The Extended-Kaleidoscope Language Specification

Version 0.3.1 - 2019-04-06
Hal Finkel and Kavon Farvardin

General Notes

- This language is an extended version of the Kaleidoscope language from LLVM's tutorial: https://llvm.org/docs/tutorial/
- Changes in version 0.3
    - No space is allowed between $ and <ident> in a variable identifier since it's silly.
    - Dropped the sfloat type for now, because we haven't been using it.
    - No more implicit booleans, i.e., we have a strongly typed bool now.
    - Clarified that || and && do not short circuit by referring to them as "bitwise".
    - Added an explicit cast operator and forbid all implicit casts.
- Changes in version 0.2.1
    - Updated the grammar for string literals. They now exclude newlines and carriage returns.
    - Clarified that local variables cannot have the same name.
    - Clarified that function parameters are to be treated as if they are local variable declarations, and that all local variables must be declared before being used.
    - Clarified case sensitivity.
    - Clarified the type of comparison expressions.
    - Added typing rules for numeric literals.
    - Added more examples of errors & non-errors.

Grammar

- Whitespace (space, newline, etc) is allowed between any two tokens in the grammar below unless otherwise noted.
- Comments begin with the "#" character and continue until the end of the line.
- All tokens (keywords, identifiers, etc) are case sensitive.

- Please keep in mind that we use regular-expression operators in this grammar. For example, `<token>*` means *zero or more appearances of <token>*, and the same goes for **+** meaning *one or more* and **?** meaning *once or none*.

```
<prog>      ::= <extern>* <func>+

<extern>    ::=  extern <type> <globid> "(" <tdecls>? ")" ";"

<func>      ::= def <type> <globid> "(" <vdecls>? ")" <blk>
<blk>       ::= "{" <stmts>? "}"
<stmts>     ::= <stmt>+

<stmt>      ::= <blk>
              | return <exp>? ";"
              | <vdecl> "=" <exp> ";"
              | <exp> ";"
              | while "(" <exp> ")" <stmt>
              | if "(" <exp> ")" <stmt> (else <stmt>)?
              | print <exp> ";"
              | print <slit> ";"

<exps>      ::= <exp> | <exp> "," <exps>
<exp>       ::= "(" <exp> ")"
              | <binop>
              | <uop>
              | <lit>
              | <varid>
              | <globid> "(" <exps>? ")"

<binop>     ::= <arith-ops>
              | <logic-ops>
              | <varid> = <exp>         # assignment
              | "[" <type> "]" <exp>    # explicit type-cast

<arith-ops> ::= <exp> * <exp>
              | <exp> / <exp>
              | <exp> + <exp>
              | <exp> - <exp>
```

```
<logic-ops> ::= <exp> == <exp>    # equality
              | <exp> < <exp>
              | <exp> > <exp>
              | <exp> && <exp>    # bitwise AND only for bools
              | <exp> || <exp>    # bitwise OR only for bools



<uop>       ::= ! <exp>          # bitwise negation on bools
              | - <exp>          # signed negation


<lit>     ::= true
            | false
            | [0-9]+(\.[0-9]+)?

<slit>    ::= "[^"\n\r]*"

<ident>   ::= [a-zA-Z_]+[a-zA-Z0-9_]*
<varid>   ::= $<ident>          # no space between $ and <ident>.
<globid>  ::= <ident>
<type>    ::= int | cint | float | bool | void | (noalias)? ref <type>
<vdecls>  ::= <vdecl> | <vdecl> "," <vdecls>
<tdecls>  ::= <type> | <type> "," <tdecls>
<vdecl>   ::= <type> <varid>
```

Typing Rules and Constraints *(Informally)*

The following table illustrates the correspondence between EK-language types and LLVM types.

EK Type	**Description**	**LLVM Type**
`int`	32-bit signed integer	`i32`
`cint`	32-bit signed integer with checked overflow.	`i32`
`float`	IEEE-754 single-precision floating-point number.	`float`

`bool`	a two-valued type	`i1`
`void`	the absence of a value	`void`
`ref T`	a reference to a value of type T	`T *`

1. In `<vdecl>, the type may not be void.`

2. In `ref <type>`, the type may not be void or itself a reference type.

3. All functions must be declared and/or defined before they are used.

4. Functions (whether extern or not) may not share the same name.

5. All local variables must be declared before they are used.

6. The scope of a local variable declaration extends from that declaring statement until the end of the block in which that statement resides. Any inner blocks within this region also have the local variable in-scope.

7. Local variables and parameters that have overlapping scopes may not have the same name. For example,

```
int $x = 0;
int $x = 1;  # error, redefinition of '$x'

{
  int $y = 0;
}
int $y = 1;  # ok, since earlier definition of '$y' is out of scope
here
```

8. A function may not return a ref type.

9. `print` prints to stdout followed by a new line.

10. Values of reference type are bound to (i.e., point to) their initialization's right-hand-side expression (or, for function arguments that have ref type, the provided function argument), which must be a variable itself. For example,

```
int $y = 0;
int $w = 1;

ref int $x = $y;  # $x is bound to $y.
ref int $z = $x;  # $z is also bound to $y.
ref int $a = 11;  # illegal, RHS must be a var.

def void foo (ref int $f, int $g) { ... }

foo ($w, $y);  # $w is passed by reference, and $y is passed by value.
```

11. Uses of the reference variable evaluate to the then-current value of the bound variable. Assignments to the reference variable set the value of the bound variable.

12. If the types of a binary operator don't match, then the expression contains a type error. An explicit cast must be present in the program to convert one of the arguments so it matches the other. Thus, no implicit casts are performed by the compiler. The following explicit casts are the *only legal casts*,

Original value's current type	**Can be casted to**
`int`	`int, cint, float`
`cint`	`int, cint, float`
`float`	<u>`int`</u>, <u>`cint`</u>, `float`
`bool`	`bool`

If the resulting type is underlined in the table above, then the cast is not always perfectly reversible, i.e., information can be lost when casting from a float to an int.

13. Integer literals are of type **int**, and number literals that contain a fractional component are type **float**. The literals 'true' and 'false' are **bool**.

14. The following table lists the types allowed for various program constructs. As a short-hand, we use N to represent the set of numeric types: int, cint, float.

Construct	**Input Type(s)**	**Output Type**
arith-ops (see grammar)	Two values of the same type from N.	The same type as its input.
`>, <`	Two values of the same type from N.	`bool`
`==`	Two values of the same type from N. Or, two bools.	`bool`
&& and `||`	Two bools.	`bool`
`!`	One bool.	`bool`
unary negation	One value from N.	The same type as was input.
`if` and `while`	One bool	statements do not have resulting values
`<varid> = <exp>`	Some value of type T, and a variable of type T.	T
`[ <type> ] <exp>`	Some expression of type T, and a type V	V

Operational Semantics *(Informally)*

- The integer types are signed and:
    a. For **int**, the behavior of the program is undefined if the value overflows.
    b. For **cint**, if the value overflows the program must print an error message to stderr and exit (the exit status of the program must indicate failure).
- Arguments to a function are evaluated left-to-right. The value of an assignment is its left-hand side after performing the assignment.
- All programs must define exactly one function named “run” which returns an int (the program exit status) and takes no arguments. This is the program entry point.
- If a reference variable, r, is declared noalias, then the programmer is promising that,

within the scope of the reference variable, the bound variable, b, is accessed only through r or a reference derived from r. When a reference variable is bound using a reference variable on the right-hand side, both refer to the same underlying variable (and this reference variable is considered to be derived from the one of the right-hand side).

- If control-flow reaches the end of the function without returning, and the function has a void return type, then a void return is implicitly assumed.
- Associativity and precedence, from highest to lowest:
    a. Right-to-left: - (unary) ! (logical not), type casts
    b. Left-to-right: * /
    c. Left-to-right: + - (binary)
    d. Left-to-right: < >
    e. Left-to-right: ==
    f. Left-to-right: &&
    g. Left-to-right: ||
    h. Right-to-left: =
- A function's parameters are treated the same as local variables declared at the start of the function's body. Their values are initialized to the corresponding values passed by the caller of the function.

Built-in External Functions

- Get the specified command-line argument as an integer: `extern int arg(int);`
- Get the specified command-line argument as a float: `extern float argf(int);`

Examples

```
def int fib (int $n) {
    if ($n < 2)
        if ($n == 0)
            return 0;
        else
            return 1;

    int $a = fib ($n - 1);
    int $b = fib ($n - 2);
    return $a + $b;
}

def void inc (ref int $n) {
  $n = $n + 1;
}

def int run () {
    print "fib(5):";
    int $val = fib(5);
    print $val;

    print "fib(5)+1:";
    inc($val);
    print $val;

    return 0;

}
```

Examples of errors and non-errors

```
# valid:
int $x = 1;
$x = 2;
$x = 3;


#invalid due to redeclarations with same name in same scope:
int $x = 1;
int $x = 2;
int $x = 3;
```

```
int $x = 1.1;  #invalid, must explicitly cast right-hand side to int
float $y = 1;  #invalid, must explicitly cast right-hand side to float
```

```
# invalid, because 2 is treated as an int and is missing a cast to cint
def cint add2 (cint $x) {
  cint $rv = $x + 2;
  return $rv;
}

# valid, since the literal 2 is explicitly cast to cint
def cint add2 (cint $x) {
  cint $rv = $x + [cint] 2;
  return $rv;
}
```