3年15組９２番　全　海成(JEON HAESUNG)
コンピュータサイエンス実習A　ソフトウェア　二回目　レポート

実習の目的
今回の実習の目的はは一回目の実習で作成した字句解析に続きC0言語の構文解析をするプログラムを作成することです。この時アルゴリズムは再帰降下方を用いて作成します。そして各構文の解析ができたらその結果を逆ポーランド記法で表します。

考え方の説明
再帰降下方のしかたは配布したプリントにあったC0の構文図を参照して書かれてる順どおりトレースするようにプログラムを作成すればいいです。構文図を見たら文の始まりをを見つけたら一次式なのか確認をすることから初め優先順位どおり解析を行っていきます。この時各構文ごとに検査関数を作り検査を行います。この検査の時式の中に式があったり文の中に文が現れることができるため検査関数の中で自分自身を呼ぶことがあります。このため再帰降下方です。
エラー処理ですが検査途中で求められてる字句がない場合は当然エラーを出してそこで終了するのではなくつ次のコードをちゃんと解析すべきです。これをやるにはエラーがあったときに次の字句を呼んでから終わるかただ終わるかをちゃんと見分ける必要があります。その理由としては以前の実習で文字を一文字づつ先読みする必要があったように今回の構文分析は字句を一つ先読みする必要があることです。
実装の話
12種類の構文の中でちょっと難しかったのが二つで残り10個は簡単に実装できます。簡単な10個は一次式を除いた全部の式です。この中でただの式と単項式以外に対する構文図は全部同じ形をしています。優先順位の高い構文から判断するために違う式に対する検査関数を呼び出して確認したあと検査する構文がないときまで繰り返します。下は関係式を検査する関数のコードです。

```
static void relational_expression() {
    int flag = -1;
    additive_expression();
    while( sy == token.LE || sy == token.LT
            || sy == token.GE || sy == token.GT ) {
        if( sy == token.LE ) flag = 0;
        else if( sy == token.LT ) flag = 1;
        else if( sy == token.GE ) flag = 2;
        else if( sy == token.GT ) flag = 3;
        get_token();
        additive_expression();
        if( flag == 0 ) {
            polish("<=");
        }
        else if( flag == 1 ) {
            polish("<");
        }
        else if( flag == 2 ) {
            polish(">=");
        }
        else if( flag == 3 ) {
            polish(">");
        }
        else {
            error("something wrong");
        }
    }
}
```

関係式より優先順位が高い加算式の検査関数を呼んだあと関係式が見当たらない時まで繰り返します。この時注意するところが分析をしたあとどんな構文なのかを逆ポーランド記法で表すために字句の情報を他に保存しておくべきです。そうしないと出力することができません。
式と単項式も簡単なため省略します。

つぎは一次式です。一次式では数と識別子と関数呼び出しと小括弧を処理します。この時識別子と関数の呼び出しの処理に注意が必要です。識別子を見つけたらその後の字句として左小括弧がすぐ出たら識別子ではなく関数の呼び出しだと判断して引数でくる式を読み取ります。下は一次式を検査する primary_expression 関数のコードの一部です。

```
static void primary_expression() {
    if( sy == token.LITERAL ) {
        polish(literal_value+"");
        get_token();
    }
    else if( sy == token.IDENTIFIER ) {
        polish(id_string);
        get_token();
        // function call
        int i = 0;
        if( sy == token.LEFT_PAREN ) {
            get_token();
            if( sy == token.RIGHT_PAREN ) {
                polish("call-0");
                get_token();
                return ;
            }
            expression();
            i++;
            while( sy == token.COMMA ) {
                get_token();
                expression();
                i++;
            }
        }
        else {
            return ;
        }
        if( sy == token.RIGHT_PAREN ) {
            polish("call-"+i);
            get_token();
        }
        else {
            error("right parenthesis expected");
        }
    }
    else if( sy == token.LEFT_PAREN ) {
        get_token();
        expression();
```

関数の呼び出しで考えられるエラーは右括弧がない場合です。
文の処理で重要なのは次の字句をどうやって保証するかです。文の処理をする statement 関数を呼び出すとき sy に処理を行うべき字句があるべきです。　これは式にも適用する話です。一つの式と文の処理が終わったら次の字句をちゃんととっているべきです。プログラムを書くときこれが一番難しかったです。式が正常的に処理されたらセミコロンが sy に入っていて文で処理が終わったらそのセミコロンの後の字句が入っているべきです。文で判断するのは IF と WHILE と空白文があってそれ以外は普通の文です。以下はコードです。エラーが出た時 get_token を呼ぶかとセミコロンを処理した後は必ず次の字句をとるべきです。

```
static void statement() {
    if( sy == token.SEMICOLON ) {
        System.out.println("empty statement");
        get_token();
    }
    else if( sy == token.IF ) {
        get_token();
        if( sy == token.LEFT_PAREN ) {                    // right path
            boolean blankflag = false;
            get_token();
            polish("if statement:");
            if( sy == token.RIGHT_PAREN ) {
                error("need condition expression");
                blankflag = true;
            }
            if(blankflag) {
                get_token();
                System.out.println();
                statement();
            }
            else {
                expression();
                if( sy == token.RIGHT_PAREN) {
                    get_token();
                }
                else {
                    error("right parenthesis expected");
                }
                System.out.println();
                statement();
            }
            if( sy == token.ELSE ) {
                System.out.println("else part");
                get_token();
                statement();
            }
            System.out.println("end if statement");
        }
        else {
            error("left parenthesis expected after if");
        }
    }
```

```
        error("left parenthesis expected after if");
    }
}
else if( sy == token.WHILE ) {
    get_token();
    if( sy == token.LEFT_PAREN ) {
        boolean blankflag = false;
        get_token();
        polish("while statement:");
        if( sy == token.RIGHT_PAREN ) {
            error("need condition expression");
            blankflag = true;
        }
        if(blankflag) {
            get_token();
            System.out.println();
            statement();
        }
        else {
            expression();
            if( sy == token.RIGHT_PAREN) {
                get_token();
            }
            else {
                error("right parenthesis expected");
            }
            System.out.println();
            statement();
        }
        System.out.println("end while statement");
    }
    else {
        error("left parenthesis expected after while");
    }
}
else if( sy == token.LEFT_BRACE ) {
    get_token();
    if( sy == token.RIGHT_BRACE) {      // empty brace
        get_token();
        return ;
    }
    while(true) {
        statement();
        if( sy == token.RIGHT_BRACE ) {
            get_token();
            return ;
        }
```

```
else if( sy == token.LEFT_BRACE ) {
    get_token();
    if( sy == token.RIGHT_BRACE) {      // empty brace
        get_token();
        return ;
    }
    while(true) {
        statement();
        if( sy == token.RIGHT_BRACE ) {
            get_token();
            return ;
        }
        if( sy == token.END_PROGRAM ) {
            error("too few right braces at end of statement list");
            sy = token.ERROR;
            return ;
        }
    }
}
else {
    expression();
    System.out.println();
    if( sy != token.SEMICOLON ) {
        error("semicolon expected");
        return ;
    }
    get_token();
}
```