Software Design Documumentation

Benji Altman

May 1, 2017

Contents

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 grammer 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 geting 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 values[1], it will break the program.

[1]Special 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 difined in the grammer. With the listed exceptions:

1. **id** is defined as **(letter | _) (letter | _ | digit)***
2. There are seperate `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 it's parent scope which allowes 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 trough each node's `toMips` function, and syntatic analyisis 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 irrelivant 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 pertinant to the entire process and those will covered below.

7.1.1 Declarations

There are a few things to cover here, to start, all variables are put on the stack, infact 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 suport usage of the heap, but for now it is but a mear 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 refered to as a function's stack pointer.

An array is stored in the program by it's first index, and then offsets are calculated from there.[2] Any accessing of arrays is always checked, a friendly error is thrown try any funny buesiness outside of arrays.

7.1.2 Stack

The stack is used for everything in this program. Every time a function is called it has space allocated for variables on the stack and then 24 bytes are allocated for some special values used by the function. These will be described in the function details.

In addition to functions, expression results are also put on the stack. An **expression** is *any part of code that returns a value.* If you look at the UML diagram for the syntax tree and find all the nodes that inhearit from `ExpressionNode`, all of these would be considered expressions. Any return from these expressions are put on the stack.

This means that to evaluate `foo + bar`, the value of `foo` is first put on the stack, then `bar`'s value is pushed onto the stack finally they both are pulled from the stack and their values are added together and the result is put back on the stak for further computation. Functions work in much the same way. All arguments are put on the stack and then copied over to where the belong when the function begins, and before the function ends the return value from the function is copied over to be returned.[3]

7.1.3 Functions

All code executing contexts[4], including the golobal one, is considered part of a function for the purpose of the compiler, however there are some obvious special cases made for the global case where some things simply aren't done, however space is still reserved for special values that it does not use.

Every function has 24 bytes of overhead (6 4-byte words) respresenting different values. Not all are used in every function, yet the space is always reserved. The values in order are listed bellow:

1. The current stack head (4 bytes) at 0 offset from `$sp`
   The current stack head points to the current point where the stack head is. The stack head moves up and down as expressions are used, however it returns to it's inital value of `$sp - 4` after every statement.

2. The return address (4 bytes) at +4 offset from `$sp`
   This is the address in the code from which we jumped to this function, upon completion of the function it is loaded into a registar and a `jr` instruction is called on it.

3. Previous stack pointer value (4 bytes) at +8 offset from `$sp`
   This is where `$sp` was pointing before the current function was called. The stack pointer is returned to pointing there post execution of this function.

4. One level up function's stack pointer (4 bytes) at +12 offset from `$sp`
   This the 'parent' function. It is used in order to find variables declared in a lower scope, for example if accessing a global variable. Because the parent function also has a pointer to it's parent, this is treated like a linked list and is iterated through in order to find lower and lower scope variables. The global scope sits at the bottom and this value is not used by it.

5. Current scope level (4 bytes) at +16 offset from `$sp`
   This integer represents what level the current scope is at, if it is a 0 then it is the global scope, if it is declared in the global scope the it is 1 and inside one of those is 2 and so on... This is used to set the above value upon function conception.

[2] An alternitive way of storing arrays would be by their zero index, even if that isn't included in the array, it could potentially be faster.

[3] This is done even if there is no return value, but the value is simply never used in this case.

[4] Anywhere a statement may be put

6. Return value (4 bytes) at +20 offset from `$sp`.
   This a special variable, it is only to be used in functions (not procedures), which for nearly all of instances are the same for running. None the less even in a procedure the return value space is still allocated. After the function completes this value is copied over to be put on the stack for the calling function.

7.1.4 Floating point opperations

Currently the language only supports 32-bit floating point values, although only even numbered floating point registars in the floating point co-processor are used. This is to make conversion to 64-bit floating point values easier in the future.