Phase 2. Parser

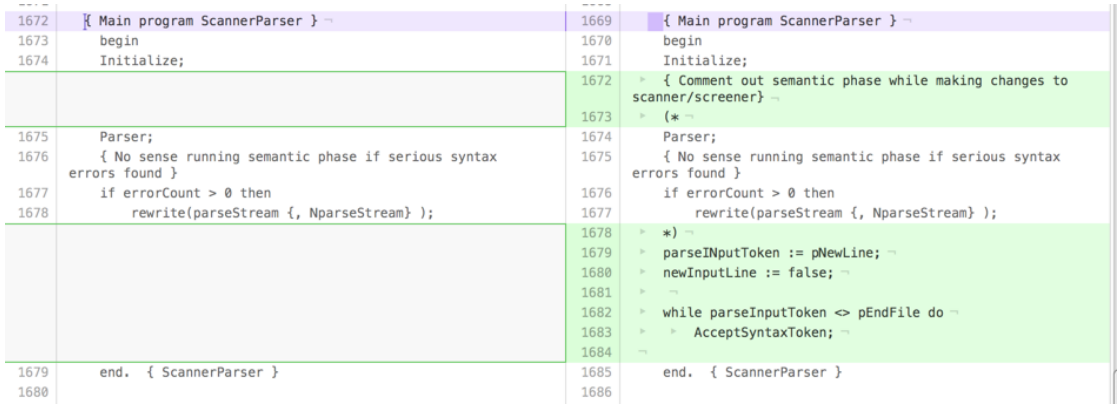
The Compildres

Arthur Margulies
Harold Treen
Paul Wells

February 25, 2015

1 - Updating Parser.pt

In the previous phase, Parser.pt was changed to only output a set of tokens. The first step to implementing the parser was rolling back these changes so the parser would also be called:



```
1672   { Main program ScannerParser } ¬
1673     begin
1674     Initialize;
1675     Parser;
1676     { No sense running semantic phase if serious syntax
       errors found }
1677     if errorCount > 0 then
1678         rewrite(parseStream {, NparseStream} );
1679     end.  { ScannerParser }
1680
```

```
1669   { Main program ScannerParser } ¬
1670     begin
1671     Initialize;
1672   ▸ { Comment out semantic phase while making changes to
       scanner/screener} ¬
1673   ▸ (* ¬
1674     Parser;
1675     { No sense running semantic phase if serious syntax
       errors found }
1676     if errorCount > 0 then
1677         rewrite(parseStream {, NparseStream} );
1678   ▸ *) ¬
1679   ▸ parseINputToken := pNewLine; ¬
1680   ▸ newInputLine := false; ¬
1681   ▸ ¬
1682   ▸ while parseInputToken <> pEndFile do ¬
1683   ▸   ▸ AcceptSyntaxToken; ¬
1684   ¬
1685     end.  { ScannerParser }
1686
```

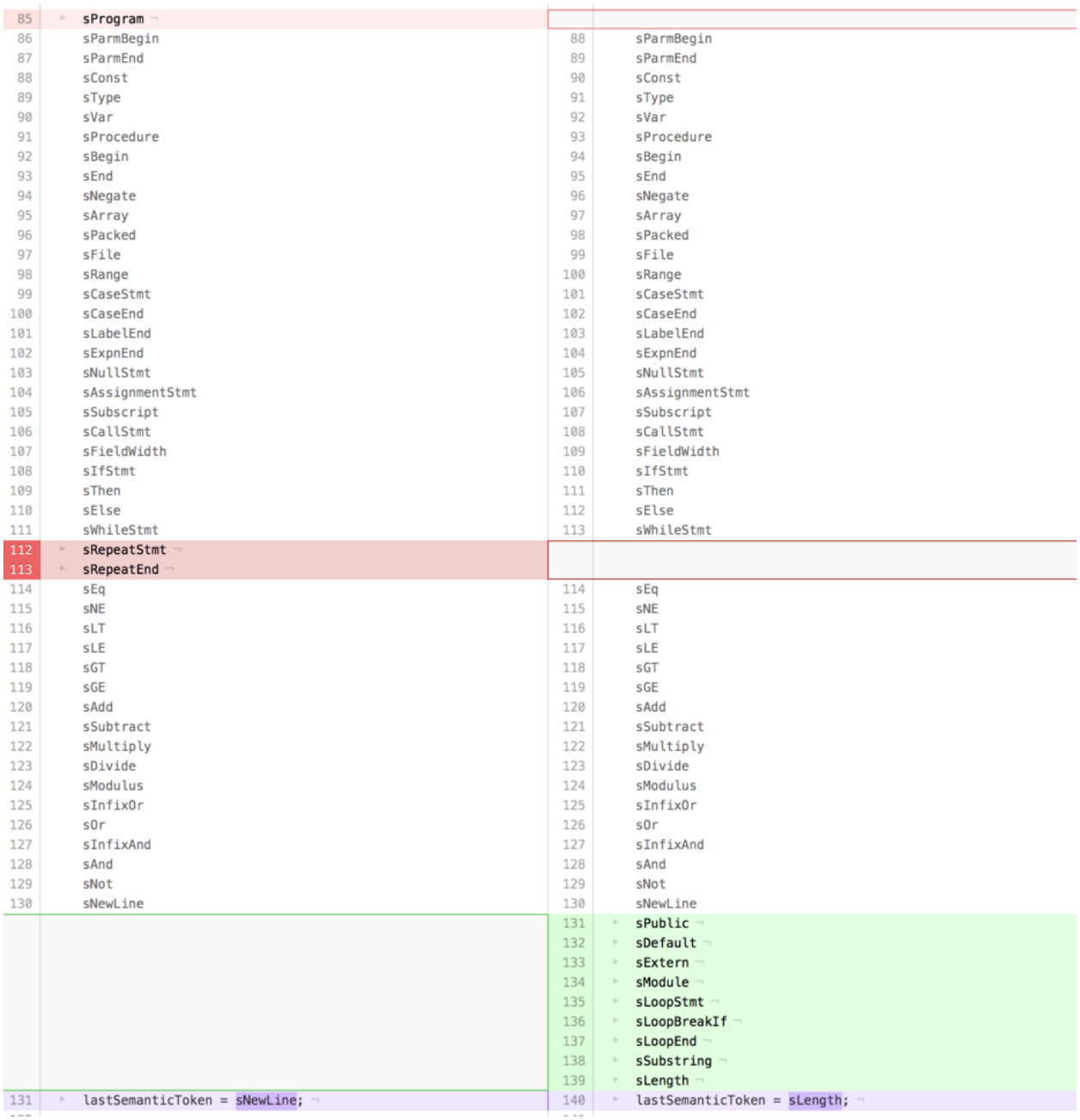
Parser.pt Diff

2 - Updating Token Definitions

The changes to the scanner resulted in a new set of tokens which would be fed to the parser. As such, the tokens defined as input to the parser needed to be updated to remove the old tokens (.pThen, .pUntil, .pDo, .pProgram, .pConst, .pProcedure, .pBegin. pColonEquals) and add the new ones (.pFunc, .pSwitch, .pDefault, .pElsif, .pLoop, .pBreak, .pModule, .pExtern, .pEquals, .pNotEqual, .pDotDot, .pHash, .pDoubleColon, .pAssign).

Along with changes to input, there was also changes to output. Tokens were removed from the list of parser outputs (.sProgram, .sRepeatStmt, .sRepeatEnd) and others added (.sPublic, .sDefault, .sExtern, .sModule, .sLoopStmt, .sLoopBreakIf, .sLoopEnd, .sSubstring, .sLength).

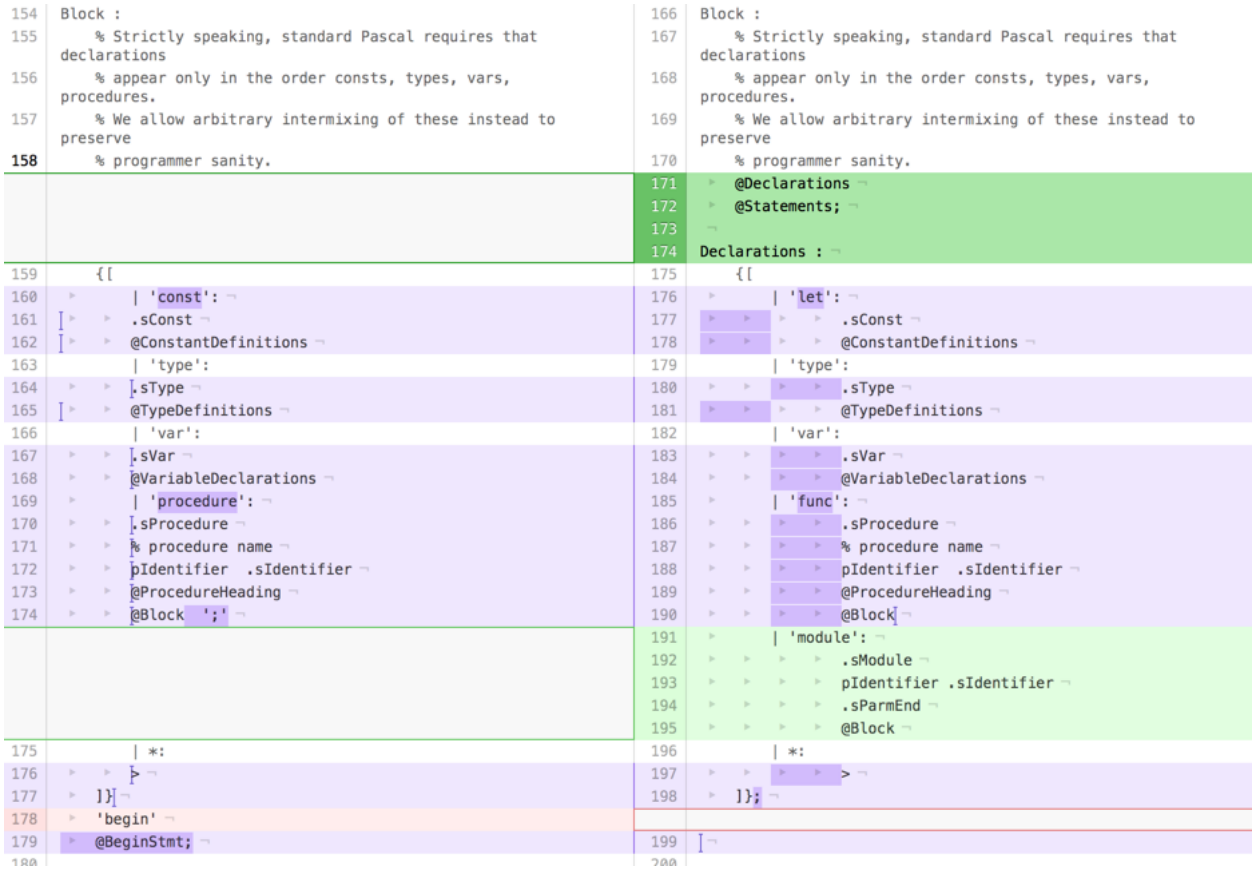
parser.ssl - Input Token Changes



parser.ssl - Output Changes

3 - Modifying The Block Rule

The block rule previously worked by parsing out declarations until it hit a begin token. With begins being removed, the structure was switched to two new rules: “Declarations” and “Statements”. The declaration rule collects all the declaration tokens and the statements rule is called after to collect the statement tokens. No begin is used to switch between the two.



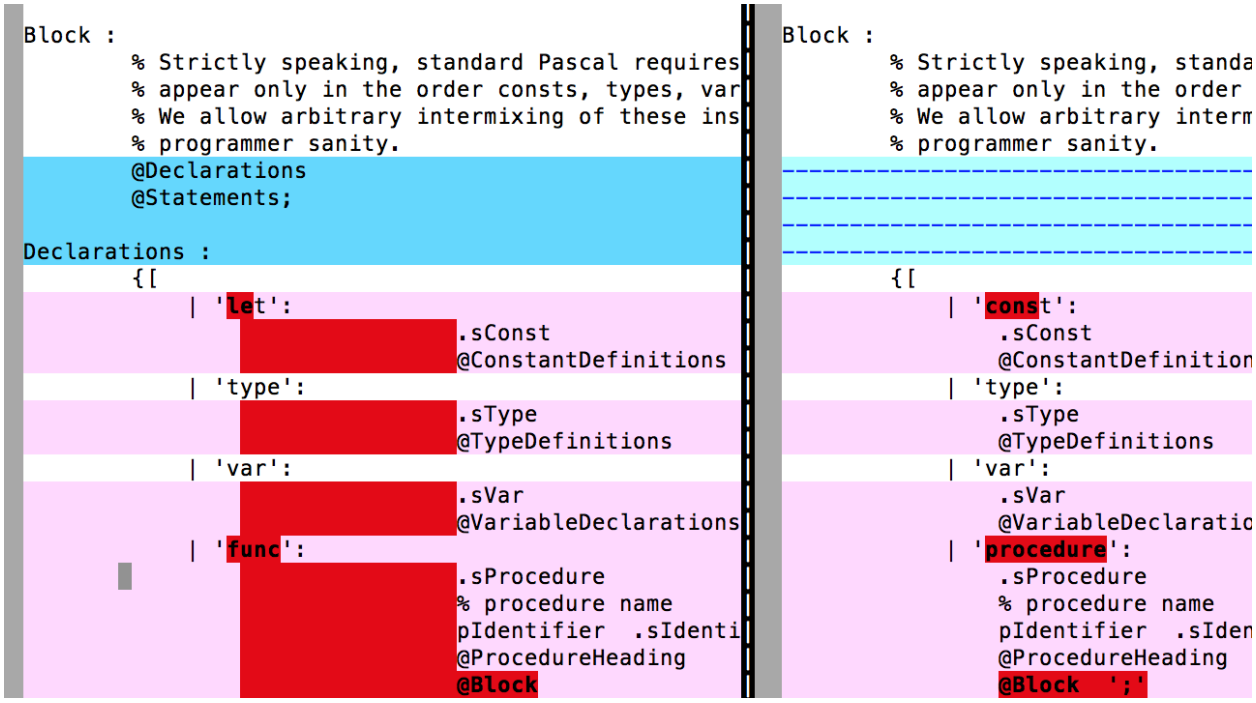
```
Block :
    % Strictly speaking, standard Pascal requires that declarations
    % appear only in the order consts, types, vars, procedures.
    % We allow arbitrary intermixing of these instead to preserve
    % programmer sanity.
    {[
        | 'const':
        .sConst
        @ConstantDefinitions
        | 'type':
        .sType
        @TypeDefinitions
        | 'var':
        .sVar
        @VariableDeclarations
        | 'procedure':
        .sProcedure
        % procedure name
        pIdentifier  .sIdentifier
        @ProcedureHeading
        @Block  ';'
        | *:
        }
    ]}
    'begin'
    @BeginStmt;
```

```
Block :
    % Strictly speaking, standard Pascal requires that declarations
    % appear only in the order consts, types, vars, procedures.
    % We allow arbitrary intermixing of these instead to preserve
    % programmer sanity.
    @Declarations
    @Statements;

Declarations :
    {[
        | 'let':
            .sConst
            @ConstantDefinitions
        | 'type':
            .sType
            @TypeDefinitions
        | 'var':
            .sVar
            @VariableDeclarations
        | 'func':
            .sProcedure
            % procedure name
            pIdentifier  .sIdentifier
            @ProcedureHeading
            @Block
        | 'module':
            .sModule
            pIdentifier .sIdentifier
            .sParmEnd
            @Block
        | *:
        >
    ]};

]
```

4 - Updating Declarations

4.1 - Updating let, var, type

Block was separated into calls to a Declarations rule and a Statements rule. The logic for let var and type was moved into the Declarations rule. The logic stayed the same except that the “const” keyword was changed to “let”.

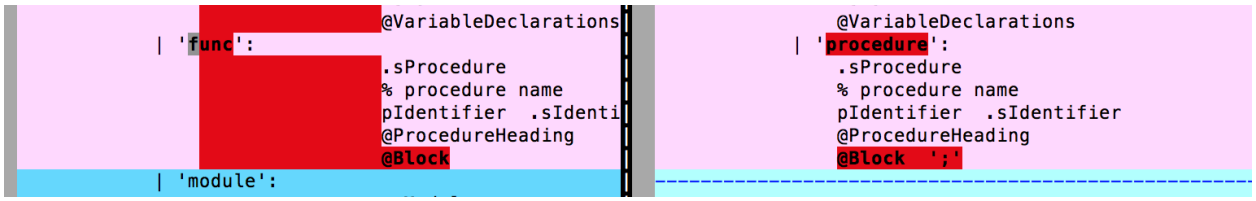


```
Block :
        % Strictly speaking, standard Pascal requires
        % appear only in the order consts, types, var
        % We allow arbitrary intermixing of these ins
        % programmer sanity.
        @Declarations
        @Statements;

Declarations :
        {[
            | 'let':
                                .sConst
                                @ConstantDefinitions
            | 'type':
                                .sType
                                @TypeDefinitions
            | 'var':
                                .sVar
                                @VariableDeclarations
            | 'func':
                                .sProcedure
                                % procedure name
                                pIdentifier  .sIdenti
                                @ProcedureHeading
                                @Block
```

```
Block :
        % Strictly speaking, standa
        % appear only in the order
        % We allow arbitrary interm
        % programmer sanity.
--------------------------------
--------------------------------
--------------------------------
--------------------------------
        {[
            | 'const':
                .sConst
                @ConstantDefinition
            | 'type':
                .sType
                @TypeDefinitions
            | 'var':
                .sVar
                @VariableDeclaratio
            | 'procedure':
                .sProcedure
                % procedure name
                pIdentifier  .sIden
                @ProcedureHeading
                @Block  ';'
```

4.2 - Updating Routines

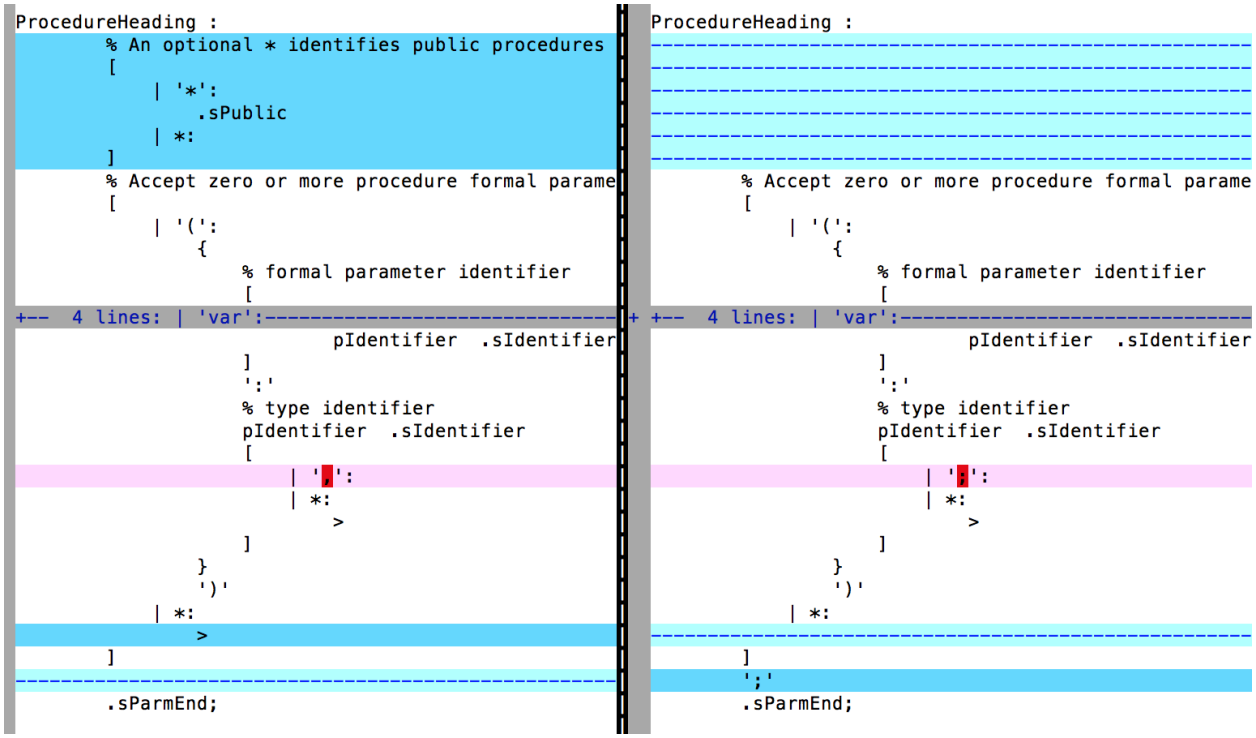
First to update routines we changed the procedure keyword to func. We also removed the required semicolon.



```
                @VariableDeclarations
            | 'func':
                .sProcedure
                % procedure name
                pIdentifier  .sIdenti
                @ProcedureHeading
                @Block
            | 'module':
```

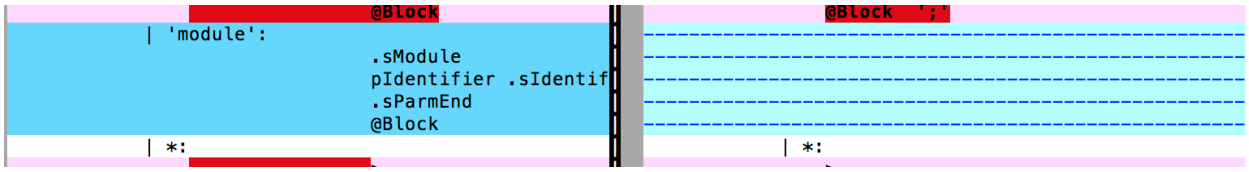
```
                @VariableDeclarations
            | 'procedure':
                .sProcedure
                % procedure name
                pIdentifier  .sIdentifier
                @ProcedureHeading
                @Block  ';'
----------------------------------------------
```

Then we edited the ProcedureHeading rule. We changed the identifier delimiter from a semicolon to a comma, added a choice action to optionally emit a sPublic token if there is a star. Finally we removed the semicolon.



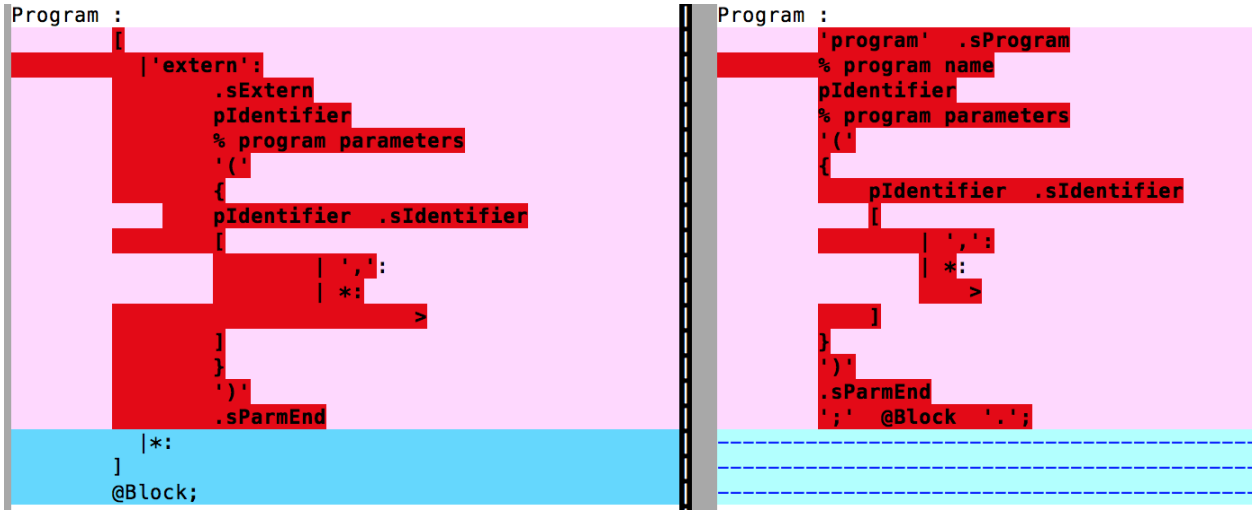
4.3 - Adding Modules

To add modules we added a ‘module’ choice to the Declarations rule. The module simply outputs an sModule token, requires an identifier, outputs sIdentifier and sParmEnd then calls the Block rule that holds the body of the module.



4.4 - Adding Externs

Adding the extern keyword syntax involved removing the required keyword “program” and adding the keyword “extern” and then emitting an sExtern token. Since extern is optional we added a choice action where the default cause did nothing. We took out the required semicolons since they are not necessary in drift. Also, the program no longer ends with a “.”

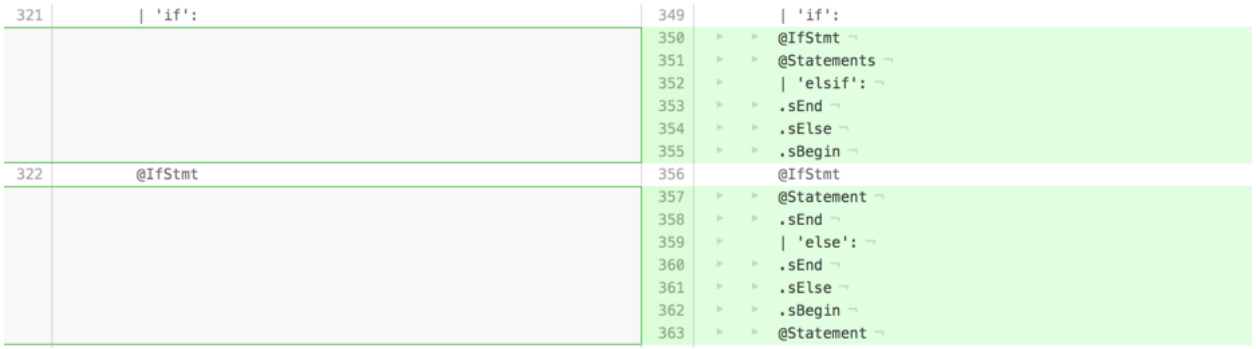


5 - Statements

Statements previously fell into 2 categories: Statements which had no begin/end, statements which had a begin/end. Previously the parser could tell the difference between the two by looking for the begin keyword, but that keyword is removed in drift and the parsing structure needed to change.

5.1 - If/Elsif Statements

In PT Pascal the “if” and “else” blocks contained explicit “begin” and “end” tokens to specify their start/end. In Drift, only “if” has a guaranteed “end”. This required a change in structure.



parser.ssl - Statement if case

If now makes a call to the @Statements rule.

```
Statements :
  {
    @Statement
    [
        | 'end':
            >
    ]
  }
  .sEnd;
```

parser.ssl - Statements rule

This rule collects all statements until it finds the matching else for the if. Previously else was a part of the if rule, but with the new syntax it is made a statement.

```
IfStmt :
    .sIfStmt
    @Expression
    .sExpnEnd
    'then' .sThen
    @Statement
    [
      | 'else':
        .sElse
        @Statement
        | *:
    ];
```

```
IfStmt :
    .sIfStmt
    @Expression
    .sExpnEnd
    .sThen


    .sBegin


    ;
```

```
Statement :
    [
        | pIdentifier:
        @AssignmentOrCallStmt
        @Statement

        | 'if':
        @IfStmt
        @Statements
        | 'elsif':
        .sEnd
        .sElse
        .sBegin
        @IfStmt
        @Statement
        .sEnd
        | 'else':
        .sEnd
        .sElse
        .sBegin
        @Statement
```

parser.ssl - Changes to if syntax

To implement elsif, we opted to convert the token into a nested if statement. When elsif is encountered the following is done:

1) Emit a .sEnd for the “if”
2) Create an “else” block
3) Call the if statement rule to emit an if token and parse the expression

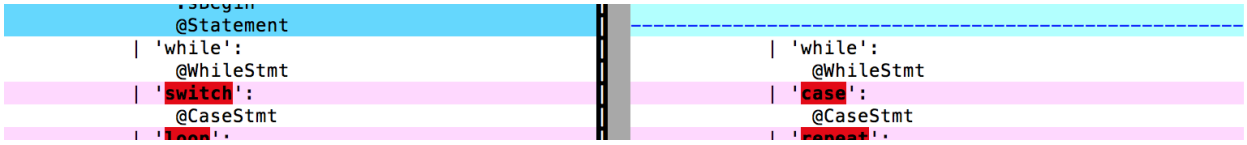
The code for this looks so:

```
| 'elsif': ¬
% Simulates the end of the previous if ¬
% And creates a new if within the else ¬
.sEnd ¬
.sElse ¬
.sBegin ¬
@IfStmt ¬
```

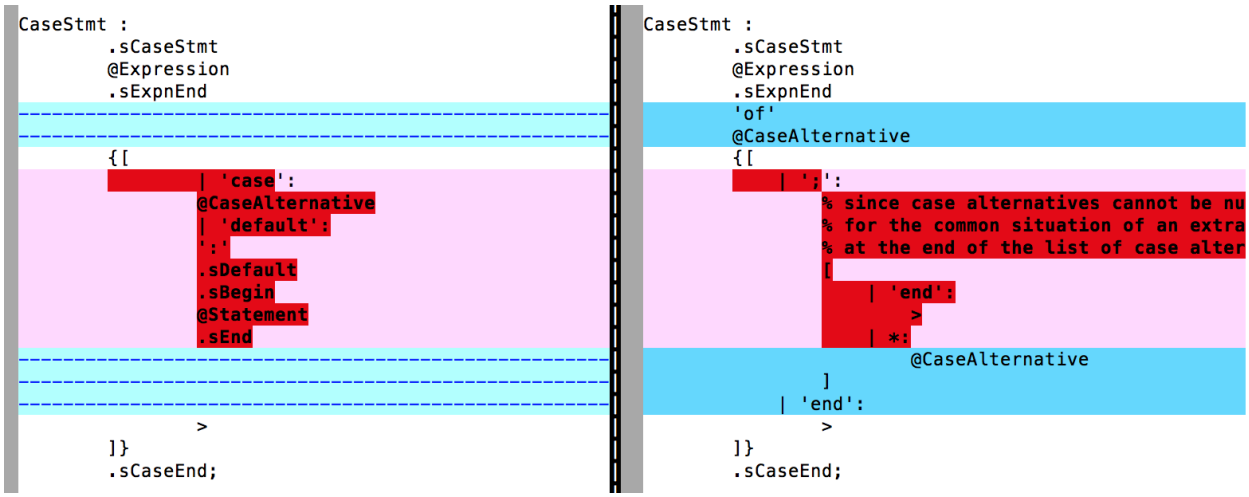
parser.ssl - Elsif implementation

5.2 - Switch Statements

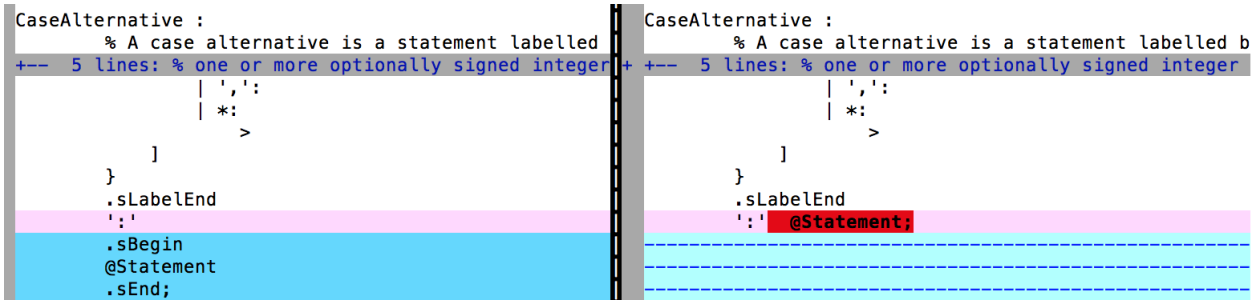
To change case statements to switch statements we first changed the ‘case’ keyword to the ‘switch’ keyword in the Statement rule.



CaseStmt now has two choices. Either a case statemnt or a default case. The default case ends the loop after emitting a statement.

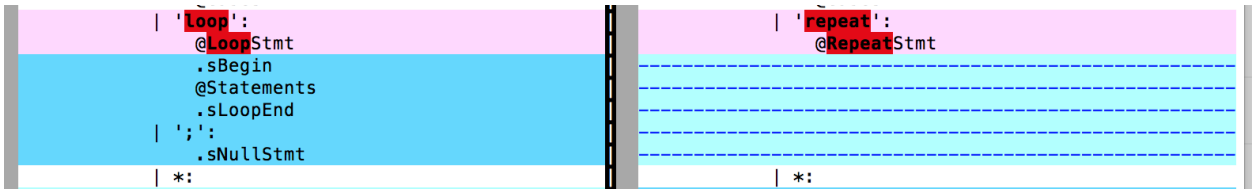


In the CaseAlternative rule we added a sBegin and sEnd before and after the expected Statement.

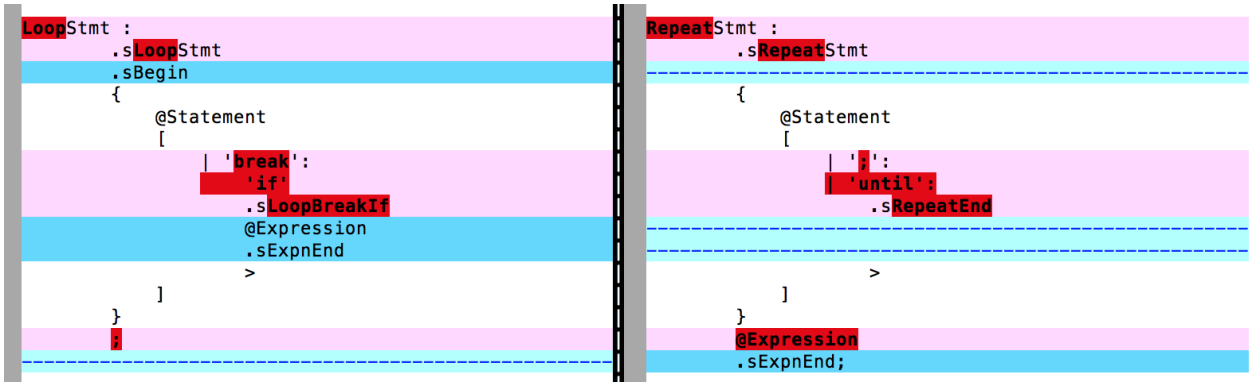


5.3 - Loop Statements

To add loop statements we started by removing the ‘repeat’ keyword from The Statement rule and replacing it with ‘loop’. Sine Drift loops can have breaks in the middle we first call the LoopStmt rule but then make a call to Statements after that and only once Statements is done do we end the loop.



Then we edited the RepeatStmt rule to become the LoopStmt rule. The main difference is that the loop now breaks on ‘break’ instead of until and there is a required expression after the break. It emits a sLoopStmt instead of a sRepeatStmt.



6 - String Type

String function changes:
We first added the # operator in the Factor block of code, as to be in the same precedence level as the not. We then added a second function specifically to add a new precedence level for the substring command between *, div, and mod but below not.

14 parser/parser.ssl View

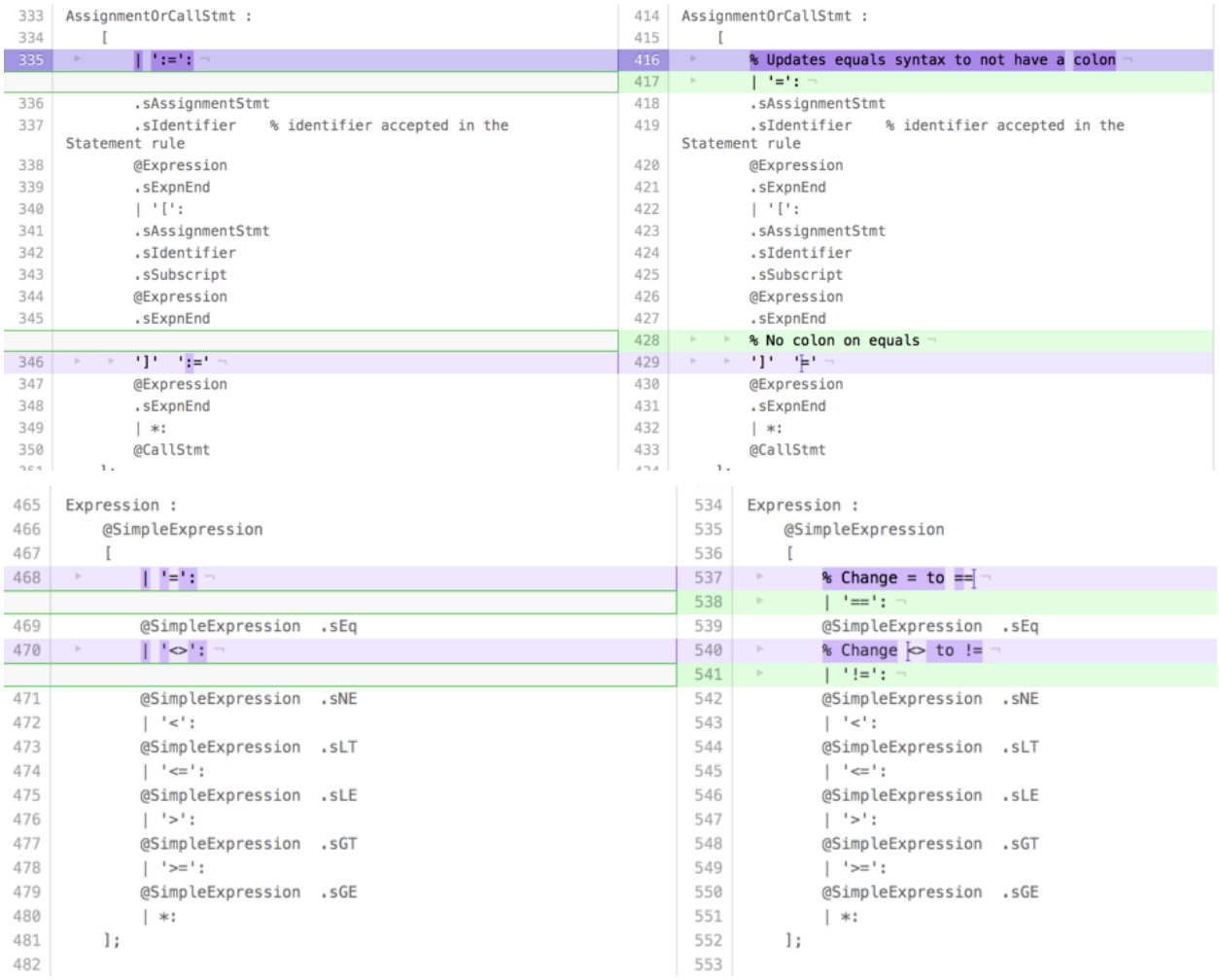
```
@@ -573,6 +573,7 @@ SimpleExpression :

 Term :
        @Factor
+       @Substring
        {[
            | '*':
                @Factor  .sMultiply
@@ -586,6 +587,16 @@ Term :
                >
        ]};

+Substring :
+       [
+           | '::':
+               @Expression
+               '..'
+               @Expression
+               .sSubstring
+           | *:
+       ];
+
 Factor :
        [
            | pIdentifier:
@@ -598,6 +609,9 @@ Factor :
                @Factor
                .sNot
+           | '#':
+               @Expression
+               .sLength
            | pLiteral:
                .sLiteral
            | 'file':
```

7 - Syntax Changes

The remaining changes involved updating the small syntactic differences. The main ones were:

1) Removing semicolons
2) Updating the = syntax to be ==
3) Changing <> to !=
4) Changing := to =

The main places where this occurred were in the expression rule and assignment rule.



parser.ssl - Syntax Changes

8 - Testing

To allow for more efficient tests, the ruby library “rspec” was added to the project. Our test suite is arranged like so:

/ptsrc/rspec - The library for rspec (Path variables added to allow rspec to be a command and include all libraries).
/ptsrc/unit_tests/parser_tests/parser_specs.rb - The rspec file which defines all the test cases.
/ptsrc/unit_tests/parser_tests/*.pt - Example programs to run tests on.
/ptsrc/unit_tests/parser_output_e/* - Expected output tokens for the test programs. Outputs are matched to programs by name.

Most specs work by sending a test file to the parser and comparing the token output stream to the file of expected tokens.

The following are all the tests written for the implemented features.

8.1 - Declarations

8.1.1 - Let, Var, Type Tests

The following files and outputs were used to test let, var and type

Test File	Expected Output
addsLet	
let c = 27	.sConst .sIdentifier .sInteger
addsType	
type t : integer	.sType .sIdentifier .sIdentifier
addsVar	
var v : string var a,b,c : integer	.sVar .sIdentifier .sIdentifier .sVar .sIdentifier .sVar .sIdentifier .sVar .sIdentifier .sIdentifier

These declarations were also used in other tests to further validate that they were parsed correctly.

8.1.2 - Routine Tests

Routines were tested by creating routines with a body and routines that were both public and private.

Test File	Expected Output
updatesRoutine	

func aFunction (var a : string) end	.sProcedure .sIdentifier .sIdentifier .sVar .sIdentifier .sParmEnd .sEnd
publicRoutines	
func aRoutine * (var param:string, var param:integer) var v:integer end	.sProcedure .sIdentifier .sPublic .sIdentifier .sVar .sIdentifier .sIdentifier .sVar .sIdentifier .sParmEnd .sVar .sIdentifier .sIdentifier .sEnd

8.1.3 - Module Tests

The module test verified the declaration of modules as well as declarations and statements within those modules.

Test File	**Expected Output**
addsModule	
module aModule let c = 20 if c == 20 a = 10 end end	.sModule .sIdentifier .sParmEnd .sConst .sIdentifier .sInteger .sIfStmt .sIdentifier .sInteger .sEq .sExpnEnd .sThen

	.sBegin .sAssignmentStmt .sIdentifier .sInteger .sExpnEnd .sEnd .sEnd

8.1.4 - Extern Test

Parsing of externs at the top of the program was done with the following test.

Test File	**Expected Output**
addsExtern	
extern someIdentifier	.sExtern .sIdentifier .sParmEnd

8.2 - Statements

8.2.1 - Switch Tests

The switch test verifies that the multiple cases are parsed along with any underlying statements.

Test File	**Expected Output**
addsSwitch	
switch x case 1: y = 1 case 2: y = 2 default: y = -1 end	.sBegin .sCaseStmt .sIdentifier .sExpnEnd .sInteger .sLabelEnd .sBegin .sAssignmentStmt .sIdentifier .sInteger .sExpnEnd .sEnd .sInteger .sLabelEnd

	.sBegin .sAssignmentStmt .sIdentifier .sInteger .sExpnEnd .sEnd .sDefault .sBegin .sAssignmentStmt .sIdentifier .sInteger .sNegate .sExpnEnd .sEnd .sCaseEnd .sEnd

8.2.2 - If/Elsif Tests

If statements were tested extensively with multiple tests to ensure the correct output for different types of nesting. Our implementation converts elsif into a nested if, so that was another point covered by the tests.

Test File	**Expected Output**
updatesIf	

if x == y
  y = 2
  x=3
else
  x = 2
end
	.sIfStmt
	.sIdentifier
	.sIdentifier
	.sEq
	.sExpnEnd
	.sThen
	.sBegin
	.sAssignmentStmt
	.sIdentifier
	.sInteger
	.sExpnEnd
	.sAssignmentStmt
	.sIdentifier
	.sInteger
	.sExpnEnd
	.sEnd
	.sElse
	.sBegin
	.sAssignmentStmt

	.sIdentifier .sInteger .sExpnEnd .sEnd
simpleIf	
if x == y y=1 end	.sIfStmt .sIdentifier .sIdentifier .sEq .sExpnEnd .sThen .sBegin .sAssignmentStmt .sIdentifier .sInteger .sExpnEnd .sEnd
addsElsif	

if x==1
  y = 2
elsif y==2
  x = 3
else
  z=4
end
	.sIfStmt
	.sIdentifier
	.sInteger
	.sEq
	.sExpnEnd
	.sThen
	.sBegin
	.sAssignmentStmt
	.sIdentifier
	.sInteger
	.sExpnEnd
	.sEnd
	.sElse
	.sBegin
	.sIfStmt
	.sIdentifier
	.sInteger
	.sEq
	.sExpnEnd
	.sThen
	.sBegin
	.sAssignmentStmt
	.sIdentifier
	.sInteger
	.sExpnEnd

	.sEnd .sElse .sBegin .sAssignmentStmt .sIdentifier .sInteger .sExpnEnd .sEnd .sEnd
addsDoubleElsif	
if x==1 y = 2 elsif y==2 x = 3 elsif y==4 y = 1 else z=4 end	.sIfStmt .sIdentifier .sInteger .sEq .sExpnEnd .sThen .sBegin .sAssignmentStmt .sIdentifier .sInteger .sExpnEnd .sEnd .sElse .sBegin .sIfStmt .sIdentifier .sInteger .sEq .sExpnEnd .sThen .sBegin .sAssignmentStmt .sIdentifier .sInteger .sExpnEnd .sEnd .sElse .sBegin .sIfStmt .sIdentifier .sInteger .sEq .sExpnEnd .sThen .sBegin

	.sAssignmentStmt .sIdentifier .sInteger .sExpnEnd .sEnd .sElse .sBegin .sAssignmentStmt .sIdentifier .sInteger .sExpnEnd .sEnd .sEnd .sEnd
addsNestedIf	
if x==1 y = 2 else if z == 2 y = 3 else y = 4 end end	.sIfStmt .sIdentifier .sInteger .sEq .sExpnEnd .sThen .sBegin .sAssignmentStmt .sIdentifier .sInteger .sExpnEnd .sEnd .sElse .sBegin .sIfStmt .sIdentifier .sInteger .sEq .sExpnEnd .sThen .sBegin .sAssignmentStmt .sIdentifier .sInteger .sExpnEnd .sEnd .sElse .sBegin .sAssignmentStmt .sIdentifier

	.sInteger .sExpnEnd .sEnd .sEnd

8.2.3 - Loop Tests

The loop test verifies correct parsing of loops containing a break statement and assignments.

Test File	Expected Output
loop x = 6 y = 7 break if x == y z = 8 a = 3 end	.sLoopStmt .sBegin .sAssignmentStmt .sIdentifier .sInteger .sExpnEnd .sAssignmentStmt .sIdentifier .sInteger .sExpnEnd .sLoopBreakIf .sIdentifier .sIdentifier .sEq .sExpnEnd .sBegin .sAssignmentStmt .sIdentifier .sInteger .sExpnEnd .sAssignmentStmt .sIdentifier .sInteger .sExpnEnd .sEnd .sLoopEnd

8.3 - String Tests

The string test checked the length functionality and substring functionality like so.

Test File	Expected Output

addsString	
#x “hello world” :: 1..2	.sIdentifier .sLength .sLiteral .sInteger .sInteger .sSubstring

8.4 - Syntax Tests

A small test was written to verify that the new syntax for equals and not equals was being parsed correctly.

Test File	**Expected Output**
changesSyntax	
r = x != y r = x == y	.sAssignmentStmt .sIdentifier .sIdentifier .sIdentifier .sNE .sExpnEnd .sAssignmentStmt .sIdentifier .sIdentifier .sIdentifier .sEq .sExpnEnd