SecLang - SECCON 2022 Finals

Introduction

SecLang is an educational programming language. SecLang provides both an interpreter and a compiler.

Overview

SecLang is a dynamically typed language. The type of a variable is associated with values at runtime. SecLang supports 5 native types (bool, byte, uint, int, func) and array of one of the types.

```
func main() {
  s = 123;
  print(s);
  s = “ Hello, World!\n”;
  print(s);
}
```

Types and Casts

Types

SecLang support the following 5 primitive types.

Type	Meaning
bool	Boolean (true or false) value
byte	Byte (8-bit) value
uint	64-bit unsigned integer
int	64-bit signed integer
func	Reference to a function

Structs, classes, or any other ways of defining a new type are not supported.

Array

In addition to the above primitive types, SecLang supports arrays. Each array has a fixed size, which can be determined at compile-time. You can initialize arrays using one of the following notations:

```
arr = [123, 234, 345];
arr = [0; 10];
```

The first example creates an array of 3 elements (123, 234, and 345). The second example creates an array of 10 elements, all of which are 0.
Arrays can only have primitive values, which means that you cannot create multidimensional arrays. Also, an array cannot contain values of different types.
Note that a string is actually an array of byte values internally.

Casts

SecLang is a dynamically typed language. The type of each variable is defined at runtime and can change during execution. You can use the ‘as’ keyword to explicitly specify (or cast a value to) a type.

```
X = -1            → x: -1
y = x as uint     → y: 0xffffffffffffffff
z = x as byte     → z: ‘\xff’
```

The use of operators between different types is usually undefined in SecLang, except for implicit type conversion, which is explained later. See the behavior of the SecLang interpreter for more information on how casts between different types are defined.

Implicit Type Conversion

The type usually does not change unless you explicitly cast it. However, there are some cases where implicit casting occurs.

The first and most obvious implicit cast is integer. Every integer is typed as a signed “int” in SecLang.

```
x = 123;
is equivalent to
x = 123 as int;
```

The second implicit casting occurs in conditional statements. When a value is used for the condition of a “while” or “if” statement, it is implicitly cast to boolean.

```
if a && b {...}
is equivalent to
if (a as bool) && (b as bool) {...}
```

The third type of casting occurs when using operators. Arithmetic and logical operations between different types usually cause undefined behavior. An exception is logical or arithmetic shift operations. (Shift is logical if the value is unsigned, otherwise it is arithmetic.)
The count of shift is automatically converted to unsigned int if it is signed int. Any type other than int or uint cannot be used as a shift count.

```
c = 8 as int;
x = 0xdeadbeef as uint;
x = x >> c;
is equivalent to
c = 8 as int;
x = 0xdeadbeef as uint;
x = x >> (c as uint)
```

Control Flow

The flow of an application can be conditionally controlled with “if” and “while” statements:

```
if x < 0 {
   print(“x is negative\n”);
} else {
   if x {
     print(“x is positive\n”);
   } else {
     print(“x is zero\n”);
   }
}

i = 0;
while f(i) {
  i = i + 1;
  if i > 100 {
    break;
  }
}
```

SecLang provides the same operators as C to test variables for equality or compare them:

- “==” and “!=” for testing equality and inequality
- “<”, “<=”, “>” and “>=” for testing less than (or equal to) and greater than (or equal to)

For combining multiple conditions, the “||” operator represents the logical *OR*, and “&&” the logical *AND*.

Variable Definition and Assignments

A variable is “defined” when it’s first assigned in the function. You can define variables anywhere within the function and outside the block. In other words, **you cannot define variables in “while” or “if” statements**.
Note that the following code is not valid in SecLang.

```
func f() {
  ...
  if x == 1 {
    y = true; → Cannot define ‘y’ in “if” statement
  } else {
    y = false; → Same
  }

  i = 0;
  while i < 10 {
    v = i;      → Cannot define ‘v’ in “while” statement
    i = i + 1;
  }
  ...
}
```

You should write it like this:

```
func f() {
  …
  y = x == 1;

  i = 0;
  v = 0;
  while i < 10 {
    v = i;
    i = i + 1;
  }
  ...
}
```

Builtin Functions

SecLang has only 3 builtin functions.

scan

[Syntax]
scan() → int

[Description]
This function reads a value from standard input, parses it as 64-bit signed integer, and returns the value as ‘int’.

print

[Syntax]
print(bool | byte | uint | int | func | array)

[Description]
This function takes an argument and prints the value to standard output. See the behavior of the SecLang interpreter or machine code of the SecLang compiler for more information on how this function handles the given value.

exit

[Syntax]
exit(int)

[Description]
This function exits the program with the value of the first argument set to status code.

About Undefined Behaviors

SecLang has a lot of undefined behavior. Any code that throws an exception on the SecLang interpreter is an undefined behavior. The SecLang compiler is still under development and cannot fully detect undefined behavior at runtime.

However, our compiler is already complete enough to pass all the testcases.

SecLang Grammar in Extended BNF

The following production rules in extended BNF fully describe the grammar of the SecLang programming language.

```
<prog> ::= { <func_decl> }

<func_decl>  ::=  ‘func’ <func_name> ‘(’ [ <param_list> ] ‘)’ <stmt>
<fucn_name>  ::= <ident>
<param_list> ::= [ <ident> ] | <ident> ‘,’ <param_list>

<stmt>        ::= ‘{’ <stmt_list> ‘}’ | <common_stmt>
<stmt_list>   ::= { <stmt> }
<common_stmt> ::= <expr_stmt> | <assign_stmt> | <if_stmt> | <while_stmt>
                | <break_stmt> | <return_stmt>
<expr_stmt>   ::= <expr> ‘;’
<assign_stmt> ::= <target> ‘=’ <expr> ‘;’
<target>      ::= <ident> | <index_expr_ref>

<if_stmt>     ::= ‘if’ <expr> <stmt> [ ‘else’ <stmt> ]
<while_stmt>  ::= ‘while’ <expr> <stmt>
<break_stmt>  ::= ‘break’ ‘;’
<return_stmt> ::= ‘return’ [ <expr> ] ‘;’

<expr>          ::= <or_or_expr>
<or_or_expr>    ::= <and_and_expr> | <or_or_expr> ‘||’ <and_and_expr>
<and_and_expr> ::= <or_expr> | <and_and_expr> ‘&&’ <or_expr>
<or_expr>       ::= <xor_expr> | <or_expr> ‘|’ <xor_expr>
<xor_expr>      ::= <and_expr> | <xor_expr> ‘^’ <and_expr>
<and_expr>      ::= <cmp_expr> | <and_expr> ‘&’ <cmp_expr>
<cmp_expr>      ::= <shift_expr> ‘==’ <shift_expr>
                  | <shift_expr> ‘!=’ <shift_expr>
                  | <shift_expr> ‘>=’ <shift_expr>
                  | <shift_expr> ‘<=’ <shift_expr>
                  | <shift_expr> ‘>’ <shift_expr> | <shift_expr> ‘<’ <shift_expr>
                  | <shift_expr>
<shift_expr>    ::= <shift_expr> ‘<<’ <add_expr> | <shift_expr> ‘>>’ <add_expr>
                  | <add_expr>
<add_expr>      ::= <add_expr> ‘+’ <mul_expr> | <add_expr> ‘-’ <mul_expr>
                  | <mul_expr>
<mul_expr>      ::= <unary_expr> | <mul_expr> ‘*’ <unary_expr>
                  | <mul_expr> ‘/’ <unary_expr> | <mul_expr> ‘%’ <unary_expr>
<unary_expr>    ::= ‘!’ <unary_expr> | ‘+’ <unary_expr> | ‘-’ <unary_expr>
                  | <cast_expr> | <index_expr> | <call_expr>
```

```
<cast_expr>     ::= <unary_expr> ‘as’ <type>
<index_expr>    ::= <unary_expr> ‘[’ <expr> ‘]’
<index_expr_ref> ::= <unary_expr> ‘[‘ <expr> ‘]’
<call_expr>     ::= <call_expr> ‘(’ [ <argument> ] ‘)’ | <primary_expr>
<argument>        ::= <expr> | <expr> ‘,’ <argument>
<primary_expr> ::= <ident> | <number_literal> | <string_literal>
                 | <boolean_literal> | <array_literal> | ‘(’ <expr> ‘)’

<ident> ::= (‘_’ | <LETTER>) { (‘_’ | <LETTER> | <DIGIT>) }
<number_literal>  ::= <DEC> | <HEX>
<string_literal>  ::= ‘”’ <STRING> ‘”’
<boolean_literal> ::= ‘true’ | ‘false’
<array_literal>   ::= ‘[’ <array_literal_values> ‘]’
<array_literal_values> ::= [ <expr> ] | <expr> ‘,’ <array_literal_values>

<type> ::= ‘bool’ | ‘byte’ | ‘uint’ | ‘int’

<HEX> ::= [ ‘+’ | ‘-’ ] ‘0x’ <HEXDIGIT>+
<DEC> ::= <DIGIT>+
<LETTER>   ::= ‘a’ | ’b’ | … | ’z’ | ’A’ | ’B’ | … | ’Z’
<DIGIT>    ::= ‘0’ | ’1’ | ’2’ | ‘3’ | ’4’ | ’5’ | ‘6’ | ’7’ | ’8’ | ‘9’
<HEXDIGIT> ::= <DIGIT> | ‘a’ | ‘b’ | … | ‘f’ | ‘A’ | ‘B’ | … | ‘F’
```