Prolog Definite Clause Grammar Example

Benjamin McLemore

ACO 240: Introduction to Programming Languages

Dr. Suzanne Dietrich

April 30, 2020

Honors Contract

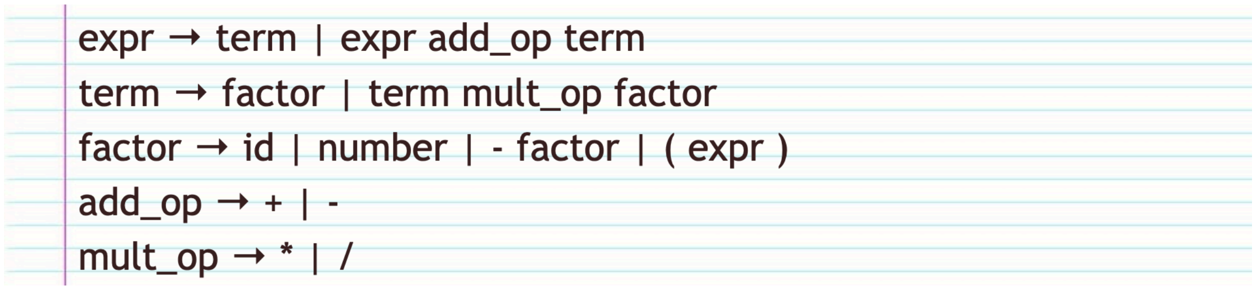
Overview

Introduction to Programming Languages is a class that taught the differences between the varieties of programming-language paradigms with examples from the three most popular. The first several weeks were dedicated to learning the terminology used to describe programming languages and to understanding at an intellectual level the difference between the various paradigms. Included in this section was the specification of the syntax of a language using grammars, for which students created a parse tree for a simple expression using a context-free grammar. The languages introduced were, in order chronologically: the imperative language Python, the functional language Erlang, and the declarative language Prolog. The latter involves defining logical predicates, which can be queried to find matches for gaps in a statement and to determine if a statement is true.

The Prolog programming language also provides inherent support for definite-clause grammar (DCG) rules for parsing statements. This honors contract created some of these DCG rules to draw a parse tree for a given mathematical expression, which is possible because of SWI-Prolog's inherent support for parse trees.

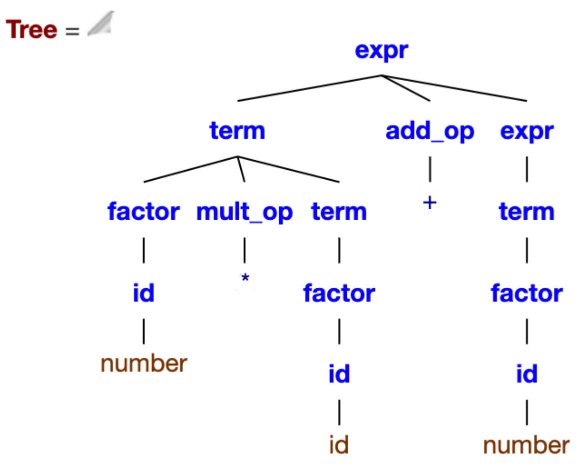
Grammars

A context-free grammar is a set of rules consisting of non-terminals and terminals that defines a grammar. A non-terminal is a statement that can be replaced with a list of one or more terminals or non-terminals. A terminal can't be replaced with anything. In the example context-free grammar below, the non-terminal *factor* can be replaced with either the terminal *id*, the terminal *number*, the terminal - and the non-terminal *factor*, or the terminal *(*, the non-terminal *expr*, and the terminal *)*. *expr*, *term*, *factor*, *add_op*, and *mult_op* are all non-terminals because they have something that can replace them, whereas *id*, *number*, *(*, *)*, +, -, *, and / are all terminals because they have nothing to replace them.



```
expr → term | expr add_op term
term → factor | term mult_op factor
factor → id | number | - factor | ( expr )
add_op → + | -
mult_op → * | /
```

To show how a particular statement follows a context-free grammar's rules, a parse tree can be drawn for that statement. In a parse tree, line are drawn underneath non-terminals to show what they are replaced with. For example, in the parse tree below which represents how the expression *number*id+number* follows the above context-free grammar's rules, the highest *expr* is connected to *term*, *add_op*, and *expr*, which replace it. Through these cycles of replacement, the expression is eventually reached. If the expression cannot be reached, then it doesn't follow the context-free grammar's rules. A parse tree is important because it shows which operations take priority, that being those lower in the tree.



Prolog's Definite Clause Grammars

In a Prolog DCG representing a context-free grammar, terminals are represented by a list of atoms, and non-terminals are represented by predicates whose parameters are variables that represent potential replacements for the non-terminal. Unlike in the example context-free grammar given above, a separate rule must be created in Prolog to represent each possible replacement for a non-terminal, because each predicate takes a different type of parameter depending on what it is trying to match. For example, *term* either takes a tuple with one variable or a tuple with three. Each predicate, which represents a rule associated with a non-terminal, takes as a parameter a tuple containing either variables representing a non-terminal or a terminal, which altogether represent the replacement for the non-terminal. This ensures that the result of parsing the list is a tuple of nested tuples, which is needed for SWI-Prolog to draw the parse tree. Below is the Prolog DCG that represents the context-free grammar given above.

```prolog
expr(expr(TERM)) --> term(TERM).
expr(expr(EXPR, ADD_OP, TERM)) --> expr(EXPR), add_op(ADD_OP), term(TERM).

term(term(FACTOR)) --> factor(FACTOR).
term(term(TERM, MULT_OP, FACTOR)) --> term(TERM), mult_op(MULT_OP), factor(FACTOR).

factor(factor(number)) --> [number].
factor(factor(id)) --> [id].
factor(factor('-', FACTOR)) --> ['-'], factor(FACTOR).
factor(factor('(', EXPR,')')) --> ['('], expr(EXPR), [')'].

add_op(add_op('-')) --> ['-'].
add_op(add_op('+')) --> ['+'].

mult_op(mult_op('*')) --> ['*'].
mult_op(mult_op('/')) --> ['/'].
```

In order for Prolog to be able to determine whether the expression above is valid, it needs to be represented in the form [number, *, id, +, number], which is a list of the atoms that make up the expression. To get the nested tuples needed to draw the parse tree, Prolog has a built-in predicate *phrase* that can be queried with the first parameter being a tuple that contains the variable that is to represent the tree with its functor being the first non-terminal, and the second parameter being the expression. *phrase* is shown below. In order to turn the resulting tuple into a parse tree, SWI-Prolog's built-in tree feature is called.

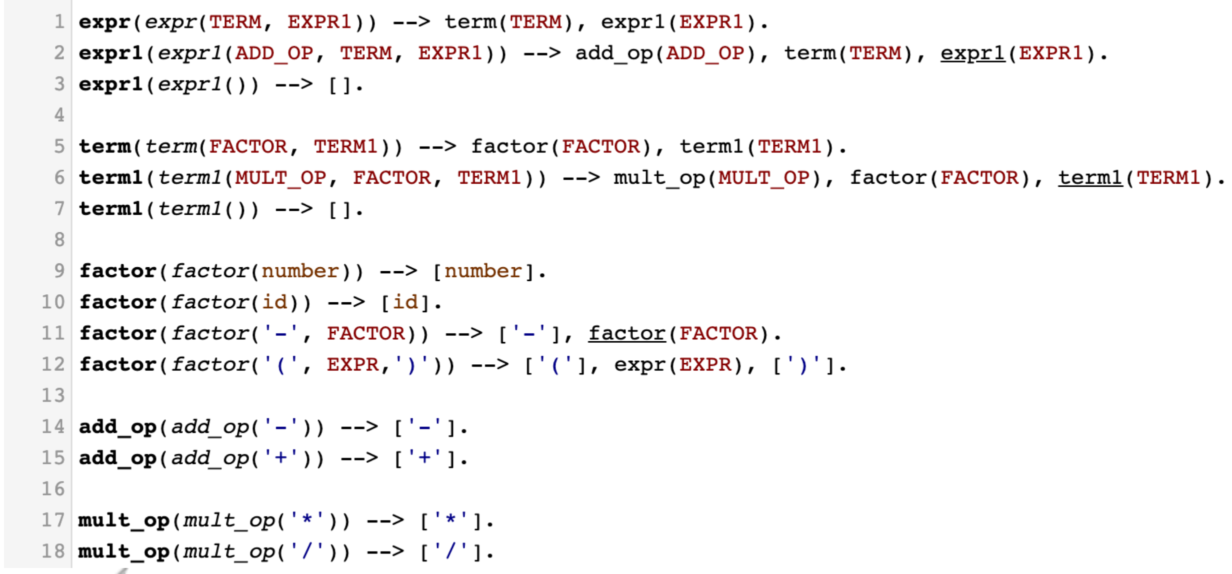
```
?- phrase(expr(Tree), [number, *, id, +, number]).
```

Left Recursion

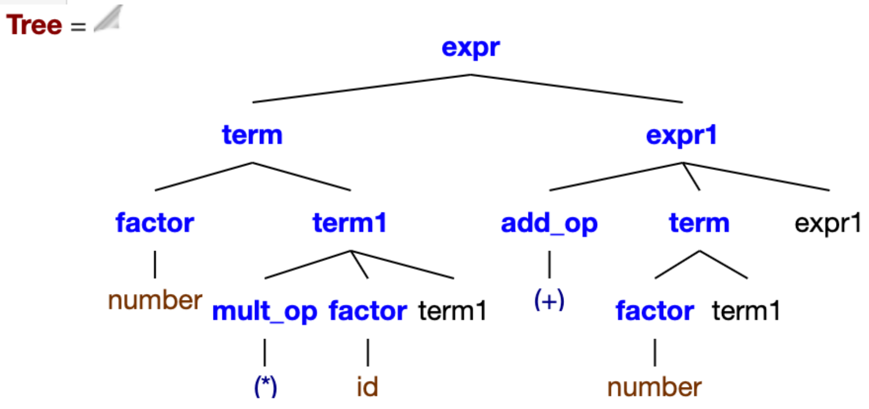
Unfortunately, this DCG wouldn't be able to parse the expression. This is due to left-hand recursion, which is a by-product of the fact that Prolog evaluates the leftmost predicate in the body before any of the others. This means that if any of the predicates recursively call themselves before anything else, Prolog will enter an infinite loop and will thus never reach the final expression. The problematic non-terminals in the example code above are *expr* and *factor*, each of which has a rule which recursively calls itself first. In order to fix this problem, the rules can either be left-factored or the left-recursion can be changed to right-recursion, that is the recursive element can be moved to the end of the body.

Left-factoring is the process by which a set of DCG rules is changed into several others such that it continues mean the same thing while eliminating left-recursion. This is done by defining the recursive element as A, the rest of the body as α, and the body of the rule without a recursive call as β. These are then rearranged into the following rules: $A \rightarrow \beta A^1$, $A^1 \rightarrow \alpha A^1$, and $A^1 \rightarrow \epsilon$, where ϵ is an empty body. Left-factoring the code above produced the following code and tree when queried with the example expression.

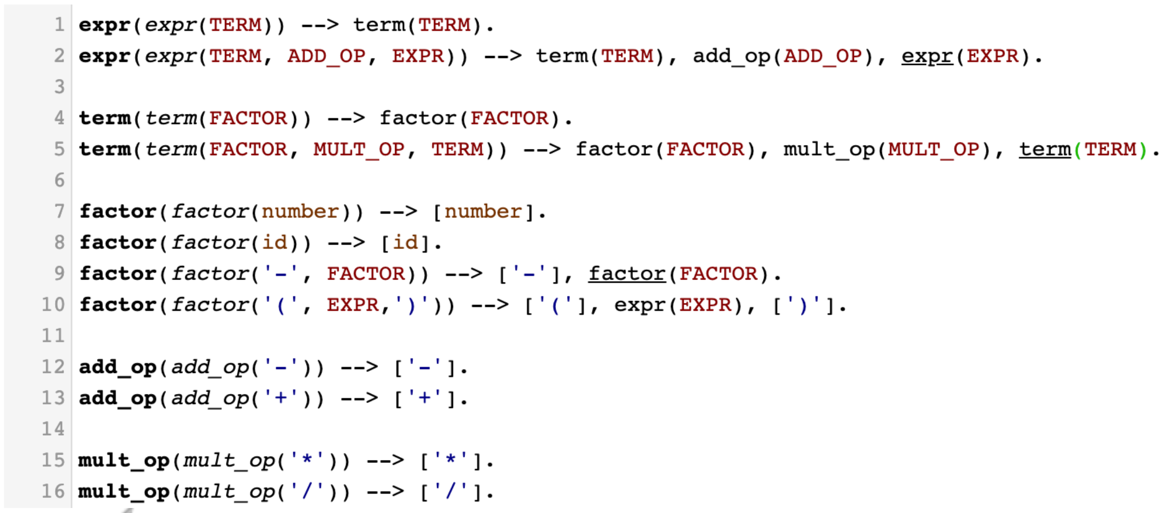
<LEFT FACTORED CODE>



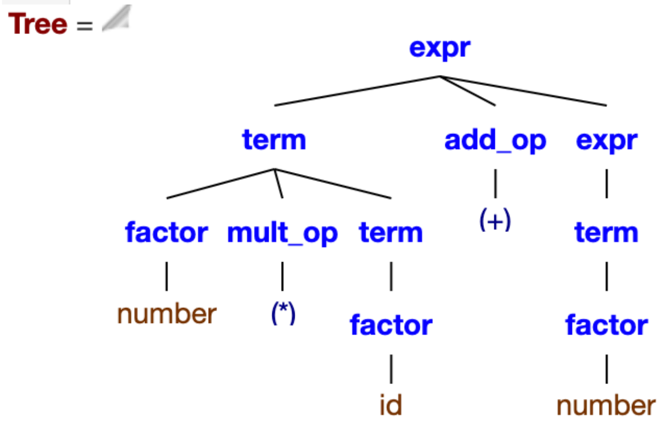
```
expr(expr(TERM, EXPR1)) --> term(TERM), expr1(EXPR1).
expr1(expr1(ADD_OP, TERM, EXPR1)) --> add_op(ADD_OP), term(TERM), expr1(EXPR1).
expr1(expr1()) --> [].

term(term(FACTOR, TERM1)) --> factor(FACTOR), term1(TERM1).
term1(term1(MULT_OP, FACTOR, TERM1)) --> mult_op(MULT_OP), factor(FACTOR), term1(TERM1).
term1(term1()) --> [].

factor(factor(number)) --> [number].
factor(factor(id)) --> [id].
factor(factor('-', FACTOR)) --> ['-'], factor(FACTOR).
factor(factor('(', EXPR,')')) --> ['('], expr(EXPR), [')'].

add_op(add_op('-')) --> ['-'].
add_op(add_op('+')) --> ['+'].

mult_op(mult_op('*')) --> ['*'].
mult_op(mult_op('/')) --> ['/'].
```



The other method of dealing with left-recursion is to change all instances of left recursion into right recursion. Because prolog checks the rightmost predicates in the body last, this doesn't cause any problems. In order to do this, the recursive element is switched with a non-recursive element on the left such that the rule is still correct. In this instance, that means switching *expr* and *term* in *expr*, and *term* and *factor* in *term*. This produces the following code and tree when queried.



```
expr(expr(TERM)) --> term(TERM).
expr(expr(TERM, ADD_OP, EXPR)) --> term(TERM), add_op(ADD_OP), expr(EXPR).

term(term(FACTOR)) --> factor(FACTOR).
term(term(FACTOR, MULT_OP, TERM)) --> factor(FACTOR), mult_op(MULT_OP), term(TERM).

factor(factor(number)) --> [number].
factor(factor(id)) --> [id].
factor(factor('-', FACTOR)) --> ['-'], factor(FACTOR).
factor(factor('(', EXPR,')')) --> ['('], expr(EXPR), [')'].

add_op(add_op('-')) --> ['-'].
add_op(add_op('+')) --> ['+'].

mult_op(mult_op('*')) --> ['*'].
mult_op(mult_op('/')) --> ['/'].
```

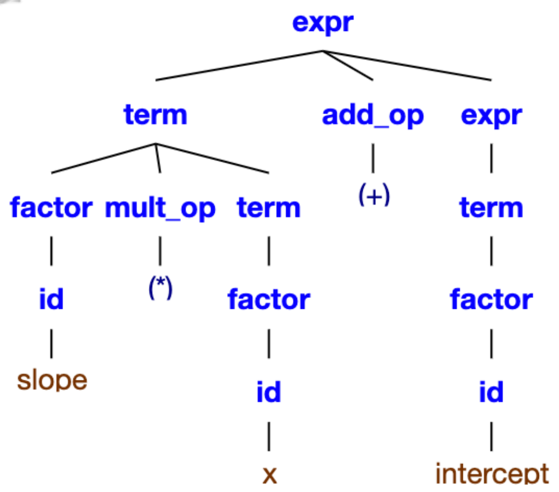


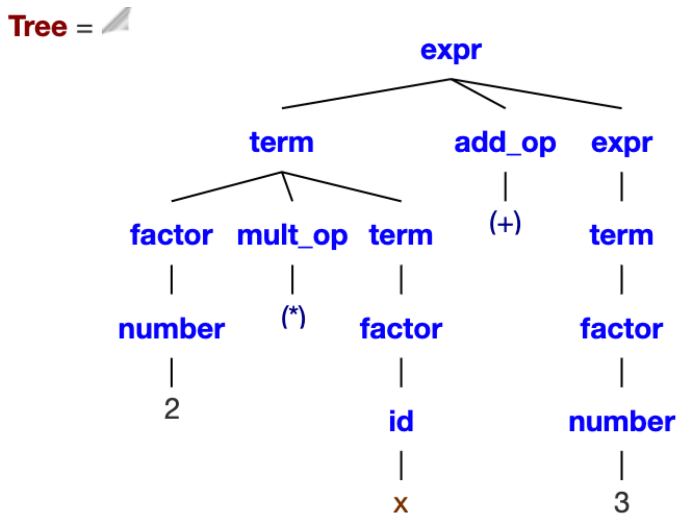
Predicates in Definite Clause Grammars

This is an accurate tree, but it can only represent expressions with *number* and *id* atoms rather than actual numbers and variables. To fix this, *number* and *id* can be turned into non-terminals to be replaced by actual numbers and variables. In order to ensure that *number* is replaced by numbers and *id* is replaced by variable names in the form of atoms, curly braces can be used to call predicates within the rules to ensure that *number* represents numbers and *id* represents atoms. This produced the following code and two trees, the first with the example expression *slope*x+intercept* and the second with *2*x+3* to show its versatility. In the former, *slope*, *x*, and *intercept* are all connected to the tree by *id*, but in the latter, *2* and *3* are connected to the tree by *number*.

```
expr(expr(TERM)) --> term(TERM).
expr(expr(TERM, ADD_OP, EXPR)) --> term(TERM), add_op(ADD_OP), expr(EXPR).

term(term(FACTOR)) --> factor(FACTOR).
term(term(FACTOR, MULT_OP, TERM)) --> factor(FACTOR), mult_op(MULT_OP), term(TERM).

factor(factor(NUMBER)) --> number(NUMBER).
factor(factor(ID)) --> id(ID).
factor(factor('-', FACTOR)) --> ['-'], factor(FACTOR).
factor(factor('(', EXPR,')')) --> ['('], expr(EXPR), [')'].

add_op(add_op('-')) --> ['-'].
add_op(add_op('+')) --> ['+'].

mult_op(mult_op('*')) --> ['*'].
mult_op(mult_op('/')) --> ['/'].

number(number(NUMBER)) --> [NUMBER], {integer(NUMBER)}.
id(id(ID)) --> [ID], {atom(ID)}.
```

Tree =





Conclusion

This honors contract detailed the exploration of using Prolog's definite clause grammar rules to represent a context-free grammar. During the research, various issues came up that required problem solving and solutions which themselves provided a more nuanced understanding of context-free grammars, DCGs, and Prolog. The result of this work was a DCG that can be used to parse the mathematical expression and draw a parse tree.