Eksempel på brug af MosML-lex og MosML-yacc

Torben Mogensen

September 2002

1 Introduktion

Som eksempel på brug af lexer- og parsergeneratorerne i Moscow ML, viser vi hvordan de kan bruges til at indlæse og beregne regneudtryk.

Vi starter med at definere grammatikken for sproget:

$$
\begin{array}{lcl}
Defs & \rightarrow & \\
Defs & \rightarrow & Def\,Defs \\
 & & \\
Def & \rightarrow & \textbf{id} = Exp \\
 & & \\
Exp & \rightarrow & \textbf{num} \\
Exp & \rightarrow & \textbf{float} \\
Exp & \rightarrow & \textbf{id} \\
Exp & \rightarrow & Exp + Exp \\
Exp & \rightarrow & Exp - Exp \\
Exp & \rightarrow & Exp * Exp \\
Exp & \rightarrow & Exp \,/\, Exp \\
Exp & \rightarrow & -\,Exp \\
Exp & \rightarrow & Exp = Exp \\
Exp & \rightarrow & Exp \texttt{ and } Exp \\
Exp & \rightarrow & Exp \texttt{ or } Exp \\
Exp & \rightarrow & \texttt{not } Exp \\
Exp & \rightarrow & \texttt{if } Exp \texttt{ then } Exp \texttt{ else } Exp \\
Exp & \rightarrow & (\ Exp\ )
\end{array}
$$

Her gælder de sædvanlige præcedensregler:

- `+`, `-`, `*` og `/` er venstreassociative.
- `*` og `/` binder stærkere end `+` og `-`.
- Unært minus binder stærkere end `*` og `/`.
- `=` er ikke-associativ og binder svagere end `+` og `-`.

- `not` binder svagere end `=`.
- `and` binder svagere end `not` og er højreassociativ.
- `or` binder svagere end `and` og er højreassociativ.

Endvidere strækker udtrykket efter `else` sig så langt som muligt. Det svarer til at `else` har lavere prioritet end alle regneoperatorer.

Heltalskonstanter og floating-point konstanter er som i SML, dog uden mulighed for at bruge hexadecimal notation for heltal. Specielt er negativt fortegn skrevet som "~".

Kommentarer starter med "\" og strækker sig til næste linieskift.

Semantikken for sproget er, at en række definitioner af variabler indlæses og beregnes og værdien af hver definition udskrives.

2 Abstrakt syntaks

Et "program" i ovenstående sprog vil indlæses som abstrakt syntaks, og beregningerne sker på denne syntaks.

Vi repræsenterer et program som en liste af definitioner, hvor hver definition er et par af et variabelnavn (en string) og et udtryk.

Udtryk repræsenteres med en datatype, der har en konstruktor for hver slags udtryk undtagen parentesudtryk. Hvert udtryk har udover deludtryk, samt attributter for navne og tal også en positionsangivelse, som består af linienummer og position på linien.

De relevante erklæringer samles i en struktur `Syntax.sml`:

```
structure Syntax =
struct

  type pos = int * int       (* position in program (line,column) *)

  datatype Exp = ICONST of int * pos
       | FCONST of real * pos
       | ID of string * pos
       | PLUS of Exp * Exp * pos
       | MINUS of Exp * Exp * pos
       | TIMES of Exp * Exp * pos
       | DIVIDE of Exp * Exp * pos
       | UMINUS of Exp * pos
       | EQ of Exp * Exp * pos
       | AND of Exp * Exp * pos
       | OR of Exp * Exp * pos
       | NOT of Exp * pos
       | IF of Exp * Exp * Exp * pos

  type Def = string * Exp

  type Pgm = Def list

end
```

Man oversætter denne fil med kommandoen

```
> mosmlc -c Syntax.sml
```

3 Parserdefinitionen

En parserdefinition i MosML-yacc består af flere sektioner, hvor nogle kan udelades. Se *Moscow ML Owner's Manual* for en komplet beskrivelse, her medtager vi blot dem vi skal bruge. Vi starter med at erklære de *tokens*, vi har brug for. Vi skal bruge en for hvert terminalsymbol i grammatikken samt for end-of-file (`EOF`). Hver token gives et navn og en type. Typen er for de fleste tokens vedkommende bare en positionsangivelse, men tokens for talkonstanter og variablenavne har også deres værdi med:

```
%token <int*(int*int)> NUM
%token <real*(int*int)> FLOAT
%token <string*(int*int)> ID
%token <(int*int)> IF THEN ELSE AND OR NOT EQ
%token <(int*int)> PLUS MINUS TIMES DIVIDE LPAR RPAR EOF
```

Herefter erklærer vi præcedens af regneoperatorer samt `else`:

```
%nonassoc ELSE
%right OR
%right AND
%nonassoc NOT
%nonassoc EQ
%left PLUS MINUS
%left TIMES DIVIDE
%nonassoc UMINUS
```

Læg mærke til erklæringen af `UMINUS`. Der er ikke nogen token med det navn, men vi skal bruge den til lokalt at ændre præcedens for `MINUS` i produktionen for unært minus, se senere. Foranstillede operatorer er her erklæret som ikke-associative. Det er egentligt ligegyldigt hvilken asssociativitet de gives, da associativitet kun giver mening for mellemstillede operatorer.

Efter præcedenserklæringerne, erklærer vi nonterminalerne. Vi starter med at angive startsymbolet og angiver derefter type for alle nonterminaler:

```
%start Defs
%type <Syntax.Pgm> Defs
%type <Syntax.Def> Def
%type <Syntax.Exp> Exp
```

Erklæringerne angiver typen af den abstrakte syntaks for hver nonterminal.

Til sidst, efterfulgt af "`%%`" kommer produktionerne i grammatikken. Den første produktion for hver nonterminal angives med nonterminalens navn, kolon, højresiden og til sidst en parser-aktion angivet mellem krølleparenteser. De efterfølgende produktioner for samme nonterminal angiver ikke nonterminalens navn, men bruger symbolet "`|`". Produktionerne for en nonterminal afsluttes med semikolon. Dette skulle fremgå tydeligt af nedenstående eksempel:

```
%%

Defs:
          EOF           { [] }
        | Def Defs      { $1 :: $2 }
;

Def:
          ID EQ Exp     { (#1 $1,$3) }
;

Exp:
          NUM           { Syntax.ICONST (#1 $1, #2 $1) }
        | FLOAT         { Syntax.FCONST (#1 $1, #2 $1) }
        | ID            { Syntax.ID (#1 $1, #2 $1) }
        | Exp PLUS Exp  { Syntax.PLUS ($1,$3, $2) }
        | Exp MINUS Exp { Syntax.MINUS ($1,$3, $2) }
        | Exp TIMES Exp { Syntax.TIMES ($1,$3, $2) }
        | Exp DIVIDE Exp
                        { Syntax.DIVIDE ($1,$3, $2) }
        | MINUS Exp %prec UMINUS
                        { Syntax.UMINUS ($2, $1) }
        | Exp EQ Exp    { Syntax.EQ ($1,$3, $2) }
        | Exp AND Exp   { Syntax.AND ($1,$3, $2) }
        | Exp OR Exp    { Syntax.PLUS ($1,$3, $2) }
        | NOT Exp       { Syntax.NOT ($2, $1) }
        | IF Exp THEN Exp ELSE Exp
                        { Syntax.IF ($2,$4,$6, $1) }
        | LPAR Exp RPAR { $2 }
;
```

Læg mærke til produktionen for foranstillet (unært) minus. Her er præcedens lokalt ændret til at være `UMINUS`'s præcedens ved hjælp af erklæringen `%prec UMINUS`. Læg også mærke til at vi sørger for at listen af definitioner, der udgør programmet, afsluttes med `EOF`.

Aktionen for en produktion (angivet mellem krølleparenteser) er et SML udtryk, der bygger værdien af produktionen, typisk den abstrakte syntaks. Symbolerne `$1`, `$2` osv. er en speciel slags variabler, der indeholder attributterne (værdierne) for hhv. det første, andet osv. symbol på højresiden.

Aktionerne for `Defs` bygger en liste af definitioner: Ved `EOF` returneres den tomme liste og ved den anden produktion sættes definitionen, som `Def` returnerer foran listen, som det rekursive kald til `Defs` returnerer.

I produktionen for `Def` har `ID` (som erklæret tidligere) en attribut, der er et par af et variabelnavn og en position. Med udtrykket `#1 $1` trækkes navnet ud af denne attribut. Navnet sættes i et par sammen med udtrykket, som findes som attribut for nonterminalen `Exp`. Da den er tredie symbol på højresiden, bruges `$3` til at hente denne attribut.

Produktionerne for `Exp` har aktioner, der bygger den abstrakte syntaks. Læg mærke til at vi sætter modulnavnet `Syntax` foran konstruktorerne. Den abstrakte syntaks har en positionsangivelse for hvert udtryk. Dette findes ved at tage positionsattributten for en af de tokens, der indgår på højresiden, typisk den første af disse. De fleste tokens har kun positionen som attribut, og denne hentes bare ved brug af den relevante dollarvariabel. Talkonstanter og variabelnavne har også deres værdi/navn som attribut, så de relevante dele hentes med operatorerne `#1` og `#2`.

Læg til sidst mærke til aktionen for udtryk i parentes. Den returnerer bare den abstrakte syntaks for det udtryk, der er i parentesen. Parentesen kan altså ikke ses i de abstrakte syntaks. Det er heller ikke nødvendigt, da formålet med parenteserne kun er at sørge for at den rigtige gruppering opnås under syntaksanalysen.

Hele parseerdefinitionen findes i filen `Parser.grm`. Parseren genereres med kaldet

```
> mosmlyac -v Parser.grm
```

Dette kald genererer filerne `Parser.sig`, `Parser.sml` og `Parser.output`. Hvis der er konflikter, vil `mosmlyac` angive antallet og typen af konflikter. I givet fald kan man bruge filen `Parser.output`, der indeholder en beskrivelse af den genererede parserautomat, inklusive hvilke NFA tilstande, der indgår i hver DFA tilstand (se "Basics of Compiler Design", afsnit 3.14 og 3.15.3). `mosmlyac` kan også rapportere andre slags fejl. Det vil i reglen klart fremgå af fejlmeddelelsen, hvad der er galt.

Parseren oversættes med kommandoerne

```
> mosmlc -c Parser.sig
> mosmlc -c Parser.sml
```

Oversættelsen af `Parser.sig` giver en *compliance warning*. Den kan ignoreres. Hvis der meldes syntaksfejl eller typefejl ved oversættelse af `Parser.sml`, ligger problemet som regel i parseraktionerne eller i erklæring af tokens eller nonterminaler.

Husk at `Syntax.sml` skal oversættes først. Lav evt. en `makefile`.

4 Lexerdefinitionen

Lexerdefinitionen er vist i figur 1. Vi gennemgår detaljerne herunder.

Udover at give parseren en sekvens af tokens, skal lexeren også angive positionen for hver af disse. MosML-Lex giver mulighed for at finde positionen af en token målt som antal tegn siden starten af inddata. Det er mere brugbart at have positionen som linienummer og position på denne linie, så vi definerer nogle hjælpefunktioner i lexeren til at holde styr på dette. Kort sagt har vi to

```
{
 open Lexing;

 val currentLine = ref 1
 val lineStartPos = ref [0]

 fun getPos lexbuf = getLineCol (getLexemeStart lexbuf)
(!currentLine)
(!lineStartPos)

 and getLineCol pos line (p1::ps) =
       if pos>=p1 then (line, pos-p1)
       else getLineCol pos (line-1) ps

 exception LexicalError of string * (int * int) (* (message, (line, column)) *)

 fun lexerError lexbuf s =
     raise LexicalError (s, getPos lexbuf)

 fun keyword (s, pos) =
     case s of
         "if"          => Parser.IF pos
       | "then"        => Parser.THEN pos
       | "else"        => Parser.ELSE pos
       | "and"         => Parser.AND pos
       | "or"          => Parser.OR pos
       | "not"         => Parser.NOT pos
       | _             => Parser.ID (s,pos);

 }

rule Token = parse
    [` ` `\t` `\r`]     { Token lexbuf } (* whitespace *)
  | [`\n` `\012`]       { currentLine := !currentLine+1;
                          lineStartPos :=  getLexemeStart lexbuf
                    :: !lineStartPos;
                          Token lexbuf } (* newlines *)
  | "\\" [^ `\n`]*      { Token lexbuf } (* comment *)
  | [`~`]?[`0`-`9`]+ { case Int.fromString (getLexeme lexbuf) of
                              NONE   => lexerError lexbuf "Bad integer"
                            | SOME i => Parser.NUM (i, getPos lexbuf)
                     }
  | [`~`]?(([`0`-`9`]+(`.`[`0`-`9`]* )?|`.`[`0`-`9`]+))
    ([`e``E`][`+``~`]?[`0`-`9`]+)?
                     { case Real.fromString (getLexeme lexbuf) of
                              NONE   => lexerError lexbuf "Bad float"
                            | SOME x => Parser.FLOAT (x, getPos lexbuf)
                     }
  | ([`a`-`z`] | [`A`-`Z`]) ([`a`-`z`] | [`A`-`Z`] | [`0`-`9`])*
                     { keyword (getLexeme lexbuf, getPos lexbuf) }
  | `+`              { Parser.PLUS (getPos lexbuf) }
  | `-`              { Parser.MINUS (getPos lexbuf) }
  | `*`              { Parser.TIMES (getPos lexbuf) }
  | `/`              { Parser.DIVIDE (getPos lexbuf) }
  | `(`              { Parser.LPAR (getPos lexbuf) }
  | `)`              { Parser.RPAR (getPos lexbuf) }
  | `=`              { Parser.EQ (getPos lexbuf) }
  | eof              { Parser.EOF (getPos lexbuf) }
  | _                { lexerError lexbuf "Illegal symbol in input" }

;
```

Figure 1: Lexerdefinitionen `Lexer.lex`

globale variabler: `currentLine` angiver det nuværende linienummer og `lineStartPos` angiver positionerne af starten af linierne (målt som antal tegn fra starten af teksten). Når vi læser et linieskifttegn, tæller vi op i `currentLine` og tilføjer den nuværende position til `lineStartPos`. Funktionen `getPos` returnerer et par bestående af `currentLine` og positionen på linien.

Vi skal også kunne rapportere leksikalske fejl. Til det formål erklærer vi en *exception* `LexicalError` og en hjælpefunktion `lexerError`. Vi vil senere se, hvordan vi fanger denne *exception*.

Dernæst erklærer vi en hjælpefunktion `keyword`. Denne bruges til at genkende nøgleord: Alle alfanumeriske tegnfølger, der starter med et bogstav genkendes med et regulært udtryk, og hjælpefunktionen finder ud af om der er tale om et nøgleord eller en variabel og returnerer den passende token. Læg mærke til at vi sætter modulnavnet `Parser` foran hvert tokennavn og giver positionen med som attribut. `Parser.ID` har endvidere navnet med som attribut.

Derefter følger de regulære udtryk, der definerer tokens. Notationen minder om notationen for produktioner: Hver linie indeholder et regulært udtryk og en aktion i krølleparenteser, og linierne adskilles med "`|`" og afsluttes med semikolon. Aktionerne er SML udtryk, som returnerer værdien af den genkendte token. Hvis man ikke vil returnere en token (f.eks. hvis det genkendte er en kommentar eller blanktegn), skriver man "`Token lexbuf`" for at kalde lexeren igen for at finde næste token. Det ses f.eks. i den første definition, der skipper *whitespace* (bestående af blanktegn, tabulatortegn og vognretur). En komplet liste af escape-sekvenser kan ses i "Moscow ML Owner's Manual". Læg mærke til at enkelttegn er omsluttet af *backquotes*, ikke almindelige anførselstegn.

Den næste definition behandler linieskift (tegn 12, *form feed*) betragtes her også som liniskift). Det er også *whitespace*, der skal skippes, men inden lexeren kaldes igen opdateres variablerne `currentLine` og `lineStartPos`.

Dernæst behandles kommentarer. De består af et *backslash* og fortsætter til næste linieskift. Derfor angives tegnene efter *backslash*'et som *alt andet end linieskift*. Dette gøres ved at starte en liste af tegn med symbolet "`^`", som inverterer den efterfølgende mængde af tegn.

Så kommer vi til de egentlige tokens. Først heltalskonstanter, der består af en ikke-tom følge af cifre, muligvis med et foranstillet minustegn. Her bruges SML's unære minus, som skrives som en tilde ("`~`"). For at få værdien kaldes SML funktionen `Int.fromString`, som konverterer en tegnfølge til et heltal. Hvis tallet er for stort eller der er andre fejl, returneres `NONE`, som her giver anledning til en leksikalsk fejl.

Floating point konstanter behandles på lignende måde, men det regulære udtryk er noget mere kompliceret. Pånær brugen af *backquotes* til at omslutte enkelttegn er notationen den samme som i "Basics of Compiler Design". Der er overlap mellem heltalskonstanter og floating-point konstanter. Da definitionen for heltalskonstanter er angivet først, vil tegnfølger i fællesmængden blive genkendt som heltal.

Hernæst behandles navne. De begynder med et bogstav og fortsætter med bogstaver eller cifre. Som tidligere nævnt, bruger vi hjælpefunktionen `keyword` til at udskille nøgleord. Derefter følger en række enkelttegn tokens og `eof`, der angiver `end-of-file`. Til sidst er der et *default pattern*, der matcher alt det, der ikke matches af de ovenstående definitioner. Det giver anledning til en leksikalsk fejl.

Der er enkelte linier i lexerdefinitionen, som ikke er beskrevet herover. De kan betragtes som

en række besværgelser, der bare skal være der for at det virker. Lexeren genereres og oversættes med kaldet

```
> mosmllex Lexer.lex
> mosmlc -c Lexer.sml
```

`mosmllex` vil ved succesfuld kørsel angive antallet af DFA-tilstande og aktioner. Sidstnævnte er antallet af lexer-definitioner (med tilhørende aktioner). Husk at parseren skal være oversat før lexeren kan oversættes, da lexeren bruger tokennavne, der er erklæret i `Parser.sig`.

5 Fortolkeren

Fortolkeren findes i filerne `Interpreter.sig` og `Interpreter.sml`. De er for store at inkludere her, men læg mærke til at positionsangivelserne i den abstrakte syntaks bruges når der rapporteres fejl.

6 Hovedprogrammet

For at samle trådene laves et hovedprogram, der læser en fil og kalder lexer og parser med denne. Hovedprogrammet ses i figur 2. Programmet består i det væsentlige af sort magi, der åbner inddatastrømme, kalder lexer og parser og behandler *exceptions*. Læg dog mærke til at parseren kaldes ved at angive startsymbolet (`Parser.Defs`) og at den har lexerens startsymbol (`Lexer.Token`) og inddatastrømmen som argumenter.

Hovedprogram oversættes med kommandoen

```
> mosmlc -o Main Main.sml
```

`Main` køres fra kommandolinien med et filnavn som argument. Den angivne fil (f.eks. `test.defs`) indlæses og køres gennem lexer og parser. Hvis der findes fejl under lexing eller parsing, rapporteres disse. Ellers beregnes definitionerne og værdierne af hver variabel udskrives. Hvis der opdages typefejl e.lign. på køretid, rapporteres disse på det tidspunkt, de findes.

Filerne `error1.defs`, `error2.defs` og `error3.defs` indeholder fejlbehæftede definitioner. Når Main kaldes med disse filer, rapporteres fejl.

```
structure Main =
struct

  fun createLexerStream ( is : BasicIO.instream ) =
      Lexing.createLexer (fn buff => fn n => Nonstdio.buff_input is buff 0 n)

  fun compile filename =
      let val lexbuf = createLexerStream (BasicIO.open_in filename)
          val defs =  (Parser.Defs Lexer.Token lexbuf)
      in
        Interpreter.evalDefs defs
      end


  fun errorMess s = TextIO.output (TextIO.stdErr,s ^ "\n");

  val _ = compile (List.nth(Mosml.argv (),1))
          handle Parsing.yyexit ob => errorMess "Parser-exit\n"
               | Parsing.ParseError ob =>
                   let val Location.Loc (p1,p2)
                             = Location.getCurrentLocation ()
                       val (lin,col)
     = Lexer.getLineCol p2
(!Lexer.currentLine)
(!Lexer.lineStartPos)
                   in
                     errorMess ("Parse-error at line "
                      ^ makestring lin ^ ", column " ^ makestring col)
                   end
               | Lexer.LexicalError (mess,(lin,col)) =>
                     errorMess ("Lexical error: " ^mess^ " at line "
                      ^ makestring lin ^ ", column " ^ makestring col)
       | Interpreter.RunError (mess,(lin,col)) =>
                     errorMess ("Runtime error: " ^mess^ " at line "
                      ^ makestring lin ^ ", column " ^ makestring col)
               | SysErr (s,_) => errorMess ("Exception: " ^ s)
end
```

Figure 2: Hovedprogrammet `Main.sml`