# FenneC Language Reference

This document describes FenneC: a small C-like language designed for educational purposes. The language was designed for the Compiler Construction course at the Vrije Universiteit. The language is named after the fennec fox, and it has a mascot!

## 1. Introduction

With educational purposes in mind, FenneC is designed to look familiar in terms of syntax and types, and as such carries a high resemblance to C. However, it attempts to avoid implementation pitfalls such as pointers and undefined variables. Furthermore, the concepts used map easily to an LLVM backend. Both the format of this document and the language specification itself are inspired by (but not related to) the CiviC language. Acknowledgements are due to Clemens Grelck, who invented CiviC for the Compiler Construction course taught at the University of Amsterdam. Special thanks go out to Mei-Li Nieuwland for creating our mascot Felix.

A FenneC program is very similar to a C program: it consists of variables and functions and its entry point is the `main` function. Different FenneC source files may be compiled separately into separate object files, to be linked together afterwards. Moreover, since we use an LLVM backend which produces native code, FenneC object files may be linked together with any native object file, allowing a FenneC function to call a C function from a separate source file (given that the function signature can be mapped to a valid FenneC type). The main practical reason for this design point is that it provides programs with access to powerful C standard library functions, such as `printf`.

Sections 2 to 6 will gradually introduce the syntax of FenneC as snippets of grammar rules preceded by corresponding descriptions. Appendix A combines the snippets into a full grammar. The grammar rules are loosely typed. Type restrictions discussed in the corresponding descriptions are (mostly) not part of the syntax, but are enforced by a separate type checking pass in the compiler. The grammar rules follow a BNF-like syntax:

- Production rules and terminals are *Capitalized*.
- Literals are **`bold`**.
- Optional elements are *[*enclosed*]* in italic square braces.
- *Term*$*$ indicates zero or more *Term*s.
- *Term*$+$ indicates one or more *Term*s.

## 2. Types

FenneC has four basic types:

- `bool`: Either `true` or `false`.
- `char`: A signed single byte value (8 bits).
- `int`: A 32-bit signed integer value.
- `float`: A double precision floating point value.

In addition, FenneC supports arrays, which are consecutive sequences of values in memory. An array type is defined by adding the suffix `[]` to a basic type. E.g., an array of integers has type `int[]`:

*Type* ⇒ **`bool`** | **`char`** | **`int`** | **`float`**
| *Type* *[* **`[]`** *]** 

Multi-dimensional arrays (arrays of arrays) are also supported. For example, the `argv` argument of `main` has type `char[][]`. Although multi-dimensional arrays exist as function parameters, however, FenneC has no syntax to define them as local variables, as an effort to reduce complexity.

## 3. Global values

A FenneC program consists of one or more declarations or definitions of global variables and/or functions. A *declaration* is prefixed with the `extern` keyword and imports a value from an external source (commonly a library). A *definition* defines a new value within the program. All definitions are exported to other modules by default, but may be prefixed with the `static` keyword to limit their visibility to the scope of the current module.

*Program* ⇒ *Declaration+*
*Declaration* ⇒ *GlobalDec* | *GlobalDef* | *FunDec* | *FunDef*

A global variable declaration may have any type, including array types (which may be used to import standard library pointers such as `environ`):

*GlobalDec* ⇒ **`extern`** *Type Id* **`;`**

A global variable definition may only have a basic type[1] and requires a constant initializer:

*GlobalDef* ⇒ *[* **`static`** *]* *Type Id* **`=`** *Const* **`;`**

A function declaration declares the function type as a return type and a number of parameter types. A `void` keyword indicates that a function does not return a value. The "varargs" parameter `...` indicates that the function has an unspecified number of additional arguments of which the types are unknown at compile time. This is the case for `printf`, for example, which is declared as `extern int printf(char[] format, ...);`:

*FunDec* ⇒ **`extern`** *ReturnType Id* **`(`** *[ParamsVararg]* **`)`** **`;`**
*ReturnType* ⇒ *Type* | **`void`**
*ParamsVararg* ⇒ *Params [* **`,`** **`...`** *]* | **`...`**
*Params* ⇒ *Param [* **`,`** *Param]**
*Param* ⇒ *Type Id*

A function definition also declares a function type, but does not allow a variable number of parameters (handling such parameters would greatly complicate the language and compiler). It also specifies a function body consisting of zero or more *statements* which are described in Section 4.

*FunDef* ⇒ *[* **`static`** *]* *ReturnType Id* **`(`** *[Params]* **`)`** **`{`** *Statement** **`}`**

[1] Global arrays are not allowed to avoid complexity in the compiler frontend: non-constant initializers must be expanded into instructions and would thus need to be moved into a function body, and constant initializers still lead to additional edge cases that would need to be handled during type checking and IR generation.

# 4. Statements

Function bodies consist of statements. The statement language of FenneC is a subset of that of C, and the syntax is almost identical:

*Statement* ⇒ *Block*
| *VarDef*
| *ArrayDef*
| *Assignment*
| *ExprStatement*
| *ControlFlow*
| *Return*

A group of statements may be grouped in curly braces ({}) to form a block:

*Block* ⇒ **{** *Statement** **}**

A local variable is defined similarly to a global variable, with a mandatory initializer. The initializer, however, need not be constant; it can be any expression that matches the defined type. An array variable is defined with a size expression (which also need not be a constant) following the type. All arrays are zero-initialized according to their type (`false`, `'\0'`, `0`, `0.0`).

*VarDef* ⇒ *Type Id* `=` *Expr* `;`
*ArrayDef* ⇒ *Type* `[` *Expr* `]` *Id* `;`

A variable may be assigned a value directly with the `=` operator, or modified relative to its current value such as with the `+=` operator. `i+=1` is simply a shorthand for `i=i+1`. The left-hand side of an assignment can be a variable or an array index, e.g., `arr[1][2]+=1` is legal syntax:

*Assignment* ⇒ *VarRef* `=` *Expr* `;`
| *VarRef Modifier Expr* `;`
*VarRef* ⇒ *Id* | *Index*
*Index* ⇒ *Expr* `[` *Expr* `]`
*Modifier* ⇒ `+=` | `-=` | `*=` | `/=` | `%=`
| `||=` | `&&=`

Any expression may be used as a statement by suffixing it with a semicolon. This forces evaluation of the expression, ignoring the resulting value. This is useful for function calls, for example:

*ExprStatement* ⇒ *Expr* `;`

Control flow statements are the same as in C, but with stricter types: since there are no type casts in FenneC, predicate expressions must have type `bool`. The body of a loop or if-statement may be any statement, including a block of statements grouped by {}.

The for-loop is a special case which is different from C (but the same as in OCaml, for instance): it always defines an induction variable of type `int`, which loops from (integer) start to stop expressions with step size 1. E.g., `for (int i = 0 to 3)` loops over indices [0, 1, 2]. The start and stop expressions are always evaluated exactly once, before the first loop iteration.

`break` immediately jumps out of the current loop and `continue` jumps to its next iteration. In a while-loop or do-while-loop, this means that `continue` skips the rest of the body and re-evaluates the predicate condition, whereas in a for-loop it first increments the induction variable:

*ControlFlow* ⇒ **`if (`** *Expr* **`)`** *Statement [*`else` *Statement]*
| **`while (`** *Expr* **`)`** *Statement*
| **`do`** *Statement* **`while (`** *Expr* **`) ;`**
| **`for (int`** *Id* **`=`** *Expr* **`to`** *Expr* **`)`** *Statement*
| **`break ;`**
| **`continue ;`**

A non-void function may return a value at any point in the function. A void function may also return before reaching the end of the function, without specifying a value:

*Return* ⇒ **`return`** *[Expr]* **`;`**

# 5. Expressions

FenneC has unary and binary operators, for which the precedence rules of C apply. Operator precedence can be overruled by the use of parentheses. Other expressions are (non-void) function calls, variable uses (see Section 4) and constants:

| | | |
|---|---|---|
| *Expr* | ⇒ | *UnaryOp Expr* |
| | \| | *Expr BinaryOp Expr* |
| | \| | `(` *Expr* `)` |
| | \| | *FunCall* |
| | \| | *VarRef* |
| | \| | *Const* |
| *FunCall* | ⇒ | *Id* `(` *[Expr [*`,` *Expr]*]* `)` |

Unary minus is defined on chars, integers and floats. Logical negation (`!`) is only defined on booleans:

| | | |
|---|---|---|
| *UnaryOp* | ⇒ | `-` \| `!` |

Binary operators are the same as in C. Equality operators are defined for all basic types. Relational and arithmetic operations are defined on chars, integers and floats. Logical operators are only valid on booleans and have implicit control flow: the right-hand side is lazily evaluated. E.g., `true && foo()` never calls `foo`. Comparison operators on `NaN` floats are always `false`, except for `!=` for which they are always `true`:

| | | |
|---|---|---|
| *BinaryOp* | ⇒ | *EqOp* \| *RelOp* \| *ArithOp* \| *LogicOp* |
| *EqOp* | ⇒ | `==` \| `!=` |
| *RelOp* | ⇒ | `<` \| `<=` \| `>=` \| `>` |
| *ArithOp* | ⇒ | `+` \| `-` \| `*` \| `/` \| `%` |
| *LogicOp* | ⇒ | `&&` \| `\|\|` |

Character and string constants are defined as in C, surrounded by single and double quotes respectively. Single-character escaping, such as `"\n"`, is supported, octal and hexadecimal escaping are not. String constants are the only constants with a non-basic type (`char[]`). They are read-only[2], allocated globally (as in C), and zero-terminated. Integer constants may be decimal or hexadecimal, the latter prefixed by `0x`. Floating point constants support notation with omitted numbers on either side of the decimal point, such as `0.` and `.0`.

| | | |
|---|---|---|
| *Const* | ⇒ | *BoolConst* \| *CharConst* \| *IntConst* \| *HexConst* |
| | \| | *FloatConst* \| *StringConst* |
| *BoolConst* | ⇒ | `true` \| `false` |
| *IntConst* | ⇒ | *Digits* |
| *HexConst* | ⇒ | `0x[0-9a-fA-F]+` |
| *FloatConst* | ⇒ | *Digits* `.` *Digits* \| *Digits* `.` \| `.` *Digits* |
| *Digits* | ⇒ | `[0-9]+` |

# 6. Comments and preprocessing

FenneC supports single-line comments by ignoring everything following `//` on a source line, and multi-line comments wrapped in `/* these delimiters */`. Its implementation also invokes the C preprocessor, so you can use preprocessor directives like `#define` and `#include`.

[2] To create a mutable string, you can define an array of sufficient size and overwrite it using a standard library function like `strcpy` with a constant string as argument.

# 7. Scoping rules

Scoping rules define how variable uses map to variable definitions, and when new variables can *shadow* variables of the same name that already in the scope. Intuitively, C scoping rules also apply to FenneC in most cases.

The *scope* is the mapping of identifiers to variable/function definitions. The scope is divided in different scoping levels. Variables and functions defined within the same level must have unique names. A definition in a higher scope can *shadow* a definition in a lower scope with the same name, meaning that it takes precedence when subsequently using a variable of that name. For example, a local variable definition `int i = 1` may legally shadow a global declaration `extern int i`, but another definition `int i = 2` following the former produces an error because it defines a duplicate variable in the same scope. The lowest scoping level is the global scope, comprising all functions and global variables. Each function defines a new scope on top of the global scope, initially containing the function itself and its parameters. Local variable definitions are added to the current scope in the order of their definitions. This means that a local variable cannot be referenced before it is defined. A compound statement (a block of statements enclosed by {}) defines a new higher scope, allowing the shadowing of variables defined earlier in the function. Finally, a for-loop defines a new scope containing its induction variable, meaning that different for-loops can use the same induction variable name.

The annotated example program below illustrates the scoping rules explained above. Each scope maintains it own symbol table, mapping variable/function names to line numbers:

```
 1                          // resolves to line // scope
 2 int i = 1;               //                  // []
 3 void foo(int p) {        //                  // [{i:2}]
 4   int i = i;             // i:2              // [{i:2}, {foo:3, p:3}]
 5   {                      //                  // [{i:2}, {foo:3, p:3}, {i:4}]
 6     int i = i;           // i:4              // [{i:2}, {foo:3, p:3}, {i:4}]
 7     foo(i);              // i:6              // [{i:2}, {foo:3, p:3}, {i:4}, {i:6}]
 8   }                      //                  // [{i:2}, {foo:3, p:3}, {i:4}]
 9   for (int i = 0 to 10)  //                  // [{i:2}, {foo:3, p:3}, {i:4}]
10     p += i;              // p:3, i:9         // [{i:2}, {foo:3, p:3}, {i:4}, {i:9}]
11   foo(i + p);            // i:4, p:3, foo:3  // [{i:2}, {foo:3, p:3}, {i:4}]
12 }                        //                  // [{i:2}]
```

# A. Complete grammar of FenneC

*Program* ⇒ *Declaration+*
*Declaration* ⇒ *GlobalDec* | *GlobalDef* | *FunDec* | *FunDef*
*GlobalDec* ⇒ `extern` *Type Id* `;`
*GlobalDef* ⇒ *[*`static`*] Type Id* `=` *Const* `;`
*FunDec* ⇒ `extern` *ReturnType Id* `(` *[ParamsVararg]* `)` `;`
*FunDef* ⇒ *[*`static`*] ReturnType Id* `(` *[Params]* `)` `{` *Statement** `}`
*ReturnType* ⇒ *Type* | `void`
*ParamsVararg* ⇒ *Params [*`,` `...`*]* | `...`
*Params* ⇒ *Param [*`,` *Param]**
*Param* ⇒ *Type Id*
*Statement* ⇒ *Block*
| *VarDef*
| *ArrayDef*
| *Assignment*
| *ExprStatement*
| *ControlFlow*
| *Return*
*Block* ⇒ `{` *Statement** `}`
*VarDef* ⇒ *Type Id* `=` *Expr* `;`
*ArrayDef* ⇒ *Type* `[` *Expr* `]` *Id* `;`
*Assignment* ⇒ *VarRef* `=` *Expr* `;`
| *VarRef Modifier Expr* `;`
*VarRef* ⇒ *Id* | *Index*
*Index* ⇒ *Expr* `[` *Expr* `]`
*Modifier* ⇒ `+=` | `-=` | `*=` | `/=` | `%=`
| `||=` | `&&=`
*ExprStatement* ⇒ *Expr* `;`
*ControlFlow* ⇒ `if` `(` *Expr* `)` *Statement [*`else` *Statement]*
| `while` `(` *Expr* `)` *Statement*
| `do` *Statement* `while` `(` *Expr* `)` `;`
| `for` `(` `int` *Id* `=` *Expr* `to` *Expr* `)` *Statement*
| `break` `;`
| `continue` `;`
*Return* ⇒ `return` *[Expr]* `;`
*Expr* ⇒ *UnaryOp Expr*
| *Expr BinaryOp Expr*
| `(` *Expr* `)`
| *FunCall*
| *VarRef*
| *Const*
*UnaryOp* ⇒ `-` | `!`
*BinaryOp* ⇒ *EqOp* | *RelOp* | *ArithOp* | *LogicOp*
*EqOp* ⇒ `==` | `!=`
*RelOp* ⇒ `<` | `<=` | `>=` | `>`
*ArithOp* ⇒ `+` | `-` | `*` | `/` | `%`
*LogicOp* ⇒ `&&` | `||`
*FunCall* ⇒ *Id* `(` *[Expr [*`,` *Expr]*]* `)`
*Const* ⇒ *BoolConst* | *CharConst* | *IntConst* | *HexConst*
| *FloatConst* | *StringConst*
*BoolConst* ⇒ `true` | `false`
*IntConst* ⇒ *Digits*
*HexConst* ⇒ `0x[0-9a-fA-F]+`
*FloatConst* ⇒ *Digits* `.` *Digits* | *Digits* `.` | `.` *Digits*
*Digits* ⇒ `[0-9]+`
*Id* ⇒ `[a-zA-Z][a-zA-Z0-9_]*`
*Type* ⇒ `bool` | `char` | `int` | `float`
| *Type [*`[]`*]**

## B. Example program

The example below demonstrates the syntax of FenneC. It does not do anything meaningful:

```
extern int printf(char[] format, ...);

static int counter = 0;

int get_counter() {
    counter += 1;
    return counter;
}

static void foo(int len) {
    int[len] arr;
    for (int i = 0 to len)
        arr[i] = i;
}

int main(int argc, char[][] argv) {
    for (int i = 0 to argc) {
        printf("argv[%d]: %s\n", i, argv[i]);
    }
    foo(10);
    return 0;
}
```