OCAML YACC
I.Savnik

2008/10/25
2012/10/11 (popravki)

1. POGLAVJE

UVOD

Ocamlyacc je splošen generator razčlenjevalnikov, ki prevede opis kontekstno neodvisne gramatike podane v LALR(1) obliki v Ocaml program.

Uporabljamo ga za razvoj enostavnih in kompleksnih programskih jezikov.

Ocamlyacc temelji na programu yacc (GNU bison), ki je del večine C programskih okolij. Prehod iz yacc na ocamlyacc bi moral biti enostaven.

Za delo z Ocamlyacc je potrebno poznavanje programskega jezika Ocaml.

2. POGLAVJE

JEZIKI IN KONTEKSTNO NEODVISNE GRAMATIKE

Ocamlyacc uporablja kontekstno neodvisne gramatike za opis jezikov.

Definiramo sintaktične skupine, ki definirajo simbole, in podamo pravila za izpeljavo simbolov. Primer sintaktične skupine, ki je uporabljena za opis programskega jezika je izraz (expression).

Pravila so pogosto rekurzivna. Obstajati mora vsaj eno pravilo, ki vodi iz rekurzije.

Najbolj pogosta oblika, ki se uporablja za opis pravil jezikov je Bacus-Naurova oblika ali na kratko BNF. Gramatika opisana v BNF je v splošnem konteksno neodvisna gramatika. Vhod v Ocamlyacc je oblika zapisa BNF.

Ocamlyacc oz. yacc zna delati z podmnožico KNG, ki se imenujejo LALR(1). Za LR(1) gramatiko je možno definirati pravilo na osnovi prvega naslednjega simbola. LALR(1) vsebuje dodatne omejitve, ki si jih lahko ogledate v knjigi [Aho,Ullman,Hopcroft:Compilers].

Pri formalnih gramatičnih pravilih označimo vsako sintaktično enoto ali skupino z imenom. Simbole, ki grupirajo manjše konstrukte v skladu z gramatičnimi pravili imenujemo neterminalni simboli. Simbole, ki jih ni mogoče naprej deliti imenujemo terminalni simboli ali žetoni.

Uporabili bomo programski jezik C kot primer jezika, kjer si bomo

pogledali kaj so terminalni in neterminalni simboli.

Žetoni programskega jezika C so identifikatorji, konstante (numerične in
nizi), aritmetične operacije in ključne besede. Poglejmo si nekaj primerov.
Terminalni simboli za identifikatorje in konstante so: 'identifier', 'string'
in 'number'. Terminalni simboli za ključne besede so: 'if', 'return', 'const', itd.

Poglejmo si primer C funkcije, ki je razbita v žetone.

```
int                 /*   keyword ‘int’ */
square (int x)      /*   identifier, open-paren, keyword ‘int’, identifier, close-paren */
{                   /*   open-brace */
   return x * x;    /*   keyword ‘return’, identifier, asterisk, identifier, semicolon */
}                   /*   close-brace */
```

Sintaktične skupine jezika C so izrazi, stavki, deklaracije in
definicija funkcije. Te skupine so v gramatiki predstavljene z
netermnalnimi simboli: 'expression', 'statement', 'declaration'
in 'function definition'.

Polna gramatika uporablja še vrsto drugih neterminalnih simbolov,
ki predstavljajo vse preostale sintaktične skupine simbolov.
Primer zgoraj je definicija funkcije. Vsaka spremenljivka
je del izraza, ki je identificiran z neterminalnim simbolom.

Vsak neterminalni simbol mora imeti gramatična pravila, ki
pove kako je sintaktični konstrukt sestavljen iz manjših komponent.
Na primer, 'return' stavek programskega jezika C, je opisan s
pravilom, ki ga neformalno beremo na sledeč način.

```
stmt:      RETURN expr SEMICOLON
           ;
```

Stavek označen z neterminalnim simbolom ‘stmt’ je lahko
sestavljen iz ključne besede 'RETURN', ki ji sledi izraz predstavljen
z neterminalnim simbolom 'exp' in končno podpiče označeno
z žetonom 'SEMICOLON'.

Vsak stavek v C je predstavljen s svojim pravilom! Pravilo
je identificirano z neterminalnim simbolom, ki definira vrsto
opisanega stavka.

Eden izmed neterminalnih simbolov mora služiti kot začetna
točka razčlenjevalnika jezika. Ta simbol imenujemo začetni
simbol. V primeru programskega jezika C imamo neterminalni
simbol 'sekvenca definicij in deklaracij', ki služi kot
začetna točka za razčlenjevanje C programov.

Poglejmo si še en primer. Izraz '1 + 2' je legalen C izraz,
ni pa veljaven kot celoten program. V okviru KNG programskega
jezika C 'izraz' ('expession') ni začetni simbol jezika.

OPIS GRAMATIKE

Vhod ocamlyacc je datoteka, ki vsebuje gramatiko. Izhod je Ocaml
izvorna datoteka, ki razčlenjuje jezik opisan z gramatiko.
To datoteko imenujemo ocamlyacc razčlenjevalnik.

Neterminalni simbol gramatike je predstavljen v Ocamlyacc datoteki

z gramatiko kot simbol v Ocaml. Simbol se ne sme končati z '
(enojni narekovaj) in se mora začeti z malimi črkami.

Terminalni simboli so v Ocamlyacc predstavljeni s tipi žetonov.
Ti morajo biti deklarirani v deklaracijski sekciji Ocamlyacc.
Definirani so kot konstruktorji konkretnih tipov žetnov.
Tipi žetonov se morajo začeti z veliko začetnico. Na primer,
tipi Integer, Identifier, itd.

Gramatična pravila so tudi del sintakse Ocamlyacc. Na primer, imamo
Ocamlyacc pravilo za C return stavek.

```
stmt:      RETURN expr SEMICOLON
           ;
```

SEMANTIČNE VREDNOSTI

Formalna gramatika opisuje žetone po klasifikaciji vhodnih besed. Na
primer, če pravilo vsebuje 'celo število' potem to pomeni, da je na
tistem mestu celo število gramatično pravilno. Natančna vrednost
celega števila kot tudi sintaktična struktura celega števila nista
pomembni.

Vrednost konstante je pomembna za pomen programa po tem, ko je
program razčlenjen. Vsak žeton mora torej imeti oboje: tip žetona kot
tudi vrednost.

Tip žetona je terminalni simbol definiran v gramatiki,
na primer, INTEGER, IDENTIFIER ali SEMICOLON. Tip pove
razčlenjevalniku vse kar je potrebno na nivoju gramatike.
Gramatična pravila ne vejo ničesar drugega o žetonih kot njihov tip.

Semantična vrednost žetona vsebuje preostale podatke o pomenu žetona,
npr. vrednost celega števila, ime identifikatorja, itd. Enostavni
žetoni kot so npr. SEMICOLON in podobni ne potrebujejo dodatnih
podatkov predstavljenih s semantično vrednostjo žetona.

Poglemo si primer. Vhodni niz znakov je klasificiran kot žeton tipa
INTEGER in ima semantično vrednost 4. Drugi vhodni niz, na primer, ima
tudi tip žetona INTEGER, vendar ima vrednost 3568. Če gramatično
pravilo pove, da se na določenem mestu sprejme žeton INTEGER, potem
sama vrednost na nivoju razčlenjevanja ni pomembna.

Vsaka semantična skupina predstavljena z neterminalnim simbolom ima
tudi lahko semantično vrednost. Na primer, v primeru kalkulatorja
predstavlja semantična vrednost izraza samo vrednost izraza. V
primeru, da pišemo prevajalnik za nek programski jezik so semantične
vrednosti izrazov tipično abstraktna sintaktična drevesa, ki pripadajo
razčlenjenem izrazu.

SEMANTIČNE AKCIJE

Program mora narediti več kot samo razčlenjevanje vhodnega niza
simbolov. Mora generirati tudi kakšen izhod na osnovi vhoda programa.
Gramatična pravila v ocamlyacc imajo lahko povezane akcije, ki se
izvedejo po tem, ko se pravilo prepozna.

Običajno je namen akcije izračun semantične vrednosti celotnega
konstrukta iz semantičnih vrednosti posameznih delov predstavljenih s

pravilom. Na primer, recimo da imamo pravilo, ki pravi, da je izraz vsota dveh izrazov.

```
expr: expr PLUS expr          { $1 + $3 }
           ;
```

Ko razčlenjevalnik prepozna pravilo ima vsaka od komponent že prirejeno semantično vrednost. Akcija sestavi delne vrednosti v končno vrednost, ki se vrne kot semantična vrednost izraza oz. vsote v našem primeru.

LOKACIJE

Veliko aplikacij kot so npr. tolmač ali prevajalnik potrebujejo natančne podatke o zvezi med terminalnimi in tudi neterminalnimi simboli in tekstom v vhodni datoteki. Ocamlyacc nudi mehanizme, ki to omogočajo.

Vsak žeton ima semantično vrednost. Podobno kot semantična vrednost je z žetonom povezana tudi lokacija z razliko, da je tip lokacije enak za vse žetone.

Generiran razčlenjevalnik ima torej vgrajeno podatkovno strukturo za shranjevanje lokacij. Podobno kot s semantičnimi vrednostmi nudi ocamlyacc dostop do lokacij znotraj akcije.

FAZE SPECIFIKACIJE ocamlyacc GRAMATIKE

Načrtovanje jezika z uporabo Ocamlyacc, od specifikacije gramatike do delovnega prevajalnika ali tolmača, ima naslednje dele:

- Formalno definiraj gramatiko v obliki, ki jo zahteva Ocamlyacc. Za vsako gramatično pravilo opiši akcijo, ki je potrebna potem, ko se pravilo prepozna. Akcija se opiše s sekvenco ocaml stavkov.

- Napiši leksikalen analizator, ki procesira vhodni niz znakov in podaja žetone razčlenjevalniku. Leksični analizator se lahko napiše z uporabo ocamllex ali kar direktno v ocaml.

- Napiši kontrolno funkcijo, ki pokliče ocamlyacc razčlenjevalnik.

- Napiši kodo za delo z napakami.

- Poženi ocamlyacc nad napisano gramatiko, da bi konstruirali razčlenjevalnik.

- Prevedi ocaml kodo za razčlenjevalnik in preostalo kodo, ki se uporablja skupaj z razčlenjevalnikom.

- Poveži objektne datote v končni izvedljiv porgram.

OSNOVNA OBLIKA ocamlyacc GRAMATIKE

Vhodna datoteka za ocamlyacc vsebuje gramatika jezika. Splošna oblika ocamlyacc datoteke je sledeča.

```
%{
      Header (Ocaml code)
%}
      Ocamlyacc declarations
%%
      Grammar rules
%%
      Trailer (Additional Ocaml code)
```

Znaki %%, %{ and %} imajo poseben pomen v datoteki. Znaka %{ in %}
definirataa začetek in konec glave datoteke z gramatiko. V glavi so
tipi, spremenljivke in funkcije, ki jih uporabimo v akcijah.

Z znaki %% ločimo preostale sekcije.

V sekciji ocamlyacc deklaracij se deklarirajo imena terminalnih in
netermnalnih simbolov, opišejo prioritete operacij ter definirajo
tipi semantičnih vrednosti žetonov.

Gramatična pravila definirajo kako se konstruira vsak posamezen
neterminalen simbol.

Zadnji del datoteke (trailer) lahko vsebuje poljubno ocaml kodo.

3. POGLAVJE

PRIMER: KALKULATOR

Prvi primer predstavlja enostaven kalkulator v obrnjenem poljskem
zapisu, ki deluje nad realnimi števili. Primer predstavlja primerno
začetno izhodišče, predvsem ker se ne bo potrebno ukvarjati s prioriteto
operacij. Naslednji primer bo prikazal tudi delo s prioritetami
operacij.

DEKLARACIJE "rpcalc"

Poglejmo si deklaracije za kalkulator v obrnjenem poljskem zapisu.

```
/* file: rpcalc.mly */
/* Reverse polish notation calculator. */
%{
open Printf
%}
%token <float> NUM
%token PLUS MINUS MULTIPLY DIVIDE CARET UMINUS
%token NEWLINE
%start input
%type <unit> input
%% /* Grammar rules and actions follow */
```

Sekcija glave ocamlyacc opisa vsebuje kodo za odpiranje Printf"
modula.

V drugi sekciji ocamlyacc deklaracij imamo definicijo tipov žetonov.

Tukaj mora biti deklariran vsak terminalni simbol gramatike. Prvi terminalni simbol je tip žetona za numerične konstante, ki imajo realno vrednost. Pripadajoče operacije so PLUS, MINUS, MULTIPLY, DIVIDE, CARET za eksponent in UMINUS za unarni minus. Terminalni simboli za opis operacij nimajo povezanih nobenih vrednosti. Zadnji terminalni simbol je NEWLINE, tip žetona za znak 'nova vrstica'.

Podati moramo tudi imena začetnih simbolov in njihovih tipov. V našem primeru imamo začetni simbol 'input', ki ima tip 'unit'. Za vsak začetni simbol je definirana funkcija razčlenjevalnika z istim imenom kot je podano.

PRAVILA GRAMATIKE rpcalc

Poglejmo si zdaj gramatična pravila za kalkulator v obrnjenem poljskem zapisu.

```
input:         /* empty */ { }
           | input line { }
;
line:          NEWLINE { }
           | exp NEWLINE { printf "\t%.10g\n" $1; flush stdout }
;
exp:           NUM   { $1 }
           |   exp exp PLUS { $1 +. $2 }
           |   exp exp MINUS { $1 -. $2 }
           |   exp exp MULTIPLY { $1 *. $2 }
           |   exp exp DIVIDE { $1 /. $2 }
          /*   Exponentiation */
           |   exp exp CARET { $1 ** $2 }
          /*   Unary minus      */
            | exp UMINUS      { -. $1 }
;
%%
```

Pravili za definicijo neterminalnega simbola 'line' sta dve.
Prvo opisuje prazno vrstico in drugo mora vsebovati izraz, ki se zaključi z NEWLINE.

Pravil za definicijo netermnalnega simbola 'exp' je več.
Povezana so z operacijo |, ki definira alternative.
V naslednjih sekcijah je predstavljeno bolj natančno kaj pomenijo predstavljena pravila.

Semantika jezika je definirana z akcijami, ki se izvedejo po tem, ko se določeno pravilo ujame na vhodnem nizu. Akcije se nahajajo v kodi, ki sledi vsakemu posameznemu pravilu.

Akcije so definirane v ocaml, vendar ocamlyacc omogoči identificiranje in uporabo posameznih rezultatov ali semantičnih vrednosti izrazov, ki se ujamejo na pripadajočih produkcijskih pravilih.

Semantične vrednosti posameznih izrazov po vrsti od leve proti desni so označene s spremenljivakami $1, $2, itd.

OPIS VHODA

Poglejmo si definicijo vhoda.

```
input:        /* empty */
            | input line
;
```

Definicija se bere na naslednji način. Kompleten vhod je bodisi prazen niz ali (rekurzivno) kompleten vhod, ki mu sledi vhodna vrstica. Ta definicija je levo rekurzivna, ker se vhod ponovno pojavi na levi strani produkcijskega pravila.

Prva alternativa je prazna ker lahko razčlenjevalnik sprejme popolnoma prazen vhod. Običajno se napiše prazna alternativa med produkcijami prva.

Druga alternativa predstavlja netrivialen vhod. Produkcijo beremo na naslednji način. Po tem ko preberemo množico vrstic, preberi še eno vrstico. Leva rekurzija naredi iz te produkcije zanko, ki se lahko ponovi poljubno-krat. Izhod iz zanke je prvo pravilo, ki sprejme prazen vhod.

Razčlenjevalnik nadaljuje s procesiranjem vhoda tako dolgo dokler ne pride do napake ali do konca vhodnega toka žetonov.

OPIS VRSTICE

Poglejmo si zdaj definicijo vrstice.

```
line:         NEWLINE          { }
            | exp NEWLINE      { printf "\t%.10g\n" $1; flush stdout }
;
```

Prva alternativa pravila za vrstico je žeton NEWLINE. V tem primeru razčlenjevalnik sprejme prazno vrstico, ki ne vsebuje nobenega izraza. Druga alternativa za razčlenjevanje stavka je neterminalni simbol 'exp', ki predstavlja izraz, in ki mu sledi žeton NEWLINE.

Semantična vrednost izraza 'exp' je vrednost $1, ker je izraz 'exp' prvi med alternativami (edini v tem primeru).
Pripadajoča akcija izpiše to vrednost.

OPIS IZRAZOV

Imamo več pravil za opis izraza 'exp', eno za vsako vrsto izraza. Prvo pravilo obdela najenostavnejše izraze: tiste, ki vsebujejo samo eno realno število. Drugo pravilo pokrije seštevanje, ki je izraženo z dvemi izrazi 'exp', ki jima sledi operacija PLUS.
Tretje pravilo predstavlja odštevanje, itd.

```
exp:      NUM { $1 }
  | exp exp PLUS { $1 +. $2 }
  | exp exp MINUS { $1 -. $2 }
  ...
```

Uporabili smo operacijo | za združitev vseh pravil za 'exp', vendar bi lahko definirali ekvivalenten opis tako, da pravila pišemo narazen.

```
exp: NUM { $1 };
exp: exp exp PLUS { $1 +. $2 };
exp: exp exp MINUS { $1 -. $2 };
  ...
```

Vsa pravila imajo akcije, ki izračunajo vrednost izraza na osnovi vrednosti delov posameznega izraza.

Na primer, v akciji, ki se drži pravila za seštevanje, se $1 referencira na prvi izraz 'exp' in $2 referencira semantično vrednost drugega izraza. Tretja komponenta PLUS nima nobene posebne semantične vrednosti povezane z žetonom. Če bi takšno vrednost imela bi se na njo lahko sklicali z imenom $3.

Ko razčlenevalnik, prepozna oba podizraza vsote, ki jima sledi operacija za vsoto, lahko izračuna vsoto vrednosti podizrazov. Vrednost vsote se vrne kot semantična vrednost vsote podizrazov.

Zapis pravil, ki je tukaj predstavljen je priporočljiv, čeprav lahko uporabljamo tudi drugačen z uporabo dodatnih belih znakov. Na primer, sledeč zapis pravil je dovoljen.

```
exp: NUM { $1 } | exp exp PLUS { $1 +. $2 } | ...
```

pomeni isto kot:

```
exp:          NUM { $1 }
           | exp exp PLUS { $1 + $2 }
           | ...
```

LEKSKALNI ANALIZATOR rpcalc

Delo leksikalnega analizatorja je nizkonivojsko razčlenjevanje. Nize znakov prevaja v žetone. Vhod ocamlyacc so žetoni, ki so rezltat leksikalnega analizatorja. .

Za delo z rpcalc poterbujemo zelo enostaven leksikalni analizator. Analizator bere nize cifer kot realna števila in v tem primeru vrača žeton NUM. Operacije '+', '-', '*', '/', '^', 'n' pretvarja v pripadajoče žetone: ADD, MINUS, MULTIPLY, DIVIDE, CARET and UMINUS. Ko leksikalni analizator pride do '\n' vrne žeton NEWLINE. Preostale bele znake kot tudi nepoznane znake preskoči.

Vrednost, ki jo vrne leksikalni analizator (funkcija) je vrednost konkretnega tipa žetona. Semantična vrednost žetona (če obstaja) se vrne skupaj z žetonom.

Poglejmo si primer leksikalnega analzatorja.

```
(* file: lexer.mll *)
(* Lexical analyzer returns one of the tokens:
     the token NUM of a floating point number,
     operators (PLUS, MINUS, MULTIPLY, DIVIDE, CARET, UMINUS),
     or NEWLINE. It skips all blanks and tabs, unknown characters
     and raises End_of_file on EOF. *)
{
    open Rtcalc (* Assumes the parser file is "rtcalc.mly". *)
}
let digit = ['0'-'9']
```

```
rule token = parse
    | [' ' '\t'] { token lexbuf }
    | '\n' { NEWLINE }
    | digit+
    | "." digit+
    | digit+ "." digit* as num
    { NUM (float_of_string num) }
    | '+' { PLUS }
    | '-' { MINUS }
    | '*' { MULTIPLY }
    | '/' { DIVIDE }
    | '^' { CARET }
   | 'n' { UMINUS }
   | _ { token lexbuf }
   | eof { raise End_of_file }
```

KONTROLNA FUNKCIJA

V okviru tega primera je kontrolna funkcija razčlenjevalnika minimalna.
Za začetek razčlenjevanja je potrebno poklicati funkcijo Parser.input
z dvemi argumenti: funkcijo leksikalnega analizatorja Lexer.token in
lexbuf tipa Lexing.lexbuf.

```
(* file: main.ml *)
(* Assumes the parser file is "rtcalc.mly" and the lexer file is "lexer.mll". *)
let main () =
   try
      let lexbuf = Lexing.from_channel stdin in
      while true do
        Rtcalc.input Lexer.token lexbuf
      done
   with End_of_file -> exit 0
let _ = Printexc.print main ()
```

POROČANJE NAPAK

Ko razčlenjevalna funkcija odkrije sintaksno napako, pokliče funkcijo
z imenom parse_error s parametrom "syntax error". Privzeta funkcija
parse_error ne naredi ničesar ter se potem vrne na klicano mesto .
Uporabnik lahko definira svojo kodo za funkcijo parse_error v sekciji
glave datoteke, ki podaja gramatiko.

```
let parse_error s = (* Called by the parser function on error *)
   print_endline s;
   flush stdout
```

Po tem ko se funkcija parse_error vrne na klicano mesto lahko
ocamlyacc razčlenjevalnik nazaj vzpostavi konsistentno stanje in
nadaljuje z razčlenjevanjem, če vsebuje gramatika primerno pravilo za
obravnavanje napake. Sicer se razčlenjevalnik in dvigne izjemo
Parsing.Parse_error.

V tem primeru nismo napisali nobenega pravila za delo z napakami,
zatorej bo nepravilen vhod povzročil dvig izjeme in se ustavil. To
sicer ni "čisto" obnašanje realnega kalkulatorja, vendar zadošča za
prvi primer.

UPORABA PROGRAMA ocamlyacc

Preden poženemo ocamlyacc, da bi dobili razčlenjevalnik, se moramo odločiti kako razvrstiti izvorno kodo po datotekah. V našem primeru bomo izdelali tri datoteke: rpcalc.mly za gramatiko yacc, lexer.mll za lekser in main.ml, ki vsebuje glavno kontrolno funkcijo.

Naslednji ukaz spremeni datoteko z gramatiko v Ocaml program za razčlenjevalnik.

```
ocamlyacc file_name.mly
```

V tem primeru je vhodna datoteka z gramatiko rpcalc.mly. Ocamlyacc izdela datoteko rpcalc.ml, ki vsebuje izvorno kodo razčlenjevalnika. Vse dodatne funkcije izvorne datoteke '.mly' so direktno prepisane v izdelano datoteko razčlenjevalnika.

PREVAJANJE DATOTEKE RAZČLENJEVALNIKA

Poglejmo si kako se prevede ocamlyacc datoteko.

```
# List files in current directory.
$ ls
.depend Makefile lexer.mll main.ml                  rpcalc.mly
# Compile the Ocamlyacc parser.
$ make
ocamlyacc rpcalc.mly
ocamlc -c rpcalc.mli
ocamllex lexer.mll
15 states, 304 transitions, table size 1306 bytes
ocamlc -c lexer.ml
ocamlc -c rpcalc.ml
ocamlc -c main.ml
ocamlc -o rpcalc lexer.cmo rpcalc.cmo main.cmo
rm rpcalc.mli lexer.ml rpcalc.ml
# List files again.
$ ls
./    .depend    lexer.cmo    main.cmi      main.ml     rpcalc.cmi rpcalc.mly
../ Makefile lexer.cmi        lexer.mll     main.cmo    rpcalc*      rpcalc.cmo
```

Datoteka rpcalc vsebuje strojno kodo, ki jo lahko izvajamo. Poglejmo si še kratko sejo s programom rpcalc.

```
$ rpcalc
4 9 +
  13
3 7 + 3 4 5 *+-
  -13
3 7 + 3 4 5 * + - n Note the unary minus, n
  13
5 6 / 4 n +
  -3.166666667
3 4 ^    Exponentiation
  81
^D    End-of-file indicator
$
```

LITERATURA

1. SooHyoung Oh, Ocamlyacc Tutorial, 2004.
2. SooHyoung Oh, Ocamllex Tutorial, 2004.
3. C.Donnelly, R.Stallman, Bison, The Yacc-compatible Parser Generator, 2012.
4. The Lex & Yacc Page, http://dinosaur.compilertools.net/, 2012.