UNIVERSITÉ LIBRE DE BRUXELLES

Info-F403 – Introduction to language theory and compiling

Genial Imperative Language for Learning and the Enlightenment of Students Part-II Report

Herma ELEZI

Gilles GEERAERTS

Mrudula BALACHANDER

Mathieu SASSOLAS

2024

Contents

0.1 Introduction

This report presents the second part of the GILLES project: the implementation of a parser for the Genial Imperative Language for Learning and the Enlightenment of Students (GILLES). The parser is built to validate the syntax of GILLES programs and generate parse trees for valid inputs.

0.2 Grammar Modifications

0.2.1 Original Grammar

The original grammar of the GILLES language is provided in Table 1. It serves as the basis for parser development.

Rule	Production
1	<Program> → LET [ProgName] BE <Code> END
2	<Code> → <Instruction> : <Code>
3	<Code> → ϵ
...	(other rules here)

Table 1: Original Grammar

0.2.2 Modified Grammar

The grammar was modified to:

a) Remove unproductive and unreachable variables.

b) Address operator associativity and precedence to eliminate ambiguity.

c) Eliminate left recursion and apply factorization for LL(1) compatibility.

The transformed grammar is shown in Table 2.

Rule	Production
1	(modified rules here)
...	

Table 2: Modified Grammar

0.3 LL(1) Analysis

0.3.1 First and Follow Sets

The First and Follow sets for each non-terminal were calculated to verify the LL(1) property of the grammar. The results are summarized below:

$$\text{First(<Program>)} = \{LET\}$$

$$\text{Follow(<Program>)} = \{\$\}$$

...(other sets here)

0.3.2 LL(1) Parsing Table

The LL(1) action table is shown in Table 3.

Non-Terminal	Input Symbol	Action
<Program>	LET	Expand
...	...	...

Table 3: LL(1) Action Table

0.4 Parser Implementation

The parser is implemented in Java as a recursive-descent LL(1) parser. Key methods include:

- `Program()`: Parses the `<Program>` rule.
- `Code()`: Handles `<Code>` and its recursive expansion.
- `Instruction()`: Implements `<Instruction>` cases such as `If`, `While`, etc.

Listing 1: Example Method: Program

```
private ParseTree Program() {
    List<ParseTree> leaves = new ArrayList<>();
    leaves.add(match(LexicalUnit.LET));
    leaves.add(match(LexicalUnit.PROGNAME));
    leaves.add(match(LexicalUnit.BE));
    leaves.add(Code());
    leaves.add(match(LexicalUnit.END));
    return new ParseTree(new Symbol(LexicalUnit.Program, "Program"), leaves);
}
```

0.5 Parse Tree Generation

Using the `ParseTree.java` class, the parser generates a LaTeX-formatted derivation tree. The tree can be visualized by compiling the LaTeX output.

0.5.1 Example Parse Tree Output

Below is an example LaTeX output for a sample GILLES program:

Listing 2: Generated LaTeX for Parse Tree

```
\documentclass{article}
\usepackage{tikz}
\begin{document}
\begin{tikzpicture}[grow=down]
\node {Program}
    child {node {LET}}
    child {node {ProgName}}
    child {node {BE}}
    child {node {Code}}
    child {node {END}};
\end{tikzpicture}
\end{document}
```

0.6 Testing and Validation

The parser was tested on various GILLES programs to ensure correctness. Table 4 summarizes the results.

Test Case	Description	Result
Test 1	Valid Program	Success
Test 2	Syntax Error	Correctly Detected

Table 4: Test Results

0.7 Conclusion

This report details the design and implementation of a recursive-descent parser for the GILLES language. Future work includes optimizing error reporting and extending the language's features.