

# Compiler Development Coursework

Component 1

**Eleftherios Kousis - 1408140**

November 6, 2017

## Introduction

In this coursework, the goal was the development of a compiler that would compile files with the extension “.tri”. This specific component was to create a scanner (Lexical Analyser) and a parser (Syntax Analyser) part of the compiler.

## Scanner

### Scanner Requirements

For the Scanner, a class was created that is responsible for the following:

1. Identify and skip whitespace
2. Identify and skip language comments
3. Identify and produce error for unknown characters in the language
4. Identify and produce error for unterminated strings.

All the above requirements were implemented successfully.

## Scanner Implementation

Requirements 1 and 2.

The way this was achieved is by including a method called ScanSeparator() in the Scanner class that identifies and skips whitespace and language comments. The following piece of code does exactly that.

```
// Skip a single separator.
void ScanSeparator()
{
    switch(_source.Current) {
        // skip comments
        case '!':
            Console.WriteLine("Skipped comment.");
            _source.SkipRestOfLine();
            _source.MoveNext();
            break;
        // space
        case ' ':
        // newline char
        case '\n':
        // return char
        case '\r':
        // tab char
        case '\t':
            Console.WriteLine("Skipped Whitespace.");
            // skip the above
            _source.MoveNext();
            break;
    }
}
```

In order to test it, some statements are included to print out the whitespace and comments when they are being skipped. The following .tri file was used as input:

```

! Comment
let
  const MAX ~ 10;
  var n: Integer
in
begin
  if (n>0) ∧ (n<=MAX) then
    while n > 0 do
      begin
        n := n-1
      end
    else
  end
end

```

And the following output was returned:

```

Skipped comment.
Kind=Let, spelling="let"
Skipped Whitespace.
Skipped Whitespace.
Skipped Whitespace.
Kind=Const, spelling="const"
Skipped Whitespace.
Kind=Identifier, spelling="MAX"

```

As it can be seen in the output above, in the **.tri** file before the **Let** there is comment which is identified and skipped by the scanner. The same happens for the whitespace before and after the **Const**.

### Requirements 3 and 4.

In order to do identify and produce the errors for the for requirements 3 and 4 , we use two methods:

For requirement 3, unknown characters:

If a one of the character is not specified in the language definition for terminals then the scanner will treat it as an error. This is done in the scanner by returning a Error as a **Tokenkind** which is when none of the case are satisfied which means that is not part of the set specified in the language definition. This is done by using a **Tokenkind** checker in the **ToString()** method of the **Token** as you it can be seen here:

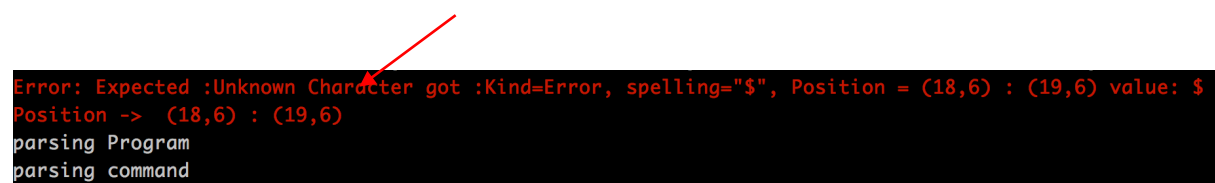
```

    mssg += "Unterminated string ";
  } else {
    mssg += "Unknown Character";
  }
  compiler._parser._errorReporter.ReportError(mssg, token, token._position);
}

```

## Testing:

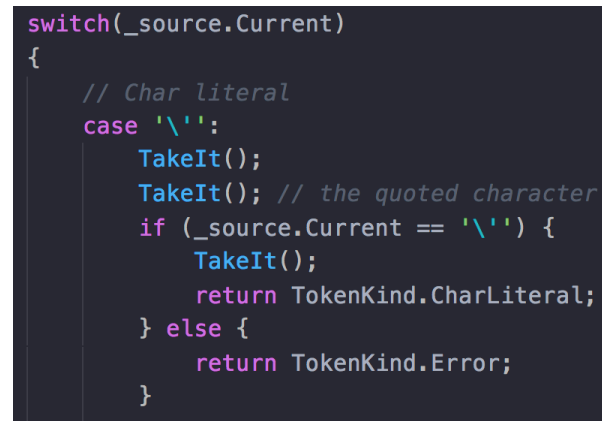
If I add the character \$ after **let** in the file used above. The scanner will identify and locate the error in the console:



```
Error: Expected :Unknown Character got :Kind=Error, spelling=\"$', Position = (18,6) : (19,6) value: $
Position -> (18,6) : (19,6)
parsing Program
parsing command
```

## For requirement 4, unterminated strings:

In order to print out any unclosed character a checker was implemented. This is achieved with the following piece of code:

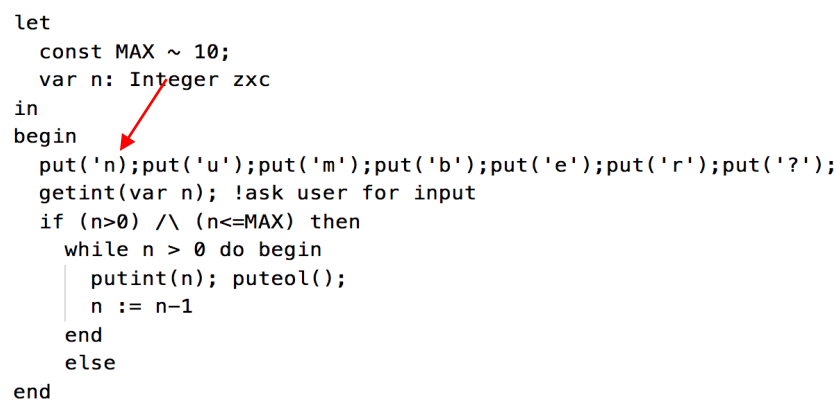


```
switch(_source.Current)
{
    // Char literal
    case '\\':
        TakeIt();
        TakeIt(); // the quoted character
        if (_source.Current == '\\') {
            TakeIt();
            return TokenKind.CharLiteral;
        } else {
            return TokenKind.Error;
        }
}
```

and after we report the error with the same way we reported the previous message using the ReportError method of the ErrorReporter class.


## Test

In order to test we include the following file in with non-closing quotes:



```
let
  const MAX ~ 10;
  var n: Integer zxc
in
begin
  put('n');put('u');put('m');put('b');put('e');put('r');put('?');
  getint(var n); !ask user for input
  if (n>0) /\ (n<=MAX) then
    while n > 0 do begin
      putint(n); puteol();
      n := n-1
    end
  else
  end
end
```

The scanner finds the missing quote and produces the following error:



```
Error: Expected :Unterminated string got :Kind=Error, spelling="'M", Position = (18,6) : (20,6) value: 'M'
Position -> (18,6) : (20,6)
parsing Program
parsing command
parsing single command
```

As it can be seen the error is identified as well as with its position.

## From Scanner to Parser

In order to introduce more advanced way to parse the language with the compiler, a new part is added called the Parser. What the parser does is provide a enhanced structure to the compiler. What the scanner did was only check for token level errors and provide the Parser with TokenKinds needed to construct the language definition representation into an *Abstract Syntax Tree*.

# Parser

## Parser requirements

1. Your Parser (Syntax Analyser) should work on the sequence of tokens created by the scanner. These tokens should be grouped in to sentences, creating a set of instructions. The instruction set will represent the purpose of the program defined in the source code.
2. Your Parser should be able to identify Syntax Errors in the source program and produce errors for instructions that do not conform to the language definition.
3. On completion of the Parser stage your compiler should maintain an Internal representation of the instruction set and write out an instruction set, in the correct order, to the console.

## Parser Implementation

Requirement 1, creating a set of instruction:

In order to fulfil this requirement the language definition provided was converted into code by creating the following subclasses:

- Parser – Commands.cs
- Parser – Common.cs
- Parser – Declarations.cs
- Parser – Expresssions.cs
- Parser – Parameters.cs
- Parser – Programs.cs
- Parser – Terminals.cs
- Parser – TypeDenoters.cs
- Parser – Vnames.cs

Each of those the above partial classes represent a different part of the language definition representation and together they make the Parser.

#### Requirement 2, language definition errors:

The classes Location, SourcePosition and ErrorReporter are used together identify and report the errors.

#### Requirement 3, instruction set:

The output of the parser is an instruction set that represented the order in which everything is being parsed.

### Testing the parser

In order to test the parser the following files provided were used along with their outputs:

- test-mini.tri
- test-mini-error.tri
- test-mini-error2.tri
- printer.tri
- print.tri

For test-mini.tri the output provided was indetical and was parsed with 0 errors:

```
parsing integer
parsing single command
Finished with 0 errors.
root@76c3c50bd163:/src/app/Triangle
```

For test-mini-error2.tri the output was exactly the same and was parsed with 1 error:

```
parsing identifier
Error: Expected :In got :Kind=If, spelling="if" value: if
Position -> (2,4) : (4,4)
parsing single command
parsing single command
Finished with 1 errors.
root@76c3c50bd163:/src/app/Triangle
```

Any error that occurs produced a message that specifies what kind of error it is what should be there instead and the location of the error.

All of the files were parsed identically to the outputs provided.

The above, underlined statement certifies that the Parser was constructed properly based on the language definition. The appendix contains all the outputs of every file and also the code used to achieve that.

## Appendix – Source files used for testing.

### Test-mini.tri

```
let
  const MAX ~ 10;
  var n: Integer
in
  begin
    if (n>0) /\ (n<=MAX) then
      while n > 0 do
        begin
          n := n-1
        end
      end
    else
      end
  end
```

### Test-mini-error.tri

```
let
  const MAX ~ 10;
  var n: Integer
in
  if (n>0) /\ (n<=MAX) then
    while n > 0 do begin
      n := n-1
    end
  else
    end
end
```

### test-mini-error2.tri

```
let
  const MAX ~ 10;
  var n: Integer
in
  begin
    if (n>0) /\ (n<=MAX) then
      while n > 0 do begin
        n := n-1
      end
    end
  end
```

### printer.tri

```
while \eol() do
  begin get(var ch); put(ch) end;
geteol(); puteol()
```

### print.tri

```
getint(var i);putint(i);puteol()
```