Assignment Report
On
Detecting Simple Syntax Errors

CSE-0302 Summer 2021

Rebecca Sultana
Dept. of CSE
State University of Bangladesh (SUB)
Dhaka, Bangladesh
rebecca.sultana.riya@gmail.com

***Abstract*—Syntax errors are very common in source programs. The main purpose of this session is to write programs to detect and report simple syntax errors.**

Assignment 1. Detecting Simple Syntax Errors

I. Introduction

A syntax error in computer science is an error in the syntax of a coding or programming language, entered by a programmer. Syntax errors are caught by a software program called a compiler, and the programmer must fix them before the program is compiled and then run.

II. Syntax Error

During the syntax analysis phase, this type of error appears. Syntax error is found during the execution of the program. Some syntax error can be:

- Error in structure
- Missing operators
- Unbalanced parenthesis

When an invalid calculation enters into a calculator then a syntax error can also occurs. This can be caused by entering several decimal points in one number or by opening brackets without closing them.

- For example 1: Using ”=” when ”==” is needed.

```
16 if (number=200)
17 count ¡¡ ”number is equal to 20”;
18 else
19 count ¡¡ ”number is not equal to 200”
```

The following warning message will be displayed by many compilers:

Syntax Warning: In this code, if expression used the equal sign which is actually an assignment operator not the relational operator which tests for equality. Due to the assignment operator, number is set to 200 and the expression number=200 are always true because the expression’s value is actually 200. For this example the correct code would be:
16 if (number==200)

- Example 2: Missing semicolon:

int a = 5 // semicolon is missing

Compiler message: ab.java:20: ’;’ expected
int a = 5

- Example 3: Errors in expressions:

```
x = (3 + 5; // missing closing parenthesis ’)’
y = 3 + * 5; // missing argument between ’+’ and ’*’
```

III. Methodology

Programming is basically solving a particular problem by giving coded instructions to the computer. Furthermore, the whole scenario of the programming cycle involves writing, testing, troubleshooting, debugging, and maintaining a computer program. Moreover, a good program should have clarity and simplicity of expressions, should make use of the proper name of identifiers, contain comments, and have a proper indentation. Besides, it should be free from all types of errors such as syntax errors, run time errors, and logical errors.

```c
#include <stdio.h>
#include <stdlib.h>

#include<stdio.h>

int brace,brack,paren;

void incomment();
void inquote(int c);
void search(int c);

int main(void)
{
    int c;

    extern int brace, brack, paren;

    while((c=getchar())!=EOF)
        if( c == '/')
            if((c=getchar())== '*')
                incomment();
            else
                search(c);
        else if( c == '\'' || c == '"')
            inquote(c);
        else
            search(c);

    if( brace < 0)
    {
        printf("Unmatched Braces\n");
        brace = 0;
```

Fig. 1. C code

```c
    }
    else if( brack < 0)
    {
        printf("Unmatched brackets\n");
        brack = 0;
    }
    else if( paren < 0)
    {
        printf("Unmatched parenthesis\n");
        paren = 0;
    }

    if(brace > 0)
        printf("Unmatched braces\n");
    else if(brack > 0)
        printf("Unmatched brackets\n");
    else if(paren > 0)
        printf("Unmatched parenthesis\n");

    return 0;
}

void incomment()
{
    int c,d;

    c = getchar();
    d = getchar();

    while(c != '*' || d != '/')
    {
        c = d;
        d = getchar();
    }
}
```

Fig. 2. C code

```c
    while(c != '*' || d != '/')
    {
        c = d;
        d = getchar();
    }
}

void inquote(int c)
{
    int d;

    putchar(c);

    while((d=getchar())!=c)
        if( d == '\')
            getchar();
}

void search(int c)
{
    extern int brace,brack,paren;

    if ( c == '(')
        --brace;
    else if ( c == ')')
        ++brace;
    else if( c == '[')
        --brack;
    else if( c == ']')
        ++brack;
    else if( c == '(')
        --paren;
    else if( c == ')')
        ++paren;
}
```

Fig. 3. C code

Assignment Report
On
CFGs for Parsing

CSE-0302 Summer 2021

Rebecca Sultana
Dept. of CSE
State University of Bangladesh (SUB)
Dhaka, Bangladesh
rebecca.sultana.riya@gmail.com

***Abstract*—In this session, we implement a simple recursive descent parser to parse a number of types of statements after exercising with simpler CFGs..**

Assignment 1. Use of CFGs for Parsing

I. Introduction

A context-free grammar is a set of recursive rules used to generate patterns of strings. ... CFG's are used to describe programming languages and parser programs in compilers can be generated automatically from context-free grammars. Two parse trees that describe CFGs that generate the string **"x + y * z"..**

II. components of CFGs

Context-free grammars have the following components:

- A set of terminal symbols which are the characters that appear in the language/strings generated by the grammar. Terminal symbols never appear on the left-hand side of the production rule and are always on the right-hand side.
- A set of nonterminal symbols (or variables) which are placeholders for patterns of terminal symbols that can be generated by the nonterminal symbols. These are the symbols that will always appear on the left-hand side of the production rules, though they can be included on the right-hand side. The strings that a CFG produces will contain only symbols from the set of nonterminal symbols.
- A set of production rules which are the rules for replacing nonterminal symbols. Production rules have the following form: variable string of variables and terminals.
- A start symbol which is a special nonterminal symbol that appears in the initial string generated by the grammar.

III. Formal Definition

A context-free grammar can be described by a four-element tuple (V, $\Sigma, R, S) where$

- V is a finite set of variables (which are non-terminal);
- Sigma is a finite set (disjoint from V) of terminal symbols;
- R is a set of production rules where each production rule maps a variable to a string s epsilon (V union sigma);
- S (which is in V) which is a start symbol)

IV. Methodology

We can think of using CFGs to parse various language constructs in the token streams freed from simple syntactic and semantic errors, as it is easier to describe the constructs with CFGs. But CFGs are hard to apply practically.

V. example of CFGs

Context-free grammars can be modeled as parse trees. The nodes of the tree represent the symbols and the edges represent the use of production rules. The leaves of the tree are the end result that make up the string the grammar is generating with that particular sequence of symbols and production rules.

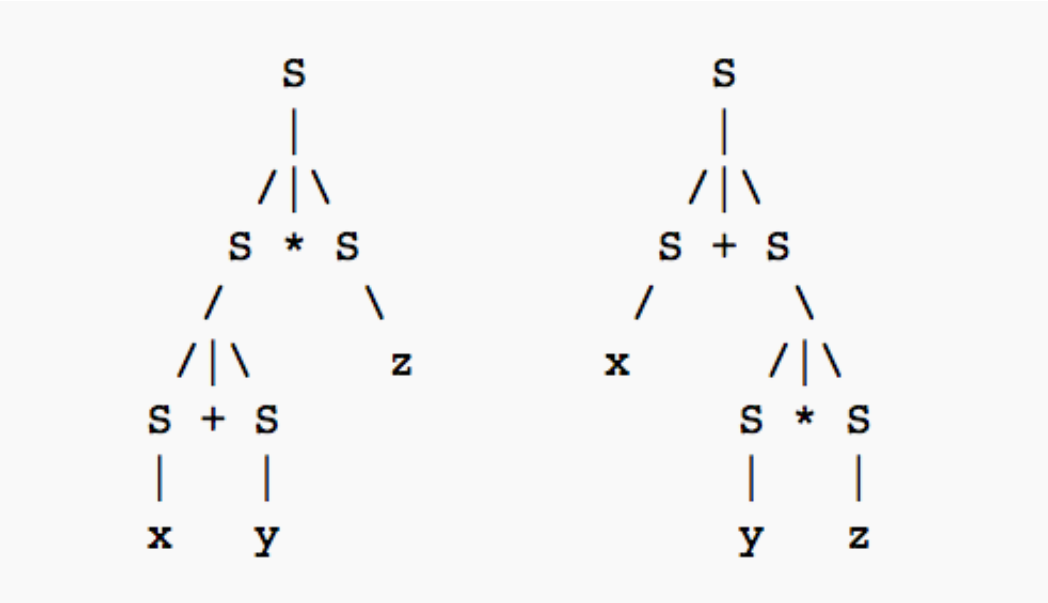

Fig. 1. Two parse trees that describe CFGs that generate the string ”x + y * z”.

Assignment Report
On
Predictive Parsing

CSE-0302 Summer 2021

Rebecca Sultana
Dept. of CSE
State University of Bangladesh (SUB)
Dhaka, Bangladesh
rebecca.sultana.riya@gmail.com

***Abstract*—Manual implementation of LL(1) and LR(1) parsing algorithms.**

Assignment 1. : Predictive Parsing

I. Introduction

Predictive parsing is a special form of recursive descent parsing, where no backtracking is required, so this can predict which products to use to replace the input string. Non-recursive predictive parsing or table-driven is also known as LL(1) parser. This parser follows the leftmost derivation (LMD).

LL(1):
here, first L is for Left to Right scanning of inputs,
the second L is for left most derivation procedure,
1 = Number of Look Ahead Symbols

II. LL(1) Algorithm

Algorithm for non recursive Predictive Parsing:

The main Concept **Maps to** With the help of FIRST() and FOLLOW() sets, this parsing can be done using just a stack that avoids the recursive calls.

For each rule, A **Maps to** x in grammar G:

- For each terminal 'a' contained in FIRST(A) add A **Maps to** x to M[A, a] in the parsing table if x derives 'a' as the first symbol.
- If FIRST(A) contains null production for each terminal 'b' in FOLLOW(A), add this production (A **Maps to** null) to M[A, b] in the parsing table.

III. The Procedure

1. In the beginning, the pushdown stack holds the start symbol of the grammar G.

2. At each step a symbol X is popped from the stack:
if X is a terminal then it is matched with the lookahead and lookahead is advanced one step,

if X is a nonterminal symbol, then using lookahead and a parsing table (implementing the FIRST sets) a production is chosen and its right-hand side is pushed into the stack.

3. This process repeats until the stack and the input string become null (empty).

IV. Advantages Disadvantages

Advantages of Predictive Parser:

- the algorithm is simple enough that we can use it to construct parsers by hand.
- we don't need automatic tools.

Disadvantages of Predictive Parser:

- To remove this recursion, we use LL-parser, which uses a table for lookup.
- Doing optimization may not be as simple as the complexity of grammar grows.
- It is inherently a recursive parser, so it consumes a lot of memory as the stack grows.

V. diagram

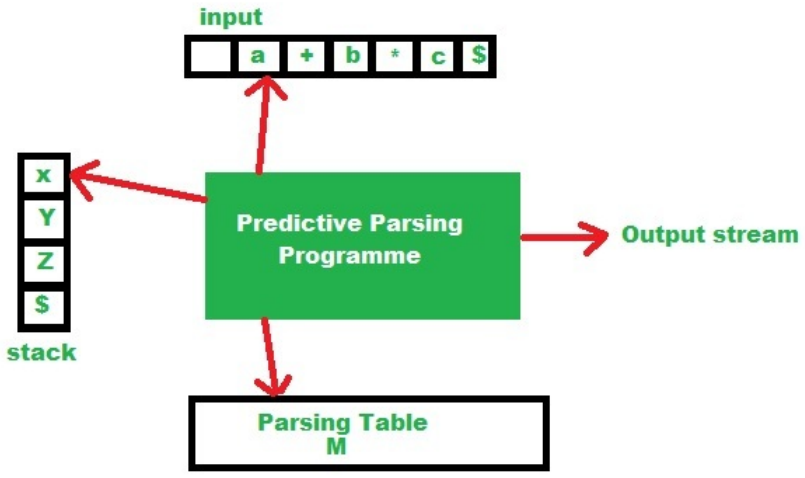

fig: Non Recursive Parser Model

Fig. 1. Non-recursive parser model diagram