COMP 442/6421 Compiler Design

Instructor: Dr. Joey Paquet paquet@cse.concordia.ca
TA: Zachary Lapointe zachary.lapointe@mail.Concordia.ca

LAB 3 – GRAMMARS AND PARSING

A2 Introduction - Key Points

- The *parser* is inherently more complex than the *lexer*
  - *CFG*s are more complex than *regular languages*
  - The language's grammar is fully featured
    - Classes, inheritance, control statements, nested function/array calls, numerical/Boolean arithmetic, etc.
      - demo
- New keyword: **void**
- No grammar design choices, unlike A1
- Testing is more complex
- Some source files provided
  - demo

Grammars

- Context free grammars are a general class of formal grammars, whose rules can be applied regardless of context
  - In the context of compiler design, we are interested in grammars which are deterministic
  - For the project, we will be using an LL(1) grammar

Grammars - LL(1)

- **L**eft to Right
  - Traversing input string from left to right
- **L**eftmost parse derivation
  - In the derivation, the leftmost non-terminal is expanded first
- **1** lookahead token
  - A unique production can be selected and applied, by looking at the next (only **1**) terminal symbol

- LL(k) grammars can exist for $k \geq 0$, and all are deterministic
- They are a proper subset of LR(k) grammars, which are a proper subset of CFGs

Grammar Transformation Tools - demo

- Dr Paquet's tool
- AtoCC grammar checker
- University of Calgary tool

Interlude - Pattern: Immutable Objects

- Idea: data whose perceived state is unchanging
  - Often, data use in a program is unchanging
- Features
  - Thread-safe (***n***-read, ***0***-write)
  - Stateless
    - No side-effects
  - Allows easy function chaining
- Implementation
  - Member variables are initialized on object creation, after which they are constant
    - Can be made public
  - Functions have no side effects, and if changes occur, they create new objects

Interlude - Immutable Token

```
public class Token{
    public final String lexeme;
    public final Type type;
    public final int line;
    public final int column;

    public Token(String lexeme, Type type, int line, int column){
        this.lexeme = lexeme;
        this.type = type;
        this.line = line;
        this.column = column;
    }
}
```

Interlude - Immutable Vector

```java
public class Vector{
    public final double x;
    public final double y;

    public Vector(){
        this(0.0, 0.0);
    }
    public Vector(double x, double y){
        this.x = x;
        this.y = y;
    }

    public double length(){
        return sqrt(x*x + y*y);
    {
    public Vector scale(double scalar){
        return new Vector(scalar*x, scalar*y);
    }
    public Vector add(Vector v){
        return new Vector(x+v.x, y+v.y);
    }

    public double complexOperation(Vector v1, Vector v2, double k){
        return v1.add(v2).scale(k).add(v2).length();
    }
}
```

Parsing

- Inputs:
  - Token stream/list
  - LL(1) grammar, first sets, follow sets
- Output:
  - Abstract syntax tree
  - Derivation proof

Parsing - Parse Trees

- The parse tree is a representation of the program’s syntactial derivation
- It is not necessarily useful to the compilation process
  - Transformed LL(1) grammars create distorted and messy parse trees
  - Many nodes of the tree are syntactically important, but semantically irrelevant or redundant
    - Brackets, semicolons, keywords, etc.
- We would like to transform it into a more useful representation
  - Abstract Syntax Trees

- The parsing process also represents a program’s syntactical derivation
  - An explicit parse tree is unnecessary
  - Instead, think along the lines of a *virtual* parse tree, from which the AST is instantly generated
    - The parse tree exist only conceptually, represented by the state of the parser
    - More on this next lab (SDT)