Compilers Project 2 ‘Parsing’ Report

2013-11431 Hyunjin Jeong

The second term project was to make a parser for tokens generated by a scanner. The input of a parser is tokens, and the output of a parser is Abstract Syntax Tree(AST)s and related symbol tables. In this report, I will write a brief summary how I implemented a parser, starting from the ‘module’ function and following the order of my source code.

- Module

module ::= “module” ident “;” varDeclaration { subroutineDecl } “begin” statSequence “end” ident “.”.

In module function, ‘varDeclaration’ is implemented directly using ‘varDecl’ function in ‘module’ function, not defining new function. ‘subroutineDecl’ is implemented by calling ‘subroutineDecl’ function. ‘statSequence’ is implemented by calling ‘statSequence’ function and using ‘SetStatementSequence’ function. At the end, parser checks whether the first ident value and the second ident value are same.

There are some predefined functions: DIM(), DOFS(), ReadInt(), WriteInt(), WriteChar(), WriteStr(), WriteLn(). They are added into the module’s symbol table.

- subroutineDecl

subroutineDecl ::= (procedureDecl | functionDecl) subroutineBody ident ";".

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

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

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

subroutineBody ::= varDeclaration "begin" statSequence "end".

If the first token is 'tProcedure' then 'procedureDecl' begins and if 'tFunction' then 'functionDecl' begins. Then parameters are declared by using 'varDeclParam' in 'procedureDecl'. It's same with 'module' except using 'varDeclParam' instead of 'varDecl'.

However, 'functionDecl' does not use 'varDeclParam' and parameters are declared directly. My parameter declaration implementation in 'functionDecl' is quite complex. Here's how I implemented it:

- There are two vectors to save parameter information: params, vars & vars_loop. 'params' is used to save parameters, and 'vars' and 'vars_loop' are used to save parameters temporarily in loop.
- Each 'varDecl' is each loop, so 'varDeclSequence' consists of loops. In each loop, idents are saving temporarily in 'vars' or 'vars_loop'. After end of each loop, parameters are saved into 'params' with index. Index is used to remember current position of parameter.
- After end of all loops, parameters in 'params' are added into symbol table.

Implementation of 'subroutineBody' is same with 'module'. 'varDeclaration' is implemented directly using 'varDecl' function and 'statSequence' is implemented by calling 'statSequence' function. At the end, parser check the first ident and the second ident have same value.

- varDeclParam & varDecl

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

These two functions are similar. 'varDeclParam' is used to declare parameters in 'procedureDecl', and 'varDecl' is used to declare local or global variables in 'module' or 'subroutineDecl'. 'varDeclParam' takes one more parameter, 'CSymProc *proc' to use 'proc->AddParam' function.

These functions consume 'tIdent' tokens and call 'type' function to get type. Then they add variables(or parameters) into symbol table.

- type

type ::= basetype | type "[" [number] "]".

basetype ::= "boolean" | "char" | "integer".

This function is used to get type. First, it consumes one token, which must be 'basetype', then it sets types by using type manager's GetType() functions(Boolean: GetBool, character: GetChar, integer: GetInt).

After that, if next token is 'tLSBrak', it means the type is array. Then a parser consumes array elements, and set array type by using type manager's GetArray() function. If isParam is true, then arrays are addressed by using type manager's GetPointer() function.

- statSequence

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

FIRST(statSequence) = {'tIf', 'tWhile', 'tReturn', 'tIdent'}.

Follow(statSequence) = {'tEnd', 'tElse'}.

If the first token is included in FIRST(statSequence), a parser sets head, and head calls 'statement' function. Then it takes multiple statements and statements are attached to head until statements ended. Statements is implemented by calling 'statement' function. Finally, this function returns head.

If the first token is not in FIRST(statSequence), this function just returns NULL.

- statement

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

FIRST(statement) = {tIf, tWhile, tReturn, tIdent}.

The 'statement' function just calls another functions according to first token type. If the token is 'tIf', it calls 'ifStatement'. Else if the token is 'tWhile', it calls 'whileStatement'. Else if the token is

'tReturn', it calls 'returnStatement'.

If the token is 'tIdent', there can be two cases: 'subroutineCall', 'assignment'. Therefore, a parser consumes first token, and peek second token. If second token is 'tLBrak', then calls 'subroutineCall', else calls 'assignment'.

If the first token is not in FIRST(statement), then it just sets error.

- subroutineCall

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

First, a parser checks undefined subroutine call and if 'ident' doesn't exists, then it sets error. Then if next token is expression(check using 'isExpr' function), parser call 'expression' function and expressions are added into CAstFunctionCall's arguments.

If expressions are array, they are addressed(add '&' operation). Then they are added into arguments, too.

- assignment

assignment ::= qualident ":=" expression.

qualident ::= ident { "[" expression "]" }.

In assignment, it looks the second token is 'tAssign' or 'tLBrak'. 'tAssign' means 'qualident' is only ident, but 'tLBrak' means 'qualident' is array.

A parser finds a symbol by identifier value and sets error if not found. Left-Hand Side(LHS) is 'CAstDesignator' in non-array case and LHS is 'CAstArrayDesignator' in array case. And the array designator takes indexes.

Expression just calls 'expression' function and it becomes Right-Hand Side(RHS).

This function returns 'CAstStatAssign' with its LHS and RHS

- ifStatement

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

Implmentation 'ifStatement' is straight-forward. 'expression' calls 'expression' function and it becomes condition. 'statSequence' calls 'statSequence' function and it becomes 'ifbody'. If 'tElse' are consumed, then another 'statSequence' function calls again, and it becomes 'elsebody'. Otherwise 'elsebody' is NULL.

This function returns 'CAstStatIf' with its condition, 'ifbody', 'elsebody'.

- whileStatement

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

'whileStatement' is easier to implement than 'ifStatement'. 'expression' calls 'expression' function and it becomes condition. 'statSequence' calls 'statSequence' calls and it becomes 'body'.

This function returns 'CAstStatWhile' with its condition, 'body'.

- returnStatement

returnStatement ::= "return" [expression].

A parser first consumes 'tReturn' token. Then if next token is 'expression' (check using 'isExpr' function) then calls 'expression' function.

This function returns 'CAstStatReturn' with its expression.

- expression

expression ::= simpleexpr [relOp simpleexpr].

'simpleexpr' calls 'simpleexpr' function. If next token is 'relOp = {=, #, <, >, <=, >=}', 'simpleexpr'

function is called again. Then this function returns 'CAstBinaryOp' with its operator, left simpleexpr, right simpleexpr.

If next token is not relOp, then this function just returns left simpleexpr only.

- simpleexpr

simpleexpr ::= ["+" | "-"] term { termOp term }.

First, a parser checks whether the first token value is "+" or "-". If then, a parser calls 'term' function. If a type of 'term' is 'number' and the first token is "-", then parser negates the number. Otherwise 'term' becomes 'CAstUnaryOp'. Then if next token is 'termOp = {+, -, ||}', this function returns 'CAstBinaryOp' with its operation, first term, second term.

- term

term ::= factor { factOp factor }.

First, a parser calls 'factor' function. Then if next token is 'factOp = {*, /, &&}', parser calls 'factor' function again. This function returns 'CAstBinaryOp' with its operation, left factor, right factor.

- factor

factor ::= qualident | number | boolean | char | string | "(" expression ")" |

subroutineCall | "!" factor.

FIRST(factor) = { tIdent, tNumber, tTrue, tFalse, tCharacter, tString, tLBrak, tEMark }.

A parser checks if the first token is in FIRST(factor).

If the first token is 'tNumber', then this function returns 'CAstConstant' with number value. If the number is bigger than 2147483648, then a parser sets error.

If the first token is 'tTrue' of 'tFalse', this function returns 'CAstConstant' and value is 1 in case

'tTrue', 0 in case 'tFalse'.

If the first token is 'tString', this function returns 'CAstStringConstant' with string value.

If the first token is 'tCharacter', then a parser escapes character values and this function returns "CAstConstant" with ASCII value of escaped character. If a length of character is zero, then the value sets '₩0' because real zero length character is handled to error in scanner.

If the first token is 'tEMark', then this function returns 'CAstUnaryOp' with 'opNot' operation and 'factor'.

If the first token is 'tLBrak', then this function returns 'expression'.

If the first token is 'tIdent', there can be two cases: qualident, subroutineCall. A parser first consumes 'tIdent', and peeks next token. If next token is 'tLBrak', then 'subroutineCall' begins. Otherwise 'qualident' begins. 'subroutineCall' in 'factor' has same procedure in 'subroutineCall' functions. 'qualident' in 'factor' does the same thing with 'qualident' in 'assignment', too.

- isExpr

FIRST(expression) = {"+", "-", FIRST(factor)}

= {"+", "-", tIdent, tNumber, tTrue, tFalse, tCharacter, tString, tLBrak, tEMark}.

This function is used to check token is in FIRST(expression). It returns true if the token is in FIRST(expression) and returns false if not.

- Type Checking

I did some type checking related implementation in this project. My parser checks duplicate parameters, variables, procedures or functions declaration. Also my parser checks undefined variables, parameters, procedures or functions usage.

And I implemented some 'GetType()' function in ast.cpp. 'GetType()' functions of 'CAstBinaryOp', 'CAstUnaryOp', 'CAstSpecialOp', 'CAstArrayDesignator', and 'CAstStringConstant' are implemented in parser.