Handling Context-Sensitive Syntactic Issues in the Design of a Front-end for a MATLAB Compiler[1]

Pramod G. Joisha, Abhay Kanhere, Prithviraj Banerjee, U. Nagaraj Shenoy, and 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

E-mail: {fpjoisha, abhay, banerjee, nagaraj, choudharg}@ece.nwu.edu

Abstract [1]

In recent times, the MATLAB language has emerged as a popular alternative for programming in diverse application domains such as signal processing and meteorology. The language has a powerful array syntax with a large set of pre-defined operators and functions that operate on arrays or array sections, making it an ideal candidate for applications involving substantial array-based processing.

Yet, for all the programming convenience that the language offers, designing a parser and scanner capable of mimicking the language's syntax has proven to be an acutely difficult task. The language has many context-sensitive constructions, and though numerous front-end implementations of MATLAB and MATLAB-like languages exist, not much has been discussed regarding the efficient compile-time parsing of such languages or how its syntax impacts the parsing process.

In this paper, we present the design and implementation of a compiler front-end for the MATLAB language. 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 complications arising in parsing command-form function invocations within a compile-time environment, the context-sensitive interpretation of the single quote character, and the translation of white space within matrices into element separators.

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.

The front-end was designed and implemented as part of the MATCH project, which addresses the translation of a MATLAB program by a compiler onto a heterogeneous target consisting of embedded and commercial-off-the-shelf processors.

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

1 Introduction

The MATCH project [1] concerns itself with the task of efficiently compiling code written in MATLAB[2] a for a heterogeneous target system comprising embedded processors, digital signal processors (DSPs) and field programmable gate arrays (FPGAs) [5]. 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 publi-

[1] This research was supported by DARPA under Contract F30602--98--2--0144.

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

cally available automatic parser and scanner generators.

MATLAB is a high performance language geared toward technical computing [12]. The language provides powerful features, which enable matrices and arrays to be efficiently and easily manipulated. The very high-level nature of these features makes the usage of the language very intuitive. In fact, the language's simplicity and ease of use are among the primary reasons behind its immense popularity in various application areas. The language's syntactic simplicity does not compromise its expressive power; this fact along with the interactive nature of the system and the fast prototyping that it empowers have made it the language of choice in research, analysis and development.

1.1 Motivation

From the perspective of syntax analysis, MATLAB offers numerous challenges whose subtlety makes them interesting exercises in parser design. Examples of these include tackling the single quote character, efficiently building uniform colon expressions and handling matrix constructs. In fact, some of the language's traits make parsing in a compile-time environment a much more complicated task and this resulted in certain modifications to the set of language features that were finally supported by our compiler.

For instance, the syntax for invoking a function in the command form is fraught with parsing ambiguity when compiled. Because of dynamic binding, a particular name could refer to either a variable or a function depending on the path of execution. As a consequence (see section 3), in certain cases, it may not be possible to establish at compile-time whether statements such as

```
A␣+1;
```

correspond to a binary addition expression or to a function invocation in the command form. In this paper, the notation ␣ will be used to denote a mixture of one or more blanks and horizontal tabs.

Another example is the single quote character which is used both as a complex conjugate transpose operator as well as a string literal demarcator [12]. This dual role can lead to lexical matching problems since the character can be associated with two tokens: the CTRANSPOSE token (corresponding to the complex conjugate transpose operator) or the TEXT token (corresponding to the string literal). Thus, while in the following MATLAB code fragment

```
Hello=1;
disp␣'*Hello';
```

the single quote character sets off a string literal[3], in

```
Hello=1;
disp=1;
disp'*Hello';
```

it denotes the complex conjugate transpose operator . In the latter code fragment, if there was no assignment to `disp` before the statement `disp'*Hello'`, an error would have resulted. That is, in the absence of a preceding assignment to `disp`, `disp'*Hello'` would not have been regarded as a function invocation even though the built-in function `disp` is visible at that point. Alternately, if an assignment to `disp` had pre- ceded the statement `disp␣'*Hello'`, an error would have occurred. This issue is further elaborated in section 5.

Matrices also pose certain non-trivial obstacles to parsing. 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 on the left 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.

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

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)`. The lines on the right show the same matrix constructed using commas and semicolons.

The above examples serve to demonstrate the complexity of the parsing and scanning process in MATLAB, especially when a formal grammar and lexical description are not publicly available, and when such a specification has to be designed, duplicating as much of the language's observed syntax and behavior as possible. Some of these problems are peculiar to a compiler framework, since in the presence of interpretation, the control flow is known by the time a statement is parsed and executed. We mention and discuss these issues in this paper, describing the solutions that we have adopted to solve them in our implementation.

[3] The MATLAB builtin function disp displays its argument.

1.2 Background

Work in building a front-end began by experimenting with the Free Software Foundation's distribution for GNU Octave [3], 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 with 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 and scripts can be processed by the front-end. Additional constructs such as `switch` and `break` statements can be easily handled with relatively little modification to the current grammar.

The main tools that were used in implementing the front-end were bison and flex. Bison is an automatic parser generator in the style of yacc [9] (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 [11] (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.

1.3 Related work

Recently, there has been much interest and work in compiling MATLAB programs into object code [2, 15, 14, 3]. However, not much has been discussed regarding the language's syntactic nuances or how they may be handled in a compile-time environment. To the best of our knowledge, we are not aware of any previously published work that discusses the parsing of the MATLAB language.

The MATLAB language is in many ways a new incarnation of the APL language [13]. Both languages advocate a functional style of programming, support a large repository of built-in (or primitive) functions and treat data in much the same way---that is, as arrays. In fact, there often exist direct correspondences between APL's primitive functions and MATLAB's built-in functions. A case in point is the ρ operator in APL. In its monadic role, it resembles MATLAB's `size` built-in function, while in its dyadic role, it behaves like MATLAB's `reshape` built-in function. Another example is the monadic ι (iota) APL primitive, also known as the index generator function. This operator resembles a specialized version of MATLAB's `colon` built-in function.

However, from the standpoint of syntax analysis, the two languages present markedly different issues. The APL language syntax is so regular that it can almost be recognized by a finite state automaton [7]. On the other hand, due to the context-sensitivity of the MATLAB language, it appears that the language's syntax cannot be described by a conventional LALR(1) specification alone. On the flip-side, the same context-sensitivity has imparted to the language an intuitiveness and richness that is among the chief reasons behind its widespread popularity. Though APL is a rich language in its own right, the same "regularness" is probably also the reason behind the language's notoriously cryptic syntactic structure.

In [8], a non-recursive parsing algorithm that uses a two-symbol lookahead and that shifts between two parsing states is briefly mentioned for a *restricted* version of the APL language. These restrictions included disallowing the usage of function names as either variable or label names, thereby eliminating the possibility of parsing ambiguities arising from late binding. Since supporting all of APL's features would necessarily entail *some* run-time parsing, researchers have proposed systems that rely on "entry-time partial parsers" and "run-time parser completers" [16] to mitigate the run-time parsing overhead. Though such methodologies can be carried over to MATLAB, it is not clear how these techniques can be incorporated in an optimizing compiler framework. Approaches such as [17] parse and compile a class of APL programs that do not utilize features that could dynamically change the syntactic meaning of the program's statements. This approach is probably justifiable in light of empirical evidence that suggests that most APL coders abstain from using language features that alter the syntactic structure of the program with each execution instance [16, 6]. By excluding support for some of MATLAB's language constructs, this is essentially the philosophy that we also adopt in the MATCH compiler.

The complete source code containing the lexical specification and the context-free grammar is available as an appendix to [10], a technical report that describes the design and implementation of the MATCH compiler front-end in greater detail.

1.4 Outline

The rest of the paper is organized as follows. In § 2, we provide an overview of the MATLAB language, describing in brief some of its lexical aspects, besides introducing the notion of M-files and showing a sample MATLAB program. Terms such as "command form" and "command-form invocation" will be explained in this section. In § 3, we consider the implications of MATLAB's command-form function invocation syntax and argue the reasons for supporting a limited version of that syntax in our compiler. In § 4, we account for the grammar rules that enable assignments to matrix-like left-hand sides which may also contain multiple variables. The dual role played by the single quote character and the issues that it entails are presented in § 5. Matrices in MATLAB and the manner in which commas are inserted to separate elements are explained in § 6. In § 7, we show how the syntax directed translation process can be leveraged to parse all colon expressions to a uniform full ternary tree form. Finally, the structure of the conditional statement as well as the shift-reduce conflicts that its grammar rules give rise to are discussed in § 8.

2 Language Preliminaries

A MATLAB program basically consists of a sequence of statements. A statement in MATLAB could be a function call, an expression, an assignment, a control construct or a `global` declaration. For example, the statement

```
disp('Hello␣World!');
```

invokes the built-in function disp that displays its argument on the standard output. The disp function does not return a value; functions that do can be used to build expressions. For instance,

```
r=rand(2)+1;
```

produces a 2 ×2 matrix of random values between 1 and 2 and assigns the result to the variable `r`. This is an example of an assignment statement.

2.1 Command/Function duality

In MATLAB, functions can be invoked in two ways. In addition to the typical way of calling a function as shown earlier with the `disp` and `rand` built-in functions, MATLAB also allows for "command-form" function invocations. The `disp('Hello␣World!')` example shown above could have been rewritten as

```
disp␣'Hello␣World!';
```

and the effect would have been the same. In general, any function *f* that accepts a string argument *e* can be invoked in the *functional form* (i.e., as *f(e)*) or in the *command form* (i.e., as *f*␣*e*) [12]. In the latter form, the invocation is called a *command-form function invocation* of *f*.

2.2 Lexical Specification Overview

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 definition*[4] `INTEGER` represents numeric quantities that are free of a decimal point and exponent; all other numeric quantities correspond to the `DOUBLE` name definition. 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`. Figure 1 formally describes some of the name definitions used in the lexical specification.

```
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]
```

Figure 1: Name Definitions

In MATLAB, there exist a couple of situations in which horizontal spaces become significant. A *horizontal space* is either a blank or a horizontal tab and is denoted by the name definition `HSPACE` shown in Figure 1. Thus, the symbol ␣ represents the lexical pattern matched by the `HSPACES` name definition, which is *at least* one horizontal space. Apart from functioning as token demarcators and matrix element separators, horizontal spaces can also influence the interpretation of succeeding character sequences. In the MATCH compiler front-end, they are cast away by

[4] Name definitions are basically shorthands that simplify the main scanner specification (see flex(1)).

the scanner, so that the parser sees a token stream free of any 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 newline, carriage-return or form-feed character (i.e., a NEWLINE lexical pattern). Everything from the ellipsis until and including the NEWLINE character—or until the end of the input—is ignored.[5] Comments likewise begin at a percent character (%) and continue until a NEWLINE character, or until the end of the input.

2.3 M-files

Statements in a MATLAB program are separated from each other by *delimiters*. Delimiters are sequences consisting of an arbitrary mixture of comma (','), semicolon (';') and LINE tokens. 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 former are often elaborately referred to as function M-files to distinguish them from built-in functions. Other than being available in files, there is no difference between the two. 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 *non-empty* line in a function must be the *function definition line* [10]. 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. A program file that is a script may either contain optional delimiters, or contain a sequence of statements optionally preceded by delimiters. Additionally, external programs can be made to act like MATLAB functions using the *shell escape* mechanism [12].

2.4 Expressions

Expressions in MATLAB are composed as operations on subexpressions. Subexpressions could be functions returning values, results of an array section operation, identifiers, matrices, string literals 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 `2j` will always be considered as an imaginary quantity irrespective of any preceding definitions against the variable `j`.

```
% Create a 1024 x 1024 array, initialized
to 0.
a(1:1024,1:1024)=0;
% Set all values along the first column to
1+i.
a(:,1)=1+i;
%
% The Jacobi iteration.
%
a(2:1023,2:1023)=(a(1:1022,2:1023) ...
+a(3:1024,2:1023)+a(2:1023,1:1022) ...
+a(2:1023,3:1024))/4;
```

Figure 2: The Jacobi Iteration

To illustrate, we show a sample MATLAB code fragment that performs the well-known Jacobi iteration in Figure 2. The Jacobi iteration executes a four-point stencil computation in which the values at each interior grid point (1022×1022 in all) are updated as an average of the values associated with its four cardinal neighbors. This averaging operation is succinctly expressed using MATLAB's vector notation. In the above code fragment, the right-hand side of the last assignment statement adds the values at corresponding grid points in four conforming array sections and averages the result. The assignment to another conforming array section achieves the desired update action.

[5] At the time of this writing, the complete line continuation specifics for MATLAB (in both versions 5.0 and 5.2) seemed non-uniform and inconsistent. This is because in certain cases, the line continuation sequence behaved as a token demarcator, while not in other cases. For instance, while the following lines

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

resulted in the value of disp1 being displayed, the following lines

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

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

3 Commands

As mentioned earlier, functions can either be invoked in the functional form or in the command form in MATLAB. 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`.

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 invocation corresponds to a normal function call.

For example, the following lines

```
disp␣1+2;
disp␣(1+2);
```

display `1+2` and `3` as their respective results.

Therefore, if `A.m` is a function M-file, the statement `A␣+1` results in an attempt to invoke the function M-file with the *string* argument `+1`. In fact, `A.m` must be defined as a function M-file that accepts at least one input argument for this particular line of input to work; otherwise MATLAB complains of an error.

```
command_form        :   name text_list
                    ;
text_list           :   TEXT
                    |   text_list TEXT
                    ;
name                :   identifier
                    ;
identifier          :   IDENTIFIER
                    ;
```

Figure 3: Productions for command form

In the MATCH compiler, sentences that derive from the command form non-terminal shown in Figure 3 require arguments to the function to be explicitly quoted.[6] That is, only strings such as `type␣'rank'` reduce to the command form non-terminal in the compiler front-end. Sentences such as `type␣rank` are not recognized as command-form function invocations and result in a parse error. The MATLAB code fragment in Figure 4 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;
```

Figure 4: A Syntactically Ambiguous Code Fragment

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 Figure 5 qualify as possible parse trees for the above statement.

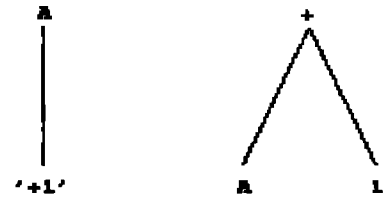

Figure 5: Candidate Parse Trees

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 left AST shown in Figure 5. That is, the statement is always considered to be a command-form function invocation irrespective of whether `A` accepts an input argument and whether `A` produces a return value. On the other hand, if the statement were `A+1`, MATLAB will always parse it into the right AST shown in Figure 5, irrespective of whether `A` accepts an input argument and whether `A` returns a value. In this case, the interpretation is that the value returned by an invocation of the function `A` without arguments is added to the

[6] We shall present grammar rules in a format compliant with bison's input syntax (see bison(1)). In general, non-terminal symbol names will be in lowercase and terminal symbol names will be in uppercase.

numeric constant 1. Hence, if it turns out that A does not return a value, the execution results in an error.

The problem cited here is essentially one stemming from delayed binding. In other words, it may not be possible to conclusively say whether the A in A⌷+1 is a variable or a function M-file until run-time. In that way, the problem has an analogy in APL. Expressions such as e+f in APL could be statically ambiguous if the binding of e is not known until run-time. Depending on whether e is bound to a monadic defined function or a variable, the expression could mean either e(+(f)) or (e)+(f).

A strategy by which our compiler could have supported the interpreter's full-fledged command-form function invocation syntax would have been to create and maintain 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, we decided to eliminate the potential for such parsing ambiguities by supporting a limited version of MATLAB's command-form function invocation syntax, and by flagging ambiguities to the programmer.

4 Assignments

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);
```

4.1 Left-hand sides

To describe the above three varieties of assignment statements, the productions shown in Figure 6 can be used. The non-terminals s_assignee_matrix and m_assignee_matrix represent left-hand sides that are enclosed within a pair of box brackets. While the former non-terminal captures single value left-hand sides, the latter expands to multiple value left-hand sides. These non-terminals are formally elaborated in Figure 7.

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

Figure 6: Productions for assignment

Do we need the non-terminals s_assignee_matrix and m_assignee_matrix shown in Figure 7 to denote box-bracketed left-hand sides? Can't we rely on productions which expand to matrices to represent such left-hand side constructions? The answer is no. Though the non-terminals s_assignee matrix and m_assignee matrix correspond to sentences that largely resemble matrices, they 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

```
reference          : name
                   | name
                   '(' argument-list ')'
                   ;
name               : identifier
                   ;
identifier         : IDENTIFIER
                   ;
argument_list      : ':'
                   | expr
                   | ':' ',' argument_list
                   | expr ',' argument_list
                   ;
s_assignee_matrix  : LD reference RD
                   ;
m_assignee_matrix  : LD reference
                     ',' reference_list RD
                   ;
reference_list     : reference
                   | reference
                   ',' reference-list
                   ;
```

Figure 7: Productions for the Left-hand Side

where the resemblance stops. First, while matrices allow for an optional comma after the last element, box-bracketed left-hand sides do not. Second, semicolons and LINE token delimiters cannot be used to separate elements in a box-bracketed left-hand side.

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. Notice that though the last three differences could have been handled by incorporating semantic checks in the action part of the matrix productions, the first two differences are purely syntactic and warrant the introduction of new non-terminals to describe such left-hand sides.

The non-terminal `expr` in the `assignment` and argument list productions corresponds to a MATLAB *expression* (see § 2.4). It must be noted that the MATCH grammar 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.

4.2 The LD and RD tokens

The tokens `LD` and `RD` shown in Figure 7 actually correspond to the lexemes [and] respectively. These lexemes also correspond to the tokens '[' and ']' that the lexical analyzer returns when matrices are encountered (see § 6). Hence, why do we need to distinguish the [and] lexemes in these two situations?

If the grammar rules for `s_assignee_matrix` and `m_assignee_matrix` assignee matrix had used the '[' and ']' tokens to enclose the elements, a reduce-reduce conflict would have arisen. To illustrate this, suppose that the token stream seen so far by the parser were '[', 'a' and ']'. Then, the parser could either be in the middle of an assignment or an expression. That is, until the '=' token is seen, the parser could either be in the midst of a sentence that ultimately reduces to the `s_assignee_matrix` or `m_assignee_matrix` non-terminals, or in the midst of a sentence that ultimately reduces to a matrix. 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` non-terminals must prevail in an assignment context, while a reduction to the matrix non-terminal must prevail in an expression context.

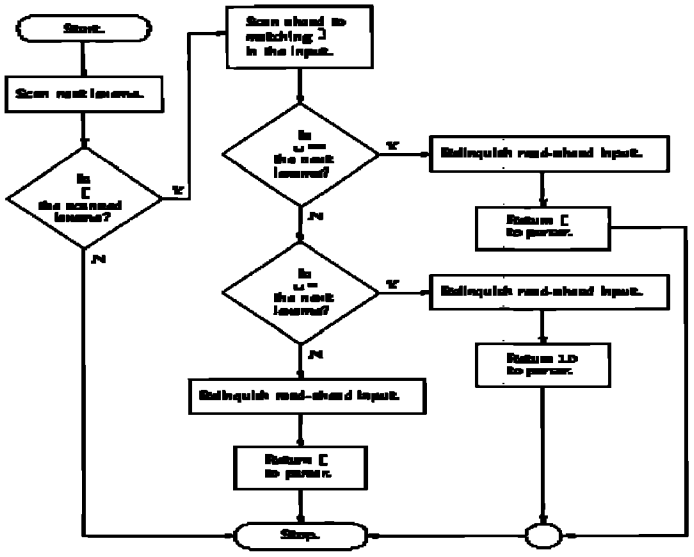

Figure 8: The LD, '[' Tokens

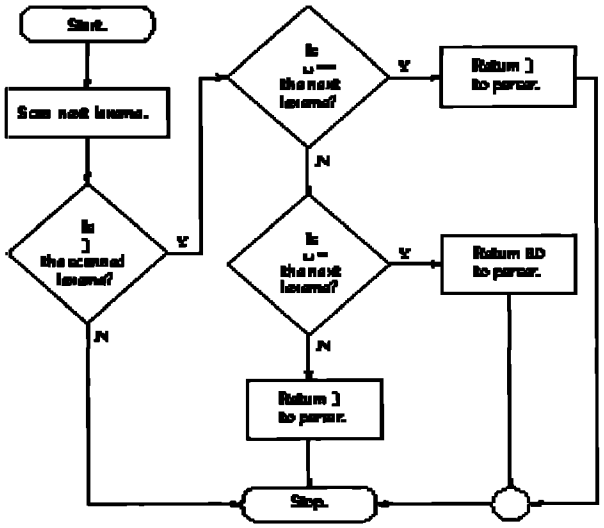

Figure 9: The RD, ']' Tokens

To remedy the above problem, the scanner returns a separate pair of tokens whenever an assignment to a box-bracketed structure is detected. The way this is done is illustrated in Figure 8 and Figure 9. The steps shown in these flow charts are implemented in the lexical specification. For instance, to determine whether an `LD` or '[' token needs to be returned on encountering the [lexeme, the scanner reads ahead until a matching box bracket is found. This read ahead is performed in the action part of the `\[` extended regular expression. If a matching closing box bracket is not found, the lexical analyzer returns the token `LEXERROR` since this is an error situation. If a matching closing box bracket is found, the scanner determines whether the next lexeme in the input is ␣ ==. This is because while `[a,b]=size(1)` is an assignment to a box-bracketed structure, `[a,b]==size(1)` is an expression. Here we use the symbol ␣ to denote *zero or more* horizontal spaces. If the next lexeme is indeed ␣ ==, then the opening box bracket is part of a matrix that occurs on the left-hand side of an equality comparison expression. Thus, the scanner needs to

return the '[' token in this situation. If the next lexeme in the input stream is instead  =, an assignment to a box-bracketed structure is detected. Therefore, the scanner returns the LD token in this case. If some other sequence of characters occurs as the next lexeme in the input stream, the scanner once again returns the '[' token. In all cases, the lexical analyzer returns control to the parser only after relinquishing the characters that were read ahead back to the input.

5 The Single Quote Character

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. 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.[7]

The MATCH scanner reproduces MATLAB's single quote semantics by using start conditions. The start condition mechanism essentially enables the scanner to "activate" only a subset of the rules in its lexical specification depending on its current state. In the absence of explicitly declared start conditions, the scanner always exists in a single state which 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

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

as `,`, `(` or `*`.[8] 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 it is not preceded by horizontal spaces. 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 Matrices

The syntactic constructs that MATLAB offers to enter matrices are arguably among the most powerful features of the language. They enable a programmer to specify a matrix in the most intuitive manner possible. For example,

```
[1␣2          [1,2;
      ⇔
3␣4]          3,4]
```

indicates a 2 by 2 matrix in MATLAB in which elements in the first row are `1` and `2`, while those in the second row are `3` and `4`. The language also provides the programmer the flexibility of using commas as demarcators among row elements, and semicolons as row separators. Thus, while the lines on the left show the matrix constructed by using horizontal spaces and newlines, the lines on the right show the same matrix constructed by using commas and semicolons instead.

While the ability to input a matrix by visually laying out its elements enhances the user-friendliness of the MATLAB language, it considerably complicates the design of a front-end for it. In the MATCH compiler front-end, the parser does not see any horizontal space. By working in tandem with the lexical analyzer, commas are either inserted between the *yet to be* scanned matrix elements, or the horizontal spaces among them are converted to commas.

Within matrices, commas can be potentially inserted 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 (`]`). We say potentially because a comma insertion is not implied in all cases. For instance, while `[1␣+2]` and `[1,+2]` are equivalent in MATLAB, the same is not true for `[1␣+␣2]` and `[1,+2]`. In MATLAB, `[1␣+␣2]` is equivalent to `[1+2]`.

6.1 Bracket nesting

Since horizontal spaces are stripped from the input by the time the parser gets to process the token stream, the front-end must know when these spaces have to be treated specially, and when they can be simply ignored. In the MATCH compiler front-end, this special treatment is indicated by what is called the *bracket nesting* of each character in the input. The bracket nesting (BN) of a character is defined to be the token corresponding to the last unmatched opening box bracket or opening parenthesis seen thus far in the input. The BN of a character can therefore be '[', LD, '(' or "undefined". We denote an undefined BN by ϕ. For example, given the input sequence `[x(1)]=2`, the character BNs are ϕ, LD, LD, '(', LD, ϕ, ϕ and ϕ respectively. That is, while `1` has a BN of '(', all the other characters within the box brackets have a BN of LD.

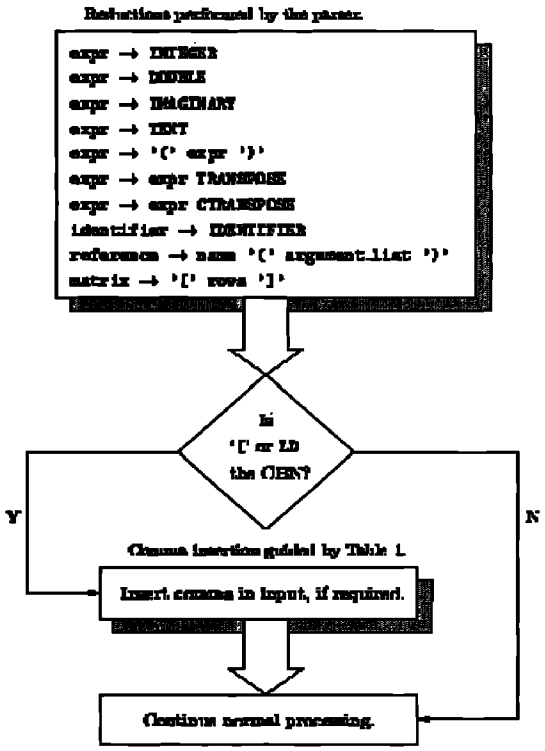

Figure 10: Comma Insertion Among Matrix Elements

The remaining characters in the given input sequence have undefined BNs. We refer to the BN of the

[8] For a full list, please refer to the lexical specification in [10].

character currently being scanned by the lexical analyzer as the *current bracket nesting* (CBN) of the input stream. We thus see that horizontal spaces are significant only when their BN is either '[' or LD. Hence, while [1␣+2] is equivalent to [1,+2], [(1␣+2)] (or [(1+␣2)] or [(1␣+␣2)]) is the same as [(1+2)].

Last Processed Token	Next Lexeme Prefix	
	Before	After
INTEGER, DOUBLE, IMAGINARY, TEXT,), CTRANSPOSE, TRANSPOSE,], IDENTIFIER	▒,	,
	▒;	;
	▒]	]
	▒*	*
	▒^	^
	▒:	:
	▒<	<
	▒>	>
	▒=	=
	▒&	&
	▒|	|
	▒/	/
	▒\	\
	▒◊	◊
	(	,(or (
	␣ (	,(or (
	'	'
	.'	.'
	␣.'	␣.'
	▒.*	.*
	▒.^	.^
	▒./	./
	▒.\	.\
	▒~=	~=
	+	+
	-	-
	␣+␣	+␣
	␣-␣	-␣
	▒*	,*

Table 1: Comma Insertion Schematics

Since the potential to insert a comma exists whenever the CBN is either '[' or LD, the input is read ahead in this situation with the aim of inserting a comma. More precisely, the input is read ahead when the CBN is either '[' or LD *and* when the parser recognizes a numeral, an imaginary quantity, a string literal, a parenthesized expression, a transpose expression, a conjugate transpose expression, a variable, a matrix, a function call or an array indexing expression. This fact is graphically shown in Figure 10.

6.2 The comma insertion mechanism

The next issue that needs consideration is the actual comma insertion specifics. Where in the unscanned input, and under what circumstances should a comma be inserted? For instance, given the input character sequence [1␣2], the front-end must insert a comma after the 1, effectively converting the character sequence to [1,2]. Notice that the lexeme 1 corresponds to the INTEGER token. On recognizing this token, the parser applies the expr → INTEGER production without consulting a lookahead.[9] Since the CBN is '[' when this reduction occurs, the front-end reads ahead, converting the yet to be scanned character sequence ␣2] to ,2]. In this way, it is ensured that the comma token (',') will be the next token that the scanner would return to the parser, rather than the INTEGER token. Table 1 summarizes all the cases that need to be considered for comma insertion. For instance, the first line of this table states that the front-end converts [a␣,b] to [a,b]. Similar remarks apply to the next thirteen lines in the table. To recapitulate, ▒ denotes zero or more horizontal spaces. While the ◊ symbol in Table 1 indicates a newline, carriage return, form-feed or percent character, the ␣t symbol represents a single horizontal space character. The "Next Lexeme Prefix" column in this table refers to the initial part of the next lexeme that the lexical analyzer would scan, *before* and *after* comma insertion; while the "Before" sub-column denotes this initial part before comma insertion, the "After" sub-column represents the initial part after comma insertion. Observe that the ␣(lexeme prefix gets converted to (or, (depending on whether the current bracket nesting is LD or not respectively. That is, while MATLAB regards [x␣ (2)] as being [x,(2)] (and not [x(2)]), it treats [x␣ (2)]=3 as an assignment to [x(2)]. Furthermore, the (lexeme

[9] In fact, the grammar has been written in such a way so that a lookahead is not consulted when any of the productions shown in Figure 10 are applied.

prefix remains as (or gets converted to , (depending on whether the last processed token is an IDENTIFIER token or not respectively. That is, while `[x(2)]` remains unchanged, `[1(2)]` gets converted to `[1,(2)]`. The last line in Table 1 denotes the comma insertion action for all other lexemes whose prefixes do not match any of the preceding rows; in this case, a comma is inserted before the first non-horizontal space character.

A special exception to the rules documented in Table 1 occurs when the last processed token is either an INTEGER or a DOUBLE and when the first character in the unscanned input is alphabetic. 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 semi-colon.`" after matching the lexeme 1. 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. 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.

7 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 basic flavors. The binary construction $\alpha:\beta$ describes the row vector

$$(\alpha,\alpha+1,\alpha+3,\ldots,\beta)$$

and the empty matrix otherwise. The ternary construction $\alpha:\sigma:\beta$ describes the row vector

$$\left(\alpha,\alpha+\sigma,\alpha+2\sigma,\ldots,\alpha+\left\lfloor\frac{\beta-\alpha}{\sigma}\right\rfloor\sigma\right)$$

if $\alpha\le\beta\wedge\sigma>0$, or $\alpha<\beta\wedge\sigma<0$, and the empty matrix otherwise. 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.

```
colon_expr      : expr ':' expr
                | colon_expr ':' expr
                ;
```

Figure 11: Productions for colon expr

The colon operator (':') is left associative. Thus, constructions such as `1:2:2:2` and `1:2:2:2:2` correspond to `(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. 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 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.

To simplify the processing of colon expressions by subsequent compiler passes, the MATCH parser always produces a full ternary tree as the AST of a colon expression. In other words, the parser retains colon expressions of the form $\alpha:\sigma:\beta$ and converts colon expressions of the form $\alpha:\beta$ to $\alpha:1:\beta$. It should be stated here that 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. An example of a colon atom is in the input `a(:)`.

The grammar rules `colon_expr` → `expr ':' expr` and `colon_expr` → `colon_expr ':' expr` shown in Figure 11 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.

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.

Consider the conditional statement. It comprises 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 non-terminal delimited_input in Figure 12.

```
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
                 ;
```

Figure 12: Productions for if.command

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

Figure 13: Productions for opt else

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. It should be noted that the expressions associated with the if and elseif parts 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" in Figure 12 generates two shift-reduce conflicts. These two conflicts occur when the right-hand side of this production has been seen until the non-terminal expr and when 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[10] so that the '+' or '-' tokens are unary operators in an expression that finally reduces to the non-terminal 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.

In a similar manner, the grammar rule "opt_else → ELSEIF expr delimited_input opt_else" 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---that is, a shift action---is the desired way in which these conflicts should be resolved. It is probably worthwhile to note here that the shift-reduce conflict problem posed by MATLAB's conditional and control statements is quite different from the usual "dangling-else" problem exhibited by similar constructs in other programming languages such as Pascal [4].

[10] The empty string is denoted by ε.

9 Summary

In this paper, we presented the design and implementation of a front-end for the MATLAB language, apart from discussing certain interesting context-sensitive syntactic issues arising from the language as well as their solutions. The MATCH compiler front-end has been implemented and tested on a variety of MATLAB programs. It is being used to compile code for embedded processors, DSPs and FPGAs [5]. The language recognized is a proper subset of MATLAB. The principal parts of the grammar and lexical specification were mentioned and explained in some depth. In particular, we justified why the parser supports a limited form of MATLAB's command-form function invocation syntax, flagging syntactic ambiguities to the programmer whenever they are detected. We also showed how commas were inserted among matrix elements so that the only delimiters visible to the parser were the comma, semicolon and LINE tokens. The dual role played by the single quote character and the syntactic issues that it gave rise to were explained. Colon expressions and their grammar rules were also briefly described. Finally, MATLAB's assignment statements and control constructs were discussed along with their respective grammar rules.

We believe that the usefulness of this work lies in aiding future front-end implementations for the MATLAB language, besides pointing out possible areas where the language may be modified or augmented so as to make it more compiler-friendly, without sacrificing its user-friendliness.

10 References

[1] http://www.ece.nwu.edu/cpdc/Match/Match. html, *The MATCH Project Home Page.*

[2] http://www.mathworks.com/, The MathWorks: Developers of MATLAB, Simulink and Stateflow for Technical Computing.

[3] http://www.che.wisc.edu/octave/, *The Octave Home Page.*

[4] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools.* Computer Science Series. Addison-Wesley PublishingCompany, Inc., Redwood City, CA 94065, USA, 1988.

[5] P. Banerjee, U. N. Shenoy, A. Choudhary, S. Hauck, C. Bachmann, M. Chang, M. Haldar, P. G. Joisha, A. Jones, A. Kanhere, A. Nayak, S. Periyacheri, and M. Walkden. "A MATLAB Compiler for Configurable Computing Systems". *Technical Report CPDC-- TR--9906-013*, Center for Parallel and Distributed Computing, Department of Electrical and Computer Engineering, Northwestern University, Evanston, IL 60208-3118, USA, Sept. 1999.

[6] A. M. Bauer and H. J. Saal. "Does APL really need run-time checking?" *Software-Practice and Experience*, 4:129-138, 1974.

[7] T. Budd. *An APL Compiler.* Springer-Verlag New York, Inc., New York City, NY 10010, USA, 1988.

[8] W.-M. Ching. "Program Analysis and Code Generation in an APL/370 compiler." *IBM Journal of Research and Development*, 30(6):594-602, Nov. 1986.

[9] S. C. Johnson. "Yacc: Yet Another Compiler Compiler." *Technical Report 32*, Bell Laboratories, Murray Hill, NJ 07974-0636, USA, July 1975.

[10] P. G. Joisha, A. Kanhere, P. Banerjee, U. N. Shenoy, and A. Choudhary. "The Design and Implementation of a Parser and Scanner for the MATLAB Language in the MATCH Compiler." *Technical Report CPDC– TR–9906–017*, Center for Parallel and Distributed Computing, Department of Electrical and Computer Engineering, Northwestern University, Evanston, IL 60208-3118, USA, Sept. 1999.

[11] M. E. Lesk. "Lex: A Lexical Analyzer Generator." *Technical Report 39*, Bell Laboratories, Murray Hill, NJ 07974-0636, USA, Oct. 1975.

[12] The MathWorks, Inc., 24 Prime Park Way, Natick, MA 01760-1500, USA. Using MATLAB-The Language of Technical Computing, Jan. 1997. *Using MATLAB* (Version 5).

[13] R. P. Polivka and S. Pakin. *APL: The Language and Its Usage.* Automatic Computation Series. Prentice-Hall, Inc., Englewood Cliffs, NJ 07458, USA, 1975.

[14] M. J. Quinn, A. Malishevsky, N. Seelam, and Y. Zhao. "Preliminary Results from a Parallel MATLAB Compiler." In *12th International Parallel Processing Symposium*, pages 81-87, Orlando, FL, USA, Apr. 1998.

[15] L. A. D. Rose. *Compiler Techniques for MATLAB Programs.* Ph.D. dissertation, University of Illinois at Urbana-Champaign, Department of Computer Science, May 1996.

[16] G. O. Strawn. "Does APL Really Need Run-time Parsing?" *Software-Practice and Experience*, 7:193-200, 1977.

[17] Z. Weiss and H. J. Saal. "Compile Time Syntax Analysis of APL Programs". *APL Quote Quad* Vol. 12 No 1 (APL81 Proceedings), pages 313-320, San Francisco, CA, USA, Oct. 1981.