

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

26	pNot	'not'	26	pNot	'not'
27	pThen	'then'			
28	pElse	'else'	27	pElse	'else'
29	pOf	'of'	28	pOf	'of'
30	pEnd	'end'	29	pEnd	'end'
31	pUntil	'until'			
32	pDo	'do'			
33	pArray	'array'	30	pArray	'array'
34	pFile	'file'	31	pFile	'file'
35	pPacked	'packed'	32	pPacked	'packed'
36	pProgram	'program'			
37	pConst	'const'			
38	pVar	'var'	33	pVar	'var'
39	pType	'type'	34	pType	'type'
40	pProcedure	'procedure'			
41	pBegin	'begin'			
42	pIf	'if'	35	pIf	'if'
43	pCase	'case'	36	pCase	'case'
44	pWhile	'while'	37	pWhile	'while'
			38	pLet	'let'
			39	pFunc	'func'
			40	pSwitch	'switch'
			41	pDefault	'default'
			42	pElsif	'elsif'
			43	pLoop	'loop'
			44	pBreak	'break'
			45	pModule	'module'
45	pRepeat	'repeat'	46	pExtern	'extern'
46	lastKeywordToken = pRepeat		47	lastKeywordToken = pExtern	
47			48		
48	pIdentifier		49	pIdentifier	
49	firstCompoundInputToken = pIdentifier		50	firstCompoundInputToken = pIdentifier	
50	pInteger		51	pInteger	
51	pLiteral		52	pLiteral	
52	lastCompoundInputToken = pLiteral		53	lastCompoundInputToken = pLiteral	
53			54		
54	pNewLine		55	pNewLine	
55	pEndFile		56	pEndFile	
56	pPlus	'+'	57	pPlus	'+'
57	pMinus	'-'	58	pMinus	'-'
58	pStar	'*'	59	pStar	'*'
59	pColonEquals	':'			
60	pDot	'.'	60	pDot	'.'
61	pComma	','	61	pComma	','
62	pSemicolon	';'	62	pSemicolon	';'
63	pColon	':'	63	pColon	':'
64	pEquals	'='	64	pEquals	'='
65	pNotEqual	'<='	65	pNotEqual	'<='
66	pLess	'<'	66	pLess	'<'
74	pDotDot	'..'	74	pDotDot	'..'
			75	pHash	'#'
			76	pDoubleColon	::
			77	pAssign	'='
75	lastSyntaxToken = pDotDot;		78	lastSyntaxToken = pAssign;	
--			--		

parser.ssl - Input Token Changes

85	▸ sProgram ↗		
86	sParmBegin	88	sParmBegin
87	sParmEnd	89	sParmEnd
88	sConst	90	sConst
89	sType	91	sType
90	sVar	92	sVar
91	sProcedure	93	sProcedure
92	sBegin	94	sBegin
93	sEnd	95	sEnd
94	sNegate	96	sNegate
95	sArray	97	sArray
96	sPacked	98	sPacked
97	sFile	99	sFile
98	sRange	100	sRange
99	sCaseStmt	101	sCaseStmt
100	sCaseEnd	102	sCaseEnd
101	sLabelEnd	103	sLabelEnd
102	sExpnEnd	104	sExpnEnd
103	sNullStmt	105	sNullStmt
104	sAssignmentStmt	106	sAssignmentStmt
105	sSubscript	107	sSubscript
106	sCallStmt	108	sCallStmt
107	sFieldWidth	109	sFieldWidth
108	sIfStmt	110	sIfStmt
109	sThen	111	sThen
110	sElse	112	sElse
111	sWhileStmt	113	sWhileStmt
112	▸ sRepeatStmt ↗		
113	▸ sRepeatEnd ↗		
114	sEq	114	sEq
115	sNE	115	sNE
116	sLT	116	sLT
117	sLE	117	sLE
118	sGT	118	sGT
119	sGE	119	sGE
120	sAdd	120	sAdd
121	sSubtract	121	sSubtract
122	sMultiply	122	sMultiply
123	sDivide	123	sDivide
124	sModulus	124	sModulus
125	sInfixOr	125	sInfixOr
126	sOr	126	sOr
127	sInfixAnd	127	sInfixAnd
128	sAnd	128	sAnd
129	sNot	129	sNot
130	sNewLine	130	sNewLine
		131	▸ sPublic ↗
		132	▸ sDefault ↗
		133	▸ sExtern ↗
		134	▸ sModule ↗
		135	▸ sLoopStmt ↗
		136	▸ sLoopBreakIf ↗
		137	▸ sLoopEnd ↗
		138	▸ sSubstring ↗
		139	▸ sLength ↗
131	▸ lastSemanticToken = sNewLine; ↗	140	▸ lastSemanticToken = sLength; ↗
---		---	

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.

154	Block :	166	Block :
155	% Strictly speaking, standard Pascal requires that declarations	167	% Strictly speaking, standard Pascal requires that declarations
156	% appear only in the order consts, types, vars, procedures.	168	% appear only in the order consts, types, vars, procedures.
157	% We allow arbitrary intermixing of these instead to preserve	169	% We allow arbitrary intermixing of these instead to preserve
158	% programmer sanity.	170	% programmer sanity.
		171	> @Declarations ↵
		172	> @Statements; ↵
		173	↵
		174	Declarations : ↵
159	{[175	{[
160	> 'const': ↵	176	> 'let': ↵
161	> .sConst ↵	177	> > .sConst ↵
162	> @ConstantDefinitions ↵	178	> > @ConstantDefinitions ↵
163	'type':	179	'type':
164	> > .sType ↵	180	> > > .sType ↵
165	> @TypeDefinitions ↵	181	> > > @TypeDefinitions ↵
166	'var':	182	'var':
167	> > .sVar ↵	183	> > > .sVar ↵
168	> @VariableDeclarations ↵	184	> > > @VariableDeclarations ↵
169	> 'procedure': ↵	185	> > 'func': ↵
170	> > .sProcedure ↵	186	> > > .sProcedure ↵
171	> > % procedure name ↵	187	> > > % procedure name ↵
172	> > pIdentifier .sIdentifier ↵	188	> > > pIdentifier .sIdentifier ↵
173	> > @ProcedureHeading ↵	189	> > > @ProcedureHeading ↵
174	> > @Block ';' ↵	190	> > > @Block ↵
		191	> > 'module': ↵
		192	> > > .sModule ↵
		193	> > > > pIdentifier .sIdentifier ↵
		194	> > > > .sParmEnd ↵
		195	> > > > @Block ↵
175	*:	196	*:
176	> > > ↵	197	> > > > ↵
177	> > > } ↵	198	> > > > } ↵
178	> 'begin' ↵		
179	> @BeginStmt; ↵	199	↵
180		200	

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”.

<pre> ProcedureHeading : % An optional * identifies public procedures ['*': .sPublic *:] % Accept zero or more procedure formal param ['(': { % formal parameter identifier [+-- 4 lines: 'var':----- pIdentifier .sIdentifier] ',': % type identifier pIdentifier .sIdentifier [' ': *: >] } ')': *: >] .sParmEnd; </pre>	<pre> ProcedureHeading : % Accept zero or more procedure formal param ['(': { % formal parameter identifier [+-- 4 lines: 'var':----- pIdentifier .sIdentifier] ',': % type identifier pIdentifier .sIdentifier [' ': *: >] } ')': *: >] .sParmEnd; </pre>
--	--

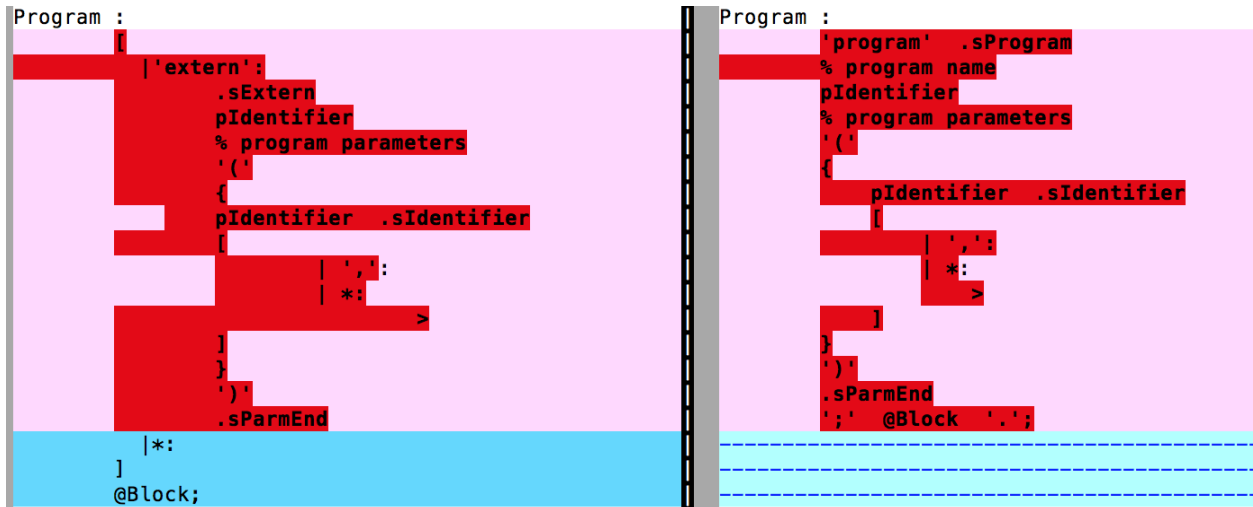
4.3 - Adding Modules

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.

<pre> 'module': .sModule pIdentifier .sIdentif .sParmEnd @Block *: </pre>	<pre> @Block ' '; *: </pre>
---	-------------------------------

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/Elseif 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.

321	'if':	349	'if':
		350	> > @IfStmt ↵
		351	> > @Statements ↵
		352	> > 'elseif': ↵
		353	> > .sEnd ↵
		354	> > .sElse ↵
		355	> > .sBegin ↵
322	@IfStmt	356	@IfStmt
		357	> > @Statement ↵
		358	> > .sEnd ↵
		359	> > 'else': ↵
		360	> > .sEnd ↵
		361	> > .sElse ↵
		362	> > .sBegin ↵
		363	> > @Statement ↵

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.

394	IfStmt :	437	IfStmt :
395	.sIfStmt	438	.sIfStmt
396	@Expression	439	@Expression
397	.sExpnEnd	440	.sExpnEnd
398	> 'then' .sThen	441	> .sThen
399	@Statement		
400	[
401	'else':	442	> .sBegin
402	> .sElse		
403	> @Statement		
404	*:		
405	> !;	443	> ;
406		444	

344	Statement :
345	[
346	pIdentifier:
347	@AssignmentOrCallStmt
348	> > @Statement
349	'if':
350	> > @IfStmt
351	> > @Statements
352	'elsif':
353	> > .sEnd
354	> > .sElse
355	> > .sBegin
356	@IfStmt
357	> > @Statement
358	> > .sEnd
359	'else':
360	> > .sEnd
361	> > .sElse
362	> > .sBegin
363	> > @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 - Elixir 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.

<pre> sBegin @Statement 'while': @WhileStmt 'switch': @CaseStmt 'end': .sEnd </pre>	<pre> 'while': @WhileStmt 'case': @CaseStmt 'end': .sEnd </pre>
---	---

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

<pre> CaseStmt : .sCaseStmt @Expression .sExpnEnd {['case': @CaseAlternative 'default': ': ' .sDefault .sBegin @Statement .sEnd]} > }} .sCaseEnd; </pre>	<pre> CaseStmt : .sCaseStmt @Expression .sExpnEnd 'of' @CaseAlternative {[': ' % since case alternatives cannot be null % for the common situation of an extra % at the end of the list of case alter ['end': > '*: @CaseAlternative]} 'end': > }} .sCaseEnd; </pre>
---	---

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

CaseAlternative :	CaseAlternative :
% A case alternative is a statement labelled	% A case alternative is a statement labelled b
+-- 5 lines: % one or more optionally signed integer +	+-- 5 lines: % one or more optionally signed integer
' , ' :	' , ' :
* :	* :
>	>
}	}
.sLabelEnd	.sLabelEnd
' :	' : @Statement;
.sBegin	
@Statement	
.sEnd;	

5.3 - Loop Statements

To add loop statements we started by removing the 'repeat' keyword from The Statement rule and replacing it with 'loop'. Since 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.

'loop':	'repeat':
@LoopStmt	@RepeatStmt
.sBegin	
@Statements	
.sLoopEnd	
' ; ' :	
.sNullStmt	
* :	* :

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.

LoopStmt :	RepeatStmt :
.sLoopStmt	.sRepeatStmt
.sBegin	
{	{
@Statement	@Statement
[[
'break':	'':
'if':	'until':
.sLoopBreakIf	.sRepeatEnd
@Expression	
.sExpnEnd	
>	>
}	}
	@Expression
.	.sExpnEnd;

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 :
573	573	
574	574	Term :
575	575	@Factor
576	576	+ @Substring
576	577	{
577	578	'':
578	579	@Factor .sMultiply
		@@ -586,6 +587,16 @@ Term :
586	587	>
587	588	}};
588	589	
	590	+Substring :
	591	+ [
	592	+ '':
	593	+ @Expression
	594	+ '..'
	595	+ @Expression
	596	+ .sSubstring
	597	+ *:
	598	+];
	599	+
589	600	Factor :
590	601	[
591	602	pIdentifier:
		@@ -598,6 +609,9 @@ Factor :
598	609	'not':
599	610	@Factor
600	611	.sNot
	612	+ '#':
	613	+ @Expression
	614	.sLength
601	615	pLiteral:
602	616	.sLiteral
603	617	'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.

333	AssignmentOrCallStmt :	414	AssignmentOrCallStmt :
334	[415	[
335	▶ '!=': ~	416	▶ % Updates equals syntax to not have a colon ~
336	.sAssignmentStmt	417	▶ '!=': ~
337	.sIdentifier % identifier accepted in the	418	.sAssignmentStmt
338	Statement rule	419	.sIdentifier % identifier accepted in the
339	@Expression	420	Statement rule
340	.sExpnEnd	421	@Expression
341	'[':	422	.sExpnEnd
342	.sAssignmentStmt	423	'[':
343	.sIdentifier	424	.sAssignmentStmt
344	.sSubscript	425	.sIdentifier
345	@Expression	426	.sSubscript
346	.sExpnEnd	427	@Expression
347	@CallStmt	428	.sExpnEnd
348];	429	▶ % No colon on equals ~
349		430	▶ '!=': ~
350		431	@Expression
351		432	.sExpnEnd
352		433	*:
353		434	@CallStmt
354		435];
465	Expression :	534	Expression :
466	@SimpleExpression	535	@SimpleExpression
467	[536	[
468	▶ '!=': ~	537	▶ % Change = to == ~
469	@SimpleExpression .sEq	538	▶ '==' ~
470	▶ '<=': ~	539	@SimpleExpression .sEq
471	@SimpleExpression .sNE	540	▶ % Change < to != ~
472	'<':	541	▶ '!=': ~
473	@SimpleExpression .sLT	542	@SimpleExpression .sNE
474	'<=':	543	'<':
475	@SimpleExpression .sLE	544	@SimpleExpression .sLT
476	'>':	545	'<=':
477	@SimpleExpression .sGT	546	@SimpleExpression .sLE
478	'>=':	547	'>':
479	@SimpleExpression .sGE	548	@SimpleExpression .sGT
480	*:	549	'>=':
481];	550	@SimpleExpression .sGE
482		551	*:
		552];
		553	

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 somelIdentifier	.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 elseif 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