Hochschule Karlsruhe
Technik und Wirtschaft
UNIVERSITY OF APPLIED SCIENCES

Systems Programming

Documentation Part II: Parser

written by: Christian Modery (???)

Henrik Mengel (52514)

Lara Speisser (54239)

Marc Ernst (52720)

Tim Glöckner (53843)

January 28, 2019

Contents

1 Introduction

Part II of the systems programming lab consists of the implementation of a parser that is based on the work of part I that performes a lexicographical analysis and generates a sequence of tokens.
The main goals of this part are the understanding of where the parser takes place in a compiler and learning to implement and understand a recursive descent which implements a LL(1)-parser. This part will also focus on the verification of the source code with the target language's grammar, an algorithm to annotate the tree, which abstracts the source code, as generated by the LL(1)-parser and verifies compatibility of the types of the nodes of the tree, as required by the language and an algorithm to generate code for the target platform, based on the annotated tree, which is an assembly dialect for an interpreter that's provided to the students for testing purposes.

Similar to part I the general conditions dictate that the implementation will be done in Linux with C++ and forbid to use any of the data-structures that the standard template library (STL) provides.

Structure of the documentation

First we will shortly describe the functionality of a parser before we introduce our implementation and explain all components that we have implemented.

2 Fundamentals

This section explains the fundamental working methods of a parser and will outline its components.

2.1 Introduction to the working method of a parser

The task incorporates the implementation of a parser that links the scanner to the parse tree by processing the tokens received from the scanner and thereof creating a parse tree that consists of nodes.

2.1.1 Parser

The parser evaluates a sequence of tokens which is received from the scanner by performing a syntactical analysis, using the grammar listed in table 1. The tokens are received calling the scanners method `nextToken()`.
Then a parse tree gets created and the tokens are passed to the parse tree for further processing. If any syntactical errors occur, the parser terminates by logging the errors.

PROG	::=	DECLS STATEMENTS
DECLS	::=	DECL ; DECLS | ε
DECL	::=	int ARRAY identifier
ARRAY	::=	[integer] | ε
STATEMENTS	::=	STATEMENT ; STATEMENTS | ε
STATEMENT	::=	identifier INDEX := EXP | write(EXP) | read(identifier INDEX) | STATEMENTS | if (EXP) STATEMENT else STATEMENT | while (EXP) STATEMENT
EXP	::=	EXP2 OP_EXP
EXP2	::=	(EXP) | identifier INDEX | integer | -EXP2 | !EXP2
INDEX	::=	[EXP] | ε
OP_EXP	::=	OP EXP | ε
OP	::=	+ | - | * | : | < | > | = | =:= | &&

Table 1: The grammer the parser has to evaluate.

2.1.2 ParseTree

The parse trees task is to perform a semantic analysis of the source code. That means, the consistency is checked by verifying the namespaces, declarations and typecasts.
This happens by calling the method `typeCheck()`, that checks, if the types of the nodes in a sub tree is consistent. If not, the check stops and logs the errors.

2.1.3 TreeNode

A tree node is part of the parse tree, or more precisely of a sub tree that is created whenever a new rule of the grammar in table 1 is recognised. This happens during the recursive descent.

3 Implementation

In this section we will discuss our implementation of the parser and all its components.

3.1 Grammar

The parser has to evaluate the grammar shown in table 1. Therefore first and follow sets are used. The follow set `follow(T)` contains all terminals that might appear right after the terminal `T`. The first set `first(T)` on the other hand, contains all legit terminals which may first appear with a specific grammar rule.
With the knowledge of those sets the order of the tokens can be checked during the recursive descent.

The functionality of the grammar class is to calculate these first and follow sets of a specific rule.

3.2 Parse Tree

3.3 Type Check

3.4 Make Code

3.5 Parser

4 Conclusion