MatLang2C

Design and Implementation

April 10, 2016

Contents

1 Specification

Design and implement a MatLang to C translator that does syntactic and semantic checks on the given MatLang source code and and translates it to the corresponding C program. For further details about MatLang language, see the compiler description.

2 Design

The main modules of the translator are as follows:

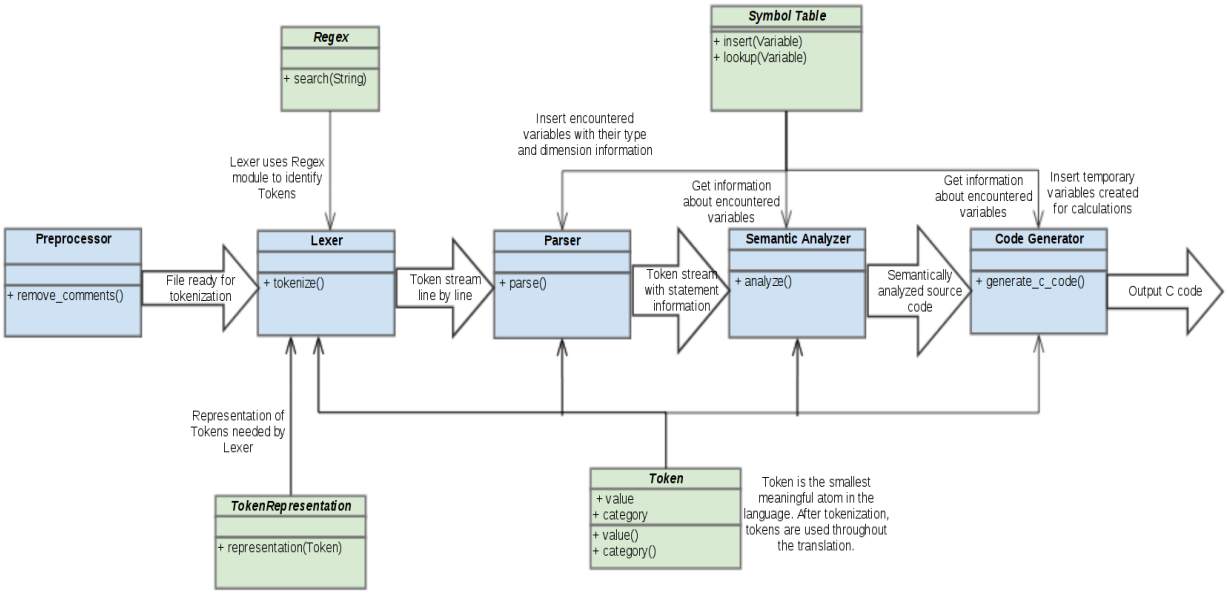
1. Preprocessor
2. Lexer
3. Parser
4. Semantic analyzer
5. Code generator

There are also two kinds of smaller modules:

1. Helper modules
   (a) Regex
2. Object modules
   (a) Token
   (b) Symbol table

Before explaining the design and functionality of each module, let us see the overall design and the interaction between modules in Figure 1.

Figure 1: Translator Pipeline



2.1 Preprocessor

Preprocessor is the module that processes the MatLang source code for the first time. It makes the file ready for tokenization.

2.2 Lexer

Lexer takes the output code from preprocessor and tokenizes the file line by line. Tokenization of a single line results in a list of tokens.

2.3 Parser

Parser takes the output of lexer which is a list of tokens and does syntactic checks and derivations on the token list. If the given token list is syntactically correct, parser figures out its statement type and returns the statement category of the token list.

2.4 Semantic Analyzer

Semantic analyzer takes the whole of source file after the whole code is syntactically checked. It does semantic checks on the given source file such as if a used variable is declared before.

2.5 Code Generator

Code generator takes the output of semantic analyzer which is the whole source file with each line's statement type determined by parser. Code generator reads the source file line by line and produces the equivalent C statements. Each C statement is written to the output C file.

2.6 Symbol Table

Symbol table is a

$$\langle Variable\_name \rangle \rightarrow \langle Variable \rangle$$

map. A *Variable* is a class that has information about the *name*, *type* and *dimension* of a variable. Whenever a new variable declaration is encountered in the program, it is put into the symbol table. Afterwards, getting type and dimension information about that variable can be done via the symbol table.

2.7 Token

Token is the smallest atom in the language. Tokens are combined to generate statements. Example tokens are

1. For keyword
2. Open parenthesis
3. Addition operator

2.8 Regex

Regex module is a wrapper class around POSIX C regex library. It provides a search method for determining if a given string matches the regular expression stored in a Regex object.

3 Implementation

3.1 Algorithms

Main algorithms used by different modules are explained below.

3.1.1 Tokenize

Tokenizer algorithm uses Regex module to match strings to token representations. Details of such matches are not shown in the pseudocode.

```
Iterator it;
String compound;
List token_vec;
while it not at the end of line do
    if line[it] is Digit or Integer or Dot then
        compound = compound + line[it];
    end
    else
        if compound matches any token representation then
            Token t(compound, category);
            token_vec.push_back(t);
            compound = "";
        end
        else if compound is empty then
            continue;
        end
        if line[it] matches any token representation then
            Token t(line[it], category);
            token_vec.push_back(t);
        end
        else if line[it] is Whitespace then
            continue;
        end
        if compound didn't match any representation then
            Error;
        end
        if line[it] didn't match any represnation then
            Error;
        end
    end
end
return token_vec;
```

3.1.2 Parse

In order to understand how parsing algorithm works, we first need to understand some implementation details of the Parser module. Parser module has a map called *prod_rules* which holds the production rule for each type of statement. For example, the following is an entry in the *prod_rules*:

$$\langle ExpressionAssignment \rangle \rightarrow \langle Identifier \rangle = \langle Expression \rangle$$

Left hand side of a production rule is the key. Right hand side of a production rule is a list of Tokens representing the derivation rule.

Another detail that needs attention is that Parser partitions the Tokens to terminals and nonterminals. Terminal tokens do not need any further derivation. They appear in the same form in the tokenized source code. However, nonterminals need further derivation.

Considering these details, now we can introduce our algorithms.

1. **Parse Algorithm**

```
result ← Derivation();
if result.index not equal to token_list.size then
    Error;
end
if result.category is Declaration Statement then
    Update the symbol table;
end
return result.category;
```

2. **Derivation Algorithm**

```
StatementType result;
Integer index ← 0;
for rule in production_rules do
    for token_category in rule.token_list do
        if token_category is terminal then
            if token_vec[index] == token_category then
                index ← index + 1;
            end
            else
                Error;
            end
        end
        else if token_category is nonterminal then
            if token_category is Expression then
                index ← parse_expression(index);
            end
            else if token_category is InitializerList then
                index ← parse_init_list(index);
            end
            else
                Error;
            end
        end
        else
            Error;
        end
    end
    if No Errors then
        result ← rule.name;
        return result;
    end
end
Error;
```

Expression and initializer list parsing algorithms are not given here because they are not part of the main algorithms translators depends upon. They can be found in Parser module source code.

3.1.3 Code Generation

As the source code progresses in the translator pipeline, more and more information about it is known to the system. When the code comes to the code generation phase, every information needed to convert it to a C program is known. Since the parsing algorithm returns the statement type of a given token list, code generation algorithm knows the statement type of each statement that comes to it. Here is the algorithm for code generation:

```
for Statement in source file do
    switch Statement.type do
        case ScalarDeclaration do
            WriteScalarDeclaration(Statement);
        end
        case MatrixDeclaration do
            WriteMatrixDeclarationStatement;
        end
        case SingleForStatement do
            WriteSingleForStatement(Statement);
        end
        ⋮
        otherwise do
            Error;
        end
    end
end
```

Each function writes its corresponding C statement to the output C file. Code generator has a special function for expression evaluation called *convert_to_c_expr* whose details are discussed in 3.2.4 Conversion Functions.

3.2 Expression Evaluation

Correct expression evaluation, syntax and semantic checks on expressions are major parts of the project. Thus, we give expression evaluation its own section where we will be discussing how operator precedences are found, how this information is maintained across modules and how it is used in finding semantic errors in expressions.

3.2.1 Expression Grammar

Following grammar is used in expression parsing.

$$
\begin{aligned}
\langle expression \rangle \rightarrow & \langle term \rangle + \langle expression \rangle \mid \\
& \langle term \rangle - \langle expression \rangle \mid \\
& \langle term \rangle \\
\langle term \rangle \rightarrow & \langle factor \rangle * \langle term \rangle \mid \\
& \langle factor \rangle \\
\langle factor \rangle \rightarrow & \langle integer \rangle \mid \\
& \langle real \rangle \mid \\
& \langle identifier \rangle [\langle expression \rangle, \langle expression \rangle] \mid \\
& \langle identifier \rangle [\langle expression \rangle] \mid \\
& \langle identifier \rangle \mid \\
& (\langle expression \rangle) \mid \\
& tr(\langle expression \rangle) \mid \\
& sqrt(\langle expression \rangle) \mid \\
& choose(\langle expression \rangle, \langle expression \rangle, \langle expression \rangle, \langle expression \rangle)
\end{aligned}
$$

Additionally, there is an extra logic for dealing with expression of the form

$$x - y + z$$

where it is transformed into the following expression

$$x + (0 - y) + z$$

3.2.2 Finding Operator Precedence

Finding operator precedence is done by transforming the given expressions to postfix notation with the help of a stack.

After implementing the above grammar with several mutually recursive functions, transforming the given expression to postfix notation is as simple as adding several lines of *stack.push()* at the correct places. For implementation details on where to add these calls, refer to the *parse_expression* and *parse_term* function in the Parser module.

As previously said, adding the extra logic for subtraction operations is again done by pushing the correct tokens to the stack at correct places.

3.2.3 Maintaining Operator Precedence Information

Postfix expression found in the previous step is saved in the expression stack. After the expression is evaluated, the original expression in token list is replaced with the postfix expression stored in the stack. This way, operator precedence is preserved across modules. Later, modules that need operator precedence information can directly use the expression. Modules that need to convert to infix notation can use expression conversion functions to do so.

3.2.4 Conversion Functions

Code generator needs the expression in infix notation to write it to the output C program. For this purpose, it has a *convert_to_c_expr()* function that converts a given postfix expression to its C counterpart in infix notation.

convert_to_c_expr() in detail This is the function used by code generator to convert expression to infix notation. A high-level view of the algorithm not explaining all of the implementation details is the following:

```
Stack expr_stack;
Iterator it;
while it not at the end of the expression do
    category ← expression[it].category;
    if category == BinaryOperator then
        left_op ← expr_stack.pop();
        right_op ← expr_stack.pop();
        if left_op is scalar AND right_op is scalar then
            Token t(left_op, operator, right_op);
            expr_stack.push(t);
        end
        else if left_op is scalar AND right_op is matrix then
            if scalar is 0 AND category == SubtractionOperator then
                NegateMatrix();
            end
            else
                Confirm operation is multiplication;
                WriteScalarMatrixMult();
            end
        end
        else if right_op is matrix AND left_op is scalar then
            Confirm operation is multiplication;
            WriteScalarMatrixMult();
        end
        else
            Confirm dimensions;
            WriteMatrixMatrixMult();
        end
    end
    else if category == CloseParenthesis then
        read a function call from expr_stack;
        write the function call to C program;
        Compound all tokens to a single token t;
        expr_stack.push(t);
    end
    else if category == CloseSquareBrackets then
        read a subscript operation from expr_stack;
        Compound all tokens to a single token t;
        expr_stack.push(t);
    end
    else
        expr_stack.push(expression[it]);
    end
end
return expr_stack.pop();
```

3.2.5 Using Operator Precedence Information

Operator precedence information coupled with the information obtained from the symbol table is enough to make semantic checks on expressions. Symbol table is used in

1. Getting type and/or dimension information about a stored variable.
2. Storing type and dimension information of temporary variables that will need to be accessed in the future.

Since we also know the order of evaluation, we can do type and dimension checking on even the most complicated expressions by following this bottom-top approach.

3.3 Design Decisions and Assumptions

This subsection describes the several design decisions or assumptions about the project.

3.3.1 Token Categories

In the current version of the translator, there are more than a couple dozens of token categories, all under the same enumeration. Although this allows for flexible token storage, it shadows the fact that a token category representing a statement type is very different than a token representing a terminal actually present in the MatLang source code. In the later versions of the translator, dividing the token categories into several enumerations may be more beneficial in terms of type safety, understanding the program logic and debugging.

3.3.2 Semantic Check in Code Generation

In the current version of the translator, the semantic check for expression evaluation is done at code generation phase. For example, if there is matrix dimension mismatch in an expression, the phase it can be detected is code generation. From the project design perspective, this looks a bit out of place and I must admit that I couldn't find a way of dividing expression evaluation and expression semantic checking into two modules.

3.3.3 Handling of Subtraction Operator

In the first draft of the expression handling grammar,

$$\begin{aligned}\langle expression \rangle \rightarrow & \langle term \rangle + \langle expression \rangle \mid \\ & \langle term \rangle - \langle expression \rangle \mid \\ & \langle term \rangle\end{aligned}$$

the expression

$$i = 1 - 2 + 3 - 4$$

results in the treatment

$$i = 1 - (2 + 3 - 4)$$

which is undesired. To fix this problem, whenever a subtraction operator is found, it is converted to a addition operation by adding a zero to the start of the expression. The previous expression becomes

$$i = 1 + (0 - 2) + 3 + (0 - 4)$$

This fixes the problem, but introduces another issue. For matrix subtraction operations, putting a scalar 0 results in matrix dimension mismatch. For example, for matrix variables x and y, the operation

$$x = x - y$$

is treated as

$$x = x + (0 - y)$$

Since 0 is a scalar and y is a matrix, this produces an error. By assumption, this operation is **allowed**. For a matrix variable i, the operation

$$i = 0 - i$$

is valid and results in a matrix whose all elements are negated.

3.3.4 Matrix or Vector to Scalar Conversion

All matrices or vectors that have only one element in them are considered as scalars. Implications of this assumption are listed but not limited to

1. Vectors or matrices with only one element **cannot** be list initialized. A vector or matrix with a single element is automatically considered a scalar and thus must be initialized with a single expression without any curly braces around it.
2. In a MatLang program, multiplication of a matrix or vector with only one element with any other matrix or vector is valid.
3. If an operation in MatLang produces a 1x1 vector or matrix, the result is considered as a scalar and all operations that are allowed on scalars are allowed on these literals as well.

3.3.5 No Integer or Double Identification

MatLang's scalar type is considered as a double value. Thus, if a scalar value corresponds to an integer is not checked at any part of the program. This has the following implication

1. Subscripting a vector or matrix with values other than integers is **possible**. Subscipts are automatically type cast to integer values in the output C program. Thus, $A[2.5]$ is the same as $A[2]$.

3.3.6 Preprocessor Output is Automatically Deleted

In the current version of the translator, there is not an option to get the preprocessor output. The only functionaly preprocessor has is deleting comments. In the later versions of the translator, getting preprocessor output may be possible.

4 Testing

Five test cases in the project description produce the expected results. Additional test cases can be found in *tests* directory in the project root directory.

4.1 Running Tests

A python script *run_tests.py* is prepared for testing purposes. In order to use the script on the newly created test files, the following steps are needed:

1. Move your test case to the *tests* directory in the project root directory. Your test MatLang source code must end with .mat extension.
2. Expected output of the test case should be put to the corresponding file with the same name but with a *test* extension. For example, if your test case is *ex3.mat*, then the result should be in *ex3.test*.

3. Run the *run_tests.py* script from the project root directory.

4. For each test case, run_tests script produces an output wher you can see the expected output and the actual ouput for the test case.

5 Acknowledgements

1. **TutorialsPoint Compiler Design**
   Information about the overall design of a compiler, which modules are necessary to implement a compiler, etc.

2. **GNU C Regular Expressions**
   Information about how to use POSIX C Regular Expression library.

3. **Wikipedia - Lexical Analysis**

4. **Wikipedia - Parsing**