Parser

Kit Lao

June 15, 2020

Goal

Given the sequence of tokens, build a sequence of AST(abstract syntax tree) for each expression.

The parser should be able to correctly build the semantics for each AST, as well as recognize syntax errors like a missing operator or expecting a specific keyword at a certain place.

The output should be a sequence of AST, where each AST is syntactically and semantically correct.

Important Stuff

What's an AST?

My understanding is that it is a tree structure that produces the semantics of the program's syntax. Basically, its things like how expressions should be prioritized, the syntax formatted for each expression, and the sub expressions allowed for each expression. There is a type of tree for every type of expression allowed in the language. So like a function call expression would be like function call node and a for-loop expression would be like a for-loop node. I'm not really an expert on this, but it's like a pretty big topic in compilers and there's like a bunch of stuff on-line about it so here's a resource that I found useful.

Grammar rules

So yea, this is pretty important in terms of parsing. I got like another section dedicated to explaining it.

Implementation

Parsing the program

So for every type of expressions, we basically want to write an algorithm for parsing it. A program will have multiple expressions listed out, where each expressions could be nested with other expressions. So this is obvious that the algorithms for each expressions would be recursive.

Primitives

Integers, floats, and Booleans are simple. If you see the token, just create a Node for it once the token is seen.

Haven't implemented strings yet.

Haven't implemented lists yet.

Arithmetic operations

When parsing arithmetic operations, the priority of the operations needs to be considered. There are 4 levels of priority, expr, term, factors, and atomic.

An expr is an arithmetic expressions where the operation is either a addition or subtraction, and the left and right operand needs to be terms. Since these operations have the lowest priority, the parser will consider this first by calling the 'expr' method. The 'method' references the grammar rule for expr by iterating the token sequence to find an addition or subtraction token, and assuming the left and right sides are terms by parsing them as terms using the 'term' method. Since priority of evaluation is from left to right, I need to make sure I build the AST for the left operand first. A 'BinaryOpNode' is used once the left and right operands are determined.

Terms are pretty much the same as expr but with multiply, divide, and modulo operation, and the left and right operands need to be factors.

Factors are either exponents, signed numbers or variables, expressions inside parenthesis, or atomic expressions. If they are signed numbers or variables, I would use the 'signed_factor' method to parse the expressions. The 'signed_factor' parses signed factors by considering the sequence of negative and positive signed in front of the the factor, and produces a 'UnaryOpNode' for every sign from right to left of the factor. Before sign operations are parsed, we need to check if there are exponent operations within the factor. the 'build_exp_ast' builds an exponent AST just like how expr and term is used, except the priority of eval-

uation is from right to left, and the only operation to consider is the exponent operation.

Atomic expressions are either expressions between parenthesis or values. Logical, arithmetic and variable assignments can be used inside parenthesis, so the 'build_var_ast' method is used after an opening parenthesis is seen. After the method is called, the next token must be a closing parenthesis. Values are simple, just build a 'ValueNode' for them.

Variables

Parsing variable declaration requires parsing the 'var' keyword, the identifier, the equal sign, followed either a variable declaration, arithmetic expression, or logical expression.

Conditional statements

Loops

Functions