NovaLang: a simple, interpreted language

Dennis Bejze

***Abstract*—As part of the study exam in the course 'Compiler Construction' I had to develop my own programming language. This paper describes the language *NovaLang*, which I developed for this purpose. NovaLang is an interpreted language that also has a simple type system. The interpreter performs syntax and type analysis before actually performing the interpretation. As the requirements have specified, the language is built using the tools *Bison* and *Flex*.**

I. Components of a source file

Source files in NovaLang follow usual principles of scripting languages by starting the interpretation from the top of the file. As the language has no concept of packages, a single source file is the only unit of interpretation. [1]

A. Comments

There are only single-line comments, that can be started with the `//` characters.

B. Semicolons

Semicolons are used to separate statements. They are not required for single expressions as last parts of a block (see Sec. V, Sec. VI).

C. Identifiers

Identifiers are sequences of letters (either upper or lower case), digits and underscores. Every identifier that is not a keyword is considered to be a variable name. Digits may be first characters in identifiers, which most programming languages do not allow. [2]

D. Keywords

The following keywords are reserved and cannot be used as identifiers:

```
if   else  for   when  default  fn  mod
```

E. Operators

The following operators are supported:

```
+   -   *    /    ==   !=    :     =   <   >
<=   >=   |>   [ ]   ->
```

F. Literals

NovaLang supports integer, boolean and string literals. Integer literals are sequences of decimal digits, but there is no support for literals of other bases. Leading zeros are not disallowed - they are simply stripped while lexing.

Boolean literals are `true` and `false`. String literals are sequences of characters enclosed in double quotes.

[1] However, there are several built-in functions available.

[2] But to be honest, this is not necessarily a good idea - I just did not had a conflict with this in development.

II. Variables

Variables are used to store values, that can be used later in the program. They are declared by using the `:` operator and can be assigned a value using the = operator.

```
1       x: int;
2       x = 5;
```

A shorthand notation for the declaration and assignment of a variable is also supported:

```
1       x := 5;
```

In this case, the type of the variable is inferred from the type of the expression on the right-hand side of the assignment.

III. Types

NovaLang has a simple type system, which includes the following basic types:

- `int` - a 32-bit signed integer
- `bool` - a boolean value
- `str` - a sequence of characters

There are additional types that are of special meaning:

- `void` - is used to indicate that a function does not return a value.
- `unknown` - is used to indicate that the type of a variable could not be inferred by the interpreter.

All special types except `void` are used internally by the interpreter and cannot be used in the source code.

A. Type compatibility

The interpreter performs type checking to ensure that the types of operands in an expression are compatible. Types of operands must be strictly equal, but type casts for string and integer types are available.

B. Arrays

The language supports arrays of any basic type. Arrays are declared by appending `[]` to the type of the array elements. The array literal lets you create an array with a list of elements.

```
1       x: int[];
2       x = [1, 2, 3];
```

The use of the shorthand notation is not supported in the case of an empty array, as the type of the array cannot be inferred when there are no elements defined.

Single elements of an array can be accessed using the `[]` operator and are treated like normal variables, in the sense that they can be assigned a new value or used in expressions.

Arrays are passed by value, so the underlying array is copied before it is assigned to another variable or passed to a function.

IV. Blocks and scopes

A. Blocks

A block is a sequence of statements or expressions declared within curly braces. Blocks can be nested, and each block introduces a new scope.

B. Scopes

A scope is a region of the program where a variable can be used. There are several types of scopes in NovaLang:

- The *global scope* is the outermost scope. This scope is implicitly created with the start of the program. Variables declared in the global scope are accessible from any part of the program.
- A *function scope* is the scope of a function. Symbols with identifiers already defined in the global scope can be redefined and result in the global symbol being shadowed.
- A *block scope* is the scope of a block. It is created by `if` statements, `for` loops and `when` expressions.
- The *local scope* is a special scope that is created for the pipe operator.

V. Functions

Functions are used to encapsulate a sequence of statements and can optionally return a value. Functions are declared using the `fn` keyword, followed by the name of the function, a list of parameters and the return type of the function.

```
fn add(a: int, b: int): int {
    a + b
}
```

The last expression of a non-`void` function is considered to be the return value of the function. Parameters are passed by value.

The language has also several built-in functions that are available to the user. Grouping of the functions is done via namespaces that correspond to a subset of types (`str`, `int`, `bool`). The notation for calling a built-in function is `namespace::function(arguments)`.

```
str::split("Hello World", " ") // ["Hello", "World"]
```

Note that every array type acts as a dedicated namespace for corresponding functions for operations on arrays.

VI. Control flow

NovaLang supports the following control flow statements:

- `if` - a conditional statement
- `for` - a loop that iterates as long as a condition is true
- `when` - the expression equivalent of a `if` statement

A. If statements & when expressions

An `if` statement is used to execute a block of code if a condition is true.

```
if x > 5 {
    print("x is greater than 5");
} else {
    print("x is less than or equal to 5");
}
```

The semantics of the `when` expression are the same as the `if` statement, but the `when` expression is used to compute a value conditionally.

```
result := when {
    x > 5 {
        "x is greater than 5"
    },
    default {
        "x is less than or equal to 5"
    };
```

Branches can omit their curly braces and preprend an arrow if they contain only a single expression.

```
result := when {
    x > 5 -> "x is greater than 5",
    default -> "x is less than or equal to 5"
};
```

VII. Expressions

Expressions are used to compute values. The type that is emitted by an expression can be used for further type inference while interpreting.

A. Arithmetic & comparison expressions

The language supports the usual arithmetic operators: `+`, `-`, `*`, `/`, `mod`. The `+` operator is also used to concatenate strings.

Comparison operators are also supported: `==`, `!=`, `<`, `>`, `<=`, `>=`.

B. Pipes

The pipe operator `|>` is used to chain expressions together. The result of the previous expression is made available through a `it` with its type being inferred from the previous expression. The last expression's value is the result of the whole pipe expression.

```
"NovaLang is cool"
    |> str::upcase(it)
    |> str::split(it)
    |> str[]::print(it); // ["NOVALANG", "IS", "COOL"]
```

Even though the pipe operator will usually be used as a expression, they can also be used as a statement. In this case, the result of the last expression is discarded.