1. Intro

In the second phase of our term project, our primary focus is on implementing a predictive top-down parser for the SnuPl/2 language which is well suited for this parsing method. We build an abstract syntax tree which consumes tokens received from the scanner we implemented in Phase 1. This report provides an overview of the work completed during this phase, including the design, implementation and testing of the parser.

2. Implementation

Before briefly explaining the process of parsing, there are few things to notify.

* All the methods created in Phase 2 is implemented in parser.cpp
* When we say “create CAstxxx” it means we create a instance which is node of the Abstract Syntax Tree, and the nodes have slightly different properties depending on its type.
* Every function mentioned is **bolded**
* The word “vector” is used as array (C++ dynamic array)

2-1. We start parsing by parsing the module.

1. Consume “module”, identifier, and semicolon
2. Create scope(CAstModule) and pass its symbol table to function **InitSymbolTable**, which initializes all the necessary keywords predefined in SnuPl/2
3. Handle declarations with while loop by checking the next token’s type. Depending on it, we handle it by **constDeclaration**, **varDeclaration**, **procDeclaration** functions
4. End the loop if type of the token is not one of the declare tokens, since Follow set of all declarations are always one of declare tokens.
5. If “begin” appears, consume it and handle the body with **statSequence**, which returns CAstStatement.
6. Set scope’s statement sequence the return value at step 5 (with predefined method of CAstModule)
7. Consume rest of the necessary tokens and check if the value of identifier consumed matches the one consumed at step 1

2-2. How we parse variable declaration:

varDeclaration = [ "var" varDeclSequence ";" ].

varDeclSequence = varDecl { ";" varDecl }.

varDecl = ident { "," ident } ":" type.

The main challenge we had in parsing variable declaration is that we need to add every variable to the scope’s(node of the AST) symbol table and therefore the type and the name of the variables are needed.

We implemented the method starting at function **varDeclaration**:

1. Consume “var”
2. Handle varDeclSequence by while loop until the next token is not a identifier
3. Inside the while loop, we declare vector of string which contains names of all the variable declared in varDecl.   
   Then, pass the vector and call **varDecl** which stores all variable names and get their type from the returned value from **varDecl**.
4. Add all variables to symbol table and consume semicolon.
5. Repeat 2~4

Now we briefly explain how function **varDecl** is implemented:

1. Consume all identifiers until colon appears  
   Add their values to the vector which is passed from **varDeclaration**
2. Handle the type by first consuming basetype
3. Then, if it is an array, we consume all brackets and size of each dimension (which is handled by **number** function)
4. After that, we use CTypeManager class methods to return appropriate CType value

\*\* At step 3, further phase we would need to handle simple expression inside the brackets but for current phase we just assume it is a number.

Const declaration is also done in similar method

2-3. How we parse subroutine declaration:

subroutineDecl = (procedureDecl | functionDecl)

( "extern" | subroutineBody ident ) ";".

procedureDecl = "procedure" ident [ formalParam ] ";".

functionDecl = "function" ident [ formalParam ] ":" type ";".

formalParam = "(" [ varDeclSequence ] ")".

The main challenge we faced in this period is we need to include the parameters of the function/procedure into its symbol table. So while parsing formalParam, we store all variables’ name and type in vector. The vector contains pair of vector(of variable names) and corresponding type, i.g. vector<pair<vector<string>, CType \*>>

Then, after some period we can get the return type of the subroutine and make CAstProcedure node and push all variables in the vector into the subroutine’s symbol table.

Rest of the step is similar to step 5~7 of module

Following steps shows what is done step by step from function **procedureDecl**:

1. Consume “function” or “procedure” and declare the type of this subroutine
2. Parse formalParam by consuming necessary tokens and while consuming identifiers inside left parens, call **varDecl** and store all variables’ name in a vector
3. Make pair of variables’ name vector and variables’ type and push it into another vector which is mentioned at 2-3 explanation
4. If subroutine type is function, get appropriate CType value
5. Create new node with CAstProcedure for this subroutine scope
6. Add all parameters declared into the scope’s symbol table
7. If “begin” appears, consume it and handle the body with **statSequence**, which returns CAstStatement.
8. Set scope’s statement sequence the return value at step 5
9. Consume rest of the necessary tokens

2-4. How we parse statSequence:

statSequence = [ statement { ";" statement } ].

statement = assignment | subroutineCall | ifStatement | whileStatement | returnStatement.

assignment = qualident ":=" expression.

subroutineCall = ident "(" [ expression {"," expression} ] ")".

ifStatement = "if" "(" expression ")" "then" statSequence [ "else" statSequence ] "end".

whileStatement = "while" "(" expression ")" "do" statSequence "end".

returnStatement = "return" [ expression ].

In statSequence, we handle the body of functions, procedures, and module. We implement statSequence as a loop. We keep a ‘head’ that points to the first statement and is returned at the end of the function( it can be NULL if no statement). In the loop, we track the end of the linked list using ‘tail’ and attach new statements to that tail.

The implementation of statSequence starts at function **statSequence**

1. Check if the next token to consume is “end” or “else”. If true, return NULL
2. Handle each statement in while loop  
   Call **ifStatement**, **whileStatement**, **returnStatement** if next token is “if”, “while”, “return”  
   If next token is identifier, get the token’s type by searching through symbol table of current scope and it’s ancestors’ scopes by using FindSymbolmethod with argument sGlobal which calls FindSymbol method from parent symbol table
3. Depending on the type, call either **subroutineCall** or **assignment**
4. If head pointer is NULL then set head as return value of the function called at 2,3
5. Modify tail pointer to appropriately maintain linked list
6. If next token is semicolon, consume it. Else, break
7. Return head pointer

Functions used here is simple and doens’t create any symbol table and quite self-explanatory

2-5. Other functions:

Here we explain some functions that needs some explanation

**qualident:**

Here we handle variable including arrays. First, we find its symbol by using FindSymbol method. We use that Csymbol pointer to create either CAstArrayDesignator or CAstDesignator depending on rather it contains brackets. It returns the created instance.

**charConst:**

This function is called when handling with character value assignment. When creating CAstConstant, we just set its value as token value’s first index character except for few cases. Since the scanner from Phase 1 output value for ‘\n’, ‘\t’, ‘\\’, ‘\0’, ‘\’’, ‘\”’ is

“\\n”, “\\t”, “\\\\”, “\\0”, “\\’”, “\””

We handle character values that it matches phase 1’s translation.