Manil S. Malla

wcrecent@gmail.com

To convert Infix expression into postfix expression using predictive parser based on grammar.

Compiler Lab 1

Infix to Postfix expression.

* **Introduction**
* Syntax Directed Definition

Syntax Directed Definition (SDD) is a kind of abstract specification. It is generalization of context free grammar in which each grammar production X –> a is associated with it a set of production rules of the form s = f(b1, b2, ……bk) where s is the attribute obtained from function f. The attribute can be a string, number, type or a memory location. Semantic rules are fragments of code which are embedded usually at the end of production and enclosed in curly braces ({ }).

Example:

E --> E1 + T { E.val = E1.val + T.val}

* Syntax Directed Translation

Syntax-directed translation refers to a method of [compiler](https://en.wikipedia.org/wiki/Compiler) implementation where the source language translation is completely driven by the [parser](https://en.wikipedia.org/wiki/Parser).

A common method of syntax-directed translation is translating a string into a sequence of actions by attaching one such action to each rule of a [grammar](https://en.wikipedia.org/wiki/Grammar). Thus, parsing a string of the grammar produces a sequence of rule applications. SDT provides a simple way to attach [semantics](https://en.wikipedia.org/wiki/Semantics) to any such [syntax](https://en.wikipedia.org/wiki/Syntax).

Syntax-directed translation fundamentally works by adding actions to the productions in a [context-free grammar](https://en.wikipedia.org/wiki/Context-free_grammar), resulting in a Syntax-Directed Definition (SDD).[[2]](https://en.wikipedia.org/wiki/Syntax-directed_translation#cite_note-Alfred-2) Actions are steps or procedures that will be carried out when that production is used in a derivation. A grammar specification embedded with actions to be performed is called a *syntax-directed translation scheme* (sometimes simply called a 'translation scheme'.)

* Predictive Parsing

The goal of predictive parsing is to construct a top-down parser that never backtracks.

[ *Backtracking is a general*[*algorithm*](https://en.wikipedia.org/wiki/Algorithm)*for finding all (or some) solutions to some*[*computational problems*](https://en.wikipedia.org/wiki/Computational_problem)*, notably*[*constraint satisfaction problems*](https://en.wikipedia.org/wiki/Constraint_satisfaction_problem)*, that incrementally builds candidates to the solutions, and abandons a candidate ("backtracks") as soon as it determines that the candidate cannot possibly be completed to a valid solution.*

*The classic textbook example of the use of backtracking is the*[*eight queens puzzle*](https://en.wikipedia.org/wiki/Eight_queens_puzzle)*, that asks for all arrangements of eight*[*chess*](https://en.wikipedia.org/wiki/Chess)[*queens*](https://en.wikipedia.org/wiki/Queen_(chess))*on a standard*[*chessboard*](https://en.wikipedia.org/wiki/Chessboard)*so that no queen attacks any other. In the common backtracking approach, the partial candidates are arrangements of k queens in the first k rows of the board, all in different rows and columns. Any partial solution that contains two mutually attacking queens can be abandoned.*]

To do so, we must transform a grammar in two ways:

1. eliminate left recursion, and
2. perform left factoring.

These rules eliminate most common causes for backtracking although they do not guarantee a completely backtrack-free parsing (called LL(1) as we will see later).

Consider this grammar:

A ::= A a

| b

It recognizes the regular expression ba\*. The problem is that if we use the first production for top-down derivation, we will fall into an infinite derivation chain. This is called left recursion. But how else can you express ba\*? Here is an alternative way:

A ::= b A'

A' ::= a A'

|

where the third production is an empty production (ie, it is A' ::= ). That is, A' parses the RE a\*. Even though this CFG is recursive, it is not left recursive. In general, for each nonterminal X, we partition the productions for X into two groups: one that contains the left recursive productions, and the other with the rest. Suppose that the first group is:

X ::= X a1

...

X ::= X an

while the second group is:

X ::= b1

...

X ::= bm

where a, b are symbol sequences. Then we eliminate the left recursion by rewriting these rules into:

X ::= b1 X'

...

X ::= bm X'

X' ::= a1 X'

...

X' ::= an X'

X' ::=

* Left Recursive Grammer

A grammar is left-recursive if and only if there exists a nonterminal symbol {\displaystyle A} that can derive to a [sentential form](https://en.wikipedia.org/wiki/Formal_grammar#The_semantics_of_grammars)

[*A sentential form is a member of {\displaystyle (\Sigma \cup N)^{\*}} that can be derived in a finite number of steps from the start symbol {\displaystyle S}; that is, a sentential form is a member of {\displaystyle \left\{w\in (\Sigma \cup N)^{\*}\mid S{\overset {\*}{\underset {G}{\Rightarrow }}}w\right\}}. A sentential form that contains no nonterminal symbols (i.e. is a member of {\displaystyle \Sigma ^{\*}}) is called a sentence.*]

with itself as the leftmost symbol.

 Symbolically,

{\displaystyle A\Rightarrow ^{+}A\alpha },

where {\displaystyle \Rightarrow ^{+}} indicates the operation of making one or more substitutions, and {\displaystyle \alpha } is any sequence of terminal and nonterminal symbols.

* **Objective**
* Translate an infix expression into postfix expression using predictive parser based on grammar.

Expr -> term R

R -> + term {print(‘+’)} R

R -> - term {print(‘-’)} R

R -> Є

Term -> 0 {print(‘0’)}

Term -> 1 {print(‘1’)}

.

.

.

Term -> 9 {print(‘9’)}

* **Code**

*#include<stdio.h>*

*#include<conio.h>*

*#include<ctype.h>*

*intlookahead;*

*int main()*

*{*

*lookahead = getchar();*

*expr();*

*putchar('\n');*

*getch();*

*return 0;*

*}*

*voidexpr()*

*{*

*term();*

*while(1)*

*{*

*if(lookahead == '+')*

*{*

*match('+');*

*term();*

*putchar('+');*

*}*

*else if(lookahead == '-')*

*{*

*match('-');*

*term();*

*putchar('-');*

*}*

*else*

*{*

*break;*

*}*

*}*

*}*

*void term()*

*{*

*if(isdigit(lookahead))*

*{*

*putchar(lookahead);*

*match(lookahead);*

*}*

*else*

*{*

*error();*

*}*

*}*

*void match()*

*{*

*int t;*

*if(lookahead == t)*

*{*

*lookahead = getchar();*

*}*

*else*

*{*

*error();*

*}*

*}*

*void error()*

*{*

*printf("\n\n\t\t\tSyntax Error!!!");*

*exit(1);*

*}*

* Output

Input = 9+5-2

Output = 95+2-