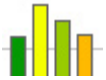


EZGraphs

A graphs and charts generating language

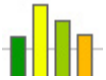
Language Reference Manual

03/05/07

Team:

Edlira Kumbarce **ek2248@columbia.edu**
Vincent Dobrev **vd2006@columbia.edu**

COMS W4115: Programming Languages and Translators
Prof. Stephen A. Edwards
Spring 2007



1. *Lexical Conventions*

1.1 Comments

Single-line comments begin with the characters "`//`" and terminate with an end-of-line marker. Multi-line comments start with the characters "`/*`" and end with "`*/`". Comments cannot be nested.

1.2 Identifiers

An identifier consists of any sequence of letters, digits, and underscores. Its first character must be a letter or an underscore. Identifiers are case-sensitive.

1.3 Keywords

Some identifiers have special meaning in the language. They are reserved as keywords and cannot be used otherwise. The following identifiers are considered keywords:

```
void            include
bool            break
char            continue
int             return
float           if
string          else
true            while
false           do
type            for
```

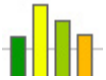
1.4 Booleans

Boolean constants can only have the values `true` or `false`.

1.5 Characters

A character constants is 1 or 2 characters enclosed in single quotes "`'`". Within a character constant a single quote must be escaped by a back-slash "`\`" as do the following characters:

"`\`"	back-slash itself
"`\b`"	backspace
"`\r`"	carriage return
"`\n`"	line feed
"`\t`"	horizontal tab



1.6 Integers

Integer constants are represented by any sequence of digits.

1.7 Floats

Floating point constants consist of any one of the following sequences:

- An integer, a decimal point "."
- An integer, a decimal point ".", a fractional part
- An integer, a decimal point ".", an exponent part
- An integer, a decimal point ".", a fractional part, an exponent part
- An integer, an exponent part
- A decimal point ".", a fractional part
- A decimal point ".", a fractional part, an exponent part

The exponent part consists of either "e" or "E", followed by an optionally signed integer.

1.8 Strings

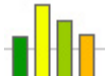
A string constant is any sequence of characters surrounded by double quotes "". Within a string constant a double quote must be escaped by a back-slash "\" as do all the characters that have to be escaped in a character constant except for a single quote "'".

1.9 Line Terminators

Lines are terminated by either one of "\r\n", "\r", or "\n". These characters are recognized and then ignored.

1.10 Whitespace

The blank space and tab characters are considered whitespace and serve to separate tokens. Like comments and line terminators, whitespace is recognized during lexical analysis and then ignored.



1.11 Operator Tokens

Operators are special symbols that perform specific operations on operands and return a result. They may be unary or binary and are evaluated based on an order of precedence, which is indicated in the table that follows, in descending order.

Operator Type	Operators
unary	`+` `-` `!` `++` `--`
multiplicative	`*` `/`
additive	`+` `-`
relational	`<` `>` `<=` `>=`
equality	`==` `!=`
logical AND	`&&`
logical OR	`||`
assignment	`=` `+=` `-=` `*=` `/=`

1.12 Separator Tokens

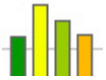
The following symbols are used to separate tokens or combinations of tokens:

```
{   }   (   )   [   ]   ;   ,   .
```

2. *Data Types*

The following data types are supported in the language:

`bool`	a boolean value (`true` or `false`)
`char`	a single character
`int`	a 32-bit integer
`float`	a single-precision floating point
`string`	a sequence of characters



3. *Expressions*

Expressions are the core components of statements. They are constructed using operators, identifiers, constants, function calls, and other expressions. Expressions evaluate to some value in the end by applying operators to constants, function calls, or other expressions using the operator precedence rules as indicated in the section on operator tokens. An expression can be of one of the following:

lvalue
lvalue ++
lvalue --
function-call
constant
unary-expression
multiplicative-expression
additive-expression
relational-expression
logical-expression
assignment-expression
(*expression*)

3.1 Primary Expressions

Primary expressions consist of l-values (identifiers or arrays), increments and decrements of l-values, function calls, constants, or other expressions enclosed in parentheses.

Function calls are of the form

identifier (*expression-list*$_{opt}$)

where *identifier* is the name of the function and *expression-list* is an optional comma-delimited list of expressions to be passed as arguments to the function that is being called.

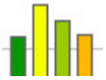
Constants can be boolean, character, integer, floating-point, or string constants.

3.2 Unary Expressions

Unary expressions are expressions prefixed by a unary operator. They are of the form

- *expression*
+ *expression*
! *expression*

The unary + operator returns the value of *expression* unchanged. The unary – operator returns the negative value of *expression*. The logical negation operator `!` returns `false` if *expression* evaluates to `true` and it returns `true` if *expression* evaluates to `false`. Unary expressions associate from right to left.



3.3 Multiplicative Expressions

Multiplicative expressions consist of two *expression* operands and one of the multiplicative operators. They can be applied to integers and floats but not to booleans, characters, or strings. Multiplicative expressions are of the form

expression * *expression*
expression / *expression*

They group from left to right.

3.4 Additive Expressions

Additive expressions consist of two *expression* operands and one of the additive operators. The + operator can be applied to integers, floats, and strings, while the – operator only applies to integers and floats. Additive expressions are of the form

expression + *expression*
expression - *expression*

They group from left to right.

3.5 Relational Expressions

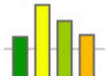
Relational expressions consist of two expression operands and one of the relational operators. They are applicable to integers and floats, and return a boolean value of `true` if:

==	the two operands are equal
!=	the two operands are not equal
<	the first operand is less than the second operand
>	the first operand is greater than the second operand
<=	the first operand is less than or equal to the second operand
>=	the first operand is greater than or equals to the second operand

3.6 Logical Expressions

Logical expressions are only applicable to boolean values. They contain two *expression* operands and either the logical AND operator (&&) or the logical OR operator (||). The AND operator has higher precedence that the OR operator. The logical expressions associate from left to right and are of the form

expression && *expression*
expression || *expression*



3.6 Assignment Expressions

Assignment expressions consist of an l-value as a left operand, one of the assignment operators, and *expression* (whose type is the same as that of the l-value) as the right operand. They group from right to left. Assignment expressions evaluate to the value contained in the left operand and are of the form

lvalue = *expression*
lvalue += *expression*
lvalue -= *expression*
lvalue *= *expression*
lvalue /= *expression*

4. Declarations

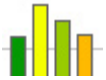
Declarations have the form

declaration:
    type-specifier declarator-list ;

type-specifier:
    `void`
    `bool`
    `char`
    `int`
    `float`
    `string`
    identifier

declarator-list:
    declarator
    declarator , *declarator-list*

declarator:
    identifier
    declarator [$constant_{opt}$]



5. Statements

Statements are building blocks of a program and are executed in the order they appear in the program provided control flow statements do not dictate jumping to different locations.

5.1 Conditional Statement

Conditional statements are used to make decisions at various points in a program. They are of the form

```
if ( expression ) statement
if ( expression ) statement else statement
```

where *expression* has to evaluate to either `true` or `false`. If *expression* evaluates to `true`, the first statement is executed. If it is `false`, the second statement is executed, if present. Conditional statements can be nested.

5.2 While Statement

The `while` statement is of the form

```
while ( expression ) statement
```

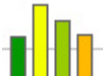
The *expression* must evaluate to a boolean and *statement* continues to be executed as long as expression is `true`. The test takes place before each execution of the *statement*.

5.3 Do Statement

The do statement is of the form

```
do statement while ( expression )
```

The *expression* must evaluate to a boolean and *statement* continues to be executed as long as expression is `true`. The test takes place after each execution of the *statement*.



5.4 For Statement

The `for` statement is of the form

```
for ( expression-1opt ; expression-2opt ; expression-3opt ) statement
```

The statement is equivalent to

```
expression-1 ;
while ( expression-2 ) {
        statement
        expression-3 ;
}
```

5.5 Break Statement

The statement

```
break ;
```

is used to exit from iterative statements such as `while`, `do`, and `for`.

5.6 Continue Statement

The statement

```
continue ;
```

is used to terminate only the current iteration of iterative statements such as `while`, `do`, and `for`.

5.7 Return Statement

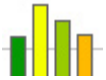
A function returns to its caller by means of the `return` statement, which has one of the forms

```
return ;
return ( expression ) ;
```

5.8 Function Call Statement

The function call statement is of the form

```
function-call ;
```



5.9 Compound Statement

A block of zero or more statements enclosed in curly brackets is also a statement.

6. *External Definitions*

A program consists of a sequence of external definitions which are as follows:

6.1 Include Declarations

The include declaration

```
include "filename" ;
```

results in the replacement of that line by the entire content of the file *fielname*.

6.2 Type Definitions

Type definitions have the form

```
type identifier { declaration-list }

declaration-list:
        declaration
        declaration declaration-list
```

6.3 Function Definitions

Function definitions have the form

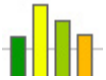
```
type-specifier identifier ( argument-list_opt ) function-body

argument-list:
        argument
        argument , argument-list

argument:
        type-specifier identifier

function-body:
        { statement-list_opt }

statement-list:
        statement
        statement statement-list
```



6.4 Declarations

Any *declaration* that appears outside the body of a function.

7. Scope Rules

The notion of static, open scoping will be applied as follows:

Variables defined in a program externally (outside of functions) are global. Their scope starts at the beginning of the file containing the program and ends at the end of the file. They are accessible from anywhere in the file, inside functions and nested blocks of statements.

Each function in the program has its own scope. Variables defined inside a function can be accessed only within the function's body. The same rule applies for parameters to functions.

Blocks of statements enclosed in curly braces also have their own scope and variables defined inside each block are accessible only within the block.

Variables that have not been initialized are invalid even within their scope area and attempts to use them will generate "uninitialized variable" errors.

8. Predefined Functions

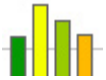
8.1 Data Acquiring Functions

`getData(filename)`

Fills an array (globally defined within the calling function) with data from *filename*.

`genData(min, max, nrPts)`

Fills an array (globally defined within the calling function) with *nrPts* elements ranging from *min* to *max*.



8.2 Drawing Functions

`drawPoint(x,y)`

Draws a point with coordinates (*x*,*y*).

`drawHLine(y)`

Draws a horizontal line with the *y*-coordinate specified as a parameter.

`drawVLine(y)`

Draws a vertical line with the *x*-coordinate specified as a parameter.

`drawRectangle(w,h)`

Draws a rectangle with width *w* and height *h*.

`grid()`

Draws a grid on the graph area.

`plotData()`

Draws a standard graph representation of the data contained in an array defined globally in the calling function.

8.3 Console Output Functions

`print(arg)`

Prints to console the string representation of the argument passed to it.

`scope()`

Prints all the variables in the current scope.