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

計算機コンパイラを作ろう

今回は計算機コンパイラを作ります。前回は四則演算をしてくれる計算機プログラムを作成したわけですが、今回はそれを発展させて実行可能なアセンブリを出力してみましょう。例によって難易度は高いですが、アセンブリでこんなことできるんだということや普段コンパイラってこんなことしてるんだということをなんとなくわかったような気になっちゃってください。

あ、前々回からひっぱってきた計算したときのスタック、やっと使いますよ、お待たせしました。

コンパイラってなんだ？

まず始めに、コンパイラって何だという話をしていきたいと思います。

```
『コンパイラ（英: compiler）は、コンピュータ・プログラミング言語の処理系（言語処理系）の一種で、
高水準言語によるソースコードから、機械語[1]あるいは元のプログラムよりも低い水準のコードに
変換（コンパイル）するプログラムである。』（Wikipedia引用）
```

らしいです。今回やりたいこととして言い換えると、数式というソースコードからアセンブリに変換するプログラムということになりますね。皆さんプログラミング言語というと、C言語やJavaなどの汎用プログラミング言語を想像すると思いますが、ドメイン固有言語という特定の処理向けに設計されたプログラミング言語もあります。そう考えると『四則演算の計算をすることに特化し数式を解釈し指示通りに計算するアセンブリを出力するプログラム』というのは十分コンパイラと言えるのではないかと思います。

じゃあとりあえず本当に簡単なコンパイラを作ってみましょうか。突然言われても困ると思いますが、大丈夫です。本当に大したことないんで。以下要件定義をします。

- 1桁の足し算をするアセンブリを出力するプログラムです
- 入力は2文字、どちらも数値です
  - 例: `45`
- 入力された文字列を二つに分け、足し算を行うプログラムです。
  - 例: `45` と入力されたら `9` と出力するアセンブリを出力する。

とりあえずこのソースコードを見てください。

```
CALC    START
        LAD     GR0, 1
        LAD     GR1, 4
        LAD     GR2, 5
        ADDA    GR1, GR2
        WRITE   GR0, GR1
        RET
CALCEND END
```

はい、何の変哲もない足し算ですね、 `LAD` 命令でそれぞれのレジスタに数値を書き込み `ADDA` で足し算をしてポート1、つまり符号付き10進数標準出力へ流し込む、そんなプログラムです。これの `4` とか `5` の部分さえ変えることが出来るなら要件は十分に満たせますよね？

```
PGM     START
        IN      BUF, =2
        LAD     GR1, 0          ; value
        LAD     GR2, 0          ; array_sub
        LAD     GR3, 0          ; port

        LAD     GR10, '\n'
        LAD     GR11, 0

        OUT     ='CALC    START\n'          , =14
```

```
        OUT     ='      LAD     GR0, 1\n' , =23
        OUT     ='      LAD     GR1, ' , =21

        LD      GR1, BUF        ; GR1 <- BUF[0]

        WRITE   GR3, GR1        ;
        WRITE   GR11, GR10      ; 改行

        OUT     ='      LAD     GR2, ' , =21

        ADDA    GR2, =1
        LD      GR1, BUF, GR2   ; GR1 <- BUF[1]
        WRITE   GR3, GR1        ;
        WRITE   GR11, GR10  ; 改行

        OUT     ='      ADDA    GR1, GR2\n', =25
        OUT     ='      WRITE   GR0, GR1\n', =25
        OUT     ='      RET\n', =12
        OUT     ='CALCEND END\n', =12

        RET
BUF     DS      2
        END
```

入力できましたか？これを`addcmp.fe`と名付けて保存しました。以下のように実行してみてください。

```
> echo "45" | python mlfe.py addcmp.fe
CALC    START
        LAD     GR0, 1
        LAD     GR1, 4
        LAD     GR2, 5
        ADDA    GR1, GR2
        WRITE   GR0, GR1
        RET
CALCEND END
```

お、何か出来そうですね。MLFEには出力を保存する機能はないのでシェルの上書き機能を使って保存し、実行してみます。

```
> echo "45" | python mlfe.py addcmp.fe > out.fe
> python mlfe.py out.fe
9
```

できましたね、これだけです。定義からすればこれも立派なコンパイラといえます。ね、たいしたことなかったでしょ？

今回作るのはもうちょっとすごいやつですが、コンパイラ作成なんて難しそう、私に出来るかなと怖がる必要は無いということだけ伝えたいです。それではコンパイラの為の新たな概念を勉強しましょう。

ポーランド記法

ポーランド記法について解説をします。その前に前々回の伏線を覚えていますか。そうあの`再起下降構文解析ってなによ`で出てきたスタックです。ちょっと構文木を作るあたりを読み返してスタックに対してどんな操作をしたのか読み返して確認してみてください。

できました？スタック操作関係の文言だけ抽出すると以下のようになります・

1. スタックに`12`を`PUSH`
2. スタックに`3`を`PUSH`
3. スタックに`4`を`PUSH`
4. スタックに`'+'`を`PUSH`
5. スタックに`'*'`を`PUSH`

こんな感じのことしてましたね。実はこの操作が分かれば、機械的に計算することはとても簡単です。

- 数値が`PUSH`された時は、素直にスタックに`PUSH`する
- 演算子が`PUSH`される時には、スタックから数値を二つ取り出し演算子に対応した演算をして、結果をスタックに`PUSH`
- スタックが数値一つだけ、かつ手順がもう無いときは計算終了。

上の法則に従って処理をしてみましょう。

1. スタックに `12` 、 `3` 、 `4` を `PUSH` する。
2. 演算子 `+` が来たので、2回 `POP` 、 `4` と `3` が取り出され足し算をすると結果は `7` 、それをスタックに `PUSH`
3. 演算子 `*` が来たので、2回 `POP` 、 `7` と `12` が取り出され掛け算をすると結果は `84` 、それをスタックに `PUSH` する
4. スタックに数値が 1 つだけかつもう手順が無いので計算終了、結果は `84` となった。

計算できましたね、どれも数値や演算子が出現するたびに操作をすれば良いので、再起下降構文解析の途中で対応するアセンブリを生成することができれば、それだけでコンパイラになります。

ちなみに、 `12 * (3 + 4)` からアセンブリを生成するとこのようになります。

```
CALC    START
        LAD     GR2, 1
        PUSH    12
        PUSH    4
        PUSH    3
        POP     GR1
        POP     GR0
        ADDA    GR0, GR1
        PUSH    0, GR0
        POP     GR1
        POP     GR0
        MULA    GR0, GR1
        PUSH    0, GR0
        POP     GR0
        WRITE   GR2, GR0
        RET
CALCEND END
```

なんかいけそうな気がしませんか？

ちなみに、スタックに積まれる内容を書き下すとこのようになります。

```
 * + 4 3 12
```

このように、演算子を前において、優先順位や括弧を気にしなくても機械的に計算できるような形式のことを前置記法、またはポーランド記法と言います。普段使っているのが中置記法と呼ばれ、また演算子を後ろに置く後置記法（逆ポーランド記法）というものもあります。
構文解析の時には結構利用する概念なので言葉だけでも覚えてください。

私たちの作るコンパイラは、再起下降構文解析、およびポーランド記法を用いて行うプログラムだということを頭の隅にでもおいておいてくださいね。では実装していきましょう。

計算機コンパイラの実装

実装を行うのですが、さっきのアセンブリの生成結果からどのようなプログラムにするべきか考えてみます。

```
CALC    START                   ; プログラムの最初に
        LAD     GR2, 1          ; STARTとLADをする

        PUSH    12              ; 数値が出たらスタックにプッシュ

        PUSH    4

        PUSH    3

        POP     GR1             ; 演算子に対しては2回ポップして
        POP     GR0
        ADDA    GR0, GR1        ; 対応する演算をする
        PUSH    0, GR0

        POP     GR1
        POP     GR0
        MULA    GR0, GR1
        PUSH    0, GR0
```

```
        POP     GR0             ; プログラムの最後に
        WRITE   GR2, GR0        ; スタックの最後の値をポップしてWRITE
        RET                     ; RETしてENDをする
CALCEND END
```

おおよそこんな感じ、FizzBuzzの実装で完成したプログラムに付け足すような形で出来そうですね。

じゃあ文字列領域を確保して、出力機能を実装すれば良いのですが、ここでちょっと標準出力機能について新たなものを紹介させてください。

それは`printf`です。

```
PGM     START
        LAD     GR1, MSG
        LD      GR2, STRING
        SVC     printf
        RET
MSG     DC      'Helloworld\n\0'
STRING  DC      's'
        END
```

`scanf`と少し似ているのですが、`GR1`に読み出したいメモリの先頭番地、`GR2`にフォーマット文字を指定することで標準出力することができます。他にも数字やアドレスを出力する機能もあるので詳しくはマニュアルに任せるのですが、今回は文字列出力に絞って解説したいと思います。

ところで、今までに習った`OUT`マクロとは大きく異なる部分があることがわかりますか？それは出力命令の際に何文字分かを指定する必要が無いことです。`printf`は区切り文字があるところまで読んでいく決まりになっており、その区切り文字はC言語ならヌル文字と呼ばれるメモリの数値の`0`となっているところまで読んでいきます。

MLFEにおいてもヌル文字は有効で、文字列中にヌル文字を入れたい場合には`\0`と入力します。逆に区切りとなるヌル文字が見つからない場合、出力はしません。

今回の計算機コンパイラはこの`printf`を使ってみたいと思います。

じゃあ文字列領域の定義を行いたいと思います。

```
MSGST   DC      'CALC    START\n\0'
MSGLAD  DC      '        LAD     GR2, 1\n\0'
MSGP0   DC      '        POP     GR0\n\0'
MSGP1   DC      '        POP     GR1\n\0'
MSGPS   DC      '        PUSH    0, GR0\n\0'
MSGPSX  DC      '        PUSH    \0'
MSGW    DC      '        WRITE   GR2, GR0\n\0'
MSGADD  DC      '        ADDA    GR0, GR1\n\0'
MSGSUB  DC      '        SUBA    GR0, GR1\n\0'
MSGMUL  DC      '        MULA    GR0, GR1\n\0'
MSGDIV  DC      '        DIVA    GR0, GR1\n\0'
MSGRET  DC      '        RET\n\0'
MSGEND  DC      'CALCEND END\n\0'
```

もうちょっとまとめることもできますが、可読性重視ということでこのような感じでOKです。

次にこれらを出力する関数を定義します。

```
; Out_START
OSTART  RPUSH   0, 2
        LAD     GR1, MSGST
        LD      GR2, STR
        SVC     printf
        RPOP    0, 2
        RET
```

例えば、プログラム最初の`START`を出力するための関数はこのようになります。もうひとつ最初の`LAD`を出力するものを付け足します。

```
; Out_START
OSTART  RPUSH   0, 2
        LAD     GR1, MSGST
        LD      GR2, STR
        SVC     printf
        RPOP    0, 2
        RET
; Out_LAD
OLAD    RPUSH   0, 2
        LAD     GR1, MSGLAD
        LD      GR2, STR
        SVC     printf
        RPOP    0, 2
        RET
```

ちょっと重複部分が多いですね、後半の`LD GR2, STR`以降は全く同じ内容です。これはひとまとめにした方が良さそうです。

```
; Out_START
OSTART  RPUSH   0, 2
        LAD     GR1, MSGST
        JUMP    OFIN
; Out_LAD
OLAD    RPUSH   0, 2
        LAD     GR1, MSGLAD
        JUMP    OFIN

; Out_FIN
OFIN    LD      GR2, STR
        SVC     printf
        RPOP    0, 2
        RET
```

OKです、すっきりしました。後は必要な出力関数をどんどん書いていきましょう。あ、一応間違えてもいいように前回のプログラム`rdp.fe`にそのまま書いていくのではなくコピーして`rdp2.fe`と名前を付けたものに記述するようにしてください。

```
; 関数名コメント省略
OSTART  RPUSH   0, 2
        LAD     GR1, MSGST
        JUMP    OFIN
OLAD    RPUSH   0, 2
        LAD     GR1, MSGMD
        JUMP    OFIN
OPOP0   RPUSH   0, 2
        LAD     GR1, MSGP0
        JUMP    OFIN
OPOP1   RPUSH   0, 2
        LAD     GR1, MSGP1
        JUMP    OFIN
OPUSH0  RPUSH   0, 2
        LAD     GR1, MSGPS
        JUMP    OFIN
OPUSHX  RPUSH   0, 2
        LAD     GR1, MSGPSX
        JUMP    OFIN
OWRITE  RPUSH   0, 2
        LAD     GR1, MSGW
        JUMP    OFIN
OADD    RPUSH   0, 2
        LAD     GR1, MSGADD
        JUMP    OFIN
OSUB    RPUSH   0, 2
        LAD     GR1, MSGSUB
        JUMP    OFIN
OMUL    RPUSH   0, 2
        LAD     GR1, MSGMUL
        JUMP    OFIN
ODIV    RPUSH   0, 2
        LAD     GR1, MSGDIV
        JUMP    OFIN
ORET    RPUSH   0, 2
        LAD     GR1, MSGRET
```

```
        JUMP    OFIN
OEND    RPUSH   0, 1
        LAD     GR1, MSGEND
        JUMP    OFIN

OFIN    LD      GR2, STR
        SVC     printf
        RPOP    0, 2
        RET
```

大部分が完成したといっても過言じゃないです。それでは各所に関数呼び出しを記述しましょう。

- 最初の部分

ここは呼び出すだけなので、特に工夫は必要ないですね。

```
; INIT          省略
INITLP  ADDA    GR2, ONE
        LD      GR1, TEXT, GR2
        CPA     GR1, ZERO
        JNZ     INITLP
        ST      GR2, LEN
        RPOP    1, 2

        CALL    OSTART
        CALL    OLAD

        RET
```

- 足し算と引き算の部分

ここは条件分岐した後の記述です。やることは決まっているので書くだけですね。

```
; EXPR          省略
; EXPRLP        省略

EXPRADD CALL    NEXT
        CALL    TERM
        ADDA    GR1, GR0

        CALL    OPOP1
        CALL    OPOP0
        CALL    OADD
        CALL    OPUSH0

        JUMP    EXPRLP
EXPRSUB CALL    NEXT
        CALL    TERM
        SUBA    GR1, GR0

        CALL    OPOP1
        CALL    OPOP0
        CALL    OSUB
        CALL    OPUSH0

        JUMP    EXPRLP

; EXPRBK        省略
```

- 掛け算と割り算の部分

ここは `EXPR` の部分とほぼ同様です。

```
; TERM          省略
; TERMLP        省略
TERMMUL CALL    NEXT
        CALL    FACT
        MULA    GR1, GR0

        CALL    OPOP1
```

```
        CALL    OPOP0
        CALL    OMUL
        CALL    OPUSH0

        JUMP    TERMLP
TERMDIV CALL    NEXT
        CALL    FACT
        DIVA    GR1, GR0

        CALL    OPOP1
        CALL    OPOP0
        CALL    ODIV
        CALL    OPUSH0

        JUMP    TERMLP
; TERMBK        省略
```

- 数値を取り出す部分

ここは取り出した数値を出力しなければいけないのでちょっとだけ工夫が必要になります。

```
; int NUMber()
NUM     RPUSH   1, 9            ; GR9まで使うのでそこまでレジスタ退避
        LAD     GR1, 0
        LAD     GR2, 0
        LAD     GR3, 0
        LAD     GR4, 0
        LAD     GR5, 0
        LAD     GR6, 0

        LAD     GR7, 1          ; 符号付き10進数標準出力
        LAD     GR8, 0          ; 標準出力
        LAD     GR9, '\n'       ; 改行文字

; NUMULP        省略

NUMBLP  CALL    GETDG
        LD      GR4, GR0
        POP     GR6
        SUBA    GR6, =48
        LD      GR1, GR6
        MULA    GR4, GR1
        ADDA    GR5, GR4
        ADDA    GR3, ONE
        CPA     GR3, GR2
        JMI     NUMBLP
        LD      GR0, GR5

        CALL    OPUSHX          ; PUSHを記述してから
        WRITE   GR7, GR0        ; 数値を出力して
        WRITE   GR8, GR9        ; 改行する

        RPOP    1, 9            ; レジスタを元に戻す
        RET
```

- 最後の部分

最後も出力することは決まっているので書くだけですが、元の計算機の解答出力機能をオフにしておきましょう。

```
FIN     LAD     GR1, 1
        ;WRITE   GR1, GR0
        CALL    OPOP0
        CALL    OWRITE
        CALL    ORET
        CALL    OEND
        RET
```

以上の変更点書けましたでしょうか？ではさっそく実行してみましょう！

実行しよう！

第4章Aでやった問題をやってましょう。

```
> echo "12 * (3 + 4)" | python mlfe.py rdp2.fe
CALC    START
        LAD     GR2, 1
        PUSH    12
        PUSH    3
        PUSH    4
        POP     GR1
        POP     GR0
        ADDA    GR0, GR1
        PUSH    0, GR0
        POP     GR1
        POP     GR0
        MULA    GR0, GR1
        PUSH    0, GR0
        POP     GR0
        WRITE   GR2, GR0
        RET
CALCEND END
```

お、出来てそうですね。ファイルに保存して実行してみましょう。

```
> echo "12 * (3 + 4)" | python mlfe.py rdp2.fe > out.fe
> python mlfe.py out.fe
84
```

ついにできました！計算機コンパイラです！結構実用的なものになったのではないでしょうか。もっと工夫すれば比較演算やビット演算など新しい機能も追加できますので是非挑戦してみてくださいね！

ついにここまできましたね、第4章完です。アルゴリズムやテクニックは理解できましたでしょうか。できてなくても全然かまわないのです。この章はアセンブリを使って何かちゃんとうごくものを作ってほしいという考えのもと作成していますので、理解しにくいアルゴリズムや小難しいテクニックは分からなくても良いのです。それよりもここまで走りきれた自分をほめてあげてください。きっと誰だってできることではないと思います。

それでは次回、アセンブリでマインスイーパを実装してみたでお会いしましょう。お疲れさまでした。

まとめ

- コンパイラとは高水準のソースコードからより低い水準のコードへ変換するプログラムのことである
- ポーランド記法とは、数式などを形式に解釈するための表現で、コンピュータが理解しやすい
- `printf` は `GR1` にメモリの先頭番地、 `GR2` にフォーマットを指定して使用する
- アセンブリで計算機とかコンパイラの実装は誰でもできることじゃないと思うので自信を持ってください！