1. Grammar

روند کلی

روند کلی برنامه به این صورت است که ابتدا شی گرامر از ورودی خوانده می‌شود سپس ساده‌سازی می‌شود، به فرم چامسکی تبدیل می‌شود و در نهایت الگوریتم CYK با توجه به رشته‌ی ورودی روی آن اجرا می‌شود.

```
int main() {
    Grammar < 10000 > g; // 10000 means maximum number of production rules
    cin >> g;
    g.simplify();
    //g.display();
    g.toCNF();
    //g.display();

    string s;
    cin >> s;
    cout << g.CYKMembership(s);
}
```

ورودی گرفتن گرامر

ابتدا همه‌ی خطوط گرامر از ورودی خوانده می‌شوند و سمت چپ production rule از سمت راست آن جدا می‌شود.

در نهایت به کمک داده‌ساختار map سمت چپ هر production rule را به سمت راست آن متناظر می‌کنیم و در شی ذخیره می‌کنیم.

متغیر شروع (Start Variable) را سمت چپ اولین production rule در نظر گرفته و به عنوان یکی از ویژگی‌های شی گرامر ذخیره می‌کنیم.

```
template <int N>
istream& operator >> (istream& in, Grammar < N > &g) {
    int n;
    cin >> n;
    g.n = n;
    cin.ignore();
    for (int i = 0; i < n; i++) {
        getline(cin, g.rawProdRules[i]);

        int j = 0;
        string left;
        for (;j < g.rawProdRules[i].size(); j++) {
            if (g.rawProdRules[i][j] == '-' && g.rawProdRules[i][j + 1] == '>') {
                left = g.rawProdRules[i].substr(0, j - 1);
                break;
            }
        } j += 3;

        string right;
        g.rawProdRules[i] += '|';
        for (;j < g.rawProdRules[i].size(); j++) {
            if (g.rawProdRules[i][j] == ' ') {
                continue;
            }
            if (g.rawProdRules[i][j] == '|') {
                g.prodRules[left].push_back(right);
                right.clear();
                continue;
            }
            right += g.rawProdRules[i][j];
        }
    }
    g.stV = g.rawProdRules[0].substr(0, 3);
}
```

ساده‌سازی گرامر

```
template <int N>
void Grammar<N>::simplify() {
    bool flg = true;
    while (flg) {
        flg = false;
        if (this -> removeNullable() || this -> removeUnitProd() || this -> removeUnreachable() || this -> removeUnterminatable())
            flg = true;
            this -> removeDuplicate();
            this -> removeInvalidProdRules();
            continue;
        }
    }
}
```

تا زمانی که یک Nullable یا Unit یا Useless بتوانیم حذف کنیم اینکار را می‌کنیم و ترتیب انجام این‌کار هم همانطور که از قبل می‌دانستیم Useless، Unit، Nullable انجام می‌دهیم. (که البته برای Useless دو متود جداگانه برای Unreachableها و آن‌هایی که به terminal ختم نمی‌شوند نوشته شده است) هر کدام از متودهای گفته شده یک مقدار boolean خروجی می‌دهند که نشان می‌دهد آیا آن متود توانسته است چیزی را ساده کند یا نه که برای فهمیدن اینکه کی ساده‌سازی به پایان می‌رسد از این خروجی‌ها استفاده می‌کنیم.

بعد از هر عملیات حذف (ساده‌سازی) یکبار بررسی می‌کنیم اگر بین production rule ها تکراری داشتیم آن را حذف می‌کنیم (removeDuplicate) و اگر در production ruleای یک متغیر نامعتبر وجود داشت (متغیری که برایش production ruleای نباشد) آن را هم حذف می‌کنیم (removeInvalidProdRules)

تبدیل به فرم نرمال چامسکی

ابتدا هر terminalای که در بین production ruleها پیدا می‌کنیم را با variable جایگزینی که برایش می‌سازیم جایگزین می‌کنیم. variableهای جایگزین را با استفاده از کاراکترهای خاص ! و $ و... می‌سازیم تا variable تکراری تولید نشود که برای اینکار از عدد اسکی ۳۳ به بعد را استفاده کردیم.

```
map < char, string > terVar;
string newVar; newVar += '<'; newVar += (char)33; newVar += '>';
map < string, vector < string > >::iterator it;
for (it = prodRules.begin(); it != prodRules.end(); it++) { // Assign t
    for (int i = 0; i < it -> second.size(); i++) {
        if (it -> second[i].size() == 1) {
            continue;
        }
        for (int j = 0; j < it -> second[i].size();) {
            if (it -> second[i][j] == '<') {
                j += 3;
            } else {
                if (terVar.find(it -> second[i][j]) == terVar.end()) {
                    terVar[it -> second[i][j]] = newVar; //cout << "tes
                    newVar[1]++;
                }
                j++;
            }
        }
    }
}
```

سپس هر production rule را تا جایی که متغیرهای سمت راست آن به دوتا برسد کاهش می‌دهیم و به ازای هر کاهش یا متغیر جدیدی برای آن می‌سازیم یا اگر قبلا production ruleای برای آن وجود داشته از همان production rule قدیمی استفاده می‌کنیم.

```
map < string, string > newProd;
for (it = prodRules.begin(); it != prodRules.end(); it++) {
    for (int i = 0; i < it -> second.size(); i++) {
        while (it -> second[i].size() > 6) {
            if (newProd.find(it -> second[i].substr(0, 6)) == newProd.end()) {
                newProd[it -> second[i].substr(0, 6)] = newVar;
                prodRules[newVar].push_back(it -> second[i].substr(0, 6));
                newVar[1]++;
            }
            string tmp = it -> second[i].substr(0, 6);
            it -> second[i].erase(0, 6);
            it -> second[i].insert(0, newProd[tmp]);
        }
    }
}
```

الگوریتم CYK Membership

ابتدا یک ماتریس که تعداد سطرها و تعداد ستون‌های آن به اندازه‌ی طول رشته ورودی و هر خانه‌ی آن یک داده‌ساختار set است می‌سازیم.

```
set < string >** dp = new set < string >* [w.size()];
for (int i = 0; i < w.size(); i++) {
    dp[i] = new set < string > [w.size()];
}
```

سپس روی طول زیررشته‌ها و همینطور نقطه‌ی شروع زیررشته for می‌زنیم.

```
for (int l = 1; l <= w.size(); l++) { // l = length of the range
    for (int i = 0; i < l + w.size() - l; i++) { // range [i, i + l - 1]
        int j = i + l - 1; // rightside position
```

در هر مرحله اگر توانستیم production ruleای پیدا کنیم که مستقیماً زیررشته‌ی w[i..j] را تولید کند آن را به dp[i][j] اضافه می‌کنیم.

```
// Base case: rightside includes of just terminals
map < string, vector < string > >::iterator it;
for (it = prodRules.begin(); it != prodRules.end(); it++) {
    for (int k = 0; k < it -> second.size(); k++) {
        if (it -> second[k] == w.substr(i, l)) {
            dp[i][j].insert(it -> first);
            //cout << "Found a terminate! " << "i:" << i <<
        }
    }
}
```

سپس همه‌ی w[i..k], w[k+1..j] ها را در نظر می‌گیریم و همه‌ی زوج مرتب‌های ممکن از dp[i][k] و dp[k+1][j] که با کنار هم گذاشتن آن‌ها به w[i][j] می‌رسیم را می‌یابیم.

```
for (int k = i; k < j; k++) { // dp[i][k], dp[k + 1][j]
    //cout << "range i:" << i << " k:" << k << " j:" << j << endl;
    set < string > targetProds;
    set < string >::iterator it2;
    set < string >::iterator it3;
    for (it2 = dp[i][k].begin(); it2 != dp[i][k].end(); it2++) {
        for (it3 = dp[k + 1][j].begin(); it3 != dp[k + 1][j].end(); it3++) {
            string target1 = *it2;
            target1 += *it3;
            string target2 = *it2;
            target2 += w.substr(k + 1, j - k);
            string target3 = w.substr(i, k - i + 1);
            target3 += *it3;
            //cout << " i,j:" << i << ' ' << j << " target1: " << target1 <<
            targetProds.insert(target1);
            targetProds.insert(target2);
            targetProds.insert(target3);
        }
    }
```

در نهایت اگر از زوج مرتب‌هایی که پیدا کردیم، یکی را در سمت راست یکی از production rulesها یافتیم، سمت چپ آن production rule را به dp[i][j] اضافه می‌کنیم.

```
// In the search of target productions among available production rules
map < string, vector < string > >::iterator it4;
for (it4 = prodRules.begin(); it4 != prodRules.end(); it4++) {
    for (int p = 0; p < it4 -> second.size(); p++) {
        if (targetProds.find(it4 -> second[p]) != targetProds.end()) {
            dp[i][j].insert(it4 -> first);
            //cout << "target acceptable found: " << "i:" << i << " j:"
        }
    }
}
```

```
return dp[0][w.size() - 1].find(stV) != dp[0][w.size() - 1].end() ? "Accepted" : "Rejected" ;
```

جوابی که در پایان از متود CYKMembership خروجی می‌دهیم در صورتی Accepted خواهد بود که متغیر شروع (stV) در dp[0][|w|] باشد یا به عبارت دیگر با شروع از stV بتوانیم زیررشته‌ی w[0..|w|] را بسازیم و در غیر این‌صورت Rejected خواهد بود.

2. Turing Machine

روند کلی

روند کلی برنامه به این صورت است که ابتدا رشته‌ی کدگذاری شده‌ی ماشین تورینگ از ورودی خوانده می‌شود و بعد مطابق فرمت خواسته شده n رشته‌ی ورودی گرفته می‌شوند و به کمک متود proc شی ماشین تورینگ پردازش می‌شوند.

```
int main() {
    Turing TM;
    cin >> TM;
    int n;
    cin >> n;
    cin.ignore();
    for (int i = 0; i < n; i++) {
        string s;
        getline(cin, s);
        cout << TM.proc(s) << endl;
    }
}
```

ورودی گرفتن ماشین تورینگ

ابتدا قسمت‌های مختلف رشته‌ی کدگذاری‌شده جدا می‌شوند و در رشته‌های جداگانه قرار می‌گیرند و سپس در delta ذخیره می‌شوند، که برای ساختار delta از داده‌ساختار map بهره گرفته شده تا هر state و حرف روی tape به حرف، حرکت و state بعدی‌اش متناظر شود. همچنین بلندترین رشته‌ای که برای stateها ورودی می‌گیریم را هم پیدا می‌کنیم و به عنوان final state ذخیره می‌کنیم.

```
istream& operator >> (istream& in, Turing &t) {
    string encoded;
    in >> encoded;
    for (int i = 0; i < encoded.size(); i++) {
        string q1;
        for (; encoded[i] != '0'; i++) {
            q1 += encoded[i];
        } i++;

        string a; // a -> b
        for (; encoded[i] != '0'; i++) {
            a += encoded[i];
        } i++;

        string q2;
        for (; encoded[i] != '0'; i++) {
            q2 += encoded[i];
        } i++;

        string b; // a -> b
        for (; encoded[i] != '0'; i++) {
```

```
t.delta[make_pair(q1, a)] = make_pair(make_pair(q2, b), mov);
if (t.finalState.size() < q1.size()) {
    t.finalState = q1;
}
if (t.finalState.size() < q2.size()) {
    t.finalState = q2;
}
```

جداسازی رشته ورودی ماشین تورینگ و آماده‌سازی برای پردازش

با توجه به اینکه کاراکترهای '0' در ورودی صرفا نقش جداکننده را دارند پس ذخیره کردن آن‌ها عملاً فایده‌ای ندارد. به همین دلیل تک تک زیررشته‌های ماکسیمال تماماً '1' را جداسازی کردیم و در یک vector ذخیره کردیم تا بعدا راحتتر بشود با آن کار کرد.

```
vector < string > w;
if (raw.size() > 0) {
    string tmp;
    for (int i = 0; i < raw.size(); i++) {
        if (raw[i] == '0') {
            w.push_back(tmp);
            tmp.clear();
        } else {
            tmp += raw[i];
        }
    } w.push_back(tmp);
}
```

پردازش رشته‌ی ورودی

```
while (true) {
    if (pos < l || pos > r) {
        cur = "1";
    } else if (pos < 0 || pos >= w.size()) {
        if (outOfPos.find(pos) == outOfPos.end()) {
            cur = "1";
        } else {
            cur = outOfPos[pos];
        }
    } else {
        cur = w[pos];
    }
    pair < string, string > curPr = make_pair(state, cur);
    if (state != finalState && delta.find(curPr) == delta.end()) {
        return "Rejected";
    }
```

برای tape ماشین یک l و r در نظر گرفتیم که مشخص کننده‌ی محدوده‌ی tape است و می‌تواند در طول writeهایی که انجام می‌شود این محدوده گسترش پیدا کند. اگر pos فعلی در محدوده رشته ورودی باشد همانجا read و write انجام می‌شود، اگر در محدوده‌ی رشته‌ی ورودی نباشد ولی چیزی غیر از blank در آن درج کرده باشیم آنگاه در outOfPos قرار می‌گیرد و آنجا read و write می‌کنیم و در غیر این‌صورت هم حتما blank است و ’1‘ در نظر می‌گیریم. اگر جایی برسیم که از state فعلی هیچ transition خروجی مناسبی نداشته باشیم و در state پایانی هم نباشیم، همانجا Rejected اعلام می‌کنیم.

```
if (pos < 0 || pos >= w.size()) {
    l = min(l, pos);
    r = max(r, pos);
    outOfPos[pos] = delta[curPr].first.second;
} else {
    w[pos] = delta[curPr].first.second;
}
if (delta[curPr].second == "1") {
    pos--;
} else {
    pos++;
}
state = delta[curPr].first.first;
if (state == finalState) {
    return "Accepted";
}
```

خطوط مقابل نشان‌دهنده‌ی نحوه‌ی جابجایی l و r و همچنین read و write و حرکت دادن pos به چپ و راست است. همچنین اگر جایی به final state رسیدیم همانجا فوراً Accepted را بر می‌گردانیم.

3. PDA Calculator

روند کلی

```
Grammar < 10000 > g; // 10000 means maximum number of production rules
stringstream in;
in << 5 << '\n';
in << "<S> -> <X> | <S>+<S> | <S>-<S> | <S>*<S> | <S>/<S> | <S>^<S> | (<S>) | sqrt(<S>) | sin(<S>) | cos(<S>) | tan(<S>) | as
in << "<X> -> -<A> | <A>\n";
in << "<A> -> 1<B> | 2<B> | 3<B> | 4<B> | 5<B> | 6<B> | 7<B> | 8<B> | 9<B> | 0 | 0.<C>\n";
in << "<B> -> 0<B> | 1<B> | 2<B> | 3<B> | 4<B> | 5<B> | 6<B> | 7<B> | 8<B> | 9<B> | .<C> | #\n";
in << "<C> -> 0<C> | 1<C> | 2<C> | 3<C> | 4<C> | 5<C> | 6<C> | 7<C> | 8<C> | 9<C> | #\n";
in >> g;
g.simplify();
g.toCNF();

string exp;
getline(cin, exp);
if (checkBadSpaces(exp)) {
exp = removeSpaces(exp);
if (!g.CYKMembership(exp)) {

const string ans = calc(split(exp));
if (ans == "INVALID") {
}
```

ابتدا به کمک کلاس grammar (که برای سوال ۱ نوشته شده بود) گرامر مربوطه را روی رشته‌ی ورودی بررسی کردیم و در صورتی که الگوریتم CYK آن را نپذیرد خروجی INVALID دادیم. قبل از ورودی دادن به گرامر، همه‌ی spaceهای موجود در رشته را پاک می‌کنیم تا کار با رشته آسان‌تر شود. ولی از آن‌جایی که ممکن است برای حالت‌هایی مثل 9 8 – 12 که خروجی باید INVALID باشد اشتباهاً خروجی مربوط به 12-89 در نظر گرفته شود پس این حالت را در تابعی به نام checkBadSpaces بررسی کردیم و سپس سراغ محاسبات رفتیم. در انتهای main هم به کمک تابع split قسمت‌های مختلف رشته‌ی ورودی (مثل عملوندها و عملگرها و پرانتزها) را جدا کردیم و در یک vector جای دادیم و سپس به تابع calc برای محاسبه حاصل پاس دادیم. در نهایت هم اگر در محاسبات میانی حالتی پیش بیاید (مثل عدم تطابق دامنه‌ی توابع tan و asin و...) که باید به INVALID ختم شود، آن را در تابع calc بررسی کردیم.

انجام محاسبات

```
const string calc(vector < string > exp) {
    vector < string > PF = postFix(exp);
    stack < string > stk;
    for (int i = 0; i < PF.size(); i++) {
```

عبارتی که قبلا توسط تابع split جداجدا شده و بصورت چند رشته درآمده را در ابتدای تابع calc به تابع دیگری به نام postFix پاس می‌دهیم که فرمت عبارت ورودی را از فرمت In Fix به Post Fix تغییر دهد، یعنی بجای اینکه یک عملگر دودویی (binary operator) بین دو عملوند خود و عملگر یکه (unary operator) قبل از عملوند خود قرار بگیرند، پس از آن‌ها قرار بگیرند. این نحوه‌ی نمایش عبارت به ما کمک می‌کند تا بتوانیم با پیمایش روی رشته‌ی جدید ابتدا عملوندها و سپس عملگر مربوط را ببینیم که در محاسبه کمک بزرگی به ما است چون می‌توانیم مثلا برای عبارت $4 \times 3 + 2$ ابتدا عملوندها را ببینیم و بعد بر اساس عملگرهایی که داریم و طبق اولویت عملگرها حاصل عبارت را محاسبه کنیم. (که در ادامه این بررسی اولویت‌ها را به کمک پشته انجام داده‌ایم)

تبدیل به فرمت Post Fix

حرف‌ها را یک به یک بررسی کردیم اگر حرف موردبررسی، عملوند باشد در نتیجه می‌گذاریم ولی اگر عملگر یا پرانتز باشد آن را داخل پشته می‌گذاریم و با توجه اولویت عملگرهایی که قرار می‌گیرند تصمیم می‌گیریم اول کدام را داخل نتیجه قرار دهیم.

(اولویت‌های عملگرها را به کمک map به کاراکترشان متناظر کردیم)

```
// If exp[i] was an operand
if (isdigit(exp[i][0]) || exp[i].size() > 1 && !isalpha(exp[i][0])) {
    res.push_back(exp[i]);
}

if (exp[i][0] == '(') {
    stk.push(exp[i]);
}

if (exp[i][0] == ')') {
    while (!stk.empty() && stk.top() != "(") {
        res.push_back(stk.top());
        stk.pop();
    }
    if (!stk.empty()) { // Pop "(" from stack
        stk.pop();
    }
}

// If exp[i] was an operator
if (precedence.find(exp[i][0]) != precedence.end() && (exp[i].size() == 1 || isalpha(exp[i][0]))) {
    if (stk.empty() || stk.top() == "(") {
        stk.push(exp[i]);
    } else {
        while (!stk.empty() && stk.top() != "(" && precedence[exp[i][0]] <= precedence[stk.top()[0]]) {
            res.push_back(stk.top());
            stk.pop();
        }
        stk.push(exp[i]);
    }
}
```

محاسبه به کمک پشته

```
// If PF[i] was an operand
if (isdigit(PF[i][0]) || PF[i].size() > 1 && !isalpha(PF[i][0])) {
    stk.push(PF[i]);
}

// If exp[i] was an binary operator
if (precedence.find(PF[i][0]) != precedence.end() && PF[i].size() == 1) {
    double a = strtod(stk.top().data(), NULL);
    stk.pop();
    double b = strtod(stk.top().data(), NULL);
    stk.pop();
    double res;
    switch (PF[i][0]) {
        case '*': {
            res = b * a;
        } break;
        case '/': {
            if (a == 0) {
                return "INVALID";
            }
            res = b / a;
```

```
// If exp[i] was an unary operator
if (precedence.find(PF[i][0]) != precedence.end() && isalpha(PF[i][0
    double a = strtod(stk.top().data(), NULL);
    stk.pop();
    double res;
    if (PF[i] == "sqrt") {
        if (a < 0) {
            return "INVALID";
        }
        res = sqrt(a);
    } else if (PF[i] == "sin") {
        res = sin(a);
    } else if (PF[i] == "cos") {
        res = cos(a);
    } else if (PF[i] == "tan") {
```

محاسبات مربوط به حاصل‌جواب را در پشته انجام دادیم به این‌صورت که هر عملوندی را از فرمت Post Fixای که بدست‌آورده بودیم به پشته اضافه کردیم و هر وقت عملگر دیدیم با توجه به اینکه binary بوده یا unary یک یا دو عملوند موردنیاز آن را از سر پشته برداشتیم و پس از انجام محاسبه آن را به پشته بازگرداندیم. با اینکار در نهایت فقط یک عملوند در پشته می‌ماند که حاصل تمام عملیات‌های انجام شده است. در حین انجام عملیات اگر عملوند یا عملوندهای موجود برای یک عملگر/تابع در دامنه‌ی مربوط به عملگر/تابع نبودند (مثل تقسیم بر صفر یا ورودی ۰ برای لگاریتم) خروجی تابع و برنامه INVALID خواهد بود.