          ;No noun, no verb
                word     Bad          ;No noun, GO
                word     Bad          ;No noun, GET
                word     Bad          ;No noun, DROP
                word     DoInventory  ;No noun, INVENTORY
                word     QuitGame     ;No noun, QUIT
                word     DoHelp       ;No noun, HELP
                word     Bad          ;N/A

NorthCmds       word     Bad, GoNorth, Bad, Bad, Bad, Bad, Bad, Bad
SouthCmds       word     Bad, GoSouth, Bad, Bad, Bad, Bad, Bad, Bad
EastCmds        word     Bad, GoEast, Bad, Bad, Bad, Bad, Bad, Bad
WestCmds        word     Bad, GoWest, Bad, Bad, Bad, Bad, Bad, Bad
LimeCmds        word     Bad, Bad, GetItem, DropItem, Bad, Bad, Bad, Bad
BeerCmds        word     Bad, Bad, GetItem, DropItem, Bad, Bad, Bad, Bad
CardCmds        word     Bad, Bad, GetItem, DropItem, Bad, Bad, Bad, Bad
SignCmds        word     Bad, Bad, GetItem, DropItem, Bad, Bad, Bad, Bad
ProgramCmds     word     Bad, Bad, GetItem, DropItem, Bad, Bad, Bad, Bad
HomeworkCmds    word     Bad, Bad, GetItem, DropItem, Bad, Bad, Bad, Bad
MoneyCmds       word     Bad, Bad, GetItem, DropItem, Bad, Bad, Bad, Bad
FormCmds        word     Bad, Bad, GetItem, DropItem, Bad, Bad, Bad, Bad
CouponCmds      word     Bad, Bad, GetItem, DropItem, Bad, Bad, Bad, Bad

; If the user enters a command we don’t know how to process, print an
; appropriate error message down here.

Bad:            printf
                byte     “I’m sorry, I don’t understand how to ‘%s’\n”,0
                dword    InputLine
                jmp      NotOverYet


; Handle the movement commands here.
; Movements are easy, all we’ve got to do is fetch the NORTH, SOUTH,
; EAST, or WEST pointer from the current room’s data structure and
; set the current room to that address. The only catch is that some
; moves are not legal. Such moves have a NULL (zero) in the direction
; field. A quick check for this case handles illegal moves.

GoNorth:        mov      si, [si].room.North
                jmp      MoveMe

GoSouth:        mov      si, [si].room.South
                jmp      MoveMe

GoEast:         mov      si, [si].room.East
                jmp      MoveMe

GoWest:         mov      si, [si].room.West
MoveMe:         test     si, si                 ;See if move allowed.
                jnz      SetCurRoom
                printf
                byte     “Sorry, you cannot go in this direction.”
                byte     cr, lf, 0
                jmp      RoomLoop

SetCurRoom:     mov      CurRoom, si  ;Move to new room.
                jmp      RoomLoop

; Handle the GetItem command down here. At this time the user
; has entered GET and some noun that the player can pick up.
; First, we will make sure that item is in this room.
; Then we will check to make sure that picking up this object
; won’t overload the player. If these two conditions are met,
; we’ll transfer the object from the room to the player.
```

```
GetItem:        mov     bx, NounPtr  ;Ptr to item user wants.
                mov     si, CurRoom
                lea     di, [si].room.ItemList;Ptr to item list in di.
                call    CheckPresence;See if in room.
                jc      GotTheItem
                printf
                byte    "Sorry, that item is not available here."
                byte    cr, lf, 0
                jmp     RoomLoop

; Okay, see if picking up this object will overload the player.

GotTheItem:     mov     ax, [bx].Item.Weight
                add     ax, CurWeight
                cmp     ax, MaxWeight
                jbe     WeightOkay
                printf
                byte    "Sorry, you are already carrying too many items "
                byte    "to safely carry\nthat object\n",0
                jmp     RoomLoop

; Okay, everything's cool, transfer the object from the room to the user.

WeightOkay:     mov     CurWeight, ax;Save new weight.
                call    RemoveItem   ;Remove item from room.
                lea     di, ItemsOnHand;Ptr to player's list.
                call    InsertItem
                jmp     RoomLoop


; Handle dropped objects down here.

DropItem:       lea     di, ItemsOnHand;See if the user has
                mov     bx, NounPtr  ; this item on hand.
                call    CheckPresence
                jc      CanDropIt1
                printf
                byte    "You are not currently holding that item\n",0
                jmp     RoomLoop

; Okay, let's see if this is the magic room where this item is
; supposed to be dropped. If so, award the user some points for
; properly figuring this out.

CanDropIt1:     mov     ax, [bx].item.key
                cmp     ax, CurRoom
                jne     JustDropIt

; Okay, success! Print the winning message for this object.

                mov     di, [bx].item.WinDesc
                puts
                putcr

; Award the user some points.

                mov     ax, [bx].item.value
                add     CurScore, ax

; Since the user dropped it, they can carry more things now.

                mov     ax, [bx].item.Weight
                sub     CurWeight, ax

; Okay, take this from the user's list.

                lea     di, ItemsOnHand
                call    RemoveItem

; Keep track of how may objects the user has successfully dropped.
```

```
; When this counter hits zero, the game is over.

                dec     TotalCounter
                jnz     RoomLoop

                printf
                byte    "Well, you've found where everything goes "
                byte    "and your score is %d.\n"
                byte    "You might want to play again and see if "
                byte    "you can get a better score.\n",0
                dword   CurScore
                jmp     Quit

; If this isn't the room where this object belongs, just drop the thing
; off. If this object won't fit in this room, ignore the drop command.

JustDropIt:     mov     di, CurRoom
                lea     di, [di].room.ItemList
                call    InsertItem
                jc      DroppedItem
                printf
                byte    "There is insufficient room to leave "
                byte    "that item here.\n",0
                jmp     RoomLoop

; If they can drop it, do so. Don't forget we've just unburdened the
; user so we need to deduct the weight of this object from what the
; user is currently carrying.

DroppedItem:    lea     di, ItemsOnHand
                call    RemoveItem
                mov     ax, [bx].item.Weight
                sub     CurWeight, ax
                jmp     RoomLoop

; If the user enters the INVENTORY command, print out the objects on hand

DoInventory:    printf
                byte    "You currently have the following items in your "
                byte    "possession:",cr,lf,0
                mov     di, ItemsOnHand[0]
                call    ShortDesc
                mov     di, ItemsOnHand[2]
                call    ShortDesc
                mov     di, ItemsOnHand[4]
                call    ShortDesc
                mov     di, ItemsOnHand[6]
                call    ShortDesc
                printf
                byte    "\nCurrent score: %d\n"
                byte    "Carrying ability: %d/4\n\n",0
                dword   CurScore,CurWeight
                inc     CurScore      ;This command is free.
                jmp     RoomLoop


; If the user requests help, provide it here.

DoHelp:         printf
                byte    "List of commands:",cr,lf,lf
                byte    "GO {NORTH, EAST, WEST, SOUTH}",cr,lf
                byte    "{GET, DROP} {LIME, BEER, CARD, SIGN, PROGRAM, "
                byte    "HOMEWORK, MONEY, FORM, COUPON}",cr,lf
                byte    "SHOW INVENTORY",cr,lf
                byte    "QUIT GAME",cr,lf
                byte    "HELP ME",cr,lf,lf
                byte    "Each command costs you one point.",cr,lf
                byte    "You accumulate points by picking up objects and "
                byte    "dropping them in their",cr,lf
                byte    " appropriate locations.",cr,lf
```

```
                    byte     "If you drop an item in its proper location, it "
                    byte     "disappears from the game.",cr,lf
                    byte     "The game is over if your score drops to zero or "
                    byte     "you properly place",cr,lf
                    byte     " all items.",cr,lf
                    byte     0
                    jmp      RoomLoop


; If they quit prematurely, let 'em know what a wimp they are!

QuitGame:           printf
                    byte     "So long, your score is %d and there are "
                    byte     "still %d objects unplaced\n",0
                    dword    CurScore, TotalCounter

Quit:               ExitPgm              ;DOS macro to quit program.
Main                endp
cseg                ends

sseg                segment  para stack 'stack'
stk                 db       1024 dup ("stack ")
sseg                ends

zzzzzzseg           segment  para public 'zzzzzz'
LastBytes           db       16 dup (?)
zzzzzzseg           ends
                    end      Main
```

16.9 Laboratory Exercises

Programming with the Standard Library Pattern Matching routines doubles the complexity. Not only must you deal with the complexities of 80x86 assembly language, you must also deal with the complexities of the pattern matching paradigm, a programming language in its own right. While you can use a program like CodeView to track down problems in an assembly language program, no such debugger exists for "programs" you write with the Standard Library's pattern matching "language." Although the pattern matching routines are written in assembly language, attempting to trace through a pattern using CodeView will not be very enlightening. In this laboratory exercise, you will learn how to develop some rudimentary tools to help debug pattern matching programs.

16.9.1 Checking for Stack Overflow (Infinite Loops)

One common problem in pattern matching programs is the possibility of an infinite loop occurring in the pattern. This might occur, for example, if you have a left recursive production. Unfortunately, tracking down such loops in a pattern is very tedious, even with the help of a debugger like CodeView. Fortunately, there is a very simple change you can make to a program that uses patterns that will abort the program an warn you if infinite recursion exists.

Infinite recursion in a pattern occurs when sl_Match2 continuously calls itself without ever returning. This overflows the stack and causes the program to crash. There is a very easy change you can make to your programs to check for stack overflow:

- In patterns where you would normally call sl_Match2, call MatchPat instead.
- Include the following statements near the beginning of your program (before any patterns):

```
DEBUG               =        0            ;Define for debugging.

                    ifdef    DEBUG
```

```
MatchPat         textequ  <MatchSP>
                 else
MatchPat         textequ  <sl_Match2>
                 endif
```

If you define the DEBUG symbol, your patterns will call the MatchSP procedure, otherwise they will call the sl_Match2 procedure. During testing, define the DEBUG symbol.

- Insert the following procedure somewhere in your program:

```
MatchSP          proc    far
                 cmp      sp, offset StkOvrfl
                 jbe      AbortPgm
                 jmp      sl_Match2

AbortPgm:        print
                 byte    cr,lf,lf
                 byte     "Error: Stack overflow in MatchSP routine.",cr,lf,0
                 ExitPgm
MatchSP          endp
```

This code sandwiches itself between your pattern and the sl_Match2 routine. It checks the stack pointer (sp) to see if it has dropped below a minimally acceptable point in the stack segment. If not, it continues execution by jumping to the sl_Match2 routine; otherwise it aborts program execution with an error message.

- The final change to your program is to modify the stack segment so that it looks like the following:

```
sseg             segment  para stack 'stack'
                 word     64 dup (?)              ;Buffer for stack overflow
StkOvrfl         word     ?                       ;Stack overflow if drops
stk              db       1024 dup ("stack   ")   ; below StkOvrfl.
sseg             ends
```

After making these changes, your program will automatically stop with an error message if infinite recursion occurs since infinite recursion will most certainly cause a stack overflow[17].

The following code (Ex16_1a.asm on the companion CD-ROM) presents a simple calculator, similar to the calculator in the section "Evaluating Arithmetic Expressions" on page 948, although this calculator only supports addition. As noted in the comments appearing in this program, the pattern for the expression parser has a serious flaw – it uses a left recursive production. This will most certainly cause an infinite loop and a stack overflow. **For your lab report:** Run this program with and without the DEBUG symbol defined (i.e., comment out the definition for one run). Describe what happens.

```
; EX16_1a.asm
;
; A simple floating point calculator that demonstrates the use of the
; UCR Standard Library pattern matching routines.  Note that this
; program requires an FPU.

                 .xlist
                 .386
                 .387
                 option     segment:use16
                 include    stdlib.a
                 includelib stdlib.lib
                 matchfuncs
                 .list
```

17. This code will also abort your program if you use too much stack space without infinite recursion. A problem in its own right.

```
; If the symbol "DEBUG" is defined, then call the MatchSP routine
; to do stack overflow checking.  If "DEBUG" is not defined, just
; call the sl_Match2 routine directly.

DEBUG           =       0           ;Define for debugging.

                ifdef   DEBUG
MatchPat        textequ <MatchSP>
                else
MatchPat        textequ <sl_Match2>
                endif

dseg            segment para public 'data'

; The following is a temporary used when converting a floating point
; string to a 64 bit real value.

CurValue        real8   0.0

; A Test String:

TestStr         byte    "5+2-(3-1)",0

; Grammar for simple infix -> postfix translation operation:
; Semantic rules appear in braces.
;
; NOTE: This code has a serious problem.  The first production
; is left recursive and will generate an infinite loop.
;
; E -> E+T {print result} | T {print result}
; T -> <constant> {fld constant} | (E)
;
;
; UCR Standard Library Pattern that handles the grammar above:

; An expression consists of an "E" item followed by the end of the string:

Expression      pattern {MatchPat,E,,EndOfString}
EndOfString     pattern {EOS}


; An "E" item consists of an "E" item optionally followed by "+" or "-"
; and a "T" item (E -> E+T | T):

E               pattern {MatchPat, E,T,Eplus}
Eplus           pattern {MatchChar, '+', T, epPlus}
epPlus          pattern {DoFadd}


; A "T" item is either a floating point constant or "(" followed by
; an "E" item followed by ")".
;
; The regular expression for a floating point constant is
;
;       [0-9]+ ( "." [0-9]* | ) ( ((e|E) (+|-| ) [0-9]+) | )
;
; Note: the pattern "Const" matches exactly the characters specified
;       by the above regular expression.  It is the pattern the calc-
;       ulator grabs when converting a string to a floating point number.


Const           pattern {MatchPat, ConstStr, 0, FLDConst}
ConstStr        pattern {MatchPat, DoDigits, 0, Const2}
Const2          pattern {matchchar, '.', Const4, Const3}
Const3          pattern {MatchPat, DoDigits, Const4, Const4}
Const4          pattern {matchchar, 'e', const5, const6}
Const5          pattern {matchchar, 'E', Succeed, const6}
Const6          pattern {matchchar, '+', const7, const8}
Const7          pattern {matchchar, '-', const8, const8}
```

```
Const8          pattern   {MatchPat, DoDigits}

FldConst        pattern   {PushValue}

; DoDigits handles the regular expression [0-9]+

DoDigits        pattern   {Anycset, Digits, 0, SpanDigits}
SpanDigits      pattern   {Spancset, Digits}

; The S production handles constants or an expression in parentheses.

T               pattern   {MatchChar, '(', Const, IntE}
IntE            pattern   {MatchPat, E, 0, CloseParen}
CloseParen      pattern   {MatchChar, ')'}


; The Succeed pattern always succeeds.

Succeed         pattern   {DoSucceed}


; We use digits from the UCR Standard Library cset standard sets.

                include   stdsets.a

dseg            ends



cseg            segment   para public 'code'
                assume    cs:cseg, ds:dseg

; Debugging feature #1:
; This is a special version of sl_Match2 that checks for
; stack overflow.  Stack overflow occurs whenever there
; is an infinite loop (i.e., left recursion) in a pattern.

MatchSP         proc    far
                cmp       sp, offset StkOvrfl
                jbe       AbortPgm
                jmp       sl_Match2

AbortPgm:       print
                byte    cr,lf,lf
                byte      "Error: Stack overflow in MatchSP routine.",cr,lf,0
                ExitPgm
MatchSP         endp

; DoSucceed matches the empty string.  In other words, it matches anything
; and always returns success without eating any characters from the input
; string.

DoSucceed       proc      far
                mov       ax, di
                stc
                ret
DoSucceed       endp

; DoFadd - Adds the two items on the top of the FPU stack.

DoFadd          proc      far
                faddp     st(1), st
                mov       ax, di                  ;Required by sl_Match
                stc                               ;Always succeed.
                ret
DoFadd          endp


; PushValue-    We've just matched a string that corresponds to a
;               floating point constant.  Convert it to a floating
```

```
;                   point value and push that value onto the FPU stack.

PushValue           proc     far
                    push     ds
                    push     es
                    pusha
                    mov      ax, dseg
                    mov      ds, ax

                    lesi     Const         ;FP val matched by this pat.
                    patgrab                ;Get a copy of the string.
                    atof                   ;Convert to real.
                    free                   ;Return mem used by patgrab.
                    lesi     CurValue      ;Copy floating point accumulator
                    sdfpa                  ; to a local variable and then
                    fld      CurValue      ; copy that value to the FPU stk.

                    popa
                    mov      ax, di
                    pop      es
                    pop      ds
                    stc
                    ret
PushValue           endp

; The main program tests the expression evaluator.

Main                proc
                    mov      ax, dseg
                    mov      ds, ax
                    mov      es, ax
                    meminit

                    finit                  ;Be sure to do this!
                    fwait

                    lesi     TestStr
                    puts                   ;Print the expression

                    ldxi     Expression
                    xor      cx, cx
                    match
                    jc       GoodVal
                    printff
                    byte     " is an illegal expression",cr,lf,0
                    ret

GoodVal:            fstp     CurValue
                    printff
                    byte     " = %12.6ge\n",0
                    dword    CurValue

Quit:               ExitPgm
Main                endp
cseg                ends

sseg                segment  para stack 'stack'
                    word     64 dup (?)    ;Buffer for stack overflow
StkOvrfl            word     ?                       ;Stack overflow if drops
stk                 db       1024 dup ("stack   "); below StkOvrfl.
sseg                ends

zzzzzzseg           segment  para public 'zzzzzz'
LastBytes           db       16 dup (?)
zzzzzzseg           ends
                    end      Main
```

16.9.2 Printing Diagnostic Messages from a Pattern

When there is no other debugging method available, you can always use print statements to help track down problems in your patterns. If your program calls pattern matching functions in your own code (like the DoFAdd, DoSucceed, and PushValue procedures in the code above), you can easily insert print or printf statements in these functions that will print an appropriate message when they execute. Unfortunately, a problem may develop in a portion of a pattern that does not call any local pattern matching functions, so inserting print statements within an existing (local) pattern matching function might not help. To solve this problem, all you need to do is insert a call to a local pattern matching function in the patterns you suspect have a problem.

Rather than make up a specific local pattern to print an individual message, a better solution is to write a generic pattern matching function whose whole purpose is to display a message. The following PatPrint function does exactly this:

```
; PatPrint- A debugging aid.  This "Pattern matching function" prints
; the string that DS:SI points at.

PatPrint         proc     far
                 push     es
                 push     di
                 mov      di, ds
                 mov      es, di
                 mov      di, si
                 puts
                 mov      ax, di
                 pop      di
                 pop      es
                 stc
                 ret
PatPrint         endp
```

From "Constructing Patterns for the MATCH Routine" on page 933, you will note that the pattern matching system passes the value of the MatchParm parameter to a pattern matching function in the ds:si register pair. The PatPrint function prints the string that ds:si points at (by moving ds:si to es:di and calling puts).

The following code (Ex16_1b.asm on the companion CD-ROM) demonstrates how to insert calls to PatPrint within your patterns to print out data to help you track down problems in your patterns. **For your lab report:** run this program and describe its output in your report. Describe how this output can help you track down the problem with this program. Modify the grammar to match the grammar in the corresponding sample program (see "Evaluating Arithmetic Expressions" on page 948) while still printing out each production that this program processes. Run the result and include the output in your lab report.

```
; EX16_1a.asm
;
; A simple floating point calculator that demonstrates the use of the
; UCR Standard Library pattern matching routines.  Note that this
; program requires an FPU.

                 .xlist
                 .386
                 .387
                 option      segment:use16
                 include     stdlib.a
                 includelib  stdlib.lib
                 matchfuncs
                 .list

; If the symbol "DEBUG" is defined, then call the MatchSP routine
; to do stack overflow checking.  If "DEBUG" is not defined, just
; call the sl_Match2 routine directly.
```

```
DEBUG           =       0           ;Define for debugging.

                ifdef   DEBUG
MatchPat        textequ <MatchSP>
                else
MatchPat        textequ <sl_Match2>
                endif

dseg            segment para public 'data'

; The following is a temporary used when converting a floating point
; string to a 64 bit real value.

CurValue        real8   0.0


; A Test String:

TestStr         byte    "5+2-(3-1)",0


; Grammar for simple infix -> postfix translation operation:
; Semantic rules appear in braces.
;
; NOTE: This code has a serious problem.  The first production
; is left recursive and will generate an infinite loop.
;
; E -> E+T {print result} | T {print result}
; T -> <constant> {fld constant} | (E)
;
; UCR Standard Library Pattern that handles the grammar above:

; An expression consists of an "E" item followed by the end of the string:

Expression      pattern {MatchPat,E,,EndOfString}
EndOfString     pattern {EOS}


; An "E" item consists of an "E" item optionally followed by "+" or "-"
; and a "T" item (E -> E+T | T):

E               pattern {PatPrint,EMsg,,E2}
EMsg            byte    "E->E+T | T",cr,lf,0

E2              pattern {MatchPat, E,T,Eplus}
Eplus           pattern {MatchChar, '+', T, epPlus}
epPlus          pattern {DoFadd,,,E3}
E3              pattern {PatPrint,EMsg3}
EMsg3           byte    "E->E+T",cr,lf,0


; A "T" item is either a floating point constant or "(" followed by
; an "E" item followed by ")".
;
; The regular expression for a floating point constant is
;
;       [0-9]+ ( "." [0-9]* | ) ( ((e|E) (+|-| ) [0-9]+) | )
;
; Note: the pattern "Const" matches exactly the characters specified
;       by the above regular expression.  It is the pattern the calc-
;       ulator grabs when converting a string to a floating point number.

Const           pattern {MatchPat, ConstStr, 0, FLDConst}
ConstStr        pattern {MatchPat, DoDigits, 0, Const2}
Const2          pattern {matchchar, '.', Const4, Const3}
Const3          pattern {MatchPat, DoDigits, Const4, Const4}
Const4          pattern {matchchar, 'e', const5, const6}
Const5          pattern {matchchar, 'E', Succeed, const6}
Const6          pattern {matchchar, '+', const7, const8}
```

```
Const7          pattern  {matchchar, '-', const8, const8}
Const8          pattern  {MatchPat, DoDigits}

FldConst        pattern  {PushValue,,,ConstMsg}
ConstMsg        pattern  {PatPrint,CMsg}
CMsg            byte     "T->const",cr,lf,0

; DoDigits handles the regular expression [0-9]+

DoDigits        pattern  {Anycset, Digits, 0, SpanDigits}
SpanDigits      pattern  {Spancset, Digits}

; The S production handles constants or an expression in parentheses.

T               pattern  {PatPrint,TMsg,,T2}
TMsg            byte     "T->(E) | const",cr,lf,0

T2              pattern  {MatchChar, '(', Const, IntE}
IntE            pattern  {MatchPat, E, 0, CloseParen}
CloseParen      pattern  {MatchChar, ')',,T3}

T3              pattern  {PatPrint,TMsg3}
TMsg3           byte     "T->(E)",cr,lf,0

; The Succeed pattern always succeeds.

Succeed         pattern  {DoSucceed}

; We use digits from the UCR Standard Library cset standard sets.

                include  stdsets.a

dseg            ends

cseg            segment  para public 'code'
                assume   cs:cseg, ds:dseg

; Debugging feature #1:
; This is a special version of sl_Match2 that checks for
; stack overflow.  Stack overflow occurs whenever there
; is an infinite loop (i.e., left recursion) in a pattern.

MatchSP         proc    far
                cmp      sp, offset StkOvrfl
                jbe      AbortPgm
                jmp      sl_Match2

AbortPgm:       print
                byte     cr,lf,lf
                byte     "Error: Stack overflow in MatchSP routine.",cr,lf,0
                ExitPgm
MatchSP         endp

; PatPrint- A debugging aid.  This "Pattern matching function" prints
; the string that DS:SI points at.

PatPrint        proc     far
                push     es
                push     di
                mov      di, ds
                mov      es, di
                mov      di, si
                puts
                mov      ax, di
                pop      di
                pop      es
                stc
                ret
PatPrint        endp
```

```
; DoSucceed matches the empty string.  In other words, it matches anything
; and always returns success without eating any characters from the input
; string.

DoSucceed        proc     far
                 mov      ax, di
                 stc
                 ret
DoSucceed        endp

; DoFadd - Adds the two items on the top of the FPU stack.

DoFadd           proc     far
                 faddp    st(1), st
                 mov      ax, di                ;Required by sl_Match
                 stc                            ;Always succeed.
                 ret
DoFadd           endp

; PushValue-     We've just matched a string that corresponds to a
;                floating point constant.  Convert it to a floating
;                point value and push that value onto the FPU stack.

PushValue        proc     far
                 push     ds
                 push     es
                 pusha
                 mov      ax, dseg
                 mov      ds, ax

                 lesi     Const         ;FP val matched by this pat.
                 patgrab                ;Get a copy of the string.
                 atof                   ;Convert to real.
                 free                   ;Return mem used by patgrab.
                 lesi     CurValue      ;Copy floating point accumulator
                 sdfpa                  ; to a local variable and then
                 fld      CurValue      ; copy that value to the FPU stk.

                 popa
                 mov      ax, di
                 pop      es
                 pop      ds
                 stc
                 ret
PushValue        endp



; The main program tests the expression evaluator.

Main             proc
                 mov      ax, dseg
                 mov      ds, ax
                 mov      es, ax
                 meminit

                 finit                  ;Be sure to do this!
                 fwait

                 lesi     TestStr
                 puts                   ;Print the expression

                 ldxi     Expression
                 xor      cx, cx
                 match
                 jc       GoodVal
                 printff
                 byte     " is an illegal expression",cr,lf,0
                 ret
```

```
GoodVal:fstp     CurValue
                 printff
                 byte     " = %12.6ge\n",0
                 dword    CurValue

Quit:            ExitPgm
Main             endp
cseg             ends

sseg             segment  para stack 'stack'
                 word     64 dup (?)              ;Buffer for stack overflow
StkOvrfl         word     ?                       ;Stack overflow if drops
stk              db       1024 dup ("stack   ")   ; below StkOvrfl.
sseg             ends

zzzzzzseg        segment  para public 'zzzzzz'
LastBytes        db       16 dup (?)
zzzzzzseg        ends
                 end      Main
```

16.10 Programming Projects

1) Modify the program in Section 16.8.3 (Arith2.asm on the companion CD-ROM) so that it includes some common trigonometric operations (sin, cos, tan, etc.). See the chapter on floating point arithmetic to see how to compute these functions. The syntax for the functions should be similar to “sin(E)” where “E” represents an arbitrary expression.

2) Modify the (English numeric input problem in Section 16.8.1 to handle negative numbers. The pattern should allow the use of the prefixes “negative” or “minus” to denote a negative number.

3) Modify the (English) numeric input problem in Section 16.8.1 to handle four byte unsigned integers.

4) Write your own “Adventure” game based on the programming techniques found in the “Madventure” game in Section 16.8.5.

5) Write a “tiny assembler” for the modern version of the x86 processor using the techniques found in Section 16.8.4.

6) Write a simple “DOS Shell” program that reads a line of text from the user and processes valid DOS commands found on that line. Handle at least the DEL, RENAME, TYPE, and COPY commands. See “MS-DOS, PC-BIOS, and File I/O” on page 699 for information concerning the implementation of these DOS commands.

16.11 Summary

This has certainly been a long chapter. The general topic of pattern matching receives insufficient attention in most textbooks. In fact, you rarely see more than a dozen or so pages dedicated to it outside of automata theory texts, compiler texts, or texts covering pattern matching languages like Icon or SNOBOL4. That is one of the main reasons this chapter is extensive, to help cover the paucity of information available elsewhere. However, there is another reason for the length of this chapter and, especially, the number of lines of code appearing in this chapter – to demonstrate how easy it is to develop certain classes of programs using pattern matching techniques. Could you imagine having to write a program like Madventure using standard C or Pascal programming techniques? The resulting program would probably be longer than the assembly version appearing in this chapter! If you are not impressed with the power of pattern matching, you should probably reread this chapter. It is very surprising how few programmers truly understand the theory of pattern matching; especially considering how many program use, or could benefit from, pattern matching techniques.

This chapter begins by discussing the theory behind pattern matching. It discusses simple patterns, known as *regular languages*, and describes how to design *nondeterministic* and *deterministic finite state automata* – the functions that match patterns described by *regular expressions*. This chapter also describes how to convert NFAs and DFAs into assembly language programs. For the details, see

- "An Introduction to Formal Language (Automata) Theory" on page 883
- "Machines vs. Languages" on page 883
- "Regular Languages" on page 884
- "Regular Expressions" on page 885
- "Nondeterministic Finite State Automata (NFAs)" on page 887
- "Converting Regular Expressions to NFAs" on page 888
- "Converting an NFA to Assembly Language" on page 890
- "Deterministic Finite State Automata (DFAs)" on page 893
- "Converting a DFA to Assembly Language" on page 895

Although the regular languages are probably the most commonly processed patterns in modern pattern matching programs, they are also only a small subset of the possible types of patterns you can process in a program. The *context free languages* include all the regular languages as a subset and introduce many types of patterns that are not regular. To represent a context free language, we often use a *context free grammar*. A CFG contains a set of expressions known as *productions*. This set of productions, a set of *nonterminal symbols*, a set of *terminal symbols*, and a special nonterminal, the *starting symbol*, provide the basis for converting powerful patterns into a programming language.

In this chapter, we've covered a special set of the context free grammars known as LL(1) grammars. To properly encode a CFG as an assembly language program, you must first convert the grammar to an LL(1) grammar. This encoding yields a *recursive descent predictive parser.* Two primary steps required before converting a grammar to a program that recognizes strings in the context free language is to *eliminate left recursion* from the grammar and *left factor* the grammar. After these two steps, it is relatively easy to convert a CFG to an assembly language program.

For more information on CFGs, see

- "Context Free Languages" on page 900
- "Eliminating Left Recursion and Left Factoring CFGs" on page 903
- "Converting CFGs to Assembly Language" on page 905
- "Some Final Comments on CFGs" on page 912

Sometimes it is easier to deal with regular expressions rather than context free grammars. Since CFGs are more powerful than regular expressions, this text generally adopts grammars whereever possible However, regular expressions are generally easier to work with (for simple patterns), especially in the early stages of development. Sooner or later, though, you may need to convert a regular expression to a CFG so you can combine it with other components of the grammar. This is very easy to do and there is a simple algorithm to convert REs to CFGs. For more details, see

- "Converting REs to CFGs" on page 905

Although converting CFGs to assembly language is a straightforward process, it is very tedious. The UCR Standard Library includes a set of pattern matching routines that completely eliminate this tedium and provide many additional capabilities as well (such as automatic backtracking, allowing you to encode grammars that are not LL(1)). The pattern matching package in the Standard Library is probably the most novel and powerful set of routines available therein. You should definitely investigate the use of these routines, they can save you considerable time. For more information, see

- "The UCR Standard Library Pattern Matching Routines" on page 913
- "The Standard Library Pattern Matching Functions" on page 914

One neat feature the Standard Library provides is your ability to write customized pattern matching functions. In addition to letting you provide pattern matching facilities missing from the library, these pattern matching functions let you add *semantic rules* to your grammars. For all the details, see

- "Designing Your Own Pattern Matching Routines" on page 922
- "Extracting Substrings from Matched Patterns" on page 925
- "Semantic Rules and Actions" on page 929

Although the UCR Standard Library provides a powerful set of pattern matching routines, its richness may be its primary drawback. Those who encounter the Standard Library's pattern matching routines for the first time may be overwhelmed, especially when attempting to reconcile the material in the section on context free grammars with the Standard Library patterns. Fortunately, there is a straightforward, if inefficient, way to translate CFGs into Standard Library patterns. This technique is outlined in

- "Constructing Patterns for the MATCH Routine" on page 933

Although pattern matching is a very powerful paradigm that most programmers should familiarize themselves with, most people have a hard time seeing the applications when they first encounter pattern matching. Therefore, this chapter concludes with some very complete programs that demonstrate pattern matching in action. These examples appear in the section:

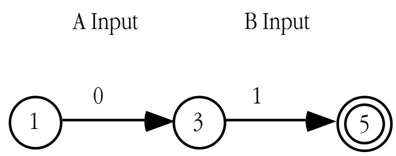
- "Some Sample Pattern Matching Applications" on page 935

16.12 Questions

1) Assume that you have two inputs that are either zero or one. Create a DFA to implement the following logic functions (assume that arriving in a final state is equivalent to being true, if you wind up in a non-accepting state you return false)

a) OR b) XOR c) NAND d) NOR

e) Equals (XNOR) f) AND



Example, A<B

2) If r, s, and t are regular expressions, what strings with the following regular expressions match?

a) $r*$ b) $r\,s$ c) r^+ d) $r \mid s$

3) Provide a regular expression for integers that allow commas every three digits as per U.S. syntax (e.g., for every three digits from the right of the number there must be exactly one comma). Do not allow misplaced commas.

4) Pascal real constants must have at least one digit before the decimal point. Provide a regular expression for FORTRAN real constants that does not have this restriction.

5) In many language systems (e.g., FORTRAN and C) there are two types of floating point numbers, single precision and double precision. Provide a regular expression for real numbers that allows the input of floating point numbers using any of the characters [dDeE] as the exponent symbol (d/D stands for double precision).

6) Provide an NFA that recognizes the mnemonics for the 886 instruction set.

7) Convert the NFA above into assembly language. Do not use the Standard Library pattern matching routines.

8) Repeat question (7) using the Standard Library pattern matching routines.

9) Create a DFA for Pascal identifiers.

10) Convert the above DFA to assembly code using straight assembly statements.

11) Convert the above DFA to assembly code using a state table with input classification. Describe the data in your classification table.

12) Eliminate left recursion from the following grammar:

```
Stmt   →   if expression then Stmt endif
       |   if expression then Stmt else Stmt endif
       |   Stmt ; Stmt
       |   ε
```

13) Left factor the grammar you produce in problem 12.

14) Convert the result from question (13) into assembly language without using the Standard Library pattern matching routines.

15) Convert the result from question (13) in assembly language using the Standard Library pattern matching routines.

16) Convert the regular expression obtained in question (3) to a set of productions for a context free grammar.

17) Why is the ARB matching function inefficient? Describe how the pattern (ARB "hello" ARB) would match the string "hello there".

18) Spancset matches zero or more occurrences of some characters in a character set. Write a pattern matching function, callable as the first field of the pattern data type, that matches one or more occurrences of some character (feel free to look at the sources for spancset).

19) Write the matchichar pattern matching function that matches an individual character regardless of case (feel free to look at the sources for matchchar).

20) Explain how to use a pattern matching function to implement a semantic rule.

21) How would you extract a substring from a matched pattern?

22) What are *parenthetical patterns*? How to you create them?