CS490

Woodrow Scott

**KNX2 Compiler Chain - Update 1**

https://github.com/knoxaramav2/KCC

**ABOUT**

**General**

The KNX2 Compiler Chain (KCC) will be designed to compile and assemble KNX2 programming code to machine code. The minimum expected deliverable is to be a working compiler to convert the KNX2 language to ARMx86 assembly, specifically targeted for the Raspian operating system. If time allows, a custom assembler will also be written, which is an expected outcome. As this may take a large amount of time, the third deliverable, the linker, is unexpected.

The name KNX2 comes from a current side project in developing a runtime language, although the languages will have little in common.

**Example Expected Workflow and Target Operation**

**Written Code**

Below is an example program written in the current KNX2 syntax. This includes example commands to compile the source code, and invoke with sample cli options and its output.

**Main.k2**

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

@use io #keyboard, console io

@use common #string/array utils

#must have a main function, does not have to be void

function<void> main(string\*args, {

string template("arg %d is %s\n");

foreach (args:s, {

int:static i;

++i;

printf(template, i, s);

});

});

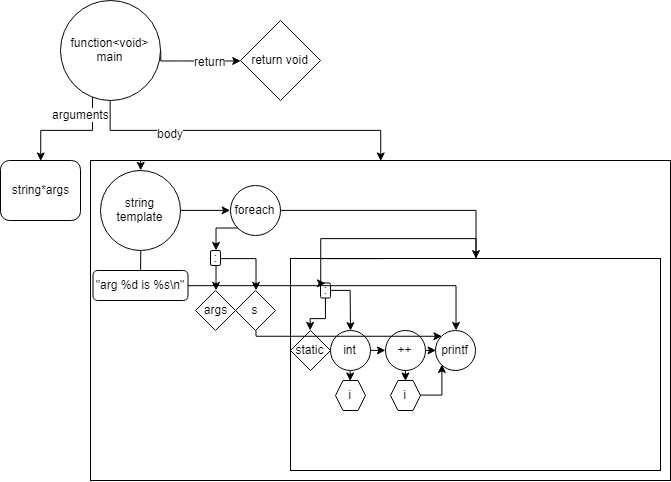
#compile options: kcc main.k2 -o main.exe

#sample cli options: ./main.exe jerry garcia

#sample cli output:

# arg 1 is jerry

# arg 2 is garcia



**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Language Syntax**

This section covers built-in functions and operators, as well as providing information about how these relate the Abstract Syntax Tree (AST) used to verify grammar and produce a reliable and versatile token tree for use in code analysis and generation. The language syntax is designed to be compatible with a modification to the Shunting-Yard algorithm.

Each keyword and operator has a nodal definition (defines the rules for how it fits into the AST) that specifies possible child leaf configurations. Abbreviated, these designations are as follows:

**NoChild (NC)**: The verb accepts no arguments and acts as a standalone.

**Unary (U)**: Accepts one argument immediately proceeding it.

**BinaryLeftRight (BLR)**: Accepts two arguments immediately on either side.

**BinaryRightRight(BRR)**: Accepts two arguments to the immediate right and proceeding thereafter.

**BinaryChain(BC)**: Used in lists, consists of the pattern A1 BC A2 BC … An-1 BC An. When one is found, every other token will be checked. When chain token is no longer found, all elements within the found range will be collected into a subtree and treated as an array.

**Keywords**

**\****arg denotes optional argument*

|  |  |  |
| --- | --- | --- |
| Keyword | Description | AST Relation |
| if(bool){} | Executes the next instruction or block if condition is non-zero. Accepts a condition (bool) and instruction. | BRR |
| else{} | Executed only if previous if statement fails evaluates zero. Accepts an instruction. | U |
| while(bool){} | Loops next instruction whilst the given argument is non-zero. | BRR |
| foreach(var:sym) | Accepts an array type variable (var) and iterates through each member, with each new iteration producing the local variable (sym). The local variable (sym) does not require explicit type declaration. | BRR |
| continue | Jump to start of closest scoped loop or switch. If used within a switch, it becomes a loop for conditions with this statement. | NC |
| break | Jump to end of closest scoped loop or switch. | NC |
| return *arg, arg,...* | Return from current function. May have zero or greater return values. Void return type is assumed if omitted. | NC/U |
| class *symbol* {} | Defines a custom class. Accepts a unique symbol name and body definition. | BRR |
| goto *tag* | Jump to a tagged point in code within the same function. | U |
| null | Alias for 0 | NC |

**Operators**

\**arg* denotes optional argument

|  |  |  |  |
| --- | --- | --- | --- |
|  | Operator | Description | AST Relation |
| = | Set | Set the value of L to R | BLR |
| + | Sum | Sum L to R operand | BLR |
| - | Difference | Subtract R from L | BLR |
| \* | Product | Multiply L and R | BLR |
| / | Quotient | Divide L by R | BLR |
| \*\* | Power | Raise L to the R power | BLR |
| % | Modulo | Return the remainder of L/R (integer only) | BLR |
| += | Set Sum | Set L = L+R | BLR |
| -= | Set Diff | Set L = L-R | BLR |
| \*= | Set Product | Set L = L\*R | BLR |
| /= | Set Quotient | Set L = L/R | BLR |
| ^ | Invert | Invert bits on operand. | U |
| && | Logical And | Return 1 if both L and R are non-zero. Otherwise, 0. | BLR |
| || | Logical Or | Return 1 if either L or R are non-zero. | BLR |
| ! | Logical Not | If immediate right is zero, return 1. Else, return 0. | U |
| !& | Nand | Return 1 if either L or R are non-zero. | BLR |
| !| | Nor | Return 1 if L and R are zero. | BLR |
| ^| | Xor | Return 1 if exactly one of L and R are 1 | BLR |
| ^! | XNor | Return 1 if both or neither of L and R are 1 or 0 simultaneously. | BLR |
| & | Bitwise And | Return bitwise AND L and R | BLR |
| | | Bitwise Or | Return bitwise OR L and R | BLR |
| c++ | Post Increment | Return c and then increment. | U |
| c-- | Post Decrement | Return c and then decrement. | U |
| ++c | Pre Increment | Increment c, and return result. | U |
| --c | Pre Decrement | Decrement c, and return result. | U |
| [\*arg] | Define Array | *Type[\*arg]symbol* Define *Symbol* as an array of *type*. If an integer *arg* is provided, the array is initialized at the provided size at default values. Otherwise, the array is empty. | BLR |
| , | List | Define a comma separated list | BC |
| @ | Reference | Used in function argument list to accept the memory address of a parameter instead of its value. This variable will then be treated as a normal variable. |  |

**PreProcessor Directives**

All directives begin with ‘@’ without whitespace. Each directive may have at most one argument; for example ‘@use io’. These directives are executed before other code is processed, and then discarded.

|  |  |
| --- | --- |
| Directive | Description |
| @use *library* | Import a library |
| @define *name value* | Define an alias with a value |

**Modifiers**

Modifiers follow the pattern *Declaration:Modifier1,Modifier2,...* and are used to set attribute flags on variable or function instances.

|  |  |  |
| --- | --- | --- |
| Modifier | Description | Supported Types |
| static | Establishes a global function within a function namespace. Establishes a global variable shared between class instances. | Variable, Function |
| public | Flags a class member as externally accessible.  Flags a translation unit function as externally accessible. | Variable, Function, Class |
| private | Flags a class member as externally hidden (default).  Flags a translation unit function as externally hidden (default). | Variable, Function, Class |

**Data Types**

|  |  |  |  |
| --- | --- | --- | --- |
| Type | Keyword | Size (Bytes) | Range |
| Integer | int | 4 | -2,147,483,648 - 2,147,483,647 |
| Short Integer | sint | 2 | -32,768 - 32,767 |
| Double | real | 8 | -1.7E+308 to +1.7E+308 |
| Character | char | 1 | -127 - 127 |
| Byte | byte | 1 | 0 - 255 |
| String | string | 1N | N bytes |
| Class | class |  |  |
| Bool | bool | 1 | Either 0 - 255 or -127 - 127 |
| Function | function<type> |  |  |