INTERNATIONAL INSTITUTE OF INFORMATION TECHNOLOGY

PROGRAMMING LANGUAGES PROJECT

CS 306 - PROGRAMMING LANGUAGES

Abstract Syntax Tree for C programming

Sri Vishnu Lahari
IMT2017041

Bishal Pandia
IMT2017010

Phani Tirthala
IMT2017031

June 2, 2020

1 Introduction

1.1 Lexer

Lexer is a software that performs lexical analysis, which is the process of converting a sequence of characters into a sequence of tokens. Tokens are strings with an assigned value and are meaningful. The tokens are then passed on to the parser as inputs.

1.2 Parser

The job of a parser is to convert the stream of tokens produced by the lexer into a parse tree representing the structure of the parsed language.

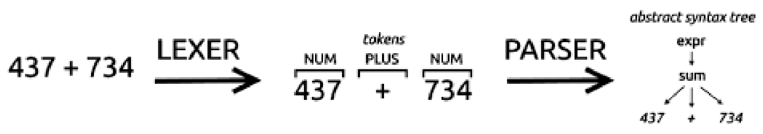
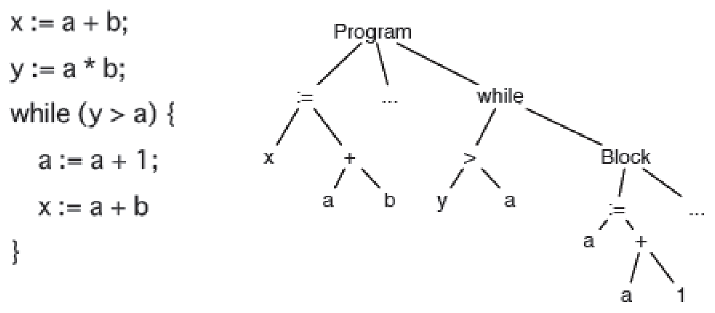


Figure 1: Example for Parsing

1.3 Abstract Syntax Tree

An abstract syntax tree (AST) is a tree that used to represent abstract syntactic structure of the source code written in a programming language which in this case is C. The syntax is "abstract" in the sense that it does not represent every detail appearing in the real syntax, but rather just the structural or content-related details. The following are the advantages of an Abstract Syntax Tree:

- Easy to visualise the flow of the source code.
- An AST usually contains extra information about the program, due to the consecutive stages of analysis by the compiler.
- An AST can be edited and enhanced with information such as properties and annotations for every element it contains.
- Compared to the source code, an AST does not include inessential punctuation and delimiters (braces, semicolons, parentheses, etc.).



```
x := a + b;
y := a * b;
while (y > a) {
   a := a + 1;
   x := a + b
}
```

Figure 2: Example for an AST

2 Construction

2.1 Description

The AST was built for all basic commands of C programming (includes if-else, while, print) using OCaml. The inputs taken are the list of tokens assuming that code was was run through the lexer. The main aim of the project is to construct an AST by parsing the given tokens.

2.2 Implementation

2.3 Input Format

The inputs will be in the form of **generic_object** and the following are considered to be the inputs:

- **Print_generic** for **printf** keyword
- **While_generic** for **while** keyword.
- **If_generic** for **if** keyword.
- **Tokens_generic** is for a list of tokens such as
  - **Operator** for arithmetic operations.
  - **Num_token** for constant values.
  - **Id_token** for variables.
- **Expression_generic** for computed set of tokens.
- **Condition_generic** for conditions.

- **Others** for the tokens that are to be ignored in the AST (;,{ , }).

2.4 Procedure

- **For expressions:**
  Parsing is done in two steps:

  1. **Infix to Prefix:**
     The operator in infix notation is in between the operands, this is commonly used so the source code will use expressions of this type.Where as prefix is notation where the operator is before it's operands, this form will helps to parse better. We are recursively converting the given infix expression to prefix, by matching with operator and then shift the operands.
  2. **Prefix to AST:**
     This is also constructed recursively, using pattern matching. If token matches with an operator then the next two objects are made as operands of that operator(at same level in the tree).

- **For print:**
  Parsing of print is done in the function **parser_print** which takes 2 arguments, which are the string **"print"** and the **statement** to be printed. This will return object of type **Generic_print**, which will have 2 children, 1^{st} is string **"print"** and 2^{nd} is the **statement** is to be printed.

- **For if-else:**
  Parsing of if block is done in these two functions **parser_if** and. **parser_if** is for if-then-else block and takes 3 arguments, which are condition for if block, statements to be executed if the condition if true and statements to be executed if condition is false. This will return object of type **Generic_if**, which will have 3 children, 1^{st} is condition, 2^{nd} is body of if block and 3^{rd} is body of else block.

- **For while:**
  Parsing of while block is done in the function **parser_while** which takes 2 arguments, which are **condition** of while block and **statements** to be executed if conditions are met. This will return object of type **Generic_while**, which will have 2 children, 1^{st} is condition and 2^{nd} is body of while block.

3 Results