Instruction Recognition for Integrated Software
IRIS - 0.1

Keywords: Compiler, Recognition, Integration, Software, Instruction, Iris, Antlr, Flex, Plex, Lex, Bison, Linux, GCC, Clang, Portugol, VisuAlg, OSS, Free Software, Open Source, Software, Parser, Lexer, Infrastructure, C, Python, Javascript, Node.js, TypeScript, Lua, C++, LuaC, TSC, Java.

Developer: Gabriel Margarido
Website: iris-sdk.gabrielmargarido.org

Version: 0.1

INSTRUCTION RECOGNITION FOR INTEGRATED SOFTWARE
IRIS 0.1

IRIS or Intruction Recognition for Integrated Software is a compiler infrastucture building based on YAML declarative files, this way IRIS ables normal people to build their own compiler that runs on Node.js Javascript Platform. IRIS is developed by Gabriel Margarido (www.github.com/polskidev) or (www.gabrielmargarido.org).

To begin with IRIS, you need to have Node.js and NPM installed on your computer, and next, download IRIS Development Kit from (iris-sdk.gabrielmargarido.org), unzip the downloaded file, enter into the unpacked directory and run the following command:
`npm install --save`

It is going to create a folder named node_modules inside the current directory.
Now you should choose, if you are going to create a compiler that supports OOP or only Procedural Programming.

1. To create an OOP compiler:

```
./iris-create <file>.yaml --oop-default
```

2. To create a procedural compiler:

```
./iris-create <file>.yaml --procedural-default
```

If everything works correctly, you will have a YAML file with the chosen name, inside it a prototype of a compiler is written in YAML.

(YAML files work like Python Programming Language)

```
function_definition:
        symbol:         "everything here"
        syntax:         "everything here"
        transpile:      "everything here"
```

Explaining:
- symbol is inside function_definition section, as well as, syntax and transpile. Each element of a section has their own value.

Normally,
Symbol stores the token or lexem of the whole command.

Syntax stores how the main lexem or token is used inside the whole command, such as, position, grammar relation, and work together with other pieces of the command.

And Transpile stores how the lexems and their grammar work together to give an output that is going to be written inside a target code file.

First of all, we need to tell our compiler which is the input extension of source-code files and the output extension of compiled files.

The used syntax is: `node <file>.js <sourcefile>.mylang`

To do this, we do:

```
input_extension: ".mylang"
output_extension: ".c"
```

Next we should tell our compiler which lexems begin and end our program based on the following structure:

```
program

      # YOUR WHOLE PROGRAM GOES HERE

endprogram
```

These main symbols/lexems/tokens are defined inside begin and end sections, you can change them if you wish.

Next section is import section, here is defined the instruction for importing other source-code files in the creating programming language.

```
#include file from path
```

name is a nickname to the file path.
path is the file path.

The comment section defines the lexem or token to start a comment.

```
// "This is my comment"
```

Basically most instructions follow the same logical structure, but some ones are more complex to understand, such as, var_assignment.

```
var_assignment:
      symbol: "="
      array_auxiliary_symbol: "[]"
      syntax: "typed indentifer = value"
      syntax_array: "typed[] identifier = value"
      null_value: "nil"
      transpile0: "let ${identifier}: ${typed}"
      transpile1: "let ${identifier}: ${typed} = ${value}"
      transpile2: "let ${identifier} = { ${value.slice(1,-1)} }"

      typedef:
            integer:
                  from: "Integer"
                  to:   "int"
            float:
                  from: "Floating"
                  to:   "float"
            string:
                  from: "String"
                  to:   "string
            bool:
                  from: "Boolean"
                  to:   "bool"
```

`symbol` stores the symbol/lexem that tells to IRIS that, this is a variable definition.

`array_auxiliary_symbol` stores the symbol/lexem that tells to IRIS that, this is a array/vector definition.

`syntax` stores the grammar and logical relation to use on variable case.

`syntax_array` stores the grammar and logical relation to use on the array case.

`null_value` stores the value that should be understood as null, undefined or none.

`transpile0` stores the equivalent instruction in the target language that tells to IRIS what it is going to do to compile the instruction on non-initialized variable case.

`transpile1` stores the equivalent instruction in the target language that tells to IRIS what it is going to do to compile the instruction on declared and initialized variable case.

`transpile2` stores the equivalent instruction in the target language that tells to IRIS what it is going to do to compile the instruction on initialized array/vector case.

`typedef` stores all datatypes and its custom conversions from your own programming language to the target language. It's structured using the following structure: `typedef -> [datatype] -> from | to`

`from` means the datatype in your own custom programming language,
and `to` means the related datatype in the target language. They are separated by the following classification: `int` `float` `string` `bool`

It is important to know that arrays/vectors are declared using brackets after the name/identifier, not before or after datatype!

IRIS Reserved Grammer Keywords:

`${typed} and typed` means the datatype of a variable, function or array
`${identifier} and identifier` means the name of a variable, function or array.
`${value} and value` means the value of a variable or array.
`${args} and args` means the arguments of a function.
`${name} and name` means the nickname of a importing related to a filepath.
`${comment}` means the content of a comment.

`${statement} and statement` means statement or expression, normally used on loops and conditionals.

`num` means an integer number used on: `5 times DO SOMETHING end`

`${iter} and iter` means an iterator (like: i, j or k) used on FOR loop.

`${low} and low` means the initial value of FOR loop.
`${max} and max` means the final value of FOR loop.

Now we must compile our YAML declarative file to a Javascript file for Node.js that is going to run our own programming language compiler. To do this you should choose some small details in our syntax, on the first mode `--include-braces` braces are used to store values inside arrays as well as parenthesis are used to store statements. However on the second mode `--exclude-braces` only parenthesis are used to store statements and arrays.

FIRST MODE:

```
if (a > 3) do
      int a[] = {0, 5, 10, 20, 25}
end
```

```
./iris-generate <file>.yaml <output>.js --include-braces
```

INSTRUCTION RECOGNITION FOR INTEGRATED SOFTWARE
IRIS 0.1

SECOND MODE:

```
if (a > 3) {
      int a[] = (0, 5, 10, 20, 25)
}
```

```
./iris-generate <file>.yaml <output>.js --exclude-braces
```

Developed by Gabriel Margarido,
Licensed under 2-Clause BSD License (FreeBSD)

Download IRIS at: iris-sdk.gabrielmargarido.org