Epilog Programming Language

Epilog is a general purpose programming language developed for the elective courses on Programming Languages at Universidad Simón Bolívar. It is inspired, in part, by the syntax of programming languages Prolog[1] and Erlang.

Epilog is imperative and strongly typed. It supports functions, procedures, logical (if) and value (case) selectors, count- (for) and condition-controlled (while) loops, recursion, line and block comments, as well as arbitrarily nested structured (record) and union (either) type definition.

Lexical considerations

Keywords

The following are keywords. This means they are all reserved and they cannot be used as identifiers nor redefined.

> `and`, `boolean`, `character`, `either`, `end`, `false`, `finish`, `float`, `for`, `function`, `if`, `integer`, `is`, `length`, `otherwise`, `or`, `not`, `print`, `procedure`, `read`, `record`, `return`, `string`, `true`, `void`, `while`, `xor`

The following are operator and punctuation marks.

`+`, `-`, `*`, `/`, `%`, `and`, `or`,`|`, `!|`, `<`, `>`, `=>`, `=<`, `=`, `!=`, `:-`, `->`, `,`, `.`, `;`, `_`

Identifiers

An identifier is a sequence of letters (`[A-Za-z]`) and digits (`[0-9]`) of any length, starting with a letter, which isn't a keyword. Identifiers are classified into Variable Identifiers and General Identifiers.

Variable Identifiers must begin with a capital letter (`[A-Z]`), and are used for naming variables.

General Identifiers must begin with a lowercase letter, and are used for naming functions, procedures, structures and union types.

Whitespace

Whitespace (i.e. spaces, tabs, newlines) serves to separate tokens and is otherwise ignored. Keywords and identifiers must be separated by whitespace or a token that is neither a keyword nor an identifier. `returniftrue` is a single identifier, not three keywords; while `if(23this` scans as four tokens.

[1] Prolog has quite a few different compilers, the most widely used of which are GNU Prolog, SWI-Prolog and SICStus.

Constants

A boolean constant is either of the keywords `true` or `false`.

A char constant is specified as `'<char>'`, where `<char>` is either a printable ascii character (An upper or lowercase letter, a digit, the space character, or a symbol different from `\`, `'`, or `"`), or an escape sequence, which is evaluated according to the following table:

Escape Sequence	Character Represented
`\n`	Newline
`\t`	Horizontal Tab
`\\`	Backslash
`\'`	Single quotation mark
`\"`	Double quotation mark

An integer constant can either be specified in decimal (base 10) or hexadecimal (base 16). A decimal integer is a sequence of decimal digits (`[0-9]`). A hexadecimal integer must begin with `0X` or `0x` (that is a zero, not the letter oh) and is followed by a sequence of hexadecimal digits. Hexadecimal digits include the decimal digits and the letters `a` through `f` (either upper or lowercase). Examples of valid integers: `8`, `012`, `0x0`, ‘0X12aE.

A float constant is a sequence of digits, a period, followed by any sequence of digits, at least one. Thus, neither `.12` nor `12.` are valid floats, but both `0.12` and `1.2` are valid. A float can also have an optional exponent, e.g., `12.2E+2` For a float in this sort of scientific notation, the decimal point is required, the sign of the exponent is optional (if not specified, `+` is assumed), and the `E` can be lower or upper case. As above, `.12E+2` and `12.E+2` are invalid, but `1.2E+2` is valid. Leading zeroes on the mantissa and exponent are allowed.

A string constant is a sequence of characters enclosed in double quotes (`"`). Strings can contain any character except a newline or double quote. A string must start and end on a single line, it cannot be split over multiple lines.

Examples:

```
boolean aBoolean is true,

character aCharacter is 'a',
character anotherCharacter is '\n',

integer aInteger is 42,
integer anotherInteger is 0x2A,

float aFloat is 4.2,
float anotherFloat is 6.02E23,
```

```
float yetAnotherFloat is 1.0E-42,

string aString is "Privyet Mir!",
string anotherString is "This is my new string\nA haiku with two newlines\nRefrigerator'
```

Operators

The operators and punctuation characters used in Epilog include

`>>` `<<`

`+`, `-`, `*`, `/`, `%`, `|`, `=`, `/=`, `=<`, `<`, `>=`, `>`, `is`, `and`, `or`, `not`, `xor`, `(`, `)`, `{`, `}`, `:`, `_`, `,`, `.`

Operators have the following procedence, from highest to lowest:

Operator	Description	Associativity
`_`	Access to record element	Left to right

Operator	Description	Associativity
-, not	Unary arithmetic and logical negation	Right to left
*,/,%	Multiplicative	Left to right
-,+	Additive	Left to right
<,=<,>,>=	Relational	Left to right
||	A|B means A divides B	Left to right
and,or,	Conjunction, disjunction,	Left to right
=,!=‘	Equality	Left to right

A boolean expression using logical AND and OR has short circuit evaluation. It means, EPILOG does not evaluate an operand unless it is neccessary to resolve the result of the expression.

Comments

Epilog allows for both single-line comments as well as block comments. A single-line comment starts with `%%` and extends to the end of the line. A block comment starts with `/%` and ends with the first subsequent `%/`. Any symbol is allowed in a comment except the sequence `%/`, which ends the current comment. Block comments do not nest.

Examples:

```
integer X, %% This is a single line comment

/%
 % While this is
 % a block comment.
 %/
```

Reference Grammar

TO DO: The grammar

Program Structure

TO DO: The structure

Scoping

TO DO: Scoping

Variables

TO DO: Variables

Scalar types

boolean

Takes either of the values `true` or `false`.

character

Represents one of the 128 ASCII characters.

integer

Represents signed integers between -2147483648 and 2147483647.

float

Represents an IEEE-754 32-bit floating point number.

Composite types

Arrays

Arrays are homogeneous, linearly indexed collections of a given type and size. The size of an array must always be an integer known at compilation time. Epilog arrays are zero-indexed, that is, the first element of an array is element 0. The declaration of an array obeys the following syntax: ~~erlang : theArray.~~

The elements of an array are accessed using the `:` operator, according to the following syntax: ~~erlang theValue is theArray:42.~~

A programmer who uses arrays can take advantage of the predefined `length` function when writing functions and procedures. This function returns the length of the array passed as argument.

Attempting to access an element outside the bounds of an array produces a runtime error.

Examples: ~erlang integer:100 myNums, for I in {0..length(myNums) - 1} -> read(myNums:I) end,

```
<...>,

integer Z,
for I in {0..length(myNums) - 1} ->
    Z is Z + myNums:I
end,

print(Z),

myNums:100 is 42. %% this produces a runtime error.
```

~

Records

Epilog records are arbitrarily nested structures. This abstraction allows Epilog programmers to define their own data types by meaningfully associating other data types, which can be native types, arrays, eithers or other records.

It is not possible, however, to define a record containing itself, either directly or indirectly, since this would consume an infinite amount of memory.

All record types must be defined in the global scope.

A record type is defined with the following syntax: ~~erlang record :- [,] .~~

To declare a variable of a record type, the usual declaration syntax is used: ~~erlang .~~

To access the entries of a record type, the _ operator is used, followed by the name of the entry. The syntax is the following: ~erlang __. ~ Accessing an entry which is not in the definition of the record produces a compilation time error.

Examples: ~erlang record person :- string Name, integer Age, boolean IsRegistered.

```
<...>

person User,
print("What is your name?"),
read(User_Name),
print("How old are you?"),
read(User_Age),
User_IsRegistered is True.
```

~

Eithers

Either types are structures which might hold a single value at a time. This allows the programmer to make efficient use of memory when she considers it necessary, since the same memory address can be used for either of the member types. The member types of an Either type can be native types, arrays, records or other eithers.

It is not possible, however, to define an either containing itself, either directly or indirectly, since this would stump the compiler indefinitely.

All either types must be defined in the global scope.

An either type is defined with the following syntax: ~~erlang either :- [,] .~~

To declare a variable of an either type, the usual declaration syntax is used: ~~erlang .~~

To access the members of an either type, the _ operator is used, followed by the name of the member. The syntax is the following: ~erlang __. ~ If the member accessed is not the same as the last member that was assigned, the value of the resulting value is undefined.

Accessing a member which is not in the definition of the either produces a compilation time error.

Examples: ~erlang either magic :- float Float, integer Integer.

```
<...>

float N,

<...>

magic Magic,
Magic_Float is N,
Magic_Integer is (1 << 29) + (Magic_Integer >> 1) - (1 << 22),

float SqrtN is Magic_Float,

print(SqrtN).
```

~

Special Types

Strings

Strings are immutable sequences of ASCII characters. They must be initialized at the site of declaration. The constraints for string constants are discussed in the section "Constants" under "Lexical considerations".

Void

TO DO: Are we even going to have the void type?

Type equivalence and compatibility

Epilog has strict types, which means that at no point are values implicitly converted or coerced from one type to another. In other words, a type is only equivalent and compatible with itself. For conversion between native types, the predefined functions `toBoolean`, `toCharacter`, `toIntger` and `toFloat` are provided, with the following semantics:

From ↓	toBoolean	toChar	toInteger	toFloat
boolean*		`'T'` if `true`, `'F'` if false	`1` if `true`, `0` if `false`	`1.0` if `true`, `0.0` if `false`
char	`true` if not `'\0'`	*	padded with zeros	toFloat . toIntger

From ↓	toBoolean	toChar	toInteger	toFloat
integer	`true` if not zero	truncated to 7 bits	*	***TO DO: define conversion***
float	`true` if not zero	toChar . toInteger	***TO DO: define conversion***	*

where the cells marked (*) produce compilation time errors for unnecessary conversions.

Equivalence between record and either types is strictly by name, which means that it is impossible for two values declared as different record types or different either types to be equivalent, even if the structures are the same.

Regarding array types, their equivalence depends completely on the equivalence of their element types, the only instance of structural equivalence in the definition of the Epilog language.

Assignment

TO DO: Assignment

Control Structures

If

The most basic conditional in Epilog is the `if` statement. It has one or more guards (boolean expressions), each with an associated branch (one or more instructions). The guards are evaluated sequentially and when a true one is found, the associated branch is executed and the rest are skipped. If no true guard is found, the program continues after the `end` token. If the programmer wishes to, she can use the keyword `otherwise` in the last guard, similarly to `else` branches in other programming languages.

Syntax: ~~erlang if -> [,][; -> [, <instruction_ij]] end~~

Examples: ˷erlang if greetUser -> print(“Hi $username!”) end,

```
if
    X > 0 -> print("X is positive."), Positives is Positives + 1;
    X = 0 -> print("X is zero."), Zeroes is Zeroes + 1;
    X < 0 -> print("X is negative."), Negatives is Negatives + 1
end,
```

```
if
    2 | Y -> print("Y is even."), Z is 2;
    3 | Y -> print("Y is an odd multiple of three.);
    otherwise -> print("Y is a boring number.")
end.
```

~

======= ### Case The `case` statement takes an integer expression which is evaluated. It then looks for the resulting value in the guards, and finally executes the branch where the value is found. If the value isn't found in any branch, the program continues after the `end` token. If the programmer wishes to, she can use the keyword `otherwise` in the last guard, similarly to the `default` case of other programming languages.

Syntax: ~~erlang case of [,] -> [,][; [,] -> [, <instruction_ij]] end~~

Examples: ~~erlang read(I), case I of 2, 3, 5, 7, 11 -> print("I love small primes."); 4, 8, 16 -> print("Well, at least it's a power of two");~~

While

The `while` statement allows for conditional iteration based on a boolean expression. When the statement is first reached, the expression is evaluated, and if it's `true`, the body is executed. After each execution of the body, the expression is evaluated again, until it becomes `false`, when the loop is broken and the first instruction after the `end` token is executed.

Syntax: ~~erlang while -> [,] end~~

Examples: ~~erlang read(I), while I /= 42 -> print("Keep trying!"), read(I) end, print("You found it!").~~

For

The `for` statement allows for count-controlled iteration. When the statement is first reached, the range is evaluated, the iteration variable is declared and assigned the first value in the range, and the body is executed. After the first iteration, the iteration variables is assigned the second value in the range, and so on, until the last value is reached. Subsequently, execution proceeds regularly after the `end` token.

If a variable with the same name as the iteration variable had been declared in the scope of the `for` statement, it is shadowed by the iteration variable.

Syntax: ~~erlang for in {..} -> [,] end~~

Examples: ~~erlang for I in {0..42} -> if 15 | I -> print("fizzbuzz"); 3 | I -> print("fizz"); 5 | I -> print("buzz"); otherwise -> print(I) e~~

Expressions

TO DO: Expressions

Procedures

A procedure is used to define a routine, so it can be called at any point of the code. Procedures are declared using the keyword `procedure` followed by it's name and parameters. Always returns void.

Syntax: ~~erlang procedure bar(integer X, float Y) :- integer Z is 3, X is X + Z, Y is Y - 1.0.~~

Functions

Functions are pure, it means that evaluating a function has not side effect. A function looks like a procedure but is declared using the keyword `function`, its arguments are read only, global variables are not allowed on its scope and the return type most always be declared explicitly. Functions can return a value of any type, except void.

Syntax: ~~erlang function foo(integer X) -> integer :- integer Y is 4, integer Z is 5, return(X+Y+Z).~~

Procedure and Function invocation

TO DO: Procedure and Function invocation

Run time checks

TO DO: Run time checks

References

Acknowledgements

The structure of this document is based, in part, on the specification of the Decaf programming language.