Analisi sintattica e lessicale in Haskell

I tool Alex e Happy

Analisi lessicale e sintattica

In precedenza abbiamo parlato del front end dei compilatori

- analisi lessicale
  - divido il codice del programma in una sequenza di parole, lessemi
  - l'insieme di lessemi è diviso in classi (identificatori, parole chiave, ...)
  - ogni classe è descritta da un'espressione regolare
  - Lex, Flex sono strumenti per costruire scanner, in C
- analisi sintattica
  - dalla sequenza di lessemi, costruisco un albero che rappresenta la struttura sintattica del programma
  - la struttura è definita da una serie di categorie sintattiche (espressioni, comandi, dichiarazioni, ...)
  - ogni categoria è definita da un non-terminale in una grammatica libera
  - Yacc, Bison sono strumenti per costruire parser, in C
- analisi semantica (principalmente type checking)

Alex e Happy

- Lex e Flex producono scanner
  - a partire da un insieme di regole
    - espressioni regolari e azioni corrispondenti
  - dichiarazioni ausiliarie
    - grammaticali
    - codice C
- Yacc e Bison, similmente, producono parser
- Alex e Happy, strumenti analoghi a Lex e Yacc, ma che producono codice Haskell
- concettualmente più semplici e puliti
- presentati abbastanza in dettaglio
- parte del corso relativa agli aspetti di compilazione del codice

Alex

Genera uno scanner

- stringa, divisa in lessemi, ogni lessema trasformato in un token

Parti principali del codice Alex:

- definizione del tipo dei token
- regole
  - espressione regolare
  - funzione che trasforma la stringa associata in token

Parti ausiliarie

- codice aggiuntivo
- definizioni ausiliarie

Alex - struttura del sorgente

```
{   -- codice Haskell da mettere in testa
module Main (main) where
}
%wrapper "basic"          -- definisce il tipo di scanner

$digit = 0-9              -- dichiarazioni ausiliarie
$alpha = [a-zA-Z]         -- insiemi di caratteri
@num   = $digit+          -- espressioni regolari

tokens :-                 -- nome arbitrario e separatore
                                  -- regole
  let             { \s -> TokenLet }    -- forma: regexp  azione
  $white+         ;                     -- azione vuota
  @num            { \s -> TokenInt (read s) }
  [\=\+\-\*]      { \s -> TokenSym (head s) }
  ...             -- ogni azione dello stesso tipo :: String -> T
```

Alex - struttura del sorgente - continua

```
{                       -- codice Haskell
data Token =            -- definisco il tipo dei token prodotti
    TokenLet            |
    TokenIn             |
    TokenSym Char       |
    TokenVar String     |
    TokenInt Int
    deriving (Eq,Show)

main = do               -- alexScanToken :: String -> [Token]
  s <- getContents
  print (alexScanTokens s)

-- lexer = alexScanToken
}
```

File Alex

- deve avere il suffisso file.x
- compilato genera file.hs

Struttura:

- inizio, codice Haskell da inserire in testa al file (Haskell)
  inserire nome modulo ed eventuali import
  - meglio non definire funzioni, possono creare conflitti
- `%wrapper` varie opzioni sul codice prodotto
- dichiarazioni ausiliarie macro, definisco
  - insiemi di caratteri
  - espressioni regolari
- segue nome mnemonico e separatore `:-`
- regole
- coda, codice Haskell da inserire in coda.

Regole Alex

- nella forma:
  - espressione regolare
  - codice associato
- codici associati tutti dello stesso tipo
  - tipo `String -> type` nel caso `%wrapper "basic"`
- se più espressioni regolari riconoscono la stringa di input
  - scelgo quella che riconosce più caratteri
  - a parità di lunghezza, scelgo la prima nella lista

Espressioni regolari di Alex

Simili a Lex, ma con alcune differenze

- separo:
  - espressioni regolari generiche
    - uso @ident come nomi per reg-expre
  - insiemi di caratteri, particolari espressioni regolari che generano singoli caratteri
    - uso $ident come nomi per insiemi

Sintassi per gli insiemi in Alex

- char – singolo carattere, se speciale preceduto da `\` caratteri non stampabili `\a, \b, \f, \n, \r, \t, \v`
  codifica Unicode `\x0A` equivalente a `\n`
- $ident – nome per un set di caratteri, definito in precedenza
- char-char – range di caratteri, tutti i caratteri nell'intervallo.
- set0 # set1 – differenza tra insiemi
- [set0set1set2] – unioni di insiemi, senza spazi
- ~set o [^set0set1set2] – complemento di insiemi

Macro

- . – tutti i caratteri escluso `\n` newline
- $printable – tutti i caratteri stampabili
- $white – tutti gli spazi `[\ \t\n\f\v\r]`.

Sintassi per le espressioni regolari

- set – un insieme
- @foo – nome per espressione regolare, definito in precedenza
- “sdf” – singola stringa
- (reg) – posso usare parentesi, per disambiguare
- reg1 reg2 – concatenazione
- reg1 | reg2 – unione
- reg* – zero o più ripetizioni
- reg+ – una o più ripetizioni
- reg? – zero o un’istanza
- reg{n} – n ripetizioni
- reg{n,} – n o più ripetizioni
- reg{n,m} – tra n e m ripetizioni

Opzioni

- `basic`
  - più semplice
  - fornisce funzione `alexScanTokens :: String -> [token]`
  - tutte le azioni devono avere tipo `String -> token`
  - per un qualche tipo token
- `posn`
  - rispetto a `basic` fornisce informazioni sulla posizione, per debugging, linguaggi sensibili all'indentazione
  - usa il tipo

```
data AlexPosn = AlexPn !Int  -- absolute character offset
                       !Int  -- line number
                       !Int  -- column number
```

  - tutte le azioni devono avere tipo `AlexPosn -> String -> Token`
  - l'argomento `AlexPosn` è relativo al primo carattere della stringa riconosciuta
  - fornisce funzione `alexScanTokens :: String -> [Token]`

Esempio posn

```
...
%wrapper "posn"
...
  "--".*                    ;
  let                       { \p s -> Let p}
  in                        { \p s -> In p}
  $digit+                   { \p s -> Int p (read s) }
...

data Token =
    Let AlexPosn            |
    Int AlexPosn Int        |
    ...
...
  print (alexScanTokens s)
```

Esercizi

Da portarne due all'esame (vedi pagina web)

- riconoscere i numeri romani
- analisi grammaticale
- riconoscere sequenze di numeri in vari formati

Altro:

- riconoscere e separare nomi di file di testo e file grafici; per ogni file generare un token con
  - tipo file
  - nome
  - estensione

Ulteriore opzione `%wrapper monad`

- Opzione per far funzionare in interleaving scanner - parser
  - il parser chiede un token alla volta allo scanner
  - in base al contesto
    - modifica il comportamento dello scanner
    - interviene sull'input
    - gestisce errori
- cambia la funzione con cui invocare lo scanner
  - in precedenza `alexScanToken :: String -> [token]`
  - ora `alexMonadScan :: Alex token` dove

```
newtype Alex a = Alex { unAlex :: AlexState
                                  -> Either String (AlexState,
```

  - dove `AlexState` descrive l'input corrente dello scanner
    - parte di testo che deve essere ancora esaminata
    - informazioni ausiliarie

Idea

- `alexMonadScan` una funzione che
  - data la stringa di input attuale
  - restituisce il primo token e la stringa di input
  - in caso di errori restituisce una stringa
- `Alex` è una monade, simile alla monade di stato
  - posso usare gli operatori sulle monadi

Happy

Strumento per costruire parser, in codice Haskell, simile a Yacc.

- interagisce con lo scanner,
  da sequenza di token, ad albero di derivazione (valore)
- si definiscono delle regole che generano una grammatica
- associa ad ogni regola una funzione che
  - a partire dall'albero di derivazione (valori) delle componenti
  - definisce l'albero di derivazione (valore) dell'espressione composta
- similitudini con la funzione `read`,
  (da una stringa di simboli ricostruisce il dato)
  - `read` algoritmo semplice, simile a un automa a pila, bottom-up, non deterministico,
  - `Happy` si basa sugli automi a pila, bottom-up, deterministico, LALR

Happy

- Strutturo la sequenza di token in un albero
  - vedo la sequenza di token come un'espressione
  - formata da un certo insieme di sottoespressioni
  - che a loro volta formate da sottoespressioni
  - fino ai token base
- Associo ad ogni espressione un valore

Definizioni principali neI file Happy

- la grammatica che guida il riconoscimento
- i tipi dei valori da associare alle varie sottoespressioni
- funzioni che per ogni produzione, calcolano
  il valore associato all'espressione a partire dai valori associati alle componenti

Esempio

- Parser per una grammatica delle espressioni `exp`.
  - interi,
  - variabili
  - operatori aritmetici `+, -, *, /,`
  - costrutto `let var = exp in exp`.
- Il parser struttura la sequenza di lessemi in un albero di espressioni e sottoespressioni
  associata ad ogni sottoespressione un valore (albero) corrispondente

Struttura del file Happy

- Intestazione - codice da inserire in testa al programma
  - dichiarazione di modulo

```
{
module Main (main) where
import Data.Char
--  import ExpressionToken
}
```

- Sequenza di dichiarazioni

```
%name calc
%tokentype { Token }
%error { parseError }
```

- definisco il nome della funzione di parsing generata
- il tipo dei Token in input `calc :: [Token] -> T`
- definisco il nome della funzione da chiamare in caso di errore

Struttura del file Happy

- lista dei terminali,
  - scopo: scrivere le regole usando la sintassi di Yacc
  - definisco la lista dei terminali della grammatica Happy
  - definisco la loro associazione con token Haskell
    - sinistra: nome usato nelle regole Happy
    - destra: token Haskell, usato nel codice
  - $$ placeholder per il valore associato dal costruttore
  - definizione del tipo token successiva

```
%token
      let             { TokenLet }
      in              { TokenIn }
      int             { TokenInt $$ }
      var             { TokenVar $$ }
      '='             { TokenEq }
      '+'             { TokenPlus }
      '-'             { TokenMinus }
      '*'             { TokenTimes }
      '/'             { TokenDiv }
```

Struttura del file Happy

- Separatore

%%

- Regole della grammatica
  - definisco come espandere (ridurre a) ogni singolo non-terminale
  - più alternative
  - l'azione definisce come costruire l'espressione associata al non-terminale a partire dai termini associati alle componenti.
  - il parser cerca di ridurre l'input a `Exp`,
    il non-terminale descritto nella prima regola, il simbolo iniziale
    restituisce il valore corrispondente.

Struttura del file Happy

```
Exp   : let var '=' Exp in Exp   { Let $2 $4 $6 }
      | Exp1                     { Exp1 $1 }

Exp1  : Exp1 '+' Term            { Plus $1 $3 }
      | Exp1 '-' Term            { Minus $1 $3 }
      | Term                     { Term $1 }

Term  : Term '*' Factor          { Times $1 $3 }
      | Term '/' Factor          { Div $1 $3 }
      | Factor                   { Factor $1 }

Factor
      : int                      { Int $1 }
      | var                      { Var $1 }
      | '(' Exp ')'              { Brack $2 }
```

Struttura del file Happy

- codice ausiliario.
  - definisco la funzione per gestire gli errori

```
{
parseError :: [Token] -> a
parseError _ = error "Parse error"
```

- definisco i tipi dei termini generati dal parser, per i vari non-terminali

```
data Exp
      = Let String Exp Exp
      | Exp1 Exp1
      deriving Show
```

segue

```
data Exp1
      = Plus Exp1 Term
      | Minus Exp1 Term
      | Term Term
      deriving Show

data Term
      = Times Term Factor
      | Div Term Factor
      | Factor Factor
      deriving Show

data Factor
      = Int Int
      | Var String
      | Brack Exp
      deriving Show
```

Struttura del file Happy

- il tipo associato ai token

```
data Token
      = TokenLet
      | TokenIn
      | TokenInt Int
      | TokenVar String
      | TokenEq
      | TokenPlus
      | TokenMinus
      | TokenTimes
      | TokenDiv
      | TokenOB
      | TokenCB
 deriving Show
```

Funzione principale

```
main = do inputString <- getContents
          print ( calc (lexer inputString)
}
```

- `calc :: [Token] -> Exp` è generata da Happy
  - `%name calc`
- `lexer :: String -> [Token]` è una funzione scanner
  - generata tramite Alex
    - importo il modulo relativo, e funzione associata `alexScanTokens`

      ```
      lexer = alexScanTokens
      ```
  - nei casi semplici può essere definita in Haskell, internamente

Definizione diretta di lexer

```
lexer :: String -> [Token]
lexer [] = []
lexer (c:cs)
      | isSpace c = lexer cs
      | isAlpha c = lexVar (c:cs)
      | isDigit c = lexNum (c:cs)
lexer ('=':cs) = TokenEq : lexer cs
lexer ('+':cs) = TokenPlus : lexer cs
lexer ('-':cs) = TokenMinus : lexer cs
lexer ('*':cs) = TokenTimes : lexer cs
lexer ('/':cs) = TokenDiv : lexer cs
lexer ('(':cs) = TokenOB : lexer cs
lexer (')':cs) = TokenCB : lexer cs
```

segue

```
lexNum cs = TokenInt (read num) : lexer rest
      where (num,rest) = span isDigit cs

lexVar cs =
   case span isAlpha cs of
      ("let",rest) -> TokenLet : lexer rest
      ("in",rest)  -> TokenIn : lexer rest
      (var,rest)   -> TokenVar var : lexer rest
```

- creare file `example.y` (stesso suffisso di Yacc)
- comando `happy example.y` genera file `example.hs`
- commando `happy example.y -i` produce un file `example.info` contenente informazioni dettagliate sul parser.

Interazione con Alex

- File Alex con nome `ModuloAlex` contiene

```
{
module ModuloAlex (Token(..), lexer)
  where
}
```

- File Happy contiene

```
{
module Main (main) where
import ModuloAlex
}
```

- È sufficiente compilare il file generato da Happy, link automatico, se il nome file Alex coincide con il nome modulo

Valutare la stringa di ingresso

- Al posto del tipo espressione, il `calc` può valutare e restituire un intero

```
Term   : Term '*' Factor          { $1 * $3 }
       | Term '/' Factor          { $1 / $3 }
       | Factor                   { $1 }
```

- variabili e costrutto `let` forzano una valutazione più complessa,
  - devo gestire un ambiente, `env` implementato come lista `[(String, Int)]`, (nome variabile, intero associato)
  - le regole associano ai non terminali una funzione `env -> Int` si sfrutta la funzionalità di Haskell

Definizioni

```
Exp   : let var '=' Exp in Exp  { \p -> $6 (($2,$4 p):p) }
      | Exp1                    { $1 }

Exp1  : Exp1 '+' Term           { \p -> $1 p + $3 p }
      | Exp1 '-' Term           { \p -> $1 p - $3 p }
      | Term                    { $1 }

Term  : Term '*' Factor         { \p -> $1 p * $3 p }
      | Term '/' Factor         { \p -> $1 p `div` $3 p }
      | Factor                  { $1 }
```

segue

```
Factor
      : int         { \p -> $1 }
      | var         { \p -> case lookup $1 p of
                              Nothing -> error "no var"
                              Just i  -> i }
      | '(' Exp ')'     { $2 }
```

Parsing sequences

Caso tipico, sequenza di comandi

- left recursion, più efficiente, inverte la sequenza

```
prods : prod                    { [$1] }
      | prods prod              { $2 : $1 }
```

- right recursion, usa più spazio

```
prods : prod                    { [$1] }
      | prod prods              { $1 : $2 }
```

Definizione delle precedenze

- Posso usare grammatiche ambigue
- togliere l'ambiguità definendo un ordine di precedenza tra operatori

```
%right in
%nonassoc '>' '<'
%left '+' '-'
%left '*' '/'
%left NEG
%%
Exp   : let var '=' Exp in Exp  { Let $2 $4 $6 }
      | Exp '+' Exp             { Plus $1 $3 }
      | Exp '-' Exp             { Minus $1 $3 }
      | Exp '*' Exp             { Times $1 $3 }
      | Exp '/' Exp             { Div $1 $3 }
      | '(' Exp ')'             { $2 }
      | '-' Exp %prec NEG       { Negate $2 }
      | int                     { Int $1 }
      | var                     { Var $1 }
```

Esercizi

Due esercizi da portare all'esame (vedi pagina web)

Altro:

- da una stringa di input rappresentante dei numeri floating point (secondo il formato standard, definito in seguito), determinare il valore associato ciascuna sottosequenza, ovvero trasformare sequenze di caratteri in numeri flaoting point.

```
floatnumber:    pointfloat | exponentfloat
pointfloat:     [intpart] fraction | intpart ["."]
exponentfloat:  (nonzerodigit digit* | pointfloat) exponent
intpart:        nonzerodigit digit* | "0"
fraction:       "." digit+
exponent:       ("e"|"E") ["+"|"-"] digit+
```