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

- 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

- Append \( d([a, b, c | T], T) \) to \( d([e, f, g, h | T'], T') \):
- Unify \( T \) with the body of the second list to get \( d([a, b, c, d, e, f, g, h | T'], T') \).
- Variable \( T \) becomes bound. \( T' \) is left unbound.

Appending Difference Lists

- For append:
  \[ \text{dappend}(d(B1, T1), d(B2, T2), d(B1, T2)) \]
- Or more simply:
  \[ \text{dappend}(d(B1, B2), d(B2, T2), d(B1, T2)). \]

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

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 grammars" (DCGs).
- Grammar parsing is based on difference lists of tokens "underneath."
Translation to Prolog DCG

- Example: Consider the following grammar for S-expressions:
  - $S \rightarrow A$
  - $S \rightarrow '(', T, ')$
  - $T \rightarrow \varepsilon$ (where $\varepsilon$ is the empty string)
  - $T \rightarrow S \cdot T$
  - $A \rightarrow 0 | 1$

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

| s --> a. |
| s --> [', (', 0, ')', ')']. |
| 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(StartSymbol, TokenList)` that produces the result of parsing from `StartSymbol` with input `TokenList`.
- Suppose the conceptual input is `[0, 0, 0]`:
  - A valid S-expression.
  - As a token list, this would be: `[', 0, '(', ')', ')']`

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

- 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 = [_, _, _]
?- 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 = 4, X = ['(', 0, 0, ')'] ;
I = 5, X = ['(', 0, 0, 0, ')'] ;
I = 6, X = ['(', 0, 0, 0, 0, ')']
```

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 DCGs 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

```
% Prolog DCG rules

s(0) --> 0. % sum
s(0) --> 0, +, s(0). % tree constructor, root +

s(0) --> t(0). % term
s(0) --> t(0), +, s(0). % tree constructor, root *

t(0) --> f(0). % factor
f(0) --> [', (', 0, ')', ')']. % parenthesized sum
```

Using DCG's for Semantics

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

```
?- 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)
Parse Examples

\[ T = x + (y + z) \]

\[ T = x \cdot (y \cdot z) \]

\[ T = x + y \cdot z \]

\[ T = (x + y) \cdot z \]

\[ T = x \]

Alternate Semantics: Evaluation, in an Environment

\[ \text{env}(\{[x, 3], [y, 5], [z, 7]\}) \]

% environment

\[ s(S) \rightarrow t(S). \]
% sum

\[ s(S) \rightarrow t(X), \[+\], s(Y), \{S \text{ is } X + Y\}. \]
% term

\[ t(F) \rightarrow f(F). \]
% factor

\[ t(P) \rightarrow f(X), t(Y), \{P \text{ is } X \cdot Y\}. \]

\[ f(S) \rightarrow \['(', s(S), '\)', '\]. \]
% parenthesized sum

\[ v(\{\text{push}(X)\}) \rightarrow \{V\}, \{\text{env}(\text{E}), \text{member}([V, X], \text{E})\}. \]
% variable

Evaluation Examples

\[ x = 3 \]

\[ x + z = 10 \]

\[ x + z + y = 38 \]

\[ (x + z) + y = 50 \]

A Third Semantics: Code Generation

\[ \text{Say we want to generate code for a stack machine, with instructions:} \]
\[ \text{push(Value)} \]
\[ \text{add} \]
\[ \text{multiply} \]

\[ \text{The value is left atop the stack.} \]
\[ \text{The code will be generated as a Prolog list.} \]

Grammar with Code Generation

\[ \text{env}(\{[x, 3], [y, 5], [z, 7]\}) \]

% environment

\[ s(S) \rightarrow t(S). \]
% sum

\[ s(S) \rightarrow t(X), \[+\], s(Y), \{\text{append([X, Y, [add]], S)}\}. \]
% term

\[ t(F) \rightarrow f(F). \]
% factor

\[ t(P) \rightarrow f(X), t(Y), \{\text{append([X, Y, [multiply]], P)}\}. \]

\[ f(S) \rightarrow \['(', s(S), '\)', \]. \]
% parenthesized sum

\[ \text{append/2 appends elements of a list of lists together.} \]

Code Generation Examples

\[ \text{push}(3) \]

\[ \text{push}(5), \text{push}(7), \text{multiply} \]

\[ \text{push}(5), \text{push}(7), \text{push}(3), \text{multiply}, \text{multiply} \]

\[ \text{push}(5), \text{push}(7), \text{add} \]

\[ \text{push}(5), \text{push}(7), \text{multiply}, \text{push}(3), \text{push}(7), \text{multiply}, \text{add} \]

Exercise

\[ \text{Write the code that simulates the stack machine.} \]

Repertoire:

\[ \text{push(Value)} \]
\[ \text{add} \]
\[ \text{multiply} \]

Assignment 10

\[ \text{The stack machine NoRM is given.} \]

\[ \text{The grammar is given, but without semantics.} \]

\[ \text{Your problem: modify the grammar so that correct code is generated.} \]

Example Test Cases

\[ \text{Example Test Cases} \]

\[ \text{test('x = 5 ;', success, \{[x, 5]\})} \]

\[ \text{The program to be compiled is 'x = 5;.'} \]

\[ \text{Upon execution, the memory location associated with x should contain 5.} \]

\[ \text{NoRM Code: [push(5), store(4), halt]} \]

\[ \text{This code is "wired in". It needs to be changed so that all examples pass.} \]
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.

- test(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).
  - Example:
    - phrase(a, Sequence) is equivalent to:
      - a(Sequence, []).
  - Recall:
    - a(X, T2) :- b(X, T1), c(X, T1, T2).

- 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(X, B1, T1), c(X, T1, T2).
    - This simply adds more arguments to the constructed Term.

- 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).

- 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 [ ... ] represent terminals. They can be variables or literals. More than one means to match each consecutively in the input.
- Terms ( ... ) are Prolog goals as is. They may share variables with the arguments of non-terminals.

Reference