Advanced Prolog

Bob Keller
April 2008
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:

  \[ d([a, b, c | T], T) \]
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:
  \[ r2d([], d(T, T)). \]
  \[ r2d([A | X], d([A | B], T)) :- r2d(X, d(B, T)). \]

?- r2d([1,2,3], X).
X = d([1, 2, 3|_G261], _G261)
Applying Difference Lists

\[ d([a, b, c \mid T1], T1) \]
\[ d([e, f, g, h \mid T2], T2) \]

Unify \( T1 \) with the body of the second list to get
\[ d([a, b, c, d, e, f, g, h \mid T2], T2) \]

Variable \( T1 \) becomes bound. \( T2 \) is left unbound.
Appending Difference Lists

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

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)
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 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”.
Example: Consider the following grammar for S-expressions:

- $S \rightarrow A$
- $S \rightarrow '(T')$
- $T \rightarrow \varepsilon$ (where $\varepsilon$ is the empty string)
- $T \rightarrow S\ T$ (Essentially $T \rightarrow S^*$)
Example: Consider the following grammar for S-expressions:

- \( S \rightarrow A \)
- \( S \rightarrow '(' T ')' \)
- \( T \rightarrow \varepsilon \) (where \( \varepsilon \) is the empty string)
- \( T \rightarrow S T \) (Essentially \( T \rightarrow S^* \))
- \( A \rightarrow 0 \mid 1 \)

Non-terminals become predicate symbols (lower-case start).
- Literals (terminals) appear as list elements [...].
- Juxtaposition on the right becomes comma-separation.

\( s \rightarrow a. \)
\( s \rightarrow '[' T ']' \).
\( t \rightarrow []. \)
\( t \rightarrow s, t. \)
\( a \rightarrow [0] \mid [1]. \)
The input is assumed to be a list of tokens. There is a built-in predicate 'phrase':

\[
\text{phrase(StartSymbol, TokenList).}
\]

Suppose the conceptual input is a valid S-expression:

\[
\text{( 0 ( ) )}
\]

As a token-list, this would be:

\[
[\text{('(', 0, '(', ')', ')')}]\]

?- phrase(s, [\text{('(', 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 may need to specify the length in advance, or depth-first search may cause infinite recursion.

```prolog
?- length(X, 3).
X = [_G235, _G238, _G241]
?- length(X, 3), phrase(s, X).
X = [', 0, ']
?- length(X, 0), phrase(s, X).
No
```
Generate all strings 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)

\[
\begin{align*}
\text{s} & \rightarrow \text{t}. & \text{% sum} \\
\text{s} & \rightarrow \text{t}, [+], \text{s}. \\
\text{t} & \rightarrow \text{f}. & \text{% term} \\
\text{t} & \rightarrow \text{f}, \text{t}. \\
\text{f} & \rightarrow \text{v}. & \text{% factor} \\
\text{v} & \rightarrow ['x'] \mid ['y'] \mid ['z']. & \text{% variable}
\end{align*}
\]

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

Yes
One Semantics: Parse Tree

\[
\begin{align*}
    s(T) & \rightarrow t(T). & \quad \% \text{sum} \\
    s(+ (T, S)) & \rightarrow t(T), [+], s(S). & \quad \% \text{tree constructor, root +} \\
    t(F) & \rightarrow f(F). & \quad \% \text{term} \\
    t(* (F, T)) & \rightarrow f(F), t(T). & \quad \% \text{tree constructor, root *} \\
    f(T) & \rightarrow v(T). & \quad \% \text{factor} \\
    f(T) & \rightarrow ['(', s(T), ')']. & \quad \% \text{parenthesized sum} \\
    v(V) & \rightarrow [V], \{\text{member}(V, [x, y, z])\}. & \quad \% \text{variable} \\
\end{align*}
\]

\% \{. . . .\} 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

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

s(S) --> t(S). % sum

s(S) --> t(X), [+], s(Y), {append([X, Y, [add]], S)}. % sum

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

t(P) --> f(X), t(Y), {append([X, Y, [multiply]], P)}. % term

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

f(S) --> ['('], s(S), [')']. % factor

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
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.

- $a \rightarrow b, c.$

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:
  - \( a(X) \rightarrow b(X), c(X) \).

- Compiles into:
  - \( a(X, B1, T2) :\text{:-} \ b(X, B1, T1), c(X, T1, T2) \).
  
- 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:
  - \( a(X) \rightarrow b(X), c(X), \{ p(X) \} \).

- 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.
  - \( a(X) \rightarrow [b], c(X). \)

- Compiles into:
  - \( a(X, [b \mid B1], T1) :- c(X, B1, T1). \)
Summary of DCG Syntax

- --> indicates a production.
- | may be used on the RHSs for disjunction.
- Terms not included in [...] or in {...} represent non-terminals in the grammar.
- 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 non-terminals.
- Non-terminals can have arguments.