Advanced Prolog

Difference Lists and Grammar Notation

Robert Keller
April 2010

Difference Lists

- The standard append concatenates two lists in time proportional to the length of the first list.
- Is this the best we can do?

Difference Lists

- Is this the best we can do?
- Represent a list by a pair

    d(B, T)

  where B ("body") looks a lot like a conventional Prolog list, but ends with **variable** T ("tail") instead of [].
- Example:

    d([a, b, c | T], T)

Appending Difference Lists

d([a, b, c | T1], T1)

d([e, f, g, h | T2], T2)

Unify T1 with the body of the second list to get

d([a, b, c, d, e, f, g, h | T2], T2)

Variable T1 becomes bound. T2 is left unbound.

Appending Difference Lists

dappend(d(B1, T1), d(B2, T2), d(B1, T2)) :- T1 = B2.

Or more simply, since B1, etc. are just variables:

dappend(d(B1, B2), d(B2, T2), d(B1, T2)).

```
?- dappend(d([1,2,3 | T1], T1), d([4,5,6,7 | T2], T2), Z).

T1 = [4, 5, 6, 7|T2],
Z = d([1, 2, 3, 4, 5, 6, 7|T2], T2)
```

Difference to Regular

- To convert a difference list to a regular list, simply **unify the tail** with []:
  - d2r(d(B, []), B).
- To convert a regular list to a difference list, use recursion to **traverse the original list**:
  - r2d([], d(T, T)).
  - r2d([A | X], d([A | B], T)) :- r2d(X, d(B, T)).

```
?- r2d([1,2,3], X).

X = d([1, 2, 3|_G261], _G261)
```

One Issue with Difference Lists

- The body of a difference list cannot be shared.
- However, the entire difference list as a term can be shared arbitrarily. When it gets used, copying may result.
- Thus the "win" with difference lists is in the case where sharing is not expected.

Prolog's Built-in Grammar Notation

- Prolog started life as a language for translating languages. The syntax was then changed to be more logic-like.
- Grammar rules were re-introduced as "definite clause grammar" (DCGs).
- Grammar parsing is based on difference lists of tokens "underneath".

DCG (Definite Clause Grammar)

- Example: Consider the following grammar for S-expressions:
  - S → A
  - S → '(' T ')'
  - T → ε (where ε is the empty string)
  - T → S T (Essentially T → S*)

Translation to Prolog DCG

- Example: Consider the following grammar for S-expressions:
  - S → A
  - S → '(' T ')'
  - T → ε (where ε is the empty string)
  - T → S T (Essentially T → S*)
  - A → 0 | 1
- Non-terminals become predicate symbols (lower-case start).
- Literals (terminals) appear as list elements [...].
- Juxtaposition on the right becomes comma-separation.
  - s --> a.
  - s --> ['('], t, [')'].
  - t --> [].
  - t --> s, t.
  - a --> [0] | [1].

Parsing with DCGs

- The input is assumed to be a list of tokens.
- There is a **special built-in predicate** 'phrase':
  phrase(StartSymbol, TokenList)
  that produces the result of **parsing from** StartSymbol with **input** TokenList.
- Suppose the conceptual input is
  (0 ( ))
  a valid S-expression.
- As a token-list, this would be:
  ['(', 0, '(', ')', ')']

```
?- phrase(s, ['(', 0, '(', ')', ')']).

Yes
```

Parsing with DCGs

- An invalid S-expression:
  (0 ()

```
?- phrase(s, ['(', 0, ')', ')']).

No
```

Reversible Parsing

- By leaving the token list as a variable, Prolog will generate strings in the language.
- We usually need to specify the length in advance, or depth-first search may cause infinite recursion.

```
?- length(X, 3).

X = [_G235, _G238, _G241]

?- length(X, 3), phrase(s, X).

X = ['(', 0, ')']

?- length(X, 0), phrase(s, X).

No
```

Example

- Generate all S expressions of length 0 to 6, inclusive:

```
?- for(I, 0, 6), length(X, I), phrase(s, X).

I = 1,
X = [0] ;

I = 1,
X = [1] ;

I = 2,
X = ['(', ')'] ;

I = 3,
X = ['(', 0, ')'] ;
. . .

I = 6,
X = ['(', 1, 1, '(', ')', ')']

Yes
```

Using DCG's for Semantics

- Syntax = Structure
- Semantics = Meaning
- Example: Arithmetic with variables (juxtaposition = multiply)

```
s --> t.                        % sum
s --> t, [+], s.

t --> f.                        % term
t --> f, t.

f --> v.                        % factor

v --> ['x'] | ['y'] | ['z'].    % variable
```

```
?- phrase(s, [x, y, +, z, x]).

Yes
```

Possible Semantics

- Parse tree (syntax checker, or type checker)
- Immediate execution (interpreter)
- Code that will execute (compiler)
- Abstract interpretation (analysis tool)

How to Get Semantics

- Prolog DCG's allow non-terminals to have **arguments**.
- These arguments can be bound to the meaning of the string being parsed.
- Additional Prolog code can help in this process.
- (This approach takes us outside of the realm of simple context-free grammars.)

One Semantics: Parse Tree

```
s(T) --> t(T).                                  % sum

s(+(T, S)) --> t(T), [+], s(S).
  % tree constructor, root +

t(F) --> f(F).                                  % term

t(*(F, T)) --> f(F), t(T).
 % tree constructor, root *

f(T) --> v(T).                                  % factor

f(T) --> ['('], s(T), [')'].           % parenthesized sum

v(V) --> [V], {member(V, [x, y, z])}.           % variable
```

% {...} means to call ordinary goal ... in Prolog

Parse Examples

```
?- phrase(s(T), [x, +, y, +, z]).
T = x+ (y+z)

?- phrase(s(T), [x, y]).
T = x*y

?- phrase(s(T), [x, y, z]).
T = x* (y*z)

?- phrase(s(T), [x, +, y, z]).
T = x+y*z

?- phrase(s(T), ['(', x, +, y, ')', z]).
T = (x+y)*z

?- phrase(s(T), [x]).
T = x
```

Alternate Semantics: Evaluation, in an Environment

```
env([[x, 3], [y, 5], [z, 7]]).                    % environment
s(S) --> t(S).                                    % sum
s(S) --> t(X), [+], s(Y), {S is X + Y}.
t(F) --> f(F).                                    % term
t(P) --> f(X), t(Y), {P is X*Y}.
f(S) --> v(S).                                    % factor
f(S) --> ['('], s(S), [')'].                      % parenthesized sum
v(X) --> [V], {env(E), member([V, X], E)}.        % variable
```

Evaluation Examples

```
?- phrase(s(X), [x]).
X = 3

?- phrase(s(X), [x, +, z]).
X = 10

?- phrase(s(X), [x, +, z, y]).
X = 38

?- phrase(s(X), ['(', x, +, z, ')', y]).
X = 50
```

A Third Semantics: Code Generation

- Say we want to generate code for a stack machine, with instructions:
  - push(Value)
  - add
  - multiply
- The value is left atop the stack.
- The code will be generated as a Prolog list.

Grammar with Code Generation

```
env([[x, 3], [y, 5], [z, 7]]).                               % environment

s(S) --> t(S).                                               % sum
s(S) --> t(X), ['+'], s(Y), {append([X, Y, [add]], S)}.
t(F) --> f(F).                                               % term
t(P) --> f(X), t(Y), {append([X, Y, [multiply]], P)}.
f(S) --> v(S).                                               % factor
f(S) --> ['('], s(S), [')'].                                 % parenthesized sum
v([push(X)]) --> [V], {env(E), member([V, X], E) }.          % variable
```

append/2 appends elements of a **list of lists** together.

Code Generation Examples

```
?- phrase(s(S), [x]).
S = [push(3)]

?- phrase(s(S), [y, z]).
S = [push(5), push(7), multiply]

?- phrase(s(S), [y, z, x]).
S = [push(5), push(7), push(3), multiply, multiply]

?- phrase(s(S), [y, +, z]).
S = [push(5), push(7), add]

?- phrase(s(S), [y, z, +, x, z]).
S = [push(5), push(7), multiply, push(3), push(7), multiply, add]
```

Exercise

- Write the code that simulates the stack machine.
- Repertoire:
  - push(Value)
  - add
  - multiply

Assignment 10

- The stack machine NoRM is given.
- The grammar is given, but without semantics.
- Your problem: modify the grammar so that correct code is generated.

Example Test Cases

- test('x = 5 ;', success, [[x, 5]]),
- The program to be compiled is 'x = 5;'.
- Upon execution, the memory location associated with x should contain 5.
- NoRM Code: [push(5), store(4), halt]   (5 is a literal value)
- This code is "wired in". It needs to be changed so that all examples pass.

Example Test Cases

- test('x = z ;', success, [[x, 60]])
- This test fails.
- NoRM Code: [push(5), store(4), halt]   (5 is a literal value)
- The initial value of z is 60,
  but x is assigned the value 5 as before.

Example Test Cases

- test('if (y = 55) x = 5 ;', success, [[x, 40]])
- This test succeeds, but only by accident.
- NoRM Code: [halt]
- The initial value of x is 40, and is unchanged when the code is run.

How Grammar Rules are Represented Underneath

- A grammar rule with ***no argument*** is written as a Prolog clause with two arguments. These correspond to the body and tail of a **difference list**.
  - a --> b, c.
- Compiles into:
  - a(B1, T2) :- b(B1, T1), c(T1, T2).

The phrase Predicate

- phrase(Symbol, Sequence) :-
  Term =.. [Symbol, Sequence, []],
  call(Term).   % call constructed term as a goal
- Example:
  - a --> b, c.
  - phrase(a, Sequence) is equivalent to:

    a(Sequence, []).
- Recall:
  - a(B1, T2) :- b(B1, T1), c(T1, T2).

How Grammar Rules are Represented Underneath

- A grammar rule ***with an argument*** just adds another argument to the head:
  - a(X) --> b, c(X).
- Compiles into:
  - a(X, B1, T2) :- b(B1, T1), c(X, T1, T2).
  - This simply adds more arguments to the constructed Term.

How Grammar Rules are Represented Underneath

- A grammar rule with a Prolog condition just adds that condition to the clause:
  - a(X) --> b(X), c(X), {p(X)}.
- Compiles into:
  - a(X, B1, T2) :- b(X, B1, T1), c(X, T1, T2), p(X).

How Grammar Rules are Represented Underneath

- A grammar rule with terminals adds constants to the front of difference lists.
  - a(X) --> [b], c(X).
- Compiles into:
  - a(X, [b | B1], T1) :- c(X, B1, T1).

Summary of DCG Syntax

- --> indicates a production.
- | may be used on the RHS for *disjunction*.
- Terms not included in [...] or in {...} represent **non-terminals** in the grammar.
- Non-terminals can have arguments.
- Terms in [...] represent terminals. They can be variables or literals. More than one means to match each consecutively in the input.
- Terms in {...} are Prolog goals as is. They may share variables with the arguments of non-terminals.

Reference

- http://en.wikipedia.org/wiki/Definite_clause_grammar