CS 246E Fall 2021 — Tutorial 7

October 27, 2021

Summary

1 Recursive Descent Parsing: Foundations

1.1 What is parsing?

Parsing involves taking a string representing some sort of data structure (usually a tree) and programmatically turning it into that data structure. Parsing is an important problem in Computer Science and many important programs, such as compilers, rely on efficient parsing algorithms to function effectively.

There are a number of parsing algorithms available to parse various inputs, many of which are fairly sophisticated and surprisingly efficient: however, we'll leave those to compilers courses (such as CS241 and CS444) and formal languages courses (such as CS360 and CS462). For our purposes, we'll focus on one of the simplest parsing algorithms in existence: recursive descent.

1.2 What is recursive descent parsing?

Recursive descent parsing is essentially the first thing you might try if you needed to write a parsing algorithm and had no better idea how to go about doing it. A series of mutually recursive functions parse the input, with each function producing a node in the tree. Information about what parts of the string go in which node is tracked implicitly in the call stack.

1.3 Terminology

- The *language* is a set of strings
- Every string in that set is also known as a *word* in that *language*
- A *word* is made up of *symbol*s
- The *grammar* is a set of rules expressing organization and placement of *symbol*s such that they form valid *word*s in the *language*

1.4 Specifying our language: the basics

For more sophisticated grammars, we might want to make sure to use a formal system for expressing our language (such as a context-free grammar). However, for simple grammars we can simply express the grammar via a set of English sentences. For example, if we wanted to specify the language of all rational numbers, we could say that:

A rational number is any of

- A natural number
- Two natural numbers seperated by a "."
- A natural number, followed by a ".", followed by infinitely many copies of a natural number

and nothing else is a rational number

1.4.1 Exercise

Try specifying the language of natural numbers which have no leading 0s (that is, the only number whose first digit is 0 is 0 itself).

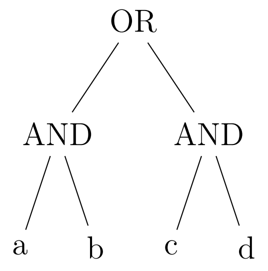
Describe this language as a regular expression.

As an aside, not all languages can be expressed as a regular expression, such as the language of rational numbers described above.

2 Recursive Descent Parsing: Trees

2.1 Parse Trees and Abstract Syntax Trees

At its core, a parse tree is a proof that a word is in a language. While we could prove that a word is in a language in any number of ways, one of the obvious ways to do so is to view the word as being partitioned into pieces by each rule describing the language. This can be efficiently represented in tree-based form and is called a parse tree. For example, consider a language of boolean formulas involving boolean variables represented using lowercase English alphabet characters and operators `AND` and `OR`. A parse tree proving that `a AND b OR c AND d` is in the language might look as follows:



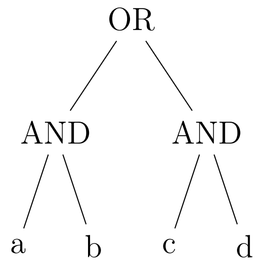
Since tree operations are a natural and foundational part of computer science, parse trees are also a very effective means of programmatically manipulating a word in a syntax-driven way, so as long as we pick a language syntax which "highlights" the interesting parts of the language a parse tree can go a long way towards helping us process it.

We usually don't actually need a full proof of the language, and it may be helpful to get rid of some information in the string which was used purely to make sure we knew how to interpret the language rules correctly but is no longer useful. For example, once we've figured out how to group operations, parentheses are no longer useful. The parse tree with these useless operations removed is called an abstract syntax tree within the compilers community, and is usually what we want to do further operations on. Despite the name "parsing," usually by the end of the parsing process we want an abstract syntax tree rather than a parse tree, since extra information like parentheses only make programmatically manipulating the parse tree more complicated.

Note that parse trees and abstract syntax trees do not need to be binary trees, and are usually general trees with an arbitrary number of children instead.

2.2 Parse Trees: Precedence and Associativity

The easiest way to understand how parse trees enforce precedence/associativity is to look at how a parse tree is used to evaluate an expression. Take the parse tree in the example above



Evaluating this tree is done in a bottom up fashion, the value of `a` and `b` are put together to create the value of `a AND b`, the value of `c` and `d` are put together to create the value of `c AND d`, and finally, the value of `a AND b` and `c AND d` are put together to create the value of `a AND b OR c AND d`. Using parenthesis to show precedence, this expression looks like

```
( ( ( a ) AND ( b ) ) OR ( ( c ) AND ( d ) ) )
```

Generalizing this, it means that nodes deeper in the tree have higher precedence than nodes at previous levels.

2.3 Specifying our language: precedence and associativity

The grammar for rational numbers defined above was very simple. However, we might need to be a bit more careful when working with more complicated examples. Take the example of boolean formulas above, call $\mathcal{L}$ the language of `AND`s and `OR`s over variables `a`; `b`; `c`, etc.

Furthermore, like in C++ each operator will have a precedence: that is, AND has higher precedence than OR, so `a AND b OR c` means `(a AND b) OR c`, not `a AND (b OR c)`. Finally, our operations will be right-associative: `a OR b OR c` will be interpreted as `a OR (b OR c)`, not `(a OR b) OR c`. Thus we can describe $\mathcal{L}$ as follows:

A word in $\mathcal{L}$ is any of

- A variable, represented by a single lowercase letter of the English alphabet
- α `OR` β, where α, β are words in $\mathcal{L}$ where α has no `OR`s in it
- α `AND` β, where α, β are words in $\mathcal{L}$ where α has no `OR`s and no `AND`s in it, and β has no `OR`s in it

and nothing else is a word in $\mathcal{L}$

2.3.1 Exercise

Try to describe the language $\mathcal{L}$ of all mathematical formulas over $\{+, -, *, /, \hat{}, !\}$ in the format of the example above. The language should have the following properties:

1. ^and ! should have the highest precedence. For example, $1 + 2\hat{}3/4$ should be interpreted as $1 + (2\hat{}3)/4$
2. * and / have the next-highest precedence. For example, $1+2*3-4$ should be interpreted as $1+(2*3)-4$
3. + and - have the lowest precedence
4. All operators are left-associative
5. Parenthese group operations in the usual way and override all precedence. For example, $(1 + 2)/3 = 3/3 = 1$
6. All operands are natural numbers

Note, we consider invalid mathematical operations such as 5.5! and 2/0 still as a mathematical formula. In general, it is not the job of the parser analyze the meaning of what is parsed.

It might be helpful to describe the natural numbers as a separate language

2.3.2 Exercise

Draw abstract syntax trees for the following words in the language from the previous exercise:

- $3 + 4 - 5\hat{}6$
- $(3 + 2)\hat{}(4/5)$

3 Programatically Representing ASTs

Now that we have inheritance in C++, programmatically representing ASTs can be done in a very natural way: we follow essentially the same basic strategy as we did to build a binary search tree, but now the nodes in a tree occupy an inheritance hierarchy and may have more or less than two children. We will also make a simplification and use `|` to represent `OR`, and `&` to represent `AND`.

For example, we could represent our tree for boolean expressions and add a few operations to it as follows:

```
class Expr {
public:
  // Pretty-print the expression, surround every binary operation with parenthesis
  virtual std::string prettyPrint() const = 0;
  // Set a specified variable to either true or false
  virtual void setValue( char var , bool value ) = 0;
  // Evaluate the expression ( unset variables default to false )
  virtual bool evaluate() const = 0;

  class Or;
  class And;
  class Var;
};

class Expr::Or : public Expr {
  std::unique_ptr<Expr> left;
  std::unique_ptr<Expr> right;
public:
  Or( std::unique_ptr<Expr> left, std::unique_ptr<Expr> right ):
    left{ std::move( left ) }, right{ std::move( right ) } {}
  std::string prettyPrint() const override {
    return "(" + left->prettyPrint() + "|" + right->prettyPrint() + ")";
  }
  void setValue( char var , bool value ) override {
    left->setValue( var, value );
    right->setValue( var, value );
  }
  bool evaluate() const override {
    return left->evaluate() || right->evaluate();
  }
};

// And and Var similar
```

3.1 Exercise

Write a set of classes which represent an AST for your language of mathematical formulas above. Include methods to pretty print it in "reverse polish notation" (where an operator follows its arguments) and to evaluate it (where ! is factorial and ^is exponentiation).

4 Recursive Descent parsing: the algorithm

Now that we have the pieces in place, the actual recursive descent algorithm is relatively simple: each rule in our English description of our grammar corresponds to a recursive function which produces trees of that type. Following the Boolean expression example we might have the following code:

```
class no_matching_expression {}; // exception class

// where curr is the current location in the input
std::unique_ptr<Expr> Or( const std::string& input, size_t& curr );
std::unique_ptr<Expr> And( const std::string& input, size_t& curr );
std::unique_ptr<Expr> Var( const std::string& input, size_t& curr );
```

Each function should recursively check whether the appropriate case applies. If it does, the function should consume input and update `curr` to reflect that, otherwise throw an exception to indicate that the attempt to apply this rule failed.

Each rule may have many valid forms, for example the left hand side of an `OR` could be an `AND` expression or a variable, and the right hand side could be an `OR` expression, an `AND` expression, or a variable. We would need to try all combinations, and only throw an exception if no combination is valid. The pseudo-code for `OR` looks something like

```
if input starting at curr matches an AND expression
    consume input corresponding to the AND expression by updating curr
else if input starting at curr matches a variable
    consume input corresponding to the variable by updating curr
else
    throw exception
fi

if can consume '|'
    consume '|' by updating curr
else
    throw exception
fi

if input starting at curr matches an OR expression
    consume input corresponding to the OR expression by updating curr
```

```
else if input starting at curr matches an AND expression
    consume input corresponding to the AND expression by updating curr
else if input starting at curr matches a variable
    consume input corresponding to the variable by updating curr
else
    throw exception
fi
```

In implementing this pseudo-code, we will assume away whitespace in the input (for example, we won't have `a |    b &   c`, but just `a|b&c`).

This is only an incomplete implementation of the algorithm, here are some of the other things to consider:

- Implementation of `And()`
- Implementation of `Or()`
- Helpers to simplify the code (you can get this down to just four lines with the right helpers)
- An entry point to the algorithm for the client to use
- Preprocessing (recall the assumption that we made earlier), can be done at the entry point part
- Additional error checking (just because `Or()` doesn't throw an error, doesn't mean the entire input is an `OR` expression, think about why)

4.1 Exercise

Write the recursive descent parsing functions for your language of mathematical formulas above.

Implementation warning: The language of mathematical formulas is known as "left recursive" which means if we read left to right, recursion on the same function may occur before any input is consumed, which will lead to infinite recursion. There is an iterative solution for this particular issue which you are free to look up and try to implement, but the easiest way to solve this problem is to read right to left, which will change the recursive structure of the algorithm so that it does not interfere with the recursive structure of the language.