Weak language draft

epoll-reactor

since December 2021

Contents

1 Scope

This document describes requirements for implementation of weak programming language.

1.0.1 Language is not required to have one implementation,
1.0.2 as well as the implementation is not required to be a compiler.

2 Lexical elements

2.1 Keywords

boolean	break	char
continue	do	else
false	float	for
if	int	return
string	struct	true
void	while	

2.2 Operators and punctuators

=	*=	/=	%=	+=	-=
<<=	>>=	&=	|=	^=	&&
||	^	&	|	==	!=
>	<	>	<=	>=	<<
>>	+	-	*	/	%
++	--	.	,	;	!
[	]	(	)	{	}

2.3 Comments

Comments are not involved into the parsing and should be processed at the lexical analysis stage.

2.3.1 All text starting with // should be ignored until the end of line.
2.3.2 All text after /* and before */ character sequences should be ignored.

3 Data types

3.0.1 Language must implement static strong type system.
3.0.2 All casts must be explicit.

3.1 Primitive

int – signed 32-bit;
float – signed 32-bit;

char – 8-bit;
bool – 8-bit;
void – empty type; used as return type only.

3.1.1 Primitive types must be trivial copyable. It means **memcpy()** must copy all primtive types between memory locations without side effects.
3.1.2 No one primitive type can be converted to another implicitly.

3.1.1 String

3.1.1.1 Character sequences must be represented as arrays of character type, for example,

```
char string[25];
```

or

```
char *string = /* Implementation - defined. */;
```

, where implementation-defined part is some memory allocation.

3.2 Aggregate

struct – aggregate type; may constructed from primitive types and from structures.

3.2.1 Structures may be nested.
3.2.2 Maximum structures depth is implementation-defined.
3.2.3 Structure may not contain fields (be empty).

3.3 Arrays

Each type except **void** can represent array, for example,

(**3.3.1**) `bool array[10];`

3.3.2 Arrays should (but not required to) be zero-initialized.
3.3.3 Arrays can be multi-dimensional with arity of any depth greather than 0.
3.3.4 Minimum array size must be at least 1 byte.
3.3.5 Maximum array size is implementation-defined.
3.3.6 Array indexing must start from 0. Zero index must point to first array element, and so on.

3.4 Pointers

(**3.4.1**) `int *ptr = ...;`

Address of the variable can be taken with **&** operator, for example

(**3.4.2**) `int * ptr = &var;`

3.4.3 Pointer cannot have **NULL** (**0**) value.
3.4.4 Pointers cannot be converted to another pointer type in any way.
3.4.5 Pointer to **void** is forbidden. All other types (including structures) are allowed to have pointer to it.
3.4.6 Pointers must allow to change the same memory location through them (for example, in other function).

```
void f(int * ptr) { *ptr = 100; }
```

Now **ptr** must be equal 100 outside this context.
3.4.7 & operator may be applied to non-pointer type.

```
int mem = ...; int * ptr = &mem;
```

3.4.8 & operator may be applied to pointer type.

```
int * ptr1 = ...; int **ptr2 = &ptr1;
```

4 Inside-iteration statements

4.1 Break

(**4.1.1**) `break;`

4.1.2 break usable only inside **while**, **do-while** and **for** statements.
4.1.3 break performs exit from a loop.

4.2 Continue

(**4.2.1**) `continue;`

4.2.2 continue usable only inside the **while**, **do-while** and **for** statements.
4.2.3 continue performs jump to the next loop iteration.

5 Iteration statements

5.1 While

(**5.1.1**) `while` (< `stmt` >) { < `stmt` > * }

5.1.2 while is the loop statement that executes its body until the condition evaluates to false.

5.2 Do-While

(**5.2.1**) `do` { < `stmt` > ∗ } `while` (< `stmt` >);

5.2.2 do-while is the loop statement with similar to **while** semantics, but it executes body before contition check at first time.

5.3 For

(**5.3.1**) `for` (< `stmt` > | ϵ; < `stmt` > | ϵ; < `stmt` > | ϵ) { < `stmt` > ∗ }

5.3.2 for is the loop statement with three initial parts and body.

5.3.3 Initial part
5.3.3.1 should (but not required to) have variable assignment;
5.3.3.2 executed once before all remaining statements in **for** loop.

5.3.4 Conditional part
5.3.4.1 should (but not required to) have expression convertible to boolean;
5.3.4.2 executed before each body part execution.

5.3.5 Incremental part
5.3.5.1 should (but not required to) have binary or unary expression.

6 Conditional statements

6.1 If

(**6.1.1**) `if` (< `stmt` >) {< `stmt` > ∗}

(**6.1.2**) `if` (< `stmt` >) {< `stmt` > ∗} `else` {< `stmt` > ∗}

6.1.3 if is the statament that executes its body if condition is true.
6.1.4 if statement optionally can have **else** part.
6.1.5 if condition is false and **else** part exists, it should be executed.

7 Jump statements

7.1 Return

(**7.1.1**) `return`;

(**7.1.2**) `return` < `stmt` >;

7.1.3 return is the statement being the end of control flow.
7.1.4 If parent function have return type other than **void**, value of this type must be returned on every exit point.
7.1.5 If parent function have **void** return type, control flow may (but not required to) break at any point whith **return;** statement.

8 Functions

(**8.0.1**) $<$ `ret − type` $>$ $<$ `id` $>$ ($<$ `parameter − list` $>$);

(**8.0.2**) $<$ `ret − type` $>$ $<$ `id` $>$ ($<$ `parameter − list` $>$) { $<$ `stmt` $>$ $*$ }

8.1 Return value

Function can return
8.1.1 any primitive type, or
8.1.2 pointer to any primitive type, or
8.1.3 structure type, or
8.1.4 pointer to structure type, or
8.1.5 nothing (**void**).

8.1.6 Return value can be ignored (if any).

8.2 Recursion

8.2.1 Function can call itself (recursive call).
8.2.2 Maximum recursive call depth is implementation-defined.

8.3 Arguments

8.3.1 Arguments count must be at least 0.
8.3.2 Maximum arguments count is implementation-defined.
8.3.3 To be called, a function call statement must have the same arguments count as its declaration.
8.3.4 To be called, a function call statement must have the same argument types as its declaration.
8.3.4 Pointers can be passed to function call statement.

9 Execution flow

9.0.1 Code can be executed only inside functions.
9.0.2 Entry point is the function

```
int main() { ... }
```

or

```
int main(int argc, char * *argv) { ... }
```

9.0.3 Main function can optionally support command-line arguments.
9.0.4 **argc** must be 1 or greater.
9.0.4 **argv[0]** must contain executable file name or any representation of program file (in case if this is an interpreter).
9.0.5 Command-line arguments are passed to main function through received from operation system values.

10 FFI

10.1 Call conventions

10.1.1 cdecl call convention must be used in order to be able to link with other shared libraries supporting this convention.

11 Grammar summary

⟨*program*⟩ ::= *(* ⟨*function-decl*⟩ | ⟨*structure-decl*⟩ *)**

⟨*structure-decl*⟩ ::= `struct` { ⟨*structure-decl-list*⟩ }

⟨*structure-decl-list*⟩ ::= `(` ⟨*decl-without-initialiser*⟩ `;`
| ⟨*structure-decl*⟩ `; )*`

⟨*function-decl*⟩ ::= ⟨*ret-type*⟩ ⟨*id*⟩ `(` ⟨*parameter-list-opt*⟩ `)` { ⟨*stmt*⟩`*`
}

⟨*ret-type*⟩ ::= ⟨*type*⟩
| ⟨*void-type*⟩

⟨*type*⟩ ::= `int`
| `float`
| `char`
| `string`
| `boolean`

⟨*void-type*⟩ ::= `void`

⟨*constant*⟩ ::= ⟨*integral-literal*⟩
| ⟨*floating-literal*⟩
| ⟨*string-literal*⟩
| ⟨*char-literal*⟩
| ⟨*boolean-literal*⟩

⟨*integral-literal*⟩ ::= ⟨*digit*⟩`*`

⟨*floating-literal*⟩ ::= ⟨*digit*⟩`* .` ⟨*digit*⟩`*`

⟨*string-literal*⟩ ::= `''( x00000000-x0010FFFF )*''`

⟨*char-literal*⟩ ::= `'ASCII(0)-ASCII(127)'`

⟨*boolean-literal*⟩ ::= `true`
| `false`

⟨*alpha*⟩ ::= `a` | `b` | `...` | `z` | `_`

⟨*digit*⟩ ::= `0` | `1` | `...` | `9`

⟨*id*⟩ ::= ⟨*alpha*⟩ `(` ⟨*alpha*⟩ | ⟨*digit*⟩ `)*`

⟨*array-decl*⟩ ::= ⟨*type*⟩ `(` `*` `)*` ⟨*id*⟩ `[` ⟨*integral-literal*⟩ `]`

⟨*var-decl*⟩ ::= ⟨*type*⟩ `(` `*` `)*` ⟨*id*⟩ `=` ⟨*logical-or-stmt*⟩ `;`

⟨*structure-var-decl*⟩ ::= ⟨*id*⟩ `(` `*` `)*` ⟨*id*⟩

⟨*decl*⟩ ::= ⟨*var-decl*⟩
| ⟨*array-decl*⟩
| ⟨*structure-var-decl*⟩

⟨*decl-without-initialiser*⟩ ::= ⟨*type*⟩ `(` `*` `)*` ⟨*id*⟩
| ⟨*array-decl*⟩
| ⟨*structure-var-decl*⟩

⟨*parameter-list*⟩ ::= ⟨*decl-without-initialiser*⟩ `,` ⟨*parameter-list*⟩
| ⟨*decl-without-initialiser*⟩

⟨*parameter-list-opt*⟩ ::= ⟨*parameter-list*⟩ | ϵ

⟨*stmt*⟩ ::= ⟨*block-stmt*⟩
| ⟨*selection-stmt*⟩
| ⟨*iteration-stmt*⟩
| ⟨*jump-stmt*⟩
| ⟨*decl*⟩
| ⟨*expr*⟩
| ⟨*assignment-stmt*⟩
| ⟨*primary-stmt*⟩

⟨*member-access-stmt*⟩ ::= ⟨*id*⟩ `.` ⟨*member-access-stmt*⟩
| ⟨*id*⟩ `.` ⟨*id*⟩

⟨*iteration-stmt*⟩ ::= ⟨*stmt*⟩
| `break;`
| `continue;`

⟨*block-stmt*⟩ ::= `{` ⟨*stmt*⟩`*` `}`

⟨*iteration-block-stmt*⟩ ::= `{` ⟨*iteration-stmt*⟩`*` `}`

⟨*selection-stmt*⟩ ::= `if (` ⟨*expr*⟩ `)` ⟨*block-stmt*⟩
| `if (` ⟨*expr*⟩ `)` ⟨*block-stmt*⟩ `else` ⟨*block-stmt*⟩

⟨*iteration-stmt*⟩ ::= `for (` ⟨*expr-opt*⟩ `;` ⟨*expr-opt*⟩ `;` ⟨*expr-opt*⟩ `)` ⟨*iteration-block-stmt*⟩
| `while (` ⟨*expr*⟩ `)` ⟨*iteration-block-stmt*⟩
| `do` ⟨*iteration-block-stmt*⟩ `while (` ⟨*expr*⟩ `) ;`

⟨*jump-stmt*⟩ ::= `return` ⟨*expr*⟩ `? ;`

⟨*assignment-op*⟩ ::= `=`
| `*=`
| `/=`
| `%=`
| `+=`
| `-=`
| `<<=`
| `>>=`
| `&=`
| `|=`
| `^=`

⟨*expr*⟩ ::= ⟨*assignment-stmt*⟩
| ⟨*var-decl*⟩

⟨*expr-opt*⟩ ::= ⟨*expr*⟩ | ϵ

⟨*assignment-stmt*⟩ ::= ⟨*logical-or-stmt*⟩
| ⟨*logical-or-stmt*⟩ ⟨*assignment-op*⟩ ⟨*assignment-stmt*⟩

⟨*logical-or-stmt*⟩ ::= ⟨*logical-and-stmt*⟩
| ⟨*logical-and-stmt*⟩ || ⟨*logical-or-stmt*⟩

⟨*logical-and-stmt*⟩ ::= ⟨*inclusive-or-stmt*⟩
| ⟨*inclusive-or-stmt*⟩ `&&` ⟨*logical-and-stmt*⟩

⟨*inclusive-or-stmt*⟩ ::= ⟨*exclusive-or-stmt*⟩
| ⟨*exclusive-or-stmt*⟩ | ⟨*inclusive-or-stmt*⟩

⟨*exclusive-or-stmt*⟩ ::= ⟨*and-stmt*⟩
| ⟨*and-stmt*⟩ `^` ⟨*exclusive-or-stmt*⟩

⟨*and-stmt*⟩ ::= ⟨*equality-stmt*⟩
| ⟨*equality-stmt*⟩ `&` ⟨*and-stmt*⟩

⟨*equality-stmt*⟩ ::= ⟨*relational-stmt*⟩
| ⟨*relational-stmt*⟩ `==` ⟨*equality-stmt*⟩
| ⟨*relational-stmt*⟩ `!=` ⟨*equality-stmt*⟩

⟨*relational-stmt*⟩ ::= ⟨*shift-stmt*⟩
| ⟨*shift-stmt*⟩ `>` ⟨*relational-stmt*⟩

```
                    | ⟨shift-stmt⟩ < ⟨relational-stmt⟩
                    | ⟨shift-stmt⟩ >= ⟨relational-stmt⟩
                    | ⟨shift-stmt⟩ <= ⟨relational-stmt⟩

⟨shift-stmt⟩        ::= ⟨additive-stmt⟩
                    | ⟨additive-stmt⟩ << ⟨shift-stmt⟩
                    | ⟨additive-stmt⟩ >> ⟨shift-stmt⟩

⟨additive-stmt⟩     ::= ⟨multiplicative-stmt⟩
                    | ⟨multiplicative-stmt⟩ + ⟨additive-stmt⟩
                    | ⟨multiplicative-stmt⟩ - ⟨additive-stmt⟩

⟨multiplicative-stmt⟩ ::= ⟨prefix-unary-stmt⟩
                    | ⟨prefix-unary-stmt⟩ * ⟨multiplicative-stmt⟩
                    | ⟨prefix-unary-stmt⟩ / ⟨multiplicative-stmt⟩
                    | ⟨prefix-unary-stmt⟩ % ⟨multiplicative-stmt⟩

⟨prefix-unary-stmt⟩ ::= ⟨postfix-unary-stmt⟩
                    | ++ ⟨postfix-unary-stmt⟩
                    | -- ⟨postfix-unary-stmt⟩
                    | * ⟨postfix-unary-stmt⟩
                    | & ⟨postfix-unary-stmt⟩
                    | ! ⟨postfix-unary-stmt⟩

⟨postfix-unary-stmt⟩ ::= ⟨primary-stmt⟩
                    | ⟨primary-stmt⟩ ++
                    | ⟨primary-stmt⟩ --

⟨primary-stmt⟩      ::= ⟨constant⟩
                    | ⟨symbol-stmt⟩
                    | ( ⟨logical-or-stmt⟩ )

⟨symbol-stmt⟩       ::= ⟨function-call-stmt⟩
                    | ⟨array-access-stmt⟩
                    | ⟨member-access-stmt⟩
                    | ⟨id⟩

⟨array-access-stmt⟩ ::= ⟨id⟩ ( [ ⟨expr⟩ ] )*

⟨function-call-arg-list⟩ ::= ⟨logical-or-stmt⟩ , ⟨function-call-arg-list⟩
                    | ⟨logical-or-stmt⟩

⟨function-call-arg-list-opt⟩ ::= ⟨function-call-arg-list⟩ | ϵ

⟨function-call-expr⟩ ::= ⟨id⟩ ( ⟨function-call-arg-list-opt⟩ )
```