虎本輪講資料 (2 章)

鵜飼昌樹

2007 年 7 月 5 日

Chapter 2 : Lexical Analysis

日本語に直訳すれば「語彙解析」あたりか。synthesis はその対になる言葉で「合成」。プログラムを翻訳するコンパイラは、まずプログラムを分解して、その構造と意味を理解する必要がある。

- Lexical analysis (語彙解析): 入力を「トークン (字句)」に分解
- Syntax analysis (構文解析): プログラムの句構造のパース (分析)
- Semantic analysis (意味解析): プログラムの意味の計算

これだけじゃさっぱりわかりませんが読み進めればわかるんでしょう。構文解析は 3 章、意味解析は 5 章。以下めんどくさいので lexical analizer を lexer (レキサ) と呼ぶ。

2.1 Lexical Tokens

token(トークン) というのは、「代用貨幣」とかいう意味でゲームセンターで使うメダルとかそんなものを指す語のようですが (よくわからん)、コンパイラの世界の文脈では、

- プログラミング言語の文法の構成単位となる文字列

だそうで。プログラミング言語ではトークンはいくつかの決まった種類に分類される。

IF や **RETURN** の類 (Punctuation Token : 句読点トークン? [1]) を**予約語**といい、識別子 (例で言うところの **ID** タイプ) にはつかえない。

コメントやプリプロセッサ処理される類のもの (C のマクロとか) [2]、空白類はトークンでは無い[3]。

[1] この Punctuation という語は C や Perl だと納得がいかない。ML では `fun` や `val` のような予約語において、直前にセミコロン—これは直感的な句点 (。) だろう—があると見なした振る舞いをするのでこう呼んでいるのではないかと思ったりした。真意は不明だけど。

[2] 語彙解析はプリプロセッサ処理後に行われるため。

[3] Whitespace という言語においては空白類はトークンだけどな。

2.2 Regular Expressions (正規表現[4])

Symbol(記号) その文字自身

Alternation(択一) 正規表現 `a|b` が表す文字列集合は “a” と “b” 。

Concatenation(結合) 正規表現 `a·b` が表す文字列集合は “ab” 。

Epsilon(ϵ, 空文字列) 正規表現 `a|`ϵ が表す文字列集合は “a” と “” (空文字列) 。

Repetition(繰り返し) 専門用語でクリーネ閉包 (Kleene closure)。M^* と書くと正規表現 M の 0 回以上の繰り返しである。すなわち $\{\ \varepsilon, M, M・M, M・M・M, ...\}$。

短縮表記 (`[]` や`+`など) を付け加えても良い (あってもなくても表現力は同じである)。また ϵ や結合子 (·) は[5] 省略して書かれる。

Fig 2.2 で出てくる正規表現の (ASCII 記述可能な) 形式は、ML-Lex [6] が解釈できる表記法なので、Perl 等の正規表現形式とは異なる部分がある。

Fig 2.2 の 5 行目はコメントと空白の場合であるが、これらの場合にはパーサには報告せずに継続する。が、一般的なコンパイラの場合、そんなに単純じゃない[7]。

(期待しない入力まで含めた場合に?) 語彙設計は**完全**である必要がある (これは DFM であるために必要な条件と思われる)。ここでは初期状態から任意の入力に対して何らかの終端にたどり着くことを言っている。

Fig 2.2 のルールはそれだけでは少しあいまいである。この曖昧性を排除するために、Lex ツールは大抵以下の 2 つのルールを適用する。

最長一致則 入力先頭からいずれかのルールにマッチする限りもっとも長い部分列になるようにトークンを取る。したがって `if8` は最長マッチ則により単一のトークン `ID(if8)` と解釈される (`if` より長い)。

優先順位則 前項に従った場合でも同一の長さのトークンが複数のルールに一致することはある。この場合は優先順位に従う。優先順位はルールの定義順。したがって `if` は `IF` と `ID` ルールに一致するが、優先順位則により `IF` となる。

2.3 Finite Automata[8](有限オートマトン)

トークンの区別を付けるために正規表現というのは使い勝手がいいんだが、プログラムに落ちてくれなければ絵に描いた餅なので、有限オートマトンを導入する。

この節では特に決定性有限オートマトン (DFA) を取り扱っている。DFA は入力を 1 個受け取ると必ず次の状態が一意に定まる有限オートマトン。DFA はステートマシンに書き下しやすい。

---

[4]正則表現とも言うらしい

[5]どうせ ASCII 文字集合では表現できないので?

[6]`http://www.smlnj.org/doc/ML-Lex/manual.html`

[7]たとえば flveri の場合、改行の数を数えて行番号を記録する等、パーサにとって不要なものといってもやることはある。って PROGRAM 節でやる Tiger 言語の解析サンプルには行番号記録が含まれている。

[8]automata は automaton の複数形

Fig 2.3 では 6 つの独立したオートマトンがあったが、これを結合して Fig 2.4 にする一般論は次の節で。とりあえず ad-hoc にやったとする。このとき、Fig 2.4 の状態機械は遷移行列 (二次元配列) にエンコードできる。遷移規則に無い文字が来たら State 0 に飛ばしている。この他に「終端」配列—たとえば State 2 なら `ID` に関連づける関数など—対応表が必要。

Recognizing the Longest Match (最長一致の認識)

文字列の受理 (or 拒否) は前述の通り (DFA が出来ていれば) 簡単である。lexer は最長一致則に従う必要がある。それを実現するためには二つの変数を導入する。

Last-Final: 最近の終端の (=トークンを切った) 状態番号

Input-Position-at-Last-Final: 入力位置

変数 Last-Final は、状態機械の終端 (となり得る位置) ごとに更新される。Input-Position は確定するたびに更新されるようだ。Fig 2.5 の図で | が Input-Position-at-Last-Final であり、`T` が Last-Final である。これと現在位置の 3 変数により、与えられた DFA から最長マッチ則を含む lexer を実装可能となる。

2.4 Nondeterministic Finite Automata (非決定性有限オートマトン (NFA))

「非決定」とは同一の入力 (あるいは入力の有無) から異なる状態に遷移可能であることを指す。絵の例では入力 `a` により左にいくか右にいくかはそれだけでは決定できない (何文字か先までいけばわかるかもしれない)。p23 の例だと入力無しで左右どちらにいくか、これも同様にそれだけでは決定できない。推測に推測を重ねて NFA を動作させるのは厳しいものがあるので、lexer は以下の作戦を採用する。

- Lexer のルールはまず形式的に NFA に変換できる (Fig 2.6)。
- NFA から DFA に変換できる。
- ゆえに Lexer のルールは DFA で書けてコード化できる

と説明している。結局難しいのは NFA - DFA 変換。

ただしこんな難しいことを知らなくても、lexer がある限りコンパイラは作れる。はず。

NFA-DFA 変換

p.25 の説明は、遷移可能な全ての状態を調べているのに相当するのだが、これを定式化すると NFA-DFA 変換になる。初期状態を例に取ると、NFA における開始点 1 と ϵ による遷移 4,9,14 の合成状態 1,4,9,14 が DFA での初期状態になる。

1. 形式的に ϵ-閉包[9]を定義する。

[9] ここでの closure は数学用語の閉包。プログラミング言語のクロージャではない。

2. 状態 s からラベル (入力) c によって到達可能な NFA 状態の集合を edge(s, c) とする[10]。たとえば edge$(1, \epsilon)$ は 1→4,1→9,1→14 だから 4,9,14 でよいのでしょうか。

3. ある状態群の集合を S としたとき、閉包 closure(S) とは S 内の状態から入力無しで到達できる状態の集合である。言い替えると矢印 ϵ だけをを辿って到達できる状態集合。

定式化すると、closure(S) は

$$T = S \cup (\bigcup_{s \in T} \mathbf{edge}(s, \epsilon))$$

を満たすもののなかで最小の集合に一致する[11]。この T の求めかたの pseudo code はこの式からは直感的に導けない (が等価なんだと思うよ)。難しいことを言っているが図 1 を簡約するための記述に過ぎない。

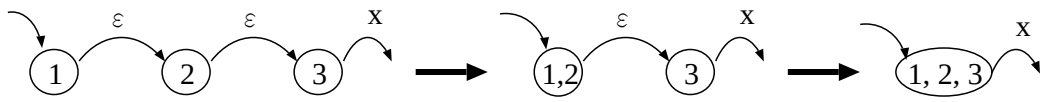

図 1: 遷移 ϵ が複数またがる場合の簡約

次に、DFAedge(d,c) を定義する。いくつかの NFA 状態名を要素に持つような集合 d をおいて、その状態集合からある入力 c によって遷移することを考える。このとき、

$$\mathbf{DFAedge}(d, c) = \mathbf{closure}(\bigcup_{s \in \mathbf{closure}(d)} \mathbf{edge}(s, c))$$

と状態集合を定義[12]すると[13]、NFA オートマトンの動作を次のアルゴリズムで書ける。

```
d := closure( {s1} );   // 初期状態
while( c := input_character ){
  d := DFAedge( d, c );
}
```

DFA への書き換えも同じことで、p.27 のアルゴリズムで書ける[14]。図 2 に示す[15]。

アルゴリズムを日本語で書くと以下の手順を踏むことになる。まず closure 変換を適用することで、ϵ の遷移をなくす。この時点ではある入力から二つ以上の状態遷移の可能性が残るので DFA ではない (例えば初期状態から入力 i があった場合には 3 通りの遷移があり得る)。したがって次に DFAedge を用いて変換を行う。

[10] 名前が奇妙だが、左辺は edge ではなく node。

[11] この式はとてもトリッキーだ。和集合の範囲式は $s \in S$ ではない!。T からの edge に ϵ が含まれていると左辺と右辺は一致せず式を満たさない事実と、明らかに $T \supseteq S$ であることが肝、かな。

[12] 教科書の記述を微修正した。教科書はおそらく d が closure であることを暗黙のうちに仮定している。

[13] これも同様に名前が奇妙だが、左辺は edge ではなく node。

[14] Σ は alphabet と言っているが a-z ではなく文字集合の意味。

[15] Fig 2.8 の DFA は誤りがある。著者のページに Errata が掲示されている通り、初期状態の次に遷移するのは {5,6,8,15} であり {10,11,13,15}

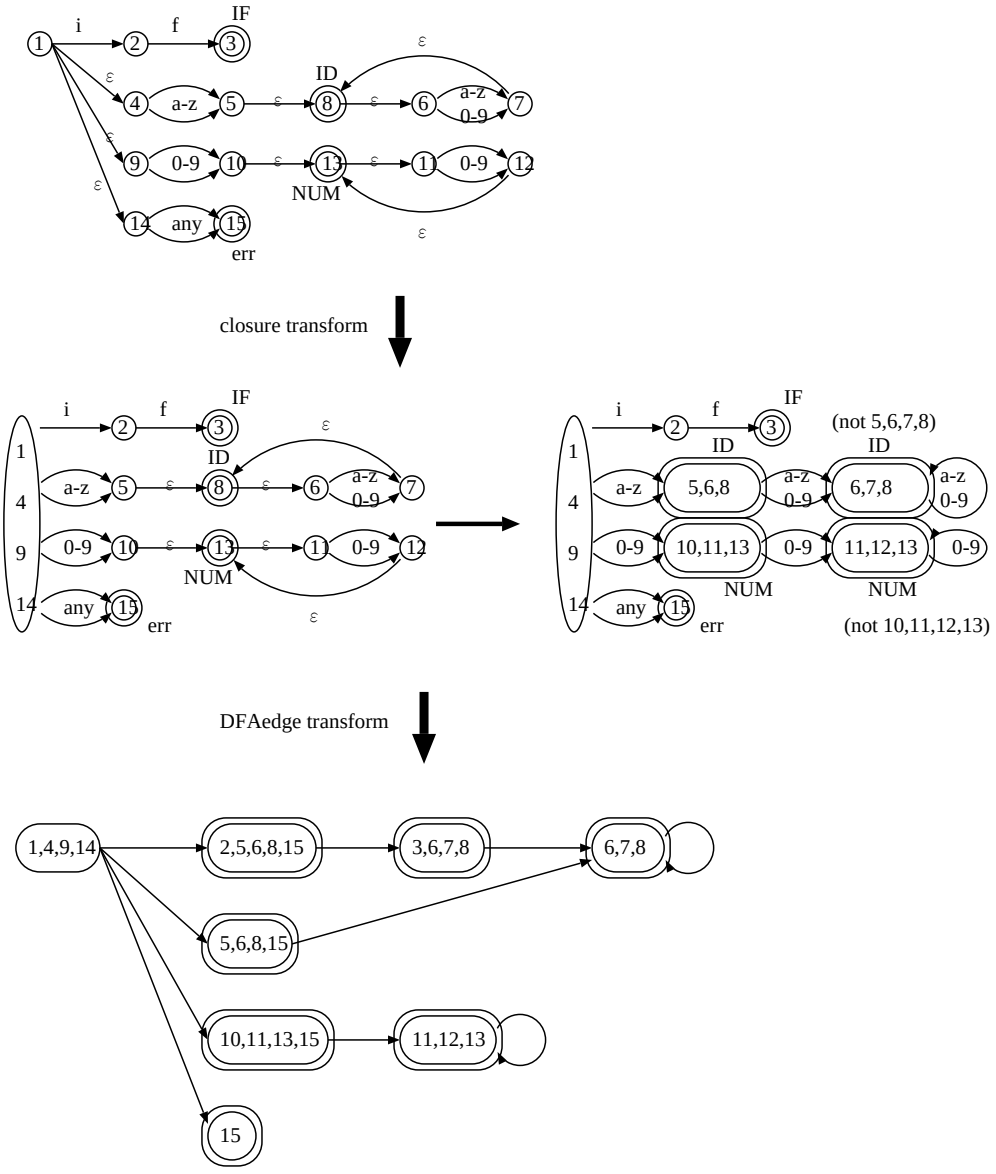

図 2: Fig 2.7 から Fig 2.8 への NFA-DFA 変換

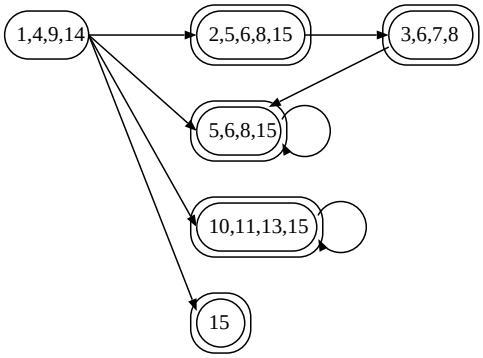

図 3: Fig 2.8 の簡約

$S1 = \mathbf{closure}(1) = \{1, 4, 9, 14\}$ に対し、DFAedge(S1,$e \in \Sigma$) は

$$S2 = \mathbf{DFAedge}(S1, e \in \{0, ..., 9\}) = \{10, 11, 13, 15\} \quad (1)$$
$$S3 = \mathbf{DFAedge}(S1, e \in \{a, ..., h, j, ..., z\}) = \{5, 6, 8, 15\} \quad (2)$$
$$S4 = \mathbf{DFAedge}(S1, e \in \{i\}) = \{2, 5, 6, 8, 15\} \quad (3)$$
$$S5 = \mathbf{DFAedge}(S1, e \notin \{0, ..., 9, a, ..., z\}) = \{15\} \quad (4)$$

同様に導かれた状態集合から生成を繰り返す。

$$S6 = \mathbf{DFAedge}(S2, e \in \{0, ..., 9\}) = \{11, 12, 13\} \quad (5)$$
$$S7 = \mathbf{DFAedge}(S3, e \in \{0, ..., 9, a, ..., z\}) = \{6, 7, 8\} \quad (6)$$
$$S8 = \mathbf{DFAedge}(S4, e \in \{f\}) = \{3, 6, 7, 8\} \quad (7)$$
$$S9(= S6) = \mathbf{DFAedge}(S4, e \in \{a, ..., e, g, ..., z, 0, ..., 9\}) = \{6, 7, 8\} \quad (8)$$
$$S10(= S6) = \mathbf{DFAedge}(S9, e \in \{0, ..., 9, a, ..., z\}) = \{6, 7, 8\} \quad (9)$$
$$S11(= S6) = \mathbf{DFAedge}(S3, e \in \{0, ..., 9, a, ..., z\}) = \{6, 7, 8\} \quad (10)$$

ただし、たとえば S2 へ入力 a があったときは架空の状態 S0 に遷移するようにしておき、エラー制御に利用できるようにする。

新たな状態集合を作らなくなったら収束であり、DFA への変換が完了する。便宜的に S9, S10, S11 を上では書いたが、実際には S6 と一致するのでアルゴリズム上は生成されない。なお、p.27 のアルゴリズムでは矢印の生成のため `tran` をメンテナンスしている。

この NFA-DFA 変換による DFA 状態木は、理論的には指数オーダの状態を作り得る (単純に n 状態の NFA からの状態集合の組み合わせ総数が 2^n だから) が、到達しえない状態がほとんどなので、実用的には問題ない規模に収まるような変換になる。

ここからさらに DFA の規模を縮小するために、等価な状態を見つけて括ることが考えられるが、それは Ex.2.6 にて。Fig 2.8 も図 3 のように簡約可能であるはず。

ところで、Closure の定義を少し変更するとこういうこと (図 4) が出来るのだがダメなのだろうか? 終端かどうかの類で問題がおきそうな気はするな。

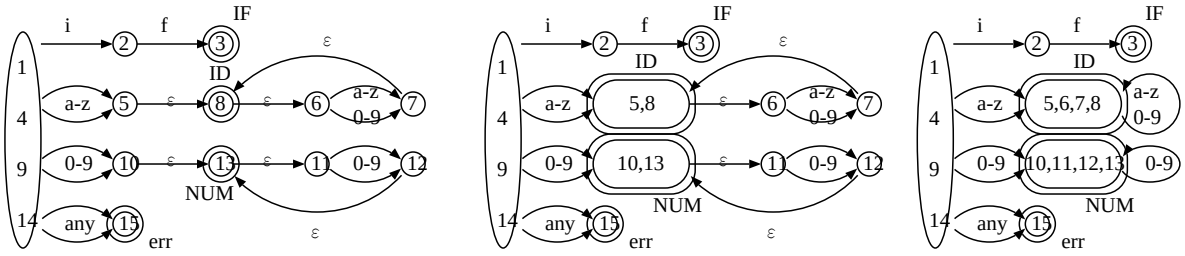

図 4: Fig 2.7 から Fig 2.8 への closure 変換もどき

2.5 ML-Lex:A Lexicai Analyzer Generator

ここでは ML-Lex という、SML(/NJ) コードを吐く Lexer を使う。開発言語が C であれば lex (GNU-flex 等) を使うことになるだろう。

一般的な lexer と同じように ML-Lex でも `%%` で区切られる 3 パートにわけて記述する構文になっている。が、それぞれのパートに書くべき内容は C の lex とは異なっている。

C の lex の場合、最初が定義部、次がルール部で、3 つ目はオプションとしてサブルーチン部を置ける。ML-Lex では、最初がユーザー宣言部、次が ML-Lex 定義部であり、これら二つをあわせたのが C の lex での定義部に当たる。3 つ目がルール部になる。サブルーチン部は無く、ルール部の右辺に書くコードは必要ならばユーザー宣言部に書くことになるようだ。

`lexresult` は `type` もしくは `datatype` による定義が必須。関数 `eof` も必要で、ファイル終端まできたときの動作を指定する必要がある。

`Token` 構造体は 3 章で出て来る parser (ML-yacc) で定義 (より正確にはパーサジェネレータで生成) されるもので、現段階では詳細はわからず天下り的である。`yytext` や `yypos` をつかってセマンティック値を渡す形式を天下り的に理解しておくしかない。

yacc / lex を使った経験があるなら大した問題はないと思うのだが、とりあえずマニュアル (`http://www.smlnj.org/doc/ML-Lex/manual.html`) 見てね、と。

後半の START STATES は、たとえば通常モードとコメント内とで文法解釈が異なるような場合を書いている。lex の解釈では「改行」だけは特別扱い[16]だから、大抵は複数行コメントだけ例のように別扱いにする。この場合、ML-Lex 定義部に `%s` で始まる状態定義が必要[17]。実際に Tiger 言語の lexer を書く場合には必須になる。複数宣言する場合には

```
%s HOGE FUGA;
```

とする。

[16] . は改行以外の 1 文字にマッチ、だ。

[17] `INITIAL` 状態はあらかじめ定義されているので不要。

PROGRAM: ML-Lex で Tiger 言語

PROGRAM 節ではもう Tiger 言語の lexer を記述させられる。Appendix.A を全部読まないと書けないとかいう罠に見える。ヒントはかなり書かれてるのと、著者のページから雛型を拾ってこれるのとで、概ね問題はなさそうだが。

ちなみに 3 章ではこの成果を元に parser を作る。

俺言語を作りたい人は言語仕様から提示してねー。

雛型に関する注意事項

`sources.cm` を `ml-build` に渡すことで build できるはずなのだが、出版当時と現在とでバージョンが違うからか、そのままでは通らない。書き換え後の `sources.cm` の中身は以下の通り。

```
Group is

driver.sml
errormsg.sml
tokens.sig
tokens.sml
tiger.lex
$/basis.cm
$/smlnj-lib.cm
```

しかし、`ml-build` にかけると手元ではどうがんばってもエラーが取れないので、インタラクティブ環境で

```
% sml
- CM.make "sources.cm";
```

としたら通った (確かにテキストにはそうしろと書いてある)。この場合スタンドアロンプログラムは生成されない。そもそもサンプルはそういうものとして提供されているようだ。スタンドアロンで動くようにするにはいくつかおまじないの必要があるようだけど面倒なのでパス。そのうち何か出てくるだろう。

結局 `tiger.lex` が完成したら以下の手順で確認できる。

```
% sml sources.cm
- Parse.parse "../testcases/test1.tig";
```

さて、Appendix.A の Tiger 言語仕様を満たすための注意点はなんだろう? [18]

[18] この課題のおかげで某コンバータのバグが見つかった!!