incremental
↳ do documentation in phase
↳ do testing incrementally

CISC/CMPE 458 Course Project Winter 2023
Phase 2. Parser
K. Jahed – February 2023

In this phase you will undertake the modifications to the Parser phase of the PT Pascal compiler to turn it into a Parser for Quby. 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 Quby 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
1 Modify the parser input token list to correspond to the new set of output tokens emitted by your Quby Scanner/Screener. Remove the unused old input tokens and add the new Quby input tokens. Make sure the two sets match exactly - this is the usual source of problems!
Remove the old PT parser output tokens *sRepeatStmt* and *sRepeatEnd* and add the new Quby parser output tokens *sPublic*, *sModule*, *sDoStmt*, *sBreakIf*, *sSubstring*, *sLength* and *sIndex*. 2

Programs
Modify the parsing of programs (the Program rule) to meet the Quby language specification. Quby main programs are different from PT, in that Quby makes no distinction between declarations and statements. So you 3 will have to change the main loop of the Block rule to accept a sequence of any number of declarations or statements in any order. I suggest that you merge the alternatives in the existing Statement rule into the alternatives in the Block rule to do this.
In order to make the differences in Quby less visible to the semantic phase, we will always output a Quby sequence of declarations and statements as if it were a PT Pascal **begin**...**end** statement 4 - that is, emit an *sBegin* token before the sequence (i.e., at the beginning of the Block rule) and and *sEnd* after it (at the end of the rule). In this way, it will look to the semantic phase like not much about main programs has changed.

sequence of declarations
only in code

For example, the Quby main program:

```
    using output
    DeclarationsAndStatements
```

Should yield the parser output token stream:

```
    sProgram
    sIdentifier     <identifier index for 'output'>
    sParmEnd
    sBegin
    DeclarationsAndStatements
    sEnd
```

Exactly like the equivalent PT main program.

Declarations
5 Modify the parsing of constant, type and variable declarations to meet the Quby 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 Quby declarations:

```
    val c = 27
    var v : string
    type t : integer
```

Should yield the parser output token stream:

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

```
     sIdentifier    <identifier index for 't'>
     sIdentifier    <identifier index for 'integer'>
```

5b

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

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

Should yield the parser output token stream:

```
     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)

6

Modify the parsing of routines (PT procedures, Quby functions) to meet the Quby language specification. The output token stream for the function 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 – that's it, you can re-use your Block rule. For example, the procedure declaration:

block rule reuse

```
     def P ()
         DeclarationsAndStatements
     end
```

Should yield the parser output token stream:

```
     sProcedure
     sIdentifier     <identifier index for 'P'>
     sParmEnd
     sBegin
     DeclarationsAndStatements
     sEnd
```

6b

For the Quby * that indicates a public function, output the *sPublic* semantic token, but following the function's identifier, for example, this Quby empty public function:

```
     def * P () end
```

Should yield the parser output token stream:

```
     sProcedure
     sIdentifier     <identifier index for 'P'>
     sPublic
     sParmEnd
     sBegin
     sEnd
```

Modules

block rule reuse

7

Add parsing of modules as specified in the Quby 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. You can re-use your Block rule, once again!

For example, the Quby module declaration:

```
     module M
         DeclarationsAndStatements
     end
```

Should yield the parser output token stream:

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

Statements

8

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

For example, the Quby if statement:

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

Should yield the parser output token stream:

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

conditional

block if true

Unless Statements

9

Quby **unless** statements are simply syntactic sugar for "**if not**". If we simply output the parser output token stream for "**unless** Expression **then**" as "**if not** Expression **then**", then we won't have to implement **unless** in the semantic phase at all.

For example, the Quby **unless** statement:

```
unless x == y then
    z = w
end
```

Should yield the parser output token stream:

```
sIfStmt
sIdentifier          <identifier index for 'x'>
sIdentifier          <identifier index for 'y'>
sEq
sNot
sExpnEnd
sThen
sBegin
sAssignmentStmt
sIdentifier          <identifier index for 'z'>
sIdentifier          <identifier index for 'w'>
sExpnEnd
sEnd
```

condition

→ not

→ end expression

Case Statements

10

The output for Quby **case** 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 Quby **else** clause, the meaning of the Quby **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 Quby, any sequence of declarations and statements can appear in each alternative, not just one statement. So how do we keep the output token stream for case alternatives the same as it was in PT? The answer is simple: re-use your Block rule, which will once again emit an *sBegin* semantic token at the beginning of an alternative, and an *sEnd* at the end. The resulting output stream looks to the semantic phase as if there were one **begin** .. **end** statement in the alternative, just like in PT.

→ block rule reuse

105

The Quby **else** clause on **case** statements is new, and we must handle it specially. But what we will do is simple - just check for an **else** following the alternatives in the case statement, and output *sElse* followed by the statements of the **else** clause (enclosing the body of the else clause with *sBegin* .. *sEnd* as usual) before outputting the *sCaseEnd* semantic token.

For example, the Quby case statement:

```
case i
    when 42 then
        DeclarationsAndStatements1
    when 43 then
        DeclarationsAndStatements2
    else
        DeclarationsAndStatements3
end
```

Should yield the parser output token stream:

```
sCaseStmt
sIdentifier     <identifier index for i>
sExpnEnd

sInteger        42
sBegin
DeclarationsAndStatements1
sEnd

sInteger        43
sBegin
DeclarationsAndStatements2
sEnd

sElse
sBegin
DeclarationsAndStatements3
sEnd
sCaseEnd
```

<u>Elsif Clauses</u>

11

The handling of **elsif** in the Quby **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 *sElsIf* semantic token and the output token stream for the **if** statement:

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

Will be:

```
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
```

x==1

emitted on elsif read

elsif conditional

elsif block

else condition

```
    sIdentifier      <identifier index for 'y'>
    sInteger         4
    sExpnEnd
    sEnd
```

elsif block

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:

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

That is to say:

```
    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
```

when elsif emit this

elsif conditional

ending the elsif

```
    sIdentifier      <identifier index for 'y'>
    sInteger         3
    sExpnEnd
    sEnd
    sElse
    sBegin
    sAssignmentStmt
    sIdentifier      <identifier index for 'y'>
    sInteger         4
    sExpnEnd
    sEnd
    sEnd
```

the else block

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!

12

<u>Do Statements</u>

Remove handling of the PT **repeat** statement, and add handling of the Quby **do** statement. The output stream should use the token *sDo* to mark the beginning and end of the statement, and the *sBreakIf* token for **break if** clauses. The end of the conditional expression following a **break if** should be marked with the *sExpnEnd* token. As always, you can use the Block rule to output an *sBegin* .. *sEnd* tokens around declarations and statements.

For example, the (silly example) **do** loop:

```
    do
        DeclarationsAndStatements1
        break if true
        DeclarationsAndStatements2
    end
```

adds new semantic tokens

sDo sBreakif

break if conditional

should yield the parser output token stream:

tag as do block

```
    sDo
    sBegin
    DeclarationsAndStatements1
    sEnd
    sBreakIf
    sIdentifier <predefined identifier index for 'true'>
    sExpnEnd
    sBegin
    DeclarationsAndStatements2
    sEnd
```

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

The string Type

13a 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 Quby operators $, ? and #. The precedence of of the $ operator (including the .. portion) should be higher (more tightly binding) than *, **div** and **mod** and lower than **not**, the precedence of the ? operator is the same as that of *, **div** and **mod**, and the precedence of # is the same as the precedence of **not**.

Hint: Adding ? to be the same precedence as *, **div** and **mod**, and # to be the same precedence as **not**, is straight forward. Adding $ involves introducing a new precedence level.

13b All three of the new operators should be converted to postfix by your parser, using the postfix operator output tokens *sSubstring*, *sIndex* and *sLength*.

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:

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

14 Other Syntactic Details

Watch out for other minor syntactic differences between PT Pascal and Quby - for example, the assignment operator is := in PT but = in Quby, the equality operator is = in PT but == in Quby, the inequality operator is <> in PT but != in Quby, and the not operator is not in PT but ! in Quby.

→ precedence:

→ not #

→ $ ..

→ * div mod ?

add # and ? first, test then introduce the new precedence level

14 make syntactic changes for updates on the operators