

158
77

**DESIGN AND CONSTRUCTION OF A
PROTOTYPE GENERAL PURPOSE SYNTAX-AWARE TEXT EDITOR**

by

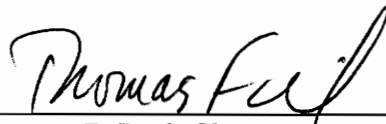
Joseph Lewis Faber III

Project and Report submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of
MASTER OF SCIENCE

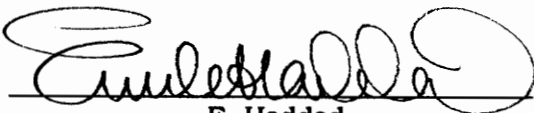
in

Computer Science

APPROVED:



T. Reid, Chairman



E. Haddad



W. May

May, 1991

Blacksburg, Virginia

c.2

LD
5655
V851
1991
F324
c.2

DESIGN AND CONSTRUCTION OF A
PROTOTYPE GENERAL PURPOSE SYNTAX-AWARE TEXT EDITOR

by

Joseph Lewis Faber III

Committee Chairman: Thomas Reid
Computer Science

(ABSTRACT)

A comparison is made between traditional text editors, syntax-directed editors and syntax-aware editors. The design for a prototype general purpose syntax-aware editor is presented. This editor utilizes a user-written language specification to continuously parse the text buffer. Errors are indicated to the user non-intrusively by modifying the display color of the text. Error messages are presented to the user as the cursor is placed over portions of the text which are in error. A description of the implementation of the editor, and a critique of its usefulness is included.

TABLE OF CONTENTS

1 Introduction	1
Traditional Compilers	1
Syntax-Directed Editors	2
Syntax-Aware Editors	2
Dynamic Syntax Specification	4
Scope	5
Report Overview	5
2 Basic Editing	7
Features	7
Data Structure	8
Implementation	8
3 Lexical Analysis	9
Scanning	10
Implementation	15
Token Use in the Editor	16
4 Syntactic Analysis	18
Extended Backus-Naur Form (EBNF)	18
Implementation	20
Error Detection and Recovery	21

5 Initial Semantic Analysis	23
Symbol Table Implementation	23
Declarations	23
6 Translation	26
Production Trees	26
Creation of the Production Tree	31
Select Sets	34
Follow-up	36
7 Syntactic Analysis of the Target Language	37
Incremental Parsing	37
Error Detection and Recovery	38
Declaration Checking	40
8 Conclusions	41
Critique	41
Usefulness	42
Further Work	43
Appendix A: Source Code	45

1 Introduction

1.1 Traditional Compilers

The detection of compilation errors in a computer program has traditionally been considered a batch process. Users enter their programs using a general purpose editor and then, when they believe the program to be syntactically correct, they compile it using either an embedded or a stand-alone compiler. If the source program contains errors, they return to the editor, modify the program and try again.

While this process has served well, it is not without inefficiency. The most opportune time for the correction of an erroneous statement is clearly when that statement is initially entered: the programmer is still focused on that region of the program. If error detection is performed only when the programmer believes that the program is "complete", there will probably be a significant time lapse between the entry of an erroneous statement and the determination that it is in error. This time lapse causes the programmer to "lose context." The details of the statement's function and environment must be remembered before the programmer can correct the error.

This problem has been addressed in several ways. Systems like Borland's Turbo Pascal feature an integrated compiler which automatically locates the erroneous statement in the source file. This, at least, relieves the programmer from having to search for the offending line. It does not, however, assist in the recovery of the "context" in which the line was entered.

1.2 Syntax-Directed Editors

Considerable work has been done on syntax-directed editors. These systems use a knowledge of the syntax of the language to ensure that statements entered are valid. The user's program is stored not as a text file, but as a derivation tree. This allows the editor to provide a host of language specific functions which generic editors are unable to match. These systems can automatically display the user's program in a "pretty-printed" style according to a set of rules embedded in the language specification. Their selection and modification functions are based on language entities (tokens, statements, procedures, etc.) rather than characters. This permits the user to work with pieces of the program which have a higher level of abstraction and are consequently more "natural".

The major complaint against these systems is that their "ruthless efficiency" at ensuring that the source program is syntactically correct adversely affects their ease of use. Some of these systems allow entry of program statements only through a menu system, thus eliminating the possibility that the user will enter an incorrect statement. Others allow local entry of text directly from the keyboard, but insist that the text be syntactically valid before including it in the source program. Both of these methods *force* the user to take additional time to correct these errors.

1.3 Syntax-Aware Editors

A syntax-aware editor is an editor which recognizes a correct program, but does not require that the entered program be correct. Simply put, it is non-intrusive. The user may enter *any* program text without regard for correctness. The editor should attempt to recognize the program as it is being entered, but should not force the user to correct any

errors that are detected. Rather, the editor should alter the text being displayed to indicate the portions of the program which appear to be incorrect. In this way, the editor supports users who wish to ensure that the program is correct at every step, but does not penalize users who prefer to postpone correction of their errors.

Such an editor could retain many of the desirable features of syntax-directed editors. The ability to view the text as a series of tokens or syntactic entities could be built into these editors in much the same way as it is built into syntax-directed editors. There are, however, several factors which complicate any syntax-aware design.

In a syntax-directed editor, the primary data structure is the derivation tree -- an abstract structure which contains the syntactic structure of the program being edited. The text of the program is secondary as it can be regenerated at any time from the derivation tree. Indeed, most systems maintain *just* the derivation tree during the editing session, recreating portions of text as they are needed to display the program to the user. This data structure is, by definition, a syntactically correct program. The editor will simply not allow any modifications which would cause the program to become invalid. In contrast, a basic feature of syntax-aware editors is their ability to work with a syntactically incorrect file. For this reason, these editors use the program *text* as their primary data structure. Derivation trees are still built to verify the program's correctness, but these trees must be sufficiently flexible to handle the errors which will be present in the program text. While the information necessary to support syntax-directed type features is present, care must be taken in the implementation of these functions to ensure that the results are meaningful when the input text does not parse correctly.

1.4 Dynamic Syntax Specification

While the value of having a syntax-directed or syntax-aware editor for a specific language is clear, it seems wasteful to build a separate editor for each language (or variant thereof) which is to be edited. Each editor would possess the same core text editing facilities, and would probably share many of the advanced editing features. Each would contain modules for scanning the program text and breaking it into tokens, although the specific rules by which this is accomplished might vary. Only in the recognition phase does the difference between the editors become apparent. Clearly this "family" of editors would be more alike than different.

One solution is to build a program "generator" which accepts a language specification (in Backus-Naur Form or some equivalent) and produces the code for an editor which will recognize the specified language. This has the advantage of allowing the "family" of editors to literally share the same code. Only those portions of the editor which are directly related to the lexical and syntactic rules of the target language need be generated. This approach is the basis for the Cornell Synthesizer Generator, a popular system which produces syntax-directed editing environments from an abstract syntax and semantic specification.

An alternative to this strategy is to build a single general purpose editor which accepts language specifications *at run time*. Such a system relieves the user from the details of the construction or compilation of the editor. If a change is desired in a target language, the user need change only the specification and the system will take care of the details. Additionally, such an editor could allow the user to edit programs in different languages simultaneously. Indeed, the tasks of recognizing and translating the language specification are in themselves parsing tasks; a general purpose editor could incorporate

code which is used in both the "startup" phase and the "user" phase. The disadvantages with this scheme are that a general purpose editor may be larger and more complex than a specialized editor for any given language, and that the postponement of language specification to run time may force the imposition of arbitrary limits on the size and nature of some data structures.

1.5 Scope

The focus of this project is the creation of a prototype general purpose syntax-aware editor. The editor will accept a language specification written in Extended Backus-Naur Form (EBNF) and parse user text in accordance with those rules. It will attempt to isolate errors in the user text and will indicate those errors by changing the color of the displayed text. The emphasis of the design is on the language recognition features of the editor, and not on the text editing capabilities. For this reason, the editor will support only a sparse set of editing features.

1.6 Report Overview

This report details the design and construction of the editor. The development was completed in several discrete steps and each will be the focus of one of the following sections. Section Two discusses the creation of the "bare bones" editor which serves as the basis of the system. Section Three covers the addition of a lexical analyzer to the editor. Section Four documents the hand-coded EBNF parser which performs initial syntax checking. Section Five addresses the incorporation of a symbol table with declaration checking. Section Six examines the translation of an EBNF specification into

extended tree form, and Section Seven describes the use of that structure to recognize user text. Finally, Section Eight presents some conclusions about the work performed. The following diagram shows the general architecture of the system.

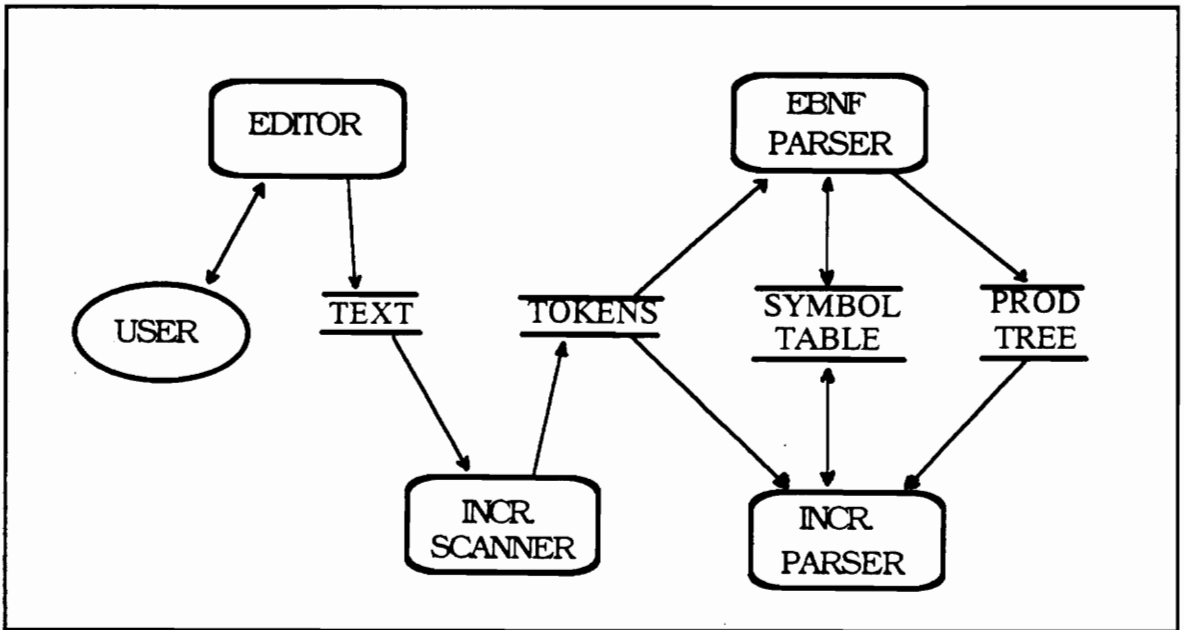


Figure 1: Data Flow Diagram

2 Basic Editing

This section discusses the creation of the basic editor to which syntax-awareness will be added. While there are many excellent existing editing systems available, the learning curve associated with mastering the internals of one of these systems would have prohibited the completion of the project in the allotted time. For that reason, I decided to build a new "bare-bones" editor to serve as the skeleton for this project. This allowed the data structures of the editor to be tailored to the specific requirements of the project, and permitted easy extensions to the editor during the course of the development.

2.1 Features

Since the editor was developed in Turbo Pascal, I chose to model the basic interface after the embedded editor within that development environment. With the constant switching between the Turbo Editor and the new editor being developed, any differences between the interfaces would only have created confusion. The specific features included in the editor were restricted to the bare minimum in the interest of completing the project in a timely fashion. A more complete set of features should certainly be included in any production version of this program, but the minimal set will suffice to demonstrate the syntax-aware features which are the focus of this project. Supported features include the following:

- cursor movement using the arrow keys, Home, End, PgUp and PgDn
- insert/overstrike modes toggled by the Insert key
- line insertion using the Enter key in insert mode
- line deletion using the Ctrl Y key

- character deletion using the Delete key or the BackSpace key
- text entry using the normal keyboard
- file read and write operations

The edit buffer is restricted to 80 characters per line, and contains only as many lines as will fit in the available memory.

2.2 Data Structure

The edit buffer is simply a doubly linked list of 80 character records. Pointers are maintained to the top of the list, to the first line on the current page, and to the line on which the cursor is presently positioned. Since no line numbering is maintained, insertion and deletion of lines of text are trivial. This structure is obviously not intended to optimize either the space requirements or performance of the editor. It merely provides a simple basis on which to build a set of complicated functions.

2.3 Implementation

While the development machine for this editor was an ISA 386, the program should run on any IBM-PC compatible machine. In order to improve performance, I have taken full advantage of the easy accessibility of the video memory, and the console I/O features of Turbo Pascal.

3 Lexical Analysis

Lexical analysis is the process whereby a stream of raw text is converted into a series of lexemes (or tokens) which are valid in the target language. In traditional batch compilers, this step is typically viewed as a kind of high-level input routine which is called by the parser whenever it requires the next token in the input stream. In many syntax-directed editors, there is little lexical analysis, per se, because the input to the editor takes the form of menu choices. In a syntax-aware editor, however, the role of the lexical analyzer (or Scanner) is quite important.

Consider a program (user text) which contains an error near its beginning which radically alters the syntactic derivation of the remainder of the program. If the scanner is implemented simply as an input filter (as in the traditional batch compiler), when the user corrects the error, the parser will force the lexical analyzer to re-scan nearly the entire program. This is quite inefficient since most of the user text (and by extension, most of the tokens) will not have changed. The answer is to retain the output of the scanner, and only re-scan a line when it has been altered. This will relieve the editor from continually re-scanning the input text.

Thus, in this editor, an input buffer of tokens is maintained in parallel with the text buffer. Whenever the text buffer changes, the scanner is called to recreate the tokens for the line(s) which have changed. The remainder of this section details the addition of a scanner to the basic editor created in Section Two.

3.1 Scanning

The design of this scanner is driven by one overriding requirement: it must be sufficiently general to handle both EBNF and whatever target language is specified. To accomplish this I created a "Language Record" which contains the information needed by the scanner to correctly convert user text into tokens. At program startup, the Language Record is loaded with the specification for EBNF, causing the generation of tokens appropriate to that language. Later, (in Section 6) when the EBNF is translated into a production tree and the editor's target language changes, the Language Record will be reloaded with the specification for the new target language. This record contains the delimiter characters for comments, strings and directives, a list of the simple tokens (for example, "==" and "{") and keywords (for example, "IF" and "THEN") used in the language, and the set of characters which may not appear in an identifier.

Using this information, the scanner converts each line of user text into a series of tokens. Each token contains an eight-bit code called a TokenID which identifies the type of the token. Each simple token has a unique TokenId between 1 and 199. Certain values above 199 identify specific classes of tokens as detailed below. In addition, each token contains the position within the text line of the first and last characters which comprise the lexeme. Except for whitespace (blanks and TABs), every character within the user text should be "covered" by a token.

3.1.1 Delimited Sequences: Comments, Strings and Directives

These three token types have the highest "precedence" in the scanner. That is, a delimited sequence of characters which is determined to fall into one of these three types will be accepted as that type of token *regardless* of the characters which are contained

within it. Thus, a delimited sequence may contain characters which would normally be recognized as a simple token, an identifier, or a numeric literal. By virtue of their positioning between the delimiters, they lose their normal significance and become part of the delimited sequence. Further, strings take precedence over directives, and comments take precedence over both strings and directives.

Directives are an extension to EBNF which allow "messages" to be passed directly to the translator during the conversion of the EBNF language specification. These messages allow the user to specify aspects of the target language which are not a part of the syntactic specification. The recognized directives are discussed in detail in Section 6. The scanner is not concerned with the contents of the directives, it simply detects their presence and generate a Directive token. Directives in EBNF are delimited by dollar signs. (For example, "\$anything\$" would be recognized as a directive.)

Strings are sequences of characters and blanks delimited by a particular character (usually a double or single quote). Two adjacent occurrences of the delimiter character within a string are taken as a literal occurrence of the delimiter within the string rather than the end of one string and the start of another. There is no provision in this scanner for strings which cross line boundaries; any string delimiter which goes unmatched for the remainder of the input line will be regarded as an error. In this case, the scanner will generate an Illegal String token which covers the characters from the start delimiter through the end of the line.

Comments, on the other hand, can span multiple lines as long as an end comment delimiter is defined for the language. The scanner will generate a separate token for each line of a multi-line comment. For languages which do not have an end comment delimiter, comments are delimited on the right by an end of line. EBNF falls into the latter category; its start comment delimiter is "#" and it has no end comment delimiter.

3.1.2 Simple Tokens

A simple token is a sequence of one or more characters which, when encountered in the text, will be recognized as having special meaning and will cause the generation of a token with a unique TokenID. These sequences can include both alphabetic and non-alphabetic characters. The simple tokens which begin with non-alphabetic characters typically do not require whitespace before them. For example, in Pascal, the sequence "Counter:=1" will be translated correctly into three tokens: "Counter" (an identifier), "!=" (the simple token), and "1" a numeric literal. There is no space required between the "Counter" and the "!=", nor is any required between the "!=" and the "1". This property is accomplished by maintaining a set (called the "Simple Set") of the start characters of all simple tokens. This set is used in addition to the whitespace characters to signal the end of an identifier. Keywords (simple tokens which start with alphabetic characters) do not have this property. Thus the sequence "CounterIF" would be treated as a single nine character identifier even though "IF" is a valid keyword in Pascal. Although keywords are stored with the non-alphabetic simple tokens, their recognition is more like that of an identifier. They will be discussed below.

There is another property associated with the non-alphabetic simple tokens: one simple token can be a prefix of another. Again, in Pascal, both ":" and "!=" are simple tokens. On detecting a ":", the scanner must not simply generate a ":" token, but must first test the next character to ensure that it is not an "=". In general, if the next input character, c , is a member of the Simple Set, and the longest simple token known to begin with c consists of N characters, then the scanner must capture the next $N-1$ contiguous characters in the text (or the remainder of the line if there are fewer than $N-1$ characters left) and attempt to match the entire sequence to a known simple token. (In this program, N (the maximum length of a simple token) will be considered constant for a language

instead of a property associated with each member of the Simple Set.) If that fails, the scanner truncates the sequence by removing the rightmost character and tries again. This process is repeated until a Simple Token is matched, or the scanner runs out of characters to truncate. When this error occurs, the scanner generates an Illegal Character token which covers only the first character, and scanning resumes with the next character.

3.1.3 Numeric Literals

For the purposes of this program, Numeric Literals are defined as sequences of digits with zero or one embedded decimal point. Sequences without a decimal point are considered Integer Literals; sequences with a decimal are considered Real Literals. The scanner does not attempt to store the value of the numbers, it merely recognizes the type and creates the appropriate token. The only complication in recognizing numerics is the common use of the decimal point character (".") as a simple token, or as the first character of a simple token (for example ".."). When the scanner encounters the first "." during recognition of a numeric, it will first determine if the "." is the beginning of a multi-character simple token. If so, the numeric will be recognized without the trailing "." as an integer. This allows the correct recognition of sequences like "3..5", where ".." is a simple token. If, on the other hand, the "." does not belong to a multi-character simple token (even if "." is a valid single character simple token), the "." will be included in the numeric token, and it will be recognized as a real. (This is a departure from the syntax of Pascal where, in order for a "." to be considered part of a real, the character immediately following it must be a digit. Thus, Pascal would not consider "3." to be a real literal. This scanner will.) Note that this rule only applies to the first "." encountered during the recognition of a particular numeric literal. Thus, the sequence "2.3.4.5" would be interpreted as the real "2.3" followed by the simple token "." followed by the real "4.5".

Note that EBNF has no need to recognize Numeric Literals. In fact, free standing numerics within a language specification should probably be regarded as terminal symbols. For this reason, a facility will be provided to disable the recognition of Numeric Literals when desired. When this feature is active, numerics will be treated as identifiers (see below), and any embedded decimal points will have no special meaning. Thus, the user could define a language where the sequence "2.3.4.5" should be recognized as the identifier "2" followed by the simple token ".", the identifier "3", the simple token ".", etc.

3.1.4 Identifiers and Keywords

Simply put, an identifier is any contiguous sequence of characters which is not otherwise recognizable by the scanner. Specifically, legal identifiers may not begin with a numeral (unless Numeric Literals are disabled); they may not contain any character which is in the Simple Set; and they may not contain any embedded whitespace. There is no specific restriction on starting an identifier with a non-alphabetic character. Thus, in EBNF, "<Expression>" is a valid identifier since the characters "<" and ">" are not contained in the Simple Set. A keyword is a sequence which appears to be an identifier, but which matches an entry in the Simple Token list. Keywords (like the other simple tokens) each have a unique TokenID.

3.1.5 End Of File Token

At the end of the last line of text, the scanner will always place a special End of file (EOF) token. This token will allow the parser to recognize when the end of the input text has been encountered without the need for "artificial" EOF tokens like the "_!_" commonly used in EBNF. Note that for compatibility with existing EBNF specs, the "_!_" is still required in the variant of EBNF recognized by this editor.

3.2 Implementation

The scanner is a procedure which accepts a pointer to a line of text and regenerates the tokens associated with that line. The tokens are represented as a singly linked list which is pointed to by the line record. In addition, each token record contains a pointer back to the source line from which it was generated. Each token record also contains an error code field which can take on values indicating different types of lexical errors (Illegal String, Illegal Character, Illegal Directive).

The scanner was integrated into the editor by way of an intermediary procedure called ScanFile which is called whenever the editor detects a break of one second or more in the user's keystrokes. ScanFile cycles through each of the text line records and checks whether that line has been changed. (The editor marks a boolean flag on each line record which it "touches".) For each line which has changed, ScanFile calls the scanner to regenerate the token sequence attached to that line.

Because comments can span multiple lines, ScanFile must perform another function: it must keep track of the "state" of the scanner each time it completes a line. Since the scanner only maintains control for a line at a time, ScanFile must tell it at each call whether it was in the middle of a comment at the end of the previous line. Moreover, since the previous line may not have required re-scanning, this information must be stored along with each line. This is accomplished by adding a state variable to each line record. Each time a line is re-scanned, the terminating state of the scanner is stored in the state variable of the following line. Thus, when it is necessary to re-scan a line, ScanFile "remembers" the correct starting state for the scanner by examining the state variable attached to the line record.

In some cases, it will be necessary to re-scan a text line even when it has *not* changed. Consider a program which contains a three-line comment. If the user deletes the start comment delimiter from the first line, the second and third lines must be re-scanned to recognize any tokens which were previously "hidden" because they were embedded in a comment. This situation can be detected by examining the state variables. If, after re-scanning a line, ScanFile detects that the current state of the scanner does not match the stored state on the next line, ScanFile must also re-scan that next line whether or not it has changed. This logic must be reapplied to each line until a line is found which has not changed *and* whose state variable matches the terminating state of the scanner after the re-scan of the previous line.

3.3 Token Use in the Editor

Even before the addition of a parser to the system, the presence of tokens allows the improvement of the display capabilities of the editor. By examining the tokens in addition to the text when redisplaying the screen, we can change the display color of certain pieces of text according to their TokenId. In the prototype, all Comments and Directives will be displayed as dim (LightGray on Blue), all literals and identifiers will be displayed as bright (White on Blue), and all simple tokens will be displayed in normal text (Yellow on Blue). In addition, tokens which indicate an error condition (Illegal String, Directive or Character) will be displayed in Red. Further, when the cursor is placed within the boundaries of a token with an error condition, an appropriate error message will appear on the status line at the bottom of the screen.

The existence of tokens also allows an improvement to the cursor movement. By traversing the tokens attached to the text buffer, the editor now supports keystrokes which

move the cursor to the Next Token, Previous Token, Next Token with an error, or Previous Token with an error.

4 Syntactic Analysis

This section describes the addition of a capability to recognize valid EBNF language specifications. Because of the requirement to translate these specifications into production trees, this capability is being implemented separately from the generalized parser which will serve to recognize text in user specified languages.

4.1 Extended Backus-Naur Form (EBNF)

Languages are described to the editor using a variant of Extended Backus-Naur Form (EBNF). This meta-language is similar to Backus-Naur Form (BNF), but also supports the specification of optional sequences, repeating sequences, and grouped sequences. The variations used in this project include the addition of directives to the allowable symbols which may appear on the right-hand side of a production, and the removal of any syntactic distinction between unquoted terminals and non-terminals. Both of these symbols are parsed as identifiers. The addition of the symbol table in the next section will allow the editor to distinguish between terminals (which do not appear on the left-hand side of a production) and non-terminals (which do). Figure 2 shows the language specification for EBNF expressed in traditional BNF, and Figure 3, the same specification in EBNF.

It should be noted that both of these language specifications have the LL(1) property. In general, a specification (or grammar) is considered LL(1) if it permits deterministic left-to-right top-down recognition with a look-ahead of only one symbol. It is this property which permits the construction of a simple recursive descent recognizer for EBNF.

<EBNF>	→	<DIRECTIVES> <PRODUCTIONS> "⌋"
<DIRECTIVES>	→	<directive> <DIRECTIVES>
		ε
<PRODUCTIONS>	→	<identifier> "::=" <EXPRESSION> "." <PRODUCTIONS>
		ε
<EXPRESSION>	→	<TERM> <EXPR_1>
<EXPR_1>	→	" " <TERM> <EXPR_1>
		ε
<TERM>	→	<SYMBOL> <TERM_1>
<TERM_1>	→	<SYMBOL> <TERM_1>
		ε
<SYMBOL>	→	<identifier>
		<string>
		<directive>
		"(" <EXPRESSION> ")"

Figure 2: EBNF specification in Backus–Naur Form

<EBNF>	::=	{ <directive> } { <PRODUCTION> } "⌋"
<PRODUCTION>	::=	<identifier> "::=" <EXPRESSION> "."
<EXPRESSION>	::=	<TERM> { " " <TERM> }
<TERM>	::=	<SYMBOL> { <SYMBOL> }
<SYMBOL>	::=	<identifier>
		<string>
		<directive>
		"(" <EXPRESSION> ")"
		"[" <EXPRESSION> "]"
		"{" <EXPRESSION> "}"
⌋		

Figure 3: EBNF specification in Extended Backus–Naur Form

4.2 Implementation

The EBNF recognizer is implemented as a simple recursive descent parser. Each time ScanFile is called to re-generate tokens, the EBNF parser is also called to check for errors. The EBNF parser is not incremental; each time it is called, it re-traverses the entire stream of tokens. This keeps the complexity of the parser minimal, although it will degrade performance when parsing extremely long EBNF specifications. Since most EBNF specifications are reasonably short, however, this is not expected to be a major problem.

The parser itself is modelled after the EBNF specification in Figure 3, containing a procedure for each of the production rules listed there. Each procedure calls the sub-procedure MATCH for every terminal to be matched, and calls the other production procedures to match non-terminals which appear in the right-hand side. Repetition and alternation are handled using the control structures WHILE and IF. Figure 4 shows the procedure EBNF_prod which implements the <EBNF> production rule.

```

procedure EBNF_prod;
begin
  while nexttoken in [Directive] do
    MATCH (Directive);
  while nexttoken in [Identifier] do
    PROD_prod;
  MATCH (EndMark); { _ }
  MATCH (EOF);
end;

```

Figure 4: Procedure EBNF_prod

Notice the correspondence between the positions of the terminals, non-terminals, and control symbols in the production rule and the positions of the procedure calls in the production procedure. This one-to-one mapping makes the translation of an EBNF language specification to a corresponding recursive descent parser nearly trivial.

The WHILE statements make reference to the variable NEXTTOKEN which contains the TokenID of the next token in the input stream. These expressions test that TokenID against the "Select Set" of the sequence within the construct. (Thus, identifier is the only member of the Select Set for the non-terminal PRODUCTION. This indicates that a PRODUCTION must always begin with an identifier, and any sequence of tokens which does not begin with an identifier cannot be a PRODUCTION.)

4.3 Error Detection and Recovery

The sub-procedure MATCH and its partner CONSUME play the role usually associated with the lexical analyzer in a traditional batch compiler. They are responsible for traversing the stream of tokens attached to the text buffer and passing them, one-at-a-time, to the production procedures of the parser. CONSUME takes care of the physical details of traversing the token stream, filtering out comment tokens and illegal character tokens generated by the scanner. (Since the illegal character tokens are already marked as being in error, it is pointless to further confuse the issue by passing them to the parser where they are incapable of being matched.) MATCH checks the current token against the TokenID passed to it by a production procedure, and marks it in error if it does not match. The error recovery techniques used by MATCH (admittedly quite "simple-minded") are detailed in the following paragraphs.

4.3.1 Insertion

Under most circumstances, MATCH will simply assume that the user has omitted a terminal. It will mark the offending token as "unexpected", but will leave it in place as the current token in the hopes that it will be matched later. Generally speaking, this

strategy is dangerous, since it allows the parser to advance within a production without consuming any of the input. The repetition and option constructs are "safe" since they cannot be entered unless the current token is a member of their Select Set. The alternation construct, on the other hand, requires the parser to choose one of its paths to traverse even if the current token is not a member of any of the Select Sets. If the chosen path contains a recursive reference, the parser could be "led" into an infinite loop. In this particular case, the only alternation construct (in the SYMBOL production) has been given a default path which contains no recursive references (SYMBOL ::= identifier). This renders the strategy "safe" by guaranteeing that the parser will not enter an infinite loop.

4.3.2 Deletion

In some cases, MATCH can determine that the token beyond the current token is a match for the requested TokenID. This is only possible when the two tokens appear on the same line of text and there are no intervening comments or illegal characters. In that situation, MATCH will mark the current token in error, then consume both it and the following token. This local lookahead can significantly improve the quality of the error handling.

4.3.3 Synchronization Tokens

This parser also supports the specification of synchronization (SYNC) tokens. When the parser reaches a state where one of these tokens is required, MATCH will skip over any tokens in the input stream until either a matching token or the EOF token is encountered. This provides a way for the parser to re-synchronize itself after an error.

5 Initial Semantic Analysis

As the last step before implementing the translation of the EBNF language specification into a production tree, we add a symbol table to the editor to keep track of identifiers. As noted in the previous section, the EBNF parser does not syntactically distinguish between non-quoted terminals and non-terminals. Both are recognized simply as identifiers. The translation process, on the other hand, must treat the two symbols quite differently. The task of differentiating between the two falls to the symbol table.

5.1 Symbol Table Implementation

The symbol table, as its name implies, is simply a listing of the symbols (identifiers) which are present within the user text. Each symbol table entry consists of an identifier name and a list of references to the tokens which match that name. Specific identifiers are "registered" as symbols when their tokens are consumed by the parser. A pointer to the appropriate symbol table entry is added to each token which is registered. A token which is being re-scanned is first "unregistered" to remove the reference to it from the symbol table. If the token being "unregistered" is the last occurrence of that symbol, the entire entry is removed from the table as well.

5.2 Declarations

In order to distinguish between terminals and non-terminals, we must introduce the concept of a declaration. In many languages, a facility exists for declaring identifiers

before their use. The distinction between an identifier *use* and an identifier *declaration* can usually be made in the syntax of the language. (In Pascal, identifiers which occur after a VAR and before a PROCEDURE, FUNCTION or BEGIN are being declared. Identifiers which occur between a BEGIN and an END are being used.) Specifically, certain occurrences of <identifier> within the grammar count as declarations, and others do not. Extending this concept to EBNF, we can assert that any identifier token which matches the <identifier> on the left-hand side of a production is a declaration of that identifier. All other identifier tokens are merely uses. The parser can distinguish between these two cases, and can pass that information to the symbol table. This allows the system to keep track of which symbols have been declared (and how many times they have been declared), and which symbols have not.

Since the set of non-terminals is simply the set of identifiers which appear on the left-hand sides of productions, those identifiers which are declared somewhere in the text must be non-terminals. Any other symbols represent un-quoted terminals.

In addition to its value during translation, the editor can also make use of the information contained in the symbol table. Since each non-terminal is restricted to one appearance on the left-hand side of a production, any identifier which violates this rule should be flagged as an error. Indeed, it is trivial to traverse the symbol table to determine which identifiers have been multiply declared. Moreover, *all* occurrences of multiply declared identifiers can be flagged simply by traversing the list of token references attached to the symbol entry.

Further, by noting which identifiers are not declared at all, the editor can alter the display color of all of the associated tokens to help the user distinguish between the non-terminals and the terminals. In the prototype, non-quoted terminals are displayed using

the same color as strings (which will be regarded as quoted terminals). Thus, all terminals appear to be the same color.

6 Translation

So far, we have taken a bare-bones text editor and added to it the ability to recognize a variant of EBNF. The editor detects lexical and syntactic errors in the EBNF specification and reports them by altering the display color of the erroneous text. Using a symbol table, we can also detect one class of semantic error (multiple declarations). In this section, we will complete the semantic analysis of the EBNF by translating it into a production tree and computing Select Sets for each decision point in the tree.

6.1 Production Trees

A production tree is simply an abstract representation of the productions which make up a grammar. Each node represents a terminal, non-terminal or control symbol in the EBNF specification, and the linkages between nodes denote the order of traversal. Although most trees are thought of as having right and left children, in production trees we will speak of the "next node" and the "alternate node". This change in terminology promotes a "horizontal" orientation which more closely mirrors that of the EBNF specification. It should also be noted that, in the strictest sense, the production tree is not a "tree" but a directed graph. As described below, some linkages between the nodes will create cycles. Figure 5 contains an example of a portion of the production tree.

6.1.1 Production Nodes

Each distinct production within a grammar is represented by a Production Node. This node serves as a target for all non-terminal references to that production, providing a pointer to the sub-tree which defines the production. The set of production nodes is

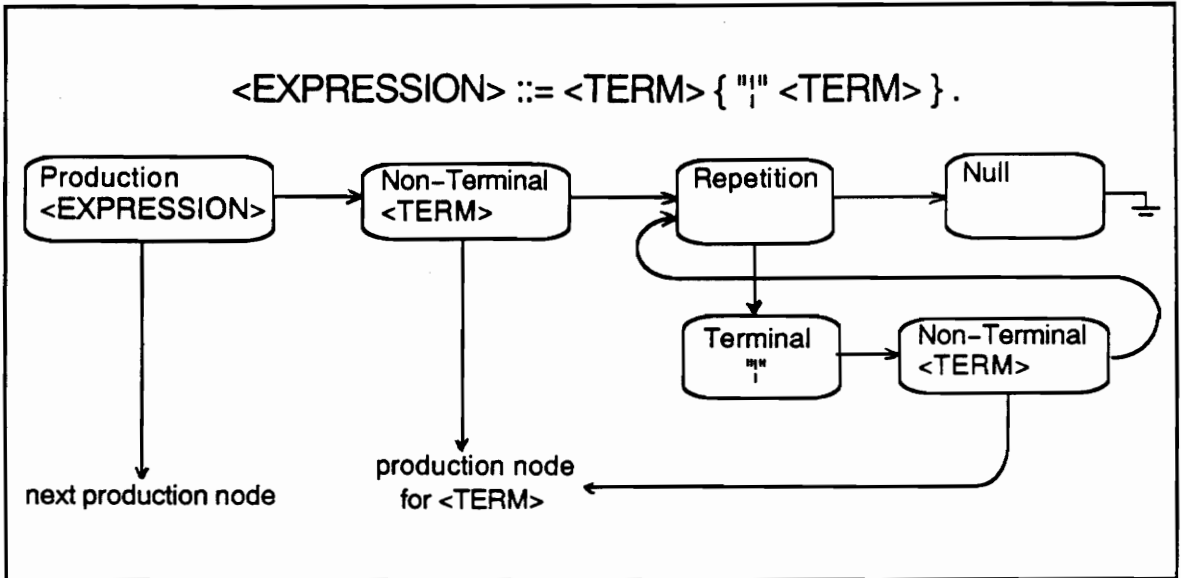


Figure 5: Example Production Sub-tree

linked together using the alternate node pointer with the Start symbol for the language at the top of the list. (The start symbol is arbitrarily defined as the non-terminal identifier which appears on the left-hand side of the first production in the EBNF specification.) The production sub-trees are reached by following the next node pointer. Within those sub-trees, nodes are traversed using the next node pointers for left-to-right sequential evaluation and the alternate node pointers for flow of control (as discussed below). The right-hand end of a production is indicated by next node which points to nil.

6.1.2 Flow of Control Tokens

There are four constructs in EBNF which permit flow of control within a production. The nodes which represent these constructs make use of the alternate node pointer to reference an optional, alternate or additional sub-tree of nodes which form a part of the production. Each of the four constructs is discussed individually below.

6.1.2.1 Grouping Meta-symbols

A sequence of symbols in a production is said to be grouped if it is surrounded by the "(" and ")" meta-symbols. Grouping is typically used in conjunction with the alternation construct to limit the scope of that alternation. (For example, in the sequence "A (B | C) D", the alternation is between B and C and does not extend outside the scope of the grouping.) The group construct is represented by two nodes: a group node and a "null" node. The group node's alternate node pointer will reference the first node of the grouped sequence, and the next node pointers of the last node(s) of the grouped sequence will refer back to the null node which follows the group node. The group's next node pointer will point to the null node.

6.1.2.2 Option Meta-symbols

A sequence of symbols in a production is said to be optional if it is surrounded by the "[" and "]" meta-symbols. This construct is represented in exactly the same manner as the group construct. (Except, of course, that the first node is an option node, not a group node.) The distinction between these two constructs is in the rules used for traversal. Option nodes represent a choice. Either the next node pointer or the alternate node pointer may be followed. In the case of a group node, there is no such choice involved. The alternate node pointer will always be the one followed.

6.1.2.3 Repetition Meta-symbols

Sequences of symbols which are surrounded by the "{" and "}" meta-symbols are said to be repeatable. These constructs are represented similarly to the option construct except that next node pointer of the last node(s) in the repeatable sub-tree points back to the repeat node instead of to the null node. This allows multiple entries to the repeatable sub-tree.

6.1.2.4 Alternation Meta-symbol

The final flow of control construct is alternation. This is represented by a node with subtrees on both the next node and the alternate node pointers. Unlike the three previous constructs, the alternation construct itself places no restrictions on the last nodes of its sub-trees, although they do inherit any restrictions from higher level constructs. Thus, if an alternation node is the first node in a production, the next node pointers of the last node(s) in each of the sub-trees will point to nil. If the same alternation were embedded in a group construct, the last node(s) would point the null node following the group. (This is what enables the scope of an alternation to be restricted.)

6.1.3 Terminals

A terminal node is placed in the production tree at any point where a terminal symbol must be matched. The next node pointer will point to the symbol immediately following the terminal in the production. The alternate node pointer for terminals is undefined; there are no choices to be made.

6.1.4 NonTerminals

A non-terminal node is placed in the sub-tree wherever a non-terminal identifier is encountered. The next node pointer points to the next node in the production, and the alternate node pointer points to the production node of the referenced non-terminal. In addition, a production reference (ProdRef) node is appended to the end of the referenced production. This node's alternate node pointer will point back to the non-terminal node, providing a cross-reference capability.

6.1.5 Declaration Directives

In the previous section, we discussed the concept of an identifier *declaration*. Recall that certain instances of the identifier token are considered to be declarations, while others are merely uses. To extend this capability to the target language, we provide two directives which may be inserted anywhere in the right-hand side of a production. `$DECLARE$` enables the declaration of subsequent identifier references, while `$NODECLARE$` disables this feature. (`$NODECLARE$` is the default behavior.) These two directives are represented in the production tree by two special node types (DeclON and DeclOFF) which appear at the appropriate location within the node sequence.

6.1.6 EOF Node

The final node type is the EOF node. This node will be automatically generated by the translator when it "augments" the grammar by creating a "meta-production" which contains a single non-terminal reference to the start production. This additional production is necessary to allow the use of the start symbol in recursive definitions. Consider the grammar:

$$\langle A \rangle ::= "(" \langle A \rangle ")" \mid "0" .$$

If the EOF node were appended directly to the end of this production, each recursive reference would require an EOF to be matched before the trailing parenthesis. By augmenting the grammar with the following meta-production:

$$\langle\langle \text{LANGUAGE} \rangle\rangle ::= \langle A \rangle \langle \text{EOF} \rangle .$$

we only require the presence of the EOF after the highest level of the start production.

6.2 Creation of the Production Tree

The actual translation of the user text into a production tree is accomplished by creating a modified set of the EBNF production procedures discussed in chapter 4. The initial versions were intended only to recognize correct EBNF. The new version will use embedded semantic actions to accomplish the translation. Each production rule and its corresponding translation procedure is discussed below.

6.2.1 Procedure EBNF

```
<EBNF> ::= { <directive> } { <PRODUCTION> } "_|_" .
```

This is the start production of the EBNF grammar. As noted, this variant allows a series of directives to precede the first PRODUCTION. These directives are used to specify the various delimiter characters which will be used by the scanner to recognize comments, strings and directives in the target language. On matching one of these leading directives, the procedure will alter the appropriate data element in the target language record. After traversing the last of the leading directives, this procedure simply calls PROD_tran (translate PRODUCTION) until an end of grammar symbol ("_|_") is detected.

6.2.2 Procedure PRODUCTION

```
<PRODUCTION> ::= <identifier> "::=" <EXPRESSION> "." .
```

This procedure is responsible for creating a production node and attaching to it the sub-tree corresponding to the EXPRESSION non-terminal. In addition, it creates the meta-production for the language the first time it is called. PROD_tran passes to EXPR_tran two parameters: the next node pointer of the newly created production node

(a synthesized or output parameter), and a pointer to a null node which will terminate the production (an inherited or input parameter).

6.2.3 Procedure **EXPRESSION**

$$\langle \text{EXPRESSION} \rangle ::= \langle \text{TERM} \rangle \{ \text{"|"} \langle \text{TERM} \rangle \} .$$

`EXPR_tran` handles the alternation construct. It calls `TERM_tran` to generate sub-trees which corresponds to that non-terminal, and attaches them to the next node pointers of alternation nodes which it generates. `EXPR_tran` will generate an alternation node before each call to `TERM_tran`, and will link the series of alternation nodes together using their alternate node pointers. This will result in one more alternation node than is strictly necessary, but will significantly simplify the routine. `EXPR_tran` accepts as input a pointer to the terminating node for the current sub-tree (which it passes on to `TERM_tran`), and returns a pointer to the first alternation node created.

6.2.4 Procedure **TERM**

$$\langle \text{TERM} \rangle ::= \langle \text{SYMBOL} \rangle \{ \langle \text{SYMBOL} \rangle \} .$$

`TERM_tran` strings together sequences of symbols using their next pointers. It returns to `EXPR_tran` a pointer to the first symbol of the sequence, and sets the next node pointer of the last symbol in the sequence to the pointer which was passed in by `EXPR_tran`. `TERM_tran` calls `SYMB_tran` to process each symbol. Since some "symbols" are actually multi-node constructs, `SYMB_tran` returns pointers to both the first and the last node of the "symbol".

6.2.5 Procedure SYMBOL

```

<SYMBOL> ::= <identifier> | <string> | <directive>
           | "(" <EXPRESSION> ")"
           | "[" <EXPRESSION> "]"
           | "{" <EXPRESSION> "}" .

```

SYMB_tran is actually the "hardest working" of the production procedures, since it is responsible for the creation of a wide variety of node types. Generally, this procedure handles either a single token (identifier, string or directive) or a set of control delimiters surrounding a sub-sequence of tokens. These two situations will be discussed separately.

6.2.5.1 Single Tokens

The single tokens recognized by SYMB_tran correspond directly to the terminals and non-terminals in the grammar. Each token will be converted into a single node in the production tree. Strings are added to the Simple Token table for the target language, and a terminal node with that entry's TokenId is generated. Identifiers which have been declared are considered non-terminals and cause the generation of a non-terminal node. Undeclared identifiers are either non-quoted terminals, or one of the special identifier values which correspond to general tokens returned by the scanner. Non-quoted terminals are treated identically to strings; special identifiers ("`<identifier>`", "`<string>`", "`<real>`", "`<integer>`", and "`<directive>`") generate terminal nodes with the appropriate general tokenId values. Directives may generate either DeclOn and DeclOff nodes, or a null node (for directives which are not recognized.) SYMB_tran will also recognize the directive "\$SYNC\$". This directive still maps to a null node, but has the effect of adding the next string token encountered to the SYNC Set of the target language. (See Paragraphs 4.3.3, 7.2.)

6.2.5.2 Control Constructs

SYMB_tran handles control constructs by creating two nodes, a control node (either option, repetition, or grouping) and a null node. It then calls EXPR_tran recursively to

generate a sub-tree which it then attaches to the alternate node pointer of the control node. By passing the appropriate second parameter to `EXPR_tran`, `SYMB_tran` governs where the next node pointers of the right-most nodes in the expression sub-tree will point. For option and grouping, they should point to the null node, indicating only one traversal of the sub-tree. For repetition, they should point back to the control node, enabling multiple traversals.

6.3 Select Sets

After the production tree has been created, all decision points within the tree must be "decorated" with the select sets which will enable a deterministic parse to take place. This is done by performing a pre-order traversal of the production tree beginning with the start production. Since the "tree" will probably contain cycles, each node is marked as it is visited to ensure only a single pass. During the traversal, all option, repetition, alternation, production and non-terminal nodes are "decorated" as they are visited.

6.3.1 Decorating Nodes

To "decorate" a node, we must compute the select sets for each of the possible paths out of that node. The select set for a particular sub-tree starts with all tokenids found in the First Set of that sub-tree. This set can be built simply by performing a depth-first traversal of the sub-tree. At terminal nodes, we truncate the traversal and add the tokenid of the terminal to the set. At option, alternation and repetition nodes, we continue the traversal down both the next and alternate node pointers. (The First Set will be the union of First Sets of the two paths since either may be traversed.) At group and non-terminal nodes, we continue the traversal along the alternate node path only.

This strategy is sufficient as long as none of the sub-trees are nullable. (That is, none of the sub-trees contain a path from the root to nil without any intervening terminal nodes.) In cases where the sub-tree is nullable, the select set must be augmented with the Follow Set of the production. The Follow Set for a production is defined as the union of the First Sets of all the sub-trees which immediately follow non-terminal references to that production. That is, the set of tokens which can legally follow that production. This is accomplished using the ProdRef (production cross-reference) nodes created with each non-terminal node.

Recall that for each non-terminal node created, there is a corresponding ProdRef node appended to the target production which contains a pointer back to the non-terminal. When attempting to compute the Select Set of a nullable sub-tree, these ProdRef nodes will be reached before the nil pointer at the right-most end of the sub-tree. By following the alternate node pointer of each of the ProdRef nodes, the sub-trees which follow all non-terminal references to the production can be traversed in order to determine the Follow Set. This set can then be added to the Select Set being computed.

6.3.2 LL(1) Error Detection

While traversing the production tree to compute the select sets, several errors can be detected. If, while computing a First Set, a node is traversed more than once, that indicates that the grammar is left recursive. That is, there is a nullable cycle in the production tree. Since the grammar is allowed only a single lookahead, it is not possible to determine how many times to traverse the cycle before continuing down another path. Note that during the computation of a Follow Set, it is valid to encounter nodes multiple times since a given production may be called from many places within a grammar.

The other error which can be detected at this point is ambiguity. If, after computing the select sets for the two paths out of a decision node (alternation, option or

repetition), it is found that the intersection of those sets is non-empty, the grammar is ambiguous at that decision. If that point in a translation is reached and the input token is a member of both select sets, it is not possible to determine which of the two paths to take.

In both of these cases, the user must be informed of the error in the grammar. This is done similarly to scanner and parser errors by marking in error the token associated with the production node where the error was detected. These error conditions will cause the tokens to be displayed in red, and will provide appropriate messages when the cursor is placed on the tokens. Since these errors are generated by the translation attempt, they will disappear the first time the source file is changed. (This is a benefit of re-parsing the entire input text each time it changes: all error flags are cleared by the parser.)

6.4 Follow-up

After the decoration of the production tree has proceeded without error, one more condition must be checked. A final pass through all the production nodes must be made to determine if they have all been traversed in the decoration. Since the decoration traversal ignores the alternate node pointers which link the production nodes together, this provides a way to check to make certain that all the productions of the language are "reachable" from the start production.

If the production tree contains no unreachable productions and is free from grammatical errors, the editor is placed into "USER" mode and the input buffer is cleared to allow the input of text in the target language.

7 Syntactic Analysis of the Target Language

In this section the use of the production tree to parse user text in the target language is detailed. Recall that the parsing of EBNF (see section 4) was accomplished using a hand-coded, recursive descent parser. This method was easy to implement and modify, but had the disadvantage of requiring the entire token stream to be re-parsed each time a change was made. The parsing of the target language is handled in an incremental fashion. Like the scanner, the parser will only re-parse those sections of the text which need to be re-parsed.

7.1 Incremental Parsing

In implementing the incremental scanner, we introduced the concept of the "state" of the scanner. In the case of the scanner, this simply consisted of a variable indicating whether or not the scanner was currently handling a comment. This same concept can be extended to the parser. We define the state of the parser after processing as consisting of the stack of all non-terminals which are currently being evaluated. In a recursive descent parser, this stack is maintained implicitly as the stack of production procedures which have been called. For this very reason, recursive descent parsers do not lend themselves to interruption or resumption (key requirements for an incremental parser).

The parser for the target language was therefore designed as a push-down automaton (PDA). The production tree built in the previous section can be used as a ready-made state-transition diagram. Terminal nodes are the only nodes which consume input tokens, and they have but a single transition. Decision nodes have two transitions, each of which is associated with one of their select sets. Non-terminal nodes are pushed

onto the stack before their alternate node transition is followed. On encountering a nil (the right-hand end of a production), the non-terminal on top of the stack is popped, and the next node transition is followed. DeclOn and DeclOff nodes (see section 6.1.5) set and clear a flag (within the state variable) which indicates whether identifiers are to be declared. The start node is, of course, the production node of the meta-production which refers to the start production; the end (accept) node is the EOF node attached to the end of that production.

By maintaining the state of the parser (the stack of non-terminals) externally, we can start it at an arbitrary point within the user text, and it can stop when it runs out of tokens. As in the scanner, we can store with each line of user text the state that the parser was in the last time it reached the beginning of that line. If a single line of the text is changed, the stack is set to the stored state and the parser is started with the first token on that line. When the parser reaches the end of the line, it simply halts. The ending state can then be compared to the start state for the next line. If they are identical, there is no need to re-parse subsequent lines. This procedure is nearly identical with the method used to incrementally scan changed text.

7.2 Error Detection and Recovery

In the recursive descent parser for EBNF, error detection and recovery were handled by the procedure MATCH which was called to match a particular terminal. While the incremental parser has its own version of procedure MATCH, it relies on another procedure called CheckNextToken to detect and report error conditions. CheckNextToken is called by MATCH when it attempts to match a terminal. In addition, it is called whenever the parser encounters a decision node (alternation, repetition, option) to verify

that the current input token is a member of one of the select sets. This allows the detection of error conditions to happen earlier than in the EBNF parser.

In the EBNF parser, each decision point had a default action and only a single select set. Thus, for a repetition construct, if the next token was a member of the select set for the repetition, the repetition was performed; if not, the default action (to skip the repetition) was performed. Thus the default action would be taken even if the next token was not a member of the select set for that branch. Because of this behavior, errors were not detected until a MATCH was encountered. Since that MATCH was for a single token only, the error message ("`<tokentype>` expected") could only contain a single valid token type. The earlier detection in the incremental parser allows the offending token to be labelled with the entire set of tokenids which would be valid in that position.

Except for the earlier detection of errors, the error recovery strategy in the incremental parser is nearly identical with that described for the EBNF parser. Either the offending input token is skipped (leaving the parser in the same state), or the state of the parser is advanced (keeping the same input token). SYNC tokens which were specified in the input grammar will cause the parser to attempt a synchronization. Only one additional feature was required.

In section 4.3.1 we briefly discussed the dangers of token insertion as an error recovery strategy. In certain situations, allowing the parser to advance without consuming an input token can cause it to enter a recursive loop. While EBNF was defined to avoid this problem, we have no guarantee that the user specified language will be similarly benign. In order to detect this problem, the routine responsible for pushing non-terminals onto the stack has been modified. Each time a non-terminal is encountered, the stack is traversed to see if that same non-terminal has been pushed before *with the same input token*. If this situation is encountered, it is clear that the parser is entering a recursive

loop. To correct the situation, the push routine clears the stack down to the duplicate entry and consumes the current input token. This can be looked on as deciding that token insertion was a bad idea, and that skipping the input token is a better option. Note that this situation will only occur during error recovery. Recursive loops in the production tree would have been detected during the translation process as an error in the grammar.

7.3 Declaration Checking

Recall the use of the declaration concept to distinguish between identifiers representing non-quoted terminals and those representing non-terminals. In most declarative languages, the lack of a declaration for an identifier is considered an error. To allow this behavior to be specified, the directives `$CHECKDECLARE$` and `$NOCHECKDECLARE$` will be used by the translator to set a global flag which will govern the actions taken in response to an undeclared identifier. In EBNF mode this flag will be cleared, indicating that undeclared identifiers are allowed. In user mode, this flag will be set by default to indicate that undeclared variables are an error and should be marked as such.

8 Conclusions

The body of this paper has described the design and construction of a general purpose syntax-aware text editor. The system was built iteratively, with increasingly abstract functions added to a bare-bones text editor. Each of the major refinements to the editor has been detailed in one of the previous chapters. In this final chapter, we will examine the resulting program, consider its usefulness, and recommend further work which might be performed in this area.

8.1 Critique

There are several shortcomings which can be noted about this editor. The first and probably most noticeable is the sparseness of the traditional text editing features. There are no block functions, no search functions, and the input area is limited to only 80 characters per line. While these features are certainly valuable in a production editor, I felt that they contributed little to the demonstration of the syntax-awareness of this editor. For reasons of clarity as well as expedience, these features were intentionally omitted from the design.

Another problem is the editor's somewhat voracious appetite for memory. The data structures (like the editing features) were designed primarily for clarity and expedience. An example of this is the record of the parser's state which is kept with each text line. The use of an array of 32 pointers to represent the non-terminal stack is certainly not an optimal use of space. It does, however, have the advantage of being easy to access and to understand. With additional work, a more space efficient data structure design could be developed.

The lexical analyzer could also be significantly improved both in functionality and in efficiency. The recognition of engineering notation and alternate base notation (hex or octal) have been omitted from this editor's design for expedience. Also, the storage and lookup of simple tokens is quite inefficient. Currently, the maximum length of a simple token (and thus, the required number of lookahead characters) is defined as a constant for the language. A modification to allow this number to vary according to the first character encountered would improve the efficiency tremendously.

The final major shortcoming of this editor is a direct result of the design and implementation strategy used. Because of the short time-frame involved, I chose to prototype the editor rather than conduct a full-fledged requirements analysis and detailed design. This is evident in the iterative approach to the system which is described in the previous chapters. The result of this strategy is that many of the design decisions were not made until well into the development of the system. The resulting program tends to be somewhat disorganized. It has been said of the prototyping methodology that one should discard one's first attempt at implementing any major system. While that may be a bit extreme, this system could certainly benefit from some strategic redesign.

8.2 Usefulness

This editor is certainly not ready for public use as a general purpose text editor. The lack of sophistication of its interface impairs its effectiveness as a programming tool. This system does, however, provide a good demonstration of the capabilities which might be included in a production editor incorporating syntax-awareness. This editor could also be used profitably in certain specialized situations.

A language developer might find this system useful to test proposed syntax specifications. The ability to switch back and forth between editing the EBNF specification and the target language text provides a quick way of trying out changes to a syntax. Using traditional compiler development tools, the developer might have to edit the EBNF, exit the editor, compile the EBNF to produce a target language compiler, then compile the target language text to see the results. This editor provides a much more integrated approach. After editing the EBNF (the editor will, of course, detect errors in the EBNF itself), the developer simply presses alt-T to translate the EBNF, then loads the target language text. The results of the EBNF changes are displayed on the screen. If further EBNF changes are required, the developer simply presses alt-E to return to EBNF mode, then reloads the specification.

This system might also prove valuable in a course on compiler construction or computational theory. The students (and the instructor) could use the editor to receive instant feedback on the correctness of their EBNF specifications. In addition, since BNF is a subset of EBNF, specifications written in that language could also be verified.

8.3 Further Work

There are many ways in which this editor might be improved. As noted above, the basic text editing features are sparse at best. The incorporation of a more complete set of these features should be easy to accomplish. In addition, there are many features usually associated with syntax-directed editors which could be added to this system. The ability to select blocks of text by syntactic association would be a valuable addition. (For example, selecting a *statement* instead of a line or a word.) This would allow users to move, copy, delete, or hide portions of their text in a much more intelligent fashion. Other

useful features include the ability to display (or go to) the declaration of a particular identifier and the ability to display (or save) a "pretty-printed" version of the source text.

There are also several enhancements which might be made to the parsing capabilities of the system. The addition of scoping to the symbol table would better facilitate the parsing of block structured languages such as Pascal. This would be a fairly trivial modification. A more ambitious undertaking might be to modify the editor to accept and process semantic attributes. This would allow the implementation of such features as type checking and array bounds checking.

Appendix A: Source Code

```
program Editor;
{ This program is a prototype General Purpose Syntax-Aware Text Editor.

  It was built in partial fulfillment of the requirements for the degree
  of MASTER OF SCIENCE in Computer Science by Joseph L. Faber III at
  the Virginia Polytechnic Institute and State University. 5/91 }

uses CRT,DOS;

const
  MAXY = 23;
  MAXTOKLEN = 10;
  StatusColor = Black + LightGray*$10;
  ConfirmColor = White + Red*$10;
  ErrorColor = Yellow + Red*$10;
  NormalColor = Yellow + Blue*$10;
  ErrCtColor = Red + LightGray*$10;
  ErrTokColor = LightRed + Blue*$10;
  TermTokColor = White + Blue*$10;
  DimTokColor = LightGray + Blue*$10;
  DosColor = LightGray + Black*$10;
  TabChar = #$09;
  EOFChar = #$16;
  EBNF_SD = ''; {string delimiter}
  EBNF_SC = '#'; {start comment}
  EBNF_EC = ''; {end comment}
  EBNF_CheckDecl = false; {undeclared ids are OK in EBNF (because
                           they're considered TERMINALS)}

  { ErrorCode Values }
  StringError: integer = 1;
  IllegalChar: integer = 2;
  IllegalCompDir: integer = 3;
  InsertOne: integer = 4;
  PrematureEOF: integer = 5;
  SkippingToNext: integer = 6;
  SkipOne: integer = 7;
  MultipleDecl: integer = 8;
  NoDecl: integer = 9;
  Unreachable: integer = 10;
  Ambiguous: integer = 11;
  LeftRecursive: integer = 12;
  ScanErrors: set of byte = [1,2,3];

  { TokenId Values }
  Affinity: byte = 000;
  IdToken: byte = 200;
  StringToken: byte = 201;
  IntToken: byte = 202;
  RealToken: byte = 203;
  Sentinel: byte = 251;
  IllegalCharToken: byte = 252;
  CommentToken: byte = 253;
  CompDirToken: byte = 254;
  EOFToken: byte = 255;

type
  FilePtr = ^FileBuffer;
  LinePtr = ^FileLine;
  TokenPtr = ^TokenRec;
  SymbPtr = ^SymbolRec;
  OccPtr = ^OccRec;
  NodePtr = ^NodeRec;

  SelectSet = set of byte;

  { ---COMPILER STATE TYPE--- }
```

```

StackEntry = record
    Node: NodePtr;
    Token: TokenPtr;
end;

StateRec = record
    Comment : boolean; { are we somewhere within a comment? }
    Declare : boolean; { are identifiers being declared? }
    ErrorFlg : boolean; { have we had any errors? }
    Node : NodePtr; { where are we in the prod tree }
    Depth : integer; { how deep is the stack }
    Stack : array[1..32] of StackEntry
end;

FileBuffer = record
    Name : string[80];
    ChgFlag : boolean;
    FirstLine: LinePtr;
end;

FileLine = record
    Line : string[80];
    NextLine : LinePtr;
    PrevLine : LinePtr;
    FirstTok : TokenPtr;
    ChgFlag : boolean;
    State : StateRec;
end;

{ ---TOKEN TYPE--- }
TokenRec = record
    TokenId : byte;
    StartCh : integer;
    EndCh : integer;
    Error : byte;
    Expected : SelectSet;
    IdError : byte;
    SymbRef : SymbPtr;
    NextTok : TokenPtr;
    SrcLine : LinePtr;
end;

EditVars = record
    FileBuf : FilePtr;
    FirstLine : LinePtr;
    CurrentLine : LinePtr;
    ChgFlag : boolean;
    ErrorFlg : boolean;
    Xpos, Ypos : integer;
    InsertOn : boolean;
end;

LanguageRec = record
    EBNFFlag: boolean; {which parse method do we use?}
    Grammar: string[12]; {what grammar file is in use?}
    CheckDecl: boolean; {are undeclared id's an error?}
    Numerics: boolean; {should the scanner recognize numerics?}
    StartComment,
    EndComment,
    StartCompDir,
    EndCompDir: string[2];
    StringChar: char; {these are delimiter sequences}
    { Simple Token Table }
    SimpleSet: set of char; {start characters}
    SimpleTokens: array [1..199] of string[MAXTOKLEN];
    SimpleCount: byte;
    SimpleMaxLen: byte;
    { Symbol Table }
    SymbolTable: SymbPtr;
    SyncTokens: set of byte;
    StartNode: NodePtr;
end;

SymbolRec = record
    IdText: string[80];
    DeclCnt: integer; {How many Declarations?}

```

```

Standard:   boolean; {Standard Id?}
FirstOcc:  OccPtr;  {ptr to reference nodes}
Error:     byte;    {multiple/no decl}
ProdPtr:   NodePtr; {for NT, ptr to production node}
NextSymb:  SymbPtr;
end;
OccRec     = record
  Decl:    boolean; {Is this token a declaration?}
  Token:   tokenptr;
  NextOcc: OccPtr;
end;

NodeType   = (Prod,NT,ProdRef,
              Opt,Grp,Rpt,Nul,Alt,Term,EOFNode,
              DeclOn,DeclOff,Aff);
NodeRec    = record
  NodeType: NodeType;
  TermId:   byte;    {expected tokenid}
  NextNode: NodePtr;
  AltNode:  NodePtr;
  NextSelect: SelectSet;
  AltSelect: SelectSet;
  SrcToken: TokenPtr;
  Visited:  Boolean;
end;

var
  I,J : integer;
  editarea: editvars;
  ch: char;
  hr,min,sec,hun: word;
  starttime, endtime: longint;
  quit: boolean;
  initialstate: staterec;
  Debug: text;
  EBNF: LanguageRec;
  UserLang: LanguageRec;
  Lang: LanguageRec;

procedure InitLang(var lang:LanguageRec);
{ initialize a language record }
begin
  with lang do begin
    EBNFFlag := false;
    Numerics := true;
    Grammar := '';
    CheckDecl := true;
    SimpleSet := [];
    SimpleCount := 0;

    StartComment := EBNF_SC;
    EndComment := EBNF_EC;

    StartCompDir := '$';
    EndCompDir := '$';

    StringChar := EBNF_SD;

    SyncTokens := [EOFToken];
    StartNode := nil;
    SymbolTable := nil;

    end;
end;

function SimpleLookup(var lang:LanguageRec;symb: string): byte;
{ return the tokenid of symb if it exists in the simple tokens of lang. }
var
  i: byte;
begin
  i := 0;
  { if it's longer than the longest token, don't bother }
  if (length(symb) > lang.SimpleMaxLen) then
    symb := '';
  while (i = 0) and (symb <> '') do begin

```

```

repeat
  i := i + 1;
until (i > lang.SimpleCount) or (lang.SimpleTokens[i] = symb);
if i > lang.SimpleCount then
  i := 0;
delete (symb,length(symb),1);
end;
SimpleLookup := i;
end;

function SimpleInsert(var lang:LanguageRec;symb: string): byte;
{insert symb into the simple tokens of lang if not already there}
var
  i,j: byte;
begin
  symb := copy (symb,1,MAXTOKLEN);
  i := SimpleLookup(lang,symb);
  if (i=0) or (lang.SimpleTokens[i] <> symb) then begin
    if not (symb[1] in ['A'..'Z']) then
      lang.SimpleSet := lang.SimpleSet+[symb[1]];
    i := lang.SimpleCount + 1;
    lang.SimpleTokens[i]:= symb;
    lang.SimpleCount := i;
    if length(symb) > lang.SimpleMaxLen then
      lang.SimpleMaxLen := length(symb);
    end;
    SimpleInsert := i;
  end;
end;

function NewNode (nt: NodeType): NodePtr;
{create and initialize a new production node}
var
  p: NodePtr;
begin
  new(p);
  with p do begin
    NodeType := nt;
    TermId := 0;
    NextNode := nil;
    AltNode := nil;
    SrcToken := nil;
    AltSelect:=[];
    NextSelect:=[];
    Visited := false;
  end;
  NewNode := p;
end;

function IntToStr (num: longint):string;
var
  outval : string[10];
begin
  str(num,outval);
  IntToStr := outval;
end;

function Upper (st: string):string;
var
  i: integer;
begin
  for i := 1 to length(st) do
    st[i] := UpCase(st[i]);
  Upper := st;
end;

procedure Beep;
begin
  sound (1000);
  delay (50);
  nosound;
end;

procedure DirectScreen (row,col,len:integer;str: string;att: byte);
{ place str directly on the screen (via memory) with the given txt attr }
var

```

```

    i,j : integer;
begin
    j := (row-1)*80*2+(col-1)*2;
    for i := 1 to len do begin
        if str <> '' then begin
            mem[$B800:j] := integer(str[i]);
            delete(str,1,1);
        end
        else
            mem[$B800:j] := integer(' ');
            mem[$B800:j+1] := att;
            j := j + 2;
        end;
    end;
end;

procedure Paint (row,col,len:integer;att: byte);
{paint the indicated screen area with the given text attribute }
var
    i,j : integer;
begin
    j := (row-1)*80*2+(col-1)*2;
    for i := 1 to len do begin
        mem[$B800:j+1] := att;
        j := j + 2;
    end;
end;

procedure StatusLine;
{display the status line to the screen}
var
    fn: string;
begin
    with Editarea do begin;
        fn := FileBuf.Name;
        if InsertOn then
            DirectScreen(1,70,3,'INS',StatusColor)
        else
            DirectScreen(1,70,3,'',StatusColor);
        if lang.EBNFFlag then
            DirectScreen(1,75,4,'EBNF',StatusColor)
        else begin
            DirectScreen(1,75,4,'',StatusColor);
            fn := fn + ' (' + lang.Grammar + ')';
        end;
        if ErrorFlg then
            DirectScreen(25,1,9,' Errors: ',ErrorColor)
        else
            DirectScreen(25,1,9,'',StatusColor);
        DirectScreen(1,5,27,fn,StatusColor);
    end;
end;

function SymbError(s_ptr: SymbPtr): boolean;
{traverse the symbol table and report if any errors are found}
begin
    if s_ptr = nil then
        SymbError := false
    else if (s_ptr^.Error <> 0) then
        SymbError := true
    else
        SymbError := SymbError(s_ptr^.NextSymb);
end;

procedure CheckSymbErrors(s_ptr: SymbPtr);
{determine whether any symbols have decl errors}
var
    ptr: OccPtr;
    err: byte;
begin
    with s_ptr do begin
        DeclCnt := 0;
        ptr := FirstOcc;
        while ptr <> nil do begin
            if ptr^.Decl then
                DeclCnt := DeclCnt + 1;
        end;
    end;
end;

```

```

    ptr := ptr^.NextOcc;
  end;
  case DeclCnt of
    0 : if Standard or not lang.CheckDecl then
        Error := 0
      else
        Error := NoDecl;
      1 : Error := 0;
      else Error := MultipleDecl;
    end;
  ptr := FirstOcc;
  while ptr <> nil do begin
    ptr^.Token^.IdError := Error;
    ptr := ptr^.NextOcc;
  end;
end;

end;

procedure AddStandardId(var ptr:SymbPtr; id: string);
{add a special ID which doesn't require a decl (like "<identifier>")}
var
  temp: SymbPtr;
begin
  if (ptr = nil) or (ptr^.IdText > id) then begin
    temp := ptr;
    new (ptr);
    with ptr^ do begin
      NextSymb := temp;
      IdText := id;
      DeclCnt := 0;
      Standard := true;
      FirstOcc := nil;
      Error := 0;
    end;
  end
  else if (ptr^.IdText < id) then
    AddStandardId(ptr^.NextSymb,id)
  else
    ptr^.Standard := true;
end;

procedure Register(var ptr:SymbPtr;t_ptr: TokenPtr;declare: boolean);
{register a symbol with the table.}
var
  id: string;

  function SymbIns(var ptr:SymbPtr;var id: string): SymbPtr;
  var
    temp: SymbPtr;
  begin
    if (ptr = nil) or (ptr^.IdText > id) then begin
      temp := ptr;
      new (ptr);
      with ptr^ do begin
        NextSymb := temp;
        IdText := id;
        DeclCnt := 0;
        Standard := false;
        FirstOcc := nil;
        ProdPtr := nil;
        Error := 0;
      end;
      SymbIns := ptr;
    end
    else if (ptr^.IdText < id) then
      SymbIns := SymbIns(ptr^.NextSymb,id)
    else
      SymbIns := ptr;
  end;

  procedure OccIns(var ptr: OccPtr; t_ptr: tokenptr; declare: boolean);
  begin
    if ptr = nil then begin
      new (ptr);

```

```

    ptr^.NextOcc := nil;
    ptr^.Token := t_ptr;
    ptr^.decl := declare;
  end
  else if ptr^.Token = t_ptr then
    ptr^.decl := declare
  else
    OccIns(ptr^.NextOcc,t_ptr,declare);
end;

begin
  with t_ptr^ do begin
    id := Upper(Copy(SrcLine^.Line,StartCh,EndCh-StartCh+1));
    SymbRef := SymbIns(ptr,id);
    OccIns(SymbRef^.FirstOcc,t_ptr,declare);
    CheckSymbErrors(SymbRef);
  end;
end;

procedure UnRegister(var ptr: SymbPtr; t_ptr: TokenPtr);
{remove an occurrence of a symbol from the table}

  procedure OccDel(var ptr: OccPtr; t_ptr: tokenptr);
  var
    temp: Occptr;
  begin
    if ptr^.Token = t_ptr then begin
      temp := ptr;
      ptr := ptr^.NextOcc;
      dispose (temp);
    end
    else if (ptr <> nil) then
      OccDel(ptr^.NextOcc,t_ptr);
  end;

  procedure SymbDel(var ptr:SymbPtr; s_ptr: SymbPtr; t_ptr: tokenptr);
  begin
    if (ptr = s_ptr) and (ptr <> nil) then begin
      OccDel(ptr^.FirstOcc,t_ptr);
      if (ptr^.FirstOcc = nil) and (not ptr^.Standard) then begin
        ptr := ptr^.nextSymb;
        dispose(s_ptr);
      end
    else
      CheckSymbErrors(ptr);
    end
    else if (ptr <> nil) then
      SymbDel(ptr^.NextSymb,s_ptr,t_ptr);
  end;

begin
  SymbDel(ptr,t_ptr^.SymbRef,t_ptr);
end;

procedure disposeTokens (var ptr: TokenPtr);
{recursively dispose all tokens in a chain}
begin
  if ptr <> nil then begin
    disposetokens(ptr^.NextTok);
    if ptr^.TokenId = IdToken then
      UnRegister(lang.SymbolTable,ptr);
    dispose(ptr);
    ptr:= nil;
  end;
end;

procedure ClearFile;
{clear the edit buffer}
var
  temp,temp2: lineptr;
begin
  with editarea do begin
    filebuf^.Name := 'NONAME.TXT';
    temp2 := filebuf^.firstline;
  end;
end;

```

```

disposeTokens(temp2^.FirstTok);
temp2^.line := '';
temp2^.ChgFlag := true;
temp := temp2^.NextLine;
temp2^.NextLine := nil;
while temp <> nil do begin
    temp2 := temp;
    temp := temp^.NextLine;
    disposeTokens (temp2^.FirstTok);
    dispose (temp2);
end;
FirstLine := filebuf^.firstline;
CurrentLine := FirstLine;
FirstLine^.ChgFlag := true;
ChgFlag := true; { cause us to reparse the edit buffer }
ErrorFlg := false;
FileBuf^.ChgFlag := false; { but we havent really chged the file }
Xpos := 1;
Ypos := 1;
end;
end;

```

```

procedure ReadFile (name: string);
{Read a file into memory and set up edit vars}
var
    infile: text;
    temp,temp2: lineptr;

```

```

begin
    ClearFile;
    with editarea do begin
        temp := filebuf^.firstline;
        filebuf^.Name := name;
        assign (infile, name);
        reset (infile);
        while not eof (infile) do begin
            new (temp^.nextline);
            temp^.nextline^.prevline := temp;
            temp^.nextline^.nextline := nil;
            temp := temp^.nextline;
            temp^.FirstTok := nil;
            temp^.ChgFlag := true;
            readln (infile, temp^.line);
            end;
            temp := filebuf^.firstline;
            if temp^.nextline <> nil then begin
                filebuf^.firstline := temp^.nextline;
                temp^.nextline^.prevline := nil;
                disposeTokens (temp^.FirstTok);
                dispose(temp);
            end;
            FirstLine := filebuf^.firstline;
            CurrentLine := FirstLine;
            ChgFlag := true; { cause us to reparse the edit buffer }
            ErrorFlg := false;
            FileBuf^.ChgFlag := false; { but we havent really chged the file }
            Xpos := 1;
            Ypos := 1;
            end;
        end;
end;

```

```

procedure WriteFile;
{ write the text buffer to disk}
var
    outfile: text;
    name: string[80];
    temp: lineptr;

```

```

begin
    name := editarea.filebuf^.name;
    temp := editarea.filebuf^.firstline;
    assign (outfile, name);
    rewrite (outfile);
    while temp <> nil do begin
        writeln(outfile,temp^.Line);
    end;
end;

```

```

    temp := temp^.nextline;
    end;
    close (outfile);
    editarea.filebuf^.ChgFlag := false;
end;

procedure JoinLines;
{ join the current and next line in the edit buffer }
var
    temp: LinePtr;
begin
    with editarea do begin
        ChgFlag := true;
        filebuf^.ChgFlag := true;
        temp := CurrentLine^.NextLine;
        CurrentLine^.Line := concat(CurrentLine^.Line,temp^.Line);
        CurrentLine^.NextLine := temp^.NextLine;
        if temp^.NextLine <> nil then
            temp^.NextLine^.PrevLine := CurrentLine;
        disposeTokens(temp^.FirstTok);
        dispose(temp);
    end;
end;

procedure ApplyColor (row:integer;ptr:TokenPtr);
{ use the errorcode and tokenId to paint tokens the right color}
begin
    if ptr <> nil then begin
        with ptr^ do begin
            if (Error <> 0) or (IdError <> 0) then
                Paint(row,StartCh,EndCh-StartCh+1,ErrTokColor)
            else if ((TokenId = IdToken) and
                ((SymbRef^.DeclCnt = 0) or lang.CheckDecl)) or
                (TokenId in [StringToken,IntToken,RealToken]) then
                Paint(row,StartCh,EndCh-StartCh+1,TermTokColor)
            else if TokenId in [CommentToken,CompDirToken] then
                Paint(row,StartCh,EndCh-StartCh+1,DimTokColor)
            end;
            ApplyColor (row,ptr^.NextTok);
        end;
    end;
end;

procedure Display;
{display the screen}
var
    i: integer;
    p: lineptr;
begin
    Window(1,2,80,25);
    TextAttr := NormalColor;
    i := 1;
    p := editarea.firstline;
    while (i <= MAXY) and (p <> NIL) do begin
        GotoXY (1,i);
        ClrEol;
        write(p^.line);
        if p^.nextline = nil then
            write (EOFChar);
        ApplyColor(1+i,p^.FirstTok);
        i := i + 1;
        p := p^.nextline;
    end;
    while (i <= MAXY) do begin
        GotoXY (1,i);
        ClrEol;
        i := i + 1;
    end;
end;

procedure UpdateDisplay;
{Just repaint. don't redisplay}
var
    i: integer;
    p: lineptr;

```

```

begin
  i := 1;
  p := editarea.firstline;
  while (i <= MAXY) and (p <> NIL) do begin
    Paint(1+i,1,80,NormalColor);
    ApplyColor(1+i,p^.FirstTok);
    i := i + 1;
    p := p^.nextline;
  end;
end;

procedure DisplayLine;
{redisplay the current line}
begin
  TextAttr := NormalColor;
  with editarea do begin
    GotoXY (1,Ypos);
    ClrEol;
    write(CurrentLine^.line);
    if CurrentLine^.nextline = nil then
      write(EOFChar);
    ApplyColor(Ypos+1,CurrentLine^.FirstTok);
  end;
end;

procedure DisplayErrorMsg;
{display the error message associated with the token under the cursor}
var
  i: byte;
  ptr: TokenPtr;
  msg: string [60];
begin
  with editarea do begin
    ptr := CurrentLine^.FirstTok;
    while ptr <> nil do
      if (ptr^.StartCh <= Xpos) and (ptr^.EndCh >= Xpos) then begin
        if (ptr^.Error <> 0) or (ptr^.IdError <> 0) then begin
          msg := '';
          if ptr^.Expected <> [] then
            for i := 1 to 255 do
              if i in ptr^.Expected then begin
                if msg <> '' then msg := ', ' + msg;
                case i of
                  1..199: if i <= lang.SimpleCount then
                    msg := '' + lang.SimpleTokens[i] + '' + msg;
                  200 : msg := '<id>' + msg;
                  201 : msg := '<str>' + msg;
                  202 : msg := '<int>' + msg;
                  203 : msg := '<real>' + msg;
                  254 : msg := '<dir>' + msg;
                  255 : msg := 'EOF' + msg;
                else;
                end;
              end;
          if ptr^.Error = IllegalChar then
            msg := 'Illegal Character'
          else if ptr^.Error = StringError then
            msg := 'Illegal String'
          else if ptr^.Error = IllegalCompDir then
            msg := 'Error in Directive'
          else if ptr^.Error = InsertOne then
            msg := 'Unexpected Token -- Inserting ' + msg
          else if ptr^.Error = PrematureEOF then
            msg := 'Premature End Of File -- Expected ' + msg
          else if ptr^.Error = SkippingToNext then
            msg := 'Unexpected Token -- Skipping to ' + msg
          else if ptr^.Error = SkipOne then
            msg := 'Unexpected Token -- Skipping'
          else if ptr^.IdError = MultipleDecl then
            msg := 'Multiple Declarations'
          else if ptr^.IdError = NoDecl then
            msg := 'Identifier not declared'
          else if ptr^.Error = Unreachable then
            msg := 'Production not reachable'
        end;
      end;
    end;
  end;
end;

```

```

        else if ptr^.Error = Ambiguous then
            msg := 'Ambiguous Grammar'
        else if ptr^.Error = LeftRecursive then
            msg := 'Left Recursive Grammar'
        else
            msg := 'Unknown Error Code';
            DirectScreen(25,10,length(msg),msg,ErrorColor);
        end;
        ptr := nil; {so we quit out of the loop}
    end
else
    ptr := ptr^.NextTok;
end;
end;

end;

procedure MoveCursor(xinc,yinc:integer);
{move the cursor around in the buffer}
var
    dispflag : boolean;
begin
    dispflag := false;
    with editarea do begin
        Xpos := Xpos+xinc;
        if (Xpos > 80) then
            Xpos := 80
        else if (Xpos < 1) then
            Xpos := 1;
        while yinc > 0 do
            if CurrentLine^.NextLine <> nil then begin
                if Ypos < MAXY then begin
                    Ypos := Ypos + 1;
                    CurrentLine := CurrentLine^.NextLine;
                end
            else begin
                dispflag := true;
                FirstLine := FirstLine^.NextLine;
                CurrentLine := CurrentLine^.NextLine;
            end;
            yinc := yinc - 1;
        end
        else
            yinc := 0;
        while yinc < 0 do
            if CurrentLine^.PrevLine <> nil then begin
                if Ypos > 1 then begin
                    Ypos := Ypos - 1;
                    CurrentLine := CurrentLine^.PrevLine;
                end
            else begin
                dispflag := true;
                FirstLine := FirstLine^.PrevLine;
                CurrentLine := FirstLine;
            end;
            yinc := yinc + 1;
        end
        else
            yinc := 0;
        if dispflag then
            Display;
        GotoXY(Xpos,Ypos);
    end;
end;

{---KEY PRESS ROUTINES---}

procedure EndKey;
var
    text: string[80];
begin
    with editarea do begin
        text := CurrentLine^.Line;
        while text[length(text)] = ' ' do
            delete(text,length(text),1);
        Xpos := length(text)+1;
        if Xpos > 80 then

```

```

        Xpos := 80;
        GotoXY(Xpos,Ypos);
    end;
end;

procedure CtrlHome;
begin
    with editarea do begin
        Xpos := 1;
        Ypos := 1;
        CurrentLine := FirstLine;
        GotoXY(Xpos,Ypos);
    end;
end;

procedure CtrlEnd;
begin
    with editarea do begin
        while (CurrentLine^.NextLine <> nil) and (Ypos < MAXY) do begin
            CurrentLine := CurrentLine^.NextLine;
            Ypos := Ypos + 1;
        end;
        EndKey;
    end;
end;

procedure CtrlPgup;
begin
    with editarea do begin
        Xpos := 1;
        Ypos := 1;
        FirstLine := filebuf^.FirstLine;
        CurrentLine := FirstLine;
        Display;
        GotoXY(Xpos,Ypos);
    end;
end;

procedure CtrlPgDn;
begin
    with editarea do begin
        while CurrentLine^.NextLine <> nil do
            CurrentLine := CurrentLine^.NextLine;
        FirstLine := CurrentLine;
        Xpos := 1;
        Ypos := 1;
        while (Ypos < MAXY) and (FirstLine^.PrevLine <> nil) do begin
            FirstLine := FirstLine^.PrevLine;
            Ypos := Ypos + 1;
        end;
        Display;
        EndKey;
    end;
end;

procedure InsKey;
begin
    with Editarea do begin
        InsertOn := not InsertOn;
        StatusLine;
    end;
end;

procedure DelKey;
begin
    with Editarea do begin
        if length(CurrentLine^.Line) >= Xpos then begin
            CurrentLine^.ChgFlag := true;
            ChgFlag := true;
        end;
    end;
end;

```

```

filebuf^.ChgFlag := true;
delete (CurrentLine^.Line.Xpos,1);
displayline;
GotoXY(Xpos,Ypos);
end
else if CurrentLine^.NextLine = nil then
  beep
else if Xpos + Length(CurrentLine^.NextLine^.Line) + 1 > 80 then
  beep
else begin
  CurrentLine^.ChgFlag := true;
  ChgFlag := true;
  filebuf^.ChgFlag := true;
  while Length(CurrentLine^.Line) < (Xpos-1) do
    CurrentLine^.Line := concat(CurrentLine^.Line, ' ');
  JoinLines;
  Display;
  GotoXY(Xpos,Ypos);
  end;
end;
end;

procedure DeleteLine;
begin
  with Editarea do begin
    CurrentLine^.ChgFlag := true;
    ChgFlag := true;
    filebuf^.ChgFlag := true;
    CurrentLine^.Line := '';
    dispoSettokens(CurrentLine^.FirstTok);
    if CurrentLine^.NextLine <> nil then begin
      CurrentLine^.FirstTok := CurrentLine^.NextLine^.FirstTok;
      CurrentLine^.NextLine^.FirstTok := nil;
      JoinLines;
      end;
    Display;
    GotoXY(Xpos,Ypos);
    end;
end;

procedure BkSpcKey;
var
  temp: LinePtr;
begin
  with Editarea do begin
    if Xpos > 1 then begin
      CurrentLine^.ChgFlag := true;
      ChgFlag := true;
      filebuf^.ChgFlag := true;
      delete (CurrentLine^.Line,Xpos-1,1);
      Xpos := Xpos - 1;
      displayline;
      GotoXY(Xpos,Ypos);
      end
    else if CurrentLine^.PrevLine = nil then
      beep
    else if Length(CurrentLine^.PrevLine^.Line) +
      Length(CurrentLine^.Line) > 80 then
      beep
    else begin
      CurrentLine^.PrevLine^.ChgFlag := true;
      ChgFlag := true;
      filebuf^.ChgFlag := true;
      if CurrentLine = FirstLine then
        FirstLine := FirstLine^.PrevLine
      else
        Ypos := Ypos - 1;
      CurrentLine := CurrentLine^.PrevLine;
      EndKey;
      JoinLines;
      display;
      GotoXY(Xpos,Ypos);
      end;
  end;
end;

```

```

end;
end;

procedure NextTokKey;
var
  t_ptr : tokenptr;
  l_ptr : lineptr;
  row   : integer;
begin
  row := 0;
  l_ptr := editarea.CurrentLine;
  t_ptr := l_ptr^.FirstTok;
  while (t_ptr <> nil) and (t_ptr^.StartCh <= editarea.Xpos) do
    t_ptr := t_ptr^.NextTok;
  while (t_ptr = nil) and (l_ptr <> nil) do begin
    row := row + 1;
    l_ptr := l_ptr^.NextLine;
    t_ptr := l_ptr^.FirstTok;
  end;
  if (t_ptr = nil) or (l_ptr = nil) then
    beep
  else
    MoveCursor(t_ptr^.StartCh - editarea.Xpos,row);
end;

procedure NextErrKey;
var
  t_ptr : tokenptr;
  l_ptr : lineptr;
  row   : integer;
begin
  row := 0;
  l_ptr := editarea.CurrentLine;
  t_ptr := l_ptr^.FirstTok;
  while (t_ptr <> nil) and
    ((t_ptr^.StartCh <= editarea.Xpos) or
     ((t_ptr^.Error = 0) and (t_ptr^.IdError = 0))) do
    t_ptr := t_ptr^.NextTok;
  while (t_ptr = nil) and (l_ptr <> nil) do begin
    row := row + 1;
    l_ptr := l_ptr^.NextLine;
    t_ptr := l_ptr^.FirstTok;
    while (t_ptr <> nil) and
      (t_ptr^.Error = 0) and
      (t_ptr^.IdError = 0) do
      t_ptr := t_ptr^.NextTok;
    end;
  if (t_ptr = nil) or (l_ptr = nil) then
    beep
  else
    MoveCursor(t_ptr^.StartCh - editarea.Xpos,row);
end;

procedure PrevTokKey;
var
  t_ptr : tokenptr;
  l_ptr,tgt_line : lineptr;
  col,row : integer;
begin
  col := 0;
  row := 0;
  l_ptr := editarea.CurrentLine;
  t_ptr := l_ptr^.FirstTok;
  while (t_ptr <> nil) and (t_ptr^.EndCh < editarea.Xpos) do begin
    col := t_ptr^.StartCh;
    t_ptr := t_ptr^.NextTok;
  end;
  while (col = 0) and (l_ptr <> nil) do begin
    row := row - 1;
    l_ptr := l_ptr^.PrevLine;
    t_ptr := l_ptr^.FirstTok;
    while (t_ptr <> nil) and (l_ptr <> nil) do begin
      col := t_ptr^.StartCh;
      t_ptr := t_ptr^.NextTok;
    end;
  end;

```

```

    end;
    if col = 0 then
        beep
    else
        MoveCursor(col-editarea.Xpos,row);
    end;

procedure PrevErrKey;
var
    t_ptr : tokenptr;
    l_ptr.tgt_line : lineptr;
    col,row : integer;
begin
    col := 0;
    row := 0;
    l_ptr := editarea.CurrentLine;
    t_ptr := l_ptr^.FirstTok;
    while (t_ptr <> nil) and (t_ptr^.EndCh < editarea.Xpos) do begin
        if (t_ptr^.Error <> 0) or (t_ptr^.IdError <> 0) then
            col := t_ptr^.StartCh;
            t_ptr := t_ptr^.NextTok;
        end;
    while (col = 0) and (l_ptr <> nil) do begin
        row := row - 1;
        l_ptr := l_ptr^.PrevLine;
        t_ptr := l_ptr^.FirstTok;
        while (t_ptr <> nil) and (l_ptr <> nil) do begin
            if (t_ptr^.Error <> 0) or (t_ptr^.IdError <> 0) then
                col := t_ptr^.StartCh;
                t_ptr := t_ptr^.NextTok;
            end;
        end;
        if col = 0 then
            beep
        else
            MoveCursor(col-editarea.Xpos,row);
    end;

procedure ReturnKey;
var
    temp: LinePtr;
begin
    with editarea do begin
        if InsertOn then begin
            CurrentLine^.ChgFlag := true;
            ChgFlag := true;
            filebuf^.ChgFlag := true;
            new(temp);
            temp^.PrevLine := CurrentLine;
            temp^.NextLine := CurrentLine^.NextLine;
            temp^.PrevLine^.NextLine := temp;
            if temp^.NextLine <> nil then
                temp^.NextLine^.PrevLine := temp;
            temp^.FirstTok := nil;
            temp^.ChgFlag := true;
            CurrentLine^.ChgFlag := true;
            temp^.Line := copy (CurrentLine^.Line,Xpos,80);
            delete(CurrentLine^.Line,Xpos,80);
            display;
            GotoXY(Xpos,Ypos);
            end;
        if CurrentLine^.NextLine = nil then
            beep
        else begin
            MoveCursor(-80,1);
            end;
        end;
    end;

end;

procedure CharKey (var Ch: char);
{any normal character}
var
    text: String[80];
begin
    with editarea do begin

```

```

text := CurrentLine^.Line;
if InsertOn then begin
  while Length(CurrentLine^.Line) < (Xpos-1) do
    CurrentLine^.Line := concat(CurrentLine^.Line, ' ');
  if (Length(CurrentLine^.Line) = 80) and
    (CurrentLine^.Line [80] <> ' ') then
    beep
  else begin
    Insert(Ch, CurrentLine^.Line, Xpos);
    Xpos := Xpos + 1;
    CurrentLine^.ChgFlag := true;
    ChgFlag := true;
    filebuf^.ChgFlag := true;
  end
end
else begin
  while Length(CurrentLine^.Line) < Xpos do
    CurrentLine^.Line := concat(CurrentLine^.Line, ' ');
  CurrentLine^.Line [Xpos] := Ch;
  Xpos := Xpos + 1;
  CurrentLine^.ChgFlag := true;
  ChgFlag := true;
  filebuf^.ChgFlag := true;
end;
if Xpos > 80 then begin
  Xpos := 80;
  beep;
end;
displayline;
GotoXY(Xpos, Ypos);
end;
end;

procedure InitEdit;
{set up the editor}
begin
  { write out borders and stuff }
  TextAttr := StatusColor;
  GotoXY(1,1);
  ClrEol;
  GotoXY(34,1);
  Write ('<Smart Edit>');
  GotoXY(1,25);
  ClrEol;
  Window(1,2,80,24);
  TextAttr := NormalColor;
  ClrScr;
  with editarea do begin
    new(filebuf);
    with filebuf^ do begin
      Name := 'NONAME.TXT';
      ChgFlag := false;
      new (FirstLine);
      with FirstLine^ do begin
        NextLine := nil;
        PrevLine := nil;
        Line := '';
        State.Comment := false;
        ChgFlag := false;
        new(FirstTok);
        with FirstTok^ do begin
          SrcLine := FirstLine;
          StartCh := 1;
          EndCh := 1;
          TokenId := EOFToken;
          Error := 0;
          IdError := 0;
          SymbRef := nil;
          NextTok := nil;
        end;
      end;
    end;
  end;
  FirstLine := filebuf^.FirstLine;
  CurrentLine := FirstLine;
  ChgFlag := true;

```

```

    ErrorFlg := false;
    Xpos := 1;
    Ypos := 1;
    display;
    GotoXY(Xpos,Ypos);
end;

end;

procedure Terminate;
{all done}
begin
    Window(1,1,80,25);
    TextAttr := DosColor;
    ClrScr;
end;

procedure OpenWindow(X1,Y1,X2,Y2,color:integer);
begin
    TextBackGround(Black);
    Window(X1+1,Y1+1,X2+1,Y2+1);
    ClrScr;
    TextBackGround(color);
    Window(X1,Y1,X2,Y2);
    ClrScr;
end;

procedure ErrorMessage (msg: string);
var
    x: char;
begin
    TextColor(Red);
    OpenWindow(75-Length(msg)-1,22,75,22,Red);
    TextColor(White);
    write (' ',msg);
    x := readkey;
    if x = #00 then
        x := readkey;
end;

procedure WarningMsg (msg: string);
var
    x: char;
begin
    TextColor(Yellow);
    OpenWindow(75-Length(msg)-1,22,75,22,Yellow);
    TextColor(Black);
    write (' ',msg);
end;

function Confirm (msg: string): boolean;
{ask a yes/no question}
var
    x: char;
begin
    beep;
    DirectScreen(25,10,length(msg),msg,ConfirmColor);
    x := #255;
    while x = #255 do begin
        x := readkey;
        case x of
            'Y','y': begin
                Confirm := true;
                x := #00;
                end;
            'N','n': begin
                Confirm := false;
                x := #00;
                end;
            else
                if x = #00 then
                    x := readkey;
                x := #255;
            end; {case}
        end; {while}
    DirectScreen(25,10,length(msg),' ',StatusColor);

```

```

end;

procedure LoadFile;
{get the file name to load}
var
  fn: string[80];
begin
  TextColor(Black);
  OpenWindow(20,10,60,12,LightGray);
  writeln;
  write(' Load file : ');
  Window(33,11,60,11);
  readln(fn);
  if FSearch(fn,'') = '' then begin
    ErrorMessage(concat('File ''',fn,''' not found'));
  end
  else begin
    ReadFile(fn);
  end;
  Display;
  GotoXY(editarea.Xpos,editarea.Ypos);
end;

procedure SaveFileAs;
{get the filename to write to}
var
  fn,msg: string[80];
  quit: boolean;
begin
  TextColor(Black);
  OpenWindow(20,10,60,12,LightGray);
  writeln;
  write(' Write file : ');
  quit := false;
  while not quit do begin
    fn := '';
    Window(34,11,60,11);
    readln(fn);
    if fn = '' then
      quit := true
    else if FSearch(fn,'') <> '' then begin
      msg := 'File '' + fn + '' already exists. Replace? (Y/N)';
      if Confirm(msg) then begin
        editarea.filebuf^.name := fn;
        WriteFile;
        quit := true;
      end
    end
    else begin
      editarea.filebuf^.name := fn;
      WriteFile;
      quit := true;
    end;
  end;
  Display;
  GotoXY(editarea.Xpos,editarea.Ypos);
end;

procedure Parse(var state: StateRec; TranslateFlag: boolean);
{EBNF PARSER: parse the entire edit buffer to ensure that it is a valid
EBNF spec. If TranslateFlag is on, also build the production tree}
var
  l_ptr: lineptr;
  t_ptr: tokenptr;
  syncflag: boolean;
  nexttoken: byte;
  token: tokenrec;

  procedure Consume;
  begin
    if nexttoken <> EOFToken then
      repeat
        if nexttoken = 0 then begin
          { set up first token }
          l_ptr := editarea.filebuf^.FirstLine;

```

```

    t_ptr := l_ptr^.FirstTok;
  end
else begin
  if (nexttoken = IdToken) and (not TranslateFlag) then
    Register(lang.SymbolTable,t_ptr,State.Declare);
  t_ptr := t_ptr^.NextTok;
  end;
  while t_ptr = nil do begin
    l_ptr := l_ptr^.NextLine;
    t_ptr := l_ptr^.FirstTok;
    end;
  token := t_ptr^;
  if not (token.Error in ScanErrors) then begin
    token.Error := 0;
    token.Expected := [];
    t_ptr^ := token;
    end;
  nexttoken := token.TokenId;
  until not (nexttoken in [CommentToken,IllegalCharToken]);
end;

procedure Match (id: integer);
begin
  if nexttoken <> id then begin
    state.ErrorFlg := true;
    { mark token with error msg }
    if token.error = 0 then begin
      token.Expected := [id];
      if nexttoken = EOFToken then
        token.error := PrematureEOF
      else if id in lang.SyncTokens then
        token.error := SkippingToNext
      else if nexttoken in lang.SyncTokens then
        token.error := InsertOne
      else if (t_ptr^.NextTok <> nil) and
        (t_ptr^.NextTok^.TokenId = Id) then
        token.error := SkipOne
      else
        token.error := InsertOne;
      t_ptr^ := token;
      end;
    if (id in lang.SyncTokens) or (token.error = SkipOne) then begin
      repeat
        Consume;
      until nexttoken in [Id,EOFToken];
      if nexttoken = Id then
        Consume
      else if token.error = 0 then begin
        token.error := PrematureEOF;
        token.Expected := [Id];
        t_ptr^ := token;
        end;
      end;
    end
  else if (nexttoken <> EOFToken) then begin
    Consume;
  end;
end;

{These are the normal productions}
procedure EBNF_prod; forward;
procedure PROD_prod; forward;
procedure EXPR_prod; forward;
procedure TERM_prod; forward;
procedure SYMB_prod; forward;

procedure EBNF_prod;
begin
  while nexttoken in [CompDirToken] do begin
    Match (CompDirToken);
  end;
  while nexttoken in [IdToken] do begin
    PROD_prod;
  end;
end;

```

```

    Match (10); { '_' }
    Match (EOFToken);
end;

procedure PROD_prod;
begin
    State.Declare := true;
    Match (IdToken);
    State.Declare := false;
    Match (7); { ::= }
    EXPR_prod;
    Match (8); { . }
end;

procedure EXPR_prod;
begin
    TERM_prod;
    while nexttoken in [9] do begin
        Match (9); { ; }
        TERM_prod;
    end;
end;

procedure TERM_prod;
begin
    SYMB_prod;
    while nexttoken in [IdToken,StringToken,CompDirToken.1,3,5] do begin
        SYMB_prod;
    end;
end;

procedure SYMB_prod;
begin
    if nexttoken in [IdToken] then begin
        Match (IdToken);
    end
    else if nexttoken in [StringToken] then begin
        Match (StringToken);
    end
    else if nexttoken in [CompDirToken] then begin
        Match (CompDirToken);
    end
    else if nexttoken in [1] then begin
        Match (1); { '(' }
        EXPR_prod;
        Match (2); { ')' }
    end
    else if nexttoken in [3] then begin
        Match (3); { '[' }
        EXPR_prod;
        Match (4); { ']' }
    end
    else if nexttoken in [5] then begin
        Match (5); { '{' }
        EXPR_prod;
        Match (6); { 'close curly brace' }
    end
    else {error trap}
        Match (IdToken);
end;

(These are the translating productions)
procedure EBNF_tran; forward;
procedure PROD_tran; forward;
procedure EXPR_tran (var n_ptr: NodePtr;follow: NodePtr); forward;
procedure TERM_tran (var n_ptr: NodePtr;follow: NodePtr); forward;
procedure SYMB_tran (var n_ptr,l_ptr: NodePtr); forward;

procedure EBNF_tran;
var
    directive: string;
begin
    while nexttoken in [CompDirToken] do begin
        directive := Upper(copy(token.SrcLine`.Line,

```

```

        token.StartCh,
        token.EndCh-token.StartCh+1));
delete(directive,1,length(lang.StartCompDir));
directive := copy(directive,1,
        length(directive)-length(lang.EndCompDir));
if copy(directive,1,13) = 'STARTCOMMENT=' then begin
    delete(directive,1,13);
    UserLang.StartComment := directive;
end
else if copy(directive,1,11) = 'ENDCOMMENT=' then begin
    delete(directive,1,11);
    UserLang.EndComment := directive;
end
else if copy(directive,1,11) = 'STRINGCHAR=' then begin
    delete(directive,1,11);
    UserLang.StringChar := directive[1];
end
else if copy(directive,1,13) = 'STARTCOMPDIR=' then begin
    delete(directive,1,13);
    UserLang.StartCompDir := directive;
end
else if copy(directive,1,11) = 'ENDCOMPDIR=' then begin
    delete(directive,1,11);
    UserLang.EndCompDir := directive;
end
else if copy(directive,1,12) = 'CHECKDECLARE' then begin
    UserLang.CheckDecl := true;
end
else if copy(directive,1,14) = 'NOCHECKDECLARE' then begin
    UserLang.CheckDecl := false;
end
else if copy(directive,1,8) = 'NUMERICS' then begin
    UserLang.Numerics := true;
end
else if copy(directive,1,10) = 'NONUMERICS' then begin
    UserLang.Numerics := false;
end;
Match (CompDirToken);
end;
while nexttoken in [IdToken] do begin
    PROD_tran;
end;
Match (10); { '_' }
Match (EOFToken);
end;

procedure PROD_tran;
var
    n_ptr: NodePtr;
    temphead,temptail: NodePtr;
begin
    n_ptr := token.SymbRef^.ProdPtr;
    if UserLang.StartNode = nil then begin
        UserLang.StartNode := NewNode(Prod);
        UserLang.StartNode^.NextNode := NewNode(NT);
        UserLang.StartNode^.NextNode^.AltNode := n_ptr;
        UserLang.StartNode^.NextNode^.NextNode := NewNode(EOFNode);
        n_ptr^.NextNode := NewNode(ProdRef);
        n_ptr^.NextNode^.AltNode := UserLang.StartNode^.NextNode;
    end;
    n_ptr^.SrcToken := t_ptr;
    Match (IdToken);
    Match (7); { ::= }
    temphead := UserLang.StartNode;
    while temphead^.AltNode <> nil do
        temphead := temphead^.AltNode;
    temphead^.AltNode := n_ptr;
    temptail := NewNode (nul);
    temphead := nil;
    EXPR_tran (temphead,temptail);
    temptail^.NextNode := n_ptr^.NextNode;
    n_ptr^.NextNode := temphead;
    Match (8); { . }
end;

```

```

procedure EXPR_tran(var n_ptr: NodePtr; follow: NodePtr);
var
  temp: NodePtr;
begin
  n_ptr := NewNode(Alt);
  TERM_tran(n_ptr^.NextNode, follow);
  temp := n_ptr;
  while nexttoken in [9] do begin
    temp^.SrcToken := t_ptr;
    Match (9); { ; }
    temp^.AltNode := NewNode(Alt);
    temp := temp^.AltNode;
    TERM_tran(temp^.NextNode, follow);
  end;
end;

procedure TERM_tran(var n_ptr: NodePtr; follow: NodePtr);
var
  temp: NodePtr;
begin
  SYMB_tran (n_ptr, temp);
  while nexttoken in [IdToken, StringToken, CompDirToken, 1, 3, 5] do begin
    SYMB_tran(temp^.NextNode, temp);
  end;
  temp^.NextNode := follow;
end;

procedure SYMB_tran(var n_ptr, l_ptr: NodePtr);
var
  directive, strlit: string;
  temp: nodePtr;
begin
  if nexttoken in [IdToken] then begin
    if token.SymbRef^.DeclCnt = 1 then begin
      n_ptr := NewNode(NT);
      n_ptr^.SrcToken := t_ptr;
      n_ptr^.AltNode := token.SymbRef^.ProdPtr;
      { Set up a reference back to this call from the tgt prod. }
      temp := n_ptr^.AltNode;
      while temp^.NextNode <> nil do
        temp := temp^.NextNode;
      temp^.NextNode := NewNode (ProdRef);
      temp^.NextNode^.AltNode := n_ptr;
    end
  else begin
    n_ptr := NewNode(Term);
    n_ptr^.SrcToken := t_ptr;
    if token.SymbRef^.IdText = '<IDENTIFIER>' then
      n_ptr^.TermId := IDToken
    else if token.SymbRef^.IdText = '<STRING>' then
      n_ptr^.TermId := StringToken
    else if token.SymbRef^.IdText = '<REAL>' then
      n_ptr^.TermId := RealToken
    else if token.SymbRef^.IdText = '<INTEGER>' then
      n_ptr^.TermId := IntToken
    else if token.SymbRef^.IdText = '<DIRECTIVE>' then
      n_ptr^.TermId := CompDirToken
    else begin
      n_ptr^.TermId := SimpleInsert(UserLang, token.SymbRef^.IdText);
      if syncflag then begin
        syncflag := false;
        UserLang.SyncTokens := UserLang.SyncTokens + [n_ptr^.TermId];
      end;
    end;
  end;
  Match (IdToken);
  l_ptr := n_ptr;
end
else if nexttoken in [StringToken] then begin
  n_ptr := NewNode(Term);
  strlit := Upper(copy(token.SrcLine^.Line,
    token.StartCh,
    token.EndCh-token.StartCh+1));
  delete(strlit, 1, length(lang.StringChar));
  strlit := copy(strlit, 1, length(strlit)-length(lang.StringChar));

```

```

n_ptr.TermId := SimpleInsert(UserLang.strlit);
if syncflag then begin
  syncflag := false;
  UserLang.SyncTokens := UserLang.SyncTokens + [n_ptr.TermId];
end;
l_ptr := n_ptr;
Match (StringToken);
end
else if nexttoken in [CompDirToken] then begin
  directive := Upper(copy(token.SrcLine`.Line,
                        token.StartCh,
                        token.EndCh-token.StartCh+1));
  delete(directive.l.length(lang.StartCompDir));
  directive := copy(directive.l,
                    length(directive)-length(lang.EndCompDir));
  if directive = 'DECLARE' then begin
    n_ptr := NewNode(DeclOn);
    l_ptr := n_ptr;
  end
  else if directive = 'NODECLARE' then begin
    n_ptr := NewNode(DeclOff);
    l_ptr := n_ptr;
  end
  else if directive = 'AFFINITY' then begin
    n_ptr := NewNode(Aff);
    l_ptr := n_ptr;
  end
  else begin
    n_ptr := NewNode(nul);
    l_ptr := n_ptr;
    if directive = 'SYNC' then
      syncflag := true;
    end;
  Match (CompDirToken);
  end
else if nexttoken in [1] then begin
  Match (1); { '(' }
  n_ptr := NewNode(Grp);
  l_ptr := NewNode(Nul);
  n_ptr.NextNode := l_ptr;
  EXPR_tran(n_ptr.AltNode.l_ptr);
  Match (2); { ')' }
  end
else if nexttoken in [3] then begin
  n_ptr := NewNode(Opt);
  l_ptr := NewNode(Nul);
  n_ptr.SrcToken := t_ptr;
  n_ptr.NextNode := l_ptr;
  Match (3); { '[' }
  EXPR_tran(n_ptr.AltNode.l_ptr);
  Match (4); { ']' }
  end
else if nexttoken in [5] then begin
  n_ptr := NewNode(Rpt);
  l_ptr := NewNode(Nul);
  n_ptr.SrcToken := t_ptr;
  n_ptr.NextNode := l_ptr;
  Match (5); { '{' }
  EXPR_tran(n_ptr.AltNode.n_ptr);
  Match (6); { 'close curly brace' }
  end
else {error trap}
  {We should NEVER get here.}
  Match (IdToken);
end;

begin {parse}
  with editarea do begin
    nexttoken := 0;
    syncflag := false;
    Match(0); {grab first token}
    if TranslateFlag then
      EBNF_tran
    else begin
      EBNF_prod;
    end
  end
end

```

```

        end
    end;
end; {parse}

function Equiv (st1,st2: staterec): boolean;
{are st1 and st2 equivalent?}
var
    i: byte;
    temp: boolean;
begin
    temp := (st1.comment = st2.comment) and
            (st1.Declare = st2.Declare) and
            (st1.Depth = st2.Depth) and
            (st1.Node = st2.Node);
    i := st1.Depth;
    while temp and (i > 0) do begin
        temp := (st1.Stack[i].Node = st2.Stack[i].Node);
        i := i - 1;
    end;
    Equiv := temp;
end;

procedure ParseLine (t_ptr: TokenPtr; var state: StateRec);
{INCREMENTAL PARSER: parse the current line of tokens, maintain staterec
to allow resumption with next line}
var
    token: TokenRec;

    procedure GetNextToken;
    begin
        while (t_ptr <> nil) and
            (t_ptr^.TokenId in [CommentToken,IllegalCharToken]) do begin
            t_ptr := t_ptr^.NextTok;
        end;
        token := t_ptr^;
        if (t_ptr <> nil) and not (token.Error in ScanErrors) then begin
            token.Error := 0;
            token.Expected := [];
            t_ptr := token;
        end;
    end;

function CheckNextToken(Allowed: SelectSet): boolean;
begin
    if (token.TokenId in Allowed) then
        CheckNextToken := true
    else begin
        CheckNextToken := false;
        state.ErrorFlg := true;
        { mark token with error msg }
        token.Expected := Allowed;
        if token.TokenId = EOFToken then
            token.error := PrematureEOF
        else if (Allowed*lang.SyncTokens <> []) then
            token.error := SkipOne
        else if token.TokenId in lang.SyncTokens then
            token.error := InsertOne
        else if (t_ptr^.NextTok <> nil) and
            (t_ptr^.NextTok^.TokenId in Allowed) then
            token.error := SkipOne
        else
            token.error := InsertOne;
        if t_ptr^.Error = 0 then
            t_ptr := token;
        if token.error = SkipOne then begin
            t_ptr := t_ptr^.NextTok;
            GetNextToken;
        end
        else
            state.Node := state.Node^.NextNode;
        end;
    end;
end;

procedure Match(id: byte);
begin

```

```

if CheckNextToken ([id]) then begin
  if (token.TokenId = IdToken) then
    Register(lang.SymbolTable.t_ptr,state.Declare);
  t_ptr := t_ptr^.NextTok;
  GetNextToken;
  state.Node := state.Node^.NextNode;
end
end;

procedure pushNode;
begin
  with state do begin
    if Depth = 32 then begin
      writeln(Debug,'StackOverflow');
      writeln('STACK OVERFLOW');
      FLUSH(debug);
      READLN;
      halt;
    end
    else begin
      { check for deja vu }
      i := Depth;
      while (i > 0) and (Stack[i].Token = t_ptr) do begin
        if Stack[i].Node = state.Node then begin
          { I have a funny feeling we've been here before...
            Skip the offending token, "pop" the stack down
            to the previous level, and continue }
          state.ErrorFlg := true;
          token.Error := SkipOne;
          token.Expected := [];
          if t_ptr^.Error = 0 then
            t_ptr := token;
          t_ptr := t_ptr^.NextTok;
          GetNextToken;
          Depth := i - 1;
        end;

        i := i - 1;
      end;
      Depth := Depth + 1;
      Stack[Depth].Node := state.Node;
      Stack[Depth].Token := t_ptr;
    end;
  end;
end;

procedure popNode;
begin
  with State do begin
    if Depth > 0 then begin
      Node := Stack[Depth].Node^.NextNode;
      Stack[Depth].Node := nil;
      Depth := Depth - 1;
    end;
  end;
end;

begin {ParseLine}
  GetNextToken;
  while t_ptr <> nil do begin
    while (t_ptr <> nil) and (state.Node <> nil) do begin
      with state.Node^ do begin
        case NodeType of
          Prod,Nul,Aff : state.Node := NextNode;
          ProdRef: popnode;
          Grp: state.Node := AltNode;
          Term: Match(TermId);
          EOFNode: Match(EOFToken);
          Rpt,Alt,Opt: begin
            if CheckNextToken(NextSelect+AltSelect) then
              if token.TokenId in AltSelect then
                state.Node := AltNode
              else
                state.Node := NextNode;
            end;
          end;
        end;
      end;
    end;
  end;
end;

```

```

    NT:   begin
          pushNode;
          state.Node := AltNode;
        end;
    DeclOn: begin {meta directive: IDs declared}
             state.Declare := true;
             state.Node := NextNode;
           end;
    DeclOff: begin {meta directive: IDs not declared}
              state.Declare := false;
              state.Node := NextNode;
            end;
          end; {case}
        end; {with}
      while (state.Node = nil) and (state.Depth <> 0) do
        popNode;
      end; {while}
    GetNextToken;
  end; {while}
end; {proc}

procedure ScanLine (ptr: LinePtr; var state: StateRec);
{ Scan the current line to build tokens}
var
  line.tokenval: string[80];
  ch: char;
  linepos: integer;
  endpos: integer;
  temp: TokenRec;
  tempptr: TokenPtr;

begin
  line := ptr^.Line;
  tempptr := nil;
  linepos := 1;
  while line <> '' do begin
    tokenval := '';
    temp.TokenId := 0;
    temp.SrcLine := ptr;
    temp.StartCh := linepos;
    temp.EndCh := 0;
    temp.Error := 0;
    temp.IdError := 0;
    temp.SymbRef := nil;

    if State.Comment or
       (copy(line,1,length(lang.StartComment)) =
        lang.StartComment) then begin
      { COMMENT }
      State.Comment := true;
      temp.TokenId := CommentToken;
      if copy(line,1,length(lang.StartComment)) =
         lang.StartComment then begin
        tokenval := tokenval + lang.StartComment;
        delete (line,1,length(lang.StartComment));
        ch := line[1];
        linepos := linepos + length(lang.StartComment);
      end;
      while (line <> '') and
            ((copy(line,1,length(lang.EndComment)) <> lang.EndComment) or
             (lang.EndComment = '')) do begin
        tokenval := tokenval + ch;
        delete (line,1,1);
        ch := line[1];
        linepos := linepos + 1;
      end;
      if line = '' then begin
        temp.EndCh := linepos - 1;
        if lang.EndComment = '' then
          State.Comment := false;
        end
      else begin
        temp.EndCh := linepos + length(lang.EndComment) - 1;

```

```

    linepos := linepos + length(lang.EndComment);
    delete (line,1,length(lang.EndComment));
    State.Comment := false;
end
end
else if line[1] = Lang.StringChar then begin
  { STRING LITERAL }
  endpos := 2;
  while (endpos <= length(line)) do begin
    if copy(line,endpos,2) =
      lang.StringChar + lang.StringChar then begin
      tokenval := tokenval + lang.StringChar;
      endpos := endpos + 2;
    end
    else if line[endpos] = lang.StringChar then begin
      delete(line,1,endpos);
      linepos := linepos + endpos;
      endpos := length(line) + 1;
      temp.TokenId := StringToken;
      temp.EndCh := linepos - 1;
    end
    else begin
      tokenval := tokenval + line[endpos];
      endpos := endpos + 1;
    end;
  end;
  if temp.TokenId = 0 then begin
    line := '';
    temp.TokenId := StringToken;
    temp.EndCh := linepos + endpos - 2;
    temp.Error := StringError;
  end;
end
else if copy(line,1,length(lang.StartCompDir)) =
  lang.StartCompDir then begin
  { COMPILER DIRECTIVE }
  temp.TokenId := CompDirToken;
  tokenval := line;
  delete(tokenval,1,length(lang.StartCompDir));
  endpos := pos(lang.EndCompDir,tokenval);
  if endpos = 0 then begin
    {illegal CompDir}
    tokenval := lang.StartCompDir;
    temp.Error := IllegalCompDir;
  end
  else begin
    tokenval := lang.StartCompDir +
      copy(line,1,endpos-1) +
      lang.EndCompDir;
  end;
  delete(line,1,length(tokenval));
  linepos := linepos + length(tokenval);
  temp.EndCh := linepos - 1;
end
else if (line[1] in ['0'..'9']) and (lang.Numerics) then begin
  { NUMERIC LITERAL }
  temp.TokenId := IntToken;
  endpos := 2;
  while (endpos <= length(line)) and
    (line[endpos] in ['0'..'9']) do
    endpos := endpos + 1;
  if (endpos <= length(line)) and (line[endpos] = '.') then begin
    i := SimpleLookup(lang,copy(line,endpos,lang.SimpleMaxLen));
    if (i = 0) or (lang.SimpleTokens[i] = '.') then begin
      temp.TokenId := RealToken;
      endpos := endpos + 1;
      while (endpos <= length(line)) and
        (line[endpos] in ['0'..'9']) do
        endpos := endpos + 1;
      end; { real }
    end;
  temp.EndCh := linepos + endpos - 2;
  linepos := linepos + endpos - 1;
  delete(line,1,endpos - 1)
end
end

```

```

else begin
  ch := line[1];
  case ch of
    ' ',TabChar: begin
      { WHITESPACE }
      delete(line,1,1);
      linepos := linepos + 1;
      end;
    '!'..'~': begin
      if not (ch in lang.SimpleSet) then begin
        { identifier }
        repeat
          tokenval := tokenval + ch;
          delete(line,1,1);
          ch := line[1];
          linepos := linepos + 1;
        until (line = '') or (ch in lang.SimpleSet + [' ',TabChar]);
        { check to make sure it is not a reserved word }
        temp.TokenId := SimpleLookup (lang.upper(tokenval));
        if temp.TokenId = 0 then
          temp.TokenId := IdToken;
        temp.EndCh := linepos - 1;
        end
      else begin
        { Simple Token }
        tokenval := copy(line,1,lang.SimpleMaxLen);
        temp.TokenId := SimpleLookup(lang.upper(tokenval));
        if temp.TokenId = 0 then begin
          { no simple token found }
          temp.TokenId := IllegalCharToken;
          tokenval := line[1];
          temp.Error := IllegalChar;
          end { if }
        else
          tokenval := lang.SimpleTokens[temp.TokenId];
          delete(line,1,length(tokenval));
          linepos := linepos + length(tokenval);
          temp.EndCh := linepos - 1;
          end; { else }
        end; { !..~ }
      end; { case }
    end; { else }
  if temp.TokenId <> 0 then begin
    if tempptr = nil then begin
      if ptr^.FirstTok = nil then begin
        new(ptr^.FirstTok);
        tempptr := ptr^.FirstTok;
        tempptr := temp;
        tempptr^.NextTok := nil;
        end
      else begin
        if ptr^.FirstTok^.TokenId = IdToken then
          UnRegister(lang.SymbolTable,ptr^.FirstTok);
        temp.NextTok := ptr^.FirstTok^.NextTok;
        ptr^.FirstTok := temp;
        tempptr := ptr^.FirstTok;
        end
      end
    else if tempptr^.NextTok = nil then begin
      new(tempptr^.NextTok);
      tempptr := tempptr^.NextTok;
      tempptr := temp;
      tempptr^.NextTok := nil;
      end
    else begin
      if tempptr^.NextTok^.TokenId = IdToken then
        UnRegister(lang.SymbolTable,tempptr^.NextTok);
      tempptr := tempptr^.NextTok;
      temp.NextTok := tempptr^.NextTok;
      tempptr := temp;
      end;
    if temp.Error <> 0 then
      state.ErrorFlg := true;
    end;
  end; { while }

```

```

{ Clean up the rest of the tokens }
if tempptr <> nil then
  disposeTokens (tempptr^.NextTok)
else
  disposeTokens (ptr^.FirstTok);
if ptr^.NextLine = nil then begin
  { add end of file token }
  temp.TokenId := EOFToken;
  temp.SrcLine := ptr;
  temp.StartCh := linepos;
  temp.EndCh := linepos;
  temp.Error := 0;
  temp.IdError := 0;
  if tempptr = nil then begin
    new(ptr^.FirstTok);
    tempptr := ptr^.FirstTok;
  end
  else begin
    new(tempptr^.NextTok);
    tempptr := tempptr^.NextTok;
  end;
  tempptr := temp;
  tempptr^.NextTok := nil;
end;

end; { ScanLine }

procedure ScanFile;
{incrementally scan text buffer. For USER mode, also call incr. parser.}
var
  ptr: LinePtr;
  state: StateRec;

begin
  state := InitialState;
  state.Node := lang.StartNode;
  with editarea do begin
    ptr := filebuf^.FirstLine;
    while ptr <> nil do begin
      if ptr^.ChgFlag then begin
        ptr^.State := state;
        ScanLine (ptr,state);
        if not lang.EBNFFlag then
          ParseLine (ptr^.FirstTok,state);
        ptr^.ChgFlag := false;
        if ptr^.NextLine <> nil then
          if not Equiv(state,ptr^.NextLine^.State) then begin
            ptr^.NextLine^.State := state;
            ptr^.NextLine^.ChgFlag := true;
          end
        end
      else
        if ptr^.NextLine <> nil then begin
          ptr^.NextLine^.State.ErrorFlg := state.ErrorFlg;
          state := ptr^.NextLine^.State;
        end;
        ptr := ptr^.NextLine;
      end;
      if lang.EBNFFlag then begin
        ErrorFlg := state.ErrorFlg;
        state := InitialState;
        state.ErrorFlg := ErrorFlg;
        Parse(state,false);
      end;
      ErrorFlg := state.ErrorFlg or SymbError(lang.SymbolTable);
    end;
  end;

end;

procedure disposenodes(var n_ptr: nodeptr);
begin
  if (n_ptr <> nil) and not (Sentinel in n_ptr^.AltSelect) then begin
    n_ptr^.AltSelect := [Sentinel];
    disposenodes(n_ptr^.NextNode);
    if n_ptr^.NodeType in [Prod.Alt,Rpt,Opt,Grp] then

```

```

        disposenodes(n_ptr^.AltNode);
    dispose(n_ptr);
    n_ptr := nil;
end;
end;

procedure Translate;
{translate the prod tree.}
var
    state: StateRec;
    s_ptr: SymbPtr;
    first: SelectSet;

    procedure CleanupAlts(var n_ptr: NodePtr);
    var
        temp: NodePtr;
    begin
        if (n_ptr <> nil) and (not n_ptr^.Visited) then begin
            if (n_ptr^.NodeType = Alt) and (n_ptr^.AltNode = nil) then begin
                temp := n_ptr;
                n_ptr := n_ptr^.NextNode;
                dispose(temp);
            end;
            if not (n_ptr^.NodeType in [ProdRef, EOFNode]) then begin
                n_ptr^.Visited := true;
                CleanupAlts(n_ptr^.NextNode);
                if n_ptr^.NodeType in [Alt, Rpt, Grp, Opt, Prod] then
                    CleanupAlts(n_ptr^.AltNode);
                n_ptr^.Visited := false;
            end;
        end;
    end;

    procedure FindFollow(var n_ptr: NodePtr;
                        var Inset: SelectSet);
    var
        temp: SelectSet;
    begin
        if (n_ptr <> nil) and not (Sentinel in n_ptr^.AltSelect) then
            with n_ptr^ do begin
                AltSelect := [Sentinel];
                if NodeType = Term then
                    Inset := Inset + [TermId]
                else if NodeType = EOFNode then
                    Inset := Inset + [EOFToken]
                else if NodeType = Prod then begin
                    FindFollow (NextNode.Inset);
                end
                else if NodeType in [Nul, DeclOn, DeclOff, Aff] then
                    FindFollow (NextNode.Inset)
                else if NodeType = ProdRef then begin
                    { The First Set we are computing contains the empty
                      production. Use the Production Reference Nodes
                      created during translation to compute the Follow
                      Set. }
                    FindFollow (AltNode^.NextNode.NextSelect);
                    FindFollow (NextNode.NextSelect);
                    Inset := Inset + NextSelect;
                end
                else if NodeType in [Opt, Alt, Rpt] then begin
                    FindFollow (AltNode, AltSelect);
                    FindFollow (NextNode, NextSelect);
                    Inset := Inset + NextSelect + AltSelect - [Sentinel];
                end
                else if NodeType in [Grp, NT] then begin
                    FindFollow (AltNode, Inset);
                end;
                AltSelect := AltSelect - [Sentinel]
            end
        else if (n_ptr <> nil) and
            (Sentinel in n_ptr^.AltSelect) and
            (n_ptr^.NodeType in [Rpt, Opt, Alt]) then begin
            { we have already seen this node, but we don't know which
              branch we went down. Since we'll stop at a prod or nt which
              contains Sentinel, go ahead and keep traversing. }

```

```

FindFollow (n_ptr^.AltNode,n_ptr^.AltSelect);
FindFollow (n_ptr^.NextNode,n_ptr^.NextSelect);
Inset := Inset + n_ptr^.NextSelect + n_ptr^.AltSelect - [Sentinel];
end

end;

procedure FindFirst(var n_ptr: NodePtr;
                   var Inset: SelectSet);
var
  temp: SelectSet;
begin
  if n_ptr <> nil then with n_ptr do begin
    if Sentinel in AltSelect then begin
      { Error: LeftRecursive Grammar };
      n_ptr^.SrcToken^.Error := LeftRecursive;
      state.ErrorFlg := true;
    end
    else if NodeType = Term then
      Inset := Inset + [TermId]
    else if NodeType = EOFNode then
      Inset := Inset + [EOFToken]
    else if NodeType = Prod then begin
      AltSelect := [Sentinel];
      FindFirst (NextNode,Inset);
      AltSelect := [];
    end
    else if NodeType in [Nul,DeclOn,DeclOff] then
      FindFirst (NextNode,Inset)
    else if NodeType = Aff then begin
      Inset := Inset + [Affinity];
      FindFirst (NextNode,Inset);
    end
    else if NodeType = ProdRef then begin
      { The First Set we are computing contains the empty
        production. Use the Production Reference Nodes
        created during translation to compute the Follow
        Set. }
      AltSelect := [Sentinel];
      FindFollow (AltNode^.NextNode,NextSelect);
      FindFollow (NextNode,NextSelect);
      AltSelect := AltSelect - [Sentinel];
      Inset := Inset + NextSelect - [Affinity];
    end
    else if NodeType in [Opt,Alt,Rpt] then begin
      AltSelect := [Sentinel];
      FindFirst (AltNode,AltSelect);
      FindFirst (NextNode,NextSelect);
      AltSelect := AltSelect - [Sentinel];
      if (AltSelect*NextSelect <> []) then begin
        { Error: Ambiguous Grammar };
        if (Affinity in NextSelect) and
           not (Affinity in AltSelect) then
          AltSelect := AltSelect - (AltSelect*NextSelect)
        else if (Affinity in AltSelect) and
           not (Affinity in NextSelect) then
          NextSelect := NextSelect - (AltSelect*NextSelect)
        else begin
          n_ptr^.SrcToken^.Error := Ambiguous;
          state.ErrorFlg := true;
        end;
      end;
      Inset := Inset + NextSelect + AltSelect;
    end
    else if NodeType in [Grp,NT] then begin
      AltSelect := [Sentinel];
      FindFirst (AltNode,Inset);
      AltSelect := AltSelect - [Sentinel]
    end;
  end;
end;

procedure Decorate(var n_ptr: NodePtr);
var

```

```

temp: SelectSet;
begin
  if (n_ptr <> nil) and
     not (n_ptr^.Visited) and
     not (n_ptr^.NodeType in [EOFNode,ProdRef]) then
    with n_ptr^ do begin
      Visited := true;
      if NodeType in [Prod,Opt,Alt,Rpt,NT] then
        FindFirst (n_ptr,temp);
        Decorate (NextNode);
      if NodeType in [Opt,Alt,Rpt,NT,Grp] then
        Decorate (AltNode);
    end;
end;

begin
ScanFile;
if editarea.ErrorFlg then
  beep
else if not lang.EBNFFlag then
  beep
else if editarea.FileBuf^.ChgFlag and
     not Confirm ('Translate: Are you sure?') then
  beep
else begin
  InitLang (UserLang);
  {pre-generate all the production nodes}
  s_ptr := lang.SymbolTable;
  while s_ptr <> nil do begin
    if s_ptr^.DeclCnt <> 0 then
      s_ptr^.ProdPtr := NewNode(Prod);
      s_ptr:=s_ptr^.NextSymb;
    end;
  state := InitialState;
  Parse(State,true);
  CleanupAlts (UserLang.StartNode);
  Decorate(UserLang.StartNode);
  s_ptr := lang.SymbolTable;
  while s_ptr <> nil do begin
    if (s_ptr^.DeclCnt <> 0) and
       (not s_ptr^.ProdPtr^.Visited) then begin
      { unreachable production }
      s_ptr^.ProdPtr^.SrcToken^.Error := Unreachable;
      state.ErrorFlg := true;
    end;
    s_ptr := s_ptr^.NextSymb;
  end;
  if state.ErrorFlg then begin
    editarea.ErrorFlg := true;
    disposenodes(UserLang.StartNode);
    editarea.Xpos := 1;
    editarea.Ypos := 1;
    editarea.FirstLine := editarea.FileBuf^.Firstline;
    editarea.CurrentLine := editarea.FirstLine;
    NextErrKey;
    DisplayErrorMsg;
  end
  else begin
    i := SimpleInsert(UserLang.UserLang.StartComment);
    i := SimpleInsert(UserLang.UserLang.StartComment);
    i := SimpleInsert(UserLang.UserLang.StringChar);
    UserLang.Grammar := editarea.FileBuf^.Name;
    ClearFile;
    Lang := UserLang;
  end;
  Display;
  StatusLine;
  GotoXY(editarea.Xpos,editarea.Ypos);
end;
end;

begin {main}
  InitLang (EBNF);
  with EBNF do begin

```

```

EBNFFlag := true;
CheckDecl := false;
Numerics := false;
Grammar := 'native';

i := SimpleInsert(EBNF, '(');
i := SimpleInsert(EBNF, ')');
i := SimpleInsert(EBNF, '[');
i := SimpleInsert(EBNF, ']');
i := SimpleInsert(EBNF, '{');
i := SimpleInsert(EBNF, '}');
i := SimpleInsert(EBNF, ':');
i := SimpleInsert(EBNF, ':=');
i := SimpleInsert(EBNF, '.');
i := SimpleInsert(EBNF, '|');
i := SimpleInsert(EBNF, '_');
i := SimpleInsert(EBNF, '_|_');

i := SimpleInsert(EBNF, StartComment);
i := SimpleInsert(EBNF, StartCompDir);
i := SimpleInsert(EBNF, StringChar);

SyncTokens := [EOFToken.10.8];

AddStandardId(SymbolTable, '<IDENTIFIER>');
AddStandardId(SymbolTable, '<STRING>');
AddStandardId(SymbolTable, '<REAL>');
AddStandardId(SymbolTable, '<INTEGER>');
AddStandardId(SymbolTable, '<DIRECTIVE>');

end;

Lang := EBNF;

Assign (Debug, 'debug.out');
Rewrite (Debug);
ClrScr;
InitEdit;
if ParamCount > 0 then begin
  if FSearch(ParamStr(1), '') = '' then begin
    ErrorMsg(concat('File ', ParamStr(1), ' not found'));
  end
  else begin
    ReadFile(ParamStr(1));
  end;
end;

Display;
CtrlHome;
Editarea.InsertOn := true;
Editarea.ErrorFlg := false;
InitialState.Comment := false;
InitialState.Declare := false;
InitialState.ErrorFlg := false;
InitialState.Depth := 0;

StatusLine;

quit := false;
repeat
  GetTime(hr, min, sec, hun);
  starttime := hun + integer(sec)*100;
  endtime := starttime;
  { wait for a bit to allow the user to type more. }
  while not KeyPressed and
    editarea.ChgFlag and
    ((endtime - starttime) mod 6000 < 100) do begin
    GetTime(hr, min, sec, hun);
    { in turbo, -1 mod 60 = -1 -- not 59 !
      add 60 seconds to make sure we get a positive difference. }
    endtime := 6000 + hun + integer(sec)*100;
  end;
  { Time's up. If nothing more to process, then go ahead and parse. }
  if (not KeyPressed) then begin
    if editarea.ChgFlag then begin
      DirectScreen(25, 10, 60, 'Scanning', StatusColor);
      ScanFile;
    end;
  end;
until quit;

```

```

UpdateDisplay:
StatusLine;
Editarea.ChgFlag := false;
end;
DirectScreen(25,10,60, '', StatusColor);
DisplayErrorMsg;
end;
DirectScreen(25,75,6, inttostr(MemAvail), StatusColor);
Ch := ReadKey;
DirectScreen(25,10,60, '', StatusColor);
case (Ch) of
  {Extended Key} #00: begin
    Ch := ReadKey;
    case (Ch) of
      {Home}      #S47: MoveCursor(-80,0);
      {Up}        #S48: MoveCursor(0,-1);
      {PgUp}      #S49: MoveCursor(0,1-MAXY);
      {Left}      #S4B: MoveCursor(-1,0);
      {Right}     #S4D: MoveCursor(1,0);
      {End}       #S4F: EndKey;
      {Down}      #S50: MoveCursor(0,1);
      {PgDn}      #S51: MoveCursor(0,MAXY-1);
      {Ins}       #S52: InsKey;
      {Del}       #S53: DelKey;
      {CtrlLeft} #S73: PrevTokKey;
      {CtrlRight} #S74: NextTokKey;
      {CtrlEnd}  #S75: CtrlEnd;
      {CtrlPgDn} #S76: CtrlPgdn;
      {CtrlHome} #S77: CtrlHome;
      {CtrlPgUp} #S84: CtrlPgup;
      {F1}       #S3B: begin end;
      {F2}       #S3C: begin end;
      {F3}       #S3D: begin end;
      {F4}       #S3E: begin end;
      {F5}       #S3F: begin end;
      {F6}       #S40: begin end;
      {F7}       #S41: PrevErrKey;
      {F8}       #S42: NextErrKey;
      {F9}       #S43: begin end;
      {F10}      #S44: begin end;
      {AltE}     #S12:
        if not editarea.Filebuf^.ChgFlag or
          Confirm('EBNF Mode: Are you Sure?') then begin
          DisposeNodes (lang.StartNode);
          ClearFile;
          lang := EBNF;
          Display;
          StatusLine;
          GotoXY(1,1);
          end;
        {AltS}     #S1F: WriteFile;
        {AltT}     #S14: Translate;
        {AltW}     #S11: SaveFileAs;
        {AltN}     #S31:
          if not editarea.Filebuf^.ChgFlag or
            Confirm('Newfile: Are you Sure?') then begin
            ClearFile;
            Display;
            GotoXY(editarea.Xpos,editarea.Ypos);
            end;
        {AltL}     #S26: {load}
          if not editarea.filebuf^.ChgFlag or
            Confirm('Load File: Are You Sure?') then
            LoadFile;
        {AltX}     #S2D: {exit}
          if not editarea.filebuf^.ChgFlag or
            Confirm('Exit: Are You Sure?') then
            quit := true;
        else
        end;
    end; {Extended Key}
  {NormalKey} ' '..'~'.TabChar: CharKey(Ch);
  {Bkspc}      #S08: BkspcKey;
  {Return}     #S0D: ReturnKey;
  {CtrlY}      #S19: DeleteLine;

```

```
    else
    end; {case}
until quit;
Terminate;
Close (Debug);
end.
```