2024 IEEE International Conference on Teaching, Assessment and Learning for Engineering (TALE) | 979-8-3503-7623-4/24/$31.00 ©2024 IEEE | DOI: 10.1109/TALE62452.2024.10834289

Learning to Design Novel Programming Languages using CodeBlock Syntax Checker

Spencer Reed
Department of Computer Science
University of Idaho, USA
reed7385@vandals.uidaho.edu

Hasan M. Jamil ✉
Department of Computer Science
University of Idaho, USA
jamil@uidaho.edu

***Abstract*—In a programmer's pursuit of using or creating new programming languages, finding errors in the syntax of code can present many issues. Languages with little to no documentation and incomprehensible exception handling and reports are frustrating to work with and can create confusion when trying to locate where in the code the program has faulted. In this paper we present *CodeBlock*, a parser generator and syntax checker for arbitrary programming languages. CodeBlock is a block based grammar builder for any programming language that constructs a parsing expression grammar for the language based on user built expressions. This grammar can then be used within the CodeBlock website or in the CodeBlock Node.JS application to test the syntax of either written code, or files containing code in the language, reporting comprehensible error messages if errors in syntax are found. Our eventual goal is to incorporate CodeBlock into a compiler design tutoring system, called *CompiTS*, in which it will play a central role in teaching students how to design new programming languages and test the effectiveness of the new language using rapid prototyping and a translational approach to implementation. This is an emerging research, and in this paper, we only focus on the syntax checking component of the CompiTS system.**

***Index Terms*—Syntax Checker, Compiler Design, Tutoring System, Language Design.**

I. Introduction

Syntax checking or parsing is fundamental to all language based computer applications [24], during system design or when instructions are supplied for machine interpretation and execution. Students build special purpose syntax checkers and analyzers at a considerable cost and effort. Computer science students also practice new language designs in courses such as compiler construction and programming languages design. The main concept they must master in these exercises is the grammar and syntax rules. Currently, they learn and practice these concepts by designing parsers and implementing them in a case by case basis. They must redesign the parsers in the event of errors or needed modifications at a considerable time cost, the time they could use to learn more useful things.

A. Background

Error handling in programming languages is the undertaking of giving feedback to programmers when errors in their code have been identified. Programming languages such as Python and Java feature built in error handling and reporting functions that check a programmer's code for any exceptions raised that would prevent the program from executing. For instance, when a syntax error is found in Python code, its parser reports to the programmer the line where the error was found and the token in this line that created the exception.

While a handful of popular programming languages contain exception handling features such as these, more often than not it is up to the programmer to locate where exactly their code has thrown the error. This is often tedious, requiring the programmer to work through their program line by line until they find where the issue has occurred.

To alleviate programmers from painstakingly searching through their code, many sites have been created to check the syntax of various programming languages. For example, various syntax checkers can be found online for challenging languages such as PHP and SQL. The downsides of these syntax checkers are that they only analyze one specific language, and they do not exist for every language. New programming languages or languages with less recognition may not have one of these tools built for them. To address these issues we present in this paper CodeBlock, a parsing expression grammar generator and syntax checker for arbitrary programming languages. Our CodeBlock system explores an alternative method of checking the syntax of any programming language by first allowing the user constructing a parsing expression grammar (PEG) for the language they would like to check the syntax of. Our system contrasts other syntax checkers found online by allowing users to build their language dynamically using a simplified, block-based format to assign the grammar rules. This allows programmers of any skill level to easily create their grammar from the ground up.

B. Related Works

Compiler design is a foundational course in all computer science curricula [11]. Yet, only a handful tutoring systems could be found [7, 8, 23]. One of the major ingredients of compiler design is syntax parser and analyzer. As Ali and Smith [3] suggests, implementation of parsers makes a difference in how the machines interpret them. Therefore, learning to implement the right grammar and parsing them accurately matters [9].

Not only novice computer scientists learn how to implement language parsers [1, 10], advanced students also implement parsers and syntax checkers for their research [4, 15, 20] as well. New languages for advanced applications are also devel-

oped routinely, yet tools to conceive new languages through syntax modeling and testing is difficult to find. In this paper, we develop a new online syntax design and checking system, called *CodeBlock*, with a goal to integrate it with a compiler tutoring system along the line of ITT [7, 8]. CodeBlock is able to help design arbitrary language syntax grammar and learn the rules, and check the syntax for admissibility under the stored rules of an input program. CodeBlock allows online submission of syntax grammars, memorizing them and downloading developed grammar to be integrated with user codes, all through an API as well as from the cloud applicatio.

C. Presentation Plan

The rest of this paper is organized as follows. In Sec II, we present an analysis of parsing expression grammars and how they are structured within CodeBlock. In Sec III, we outline the method of generating these grammars using a block-based format. Sec IV gives a more in depth analysis of how this grammar is translated from simple blocks to parser generation strings. In Sec V, we present the ASP.NET Core web API developed for database use in the system. In Sec VI, the methods of receiving and uploading grammars constructed using CodeBlock is presented for use in subsequent grammar construction and syntax checking sections. Sec VII presents an overview of a Node.JS application developed in order to check the syntax of code aligned to defined CodeBlock grammars without the need of accessing the CodeBlock site every time. Finally, we briefly outline the design of the language design tutoring system in which CodeBlock is a component in Sec VIII before we conclude in Sec IX.

II. Parsing Expression Grammars for use in CodeBlock

Parsing expression grammars, or PEGs, are formal grammar descriptions of the rules used for recognizing strings in some programming language. These descriptions contain parsing expressions, or patterns that a given string can either match, or not match, with no in-between [21]. PEGs contain designated starting expressions as well, where expressions that reference each other can start from. The grammar will accept any strings that start with these expressions and match the rules following it. PEGS are similar to widely used context-free grammars (CFGs), however PEGs do not allow for any ambiguity in the strings passed to it.

An example of a parsing expression grammar is found in the following example:

- exp → sum ;
- sum → addop (("+" | "-") addop)* ;
- addop → mulop (("*" | "/") mulop)* ;
- mulop → value ("^" mulop)? ;
- value → [0-9]+ | "(" exp ")" ;

The string 4 + 5 / 3 is valid in this grammar. The parse tree for this string would be as shown in Fig 1.

These grammars do not allow for more than one parse tree to be created for them, and as such parsing expression grammar do not allow for ambiguity in design. PEGs also function

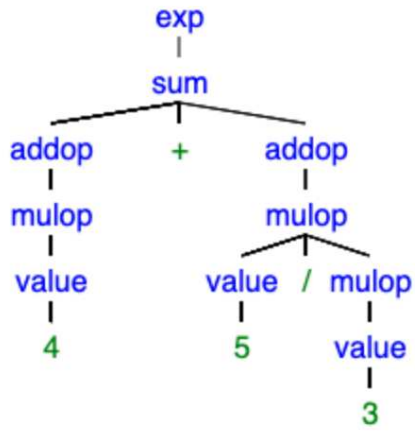

Fig. 1. Parse tree for "4 + 5 / 3" in a parsing expression grammar

based on "ordered choice" parsing. This means that the parser will check the first expression in a grammar rule first, and if the string it is parsing does not pass it moves onto the next, accepting the first expression found that works for the string given. This allows for a great deal of control when structuring a grammar, with the programmer easily able to change what will be checked first when parsing.

In this section, we present the structure of the parsing expression grammar constructed by the CodeBlock system. To follow up, we discuss how this grammar is passed into and parsed using PeggyJS, a Javascript API tool used to parse PEGs on runtime.

A. CodeBlock Grammar

A CodeBlock grammar is constructed in three separate chunks.

1) Keywords
2) Initial Keywords
3) Grammar Rules

The keywords section of a CodeBlock grammar contains the expression tokens used in the grammar rules. These tokens each define a rule in the grammar, and are able to reference each other within their rules in the file created when downloading a grammar from CodeBlock as follows.

Keywords:
start token1 token2 token3

The initial keywords section of a CodeBlock grammar contains the tokens for the starting expressions of the grammar. These tokens define the expressions that the grammar can start with when parsing a string.

Initial Keywords:
start

The start keyword in the above example would be the name of the expression used to start the parsing process of the grammar.

Lastly, the grammar rules section of a CodeBlock grammar contains all of the names and definitions for each expression

within the grammar, and what each expression allows according to the grammar. These rules are constructed in a similar manner to Backus-Naur Form, using #= to separate the left and right hand sides rather than ::=. The rules section is structured as in the following example:

```
Rules:
start #= token1 / token2
token1 #= [a-zA-Z]+ token2
token2 #= token3 'b' token3
token3 #= [0-9]
```

In this grammar, a string could either start with the expression defined by token1 or the expression defined by token1 through *start*'s definition. The token *token1* can be 1 or more letters followed by the token *token2*, which can be any number as defined in the token *token3*, the letter b, then any number again. The user must construct this grammar by defining its keywords and rules themselves initially in the manner described in Sec III.

These three sections are separated by the CodeBlock system when the user defines them, or when a grammar file is fed into the system. A full grammar file in this format would combine the previous sections and be structured as in the example below.

```
Keywords:
start token1 token2 token3
Initial Keywords:
start
Rules:
start #= token1 / token2
token1 #= [a-zA-Z]+ token2
token2 #= token3 'b' token3
token3 #= [0-9]
```

This file is then traversed when the user desires to check the syntax of code, reading each section and storing their values into different arrays to be used further. The way this is done is described in further detail in Sec VI.

B. Peggy Utilization

When a grammar in the format defined in Sec II-A is read into the CodeBlock website, the *Rules* section is manipulated into a string readable by the PeggyJS API [16]. This API allows for generating PEGs while a program is running by feeding it a string containing the tokens and rules of a language. It then allows a user to parse strings in the grammar, returning error messages when a syntax error is found. Strings used to generate PEGs in this manner must start with the initial keywords contained in parentheses, followed by a newline, and followed further by definitions for each token. Each definition must also be separated by a newline character. Ana string creating a grammar using this API can be found in the following example, with newline characters added in for better visibility of the string.

```
"start = (programStart)
programStart = type' 'funcStart' 'funcBody' "}'
type = 'int' / 'void'
varDecl = ID' "=" '[0-9]+
ID = [a-zA-Z]+
funcStart = [a-zA-Z]+'(){'
funcBody = "varDecl';'"
```

This example showcases a string capable of generating a PEG for a small language using the PeggyJS API able to define an initial function with a type of int or void followed by an id for the function as well as the interior of the function, allowing for a variable definition. The string "void updateVar() var = 91; " is a valid string in this grammar, with the syntax tree in Fig 2.

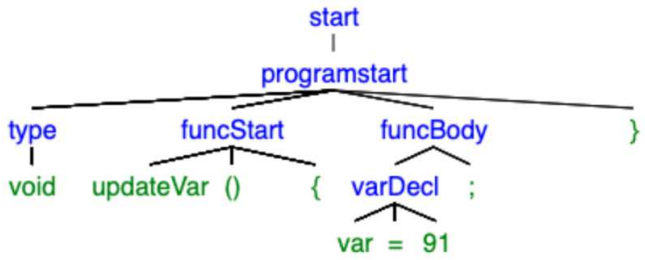

Fig. 2. Parse Tree for void updateVar() var = 91;

In the CodeBlock system, the Rules section of the grammar file is traversed and each expression found is added to a string in this format. This string is then fed into the API's .generate() function, creating a parser for the grammar. Strings in the language are then able to be used with this parser. The CodeBlock system then returns "Correct Syntax!" to the user if no errors are found, or customized error messages to the user displaying the location of a syntax error if one is discovered by the parser.

III. Block-Based Grammar Generation

To create a program for building functional grammars usable by programmers of any level, a method to construct the grammars in an easily digestible manner had to be considered. This section describes the method of generating PEGs within CodeBlock. To create the grammars, a block-based drag and drop method has been developed for use in the CodeBlock website. This has been done in order for ease of use, regardless of skill in coding languages.

Block-based programming works by dragging predefined 'blocks' within a designated coding area, as opposed to typing code directly. An example of this coding method is the *Scratch* language. Scratch works by dragging jigsaw style blocks with code baked into them in its coding area. These blocks slot in place within and next to each other, building code simply by reading these blocks in order.

CodeBlock employs a similar method to Scratch for the purpose of generating its grammar to use within the system. The CodeBlock website features a coding area used to define

individual expression rules. Users of the website are able to drag keyword blocks, defined initially by the user, into this area. To create these blocks, the user writes each keyword they would like to use in the grammar within an input box labeled ”Input Keywords”, separated by spaces or commas. This generates unique draggable token blocks on the left-hand side of the website UI for each keyword input using the HTML Drag and Drop API script.

Once these keywords have been input, the corresponding blocks can be drug into the grammar construction area on the site. A definition block such as ”is defined by” can then be dragged to this area as well. Continuing from this set-up, additional blocks can be added to this area, outlining the right-hand side of the expression rules. The blocks ”any letter”, ”an integer value” and ”the character _” can be found within the list of blocks and once drug into the rule definition, the regular expressions corresponding to them used in the system are added to the parsing expression.

An example of this block-based rule building can be found in the Fig 3, showing the definition of ”varDecl” from the grammar in Sec II-B.

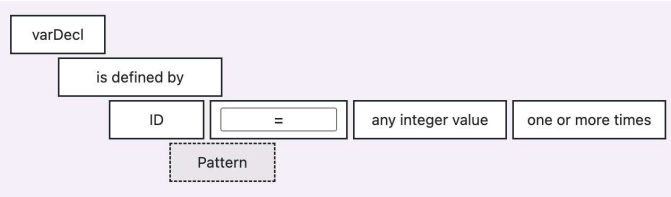

Fig. 3. Expression definition for a ”varDecl” rule

To define more than one rule that a single expression can move through, there exists an ”or defined by” block in the CodeBlock grammar construction area that adds a ’/’ character to the expression string. This character allows for ”or” statements in Peggy’s grammar generator, producing a separate path for an expression to follow when parsing a string.

After an expression has been fully defined, it is submitted and added to the site’s memory. It is also pushed into the CodeBlock web API’s database memory separately, defined in Sec V. A button on the site labeled ”Clear Blocks” can be pressed to erase the blocks in the construction area, allowing for new expressions to be built and added to memory.

To ensure that the PEG reads strings starting with the correct expression, a separate input bar was added to the UI labeled ”Initial Keywords”. Listing keywords here in the same way as the other keyword input bar and submitting the keywords places the initial expression names into a separate Javascript string array, read when creating the grammar string for passing to the generate function.

IV. Block to String Literal Translation

To build the grammar string of words and regular expressions passed into the Peggy API to generate a parser, the arrays of strings mentioned are built and updated dynamically whenever users add keywords, initial keywords or rules using the blocks on the site. When initiating a syntax check for the grammar, the Initial Keywords and Rules arrays are read into different parts of the overall grammar string.

A. Starting Expression Strings

The starting expression is created first by reading the Initial Keywords array. In the expression generated based on the example shown in Sec II-A, the starting expression string generated would be as follows.

”start = (programStart)”

In this string, the Peggy parser knows to start with the expression named ”programStart”

Another example of a starting expression generated when multiple initial keywords are found instead of a singular starting expression would be structured as follows.

”start = (programStart / startExp)”

This string teaches the parser that strings can either start following the ”programStart” or ”startExp” expressions and their rules.

As mentioned previously, starting keywords can be updated at any time, which will update the string as well allowing for more starting expressions dynamically. Regardless of how many expression names are held in the array, this initial expression will follow the same format, with more ’/’ characters separating additional expression names.

B. Expression Rules

When constructing expressions as in Fig 1, each block is read separately into a temporary string array when the rule is submit. For the example in the Fig 1, the array would have the following contents:

1) tempRuleArray[0] = ”varDecl ”
2) tempRuleArray[1] = ”= ”
3) tempRuleArray[2] = ”ID ”
4) tempRuleArray[3] = ” ’=’ ”
5) tempRuleArray[4] = ”[0-9]”
6) tempRuleArray[5] = ”+”

These are then combined into the ruleGrammar array in one index location with space characters being defined simply as ’ ’ for readability in the parser generator. The final outcome for this singular expression is what is shown in Sec II-B:

ruleGrammar[3] = ”varDecl = ID’ ”=” ’[0-9]+”

All of the above strings in the ruleGrammar are combined once a syntax check has been initiated into a singular string, then used with the generate function in order to create a functional parser using the Peggy API.

The ’ ’ chunks must be added in the string translation in order to read spaces in input. Each character, including spaces read as character literals in the grammar must be encapsulated with single quotation marks, or the character will be interpreted as an expression name. In the previous example, ID is a token, while the character = must be written as it is a terminal. If the = character were not surrounded by single

quotes, it would be read as a non-terminal token, halting the flow of the parser.

With all of this done, the above expression can read the string as a correct input:

”x = 20”

This string will return ”Correct Syntax!” to the user if varDecl is specified as a starting expression as it passes the right-hand side rules of the expression. x is the ID token, = is the ’=’ component, and 20 is the [0-9]+ regular expression.

V. CodeBlock WebAPI

In addition to the features of CodeBlock mentioned in the previous sections, the CodeBlock system also contains a Web API to hold all of the rules created by a user. These rules are added automatically when submitting a rule from the grammar construction area, each with their own unique identification number. This section will describe the back-end activities of the API, as well as the uses of this API within CodeBlock.

A. ASP.NET Core Integration

The CodeBlock Web API is developed using the ASP.NET Core framework for creating and update a database of grammar rules. ASP.NET Core uses a Model-View-Controller pattern in order to create an easily testable separation between a program’s API code and client-side code. With this framework, development of the API was expedited and made much easier to update and add features to. The CodeBlock website hosts a CodeBlockRules model for API rules, as well as a CodeBlockController controller holding the code for accessing and manipulating this model through different aspects of the site itself.

B. Client-Side API Database Manipulation

With the integration of this API within the CodeBlock system, users are able to add, remove and update the API database in a variety of different ways. By doing this, they are able to keep these rules held in the database to view even when the page itself is refreshed.

When users create rules for their grammar in the grammar construction area, an HttpPost method defined in the CodeBlockController is called, placing a string following the structure defined in the ”Rules” portion of Sec III into the database with a unique identification number to be referenced in later methods.

Next, when there are rules within the database, a button on the top-right side of the website can be pressed, bringing up a box featuring both a syntax checker for the grammar, as well as each rule found in the grammar. This screen can be seen in Fig 2.

These rules show the ID for the rule on the left with a ”ID).” format, followed by the expression string, and a ”Delete” button on the right of the rule. To show these rules to the user, an HttpGet method is called when the button to show the rules is initially pressed. This method iterates through the database item by item until no more items are found. When a rule is found, an html row object is created and placed in a table object with columns for the id number, expression string and delete button. This allows for deletion of rules from the database with the delete button to remove the object visually from the table without removing the entirety of the rules from the user’s view.

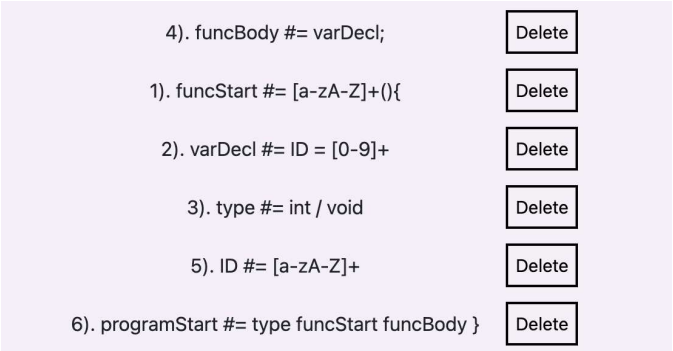

Fig. 4. The rules of a grammar pulled from the webAPI database

The delete button itself calls an HttpDelete method within the API as well. This method finds the rule by the ID number within the database, then removes it from the database. This allows for dynamic updating and removal of rules, without having to worry about duplicate rules or errors when building the expression. Since the rules are in individual rows of the table, each row is assigned a number corresponding to the id of the rule in the database that will be used as input for the delete method.

Additionally, to create the string that the PeggyJS parser is generated from, the HttpGet method is called in a similar way to displaying each expression in the rules visualization portion of the website. To construct the grammar string as in Sec II-B, the API database is traversed and each rule found within is placed into an array of strings. After this, the rule strings are manipulated into Peggy format and appended to the starting expression string separated by ’+’ characters. This string is then fed into the Peggy parser generator and a parser is generated from it to then check for syntax errors.

VI. Downloading and Uploading CodeBlock Grammars

To ensure ease of use across sessions using the CodeBlock system, users are both able to download grammars directly from the website as well as upload them to website. Both of these processes function based on simple .txt files read by the system. This section will delve further into the processes for both of these systems in individual subsections.

A. Grammar Downloading

Once a user has populated their CodeBlock system with Keywords, Initial Keywords and Rules for their grammar in a session, they may want to save their progress for later while needing to shut down their system. While the CodeBlock API shown in Sec V works across page refreshes, it is unfortunately impossible to store data across sessions. To

circumvent this issue, code was developed allowing users to be able to download their grammar in a text file to be uploaded when their system is running the CodeBlock website once more.

When users add items to any of the three key sections of the system, they are stored in individual Javascript string arrays. There exists a keywords array, an initial keywords array and a rules array in the code. To create the file that will be downloaded by the user, once the button is pressed to download the grammar these arrays are read and placed into the body of a New Grammar.txt file. An example of this text file is shown in Fig 3.

```
Keywords:
programStart funcStart funcBody ID type varDecl
Initial:
programStart
Rules:
type #= int / void
funcStart #= [a-zA-Z]+(){
funcBody #= varDecl;
varDecl #= ID = [0-9]+
ID #= [a-zA-Z]+
programStart #= type funcStart funcBody }
```

Fig. 5. A simple downloaded grammar file

The file is structured exactly as in the full grammar example in Sec III, with ”Keywords:” being written first and the keywords array’s strings following on the next line separated by spaces. Next, ”Initial Keywords:” is written on a new line with the initial keywords array’s strings following on another newline separated by spaces as well. Finally, ”Rules:” is written on yet another new line, followed by the strings found in the rules array separated by new lines rather than spaces.

When downloading the grammar, another file named ”cbGrammar.js” is downloaded as well featuring only the expressions of this text file without the ”Rules:” heading for use in a Node JS application described in Sec VII.

B. Grammar Uploading

When a new session of the CodeBlock website is initiated the user may wish to continue updating a grammar or testing one without wanting to create the expressions from the ground up once more. To allow the user to do this, an upload function was added to the CodeBlock website. With this function, the downloaded grammar text file mentioned previously is taken and separated into portions readable and usable by the system for additional use of the grammar on the site.

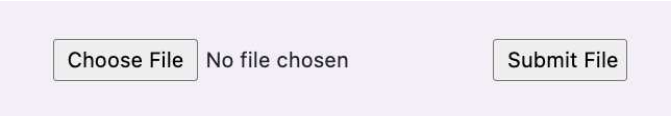

Fig. 6. The Uploading section of the CodeBlock website

This section is found under the ”Input Keywords” input box and can accept any .txt file, but will cancel the file search if ”Keywords:” is not the initial line, to stop non downloaded grammar files from being read.

Once a grammar file has been uploaded it is read section by section. When the system reads the line ”Keywords:” it then takes every string separated by a space and places it into the global keywords string array. With this array, the separated strings are then made into draggable blocks that can start an expression in the grammar construction area. Each of these blocks is a separate HTML paragraph tag appended to the ”All Keywords” section of the site.

Once all of the keywords in the text file have been read into blocks, the program reads the next line of the text file. This should be just the words ”Initial Keywords:”. Once these words are read, the program moves along to the next line which holds all of the individual initial keywords used as starting expressions for the grammar to parse. These keywords are placed into the array holding initial keywords, and then read into the ”Initial Keywords” section of the site for viewing by the user.

Finally, once no additional initial keywords are read in the text file the program moves to the next line holding the string ”Rules:”. Once read, the system will move to the final set of lines, each holding individual expressions as well as their terminal and non-terminal rules. These expressions are read line by line and both placed into the expression rules array as well as uploaded to the API’s database.

VII. NodeJS Development for CodeBlock

When a user has fully defined a parsing expression grammar on the CodeBlock website, they may want to check the syntax of their grammar without wanting to initiate a session of the website. Rather, they may want to check the syntax of their programming language from within their chosen coding environment. To allow users to do this, a NodeJS application has been developed for users to download and use to check the syntax on their system. This section will outline the uses of the application for syntax checking natively as well as the inner workings of the application when run.

```
start = (programStart)
varDecl = ID' ''=' '[0-9]+
type = 'int' / 'void'
programStart = type' 'funcStart' 'funcBody' ''}'
ID = [a-zA-Z]+
funcStart = [a-zA-Z]+'(){'
funcBody = ''varDecl';'
```

Fig. 7. Grammar File for Node JS Use

When downloading their grammar from the CodeBlock website, the user will receive an additional file to their ”New Grammar.txt” file called ”cbGrammar.js”. This file holds simply the expressions from the .txt file, without the ”Rules:” line at the top, in the format of a Peggy JS grammar string. The user can then download the codeBlockGrammar.js node

application and use it to parse their grammar at any point, even if the website is not up and running on their system. If the user has already downloaded node on their system, they can simply run the application and start checking the syntax of input lines or files written in the language the grammar corresponds to.

When running the application, it will initially ask the user to input the name of the .js file containing their grammar. With this input, the application will run a search on the system for a file with a name matching the one input. If one is found, it will move on to the next portions of the application, if not, it will ask for input again.

Once a file has been found containing the user's grammar, the application will read the file line by line. This will construct the grammar string in full by concatenating each line together separated by '+' characters. This string will then be passed into the API for PeggyJS once more to generate a parser in the same way as in Sec II-B. The application will then ask the user if they would like to check the syntax of a file, input, or to exit. If they select to check the syntax of input, they will be asked to enter a string. Once entered, the string will be passed into the generated parser and its syntax will be checked. If syntax is correct it will simply return "Correct Syntax!" to the user. If it is not correct, a custom error message will be returned to the user showing them where the parser ran into an error and what it was expecting. The user can then check additional lines, quit, or check the syntax of files.

```
Please enter the .js file containing your parser grammar
> cbGrammar.js
Searching for cbGrammar.js...
File found!
Would you like to test the syntax of command line input (1) or a file (2)?
> 1
Input line to check it's syntax:
> int main(){ a = 20 }
Error at: (){ a = 20 }
Expected ";" or [0-9] but " " found.
Press 1 to continue, 2 to check file syntax or 0 to quit:
> 1
Input line to check it's syntax:
> int main(){ a = 20; }
Correct Syntax!
```

Fig. 8. Input Syntax Check Feature of the CodeBlock Node JS Application

To check the syntax of written files, the user can either select to check a file initially when running the application and supplying a proficient grammar file, or select to check a file after checking the syntax of an input string. When a file name is given and a file matching that name is found on the user's system, it will begin to check the syntax of the code file. To accomplish this, the file will be converted to a single string, removing the line breaks in order to create a parseable line to be fed into the generated parser. The application will then return "Current Syntax!" if there are no errors found, and an error message if an error is thrown by the parser. An example of this feature is shown in Fig 9.

In this example, two files are used, the first having the same code as in Fig 8 having the syntax error named "grammarTestSyntaxError.txt". The second file, having the same code as in Fig 8 which does not feature any syntax errors named "grammarTestFile.txt" is then checked for syntax errors. As we

```
Press 1 to continue, 2 to check file syntax or 0 to quit:
> 2
What file would you like the check the syntax of?
> grammarTestSyntaxError.txt
Testing syntax of file
Error at: { var = 20 }
Expected ";" or [0-9] but " " found.
Press 1 to continue, 2 to check input syntax or 0 to quit:
> 1
What file would you like the check the syntax of?
> grammarTestFile.txt
Testing syntax of file
Correct Syntax!
```

Fig. 9. Written File Syntax Check Feature of the CodeBlock Node JS Application

can observe from the figure, when the syntax error is found in the first file it returns the text of the portion that threw an error as well as what the parser was expecting after checking every available expression. With these two features a user is able to comfortably check syntax of code following their defined grammar without the need to open the CodeBlock site every time.

VIII. Discussion

CodeBlock is the syntax checker backbone of a compiler design tutoring system, *CompiTS* (stands for Compiler Tutoring System), we are currently developing. In contrast to compiler tutoring systems such as ITT [8], our goal is to help students design new languages, experiment with the new language and observe how the semantic interpretation of the newly designed language aligns with the students' design goals. In other words, CompiTS focuses on helping students experience the features of novel language they design.

CompiTS support this vision by allowing the students also design a translation function τ that will correctly map the new language to an existing language for which we have a robust execution engine and has a known semantics. A graphical user interface for τ supports writing translation rules visually, and testing their execution in target languages. The translation function is based on semantic equivalence preservation [2, 13], and allowing rapid prototyping of novel languages. Language translation [14, 18] is a convenient method to prototype a new language, and has been used [17] for both imperative [12] and declarative [5, 6, 19, 22] language implementations.

IX. Conclusion

Though we defer a discussion on CompiTS and its support for translational approach to novel language prototyping as a learning exercise to a future article, we believe we have adequately discussed CodeBlock in this paper and demonstrated how it can be used to check syntactic correctness of new languages in very flexible and intuitive ways. As a tool, it allows multiple useful functions and features to make it an effective syntax checking and grammar management toolkit.

Our future plan is to incorporate CodeBlock in CompiTS language design pipeline. We aim to support a GUI for the design of translation rules, and what it means for two languages to be equivalent. Then support a testing platform to

verify semantic equivalence by comparing execution equality with expected reference outcomes over identical datasets, in the spirit of query equivalence. We also seek to investigate if an abstract equivalence testing process could be developed from user supplied equivalence descriptions. Such a tool could help students learn and practice the theories of program equivalence and machine translation of languages.

ACKNOWLEDGEMENT

This Research was supported in part by a National Institutes of Health Institutional Development Award (IDeA) #P20GM103408, a National Science Foundation CSSI grant OAC 2410668, and a US Department of Energy grant DE-0011014.

REFERENCES

[1] R. Aarssen, J. J. Vinju, and T. van der Storm. Concrete syntax with black box parsers. *Art Sci. Eng. Program.*, 3(3):15, 2019.

[2] A. Ahmed and M. Blume. An equivalence-preserving CPS translation via multi-language semantics. In M. M. T. Chakravarty, Z. Hu, and O. Danvy, editors, *ACM SIGPLAN ICFP 2011, Tokyo, Japan, September 19-21, 2011*, pages 431–444. ACM, 2011.

[3] S. Ali and S. W. Smith. A survey of parser differential anti-patterns. In *2023 IEEE Security and Privacy Workshops (SPW), San Francisco, CA, USA, May 25, 2023*, pages 105–116. IEEE, 2023.

[4] M. Alrammal and G. Hains. Syntax analyzer & selectivity estimation technique applied on wikipedia XML data set. In *DeSE 2013, Abu Dhabi, United Arab Emirates, December 16-18, 2013*, pages 3–8. IEEE, 2013.

[5] X. D. Carlos, G. Sagardui, and S. Trujillo. Mqt, an approach for run-time query translation: From EOL to SQL. In A. D. Brucker, C. Dania, G. Georg, and M. Gogolla, editors, *Workshop on OCL and Textual Modelling (@MODELS 2014), Valencia, Spain, September 30, 2014*, volume 1285 of *CEUR Workshop Proceedings*, pages 13–22. CEUR-WS.org, 2014.

[6] H. C. Chan. Translational semantics for a conceptual level query language. *J. Comput. Sci. Technol.*, 10(2):175–187, 1995.

[7] R. del Vado Vírseda. An interactive tutoring system for learning language processing and compiler design. In M. N. Giannakos, G. Sindre, A. Luxton-Reilly, and M. Divitini, editors, *ITiCSE 2020, Trondheim, Norway, June 15-19, 2020*, page 552. ACM, 2020.

[8] R. del Vado Vírseda. ITT: an interactive tutoring tool to improve the learning and visualization of compiler design theory from implementation. In L. Merkle, M. Doyle, J. Sheard, L. Soh, and B. Dorn, editors, *SIGCSE 2022, Providence, RI, USA, March 3-5, 2022, Volume 2*, page 1074. ACM, 2022.

[9] L. Diekmann and L. Tratt. Don't panic! better, fewer, syntax errors for LR parsers. In R. Hirschfeld and T. Pape, editors, *ECOOP 2020, November 15-17, 2020, Berlin, Germany (Virtual Conference)*, volume 166 of *LIPIcs*, pages 6:1–6:32. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020.

[10] M. S. Farooq, A. Abid, and R. K. Fox. A formal design for the lexical and syntax analyzer of a pedagogically effective subset of C++. In *ICMLA 2016, Anaheim, CA, USA, December 18-20, 2016*, pages 420–425. IEEE Computer Society, 2016.

[11] C. Fox and R. L. Lancaster. Use of a syntax checker to improve student access to computing. In R. H. Austing, L. N. Cassel, and J. C. Little, editors, *SIGCSE 1984, Philadelphia, PA, USA, February 16-17, 1984*, pages 65–68. ACM, 1984.

[12] L. Fredlund, B. Jonsson, and J. Parrow. An implementation of a translational semantics for an imperative language. In J. C. M. Baeten and J. W. Klop, editors, *CONCUR '90, Theories of Concurrency: Unification and Extension, Amsterdam, The Netherlands, August 27-30, 1990, Proceedings*, volume 458 of *LNCS*, pages 246–262. Springer, 1990.

[13] E. Gansner, J. R. Horgan, C. M. R. Kintala, D. J. Moore, and P. Surko. Semantics and correctness of a query language translation. In R. A. DeMillo, editor, *ACM POPL, Albuquerque, New Mexico, USA, January 1982*, pages 289–298. ACM Press, 1982.

[14] M. Héon, J. Aubut, and S. Gaudreau. UMLS-OWL: an OWL 2 translation of the unified medical language system (umls®) semantic-network and metathesarus for publishing in the semantic web. In N. Nikitina, D. Song, A. Fokoue, and P. Haase, editors, *ISWC 2017, Vienna, Austria, October 23rd - to - 25th, 2017*, volume 1963 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2017.

[15] B. Hui, R. Geng, L. Wang, B. Qin, Y. Li, B. Li, J. Sun, and Y. Li. S^2sql: Injecting syntax to question-schema interaction graph encoder for text-to-sql parsers. In S. Muresan, P. Nakov, and A. Villavicencio, editors, *Findings of the Association for Computational Linguistics: ACL 2022, Dublin, Ireland, May 22-27, 2022*, pages 1254–1262, 2022.

[16] D. Majda. Peggy: Parser generator for javascript. https://peggyjs.org/, 2024. Accessed: 5/29/2024, at https://github.com/peggyjs.

[17] I. A. Mason and C. L. Talcott. Actor languages their syntax, semantics, translation, and equivalence. *Theor. Comput. Sci.*, 220(2):409–467, 1999.

[18] Y. Qiu, K. Zhang, H. Zhang, S. Wang, S. Xu, Y. Xiao, B. Long, and W. Yang. Query rewriting via cycle-consistent translation for e-commerce search. In *37th IEEE International Conference on Data Engineering, ICDE 2021, Chania, Greece, April 19-22, 2021*, pages 2435–2446. IEEE, 2021.

[19] C. Sharma. FLUX: from SQL to GQL query translation tool. In *35th IEEE/ACM International Conference on Automated Software Engineering, ASE 2020, Melbourne, Australia, September 21-25, 2020*, pages 1379–1381. IEEE, 2020.

[20] M. Shin, J. Kim, A. Mohaisen, J. Park, and K. Lee. Neural network syntax analyzer for embedded standardized deep learning. In *EMDL@MobiSys 2018, Munich, Germany, June 15, 2018*, pages 37–41. ACM, 2018.

[21] D. Tiselice. A thoughtful introduction to the pest parser. https://pest.rs/book/, 2018. Accessed: 5/29/2024.

[22] S. D. Urban and T. B. Abdellatif. Object-oriented query language access to relational databases: A semantic framework for query translation. *J. Syst. Integr.*, 5(2):123–156, 1995.

[23] J. P. Yoo, D. Smith, S. Yoo, and T. Cheatham. C-language syntax tutoring using machine learning techniques. In T. J. Cheatham, C. C. Pettey, L. W. Dowdy, and J. P. Yoo, editors, *Proceedings of the 35th Annual Southeast Regional Conference, Murfreesboro, Tennessee, USA, April 2-4, 1997*, pages 36–40. ACM, 1997.

[24] Y. Zaki, H. Hajjar, M. Hajjar, and G. Bernard. A survey of syntactic parsers of arabic language. In D. E. Boubiche, H. Hamdan, and A. Bounceur, editors, *BDAW 2016, Blagoevgrad, Bulgaria, November 10-11, 2016*, pages 31:1–31:10. ACM, 2016.