手を動かして学ぶ！コンピュータアーキテクチャとアセンブリ言語プログラミングの基本

計算機を作ろう（要件定義編）

皆さんは今までアセンブリのプログラミングをしてきてどうでしたか？「難しいし勘弁してほしい」とか「おもしろいけど実用的なものを作れる気がしない」とか思っているかもしれませんが、今回はおもいきって計算機を実装してみたいと思います。
どんな計算機かというと、四則演算ができる計算機です。括弧をつけて順序を制御したりできます。

あらかじめご了承いただきたいのは、今回は前回よりも難易度がとても高いです。無理に理解しなくても大丈夫ですが、よければ是非アセンブリで実用的なものを作るという体験をしてみてください。

計算機の要件定義

それでは、計算機の要件定義をしていきます。ところで、皆さんは計算機ってどういう風に動作して計算しているか知っていますか？与えられた文字列からルールに則ってプログラムに何をさせたいのか判定する処理のことを構文解析と言います。今回は計算機用の構文解析を行うということです。

計算機用の構文解析の手法にはいくつか種類がありますが、今回挑戦するのは再帰下降構文解析という手法です。専門用語を省いて簡単に説明するなら、形式的な表現方法をそのままプログラムに落とし込めるようなシンプルな構文解析です。

再帰下降構文解析という単語を覚えたうえで、計算機の要件定義を行います。

- 四則演算が可能な計算機
- 再帰下降構文解析を用いて実装をする。
- 入力は標準入力から読み込む
- 出力は標準出力へ送る
- 与えられた入力はエラーは無いものとする。
- 二項演算のみをサポートする（-1 + 5みたいな書き方はできない）

一番大切なのは再起下降構文解析についてです。今回はこれについて全体像をつかむために座学をしっかりやることにします。

再起下降構文解析ってなによ

再起下降構文解析とは、相互再帰的な手続きで構成されるLL法のトップダウン構文解析であり、各プロシージャが文法の各生成規則を実装することが多いです。

生成規則とは、例えば足し算や掛け算の順序を定義するものです。

トップダウン構文解析とは大きな要素から開始して徐々に細部に分解していく構文解析のことです。

LL法とは、終端ではない記号を取り出した時生成規則に従い出力した記号列からまた終端ではない記号を取り出すことを繰り返し構文木の根から葉まで探索を繰り返す構文解析のことです。

相互再帰とは、関数などの一連の処理のかたまりが、互いに呼び出しあうことです。

再起下降構文解析の特徴として、バッカス・ナウア記法で示される形式言語のまま実装が進められることにあり、実装が分かりやすい点にあります。

難しい事をまくしたてましたが、要は構文解析を手で実装するなら比較的簡単でおすすめってことです。

今回実装する計算機の生成規則を拡張バッカス・ナウア（EBN）記法で示すと以下のようになります。

```
EXPR = TERM, {("+"), TERM | ("-", TERM)}
TERM = FACT, {("*", FACT) | ("/", FACT)}
FACT = [SPACE], ("(", EXPR, ")") | NUM, [SPACE]
SPACE = ? white space charactores ? (*スペース文字を除去する処理*)
NUM = ? 0 ~ 2,147,483,647 ? (*符号付き32bit整数の上限*)
```

表記	意味
=	定義する
,	連結
|	区切る
[...]	オプション
{...}	繰り返す
(...)	グループ化
"..."	文字
?...?	例外
(*...*)	コメント

以上の生成規則を基に下の文字列を解釈して構文木を作ってみたいと思います。

```
12 * (3 + 4)
```

文字の配列でいうとこのようになります。

0	1	2	3	4	5	6	7	8	9	10	11
1	2		*		(	3		+		4	)

今何文字目に着目しているか示す変数 `P` についてまず定義しておきます。初期状態は `0` 、つまり0文字目の `1` を指し示しています。 また、計算を便利にするためのスタックを定義します。スタックはご存じの通り `PUSH` で記号を積み `POP` で積まれた一番上の記号を取り出します。このスタックは後々使います。。

まず、 `EXPR` からはじまります、 `EXPR` は何が書いてあるのかというと、まず `TERM` を取り出して、そのあと `+` を取り出して `TERM` を取り出す、もしくは `-` を取り出して `TERM` を取り出すということが書かれています。

```
EXPR
```

では `TERM` を取り出しますね。 `TERM` ではまず `FACT` を取り出して、そのあと `*` を取り出して `FACT` を取り出す、もしくは `/` を取り出して `FACT` を取り出します。

```
EXPR -> TERM
```

では `FACT` を取り出しましょう。 `[SPACE]` はスペースが入っていたら無視するということが書かれています。次の `("(", EXPR, ")")` は `(` があったら、次に `EXPR` を取り出してから `)` を取り出すということが書かれています、もしくは `NUM` を取得してからスペースの除去と言うように処理が流れていきます。

```
EXPR -> TERM -> FACT
```

現在 `P` は `0` つまり文字列の `1` を指しています。 `TERM` から下っていき、 `FACT` の `NUM` が適合します。そのため `NUM` で処理され、 `12` という数値が出されます。 `P` は `2` まで進み、スタックに `12` を `PUSH` します。

```
EXPR -> TERM -> FACT -> 12
```

`P` は2文字目のスペースを指し示しています。そのため `FACT` の `NUM` のあとの `[SPACE]` が適応されます。スペース除去です。 `P` は `3` になります。

呼び出された関数の途中で `return` されるのをイメージしてください。 `[SPACE]` まで適合したところで、 `TERM` に戻されます。 `FACT` の部分には `12` が入っています。次に `*` か `/` が適合するとそのあとの処理に進みますが、 `P` は `3` 、つまり `*` を指していますので、あらためて `FACT` を取り出します。

```
EXPR -> TERM -> FACT -> 12
             -> *
             -> FACT
```

スペース除去をして、`P`を`5`に更新します、`(`が見つかったので`EXPR`を呼び出します。`EXPR`の中でまた`TERM`が呼ばれ、`TERM`の中で`FACT`が呼ばれます。`P`を`6`に更新して、`3`が見つかりました。スタックに`PUSH`しておきます。

```
EXPR -> TERM -> FACT -> 12
             -> *
             -> FACT -> (
                     -> EXPR -> TERM -> FACT -> 3
```

スペースの除去を行い、`return`します。`TERM`に帰ってきました。`P`は`8`、つまり`+`を指しています。`TERM`に適合するものはありませんのでまだ`return`します。`EXPR`に戻ってきました。適合する`+`がありますのでその処理の流れに進みます。

```
EXPR -> TERM -> FACT -> 12
             -> *
             -> FACT -> (
                     -> EXPR -> TERM -> FACT -> 3
                             -> +
                             -> TERM
```

スペースの除去を行います。`P`の値は`10`、つまり`4`を指し示します。`TERM`内では`FACT`が呼び出され、`NUM`が適合するのでそれを取り出します。スタックに`4`を`PUSH`しておきます。`EXPR`内の`+`の後の`TERM`も取り出し終わったので、ここでスタックに`+`を`PUSH`しておきます。

```
EXPR -> TERM -> FACT -> 12
             -> *
             -> FACT -> (
                     -> EXPR -> TERM -> FACT -> 3
                             -> +
                             -> TERM -> FACT -> 4
```

`NUM`までいってスペースもないので`return`します。どこまで帰るのかというと、`"(", EXPR, ")"`までです。

```
EXPR -> TERM -> FACT -> 12
             -> *
             -> FACT -> (
                     -> EXPR -> TERM -> FACT -> 3
                                     -> +
                                     -> TERM -> FACT -> 4
                     -> )
```

`P`は`11`、つまり最後まで来ました。ここで最初の`EXPR`になるまで`return`します。道中のプロシージャ、`*`を取りだした`TERM`の後半の`FACT`を取り出したので、スタックに`*`を`PUSH`しておきます。これで構文解析は終了です。

構文木を取り出してみましょう。

```
*   ->  12
    ->  +   ->  3
            ->  4
```

このようになります。これが再起下降構文解析を用いた構文解析です。何となく流れは掴めましたでしょうか？

計算するには？

さて、計算するにはという話ですが、足し算と掛け算ができる、以下のような関数群を考えてみます。

```
//  ポインタをもとに指している文字を取り出す
int peek(){
    return string[P];
}

// ポインタを次に進める
```

```
void next(){P++;}


// EXPR = TERM, "+", TERM
int expr(){
    int x = term();
    for(;;){
        if(peek() == '+'){
            next();
            x += term();
            continue;
        }
        break;
    }
    return x;
}

// TERM = FACT, "*", FACT
int term(){
    int y = fact();
    for(;;){
        if(peek() == '*'){
            next();
            y *= fact();
            continue;
        }
        break;
    }
    return y;
}

// FACT = ("(", EXPR, ")") | NUM
int fact(){
    if(peek() == '('){
        next();
        int z = expr();
        if(peek() == ')'){
            next();
        }
        return z;
    }
    return number();
}
```

`string`は数式の入った文字列、`P`は指し示す為のポインタ、`number`関数は次の数値を取り出すための処理です。

これらの関数があるとして、上の`12*(3+4)`の処理が進んでいくことを考えます。説明を簡略化するためスペースは無いこととします。

エントリーポイントは、`expr`関数からです。まず`x`に`term`からの返却値をいれます。`term`関数では`y`に`fact`から返却値をいれます。`fact`関数では`number`関数を呼び出し、次の数値を取り出し返却します。

`number`で取り出されたのは、`12`です。`fact`から`term`に`return`され、変数`y`として保持されます。`term`関数内で`*`という条件に適合しますので、また`fact`関数が呼び出され、次は`(`が適合します。

括弧が検出されたので`expr`関数が呼び出されます。`expr`の中で`term`が呼ばれ、`term`の中で`fact`が呼ばれ、`3`が取り出されます。`return`して`expr`の中で`x`に値が保持されます。次に出てくるのは`+`で`expr`内の条件に適合しますので、また`term`、`fact`と呼び出され、`4`が返却されます。

`expr`の中で`3`と`4`がそろいますので足して`7`となります。閉じ括弧が見つかりますので、`7`が返却され、`fact`の`z`に保持され、返却されます。この`7`は`term`の後半の数値になるので、`12 * 7`が成立、`84`という答えが導かれるわけです。

文字で説明してみましたが何となくわかりましたか？複雑なので、関数を見ながらどのように処理が流れているのか確認してみてください。上の関数は足し算と掛け算しか出来ませんが、実装するプログラムでは上の`EBN表記`を基に四則演算が出来るプログラムを実装していきます。

まとめ

- 再起下降構文解析という方法がある
- 解析のルールを定めるEBN記法というものがある
- 再起下降構文解析はEBNをそのまま書くように実装することが出来る

- 構文解析を施すことで数式から構文木を作ることが出来る
- 構文解析中の各プロシージャで変数を保持することで計算することが出来る
- スタックはあとで使います