Phase 2 : Parsing

Team 7 : Jae Eun Kim, Eun Do Lee

1. Introduction

For this phase project, our team implemented a parser that supports the SnuPL/1 language. A parser constructs a tree from a stream of tokens according to its grammatical structure. Each non-terminal can be modeled as a function. The relationship between non-terminals and terminals, specified by productions, can be implemented by function calls.

2. Basic Function Implementation

The following is the list of functions we implemented for the parser. The last four functions are for creating nodes of terminals rather than non-terminals. More on nodes would be explained later.

Implementing the function itself is, in principle, quite simple. We implement it according to the right hand side of the non-terminal's production. When a non-terminal is requested, the parser calls the corresponding function. When a terminal is requested, it consumes the token and checks if the type matches the one the production requested. The following is an example from SnuPl/1. There is a production 'assignment = qualident ":=" expression'. 'assignment', 'qualident' and 'expression' are non-terminals whereas ":=" is a terminal. The function 'assignment' first calls the function 'qualident'. If 'qualident' does not throw any error, 'assignment' consumes the next token and checks if it is ":=", the assignment token. If it is not, it throws an error. Then 'assignment' calls the 'expression' function. If 'expression' throws no error, 'assignment' is complete.

3. FIRST and FOLLOW

This process was not done without careful planning. In many cases, it was necessary to figure out the FIRST and the FOLLOW of the non-terminal before implementing the function. For example, some non-terminals were optional. Then the simple thing would be to move on if the next token was not in the FIRST of such non-terminal. In another case, a non-terminal had multiple productions. An example is 'statement = assignment | subroutineCall | ifStatement | whileStatement | returnStatement'. We solved the dilemma of which one to call by comparing the next token and the FIRST sets of the non-terminals on the right hand side. In the SnuPL/1 language, we didn't have to left factor any productions. FIRST can also be useful when spotting errors.

FOLLOW was useful when implementing non-terminals that ended with the form '{α}', where α denotes some grammar. Such cases were statSequence, constDeclSequence, and varDeclSequence. We implemented them by loops, and broke the loop when next token was in the FOLLOW set. Simply stopping when an unexpected token appears would be unwise because the token may be an error.

4. Implementing LL(2)

One of the most difficult sequences to implement was varDeclSequence because it requires LL(2). Its production is ‘varDeclSequence = varDecl { “;” varDecl }’. When the parser has gone through varDecl and the next token is “)”, it is clear that varDeclSequence is finished. The problem is when the next token is “;”. It is impossible to tell whether it is a separator between the last varDecl and the next varDecl or it is the last token of varDeclaration. The parser has to look at one more token ahead. If it is an identifier, the parser has to call another varDecl. If it is one of the elements of FOLLOW(varDeclaration) = {tBegin, tProcedure, tFunction}, varDeclSequence is over. The semicolon was the last token of varDeclaration. Implementing this neatly was not easy. The scanner can peek only one token. In order to know the next token of the next token, the parser has to consume the semicolon even though it might belong to varDeclaration. That is why our ‘varDeclaration’ function does not consume tSemicolon at the end. It is done during the varDeclSequence stage, and the parser decides what to do next then. To minimize complicated condition statements, we separated varDeclSequence into two functions, ‘varDecl’ and ‘varDeclSequenceAux’. ‘varDeclSequenceAux’ is called when the non-terminal varDeclSequence is requested. ‘varDecl’ is the exact implementation of the non-terminal varDecl according to its production. It consumes tokens and calls functions accordingly, and adds symbols to symbol tables. ‘varDeclSequenceAux’ is the function that decides what to do next after each varDecl. To make things simple, it accepts a boolean parameter ‘formalParam’. ‘varDeclSequenceAux’ has a loop whose condition is that the next token is tSemicolon. If it is, it first consumes the semicolon. Then it checks if ‘formalParam’ is negative and the next token is not an identifier. If both are true, the loop is terminated and so is varDeclSequence. Otherwise ‘varDecl’ is called and varDeclSequence continues.

5. Eliminating Left Recursion

It is crucial that there is no left recursion when building a predictive parser. However, the ‘type’ non-terminal had a left recursion, so we had to fix it. Originally, the production of type was ‘type = basetype | type “[“ [simpleexpr] “]”. We split it into two productions to eliminate left recursion :

```
type = basetype arraytype
arraytype = “[“ expression “]” arraytype
```

We made a separate function named ‘arraytype’ to efficiently implement the ‘type’ function.

6. Nodes and Scope

Certain functions create and return nodes of our parsing tree. Return types that start with letters ‘CAst’ are nodes.

Of these nodes, CAstModule and CAstProcedure are special because they have their own scope and symbol table. CAstModule’s scope is global, while CAstProcedure’s is local. Scope is passed down from caller to callee via parameter CastScope *s. Whenever a symbol table is created, it is initialized so that predefined function symbols are included. That way, functions of the same name cannot be declared by the user.

7. Symbols

Symbols must be created according to their type and inserted into the correct symbol table. For most symbols, this was not too difficult. Type could be set using the type manager, and local symbols simply get inserted in s→GetSymbolTable(). Global symbols, notably CSymProc, automatically get inserted to the global symbol table. Constant symbols(CSymConst), variable symbols(CSymbol) and parameter symbols(CSymParam) were the difficult ones because they were created and inserted in sequences. The following are the details of their implementations.

a) Constant Symbols

Constant symbols are created and stored at function ‘constDecl’. Production of constDecl is ‘constDecl = varDecl “=” expression’. The ‘constDecl’ function does not call the ‘varDecl’ function because it must create constant symbols, not variable symbols. We simply implemented constDecl’s own version of varDecl. Production of varDecl is ‘varDecl = ident { “,” ident } “:” type’. Until the parser consumes a colon, it keeps consuming identifier tokens and storing them in a vector. After consuming a colon, it calls the ‘type’ function and stores the returning node. Then varDecl is over, and the parser consumes tRelOp(=) and calls ‘expression’. The returning node of ‘expression’ is stored and the node has an Evaluate function which returns the value of the expression. After that, the parser brings s→ GetSymbolTable(), where it will store the symbols. Until the vector is empty, the parser pops an identifier token, creates a symbol of type CSymConstant with the value of the token as its name, returning value of GetType() of the type node as its type, and its data initialized as the returning value of the Evaluate function of the expression node. However, for this phase, we initialized the data to 1. The data initializing was done with the help of CDataInitializer. After the symbol is created, the parser inserts it into the symbol table and shows an error message if the same symbol is already in the same table. This way we can prevent re-defined variables and constants.

b) Variable Symbols and Parameter Symbols

Creating and inserting variable and parameter symbols are actually implemented at the same location, the ‘varDecl’ function because the process is the basically the same. The difference is the type of the symbols and the fact that parameter symbols must be included to their corresponding procedure symbol. As mentioned above, the ‘varDecl’ function is the implementation of the production of the non-terminal varDecl. Therefore the overall procedure is similar to the ‘constDecl’ function without consuming tRelOp(=) and calling ‘expression’. We will only mention what is different. One major difference is that the ‘varDecl’ function uses a boolean parameter ‘formalParam’. If ‘formalParam’ is false, variable symbols are being requested, and the parser creates symbols of the type CSymbol and stores them at s→GetSymbolTable(). If ‘formalParam’ is true, parameter symbols are being requested, so the parser first creates CAstProcedure *cap = dynamic_cast<CAstProcedure*>(s) and brings symbol table cap → GetSymbolTable(). The parser then creates symbols of the type CSymParam, stores them at the symbol table, and adds them to the parameters of the corresponding procedure symbol obtained by cap→GetSymbol(). Unlike constant symbols, variable and parameter symbols do not require data initializing.

8. Array Type

There is one more function that accepts the boolean parameter ‘formalParam’, the ‘type’ function. Normally open arrays are not allowed, but when ‘formalParam’ is true, the parser accepts open arrays because they are allowed in parameter definitions.

9. Conclusion

We tested the parser on all test inputs in the test/parser directory and made sure that it produces the same results as the reference binary unless the difference has something to do with future phases. We manually compared the parsing tree and the input text from test01, test02, test03 and test04. We checked that each nested scope and symbol table is created, symbols of the correct type are created and inserted in their correct symbol table, and made sure that each statement has correctly ordered operations. We also checked that our parser produces the correct error messages.