Fakulteten för ekonomi, kommunikation och IT

Programspråk

Laboration 3

Datum:	2007-09-03
Namn:	Henrik Bäck
	Mathias Andersson
Kurs:	DAV C02

Innehållsförteckning

Parsing av Pascalprogram

Lexer

Lexerns uppgift är att serva Parsern med tokens från den aktuella programkodskällan. När programmet öppnas läses det in till en databuffer. Från denna buffer hämtas sedan tecken för att tolkas till tokens. Större delen av lexern är inte skriven på egen hand utan var tillhandahållen vid start.

Parsern

Genom att omforma grammatikreglerna för Pascal så att de kan användas med högerrekursion blir skapandet av själva parsern mycket simpel. När detta var gjort är det möjligt att bygga upp funktioner som motsvarar enskilda delar av grammatiken. Det här leder i sin tur till att programkoden för varje funktion blir mycket liten och varje funktion består i stort sett av ett antal anrop till andra funktioner. I slutänden används en funktion för att jämföra att den på platsen förväntade token också finns där i den filen som parsas. På så sätt upptäcks eventuella avvikelser i det inlästa programmet från den korrekta grammatiken.

Parsning av ett program

Parsern börjar med att kontrollera så att programhuvudet motsvarar det förväntade utseendet för ett programhuvud. Detta gör den enkelt genom att kontrollera så att ta första elementet i listan och kontrollera det mot det förväntade värdet. Vid varje matchning kommer denna token att tas bort från listan.

På detta sätt kontrolleras även resten av programkoden. Eftersom ingen kontroll görs så att variabler är definierade eller av rätt typ kommer enbart varje token att kontrolleras mot det som är förväntat. Detta gäller både för variabel- och för program-delen av programmet.

Det sista som sker är att programmets fot kontrolleras. Här måste det förekomma ett `end` följt av en punkt (`.`). När `end` följt av en punkt (`.`) har påträffats så är kontrollen av programmet över och parsen avslutas. Om det skulle finnas extra tecken efter detta så kommer det resultera i ett fel.

Så fort ett fel påträffas någonstans i koden kommer funktionen för match att returnera falskt. Detta gör att parsningen avbryts.

Bilaga – Programkod

Parser.lsp

```
;===========================================================
; Denna funktion laeser fraan stream. Extraherar ett lexeme.
; skippa white-space tecken
;===========================================================

(defun stream-collect-lexeme (stream ws-characters)

  (when (not (eq 'EOF (loop
                         (let ((c (peek-char nil stream nil 'EOF)))
                          (if (or (eq c 'EOF) (not (member c ws-characters)))
                              (return c)
                            (read-char stream))))))
    (let ((lexeme ""))
      (loop
       (let ((c (peek-char nil stream nil 'EOF)))
            (if (or (eq c 'EOF) (member c ws-characters))
                (return lexeme)
              (progn
                (read-char stream)
                           ; Gah! Var fan finns string-append-char!??!?!!?
                           ; Taenk: (setf lexeme (string-append-char lexeme c))
                (setf lexeme
                    (concatenate 'string lexeme
                         (make-string 1 :initial-element c))))))))))

;===========================================================
; Returvaerde fraan get-next-token aer alltid en lista.
; car  returvaerde = Token   (ex: 'IF ':= '< osv.)
; cadr returvaerde = Lexeme  (ex: "IF"  ":=" "<")
;===========================================================

(defun get-next-token (stream lexeme-mapper)
  (let ((result (stream-collect-lexeme stream '(#\Space #\Tab #\Newline))))
    (if (null result) (list 'EOF "")(funcall lexeme-mapper result))
  )
 )

;===========================================================
;Haer skapas en lista innehaallande tvaa element:
;(symbol, lexeme)
;===========================================================

(defun map-lexeme (lexeme)
;(format t "The incoming lexeme is: ~S ~%" lexeme)
   (list (cond

      ((string= lexeme "program")    'PROGRAM)
      ((string= lexeme "var")        'VAR)
      ((string= lexeme "input")          'INPUT)
      ((string= lexeme "output")   'OUTPUT)
      ((string= lexeme ":=")         'ASSIGN)
      ((string= lexeme ",")          'COMMA)
      ((string= lexeme ";")          'SCOLON)
      ((string= lexeme ":")          'COLON)
      ((string= lexeme "(")          'LP)
      ((string= lexeme ")")          'RP)
      ((string= lexeme "+")          'ADD)
      ((string= lexeme "*")          'TIMES)
      ((string= lexeme "integer")    'INTEGER)
      ((string= lexeme "real")     'REAL)
      ((string= lexeme "begin")      'BEGIN)
      ((string= lexeme ".")        'DOT)
      ((string= lexeme "end")        'END)

      ((is-id lexeme)                'ID)
      ((is-number lexeme)            'NUMBER)

      (t                             'UNKNOWN)
```

```
      )
    lexeme)
)

;===========================================================
;Jaemfoer om foersta tecknet i straengen aer en bokstav
;samt att resterande tillhoer maengden {a-z, A-Z, 0-9}
;===========================================================

(defun is-id (str)
           (and (alpha-char-p (char str 0))
                    (every #'is-alphanum str)
           )
)

(defun is-alphanum(char)
           (cond
                  ((digit-char-p char) t)
                  ((alpha-char-p char) t)
                  (t                                    nil)
           )
)


(defun is-number (str)
           (every #'digit-char-p str)
)

;===========================================================
;Skapar en strukt med tvaa faelt:
;lookahead aer en lista (symbol, lexeme), stream aer en
;filstroem.
;===========================================================

(defstruct pstate
    (lookahead)
    (stream)
)

;===========================================================
;Konstruktor foer strukten pstate, initialiserar
;stream till inskickad filstroem och lookahead
;till foersta inlaesta token (symbol, lexeme)
;===========================================================

(defun create-parser-state (stream)

    (make-pstate :stream stream
                 :lookahead (get-next-token stream #'map-lexeme)
    )

)

;===========================================================
;match haemtar ett nytt token fraan filstroemmen om
;inskickad symbol matchar lookahead, annars har det
;uppstaatt ett syntax error.
;===========================================================

(defun match (state symbol)

    (if (eq symbol (first (pstate-lookahead state)))
        (setf (pstate-lookahead state)
            (get-next-token (pstate-stream state) #'map-lexeme)
        )
        (format t "*** Expected: ~S, found: ~D. ~%~%" symbol (first (pstate-lookahead
state)))
    )

)

;===========================================================
;Startat programmet genom att man vid promten skriver:
;
;(parser)
```

```
;
;Programmet kommer daa att fraaga efter ett filnamn, finns
;filen kommer den att bli parserad och ett meddelande
;kommer att skrivas ut som meddelar om parsningen gick bra.
;==========================================================

(defun parser (file)
           (format t "--- Parsing file: ~S --- ~%" file)
    (with-open-file (stream file :direction :input)
       (if  (program (create-parser-state stream))
                     (format t "*   Computer says yes!  (Parser says true). ~%~%")
                     (format t "*   Computer says no!   (Parser says false). ~%~%")
       )
    )
)

;==========================================================
;[prog header] ::= program id ( input , output );
;==========================================================
(defun progHeader(state)

           (and    (match state 'PROGRAM)
                          (match state 'ID)
                          (match state 'LP)
                          (match state 'input)
                          (match state 'COMMA)
                          (match state 'output)
                          (match state 'RP)
                          (match state 'SCOLON)
           )

)

;==========================================================
;[var part] ::= var [var dec list]
;==========================================================
(defun varPart(state)

           (and    (match state 'VAR)
                          (varDecList state)
           )
)

;==========================================================
;[var dec list] ::= [var dec] | [var dec list] [var dec]
;==========================================================
(defun varDecList(state)

           (and    (varDec state)
                          (if (not (eq 'BEGIN (first (pstate-lookahead state))))
                                 (VarDecList state)
                                 t
                          )
           )
)

;==========================================================
;[var dec] ::= [id list] : [type]  ;
;==========================================================
(defun varDec(state)

           (and (idList state)
                    (match state 'COLON)
                    (typeEn state)
                    (match state 'SCOLON)
           )
)

;==========================================================
;[id list] ::= id | [id list] , id
;==========================================================
(defun idList(state)

           (and    (match state 'ID)
                          (if (eq 'COMMA (first (pstate-lookahead state)))
```

```
                                  (and (match state 'COMMA)
                                        (idList state)
                                  )
                                  t
                          )
           )
)

;===========================================================
;[type] ::= integer | real
;===========================================================
(defun typeEn(state)

           (if (eq 'INTEGER (first (pstate-lookahead state)))
                  (match state 'INTEGER)
                  (match state 'REAL)
           )
)

;===========================================================
;[stat part] ::= begin [stat list] end .
;===========================================================
(defun statPart(state)

           (and    (match state 'BEGIN)
                          (statList state)
                          (match state 'END)
                          (match state 'DOT)
           )

)

;===========================================================
;[stat list] ::= [stat] | [stat list]  ;  [stat]
;===========================================================
(defun statList(state)

           (and    (stat state)
                          (if (eq 'SCOLON (first (pstate-lookahead state)))
                                  (and  (match state 'SCOLON)
                                             (statList state)
                                  )

                                  t
                          )
           )
)

;===========================================================
;[stat] ::= [assign stat]
;===========================================================
(defun stat(state)

           (assignStat state)
)

;===========================================================
;[assign stat] ::= id := [expr]
;===========================================================
(defun assignStat(state)

           (and    (match state 'ID)
                          (match state 'ASSIGN)
                          (expr state)
                          )
)

;===========================================================
;[expr] ::=  [operand] | [expr] [operator] [operand]
;===========================================================
(defun expr(state)

           (and (operand state)
                   (if    (or  (eq 'TIMES (first (pstate-lookahead state)))
                                       (eq 'ADD (first (pstate-lookahead state)))
```

```
                                   )
                           (and (operatorn state)
                                    (expr state)
                           )
                           t
                   )
           )
)

;==========================================================
;[operand] ::= id | number
;==========================================================
(defun operand(state)

           (if (eq 'ID (first (pstate-lookahead state)))
                  (match state 'ID)
                  (match state 'NUMBER)
           )

)
;==========================================================
;[operator] ::=  + | *
;==========================================================
(defun operatorn(state)

           (if (eq 'ADD (first (pstate-lookahead state)))
                  (match state 'ADD)
                  (match state 'TIMES)
           )
)

;==========================================================
;<program> --> <program-header><var-part><stat-part>EOF
;==========================================================

(defun program (state)

   (and (progHeader state)
                  (varPart state)
                  (statPart state)
        (match state 'EOF)
   )

)

(defun start()

           (mapcar  #'parser  '(Test/testok1.pas  Test/testok2.pas  Test/testok3.pas
Test/testok4.pas  Test/testok5.pas  Test/testok6.pas  Test/testok7.pas  Test/fun1.pas
Test/fun2.pas Test/fun3.pas Test/fun4.pas Test/fun5.pas Test/testa.pas Test/testb.pas
Test/testc.pas    Test/testd.pas    Test/teste.pas    Test/testf.pas    Test/testg.pas
Test/testh.pas    Test/testi.pas    Test/testj.pas    Test/testk.pas    Test/testl.pas
Test/testm.pas    Test/testn.pas    Test/testo.pas    Test/testp.pas    Test/testq.pas
Test/testr.pas    Test/tests.pas    Test/testt.pas    Test/testu.pas    Test/testv.pas
Test/testw.pas    Test/testx.pas    Test/testy.pas    Test/testz.pas    Test/sem1.pas
Test/sem2.pas Test/sem3.pas Test/sem4.pas Test/sem5.pas))


)

(start)
```

Bilaga – Körning

```
--- Parsing file: TEST/TESTOK1.PAS ---
*   Computer says yes!  (Parser says true).

--- Parsing file: TEST/TESTOK2.PAS ---
*   Computer says yes!  (Parser says true).

--- Parsing file: TEST/TESTOK3.PAS ---
*   Computer says yes!  (Parser says true).
```

```
--- Parsing file: TEST/TESTOK4.PAS ---
*   Computer says yes!  (Parser says true).

--- Parsing file: TEST/TESTOK5.PAS ---
*   Computer says yes!  (Parser says true).

--- Parsing file: TEST/TESTOK6.PAS ---
*   Computer says yes!  (Parser says true).

--- Parsing file: TEST/TESTOK7.PAS ---
*   Computer says yes!  (Parser says true).

--- Parsing file: TEST/FUN1.PAS ---
*   Computer says yes!  (Parser says true).

--- Parsing file: TEST/FUN2.PAS ---
*** Expected: ID, found: UNKNOWN.

*   Computer says no!   (Parser says false).

--- Parsing file: TEST/FUN3.PAS ---
*** Expected: ASSIGN, found: COLON.

*   Computer says no!   (Parser says false).

--- Parsing file: TEST/FUN4.PAS ---
*   Computer says yes!  (Parser says true).

--- Parsing file: TEST/FUN5.PAS ---
*** Expected: EOF, found: UNKNOWN.

*   Computer says no!   (Parser says false).

--- Parsing file: TEST/TESTA.PAS ---
*** Expected: PROGRAM, found: EOF.

*   Computer says no!   (Parser says false).

--- Parsing file: TEST/TESTB.PAS ---
*** Expected: PROGRAM, found: ID.

*   Computer says no!   (Parser says false).

--- Parsing file: TEST/TESTC.PAS ---
*** Expected: ID, found: LP.

*   Computer says no!   (Parser says false).

--- Parsing file: TEST/TESTD.PAS ---
*** Expected: LP, found: INPUT.

*   Computer says no!   (Parser says false).

--- Parsing file: TEST/TESTE.PAS ---
*** Expected: INPUT, found: COMMA.

*   Computer says no!   (Parser says false).

--- Parsing file: TEST/TESTF.PAS ---
*** Expected: COMMA, found: OUTPUT.

*   Computer says no!   (Parser says false).

--- Parsing file: TEST/TESTG.PAS ---
*** Expected: OUTPUT, found: RP.

*   Computer says no!   (Parser says false).

--- Parsing file: TEST/TESTH.PAS ---
*** Expected: RP, found: SCOLON.

*   Computer says no!   (Parser says false).

--- Parsing file: TEST/TESTI.PAS ---
*** Expected: SCOLON, found: VAR.
```

```
*   Computer says no!   (Parser says false).

--- Parsing file: TEST/TESTJ.PAS ---
*** Expected: VAR, found: ID.

*   Computer says no!   (Parser says false).

--- Parsing file: TEST/TESTK.PAS ---
*** Expected: ID, found: COMMA.

*   Computer says no!   (Parser says false).

--- Parsing file: TEST/TESTL.PAS ---
*** Expected: COLON, found: ID.

*   Computer says no!   (Parser says false).

--- Parsing file: TEST/TESTM.PAS ---
*** Expected: COLON, found: INTEGER.

*   Computer says no!   (Parser says false).

--- Parsing file: TEST/TESTN.PAS ---
*** Expected: REAL, found: SCOLON.

*   Computer says no!   (Parser says false).

--- Parsing file: TEST/TESTO.PAS ---
*** Expected: SCOLON, found: BEGIN.

*   Computer says no!   (Parser says false).

--- Parsing file: TEST/TESTP.PAS ---
*** Expected: COLON, found: ASSIGN.

*   Computer says no!   (Parser says false).

--- Parsing file: TEST/TESTQ.PAS ---
*** Expected: ID, found: ASSIGN.

*   Computer says no!   (Parser says false).

--- Parsing file: TEST/TESTR.PAS ---
*** Expected: ASSIGN, found: UNKNOWN.

*   Computer says no!   (Parser says false).

--- Parsing file: TEST/TESTS.PAS ---
*** Expected: ASSIGN, found: COLON.

*   Computer says no!   (Parser says false).

--- Parsing file: TEST/TESTT.PAS ---
*** Expected: ASSIGN, found: ID.

*   Computer says no!   (Parser says false).

--- Parsing file: TEST/TESTU.PAS ---
*** Expected: NUMBER, found: ADD.

*   Computer says no!   (Parser says false).

--- Parsing file: TEST/TESTV.PAS ---
*** Expected: END, found: ID.

*   Computer says no!   (Parser says false).

--- Parsing file: TEST/TESTW.PAS ---
*** Expected: END, found: UNKNOWN.

*   Computer says no!   (Parser says false).

--- Parsing file: TEST/TESTX.PAS ---
*** Expected: ID, found: UNKNOWN.
```

```
*   Computer says no!   (Parser says false).

--- Parsing file: TEST/TESTY.PAS ---
*** Expected: END, found: DOT.

*   Computer says no!   (Parser says false).

--- Parsing file: TEST/TESTZ.PAS ---
*** Expected: DOT, found: EOF.

*   Computer says no!   (Parser says false).

--- Parsing file: TEST/SEM1.PAS ---
*   Computer says yes!  (Parser says true).

--- Parsing file: TEST/SEM2.PAS ---
*   Computer says yes!  (Parser says true).

--- Parsing file: TEST/SEM3.PAS ---
*** Expected: REAL, found: ID.

*   Computer says no!   (Parser says false).

--- Parsing file: TEST/SEM4.PAS ---
*   Computer says yes!  (Parser says true).

--- Parsing file: TEST/SEM5.PAS ---
*   Computer says yes!  (Parser says true).
```