

Software Design Documentation

Benji Altman

May 1, 2017

Contents

1	Introduction	2
2	General	2
2.1	Main	2
2.2	Unique Identifier Generator	2
2.3	Pair	2
3	Scanner	2
3.1	TokenType	2
3.2	Token	2
3.3	Lookup	2
3.4	Scanner	3
4	Parser	3
5	Symbol Table	3
5.1	Scope	3
6	Syntax Tree	3
7	Details	3
7.1	toMips	3
7.1.1	Declarations	3
7.1.2	Stack	3

1 Introduction

This is the software design documentation, it will describe on a high level view how each package works. The entire program is based off the grammar rules defined in this pdf.

2 General

The **general** package contains non-program specific classes, as well as the main class **Main**.

2.1 Main

The **Main** class is static and has only general static functions, as well as the **main** function, the program's entry point.

2.2 Unique Identifier Generator

Each instance of this class keeps track of a list of given IDs, and makes sure that it never gives the same ID twice. When getting a new ID you may make a request, if you make a request it will try and give you a string close to what you gave it. In the current implementation it will first attempt to return the string you requested, and if that is not available it will append something to that string to make it unique.

By default the UID considers all alphanumeric characters and underscore to be valid characters in an identifier. All strings where you don't request a specific string are prepended with an underscore so that it does not produce an identifier that begins with a number. In future iterations of this it should only produce strings that match a regex given at creation time.

2.3 Pair

This very basic class simply is a pair of two genericed objects.

3 Scanner

This package maintains classes that allow you to scan through a file and produce tokens.

3.1 TokenType

This is an enum that contains a value for all symbols, and then a special value for each type of literal as well as a special value for identifiers.

3.2 Token

This class represents tokens that are parsed from an input Pascal file. It contains the string that was parsed to produce this value, a type which is a **TokenType**, and a line and column number which is used in error messages.

3.3 Lookup

This class acts much like a static class. It can not be instantiated as it's default constructor is private, although it does contain a static instance of itself named **LOOKUP** that can be used. It allows you to lookup **TokenTypes** based on their strings, and allows you to do the reverse. The lookup may be done with the **get** function and reverse lookup is done with **teg**, although if you call **teg** on any special values¹, it will break the program.

¹Special values are: any types representing literals, and identifier

3.4 Scanner

This class is computer generated from the file `Scanner.jflex` by the program `jflex-1.6.1.jar`. It turns text in a file into tokens as defined in the grammar. With the listed exceptions:

1. `id` is defined as `(letter | _)(letter | _ | digit)*`
2. There are separate `TokenType`'s for both integer and real literals, instead of just a num `TokenType`.
 - (a) Integer literals are defined as `[+-λ] digit digit*`
 - (b) Real literals are defined as `[+-λ] digit digit* . digit digit*`

4 Parser

The parser package maintains support for taking tokens from the scanner and populating symbol tables and generates a syntax tree.

5 Symbol Table

This package contains support for a symbol table specific to the needs of this program.

5.1 Scope

This class represents a single scope level. It contains a pointer to its parent scope which allows you to find identifiers from higher scopes. `Scope` also contains a map that maps from identifiers to information about them, which is stored in a private `Symbol` class contained in the `Scope` class. Symbols contain all information that any identifier might need.

6 Syntax Tree

This maintains all node types that are in the syntax tree as well as abstract classes and interfaces that nodes inherit from for purposes abstraction and grouping. All classes are documented in the `JavaDoc`. A UML diagram of this package can be found in this PNG file.

Code generation is maintained through each node's `toMips` function, and syntactic analysis is largely done in constructors for the syntax tree.

7 Details

Here I will go through details of how the program runs, knowledge of this is largely irrelevant for use of the program however would be integral to contributing.

7.1 toMips

As stated above the `toMips` function in each node supports produces assembly to complete each node's task, as *should* be documented in the `JavaDoc`, however some details are pertinent to the entire process and those will be covered below.

7.1.1 Declarations

There are a few things to cover here, to start, all variables are put on the stack, in fact the entire language, as will be explained below in the section on the stack, is entirely stack based.

The first thing that this means is that the heap and `.data` section really aren't used. It is possible that future versions of this program will support usage of the heap, but for now it is but a mere pipe dream. For now all variables follow one simple rule

ALL arguments, arrays, and declared variables, exist in the stack, and are stored as an offset from what will be referred to as a function's stack pointer.

7.1.2 Stack

hi