In this phase you will undertake the modifications to the Parser phase of the PT Pascal compiler to turn it into a Parser for Drift. These changes will be a little more extensive than those of phase 1. The following suggestions are provided to guide you in this phase.

Suggestions for Implementing Phase 2

General Strategy

In general, the strategy will be to have the Drift parser output, as much as possible, the same semantic token stream that would be output by the PT parser for the equivalent features of PT Pascal. (In that way, we will have fewer things to change in the semantic phase.)

Token Definitions

Modify the parser input token list to correspond to the new set of output tokens emitted by your Drift Scanner/Screener. Remove the unused old input tokens and add the new Drift input tokens. Make sure the two sets match exactly - this is the usual source of problems!

Remove the old PT parser output tokens `sProgram`, `sRepeatStmt` and `sRepeatEnd`.

Add the new Drift parser output tokens `sPublic`, `sDefault`, `sExtern`, `sModule`, `sLoopStmt`, `sLoopBreakIf`, `sLoopEnd`, `sSubstring` and `sLength`.

858 teams should additionally add the new Drift 858 parser output token `sReturn`.

Programs

Modify the parsing of programs to meet the Drift language specification. You will have to change the Block rule to be split into two rules, one to accept a sequence of Declarations and a second to accept a sequence of Statements (not a begin statement or a single statement).

In order to make the differences in Drift less visible to the semantic phase, always output a sequence of statements as if it were a PT Pascal begin statement - that is, with an `sBegin` before the statements and an `sEnd` after them.

Declarations

Modify the parsing of constant, type and variable declarations to meet the Drift language spec. The output token streams for these declarations should be the same as for the equivalent PT declarations, in order to minimize the changes we'll have to make to the semantic phase.

For example, the Drift declarations:

```plaintext
let c = 27
var v : string
type t : integer
```

Should yield the parser output token stream:

```plaintext
sConst
sIdentifier <identifier index for 'c'>
sInteger 27
sVar
sIdentifier <identifier index for 'v'>
sIdentifier <identifier index for 'string'>
```

To assist the semantic phase, if there is more than one identifier declared in a single Drift `var` declaration, then each one should be output with an `sVar` token, for example:

```plaintext
var a,b,c : integer
```

Should yield the parser output token stream:

```plaintext
sVar
sIdentifier <identifier index for 'a'>
sVar
sIdentifier <identifier index for 'b'>
sVar
sIdentifier <identifier index for 'c'>
sIdentifier <identifier index for 'integer'>
```

Routines (Procedures)

Modify the parsing of routines (PT procedures, Drift funcs) to meet the Drift language specification. The output token stream for the func should be the same as for the equivalent PT procedure, in order to minimize the changes we'll have to make to the semantic phase. In particular, the procedure's statements should be output preceded by an `sBegin` token and ended by an `sEnd` token as if `begin .. end` were still in the language. If you have already written a Statements rule that does this as suggested above, you can simply call it.

For example, the procedure declaration:

```plaintext
func P
  Declarations
  Statements
end
```

Should yield the parser output token stream:

```plaintext
sProcedure
sIdentifier <identifier index for 'P'>
sParmEnd
sBegin
sIdentifiers 0
sEnd
```

For the Drift `""` that indicates a public procedure, output the `sPublic` semantic token.

Modules

Add parsing of modules as specified in the Drift language specification. The output stream should use the token `sModule` to mark the beginning of the module. The statements part of the module should be preceded by an `sBegin` token and ended by an `sEnd` token as if `begin .. end` were still in the language (i.e., you should once again call your Statements rule to handle them).

For example, the Drift module declaration:

```plaintext
module M
  Declarations
  Statements
end
```
Should yield the parser output token stream:

```
sModule
sIdentifier <identifier index for 'M'>
Declarations
sBegin
Statements
sEnd
```

**Main Programs**

Drift main programs are like the body of a module except that they have the `extern` statement before the declarations, which should be output using the `sExtern` semantic token and ended using the `sParmEnd` semantic token.

For example, the Drift main program:

```
extern output
Declarations
Statements
```

Should yield the parser output token stream:

```
sExtern
sIdentifier <identifier index for 'output'>
sParmEnd
Declarations
sBegin
Statements
sEnd
```

**Statements**

Modify the parsing of `if`, `case`, `while`, `repeat` and `begin` statements to meet the Drift language specification for Drift `if`, `switch`, `while` and `loop` statements. For `if`, `switch` and `while`, the goal is to have the output token stream for the Drift parser be, as much as possible, identical to the output token stream for the corresponding statement in the PT parser. In this way, we will minimize the changes necessary in the semantic phase.

For example, the Drift `if` statement:

```
if x == y
  z = w
  w = 2
else
  z = 1
end
```

Should yield the parser output token stream:

```
sIfStmt
sIdentifier <identifier index for 'x'>
sEq
sExpnEnd
sThen
sAssignmentStmt
sIdentifier <identifier index for 'z'>
sIdentifier <identifier index for 'w'>
sExpnEnd
sAssignmentStmt
sIdentifier <identifier index for 'w'>
sInteger 2
sExpnEnd
sEnd
sElse
sBegin
sAssignmentStmt
sIdentifier <identifier index for 'z'>
sInteger 1
sExpnEnd
sEnd
```

**Statement Sequences**

Drift statement sequences replace the PT Pascal `begin...end` statement, and we can save ourselves a lot of work in the Semantic phase by fooling it into thinking nothing has changed by outputting all statement sequences with `sBegin .. sEnd` around them as shown in this example, by using a Statements rule such as the one suggested above.

Remember that in Drift, there are no semicolons, except the null statement, which consists of a semicolon only. So all semicolons and semicolon separators should be removed from the grammar, and the null statement added as just a semicolon.

**Switch Statements**

The output for Drift `switch` statements should also be the same as the corresponding `case` statements of PT Pascal, using the old `sCase`, `sLabelEnd` and `sCaseEnd` semantic tokens. Even though they look different, except for the addition of the Drift `else` clause, the meaning of the Drift `case` statement and the PT `case` statement is identical - so the semantic token stream can be the same.

A tricky part of this translation is the fact that PT `case` statements take only one statement in each alternative - usually a `begin..end` statement. In Drift, any sequence of statements can appear in each alternative, not just one. So how do we keep the output token stream for case alternatives the same as it was in PT?

The answer is simple: we once again use our Statements rule for statement sequences, which will emit an `sBegin` semantic token at the beginning of the statements in the alternative, and an `sEnd` at the end of them. The resulting output stream looks to the semantic phase as if there were one `begin..end` statement in the alternative, just like in PT.

The Drift `default:` clause on `switch` statements is new, and we must handle it specially. But what we will do is simple - just check for a `default:` following the alternatives in the case statement, and output `sDefault` followed by the statements of the `default:` using the Statements rule to enclose them in `sBegin .. sEnd` as usual before outputting the `sCaseEnd` semantic token.

**Elsif Clauses**

The handling of `elsif` in the Drift `if` statement presents us with a choice. We can either:

(a) use a new semantic token `sElsif` to represent `elsif`, and modify the semantic phase of the compiler to handle it in the next phase, or:

(b) not use any new semantic tokens, and output the token stream corresponding to the equivalent PT Pascal nested `if` statements, so that the semantic phase will not have to be modified to handle `elsif` at all.
If you decide on the first alternative, then you will have to add an `elsif` semantic token and the output token stream for the `if` statement:

```plaintext
if x == 1
  y = 2
elsif z == 2
  y = 3
else
  y = 4
end
```

Will be (note that once again Drift allows a statement sequence in each part, which we bracket in `sBegin..sEnd` in the output using our Statements rule as usual):

```plaintext
sIfStmt
  sIdentifier <identifier index for 'x'>
  sInteger 1
  sEq
  sExpnEnd
  sThen
  sBegin
  sAssignmentStmt
    sIdentifier <identifier index for 'y'>
    sInteger 2
    sExpnEnd
  sEnd
  sElsif
    sIdentifier <identifier index for 'z'>
    sInteger 2
    sEq
    sExpnEnd
    sThen
    sBegin
    sAssignmentStmt
      sIdentifier <identifier index for 'y'>
      sInteger 3
      sExpnEnd
    sEnd
    sElse
    sBegin
    sAssignmentStmt
      sIdentifier <identifier index for 'y'>
      sInteger 4
      sExpnEnd
    sEnd
  sEnd
end
```

However, if you decide on the second alternative, then the parser output token stream for the example should be the same as the output stream for the equivalent nested `if` statement:

```plaintext
if x == 1
  y = 2
else
  if z == 2
    y = 3
  else
    y = 4
end
```

That is to say:

```plaintext
sIfStmt
  sIdentifier <identifier index for 'x'>
  sInteger 1
  sEq
  sExpnEnd
  sThen
  sBegin
  sAssignmentStmt
    sIdentifier <identifier index for 'y'>
    sInteger 2
    sExpnEnd
    sEnd
  sElse
  sBegin
  sIfStmt
    sIdentifier <identifier index for 'z'>
    sInteger 2
    sEq
    sExpnEnd
    sThen
    sBegin
    sAssignmentStmt
      sIdentifier <identifier index for 'y'>
      sInteger 3
      sExpnEnd
      sEnd
    sElse
    sBegin
    sAssignmentStmt
      sIdentifier <identifier index for 'y'>
      sInteger 4
      sExpnEnd
    sEnd
  sEnd
end
```

If you choose this way, you won’t have to implement `elsif` in the semantic phase at all, because it will never see it. This is typical of decisions made by compiler writers - many language features can be implemented either in one phase or in the next. In this case, we can either implement `elsif` in the parser (this phase) or in the semantic analyzer (next phase).

Neither decision is strictly the right one, and neither is wrong. The amount of work to implement the feature is about the same either way. It will be up to you to decide which you want to do, but whichever decision you make, make it clear to your TA when you hand in your phases!

**Loop Statements**

Remove handling of the PT `repeat` statement, and add handling of the Drift `loop` statement. The output stream should use the tokens `sLoop` and `sEndLoop` to mark the beginning and end of the loop, and the `sLoopBreakIf` token for `break if` clauses. The end of the conditional expression following a `break if` should be marked with the `sExpnEnd` token. As usual, use your Statements rule to handle the statement sequences before and after the exit, which will output `sBegin..sEnd` tokens around the sequences.
For example, the (silly) loop:

```pascal
loop
  Statements1
break if true
  Statements2
end
```

should yield the parser output token stream:

```token
sLoop
sBegin
  Statements1
sEnd
sLoopBreakIf
sIdentifier <predefined identifier index for 'true'>
  sExpnEnd
sBegin
  Statements2
sEnd
sEndLoop
```

Note that we can't pull the same trick of making the Semantic phase think we still have PT for the `loop` statement - there is no PT `while` or `repeat` statement equivalent to an Drift `loop`, so we'll just have to leave it until the Semantic phase.

### The string Type

Remove handling of the old PT `char` data type and `char` literals, and add handling of the `string` data type and `string` literals. Add handling of the new Drift operators `::` and `#`. The precedence of `::` operator (including the `..` portion) should be higher (more tightly binding) than `*`, `div` and `mod` and lower than `not`, and the precedence of `#` is the same as the precedence of `not`.

(Hint: Adding `#` to be the same precedence as `not` is straightforward. Adding `::` involves introducing a new precedence level.)

Both of the new operators should be converted to postfix by your parser, using the postfix operator output tokens `sLength` and `sSubstring`.

The operator `::` is somewhat unusual in that it takes three operands, but this does not affect the form of the postfix output, which should consist of the three operands followed by the operator.

For example, the substring operation:

```
"Hi there" :: 1..2
```

Should yield the parser output stream:

```token
sLiteral  "Hi there"
  sInteger  1
  sInteger  2
  sSubstring
```

### Other Syntactic Details

Watch out for other minor syntactic differences between PT Pascal and Drift - for example, the separator between formal parameters is semicolon in PT whereas it is comma in Drift, the equality operator is `=` in PT but `==` in Drift, and the inequality operator is `<>` in PT but `!=` in Drift.

### Functions (858 only)

Adding Drift functions involves three main changes: handling the return type following the parameters, handling the return statement at the end of the function, and changing the grammar of function calls to allow for multiple actual parameters.

The output stream for the function declaration should append the type identifier for the return type following the output tokens for the parameter list. The `return` statement should be output following the `sEnd` semantic token for the end of the statements in the routine, and should begin with the new `sReturn` token followed by the result expression ended with an `sEndExpn` token.

For example, the function:

```pascal
func f (i : string) : integer
  Declarations
  Statements
  return Expn
end
```

Should yield the parser output token stream:

```token
sFunction
  sIdentifier <identifier index for f>
  sIdentifier <identifier index for i>
  sIdentifier <predefined identifier index for string>
  sParmEnd
  sIdentifier <predefined identifier index for integer>
  sIdentifiers
  sBegin
  sEnd
  sReturn
  sExpn
  sEndExpn
```

The syntax for function call actual parameter lists in expressions should be changed to allow multiple parameters, using `sParmEnd` to mark the end of the actual parameter list in the same way as for call statement parameters.

For example, the assignment:

```
x  =  f (z, w) + 1
```

Should yield the parser output token stream:

```token
sAssignmentStmt
  sIdentifier <identifier index for x>
  sIdentifier <identifier index for f>
  sParmBegin
  sIdentifier <identifier index for z>
  sExpnEnd
  sIdentifier <identifier index for w>
  sExpnEnd
  sParmEnd
  sInteger  1
  sAdd
  sExpnEnd
```