ML-Yacc の使いかた

東北大学 電気通信研究所 篠埜 功

平成 18 年 11 月 28 日

1 はじめに

この資料では、ML-Yacc を使って、ML で書かれた構文解析器 (syntax analyser あるいは parser) を作成する方法を説明する。構文解析器とは構文解析 (syntax analysis あるいは parsing) を行うプログラムである。構文解析とは、字句列 (token stream) を受けとって、それが、構文 (syntax) に従った字句列になっている場合、その木構造を結果として返し、そうでない場合には構文エラー (syntax error) を出力するものである。構文は、生成規則の集合によって表され、それを文法 (grammar) と呼ぶ。文法が、あるクラスに属している場合、文法から機械的に構文解析器を作成することができる。これを行うものを構文解析器生成系 (parser generator) といい、通常、プログラミング言語処理系の作成時には手で一から書くことはせず、構文解析器生成系を用いる。

構文解析器生成系として、Yacc が有名であり、これは、LALR という文法クラスを対象とするものであり、C で書かれた構文解析器を生成する。この講義では、ML を用いてプログラミング言語処理系を記述するため、ML で書かれた構文解析器を生成する、ML-Yacc という、Yacc の ML 版を使用する。ML-Yacc は、Standard ML of New Jersey(SML/NJ と略記) という、Standard ML の compiler に付随している。また、字句解析器生成系としては lex の ML 版である ML-Lex を用いる。

この資料は ML-Yacc のマニュアル [1] および ML-Lex のマニュアル [2] を参考に作成されており、詳しくはそちらを参照されたい。

2 ML-Yacc における文法の記法

ML-Yacc を用いる場合、lambda.grm のような、.grm という suffix のついたファイルを作成する。このファイルは、次のように、%%で区切られた、３つの部分からなる。

SML のコード (各規則に付随する意味記述に用いる)
%%
規則に関する種々の宣言
%%
文法規則

最初の SML コードの部分は各規則に対応する意味記述に用いる関数の定義部であり、Yacc においては C プログラムを記述するところである。Yacc は属性文法 (attribute grammar) に基づいており、意味は属性文法における合成属性 (synthesized attribute) である。

２番目の宣言の部分では、

%name parser の名前
%term 終端記号 (terminal symbol) の列
%nonterm 非終端記号 (nonterminal symbol) の列
%pos 位置情報の型

という宣言が含まれている必要がある。

%term のところで terminal symbol を定義するが、これに対応して、生成される parser (lambda.grm.sml) において、`Tokens` という structure が定義される。その structure 中で、例えば、

```
fun ID (i,p1,p2) = Token.TOKEN (ParserData.LrTable.T 0,
                                (ParserData.MlyValue.ID (fn () => i),p1,p2))
```

のように、terminal symbol 1つにつき1つの関数が生成される。ここでは、`i` は terminal symbol の属性 (字句に付随する情報) であり、`p1`, `p2` は、source program における、字句の始まりと終わりの位置情報である。この関数を用いて、構文木の葉の部分が生成される。

structure `Tokens` に対する signature は、lambda.grm.sig に生成される。signature 名は、.grm ファイルにおいて指定された名前を Lambda とすると、Lambda_TOKENS である。

必須ではない宣言として、%verbose があるが、これを宣言部に入れておくと、lambda.grm.desc というファイルが生成され、状態遷移表などの情報が得られる。

3 例

ここでは、例として、ラムダ式を入力とし、字句解析、構文解析して構文木を作成し、それを単に再帰的にたどって表示するプログラムを書いてみる。

まず、lambda.grm を図 1 のように記述する。構文解析の結果として返す構文木の型として、Exp データ型を定義しておく。

Exp 型の値は、構文解析後、次のフェーズへ渡すため、別の sml のファイルの中で、structure を定義し、その中で定義する。例えば、exp.sml というファイルを作成し、図 2 のように、`Exp` structure を作成する。この structure では、`Exp` から `string` への変換関数 `exp2string` も定義してある。

図 1 において、Exp.Var, Exp.Lam 等は、Exp structure 中の、`Var`, `Lam` である。mlyacc の起動時

```
> mlyacc lambda.grm
```

には `Exp` structure の定義は不要だが、mlyacc の生成した lambda.sml ファイルのコンパイル時に、Exp の定義が必要となる。コンパイル方法は後で述べる。

構文解析の結果、属性値として `Exp` 型の値が得られるが、これをコンパイルの次の処理へ渡せばよい。この例では、`Exp` 型の値を string に変換する関数 `exp2string` を用いて、print して終了することを想定している。

4 構文解析器の生成

この節では、ML-Lex, ML-Yacc を用いて構文解析器を生成する方法を説明する。ML-Lex の仕様ファイルを lambda.lex, ML-Yacc の仕様ファイルを lambda.grm とする。

4.1 Parser 用の Functor の生成

ML-Yacc により自動生成されるのは、`LambdaLrValsFun`(`Lambda` の部分は%name の指定によって変わる) という Functor であり、これを用いて構文解析器を定義する。

この Functor は、

```
functor LambdaLrValsFun(structure Token : TOKEN)
 : sig structure ParserData : PARSER_DATA
       structure Tokens : Lambda_TOKENS
   end
```

```
%%

%eop EOF SEMI
%pos int

%term ID of string | LAMBDA | DOT | EOF | SEMI
%nonterm EXP of Exp.Exp | START of Exp.Exp

%name Lambda

%noshift EOF
%nodefault
%%

START : EXP (EXP)
EXP : ID (Exp.Var ID)
    | LAMBDA ID DOT EXP (Exp.Lam (ID, EXP))
    | EXP EXP (Exp.App (EXP1,EXP2))
```

図 1: lambda.grm

```
structure Exp =
struct

type Id = string
datatype Exp = Lam of Id * Exp
             | Var of Id
             | App of Exp * Exp

fun exp2string (Lam (x,e)) =
       "Lam " ^ x ^ ". " ^ exp2string e
  | exp2string (Var x) =
       "Var " ^ x
  | exp2string (App (e1, e2)) =
       "App (" ^ exp2string e1 ^ ", " ^ exp2string e2 ^ ")"

end
```

図 2: exp.sml

という signature をもっており、TOKEN という、/usr/share/smlnj/src/ml-yacc/lib/base.sig (/usr/share/... 等の path は SML/NJ の install 時の指定に依存する。以下同様) で定義されている signature を持つ structure を引数としてとり、ParserData : PARSER_DATA, Tokens : Lambda_TOKENS からなる structure を返す。

引数としては、LrParser (/usr/share/smlnj/src/ml-yacc/lib/parser2.sml に定義されている) という structure の中に定義されている LrParser.Token : TOKEN という structure を与え、Parser 用 structure を得る。

```
LambdaLrValsFun(structure Token = LrParser.Token)
```

LrParser.Token においては、構文解析テーブル用 structure、字句用データ型、字句同士の等しさの判定関数が定義されている。

字句用データ型では、TOKEN という構成子が定義されており、各字句データ生成関数の定義に用いられる。例えば、さきほど挙げた例においては、

```
fun ID (i,p1,p2) = Token.TOKEN (ParserData.LrTable.T 0,
                                (ParserData.MlyValue.ID (fn () => i),p1,p2))
```

のように使われている。

4.2 ML-Lex について

ML-Lex において、規則は、

```
<状態リスト> 正規表現 => (ML コード)
```

の形の列で記述する。例えば、

```
\n => (pos := (!pos) + 1; lex());
{ws}+ => (lex());
";" => (Tokens.SEMI(!pos,!pos));
{alpha}+ => (Tokens.ID(yytext,!pos,!pos));
"." => (Tokens.DOT(!pos,!pos));
"\\" => (Tokens.LAMBDA(!pos,!pos));
```

のように記述する。

```
%s A,B,C
```

等で状態を定義でき、動作記述部分に

```
YYBEGIN A;
```

等と書くことにより状態遷移を行うことができる。

ML コードの中では、yytext を用いることができ、これは字句に対応する文字列がその値となっている。また、lex() という関数呼び出しを用いることができる。lex という関数は字句を一つ認識する関数であり、ML-Lex によって生成される字句解析器において定義される関数である。.lex の各規則記述中の ML コードはこの lex という関数の一部分になるので、lex() が規則のコード中にある場合、lex() を再帰呼び出しすることになり、読み込んだ文字列を捨てることになる。これは、Yacc において、規則のコードに return 文がない場合に相当している。

なお、.lex ファイルにおいて%arg が宣言されている場合、lex() ではなく、continue() という関数を代わりに用いる。

ML-Lex により、makeLexer という関数が.lex.sml ファイルの中の、LambdaLexFun という Functor 定義の中に定義される。この関数は

```
val makeLexer : (int -> string) -> yyarg -> lexresult
```

という型を持っており、`int -> string` 型の関数を引数にとって字句解析器を返す。例えば、

```
makeLexer (fn _ => TextIO.inputLine TextIO.stdIn)
```

は、ユーザからの入力に対して字句解析を行う。int 型の引数は、何文字読み込むかを指定するために設けられているが、ここでは使われていない。

4.3 Lexer 用の Functor の生成

ML-Lex により自動生成されるのは、`LambdaLexFun`(`Lambda` の部分は%name の指定によって変わる) という Functor である。引数に、Parser 用の Functor から生成した `LambdaLrVals` という structure 中の、`Tokens` という structure を引数に与えることにより、字句解析器の structure が作成される。

```
LambdaLexFun(structure Tokens = LambdaLrVals.Tokens)
```

`LambdaLrVals.Tokens` において、各字句に対応するデータを生成する関数が定義されており、それらを ML-Lex の規則記述に用いる。さきほどの ML-Lex の例だと、

```
{alpha}+ => (Tokens.ID(yytext,!pos,!pos));
```

のように使われている。

4.4 Parser 作成

これまでに作成した、Lexer 用の Functor と、Parser 用の Functor を用いて、Parser を作成する。まず、それぞれの Functor に以下のように Token の structure を与え、structure を作成する。

```
structure LambdaLrVals =
LambdaLrValsFun(structure Token = LrParser.Token)

structure LambdaLex =
LambdaLexFun(structure Tokens = LambdaLrVals.Tokens)
```

これらと、Parser の本体である structure `LrParser` を組合せて parser を作成することになるが、これらの中の、型 pos や、字句の型などが一致している必要があるため、functor の sharing という機構を用いて、一つの structure にまとめる。これを行う Functor として、`Join` と `JoinWithArg` が/usr/share/smlnj/src/ml-yacc/lib/join.sml (/usr/share/... 等の path は SML/NJ の install 時の指定に依存する。) に定義されている。`Join` は.lex において %arg の宣言をしなかった場合、`JoinWithArg` は%arg の宣言をした場合に用いる。%arg の宣言をした場合、関数 makeLexer の第二引数に、%arg で指定した引数を与えることになる。例えば、解析対象の文字列をファイルから得る場合などに、file 名を引数として与える場合などに使われる。%arg なしの場合は、

```
structure LambdaParser =
  Join(structure LrParser = LrParser
       structure ParserData = LambdaLrVals.ParserData
       structure Lex = LambdaLex)
```

のようにし、%arg ありの場合は、

```
structure LambdaParser =
  JoinWithArg(structure LrParser = LrParser
              structure ParserData = LambdaLrVals.ParserData
              structure Lex = LambdaLex)
```

のようにすればよい。LambdaLex.makeLexer は、一つの字句を返す lexer を作成するものだが、Join Functor の中では、LambdaLex.makeLexer が stream 化された makeLexer が新たに定義される。この stream 化された makeLexer を用いて、例えば

```
val tokenstream = LambdaParser.makeLexer
                  (fn _ => TextIO.inputLine TextIO.stdIn)
```

のようにして token の stream を作成する。この tokenstream は、標準入力からの入力を字句解析した結果の字句の stream が値となる。

parsing 関数の方は、生成された Functor 中の parser のテーブル、意味動作と、parser 本体 LrParser.parse を用いて、Join functor により LambdaParser.parse が定義される。この parse 関数は、構文エラーの修正のために先読を何文字まで許すかを表わす int 型の値、さきほど作成した字句 stream、エラー表示関数、%arg で指定された値の４つ組を引数としてとる。エラー表示関数は、表示するメッセージと、エラーの発生した個所の左端と右端を示す値の組を引数としてとる。４番目の引数は、%arg で指定されたものだが、%arg 宣言がない場合は () を与えればよい。例えば、

```
LambdaParser.parse(0,tokenstream,print_error,())
```

のように引数を与えると、parse 結果の属性値と、字句 stream の残りの対が結果として得られる。最初の引数が 0 の場合、構文エラーの修正は試みられない。ユーザとの対話的環境の場合、字句 stream の残りを次のループで処理するため、字句 stream の残りが必要になる。今回 parse された最後の字句が、字句 stream の残りの先頭になる。このため、次のループに入る前に字句を一つ読み込む必要があり、`LambdaParser.Stream.get` がこれを行う。

```
LambdaParser.Stream.get tokenstream
```

により、字句 stream の先頭と残りの対が得られ、先頭の字句で対話環境を終了するかどうかの判定などを行うことができる。

4.5 コンパイル方法

以上のような parsing 関数の定義を lambda.sml に記述することにすると、以下のようにしてコンパイルすることができる。

まず、sources.cm というファイルを lambda.lex, lambda.grm と同じ directory に作成し、

```
Group is
ml-yacc-lib.cm
lambda.grm
lambda.lex
exp.sml
lambda.sml
```

のように記述する。.cm ファイルは、SML/NJ の compilation manager の動作を記述するものである。SML/NJ の prompt において、

```
- CM.make();
```

のように入力することによって、compilation manager は、sources.cm の内容に従って compile を行う。または、

```
- CM.make' "sources.cm";
```

のようにして、読み込む.cm ファイルを指定する方法もある。Compilation manager は、まず、

```
> mlyacc lambda.grm
> mllex lambda.lex
```

を実行し、lambda.grm.sml, lambda.grm.sig, lambda.lex.sml を生成する。その後、これら、生成された.sml ファイルは、ml-yacc-lib.cm で指定された ML-Yacc library と exp.sml を読み込んでからコンパイルされる。

上記を compilation manager を用いずに行うとすれば、

```
> mlyacc lambda.grm
> mllex lambda.lex
```

を手動で実行したのち、

```
use "/usr/share/smlnj/src/ml-yacc/lib/base.sig";
use "/usr/share/smlnj/src/ml-yacc/lib/join.sml";
use "/usr/share/smlnj/src/ml-yacc/lib/lrtable.sml";
use "/usr/share/smlnj/src/ml-yacc/lib/stream.sml";
use "/usr/share/smlnj/src/ml-yacc/lib/parser2.sml";
use "exp.sml";
use "lambda.grm.sig";
use "lambda.grm.sml";
use "lambda.lex.sml";
use "lambda.sml";
```

のようにすればよい。(/usr/share/... 等の path は SML/NJ の install 時の指定に依存する。) Compilation manager を用いれば、一度 compile して変更がなかったファイルは以前の compile 結果が用いられるので効率がよい。

参考文献

[1] David R. Tarditi and Andrew W. Appel. ML-Yacc User's Manual Version 2.4. `http://www.smlnj.org/doc/ML-Yacc/index.html`, April 24, 2000.

[2] Andrew W. Appel, James S. Mattson, and David R. Tarditi. A lexical analyzer generator for Standard ML. Version 1.6.0. `http://www.smlnj.org/doc/ML-Lex/manual.html`, October 1994.