published on Novermber 15, 2015

Programming Languages: Assignment #3

Due December 1, 2015 at 11pm

Description

In this assignment you are requested to implement a simple language with integers and arrays. You will practice in this assignment with flex and bison. Most of the work of this assignment should be done by flex and bison, in other words, the lesser C/C++ code the better your work is.

Definitions

Whitespace:

`' ' (space), \t or (tab)`

End of line (EOL):

`\n`

Types

Integer

Integer literal has multiple representations:

1. Decimal: a sequence of digits (at least one). For example: 13. If the sequences length is more than 1 than the first character cannot be 0, i.e., 06 is NOT a valid integer number.
2. Hexadecimal: a sequence of $0x$ followed by a non empty sequence of $[0-9a-fA-F]$. For example: $0x13a$ $(=314)$, $0x012F$ $(=303)$.
3. Binary: The sequence of $0b$ followed by a non empty sequence of $0|1$. For example: $0b1010$ $(=5)$, $0b010$ $(=2)$.

Array

The type of elements in array is limited only to integer. Array literal:

1. Empty array: [ ] or *nil*.
2. Non-empty array: array elements enclosed by square brackets and separated by commas. Example: $[1,2,3]$, $[1+1,2*5]$ $(=[2,10])$.

Operators

Table 1: Integer Operations

Priority	Character	Description	Association
Highest	( )	Grouping	
	~ $	self decrement (~), increment ($)	
	^	Power	right
	- +	Unary Operations	
	* /	Multiplication, Division	left
	+ -	Addition and Subtraction	left
	== != < > <= >=	Comparison operators	
	&&	Logic And	left
Lowest	||	Logic Or	left

Remarks:

1. `^` is power operator (e.g. `2^3 = 8`). You can assume the exponent is always positive integer (including 0).
2. The `~`, `$` operation are the equivalent of `--`, `++` in C/C++, respectively. Can operate only on variables.
3. For the comparison and logical operations integral, value $! = 0$ means *true* and $== 0$ *false*.
4. In `/` (division) you need to check for zero division. Also, always use floor for division (e.g. $2/3 = 0$).
5. Evaluating Boolean arithmetic is according to short circuit rule
   (a) expr1 `&&` expr2: if expr1 is false, then expr2 is not exaluated.
   (b) expr1 || expr2: if expr1 is true, then expr2 is not exaluated.

Integer/char operations examples

```
1 + 3 * 5^2                 # = 76
(1 + 0x3 * 5)^2             # = 256
1 / 2 + 0b101               # = 5
-2 - 5                      # = -7
-2 - - 5                    # = 3
-2 - - -5                   # = -7
-2 + + -5                   # = -7
11 == 1 + 6                 # = 0 (false)
11 == 5 + 6                 # = 1 (true)
6 && 5                      # = 1
6 && 5 && 0                 # = 0
6 && 5 || 0                 # = 1
-6 && (5 || 0)              # = 1
```

Table 2: Array Operations

Priority	Character	Description	Association
Highest	( )	Grouping	
	::	Appends an element to the beginning of an array	right
	<<	Appends an element to the end of an array	left
	[ ]	Range Operator	left
Lowest	@	Concatenate 2 arrays	left

The operator `{}` is used to access to an element in an array. For example `[10,20,30]{1}` returns 20.

Array operations examples

```
[1,2,3] @ [4,5]                 # = [1,2,3,4,5]
[1,2,3]{0}                      # = 1
[1,2,3]{2}                      # = 3
[1,2,3]{3}                      # error (out of array range)
nil                             # = []
1::nil                          # = [1]
1::2::3::nil                    # = [1,2,3]
[]<<1<<2<<3                     # = [1,2,3]
[1,2,3,4,5][1..3]               # = [2,3,4] (returns elements from 1 to 3, including 3)
[1,2,3,4,5][1..<3]              # = [2,3]  (returns elements from 1 to 3, excluding 3)
1::[2,3,4]<<5                   # = [1,2,3,4,5] (notice that "::" is applied first)
1::[2,3,4][1..2]<<4[0..1]       # = [2,3]
[1] @ [2]<<5 @ 0::[3]           # = [1,2,5,0,3]
```

Remarks:

1. Notice that `[1,2,3]{0}` returns int, while `[1,2,3][0..0]` returns array
2. Range access operator receives integer literals only (e.g. `[1,2,3][0..(1+1)]` is illegal).

Variables

The syntax for declaring a variable is: `var <VAR_NAME> = expr`.
Right-hand `expr` might be of type *int* or *array*.
Variable name `<VAR_NAME>` is composed of the following characters:`'a'-'z'`, `'0'-'9'` and `'_'` (i.e., a composition of lower case letters, digits and underscore). A variable name must begin with a letter. Examples of valid names are: `"var24", "t", "myvar"`. Examples of invalid names are: `"88", "69_", "aFG", "K", "8i"`.

Examples

```
var x = 99
var y = 9 * 5
var arr = [1,2]
var a = [1,2] @ [3]
```

Alternatively, a variable can be declared along with its type. The syntax for declaring a variable with a type is: `<TYPE> <VAR_NAME> = expr`, where `<TYPE>` is *int* or *Array*. The left-hand side type must match the type of the right-hand side expression.

Examples

```
int     x  =  99        # OK
var     xx =  []        # OK
Array   a  =  []        # OK
int     y  =  []        # error!
Array   z  =  1         # error!
```

The syntax for variable assignment is: `<VAR_NAME> = expr`. Notice that in assignment the left-hand side variable must be previously declared and its type must match the type of the right-hand side expression.

Examples

```
int x = 99
var y = 100
x = 15              # now x equals to 15
y = 16              # OK, the type of 'y' is int
x = []              # error!
```

Whitespace

Whitespace `' '` (space) or `\t` (tab) are ignored (only useful to separate tokens).

If-else expression

The syntax for if-then-else is: `if int_expr then int_expr else int_expr`. In case the `int_expr` after the 'if' keyword is evaluated to 1 (true) the expression for 'then' is evaluated, otherwise the `int_expr` for the 'else' is evaluated. An If-then expression (i.e. without else) is not supported.

```
if 1 then 2 else 3                      # = 2
if 0 then 2 else 3                      # = 3
if 0 then 1 else if 0 then 2 else 4     # = 4
if 1 then if 0 then 2 else 3 else 4     # = 3
```

The precedence of if-then-else is the lowest (in other words the precedence of the 'else' token is the lowest).

```
if 1 then 2 else 2 || 3                 # = if 1 then 2 else (2 || 3) = 2
if 1 then 1 else 2 + 3                  # = if 1 then 1 else (2 + 3) = 1
(if 1 then 1 else 2) + 3                # = 4
```

Notice that if, for example, the "then-expression" is evaluated, it follows that the "else-expression" must not be evaluated.

```
var i = 0
if 1 then $i else ~i                    # i = 1
```

Output

Each line of expression or statement ends by `'\n'` or `';'`. Only lines that are ended with `'\n'` are outputted. Expressions that end with `';'` are not outputted even if after `';'` a `'\n'` is followed. Empty lines that contain only `'\n'` or `';'` are OK (no error and no feedback).

Print int expression

Format: `it=<int expression>`

```
1 + 5 * 1
it=6
```

Print array expression

Format: `it=[e1,e2,...,e_n]` (no space between commas)

```
[1,2 + 3]
it=[1,5]
[1,2] @ [3,4]
it=[1,2,3,4]
```

Print variable declaration/assigment

Format: `<var name>=<expression>`

```
var x = 12 * 3 +5
x=41
x = 99
x=99
var y = x * 2
y=198
var arr = [1,2,3]
arr=[1,2,3]
Array a = [1,2,4]
a=[1,2,4]
```

Handling Errors

Error may occur:

- Illegal syntax (discovered by flex/bison)
- Duplicate declaration of a variable
- Using undefined variable in expression or assignment
- Left-hand side variable type does not match right-hand side expression
- Zero division
- Out of range array access

- Using `$` or `~` on non-variable

Lines that produce feedback are printed regularly until an error discovered. On error, just print "error" and exit.

More Examples

```
var x = 0
x=0
if x then $x else ~x;x
it=-1
if x then $x else ~x;x
it=0
$x && $x; x
it=2
x=0;x$ && $x; x
it=1
if if 1 then 1 else 0 then 1 else 2
it=1
int y=  1; x=2;([x,y, x+y] @ 0::[1,-2^3]<<5)
it=[2,1,3,0,1,-8,5]
```

Submission Notes

- Assignment name: Ex3
- Assignment weight: `24%`
- The assignment should run on the department computer: u2
- Submission via 'submit' system (submit.cs.biu.ac.il).
- You must supply (at least) the following files: makefile, ex3.lex ex3.y
- At the head of your Flex/Bison files, add a remark with your: name, ID, group, user-name
- The running executable must bear the name: a.out

General Notes

1. You can use (and it's recommended) C/C++ standard libraries (e.g. C++ STL: string, set, map, vector, etc.). Normally, the libraries are used for the grammar semantic (i.e. the actions callback), which should be kept minimal (most of your work is to define Flex/Bison rules).
2. Do try and write lots of test cases and check your work. The one supplied by me, does not pretend to check even `1.78%` of what you will be checked later (after the deadline).
3. You can look at the make file at the end of the PPT `#3`
4. You may consult others (friends, internet, etc...) but copying or communal work is forbidden. Please do not plagiaries. We make all effort to catch such cases.
5. Some characters are different between Linux and Windows OS (e.g., EOL). Thus, when you write your own test files, dont forget to 'dos2unix' when migrating them from your Win platform to Planets Linux.
6. For Boolean arithmetic and if-then-else this may help you: Section [3.4.8] 'Actions in Mid-Rule': http://www.gnu.org/software/bison/manual/bison.html
7. You may consult others (friends, internet, etc...) but copying or communal work is forbidden. Please do not plagiarize. We make all effort to catch such cases (remember, we have accumulated students work for already 5 years).
8. It is a long assignment (definition and implementation). Please read it carefully. Also, make sure to get update on the Q & A.
9. See the published input/output examples. Your grade should be (at least) 100 on those. Later, when coming to grade you, we will use a different, larger and much more vicious test files.