Syantx Analysis

Oh Gyuhyeok (2020-10485) and Hwang SeungJun (2020-15313)

SNUCSE

Abstract

This report presents the design and implementation of a compiler for the SNUPL/2 language, a powerful type-based procedural language designed for educational purposes. The compiler consists of several key components, including a scanner, parser, optimizer, and code generator. In this report, we primarily focus on the parser, which is responsible for translating high-level source code into an abstract syntax tree (AST). The main objectives of this phase of the project were to design and implement a top-down parser for the SNUPL/2 language, generate an AST in both textual and graphical forms, perform syntactical checks, and enforce constraints related to the language's syntax. The parser phase involves parsing modules, declarations, subroutines, statements, and expressions. It handles various types of declarations, including variables, constants, procedures, and functions. Special attention is given to the parsing of formal parameters for subroutines, ensuring that the symbol table is appropriately populated. Semantic correctness and error checking are integral parts of the parser's functionality. The report provides a detailed overview of the parser's implementation, including the structure and logic of the parsing process. It also highlights the challenges faced and solutions devised during the implementation. The system's robustness and correctness are validated through a range of test cases, including both provided and custom scenarios. This work represents a crucial step in the development of a SNUPL/2 compiler, laying the foundation for subsequent phases, including optimization and code generation.

1 Introduction

Compiler is a complex program that is one of computer science as an enemy effort. It is a program that translates a program written in a high-level language into a program written in a low-level language. For understanding of this compiler, you can learn this by implementing the compiler of SNUPL/2[3], a powerful type-based procedural language designed for educational purposes.

The compiler consists of several components: scanner, parser, optimizer and code generator[1]. In the first phase of our term project, we implemented a scanner for the SnuPl/2 language. In the second phase of our term project, our primary focus is on implementing a predictive top-down parser[2] for the SnuPl/2 language which is well suited for this parsing method. We focus on the syantax, which is the structure of the program, and we build an abstract syntax tree which consumes tokens received from the scanner we implemented in Phase 1. Next phase, we will focus on semantic analysis, which is the meaning of the program, and we will build a symbol table and perform type checking.

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.

1.1 Progress Overview

The main objectives of this phase were as follows:
1. Design and implement a top-down parser for the SnuPL/2 language based on Appendix.A.1.
2. Generate an abstract syntax tree (AST) in both textual and graphical forms.
3. Perform syntactical checks, such as ensuring symbols are defined before use.
4. Enforce constraints related to the SnuPL/2 syntax, such as matching declaration and end identifiers.

2 Implementation

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

1. All the methods created in Phase 2 is implemented in parser.cpp
2. 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.
3. Every function mentioned is bolded
4. The word “vector” is used as array (C++ dynamic array)

Parsing step is below and we will explain each step in detail in the following sections.

2.1 Parsing module

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 Parsing declaration

There are two different type of declartion:variables containing const varaible and subroutine: procedure and function. We will explain how we parse each declaration in detail in the following sections.

2.2.1 Parsing variable declaration

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 will explain how we parse each declaration in detail in the following sections.

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 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

Before we explain how function varDecl is implemented, we need to explain how we handle type of the variable. We need to know the variable and its type before we can know the symbol table of the variable for function declaration. So we need to save variables and the their type.

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

Const declaration is also done in similar method except considering the value of the constant. We save the value of the constant with `CDataInitializer` which initializes the value with the expression.
In further phase, we would need to handle simple expression inside the brackets but for current phase we just assume it is a number.

Parsing subroutine declaration

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

2.3 Sequence of statements

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 FindSymbol method 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

2.4 Expression

The implmentation of parsing expression starts at function `expression`.
expression:

1. Call simpleexpr for left.
2. If next token is relational operator, consume it and call simpleexpr for right.
3. Create appropriate node depending on the operator and return it.

simpleexpr:

1. If next token is plus or minus, consume it and call term for left.
2. If next token is term operator, consume it and call term for right

term:

1. Call factor for left.
2. If next token is factor operator, consume it and call factor for right.
3. Create appropriate node depending on the operator and return it.

factor:

1. If next token is identifier, call qualident.

2. If next token is number, call number.
3. If next token is boolean, call boolean.
4. If next token is character, call character.
5. If next token is string, call string.
6. If next token is left parenthesis, consume it and call expression. Then consume right parenthesis.
7. If next token is exclamation mark, consume it and call factor.

unaryOp: For sign of the number
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:

We handle character values that it matches phase 1's translation.
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", "\\'", "\\""

other consts:

We handle other constants such as integer, boolean, string.
We just create `CAstConstant` with appropriate value and return it.
For string, we need to save the value of the string to the symbol table. So we use diffirent class: `CAstStringConstant`.

3 Result

We utilized a provided test program that printed the AST to validate our parser's functionality. The test program allowed us to examine the generated AST for different test cases. In addition to the provided test cases, we created custom test cases to cover more complex and special scenarios, ensuring the robustness of our parser.

References

[1] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compiler: Principles, Techniques, and Tools*. Pearson Addison-Wesley, 2007.

[2] Bernhard Egger. *Compiler Lecture Note*. 2023.

[3] Bernhard Egger. Snupl2 compiler, Sep 2023.

A Appendix

A.1 Grammer

EBNF notation is used to describe the grammar of the SnuPL/2 language.

```
module              = “module” ident “; ” {constDeclaration | varDeclaration | subroutineDecl}
                     [ “begin” statSequence ] “end” ident “.”

letter              = “A”..“Z” | “a”..“z” | “_”
digit               = “0”..“9”
hexdigit            = digit | “A”..“F” | “a”..“f”
character           = LATIN1_char | “\n” | “\t” | “\”” | “\’” | “\\” | hexencoded
string              = “”” {character} “””

ident               = letter {letter | digit}
number              = digit {digit} [ “L” ]
boolean             = “true” | “false”
type                = basetype | type “[” [ simpleexpr ] “]”
basetype            = “boolean” |  “char” | “integer” | “longint”

qualident           = ident {“[” simpleexpr “]”}
factOp              = “ * ” | “/” | “&&”
termOp              = “ + ” | “ − ” | “||”
relOp               = “ = ” | “#” | “ < ” | “ <= ” | “ > ” | “ >= ”

factor              = qualident | number | boolean | char | string | “(” expression “)” | subroutineCall | “!” facto
term                = factor {factOp factor}
simpleexpr          = [“ + ” | “ − ”] term {termOp term}
expression          = simpleexpr [relOp simpleexpr]

assignment          = qualident “ := ” expression
subroutineCall      = ident “(” [expression {“, ” expression} ]
ifStatement         = “if” “(” expression “)” “then” statSequence [“else” statSequence] “end”
whileStatement      = “while” “(” expression “)” “do” statSequence “end”
returnStatement     = “return” [ expression ]

statement           = assignment | subroutineCall | ifStatement | whileStatement | returnStatement
statSequence        = [statement {“; ” statement }]

constDeclaration    = [ “const” constDeclSequence ]
```

```
constDeclSequence   = constDecl “; ” {constDecl “; ”}
constDecl           = varDecl “ = ” expression

varDeclaration      = [ “var” varDeclSequence “; ” ]
varDeclSequence     = varDecl {“; ” varDecl }

subroutineDecl      = (procedureDecl | functionDecl) (“extern” | subroutineBody ident ) “; ”
procedureDecl       = “procedure” ident [ formalParam ] “; ”
functionDecl        = “function” ident [ formalParam ] “ : ” type “; ”
formalParam         = “(” [ varDeclSequence ] “)”
subroutineBody      = constDeclaration varDeclaration “begin” statSequence “end”

comment             = “//” {[^\n]] \n
whitespace          = {“ ” | \t | \n}
```

A.2 Method

We implemented additional method to implement parser. List of method is below.

- `parseModule` : parse module
- `parseConstDecl` : parse const declaration
- `parseVarDecl` : parse variable declaration
- `parseSubroutineDecl` : parse subroutine declaration
- `parseFormalParam` : parse formal parameter
- `parseSubroutineBody` : parse subroutine body
- `parseStatSequence` : parse statement sequence
- `parseStatement` : parse statement
- `parseAssignment` : parse assignment
- `parseSubroutineCall` : parse subroutine call
- `parseIfStatement` : parse if statement
- `parseWhileStatement` : parse while statement
- `parseReturnStatement` : parse return statement
- `parseSimpleExpr` : parse simple expression
- `parseTerm` : parse term

- `parseFactor` : parse factor
- `parseQualident` : parse qualident
- `parseNumber` : parse number
- `parseBoolean` : parse boolean
- `parseChar` : parse char
- `parseString` : parse string
- `parseType` : parse type
- `parseBasetype` : parse basetype
- `parseIdent` : parse identifier
- `parseRelOp` : parse relational operator
- `parseTermOp` : parse term operator
- `parseFactOp` : parse factor operator
- `constChar` : parse constant char

A.3 Additional Test Cases

Input:

```
    # Any additional test case
```

Output:

```
result
```