Tiny Ruby コンパイラの作成

Ruby のコンパイラをデカルト言語を使って作りましょう。
しかし、フルセットの偉大なる ruby の仕様では、ちょっとここで作って説明するというわけにはいきませんし、私には完璧な Ruby を作る知識も能力もございません。そこで、思い切り簡略化した仕様とします。
昔々、Tiny BASIC という、とてもかわいい BASIC 言語があったと聞きます。なんと、8bit プロセッサでメモリが 4KB しかなくても動いたそうです。
Tiny BASIC のような簡単で簡略化された Tiny Ruby のコンパイラを作ることとしましょう。
簡略化の過程で本来の Ruby のもつ特徴や長所が失われてしまいますが、ちょっと見た目が似ているソースがコンパイルできるくらいのものにはしてみましょう。

1. Tiny Ruby の仕様

次に示すような仕様とします。

1. データ型は整数(int)のみ
2. 変数はローカル変数のみで、変数名は英字 1 文字で a から z までのみ
3. print メソッドが使える
4. print メソッドの中で文字列が使える
5. 制御構文は if, while が使える。elsif は使えない。
6. times メソッドが使える。
7. メソッドとクラスの新規定義はできない。
8. 条件として比較演算は使えるが and, or, not のような論理演算は使えない。
9. コメントは、#から行末まで。
10. ソースコードを読み込み、C のソースを出力する。

この仕様だと、以下のようなプログラムが書けるのですが、はたして、Ruby のソースに見えるでしょうか？

```
# Tiny Ruby の例題
a=1; b=2
c= a + b
print "display ", c, "¥n";
10.times do
  print a
  a = a + 1
end
print "¥n"
```

このようなコンパイラが、デカルト言語では割と簡単につくることができるのです。

2. Tiny Rubyの字句解析

デカルト言語では、字句解析(入力ソースをトークンに分解する処理)は、非常に簡単に書けます。トークンとは、"if", "else","end", 演算子, コンマ, セミコロン, 数字列, 変数などの構文を解析するときの、文字の塊の最小単位です。
"if", "else","end"といったような決まった固定された文字列の場合は、そのまま文字列として書けば良いです。

```
"if"
"else"
"end"
```

演算子、コンマ、セミコロンも同様です。

```
"*"
"_"
","
";"
```

実は、上のような英字だけのトークンや、記号１字のトークンの場合は、デカルト言語では""でくくる必要もありません。"if"でもifでも大丈夫です。両者は同じものとされます。
英字と記号が組み合わされたトークンでは、たとえばaの１番を表す"a-1"というトークンがあった場合、デカルト言語はa-1と書いてあるとトークンを aと-1に分解してしまうので"a-1"と""で括らなければならないのです。しかし、今回のTiny Rubyでは、トークンがトークンであることを目立たせるために、基本的に英記号トークンは""でくくることとします。
数字列については、組み込み述語である<NUM>が使えます。

```
<NUM>
```

""で括られた文字列は、組み込み述語<STRINGS>が使えます。
変数は、小文字英字一文字と決めたのですが、このような可変な構造を持ったトークンを任意に設定するには、<TOKEN>述語を使います。<TOKEN>述語の引数に字句解析の構文を付けると任意のトークンを合成できます。ここでは、一定の範囲の文字を字句解析できる<RRANGE>組み込み述語を使うことにしましょう。

```
<TOKEN _ <RANGE _ a z>>
```

これで、a から z までの一文字を抜き出します。_は無名変数であり、字句解析した結果が設定されます。しかし、今回の場合は、TOKEN の無名変数には、RANGE の無名変数と同じものが設定されることになります。よって、TOKEN 述語は無くても問題ないため、変数は RANGE 述語だけで構成することになります。

〈RANGE _ a z〉

詳しくは、次の「3. Tiny Ruby の構文解析」を見てから、もう一度本項を見ると分かりやすいと思います。

3. Tiny Rubyの構文解析

さっそくですが、Tiny Rubyの構文をデカルト言語で書いてみます。

〈TinyRuby〉　〈プログラム〉〈EOF〉;

〈プログラム〉　{[〈実行文〉{ ";"〈実行文〉}] (〈コメント〉 | 〈CR〉) };

〈実行文〉　〈if 文〉 | 〈while 文〉 | 〈print 文〉 | 〈times 文〉 | 〈代入文〉 | 〈数式〉;

〈if 文〉　"if"〈条件式〉〈プログラム〉["else"〈プログラム〉] "end";

〈while 文〉　"while"〈条件式〉〈プログラム〉"end";

〈times 文〉　(〈NUM〉 | 〈変数〉) "." "times" "do"〈プログラム〉"end";

〈print 文〉　"print" 〈表示項目〉{ "," 〈表示項目〉} ;

〈表示項目〉　(〈STRINGS〉 | 〈NUM〉 | 〈数式〉);

〈代入文〉　〈変数〉"="〈数式〉;

〈数式〉　〈expradd〉;

〈expradd〉　〈exprmul〉{ "+"〈exprmul〉 | "-"〈exprmul〉 };

〈exprmul〉　〈exprID〉{ "*"〈exprID〉 | "/"〈exprID〉 };

〈exprID〉　"+"〈exprterm〉 | "-"〈exprterm〉 | 〈exprterm〉;

〈exprterm〉　"("〈数式〉")" | 〈NUM〉 | 〈変数〉;

〈条件式〉　〈数式〉(">=" | ">" | "==" | "!=" | "<=" | "<")〈数式〉;

〈変数〉　〈RANGE _ a z〉;

〈コメント〉　　″#″ 〈SKIPCR〉;

文法規則を書いただけのようにも見えますが、デカルト言語では、この規則をプログラムとして直接受け付けることができます。
これで、「1. Tiny Ruby の仕様」で日本語で書かれていたときには曖昧であった部分がなくなり、Tiny Ruby の言語仕様が厳密に定義されました。どのような構文をどのような順序で、何を書けばよいのかがすべて定義されているのです。
ただし、これは、構文を定義したものであり、まだ、コンパイル結果を出力する機能は付けてない状態です。これだけでは、Tiny Ruby のソースを入力しても、コンパイルはできません。コンパイル結果を出力する部分や、入力ファイルを準備してこの構文にインプットする部分のプログラムが必要となります。
詳細な説明は、次項で行っていきます

4. 構文の詳細

順番に構文を見ていきましょう。

〈TinyRuby〉　　〈プログラム〉〈EOF〉;

これは、TinyRuby は、プログラムと最後の EOF でできていることを表しています。
EOF とは、End Of File のことでソースファイルの末尾に来たことを意味します。

〈プログラム〉　　{[〈実行文〉{ ″;″〈実行文〉}] (〈コメント〉｜〈CR〉) };

{} の括弧は 0 回以上の繰り返しを意味します。
プログラムは、実行文をセミコロン(;)か改行で区切ったものの繰り返しです。
プログラムの後ろにはコメントが付けられます。実行文がないコメントだけの行もありえます。

〈実行文〉　　〈if 文〉｜〈while 文〉｜〈print 文〉｜〈times 文〉｜〈代入文〉;

実行文は、if 文、while 文、print 文、times 文、代入文のいずれかです。

〈if 文〉　　″if″〈条件式〉〈プログラム〉[″else″〈プログラム〉] ″end″;

[]の括弧は省略可能であることを意味します。
if 文は、見てのとおりです。else は省略できます。

〈while 文〉　　"while"〈条件式〉〈プログラム〉"end";

〈times 文〉　　(〈NUM〉|〈変数〉)"."　"times"　"do"〈プログラム〉"end";

〈print 文〉　　"print"　〈表示項目〉{ ","　〈表示項目〉} ;

while, times, print も、構文の並びをそのまま表示しているだけです。 times 文で、先頭が数字 NUM か変数しか使えないのは大きな制限です。

〈数式〉　　〈expradd〉;

〈expradd〉　　〈exprmul〉{ "+"〈exprmul〉| "-"〈exprmul〉};

〈exprmul〉　　〈exprID〉{ "*"〈exprID〉| "/"〈exprID〉};

〈exprID〉　　"+"〈exprterm〉| "-"〈exprterm〉| 〈exprterm〉;

〈exprterm〉　　"("〈数式〉")"| 〈NUM〉| 〈変数〉;

数式は、優先度の低いほうから高いほうに向かって処理を積み上げています。
最も優先度の高いのは expterm であり、() で括られた数式や数値と変数を処理します。
次に単項の+-を処理する exprID です。+1 とか-a とかの符号付きの項目を処理します。
次が exprmul の乗除算であり、最後が expradd の加減算となります。数式の処理では、掛け算と割り算を足し算と引き算より優先して計算しますね。

〈変数〉　　〈RANGE _ a z〉;

変数は、a から z までの小文字とするために、デカルト言語特有の RANGE 述語を使って字句解析します。RANGE は２つの引数の範囲にある文字と合致します。

〈コメント〉　　"#"　〈SKIPCR〉;

コメントは#から、改行までです。SKIPCR は、改行まで構文解析を行わずにスキップさせます。
その他構文は、ほぼそのまま構文を並べているだけです。
では、この構文の定義に実際にコンパイルするための処理を追加しましょう。

5. コンパイラ結果出力機能

前項で作成した構文プログラムに、ソースをコンパイルした結果を出力する機能を付けていきます。
print 述語と print 述語で、該当する文法に対するコンパイル結果を出力します。
print 述語は、単純な文字列の出力に使われ、自動的に出力後に改行されます。
printf 述語は、引数に%フォーマットと¥エスケープシーケンスを使って、可変な引数を柔軟に出力するのに使います。たとえば、C 言語で printf("%d¥n", num);は、デカルト言語では、<printf <%_"%d" num> <¥_n>>となります。
３つめのインデントで print 系の処理を行っているところが C 言語の結果を出力しているところです。元の Ruby 言語のソースがどのような C 言語のソースに変換されているかを想像しながら見てください。
「３．Tiny Ruby の構文解析」の構文と見比べると、分かりやすいかも知れません。
最初に、<TinyRuby>を実行したときの処理について示します。

```
<TinyRuby>                          <print "#include <stdio.h>">
                                     <print "int main() {">
                                     <print "int a,b,c,d,e,f,g,h,i,j,k,l,m;">
                                     <print "int n,o,p,q,r,s,t,u,v,w,x,y,z;">
              <プログラム> <EOF>     <print "exit(0);">
                                     <print "}">
              ;
```

この部分は、Tiny Ruby の初期化と終了処理を行っています。
実行されると次に示すような C 言語に変換されます。

```
#include <stdio.h>
int main() {
        int a,b,c,d,e,f,g,h,i,j,k,l,m;
        int n,o,p,q,r,s,t,u,v,w,x,y,z;
        途中の処理(<プログラム> <EOF>)
        exit(0);
}
```

変数はすべて事前に定義しています。途中の処理には、Tiny Ruby のプログラムをコンパイルした結果が入ります。

〈プログラム〉　{[〈実行文〉{ ″;″〈実行文〉}] (〈コメント〉｜〈CR〉)};
〈実行文〉　〈if 文〉｜〈while 文〉｜〈print 文〉｜〈times 文〉｜〈代入文〉;

上記の２つの処理には、とくにコンパイル結果を出力する処理はありません。

```
〈if 文〉　　″if″　　〈printf ″if (″〉
　　〈条件式〉　　〈printf ″) {″ 〈¥_n〉〉
　　〈プログラム〉
　　[
　　　　″else″　　〈printf ″} else {″ 〈¥_n〉〉
　　　　〈プログラム〉
　　]
　　″end″　　〈printf ″}″ 〈¥_n〉〉
　　;
```

if 文は、以下のようなＣ言語に変換されます。

```
if(条件式) {
　　プログラム
}
```

条件式とプログラムの部分は、後に説明する〈条件式〉と〈プログラム〉に記述されている処理が変換されて展開されます。
else 節がある場合には以下のようなＣ言語に変換されます。

```
if(条件式) {
　　プログラム
} else {
　　プログラム
}
```

次は while 文を見ていきましょう。

<while 文>	″while″	<printf ″while (″>
	<条件式>	<printf ″) {″ <¥_n>>
	<プログラム>	
	″end″	<printf ″}″ <¥_n>>
	;	

while 文は、以下のような C 言語の出力を生成させます。

```
while(条件式){
        プログラム
}
```

どんどん行きます。次は times メソッドです。

```
<times 文>      (<NUM #n>｜<変数 #n>)
                ″.″ ″times″            <printf ″{int ct; for (ct=0; ct<″>
                                        <printf #n>
                                        <printf ″; ct++) {″ <¥_n>>
                ″do″
                <プログラム>
                ″end″                   <printf ″}}″ <¥_n>>
                ;
```

times メソッドは、for 文に変換されるのですが、ループ回数をどうやって制御するかが問題となります。
ここではカウンタとして変数 ct をローカルな変数として使います。このように{int ct; 〜}としておくと、複数の times メソッドをネストさせても大丈夫です。
以下のように C 言語に変換されます。

```
{
        int ct;
        for (ct=0; ct<回数; ct++) {
                プログラム
        }
}
```

print 文は、文字列 STRINGS, 数値 NUM, 変数を表示できるように printf 文に展開されます。

```
<print 文>      "print"<表示項目>{","  <表示項目>};

<表示項目>
                  <STRINGS #s>
                            <printf 'printf("%s", "' #s '");'
                                                <¥_n>>
                  | <NUM #n>
                            <printf 'printf("%d", ' #n ');'
                                                <¥_n>>
                  | <変数 #v>
                            <printf 'printf("%d", ' #v ');'
                                                <¥_n>>
                  ;
```

次は代入文です。変数と=と数式で構成されます。

```
〈代入文〉       〈変数 #v〉
                ″=″                     〈printf #v〉
                                        〈printf ″ = ″〉
                〈数式〉                 〈printf ″;″ 〈¥_n〉〉
                ;
```

C 言語では、以下のようになります。

```
        変数 = 数式;
```

数式は、Tiny Ruby の構文の中では最も複雑なものです。しかし、どのような C 言語に変換されるか想像しながら、ひとつひとつ変換のための printf 文を追加していけば、そんなに複雑でもありません。

```
〈数式〉         〈expradd〉;
〈expradd〉      〈exprmul〉
                { ″+″                   〈printf ″+″〉
                        〈exprmul〉
                | ″-″                   〈printf ″-″〉
                        〈exprmul〉
                }
                ;
〈exprmul〉      〈exprID〉
                { ″*″                   〈printf ″*″〉
                        〈exprID〉
                | ″/″                   〈printf ″/″〉
                        〈exprID〉
                | ″/″                   〈printf ″/″〉
                        〈exprID〉
                }
                ;
```

```
<exprID>        "+" <exprterm>
                | "-"                   <printf "-">
                        <exprterm>
                | <exprterm>
                ;
<exprterm>      "("                     <printf "(">
                        <数式>
                ")"                     <printf ")">
                | <NUM>                 <GETTOKEN #n>
                                        <printf #n>
                | <変数 #v>             <printf #v>
                ;
<条件式>        <数式>
                (">=" | ">" | "==" | "!=" | "<=" | "<")
                                        <GETTOKEN #op>
                                        <printf #op>
                <数式>
                ;
<変数 #x>
                <RANGE #x a z>
                ;
```

最後です。
コメントは、デカルト言語の改行まで読み飛ばす SKIPCR 述語を使います。
そのためコメント部分は、C 言語として何もは出力されません。

```
<コメント>      "#" <SKIPCR>
                ;
```

これで C 言語へ変換する構文を一通り追加しました。お疲れさまです。
次は、コンパイルするために入力ソースと出力ファイルを処理する部分を追加します。

6. 引数、サフィックス変換、入力ファイルと出力ファイル

コンパイラ本体である<TinyRuby>は以下の処理から起動されます。

```
<compile>
        ::sys<args #x>
        ::sys<nth #inputfile #x 1>
        ::sys<suffix #outputfile #inputfile c>
        <print inputfile #inputfile>
        <print outputfile #outputfile>
        ::sys<openw #outputfile
                ::sys<openr #inputfile <TinyRuby>>>
        ;

?<compile>;
```

順に説明していきましょう。

1. args 述語は、コマンドラインの引数を変数#x に設定してます。
2. 次の nth 述語は args で得た引数の２つ目(nth では最初の要素が 0 から始まる)を取り出して入力ファイル#inputfile に設定しています。
3. suffix 述語は#inputfile の拡張子を c に変換して、出力ファイル#outputfile に設定します。
4. print で入力ファイルと出力ファイルを画面に表示します。
5. openw 述語で出力ファイルをオープンし、openr 述語で入力ファイルをオープンして、コンパイラ本体である TinyRuby 述語を呼び出します。
6. ?<compile>で、上の処理を実行しています。

これですべての処理の説明が終わりました。

7. Tiny Ruby のすべて

ソース全体をここに置きます。

```
<compile>
        ::sys<args #x>
        ::sys<nth #inputfile #x 1>
        ::sys<suffix #outputfile #inputfile c>
        <print inputfile #inputfile>
        <print outputfile #outputfile>
        ::sys<openw #outputfile
                ::sys<openr #inputfile <TinyRuby>>>
        ;


<TinyRuby>                              <print "#include <stdio.h>">
                                        <print "int main() {">
                                        <print "int a,b,c,d,e,f,g,h,i,j,k,l,m;">
                                        <print "int n,o,p,q,r,s,t,u,v,w,x,y,z;">
                <プログラム> <EOF>       <print "exit(0);">
                                        <print "}">
                ;

<プログラム>    {[<実行文> { ";" <実行文>}] (<コメント> | <CR>)};


<実行文>        <if 文> | <while 文> | <print 文> | <times 文> | <代入文> ;


<if 文>         "if"                    <printf "if (">
                <条件式>                <printf ") {" <¥_n>>
                <プログラム>
                [
                        "else"          <printf "} else {" <¥_n>>
```

```
                <プログラム>
            ]
            "end"                   <printf "}" <¥_n>>
            ;

<while 文>  "while"                 <printf "while (">
            <条件式>                <printf ") {" <¥_n>>
            <プログラム>
            "end"                   <printf "}" <¥_n>>
            ;

<times 文>  (<NUM #n> | <変数 #n>)
            "." "times"             <printf "{int ct; for (ct=0; ct<">
                                    <printf #n>
                                    <printf "; ct++) {" <¥_n>>
            "do"
            <プログラム>
            "end"                   <printf "}}" <¥_n>>
            ;

<print 文>  "print" <表示項目> {"," <表示項目>};

<表示項目>
              <STRINGS #s>          <printf 'printf("%s", "' #s '");' <¥_n>>
            | <NUM #n>              <printf 'printf("%d", ' #n ');' <¥_n>>
            | <変数 #v>             <printf 'printf("%d", ' #v ');' <¥_n>>
            ;

<代入文>    <変数 #v>
            "="                     <printf #v>
                                    <printf " = ">
            <数式>                  <printf ";" <¥_n>>
            ;

<数式>      <expradd>;
```

```
<expradd>       <exprmul>
                { "+"                   <printf "+">
                        <exprmul>
                | "-"                   <printf "-">
                        <exprmul>
                }
                ;

<exprmul>       <exprID>
                { "*"                   <printf "*">
                        <exprID>
                | "/"                   <printf "/">
                        <exprID>
                }
                ;

<exprID>        "+" <exprterm>
                | "-"                   <printf "-">
                        <exprterm>
                | <exprterm>
                ;

<exprterm>      "("                     <printf "(">
                        <数式>
                ")"                     <printf ")">
                | <NUM>                 <GETTOKEN #n>
                                        <printf #n>
                | <变数 #v>             <printf #v>
                ;

<条件式>        <数式>
                (">=" | ">" | "==" | "!=" | "<=" | "<")
                                        <GETTOKEN #op>
                                        <printf #op>
                <数式>
                ;
```

```
<変数 #x>
                <RANGE #x a z>
                ;

<コメント>      "#" <SKIPCR>
                ;

?<compile>;
```

8. Tiny Rubyの実行

前項で示したTiny Rubyコンパイラをファイルtinyrubyとして保存してください。
ここでは、例としてRubyのソースとして、trb1.rbを例題として使います。
trb1.rbには以下を保存しておいてください。

```
# Tiny Ruby example No.1
a=1; b=2
c= a + b
print "display ", c, "¥n";
10.times do
        print a
        a = a + 1
end
print "¥n"
```

実行します。

1. デカルト言語の引数に tinyruby のプログラムとコンパイルする
   Ruby ソースを指定します。

```
$ descartes tinyruby trb1.rb
```

2. ソースに指定した Ruby ソースの拡張子を c に変換した C 言語のソースが
   出力されているはずです。
   それを、C コンパイラでコンパイルします。

```
$ gcc trb1.c -o trb1
```

3. 出来上がったプログラムを実行してみてください。

```
$ ./trb1
 display 3
 12345678910
```

4. 試しに本家ホンモノの ruby で実行してみます。

```
$ ruby trb1.rb
 display 3
 12345678910
```

同じ結果が出ました。感激です。

9. Tiny Rubyのコンパイル出力

「8. Tiny Rubyの実行」で、trb1.rbをコンパイルした結果出力であるtrb1.cの中身を見てみましょう。

```
#include <stdio.h>
int main() {
int a,b,c,d,e,f,g,h,i,j,k,l,m;
int n,o,p,q,r,s,t,u,v,w,x,y,z;
a = 1;
b = 2;
c = a+b;
printf("%s", "display ");
printf("%d", c);
printf("%s", "¥n");
{int ct; for (ct=0; ct<10; ct++) {
printf("%d", a);
a = a+1;
}}
printf("%s", "¥n");
exit(0);
}
```

インデントが無く読みづらいですが、確かにC言語のソースに見えます。
タブを適切に入れてインデントを直してみます。

```
#include <stdio.h>
int main()
{
    int a, b, c, d, e, f, g, h, i, j, k, l, m;
    int n, o, p, q, r, s, t, u, v, w, x, y, z;
    a = 1;
    b = 2;
    c = a + b;
    printf("%s", "display ");
    printf("%d", c);
    printf("%s", "¥n");
    {
        int ct;
        for (ct = 0; ct < 10; ct++) {
            printf("%d", a);
            a = a + 1;
        }
    }
    printf("%s", "¥n");
    exit(0);
}
```

どうでしょうか、元のTiny Rubyのソースが、C言語のソースに変換されているのが分かるでしょうか。