Linguaggi di Programmazione 2015/2016

Prolog e Programmazione Logica VI

Marco Antoniotti
Gabriella Pasi

Hands-on Predicates

- Nowadays, writing effective programs in any language requires mastering the underlying language ecology; i.e., the language libraries

- In the following, we will see several predicates that will be useful in general and for the project as well (some of these predicates are SWI-Prolog only)

- Next we well see a few predicates that informally illustrate the notion of **Recursive Descent Parser** and the dirty details needed to actually get things done

Characters and Codes

- Prolog used to represent strings as lists of character (ASCII) codes
- SWI Prolog defaults to a different setting with strings being proper objects on their own right
- SWI Prolog has several predicates that deal with strings; those that we need most are

```
atom_string/2
number_string/2
string_codes/2
```

- The complete documentation is in the Manual in Chapter 5.2 "*The string type and its double quoted syntax*"

Characters and Codes

- The three predicates are "invertible" (as long as one of the two arguments is fully instantiated)

```
atom_string/2
number_string/2
string_codes/2
```

- **Examples:**

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

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

Reading Strings and Files

- The main predicate to read a string from a stream is

  `read_string(InputStream, Length, String).`

  Note that the first argument is a stream; we must open one to use `read_string/3`

- Examples

```
?- 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"
```

Reading Strings and Files

- We can therefore write two predicate **`read_file_from_string/2`** and **`read_file_from_string/3`** as follows

```
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).
```

- These predicates read the content of a (text) file into a single string.

Reading Strings and Files

- We said that parsing is easier to implement in Prolog when dealing with *lists of codes* (i.e., *character codes*)
- SWI-Prolog provides a predicate in the readutil library:

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

- With this predicate we can obtain the list of codes we need to parse

- **Example**

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

Checking Characters

- A very useful predicate that is used to classify characters is

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

- The predicate can be used on characters codes, single characters atoms and single character string
- Note that a character may have several types associated; try the example below.

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

Parsing Integers

- Now we can proceed to define a predicate that can parse integers
- The predicate packs a number of programming techniques which are represented in Prolog form
- We will proceed top-down to define the predicates

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

Parsing Integers

- The first predicate, **parse_integer/3**, is a convenience predicate that sets up a call to **parse_integer/5**

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

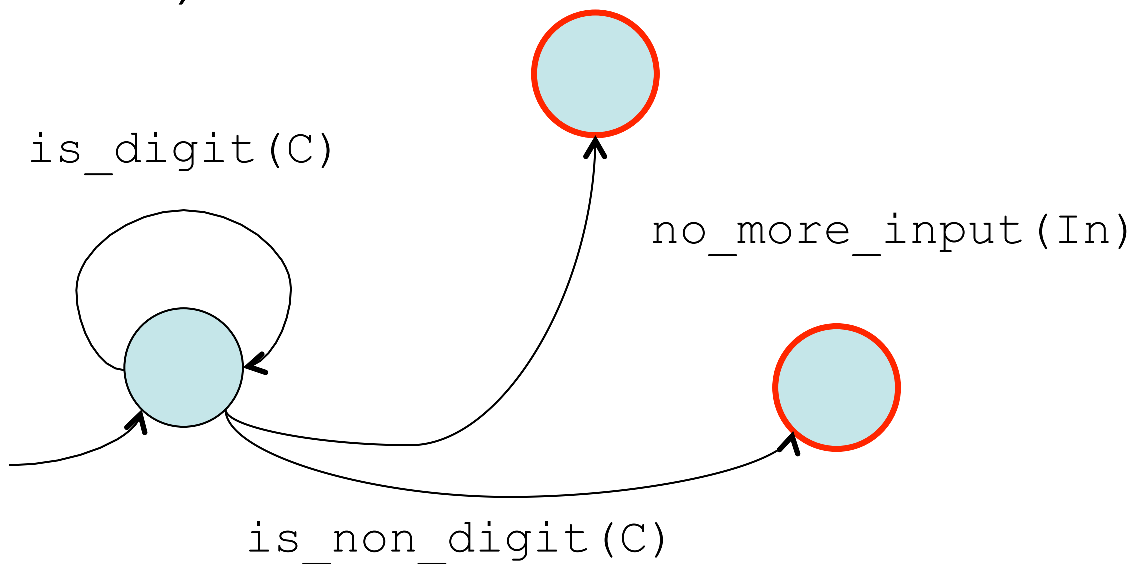
- The predicate **parse_integer/5** needs to be initialized with an empty accumulator in the above call, and we do not care for the content of the third argument.

Parsing Integers

- The predicate `parse_integer/5` has the following arguments.
  1. The list of character codes being scanned.
  2. An accumulator (i.e., a list) of characters (i.e., characters codes of the digits) seen up to a point.
  3. The actual integer parsed.
  4. A list of the digit character codes that make up the integer.
  5. The character codes starting from the first non digit character code after the integer.

Parsing Integers

- The predicate **`parse_integer/5`** scans character codes left to right and essentially implements the following automata (red bordered states final)



Parsing Integers

- The three states are translated into three rules
- The first rule collects integer digits

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

- As you see the digit codes are consumed before the recursive call

Parsing Integers

- The second rule correspond to the first final state, where, by combination of the rule order and the cuts, we know we do not have a digit code in C to deal with

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

- This is a base case for the recursion, and thus we need to produce results; this is done in three steps
  1. The collected digits are reversed (into **`Digits`**)
  2. The integer **`I`** is finally produced (via **`number_string/2`**)
  3. The character **`C`** is "*put back*" on the input

Parsing Integers

- The third rule correspond to the final state, where we may have consumed all the input

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

- This base case for the recursion just need to perform two steps, as there is nothing to be "*pushed back*" onto the input
  1. The collected digits are reversed (into **`Digits`**)
  2. The integer **`I`** is finally produced (via **`number_string/2`**)

Parsing Integers and Floats

- The predicate is mostly correct; it does not parse correctly negative integers, such as `-42`
  - You just need to add rules to that effect

- Parsing "simple" floats of the form

  `Float ::= Digit+ ['.' Digit+]`

  can be done in the same way

- Nevertheless, to illustrate a principle, here is a version that reuses **`parse_integer/5`**

Parsing Floats

- Let's proceed to define a predicate that can parse floats

- Again, we will proceed top-down to define the predicates

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

Parsing Floats

- Let's proceed to define a predicate that can parse floats

- Again, the main predicate is **`parse_float/4`**, which reuses **`parse_integer/5`**; the strategy is to do two things.
  1. Parse the integral part using **`parse_integer/5`**
  2. Parse the decimal part using a new predicate **`parse_float_decimal/4`**

- The code for **`parse_float/3`** is the following

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

Parsing Floats

- The code for **parse_float/4** is the following

```
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).
```

- As promised, we first parse the integral part, and afterword, we parse the decimal part
  - Note again that we do not handle negative numbers; it is a useful exercise to add them

Parsing Floats

- Let's now see the code for **parse_float_decimal/4** which has two base cases and which – again – reuses **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).
```

- The predicate first checks to see if a dot '.' in on the input and, if so, it proceeds to parse the decimal part; eventually it has to produce a proper decimal float

Parsing Floats

- The two base cases for `parse_float_decimal/4` are as follows

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

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

- Again the two base cases correspond to a case where `D` is something after the decimal part (because of the cut in the previous rule), or when the input is empty

Conclusion

- We have just gone through a crash course on recursive descent parsing

- Although the languages (integers, floats) we recognized are regular, the techniques shown point the way towards the construction of more general Recursive Descent Parsers (RDPs)
  - Note that we did not properly discuss all the theory behind RDPs, but you should now be able to tackle projects like JSON parsing while *avoiding* the pitfalls of simple string searches, even if based on regular expressions
  - Remember that JSON is a *Context Free Language*, fully parenthesized

HAPPY HACKING