CISC / CMPE 458 Course Project Winter 2019

Phase 2. Parser

(Due 4:30pm Wednesday February 27)

J.R. Cordy – February 2019

In this phase you will undertake the modifications to the Parser phase of the PT Pascal compiler to turn it into a Parser for MiniT. 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

Token Definitions

Modify the parser input token list in *parser.ssl* to correspond to the new set of output tokens emitted by your MiniT Scanner/Screener. Remove the unused old parser input tokens and add the new MiniT 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!

Remove the old parser output token *sPacked*, which is not in MiniT. Replace the old PT parser loop statement output tokens *sWhileStmt*, *sRepeatStmt* and *sRepeatEnd* with the new MiniT loop statement tokens *sLoopStmt* and *sLoopExitWhen*. Add new parser output tokens for MiniT modules, *sModule* and *sPublic*, for the MiniT default case alternative *sCaseOtherwise*, and for the MiniT string operations *sSubstring* and *sLength*. Finally, add new *sIncrement* and *sDecrement* output tokens for the MiniT ++ and -- statements.

Programs

Modify the parsing of programs to meet the MiniT language specification. MiniT main programs are very similar to PT, but you will have to change the *Block* rule to accept a sequence of any number of statements (not just a single *begin* statement) after the declarations. I suggest that you change the existing *Statement* rule to *Statements* for this purpose, and modify it to accept a sequence of any number of MiniT statements.

In order to make the differences in MiniT less visible to the semantic phase, always output a MiniT sequence of statements as if it were a PT Pascal *begin* statement - that is, emit an *sBegin* before the sequence of statements and an *sEnd* after them. In this way, it will look to the semantic phase like nothing about main programs has changed.

For example, the MiniT main program :

```
external output
Declarations
Statements
```

Should yield the parser output token stream :

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

Declarations

Modify the parsing of constant, type and variable declarations to meet the MiniT language specification in the document *"CISC/CMPE 458 Course Project Winter 2019: The MiniT Programming Language"*. 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 MiniT declarations :

```
const 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'>
```

Remember that while PT Pascal allows for multiple declarations in each **const**, **var** or **type** (e.g., **var** x: integer; y: char; **const** a=1; b=2; **type** t=integer; u=char;), MiniT allows only one (e.g., **var** x: integer **var** y: char **const** a := 1 **const** b := 2 **type** t: integer **type** u: char)

MiniT does however allow multiple identifiers of the same type in a **var** decaration (e.g., **var** a,b: integer), which PT does not. To assist the semantic phase, if there is more than one identifier in a single MiniT **var** declaration, 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)

Modify the parsing of routines (PT procedures, MiniT methods) to meet the MiniT language specification. The output token stream for a MiniT method should be the same as for the equivalent PT procedure, in order to minimize the changes 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 modified the *Block* rule to use a *Statements* rule to do this as suggested above, you can simply call it.

For example, the MiniT procedure declaration :

```
procedure P
     Declarations
     Statements
end procedure
```

Should yield the parser output token stream :

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

For the MiniT ***** modifier that indicates an exported (public) procedure, output the *sPublic* semantic token, but following the procedure's identifier, for example, the MiniT public routine:

```
procedure * P
  Statements
end procedure
```

Should yield the parser output token stream :

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

Modules

Add parsing of modules as specified in the MiniT language specification. A module is like an embedded whole program, but without the program parameters (i.e., without an *external* statement). 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 modified *Block* rule that uses *Statements* to handle them).

For example, the MiniT module declaration :

```
module M
     Declarations
     Statements
end module
```

Should yield the parser output token stream :

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

Statement Sequences

MiniT 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 using a *Statements* rule such as the one suggested above.

Statements

Modify the parsing of **if**, **case**, **while**, **repeat** and **begin** statements to instead meet the MiniT language specification of MiniT **if, case** and **loop** statements. The goal is to have the output token stream for the MiniT 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 MiniT **if** statement :

```
if x = y then
     Statements1
else
     Statements2
end if
```

Should yield the parser output token stream :

```
sIfStmt
sIdentifier  <identifier index for 'x'>
sIdentifier  <identifier index for 'y'>
sEq
sExpnEnd
sThen
sBegin
  Statements1
sEnd
sElse
sBegin
  Statements2
sEnd
```

Elsif Clauses

The handling of **elsif** in the MiniT **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.

The first alternative would add a new *sElsIf* 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 **elsif**, such as this one:

```
if x = 42 then
     Statements1
elsif y = 71 then
     Statements2
else
     Statements3
end if
```

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

```
        if x = 42 then
            Statements1
        else
            if y = 71 then
                Statements2
            else
                Statements3
            end if
        end if
```

That is to say :

```
        sIfStmt
        sIdentifier  <identifier index for 'x'>
        sInteger  42
        sEq
        sExpnEnd
        sThen
        sBegin
          Statements1
        sEnd
        sElse
        sBegin
          sIfStmt
          sIdentifier  <identifier index for 'y'>
          sInteger  71
          sEq
          sExpnEnd
          sThen
          sBegin
            Statements2
          sEnd
          sElse
          sBegin
            Statements3
          sEnd
        sEnd
```

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 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!

Case Statements

The output for MiniT **case** 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 MiniT **case** statement and the PT **case** statement is identical, except for the MiniT *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 MiniT, 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 MiniT *default* alternative (**label :**) on **case** statements is new, and we must handle it specially. But what we will do is simple - just check for a **label :** following the other alternatives in the case statement, and output *sCaseOtherwise* followed by the statements of the **label :** clause, once again using the *Statements* rule to enclose them in *sBegin* .. *sEnd*. before outputting the *sCaseEnd* semantic token.

For example, the MiniT **case** statement:

```
        case i of
          label 42:
             Statements1
          label 43:
             Statements2
          label :
             Statements3
        end case
```

Should yield the parser output token stream :

```
        sCaseStmt
        sIdentifier  <identifier index for i>
        sExpnEnd
        sInteger 42
        sBegin
          Statements1
        sEnd
        sInteger 43
        sBegin
          Statements2
        sEnd
        sCaseOtherwise
        sBegin
          Statements3
        sEnd
        sCaseEnd
```

Loop Statements

Remove handling of the PT **while** and **repeat** statements, and add handling of the MiniT **loop** statement. The output stream should use the token *sLoopStmt* to mark the beginning of the statement, and the *sLoopExitWhen* token for **exit when** *expression* clauses. The end of the conditional expression following **exit when** should be marked with the *sExpnEnd* token, as it is for the old PT Pascal **while** statement. Use your *Statements* rule to handle the statement sequences before and after the **exit when**, which will automatically output *sBegin* .. *sEnd* tokens around the sequences.

For example, the MiniT loop:

```
        loop
            Statements1
            exit when Expression
            Statements2
        end loop
```

Should yield the parser output token stream :

```
sLoopStmt
sBegin
  Statements1
sEnd
sLoopExitWhen
  Expression
sExpnEnd
sBegin
  Statements2
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 MiniT **loop**, so we'll just have to leave it until the Semantic phase.

Short Form Assignments

Add the parsing of MiniT 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
```

Iterator Statements

Add the parsing of the MiniT iterator statements **++** and **--**. In this case we can't fool the Semantic phase by converting to the equivalent addition and subtraction assignments, because there's no way in an S/SL parser to generate the integer literal 1 from no input token. (We could add a semantic operation to do this, but that's cheating, in a parser.)

So instead, for the iterator ++, we will generate an assignment statement, but use the new semantic tokens *sIncrement* for ++ and *sDecrement* for --. For example, for the MiniT iterator statement `i++`, we will output the semantic token stream:

```
sAssignmentStmt
sIdentifier  <identifier index for 'i'>
sIdentifier  <identifier index for 'i'>
sIncrement
sExpnEnd
```

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 MiniT string operators *substring* (s @ *expression* **..** *expression*) and *length* (**?** s).

The precedence of the substring operator **@** *expression* **..** *expression* should be the same as *, **div**, **mod** and **and**, and the precedence of the length operator **?** is the same as the precedence of **not**. Both of the new operators should be converted to postfix by your parser, using the postfix operator output tokens *sSubstring* and *sLength.*

The **@** 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
```

Other Syntactic Details

Watch out for other minor syntactic differences between PT Pascal and MiniT - for example, semicolon is a separator between statements in PT, but is a null declaration and a null statement in MiniT.

Whenever you have any questions about what is allowed or not allowed in MiniT, refer to the MiniT language specification in the document *"CISC/CMPE 458 Course Project Winter 2019: The MiniT Programming Language"*. Any forms not explicitly defined there retain their original PT Pascal form and meaning.

No modifications, extensions or changes to the MiniT language are allowed in your compiler; you must implement the language exactly as specified.