Abstract Syntax Trees

https://courses.missouristate.edu/anthonyclark/333/
Outline

Topics and Learning Objectives

• Know the difference between a parse tree and an AST
• Learn how to draw an AST
• Learn how to implement an AST in Rust

Assessments

• Drawing an AST
Convert Grammar Into Source Code

1. Each rule defined in the grammar becomes a function/method
2. References to that rule become a function call
3. The body of a rule function follows the flow of the rule’s RHS
4. Alternatives (a1 | a2 | ...) are handled with branches (if-then-else, switch statements, match expressions, etc.)
5. An optional grouping (...)* becomes a while loop that can loop zero or more times
6. Epsilon transitions enable a function to return without doing anything
7. Each reference to a terminal (token) requires you to check that the current token is correct, and then either report an error or advance to the next token
expr  :  term  ( ('+' | '-')  term)*;
term  :  factor  ( ('*' | '/')  factor)*;
factor :  Number  |  '('  expr  ')';
```plaintext
expr : term ( ('+' | '-') term*)
term : factor ( ('*' | '/') factor)*
factor : Number | '(' expr ')'
```

FUNCTION Expr()
  Term()
  LOOP
    MATCH current_token
      '+' | '-' ->
        Consume()
        Term()
      _ -> BREAK
```
FUNCTION Expr()
   Term()
END_FUNC

FUNCTION Term()
   Factor()
END_FUNC

expr   : term ( ('+' | '-' ) term )*;
term   : factor ( ('*' | '/') factor )*;
factor : Number | `'expr'`;
expr : term ( ('+' | '-') term)*;

term : factor ( ('*' | '/') factor)*;

factor : Number | '(expr')';

FUNCTION Expr()
    Term()
    LOOP
        MATCH current_token
            ' +' | '-' ->
                Consume()
                Term()
            _ -> BREAK
    LOOP

FUNCTION Term()
    Factor()
    LOOP
        MATCH current_token
            '*' | '/' ->
                Consume()
                Factor()
            _ -> BREAK

FUNCTION Factor()
    MATCH current_token
        Number -> Consume()
        '(' ->
            Expr()
            MATCH current_token
                ')' -> Consume()
            _ -> Error()
        _ -> Error()
Parse Trees

expr : expr ('+' | '-') term | term
term : term ('*' | '/') factor | factor
factor : NUMBER | ID | ('(' expr ')')

Input: 1 * (3 + 4)
Parse Trees

expr : expr ('+' | '−') term | term
term : term ('∗' | '÷') factor | factor
factor : NUMBER | ID | '(expr expr)'

Input: 1 * (3 + 4)

3 + 4

How much of this tree is useful information?
Parse Trees vs Abstract Syntax Trees

Why don’t we keep the parentheses?
Parse Trees: Example

```
prog : (stat)+
stat : expr 'n'
    | ID '=' expr 'n'
    | 'n'
expr : term [(+ | -) term]*
term : atom [(* | '/') atom]*
atom : INT | ID | (' expr ')'
```

Parse Trees

\[
\begin{align*}
\text{prog} & : \ (\text{stat})^+ \\
\text{stat} & : \ \text{expr} \ n' \\
& \quad | \ \text{ID} \ = \ \text{expr} \ n' \\
& \quad | \ n' \\
\text{expr} & : \ \text{term} \ [\ (\ + \ | \ - \ ) \ \text{term}]^* \\
\text{term} & : \ \text{atom} \ [\ (\ * \ | \ / \ ) \ \text{atom}]^* \\
\text{atom} & : \ \text{INT} \ | \ \text{ID} \ | \ \text{'}(\ ' \ \text{expr} \ ')\ '\n\end{align*}
\]

Activity: Draw the AST
\[ \text{prog} = x \cdot 1 = y \cdot 2 + 3 \cdot (x + y) \]
Parser Output

Parse tree vs AST

• Parse Tree: All tokens and complete derivation
• AST: Abstracts away artifacts (drops some tokens and nodes)

How does this affect later stages (interpretation)? An AST

• Gives a smaller tree to traverse (minor efficiency boost)
• Gives a simpler tree to traverse (much easier to interpret!)
Our Grammar

What nodes do we need in our AST?

\[
\begin{align*}
\text{expr} & : \text{term} \ [ (\text{`+`} \mid \text{`-`} ) \text{term} \ ]^* \\
\text{term} & : \text{factor} \ [ (\text{`*`} \mid \text{`/`} ) \text{factor} \ ]^* \\
\text{factor} & : \text{NumberLiteral} \mid (\text{`(` expr `)`)}
\end{align*}
\]
Our Grammar

What nodes do we need in our AST?

```plaintext
expr : term [ ('+' | '-' ) term ]*
term  : factor [ ('*' | '/' ) factor ]*
factor : NumberLiteral | '(' expr ')'
```

We will create nodes from all terminals (tokens) that we need to keep.
Representing an AST in Code

• An AST is a tree data structure
• Some nodes have children, some do not
• Nodes that have children can point to nodes of any other type

• What should we use to represent an AST in Rust?
Two Options For Representing an AST in Rust

Struct
• Explicit inheritance
• Nicer object hierarchy
• Each node will use only the amount of space that it requires
• Requires a fair amount of code

Enum
• Implicit inheritance (ish)
• Odd object hierarchy
• Each node will use the same amount of space as the largest variant (not a problem since we’ll use pointers anyway)
• Requires very little coding
Representing an AST in Rust

trait ASTNode {
    fn evaluate(&self) -> f64;
}

struct Addition(Box<ASTNode>, Box<ASTNode>);
struct Multiplication(Box<ASTNode>, Box<ASTNode>);
struct Number(f64);

impl ASTNode for Addition {
    fn evaluate(&self) -> f64 {
        self.0.evaluate() * self.1.evaluate()
    }
}

impl ASTNode for Multiplication {
    fn evaluate(&self) -> f64 {
        self.0.evaluate() * self.1.evaluate()
    }
}

impl ASTNode for Number {
    fn evaluate(&self) -> f64 {
        self.0
    }
}

#[derive(Debug)]
enum ASTNode {
    Addition(Box<ASTNode>, Box<ASTNode>),
    Multiplication(Box<ASTNode>, Box<ASTNode>),
    Number(f64),
}

fn evaluate(node: ASTNode) -> f64 {
    match node {
        ASTNode::Addition(lhs, rhs) => evaluate(&lhs) + evaluate(&rhs),
        ASTNode::Multiplication(lhs, rhs) => evaluate(&lhs) * evaluate(&rhs),
        ASTNode::Number(val) => val,
    }
}
Rust Box

- Enums in Rust cannot reference themselves
  - Why?
    - It would lead to a recursive calculation of their size
- But they can have pointers to anything
- So, we can have them point to AST nodes
- Why is this different?
  - What is the size of a pointer?
- How would you calculate the size of an enum variant?
An Enum for ASTs

```
#[derive(Debug)]
enum ASTNode {
    Addition(Box<ASTNode>, Box<ASTNode>),
    Subtraction(Box<ASTNode>, Box<ASTNode>),
    Multiplication(Box<ASTNode>, Box<ASTNode>),
    Division(Box<ASTNode>, Box<ASTNode>),
    Number(f64),
}
```
LL Grammar

expr : term expr'
expr' : (+ | -) term expr' | ep

term : factor term'
term' : (* | /) factor term' | ep

factor : NUMBER | ID | '(' expr ')'
BNF LL Grammar

expr : term expr'
expr' : (+ | -) term expr' | ep
term : factor term'
term' : (*) | (/) factor term' | ep
factor : NUMBER | ID | ( expr )

BNF LR Grammar

expr : expr (+ | -) term | term
term : term (*) | (/) factor | factor
factor : NUMBER | ID | ( expr )

EBNF Grammar

expr : term [ (+ | -) term ]*
term : factor [ (*) | (/) factor ]*
factor : NUMBER | ID | ( expr )