Vademecum
Linguaggi e Compilatori

Andrea Mansi UNIUD – 2021

Contenuti:

Haskell
BNFC
Alex & Happy
FLex & Bison

Indice

Introduzione

Lo scopo di questo documento è fornire un breve e sintetico sommario delle principali caratteristiche del linguaggio di programmazione Haskell. Tale vademecum è pensato come supporto all'attività di programmazione per gli studenti dei corsi di *Linguaggi & Compilatori* e/o *Linguaggi di Programmazione*.

1. Programmare in Haskell

Un utile strumento per iniziare a programmare in haskell è **Stack**. Stack è una piattaforma cross-platform pensata per semplificare lo sviluppo di progetti in haskell. Stack utilizza la libreria Cabal.

Stack è ottenibile al link: https://www.haskell.org/downloads/#stack

Haskell permette due modalità di esecuzione del codice: tramite interprete o tramite compilazione. Una volta installato stack, sarà possibile interagire con il proprio codice sorgente tramite interprete interattivo nel seguente modo:

```
<<aprire shell cmd e posizionarsi nella directory del sorgente>>

stack ghci                      -> avvio dell'interprete
:l <<nome_file.s>>              -> caricamento del sorgente
<<nome_funzione>>               -> per eseguire una funzione del nostro codice
```

Per compilare il codice e produrre un file .exe sarà sufficiente eseguire il seguente comando:

```
<<aprire shell cmd e posizionarsi nella directory del sorgente>>

stack ghc <filename.hs>         -> avvio compilazione
```

In questo caso stack si occuperà di gestire le varie dipendenze. Si noti che il sorgente richiede che la funzione main sia definita per poter essere compilato in .exe.

2. Haskell in a nutshell

Haskell è un linguaggio "typeful": i tipi sono pervasivi. Il sistema di tipi di Haskell è abbastanza potente da permetterci di evitare di includere le firme di tipo (quasi sempre).

Il sistema di tipi deduce (tramite inferenza) i tipi corretti al posto nostro. Tuttavia, il posizionamento delle firme di tipo è una buona idea, includerle è un modo efficace di produrre documentazione e aiuta a portare alla luce gli errori di programmazione.

Haskell è un linguaggio puramente funzionale. Tutte le computazioni vengono svolte tramite la valutazione di espressioni (termini sintattici) per produrre valori.

Ogni valore ha un tipo associato. I tipi possono essere espressi tramite espressioni di tipo che sono termini sintattici che denotano il tipo dei valori, ad esempio: `Int, Char, Bool` (tipi atomici). `Int->Char` (tipi funzione). `[Int]` (tipi strutturati). Possiamo eseguire il typing con due doppi punti consecutivi.

Ad esempio con `5 :: Int` stiamo dicendo che il valore `5` è di tipo Int.

In Haskell le funzioni sono tipicamente definite da una serie di equazioni che formano una dichiarazione di funzione.

Ad esempio:

```
inc x = x+1
```

oppure:

```
fact 0 = 1
fact n = n * fact (n-1)
```

Si noti che la sintassi di Haskell richiede che gli identificatori di tipo inizino con la lettera maiuscola e gli identificatori di funzioni con la lettera minuscola. La firma di tipo (type signature) della precedente funzione può essere espressa come:

```
fact :: Int -> Int
```

Polimorfismo di tipo

Haskell incorpora anche i tipi polimorfi parametrici, che sostanzialmente descrivono famiglie di tipi. Ad esempio `[a]` rappresenta la famiglia di liste di qualsiasi tipo. Identificatori come a vengono chiamati variabili di tipo.

Un esempio di funzione polimorfa:

```
head :: [a] -> a
head (x:xs) = x
```

Le parti sinistre delle equazioni contengono pattern come ad esempio `(x:xs)`. In un applicazione della funzione questi pattern vengono confrontati con i parametri attuali e in caso di match viene eseguita la parte destra dell'equazione.

Notazione curried

Si considerino due definizioni della funzione che somma due argomenti:

```
add :: Int -> ( Int -> Int )
add x y = x + y

add' :: (Int , Int) -> Int
add' (x,y) = x + y
```

La prima è un esempio di funzione curried. Applicare `add` ad un argomento porta a una nuova funzione che viene applicata al secondo argomento. L'applicazione delle funzioni è associativa a sinistra mentre l'operatore `->` associa a destra.

Lambda espressioni

In Haskell le funzioni sono direttamente esprimibili tramite la scrittura di espressione il cui valore è una funzione. Questo è possibile grazie alle lambda espressioni. Un esempio:

```
\x->x+3
```

Operatori infissi

Gli operatori infissi sono semplicemente funzioni, e possono essere definiti usando equazioni. Ad esempio:

```
[] ++ xs = xs
(x:xs) ++ ys = x : (xs ++ ys)
```

Gli operatori infissi devono essere costituiti solo da simboli.L'applicazione parziale di un operatore infisso viene detta sezione (section).

```
(x+) = \y->x+y
(+y) = \x->x+y
(+) = \x y->x+y
```

Una sezione viene utilizzata per passare una primitiva come argomento a una funzione, come **`map (+1)`** e per dichiarare il tipo di una primitiva come nell'esempio:

```
(++) :: [a] -> [a] -> [a]
```

Possiamo costringere (coercion) una funziona binaria in operatore infisso incapsulando l'identificatore tra backquotes, ad esempio:

```
x `add` y = add x y
```

Sequenze aritmetiche

Haskell possiede una sintassi specifica per le sequenze aritmetiche. Il miglior modo di illustrarle è tramite esempi:

```
[1..10] =) [1,2,3,4,5,6,7,8,9,10]
[1,3..10] =) [1,3,5,7,9]
[1,3..] =) [1,3,5,7, : : : (seq. infinita)
```

List comprehensions

Con List comprehensions ci si riferisce allo zucchero sintattico come da esempio:

```
[ f x | x <- xs ]
[ (x,y) | x <- xs , y <- ys ]
[ (x,y) | x <- xs , y <- ys , x+y >3 ]
```

La frase `x <- xs` è chiamata generatore. Oltre ai generatori, sono consentite espressioni booleane chiamate guardie, utilizzate per impostare dei vincoli sugli elementi generati.

Le stringe sono un altro esempio di zucchero sintattico in Haskell (per i tipi predefiniti). La stringa

```
"Ciao"
```

non è altro che zucchero sintattico della lista `['C','i','a','o']`. Quindi String non è altro che un sinonimo di tipo predefinito per `[Char]`.

Tipi definiti dall'utente

Haskell permette all'utente di definire i propri tipi monomorfi e polimorfi. Esempi:

```
data Bool = False | True
data Color = Red | Green | Blue | Indigo
data Pixel = Grey Int | RGB Int Int Int
data Point a = TwoD a a | ThreeD a a a
data BST a = Void | Node a (BST a) (BST a)
```

"`Point`" è un costruttore di tipo unario, dal tipo **`t`** costruisce un nuovo tipo **`Point t`.**

"TwoD" è un costruttore binario curried. **`TwoD :: a -> a -> Point a.`**

È importante distinguere tra applicare un costruttore di dato per ottenere un valore e applicare un costruttore di tipo per ottenere un tipo. Il primo avviene a run-time ed è come Haskell esegue le computazioni. La seconda succede in tempo di compilazione per assicurare la "type-safety".

I costruttori di tipo e i costruttori di dati sono in namespace diversi, questo permette l'uso dello stesso nome per dichiarare sia costruttore di tipo che di dato. Il contesto sintattico permette di determinare quale dei due nomi utilizzare. Nonostante come pratica sembri confusionaria, questo permette di collegare (ai fini della leggibilità del codice) il costruttore di tipo e quello dei dati.

Field names

I costruttori nella dichiarazione di dati possono essere dichiarati con dei campi (field names). Questi campi identificano le componenti del costruttore tramite nome e non tramite posizione. Esempio:

Invece di:

```
data BST a = Void | Node a ( BST a) (BST a)
```

Possiamo usare:

```
data BST a = Void | Node {
value :: a,
left, right :: BST a
}
```

Questa dichiarazione definisce anche i nomi `value`, `left` e `right`. Questi campi possono essere usati come selettori di funzione per estrarre un componente dalla struttura.

Quindi "`value tree`" valuta la radice di valore "`tree`". È tuttavia importante tenersi nota del fatto che le etichette di campo condividono il namespace di top-level con le variabili ordinarie e le classi di metodi e di conseguenza non possono essere usati più nomi di campo nello stesso scope tra più tipi di dato.

Nello stesso tipo di dato può essere usato in più di un costruttore a patto che il suo tipo coincida sempre. Le etichette di campo si possono utilizzare anche per costruire nuovi valori, come:

```
Node{ left =Void , value =3, right = Void }
```

La definizione di funzioni usa una sintassi simile, ad esempio:

```
size Node { right =r, left =l} = 1 + size l + size r
```

o come:

```
size Node {..} = 1 + size left + size right
```

Esiste un costrutto "update" che use i valori dei campi di un valore esistente per riempire i componenti di una nuova struttura.

Sinonimi di tipo

Haskell fornisce un modo di definire sinonimi di tipo, ad esempio nomi per tipi usati comunemente. I sinonimi di tipo si creano usando una dichiarazione di tipo come:

```
type String = [ Char ]
type Person a = (Name , Address ,a)
type Name = String
type Address = Maybe String
data Maybe a = Nothing | Just a
```

I sinonimi di tipo non definiscono nuovi tipo, ma semplicemente danno un nuovo nome a tipi esistenti. Ad esempio:

Il tipo `Person Int -> Name`

è precisamente equivalente a:

```
(String,Address,Int) -> String
```

come anche a:

```
([Char],Maybe [Char],Int) -> [Char].
```

Valutazione dei parametri attuali

Partendo da un esempio di esecuzione, ci accorgiamo subito che definendo `const x = 1` ed eseguendo `const (fact 1000000)` haskell restituisce subito 1. Questo è dovuto al metodo di valutazione di Haskell, detto "lazy", ovvero valutare i parametri attuali solo quando è realmente necessario conoscerne il valore. Questo permette la definizione e l'utilizzo di dati di lunghezza infiniti, come ad esempio la lista di Fibonacci.

Semantica del pattern matching

Partiamo elencando alcune caratteristiche dei pattern: è permesso il nesting di pattern multipli, a profondità arbitraria. In tutti i pattern di una equazione i nomi delle variabili devono essere unici. Il meccanismo di valutazione di Haskell è basato sul pattern matching, il processo che: dato un pattern t da matchare con un'espressione e, prova a matchare e con la struttura di t e lega la variabile di t a una valida sotto-espressione di e.

Il pattern matching può fallire, avere successo o divergere. Un match di successo lega il parametro formale nel pattern. La divergenza accade quando un valore necessario dal pattern diverge (ad esempio dato infinito).

Dato una funzione definita per pattern, il processo di esecuzione procede top-down e left-to-right. Haskell prova ad eseguire il match tutti i pattern con tutti i parametri, se il match avviene si valuta la parte destra dell'espressione del pattern. Se tutte le equazioni falliscono nel match, viene restituito un codice di errore.

As-patterns e wild-cards

A volte è conveniente nominare un pattern per l'uso nel lato destro di una equazione. Ad esempio, una funzione che duplica il primo elemento in una lista può essere scritta come:

```
f (x:xs) = x:x:xs
```

Per migliorare la leggibilità, possiamo preferire di scrivere `x:xs` una sola volta, risultato ottenibile usando un as-pattern come segue:

```
f ys@(x:xs) = x:ys
```

Un'altra situazione comune è il matching su un valore che non è di nostro interesse. Ad esempio, nella definizione precedente non usiamo la variabile xs nel lato destro, la definizione può essere scritta come:

```
f ys@(x:_) = x:ys
```

Ogni wild-card matcha indipendentemente nulla, ma in contrasto a una variabile ciascuna non lega con niente, quindi ne è consentito l'uso multiplo.

Equazioni con guardie

I pattern top-level possono avere una sequenza di guardie booleane, come:

```
sign x | x > 0 = 1
       | x == 0 = 0
       | otherwise = -1
```

le guardie sono valutate top-down e la prima a valutare True risulta nel successo di un match.

Case expressions

Le case expressions sono una generalizzazione delle dichiarazioni "case" dei linguaggi imperativi in espressioni, dove i selettori sono pattern. Un esempio:

```
foo (x:_) = 2 + case x of
       (n,0) | n >=0        -> n
             | otherwise  -> -n
      (0,m) -> 0
      (n,m)  | n==m        -> n*(-n)
             | otherwise  -> n*m
      _ -> 0
```

I tipi della parte destra di un'espressione case devono essere tutti compatibili con un tipo principale in comune.

Let expressions

Le espressioni let sono espressioni nella forma `let <dichiarazioni> in e`, dove nello scope di e aggiungiamo le dichiarazioni <dichiarazioni>. Ad esempio:

```
g z = 3+
      let y     = a*b
            f x = (x+y)/y
      in f c + f d
```

Clausole Where

Sono una parte opzionale delle dichiarazioni di funzioni e delle espressioni case. Le dichiarazioni locali delle clausole where sono visibile nel pattern e nel corpo delle equazioni. Ad esempio:

```
f x y | y>z  = y-z
      | y==z = ...
      | y<z  = ...
where z = x*x
```

Layout

Haskell usa una sintassi bidimensionale chiamata layout che essenzialmente fa affidamento nelle dichiarazioni allineate in colonne. Il layout è una scorciatoia per un raggruppamento meccanico esplicito, usando `{ ;` e `}`, come in:

```
let { y = a*b ; f x = (x+y)/y } in f c + f d
```

Il lexer di Haskell converte i layout in sintassi esplicita, seguendo le seguenti regole:

- La colonna di partenza per le dichiarazioni locali è determinata dal prossimo carattere che segue una delle keyword where, let o of;
- Tale colonna iniziale deve essere più a destra della colonna iniziale associata alla clausola immediatamente circostante;
- La "risoluzione" di una dichiarazione avviene quando qualcosa appare in o a sinistra della colonna iniziale della dichiarazione locale.

Classi di tipo

Le classi di tipo forniscono un modo strutturato per controllare l'overloading. Esse dichiarano il nome e il tipo delle operazioni overloaded, anche chiamate metodi. Ad esempio, definiamo una classe contenente un operatore di eguaglianza:

```
class Eq a where
(==) :: a -> a -> Bool
(/=) :: a -> a -> Bool
```

Questa dichiarazione può essere letta come "u tipo a è una istanza della classe Eq in cui ci sono due operatori (overloaded) == e /=, del tipo appropriato, definito in essa".

Contesti

`Eq a` non è un'espressione di tipo, ma piuttosto è un'espressione con un vincolo sul tipo, ed è chiamato contesto. I contesti sono posizionati davanti alle espressioni. I vincoli si propagano dall'utilizzo dei metodi al tipo di funzioni che li utilizzano.

Le classi di metodi possono avere anche vincoli di classe aggiuntivi in ogni tipo di variabile eccetto quella definita dalla classe stessa. Ad esempio:

```
class C a where
m :: Show b => a -> b
...
```

quindi il tipo è `m :: (C a,Show b) => a -> b`

I metodi di classe sono trattai sempre come dichiarazioni top-level. Condividono lo stesso namespace delle varaibili ordinarie quindi un nome non può essere usato per denotare un metodo e una variabile o metodi di differenti classi.

Dichiarazione di istanze

Una dichiarazione di istanza specifica che un tipo è una istanza di una classe, e il comportamento dei metodi di ciascun tipo, ad esempio:

```
instance Eq Integer where
      x == y = integerEq x y
      x /= y = not ( integerEq x y)
```

Le istanze possono anche avere un contesto. Ad esempio:

```
instance Eq a => Eq (BST a) where
      x == y = treeEq x y
```

Dichiarazione dei metodi di default

All'interno della dichiarazione di una classe possono essere dichiarati dei metodi di default, ad esempio:

```
class Eq a where
      (==) :: a -> a -> Bool
      (/=) :: a -> a -> Bool
      x /= y = not (x == y)
      x == y = not (x /= y)
```

Se un metodo per una particolare operazione è ommesso nella dichiarazione di istanza allora viene usato il metodo di default, dichiarato nella dichiarazione della classe (se esiste).

Ad esempio, per via dei metodi default di Eq, la seguente istanza è equivalente a quella precedente:

```
instance Eq Integer where
x == y = integerEq x y
```

È importante stare attenti alle dichiarazioni cicliche.

Ereditarietà

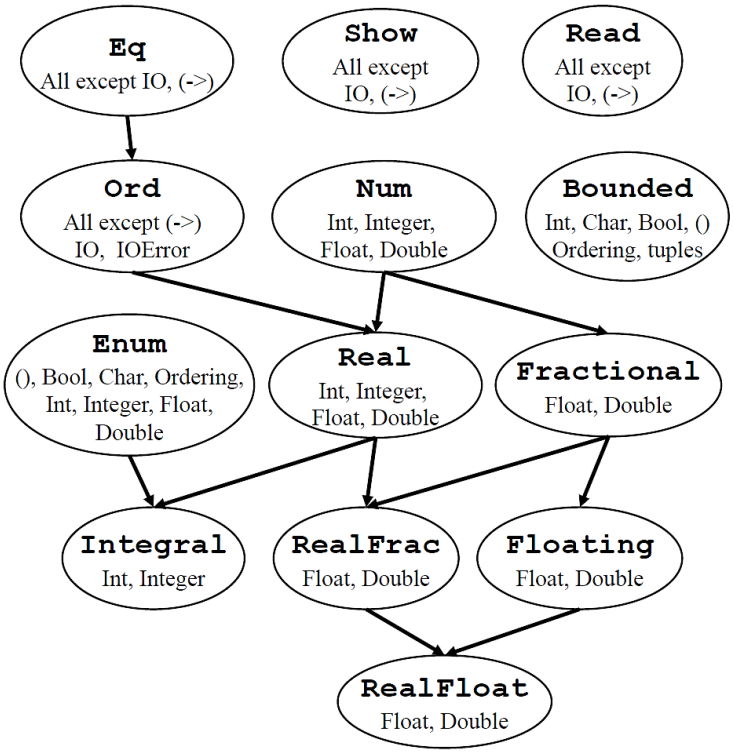
Haskell supporta l'estensione di classe, ovvero la dichiarazione di una classe che eredita tutte le operazioni di un'altra classe. Esempio:

```
class Eq a => Ord a where
      compare :: a -> a -> Ordering
      (<), ( <=), (>), ( >=) :: a -> a -> Bool
      max , min :: a -> a -> a
data Ordering = LT | EQ | GT
```

Haskell permette anche ereditarietà multipla, dato che le classi possono avere più di una superclasse. Esempio:

```
class (Eq a, Show a) => Class a where ...
```

Segue un grafo delle classi predefinite in Haskell:



Costrutto deriving

Le dichiarazioni di tipi possono automaticamente definire istanze di `Eq`, `Ord`, `Enum`, `Bounded`, `Show` e `Read` con il costrutto deriving. Ad esempio:

```
data BST a = Void | Node a ( BST a) (BST a)
deriving (Eq ,Ord)
```

Per ogni classe consentita nel deriving, Haskell specifica come le istanze sono definite.

- == è definito per eguaglianza strutturale
- I sette operatori di Ord sono definiti in modo da confrontare i loro argomenti lessicograficamente rispetto all'insieme di costruttori dato, con in primi costruttori nella dichiarazione del tipo di dati che contano come più piccoli di quelli successivi. Per esempio: `Void < Node e1 e2 e3`

Input/Output

Il sistema I/O in Haskell è puramente funzionale, ma ha tutto l'espressività che si può trovare nei linguaggi di programmazione convenzionali. Le azioni di I/O sono nettamente separate dal nucleo puramente funzionale del linguaggio. Le azioni di I/O vengono definite anziché richiamate. Valutare la definizione di un azione in realtà non provoca il verificarsi dell'azione. Ogni azione di I/O restituisce un valore. Il valore restituito è "taggato" con il tipo `IO`, distinguendo così le azioni da altri valori. Per esempio,

```
getChar :: IO Char
```

indica che `getChar`, quando invocato, esegue alcune azioni di I/O che restituisce un carattere.

Le azioni di I/O che non restituiscono valori interessanti usano il tipo di unità `()` come ad esempio:

```
putChar :: Char -> IO ()
```

La funzione `putChar` accetta un carattere come argomento ma non restituisce nulla di utile. Il tipo di unità `()` è simile a `void` in altre lingue.

Le azioni di I/O vengono sequenziate utilizzando la notazione `do` che introduce una sequenza di istruzioni che vengono eseguite in ordine. Una dichiarazione è:

- una azione di I/O, o
- un pattern bound al risultato di una azione usando `<-`, o
- un insieme di definizioni locali introdotte usando `let`, o
- una dichiarazione di ritorno `return :: a -> IO a`.

La notazione `do` usa un layout nello stesso modo che `let` e `where`. Le variabili definite da `<-` sono solo nello scope delle istruzioni seguenti (questo sembra contravvenire alla regola generale di scoping ma vedremo più avanti che questo non è il caso).

```
getLine :: IO String
getLine = do
      c <- getChar
      if c == ’\n’
            then return ""
            else do
                  cs <- getLine
                  return (c:cs)
```

Si noti il secondo `do` nella clausola else. Ciascuno introduce una singola catena di dichiarazioni. Qualsiasi costrutto intermedio, come if, deve usare un nuovo do per avviare ulteriori sequenze di azioni di I/O.

Le azioni di I/O sono valori ordinari in Haskell. Possono essere passati a funzioni, messi nelle strutture e usati come qualsiasi altro valore.

Possiamo combinare funzioni di ordine superiore per ottenere codice utile, come:

```
sequence_ . map putChar
```

richiamando che:

```
sequence_ :: [IO ()] -> IO ()
map :: (a->b) -> [a] -> [b]
putChar :: Char -> IO ()
(.) :: (b -> c) -> (a -> b) -> a -> c
```

e quindi il tipo di:

```
sequence_ . map putChar è [Char] -> IO () .
```

Cos’è? è una funziona che manda in output tutti i caratteri di una stringa (predefinita in Haskell con il nome di putStr).

Main

`main :: IO()` è l’entry point di un programma Haskell compilato, lanciato quando un binario haskell viene lanciato.

Files – System.IO library

Primitive per i file:

```
type FilePath = String

readFile :: FilePath -> IO String
writeFile :: FilePath -> String -> IO ()
appendFile :: FilePath -> String -> IO ()
openFile :: FilePath -> IOMode -> IO Handle

data IOMode = ReadMode
      | WriteMode
      | AppendMode
      | ReadWriteMode

stdin , stdout , stderr :: Handle

hClose :: Handle -> IO ()
hGetChar :: Handle -> IO Char
hGetLine :: Handle -> IO String
hGetContents :: Handle -> IO String
hPutChar :: Handle -> Char -> IO ()
hPutStr :: Handle -> String -> IO ()
hPutStrLn :: Handle -> String -> IO ()
```

Classe Show

I tipi della classe Show sono quei tipi che possono essere convertiti in stringhe di caratteri.

```
class Show a where
      show :: a -> String
      showsPrec :: Int -> a -> ShowS
      showList :: [a] -> ShowS
type ShowS = String -> String
shows = showsPrec 0
```

Istanze automatiche di tipi di dato definiti dall'utente possono essere definite tramite deriving. La serializzazione è essenziale per printare valori.

Type defaulting

Il type defaulting è introdotto per risolvere i problemi di ambiguità di tipo causati dalle classi di tipo. Ad esempio, l'espressione `e = show 4` è ambigua perché il letterale 4 è di tipo `Num a => a`. Nell'espressione e, il letterale 4 può essere `Int`, `Float` o qualsiasi altra istanza di `Num` e `Show`.

In ogni caso e può essere tipata correttamente in stringa ma il risultato non è uguale. Possiamo risolvere questo problema esplicitando il tipo come in `show(4::Int)`. Tuttavia, per i tipi numerici ci sono dei tipi default nel caso di ambiguità. Per `Num` il default è `Integer`, a per `Floating` il default è `Double`.

Moduli

Un programma Haskell consiste in una collezione di moduli. Ogni modulo definisce una collezione di valori, tipi di dato, sinonimi di tipo, classi, istanze e sono identificati da un nome. I nomi dei moduli possono essere alfanumerici e devono iniziare con la lettera maiuscola.Il namespace dei moduli è completamente "piatto".

Un modulo inizia con la keyword `module`, come:

```
module BinarySeachTrees (BST,insert,empty) where ...
```

Si noti dopo il nome del modulo la export list. Si noti che il nome di un tipo di dato nella export list non esporta i nomi dei suoi costruttori. Questo può essere fatto aggiungendo `(..)`, come `BST(..)`. Se la export list viene ommessa, tutti i nomi al top-level del modulo vengono esportati.

Nelle linee che seguono abbiamo la possibilità di una lista vuota di import.

```
import name (lista di nomi)
```

Se la import list (opzionale) è presente, allora vengono importati solo i nomi specificati. È possibile dichiarare con la parola `hiding(lista di nomi)` una lista di nomi da non importare.

Nomi qualificati

Haskell usa nomi qualificati per risolvere il problema di avere lo stesso nome legato a differenti entità importate da due moduli. Un nome qualificato è un prefisso di un modulo seguito dal carattere punto senza whitespace. Ad esempio:

```
BinarySearchTrees.insert e BinarySearchTrees.BST
```

Una dichiarazione di import può usare la keyword `qualified` per importare nomi solo in forma qualificata. Altrimenti entrambi i nomi qualificati e non vengono importati. Un `import` può contenere una clausola `as` per specificare un differente qualificatore più corto del nome del modulo da cui si importa. Ad esempio:

```
import qualified BinarySearchTrees as BST
```

a quindi `BST.Insert` e `BST.BST` possono essere usati.

Un import esplicito del modulo prelude sovrascrive l'import implicito di tutti i nomi del Prelude.

3. Syntax directed compilation techniques

BNFC

Il Converter BNF è uno strumento di costruzione di compiler (lexer e parser) front-end da una grammatica BNF. Originariamente scritto per generare codice Haskell, è stato aggiornato per generare codice in altri linguaggi come C++ o Java.

BNFC genera:

- un **lexer** (il sourcecode per uno strumento che genera il lexer);
- un **parser** (il sourcecode per uno strumento che genera il parser);
- un **albero di sintassi astratto**: definisce i data types per ogni non terminale usando le etichette come costruttori;
- un **serializzatore** (pretty-printer) di tale albero di sintassi astratto.

https://hackage.haskell.org/package/BNFC

Per installare il pacchetto con **Stack**:

```
stack install BNFC
```

Nel caso di Haskell, i sourcecode relativi al lexer e parser sono il codice sorgente per i due strumenti **Alex** e **Happy**. Quindi BNFC non crea direttamente il lexer e il parser, ma prepara l'input per questi due strumenti.

In questo capitolo vengono illustrate le principali caratteristiche della sintassi di questo tool/package.

Grammatica BNF

Una grammatica BNF è una sequenza di regole grammaticali nella forma:

```
label.      NonTerminal ::= (Terminal | NonTerminal)* ;
```

le etichette (`label`) e i non terminali (`NonTerminal`) sono identificatori composti da lettere, inizianti con una lettera maiuscola. I non terminali possono finire con un numero. L'etichetta è usata per costruire un albero di sintassi i cuoi sottoalberi sono dati dalle regole dei non terminali. I terminali (`Terminals`) sono stringhe.

Ad esempio:

```
AddOp. E ::= E "+" E1;
MulOp. E1 ::= E1 "*" E2;
IntVal. E2 ::= Integer;
Identifier. E2 ::= Ident;
```

Non-terminali predefiniti di base:

- **Integer** per **integers**
- **Double** per **numeri floating point**
- **Char** per **characters** (in single quotes)
- **String** per **strings** (in double quotes)
- **Ident** per **identifiers**

Questi non terminali, eccetto **Ident**, sono rappresentati dai corrispondenti tipi del linguaggio target.

Semantic dummies

Nella specifica delle regole, le etichette possono essere _ se abbiamo una produzione che nel suo lato destro ha terminali e solo una occorrenza di non terminali della parte sinistra:

```
_. E ::= E1;
_. E1 ::= E2;
_. E2 ::= "("E")";
```

Vediamo ora una serie di macro, zucchero sintattico per sintetizzare la scrittura di regole della grammatica.

Non-terminali indicizzati

I non-terminali indicizzati sono comunemente utilizzati per i livelli di precedenza. Un simbolo non-terminale che finisce con un indice intero è trattato come sinonimo di tipo del corrispondente simbolo non-indicizzato.

Coercizioni

Sono la scorciatoia per un gruppo di regole relativi alle precedenze, ad esempio:

```
coercions Exp 3;
```

è equivalente a:

```
_. Exp ::= Exp1;
_. Exp1 ::= Exp2;
_. Exp2 ::= Exp3;
_. Exp3 ::= "(" Exp ")";
```

Terminator

La macro “`terminator NT T;`” definisce regole appropriate per il simbolo `[NT]` per produrre una (anche vuota) sequenza di NT dove ciascun NT termina con T, ad esempio:

```
terminator Stm ";" ;
```

definisce:

```
[].  [Stm] ::= ;
(:). [Stm] ::= Stm ";" [Stm] ;
```

È possibile usare il qualificatore `nonempty` per scartare le sequenze vuote, come:

```
terminator nonempty Stm ";" ;
```

definisce:

```
(:[]). [Stm] ::= Stm ";" ;
(:).   [Stm] ::= Stm ";" [Stm] ;
```

T può essere specificato come stringa vuota “ ” così che `[NT]` sia una sequenza di NT.

Separator

La macro `separator` è simile a `terminator`, eccetto che il token di separazione non è attaccato all'ultimo elemento, quindi:

```
separator Stm ";" ;
```

significa

```
[].    [Stm] ::= ;
(:[]). [Stm] ::= Stm ;
(:).   [Stm] ::= Stm ";" [Stm] ;
```

dove:

```
separator nonempty Stm ";" ;
```

significa:

```
(:[]). [Stm] ::= Stm ;
(:).   [Stm] ::= Stm ";" [Stm] ;
```

Anche in questo caso possiamo specificare come terminatore la stringa vuota.

Rules

La macro `rules` è l'abbreviazione per un insieme di regole le cui etichette sono generate automaticamente, ad esempio:

```
rules Type ::= Type "[" "]" | "float" | "int" ;
```

è l'abbreviazione di:

```
Type1.Type ::= Type "[" "]" ;
Type_float. Type ::= "float" ;
Type_int.   Type ::= "int" ;
```

(Non vengono eseguiti controlli globali nella generazione delle etichette).

Commenti

La regola "`comment T ;`" istruisce il generatore di lexer a trattare il testo da T alla fine della linea come commento.

La regola "`comment T T' ;`" istruisce per multiple linee di commenti che iniziano con T e finiscono con T'.

Entrypoints

Il convertitore BNF genera, per impostazione predefinita, un parser per ogni non terminale nella grammatica. Il pragma degli entrypoints definisce quali sono i parser effettivamente generati. Per esempio:

```
entrypoints Stm, Exp;
```

In realtà questa funzionalità dipende dal generatore di parser utilizzato in base al linguaggio di destinazione scelto. Ad esempio, Happy supporta questa funzione mentre Bison no.

Happy

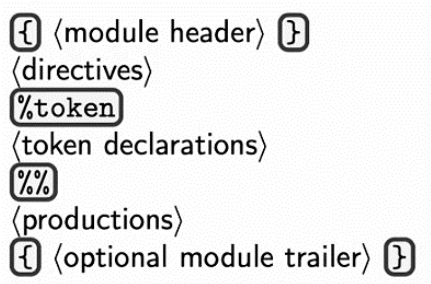
Happy è un generatore di parser per Haskell e viene utilizzato da BNFC per tale componente del front-end di un compilatore. Happy prende in input un file con una grammatica BNF e produce un modulo haskell contenente il parser per tale grammatica.

Happy genera parser per grammatiche LALR.

Happy viene installato automaticamente da BNFC ma è possibile installarlo manualmente tramite Stack con il comando:

```
stack install happy
```

Struttura del sorgente happy



Il `<module header>` è l'header di un modulo haskell. Questo modulo è escluso nel modulo generato.

In `<directives>` abbiamo obbligatoriamente le direttive `%tokentype{<type>}` che dichiarano il tipo dei token che il parser riceve in input.

La direttiva `%name <identifier> <non-terminal>` definisce il codice del parser per il simbolo iniziale `<non-terminale>` con il nome `<identifier>.`

La dichiarazione di un token consiste in:

- un simbolo S che verrà usato per riferirsi al token nel resto della grammatica.
- un pattern Haskell P che matcha il token:
  - un input che matcha P è un'occorrenza di S nella grammatica;
  - variabili del pattern P dovrebbero essere solo anonime, o…
  - …il simbolo $$ che rappresenta il valore del token (o il valore è il token stesso).

Nella sezione `<productions>` troviamo le produzioni della grammatica. Ogni produzione consiste di un simbolo non terminale NT seguito da `:`, seguito da un o più espansioni sulla destra separate da `|`. Ciascuna espansione ha del codice haskell da associare con la regola (messa tra graffe). Questo codice serve per la generazione della sintassi astratta. Il valore del simbolo i-esimo è referenziabile usando `$i`.

```
Exp : Exp '+' Exp1 { Plus $1 $3 }
    | Exp1         { $1 }
```

Come viene usata l'espressione specificata nella produzione: se il parser esegue una riduzione di una produzione `NT : a_1;...; a_n {expr}` e il valore generato per ciascun `a_i` è `v_i`, allora genera il valore dell'espressione `expr` dove ogni `$i` è sostituito da `v_i`.

(Considerando che haskell è lazy, la frase "genera il valore" sarebbe "definisce il valore").

Le produzioni possono essere tipate (raccomandato) come in:

```
Exp :: { Expression }
Exp : Exp '+' Exp1 { Plus $1 $3 }
    | Exp1 { $1 }
```

se il tipo è presente allora la testa della produzione è opzionale, come in:

```
Exp :: { Expression }
    : Exp '+' Exp1 { Plus $1 $3 }
    | Exp1 { $1 }
```

Produzioni vuote sono consentite: `A : { expr }` corrisponde alla produzione `A->epsilon`. È consigliato scrivere come `A : {-empty-} { expr }` per rendere l'epsilon evidente.

Uso delle precedenze

Happy permette di specificare precedenze di operatori per risolvere conflitti, usando le direttive: `%left`, `%right` o `%nonassoc` seguite da una lista di terminali. Esse dichiarano tutti i token come di sinistra, destra o non associativi.

Le precedenze di questi token rispetto agli altri token è stabilita dall'ordine delle direttive: prima significa minor precedenza, cioè l'operatore si lega meno strettamente.

Ad esempio:

```
% left      '||'
% left      '&&'
% left      '!'
% nonassoc  '==' '!=' '<' '<=' '>' '>='
% left      '+' '-'
% left      '*' '/' '%'
% right     '^'
```

La precedenza di una regola individuale può essere sovrascritta (overridden), usando le precedenze di contesto. Sono utili quando un token ha differente precedenza in base al contesto in cui si trova.

La direttiva `%prec` viene attaccata alla regola che sovrascrive la precedenza di default.

Un esempio tipico è il segno meno: ha precedenza alta quando usato come prefisso di negazione ma minore quando usato come simbolo unario di sottrazione. Mostriamo l'esempio:

Inventiamo un nuovo token NEG come placeholder. Il token NEG non ha bisogno di apparire in una direttiva `%token`.

```
%left '+' '-'
  ...
%left NEG
```

Attacchiamo la direttiva `%prec NEG` al prefisso della regola di negazione.

```
Expr : Expr '-' Expr { ... }
     | '-' Expr % prec NEG { ... }
```

Conflitti irrisolti

In alcuni casi in cui i conflitti non possono essere risolti Happy riporta un warning ma il codice del parser viene comunque generato.

- In caso di un conflitto **shift/reduce,** viene eseguito lo shift.
- Nel caso di un conflitto **reduce/reduce** viene eseguita l'azione con la regola definita prima nel sourcecode.

Ovviamente il risultato potrebbe non generare il risultato voluto.

Produzioni parametrizzate

Happy supporta le produzioni parametrizzate che forniscono una notazione conveniente per catturare pattern ricorrenti nelle grammatiche CF.

```
opt (p)
  : { Nothing }
  | p { Just $1 }

rev_list1 (p)
  : p { [$1] }
  | rev_list1 (p) p { $2 : $1 }
```

La produzione `opt(p)` è usata come componenti opzionali della grammatica. La produzione `rev_list1` è per il parsing di una lista di 1 o più occorrenze di `p` (produce una lista in ordine inverso).

Happy non supporta la firma di tipo per produzioni parametrizzate.

Alex

Alex è un tool per generare un analizzatore lessicale in Haskell. Alex prende in input una descrizione di token basati su espressioni regolari e genera un modulo Haskell contenente codice per scannerizzare il testo efficientemente.

Alex viene installato automaticamente da BNFC ma è possibile installarlo manualmente tramite Stack con il comando:

```
stack install alex
```

Struttura di un sorgente Alex

```
{ ⟨module header⟩ }
⟨directives⟩
⟨macro definitions⟩
⟨identifier⟩ :-
⟨token definitions⟩
{ ⟨optional module trailer⟩ }
```

`<module header>` è un header di modulo Haskell che viene ommesso nel modulo generato.

Nella sezione `<directives>` abbiamo le direttive di tipo `%wrapper "<name>"` che controllano che tipo di codice di supporto Alex deve produrre insieme allo scanner base.

Nella sezione `<macro definitions>` definiamo le macro da utilizzare nelle definizioni dei token.

Lo scanner è specificato come una serie di definizioni di token. Ogni definizione prende la forma:

```
regexp { code }
```

ovvero "se l'input fa match con regexp ritorna code"

In fondo al file abbiamo un altro frammento di codice tra graffe. Qui definiamo il tipo dei token e aggiungiamo tutto il codice di supporto. Esistono due tipologie di macro:

- Character set expressions: `$<identifier> = <set expression>`
- Regular expressions: `@<identifier> = <regular expression>`

Le set expressions sono:

- **char** --> un singolo carattere unicode (quelli speciali richiedono di essere escaped).
- **char-char** --> un range di caratteri seguendo l'ordine unicode (inclusi gli estremi).
- **.** --> matcha tutti I caratteri eccetto \n.
- **set0 # set1** --> matcha I caratteri in set0 ma che non sono in set1.
- **[sets]** --> unione di sets.
- **~set** --> complemento di sets.
- **[^sets]** --> complemento dell'unione di sets.
- **$ident** --> espande la def. dell'appropriata set expression macro.

Le espressioni regolari sono:

- **set** --> matcha ciascuno dei char nel set.
- **@ident** --> espande la def. dell'appropriata ER macro.
- **"…"** --> matcha la seq. di caratteri nella stringa in quell'ordine.
- **r1r2** --> matcha l'unione.
- **r1|r2** --> matcha r1 o r2.
- **r*** --> matcha 0 o + occorrenze di r.
- **r+** --> matcha 1 o + occorrenze di r.
- **r?** --> matcha 0 o 1 occorrenze di r.
- **r{n}** --> matcha n occorrenze di r.
- **r{n,}** --> matcha n o + occorrenze di r.
- **r{n,m}** --> matcha da n a m (inclusi) occorrenze di r.

Regole per la definizione dei token

La sintassi (ver. base) della sintassi delle regole è: `<regular expression> {(action)}`. Ogni regola definisce un token nella specificazione lessicale. Quando l'input stream combacia con l'ER in una regola, il lexer ritorna il valore dell'azione (l'espressione a destra).

- L'azione può essere un'espressione haskell. Alex ha una sola restrizione sull'azione: ogni azione deve avere lo stesso tipo.
- L'azione può essere ommessa, indicata rimpiazzandola con ';'. In questo caso il token viene skippato dall'input stream.

Alex trova sempre il modo ti eseguire il match più ampio di ogni ER. Quando l'input stream matcha più di una regola, la regola con il prefisso più lungo dell'input stream vince. Se ci sono regole che matchano in egual numero di caratteri allora vince la regola che compare prima nel file di input.

Wrapper "basic"

Il wrapper "basic" è un buon modo per ottenere da una specifica di lexer, con un po' di confusione, una funzione `alexScanToken` di tipo `String -> [t]` che prende una stringa in input e ritorna una lista dei token che contiene. Tutte le azioni nella specificazione lessicale devono avere tipo `String -> t`. Il lessema che attualmente matcha la testa della regola è passato come argomento all'azione.

Esempio:

```
% wrapper " basic "

$digit = 0-9            -- digits
$alpha = [a-zA -Z]      -- alphabetic characters

tokens :-
      $white+            ;
      " - -".*           ;
      l e t        { \s -> Let }
      i n                { \s -> In }
      $digit+            { \s -> Int ( read s) }
      [\=\+\ -\*\/\(\)]  { \s -> Sym ( head s) }
      $alpha [ $alpha $digit \_ \ ']* { \s -> Var s }
                  -- this rule MUST be last!
{
data Tok = Let | In | Sym Char | Var String | Int Int
}
```

BNFC in ambiente C/C++

Passiamo ora ad analizzare i tool per la generazione di lexer e parser in ambiente C/C++. I tool di default per cui BNFC genera gli input; se invocato con il parametro **--cpp** o **--c** sono i seguenti:

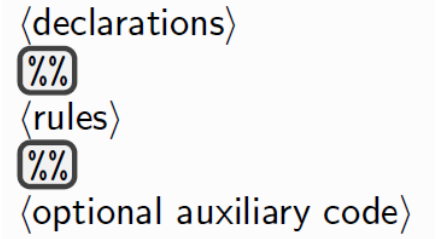
- Lexer: Flex
- Parser: Bison (Yacc)

Flex

Il tool Flex è un tool per generare analizzatori lessicali in codice C o C++. Similmente ad Alex, Flex prende in input una descrizione di token basate su espressioni regolari e genera un modulo C/C++ contenente del codice per scannerizzare il codice in modo efficiente.

Flex è ottenibile dal link: **http://gnuwin32.sourceforge.net/packages/flex.htm**

Struttura di un sorgente Flex



Nella sezione <declarations> definiamo le macro da utilizzare nella sezione rules. Ogni dichiarazione segue la seguente sintassi:

<identifier> <regular expression>

Ogni regola, nella sezione <rules> segue la sintassi:

<regular expression> <code>

la cui semantica è la seguente: se l'input matcha l'ER, allora esegui il codice. Si noti che ora, essendo in ambiente C/C++ il paradigma imperativo fa sì che il termine più appropriato sia "esegue" e non "ritorna".

In fondo al file abbiamo un frammento di codice che viene riportato tale e quale nel modulo generato.

Nelle prime due sezione, ogni testo indentato o ogni testo racchiuso tra %{. . .}% viene incluso nel modulo generato. Bisogna fare quindi attenzione: le regole e le dichiarazioni devono iniziare a inizio linea.

Le espressioni regolari sono:

- **char** --> un singolo carattere (i caratteri speciali vanno escaped)
- **.** --> tutti i caratteri escluso newline \n
- **[ranges]** --> matcha tutti i caratteri specificati nel range; che è una sequenza o di caratteri singoli o char-char (in base alla codifica ASCII)
- **[^ranges]** --> not della regola precedente
- **{ident}** --> espande la definizione della macro appropriata
- **"…"** --> matcha la sequenza di caratteri nella stringa tra virgolette, in quell'esatto ordine
- **r1 r2** --> matcha la concatenazione di r1 e r2
- **r1|r2** --> matcha r1 o r2
- **r*** --> matcha 0 o più occorrenze di r
- **r+** --> matcha 1 o più occorrenze di r
- **r?** --> matcha 0 o una occorrenza di r
- **r{n}** --> matcha n occorrenze di r
- **r{n,}** --> matcha n o più occorrenze di r
- **r{n,m}** --> matcha tra n e m occorrenze di r (inclusi n e m)

Come funziona FLex

Il matching dell'input avviene nel seguente modo:

- Flex trova sempre il match più lungo di ogni ER.
- Quando l'input stream matcha più di una regola, la regola che matcha il prefisso più lungo viene accettata.
- Nel caso in cui due regole matchino lo stesso numero di caratteri, vince la regola che è stata definita per prima.

Cosa fa:

- Quando un match viene determinato, il lessema è reso disponibile tramite il puntatore a carattere predefinito **yytext**, e la sua lunghezza è memorizzata nella variabile intera globale **yyleng**. L'azione corrispondente al match viene eseguita, e quindi viene scannerizzato l'input rimanente.
- Le azioni potrebbero impostare la variabile **yylval** che il parser può usare (tipicamente usata per attributi dei token).
- Se nessun match viene trovato, di default viene eseguita la regola di default: il prossimo carattere in input viene considerato matchato e copiato su standard output (stdout).

Bison (Yacc)

Bison è un generatore di parser general-purpose che converte una grammatica context-free (CF) in un parser (deterministico) LALR(1). Bison supporta la retro compatibilità con Yacc: ogni grammatica scritta rispettando le specifiche di Yacc è un valido input per Bison. I linguaggi target di Bison sono C e C++ e (in versione sperimentale) Java (le cui soluzioni più popolari sono CUP e JLex).

Bison è ottenibile dal link: **http://gnuwin32.sourceforge.net/packages/bison.htm**

Struttura del sorgente Bison

```
%{ ⟨Prologue⟩ %}
⟨directives⟩
%token ⟨token declarations⟩
%%
⟨productions⟩
%%
⟨Epilogue⟩
```

Nella sezione <Prologue> si possono definire tipi e variabili da utilizzare nelle azioni. È possibile utilizzare anche comandi del preprocessor. È necessario dichiarare l'analizzatore lessicale **yylex** e l'error printer **yyerror** in questa sezione, insieme ad ogni identificatore globale utilizzare nelle azioni della grammatica.

In <directives> sono incluse direttive di tipo **%left**, **%right** o **%nonassoc** analogamente ad Happy.

La dichiarazione di un token non è altro che una enumerazione di identificatori (per convenzione tutti uppercase) che Bison associa a valori interi (crea anche un file con le definizioni da utilizzare nel codice del lexer -- si noti che quindi in fase di compilazione è necessario compilare prima il sorgente Bison e poi quello di FLex).

Nelle produzioni, letterali tipo '+' possono essere utilizzati senza dichiarazioni e Bison li associa automaticamente con l'intero corrispondente nella codifica ASCII.

Ciascuna produzione consiste di un simbolo non terminale **NT** seguito dai due punti. I due punti sono a sua volta seguite da una o più espansioni sulla destra, separate da **|**.

Ciascuna espansione ha del codice associato, chiuso tra parentesi graffe. Il parser esegue tale codice ogni qual volta riduce la regola.

- Il codice può referenziare i valori dell'i-esimo simbolo (sia token che non-terminali) utilizzando **$i**
- Il codice può definire il valore di **NT** usando **$$=expr**.

Esempio:

```
Exp : Exp '+' Exp1      { $$ = mkPlusNode ($1 , $3) }
    | Exp1              { $$ = $1 }
```

Se il parser svolge una riduzione di una produzione **NT: a_1, …, a_n {code}** e il valore generato da ciascun **a_i** è **v_i** allora esegue il codice dove ciascun **$i** è stato rimpiazzato da **v_i**. Un assegnamento a **$$** definisce il valore di NT.

Il tipo dei valori generati è **int** di default fino a che nel preambolo non viene usata la direttiva:

```
#define YYSTYPE …
```

Se non viene specificato del codice allora il codice di default è **{$$ = $1}**.

Le produzioni vuote sono consentite: **A:{code}** e corrispondono alla produzione **A**-->**epsilon**. È consigliato esplicitare la cosa scrivendo **A: {-empty-}{code}** per rendere l'epsilon evidente e rendere il codice più leggibile.

Il codice può anche essere inserito tra i simboli, ad esempio:

```
Exp : Exp '+' { printf ("+"); } Exp1 { mk($4 ); }
```

(per utilizzare tale feature è consigliato conoscere bene le grammatiche L-attributed).

Ogni pezzo di codice conta come terminali o non terminali (infatti sopra **$4** si riferisce a Exp1).

Modello master-slave di Bison e FLex

Il parser generato da Bison e il lexer generato da Flex funzionano secondo un modello **master-slave**. Vale a dire che ogni volta che il parser ha bisogno di un nuovo token lookahead, chiama il lexer che consuma abbastanza input (concreto) per generare un nuovo token e quindi restituisce il controllo al parser.

Questo è possibile poiché il lexer può avere l'effetto collaterale (side-effect) di consumare l'input; mentre il parser Bison funziona, può inoltre alterare il comportamento del lexer Flex (di modifica delle variabili globali).

Dalla parte di Haskell (fino ad ora) abbiamo avuto un lexer puramente funzionale che è condannato a prendere l'intero input e produrre l'intero output (che è una lista di tokens).

Tuttavia, per la lazyness del linguaggio, il codice del lexer viene eseguito gradualmente man mano che nuovi token risultano necessari al parser.

Ad ogni mod, nonostante la lazyness, o non è possibile per il parser Happy modificare il comportamento in progresso di Alex lexer.

Uso delle precedenze

Bison permette di specificare le precedenze degli operatori per risolvere i conflitti, utilizzando le direttiva **%left**, **%right** e **%nonassoc** seguite da una lista di terminali. Esse dichiarano il lato dell'associatività di tali non terminali. La precedenza di questi token in relazione ad altri token è stabilita dall'ordine delle direttive: prima vuol dire meno precedenza. Esempio:

```
% left OR
% left AND
% left NOT
% nonassoc EQUAL NOTEQUAL LESS LESSEQUAL GREATER
% left PLUS MINUS
% left MUL DIV MOD
% right CIRCUMFLEX
```

Precedenze dipendenti dal contesto

La precedenza di una regola individuale può essere sovrascritta, usando la precedenza di contesto. Questa feature è utile quando un token ha differenti precedenze in base al contesto. La direttiva **%prec** è associata alla regola da sovrascrivere (sovrascrivere la precedenza di default).

Un esempio comune è il segno meno: ha precedenza più alta quando usato come prefisso di negazione, ma minore quando usato come sottrazione binaria.

Potremmo inventarci un nuovo token **NEG** da usare come rimpiazzo. Il token **NEG** non deve per forza apparire in una direttiva %token.

```
% left PLUS MINUS
...
% left NEG
```

Associamo la direttiva **%prec NEG** al prefisso della negazione:

```
Expr : Expr MINUS Expr { ... }
     | MINUS Expr % prec NEG { ... }
```

Conflitti irrisolti

Nel caso in cui alcuni conflitti non possano essere risolti, Bison (esattamente come fa Happy nella controparte in Haskell) riporta dei warning ma il codice del parser viene comunque generato seguendo le regole elencate:

- Nel caso di conflitto **shift/reduce** viene attuato uno shift;
- Nel caso di conflitto **reduce/reduce** viene attuato una azione di riduzione con la regola definita prima nel sourcecode.

Ovviamente questa soluzione non produce per forza il risultato voluto e potrebbe richiedere modifiche manuali.

Error Recovery

Il parser Bison genera un token speciale **error** ogni qual volta viene generato un syntax error. Questo simbolo terminale è sempre definito e riservato per la gestione degli errori.

Fasi:

- Il parser scarta gli stati e oggetti dallo stack affinché non torni indietro allo stato in cui il token **error** è accettato; uno stato con un item **A** --> **error a**.
- Il token error viene shiftato.
- Dopo, il parser legge i token e li scarta fino a che può ridurre ad **a**. In caso a = epsilon nulla viene scartato.
- Reduce con la regola **A** --> **error a**.

Esempio:

```
Stmt : error ';' { ... }
```

In caso di errore scarta tutto l'input fino al primo **;** e continua come se un **Stmt** sia stato riconosciuto.