DigitalDiscord Language Guide

SirWolf

April 15, 2021

Contents

1 Introduction

1.1 Words from the Dev

This guide should teach you how to use the DigitalDiscord language-parser from the DigitalDiscord project.
You can use this also in your project (licensed with MIT) but please credit me there, thank you.

Just so that you know: this here is very assambly-bash-like.
We use flags and nearly every command in the standard-library is only 3 chars long.
So please write in this style if you use the standard-lib.

1.2 Requirements

- some knowledge in C++
- a C++ compiler (g++/clang are tested)
- the newest version of DigitalDiscord

1.3 System

The System behind the whole parser thing is very simple.
There are only 3 types you have to know about:
- Parser (LangParser)
- Command (Command)
- Argument (Arg)

Normaly it looks like this:

```
<command> <arg1> <arg2> ...
<command> <arg1> <arg2> ...
```

The parser checks if the first tokens (splited by spaces) is a known command, and if yes
it triggers his arguments with the value of the string thats at the argument possition (<arg1>, <arg2>, ..).
Every argument returns a value to the command, and at the end, the command handles the
argument-return values and do whatever the command is supposed to do.

2 Guide

Here you will find some tutorials for the DigitalDiscord language-parser.
Have fun with it.

2.1 Hello World

First things first:
we need to make a file (for this tutorial we will name it "Tutorial.h").
Now we will add the include-guard and all the includes from the lib:

```
#ifndef TUTORIAL_H_
#define TUTORIAL_H_
#include "/src/InternalLib.h"



#endif
```

After that we'll add our parser:

```
inline InternalLib::LangParser tutorial;
```

And now a object...

```
inline InternalLib::LangParser tutorial = InternalLib::LangParser()
```

Now we can start to make some settings.
For example we can set the file that we want to parse!
Let's say it's "test.txt"

```
inline InternalLib::LangParser tutorial = InternalLib::LangParser()
                .setFile("test.txt")
```

(Information: nearly every function returns "this". So we use a rust-like build-pattern.
Everytime we make a new instance of a class we add one tab, so that it doesn't look like a mess.)
Or we can add a Command!

```
->addCommand(InternalLib::LangParser::Command()
```

Ok, just for showing, we will make a command that writes into stdout.
(or simplified: another std::cout)
Let's call it "out".

```
->addCommand(InternalLib::LangParser::Command()
                        .setName("out")
                       )
```

Now we need something to get the text we want to print.
We can do that with arguments. Let's add one with the name "text".

```
->addArg(InternalLib::LangParser:.Command::Arg()
        .setName("text")
                )

#endif
```

Every argument needs a type.
There are only 3 types:
- GET
- SET
- TAG

We're only going to talk about GET.
We will talk about SET and TAG later.

GET means that the argument gets the string thats at the current position.
Like:

```
out Hello_World
```

"Hello_World" is what the argument "text" gets.
So let's add the type:

```
//using AGET = InternalLib::LangParser::Command::Arg::AGET;
->addArg(InternalLib::LangParser:.Command::Arg()
        .setName("text")
        ->setType(AGET)
                )
```

Last thing we have to do is go give the argument informations about how to handle the input.
In our case: "return the current string".

```
->addArg(InternalLib::LangParser:.Command::Arg()
        .setName("text")
        ->setType(AGET)
        ->sgetFun([](...) {
                arg.setEntry(tokens[pos]);
        })
        )

#endif
```

Now you may ask yourself: "Why are there "..." in the lambda?".
Well, is difficult, I cant paste the whole paramether-list there because it's to long, but I will list they here for you:

```
(
        Arg& arg,
        tokenType& tokens,
        Register& vars,
        size_t& pos,
        const std::vector<Operator> operators,
        LangParser* parser
)
```

"arg" is the current argument.
"tokens" is from type "tokenyType" (aka std::vector<std::string>) and containes the whole lexed file. "vars" is from type "Register" (aka std::map<std::string, std::string>) and containes all existing variables.
(Every variable is a string!)
"pos" is the current possition in "tokens".
"operators" stores all the existing operatores from the parser.
"parser" is a pointer to the current parser.

The get-function will be executed by the parser during the parse() process.
With the function "setEntry" are we returning the current token to the command.

Now we will need to tell the command what it has to do with our argument-input.
We can do that really easy with the "setExecute()" from the class "Command":

```
->setExecute([](...) {

})
```

This time the arguments are the following:

```
(
Command* com,
Register& vars,
Register& entry,
LangParser* parser
)
```

"com" is a pointer to the current command. "vars" is where the variables are stored. "entry" is a map between the name of the "Arg"s and their return type. "parser" is a pointer to the current parser.

We want to write our input in stdout, so we have to take our value (entry["text"]) and print it!

```
->setExecute([](...) {
        std::cout << entry["text"] << "\n";
})
```

So, there we got it, our first command.
Now we only have to add a "->toInstance()" to everything because we want to convert
the pointers to a normal instance.

At the end it should look like this:

```
inline InternalLib::LangParser tutorial = InternalLib::LangParser()
        .setFile("test.txt")
        .addCommand(l::LangParser::Command()
                .setName("out")
                ->addArg(InternalLib::LangParser::Command::Arg()
                    .setName("text")
                    ->sgetFun([](...) {
                        arg.setEntry(tokens[pos]);
                })
                ->setType(InternalLib::LangParser::Command::Arg::AGET)
                ->toInstance()
            )
            ->setExecute([](...) {
                std::cout << entry["text"] << "\n";
            })
            ->toInstance()
        )
        ->toinstance();
```

If we want to test it now, we only have to call the "parse()"
and write a programm like this into "test.txt":

```
out hello
out world
out This_will_work_i_guess
```

And we will get the output:

```
hello
world
This_will_work_i_guess
```

This is the basic on how to make a simple Command that can print
out stuff!
See the other sections to learn more.

2.2 Include from the standard-library

The DigitalDiscord-project has some standard commands that you can include easily.
Just use the function "include" from "LangParser".
A list of standard Command-libraries:
- math → some math functions that you can't call with operators.
- system → if/else, loops, exit, scopes and other things.
- io → standard input/output
- variables → normal variable-handeling

Let's see how we can do that:
At first we have to include "InternalStandardParserLibrary.h" from "src", and include an parser:

```
#include "src/InternalStandardParserLibrary.h"

inline InternalLib::LangParser test = InternalLib::LangParser()
        .include(STDParser::io)
        ->include(STParser::system)
                ...
```

The function "include" copies all commands in the given parser into the own vector of commands.
Warning: DONT CALL COMMANDS THE SAME AS THEY IN THE NAMESPACE STDParser IF YOU INCLUDE THEM!!
That won't work out fine at all!

2.3 ASET/ATAG

ASET and ATAG are types of arguments, which can help you to add bash-like flags.
An ASET is for example:

```
g++ -o main ...
       ^~~~~~~
```

And a ATAG is the same as a ASET but doesn't get any input.
It's just a bool-switch and will always return "1" if existing.

The special about ASET and ATAG is that they will be ignored if there isn't any argument-token that matches with them.
For short: they are always optional!

You can set the call for them (in bash: "-") by using the function: "setArgPrefix" in LangParser

An argument with the tag ATAG doesn't need a getFun because it only returns 1/0.

2.4 Handle variables

In the namespace "Handlers" are some usefull handlers, for example a Handler for variables.
There are 4 Handlers, but we will only use one.
This one is called "checkvariables" an has the following paramethers:

```
(
char call,
std::string check,
Register& reg
)
```

"call" is the char that calls a variable (in bash it would be "$")
"check" is the string to check a variable for.
"reg" is the variable register where the programm is looking at if a variable is found.

It returns a string, with is either the value of the variable or just the given string.
This function throws an instance of "VariableAccessError" if the requestet variable doesn't exist.

2.5 Handle expression

In the namespace "Handlers" are some functions to handle expressions, but only 2 are important for us.
These are "handleExpression" and "checkNormalBexpr".
Paramethers of "handleExpression":

```
(
std::string str,
const std::vector<Operator>& operators,
char variablecall,
Register& vars
)
```

"str" is the expression as string.
"operators" is the list of operators.
"variablecall" is how you want to call variables. (in bash: $)
"vars" is the register of all existing variables.

return value:
int

"handleExpression" handles the given expression and returns the result. Warning: it doesn't know the "Dot before dash"-prinzip, but you can solve this by using normal braces.

Paramethers of "checkNormalBexpr":

```
(
        //same as "handleExpression"
)
```

return type:

```
std::tuple<int,std:string>
```

If "str" is an expression (starts and ends with normal braces), the result will be pushed in int
and std::string will be set to "FALSE".
If not, std::string will be the str and the int 0.

2.5.1 Operators

"Operators" is a class in the namespace "Handlers" and can be easily setted up by his constructor.

For example, we are going to make an operator that can add 2 number together:

```
Operator coolNewOperatorThatCanDoThings();
```

Now we have to give it a name, like "+","-",...
Let's call it "+", to keep it simple.

```
Operator coolNewOperatorThatCanDoThings("+");
```

And now we need to say the Operator, what it has to do:

```
Operator coolNewOperatorThatCanDoThings("+",[](int left, int right)->int {

});
```

Now make the operation...

```
Operator coolNewOperatorThatCanDoThings("+",[](int left, int right)->int {
        return left + right;
});
```

Here we go, a cool operator that can add two numbers.
The only thing left is to add it into our parser...

```
parser.addOperator(coolNewOperatorThatCanDoThings);
```

And Done!
Warning: only operators which takes 2 number can be added as an operator,
everything else you need is in "STDParser::math".

2.6 LineRegister

"LineRegister" is a member of "LangParser" and can convert a line into the internal token possition.
For example: if you want to know to what you have to set "pos" to, to skip some lines,
just do "parser.lineRegister[<line>]" and you will get the size_t for "pos".

Good example is "jmp" in "STDParser::system".

2.7 off/unlock

You can lock the parsing progress with "off" and "unlock".
The parser will only execute commands if the private member variable "stoping" is false.
off → true
unlock → false

3 STDParser/Operator list

3.1 math

3.1.1 sqrt

Calculates the square-root of arg1 and writes it into arg2. Variable-compatible.

3.1.2 not

Performs a binary-not on arg1 and write it in arg2. Variable-compatible.

3.2 variables

3.2.1 set

Set arg1 to arg2. Variable-compatible

3.2.2 del

Deletes arg1.

3.2.3 vex

Short for "variable-exists".
Write in arg2 "1" if arg1 exists, else "0".

3.3 system

3.3.1 ext

Exit the programm.
You can set a exit-code with "-c". Variable-compatible.

3.3.2 exe

Makes a scope with arg1.
Save all created variables with "-sv".
Ignores "ext" with the flag "-no".

3.3.3 con

Typical if/else.
If arg1 == "1", run arg2, else run SET "-e".
"-n" flips arg1.
"-sv"-flag saves all new-created variables.

3.3.4 slp

Sleeps for arg1 seconds. Variable-compatible.

3.3.5 spt

Will trigger a normal pause-screen. (selfmade)

3.3.6 run

Will run arg2, arg1 times.
arg1 = expression.
arg2 commands.
If flag "-e", arg2 will only run if arg1 is "1".
(Will set "__loop_val__" to the current round-counter)

3.3.7 jmp

Jumps to line arg1.
If flag "-d" == "1", it will jump to "__line__" + arg1.
Variable-compatible.

3.4 io

3.4.1 out

Will write arg1 in stdout. Variable-compatible.
Flag "-nnl" will cancel the newline print at the end.

3.4.2 red

Will get an user input and write it in arg1.
You can print text befor reading the input by uisng the SET "t".
Variable-compatible.

3.4.3 fout

Copy read() from the file named arg1 into arg2.
Variable-compatible.

3.4.4 fin

Write arg2 into the file named arg1.
Variable-compatible.
If the flag "-t" is on, it will clear the file befor writing.

3.5 external

3.5.1 inc

Includes the file arg1 into your program.
If the flag "-lib" is on, it will take the file from the
linked standard-library.
Variable compatible.

3.5.2 lik

Links the path arg1 as standard-library.
Variable-compatible.

3.6 Operators

3.6.1 plus

Returns left + right.

3.6.2 minus

Returns left - right.

3.6.3 times

Returns left * right.

3.6.4 div

Returns left / right.

3.6.5 power

Returns left r̂ight.

3.6.6 band

Returns left & right.

3.6.7 bor

Returns left | right.

3.6.8 mod

Returns left % right.

3.6.9 bxor

Returns left xor right.

3.6.10 equal

Returns left == right.

3.6.11 nequal

Returns left != right.

3.6.12 land

Returns left && right.

3.6.13 lor

Returns left || right.

4 Sytsem-variables

4.1 __file__

Stores the current filename.

4.2 __line__

Stores the current line.

4.3 __main__

Stores if the current file is includet or not.

4.4 __start_time__

Stores the start time of the program in UNIX-style.

4.5 __username__

Stores the system username.

4.6 __loop_val__

In a loop (manual): stores the run-counter.