Parsing

Overview

- **parser:** program extracting structure from linear sequence of elements
  - e.g. transforming string "3+4*5" to a tree representing the expression
- **domain specific language (DSL):** small programming language for a narrow domain
  - often embedded in existing languages, adding new features particular to the domain, while otherwise using existing functionality
- if DSL can be parsed by extending host language parse, its much more convenient to use
- Prolog handles this well:
  - `read/1` reads a term
  - `op/3` allows you to extend the language by defining new operators

Operator Precedence

- **operator precedence:** simple parsing technique based on operator's:
  - **precedence:** which operator binds tightest
  - **associativity:** if repeated infix operators associate to left/right/neither
    - i.e. $a - b - c$ is $(a - b) - c$ or $a - (b - c)$ or an error
  - **fixity:** infix, prefix, postfix

Prolog Operators

- Prolog's `op/3` predicate declares an operator
  - precedence:
    - larger numbers are lower precedence
    - 1000: goal precedence
  - fixity: 2/3 letter symbol giving fixity and associativity
    - `f`: operator
    - `x`: subterm at lower precedence
    - `y`: subterm at higher precedence
  - operator: operator to declare

```
:- op(precedence, fixity, operator).
```

e.g. Prolog imperative for loop

```
:- op(950, fx, for).
:- op(940, xfx, in).
:- op(600, xfx, '..').
:- op(1050, xfy, do).

for Generator do Body :-
    (   call(Generator),
        call(Body),
        fail
    ;   true
    ).

Var in Low .. High :-
    between(Low, High, Var).

Var in [H|T] :-
    member(Var, [H|T]).
```

Haskell Operators

- simpler and more limited than Prolog
- only supports infix operators
- declare as `associativity precedence operator`
- associativity can be:
  - `infixl`: left associative infix operator
  - `infixr`: right associative infix operator
  - `infix`: non-associative infix operator
- precedence: integer 1-9
  - lower numbers are lower precedence (looser)

e.g. define `%` as synonym for `mod`

```
infix 7 %

(%) :: Integral a => a -> a -> a
a % b = a `mod` b
```

Grammars

- parsing is based on a **grammar** which specifies the language to be parsed
- **terminals:** symbols of the language
- **non-terminals:** specify a linguistic category
- grammar comprised of set of rules

$$(\text{non-terminal} \cup \text{terminal})^* \rightarrow (\text{non-terminal} \cup \text{terminal})^*$$

- most commonly, LHS of arrow is a single non-terminal:

$$\text{expression} \rightarrow \text{expression}'+'\text{expression}\text{expression} \rightarrow \text{expression}'-'\text{expression}\text{expression} \rightarrow \text{expression}'*'\text{expressi}$$

Definite Clause Grammars

- Prolog directly supports **definite clause grammars**, which adhere to the following rules:
  - Non-terminals are written using goal-like syntax
  - Terminals are written between single quotes
  - LHS and RHS separated with `-->`
  - parts on RHS separated with `,`
  - empty terminal: `[]` or `''`
- e.g. expression grammar as Prolog DCG:

```
expr --> expr, '+', expr.
expr --> expr, '*', expr.
expr --> expr, '-', expr.
expr --> expr, '/', expr.
expr --> number.
```

- note this can only test whether a given string is an element of the language
- to produce a **parse tree**, i.e. a data structure representing the input, add arguments to the non-terminals

```
expr(E1+E2) --> expr(E1), '+', expr(E2).
expr(E1*E2) --> expr(E1), '*', expr(E2).
expr(E1-E2) --> expr(E1), '-', expr(E2).
expr(E1/E2) --> expr(E1), '/', expr(E2).
expr(N) --> number(N).
```

Recursive Descent Parsing

- **recursive descent parsing:** DCGs map each non-terminal to a predicate that nondeterministically parses one instance of that non-terminal
- to use a grammar, you use the `phrase/2` predicate: `phrase(nonterminal, string).`
- recursive descent parsing cannot handle left recursion

Left Recursion

- `expr(E1+E2)--> expr(E1), '+', expr(E2).` is left recursive
  - before parsing any terminals, it calls itself recursively
  - as DCGs are transformed to ordinary Prolog code, this becomes a clause that calls itself recursively consuming no input: infinite recursion
- DCGs can be transformed to remove left recursion:
  - rename left recursive rules to `A_rest` and remove the first non-terminal
  - add a rule for `A_rest` matching empty input
  - add `A_rest` to the end of the non-left recursive rules
- DCGs with arguments: non-left recursive rules
  - replace argument of non-left recursive rules with a fresh variable
  - use original argument of `_rest` added non-terminal
  - add fresh variable as second argument of `_rest` added non-terminal e.g.

```
expr(N) --> number(N).
% becomes
expr(E) --> number(N), expr_rest(N, E).
```

- DCGs with arguments: left recursive rules
  - use argument of left-recursive non-terminal as first head argument, and fresh variable as second
  - use original argument of head as first argument of `_tail` call, and fresh variable as second argument of head and `_tail` call

```
expr(E1+E2) --> expr(E1), '+', expr(E2).
% becomes
expr_rest(E1,R) --> '+', expr(E2), expr_rest(E1+E2, R).
```

Disambiguating a grammar

- original grammar is ambiguous: `expr(E1-E2)--> expr(E1), '-', expr(E2).`
  - applied to "3-4-5" allows E1 to be "3-4" or "4-5"
- ensure only desired one is possible by splitting ambiguous non-terminal into separate non-terminals for each precedence level
- becomes (before elimination of left recursion)

```
expr(E-F) --> expr(E), '-' factor(F)
```

Final Grammar

```
expr(E) --> factor(F), expr_rest(F, E).

expr_rest(F1, E) --> '+', factor(F2), expr_rest(F1+F2, E).
expr_rest(F1, E) --> '-', factor(F2), expr_rest(F1-F2, E).
expr_rest(F, F) --> [].

factor(F) --> number(N), factory_rest(N,F).

factor_rest(N1, F) --> '*', number(N2), factor_rest(N1*N2, F).
factor_rest(N1, F) --> '/', number(N2), factor_rest(N1/N2, F).
factor_rest(N, N) --> [].
```

Tokenisers

- **syntax analysis = lexical analysis/tokenising + parsing**
- **lexical analysis:** uses simpler class of grammar to group characters and tokens
  - eliminates meaningless text (whitespace, comments)
- you can use `'strings'` as terminals or lists if you need to
- you can also write normal Prolog code in a DCG wrapped in `{ }`
  - if it fails, the rule fails

```
number(N) -->
    [C],
    { '0' =< C, C =< '9'},
    { N0 is C - '0'},
    number_rest(N0, N).

number_rest(N0, N) -->
```

```
    (   [C],
        { '0' =< C, C =< '9'}
    ->  { N1 is N0 *10 + C - '0' },
        number_rest(N1, N),
    ;   { N = N0}
    ).
```

Working parser

```
?- phrase(expr(E), '3+4*5'), Value is E.
E = 3+4*5,
Value = 23;
false.
```

Extras

- DCGs can run backwards to generate text from structure

```
flatten(empty) --> []
flatten(node(L, E, R)) -->
    flatten(L),
    [E],
    flatten(R).
```

- parsing in Haskell
  - `ReadP`, `Read`, `Parsec`