Weak language draft

epoll-reactor

since December 2021

Contents

1 Scope

This document describes requirements for implementation of weak programming language.

2 Lexical elements

2.1 Keywords

boolean	**break**	**char**
continue	**do**	**false**
float	**for**	**if**
int	**return**	**string**
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.

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

3 Grammar summary

⟨*program*⟩ ::= ⟨*function-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*⟩
	|	⟨*boolean-literal*⟩
⟨*integral-literal*⟩	::=	⟨*digit*⟩ *
⟨*floating-literal*⟩	::=	⟨*digit*⟩ * **.** ⟨*digit*⟩ *
⟨*string-literal*⟩	::=	¨ *(* `\x00000000-\x0010FFFF` *)** ¨
⟨*boolean-literal*⟩	::=	***true***
	|	***false***
⟨*alpha*⟩	::=	***a*** | ***b*** | ... | ***z*** | _
⟨*digit*⟩	::=	***0*** | ***1*** | ... | ***9***
⟨*id*⟩	::=	⟨*alpha*⟩ *(* ⟨*alpha*⟩ | ⟨*digit*⟩ *)**
⟨*array-decl*⟩	::=	⟨*type*⟩ ⟨*id*⟩ *[* ⟨*digit*⟩ * *]*
⟨*var-decl*⟩	::=	⟨*type*⟩ ⟨*id*⟩ `=` ⟨*logical-or-expr*⟩
⟨*var-decl-without-initialiser*⟩	`::=`	⟨*type*⟩ ⟨*id*⟩
⟨*parameter*⟩	`::=`	⟨*var-decl-without-initialiser*⟩
	|	⟨*array-decl*⟩
⟨*parameter-list*⟩	`::=`	⟨*parameter*⟩ **,** ⟨*parameter-list*⟩
	|	⟨*parameter*⟩
⟨*parameter-list-opt*⟩	`::=`	⟨*parameter-list*⟩ | ϵ
⟨*stmt*⟩	`::=`	⟨*selection-stmt*⟩
	|	⟨*array-access-stmt*⟩
	|	⟨*iteration-stmt*⟩
	|	⟨*jump-stmt*⟩
	|	⟨*var-decl*⟩
	|	⟨*expr*⟩
	|	⟨*unary-expr*⟩
⟨*array-access-stmt*⟩	`::=`	⟨*id*⟩ `[` ⟨*logical-or-expr*⟩ `]`

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

⟨selection-stmt⟩       ::= if ( ⟨expr⟩ ) { ⟨stmt⟩* }
                         | if ( ⟨expr⟩ ) { ⟨stmt⟩* } else { ⟨stmt⟩* }

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

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

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

⟨expr⟩                 ::= ⟨assignment-expr⟩

⟨expr-opt⟩             ::= ⟨expr⟩ | ϵ

⟨assignment-expr⟩      ::= ⟨logical-or-expr⟩
                         | ⟨unary-expr⟩ ⟨assignment-op⟩ ⟨assignment-expr⟩

⟨logical-or-expr⟩      ::= ⟨logical-and-expr⟩
                         | ⟨logical-or-expr⟩ || ⟨logical-and-expr⟩

⟨logical-and-expr⟩     ::= ⟨inclusive-or-expr⟩
                         | ⟨logical-and-expr⟩ && ⟨inclusive-or-expr⟩

⟨inclusive-or-expr⟩    ::= ⟨exclusive-or-expr⟩
                         | ⟨inclusive-or-expr⟩ | ⟨exclusive-or-expr⟩

⟨exclusive-or-expr⟩    ::= ⟨and-expr⟩
                         | ⟨exclusive-or-expr⟩ ^ ⟨and-expr⟩

⟨and-expr⟩             ::= ⟨equality-expr⟩
                         | ⟨and-expr⟩ & ⟨equality-expr⟩
```

```
⟨equality-expr⟩          ::= ⟨relational-expr⟩
                           | ⟨equality-expr⟩ == ⟨relational-expr⟩
                           | ⟨equality-expr⟩ != ⟨relational-expr⟩

⟨relational-expr⟩        ::= ⟨shift-expr⟩
                           | ⟨relational-expr⟩ > ⟨shift-expr⟩
                           | ⟨relational-expr⟩ < ⟨shift-expr⟩
                           | ⟨relational-expr⟩ >= ⟨shift-expr⟩
                           | ⟨relational-expr⟩ <= ⟨shift-expr⟩

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

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

⟨multiplicative-expr⟩    ::= ⟨unary-expr⟩
                           | ⟨multiplicative-expr⟩ * ⟨unary-expr⟩
                           | ⟨multiplicative-expr⟩ / ⟨unary-expr⟩
                           | ⟨multiplicative-expr⟩ % ⟨unary-expr⟩

⟨unary-expr⟩             ::= ⟨postfix-expr⟩
                           | ++ ⟨unary-expr⟩
                           | -- ⟨unary-expr⟩

⟨postfix-expr⟩           ::= ⟨primary-expr⟩
                           | ⟨postfix-expr⟩ [ ⟨expr⟩ ]
                           | ⟨postfix-expr⟩ ++
                           | ⟨postfix-expr⟩ --

⟨primary-expr⟩           ::= ⟨constant⟩
                           | ⟨id⟩
                           | ( ⟨expr⟩ )
```

4 Environment

4.1 Backend

The language use the LLVM backend, although another backend can be implemented (including self-written one).

4.2 Data types

The language must implement static strong typing. All casts must be explicit.

- **Int** – Signed 32-bit;
- **Float** – Signed 32-bit;
- **Bool** – 8-bit;
- **String** – Character sequence, that ends with Null character;
- **Void** – Empty type, used as return type only.

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

bool array[10];

.

4.3 Inside-iteration statements

- **Break** – Usable only inside the **while**, **do-while** and **for** statements. Performs exit from a loop.
- **Continue** – Usable only inside the **while**, **do-while** and **for** statements. Performs jump to the next iteration.

4.4 Iteration statements

- **While** – Loop statement that performs its body until the condition evaluates to true.
- **Do-While** – Loop statement with similar to **While** semantics, but it executes body before contition check at first time.
- **For** – Loop statement with three initial parts and body. This includes:
  - **Initial** part with the variable assignment;
  - **Conditional** part with the some condition;
  - **Incremental** part with the some statement, that should change assigned variable.

  All parts are optional.

4.5 Conditional statements

- **If** – Conditional statement, that should execute If-part when it's condition evaluates to true. Otherwise, Else-part should be executed.

4.6 Jump statements

- **Return** – The end point of control flow, may return value, may not (void functions).

5 Semantics

5.1 Types

- Integer (boolean, integer, floating point) types are simple numeric types, which can be copied in trivial way (with memcpy and so on).
- String type initially is a pointer to string literal. However, once contents under this pointer are "modified", copy of literal created and emplaced onto stack. After that, all operations on string variable affecting local copy.

5.2 Function parameters

- All types including arrays are copied to function parameters during call.

6 FFI

6.1 Linking with C

The language have FFI with the GNU C Library and with other C libraries in general. This mean, that **cdecl** call convention is used.