

**Department of Computer Engineering**  
**University of Peradeniya**  
**CO523 - Programming Languages**

## Lab 02: Syntax, Semantics, and Parsing

---

### 1. Objectives

- Understand the distinction between syntax and semantics.
- Define programming languages using Context-Free Grammars (BNF).
- Visualize program structure using Parse Trees.
- Implement a Recursive Descent Parser for arithmetic expressions.

### 2. Syntax vs. Semantics

A language definition is loosely divided into two parts: syntax and semantics.

#### 2.1. Syntax

This refers to the structure of the language. It describes how different parts of the language (like words and symbols) may be combined to form phrases and sentences. For example, the syntax of a C if statement requires the word "if" followed by an expression in parentheses.

#### 2.2. Semantics

This refers to the meaning of the language. Describing semantics involves specifying the effects of executing the code. For example, the semantics of an if statement describe how the expression is evaluated and which statement is executed based on the result..

### 3. Lexical Structure

The lexical structure of a programming language is the structure of its tokens, or words. A language's lexical structure and its syntactic structure are two different things, but they are closely related. Typically, during its scanning phase, a translator collects sequences of characters from the input program and forms them into tokens. During its parsing phase, the translator then processes these tokens, thereby determining the program's syntactic structure.

Tokens generally fall into several distinct categories. Typical token categories include the following:

- **reserved words**, sometimes called **keywords**, such as `if` and `while`
- **literals** or **constants**, such as `42` (a numeric literal) or `"hello"` (a string literal)
- **special symbols**, such as `";"`, `<=`, or `+"`
- **identifiers**, such as `x24`, `monthly_balance`, or `putchar`

## 4. Context-Free Grammar

A context-free grammar is said to define the syntax of the language. A context-free grammar consists of a series of grammar rules.

- Terminals – Base tokens of the language. They cannot be broken down further. (eg: keywords, operators, characters, etc).
- Nonterminals – Non-terminals represent larger structures, phrases, or groups of terminals. They appear on the left side of a grammar rule and are defined by the structure on the right side. they are broken down into further phrase structures.
- Productions - Each of the rules in a context-free grammar is known as a production.

### 4.1. Backus-Naur Form (BNF)

Backus-Naur Form is a notation for context-free grammars.

#### 4.2. Elements of BNF

Terminals are simply written out. (eg: `while`)

The Nonterminals are enclosed in angle brackets. (eg. `<statement>`)

Productions are in the form:

`<nonterminal> ::= <sequence of terminals or nonterminals>`

Symbol `|` represents or

eg:

```
expr ::= expr + expr | expr * expr | ( expr ) | number
number ::= number digit | digit
digit ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

### 4.3. Parse Trees

A parse tree hierarchically maps the linear string of code (tokens) to the rules of the grammar.

- **Root:** The top node of the tree, representing the "Start Symbol" of the grammar (the entire program or expression).
- **Internal Nodes:** These represent Nonterminals. They show how a complex structure is broken down into smaller parts.
- **Leaves:** The bottom nodes, representing Terminals (tokens). These are the actual values, operators, or keywords found in the source code

eg: 4.2.1

#### Grammar:

```
<assignment-stmt> ::= <identifier> = <expr>;  
  
<expr> ::= <expr> + <expr> | <expr> - <expr> | (<expr>) | <identifier> |  
<integer>  
  
<identifier> = a | b | c
```

**Sentence:** c = a + 10;

#### Parse Tree:

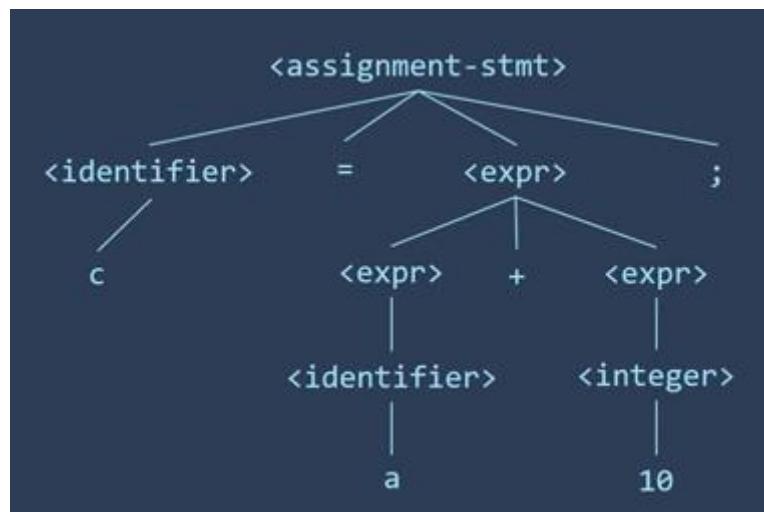


Figure 1: Parse Tree for eg:4.2.1

eg:2.4.2

**Grammar:**

```

<assignment-stmt> ::= <identifier> = <expr>;

<expr> ::= <expr> + <expr> | <expr> - <expr> | (<expr>) | <identifier>
          | <integer>

<identifier> = a | b | c
  
```

**Sentence:** a + b - (c - a) + 11

**Parse Tree:**

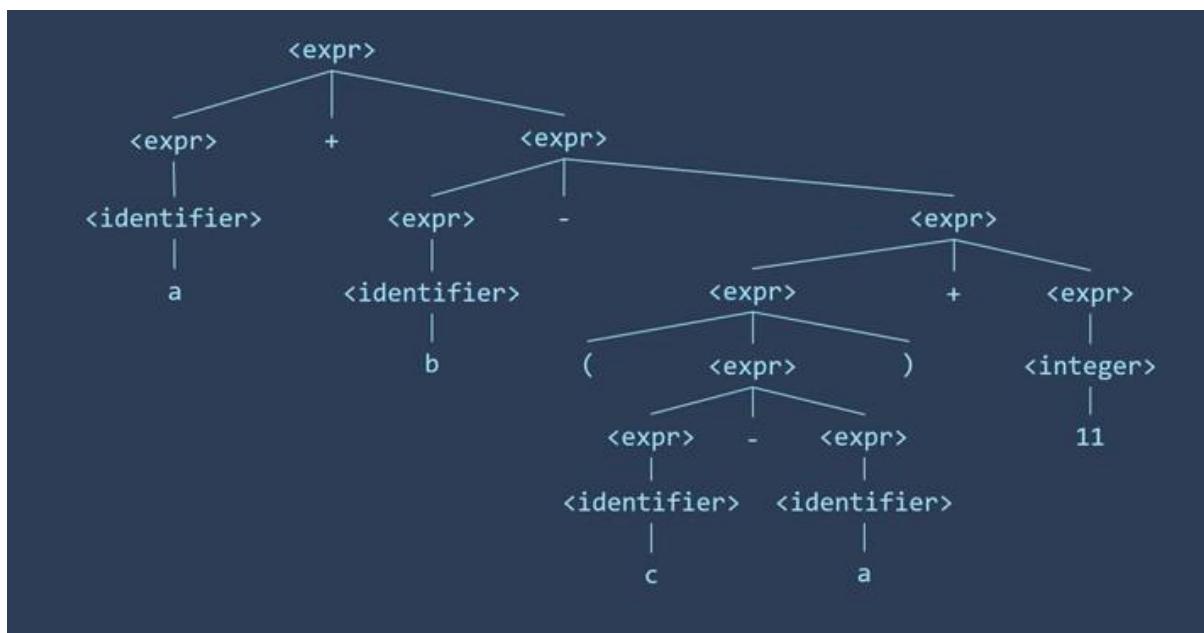


Figure 2: Parse Tree for eg:4.2.2