Middle East Technical University **Department of Computer Engineering**

CENG444 - Language Processors

2024-2025 Spring
Project IV
Simple Optimizations, Linear IR, and Code Generation

Problem

In this assignment, you are required to write a C++ program that generates linear IR after performing simple optimizations. Your implementation will be based on your Project III implementation. There is no change in the language specification.

Constant folding

Your solution will have the capacity to reduce the code requirement to evaluate the expressions based on the roles of constants found in an expression. The result that is going to be evaluated by an operator can be evaluated ahead of time by using constant folding. Below is the list of the operators and constant dependent contexts where constant folding can be applied and how.

Ternary Operator

```
true?<expr1>:<expr2> => <expr1>
false?<expr1>:<expr2> => <expr2>
```

Comparison Operators

```
const <comp-op> const => either true or false
```

Boolean and Operator

```
<expr> && true => <expr>
true && <expr> => <expr>
<expr> && false=> false
false && <expr> => false
```

Boolean or Operator

```
<expr> || false => <expr>
false || <expr> => <expr>
<expr> || true => true
true || <expr> => true
```

Addition / concatenation operator

```
<const> + <const> => <const>
<expr> + 0 => <expr>
0 + <expr> => <expr>
```

```
<expr> + "" => <expr>
"" + <expr> => <expr>
```

Subtraction Operator

```
<const> - <const> => <const>
<expr> - 0 => <expr>
0 - <expr> => -<expr>
```

Multiplication Operator

```
<const> * <const> => <const>
<expr> * 0 => 0
0 * <expr> => 0
```

Division Operator

```
<const> / 0 => 3 with and error message. 3 is an ordinary value
that does not trigger further constant folding.
<const> / <const> => <const>
0 / <expr> => 0
<expr> / 1 => <expr>
```

Unary Minus Operator

```
- number => -number
```

The Not Operator

```
! const => !const
```

Dead code elimination

Ineffective Statements (-p1 / OPTIMIZE_DC_STATEMENT)

A statement will be considered as effective when any combinations of the following conditions are met:

- The statement is predecessor of another statement.
- The statement has a function call.
- The statement has an assignment.

There is no need to execute an ineffective statement. Even when a statement is found as ineffective, the graphical IR will contain fully analyzed statement and expression nodes. However, the IC will not contain any code from the statements that are found to be eliminated. See Appendix 4: Code Samples to Demonstrate Optimizations.

Ineffective Expression Parts (-p2 / OPTIMIZE_DC_EXPPART)

Another form of a dead code elimination may occur where an expression with comma operator(s) found out of contexts for array literals and calls.[1] In such cases, an expression part is said to be effective when any combinations of the following conditions are met:

[1] Keep in mind that array building literals are also translated as compiler runtime calls, which correspond to implicit call nodes.

- The expression part has a call.
- The expression part has an assignment.

Similarly, there is no need to execute an ineffective expression part. Even when an expression part is found as ineffective, the graphical IR will contain fully analyzed expression nodes. However, the IC will not contain any code from the ineffective expression parts.

See Appendix 4: Code Samples to Demonstrate Optimizations.

IC Generation

Your program will generate intermediate code and present it both in a JSON file, which becomes the inspection tool for the intermediate representation, and in a human readable text file.[2] An instruction will have a sequence number, op-code, a simple type-tag, and an optional parameter. See Appendix 2: Node Op-Codes and IC Instruction Set for a concise description of the instruction set.

The linear IC that will be generated for a stack machine that will be discussed in the lab hours with help of example code fragments.

Most of the instructions correspond to the operators defined in the supplemental code. A few of the expression tree patterns are translated compiler runtime calls. These are

- Array builder from array literals.
- Array push operations that appear as OP_ADD with an array on LHS.
- Comparison op-codes with array parameters.
- String concatenations that appear as OP_ADD with array operands.
- Comparison op-codes with string parameters.
- Number to string comparisons that appear as OP_ADD with a string one side and a number on the other.

There is a small set of IC instructions that neither appear as direct op-codes in expression nodes nor translated to compiler runtime calls. These are the instructions to perform jumps and stack manipulation.

Peephole optimization

The peephole optimization that will be applied by your solution will have a three-instruction window just to identify two kinds of ineffective code.

Ineffective Store Load (-p4 / OPTIMIZE_PH_OFFLOAD)

A pop operation may occur right after an assignment. In some circumstances, the popped value may be re-invoked on TOS with an “insid” op-code. In such cases the “pop” and the “insid” instructions can be eliminated. See Appendix 4: Code Samples to Demonstrate Optimizations.

Constant Value Sink (-p8 / OPTIMIZE_PH_CONSTSINK)

A pop operation may occur right after pushing a constant. In such cases both instructions can be eliminated. See Appendix 4: Code Samples to Demonstrate Optimizations.

[2] Appendix 4: Code Samples to Demonstrate Optimizations contain plenty of instruction lines to give you an idea on how the human readable text file looks like.

X64 Code Generation (Bonus)

Your solution will translate the intermediate code into x64 instructions into an executable memory block and execute it. The code templates and methodologic guidance will be provided in the lab hours with additional code support.

Your Assignment

You will develop a C++ program that accepts one or two command line arguments. When run with single argument, your program will treat the parameter as the file name. When run with two parameters, the first parameter will be in form of -p<number> where the number is the bitwise encoded optimization flag. In this case, the second parameter will become the file name.

The file name parameter will be used to generate the full name of the input file and the output files. The input file will be <file name>.txt, the first output file will be <file name>.json, and the second output file will be <file name>-IC.txt. The first output file having the JSON format will be similar to the output of the Project III implementation. The second output file will be a text file that contains easy to read IC listing.

When specified the bitwise specified optimization flag will tell your program the optimizations that will be applied while generating the IC.

Argument	**Macro in dgevalsup.h**	**Meaning**
-p1	`OPTIMIZE_DC_STATEMENT`	The processor will apply dead code elimination on ineffective statements.
-p2	`OPTIMIZE_DC_EXPPART`	The processor will apply dead code elimination on ineffective expression parts.
-p4	`OPTIMIZE_PH_OFFLOAD`	The processor will apply store / load elimination on the IC.
-p8	`OPTIMIZE_PH_CONSTSINK`	The processor will apply constant sink elimination on the IC.

Note that the bitwise encoded arguments must be combined to activate more than one optimization. Specifying -p15 will activate all of the optimizations. Specifying -p0 will disable all of the optimizations. Your program will apply all kind of optimizations when run in single parameter configuration.

Example run:

When your program is compiled and run as "`project4 test1`", your program will read the input file "`test1.txt`", and generate the outputs as "`test1.json`" and "`test1-IC.txt`". In case your program fails to open the input file, your program will print "File not found!" message and terminate.

Processor Output

The JSON file will meet all of the requirements of the Project III. The followings define additional or modified entities that will be found:

- Each reported statement must have a count describing how many times the statement found as a predecessor of another statement.

- Each reported statement must have a count describing the number of the functions calls in its expression.
- Each reported statement must have a count describing the number of the assignments found in its expression.
- There will be a root level array containing the IC instructions. Instruction reporting format will be supplied. There will be no instruction coming from an eliminated statement.
- There will be mutations and transformations on the expression trees as result of processing. The final version of the expression trees must be observed in the JSON file.

The output text file will contain instruction lines. Each instruction line will contain sequence number, instruction mnemonic, type and parameter information as appropriate. The example format for an instruction line will be supplied.

Finally, the processor must report to the console the messages placed in the JSON file in a human readable format. This report must be generated before calling the dynamically generated x64 code if bonus part was implemented.

Bonus Part Output

Those who intends two receive bonus points must note in the README.txt file their intention. For the submissions with bonus part, the program is expected to run the dynamically generated code after generating the standard processor outputs. Note that execution of dynamically generated code is requested for error free DGEval code only. It is student's responsibility to implement the DGEval runtime library functions (see Appendix 1: DGEval Runtime Library). Further guidance will be provided in implementation of these functions during the lab hours.

The evaluation process will use the print function to observe correct execution. The README.txt file must provide the evaluator where to set a breakpoint and the variable that points to the memory block that contains the dynamically generated x64 code.

Regulations & Hints

- **Implementation:** You should use Flex, Bison with C++ to develop your program. Make sure that your program will accept the name of the input file. In case the program is run without a parameter or with more than one parameter your program must terminate immediately with appropriate error message sent to the standard output.
- **Supplemental Material:** You will be provided with a supplemental code and an example set for support and standardization. The code will cover
    - A standardized grammar for DGEval
    - A small library for fundamental data structures, runtime functions, and the language runtime.

    Note that you may be asked to develop your own extensions to any of these components.
- **Evaluation:** The evaluation will be based on
    - Presence and correctness of the symbol table in the JSON file when the input is free of syntactic problems.
    - Presence and correctness of the statement lists in the JSON file for both circular and drafted ones when the input is free of syntactic problems.

- Presence and correctness of the expression trees in the JSON file for each statement when the input is free of syntactic problems.
- Presence and the correctness of the linear IR both in the JSON file and in the more human readable text file.
- Factual presence of the messages in the JSON file due to syntactic and semantic checks with line number and descriptions.

Keep in mind that correctness checks will also require the coherence between the IC generated and the optimization flags specified.

- **Submission:** You need to submit all relevant files you have implemented (`.cpp, .h, .l, .y`, etc.) as well as a `README` file with instructions on how to build and run, in a single `.zip` file named `<your studentID>`.

Following notes are for more clarity to address possible concerns and/or questions:

- Use Flex and Bison to generate C++ not C file. C implementation will not be accepted.
- Never modify the automatically generated files generated by Flex and Bison. You may prefer consulting the recitation material and the sample project on semantic analysis.
- Describe precisely the steps to build and run your program in `README.txt` file, which is essential part of your submission. Provide any configuration items such as scripts, configuration files that will be necessary.

Appendix 1: DGEval Runtime Library

DGEval runtime library includes a limited set of functions that can be accessed by using function call operator.

Index	Prototype	Description
0	number stddev(array)	Calculates standard deviation of the values in an array of numbers.
1	number mean(array)	Calculates the mean of the values in an array of numbers.
2	number count(array)	Calculates the count of the values in an array.
3	number min(array)	Calculates the minimum value found in an array.
4	number max(array)	Calculates the minimum value found in an array.
5	number sin(number)	Calculates the sine of a given number.
6	number cos(number)	Calculates the cosine of a given number.
7	number tan(number)	Calculates the tangent of a given number.
8	number pi()	Returns pi.
9	number atan(number)	Calculates the invers tangent of a given number.
10	number asin(number)	Calculates the inverse sine of a given number.
11	number acos(number)	Calculates the inverse cosine of a given number.
12	number exp(number)	Evaluates e powered to the given parameter.
13	number ln(number)	Calculates natural logarithm of the given parameter.
14	number print(string)	Prints the passed string on the console. Returns 1.
15	number random(number)	Returns a random number x for a given number y with the following condition. 0 <= x < y.
16	number len(string)	Returns the length of a given string.
17	string right(string, n)	Returns a new string containing rightmost n characters of the first parameter.
18	string left(string, n)	Returns a new string containing leftmost n characters of the first parameter.

Appendix 2: Node Op-Codes and IC Instruction Set

Below is the table of the op-codes that can be found in an expression tree and their corresponding mnemonics and semantics.

Op Code	mnemonic	Functional Description
OP_NOP	nop	Does nothing.
OP_COMMA	comma	Accumulates expression value on top of stack. The newly evaluated value remains as word on TOS.
OP_ASSIGN	assign	Assigns the value at the top of the stack to the value. The instruction p1 parameter is the word index. When set, the strConst member of the instruction is the name of the word.
OP_COND	cond	Appears in expression trees to denote the conditional evaluation (ternary) operator. This is not an IC instruction.
OP_ALT	alt	Appears in expression trees to denote the alternatives for conditional evaluation (ternary) operator. This is not an IC instruction.
OP_BAND	band	Applies logical and to the two words at the top of the stack. It effectively pops these two values and pushes the result.
OP_BOR	bor	Applies logical or to the two words at the top of the stack. It effectively pops these two values and pushes the result.
OP_EQ	eq	Compares two words at the top of the stack. It effectively pops these two values and pushes the result. The result is logical true if both operands are equal, false otherwise. This op-code must be replaced by a proper LRT instruction by the processing subsequent to the initial expression tree construction for the operations involving strings and arrays.
OP_NEQ	neq	Compares two words at the top of the stack. It effectively pops these two values and pushes the result. The result is logical true if both operands are not equal, false otherwise. This op-code must be replaced by a proper LRT instruction by the processing subsequent to the initial expression tree construction for the operations involving strings and arrays.
OP_LT	lt	Compares two words at the top of the stack. It effectively pops these two values and pushes the result. The result is logical true if the first operand is less than the second, false otherwise.
OP_LTE	lte	Compares two words at the top of the stack. It effectively pops these two values and pushes the result. The result is logical true if the first operand is less than the second operand or equal to the second operand, false otherwise.
OP_GT	gt	Compares two words at the top of the stack. It effectively pops these two values and pushes the result. The result is logical true if the first operand is greater than the second, false otherwise.
OP_GTE	gte	Compares two words at the top of the stack. It effectively pops these two values and pushes the result. The result is logical true if the first operand is greater than the second operand or equal to the second operand, false otherwise.
OP_ADD	add	Adds two words at the top of the stack. It effectively pops these two values and pushes the result. This op-code must be replaced by a proper LRT instruction by the processing subsequent to the initial expression tree construction for the operations involving strings and arrays.
OP_SUB	sub	Subtracts the word at the top of the stack from the word that was pushed before the word appearing at the top of the stack. It effectively pops two values and pushes the result.

Op Code	mnemonic	Functional Description
OP_MUL	mul	Multiplies the word at the top of the stack by the word that was pushed before the word appearing at the top of the stack. It effectively pops two values and pushes the result.
OP_DIV	div	Divides op1 by op2 where the word at the top of the stack is op1, and op2 is the word that was pushed before op2. It effectively pops two values and pushes the result.
OP_MINUS	minus	Changes the value of the word appearing at the top of the stack as negation of its current value. Stack load does not change.
OP_NOT	not	Changes the value of the word appearing at the top of the stack as logical negation of its current value. Stack load does not change.
OP_AA	aa	This denotes the array access operator. This op-code must be replaced by a proper LRT instruction by the processing subsequent to the initial expression tree construction.
OP_CALL	call	Calls the function having the registration index number stored in the numConstant member. The p1 parameter contains number of the actual parameters.
JMP	jmp	Performs jump to the linear IR code location pointed by the p1 member.
JF	jf	When the value of the word at the top of the stack is logical false, this instruction performs a jump to the linear IR code location pointed by the p1 member. This instruction pops the word inspected for the logical check.
JT	jt	When the value of the word at the top of the stack is logical false, this instruction performs a jump to the linear IR code location pointed by the p1 member. This instruction pops the word inspected for the logical check.
INSID	id	Pushes the value of the word implied by the p1 member onto the stack.
CONST	const	Pushes the immediate value stored in the instruction onto the stack.
LRT	lrt	Performs a call to the language runtime function implied by the p1 member, which must be a valid index to an LRT function. Below is the list of additional critical members in an LRT instruction that need to be generated. • LRT 0 must have type member set correctly. Example: For an array of strings, type will be DGString, and dim will be 1. • LRT 1 must have type member set correctly. Example: For an array of strings, type will be DGString, and dim will be 0. • LRT 3 must have the strConst member must point to the string that will be allocated. The runtime needs to create a copy of string constant. • LRT 6 needs comparison type (test) in the numConstant member.
POP	pop	Pops the value(s) from the stack. The p1 member tells the instruction how many values will be popped. This instruction does not store the popped value any other location. The operation is just adjusting the stack pointer as requested.

Appendix 3: Language Runtime

The functions in the language runtime are not directly called by the DGEval call operator. These are the functions are called implicitly to implement related operators and memory management. These operators are given in the dgevalsup.h between the comments // LRT Start and // LRT End. A short description for each function is also given. Each description starts with index value of the function.

```
// LRT Start

// 0: Allocates an array with the elements on the stack and the supplied type descriptor.
static DGEvalArray *allocatearray(DGEval *dgEval, DGTypeDesc signature, int itemCount, uint64_t
*base);

// 1: Returns an element in an array. Returns the array object passed (arr).
static uint64_t arrayelement(DGEval *dgEval, DGEvalArray *arr, int64_t index);

// 2: Appends an element to an array. Returns the array object passed (arr).
static DGEvalArray *appendelement(DGEvalArray *arr, uint64_t para);

// 3: Allocates a string given a string constant detected in source.
static string *allocatestring(DGEval *dgEval, string *s1);

// 4: Concatenates two strings and returns the concatenated string.
static string *catstring(DGEval *dgEval, string *s1, string *s2);
```

```
// 5: Converts a number to string.
static string *number2str(DGEval *dgEval, double n);

// 6: Compares two strings an returns the test result as true or false.
// test parameter encoding: 0->EQ, 1:NEQ, 2->GT, 3->LT, 4->GTE, 5->LTE
static int64_t strcmp(string *s1, string *s2, int64_t test);

// 7: Compares two arrays. Returns true in case of equality, false otherwise.
static int64_t arrcmp(DGEvalArray *arr1, DGEvalArray *arr2);

// 8: Performs clean-up after execution. This must be called before the final return instruction.
static int64_t postexecutecleanup(DGEval *dgEval);

// 9: Checks for an exception and returns true if an exception record was generated.
static int64_t checkexception(DGEval *dgEval);

// LRT End
```

Appendix 4: Code Samples to Demonstrate Optimizations

Below is the example source used for each optimization option applied in this appendix.

Source	IC with no optimization

`"3"+"24";`
`a+4*(2/4);`
`a=(2)+(1);`
`k=(a+1>2?a:2,a*2);`
	`00000 lrt 3 type:[string:0] "324"`
	`00001 pop 1 type:[none:0]`
	`00002 const 0 type:[number:0] 3`
	`00003 assign 0 type:[number:0] "a"`
	`00004 pop 1 type:[none:0]`
	`00005 id 0 type:[number:0] "a"`
	`00006 const 0 type:[number:0] 1`
	`00007 add 0 type:[number:0]`
	`00008 const 0 type:[number:0] 2`
	`00009 gt 0 type:[number:0]`
	`00010 jf 13 type:[number:0]`
	`00011 id 0 type:[number:0] "a"`
	`00012 jmp 14 type:[number:0]`
	`00013 const 0 type:[number:0] 2`
	`00014 id 0 type:[number:0] "a"`
	`00015 const 0 type:[number:0] 2`
	`00016 mul 0 type:[number:0]`
	`00017 pop 1 type:[none:0]`
	`00018 assign 1 type:[number:0] "k"`
	`00019 pop 1 type:[none:0]`
	`00020 id 0 type:[number:0] "a"`
	`00021 const 0 type:[number:0] 2`
	`00022 add 0 type:[number:0]`
	`00023 pop 1 type:[none:0]`
	`00024 lrt 8 type:[none:0] 0`

Ineffective Statements (-p1 / OPTIMIZE_DC_STATEMENT)	Ineffective Expression Parts (-p2 / OPTIMIZE_DC_EXPPART)
00000 const 0 type:[number:0] 3 00001 assign 0 type:[number:0] "a" 00002 pop 1 type:[none:0] 00003 id 0 type:[number:0] "a" 00004 const 0 type:[number:0] 1 00005 add 0 type:[number:0] 00006 const 0 type:[number:0] 2 00007 gt 0 type:[number:0] 00008 jf 11 type:[number:0] 00009 id 0 type:[number:0] "a" 00010 jmp 12 type:[number:0] 00011 const 0 type:[number:0] 2 00012 id 0 type:[number:0] "a" 00013 const 0 type:[number:0] 2 00014 mul 0 type:[number:0] 00015 pop 1 type:[none:0] 00016 assign 1 type:[number:0] "k" 00017 pop 1 type:[none:0] 00018 lrt 8 type:[none:0] 0	00000 lrt 3 type:[string:0] "324" 00001 pop 1 type:[none:0] 00002 const 0 type:[number:0] 3 00003 assign 0 type:[number:0] "a" 00004 pop 1 type:[none:0] 00005 id 0 type:[number:0] "a" 00006 const 0 type:[number:0] 1 00007 add 0 type:[number:0] 00008 const 0 type:[number:0] 2 00009 gt 0 type:[number:0] 00010 jf 13 type:[number:0] 00011 id 0 type:[number:0] "a" 00012 jmp 14 type:[number:0] 00013 const 0 type:[number:0] 2 00014 assign 1 type:[number:0] "k" 00015 pop 1 type:[none:0] 00016 id 0 type:[number:0] "a" 00017 const 0 type:[number:0] 2 00018 add 0 type:[number:0] 00019 pop 1 type:[none:0] 00020 lrt 8 type:[none:0] 0

Ineffective Statements and Ineffective Expression Parts (-p3 / OPTIMIZE_DC_STATEMENT | OPTIMIZE_DC_EXPPART)	Ineffective Store Load (-p4 / OPTIMIZE_PH_OFFLOAD)
00000 const 0 type:[number:0] 3 00001 assign 0 type:[number:0] "a" 00002 pop 1 type:[none:0] 00003 id 0 type:[number:0] "a" 00004 const 0 type:[number:0] 1 00005 add 0 type:[number:0] 00006 const 0 type:[number:0] 2 00007 gt 0 type:[number:0] 00008 jf 11 type:[number:0] 00009 id 0 type:[number:0] "a" 00010 jmp 12 type:[number:0] 00011 const 0 type:[number:0] 2 00012 assign 1 type:[number:0] "k" 00013 pop 1 type:[none:0] 00014 lrt 8 type:[none:0] 0	00000 lrt 3 type:[string:0] "324" 00001 pop 1 type:[none:0] 00002 const 0 type:[number:0] 3 00003 assign 0 type:[number:0] "a" 00004 const 0 type:[number:0] 1 00005 add 0 type:[number:0] 00006 const 0 type:[number:0] 2 00007 gt 0 type:[number:0] 00008 jf 11 type:[number:0] 00009 id 0 type:[number:0] "a" 00010 jmp 12 type:[number:0] 00011 const 0 type:[number:0] 2 00012 id 0 type:[number:0] "a" 00013 const 0 type:[number:0] 2 00014 mul 0 type:[number:0] 00015 pop 1 type:[none:0] 00016 assign 1 type:[number:0] "k" 00017 pop 1 type:[none:0] 00018 id 0 type:[number:0] "a" 00019 const 0 type:[number:0] 2 00020 add 0 type:[number:0] 00021 pop 1 type:[none:0] 00022 lrt 8 type:[none:0] 0

Constant Value Sink (-p8 / OPTIMIZE_PH_CONSTSINK)	Ineffective Store Load and Constant Value Sink (-p12 / OPTIMIZE_PH_OFFLOAD | OPTIMIZE_PH_CONSTSINK)
00000 const 0 type:[number:0] 3	00000 const 0 type:[number:0] 3
00001 assign 0 type:[number:0] "a"	00001 assign 0 type:[number:0] "a"
00002 pop 1 type:[none:0]	00002 const 0 type:[number:0] 1
00003 id 0 type:[number:0] "a"	00003 add 0 type:[number:0]
00004 const 0 type:[number:0] 1	00004 const 0 type:[number:0] 2
00005 add 0 type:[number:0]	00005 gt 0 type:[number:0]
00006 const 0 type:[number:0] 2	00006 jf 9 type:[number:0]
00007 gt 0 type:[number:0]	00007 id 0 type:[number:0] "a"
00008 jf 11 type:[number:0]	00008 jmp 10 type:[number:0]
00009 id 0 type:[number:0] "a"	00009 const 0 type:[number:0] 2
00010 jmp 12 type:[number:0]	00010 id 0 type:[number:0] "a"
00011 const 0 type:[number:0] 2	00011 const 0 type:[number:0] 2
00012 id 0 type:[number:0] "a"	00012 mul 0 type:[number:0]
00013 const 0 type:[number:0] 2	00013 pop 1 type:[none:0]
00014 mul 0 type:[number:0]	00014 assign 1 type:[number:0] "k"
00015 pop 1 type:[none:0]	00015 pop 1 type:[none:0]
00016 assign 1 type:[number:0] "k"	00016 id 0 type:[number:0] "a"
00017 pop 1 type:[none:0]	00017 const 0 type:[number:0] 2
00018 id 0 type:[number:0] "a"	00018 add 0 type:[number:0]
00019 const 0 type:[number:0] 2	00019 pop 1 type:[none:0]
00020 add 0 type:[number:0]	00020 lrt 8 type:[none:0] 0
00021 pop 1 type:[none:0]	
00022 lrt 8 type:[none:0] 0	

All optimizations (-p15 or without optimization argument)	
00000 const 0 type:[number:0] 3 00001 assign 0 type:[number:0] "a" 00002 const 0 type:[number:0] 1 00003 add 0 type:[number:0] 00004 const 0 type:[number:0] 2 00005 gt 0 type:[number:0] 00006 jf 9 type:[number:0] 00007 id 0 type:[number:0] "a" 00008 jmp 10 type:[number:0] 00009 const 0 type:[number:0] 2 00010 assign 1 type:[number:0] "k" 00011 pop 1 type:[none:0] 00012 lrt 8 type:[none:0] 0	