Linguaggi di Programmazione 2022-2023

Prolog e Programmazione Logica VI

Marco Antoniotti
Gabriella Pasi
Fabio Sartori

Hands-on Predicates

- Per scrivere programmi efficaci in qualunque linguaggio di programmazione dobbiamo padroneggiare l'«ecologia» dell'ambiente complessivo creato dal linguaggio stesso; ovvero dobbiamo padroneggiare le librerie accessibili dal linguaggio

- Di seguito vedremo molti predicati di utilità generale e per i progetti in particolare (alcuni di questi predicati sono specifici di SWI Prolog)

- Successivamente, vedremo alcuni predicati che ci permettono di rivedere informalmente la nozione di **Recursive Descent Parser** ed i dettagli necessari per far funzionare effettivamente le cose

Caratteri e Codici

- I sistemi Prolog originali rappresentavano le «stringhe» come liste di codici (ASCII) di caratteri
- SWI Prolog ha un default diverso con le stringhe rappresentate come oggetti a sé stanti
- SWI Prolog ha molti predicati per gestire stringhe; tra quelli più utili abbiamo i seguenti:

  `atom_string/2`

  `number_string/2`

  `string_codes/2`

- La documentazione completa si trova «el manuale SWI Prolog al capitolo 5.2 «*The string type and its double quoted syntax*»

Caratteri e Codici

- I tre predicate sono invertibili (a patto di avere uno degli argomenti completamente istanziato)

  atom_string/2

  number_string/2

  string_codes/2

- **Esempi:**

```
?- number_string(QD, ”42.0”).
QD = 42.0

?- string_codes(”42”, Cs).
Cs = [52, 50]
```

Leggere stringhe e files

- Il predicato principale per leggere una stringa da uno «stream» è il seguente

```
read_string(InputStream, Length, String).
```

Notate che il primo argomento è uno «stream»; dobbiamo aprirne uno per usare `read_string/3`

- **Esempio**:

```
?- open('inferno.txt', read, In),
   read_string(In, _, Nel_mezzo),
   close(In).
In = <stream>(0x103466c00),
Nel_mezzo = "Nel mezzo del cammin di nostra vita\nMi
ritrovai...\n"
```

Leggere stringhe e files

- Ora possiamo scrivere i due predicate (ed altri ancora, ovviamente)
  `read_file_from_string/2` and `read_file_from_string/3` come segue

```
read_file_to_string(Filename, Result) :-
    read_file_to_string(Filename, Result, []).
read_file_to_string(Filename, Result, Options) :-
    open(Filename, read, In, Options),
    read_string(In, _, Result),
    close(In).
```

- Questi predicati leggono il contenuto di un file (di testo) in una singola stringa

Leggere stringhe e files

- Fare «parsing» in Prolog è più semplice quando si manipolano *liste* di *codici* (ovvero, *codici* di *caratteri*)
- SWI-Prolog fornisce un predicato nella libreria `readutil`:

```
read_file_to_codes(Filename, Codes, Options)
```

- Con questo predicato possiamo ottenere la lista di codici da passare al «parser»

- **Esempio**

```
?- read_file_to_codes('inferno.txt', Codes, []).
Codes = [78, 101, 108, 32, 109, 101, 122, 122|...].
```

Controllare le caratteristiche dei caratteri

- Un predicato molto utile che è usato per classificare i caratteri è

```
char_type(Character, Type).
```

- Il predicato può essere usato su codici, caratteri singoli, atomi con nome di un singolo carattere e stringhe di lunghezza 1

- Si noti che un carattere può avere molti «tipi» associati; provate l'esempio qui sotto

```
?- char_type(C, punct),
   write('Char: '),
   writeln(C),
   fail.
```

Parsing di numeri interi

- Ora possiamo procedere a definire un predicato che può far eil «parsing» di una stringa in un numero intero

- Il predicato utilizza delle tecniche di programmazione rappresentate in forma Prolog

- Procederemo in modo «top-down» a definire i predicati

```
parse_integer(Chars, I, MoreChars).
parse_integer(Chars,
              DigitsSoFar,
              I,
              IntegerCodes,
              MoreChars).
```

Parsing di numeri interi

- Il primo predicato, **`parse_integer/3`**, è un predicato di convenienza che prepara la chiamata a **`parse_integer/5`**

```
parse_integer(Chars, I, MoreChars) :-
    parse_integer(Chars, [], I, _, MoreChars).
```

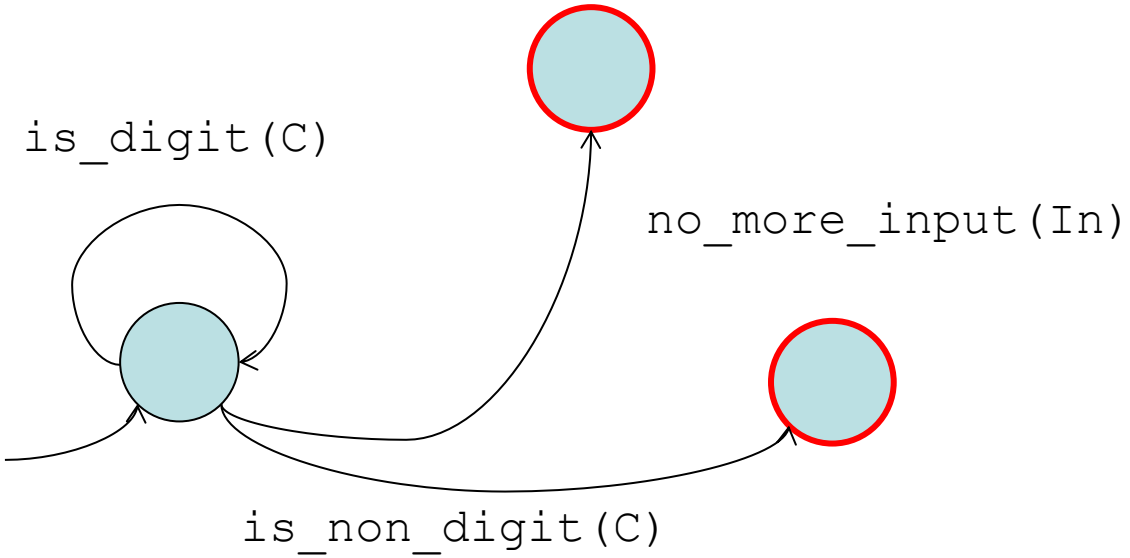
- Il predicato **`parse_integer/5`** ha bisogno di essere inizializzato con un accumulatore vuoto nella chiamata qui sopra e non ci interessa il valore finale del terzo argomento

Parsing di numeri interi

- Il predicato `parse_integer/5` ha i seguenti argomenti
  1. La lista di codici di caratteri che si sta scansionando.
  2. Un accumulatore (i.e., una list) di caratteri (i.e., codici di carattere delle cifre) già viste fino al momento della chiamata (ricorsiva)
  3. L'intero che si è riconosciuto (ovvero «parsato»)
  4. Una lista dei codici di carattere delle cifre che costituiscono l'intero
  5. I codici di carattere a partire dal primo codice di un carattere diverso da '0' a '9' trovato dopo l'intero

Parsing di numeri interi

- Il predicato **`parse_integer/5`** scansiona la lista di codici di carattere da sinistra a destra e, de facto, implementa il seguente automa (con stati finali bordati di rosso)



Parsing di numeri interi

- I tre stati sono tradotti in tre regole
- La prima regola raccoglie le cifre («digits») del numero intero

```
parse_integer([D | Ds], DsSoFar, I, ICs, Rest) :-
    is_digit(D),
    !,
    parse_integer(Ds, [D | DsSoFar], I, ICs, Rest).
```

- Notate come i codici delle cifre sono consumati prima della chiamata ricorsiva

Parsing di numeri interi

- La seconda regola corrisponde al primo stato finale, in cui, data la combinazione dell'ordine delle regole stesse e dei tagli, sappiamo di non avere un codice di una cifra («digit») in `C` da gestire

```
parse_integer([C | Cs], DsR, I, Digits, [C | Cs]) :-
    % not_is_digit(C),
    !,
    reverse(DsR, Digits),
    number_string(I, Digits).
```

- Questo è un caso base della ricorsione e quindi prodice dei risultati; tre in particolare
  1. La lista delle cifre raccolte è invertite (nella lista **`Digits`**)
  2. L'intero **`I`** viene finalmente costruito (via **`number_string/2`**)
  3. Il carattere **`C`** è "*put back*" nell'input

Parsing di numeri interi

- Le terza regola corrisponde allo stato finale in cui abbiamo consumato tutto l'input

```
parse_integer([], DsR, I, Digits, []) :-
    !,
    reverse(DsR, Digits),
    number_string(I, Digits).
```

- Questo caso base per la ricorsione deve fare solo due cose, dato che non c'è nulla da "*pushed back*" in input
  1. La lista delle cifre raccolte è invertite (nella lista **`Digits`**)
  2. L'intero **`I`** viene finalmente costruito (via **`number_string/2`**)

Parsing di interi e «floats»

- Il predicato presentato è abbastanza corretto; non riconosce correttamente i numeri negativi, quali `-42`
  - Sono necessarie altre regole per aggiungere tale capacità

- Il parsing di “semplici” float del tipo

  `Float ::= Digit+ [‘.’ Digit+]`

  può essere implementato da zero come nel caso precedente

- Tuttavia, al fine di illustrare un principio, nel seguito vedremo una versione che riusa **`parse_integer/5`**

Parsing Floats

- Procediamo col definire un predicato che può riconoscere dei floats

- Procediamo ancora con una definizione top-down

```
parse_float(Chars, F, MoreChars).
parse_float(Chars,
            F,
            FloatCodes,
            MoreChars).
```

Parsing Floats

- Come per il caso precedente, il predicato principale è **`parse_float/4`**, che riusa **`parse_integer/5`**; la strategia è di fare due cose in sequenza:
  1. Riconoscere la parte intera usando **`parse_integer/5`**
  2. Riconoscere la parte decimale usando un nuovo predicato **`parse_float_decimal/4`**

- Il codice per **`parse_float/3`** è il seguente

```
parse_float(Chars, F, MoreChars) :-
    parse_float(Chars, F, _, MoreChars).
```

Parsing Floats

- Il codice per `parse_float/4` è il seguente

```
parse_float(Chars, F, FloatCodes, MoreChars) :-
    parse_integer(Cs, [], _I, IntegerDigits, MoreInput),
    parse_float_decimal(MoreInput,
                        _Decimal,
                        DecimalChars,
                        DecimalRestCs),
    append(IntegerDigits, DecimalChars, FloatChars),
    number_string(F, FloatChars).
```

- Come preannunciato, prima riconosciamo la parte intera e successivamente la parte decimale
  - Si noti ancora che il predicato è incompleto e non gestiscenumeri negativi; aggiungere questa parte è un utile esercizio

Parsing Floats

- Ora vediamo il codice per `parse_float_decimal/4` che ha due casi base e che – ancora – riusa `parse_integer/5`

```
parse_float_decimal([D | Ds],
                    Decimal,
                    [0'. | DigitChars],
                    AfterDecimalCodes) :-
    is_dot(C),
    !,
    parse_integer(Ds, [], I, DigitChars, AfterDecimalCodes),
    length(DigitChars, NDs),
    Decimal is I * (10.0 ** NDs).
```

- Il predicato per prima cosa controlla che ci sia un punto '.' in input e, se c'è, procede a parsare la parte decimale; infine deve produrre un la parte decimale in formato float appropriato

Parsing Floats

- I due casi base per `parse_float_decimal/4` sono i seguenti

```
parse_float_decimal([D | Ds], 0.0 , [0'., 0'0], [D | Ds]).

parse_float_decimal([], 0.0 , [0'., 0'0], []).
```

- Ancora una volta, i due casi base sono (1) il caso in cui `D` è dopo il punto decimale (per via del cut nella regola precedente), o (2) quando l'input è vuoto

Conclusioni

- Abbiamo appena fatto un corso super accelerato sul *recursive descent parsing*

- Sebbene i linguaggi (interi, float) che abbiamo riconosciuto siano **regolari**, le tecniche mostrate vi indicano coem procedere alla costruzione di Recursive Descent Parsers (RDPs) più generali
  - Notate che non abbiamo discusso tutta la teoria alla base del RDPs, ma ora dovreste essere in grado di attaccare progetti (e.g., parsing to JSON) *evitando* i problemi derivanti da semplici ricerche su stringhe, anche se basati su espressioni regolari
  - Ricordare che JSON è un *Context Free Language*, completamente(e ricorsivamente) parentesizzato

HAPPY HACKING