# SYNTAX COLORING AND INCREMENTAL PARSING FOR THE UNICON LANGUAGE

Luis Alvidres

December, 2006

## ABSTRACT

The Unicon language incorporates many elegant ideas that provide productivity gains. Programs in this language tend to be more readable and maintainable[1]. The Unicon language IDE was developed by Nolan Clayton and Clinton Jeffery and contains a set of tools like class browser, edit box and menus that invoke the compiler on the code, using system interface functions to call the compiler executable and providing it with the arguments depending on what option was selected. This project adds two new tools to the existing IDE and removes the need for some system interface calls thus providing additional performance and productivity gains. These new tools are syntax coloring and incremental parsing which interact with the lexical and syntax analyzers of the Unicon compiler without the need of system interface functions.

## INTRODUCTION

Unicon is an object-oriented, goal-directed programming language based on the Icon programming language that originated at the University of Arizona [1]. This language incorporates many elegant ideas that provide productivity gains due to the fact that programs tend to be more readable and maintainable than similar programs written in other very high-level languages[1].

The Unicon language contains an integrated development environment that assists computer programmers in developing software. The IDE contains a set of tools which makes developing in this language even easier. The goal of this environment focuses again on helping developers to achieve faster results and provides a solution to the needs of complex applications.

This project adds two new tools to the existing integrated development environment in order to provide more productivity and performance gains. These new tools are syntax coloring and incremental parsing which interact with the lexical and syntax analyzers of the Unicon compiler and the integrated development environment.

## THE IDE SYNTAX COLORING AND INCREMENTAL PARSING BENEFITS

The growth in both the number and complexity of applications has pushed the need for more sophisticated tools that aids the computer programmers in developing software. An IDE typically provides large numbers of features for authoring, modifying, compiling, deploying and debugging software. Tight integration of various development tasks can lead to further productivity increases[6].

Syntax coloring is a feature that displays source code in different colors and fonts according to the category of terms. This feature eases writing in a structured language such as a programming language as both structures and syntax errors are visually distinct[6]. When looking at pages and pages of code, syntax coloring greatly improves the readability and context of the text. The reader can automatically ignore large sections of comments or code, depending on what one desires.

Incremental parsing is a feature that compiles code while it is being written into the text editor, providing instant feedback to the developer on syntax errors[4].

## THE ORIGINAL IDE INTERACTION WITH THE COMPILER

The starting point for this project was an IDE developed by Nolan Clayton and Clinton Jeffery that contains a set of tools like class browser, edit box and menus that interact with the compiler against the written code using system interface functions to call the compiler executable and providing it with the arguments depending on what option was selected.

The general idea of how the IDE goes about parsing the code and checking for any errors is:

1. User clicks on the `Compile Only` menu option.
2. The `Compile Only` method calls the save method which saves the current code into a file.
3. The `Compile Only` method then uses the system interface function to call the Unicon compiler executable with the appropriate parameters.
4. The Unicon compiler executable opens the file and reads the entire contents, parses it, and generates code.
5. The Unicon compiler creates a log file which the IDE passes as a parameter to the `showanyerror()` function that extracts the errors written into this file by the Unicon compiler and then they are posted into a message box inside the IDE.

This coding example shows how the original IDE calls the Unicon compiler executable using the system interface function.

```
procedure compile()
  system("wunicon -c -quiet -log " || wiconlog || " " || targs ||
        " " || comp1file(current_file), log)

  log := readin(wiconlog)
  showanyerror(log)
end
```

Therefore, every time the user wants to check if the coding is syntactically correct, a manual interaction between the user and the IDE needs to be performed. At this point, the syntactical errors are shown in the form of a list inside the message box.

## IMPLEMENTING SYNTAX COLORING

The re-use of programming code is a common technique which attempts to save time and energy by reducing redundant work thus avoiding "re-inventing the wheel". In this approach the main idea was to reutilize the already built Unicon lexical analyzer and incorporate it into the extended editable text list class that is used by the IDE as the coding typing text area.

The extended editable text list also contains a draw function that overrides the original editable text list class draw function. This new draw function incorporates line numbering to it by dividing the text area in to two. The left side of the text area which can grow or shrink depending on the amount of digits needed in order to represent the amount of lines and the right side of the text area which contains the code. This function also calls the left_string function from the gui package which is in charge of printing the text in the editable text list area.

Therefore, in order to implement the syntax coloring tool several steps were needed. The first step was to create a UniconPackage package which includes all the files needed by the compiler and then include it in the editable text list class as an import statement in order to reutilize the compiler's lexical analyzer.

The second step was to create a new class variable called `errorLineNumber` and add it to the extended editable text list class in order to keep track of where the error is going to be set. This variable contains the line number that is going to be drawn as red.

```
import UniconPackage
# buffertextlist.icn - modified editabletextlist
#
# A scrollable editable text area.  An {Event} is generated
# whenever the contents are changed by the user.
#
$include "guih.icn"
$include "ytab_h.icn"

class BufferTextList : EditableTextList(
      highlightcolor,
      autoindent,
      scroll_y,
      errorLineNumber,
      doReparse
      )

  . . .
end
```

The third step was to create a helper function which decides the color of the text based upon the tokens being returned by the lexical analyzer. The yyin global variable and yylex_reinit function, from the lexical analyzer, are set and called inside this helper function with the text of the current line being processed. Additionally, other modifications were needed in the yyerror.icn file which contains the yyerror function that reports the errors encountered by the lexical and syntax analyzers. An error reinitialization mechanism was needed in this yyerror function in order to avoid the istop function from being called when the merr error counter got to 10 thus stopping the lexical analyzer from finishing the entire line.

```
method left_string_unicon(win, x, y, s, currentLine)
  . . .
  # Check if an error line has been set.
  if \errorLineNumber then {
    # Check if this string belong to the error line
    if ( currentLine = errorLineNumber ) then {
      Fg( win, "Red" )              # Set error line color (red)
      DrawString( win, x, y, s )    # Print the string
      return                        # Exit this method
    }
  }
  # Reinitilize error counter in the yyerror function and the
  # lexical analyzer.
  yyerror( "reinitilize merr errors" )
  yyin := s
  yylex_reinit()
  . . .
  # Get string s tokens
  while ( (token := yylex()) ~=== EOFX ) do {
    case ( token ) of {
      ABSTRACT | BREAK | BY | CASE | CLASS | CREATE |
      DEFAULT | DO | ELSE | END | EVERY | FAIL |
      GLOBAL | IF | IMPORT | INITIALLY |
      iconINITIAL | INVOCABLE | LINK | LOCAL |
      METHOD | NEXT | NOT | OF | PACKAGE |
      PROCEDURE | RECORD | REPEAT | RETURN | STATIC |
      SUSPEND | THEN | TO | UNTIL | WHILE |
      LOCAL                                     : Fg(win, "Blue")
      STRINGLIT | CSETLIT                       : Fg(win,"Dark Red")
      default                                   : Fg(win, "Black")
    }
    . . .
    # Print the string
    DrawString(win, x, y, s[last_s_Position:(new_s_Position+1)])
    . . .
  }
  # Draw the rest of the string s that was not a token
  Fg( win, "Dark Green" )
  DrawString( win, x, y, s[ last_s_Position : ( *s + 1 ) ] )

end
```

Finally, the last step was to modify the draw function in order to utilize the new functionality.

```
method draw(s, left_pos, yp, i)
  local s1, s2, newp, fh, asc, desc, yp2
  . . .
  . . .
  . . .
  # Check if an error line has been set.
  if \errorLineNumber then{
    # Check if this string belong to the error line
    if ( i = errorLineNumber ) then {
      # Set error line color (red)
      Fg( self.cbwin, "Red" )
    }
  }
  # Print line numbering
  left_string(self.cbwin, left_pos -TextWidth(self.cwin, i)-3, yp, i)

  # Print code
  left_string_unicon ( self.cbwin, left_pos, yp, s, i )
  . . .
  . . .
  . . .
end
```

The Unicon language tokens were divided inside the syntax coloring function into 5 main categories:

*Control structures and reserved words*

Unicon has many reserved words. Some are used in declarations, but most are used in control structures. These reserved words were given the blue color.

*Strings and Csets*

The non-numeric atomic types available in Unicon are character sequences (strings) and character sets (csets). String literals are enclosed in double quotes, while cset literals are enclosed in single quotes. This group was given the dark red color.

*Comments*

Unicon programming language has a construct that provides a mechanism for embedding information in the source code. Comments begin with the # character and extend to the end of the line on which they appear. The compiler ignores them. This group was given the dark green color.

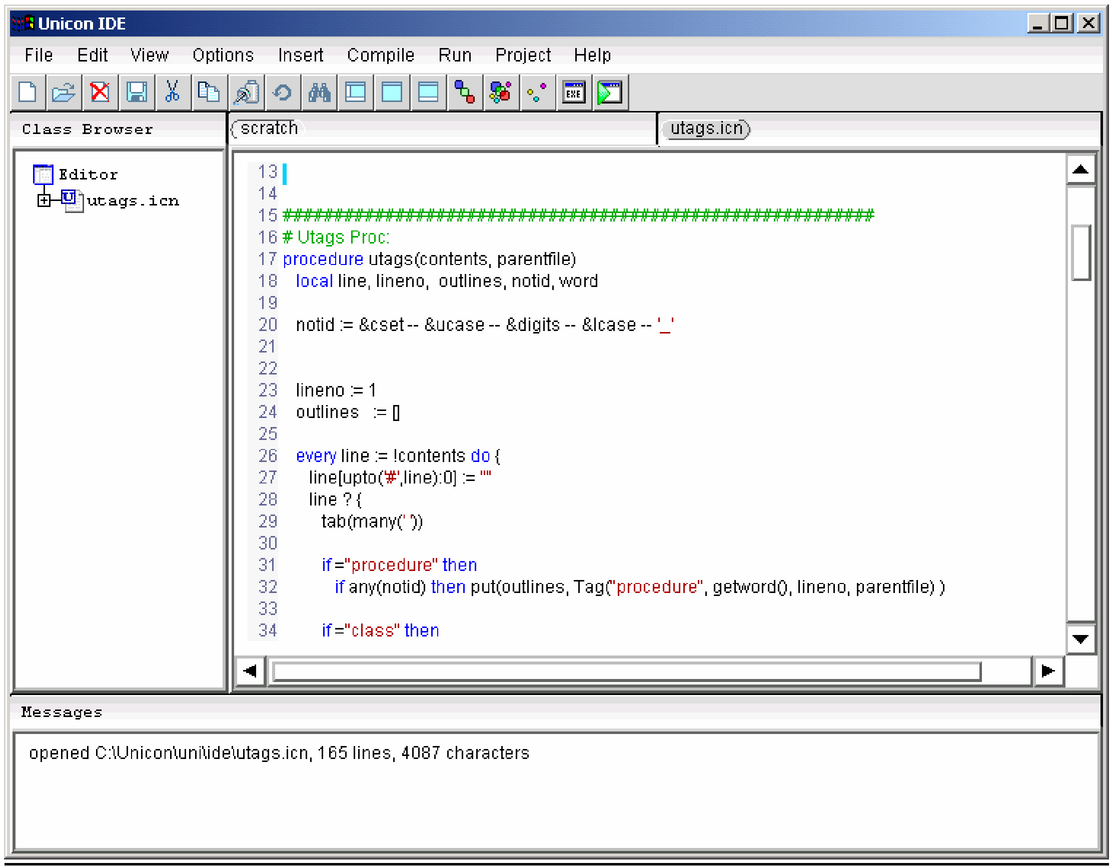
*Syntax Errors*

A syntax error refers to a mistake in a statement's syntax and needs to be corrected otherwise a compilation error would result. This group was given the red color.

*The Rest*

The rest of the language including keywords, identifiers, operators, preprocessor commands, predefined symbols and built-in functions were put in this group. This group was given the black color.

An example of the color coding is presented here:



```


###########################################################
# Utags Proc:
procedure utags(contents, parentfile)
  local line, lineno,  outlines, notid, word

  notid := &cset -- &ucase -- &digits -- &lcase -- '_'


  lineno := 1
  outlines   := []

  every line := !contents do {
    line[upto('#',line):0] := ""
    line ? {
      tab(many(' '))

      if ="procedure" then
        if any(notid) then put(outlines, Tag("procedure", getword(), lineno, parentfile) )

      if ="class" then
```

```
opened C:\Unicon\uni\ide\utags.icn, 165 lines, 4087 characters
```

Figure 1: Syntax coloring example

## IMPLEMENTING INCREMENTAL PARSING

The main goal behind the implementation of incremental parsing is to compile the code while is being written into the text editor, providing instant feedback to the developer on syntax errors. Several steps were taken in order to acomplish this task.

### *Baseline*

The current starting point of this project is to call the compiler executable using the `system()` function against the Unicon source file being edited inside the IDE. The performance baseline for this project was obtained from a study of 1628 Unicon source files that contained from 3 to 7000 lines of code.

The study consisted on compiling each of the 1628 files using the starting point IDE with some minor modification to the `handle_compile_menu_item(ev)` method and the addition of another method called `handle_statistics_menu_item(ev)`. The second method was in charge of opening, calling the `handle_compile_menu_item(ev)` method, obtaining the parsing time, storing the parsing time into a file and closing each of the Unicon source files which were previously recollected. The results are shown in Figure 2.

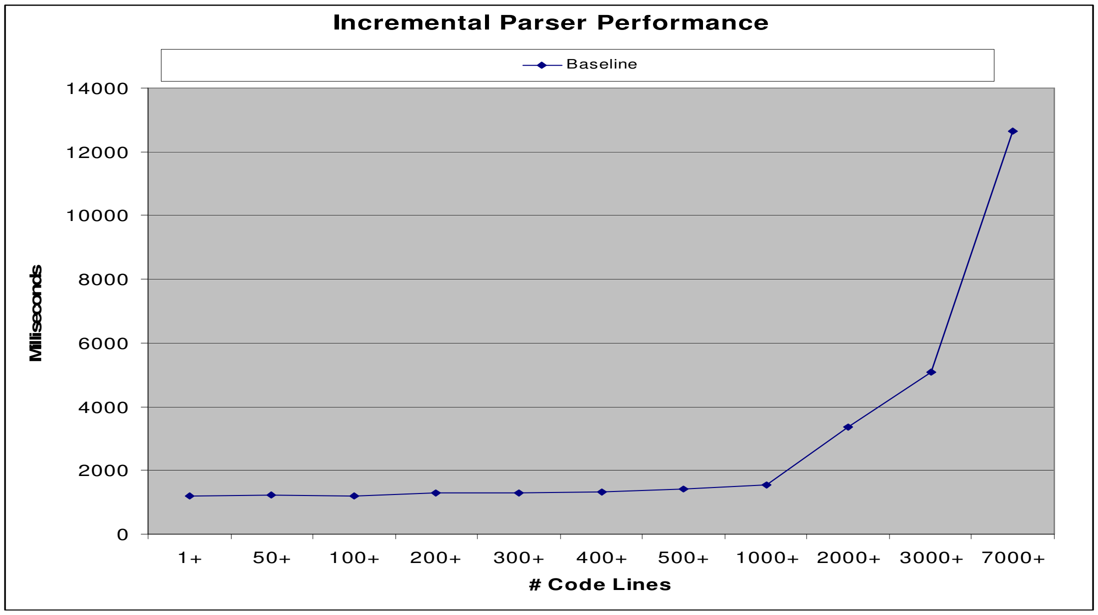


Figure 2: Baseline performance (parsing time in milliseconds)

*First Approach*

The first attempt to improve on the baseline was to modify the compiler so that when the contents change inside the editable text list, the code then gets saved into a temporary file and fed to the compiler which needs to get called without using any `system()` functions.

This approach includes the modification of the compiler `main()` procedure by adding an additional function called `unicon()` which becomes the liaison function between the IDE and the compiler. Another modification was the creation of a UniconPackage package that includes all the files needed by the compiler and then it was included in the IDE as an import statement in order to avoid problems with global variables utilized by the IDE and the compiler. Finally, add the functionality inside the editable text list so that when the contents change, a temporary file is created and passed to the compiler newly created liaison function.

```
procedure main ( argv )
  return unicon ( argv )
end
```

Pros
- Minimal changes to the IDE and to the compiler.
- No need for `system()` function calls.

Cons
- Opening, writing and closing the temporary files.
- Since the compiler also needs to open, read and close the temporary file in this approach it takes more time to parse the code.

Metrics

- Based upon a study on 1628 Unicon source files that contained from 3 to 7000 lines of code:
    - A 90% increase performance for small Unicon source files.
    - A 40% increase performance for bigger Unicon source files.

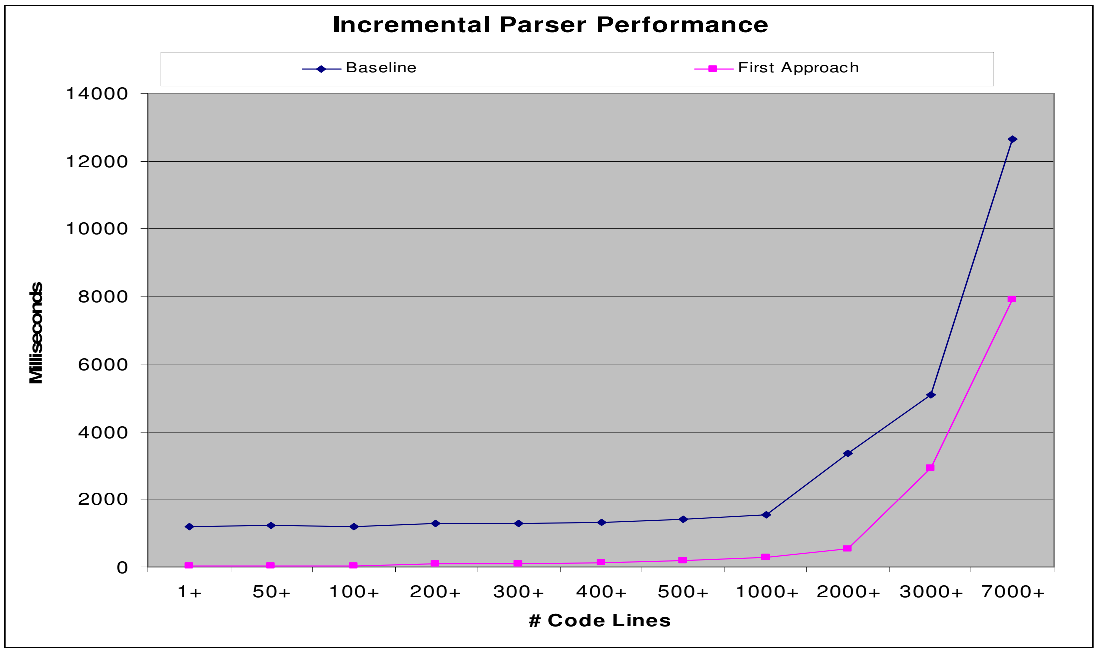


Figure 3: Baseline performance Vs First Approach performance

Conclusions

- Great performance improvement for small Unicon source files but not so great for larger files.
- The overhead of calling `system()` function is almost 2 seconds, on a Windows 2000 Professional Edition machine with a 2.5 GHz processor and 512 MB of RAM, and dominates cost except on the largest files.
- In the real world, programs tend to have more than a thousand lines of code, therefore more work is needed.

### *Second Approach*

Instead of the code being written into a temporary file and then passed to the compiler, the code is obtained from the editable text list and then passed in the form of a string to the compiler thus avoiding the need to create a temporary file and prevent the compiler from trying to open a file in order to read the code.

This approach includes many modifications to the compiler and the preprocessor programs since both needs to be aware of the new type of data being received.

Many of the modifications done in the two programs refer to the fact that both programs are expecting a file to open and read from. So, many of the efforts were done in identifying the type of data being received and how it was going to be processed.

Pros

- Avoid creating a temporary file to store the code.
- Minimal changes to the IDE.
- Less parsing time since the compiler does not need to open, read and close a file.

Cons

- Many changes to the compiler and preprocessor programs.

Metrics

- Based upon a study on 1628 Unicon source files that contained from 3 to 7000 lines of code:
    - Saving time varied from 10 to 40 milliseconds.

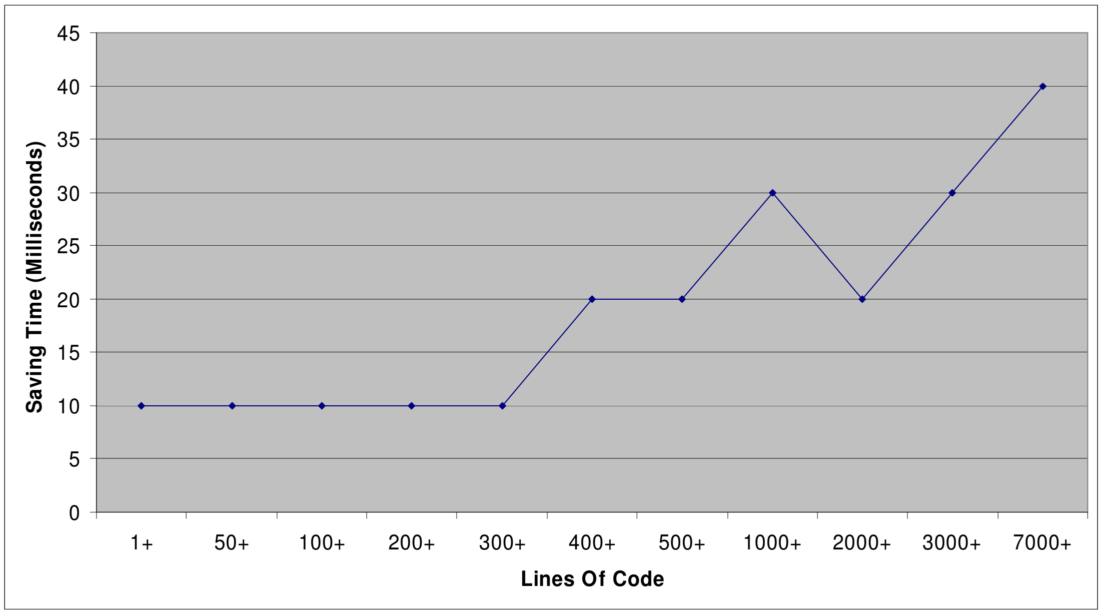


Figure 4: Saving times for temporary files.

- A 2% performance increase versus the first approach.

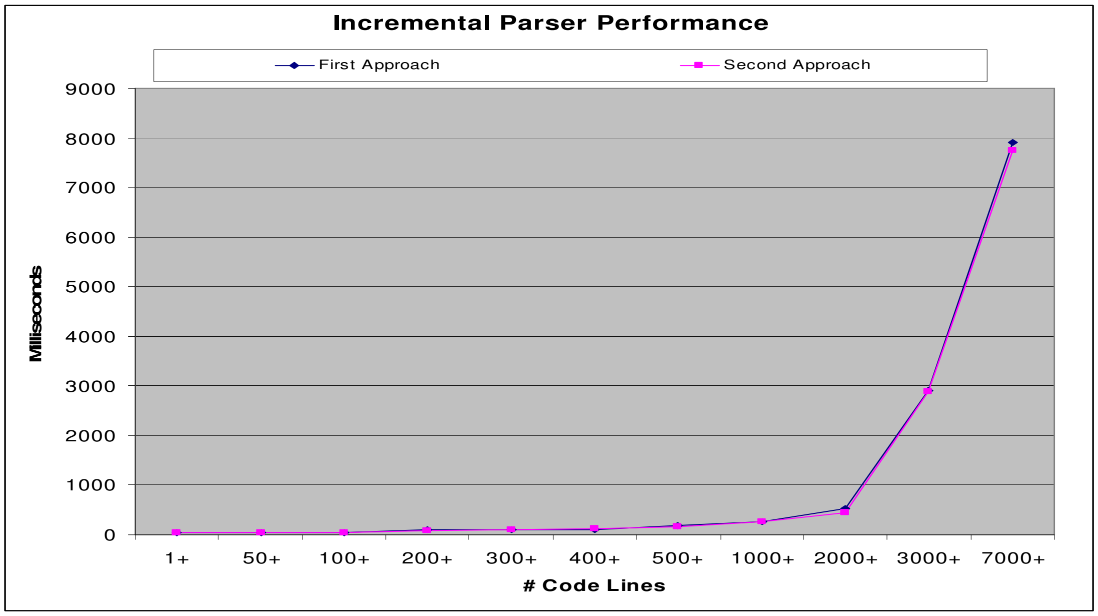


Figure 5: First Approach Vs Second Approach performance

Conclusions

- Only a 2% increase in performance was obtained due to the fact that the time needed to save the code in a temporary file it is very low compared to the amount of time it takes the compiler to parse the entire source file.

- A performance issue is still present for larger files.

*Third approach*

Instead of passing to the compiler the entire source code contained in the editable text list, a portion of the code, which is currently being changed, is extracted and passed to the compiler.

In this approach, a segment of the code is extracted from the editable text list that is surrounded by enclosing statements like `class -> end`, `procedure -> end` or `method -> end` thus focusing only on the segment of code that is being changed. Segmenting the code reduces the amount of code to be parsed thus reducing the amount of time the compiler needs in order to run it against the parser.

Some of the modifications needed in this approach are located in the IDE. A function called G`etCode()` was created and is in charge of going thru the code in search for the enclosing statements. Once again the lexical analyzer is re-used in order to identify the enclosing statements tokens.

Pros
- Avoid creating a temporary file to store the code.
- Less parsing time since the compiler does not need to open, read and close a file.
- Less parsing time since the compiler does not need to compile the entire source code.

Cons
- Many changes to the IDE.
- Time needed to get the segment of code that is going to be parsed.

Metrics
- Based upon the feedback from 3 developers 90 milliseconds waiting time was a good response time from the IDE attempting incremental parsing.

- Based upon a study on 9227 methods and procedures from different Unicon source files, here are some interesting results:

    - Procedures/Methods that have equal or less than 100 lines of code take less than 90 milliseconds to parse.

    - 98% of methods and procedures were less than 100 lines of code.

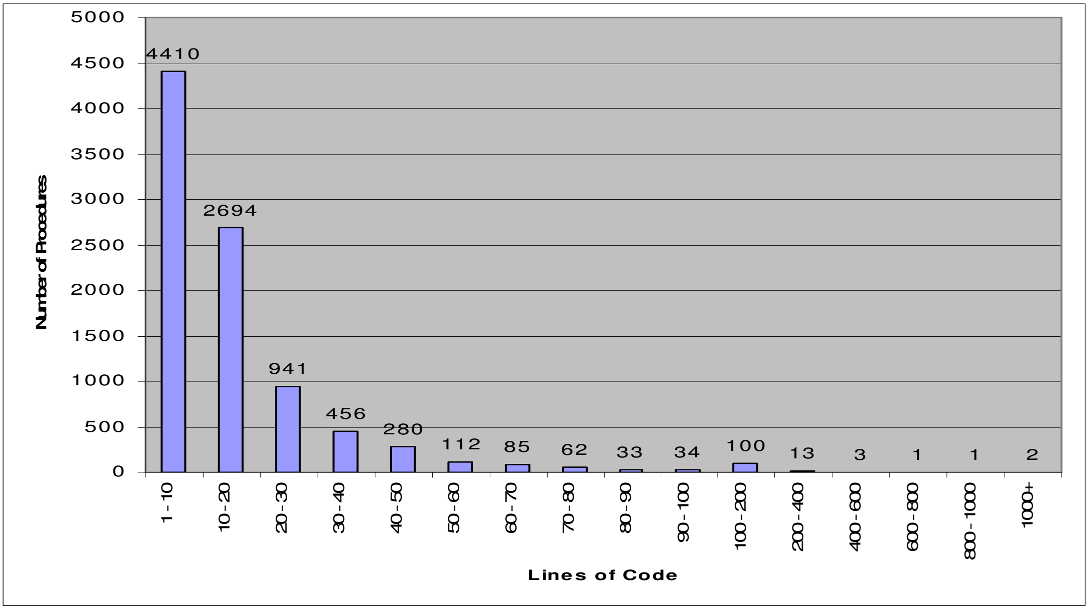


Figure 6: Procedures/Methods grouped by the amount of lines of code.

- 98% of methods and procedures take less than 90 milliseconds to parse.

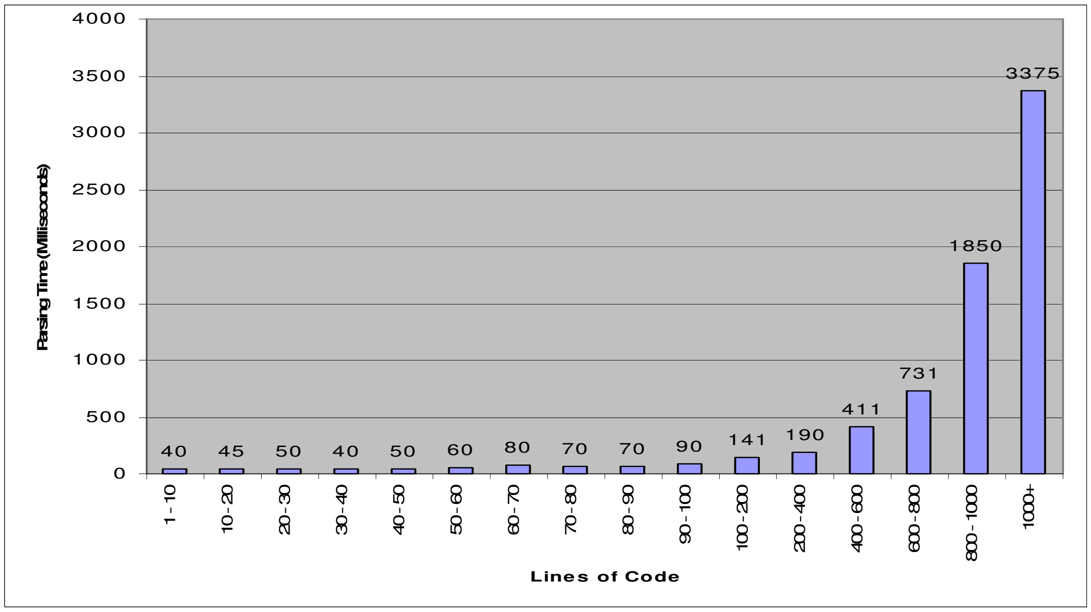


Figure 7: Procedures/Methods parsing time based upon the number of lines.

- Based upon a study on 1628 Unicon source files that contained from 3 to 7000 lines of code:
  - As Unicon source files get bigger the performance increase percentage grows exponentially.

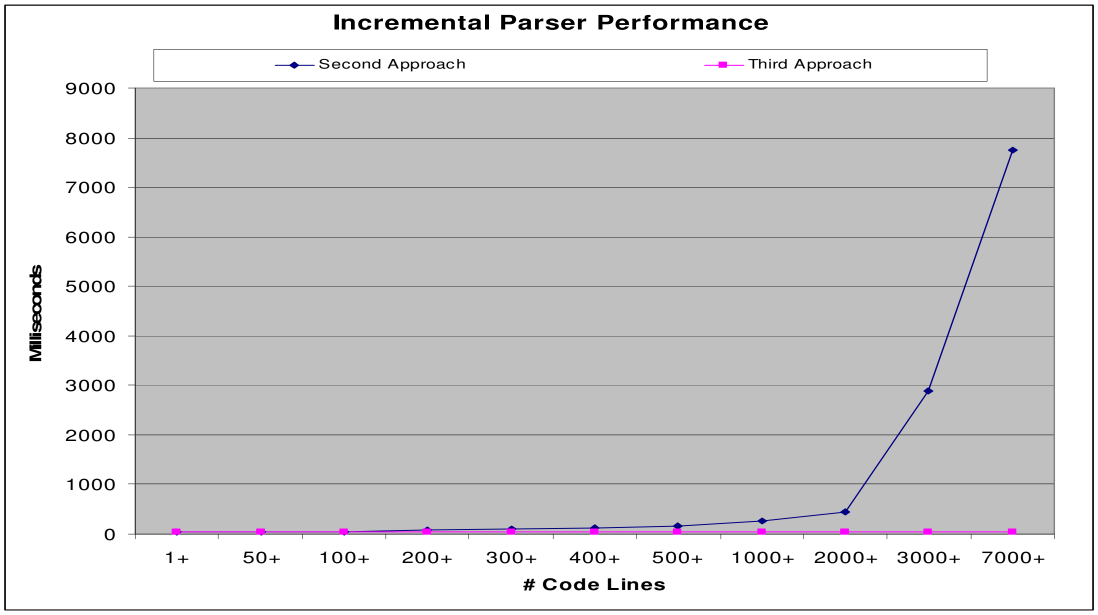


Figure 8: Second Approach Vs Third Approach performance

- Based upon a study on 1628 Unicon source files that contained from 3 to 7000 lines of code:
    - Looking at all of the approaches performance at once.

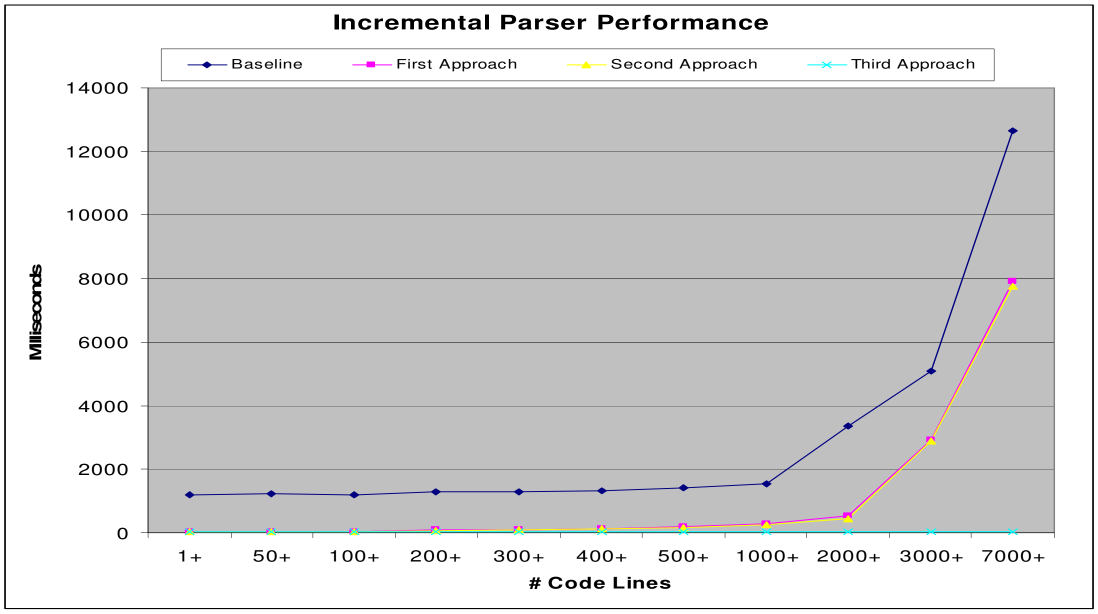


Figure 9: All approaches performance

Conclusions

- Parsing time was dramatically improved by just parsing a portion of the code being edited.

- 98% of the time the compiler will only need to parse at most 100 lines of code which has a response time of about 90 milliseconds even if your programs have thousands of lines of code.

- 2% of the time the compiler will have to parse more than 100 lines of code and therefore increase the response time from the parser giving the sense that the IDE is having technical difficulties. This was detected on pathologically large procedures/methods, which do exist but are usually machine generated.

## FUTURE WORK

There is still more work to be done in both of the tools created in this project. The syntax coloring tool could be extended in order to color or highlight more types of tokens like operators or even better, color the background of methods, procedures, and classes thus greatly improving readability.

As for the incremental parsing tool, there is still a 2% chance, based upon the study of the second approach, that a method or procedure goes beyond the 90 milliseconds response time and therefore creating a bad user experience.

A code injection approach could be perused. By just extracting small segments of code, like in the third approach but without going all the way to the start or the end of a procedure or method, and then deciding which enclosing statements are needed for them to be compiled, it reduces the amount of lines to be parsed, thus avoiding a longer waiting times for these 2% of methods and procedures that take more than 90 milliseconds to compile.

Additional variables would be needed in order to keep track of which lines of code were injected so when an error occurs, the real line number can be obtained and the injected lines can be removed. The injected code will only be fed into the compiler and it will not appear in the source code shown in the IDE.

Finally, some other approaches could be taken in order to avoid exceeding the 90 milliseconds response time like preventing the compilation based upon the number of lines being compiled or providing the user with the capability of deciding if they wish to wait for longer periods of time.

## CONCLUSION

The growth in both the number and complexity of application has pushed the need for more sophisticated tools that aids the computer programmers in developing software. Although Icon was developed to obtain productivity gains from its language, syntax coloring and incremental parsing are other ways to gain productivity by providing instant feedback to the developer on syntax errors.

## REFERENCES

1. Jeffery, C.; Mohamed, S.; Pereda, R.; Parlett R. **Programming with Unicon.** Available at: http://www.unicon.org

2. Rekers, J.; **Parser Generation for Environments.** Amsterdam: University of Amsterdam, 1992

3. J. Heering, P. Klint, J. Rekers; **Lazy and incremental program generation.** *ACM Press 1994, New York, NY, USA*
   Research paper at:
   http://portal.acm.org/citation.cfm?id=177750&coll=portal&dl=ACM&CFID=2773302&CFTOKEN=24996120

4. M. F. Cowlishaw; **LEXX – A programmable structured editor.**
   *IBM J. RES. DEVELOP. VOL. 31 NO, I JANUARY 1987*
   Research paper at:
   http://www.research.ibm.com/journal/rd/311/ibmrd3101G.pdf

5. Boshernitsan M., Graham S.; **Interactive Transformation of Java Programs. ,** *OOPSLA'06, Oct 22–26, 2006, Portland, Oregon, USA.*
   Research paper at: Department of Electrical Engineering and Computer Science University of California, Berkeley.

6. From Wikipedia, the free encyclopedia; **Syntax Highlighting and Parsing.** Available at: http://en.wikipedia.org/wiki/Syntax_highlighting