Lexical Analysis

#include <iostream>

#include <cctype>

#include <string>

using namespace std;

int main() {

string s;

cout << "Enter the code:\n";

getline(cin, s);

for (size\_t i = 0; i < s.size();) {

if (isspace(s[i])) { i++; continue; }

if (isalpha(s[i])) {

string id;

while (isalnum(s[i])) id += s[i++];

cout << "Identifier: " << id << endl;

} else if (isdigit(s[i])) {

string num;

while (isdigit(s[i])) num += s[i++];

cout << "Number: " << num << endl;

} else {

cout << "Symbol: " << s[i++] << endl;

}

}

return 0;

}

In : Enter the code:

int x = 10;

Symbol Table

#include <iostream>

#include <vector>

#include <sstream>

#include <string>

using namespace std;

struct Symbol {

string name;

string type;

string address;

int index;

string value;

string next;

string status;

};

int main() {

vector<Symbol> table;

string line;

cout << "Enter declarations (type 'end' to finish):\n";

int addr = 1000;

while (getline(cin, line)) {

if (line == "end") break;

stringstream ss(line);

string type, word;

ss >> type;

while (ss >> word) {

// Remove commas and semicolons

if (word.back() == ',' || word.back() == ';') word.pop\_back();

Symbol sym;

sym.name = word;

sym.type = type;

sym.address = to\_string(addr);

addr += 4; // Assuming each variable takes 4 bytes

sym.index = table.size();

sym.value = "0";

sym.next = "-";

sym.status = "active";

table.push\_back(sym);

}

}

cout << "\nSymbol Table:\n";

cout << "Name\tType\tAddress\tIndex\tValue\tNext\tStatus\n";

for (auto &s : table) {

cout << s.name << "\t" << s.type << "\t" << s.address << "\t" << s.index << "\t"

<< s.value << "\t" << s.next << "\t" << s.status << "\n";

}

return 0;

}

In : Enter declarations (type 'end' to finish):

int a, b;

float c;

char d;

End

Predictive Parser

Grammar : E -> T E'

E' -> + T E' | ε

T -> F T'

T' -> \* F T' | ε

F -> ( E ) | id

#include <iostream>

#include <string>

using namespace std;

string input;

int i = 0;

bool E();

bool E\_();

bool T();

bool T\_();

bool F();

bool match(char expected) {

if (input[i] == expected) {

i++;

return true;

}

return false;

}

bool E() {

if (T()) {

if (E\_()) {

return true;

}

}

return false;

}

bool E\_() {

if (match('+')) {

if (T()) {

if (E\_()) {

return true;

}

}

return false;

}

// E' -> ε

return true;

}

bool T() {

if (F()) {

if (T\_()) {

return true;

}

}

return false;

}

bool T\_() {

if (match('\*')) {

if (F()) {

if (T\_()) {

return true;

}

}

return false;

}

// T' -> ε

return true;

}

bool F() {

if (match('(')) {

if (E()) {

if (match(')')) {

return true;

}

}

return false;

} else if (isalnum(input[i])) {

i++;

return true;

}

return false;

}

int main() {

cout << "Enter the string to parse:\n";

getline(cin, input);

input += '$'; // End marker

if (E() && input[i] == '$') {

cout << "Accepted\n";

} else {

cout << "Rejected\n";

}

return 0;

}

In : Enter the string to parse:

id+id\*id

Recursive Parser : Same as above

Direct DFA

#include <iostream>

#include <vector>

#include <map>

#include <set>

using namespace std;

int main() {

int num\_states, num\_transitions, num\_final\_states;

cout << "Enter the number of states:\n";

cin >> num\_states;

cout << "Enter the number of transitions:\n";

cin >> num\_transitions;

map<pair<int, char>, int> transitions;

cout << "Enter the transitions (format: from\_state input\_symbol to\_state):\n";

for (int i = 0; i < num\_transitions; i++) {

int from, to;

char symbol;

cin >> from >> symbol >> to;

transitions[{from, symbol}] = to;

}

cout << "Enter the number of final states:\n";

cin >> num\_final\_states;

set<int> final\_states;

cout << "Enter the final states:\n";

for (int i = 0; i < num\_final\_states; i++) {

int state;

cin >> state;

final\_states.insert(state);

}

cin.ignore(); // Clear input buffer

string input\_string;

cout << "Enter the input string:\n";

getline(cin, input\_string);

int current\_state = 0; // Assuming 0 is the start state

bool accepted = true;

for (char c : input\_string) {

if (transitions.find({current\_state, c}) != transitions.end()) {

current\_state = transitions[{current\_state, c}];

} else {

accepted = false;

break;

}

}

if (accepted && final\_states.count(current\_state)) {

cout << "Accepted\n";

} else {

cout << "Rejected\n";

}

return 0;

}

In : Enter the number of states:

3

Enter the number of transitions:

4

Enter the transitions (format: from\_state input\_symbol to\_state):

0 a 1

1 b 2

2 a 2

2 b 2

Enter the number of final states:

1

Enter the final states:

2

Enter the input string:

aba

Shift Reduce Parser

Grammar : S -> a S b | a b

#include <iostream>

#include <string>

#include <vector>

using namespace std;

int main() {

string input;

cout << "Enter the input string:\n";

cin >> input;

input += '$';

vector<char> stack;

int i = 0;

cout << "Stack\tInput\tAction\n";

while (i < input.size()) {

stack.push\_back(input[i]);

cout << string(stack.begin(), stack.end()) << "\t" << input.substr(i+1) << "\tShift\n";

i++;

// Try to reduce

bool reduced;

do {

reduced = false;

if (stack.size() >= 2 && stack[stack.size() - 2] == 'a' && stack.back() == 'b') {

stack.pop\_back();

stack.pop\_back();

stack.push\_back('S');

cout << string(stack.begin(), stack.end()) << "\t" << input.substr(i) << "\tReduce S -> a b\n";

reduced = true;

} else if (stack.size() >= 3 && stack[stack.size() - 3] == 'a' && stack[stack.size() - 2] == 'S' && stack.back() == 'b') {

stack.pop\_back();

stack.pop\_back();

stack.pop\_back();

stack.push\_back('S');

cout << string(stack.begin(), stack.end()) << "\t" << input.substr(i) << "\tReduce S -> a S b\n";

reduced = true;

}

} while (reduced);

}

if (stack.size() == 2 && stack[0] == 'S' && stack[1] == '$') {

cout << "Input Accepted\n";

} else {

cout << "Input Rejected\n";

}

return 0;

}

In : Enter the input string:

aabb

LR Parser

Grammar : E -> E + T | T

T -> T \* F | F

F -> ( E ) | id

#include <iostream>

#include <string>

#include <stack>

using namespace std;

// Parsing table and grammar are hardcoded for simplicity.

int main() {

string input;

cout << "Enter the input string (e.g., id+id\*id):\n";

cin >> input;

input += "$";

stack<int> states;

states.push(0);

int i = 0;

string action;

while (true) {

int state = states.top();

char symbol = input[i];

// Simplified action lookup

if (state == 0 && symbol == 'i') {

states.push(5);

i += 2; // Assuming 'id' is two characters

cout << "Shift 'id', push state 5\n";

} else if (state == 5 && (symbol == '+' || symbol == '$')) {

states.pop();

states.push(3);

cout << "Reduce F -> id, push state 3\n";

} else if (state == 3 && symbol == '\*') {

states.push(7);

i++;

cout << "Shift '\*', push state 7\n";

} else if (state == 7 && symbol == 'i') {

states.push(5);

i += 2;

cout << "Shift 'id', push state 5\n";

} else if (state == 5 && (symbol == '+' || symbol == '$')) {

states.pop();

states.push(3);

cout << "Reduce F -> id, push state 3\n";

} else if (state == 3 && (symbol == '+' || symbol == '$')) {

cout << "Accept\n";

break;

} else {

cout << "Error\n";

break;

}

}

return 0;

}

LL1 : Same as Predictive

Operator Precedence Parser

#include <iostream>

#include <string>

#include <stack>

using namespace std;

int prec(char op) {

if (op == '+' || op == '-') return 1;

if (op == '\*' || op == '/') return 2;

if (op == '$') return -1; // End marker

return 0;

}

int main() {

string input;

cout << "Enter the expression:\n";

cin >> input;

input += '$';

stack<char> op\_stack;

op\_stack.push('$');

int i = 0;

cout << "Stack\tInput\tAction\n";

while (i < input.length()) {

char top = op\_stack.top();

char current = input[i];

int top\_prec = prec(top);

int current\_prec = prec(current);

if (isdigit(current)) {

cout << string(1, top) << "\t" << input.substr(i) << "\tShift " << current << "\n";

i++;

op\_stack.push('n'); // 'n' represents a number

} else if (current\_prec > top\_prec) {

op\_stack.push(current);

cout << string(1, top) << "\t" << input.substr(i) << "\tShift " << current << "\n";

i++;

} else {

op\_stack.pop();

cout << string(1, top) << "\t" << input.substr(i) << "\tReduce\n";

}

}

while (op\_stack.top() != '$') {

cout << string(1, op\_stack.top()) << "\t$\tReduce\n";

op\_stack.pop();

}

cout << "Accepted\n";

return 0;

}

In : Enter the expression:

3+4\*5

Ambiguity Detection :

#include <iostream>

#include <string>

using namespace std;

bool isAmbiguous(const string &expr) {

int operatorCount = 0;

for (size\_t i = 0; i < expr.length(); i++) {

if (expr[i] == '+' || expr[i] == '\*') {

operatorCount++;

if (operatorCount >= 2) {

// Check if there are no parentheses enforcing associativity

if (expr.find('(') == string::npos) {

return true;

}

}

}

}

return false;

}

int main() {

string s;

cout << "Enter an expression:\n";

getline(cin, s);

if (isAmbiguous(s)) cout << "The expression is ambiguous.\n";

else cout << "The expression is not ambiguous.\n";

return 0;

}

In : Enter an expression: a+a+a