日本語プログラミング言語コンパイラの開発

武山　秀寛

目　次

――日本語プログラミング言語コンパイラの開発――

1　序論

(1) 目的

現在のプログラミング言語の多くは、英語をベースとしている。使用する人が日本人であったとしても、入力の容易さや表現の一貫性を考えると、英語を使用する事は非常に理にかなっているといえる。しかし、全くプログラミングに触れた事のない初心者にとっては、アルファベットで書かれたコードは意味不明な記号の羅列にしか見えないようである。

プログラミング自体は、全くの初心者にとっても決して難しいものではない。しかし、特に英語力不足が叫ばれる日本人にとっては、専門性の高い英単語を多用したプログラムの理解には 1 つの壁があるのもまた事実である。そのような言葉の壁に加え、プログラムの構造自体を理解する際にもまた壁が存在する。日本人の初心者がプログラミングを学ぶ際には、この 2 つの壁を同時に乗り越えなければならない。

そこで、言語の違いという壁を取り払い、プログラミングそのものの学習を進めるための言語として、日本語プログラミング言語を開発した。

(2) 既存のものとの比較

既存の日本語プログラミング言語としては、「ひまわり」や、その後継言語である「なでしこ」等が挙げられる。これらはそれ単体での完成度を高める事を重視しており、その文法等にはかなりの独自性がみられる。例えばなでしこでは、**リスト 1** のように書くと「こんばんは」という文字列が 10 回表示される。これと同じ事を本言語では、**リスト 2** の

ような形で記述する。一見すると前者のほうがスマートで分かりやすいが、一般的なプログラミング言語では、このように回数を指定して繰り返す事ができるような構文が用意されているとは限らず、後者のように実行回数を数えるための変数を用意する事が多い。例えば C 言語では **リスト 3** のように記述する。特に、`while` を使った例は本言語での記述とほぼ同一である。

私の日本語プログラミング言語においては、それ自身で全ての事ができるほどの完成度を持たせる事を目的とはしていない。初めてプログラムに触れるような初心者が、この言語を利用してプログラミングの感触をつかみ、またその楽しさを知ってもらう事で、他の言語の学習をシームレスに進めるための地盤を作成する。最終的には C や Python などの一般的な言語の学習へと繋げてゆけるようにする事が目的である。

また、ひまわりやなでしこが GUI アプリケーションの作成を中心とする中で、本言語ではコンソールアプリケーションの作成を目的とする。コンソールアプリケーション作成がプログラミングの基本であるという事も CUI を使用する理由の 1 つではあるが、将来的にウェブ系のアプリケーション作成へつなげてゆきたいという事が大きな理由である。この点において、本言語は他の日本語プログラミング言語と比較して初心者には少し厳しいものになっているといえる。しかし、プログラミングの学習と比較すれば、最低限のコマンドプロンプト使用法を学ぶ事はそれほど困難ではないはずである。

2　Intp の拡張

(1) Intp とは

日本語プログラミング言語を作成するにあたり、そのベースとして、『Ruby を 256 倍使うための本 無道編』（青木峰郎著）[(1)] の中で作成する、Intp というコンパイラを利用する事とした。この本は、Ruby 版の Yacc である Racc の使い方を、Racc の開発者である青木自らが説いたものである。青木は、Racc の使い方を説明する際に、実際に小規模なコンパイラを作成している。その結果として出来上がったものが Intp である。

Intp の構文規則は Ruby を基本としており、関数呼び出しの方法や繰り返しの構文などは、ほとんどが Ruby での記述方法と同一である。初心者向けの 1 冊の書籍で解説できる範囲であるという事もあり、必要最低限の機能しか実装されていないが、わずかな拡張を行えば、日本語プログラミング言語を開発するためのベースとして十分な機能を持つ。まずは、Intp に実用的なプログラミング言語としての機能を持たせるための拡張作業を行う。

(2) 比較演算子

Intp においては、比較演算子として==演算子のみが実装されている。比較演算子は、条件分岐や繰り返しの条件を記述するために使用される事が多い。==演算子だけで条件を記述する事も不可能ではないと思われるが、これからプログラミングを始めようという初心者にそれを求める事は酷である。==以外の一般的な比較演算子である「>，<，>=，<=」は、初心者のプログラミング作業においても頻繁に使用されるであろう

し、また他のプログラミング言語の多くがこのような比較演算子を実装している事からも、これは利用者の使い方でカバーすべき問題ではない。コンパイラ側で実装すべきものである。

比較演算子の実装は容易である。まず、Intp における==演算子の実装を**リスト 4** に示す。このコードでは、字句解析部において、まず正規表現によって前文一致で文字列'==' とのマッチングを行う。そして、一致した場合には、イコールを示す EQ という識別子と共に、それが見つかった行の行番号と文字列'==' を構文解析部に送っている。

それを受け取った構文解析部は、送られてきた識別子を並べ、構文規則に一致するかどうかの確認を行う。今回の場合、==を示す識別子は EQ である。また、その前後にある expr は、文を表す。文には変数名や数値等も含まれるため、今回の「expr EQ expr」という規則の場合、「a == 10」などが該当する。勿論、このような条件文は単独で使われる事は少なく、多くの場合は if 文や while 文などの条件部分において使われるものである。

ここに他の比較演算子を追加する事は容易である。まず、字句解析部において、「>, <, >=, <=」のそれぞれにマッチする正規表現を追加する。この 4 つは、\A[><]=?という 1 つの表現で表記する事が可能である。==も含めて 1 つの正規表現で記述する事も容易であるが、Ruby の when 文では複数条件を指定する事ができるため、可読性等も考慮して、従来の「==」と今回の「大なり小なり」を併記する形とした。

また、Intp においては、識別子としてイコールを表す EQ を使用していたが、他の比較演算子を追加する事に伴い、これを COMPAIR に変更した。さらに、Intp において'==' という形で直接文字列を送っていた部分を$&.intern という形にし、正規表現にマッチした文字列全体を

送るように変更した。これにより、構文解析部ではこれら全ての比較演算子を意識せずとも、COMPAIR という 1 つの識別子を用いるだけで、全ての比較演算子に関する構文を記述できる。変更後の字句解析部は**リスト 5** の通りである。

字句解析部の変更にあわせて構文解析部も変更する。Intp における構文解析部では、「expr EQ expr」という構文規則で== 演算子を用いた条件式を記述していた。字句解析部での識別子名変更に伴い、これを「expr COMPAIR expr」と変更する。

また、Intp では FuncallNode.new の第 3 引数として'==' という文字列を直接記述していたが、これを字句解析部から送られてきたものに変更する必要がある。Racc においては、構文解析部でそれぞれの識別子の持っている値は val という配列に入れられており、それが字句解析部で送られるデータと対応している。今回の場合、expr、COMPAIR、expr という 3 つの識別子にそれぞれ val[0]、val[1]、val[2] が対応する。よって、COMPAIR に対応する値は val[1] である。また、この val[1] の値は字句解析部の [lineno, $&.intern] にあたるものである。そのため、$&.intern に対応するのは val[1][1] である。これを適用すると、構文解析部は**リスト 6** のように記述できる。

以上で、比較演算子の実装に関わるほとんどの変更は終了したが、続いて優先度の設定をしなければならない。Intp では、演算子それぞれに優先度を設定している。最優先されるのが単項マイナスであり、次が乗除算、さらに加減算、イコール演算子の順である。また、それを表すコードは**リスト 7** の通りである。

ここにその他の比較演算子を追加する。イコールを含む比較演算子を示す識別子は COMPAIR であるため、単純に EQ を COMPAIR に置き換え

るだけで、全ての比較演算子を同一の優先度に設定できる。よって、修正後のコードは**リスト 8** のようになる。以上で優先度の設定は完了である。

(3) elsif

続いて、elsif の実装を行う。Intp の if 文には、elsif に相当する機能が実装されていない。if 文と else 節は利用可能であるため、それを入れ子にしてゆけば、elsif を使わずとも同等の処理を行う事は可能である。しかし、利便性を考慮すると elsif の実装は不可欠であると考える。

elsif の実装は、前述の比較演算子のような単純な置き換えでは実現できない。字句解析部、構文解析部の変更と共に、構文木の if ノードを変更する必要がある。字句解析部の変更は、ハッシュ RESERVED 中に「'elsif' => :ELSIF,」の 1 行を加えるだけである。

構文解析部には、比較的大規模な変更が必要である。まず、elsif の構文全体を示す elsif_stmt という構文の構造を記述する。そして、if_stmt の中に elsif_stmt を組み込む。変更後の if_stmt の構文規則を**リスト 9** に示す。この構文規則の中で elsif_stmt の位置を見てみると、実際の文法と全く同様な場所に位置している事が分かる。

Intp において、IfNode.new の引数として送られていたのは stmt、stmt_list、else_stmt の 3 つの値である。elsif の追加に伴い、ここに elsif_stmt が追加される。上記の例では、val[5] がこれにあたる。

次に、elsif_stmt の最も単純な形を**リスト 10** に示す。elsif_stmt には、最初に elsif があり、条件文となる stmt がそれに続く。そし

て、条件文の終わりを告げる then があり、行の終わりを示す EOL によって行が終わる。そして、続く stmt_list が、条件が真の場合に実行される一連のコードを表している。

しかし、この例では elsif 節は必ず 1 回でなければならず、elsif 節のない if 文や、elsif が複数回繰り返される場合は表現できていない。elsif は 1 つの if 文に複数存在する可能性があり、またその個数はいくつであるのか分からない。さらに、elsif が存在しない可能性もある。そのため、再帰的表現による記述が必要である。これは必ずしも elsif_stmt の構文規則において定義しなくとも、if_stmt の構文規則において定義する事も可能である。しかし、elsif_stmt の中でどのような処理が行われているのかを if_stmt 側で意識する事なく利用できるようにする事は、プログラム強度を高める事に繋がるため、可能な限り elsif_stmt の中で完結するようなものとする事が望ましい。

elsif_stmt において、elsif がない場合及び複数回の elsif をサポートする構文規則は**リスト 11** の通りである。1 回目の elsif を示すのが、「ELSIF stmt THEN EOL stmt_list」という規則である。ここで、条件文と、条件が真の場合に実行される一連のコードを配列にし、さらにそれを配列にして result に代入する。そして、2 回目以降は「elsif_stmt ELSIF stmt THEN EOL stmt_list」が使われ、1 回目に作られた配列に、条件文及び条件が真の場合に実行される一連のコードを示す配列を追加する。result の初期値は val[0] である事から、elsif_stmt の値、すなわち前回までに作られた配列が result の初期値である。これに、push メソッドを利用して配列を追加する。この方法であれば、elsif の数がいくら増えようとも対応できる。なお、elsif がない場合を示すのは最下の「|」のみの行である。これは、

elsif_stmt として「何もないもの」が認められるという事を示している。

続いて、node.rb の修正を行う。IfNode.new に引数を追加した事に伴い、IfNode の定義を変更する必要がある。また、同時に elsif を処理するためのコードも記述しなければならない。修正後の IfNode は**リスト 12** である。

まず、initialize の引数として elsif_stmt を追加し、さらに「@elsif_stmt = elsif_stmt」という行を追加した。

続く evaluate には、全ての elsif_stmt を評価するためのコードに加え、else を実行するか否かを決めるフラグを置いた。追加した部分のコードを、**リスト 13** に示す。

3　日本語による表現の実装

以上で Intp の拡張が一通り完了し、最低限の機能を持った教育用言語として使用できるようになった。これより、それをベースとして、いかにして日本語による表現を実装するのかという点に論点を移す。

(1) 字句解析部

(i) Intp における字句解析

日本語による表記を実装する事に先立ち、まずは Intp における字句解析の手法を改めて確認する。Intp における字句解析は、1 行を単位として行われる。読み込まれた行は、まず strip!メソッドによって前後の空白文字を取り除かれる。次に、正規表現によって前方一致でトークンとのマッチングが行われ、マッチした場合にはその識別子と行番

号、そして予約語名や数値、文字列といったデータが構文解析へと送られる。

数字及び「.」のみからなる数値や、引用符で囲まれた文字列などの判定は容易である。しかし、if や while 等の予約語名、i 等の変数名、puts 等の関数名はいずれも同様な英数字の並びであり、それらを適切に判別する事は比較的困難である。予約語に関しては、使われる予約語名が決まっているため、字句解析部での判別が比較的容易である。しかし、関数名に関してはその数が膨大であり、また、利用者が対象ソース中において関数定義を行う場合には、字句解析部では対処しきれない等の問題が生じる。

これらに対処するため、Intp においては以下のような 2 つの方法が取られている。まず、予約語を判別する方法として用いられているのが、予約語名の一覧を事前に RESERVED という名のハッシュに格納しておくという方法である。そして、変数名と関数名を判別する作業に関しては、字句解析部では行わずに構文解析部に任せてしまうという手法をとる。

仮に対象ソース中に'if' という文字列があった場合、字句解析部はまず、ハッシュ RESERVED 中に if が含まれているかどうかを確認する。そして、もしそこに if が含まれていれば、if は予約語であると断定する。その場合、RESERVED 内から if をキーとして取り出せる、IF という識別子を構文解析部に送る処理を行う。そして、それ以外の場合には、変数または関数を示す IDENT という識別子と共に、その文字列を構文解析部へと送る。この際、それが変数であるのか関数であるのかという点に関しては、字句解析部は一切関知しない。

(ii) 日本語の字句解析

以上が Intp における字句解析の概要である。これを日本語表記に対応させるためには、いくつかの方法が考えられる。まず、単純に RESERVED の中に日本語を表記する方法がある。例えば、Intp において「'if' => :IF」という対応であったものを、「'**もし**' => :IF」と変更すれば、あとは\A[a-zA-Z][a-zA-Z0-9_]*となっている正規表現部分を、半角英数及び全角の全ての文字の 1 回以上の繰り返しを表す\A\w+という表現に変更するだけである。

しかし、この方法では柔軟な日本語表現は不可能である。その理由としては、第一に、日本語には単語間のスペースがない事が挙げられる。前述の正規表現によるマッチングでは、条件に当てはまる最長の文字列がマッチする。すなわち、自然な日本語文のような連続した全角文字の場合、その文全体にマッチしてしまう事になる。そのため、日本語で記述する際にも単語間にスペースを入れる事が求められる事になる。

第二に、同義語の使用に関する制限が挙げられる。例えば、先ほどの例における if に相当する日本語を記述する場合、必ず「**もし**」と記述しなければならず、「**もしも**」等の同義表現はそのままでは利用できない。

これをサポートするためには、**リスト 14** のように予約語を追加する事が考えられる。しかし、この方法ではハッシュ RESERVED の大きさが莫大なものとなり、可読性を損なう。また、正規表現と比較すると冗長な表現が多く含まれ、コードを記述する際の効率が悪い。

これらの問題を解決するため、多少の手間はかかるが、全ての予約語をそれぞれ正規表現で記述し、またそれら 1 つ 1 つに対して、英数字からなる一意の識別子を設定する方針とした。また、関数に関しては最低限必要な一部関数の実装のみに止め、これも全てを正規表現で記述し

た。なお、変数名としては Intp 同様に半角英数およびアンダーラインを使用する事としたため、変数を認識する部分に関しては Intp のものがそのまま利用可能である。

(2) 構文解析部

(i) 日本語の構文解析

構文解析部において、日本語をそのまま記述する事は少々難しい。まず、Intp においては代入を示す構文を**リスト 15** のように記述している。Intp の字句解析部では、対象の文字列が変数名や予約語、数値、文字列などであるかどうかを比較した後、そのどれにも当てはまらなかったものに対して、`\A.` という正規表現を用いてマッチングを行う。すなわち、それがどのような文字であるかに関わらず、先頭の 1 文字のみを取り出して構文解析部へと送る。その文字が構文解析部で利用される場合、上記の例のように`'='` といった表記となる。

正規表現の`\A.` は、日本語での表記にも対応している。日本語の場合でも、1 バイト分のみにマッチするのではなく、1 文字分にマッチする。これを利用すれば、例えば**リスト 16** のように構文を記述可能である。

これを拡張し、全角 1 文字に対応する正規表現のみでなく、それぞれの語句に対応する正規表現を用意すれば、より日本語の文章に近い形で構文規則を記述する事ができる。一例を挙げると、「`IDENT` **に** `expr` **を代入する**」という前述の文を表す規則は**リスト 17** のように記述する。

この方法を用いると、より日本語の表現に近く、理解しやすい形で構文規則を記述する事が可能である。しかし、その反面、文字列を直接利用しているために不必要な処理速度の低下を招く事となる。また、現在の開発プラットフォームである Ruby ではこのような文字列による処

理が可能であっても、今後移植する可能性の高い C 言語上の Yacc 等では、このような動作が可能かどうか定かではない。

そのため、今回は Intp 同様にシンボルを受け渡す形とした。これにより、不必要な処理速度の低下を防ぐ事ができ、また同時に他言語への移植性を高められる。シンボルを用いた実際の記述例は**リスト 18** の通りである。

(ii) 同義語及び文末表現の差異の吸収

通常のプログラミング言語では、ある 1 つの処理を行うための表現は基本的には 1 つのみである。例えば、`'hello'` という文字列を表示したい場合、Ruby においては**リスト 19** のような記述を行う。Ruby には、`print` や `p` のように、同様の処理を行うかのようにも見える関数も存在するが、厳密にはこれらは全て異なる。`puts` では表示後に改行が行われるのに対し、`print` では改行が行われない。また、`p` においては、引数として与えられた値が文字列の場合は引用符を付け、数値の場合は引用符を付けずに表示する。よって、これらはそれぞれ別の処理を行っているといえる。例外として、コマンドライン等から与えられる引数を収めた組み込み変数である`$*`と `ARGV` のように、異なる表現であるにも関わらず、全く同じものを示すものも存在する。しかし、大部分においては、表現は一意に定まっていると言って差し支えない。

日本語の自然言語表現においては、多数の同義語が使用される可能性がある。また、その文末表現も多様である。例えば、「変数 `a` に文字列`'hello'` を代入する」という処理を行う場合、Ruby においては**リスト 20** のように記述する事が通常である。これに対して、同様な表現を日本語で記述しようとする場合には、**リスト 21** のような表現が例として挙げられる。これら 4 つの表現の意味する事は完全に同一である。日

本語の自然言語表現によるプログラムの記述を可能にするためには、このような表現の違いを吸収し、これらを同一のものとして認識しなければならない。

この問題をクリアして日本語プログラミングを行うためには、2 つの方法が考えられる。1 つは、同義語の使用の制限ならびに文末表現の統一を行い、決められた表現以外は全く認めないという方法である。一般的なプログラミング言語はこの方法を採用してるといえる。これにより、コンパイラでの処理が単純化できると共に、誰が書いたソースコードであっても、基本的に同じ内容になるというメリットを享受できる。

もう 1 つの方法は、同義語や文末表現の多くをコンパイラにおいてサポートするというものである。あらゆる日本語表現の可能性を考え、それをコンパイラに実装する事ができれば、日本語の表現の違いを意識する事なくプログラムを記述する事が可能になる。

これを実装するためには、2 つの方法が考えられる。1 つは字句解析部に正規表現を用いて実装する方法であり、もう 1 つは構文解析部において BNF 記法を用いて記述する方法である。

まず、字句解析部で用いる方法を挙げる。字句解析部において、全ての予約語及び関数を正規表現で記述する事により、文末表現等に自由度を持たせる事が容易となる。一例を挙げると、「**〜を表示する**」という、puts に相当する関数を実装する場合に、「**\A を表示（する|して改行（する）?)?**」という正規表現を用いて表現する事で、「**を表示**」「**を表示する**」「**を表示して改行**」「**を表示して改行する**」という 4 種の記述方法をまとめて扱う事ができる。代入の例においては、**リスト 22** のような 2 つの記述における言い回しの違いを吸収する事ができる。

上記のような正規表現による実装においては、単純な言い回しの違い

や同義語による差異が吸収可能である。しかし、それだけでは、助詞の順番の違いによる語順の変化等への対応が不十分である。例えば、**リスト 23** のような違いには対応できない。この 2 文の意味は全く同一である。しかし、その語順は異なっており、これを同一のものとして扱うためには構文規則の記述が必要である。具体的には、**リスト 24** のような記述を行う。

字句解析部では、正規表現にマッチした文字列に対応した識別子を構文解析部へと送る処理を行う。これは今までに述べた通りである。そして、構文解析部において、識別子の並びが構文規則と一致するかどうかを判定する。

今回の場合、Intp に実装されていた構文規則は「IDENT '=' expr」という最初の 1 行のみである。それを残したまま、日本語によるプログラムの並び方を示す新たな構文規則を並列に記述した 。「IDENT NI expr WO DAINYU」は「a **に** 'hello' **を 代 入** 」といった文にマッチし、また「expr WO IDENT NI DAINYU」は「'hello' **を** a **に代入**」などの文とマッチする。そして、そのどちらにマッチしようとも、一律に assign に還元される。上流では、この assign が代入を示す文として利用されるが、その時点においては、「IDENT NI expr WO DAINYU」と「expr WO IDENT NI DAINYU」のどちらにマッチたのかは一切意識されず、どちらも全く同じ内容をもつ assign として処理される。

以上で、表現の差異の吸収に関する基本的な方法の記述は終了である。なお、今回の開発にあたっては、正規表現で記述可能な部分は、そのほとんどを字句解析部で処理している。しかし、処理の一貫性や移植性、また日本語処理という観点でみると、本来であれば全てを BNF 記

法による構文規則で記述するべきである。今回は時間の関係で断念したが、いずれは正規表現で行っている処理の全てを BNF を用いた表記へと書き換えたい。

4　言語仕様の策定

これより、今まで考えてきた方針を元に、実際に日本語プログラミング言語の言語仕様策定を行う。なお、今までにも日本語のコードを使用した説明を一部で行ったが、それらはこれ以降で策定する言語仕様に基づいている。

(1) 代入

代入文の仕様を決定する。まず、Ruby における代入文の例を**リスト 25** に示す。さらに、これを日本語で表現したものを**リスト 26** で示す。また、これを実現するために必要な字句解析部及び構文解析部は**リスト 27** の通りである。字句解析部においては、正規表現によって、「**する**」の有無を吸収している。また、構文解析部では、「**が**」と「**を**」の位置の違いを吸収する。

(2) 比較

条件分岐や繰り返しの条件に使用する、比較演算子の実装を行う。前述の Intp 拡張時に、「**>，<，>=，<=**」という 4 つの比較演算子を利用可能とした。これだけでも実用上それほど問題はないが、自然な文章に近づけるためには、これらにもそれぞれ日本語の表現を加えたほうが良い。そこで、**リスト 28** のような条件文が利用できるようにする。その

字句解析部、構文解析部は**リスト 29** の通りである。

字句解析部において特筆すべき点は、「より小さい」と「未満」を並列に記述している点である。また、構文解析部においては、`FuncallNode.new` の第 3 引数に比較演算子を示す文字列を置いている。機能補完後の Intp では、この部分を`$&.intern` という形にし、正規表現にマッチした文字列を用いていた。しかし、日本語のプログラムではそれが不可能なため、オリジナルの Intp と同様な形に戻している。

なお、本来ではここに、否定を表す「`a` **が** `b` **ではない**」という表現を加えるべきである。否定を表す比較演算子は`!=`だが、他の比較演算子と同様な記述を行っても動作しなかった。この機能の実装は今後の課題としたい。

(3) 条件分岐

まず、Ruby における条件分岐の基本的な構文を**リスト 30** に示す。これを日本語したものは**リスト 31** の通りである。また、字句解析部及び構文解析部は**リスト 32** のように記述した。

今まで行ってきた代入や比較の構文規則定義においては、Intp に実装されていた構文規則に新たな日本語規則を併記する事が通常である。その理由としては、Ruby の構文規則における各要素の並び順と、日本語プログラムにおけるそれが異なっている事が挙げられる。しかし、今回の条件分岐においては、機能補完版 Intp で用いていた `IF`、`THEN`、`ELSIF`、`ELSE` のそれぞれの識別子をそのまま利用している。それは、Ruby における要素の並び順と、日本語プログラムにおける要素の並び順が全く同一であり、単純な置き換えで日本語化が実現できたためである。

また、Ruby の end にあたる、条件分岐の終了を示すものの日本語化においては、基本的には「**終了**」という言葉を用いる事とした。しかし、終わりを示す日本語は多数存在するため、「**完了**」や「**以上**」のような、その他の記述でも動作するようにした。

(4) 繰り返し

Ruby における、while 文の基本文法は**リスト 33** のようなものである。また、これを日本語化すると**リスト 34** のようになる。これを実現するために追加すべきコードは**リスト 35** の通りである。なお、字句解析部の正規表現を展開すると出てくる「行がある間」という部分は、後述するファイル読み込み関数のために用意している。

(5) 関数

続いて、関数の実装に移る。まず。Ruby における一般的な関数の使用方法を**リスト 36** に示す。1 つ目の例が、最も基本となる例である。関数名のあとに括弧があり、その中に引数が収められている。そして、次のものがその括弧を省略したものである。また、4 つ目の例のように、引数がない場合には括弧を省略して関数名のみを記述しても良い。

これを日本語化する場合について考える。上の 2 つのように引数がある場合に関しては、**リスト 37** のような日本語が適応する。ここでは、'hello' が引数であり、また「**を表示**」が関数名である。これを表す構文規則は**リスト 38** の通りである。

なお、本来であれば、stmt ではなく、引数のリストを表す args を使いたい。しかし、Intp とは異なって引数が最初に来ているため、これではうまく認識できないようである。

また、関数を示す識別子として、FUNCTION を利用する事とする。なお、日本語のローマ字表記による識別子と英語表記によるものの使い分けとして、ある 1 つの言葉に対応する識別子はローマ字表記とし、今回の FUNCTION のように複数の言葉に対応するものは英語表記にする事を基本とする。

字句解析部から送られる word.intern は、その日本語関数に対応した Ruby の関数名をシンボル化したものである。Intp においては、正規表現で取得した関数名を利用していたが、今回は関数名を文字列で直接記述している。

(i) 〜を表示する

これは、Ruby の puts に相当する関数であり、記述方法は**リスト 39** の通りである。なお、括弧で括られた部分は省略可能である。また、「**〜**」の部分は引数を示す。

字句解析部は**リスト 40** の通りである。なお、構文解析部に関しては、前述の「stmt FUNCTION」を用いる。

(ii) 〜を改行なしで表示する

この関数は、Ruby の print 関数に相当する。すなわち、改行を伴わない表示である。記述方法及び字句解析部を**リスト 41** に示す。

(iii) 改行

この関数は、改行のみを行う関数である。すなわち、引数を伴わない puts と同義である。記述方法を**リスト 42** に示す。

また、字句解析部及び構文解析部は**リスト 43** の通りである。他の関数とは異なり、識別子 FUNCTION は利用せず、KAIGYO という新たな識別子を用いている。これは、他の関数とは異なり、引数を持たないためである。現在のところは引数なしの関数がこれのみであるために専用の

識別子を用いているが、同様の関数を追加する場合には、構文解析部を共用できるように、汎用性のある識別子名にすべきである。

(iv) ファイルから 1 行ずつ〜に読み込む

この関数では、コマンドラインから入力された引数で指定されたファイルを 1 行ずつ読み込み、指定された変数に代入する。Ruby における gets に相当する関数である。まず、引数にファイル名を指定する例を**リスト 44** に示す。src.txt が日本語プログラムのソースであり、そのプログラム内で読み込みたいファイルが data.txt である。また、プログラム内での記述方法は**リスト 45** の通りである。

なお、細かい日本語表現の違いに対応できるように正規表現を作成しているため、厳密に上記と同じ構文でなくとも動作する。上記以外の日本語表現に関しては、**リスト 46** で挙げる字句解析部の正規表現を参照して欲しい。

この関数は、字句解析部、構文解析部共に、通常の関数とは大きく異なった内容である。字句解析部においては、正規表現中に、変数名を示す [a-zA-Z][a-za-z0-9_]* という表現がある。これはグループ化されており、$3 という特殊変数を用いてマッチした文字列を取り出す事が可能である。GETS という識別子と共に、ここで得た変数名を構文解析部へと送る事になる。

それを受け取った構文解析部では、2 つの処理を行う。gets 関数の呼び出しと、その結果の変数への代入である。通常、1 箇所のアクションで実行される処理は 1 つである。そして、その結果が result 変数に代入され、上の階層へと送られてゆく。しかし、今回のように 2 つのアクションを行う場合には、最終的な結果のみを result 変数に代入しなければならない。

そのため、テンポラリ変数として tmp を用意した。通常の関数呼び出しと同様に、FuncallNode.new を呼び出して関数 gets を実行し、その結果を変数 tmp に保存する。そして、次の AssignNode.new において、tmp の値を変数に代入する。その結果が、最終的に result の値となる。

ファイル読み込み変数を実装するにあたり、さらに変更しなければならない部分がある。それは、引数を示す組み込み変数 ARGV である。Ruby の gets 関数は、引数として与えられたファイルの中身を読み込む。例えば、「ruby jpprog src.txt data.txt」という形でスクリプトが実行された場合、Ruby から見た引数は src.txt と data.txt の 2 つである。そのため、gets では最初の src.txt を読み込んでしまう。これを回避し、data.txt を読み込むためには、ARGV の値を 1 つずらしてやる必要がある。そこで利用するのが、shift メソッドである。具体的には、parser.y 中の parse 関数定義の最初に**リスト 47** の 1 文を追加する。

なお、ファイル名を引数として指定する方法の他に、プログラム中に直接ファイル名を記述する方法に関しても検討を行った。しかし、この方法を用いるためには、ファイルのオープンとクローズの処理を行わなければならない。現在の Intp では、ファイルをクローズするために用いる、f.close のようなメソッドが実行できない。そのため、意図的にファイルをクローズする必要のない、引数を利用したファイル読み込み関数を採用した。

以上で、関数の実装は終了である。なお、今回の関数の実装は、最低限必要と思われるもののみに止めた。このままでもプログラミング導入用言語として最低限の機能は備えているが、今後はさらに多くの関数を

実装すべきである。

(6) エラーメッセージ

Intp において、構文エラー等が発生したときに表示されるエラーメッセージは**リスト 48** のようなものである。ここで、`#{@fname}`はソースファイル名を示し、`#{line}`は行番号を示す。そして、`#{v.inspect}`が、エラーが発生したトークンである。これを**リスト 49** のように日本語化した。また、エラーメッセージが見やすいように、前後に空行を挿入した。

Intp では、`#{v.inspect}`に、エラーが発生した時点での関数名等が収められている。しかし、日本語プログラミング言語においては、プログラムのソースに書かれている日本語プログラムと、実際に実行される Ruby のコードが異なるという問題がある。Intp と同じ仕様のまま実行すると、Ruby のコードが表示されてしまい、利用者にとっては分かりにくいものとなる。ここでは、日本語のコードを表示するべきである。

これを実現する手段の大部分は、既に今までのコード中に示してある。**リスト 50** に一例を挙げる。このトークンに関しては、「**もし**」という日本語と、`IF` という識別子が一意に対応している。すなわち、本来であれば識別子の利用のみで足り、関数のように関数名などを送る必要はない。にも関わらず、ここではあえて、`$&.intern` という、正規表現にマッチした全体、すなわち「**もし**」という文字列をシンボル化して送っている。これは、プログラムを実行するためには全く必要のない処理である。しかし、エラーメッセージを表示する際には、ここで送られるシンボルがエラー発生個所として表示されるため、エラーメッセージの日本語化を行う際には重要である。

なお、関数に関しては、Ruby の関数名を送る必要があるため、エラーメッセージの日本語化は行っていない。この部分の日本語化に関しては、送信データを配列化するなどして今後実装したい。

(7) 実行方法

以上で、日本語プログラミング言語の開発が一通り終了した。これより、実際にこの言語を実行する方法を示す。

本言語は Ruby で書かれている。よって、実行するためには Ruby をインストールする必要がある。Windows であれば、mswin32 版の Ruby[(2)] が利用可能である。また、本日本語プログラミング言語のソースファイルは、本論文の資料として添付する CD-ROM 及び、筆者の Web サイト[(3)]から入手可能である。

環境が整ったら、テキストファイルに日本語プログラムを記述し、実行する。パスが通っていれば、**リスト 51** のようにコマンドプロンプトに打ち込むだけで実行される。なお、ここでは日本語プログラムを記述したファイルのファイル名は `src.txt` とする。

5 結論

(1) 他者からの評価

この日本語プログラミング言語を、プログラミングを 1 年間学んでいる学生数人に試用してもらった。その結果、多くの人は違和感なく日本語でのプログラミングに溶け込めたようである。また、可読性という点では、日本人にとって日本語表記が大変有利であるという意見を多数頂いた。

しかし、プログラミング経験者であるが故の問題点も指摘された。それは、いかに日本語であろうとも、これもまた構文規則に縛られた新たなプログラミング言語であり、それを覚える事にも労力がかかるという点である。そのため、既に他の言語を理解している人にとっては、逆に効率が悪い。

また、日本語のバリエーションの少なさも指摘された。これが、前述のような、構文規則に縛られて直感的な記述ができないという問題に繋がっている。今後は可能な限り言葉のバリエーションを増やし、どのような表現で記述しても動作するようにしてゆきたい。

(2) 自らの評価

今回の日本語プログラミング言語開発にあたっては、初心者のプログラミング導入学習向けにターゲットを絞った。そういった意味では、目的を達成するだけの機能は実装できたように思う。しかし、汎用的な言語として利用するためにはまだまだ機能が足りない。

また、本来であればゼロからコンパイラを作成するはずだったが、時間の関係上、青木製作の Intp という小規模コンパイラを改変するに止まってしまったことは残念である。

(3) 今後の課題

時間の関係等で今回は実装できなかった、今後の課題となる機能は次のようなものである。

まず、比較演算子!=が挙げられる。他の比較演算子と同様に記述した

ものの、なぜか!=のみ動作しなかった。また、ファイル読み込み関数に関しては、メソッドが使用できないために、プログラム中でファイル名を指定してのファイル読み込みができない。さらに、関数定義の実装も課題である。

また、日本語の表現のバリエーションがまだまだ不十分である。思いつくままに日本語の表現を実装したが、まだ不足があると思われる。さらに、現在は正規表現を用いて表現の差異の吸収を行っているが、これを BNF 記法による表記に変更する。また、エラーメッセージの関数部分の日本語化も課題である。

注

（1）http://ruby256.hp.infoseek.co.jp/

（2）http://www.garbagecollect.jp/ruby/mswin32/ja/

（3）http://www.geocities.jp/japanese_programing/

文献表

青木峰郎 2001　『Ruby を 256 倍使うための本 無道編』アスキー

Brian W. Kernighan, Rob Pike 1985　『UNIX プログラミング環境』
アスキー