F28PL1 Programming Languages

Lecture 19: Prolog 4

Syntax

- *grammar*
- set of rules to determine if a sequence of symbols is well formed
- in grammar, distinguish:
  - *terminal* symbol == words, punctuation, numbers etc
  - *non-terminal* symbols == names of grammar rules

Syntax

- rule has form:

non-terminal -> *option1* | *option2* | ...

- to match *non-terminal*, match *option1* or *option2* or...
- *optioni* is a sequence of one or more terminal and non-terminal symbols
  - *sentential form*

article -> `the` | `a`

noun -> `cat` | `dog`

verb -> `saw` | `chased`

sentence -> *article noun verb article noun*

Syntax analysis

- *parsing*
- use rules to decide if terminal symbol sequence is well formed
- to match *non-termina*l, match *option1* or *option2* or...
- *sentence* rule
  - rule from which matching starts
- at each stage
  - consume some symbols from start of sequence
  - return rest of sequence to continue analysis

Syntax analysis

- is `the cat saw a dog` a *sentence*?
- try: *sentence*
  - try: *article*
  - try: `the` – success – `cat saw a dog`
  - try: *noun*
  - try: `cat` – success – `saw a dog`
  - try: *verb*
  - try: `saw` – success – `a dog`

Syntax analysis

- try: *article*
- try: `the` – fail & backtrack – `a dog`
- try: `a` – success – `dog`
- try: *noun*
- try: `cat` – fail & backtrack - `dog`
- try: `dog` – success -

Prolog syntax analysis

- Prolog well suited to syntax analysis
- close correspondence between:
  - syntax analysis behaviour
  - pattern matching with backtracking
- for each grammar rule non-terminal option:
  - disjunction of Prolog clauses
- for a rule option:
  - conjunction of Prolog terms

Prolog syntax analysis

- to match a terminal:
  - *non-terminal* -> *terminal*

`non-terminal([terminal|T],T).`

- symbol list starts with `terminal` in head
- after recognising `terminal` left with symbol list in tail *T*
- e.g. `article([the|T],T).`

```
| ?- article([the,cat,saw,a,dog],S).
S = [cat,saw,a,dog]
```

Prolog syntax analysis

- to match a non-terminal sequence
- consume from input *S* to return output *F*

```
non-terminal(S,F) :-
 non-terminal1(S,S1),
 non-terminal2(S1,S2)...
 non-terminalN(SN,F).
```

- each right hand side *non-terminali*
  - consumes symbols from *Si*
  - passes remaining symbols *Si+1* onto next *non-terminali+1*

Prolog syntax analysis

```
article([the|T],T).
article([a|T],T).
noun([cat|T],T).
noun([dog|T],T).
verb([saw|T],T).
verb([chased|T],T).
sentence(L,S) :-
 article(L,S1),noun(S1,S2),
 verb(S2,S3),
 article(S3,S4),noun(S4,S).
```

Prolog syntax analysis

`| ?- sentence([the,cat,saw,a,dog],F).`

`F = []`

- try: `sentence([the,cat...],F) :-`

  `article([the,cat...],S1),noun(S1,S2),`

  `verb(S2,S3),article(S3,S4),noun(S4,F)`

  - try: `article([the,cat...],S1)`
  - matches: `article([the|[cat...]],[cat...])`
  - – `S1` instantiated to `[cat...]`
  - `try: noun([cat,saw...],S2)`
  - matches: `noun([cat|[saw...]],[saw...])`
  - – `S2` instantiated to `[saw...]`

Prolog syntax analysis

- try: `verb([saw,a...],S3)`
- matches: `verb([saw|[a...]],[a...])`
- `-` `S3` instantiated to `[a...]`
- try: `article([a,dog],S4)`
- matches: `article([a|[dog]],[dog])`
- `-` `S4` instantiated to `[dog]`
- try: `noun([dog],F)`
- matches: `noun([dog|[]],[])`
- `-` `F` instantiated to `[]`

Definite Clause Grammars

- special Prolog notation for parsing

non-terminal -> *option1* | ... ==>

non-terminal -->option1.

...

- in *optioni*

terminal ==>

[*terminal*]

Definite Clause Grammars

```
article --> [the].
article --> [a].
noun --> [cat].
noun --> [dog].
verb --> [saw].
verb --> [chased].
sentence -->
 article,noun,verb,article,noun.
```

Definite Clause Grammar

- compiled into equivalent Prolog
  - with appropriate input & output arguments for each term
- must call with explicit arguments

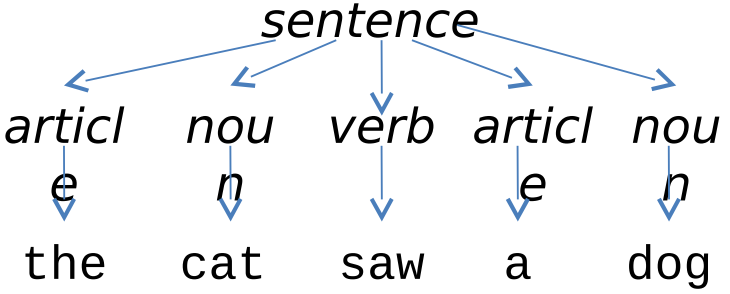
```
| ?- sentence([a,dog,chased, the cat],F).

F = []
```

Syntax trees

- don't just want to know that symbol sequence is well formed
- find out what has been recognised
- build a structure to represent the syntax

```
sentence(article(the),
            noun(cat),
            verb(saw),
            article(a),
            verb(dog))
```



Syntax trees

- add another parameter to the rules to return the corresponding structure
- compiled into equivalent Prolog
- with new parameters at start

`non-terminal(term) --> body ==>`

`non-terminal(term,input,output) --> body`

- where *body* may refer to components of *term*

Syntax trees

```
article(article(the)) --> [the].
article(article(a)) --> [a].
noun(noun(cat)) --> [cat].
noun(noun(dog)) --> [dog].
verb(verb(saw)) --> [saw].
verb(verb(chased)) --> [chased].
sentence(sentence(A1,N1,V,A2,N2)) -->
 article(A1),noun(N1),
 verb(V),article(A2),noun(N2).
```

Syntax trees

```
| ?- sentence(T,[the,cat,saw,a,dog],F).
T = sentence(article(the),noun(cat),
             verb(saw),article(a),noun(dog))
F = []
```

- try: `sentence(sentence(A1,N1,V,A2,N2),`
  `[the,cat...],F) :-`
  `article(A1,[the,cat...],S1),noun(N1,S1,S2),`
  `verb(V,S2,S3),article(A2,S3,S4),noun(N2,S4,F)`
  - try: `article(A1,[the,cat...],S1)`
  - matches: `article(article(the),[the|[cat...]],[cat...])`
  - `-` `A1` instantiated to `article(the)` & `S1` instantiated to `[cat...]`

Syntax trees

- try: `noun(N1,[cat,saw...],S2)`
- matches: `noun(noun(cat),[cat|[saw...]],[saw...])`
- – `S2` instantiated to `[saw...]` & `N1` instantiated to `noun(cat)`
- try: `verb(V,[saw,a...],S3)`
- matches: `verb(verb(saw),[saw|[a...]],[a...])`
- – `S3` instantiated to `[a...]` & `V` instantiated to `verb(saw)`
- try: `article(A2,[a,dog],S4)`
- matches: `article(article(a),[a|[dog]],[dog])`
- – `S4` instantiated to `[dog]` & `A2` insantiated to `article(a)`
- try: `noun(N2,[dog],F)`
- matches: `noun(noun(dog),[dog|[]],[])`
- – `F` instantiated to `[]` & `N2` instantiated to `noun(dog)`

Arithmetic

digit -> `0` | `1` | …

number -> *digit number* | *digit*

base -> `(` *exp* `)` | *number*

term -> *base* `*` *term* | *base* `/` *term* | *base*

exp -> *term*`+` *exp* | *term*`-`*exp* | *term*

e.g. `(12+34)*56`

Arithmetic

```
digit(0) --> [0].
digit(1) --> [1].
digit(2) --> [2].
digit(3) --> [3].
digit(4) --> [4].
digit(5) --> [5].
digit(6) --> [6].
digit(7) --> [7].
digit(8) --> [8].
digit(9) --> [9].
```

```
| ?- digit(T,[1,2,3],L).
T = 1
L = [2,3]
```

Arithmetic

```
number([D|N]) --> digit(D),number(N).
number([D]) --> digit(D).

| ?- number(T,[1,2,3,*,4,5,6],L).
T = [1,2,3]
L = [*,4,5,6]
```

Arithmetic

```
base(n(N)) --> number(N).


| ?- base(T,[1,2,3,+,4,5,6],L).
T = n([1,2,3])
L = [*,4,5,6]
```

Arithmetic

```
term(*(B,T)) --> base(B),[*],term(T).
term(/(B,T)) --> base(B),[/],term(T).
term(B) --> base(B).

| ?- term(T,[1,2,3,*4,5,6,+7,8,9],L).
T = n([1,2,3])*n([4,5,6])
L = [+,7,8,9]
```

Arithmetic

```
exp(+(T,E)) --> term(T),[+],exp(E).
exp(-(T,E)) --> term(T),[-],exp(E).
exp(T) --> term(T).

| ?- exp(T,[1,2,3,*,4,5,6,+,7,8,9],L).
T = n([1,2,3])*n([4,5,6])+n([7,8,9])
L = []
```

Arithmetic

```
base(B) --> ['('],exp(B),[')'].


| ?- exp(T,[‘(‘,1,2,+,3,4,’)’,*,5,6],
          L).
T = (n([1,2])+n([3,4]))*n([5,6])
L = []
```

Evaluate expression tree

- convert list of digits to integer value
- keep track of value so far
- for empty list, value is value so far
- for non-empty list, value is from tail with 10 * value so far + first digit

```
evalNumb([],V,V).

evalNumb([H|T],V1,V) :-

 V2 is 10*V1+H,evalNumb(T,V2,V).

?- evalNumb([1,2,3],0,V).

V = 123
```

Evaluate expression tree

- try: `evalNumb([1|[2,3]],0,V) :-`

  `V2 is 10*0+1,evalNumb([2,3],V2,V).`

  – try: `V2 is 10*0+1`
  – `V2` instantiated to `1`
  – try: `evalNumb([2,3],1,V)`
  – try: `evalNumb([2|[3]],1,V) :-`
  – `V2’ is 10*1+2, evalNumb([3],V2’,V)`

    try: `V2’ is 10*1+2`

    – `V2’` instantiated to `12`

    try: `evalNumb([3],12,V)`

Evaluate expression tree

- try: `evalNumb([3|[]],12,V) :-`
- `V2’’ is 10*12+3,`
- `evalNumb([],V2’’,V)`

try: `V2’’ is 10*12+3`

- `V2’’` instantiated to `123`

try: `evalNumb([],123,V)`

- matches: `evalNumb([],123,123)` – `V` instantiated to `123`

Evaluate expression tree

```
eval(E1+E2,V) :-
 eval(E1,V1),eval(E2,V2),V is V1+V2.
eval(E1-E2,V) :-
 eval(E1,V1),eval(E2,V2),V is V1-V2.
eval(E1*E2,V) :-
 eval(E1,V1),eval(E2,V2),V is V1*V2.
eval(E1/E2,V) :-
 eval(E1,V1),eval(E2,V2),V is V1/V2.
eval(n(N),V) :- evalNumb(N,0,V).
```

Evaluate expression tree

```
run(L,V) :- exp(T,L,_),eval(T,V).


| ?-
run([‘(‘,1,2,+,3,4,’)’,*,5,6],V).
V = 2576
```

Prolog summary: types

- weak, dynamic types
- base types
  - integer, atom
- structured types
  - structure, list
- ad-hoc polymorphism
  - arbitrary types can appear in structures

Prolog summary: data abstraction

- variable
  - name/value association
  - changed by backtracking
  - variable sharing
- memory not visible

Prolog summary: data abstraction

- variable introduction
  - term
- scope
  - lexical
- extent
  - goal of which term is part

Prolog summary: control abstraction

- term
  - abstracts from arithmetic/logic/flow of control sequences
- DB
  - disjunction of facts/rules
- rule body/question
  - conjunction of terms
- pattern matching
  - abstracts from constant matching

Prolog summary: control abstraction

- question/goal/sub-goal
  - analogous to function call
  - binds variables to terms
- recursion
- backtracking
  - reverses variable bindings

Prolog: pragmatics

- higher level than imperative programs
- many to one mapping from expression to machine code
- must be compiled to machine code (or interpreted)
- very succinct
- good correspondence between program structure & data structure
- automatic memory management
  - garbage collection

Prolog: pragmatics

- weak types/static typing/ad-hoc polymorphism
  - space overhead: must have explicit representation on type in memory
  - time overhead: must check types for typed operation
- backtracking
  - space overhead: must keep track of execution history
- garbage collection overhead

Prolog: pragmatics

- very different model to imperative/functional languages
  - long way from von Neumann model
  - key concepts drawn from logic
- claims to be entirely implementation independent but need to understand:
  - variable sharing
  - backtracking

Prolog: pragmatics

- CPU independent
- used for:
  - rapid prototyping
  - AI