HULK

Rafael A. Sanchez Martinez
University of Havana
Group: C-113

Figure 1: Havana University Language for Kompilers

1 Introduccion

Welcome to the user manual of HULK, the programming language of the University of Havana. This manual is divided into 5 sections, each one dedicated to a part of the interpreter. The first section explains how to install the interpreter, the second section explains how to use the interpreter, the third section explains how to use the lexical analyzer, the fourth section explains how to use the syntax analyzer and the fifth section Explains how to use the expression evaluator.

1.1 Start

- To start using the interpreter, you must download the source code of the interpreter from the project's github repository. Once the source code has been downloaded, the project must be run using the 'dotnet run' command from a terminal open in the project folder.
- Another option is to run the script which will take care of everything
- See previous report in `https://github.com/ARKye03/mini_kompiler.git`

1.2 Summary

HULK (Havana University Language for Compilers) is an educational, type-safe, object-oriented and incremental programming language, designed for the Introduction to Compilers course of the Computer Science degree at the University of Havana.

A simple 'Hola Mundo' in HULK looks like this:
print('Hola mundo')

From a bird's eye view, HULK is an object-oriented programming language, with simple inheritance, polymorphism, and class-level encapsulation. Additionally, in HULK it is possible to define global functions outside the scope of all classes. It is also possible to define a single global expression that constitutes the entry point to the program.

Most syntactic constructs in HULK are expressions, including conditional statements and loops. HULK is a statically typed language with optional type inference, meaning that some (or all) parts of a program can be annotated with types and the compiler will check all operations for consistency.

But, in this case, since it is a project for 1st year students, it is a bit simplified. In this case, let's say, we have a one-line interpreter, which allow us, to evaluate expressions like:

- Like I said above, print("Hello World")
  - Which is equivalent to $print(\text{“}Hello'' + \text{“}World'')$
  - And print("Hello" @ " World")
- All basics 'instructions' can handle simple math expressions like:
  - $\text{“}2 \pm 5;''$ —— $\text{“}147.6 \pm -63;''$
  - $\text{“}8 \times 2;''$ —— $\text{“}200 \div 3;''$
  - $\text{“}2^{64};''$ —— $\text{“}sqrt(3969)'' \rightarrow \text{“}\sqrt{3969}''$ —— $\text{“}\log(2, 10);''$
  - Simple functions to handle trigonometrics expressions.
    - $\text{“}\sin(\pi)''$
    - $\text{“}\cos(\pi)''$
    - $\text{“}\tan(\pi)''$
    - $\text{“}\cot(\pi)''$
    - Others can be defined on the way too.
  - And thats pretty much it $\wedge \times \wedge$

- Evaluable expressions are:
  - Printing:

    This expression keyword receives another expressions as argument, and evaluates it, and print the returned value

    $print(55)$; shows 55 —— $print(\text{“}Kitty\text{”})$; shows Kitty
  - Variables:

    This one, contains two important keywords, “LetKeyword” and “InKeyword”.

    A basic let-in expression is: “let $< var\_name >$ in $< statement >$”

    Example: “let x = $(63 \pm 109)^2$ in $print(x)$; ”
  - Conditions:

    Basic and classic conditions:

    if (condition) $< do\_if\_true >$ else $< do\_if\_false >$

    “if (1024 % 2 == 0) $print(\text{“}Even\text{”})$ else $print(\text{“}Odd\text{”})$; ”
  - Functions:
    - Declare functions:

      To declare functions simple do:

      function $function\_name$(arguments) => $< statement >$;

      Example: function Pow(x, y) => x^y;

      This can be used like:

      “let number = $Pow(2, 5)$ in $print(number)$; ”
  - Note: A statement is basically another instruction or expression.
- Some expressions can be:
  1. print(“Hello World”);
  2. print($((((1 + 2)^3) * 4) / 5$);
  3. print($\sin(2 * PI)^2 + \cos(3 * PI / \log(4, 64))$);
  4. function $\tan(x)$ => $\sin(x) / \cos(x)$;
  5. let x = PI/2 in $print(\tan(x))$;
  6. let number = 42, text = “The meaning of life is” in print(text @ number);
     - let number = 42 in (let text = “The meaning of life is” in (print($text$ @ $number$)));
  7. print($7 + (let\ x = 2\ in\ x * x)$);
  8. let a = 42 in if (a % 2 == 0) print($\text{“}Even\text{”}$) else print($\text{“}odd\text{”}$);
  9. let a = 42 in print(if (a % 2 == 0) “even” else “odd”);
  10. function fib(n) => if (n > 1) fib$(n - 1)$ + fib$(n - 2)$ else 1;
- And on and on, use your imagination, except if you are my programming teacher, please don’t get creative, I beg you.

2 Lexer

2.1 Lexer Summary

A Lexer, also known as a lexical analyzer, tokenizer, or scanner, is a program or function that breaks down input code into a sequence of tokens. These tokens are meaningful units of code.

In the process of parsing programming languages, this task is traditionally split up into two phases: the lexing stage and the parsing stage. The lexer's job is simpler than the parser's. It turns the meaningless string into a flat list of things like "number literal", "string literal", "identifier", or "operator". It can also recognize reserved identifiers ("keywords") and discard whitespace.

The output of the lexer is used as the input for the parser. The parser then has the much harder job of turning the stream of "tokens" produced by the lexer into a parse tree representing the structure of the parsed language. This separation allows the lexer to do its job well and for the parser to work on a simpler, more meaningful input than the raw text.

More info about:

- What's a Lexer
- The Lexer - ACCU
- Extra blog

2.2 Now, my Lexer

First I've defined some simple Token types in this "enum":

```
public enum TokenType
{
    FLinq, //Link token for function declaration
    ComparisonOperator,
    Number, // Number
    StringLiteral, // Literally a string
    FunctionDeclaration, // Well, deprecated
    LetKeyword,
    IfKeyword,
    ElseKeyword,
    PrintKeyword,
    InKeyword,
    FunctionKeyword,
    Operator, // '+','-','*','/','^', '='...
    Punctuation, // '(',')' & ','
    Identifier, //Used for variables name and functions value
    EOL, //';'
    EOF, //Not yet
    Semicolon, //Idk man/woman
    Separator // '@', and in a future, "||", "&&"...
}
```

Basically, my Tokens, have a few properties:

```
public TokenType type; // From enum
public string value; // Text of the token
public int line; // Line position, currently 1, always
public int column; // Column position
```

`Note:` ToString override is there to test tokens only.

Some functions of my lexer

- One char to the right:

```
private void advance()
{
    pos++;
    column++;
    if (pos > text.Length - 1)
    {
        current_char = '\0';
    }
    else
    {
        current_char = text[pos];
    }
}
```

- Jumps to next non-blank char:

```
private void skip_whitespace()
{
    while (current_char != '\0' && char.IsWhiteSpace(current_char))
    {
        advance();
    }
}
```

- Takes a look to next token, without taking it:

```
private char peek()
{
    var peek_pos = pos + 1;
    if (peek_pos >= text.Length)
    {
        return '\0';
    }
    else
    {
        return text[peek_pos];
    }
}
```

- Drop a token to be used later:

```
public void unget_token(Token token)
{
    readTokens.Insert(0, token);
}
```

The main function

The get_next_token() method it's actually a little bit big, it used several times in the 'Interpreter' itself, since it is in charge of tokenizing/lexing the expression as it is parsed.

3 Introduction

This is the report of my compiler project. The compiler is written in C++ and

4 Introduction

This is the report of my compiler project. The compiler is written in C++ and

5 Introduction

This is the report of my compiler project. The compiler is written in C++ and