|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | | | | | | | | |
|  | | Parser Project | | | | |  | |
| Several white paper cutout snowflakes, on a white background |  | | | | | | |  |
|  | | | |  |  | | | |
|  | | | | Syed Faheel Aziz |  | | | |
|  | | | | 04/29/2022—CSCI 330 Programming Language Concept—**Dr. ALSAQAA HAZIM** |  | | | |
|  | | |  | | |  | | |

Contents:

1. Project Objective…….
2. Introduction……….
3. Functions and Data Structures …..
4. BNF Rules…….
5. Prolog Code with updated statements………
6. Code…………………
7. Execution…………
8. Output……..
9. Code in C++……..
10. Output………..
11. User Doc……
12. Summary…….
13. Citation……

Project Objective:

The parser project is divided into two stages: scanning and parsing. The scanner, accepts a string and converts it into a list of tokens. The input text is a C++ program made up of C++ statements, like: if, else if, for, while, variable/constant declarations, assignment, expressions and other statements. Expressions are addition, subtraction … etc. Here's how this project works: • The scanner takes a list of tokens and tests if it can be transformed into a list of statements, then takes the same list of tokens and tests if the tokens can be divided up into individual statements, each statement ending with a semicolon. • The parser tests if the first tokens of the token list make up a legal statement. If so, the statement is returned in a structure and the remaining tokens are tested again. For more details see Parsing1.pdf and Parsing2.pdf posted in D2L.

Requirements:

1. Parsing program for C++ like statements in Prolog language.
2. Documentation Report

Introduction:

Parsing entails translating the input stream into words in the target language. A parser is often used as part of an interpreter or compiler. There are three stages to the parsing process:

1. **Lexical Analysis**: A lexical analyzer is used to produce tokens from a stream of input

string characters, which are broken into small components to form meaningful

expressions.

1. **Syntactic Analysis**: Checks whether the generated tokens form a meaningful expression.

This makes use of a context-free grammar that defines algorithmic procedures for

components. These work to form an expression and define the order in which tokens

must be placed.

1. **Semantic Parsing**: The final parsing stage in which the meaning and implications of the

validated expression are determined, and necessary actions are taken.

The main goal of a parser is to see whether input data can be deduced from the grammar's start symbol. If that's the case, how can this input data be derived? This is accomplished in the following way:

* **Top-Down Parsing**: Involves searching a parse tree to find the left most derivations of an input stream by using a top-down expansion. Examples include LL parsers and recursive-descent parsers.
* **Bottom-Up Parsing**: Involves rewriting the input back to the start symbol. This type of parsing is also known as shift-reduce parsing. One example is a LR parser.

The inverse of templating is parsing: defining the structure and extracting the data. The inverse of templating is parsing: defining the structure and extracting the data. When parsing, the model must be determined from the raw representation. Parser may refer to both the program that performs the entire process and just the proper parser, which analyzes the tokens generated by the entire process and just the proper parser in the sense of parsing. Scanning and parsing are the two phases of the parser project.

The scanner takes a string and turns it into a token list. The input text consists of C++ statements such as if, else, for, while, variable/constant declarations, assignment, expressions, and other statements. In any case, you're referring to the parser. Its output is typically a tree, which is an ordered code structure. A parse tree or an abstract syntax tree may be used. They're both trees, but the degree to which they reflect the actual code written and the parser's intermediate elements differs. The distinction between the two can be hazy at times; we'll look at their differences in more detail in a later paragraph.

The parser needs the lexer since it does not function directly on the text but rather on the lexer's performance. A lexer and a parser function in tandem: the lexer scans the input and generates matching tokens, while the parser scans the tokens and generates the parsing result. Parser may refer to both the program that performs the entire process and only the proper parser that analyzes the tokens provided by the lexer in the sense of parsing.

Functions and data structures:

If else statements and do when statements were used. A list is a data structure that serves as a container for elements that can be added, removed, or used to perform operations within a sequence or iteration. There are facts, laws, and queries in prolog. Predicates are the terms used to describe the relationships between objects. When these relationships are believed to be valid all of the time, they can be considered truth. If and only if the predicate is valid, clauses are also the relation between objects. This is often referred to as a norm. Queries are what happen when a consumer asks a question about the truth.

Graphical user interface

Description automatically generated with medium confidence

BNF Rules

expr -> expr + term | expr – term | term

term -> term\*power | term / power | term % power | power

power -> factor ^ power | factor

factor -> (expr) | number

num -> number digit | digit

digit -> 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 0

Exp -> List2 | List3 + (Exp1 Exp2) | Exp

Exp -> List2 | List3 - (Exp1 Exp2) | Exp

Exp -> List2 | List3 \* (Exp1 Exp2) | Exp

Exp -> List2 | List3 / (Exp1 Exp2) | Exp

Exp -> List2 | List3 % (Exp1 Exp2) | Exp

Exp -> List2 | List3 > (Exp1 Exp2) | Exp

Exp -> List2 | List3 < (Exp1 Exp2) | Exp

Prolog Code with updated statements:

These statements are being added to original code:

for(exp.statement); %for loop

remainder(exp,exp); % &

s\_statement(["switch"|List6],List1,switch(Exp,Statement1,Statement2,Statement3)):-

s\_exp(List4,List5,Exp),

List2=["case1"|List4],

s\_statement(List2,List3,Statement1),

List4=["case2"|List1],!,

s\_statement(List4,List5,Statement2),

List6=["default"|List7],!,

s\_statement(List6,List7,Statement3),

List8=["endSwitch"|List9].

s\_statement(["if"|List1],List7,if\_then\_else(Exp,Statement1,Statement2)):-

s\_exp(List1,List2,Exp),

List2=["then"|List3],

s\_statement(List3,List4,Statement1),

List4=["else"|List5],!,

s\_statement(List5,List6,Statement2),

List6=["fi"|List7].

Code:

DOMAINS

toklist = string\*

PREDICATES

tokl(string,toklist)

CLAUSES

tokl(Str,[H|T]):-

fronttoken(Str,H,Str1),!,

tokl(Str1,T).

tokl(\_,[]).

/\* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \*

\* This second part of the program is the parser \*

\* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \*/

DOMAINS

program = program(statementlist)

statementlist = statement\*

/\* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \*

\* Definition of what constitutes a statement \*

\* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \*/

statement = if\_Then\_Else(exp,statement,statement);

if\_Then(exp,statement);

while(exp,statement); %for while loop

switch(exp,statement,statement,statement); %for switch

assign(id,exp)

/\* \* \* \* \* \* \* \* \* \* \* \* \* \* \*

\* Definition of expression \*

\* \* \* \* \* \* \* \* \* \* \* \* \* \* \*/

exp = plus(exp,exp);

minus(exp,exp);

var(id);

int(integer);

multiply(exp,exp);

divide(exp,exp);

greaterThan(exp,exp);

lessThan(exp,exp);

modulus(exp,exp)

id = string

PREDICATES

s\_program(toklist,program)

s\_statement(toklist,toklist,statement)

s\_statementlist(toklist,toklist,statementlist)

s\_exp(toklist,toklist,exp)

s\_exp1(toklist,toklist,exp,exp)

s\_exp2(toklist,toklist,exp)

CLAUSES

s\_statement(["switch"|List6],List1,switch(Exp,Statement1,Statement2,Statement3)):-

s\_exp(List4,List5,Exp),

List2=["case1"|List4],

s\_statement(List2,List3,Statement1),

List4=["case2"|List1],!,

s\_statement(List4,List5,Statement2),

List6=["default"|List7],!,

s\_statement(List6,List7,Statement3),

List8=["endSwitch"|List9].

s\_statement(["if"|List1],List7,if\_then\_else(Exp,Statement1,Statement2)):-

s\_exp(List1,List2,Exp),

List2=["then"|List3],

s\_statement(List3,List4,Statement1),

List4=["else"|List5],!,

s\_statement(List5,List6,Statement2),

List6=["fi"|List7].

s\_program(List1,program(StatementList)):-

s\_statementlist(List1,List2,StatementList),

List2=[].

s\_statementlist([],[],[]):-!.

s\_statementlist(List1,List4,[Statement|Program]):-

s\_statement(List1,List2,Statement),

List2=[";"|List3],

s\_statementlist(List3,List4,Program).

s\_statement(["if"|List1],List7,if\_then\_else(Exp,Statement1, Statement2)):-

s\_exp(List1,List2,Exp),

List2=["then"|List3],

s\_statement(List3,List4,Statement1),

List4=["else"|List5],!,

s\_statement(List5,List6,Statement2),

List6=["fi"|List7].

s\_statement(["if"|List1],List5,if\_then(Exp,Statement)):-!,

s\_exp(List1,List2,Exp),

List2=["then"|List3],

s\_statement(List3,List4,Statement),

List4=["fi"|List5].

s\_statement(["do"|List1],List4,while(Exp,Statement)):-!,

s\_statement(List1,List2,Statement),

List2=["while"|List3],

s\_exp(List3,List4,Exp).

s\_statement([ID|List1],List3,assign(Id,Exp)):-

isname(ID),

List1=["="|List2],

s\_exp(List2,List3,Exp).

s\_exp(LIST1,List3,Exp):-

s\_exp2(List1,List2,Exp1),

s\_exp1(List2,List3,Exp1,Exp).

s\_exp1(["+"|List1],List3,Exp1,Exp):-!,

s\_exp2(List1,List2,Exp2),

s\_exp1(List2,List3,plus(Exp1,Exp2),Exp).

s\_exp1(["-"|List1],List3,Exp1,Exp):-!,

s\_exp2(List1,List2,Exp2),

s\_exp1(List2,List3,minus(Exp1,Exp2),Exp).

s\_exp1(List,List,Exp,Exp).

s\_exp2([Int|Rest],Rest,int(I)):-

str\_int(Int,I),!.

s\_exp2([Id|Rest],Rest,var(Id)):-

isname(Id).

Goal tokl("b=2; if b then a=1 else a=2 fi; do a=a-1 while a;",Ans),

s\_program(Ans,Res).

Execution:

The software was successfully compiled after being run through the compiler. A collection of tokens was fed into the scanner, which was then converted into statements. The first token's parser test resulted in a legal argument, and the remaining tokens were checked once more.

Graphical user interface, text, application

Description automatically generated

Output:

Ans=["b","=","2",";","if","b","then","a","=","1","else","a","=","2","fi",";","do","a","=","a","-","1","while","a",";"], Res=program([assign("b",int(2)),if\_then\_else(var("b"),assign("a",int(1)),assign("a",int(2))),while(var("a"),assign("a",minus(var("a"),int(1))))])

1 Solution

Finally, we completed the project effectively with the help of the professor's lectures videos and power point presentations, as well as a few websites (listed below). It was initially difficult to identify the flaws, but after looking over the problem again and amending the text, everything worked great. The project provided a solid foundation in parsing, BNF, Prolog, and C++ coding. The application appears to be attempting to illustrate how to utilize if else, while, for, and assignments statements in visual prolog.

Code in C++:

#include <stdbool.h>

#include <stdio.h>

#include <string.h>

#include <stdlib.h>

// Returns 'true' if the character is a DELIMITER.

bool isDelimiter(char ch)

{

   if (ch == ' ' || ch == '+' || ch == '-' || ch == '\*' ||

       ch == '/' || ch == ',' || ch == ';' || ch == '>' ||

       ch == '<' || ch == '=' || ch == '(' || ch == ')' ||

       ch == '[' || ch == ']' || ch == '{' || ch == '}')

       return (true);

   return (false);

}

// Returns 'true' if the character is an OPERATOR.

bool isOperator(char ch)

{

   if (ch == '+' || ch == '-' || ch == '\*' ||

       ch == '/' || ch == '>' || ch == '<' ||

       ch == '=')

       return (true);

   return (false);

}

// Returns 'true' if the string is a VALID IDENTIFIER.

bool validIdentifier(char\* str)

{

   if (str[0] == '0' || str[0] == '1' || str[0] == '2' ||

       str[0] == '3' || str[0] == '4' || str[0] == '5' ||

       str[0] == '6' || str[0] == '7' || str[0] == '8' ||

       str[0] == '9' || isDelimiter(str[0]) == true)

       return (false);

   return (true);

}

// Returns 'true' if the string is a KEYWORD.

bool isKeyword(char\* str)

{

   if (!strcmp(str, "if") || !strcmp(str, "else") ||

       !strcmp(str, "while") || !strcmp(str, "do") ||

       !strcmp(str, "break") ||

       !strcmp(str, "continue") || !strcmp(str, "int")

       || !strcmp(str, "double") || !strcmp(str, "float")

       || !strcmp(str, "return") || !strcmp(str, "char")

       || !strcmp(str, "case") || !strcmp(str, "char")

       || !strcmp(str, "sizeof") || !strcmp(str, "long")

       || !strcmp(str, "short") || !strcmp(str, "typedef")

       || !strcmp(str, "switch") || !strcmp(str, "unsigned")

       || !strcmp(str, "void") || !strcmp(str, "static")

       || !strcmp(str, "struct") || !strcmp(str, "goto"))

       return (true);

   return (false);

}

// Returns 'true' if the string is an INTEGER.

bool isInteger(char\* str)

{

   int i, len = strlen(str);

   if (len == 0)

       return (false);

   for (i = 0; i < len; i++) {

       if (str[i] != '0' && str[i] != '1' && str[i] != '2'

           && str[i] != '3' && str[i] != '4' && str[i] != '5'

           && str[i] != '6' && str[i] != '7' && str[i] != '8'

           && str[i] != '9' || (str[i] == '-' && i > 0))

           return (false);

   }

   return (true);

}

// Returns 'true' if the string is a REAL NUMBER.

bool isRealNumber(char\* str)

{

   int i, len = strlen(str);

   bool hasDecimal = false;

   if (len == 0)

       return (false);

   for (i = 0; i < len; i++) {

       if (str[i] != '0' && str[i] != '1' && str[i] != '2'

           && str[i] != '3' && str[i] != '4' && str[i] != '5'

           && str[i] != '6' && str[i] != '7' && str[i] != '8'

           && str[i] != '9' && str[i] != '.' ||

           (str[i] == '-' && i > 0))

           return (false);

       if (str[i] == '.')

           hasDecimal = true;

   }

   return (hasDecimal);

}

// Extracts the SUBSTRING.

char\* subString(char\* str, int left, int right)

{

   int i;

   char\* subStr = (char\*)malloc(

               sizeof(char) \* (right - left + 2));

   for (i = left; i <= right; i++)

       subStr[i - left] = str[i];

   subStr[right - left + 1] = '\0';

   return (subStr);

}

// Parsing the input STRING.

void parse(char\* str)

{

   int left = 0, right = 0;

   int len = strlen(str);

   while (right <= len && left <= right) {

       if (isDelimiter(str[right]) == false)

           right++;

       if (isDelimiter(str[right]) == true && left == right) {

           if (isOperator(str[right]) == true)

               printf("'%c' IS AN OPERATOR\n", str[right]);

           right++;

           left = right;

       } else if (isDelimiter(str[right]) == true && left != right

               || (right == len && left != right)) {

           char\* subStr = subString(str, left, right - 1);

           if (isKeyword(subStr) == true)

               printf("'%s' IS A KEYWORD\n", subStr);

           else if (isInteger(subStr) == true)

               printf("'%s' IS AN INTEGER\n", subStr);

           else if (isRealNumber(subStr) == true)

               printf("'%s' IS A REAL NUMBER\n", subStr);

           else if (validIdentifier(subStr) == true

                   && isDelimiter(str[right - 1]) == false)

               printf("'%s' IS A VALID IDENTIFIER\n", subStr);

           else if (validIdentifier(subStr) == false

                   && isDelimiter(str[right - 1]) == false)

               printf("'%s' IS NOT A VALID IDENTIFIER\n", subStr);

           left = right;

       }

   }

   return;

}

// DRIVER FUNCTION

int main()

{

   // maximum legth of string is 100 here

   char str[100] = "int a = b + 1c; ";

   parse(str); // calling the parse function

   return (0);

}

Output:

Text

Description automatically generated

User Doc.

* 1. Compile Program using “g++ parse.cpp”
  2. Run the program, using “./a.out.”
  3. Run code in visual prolog, | options -> build -> test goal |

Summary:

Finally, we completed the project effectively with the help of the professor's lectures videos and power point presentations, as well as a few websites (listed below). It was initially difficult to identify the flaws, but after looking over the problem again and amending the text, everything worked great. The project provided a solid foundation in parsing, BNF, Prolog, and C++ coding. The application appears to be attempting to illustrate how to utilize if else, while, for, and assignments statements in visual prolog.

Citation :

1. <https://www.geeksforgeeks.org/c-program-detect-tokens-c-program/>
2. <https://www.tutorialspoint.com/compiler_design/compiler_design_lexical_analysis.htm>