Parsing

Introduction

- Parsing is the problem of determining whether a given string conforms to the syntax rules of a language
- It is an good application for logic programming, because the rules of a language can be expressed as clauses in a logic program, and (at least in principle) parsing a string amounts to solving a goal with that logic program.

Example: Arithmetic Expressions

- As an example, we shall use the following set of rules for the syntax of arithmetic expressions in the variables x and y:

$$\begin{aligned} expr &::= term \mid term\ \text{`+'}\ expr \mid term\ \text{`−'}\ expr \\ term &::= factor \mid factor\ \text{`*'}\ term \mid factor\ \text{`/'}\ term \\ factor &::= \text{`x'} \mid \text{`y'} \mid \text{`('}\ expr\ \text{`)'} \end{aligned}$$

Example: Arithmetic Expressions…

- The first rule says that an expression (*expr*) may be either a term, or a term followed by a plus sign and another expression, or a term followed by a minus sign and another expression. Thus an expression is a sequence of terms separated by plus and minus signs
- A *term* is a sequence of factors separated by multiplication and division signs
- A *factor* is either a variable ('x' or 'y'), or an expression in parentheses

Example: Arithmetic Expressions…

- The simplest way to translate these rules into a logic program is to make each syntactic class such as *expr* or *term* correspond to a one-argument relation, arranging that *expr (A)* is true if and only if the string (list of characters) *A* forms a valid member of the class *expr* , and so on

Example: Arithmetic Expressions…

- Because one form of expression is simply a term, we can write down the clause

$$expr(A) :- term(A).$$

$$\begin{array}{l} expr(A) :- term(A). \\ expr(A) :- \\ \quad append(B, C, A), term(B), \\ \quad append(\text{“+”}, E, C), expr(E). \\ expr(A) :- \\ \quad append(B, C, A), term(B), \\ \quad append(\text{“−”}, E, C), expr(E). \end{array}$$

Example: Arithmetic Expressions…

$$\begin{array}{l} term(A) :- factor(A). \\ term(A) :- \\ \quad append(B, C, A), factor(B), \\ \quad append(\text{“*”}, E, C), term(C). \\ term(A) :- \\ \quad append(B, C, A), factor(B), \\ \quad append(\text{“/”}, E, C), term(C). \end{array}$$

$$\begin{array}{l} factor(\text{“x”}) :- . \\ factor(\text{“y”}) :- . \\ factor(A) :- \\ \quad append(\text{“(”}, B, A), append(C, \text{“)”}, B), expr(C). \end{array}$$

Example: Arithmetic Expressions…

- Another possibility for an expression is a term followed by a plus sign and another expression. This can be expressed using the *append* relation:

$$\begin{array}{l} expr(A) :- \\ \quad append(B, C, A), term(B), \\ \quad append(D, E, C), D = \text{“+”}, expr(E). \end{array}$$

Example: Arithmetic Expressions…

- To be a valid expression of this kind, a string a must split into two parts B and C, where B is a valid term, and C consists of a plus sign followed by another expression
- This last condition is expressed using another instance of *append*.
- Fixed symbols like '+' and 'x' can be translated by constant strings
- A useful notation uses double quotes for strings, so that:
  - "+" means '+':*nil*
  - "mike" means 'm':'i':'k':'e':*nil*

Example: Arithmetic Expressions…

- Using this notation, we can translate the whole set of rules to give the logic program shown above
- This translation is correct in a logical sense, but it is very inefficient when run as a program
- For example, to parse the string "x*y+x", we must use the second clause for *expr* , splitting the string into a part "x*y" that satisfies *term*, and a part "+x" that is a plus sign followed by an *expr*

Example: Arithmetic Expressions…

- The Prolog strategy uses backtracking to achieve this, splitting the input string at each possible place until it finds a split that allows the rest of the clause to succeed. This means testing each of the strings " ", "x", "x*" with the relation *term*, before finally succeeding with "x*y"
- Testing the subgoal *term*("x*y") leads to even more backtracking, so the whole process is extremely time-consuming

Example: Difference Lists

- An equivalent but more effective translation uses a technique called *difference lists* to eliminate the calls to *append* and drastically cut down the amount of backtracking
- The idea is to define a new relation *expr2* (A,B) that is true if the string a can be split into two parts: the first part is a valid expression, and the second part is the string B. This relation could be defined by the single clause

$$expr2(A, B) :- append(C, B, A), expr(C).$$

Example: Difference Lists…

- But we can do better than this by defining *expr2* directly, without using *append* or *expr* . For example, the second clause for *expr* leads to this clause for *expr2* :

$$expr2(A, D) :- term2(A, B), eat(\text{‘+’}, B, C), expr2(C, D).$$

- Here we have used a relation *term2* that is related to *term* as *expr2* is related to *expr* , and a special relation *eat*

Example: Difference Lists…

- The whole clause can be read like this: to chop off an expression from the front of A, first chop off a term to give a string B, then chop off a plus sign from B to give a string C, and finally chop of an expression from C to give the remainder D
- The technique is called ‘difference lists’ because the pair (A,D) represents a list of characters that is the difference between A and D
- The relation *eat* is defined by the single clause

$$eat(X, A, B) :- A = X{:}B.$$

Example: Difference Lists…

- It is true if the string B results from chopping off the single character X from the front of A.
- Other rules can be re-formulated in a similar way. For example, the rule

$$factor ::= \text{‘(’}\ expr\ \text{‘)’}$$

can be re-formulated as

$$factor2(A, D) :- eat(\text{‘(’}, A, B), expr2(B, C), eat(\text{‘)’}, C, D).$$

Example: Difference Lists…

- Below is the complete set of rules translated in this style. In order to test a string such as “(x+y)-x” for conformance to the syntax rules, we formulate the query $\# :- expr2(\text{“x*y+x”}, \text{“”}).$

$$\begin{aligned}
&expr2(A, B) :- term2(A, B).\\
&expr2(A, D) :- term2(A, B), eat(\text{‘+’}, B, C), expr2(C, D).\\
&expr2(A, D) :- term2(A, B), eat(\text{‘-’}, B, C), expr2(C, D).\\
\\
&term2(A, B) :- factor2(A, B).\\
&term2(A, D) :- factor2(A, B), eat(\text{‘*’}, B, C), term2(C, D).\\
&term2(A, D) :- factor2(A, B), eat(\text{‘/’}, B, C), term2(C, D).\\
\\
&factor2(A, B) :- eat(\text{‘x’}, A, B).\\
&factor2(A, B) :- eat(\text{‘y’}, A, B).\\
&factor2(A, D) :- eat(\text{‘(’}, A, B), expr2(B, C), eat(\text{‘)’}, C, D).
\end{aligned}$$

Example: Difference Lists…

- This asks whether it is possible to chop off an expression from the front of "x*y+x" and leave the empty string; in other words, whether "x*y+x" is itself a valid expression
- Solving this goal involves backtracking among the different rules, but much less than before

Example: Expression Trees

- In applications such as compilers, it is useful to build a tree that represents the structure of the input program. In our example of arithmetic expressions, we might represent the expression "x*y+x" by the term

$$add(multiply(vbl(x), vbl(y)), vbl(x)).$$

- Representing the expression like this makes it easy to evaluate it for given values of x and y, or to translate it into machine code in a compiler

Example: Expression Trees…

- We can extend our parser so that it can build a tree like this, in addition to checking that a string obeys the language rules
- We extend the relation *expr2* (A,B) into a new relation *expr3* (T,A,B) that is true if the difference between string A and string B is an expression represented by T.
- One clause in the definition of *expr3* is this:

$$\begin{array}{l} expr3(add(T_1, T_2), A, D) :- \\ \quad term3(T_1, A, B), eat(\text{'+'}, B, C), expr3(T_2, C, D). \end{array}$$

Example: Expression Trees…

- As before, this says that an expression may have the form *term* '+' *expr* .
- The added information is that if the term on the left of '+' is represented by the tree T1, and the expression on the right is represented by T2, then the whole expression is represented by the tree *add*(T1,T2).

Example: Expression Trees…

- Other clauses in the parser can be augmented in similar ways
- One clause allows an expression in parentheses to be used as a factor; it turns into the new clause

$$\begin{array}{l} \mathit{factor3}(T, A, D) :- \\ \quad \mathit{eat}(\text{`('}, A, B), \mathit{expr3}(T, B, C), \mathit{eat}(\text{`)'}, C, D). \end{array}$$

Example: Expression Trees…

- The tree for the whole factor is the same as the tree for the expression inside
- In this way, we can be sure that parentheses have no effect on the 'meaning' of an expression, except insofar as they affect the grouping of operators

Example: Expression Trees…

- Once the whole parser has been augmented in this way, we can use it to analyse strings and build the corresponding tree. For example, the goal

$$\# :- \mathit{expr3}(T, \text{“x*(y+x)”}, \text{“”}).$$

will succeed, with the answer

$$T = \mathit{multiply}(\mathit{vbl}(x), \mathit{add}(\mathit{vbl}(y), \mathit{vbl}(x))).$$

Example: Expression Trees…

- Rather unusually, the parser can also be used 'backwards', producing a string from a tree. For example, the goal

$$\# :- \mathit{expr3}(\mathit{add}(\mathit{vbl}(x), \mathit{multiply}(\mathit{vbl}(x), \mathit{vbl}(y))), A,).$$

has several answers, and the first one found by Prolog is a = “x+x*y”

- The other answers have extra parentheses added around various sub-expressions.
- This 'unparsing' function might be useful for generating error messages in a compiler, or for saving expression trees in a text file so they could be parsed again later

Example: Expression Trees…

- The parser for expressions has an unfortunate flaw. The expression “x-y-x” would be assigned the tree

$$subtract(vbl(x), subtract(vbl(y), vbl(x))),$$

that is, the same tree as would be assigned to the expression “x-(y-x)”.

- This is wrong, because the usual convention is that operators ‘associate to the left’, so the correct tree would be

$$subtract(subtract(vbl(x), vbl(y)), vbl(x)),$$

the same as for the expression “(x-y)-x”.

Example: Expression Trees…

- The problem is with the syntax rule

$$expr ::= term \text{ ‘−’ } expr,$$

and others like it

- This rule suggests that where several terms appear interspersed with minus signs, the most important operator is the leftmost one
- The other minus signs must be counted as part of the *expr* in this rule, not part of the *term*, because a term cannot contain a minus sign except between parentheses

Example: Expression Trees…

- We could correct the syntax rules by replacing this rule with

$$expr ::= expr \text{ ‘−’ } term,$$

but unfortunately this would lead to the clause

$$expr(A, D) :- expr(A, B), eat(\text{‘−’}, B, C), term(C, D).$$

- This clause behaves very badly under Prolog’s left-to-right strategy, because a call to *expr* leads immediately to another call to *expr* that contains less information

Example: Expression Trees…

- For example, the goal *expr* (“x-y”, “ ”) immediately leads to the subgoal *expr* (“x-y”,b), and so to an infinite loop
- This is called *left recursion*, because the body of the rule for *expr* begins with a recursive call
- Left recursion causes problems for top–down parsing methods like the one that naturally results from Prolog’s goal-directed search strategy

Example: Expression Trees…

- The solution to this problem is to rewrite the grammar, avoiding left recursion
- The following syntax rules are equivalent to our original ones, in that they accept the same set of strings:

```
    expr ::= term exprtail
exprtail ::= empty | '+' term exprtail | '−' term exprtail
    term ::= factor termtail
termtail ::= empty | '*' factor termtail | '/' factor termtail
  factor ::= 'x' | 'y' | '(' expr ')'
```

Example: Expression Trees…

- The idea here is that an *exprtail* is a sequence of terms, each preceded by a plus or minus sign
- In order to build the tree for an expression, we translate the rules for *exprtail* into a four-argument relation *exprtail*(T1,T,A,B) that is true if the difference between A and B is a valid instance of *exprtail* , and t is the result of building the terms onto the tree T.
- By building on the terms in the right way, we obtain the correct tree for each expression. The complete translation of the new set of rules is shown in the figure below

Example: Expression Trees…

```
exprtail(T1, T1, A, A) :- .
exprtail(T1, T, A, D) :-
    eat('+', A, B), term(T2, B, C),
    exprtail(add(T1, T2), T, C, D).
exprtail(T1, T, A, D) :-
    eat('-', A, B), term(T2, B, C),
    exprtail(subtract(T1, T2), T, C, D).

term(T, A, C) :-
    factor(T1, A, B), termtail(T1, T, B, C).

termtail(T1, T1, A, A) :- .
termtail(T1, T, A, D) :-
    eat('*', A, B), factor(T2, B, C),
    termtail(multiply(T1, T2), T, C, D).
termtail(T1, T, A, D) :-
    eat('/', A, B), factor(T2, B, C),
    termtail(divide(T1, T2), T, C, D).

factor(vbl(x), A, B) :- eat('x', A, B).
factor(vbl(y), A, B) :- eat('y', A, B).
factor(T, A, D) :-
    eat('(', A, B), expr(T, B, C), eat(')', C, D).
```

END