構文解析 Howto

gistfile1.rst

構文解析 Howto

Author:	draftcode
Date:	2011-11-11T13:18:07+09:00
ID:	289a0136-0c1c-11e1-a06b-040ccee352e6

こうぶん、かいせきー

構文解析なんてやるだけゲーです。99%やるだけゲーです。問題はそれがどれぐらい大きいのかとか、それと複合してほかの問題を解かなければいけない(しかも方程式を解いたりするものがおおい)とかそんなのばっかです。一問だけ難しい構文解析ゲーがありますが、AOJで未だに一人しかACしてないです。そんなの解けなくてもいいです。

ただ、構文解析には人によって、書き方みたいなのがいくつかあるらしくて、基本的な 方針は同じなのに、複雑になっていたりします。このHowtoは複雑なのをできるだけ避 ける書き方をしてみたいと思います。

方針

構文解析というと、字句解析をして、構文木をつくってみたいなガチなのが思いだされ ますが、ICPCで出る構文解析は99%そんなことしなくてもいいはずなので、省きます。 直接文字列をパースして、結果を直接作ります。

例題として普通の四則演算をやりましょう。AOJ0109が簡単です。ICPCの構文解析ゲー はほぼすべての問題が、四則演算の一部分を変えただけの問題です。

四則演算は次のような見方をします:

```
<四則演算の式> ::= <乗算除算の式> (+ or -) <乗算除算の式> (+ or -) ...
<乗算除算の式> ::= <数> (* or /) <数> (* or /) ...
```

```
<数>            ::= ...
```

これに従うと、乗算除算の方が優先されるというルールにそのまま適合します。しかも それぞれが式なので、その式を計算した結果を構文解析の結果として返してあげればよ いのです。例えば:

```
2*3 + 4*5 + 6*7
-> 6 + 20 + 42
-> 68
```

という風に、部分式を適当に評価してあげるということをすることで、勝手に四則演算 の優先順位が守られます。この方針で行きましょう。

テンプレート

なにはともあれこれを書きましょう:

```
#include <string>
#include <cctype>
typedef string::const_iterator State;
class ParseError {};
```

構文解析は途中経過の状態をどんどん進めながらやっていきますので状態が必要です。 あと途中でエラーだったというときに投げるための例外を定義しておきましょう。

部分式のパース

先の方針だと、<四則演算の式>と<乗算除算の式>と<数>を分けてパースしてあげること で、四則演算の優先順位を守らせるというものでしたので、それに従って次のように関 数を書きます:

```
// 四則演算の式をパースして、その評価結果を返す。
int expression(State &begin) {
}

// 乗算除算の式をパースして、その評価結果を返す。
int term(State &begin) {
}

// 数字の列をパースして、その数を返す。
int number(State &begin) {
}
```

expressionとかtermは構文解析でよく使われる名前なので、適当に従っておきましょう 。3問ぐらい解けばなれます。

ここで重要なのはStateが参照であるということと、結果がintであるということです。パースした結果、ここまでパースできましたというのを返さなければいけないのですが、Stateの参照を渡しておくことで、begin++とかやりながらパースしていけば、勝手に 呼び出し元にどこまでパースしたのかというのがわかるようになります。またintを返 すのは、ここでほしいのが評価結果なので、返すべきは数値です。なのでintになりま す。問題によっては、「この部分の式をパースした結果、こういうのがほしい!」みたいなのがありますので、この部分を変更することになります。(ただし50%ぐらいは数値 です。)

数のパース

さすがにやるだけです。ここでcctypeのisdigitを使っていきましょう:

```
int number(State &begin) {
    int ret = 0;

    while (isdigit(*begin)) {
        ret *= 10;
        ret += *begin - '0';
        begin++;
    }

    return ret;
}
```

beginは参照でしたので、begin++とすると、文字列のイテレータがずれていき、しかも それが呼び出し元にまで影響します。なので、numberがリターンしたときには、数とい うのが、パースできるところまでパースされていて、現在は数字ではない、isdigitが 真でない文字を指しているはずです。

乗算除算の式のパース

乗算除算の式は次のようなものでした:

```
<乗算除算の式> ::= <数> (* or /) <数> (* or /) ...
```

これに従って関数を埋めてみましょう:

```
int term(State &begin) {
    int ret = number(begin);

    for (;;) {
        if (*begin == '*') {
```

```
            begin++;
            ret *= number(begin);
        } else if (*begin == '/') {
            begin++;
            ret /= number(begin);
        } else {
            break;
        }
    }

    return ret;
}
```

上の式と見比べてみてください。本当に書き下すだけでできます。

四則演算の式のパース

上の関数を埋めていきましょう:

```
int expression(State &begin) {
    int ret = term(begin);

    for (;;) {
        if (*begin == '+') {
            begin++;
            ret += term(begin);
        } else if (*begin == '-') {
            begin++;
            ret -= term(begin);
        } else {
            break;
        }
    }

    return ret;
}
```

ほぼ同じです。

構文解析の開始

ここまでで構文解析部分はできてしまいました。最後に文字列を読み込んでパースさせてみましょう:

```
int main(void) {
    int N;
    cin >> N;
    cin.ignore()
```

```
    for (int i = 0; i < N; i++) {
        string s;
        getline(cin, s);

        State begin = s.begin();
        int ans = expression(begin);
        cout << ans << endl;
    }
    return 0;
}
```

ここでなぜ"cin >> s"を使わないのでしょうか。今回のAOJの問題では空白が入ることはありませんが、問題によっては入力の途中に空白が入ったりします。"cin >> s"だと空白で区切ってしまうので、そのような問題を読み込むのに適切ではありません。この ためgetlineを使って1行ずつ読み込みます。また、getlineとcinからの読み込みは、空 白部分でちょっと相性が悪いです。なので、「cinで読み込んだ後」はgetlineをする前 にignoreを呼びましょう。cinで何かを読み込んだ後の空白や改行などをスキップして あげないと、getlineで空行を読み込んでしまうという罠にはまってしまうのです。

構文解析の拡張

実はこれだけではAOJの問題は解けません。括弧による優先順位の変更が必要です。しかし、この対応はnumberとtermの間に一つ挟むことで対応できます。次のように構文を 変更しましょう:

```
<四則演算の式> ::= <乗算除算の式> (+ or -) <乗算除算の式> (+ or -) ...
<乗算除算の式> ::= <括弧か数> (* or /) <括弧か数> (* or /) ...
<括弧か数>     ::= '(' <四則演算の式> ')' or <数>
<数>           ::= ...
```

また、<括弧か数>に対応する関数は次のようにします:

```
// 括弧か数をパースして、その評価結果を返す。
int factor(State &begin) {
    if (*begin == '(') {
        begin++; // '('を飛ばす。
        int ret = expression(begin);
        begin++; // ')'を飛ばす。
    } else {
        return number(begin);
    }
}
```

factorというのもexpressionやtermと同様に、構文解析でよくある名前です。

最後にtermの中のnumberの呼び出しをfactorに置き換えてあげれば完成です。実行して SubmitしてACしましょう!

方針のまとめ

1. テンプレートを書く
2. expression, term, factor, numberのスケルトンを書く
3. factorとnumberを埋める
4. expressinとtermを埋める(最初に一つ下の部分式を一つだけパースした後に、あとはループで回しながら、足したり引いたりする。)
5. 文字列の入力を行う

構文解析のデバッグ手法

構文解析はやるだけゲーですが、安心しているとバグにはまってしまったりします。ここで紹介している書き方は、各部分式に対応する関数を呼び出すための引数も少なく、 複雑な値をリターンすることもないので、シンプルに書けると思いますが、それでもや っぱりバグは混入するものです。

構文解析でよくあるバグは次の二つです。

1. begin++をし忘れる
2. getlineとcinの混合

各部分式の関数の途中途中で、cerrにいろいろ出力する感じになると思います。が、とりあえず、構文解析でやるのは「これから読み込もうとする文字が本当に正しいか確認すること」です。このために、次のような関数を用意しましょう:

```
// beginがexpectedを指していたらbeginを一つ進める。
void consume(State &begin, char expected) {
    if (*begin == expected) {
        begin++;
    } else {
        cerr << "Expected '" << expected << "' but got '" << *begin << "'"
            << endl;
        cerr << "Rest string is '";
        while (*begin) {
            cerr << *begin++;
        }
        cerr << "'" << endl;
        throw ParseError();
    }
}
```

この関数は、次に読み込む文字が期待するものでなかった場合はエラーを出力して例外 を投げます。(例外を受け取った場所で出力した方がいいと思いますが、ほとんどどう せParseErrorはキャッチしないので。) begin++をしているところを、この関数で置き換えることで、多くのエラーを防ぐことができます。例えば:

```
// 括弧か数をパースして、その評価結果を返す。
int factor(State &begin) {
    if (*begin == '(') {
        consume(begin, '(');
        int ret = expression(begin);
        consume(begin, ')');
    } else {
        return number(begin);
    }
}
```

expressionを評価した後に、きちんと閉じ括弧がきていなければ、どこか括弧の中の途 中のところでパースできなかった部分が存在するはずです。その部分を特定してバグを つぶしていきます。

また、最初の構文解析を始めるところでもチェックを行うことができます:

```
State begin = s.begin();
int ans = expression(begin);
consume(begin, '=');
cout << ans << endl;
```

今回の問題では最後に=がくるのでこのようにチェックできます。そうでない場合でも、beginがs.end()と一致しているかを判定することで同様のチェックを行うことができます。

応用: 数字以外を返す構文解析ゲー

数字以外を返し、さらにパースする次の式が決定的ではない例をやってみましょう。AOJ1282の問題がよい例です。

ayaya09 commented on Feb 13, 2020

factorの')'を飛ばした後return ret;が抜けていると思います