FlatShell の開発

FURUKAWA <flucium@flucium.net>

April 23, 2024

はじめに

Shell とは，オペレーティングシステム (OS) とユーザー間のインフェーフェイスとして機能するプログラムのことである．
よく知られているものとして，Unix shell である Bash や Zsh，Microsoft の PowerShell などがある．
FlatShell とは，独自の Shell であり，Unix shell などには該当しない．また，Unix shell を比較対象としている．

1　FlatShell の思想

“フラットな状態は良い”という思想のもとで開発している．
構文はもちろんのこと，処理系などの実装においても，可能な限りネストを深くしないように心掛けている．また，Unix shell の構文と比較して，自由な形式で書けることを意識している．

2　字句解析器

FlatShell では，入力に対してプリプロセッシングという事前処理を多段階で行い，その結果に対してスキャナとトークナイザを走らせ，トークンを得るようにしている．
以降，順を追って説明する．

2.1　プリプロセッシング

1. 入力を受ける．
2. 先頭から順にリードし，Sharp(#) がリードされた時点で，LF，CR（\n , \r），Semicolon(;) まで読み飛ばす．
3. 各行をみていく．空の行を読み飛ばしていく．（又は，空では無い行のみを拾い上げていく）．
4. LF，CR を Semicolon に置換する．
5. 最後に，ここまで処理した文字列を char のベクタに変換する．

2.2　スキャナ

条件一致するまで，char のベクタをリードし続ける．リードされた char は，順にバッファへと入れていく．条件一致した時点で処理を終了させる．
バッファを１つ目の戻り値とする．条件一致した char を２つ目の戻り値とする．条件一致した char にポジションは移動していない．

2.3 トークナイザ (小)

対象ごとに，小さなトークナイザを定義する．

1. 文字列と仮定する．スキャナは Whitespace 又は SYMBOLTABLE に当たらない限り，スキャンし続ける．スキャンし，得られた char のベクタを文字列に変換し，String Token とする．
   Whitespace 及び SYMBOLTABLE に関する例外として，Double quote(") 又は Single quote(') の範囲は，全て文字列として解釈する．

2. 数列と仮定する．Current char を確認し，10 進数ならば数列と再仮定する．スキャナは Whitespace 又は SYMBOLTABLE に当たらない限り，スキャンし続ける．スキャンし，得られた char のベクタを文字列に変換する．文字列を数列にパースする．パースに成功したら，Number Token とする．

3. 変数名と仮定する．Current char を確認し，Dollar($) ならば以降を変数名であると再仮定する．Dollar 以降を対象とする．スキャナは Whitespace 又は SYMBOLTABLE に当たらない限り，スキャンし続ける．スキャンし，得られた char のベクタを文字列に変換する．文字列がアルファベットから始まる場合には，それを Ident Token とする．

4. ファイルディスクリプタと仮定する．Current char を確認し，Att(@) ならば以降をファイルディスクリプタであると再仮定する．以降を対象とする．スキャナは Whitespace 又は SYMBOLTABLE に当たらない限り，スキャンし続ける．スキャンし，得られた char のベクタを文字列に変換する．文字列を数列にパースする．パースに成功したら，FD Token とする．

5. 特殊文字の扱い．以下の通りである．

   (a) Semicolon ならば，Semicolon Token とする．

   (b) Equal(=) がならば，Equal Token とする．

   (c) Ampersand（&）ならば，Ampersand Token とする．

   (d) Vertical bar（|) ならば，Pipe Token とする．

   (e) Greater than （>）ならば，Gt Token とする．(Version 0.0.1 では Redirect のオペレータとして使用する．)

   (f) Less than（<）ならば，Lt Token とする．(Version 0.0.1 では Redirect のオペレータとして使用する．)

   (g) Att ならば，4. を試みる．4. に失敗した場合には，２つの条件で処理を変える．
   条件１，ファイルディスクリプタをリードする関数そのものがエラーを返さずに，空の値（None）を返した場合には，End Of File として解釈する．つまり，EOF Token とする．
   条件２，ファイルディスクリプタをリードする関数がエラーを返した場合には，仮リード（peek char）を行い，その結果に基づいて処理を変える．仮リードの結果が Some であり尚且つ Whitespace，又は仮リードの結果が存在しない（None）場合には，Att を文字列として解釈する．つまり，String Token とする．
   それ以外は，エラーとする．

   (h) Dollar ならば，3. を試みる．3. に失敗した場合には，２つの条件で処理を変える．
   条件１，変数名をリードする関数そのものがエラーを返さずに，空の値（None）を返した場合には，End Of File として解釈する．つまり，EOF Token とする．条件２，変数名をリードする関数がエラーを返した場合には，仮リードを行い，その結果に基づいて処理を変える．仮リードの結果が Some であり尚且つ Whitespace，又は仮リードの結果が存在しない（None）場合には，Dollar を文字列として解釈する．つまり，String Token.
   それ以外は，エラーとする．

   (i) Double quote 又は Single quote ならば，1. に準拠する．

2.4 トークナイザ

トークナイザは，スキャナやトークナイザ（小）よりも先に Current char を確認し，適切なトークナイザ（小）を呼び出す．
もし，Current char が Whitespace ならば，Skip をする．

3　構文解析器

入力を受け取り，字句解析器に渡す．字句解析器は，スキャナとトークナイザを組み合わせて，入力をトークナイズする．
構文解析器は，トークナイズによって得られたトークン列に対して解析を行う．

3.1　ライトパーサ

ライトパーサといわれる，対象を限定した小さなパーサ群を定義する．

1. 文字列と仮定する．入力されたトークンが String Token ならば，String Expr とする．
2. 変数名と仮定する．入力されたトークンが Ident Token ならば，Ident Expr とする．
3. 数列と仮定する．入力されたトークンが Number Token ならば，Number Expr とする．
4. ファイルディスクリプタと仮定する．入力されたトークンが FD Token ならば，FD Expr とする．
   (a) Expr と仮定する．入力されたトークンが String，Ident，Number，FD Token のいずれかであれば，1.〜4. のライトパーサーにトークンを渡して，パースする．
   また，このライトパーサを Expr パーサとする．対象外としたいトークンを指定することができる．つまり，条件一致で Expr のパースを行うことができる．
   (b) Assign と仮定する．入力として受け取れるトークンの数は３つであり，配列として受け取る．つまり，[Token;3] である．３は定数とする．
   Left は Ident Token であることが期待され，Middle は Equal Token であることが期待される．Right は String，Number，FD Token であることが期待される．その上で，Left と Right を適切なライトパーサに渡し，パースを行う．また，Middle が Equal であるかも確認する．
   (c) Redirect と仮定する．２つの形式を想定する必要がある．入力として受け取れるトークンの数は，２または３である．入力されたトークンの数を確認し，２なら形式 i . へ，３なら形式 ii . へ渡す．
      i. オペレータが Lt ならば Left を FD0 とし，Gt ならば Left を FD1 する．Right は Expr パーサへ入力し，Expr を得る．オペレータが Lt または Gt でなければ，エラーとする．
      ii. オペレータが Lt または Gt であるかを確認し，そうでなければエラーとする．Left は FD Token であることが期待され，Right は String, Number, Ident, FD Token であることが期待される．Left と Right を適切なライトパーサへ渡して，パースする．
5. コマンド列と仮定する．Command，Args，Redirects，Background をそれぞれパースしていく．
   (a) Background の判定．コマンド列と仮定されたトークン列の最後尾に Ampersand が存在するかを確かめ，存在する場合には，Ampersand をトークン列から除外し，尚且つ Command に Background で処理するよう設定を行う．
   (b) Command のパース．String, Number, Ident Token であることが期待される．ライトパーサへ渡し，パースし Expr を得る．
   (c) Args のパース．Args は Optional である．Args が 0 以上の場合には，Arg が String，Number, Ident Token であることが期待される．それぞれを適切なライトパーサへと渡し，パースする．
   (d) Redirects のパース．コマンドには複数の Redirect を含めることができる．Redirects が 0 以上の場合には，Redirect を適切なライトパーサへ渡して，パースする．
6. Pipe と仮定する．トークン列を受け取り，トークン列が０または１，２以上かで処理を変える．トークン列に Pipe Token が存在し，尚且つ Pipe Token 以外の箇所をコマンド列と仮定するならば，このトークン列は Pipe Token を Pivot とした，２次元のコマンド列であると解釈する．
   (a) トークン列が０．エラーとする．
   (b) トークン列が１．尚且つ Pipe Token ならエラーとする．
   (c) トークン列が１．尚且つ Pipe Token 以外ならバッファに入れ，バッファを戻り値とする．
   (d) トークン列が２以上．Pipe Token を Pivot に，トークン列を再帰的に分割する．分割したトークン列をコマンド列であると仮定し，適切なライトパーサへと渡して，パースする．

3.2 パーサ

入力を受け取り，字句解析器へ渡す．トークナイズを行い，トークン列を得る．得たトークン列の最後尾が EOF Token かを確認する．EOF Token ならば，EOF Token を除外する．そうでなければ，エラーとする．
Semicolon Token を Pivot に，トークン列を再帰的に分割する．このトークン列は，Semicolon を Pivot とした 2 次元のトークン列であると解釈する．
entries = [[...], ...] となっている．
各トークン列を，適切なライトパーサでパースしていく．パースに成功すると，Pipe，Assign，Command のいずれかを得ることができる．それらをバッファに入れていく．バッファは，Semicolon を意味する AST の Node である．そのバッファをルートとする．
パースに成功した結果，AST を返す．

4 内部状態

FlatShell では，fsh-engine というライブラリ（crate）で処理系が定義され実装されている．
また, その中で State という構造体があり, State が内部状態を司ることになる．State は, Process handler, Current directory，Pipe のみを内包している．State とは別に，ShVars という Shell 変数が定義され，実装されている．
fsh-engine には eval という関数があり，eval 関数は，AST，State，ShVars を入力として受け取る．AST は eval 関数の中で処理され，適切な処理系に渡される．State は状態を保持する．eval 関数が終了しても，State の状態は保たれている．そのため，eval 関数が予期しないエラーを引き起こしたとしても，eval 関数とは別の場所で State にイニシャライズ等をかけることができる．ShVars に関しても同様である．

5 Unix shell との違い

5.1 Whitespace の考え方

FlatShell では，可能な限り Whitespace を認めることとしている．また，それが良いと考えている．
なぜ，Whitespace を認めた自由形式である方が良いと考えるのか？ 最も，Bug を減らすことができるからである．Whitespace を認めていない場合において，$A=Hello としたつもりが，$A = Hello と Typo されていたとき，多くの Unix shell ではエラーとして扱われる．
しかし，処理系などの実装ミスによっては，$A= Hello をコマンド等と解釈されてしまう可能性も否定できない．
構文を緩くすることで，処理系などの実装を容易にし，実装ミスを減らすことが可能であると考えている．Whitespace は，その 1 つである．

5.2 Unix shell における Whitespace

なぜ，Unix shell では Whitespace を認めていない箇所が多いのだろうか？ または，変則的ともいえるような構文となっているのだろうか？
主に 2 つの理由から，Whitespace を認めていない思われる．

1. Whitespace を Skip する動作を必要とする．その動作が入る可能性があり，Whitespace を認めていない場合とでは，少なからずリソース消費が多くなる．
2. 字句及び構文解析を容易にするため．例えば，A=Hello という入力を受け取ったとき，その時点では A が何を意味する文字（または文字列）なのかを理解していない．A の次をリードし，それが Equal ならば，A を変数名だと直ぐに確定させることができる．

5.3 Unix shell における変数の代入式

代入式は，<Variable name><Equal><Value>となっている．
具体的には， A=Hello と書く．
ここには，Whitespace を含めてはいけないという決まりがある．

5.4 Unix shell における変数の参照

参照は，<Dollar><Variable name>となっている．
具体的には，$A や $PATH と書く．
勿論，Dollar と変数名の間に Whitespace を入れることは認められない．これには同意である．
Whitespace を認めてしまうと，<Dollar><Whitespace><Variable name>となれば，Dollar がコマンド名なのか，変数名を意味する Dollar なのか，または変数名なのかを判定するのが難しくなる．困難とまでは言わないが，無理に容認すれば思わぬ Bug につながる可能性がある．場合によっては，セキュリティ上のリスクとなるかもしれない．

6 コンピュータリソースの今と昔

5.1，5.2 で触れた通り，Unix shell では Whitespace を認めていない箇所が多々ある．
C 言語の標準 API が定義された IEEE Std 1003.1-1988，Shell の仕様が追加された POSIX.2（つまり 1992 年）と 2024 年の今では，コンピュータリソースに大きな差がある．
現代のコンピュータリソースを持ってして，Whitespace の Skip に費やされるリソースを惜む必要はないだろう．特に，オプティマイズすることを前提とするならば，尚更である．

7 今後について

追加予定：分岐構造，反復構造，Shell 変数とは独立した変数（Shell script で使用するため）．

References

1. https://github.com/flucium/flatshell