CISC / CMPE 458 Course Project Winter 2020

Phase 2. Parser (CORRECTED)

(Due 4:30pm Wednesday February 26)

J.R. Cordy – February 2020

In this phase you will undertake the modifications to the Parser phase of the PT Pascal compiler to turn it into a Parser for Qust. These changes will be much more extensive than those in phase 1. The following suggestions are provided to guide you in this phase, but remember - the goal is to implement a parser for Qust as described in the Qust language specification.

Suggestions for Implementing Phase 2

Token Definitions

Modify the parser input token list in *parser.ssl* to correspond to the new set of output tokens emitted by your Qust Scanner/Screener. Remove the unused old parser input tokens and add the new Qust input tokens. Make sure the two token sets (*scan.ssl* output tokens, *parser.ssl* input tokens) match exactly - this is the usual source of problems! Don't forget to add the string synonyms (e.g. *pElse 'else', pPlusEquals '+='*) for all the new tokens of Qust.

Replace the old PT parser repeat statement output tokens *sRepeatStmt* and *sRepeatEnd* with the new Qust loop statement tokens *sLoopStmt, sLoopBreakIf* and *sLoopEnd.* Add new parser output tokens for Qust modules, *sModule* and *sPublic*, for the Qust default case alternative *sCaseOtherwise*, and for the Qust string operations *sSubstring* and *sLength*. Finally, add new *sMutable* and *sInitialValue* output tokens for Qust variable declarations.

Programs

Modify the parsing of programs to meet the Qust language specification. Qust main programs are different from PT, in that Qust makes no distinction between declarations and statements. So you 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 after the alternatives in the *Block* rule to do this.

In order to make the differences in Qust less visible to the semantic phase, we will always output a Qust sequence of declarations and statements as if it were a PT Pascal **begin**...**end** statement - that is, emit an *sBegin* token before the sequence (i.e., at the beginning of the *Block* rule) and an *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.

For example, the Qust main program :

```
mod main (output) {
  DeclarationsAndStatements
}
```

Should yield the parser output token stream :

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

Declarations

Modify the parsing of constant, type and variable declarations to meet the Qust language specification in the document *"CISC/CMPE 458 Course Project Winter 2020: The Qust Programming Language"*. The output token streams for these declarations should be similar to the equivalent PT declarations, in order to minimize the changes we'll have to make to the semantic phase.

For example, the Qust declarations :

```
const c = 27;
type t = int;
let v : int;
```

Should yield the parser output token stream :

```
sConst
sIdentifier  <identifier index for 'c'>
sInteger 27
sType
sIdentifier  <identifier index for 't'>
sIdentifier  <identifier index for 'int'>
sVar
sIdentifier  <identifier index for 'v'>
sIdentifier  <identifier index for 'int'>
```

Although Qust uses **let** for variable declarations, we continue to use PT's *sVar* semantic token to minimize changes to the Semantic phase. Remember that while PT Pascal allows multiple declarations in each **type** (e.g., **type** t=integer; u=char;), Qust allows only one (e.g., **type** t=int; **type** u=str;). Also notice that while Qust still allows multiple declarations in each **const** and **let**, it separates them using commas, not semicolons (e.g., PT's **const** a = 1; b = 2; becomes Qust's **const** a=1, b=2;).

Qust **let** declarations differ from PT's **var** declarations in two ways. First, Qust variables can be either *mutable* (i.e., assignable, declared using **mut**) or immutable (i.e. assigned only once using an initial value). To mark mutable **let** declarations, we'll use the new *sMutable* semantic token, emitted at the end of the declaration, for example:

```
let mut v : int;
```

Should yield the output token stream:

```
sVar
sIdentifier   <identifier index for 'v'>
sIdentifier   <identifier index for 'int'>
sMutable
```

Second, Qust optionally allows expressions as initial values for variables, for example:

```
let n : int = 64;
```

To denote this, we'll add the new *sInitialValue* semantic token to mark the beginning of the initial value expression, and we'll use the existing *sExpnEnd* token to mark the end, for example:

```
sVar
sIdentifier   <identifier index for 'n'>
sIdentifier   <identifier index for 'int'>
sInitialValue
sInteger 64
sExpnEnd
```

Types

Modify the parsing of types to conform to Qust syntax. Unlike PT, array subscript ranges in Qust always begin with 1 and end with an upper bound, rather than specifying both a lower and and upper bound using a subrange type. PT subrange types (e.g., **var** x: 10 **..** 20;) are removed from Qust, and simple types are always just a type identifier. Thus array types should be output with just the element type and the upper bound. For example, the Qust array declaration :

```
let mut a: [int:10];
```

Should yield the parser output token stream :

```
sVar
sIdentifier  <identifier index for 'a'>
sArray
sIdentifier  <identifier index for 'int'>
sRange
sInteger 10
sMutable
```

Routines

Modify the parsing of routines (PT procedures, Qust functions) to meet the Qust language specification. (In particular, note that Qust function declarations require ( ) for parameters even if there are none, and use { } around the body of the function.)

As much as possible, the output token stream for a Qust function should be the same as for the equivalent PT procedure, in order to minimize the changes to the semantic phase. In particular, you should use your modified *Block* rule as suggested above so that the declarations and statements in the body of the function are output surrounded by *sBegin* and *sEnd*.

For example, the Qust function declaration :

```
fn P() {
  DeclarationsAndStatements
}
```

Should yield the parser output token stream :

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

For the Qust **pub** modifier that indicates a public function, output the *sPublic* semantic token, but following the procedure's identifier, for example, the Qust public routine:

```
pub fn P() {
  DeclarationsAndStatements
}
```

Should yield the parser output token stream :

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

Modules

Add parsing of modules as specified in the Qust language specification. A module is like a Qust whole program, but without the program parameters. The output stream should use the token *sModule* to mark the beginning of the module. The body of the module should be preceded by an *sBegin* token and ended by an *sEnd* token(i.e., you should once again use your modified *Block* rule to handle the body of the module).

For example, the Qust module declaration :

```
mod M {
  DeclarationsAndStatements
}
```

Should yield the parser output token stream :

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

Statement Sequences

PT Pascal normally allows only single statements in control statements, and the special **begin**...**end** statement (kind of like {...} in C) allows for sequences of statements. While **begin**...**end** is removed from Qust, we can save ourselves a lot of work in the Semantic phase by outputting all declaration-or-statement sequences in Qust with *sBegin .. sEnd* semantic tokens around them, so that it still thinks it's handling PT. We can do this simply by calling the *Block* rule suggested above wherever *DeclarationsAndStatements* appears in the Qust syntax.

Statements

Modify the parsing of **if**, **while**, **repeat** and **case** statements to instead meet the Qust language specification for Qust **if, while, loop** and **match** statements. Remove the PT begin...end statement. The goal is to have the output token stream for the Qust parser be, as much as possible, the same as 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 Qust **if** statement :

```
if x == y {
  DeclarationsAndStatements1
} else {
  DeclarationsAndStatements2
}
```

Should yield the parser output token stream :

```
sIfStmt
sIdentifier  <identifier index for 'x'>
sIdentifier  <identifier index for 'y'>
sEq
sExpnEnd
sThen
sBegin
  DeclarationsAndStatements1
sEnd
sElse
sBegin
  DeclarationsAndStatements2
sEnd
```

Else-If Clauses

The handling of **else if** in the Qust **if** statement presents us with a choice. We can either :

(a) use a new semantic token *sElseIf* to represent **else if**, 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 **else if** at all.

The first alternative would add a new *sElseIf* semantic token to the output token stream for the **if** statement, and handle it in the semantic phase. If instead we choose the second alternative, the parser output token stream for an **if** statement with an **else if**, such as this one:

```
if x == 42 {
   DeclarationsAndStatements1
} else if y == 71 {
   DeclarationsAndStatements2
} else {
   DeclarationsAndStatements3
}
```

Should be the same as the output stream for the equivalent nested **if** statement :

```
if x == 42 {
   DeclarationsAndStatements1
} else {
   if y == 71 {
      DeclarationsAndStatements2
   } else {
      DeclarationsAndStatements3
   }
}
```

That is to say :

```
sIfStmt
sIdentifier  <identifier index for 'x'>
sInteger  42
sEq
sExpnEnd
sThen
sBegin
  DeclarationsAndStatements1
sEnd
sElse
sBegin
  sIfStmt
  sIdentifier  <identifier index for 'y'>
  sInteger  71
  sEq
  sExpnEnd
  sThen
  sBegin
    DeclarationsAndStatements2
  sEnd
  sElse
  sBegin
    DeclarationsAndStatements2
  sEnd
sEnd
```

This way, you won't have to implement **else if** 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 **else if** 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 is 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!

Match Statements

The output for Qust **match** statements should be the same as the corresponding **case** statements of PT Pascal, using the old PT *sCase* and *sCaseEnd* semantic tokens. The meaning of the Qust **match** statement and the PT **case** statement is identical, except for the Qust *default* alternative - 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 Qust, any sequence of declarations-and-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 *Block* rule, which will emit an *sBegin* semantic token at the beginning of the declarations-and-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 Qust *default* alternative (| _ **=>**) in **match** statements is new, and we must handle it specially. But what we will do is simple - just check for | _ **=>** following the other alternatives in the case statement, and output *sCaseOtherwise* followed by the statements of the | _ **=>** clause, once again using the *Block* rule to enclose them in *sBegin .. sEnd,* before outputting the *sCaseEnd* semantic token.

For example, the Qust **match** statement:

```
match i {
  | 42 | 43 => {
      DeclarationsAndStatements1
    }
  | 44 => {
      DeclarationsAndStatements2
    }
  | _ => {
      DeclarationsAndStatements3
    }
}
```

Should yield the parser output token stream :

```
sCaseStmt
sIdentifier  <identifier index for i>
sExpnEnd
sInteger 42
sInteger 43
sLabelEnd
sBegin
  DeclarationsAndStatements1
sEnd
sInteger 44
sLabelEnd
```

```
sBegin
  DeclarationsAndStatements2
sEnd
sCaseOtherwise
sBegin
  DeclarationsAndStatements3
sEnd
sCaseEnd
```

Loop Statements

Remove handling of the PT **repeat** statements, and add handling of the Qust **loop** statement. The output stream should use the tokens *sLoopStmt* and *sLoopEnd* to mark the beginning and end of the statement, and the *sLoopBreakIf* token for **break if** *expression* clauses. The end of the conditional expression following **break if** should be marked with the *sExpnEnd* token, just as it is in the PT Pascal **while** statement. Use your *Block* rule to handle the declaration—or-statement sequences before and after the **break if**, which will automatically output *sBegin* .. *sEnd* tokens around the sequences.

For example, the Qust loop:

```
loop {
   DeclarationsAndStatements1
   break if Expression;
   DeclarationsAndStatements2
}
```

Should yield the parser output token stream :

```
sLoopStmt
sBegin
  DeclarationsAndStatements1
sEnd
sLoopBreakIf
  Expression
sExpnEnd
sBegin
  DeclarationsAndStatements1
sEnd
```

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 a Qust **loop**, so we'll just have to leave it until the Semantic phase.

Short Form Assignments

Add the parsing of Qust short form assignment statements **+=** and **–=**. This is another case where we can save ourselves work in the Semantic phase by simply outputting the semantic token stream for a regular assignment, so that the Semantic phase doesn't have to handle iterative assignments at all.

For example, for the iterative assignment `i += 10`, we will output the semantic token stream for the equivalent regular assignment `i = i + 10`, that is:

```
sAssignmentStmt
sIdentifier  <identifier index for 'i'>
sIdentifier  <identifier index for 'i'>
sInteger  10
sAdd
sExpnEnd
```

The String Type

Remove handling of the old PT ***char*** data type and ***char*** literals, and add handling of the ***str*** data type and ***str*** literals. Add handling of the new Qust string operators *substring* (s **/** *expression* **:** *expression*) and *length* (**?** s).

The precedence of the substring operator **/** *expression* **:** *expression* is the same as integer /, and the precedence of the length operator **?** is the same as the precedence of boolean not (!). Both of the new operators should be converted to postfix by your parser, using the postfix operator output tokens *sSubstring* and *sLength.*

The */ expression : expression* operator is somewhat unusual, in that it takes three operands (the string and two integer expressions), 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
```

Operator Syntax

Change the syntax of the PT operators to the corresponding Qust operators described in the Qust language specification. For example, PT **and, or,** and **not** are &&, ll and ! in Qust, PT **div** and **mod** are / and % in Qust, and PT :=, =, and <> are =, ==, and != respectively in Qust.

Other Syntactic Details

Watch out for other minor syntactic differences between PT Pascal and Qust - for example, semicolon is a separator between statements in PT, but is part of some statements (including the null statement) in Qust, and Qust requires empty parameter and argument lists ( ) for function declarations and calls when there are no parameters.

Whenever you have any questions about what is allowed or not allowed in Qust, refer to the Qust language specification in the document *"CISC/CMPE 458 Course Project Winter 2020: The Qust Programming Language"*. Any forms not explicitly defined there retain their original PT Pascal form and meaning.

No modifications, extensions or changes to the Qust language are allowed in your compiler; you must implement the language exactly as specified. If you have doubts about what is intended, post a question on the course forum for an interpretation.