The Design and Implementation of a Parser and Scanner for the MATLAB Language in the MATCH Compiler

Pramod G. Joisha *Abhay Kanhere* *Prithviraj Banerjee*
U. Nagaraj Shenoy *Alok Choudhary*

Technical Report No. CPDC–TR–9909–017

September 1999

Center for Parallel and Distributed Computing
Department of Electrical & Computer Engineering,
Technological Institute,
2145 Sheridan Road,
Northwestern University,
IL 60208–3118.

The Design and Implementation of a Parser and Scanner for the MATLAB Language in the MATCH Compiler*

Pramod G. Joisha Abhay Kanhere Prithviraj Banerjee
U. Nagaraj Shenoy Alok Choudhary

Center for Parallel and Distributed Computing, Electrical and Computer Engineering Department, Technological Institute, 2145 Sheridan Road, Northwestern University, IL 60208–3118.
Phone: (847) 467-4610, Fax: (847) 491-4455
Email: `[pjoisha, abhay, banerjee, nagaraj, choudhar]@ece.nwu.edu`

Abstract

In this report, we present the design and implementation of the MATCH compiler front-end. We discuss in detail both the indigenously designed grammar responsible for syntax analysis as well as the lexical specification that complements the grammar. In the course of our attempts to emulate MATLAB's syntax, we were able to unravel certain key issues relating to its syntax, such as the parsing of command-form function invocations and how the single quote character is differently construed depending on the context. The front-end effects a conversion of the original source to an intermediate form in which statements are represented as abstract syntax trees and the flow of control between statements by a control-flow graph. All subsequent compiler passes work on this intermediate representation.

Keywords syntax analysis for MATLAB, command-form function invocations, single quote character, matrices, colon expressions, assignments, control constructs

1 Introduction

The MATCH project [MAT] concerns itself with the task of efficiently compiling code written in MATLAB[b] for a heterogeneous target system comprising embedded and commercial-off-the-shelf (COTS) processors. Since the language is proprietary, the project also faced the additional onus of designing the grammar and the lexical specification for it, in addition to actually implementing the specifications using publically available automatic parser and scanner generators.

MATLAB is a high performance language geared toward technical computing [Mat97]. The language provides powerful features that among other things, enable matrices and arrays to be efficiently and easily manipulated. These features are also high level and this makes the usage of the language very intuitive. More importantly, programs written in this language are usually interpreted.

*This research was supported by DARPA under Contract F30602–98–2–0144.

[b]MATLAB is a registered trademark of The MathWorks, Inc.

From the perspective of syntax analysis, MATLAB offers numerous challenges whose subtlety make them interesting exercises in parser design.[c] Examples of these include tackling the single quote character, efficiently building uniform colon expressions and handling matrix constructs (see § 6.2, 6.3 and 6.1). In fact, some of the language's attributes make parsing in a compilation environment a much more complicated task (see § 5) and this resulted in certain modifications to the set of language features that were finally supported by our compiler.

2 Background

Work in building a front-end began by experimenting with the Free Software Foundation's distribution for GNU Octave [OCT], a language having much of MATLAB's syntactic and semantic features. Beginning with Octave's grammar, a core set of productions were retained and modified and many more added to capture MATLAB's syntax as faithfully as possible. The lexical specification was written from scratch.

The front-end currently supports only a proper subset of the MATLAB language. Support for structures and cell arrays is presently unavailable in the front-end. Furthermore, the current version of the parser recognizes expressions, assignments, `for` loops, `if` statements, `global` declarations, `while` loops, `return` statements and a limited form of function invocations in the command form. Both functions as well as scripts can be processed by the front-end. Additional constructs such as `switch` statements can be easily handled with relatively little modification to the current grammar.

The expository approach that we take in this report is to present the lexical specification and the context-free grammar written for flex and bison [FXB]. We then dissect the specification and the grammar, explaining the lexical and parser rules in detail. The focus will be on the syntactic points and the rules *per se*, with descriptions of some of the semantic actions. The complete source code containing the lexical specification and the context-free grammar is available as an appendix to this report.

The main tools that were employed in implementing the front-end were bison and flex. Bison is an automatic parser generator in the style of yacc [Joh75] (see `yacc(1)`). Flex, which is a contraction of "Fast Lexical Analyzer," is an automatic scanner generator that was implemented as a rewrite of AT&T's lex tool [Les75] (see `lex(1)`), with some useful extensions over lex as well as some incompatibilities. While bison was primarily written by Richard Stallman as part of the GNU project, flex was authored by Vern Paxson when at Lawrence Berkeley Laboratory.

2.1 Bison

Both bison and yacc convert a description of an LALR(1) context-free grammar [ASU88] to a C program called the parser. The parser accepts strings belonging to the language generated by the grammar and rejects all other strings. The grammar can be ambiguous and precedences are used to resolve the resulting conflicts. The grammar file contains the language specification and this forms the input to both bison and yacc.

The grammar file essentially consists of four sections. The first section usually contains `#include` lines, macro definitions and C declarations of variables used in the generated parser. The second section contains bison- or yacc-related declarations. These could be token declarations and operator precedences among others. The third section contains the actual grammar rules and their associated semantic actions, and the last section contains additional C code that commonly corresponds to functions invoked from the semantic actions.

2.2 Flex

Both flex and lex are used to generate programs called scanners or lexical analyzers that match lexical patterns occurring in textual input. Parsers use the services of the scanner to process an input stream. Parsers typically work at the level of *tokens* that correspond to character sequences in the input called *lexemes*. Lexemes for a particular token conform to a particular lexical pattern and this is represented by a *regular expression* [ASU88].

[c] So much so, that many of these issues were overlooked in the first version of the front-end.

The input file to flex and lex comprises three sections. The first section is called the *definitions section* and contains declarations of *name definitions* as well as start conditions. Name definitions are basically shorthands that simplify the main scanner specification. The rules that constitute the chief part of the lexical specification are contained in the second section of the input file. The lexical rules themselves have two parts: an *extended regular expression* (see `regexp(5)`) that denotes a lexical pattern and an *action* that contains user code that must be executed whenever that specific lexical pattern is detected in the input. The third section contains additional C code that once again usually corresponds to functions that are invoked from the action parts of the lexical rules.

3 Language Preliminaries

A MATLAB identifier consists of a letter followed by zero or more underscores, letters or digits. A MATLAB numeric quantity can be free of a decimal point and an exponent, or consist of either a decimal point or an exponent or both. In the MATCH lexical specification, the name definitions `INTEGER` and `DOUBLE` correspond to the former and latter respectively. For example, the character sequences `1e-2` and `1.` associate with the name definition `DOUBLE`, while the character sequence `1` associates with the name definition `INTEGER`.

```
HSPACE                         [ \t]
HSPACES                        {HSPACE}+
NEWLINE                        \n|\r|\f
NEWLINES                       {NEWLINE}+


ELLIPSIS                       \.\.\.
CONTINUATION                   {ELLIPSIS}[^\n\r\f]*{NEWLINE}?


COMMENT                        \%[^\n\r\f]*{NEWLINE}?


IDENTIFIER                     [a-zA-Z][_a-zA-Z0-9]*


DIGIT                          [0-9]
INTEGER                        {DIGIT}+
EXPONENT                       [DdEe][+-]?{DIGIT}+
MANTISSA                       ({DIGIT}+\.)|({DIGIT}*\.{DIGIT}+)
FLOATINGPOINT                  {MANTISSA}{EXPONENT}?
DOUBLE                         ({INTEGER}{EXPONENT})|{FLOATINGPOINT}
NUMBER                         {INTEGER}|{DOUBLE}
IMAGINARYUNIT                  [ij]
```

In MATLAB, there exist a couple of situations in which horizontal spaces become significant. Apart from functioning as token demarcators and element separators in certain circumstances, horizontal spaces can also influence the interpretation of a succeeding character sequence in certain other cases. They are cast away by the scanner, so that the parser sees a token stream free of any horizontal space. A *horizontal space* is either a blank or a horizontal tab and is denoted by the name definition `HSPACE` shown above. We shall often use the symbol ␣ to represent the lexical pattern matched by the `HSPACES` definition (which is *at least* one horizontal space).

Input lines in MATLAB can be continued onto multiple lines. This "breaking" of long statements is accomplished by using a contiguous sequence of three periods, subsequently followed by a new-line, carriage return or form-feed character (i.e., a `NEWLINE` lexical pattern). Everything beginning from the ellipsis until and including the `NEWLINE` character, *or until the end of the input*, are ignored.[d] Comments likewise begin at a percent character ('%') and continue until a `NEWLINE` character, or until the end of the input.

4 M-Files

Input files that contain code written in MATLAB are called *M-files.* M-files can either be *functions* (which *may* accept input arguments and which *may* return output arguments), or scripts (which neither accept inputs nor produce outputs). The main distinction between functions and scripts is that while the former execute in a workspace independent of the caller's environment, the latter execute in the caller's workspace. Syntactically, functions and scripts are the same, except that the first *nonempty* line in a function must be the *function definition line.* A `LINE` token, which is returned by the lexical analyzer to the parser whenever a `COMMENT` or a `NEWLINES` lexical pattern is scanned by it, signifies an empty line.

```
input                   : scriptMFile
                        | functionMFile
                        | parse_error
                        ;

scriptMFile             : opt_delimiter
                        | opt_delimiter statement_list
                        ;

functionMFile           : empty_lines f_def_line f_body
                        | f_def_line f_body
                        ;

f_def_line              : FUNCTION f_output '=' IDENTIFIER f_input
                        | FUNCTION IDENTIFIER f_input
                        ;

f_body                  : delimiter statement_list
                        | opt_delimiter
                        ;
```

A MATLAB program basically consists of a sequence of statements. Statements are separated from each other by *delimiters.* Delimiters are sequences consisting of an arbitrary mixture of comma (','), semicolon

[d]At the time of this writing, the complete line continuation specifics for MATLAB seemed nonuniform and were unclear. This is because in certain cases, the line continuation sequence behaves as a token demarcator, while not in other cases. For instance, while the following lines

```
disp1=2;
disp...
1
```

result in the value of `disp1` being displayed, the following lines

```
disp1=2;
a=disp...
1
```

result in an error situation when an assignment to `a` is attempted. More esoteric behavior is observed when the line continuation sequence occurs within matrices and among MATLAB keywords such as `global` and `for`.

(';') and LINE tokens.

```
opt_delimiter           :
                        | delimiter
                        ;

delimiter               : null_lines
                        | empty_lines
                        | null_lines empty_lines

null_lines              : null_line
                        | null_lines null_line
                        ;

null_line               : ','
                        | ';'
                        | empty_lines ','
                        | empty_lines ';'
                        ;

empty_lines             : LINE
                        | empty_lines LINE
                        ;

statement_list          : statement opt_delimiter
                        | statement delimiter statement_list
                        ;

parse_error             : LEXERROR
                        | error
                        ;

statement               : command_form
                        | expr
                        | assignment
                        | for_command
                        | if_command
                        | global_command
                        | while_command
                        | return_command
                        ;
```

A program file that is a script may either contain optional delimiters (`opt_delimiter`), or contain a sequence of statements (`statement_list`) optionally preceded by delimiters. Note that `error` is a special token reserved by bison for error handling; the parser generates this token whenever a syntax error occurs. Thus, a reduction to the `parse_error` nonterminal occurs either when the lexical analyzer detects an invalid character sequence, in which case, the token `LEXERROR` is returned by it, or if the parser discovers a syntax error. Thus, as part of the semantic action of the production `input` $\rightarrow$ `parse_error`, error messages may be emitted.[e]

[e] While discussing the rules, the semantic actions will be omitted in most cases. The source code in the appendix shows the rules as well as the associated actions in their entirety.

A statement in MATLAB can either be a command-form function invocation, an expression, an assignment, a control construct or a `global` declaration. Among the control constructs, the `for` loop, the `if` statement, the `while` loop and the `return` statement are currently supported. Also, support for cell arrays and structures currently does not exist.

5 Commands

In MATLAB, functions can either be invoked in the functional form or in the command form. For instance, we could invoke the built-in function `type` either like `type('rank')`, which corresponds to the functional form, or like `type␣rank`, which corresponds to the command form. A statement such as `type␣'rank'` also qualifies as a command-form function invocation and is equivalent to `type␣rank`.

```
command_form            : identifier text_list
                        ;

text_list               : TEXT
                        | text_list TEXT
                        ;

identifier              : IDENTIFIER
                        ;
```

In the MATCH compiler, sentences that derive from the `command_form` nonterminal shown above require arguments to the function to be explicitly quoted. That is, only strings such as `type␣'rank'` reduce to the `command_form` nonterminal. Sentences such as `type␣rank` are not recognized as command-form function invocations and result in a parse error. The following MATLAB code fragment provides the reason behind the imposition of this constraint.

```
% Suppose that a function M-file called
% A.m exists in the MATLAB M-file search
% path.

if (rand > 0.50)
   A=1;
end;

A +1;
```

Control can reach the statement `A␣+1;` either after executing the control construct, in which case, `A` will be treated as a variable, or without executing the `if` statement body, in which case, `A` will be regarded as a function. Thus, on reaching the statement `A␣+1;`, it is not clear to the compiler whether to parse the statement as a function invocation in which the function `A` is invoked with the string literal `+1` as the only argument, or as the binary addition of the variable `A` with the numeric constant `1`. Hence, both the abstract syntax trees (ASTs) shown in Figures 1 and 2 qualify as possible parse trees for the above statement.

It must be emphasized that this is a *parsing* problem in addition to an inferencing one. This problem is peculiar to the compiler and the MATLAB interpreter is not faced with the same predicament because the parsing of `A␣+1;` begins only after control reaches it, by which time the parser is aware of the nature of `A`. An interesting sidebar to this discussion is that if `A␣+1;` were the only statement in the above code fragment and if `A.m` were a function M-file in the MATLAB search path, MATLAB *always* parses it into the AST shown in Figure 1. That is, the statement is actually considered to be a command-form function invocation. On the other hand, if the statement were `A+1;`, MATLAB will always parse it into the AST shown in Figure 2. In

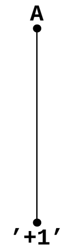

FIGURE 1: Function Invocation

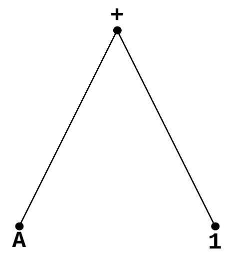

FIGURE 2: Binary Addition

this case, the interpretation is that the value returned by an invocation of the function A without arguments is added to the numeric constant 1.[f]

A strategy by which our compiler could have supported the interpreter's full-fledged command-form function invocation syntax would have been by creating and maintaining both ASTs against a statement when in doubt. Apart from the fact that the parser would now have to detect such syntactic ambiguities, the creation and maintenance of two ASTs would have further complicated analysis by subsequent compiler passes. Hence, it was decided to eliminate the potential for such parsing ambiguities by supporting a limited version of MATLAB's command-form function invocation syntax.

The scanner returns the token TEXT whenever a string literal is matched by it. A string literal in MATLAB consists of a sequence of characters enclosed between a pair of single quotes. To include a single quote as a character in a string literal, its meaning as a string demarcator must first be escaped and this is done by preceding the single quote by another single quote. Thus, the character sequence 'Giordano''s⊔Pizzas!' actually corresponds to the string literal Giordano's⊔Pizzas!.[g] Hence, there are two extended regular expressions that are together responsible for the TEXT token in the lexical specification. The first (i.e., <INITIAL>'[^'\n\r\f]*'/') enables single quotes to be escaped, while the second (i.e., <INITIAL>'[^'\r\f\n]*') actually causes the scanner to return the TEXT token. The presence of the prefix <INITIAL> to both of these extended regular expressions indicates that the match will only occur when the scanner is in the INITIAL *start condition* (see flex(1)). The reason for this "selective" matching is because the single quote character is also used as the complex conjugate transpose operator [Mat97] and this dual role can lead to lexical matching problems; this issue is further elaborated in § 6.2.

6 Expressions

Expressions in MATLAB are composed as operations on subexpressions; ultimately, they consist of colon "atoms," string literals, identifiers and quantities that could either be numeric or imaginary. The MATCH lexical specification enables two types of numeric quantities to be identified (INTEGER and DOUBLE). In addition, the scanner identifies any number (i.e., NUMBER) followed immediately by the imaginary unit as an imaginary quantity. The imaginary unit is specified by either of the characters 'i' or 'j'. This is a lexical match and is not affected by any reaching definition against the variables i or j. For instance, the character sequence

[f] A command-form invocation of a function occurs whenever the first token on the MATLAB input line is an identifier that corresponds to a function, and if the second token is separated from the first by horizontal spaces. The only exception to this rule is when the second token is an opening parenthesis. In this case, the function is invoked only after evaluating a parenthesized expression. For example, the following lines

```
disp␣1+2;
disp␣(1+2);
```

display 1+2 and 3 as their respective results.

[g] We use the symbol ⊔ to explicitly represent a horizontal space (i.e., HSPACE). As before, the symbol ␣ will be used to denote the lexical pattern matched by the HSPACES definition.

2j will always be considered as an imaginary quantity irrespective of any preceding definitions against the variable j.

6.1 Matrices

The constructions that MATLAB offers to represent matrices are very user-friendly. Though quite intuitive, these constructions add significance to the actual textual layout of the matrix, making the recognition of these structures much harder for the parser-scanner duo. For example, the following lines define a 3 by 3 matrix having `1`, `2`, and `-3+4` as the elements in the first row, `0.1`, `+.1i` and `.2` as the elements in the second row, and `a`, `(3)` and `b(3)` as the elements in the third row. What should be noted here is that the second element in the first row is *not* `2-3+4`, that the first element in the second row is *not* `0.1+.1i` and that the first element in the third row is *not* `a(3)`.

```
[1, 2 -3 + 4,
0.1 +.1i .2
a (3) b(3);]
```

A matrix is formed by surrounding one or more rows of elements with box brackets (`[`, `]`). Each row except the last must be terminated either by a semicolon or a `LINE` token.

```
matrix                  : '[' boxes1 rows
                        {
                         BracketDelimiter = $2;
                         } ']'
                        ;

boxes1                  :
                        {
                         $$ = BracketDelimiter;

                         BracketDelimiter = '[';
                         }
                        ;

rows                    :
                        | row
                        | rows ';'
                        | rows ';' row
                        | rows LINE
                        | rows LINE row
                        ;

row                     : expr
                        | row_with_commas
                        | row_with_commas expr
                        ;

row_with_commas         : expr ','
                        | row_with_commas expr ','
                        ;
```

A row could either be empty or consist of a sequence of one or more elements separated from each other by a comma. A row containing at least one element could have a comma after the last element. Finally, elements are themselves expressions.

The parser does not see any horizontal space. By working in tandem, the parser and lexical analyzer either insert commas between the yet-to-be scanned matrix elements or convert horizontal spaces among the yet-to-be scanned matrix elements to commas. To see how this is achieved, we must first understand how horizontal spaces among matrix elements affect the parsing process in MATLAB.

Within matrices, commas can be potentially inserted immediately after a numeral, an identifier, an imaginary quantity, a string literal, a closing parenthesis (`)`), a transpose operator (`.'`), a complex conjugate transpose operator (`'`) or a closing box bracket (`]`). Horizontal spaces among matrix elements are significant only if the preceding token is *immediately* enclosed within a pair of box brackets. We then say that the *bracket nesting* of the preceding token is '`[`'. For example, given the input stream `[1␣+2];`, there are six tokens that the scanner returns to the parser: `[`, `1`, `+`, `2`, `]` and `;`. The bracket nesting of the first token is considered to be null, that of the next three '`[`', and that of the last two null.

The `boxes1` nonterminal shown above, along with the `parenthesis` and `boxes2` nonterminals, keeps track of the current bracket nesting. Reductions to these nonterminals always occur without consulting a lookahead token.[h] That is, the productions `boxes1` $\rightarrow$ ϵ,[i] `parenthesis` $\rightarrow$ ϵ and `boxes2` $\rightarrow$ ϵ are applied immediately after an opening box bracket or parenthesis is seen and immediately before the next token is requested from the lexical analyzer. Thus, as part of the semantic action of these productions, the current bracket nesting is stored on the stack. This is what the line,

```
$$ = BracketDelimiter;
```

in the semantic action of the `boxes1` production does. The global variable `BracketDelimiter` can be queried to find the current bracket nesting. The new bracket nesting is then set to either '`(`' or '`[`'. For instance, the line

```
BracketDelimiter = '[';
```

in the semantic action of the `boxes1` production sets `BracketDelimiter` to '`[`'. The *mid-rule action* (see `bison(1)`) in the `matrix` production properly resets `BracketDelimiter` by popping the previous bracket nesting off the stack whenever a closing box bracket occurs as the next token in the input stream.

The function `yyinsert_comma_in_input()` is invoked by the parser whenever the following productions are applied by the parser.

```
expr → DOUBLE
expr → IMAGINARY
expr → TEXT
expr → '(' parenthesis expr ')'
expr → expr CTRANSPOSE
expr → expr TRANSPOSE

reference → identifier '(' parenthesis argument_list ')'

matrix → '[' boxes1 rows ']'
```

Whenever any of the reductions corresponding to these productions is performed, the parser does not consult a lookahead token. Consequently, the semantic actions of these productions are ideal locations to incorporate source code to either actually convert any immediately following horizontal space in the input to a comma or to simply insert a comma. This is exactly what the `yyinsert_comma_in_input()` calls accomplish. The argument to this function is an integer that designates the last token processed by the scanner. In other words, this is the last token matched by the lexical analyzer and that is either returned, or will be returned, by `yylex()` (see `flex(1)`) to the parser. There exist two more `yyinsert_comma_in_input()` call-sites. These are in the action parts of the `{INTEGER}` and `{IDENTIFIER}` extended regular expressions in the lexical specification.

[h] This can be verified by analyzing the grammar as well as by consulting the verbose description file (having the extension `.output`) generated by bison.

[i] The *empty string* is denoted by ϵ.

A few points are in order regarding `yyinsert_comma_in_input()`.[j] The function first checks to see if the current bracket nesting is either '[' or LD (see § 7). If not, the function immediately returns, allowing for the usual processing of horizontal spaces by the lexical analyzer. Otherwise, the function reads ahead until it arrives at the first non-horizontal space character. This is stored in the integer variable `current` that is local to `yyinsert_comma_in_input()`. The function reads ahead to the first non-horizontal space character with the aim of determining the next token that the scanner *would normally have returned* to the parser. If this happens to be the comma token, the semicolon token, the closing box bracket token, or binary operator tokens such as '*', POWER, ':', LTHAN, LTHANE, GTHAN, GTHANE, EQUAL, AND, OR, '\', and '/', or a LINE token, `yyinsert_comma_in_input()` relinquishes the non-horizontal space character to the input and returns control. That is, if any of the above tokens were to be returned by the scanner as the next token, inserting a comma in the input would either be unnecessary (such as before a LINE token) or be syntactically illegal (such as before a '*' token).

If the first non-horizontal space character is a single quote, the next token in the input stream could either be the complex conjugate transpose operator or a string literal. Its meaning is the former if not preceded by horizontal spaces; otherwise, it is regarded to be the latter. The justification for this treatment is that this is how MATLAB interprets the single quote character, a topic that is discussed in detail in § 6.2. The local integer variable `count` records the total number of characters thus far read by `yyinsert_comma_in_input()` from the input. If this value is 1, (at this point in the code, `count` will always be greater than or equal to 1), and if the character stored in `current` is '`''`', the single quote is not preceded by horizontal spaces and therefore, the next token must be the complex conjugate transpose operator. Hence, the function returns the single quote character to the input before returning control.

Additionally, if the non-horizontal space character is '(', the opening parenthesis in the input could either be the start of a new matrix element (in which case, the new matrix element is a parenthesized expression), or be the start of the argument list of a function invocation or an array indexing operation that belongs to the current matrix element. If the value of `count` is 1, the opening parenthesis immediately follows the last token matched by the scanner. If the last token processed by the parser is the IDENTIFIER token and if this token is not followed by horizontal space, or if the current bracket nesting is LD, the function also relinquishes the non-horizontal space character to the input and returns control. This particular behavior is because MATLAB regards `[x␣(2)];` as being `[x,(2)];` and not `[x(2)];`. On the other hand, it treats `[x␣(2)]=3;` as an assignment to `[x(2)]`; hence the check on whether the current bracket nesting is LD. This is explained further in § 7. Moreover, MATLAB considers `[a␣b];` as being equivalent to `[a,b];`. By invoking `yyinsert_comma_in_input()` from the action part of the {IDENTIFIER} extended regular expression in the lexical specification, the above semantics are ensured.

Another case that needs to be taken into account is the possibility that the current character is alphabetic and the last processed token is either an INTEGER or a DOUBLE. This is because character sequences such as `[1a];` would otherwise be parsed as `[1,a];`. When provided with this character sequence, MATLAB complains with the message `'Missing operator, comma, or semicolon.'` after matching the lexeme 1.[k] This "anomalous" behavior was noticed only when a lexeme corresponding to the IDENTIFIER token immediately followed a lexeme corresponding to the INTEGER or DOUBLE tokens in the input. To exactly duplicate this behavior, we need to also check whether `count` equals 1. If so, and if the current character is alphabetic, an IDENTIFIER token immediately follows the last token matched by the scanner. If the last token is either an INTEGER or a DOUBLE, the function simply relinquishes the character stored in `current` to the input before returning.

The only remaining tokens that need explicit consideration are those corresponding to the lexemes `.'`, `.*`, `.^`, `./`, `.\`, `~=`, `+` and `-`. To determine whether a comma is to be inserted in the case of each of these tokens, `yyinsert_comma_in_input()` reads the next character from the input and stores it in the local integer variable `next`.

[j] To follow this discussion, it is strongly recommended that the reader consult the function's source code, available in the lexical specification.

[k] Yet, when provided with character sequences such as `[1.1.1]`, `[a.1]`, `[1[1]]` and `[1i1]`, MATLAB parses them in the "expected" way—as `[1.1,.1]`, `[a,.1]`, `[1,[1]]` and `[1i,1]` respectively.

If the character in current is '.' and the character in next is a digit, the DOUBLE token is the next token that the scanner would normally have returned. Therefore, in this case, the comma insertion point should be *prior* to the period. If not a digit, the character in next can legally be ''', '*', '^', '/', '\' or '.'. In the latter case, no comma insertion is necessary and both next and current are simply relinquished in that order to the input before the function returns.[1] If the character in next is ''', in addition to relinquishing both next and current, the value of count is also checked. If greater than 2, we have a lexical error situation since MATLAB does not allow horizontal spaces to precede a transpose operator (see § 6.2). To ensure that the scanner does detect it, a blank character is inserted in the input before yyinsert_comma_in_input() returns. If this blank were not inserted, the front-end would have parsed [1␣.'2] as [1.',2] instead of complaining of a syntax error as MATLAB does.

If the character in current is the ~ character, the next token in the input stream could either be the UNEQUAL token (i.e., corresponding to the lexeme ~=) or the NOT token (i.e., corresponding to the lexeme ~). If the character stored in next is '=', no comma insertion is necessary and the function once again relinquishes next followed by current to the input before returning.

Finally, if the character in current is either '+' or '-', the next lexeme that the scanner will match can be either the binary addition operator or the unary plus operator. Depending on the value of count and whether the character in next is a horizontal space, a comma is either inserted before the operator or not inserted at all. This is because MATLAB regards [1␣+2]; as being [1,+2]; whereas [1+␣2]; and [1␣+␣2]; are both equivalent to [1+2];.

If the character in current is none of the above, the next token in the input stream could be either IMAGINARY, INTEGER, DOUBLE, IDENTIFIER, '[', ')', or a LEXERROR token. In each of these cases, the function inserts a comma before the token's lexeme and returns control. It does not matter that the function inserts a comma before the ')' and LEXERROR tokens, since an occurrence of these tokens in this context (that is, with BracketDelimiter equal to '[' or LD) would in any case produce a parse error. In fact, an occurrence of the LEXERROR token in *any* context will always produce a parse error.

The above mentioned scheme mimics all of MATLAB's comma insertion semantics and Table 1 summarizes yyinsert_comma_in_input()'s overall operation. In this table, we denote *zero or more* horizontal spaces by ␣ and a new-line, carriage return, form feed or percent character by ◇. The "Next Lexeme Prefix" column refers to the initial part of the character sequence that will be scanned by the lexical analyzer on the next invocation of yylex(). The "Before" and "After" subcolumns indicate the status of the yet-to-be scanned input before and after yyinsert_comma_in_input() executes. Note that the ␣(lexeme prefix gets converted to (or ,(depending on whether the current bracket nesting is LD or not respectively. Also, the (lexeme prefix remains as (or gets converted to ,(depending on whether the last processed token was an IDENTIFIER token or not respectively. The last line in the table denotes yyinsert_comma_in_input()'s action for all other lexemes whose prefixes do not match any of the preceding rows; in this case, yyinsert_comma_in_input() inserts a comma before the first non-horizontal space character.

The yyinsert_comma_in_input() call-sites could have been moved from the semantic actions of the expr → DOUBLE, expr → IMAGINARY, expr → expr TRANSPOSE and expr → expr CTRANSPOSE productions to the action parts of the {DOUBLE}, {NUMBER}{IMAGINARYUNIT}, \.' and <QuoteSC>' extended regular expressions. The effect would have been the same. We could have also moved the call to yyinsert_comma_in_input() from the action part of the {INTEGER} extended regular expression to the semantic action of the expr → INTEGER production. Though the effect would have been the same, from the viewpoint of efficiency, this move is not advisable. This is because, the INTEGER token is also returned by the scanner whenever the {DIGIT}+/\.[*/\\^] and {DIGIT}+/\.' extended regular expressions are matched. However, invoking yyinsert_comma_in_input() in the latter two cases would be unnecessary since these cases offer no scope for comma insertion. Also, moving the call to yyinsert_comma_in_input() from the semantic action of the expr → TEXT production to the action part of the <INITIAL>'[^'\r\f\n]*' extended regular expression is not recommended since the function would also be (unnecessarily) invoked whenever the production command_form

[1] If the character stored in next is '.', the line continuation sequence could be the next character sequence that the lexical analyzer scans. However, the current comma insertion methodology does not capture all of MATLAB's line continuation specifics, but this issue will be addressed in the next version of the parser when all details pertaining to MATLAB's line continuation syntax become known.

TABLE 1: Conversion of “Unscanned” Input by `yyinsert_comma_in_input()`

<table>
<tr><th rowspan="2">Last Processed Token</th><th colspan="2">Next Lexeme Prefix</th></tr>
<tr><th>Before</th><th>After</th></tr>
<tr><td rowspan="29">INTEGER,
DOUBLE,
IMAGINARY,
TEXT,
‘)’,
CTRANSPOSE,
TRANSPOSE,
‘]’,
IDENTIFIER</td><td>␣,</td><td>,</td></tr>
<tr><td>␣;</td><td>;</td></tr>
<tr><td>␣]</td><td>]</td></tr>
<tr><td>␣*</td><td>*</td></tr>
<tr><td>␣^</td><td>^</td></tr>
<tr><td>␣:</td><td>:</td></tr>
<tr><td>␣<</td><td><</td></tr>
<tr><td>␣></td><td>></td></tr>
<tr><td>␣=</td><td>=</td></tr>
<tr><td>␣&</td><td>&</td></tr>
<tr><td>␣|</td><td>|</td></tr>
<tr><td>␣/</td><td>/</td></tr>
<tr><td>␣\</td><td>\</td></tr>
<tr><td>␣◇</td><td>◇</td></tr>
<tr><td>(</td><td>,(or (</td></tr>
<tr><td>␣(</td><td>,(or (</td></tr>
<tr><td>'</td><td>'</td></tr>
<tr><td>.'</td><td>.'</td></tr>
<tr><td>␣.'</td><td>⊔.'</td></tr>
<tr><td>␣.*</td><td>.*</td></tr>
<tr><td>␣.^</td><td>.^</td></tr>
<tr><td>␣./</td><td>./</td></tr>
<tr><td>␣.\</td><td>.\</td></tr>
<tr><td>␣~=</td><td>~=</td></tr>
<tr><td>+</td><td>+</td></tr>
<tr><td>-</td><td>-</td></tr>
<tr><td>␣+⊔</td><td>+⊔</td></tr>
<tr><td>␣-⊔</td><td>-⊔</td></tr>
<tr><td>␣⋆</td><td>,⋆</td></tr>
</table>

→ `identifier text_list` is being processed. The call to `yyinsert_comma_in_input()` cannot be moved from the semantic action of the `expr` → `'(' parenthesis expr ')'` production to the action part of the `\)` extended regular expression because the mid-rule action in the above production is performed *after* the scanner returns the closing parenthesis token. In other words, the current bracket nesting is correctly reset only after the token ‘`)`’ is returned. Therefore, `yyinsert_comma_in_input()` should not be invoked from the action part of the `\)` extended regular expression because the value of `BracketDelimiter` will not correctly reflect the current bracket nesting until after the mid-rule action in the `expr` → `'(' parenthesis expr ')'` production has executed. Thus, in the case of the ‘`)`’ token, the calls to `yyinsert_comma_in_input()` should be from the semantic actions of the relevant productions (the only other relevant production in this case is `reference` → `identifier '(' parenthesis argument_list ')'`). Likewise, the call to `yyinsert_comma_in_input()` cannot be moved from the semantic action of the `expr` → `'[' boxes1 rows ']'` production to the action part of the `\]` extended regular expression because the mid-rule action in this production is performed after the scanner returns the closing box bracket token.

Finally, it must be mentioned that `yyinsert_comma_in_input()` carries no checks for an *end-of-file* situation. This is because whenever `yyinput()` (see `flex(1)`) is called in this function, the current bracket nesting will be either '`[`' or `LD`. The extended regular expression `\[` is responsible for both of these tokens and its action part guarantees the existence of a matching '`]`' or `RD` token in the input stream. Hence, `yyinsert_comma_in_input()` can safely read until the first non-horizontal space character without having to check for the end-of-file condition. If this character happens to be '`]`', the function does not read further from the input; otherwise, it can read at least one more character from the input without hazarding an end-of-file situation.

6.2 The Single Quote Character

As mentioned earlier in § 5, the single quote character can be associated with two tokens: the `TEXT` token or the `CTRANSPOSE` token. For example, in the MATLAB statement

```
disp␣'Hello␣World!';
```

it demarcates a string literal, while in

```
disp=1;
disp'';
```

it denotes the complex conjugate transpose operator.

The actual role played by the single quote character is determined by horizontal spaces that precede it. In a non-matrix scenario, horizontal spaces within expressions are usually inconsequential. For instance, the MATLAB expression statements `1+2;` and `1␣+2;` and `1␣+␣2;` are all the same, and it does not matter whether horizontal spaces surround the binary plus operator. In most cases, horizontal spaces occurring among the tokens of a valid expression have no effect whatsoever on the token stream scanned by the lexical analyzer. However, there are situations wherein horizontal spaces within expressions *do* become significant. Apart from the obvious case of a matrix in which horizontal spaces serve as element separators, the case of the single quote character is another interesting instance where horizontal spaces actually determine how a character sequence must be interpreted.

Consider the following character sequence: `A␣'+1';`. And suppose that the front-end has been able to ascertain that the identifier `A` in the above character sequence actually corresponds to the function M-file `A.m`. In addition, let the function definition line in `A.m` specify a single input argument and a single output argument. Therefore, the function M-file can be invoked either with one argument or no arguments. Consequently, how should the parser-scanner pair process the above character sequence? Should the lexical analyzer return an `IDENTIFIER` token followed by a `TEXT` token finally followed by a semicolon token so that the parser recognizes a command-form invocation of a function, or should the lexical analyzer return the token stream `IDENTIFIER`, `CTRANSPOSE`, '`+`', `INTEGER`, `CTRANSPOSE`, '`;`' so that the parser recognizes an expression? Should the presence of the horizontal spaces between the characters '`A`' and '`'`' be ignored so that the character sequence `A'+1'` is also treated in the same way?

MATLAB interprets the single quote character by always applying a simple rule: if a single quote character immediately follows an `INTEGER`, `DOUBLE`, `IMAGINARY`, `IDENTIFIER`, `TRANSPOSE`, `CTRANSPOSE`, '`]`' or '`)`' token, it is regarded to be the `CTRANSPOSE` token. Otherwise, it is considered to be the starting demarcator of a string literal. Notice that this rule resolves the above mentioned ambiguity. That is, the character sequence `A␣'+1';` is scanned into the token stream `IDENTIFIER`, `TEXT`, '`;`' by this rule, whereas the same rule causes the character sequence `A'+1';` to be scanned into the token stream `IDENTIFIER`, `CTRANSPOSE`, '`+`', `INTEGER`, `CTRANSPOSE`, '`;`'. Notice also that by this rule, the character sequence `1␣';` produces a syntax error.[m]

The MATCH scanner reproduces MATLAB's single quote semantics by using start conditions. The start condition mechanism basically enables the scanner to "activate" only a subset of the rules in its lexical

[m]This is quite contrary to what one would intuitively expect!

specification depending on its current state. In the absence of explicitly declared start conditions,[n] the scanner always exists in a single state that is associated with the start condition `INITIAL`. A scanner can be transited from one start condition to another by `BEGIN` commands (see `flex(1)`). Extended regular expressions that are prefixed by the construction `<SC>` where `SC` is a declared start condition, are active only when the scanner is in the start condition `SC`. Extended regular expressions that are not prefixed by `<SC>` are either active or inactive in the start condition `SC` depending on whether the start condition is inclusive or exclusive. The predefined start condition `INITIAL` is inclusive.

To imitate MATLAB's single quote semantics, the scanner is always in one of two start conditions. These are referred to as `INITIAL` and `QuoteSC` in the lexical specification. When in `INITIAL`, the scanner regards the single quote character as the demarcator of a string literal. When in `QuoteSC`, the scanner considers the single quote character as the `CTRANSPOSE` token. We thus have the extended regular expressions `<INITIAL>'[^'\r\f\n]*'` and `<QuoteSC>'` whose action parts return the tokens `TEXT` and `CTRANSPOSE` respectively. Since a single quote character immediately after an `INTEGER`, `DOUBLE`, `IMAGINARY`, `IDENTIFIER`, `TRANSPOSE`, '`]`', '`)`' or `CTRANSPOSE` token should be regarded as the `CTRANSPOSE` token, the action parts of the extended regular expressions responsible for each of the above tokens except the last contain a `BEGIN` command that changes the start condition to `QuoteSC`. Once the scanner enters the `QuoteSC` start condition, the only way it can exit this start condition and thus enter the start condition `INITIAL` is if it scans lexemes such as `,`, `(` or `*`.[o] More importantly, a horizontal space in the input causes the scanner to enter the start condition `INITIAL`. Thus, while the character sequence `A''` is scanned by the lexical analyzer into the token stream `IDENTIFIER`, `CTRANSPOSE`, `CTRANSPOSE` (and not into the token stream `IDENTIFIER`, `TEXT`), the character sequence `A␣''` is scanned into the token stream `IDENTIFIER`, `TEXT` (and not `IDENTIFIER`, `CTRANSPOSE`, `CTRANSPOSE`).

Finally, it must be mentioned that MATLAB considers the character sequence `.'` to be the transpose operator only if not preceded by horizontal spaces. If preceded by horizontal spaces, it is also treated like the starting demarcator of a string literal. For example, when provided with the character sequence `A␣.'+1'`, MATLAB parses it into the token stream `IDENTIFIER`, `TEXT`. In this case, the lexeme associated with the `TEXT` token is `.+1`.

6.3 Colon Expressions

Colon expressions are a useful way to succinctly describe row vectors in which the elements form an arithmetic progression. For example, the statement

```
a=1:4;
```

assigns the same value to `a` as does the assignment

```
a=[1,2,3,4];
```

A *stride* could also be provided, so that `a=1:2:4;` is equivalent to `a=[1,3];`. More precisely, colon expressions come in two flavors:

$$\alpha:\beta = \begin{cases} \big(\alpha, \alpha+1, \alpha+2, \ldots, \beta\big) & \text{if } \alpha \le \beta, \\ \big(\ \big) & \text{otherwise.} \end{cases}$$

$$\alpha:\sigma:\beta = \begin{cases} \big(\alpha, \alpha+\sigma, \alpha+2\sigma, \ldots, \alpha+\lfloor\frac{\beta-\alpha}{\sigma}\rfloor\sigma\big) & \text{if } \alpha \le \beta \wedge \sigma > 0 \text{ or } \alpha > \beta \wedge \sigma < 0, \\ \big(\ \big) & \text{otherwise.} \end{cases}$$

As an example, `2:4` results in a row vector with three elements: 2, 3 and 4. On the other hand, `2:0:4` produces the empty matrix as the result.

[n] An *inclusive* start condition is declared using the `%s` directive in the definitions section of the lexical specification.
[o] For a full list, please refer to the lexical specification.

```
colon_expr              : expr ':' expr
                        {
                         $$.start = $1;

                         if (LOOKAHEAD != ':')
                            {
                             $$.stride =
                             build_expr_ATOM(build_atom_INTEGER(1));
                             $$.stop = $3;
                             }
                         else
                            {
                             $$.stride = $3;
                             $$.stop = 0;
                             }
                         }
                        | colon_expr ':' expr
                        {
                         if ($1.stop)
                            {
                             $$.start =
                             build_expr_ternary_op(COLONexpr,
                             $1.start, $1.stride, $1.stop);

                             if (LOOKAHEAD != ':')
                                {
                                 $$.stride =
                                 build_expr_ATOM(
                                 build_atom_INTEGER(1));
                                 $$.stop = $3;
                                 }
                             else
                                {
                                 $$.stride = $3;
                                 $$.stop = 0;
                                 }
                             }
                         else
                            {
                             $$.start = $1.start;
                             $$.stride = $1.stride;
                             $$.stop = $3;
                             }
                         }
                        ;
```

The colon operator (‘`:`’) is left associative. Thus, constructions such as `1:2:2:2` and `1:2:2:2:2` are evaluated in the same way as `(1:2:2):2` and `(1:2:2):2:2` respectively. When evaluated, the former yields a row vector with two elements, whereas the latter produces a scalar.[p]

[p]In a colon expression, the start, stride (if present) and stop values must all be scalars. If any of these are not scalars, MATLAB issues a warning and considers their respective *first* elements to evaluate the colon expression. Thus, when provided

To simplify the processing of colon expressions by subsequent compiler passes, the parser always produces a full ternary tree as the AST of a colon expression.[q] In other words, the parser retains colon expressions of the form $\alpha : \sigma : \beta$ and converts colon expressions of the form $\alpha : \beta$ to $\alpha : 1 : \beta$.

Token streams corresponding to colon expressions eventually reduce to the `colon_expr` nonterminal and ultimately to the `expr` nonterminal. The LALR(1) parser generated by bison processes a colon expression from left to right, by first applying the production `colon_expr` → `expr ':' expr`, and by subsequently applying the production `colon_expr` → `colon_expr ':' expr` in succession. Finally, the production `expr` → `colon_expr` is applied. For example, given `1:2:2:2`, the following shows the sequence of reductions performed by the parser.

```
expr → 1
expr → 2
colon_expr → expr ':' expr
expr → 2
colon_expr → colon_expr ':' expr
expr → 2
colon_expr → colon_expr ':' expr
expr → colon_expr
```

Thus, in the production `colon_expr` → `expr ':' expr`, the first `expr` nonterminal must be a start value. Hence the assignment

```
$$.start = $1;
```

in the semantic action of this production. Now, the parser applies this production only after consulting the lookahead token. To understand the reason for this, suppose that the parser has thus far recognized the *sentential form* `expr ':' expr`. If the next token in the input stream were '`+`', the parser should perform a shift action, since the binary addition operator has a higher precedence than the colon operator. Stated alternately, the second `expr` nonterminal in the sentential form `expr ':' expr` is a subexpression of the colon expression's second operand. If the next token in the input stream were instead '`<`', the parser should perform a reduce action, since the binary less-than operator has a lower precedence than the colon operator. Therefore, when the reduction `colon_expr` → `expr ':' expr` is applied, the lookahead token can legally be either a '`:`' or some other token corresponding to an operator having a lower precedence than the colon operator. If the lookahead token[r] is not '`:`', the second `expr` nonterminal must be the colon expression's stop value. Hence the assignment

```
$$.stop = $3;
```

in the conditional part of this production's semantic action. However, if the lookahead token were indeed '`:`', the second `expr` nonterminal must be the colon expression's stride value, and a stop value is expected in the input.

As far as the production `colon_expr` → `colon_expr ':' expr` is concerned, two possibilities may arise in its semantic action. The right-hand side nonterminal `colon_expr` could either be a *partial* colon expression—one whose stop value is yet to be ascertained—or it could be a *complete* colon expression having all three components. In the former case, the nonterminal `expr` corresponds to the partial colon expression's expected stop value. This results in a complete colon expression against the left-hand side nonterminal `colon_expr`. In the latter case, the complete colon expression associated with `colon_expr` is itself the start value of a new colon expression. In this new colon expression, `expr` corresponds to either the stride or stop value depending on whether '`:`' is the lookahead token.

---

with the input `1:2:3:4:5`, MATLAB issues an alert ('`Warning: COLON arguments should be real scalars.`') and produces a row vector having 1 and 5 as its elements.

[q]A colon expression is different from a colon atom. The latter is employed in array indexing operations to denote the entire extent of a particular array dimension.

[r]The macro `LOOKAHEAD` substitutes to `yychar`; `yychar` is a variable that is always set to the parser's lookahead token if any, and `YYEMPTY` otherwise (see `bison(1)`).

The productions `colon_expr` → `expr ':' expr` and `colon_expr` → `colon_expr ':' expr` give rise to a shift-reduce conflict. This is because if `colon_expr` is the sentential form thus far seen by the parser and if ‘`:`’ is the lookahead token, the parser could either choose to shift the lookahead token so as to subsequently apply the `colon_expr` → `colon_expr ':' expr` production, or choose to immediately apply the `expr` → `colon_expr` production and later the `colon_expr` → `expr ':' expr` production. However, to identify the stride and stop values of colon expressions having more than two operands, the parser should perform the shift action rather than the reduce action. This is in fact the default course of action in the event of a conflict. By doing so, the syntax-directed translation process is exploited to efficiently determine whether the expression following a colon operator is a stride or stop value.

We could have replaced the above pair of grammar rules by the single production `expr` → `expr ':' expr` and the parser would have supported the same colon expression syntax. In fact, this replacement would have eliminated the previously mentioned shift-reduce conflict. However, casting the recognized colon expression to the $\alpha : \sigma : \beta$ form becomes a complicated affair involving the maintenance of some kind of book-keeping information, and/or the allocation and deallocation of temporary expressions.

7 Assignments

Assignments are important in MATLAB since they are the only way by which a variable can be defined. None of the arithmetic, relational and logical operators in MATLAB produce side effects. Thus, assignments are also the only means by which a variable's value can be altered. That is, MATLAB does not offer side-effect producing operators such as the increment and decrement operators available in some languages such as C. Hence, if a function M-file has side effects, its execution must eventually involve an assignment.

Assignments in MATLAB come in three variations. The first variation allows the assignment of an arbitrary expression to a variable or an array section. As an example,

```
clear a;
a(1:2:3,1:3) = 2;
```

creates a 3 by 3 matrix against the variable `a` having the element 2 along the first and third rows and the element 0 in the remaining rows and columns. The second variation is essentially the same as the first, except that the variable or array section syntax is enclosed within a pair of box brackets. For instance, the previous example could have also been written as

```
clear a;
[a(1:2:3,1:3)] = 2;
```

and the same value would have been assigned to `a`. The third form allows a function to return more than one value. In this form, the right-hand side of the assignment is restricted to be a function invocation. This is illustrated in the following example using the built-in function `size`.

```
[x y] = size(1);
```

In terms of a grammar, the above three varieties of assignment are expressed using three productions. The nonterminals `s_assignee_matrix` and `m_assignee_matrix` correspond to sentences that largely resemble the sentences associated with the nonterminal `matrix`, but that are not exactly the same. Both sentences consist of a sequence of one or more elements that are separated from each other by horizontal spaces or commas and that are enclosed within a pair of box brackets. But that is where the resemblance stops.

```
assignment            : reference '=' expr
                      | s_assignee_matrix '=' expr
                      | m_assignee_matrix '=' reference
                      ;
```

For one, semicolon and LINE token delimiters are not allowed among the elements of an m_assignee_matrix. Second, a comma is not allowed after the last element in an s_assignee_matrix or m_assignee_matrix sentence. Third, the elements in an s_assignee_matrix or m_assignee_matrix sentence can only be variables or array sections. Fourth, any expression may be assigned to an s_assignee_matrix whereas the same is not true for an m_assignee_matrix. Fifth, assignments to empty matrices are invalid. In fact, the first two reasons made it imperative that new nonterminals be introduced (in the form of s_assignee_matrix and m_assignee_matrix), rather than rely on the matrix nonterminal to describe the left-hand sides of such assignments.

```
s_assignee_matrix       : LD boxes2 reference RD
                        ;

m_assignee_matrix       : LD boxes2 reference ',' reference_list RD
                        ;

boxes2                  :
                        ;

reference_list          : reference
                        | reference ',' reference_list
                        ;
```

The tokens LD and RD actually correspond to the lexemes [and] respectively. Notice that the parser expects the lexical analyzer to return the tokens '[' and ']' whenever these lexemes are encountered in the case of matrices. If the grammar rules for s_assignee_matrix and m_assignee_matrix had also used the same tokens to enclose the elements, a reduce-reduce conflict would arise due to the productions reference_list → reference and expr → reference. This is because until an assignment operator is encountered, the parser has no way of determining whether an assignment or an expression is currently being processed. That is, until the '=' token is seen, the parser could be in the midst of a sentence that ultimately reduces to the s_assignee_matrix or m_assignee_matrix nonterminals, or in the midst of a sentence that ultimately reduces to the matrix nonterminal. The default course of action taken by the parser in such situations cannot be relied on, since a reduction to the s_assignee_matrix or m_assignee_matrix nonterminals must prevail in an assignment context, while a reduction to the matrix nonterminal must prevail in an expression context.

To remedy the above problem, the scanner returns a separate pair of tokens whenever an assignment to a box-bracketed structure is detected. Thus, in the action part of the extended regular expression \[, the input is read ahead until a matching closing box bracket is encountered. If a matching closing box bracket is not found, the token LEXERROR is returned, since this is an error situation. If a matching closing box bracket is found, the scanner reads ahead to the next non-horizontal space character or until the end of the input. If a non-horizontal space character is found, and if it happens to be '=', a potential assignment to a box-bracketed structure arises. This is because while [a,b]=size(1); corresponds to an assignment to a box-bracketed structure, [a,b]==size(1); is an expression. Hence, on encountering the '=' character, the scanner attempts to read the next character in the input stream. If one exists and if it is not '=', an assignment to a box-bracketed structure is detected. The scanner therefore returns the LD token. Otherwise, the scanner returns the '[' token. In both cases, the scanner returns control only after relinquishing the characters that were read ahead back to the input.

Similarly, the RD token is returned only when the extended regular expression \]/{HSPACE}*=[^=] is matched. Notice that once an LD token is returned by the scanner, it is guaranteed that this extended regular expression will be subsequently matched. All other closing box brackets will match the extended regular expression \], whose action part returns the ']' token to the parser.

The scanner maintains the current nesting level of a box bracket. This information is available in the static integer variable Depth. This information enables an optimization that avoids reading ahead on every opening box bracket. Whenever an opening box bracket is encountered, Depth is incremented by one. Whenever a closing box bracket is encountered, `Depth` is decremented by one. Also, during a read-ahead, the scanner matches every opening box bracket with a closing box bracket. Each of the inner box brackets is assigned a nonnegative integer called the "depth" that corresponds to how deep the box bracket is in the box bracket nesting. Hence, on encountering a box bracket, a read-ahead needs to be done only if the current value of `Depth` is zero.

It must be noted that the grammar presented here does not permit assignments to be a part of expressions. In other words, statements such as `(a=1)+1;` are not allowed. This restriction is in accordance with that observed in MATLAB.

8 Control Statements

The control constructs that are currently supported in the MATCH compiler enable the conditional (`if`) or iterative (`for`, `while`) execution of a body of statements. From a syntactic perspective, these statements do not pose a problem except that the grammar rules responsible for each of these constructs introduce shift-reduce conflicts.

```
if_command              : IF if_cmd_list END
                        ;

if_cmd_list             : expr delimited_input opt_else
                        ;

delimited_input         : opt_delimiter
                        | opt_delimiter delimited_list
                        ;

delimited_list          : statement delimiter
                        | statement delimiter delimited_list
                        ;
```

Consider the conditional statement. It is consists of an expression associated with the `if` part of the statement and a body of statements that is executed only if this expression evaluates to true at run time. The body may be empty and is represented by the nonterminal `delimited_input` shown above.

The conditional statement may also have multiple `elseif` parts and an `else` part, but these are optional. If present, each of the `elseif` parts possesses its own expressions.

```
opt_else                :
                        | ELSE delimited_input
                        | ELSEIF expr delimited_input opt_else
                        ;
```

Notice that the expressions associated with the `if` and `elseif` parts of the conditional statement can be separated from their respective statement bodies by only horizontal spaces. For instance,

```
if a (2); end;
```

is a valid conditional statement in which the `elseif` and `else` parts are absent. The lexeme `a` forms the conditional statement's expression. The conditional statement's body is a single parenthesized expression. Since the expressions associated with the `if` and `elseif` parts can be separated from their respective bodies by only horizontal spaces, this gives rise to shift-reduce conflicts in the grammar. For example, if we were to consider the following code fragment,

```
if 1 +2; end;
```

should this be treated as a conditional statement in which the expression is `1` and the body is the single expression statement `+2;`, or should this be regarded as a conditional statement in which the expression is `1+2` and the body is empty? The production `if_cmd_list` → `expr delimited_input opt_else` generates two shift-reduce conflicts. These two conflicts occur when the right-hand side of this production has been seen until the nonterminal `expr` and if the next token is either a '`+`' or a '`-`'. In such a situation, the parser could either choose to shift the token (the default action) so that the expression recognized thus far becomes a subexpression of a binary addition operation, or choose to apply the reduction `opt_delimiter` → ϵ so that the '`+`' or '`-`' tokens are unary operators in an expression that finally reduces to the nonterminal `delimited_input`. As it turns out, the default course of action taken by the parser to resolve this conflict suffices since this duplicates MATLAB's behavior.

The production `opt_else` → `ELSEIF expr delimited_input opt_else` similarly introduces a pair of shift-reduce conflicts. Thus, the grammar rules behind the conditional statement give rise to four shift-reduce conflicts. Productions for the `for` and `while` statements similarly give rise to two shift-reduce conflicts each. The default course of action that the parser takes in each of these cases—a shift action—is the desired way in which these conflicts should be resolved.

As an aside, it must be mentioned that the grammar described in the appendix has ten shift-reduce conflicts. In addition to the eight mentioned above, the other two are due to the productions responsible for the colon expression (see § 6.3) and the `reference` productions. In all these cases, the default action of a shift is the correct way in which the conflicts should be resolved.

9 Summary

In this report, we have presented the grammar and lexical specification behind the MATCH compiler front-end. The language recognized is a proper subset of MATLAB. The principal parts of the grammar and lexical specification were mentioned and discussed in some depth. In particular, we justified why the parser supports a limited form of MATLAB's command-form function invocation syntax. We also show how commas are inserted among matrix elements so that the only delimiters visible to the parser are the comma, semicolon and `LINE` tokens. The dual role played by the single quote character and the syntactic issues that it gives rise to were also explained. The scheme by which all colon expressions are converted to a uniform form was also described. Finally, assignment statements and control constructs and their respective grammar rules were discussed.

References

[ASU88] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. **Compilers: Principles, Techniques, and Tools**. Computer Science Series. Addison-Wesley Publishing Company, Inc., 1988. ISBN 0–201–10088–6.

[FXB] At `http://www.uman.com/lexyacc.shtml`, Microman Links and Download Examples for Lex and Yacc.

[Joh75] S. C. Johnson. "Yacc: Yet Another Compiler Compiler". Technical Report 32, Bell Laboratories, July 1975.

[Les75] M. E. Lesk. "Lex: A Lexical Analyzer Generator". Technical Report 39, Bell Laboratories, October 1975.

[MAT] At `http://www.ece.nwu.edu/cpdc/Match/Match.html`, The MATCH Project Home Page.

[Mat97] The MathWorks, Inc. *MATLAB: The Language of Technical Computing*, January 1997. Using MATLAB (Version 5).

[OCT] At `http://www.che.wisc.edu/octave/`, The Octave Home Page.

A Lexical Specification for the MATCH Scanner

```
%{

/*
 * Lexical specification for the MATCH scanner written for
 * flex (GNU Flex version 2.5.4).
 *
 * Command-line options: -f
 */

#include <stdio.h>
#include <string.h>
#include <ctype.h>
#include <stdlib.h>


#include "basic.h"


#include "y.tab.h"


#define Return(a, b)    \
{ \
 DB1(DB_HPM_lex, "Source line: %d, returned token: "a".\n", \
 SourceLine); \
 return b; \
 }


void yyinsert_comma_in_input(const int);
unsigned int SourceLine = 1;

static int Depth;

%}


%option noyywrap
%s QuoteSC


HSPACE                          [ \t]
HSPACES                         {HSPACE}+
NEWLINE                         \n|\r|\f
NEWLINES                        {NEWLINE}+


ELLIPSIS                        \.\.\.
```

```
CONTINUATION                    {ELLIPSIS}[^\n\r\f]*{NEWLINE}?


COMMENT                         \%[^\n\r\f]*{NEWLINE}?


IDENTIFIER                      [a-zA-Z][_a-zA-Z0-9]*


DIGIT                           [0-9]
INTEGER                         {DIGIT}+
EXPONENT                        [DdEe][+-]?{DIGIT}+
MANTISSA                        ({DIGIT}+\.)|({DIGIT}*\.{DIGIT}+)
FLOATINGPOINT                   {MANTISSA}{EXPONENT}?
DOUBLE                          ({INTEGER}{EXPONENT})|{FLOATINGPOINT}
NUMBER                          {INTEGER}|{DOUBLE}
IMAGINARYUNIT                   [ij]


%%


"for"                           {
                                 Return("FOR", FOR);
                                 }


"end"                           {
                                 Return("END", END);
                                 }


"if"                            {
                                 Return("IF", IF);
                                 }


"elseif"                        {
                                 Return("ELSEIF", ELSEIF);
                                 }


"else"                          {
                                 Return("ELSE", ELSE);
                                 }


"global"                        {
                                 Return("GLOBAL", GLOBAL);
                                 }
```

```
"while"                        {
                                Return("WHILE", WHILE);
                                }


"function"                     {
                                Return("FUNCTION", FUNCTION);
                                }


"return"                       {
                                Return("RETURN", RETURN);
                                }


<INITIAL>'[^'\n\r\f]*'/'       {
                                yymore();
                                }


,                              {
                                BEGIN(INITIAL);

                                Return("','", ',');
                                }


;                              {
                                BEGIN(INITIAL);

                                Return("';'", ';');
                                }


<INITIAL>'[^'\r\f\n]*'         {
                                register int i, size;
                                char* modified;

                                const int length = yyleng-2;

                                for (size = 0, i = 1;
                                i <= length; size++, i++)
                                if (*(yytext+i) == '\'')
                                   i++;

                                modified = alloc_string(size+1);
                                *(modified+size) = '\0';

                                for (size = 0, i = 1;
                                i <= length; size++, i++)
```

```
                                 {
                                  *(modified+size) = *(yytext+i);

                                  if (*(yytext+i) == '\'')
                                     i++;
                                  }

                                 yylval.text = modified;

                                 Return("TEXT", TEXT);
                                 }


{NUMBER}{IMAGINARYUNIT}         {
                                 BEGIN(QuoteSC);

                                 *(yytext+yyleng-1) = '\0';

                                 yylval.imaginaryQ =
                                 atof(yytext);

                                 Return("IMAGINARY", IMAGINARY);
                                 }


{DIGIT}+/\.[*/\\^]              {
                                 yylval.integerQ =
                                 atoi(yytext);

                                 Return("INTEGER", INTEGER);
                                 }


{DIGIT}+/\.'                    {
                                 yylval.integerQ =
                                 atoi(yytext);

                                 Return("INTEGER", INTEGER);
                                 }


{INTEGER}                       {
                                 BEGIN(QuoteSC);

                                 yylval.integerQ =
                                 atoi(yytext);

                                 yyinsert_comma_in_input(INTEGER);

                                 Return("INTEGER", INTEGER);
                                 }
```

```
{DOUBLE}                      {
                               BEGIN(QuoteSC);

                               yylval.doubleQ =
                               atof(yytext);

                               Return("DOUBLE", DOUBLE);
                               }


{HSPACES}                     {
                               BEGIN(INITIAL);
                               }


{CONTINUATION}                {
                               SourceLine++;
                               }


{NEWLINES}                    {
                               BEGIN(INITIAL);

                               SourceLine += yyleng;

                               Return("LINE", LINE);
                               }


{IDENTIFIER}                  {
                               BEGIN(QuoteSC);

                               yylval.symbol =
                               strcpy(alloc_string(yyleng+1),
                               yytext);

                               yyinsert_comma_in_input(IDENTIFIER);

                               Return("IDENTIFIER", IDENTIFIER);
                               }


{COMMENT}                     {
                               BEGIN(INITIAL);

                               SourceLine++;

                               Return("LINE", LINE);
                               }
```

```
&                               {
                                 BEGIN(INITIAL);

                                 Return("AND", AND);
                                 }


\|                              {
                                 BEGIN(INITIAL);

                                 Return("OR", OR);
                                 }


\<                              {
                                 BEGIN(INITIAL);

                                 Return("LTHAN", LTHAN);
                                 }


\<=                             {
                                 BEGIN(INITIAL);

                                 Return("LTHANE", LTHANE);
                                 }


>                               {
                                 BEGIN(INITIAL);

                                 Return("GTHAN", GTHAN);
                                 }


>=                              {
                                 BEGIN(INITIAL);

                                 Return("GTHANE", GTHANE);
                                 }


==                              {
                                 BEGIN(INITIAL);

                                 Return("EQUAL", EQUAL);
                                 }
```

```
~=                           {
                              BEGIN(INITIAL);

                              Return("UNEQUAL", UNEQUAL);
                              }


:                            {
                              BEGIN(INITIAL);

                              Return("':'", ':');
                              }


\+                           {
                              BEGIN(INITIAL);

                              Return("'+'", '+');
                              }


-                            {
                              BEGIN(INITIAL);

                              Return("'-'", '-');
                              }


\*                           {
                              BEGIN(INITIAL);

                              Return("'*'", '*');
                              }


".*"                         {
                              BEGIN(INITIAL);

                              Return("EMUL", EMUL);
                              }


\/                           {
                              BEGIN(INITIAL);

                              Return("'/'", '/');
                              }


"./"                         {
                              BEGIN(INITIAL);
```

```
                                  Return("EDIV", EDIV);
                                  }


\\                               {
                                  BEGIN(INITIAL);

                                  Return("'\\'", '\\');
                                  }


\.\\                             {
                                  BEGIN(INITIAL);

                                  Return("ELEFTDIV", ELEFTDIV);
                                  }


{HSPACES}\.'                     {
                                  Return("LEXERROR", LEXERROR);
                                  }


\.'                              {
                                  BEGIN(QuoteSC);

                                  Return("TRANSPOSE", TRANSPOSE);
                                  }


\.^                              {
                                  BEGIN(INITIAL);

                                  Return("EPOWER", EPOWER);
                                  }


\^                               {
                                  BEGIN(INITIAL);

                                  Return("POWER", POWER);
                                  }


~                                {
                                  BEGIN(INITIAL);

                                  Return("NOT", NOT);
                                  }
```

```
<QuoteSC>'                    {
                               Return("CTRANSPOSE", CTRANSPOSE);
                               }


<INITIAL>'                    {
                               Return("LEXERROR", LEXERROR);
                               }


\[                            {
                               if (Depth)
                                  {
                                   Depth++;

                                   Return("'['", '[');
                                   }

                               int current = 0, next = 0;
                               char* buffer = 0;
                               int level = 1, length = 0;

                               while (level &&
                               (current = yyinput()) != EOF)
                               {
                                buffer =
                                realloc_string(buffer, ++length);

                                *(buffer+length-1) = current;

                                if (current == '[')
                                   level++;

                                if (current == ']')
                                   level--;
                                }

                               if (level)
                                  {
                                   Return("LEXERROR", LEXERROR);
                                   }

                               while ((current = yyinput()) != EOF)
                               {
                                buffer =
                                realloc_string(buffer, ++length);

                                *(buffer+length-1) = current;

                                if (current != ' ' &&
```

```
                                     current != '\t')
                                        break;
                                     }

                                    if ((next = yyinput()) != EOF)
                                       {
                                        buffer =
                                        realloc_string(buffer, ++length);

                                        *(buffer+length-1) = next;
                                        }


                                    for (; length > 0; length--)
                                    unput(*(buffer+length-1));

                                    free(buffer);

                                    Depth = 1;

                                    if (current == '=' && next != '=')
                                       {
                                        Return("LD", LD);
                                        }
                                    else
                                       {
                                        Return("'['", '[');
                                        }
                                    }


\]/{HSPACE}*=[^=]                  {
                                    BEGIN(INITIAL);

                                    Depth--;

                                    Return("RD", RD);
                                    }


\]                                 {
                                    BEGIN(QuoteSC);

                                    Depth--;

                                    Return("']'", ']');
                                    }


\(                                 {
                                    BEGIN(INITIAL);
```

```
                                  Return("'('", '(');
                                  }


\)                               {
                                  BEGIN(QuoteSC);

                                  Return("')'", ')');
                                  }


=                                {
                                  BEGIN(INITIAL);

                                  Return("'='", '=');
                                  }


.                                {
                                  Return("LEXERROR", LEXERROR);
                                  }


%%


void yyinsert_comma_in_input(const int lastToken)
{
 int count, current, next;

 extern int BracketDelimiter;


 DB0(DB_HPM_lex, "Entering <yyinsert_comma_in_input> ...\n");

 if (!(BracketDelimiter == '[' || BracketDelimiter == LD))
    return;

 for (count = 1; ; count++)
 {
  current = yyinput();

  if (current != ' ' && current != '\t')
     break;
  }

 if (current == ',' || current == ';' || current == ']' ||
 current == '*' || current == '^' || current == ':' ||
 current == '<' || current == '>' || current == '=' ||
 current == '&' || current == '|' ||
```

```
current == '/' || current == '\\' ||
current == '\n' || current == '\r' || current == '\f' ||
current == '%' ||
(current == '\'' && count == 1))
   {
    unput(current);

    return;
    }

if (current == '(' &&
(BracketDelimiter == LD ||
(lastToken == IDENTIFIER && count == 1)))
   {
    unput(current);

    return;
    }

if (isalpha(current) &&
(lastToken == INTEGER || lastToken == DOUBLE) && count == 1)
   {
    unput(current);

    return;
    }


next = yyinput();
++count;


if (current == '.' &&
(next == '\'' || next == '*' || next == '^' ||
next == '/' || next == '\\' || next == '.'))
   {
    unput(next);
    unput('.');

    if (next == '\'' && count > 2)
       unput(' ');

    return;
    }

if (current == '~' && next == '=')
   {
    unput('=');
    unput('~');

    return;
```

```
    }

if (current == '+' || current == '-')
   {
    if (count == 2 || next == ' ' || next == '\t')
       {
        unput(next);
        unput(current);

        return;
        }
    }

unput(next);
unput(current);
unput(',');
}
```

B Context-Free Grammar for the MATCH Parser

```
%{

/*
 * Context-free grammar for the MATCH parser written for
 * bison (GNU Bison version 1.24).
 *
 * Command-line options: --yacc -d -t
 */

#include <stdlib.h>
#include <string.h>

#include "basic.h"
#include "cpp_classes.h"


#define LOOKAHEAD       yychar


statement_t* InputSequence;
int BracketDelimiter = '\0';


extern int yylex(void);
extern void yyinsert_comma_in_input(const int);


static void yyerror(const char*);

static const char* FunctionName;


/*
 * MATLAB's expression operators fall into three categories:
 *
 *      + Arithmetic operators.
 *      + Relational operators.
 *      + Logical operators.
 *
 * The precedences documented in the manual appear to be wrong. For
 * instance, according to the manual, arithmetic operators have the
 * highest precedence, followed by relational operators, followed by
 * logical operators. This would mean that the MATLAB interpretor
 * should consider the token sequence NOT INTEGER '+' INTEGER as being
 * equivalent to NOT '(' INTEGER '+' INTEGER ')', and not as '(' NOT
 * INTEGER ')' '+' INTEGER. However, the MATLAB interpretor actually
 * treats the token sequence in the latter manner.
 *
```

```
 * The following were the observed precedence levels of operators in
 * the MATLAB 5.0 interpretor, arranged in decreasing order:
 *
 *      1. Power (EPOWER), matrix power (POWER).
 *
 *      2. Transpose (TRANSPOSE),
 *      complex conjugate transpose (CTRANSPOSE).
 *
 *      3. Logical negation (NOT),
 *      unary plus ('+'), unary minus ('-').
 *
 *      4. Matrix multiplication ('*'), multiplication (EMUL),
 *      matrix division ('/'), division (EDIV),
 *      matrix left division ('\\'), left division (ELEFTDIV).
 *
 *      5. Addition ('+'), subtraction ('-').
 *
 *      6. Colon operator (':').
 *
 *      7. Less than (LTHAN), less than or equal to (LTHANE),
 *      greater than (GTHAN), greater than or equal to (GTHANE),
 *      equal to (EQUAL), not equal to (UNEQUAL).
 *
 *      8. Logical and (AND), logical or (OR).
 *
 * Within each precedence level, operators have left associativity,
 * except for the level 2 and level 3 operators, which are
 * non-associative.
 */

%}


%start input


%token LEXERROR

%token LINE
%token LD RD

%token FOR
%token END
%token IF ELSEIF ELSE
%token GLOBAL
%token WHILE

%token FUNCTION
%token RETURN
```

```
%union {
        char* text;
        char* symbol;

        int integerQ;
        double doubleQ;
        imaginary_t imaginaryQ;

        struct {
                int size;
                char** entries;
                } variables;

        struct {
                int size;
                vector_t** vectors;
                } rows;

        matrix_t* matrix;

        struct {
                expr_t* start;
                expr_t* stride;
                expr_t* stop;
                } colon;

        expr_t* expression;

        int delimiter;

        var_expr_t* vexpression;

        struct {
                int size;
                expr_t** expressions;
                } singletons;

        statement_t* statement;

        for_t* forStatement;
        if_t* ifStatement;
        global_t* globalStatement;
        while_t* whileStatement;
        }


%token <text> TEXT
%token <symbol> IDENTIFIER

%token <integerQ> INTEGER
%token <doubleQ> DOUBLE
```

```
%token <imaginaryQ> IMAGINARY

%type <statement> input functionMFile scriptMFile

%type <symbol> f_def_line
%type <statement> f_body
%type <vexpression> f_output f_input f_argument_list

%type <statement> delimited_input delimited_list
%type <statement> statement_list
%type <statement> statement

%type <statement> command_form
%type <statement> for_command for_cmd_list
%type <statement> if_command if_cmd_list opt_else
%type <statement> global_command
%type <statement> while_command while_cmd_list
%type <statement> return_command

%type <expression> expr reference identifier
%type <colon> colon_expr
%type <expression> assignment

%type <delimiter> parenthesis boxes1 boxes2

%type <vexpression> argument_list text_list
%type <vexpression> reference_list

%type <variables> global_decl_list

%type <matrix> matrix s_assignee_matrix m_assignee_matrix

%type <rows> rows

%type <singletons> row row_with_commas


%left AND OR
%left LTHAN LTHANE GTHAN GTHANE EQUAL UNEQUAL
%left ':'
%left '-' '+'
%left '*' EMUL '/' EDIV '\\' ELEFTDIV
%nonassoc NOT UNARYPLUS UNARYMINUS
%nonassoc TRANSPOSE CTRANSPOSE
%left EPOWER POWER


%%


input                   : scriptMFile
```

```
                     {
                      $$ = InputSequence = $1;
                      }
                     | functionMFile
                     {
                      $$ = InputSequence = $1;
                      }
                     | parse_error
                     {
                      $$ = InputSequence = NULL;
                      }
                     ;


scriptMFile          : opt_delimiter
                     {
                      FunctionName = "__main__";
                      cpp_function(FunctionName, 0, 0, 0);

                      $$ = 0;
                      }
                     | opt_delimiter statement_list
                     {
                      FunctionName = "__main__";
                      cpp_function(FunctionName, $2, 0, 0);

                      $$ = $2;
                      }
                     ;


opt_delimiter        :
                     | delimiter
                     ;


delimiter            : null_lines
                     | empty_lines
                     | null_lines empty_lines
                     ;


null_lines           : null_line
                     | null_lines null_line
                     ;


null_line            : ','
                     | ';'
                     | empty_lines ','
                     | empty_lines ';'
```

```
                        ;

empty_lines             : LINE
                        | empty_lines LINE
                        ;

statement_list          : statement opt_delimiter
                        {
                         $$ = $1;
                         }
                        | statement delimiter statement_list
                        {
                         STATEMENT_NEXT($1) = $3;
                         STATEMENT_PREV($3) = $1;

                         $$ = $1;
                         }
                        ;

statement               : command_form
                        {
                         $$ = $1;
                         }
                        | expr
                        {
                         statement_t* const statement =
                         alloc_statement_t(EXPRESSIONstatement);

                         STATEMENT_EXPR(statement) = $1;

                         EXPR_PARENT_STATEMENT($1) = statement;

                         $$ = statement;
                         }
                        | assignment
                        {
                         statement_t* const statement =
                         alloc_statement_t(EXPRESSIONstatement);

                         STATEMENT_EXPR(statement) = $1;

                         EXPR_PARENT_STATEMENT($1) = statement;

                         $$ = statement;
                         }
                        | for_command
                        {
                         $$ = $1;
```

```
                    }
                   | if_command
                   {
                    $$ = $1;
                    }
                   | global_command
                   {
                    $$ = $1;
                    }
                   | while_command
                   {
                    $$ = $1;
                    }
                   | return_command
                   {
                    $$ = $1;
                    }
                   ;


command_form       : identifier text_list
                   {
                    statement_t* const statement =
                    alloc_statement_t(EXPRESSIONstatement);

                    expr_t* const expr =
                    build_expr_nary_op(FUNCTIONARRAYexpr,
                    $1, $2);

                    STATEMENT_EXPR(statement) = expr;

                    EXPR_PARENT_STATEMENT(expr) = statement;

                    $$ = statement;

                    dealloc_var_expr_t_list($2);
                    }
                   ;


text_list          : TEXT
                   {
                    expr_t* expr;

                    if (strlen((const char*)$1) == 1)
                       expr = build_expr_ATOM(
                       build_atom_CHARACTER(*($1)));
                    else
                       expr = build_expr_ATOM(
                       build_atom_STRING($1));
```

```
                        $$ =
                        alloc_var_expr_t(expr);
                        }
                       | text_list TEXT
                       {
                        expr_t* expr;

                        if (strlen((const char*)$2) == 1)
                           expr = build_expr_ATOM(
                           build_atom_CHARACTER(*($2)));
                        else
                           expr = build_expr_ATOM(
                           build_atom_STRING($2));

                        VAR_EXPR_NEXT($1) =
                        alloc_var_expr_t(expr);

                        VAR_EXPR_PREV(VAR_EXPR_NEXT($1)) = $1;

                        $$ = $1;
                        }
                       ;


expr                   : INTEGER
                       {
                        $$ =
                        build_expr_ATOM(build_atom_INTEGER($1));
                        }
                       | DOUBLE
                       {
                        yyinsert_comma_in_input(DOUBLE);

                        $$ =
                        build_expr_ATOM(build_atom_DOUBLE($1));
                        }
                       | IMAGINARY
                       {
                        yyinsert_comma_in_input(IMAGINARY);

                        $$ =
                        build_expr_ATOM(build_atom_IMAGINARY($1));
                        }
                       | TEXT
                       {
                        expr_t* expr;

                        if (strlen((const char*)$1) == 1)
                           expr = build_expr_ATOM(
                           build_atom_CHARACTER(*($1)));
                        else
```

```
   expr = build_expr_ATOM(
   build_atom_STRING($1));

yyinsert_comma_in_input(TEXT);

$$ = expr;
}
| '(' parenthesis expr
{
BracketDelimiter = $2;
} ')'
{
yyinsert_comma_in_input(')');

$$ = $3;
}
| reference
{
$$ = $1;
}
| matrix
{
$$ =
build_expr_MATRIX($1);
}
| expr EPOWER expr
{
$$ =
build_expr_binary_op(EPOWERexpr, $1, $3);
}
| expr POWER expr
{
$$ =
build_expr_binary_op(BIPOWERexpr, $1, $3);
}
| expr TRANSPOSE
{
yyinsert_comma_in_input(TRANSPOSE);

$$ =
build_expr_unary_op(TRANSPOSEexpr, $1);
}
| expr CTRANSPOSE
{
yyinsert_comma_in_input(CTRANSPOSE);

$$ =
build_expr_unary_op(CTRANSPOSEexpr, $1);
}
| NOT expr
{
```

```
 $$ =
 build_expr_unary_op(NOTexpr, $2);
 }
| '+' expr %prec UNARYPLUS
{
 $$ =
 build_expr_unary_op(UNARYPLUSexpr, $2);
 }
| '-' expr %prec UNARYMINUS
{
 $$ =
 build_expr_unary_op(UNARYMINUSexpr, $2);
 }
| expr '*' expr
{
 $$ =
 build_expr_binary_op(BIMULexpr, $1, $3);
 }
| expr '/' expr
{
 $$ =
 build_expr_binary_op(BIDIVexpr, $1, $3);
 }
| expr '\\' expr
{
 $$ =
 build_expr_binary_op(BILEFTDIVexpr, $1, $3);
 }
| expr EMUL expr
{
 $$ =
 build_expr_binary_op(EMULexpr, $1, $3);
 }
| expr EDIV expr
{
 $$ =
 build_expr_binary_op(EDIVexpr, $1, $3);
 }
| expr ELEFTDIV expr
{
 $$ =
 build_expr_binary_op(ELEFTDIVexpr, $1, $3);
 }
| expr '+' expr
{
 $$ =
 build_expr_binary_op(BIPLUSexpr, $1, $3);
 }
| expr '-' expr
{
 $$ =
```

```
 build_expr_binary_op(BIMINUSexpr, $1, $3);
 }
| colon_expr
{
 $$ =
 build_expr_ternary_op(COLONexpr,
 $1.start, $1.stride, $1.stop);
 }
| expr LTHAN expr
{
 $$ =
 build_expr_binary_op(LTHANexpr, $1, $3);
 }
| expr LTHANE expr
{
 $$ =
 build_expr_binary_op(LTHANEexpr, $1, $3);
 }
| expr GTHAN expr
{
 $$ =
 build_expr_binary_op(GTHANexpr, $1, $3);
 }
| expr GTHANE expr
{
 $$ =
 build_expr_binary_op(GTHANEexpr, $1, $3);
 }
| expr EQUAL expr
{
 $$ =
 build_expr_binary_op(EQUALexpr, $1, $3);
 }
| expr UNEQUAL expr
{
 $$ =
 build_expr_binary_op(UNEQUALexpr, $1, $3);
 }
| expr AND expr
{
 $$ =
 build_expr_binary_op(ANDexpr, $1, $3);
 }
| expr OR expr
{
 $$ =
 build_expr_binary_op(ORexpr, $1, $3);
 }
;
```

```
parenthesis             :
                        {
                         $$ = BracketDelimiter;

                         BracketDelimiter = '(';
                         }
                        ;


reference               : identifier
                        {
                         $$ = $1;
                         }
                        | identifier '(' parenthesis argument_list
                        {
                         BracketDelimiter = $3;
                         } ')'
                        {
                         yyinsert_comma_in_input(')');

                         $$ =
                         build_expr_nary_op(FUNCTIONARRAYexpr,
                         $1, $4);

                         dealloc_var_expr_t_list($4);
                         }
                        ;


identifier              : IDENTIFIER
                        {
                         $$ =
                         build_expr_ATOM(build_atom_VARIABLE($1));
                         }
                        ;


argument_list           : ':'
                        {
                         $$ =
                         alloc_var_expr_t(build_expr_ATOM(
                         build_atom_COLON()));
                         }
                        | expr
                        {
                         $$ =
                         alloc_var_expr_t($1);
                         }
                        | ':' ',' argument_list
                        {
                         var_expr_t* const varExpr =
```

```
                             alloc_var_expr_t(build_expr_ATOM(
                             build_atom_COLON()));

                             VAR_EXPR_NEXT(varExpr) = $3;
                             VAR_EXPR_PREV($3) = varExpr;

                             $$ = varExpr;
                             }
                            | expr ',' argument_list
                            {
                             var_expr_t* const varExpr =
                             alloc_var_expr_t($1);

                             VAR_EXPR_NEXT(varExpr) = $3;
                             VAR_EXPR_PREV($3) = varExpr;

                             $$ = varExpr;
                             }
                            ;


matrix                      : '[' boxes1 rows
                            {
                             BracketDelimiter = $2;
                             } ']'
                            {
                             yyinsert_comma_in_input(']');

                             $$ =
                             build_matrix_t($3.size, $3.vectors);
                             }
                            ;


boxes1                      :
                            {
                             $$ = BracketDelimiter;

                             BracketDelimiter = '[';
                             }
                            ;


rows                        :
                            {
                             $$.vectors = 0;

                             $$.size = 0;
                             }
                            | row
                            {
```

```
                       $$.vectors = (vector_t**)
                       alloc_pointers(1);

                       $$.size = 1;

                       *($$.vectors+$$.size-1) =
                       build_vector_t($1.size, $1.expressions);
                       }
                      | rows ';'
                      {
                       $$ = $1;
                       }
                      | rows ';' row
                      {
                       $$.vectors = (vector_t**)
                       realloc_pointers((void*)$1.vectors,
                       $1.size+1);

                       $$.size = $1.size+1;

                       *($$.vectors+$$.size-1) =
                       build_vector_t($3.size, $3.expressions);
                       }
                      | rows LINE
                      {
                       $$ = $1;
                       }
                      | rows LINE row
                      {
                       $$.vectors = (vector_t**)
                       realloc_pointers((void*)$1.vectors,
                       $1.size+1);

                       $$.size = $1.size+1;

                       *($$.vectors+$$.size-1) =
                       build_vector_t($3.size, $3.expressions);
                       }
                      ;


row                   : expr
                      {
                       $$.expressions = (expr_t**)
                       alloc_pointers(1);

                       $$.size = 1;

                       *($$.expressions+$$.size-1) = $1;
                       }
                      | row_with_commas
```

```
                        {
                         $$ = $1;
                         }
                        | row_with_commas expr
                        {
                         $$.expressions = (expr_t**)
                         realloc_pointers((void*)$1.expressions,
                         $1.size+1);

                         $$.size = $1.size+1;

                         *($$.expressions+$$.size-1) = $2;
                         }
                        ;


row_with_commas         : expr ','
                        {
                         $$.expressions = (expr_t**)
                         alloc_pointers(1);

                         $$.size = 1;

                         *($$.expressions+$$.size-1) = $1;
                         }
                        | row_with_commas expr ','
                        {
                         $$.expressions = (expr_t**)
                         realloc_pointers((void*)$1.expressions,
                         $1.size+1);

                         $$.size = $1.size+1;

                         *($$.expressions+$$.size-1) = $2;
                         }
                        ;


colon_expr              : expr ':' expr
                        {
                         $$.start = $1;

                         if (LOOKAHEAD != ':')
                            {
                             $$.stride =
                             build_expr_ATOM(build_atom_INTEGER(1));
                             $$.stop = $3;
                             }
                         else
                            {
                             $$.stride = $3;
```

```
                              $$.stop = 0;
                              }
                          }
                        | colon_expr ':' expr
                        {
                         if ($1.stop)
                            {
                             $$.start =
                             build_expr_ternary_op(COLONexpr,
                             $1.start, $1.stride, $1.stop);

                             if (LOOKAHEAD != ':')
                                {
                                 $$.stride =
                                 build_expr_ATOM(
                                 build_atom_INTEGER(1));
                                 $$.stop = $3;
                                 }
                             else
                                {
                                 $$.stride = $3;
                                 $$.stop = 0;
                                 }
                             }
                         else
                            {
                             $$.start = $1.start;
                             $$.stride = $1.stride;
                             $$.stop = $3;
                             }
                         }
                        ;


assignment              : reference '=' expr
                        {
                         $$ =
                         build_expr_binary_op(ASSIGNMENTexpr, $1, $3);
                         }
                        | s_assignee_matrix '=' expr
                        {
                         $$ =
                         build_expr_binary_op(ASSIGNMENTexpr,
                         build_expr_MATRIX($1), $3);
                         }
                        | m_assignee_matrix '=' reference
                        {
                         $$ =
                         build_expr_binary_op(ASSIGNMENTexpr,
                         build_expr_MATRIX($1), $3);
                         }
```

```
                        ;


s_assignee_matrix       : LD boxes2 reference
                        {
                         BracketDelimiter = $2;
                         } RD
                        {
                         expr_t** const singletons = (expr_t**)
                         alloc_pointers(1);

                         *singletons = $3;

                         vector_t** const vectors = (vector_t**)
                         alloc_pointers(1);

                         vectors[0] =
                         build_vector_t(1, singletons);

                         $$ =
                         build_matrix_t(1, vectors);
                         }
                        ;


m_assignee_matrix       : LD boxes2 reference ',' reference_list
                        {
                         BracketDelimiter = $2;
                         } RD
                        {
                         var_expr_t* varExpr;
                         int length;

                         for (length = 1, varExpr = $5;
                         VAR_EXPR_NEXT(varExpr);
                         length++, SET_TO_NEXT(varExpr));

                         expr_t** const singletons = (expr_t**)
                         alloc_pointers(length+1);

                         *singletons = $3;

                         for (length = 1, varExpr = $5;
                         VAR_EXPR_NEXT(varExpr);
                         length++, SET_TO_NEXT(varExpr))
                         *(singletons+length) =
                          VAR_EXPR_DATA(varExpr);

                         *(singletons+length) =
                          VAR_EXPR_DATA(varExpr);
```

```
                        vector_t** const vectors = (vector_t**)
                        alloc_pointers(1);

                        *vectors =
                        build_vector_t(length+1, singletons);

                        $$ =
                        build_matrix_t(1, vectors);

                        dealloc_var_expr_t_list($5);
                        }
                       ;


boxes2                 :
                       {
                        $$ = BracketDelimiter;

                        BracketDelimiter = LD;
                        }
                       ;


reference_list         : reference
                       {
                        $$ =
                        alloc_var_expr_t($1);
                        }
                       | reference ',' reference_list
                       {
                        var_expr_t* const varExpr =
                        alloc_var_expr_t($1);

                        VAR_EXPR_NEXT(varExpr) = $3;
                        VAR_EXPR_PREV($3) = varExpr;

                        $$ = varExpr;
                        }
                       ;


for_command            : FOR for_cmd_list END
                       {
                        $$ = $2;
                        }
                       ;


for_cmd_list           : identifier '=' expr delimited_input
                       {
                        statement_t* const statement =
```

```
                          alloc_statement_t(FORstatement);

                          for_t* const forSt =
                          alloc_for_t();

                          STATEMENT_FOR(statement) = forSt;

                          FOR_VARIABLE(forSt) = $1;
                          FOR_EXPRESSION(forSt) =
                          build_expr_binary_op(ASSIGNMENTexpr, $1, $3);

                          FOR_PARENT(forSt) = statement;
                          FOR_BODY(forSt) = $4;

                          set_owner_of_list(statement, $4);

                          EXPR_PARENT_STATEMENT(FOR_EXPRESSION(forSt)) =
                          statement;

                          $$ = statement;
                          }
                         ;


if_command               : IF if_cmd_list END
                         {
                          $$ = $2;
                          }
                         ;


if_cmd_list              : expr delimited_input opt_else
                         {
                          statement_t* const statement =
                          alloc_statement_t(IFstatement);

                          if_t* const ifSt =
                          alloc_if_t();

                          STATEMENT_IF(statement) = ifSt;

                          IF_CONDITION(ifSt) = $1;
                          IF_PARENT(ifSt) = statement;
                          IF_BODY(ifSt) = $2;

                          IF_ELSE_BODY(ifSt) = $3;

                          set_owner_of_list(statement, $2);
                          set_owner_of_list(statement, $3);

                          EXPR_PARENT_STATEMENT($1) = statement;
```

```
                         $$ = statement;
                         }
                        ;


opt_else                :
                        {
                         $$ = 0;
                         }
                        | ELSE delimited_input
                        {
                         $$ = $2;
                         }
                        | ELSEIF expr delimited_input opt_else
                        {
                         statement_t* const statement =
                         alloc_statement_t(IFstatement);

                         if_t* const ifSt =
                         alloc_if_t();

                         STATEMENT_IF(statement) = ifSt;

                         IF_CONDITION(ifSt) = $2;
                         IF_PARENT(ifSt) = statement;
                         IF_BODY(ifSt) = $3;

                         IF_ELSE_BODY(ifSt) = $4;

                         set_owner_of_list(statement, $3);
                         set_owner_of_list(statement, $4);

                         EXPR_PARENT_STATEMENT($2) = statement;

                         $$ = statement;
                         }
                        ;


global_command          : GLOBAL global_decl_list
                        {
                         global_t* const globalSt =
                         alloc_global_t();

                         GLOBAL_LENGTH(globalSt) = $2.size;
                         GLOBAL_ENTRIES(globalSt) = $2.entries;

                         statement_t* const statement =
                         alloc_statement_t(GLOBALstatement);
```

```
                         STATEMENT_GLOBAL(statement) = globalSt;
                         GLOBAL_PARENT(globalSt) = statement;

                         $$ = statement;
                         }
                        ;


global_decl_list        : IDENTIFIER
                        {
                         $$.entries = (char**)
                         alloc_pointers(1);

                         $$.size = 1;

                         *($$.entries+$$.size-1) =
                         strcpy(alloc_string(strlen($1)+1), $1);
                         }
                        | global_decl_list IDENTIFIER
                        {
                         $$.entries = (char**)
                         realloc_pointers((void*)($1.entries),
                         $1.size+1);

                         $$.size = $1.size+1;

                         *($$.entries+$$.size-1) =
                         strcpy(alloc_string(strlen($2)+1), $2);
                         }
                        ;


while_command           : WHILE while_cmd_list END
                        {
                         $$ = $2;
                         }
                        ;


while_cmd_list          : expr delimited_input
                        {
                         statement_t* const statement =
                         alloc_statement_t(WHILEstatement);

                         while_t* const whileSt =
                         alloc_while_t();

                         STATEMENT_WHILE(statement) = whileSt;

                         WHILE_CONDITION(whileSt) = $1;
                         WHILE_PARENT(whileSt) = statement;
```

```
                         WHILE_BODY(whileSt) = $2;

                         set_owner_of_list(statement, $2);

                         EXPR_PARENT_STATEMENT($1) = statement;

                         $$ = statement;
                         }
                        ;


return_command          : RETURN
                        {
                         statement_t* const statement =
                         alloc_statement_t(RETURNstatement);

                         return_t* const returnSt =
                         alloc_return_t();

                         STATEMENT_RETURN(statement) = returnSt;

                         $$ = statement;
                         }
                        ;


delimited_input         : opt_delimiter
                        {
                         $$ = 0;
                         }
                        | opt_delimiter delimited_list
                        {
                         $$ = $2;
                         }
                        ;


delimited_list          : statement delimiter
                        {
                         $$ = $1;
                         }
                        | statement delimiter delimited_list
                        {
                         STATEMENT_NEXT($1) = $3;
                         STATEMENT_PREV($3) = $1;

                         $$ = $1;
                         }
                        ;
```

```
functionMFile        : empty_lines f_def_line f_body
                     {
                      FunctionName = $2;

                      $$ = $3;
                      }
                     | f_def_line f_body
                     {
                      FunctionName = $1;

                      $$ = $2;
                      }
                     ;


f_def_line           : FUNCTION f_output '=' IDENTIFIER f_input
                     {
                      extern file* mfile;

                      cpp_set_file(mfile);
                      cpp_function($4, 0, $2, 0);
                      cpp_set_args($5);

                      $$ = $4;

                      dealloc_var_expr_t_list($5);
                      }
                     | FUNCTION IDENTIFIER f_input
                     {
                      extern file* mfile;

                      cpp_set_file(mfile);
                      cpp_function($2, 0, 0, 0);
                      cpp_set_args($3);

                      $$ = $2;

                      dealloc_var_expr_t_list($3);
                      }
                     ;


f_output             : identifier
                     {
                      $$ =
                      alloc_var_expr_t($1);
                      }
                     | LD f_argument_list RD
                     {
                      $$ = $2;
                      }
```

```
                        ;


f_input                 :
                        {
                         $$ = 0;
                         }
                        | '(' ')'
                        {
                         $$ = 0;
                         }
                        | '(' f_argument_list ')'
                        {
                         $$ = $2;
                         }
                        ;


f_argument_list         : identifier ',' f_argument_list
                        {
                         var_expr_t* const varExpr =
                         alloc_var_expr_t($1);

                         VAR_EXPR_NEXT(varExpr) = $3;
                         VAR_EXPR_PREV($3) = varExpr;

                         $$ = varExpr;
                         }
                        | identifier
                        {
                         $$ =
                         alloc_var_expr_t($1);
                         }
                        ;


f_body                  : delimiter statement_list
                        {
                         cpp_set_stmt_list((const statement_t*)$2);

                         $$ = $2;
                         }
                        | opt_delimiter
                        {
                         cpp_set_stmt_list((const statement_t*)0);

                         $$ = 0;
                         }
                        ;
```

```
parse_error             : LEXERROR
                        {
                         yyerror("Lexical error!");
                         }
                        | error
                        ;


%%


static void yyerror(const char* message)
{
 extern unsigned int SourceLine;


 DB0(DB_HPM_yacc, "Entering <yyerror> ...\n");

 if (InputFileName)
    fprintf(stderr, "Error in %s around line %u: %s\n",
    InputFileName, SourceLine, message);
 else
    fprintf(stderr, "Error around line %u: %s\n",
    SourceLine, message);

 exit(1);
 }
```