Decaf Programming Language Specification Fall 2020

Introduction

This is the reference manual for the Decaf programming language which is the programming language specifically for the CMPT 379: Compilers (http://anoopsarkar.github.io/compilers-class) course taught by Anoop Sarkar (http://www.cs.sfu.ca/~anoop/) at the SFU Computing Science School (Burnaby campus). For more information and other documentation, see the course web page.

Decaf is a strongly typed C-like language. The feature set is trimmed down considerably from what is usually part of a full-fledged programming language. This is done to keep the programming assignments manageable. Despite these limitations, Decaf will be able to handle interesting and non-trivial programs.

Here is an example Decaf program:

```
extern func print_int(int) void;

package GreatestCommonDivisor {
    var a int = 10;
    var b int = 20;

    func main() int {
        var x, y, z int;
        x = a;
        y = b;
        z = gcd(x, y);

        // print_int is part of the standard input-output library
        print_int(z);
    }

    // function that computes the greatest common divisor
    func gcd(a int, b int) int {
        if (b == 0) { return(a); }
        else { return( gcd(b, a % b) ); }
    }
}
```

Notation

The syntax is specified using Extended Backus-Naur Form (http://en.wikipedia.org/wiki/Extended_Backus-Naur_Form) (EBNF):

```
Production  = production_name "=" [ Expression ] "." .
Expression  = Alternative { "|" Alternative } .
Alternative = Term { Term } .
Term        = production_name | token [ "..." token ] | Group | Option | Repetition | Repetit
ion+ | CommaList .
Group       = "(" Expression ")" .
Option      = "[" Expression "]" .
Repetition  = "{" Expression "}" .
Repetition+ = "{" Expression "}+" .
CommaList   = "{" Expression "}+," .
```

Productions are expressions constructed from terms and the following operators, in increasing precedence:

```
|    alternation
()   grouping
[]   option (0 or 1 Expression)
{}   repetition (0 to n Expressions)
{}+  repetition (1 to n Expressions)
{}+, comma list (1 to n Expressions comma separated, e.g. x, y, z)
```

Lower-case production names are used to identify lexical tokens. Non-terminals are in CamelCase. Lexical tokens are enclosed in double quotes “” or back quotes ``.

The form `a ... b` represents the set of characters from a through b as alternatives. The horizontal ellipsis `...` is also used elsewhere in the spec to informally denote various enumerations or code snippets that are not further specified. The character `...` is not a token of the Decaf language.

Source code representation

Decaf source code is encoded as ASCII text. Upper and lower case characters are considered different characters. For example, `if` is defined as a keyword, but `IF` would be considered an identifier.

ASCII table

The ASCII table and decimal equivalent for each character is shown below:

```
  0 nul    1 soh    2 stx    3 etx    4 eot    5 enq    6 ack    7 bel
  8 bs     9 ht    10 nl    11 vt    12 np    13 cr    14 so    15 si
 16 dle   17 dc1   18 dc2   19 dc3   20 dc4   21 nak   22 syn   23 etb
 24 can   25 em    26 sub   27 esc   28 fs    29 gs    30 rs    31 us
 32 sp    33  !    34  "    35  #    36  $    37  %    38  &    39  '
 40  (    41  )    42  *    43  +    44  ,    45  -    46  .    47  /
 48  0    49  1    50  2    51  3    52  4    53  5    54  6    55  7
 56  8    57  9    58  :    59  ;    60  <    61  =    62  >    63  ?
 64  @    65  A    66  B    67  C    68  D    69  E    70  F    71  G
 72  H    73  I    74  J    75  K    76  L    77  M    78  N    79  O
 80  P    81  Q    82  R    83  S    84  T    85  U    86  V    87  W
 88  X    89  Y    90  Z    91  [    92  \    93  ]    94  ^    95  _
 96  `    97  a    98  b    99  c   100  d   101  e   102  f   103  g
104  h   105  i   106  j   107  k   108  l   109  m   110  n   111  o
112  p   113  q   114  r   115  s   116  t   117  u   118  v   119  w
120  x   121  y   122  z   123  {   124  |   125  }   126  ~   127 del
```

The set of valid characters in Decaf is all the ASCII characters:

```
all_char = /* all ASCII characters from 7 ... 13 and 32 ... 126 */ .
char = /* all ASCII characters from 7 ... 13 and 32 ... 126 except char 10 "\n", char 92 "\"
 and char 34 """ */
char_lit_chars = /* all ASCII characters from 7 ... 13 and 32 ... 126 except char 39 "'" and
 char 92 "\" */ .
char_no_nl = /* all ASCII characters from 7 ... 13 and 32 ... 126 except char 10 "\n" */ .
```

Implementation restriction: For compatibility with other tools, a compiler should always disallow the `nul` character (decimal: 0) in the source text.

Letters and Digits

The underscore character `_` is considered a letter.

```
letter        = "A" ... "Z" | "a" ... "z" | "_" .
decimal_digit = "0" ... "9" .
hex_digit     = "0" ... "9" | "A" ... "F" | "a" ... "f" .
digit         = "0" ... "9" .
```

Lexical Elements

Comments

Decaf only has line comments that start with the character sequence `//` and stop at the newline character. The newline character is assumed to be part of the comment. The comment representation is as follows:

```
// this is a line comment and it includes the newline at the end of the line\n

comment = // { char_no_nl } \n
```

Whitespace

Whitespace is used to separate tokens, and is defined as follows:

```
newline         = /* ASCII character nl : '\n' */ .
carriage_return = /* ASCII character cr : '\r' */ .
horizontal_tab  = /* ASCII character ht : '\t' */ .
vertical_tab    = /* ASCII character vt : '\v' */ .
form_feed       = /* ASCII character np : '\f' */ .
space           = /* ASCII character sp : ' ' */ .
whitespace      = { newline | carriage_return | horizontal_tab | vertical_tab | form_feed | s
pace }+ .
```

The following are special characters that are not part of white space:

```
bell        = /* ASCII character bel : '\a' */ .
backspace   = /* ASCII character bs : '\b' */ .
```

Tokens

Tokens are the vocabulary of the Decaf language. There are four classes: identifiers, keywords, operators and literals. White space is ignored except as it separates tokens that would otherwise combine into a single token. For example, `int3` is a single token but `int 3` is two tokens, a keyword `int` and integer `3`; and `int(3)` is a sequence of four tokens: `int`, `(`, `3` and `)`.

While breaking the input into tokens, the next token is the longest sequence of characters that form a valid token.

Semicolons

The Decaf language uses semicolons `;` as a terminator in a number of productions.

Identifiers

Identifiers name program entities such as variables and types. An identifier is a sequence of one or more letters and digits. The first character in an identifier must be a letter.

```
identifier = letter { letter | digit } .
```

For example:

```
a
x9
_x9
ThisVariableIsInCamelCase
```

Type and constant identifiers are predeclared.

Keywords

The following keywords are reserved and may not be used as identifiers.

```
bool    break   continue  else   extern  false
for     func    if        int    null    package
return  string  true      var    void    while
```

Operators and Delimiters

The following character sequences represent operators (see Operators section below) and delimiters.

```
{  }   [   ]   ,   ;   (   )  =
-  !   +   *   /   <<  >>  <  >
%  <=  >=  ==  !=  &&  ||  .
```

Integer literals

An integer literal is a sequence of digits representing an integer constant. An optional prefix sets a non-decimal base: 0x or 0X for hexadecimal. In hexadecimal literals, letters a-f and A-F represent values 10 through 15.

```
int_lit     = decimal_lit | hex_lit .
decimal_lit = { decimal_digit }+ .
hex_lit     = "0" ( "x" | "X" ) { hex_digit }+ .
```

For example, the following are integer literals:

```
42
0xBadFace
170141183460469231731687303715884105727
```

For integer literals, the semantics of range checking occurs later, so that a long sequence of digits such as the last example above which is clearly out of range is still scanned as a single token. The semantic analyzer will come in later and reject this lexeme value as a valid integer constant.

Character literals

A character literal represents a character constant (see Constants section), which is an integer value that identifies an ASCII eqivalent. A character literal is expressed as one or more characters enclosed in single quotes. Within the quotes, any character may appear except single quote. A single quoted character represents the ASCII value of the character itself, while multi-character sequences beginning with a backslash encode special ASCII values as escaped characters.

The simplest form represents the single character within the quotes which is equal to the integer ASCII value. For example:

```
'a' // equal to ASCII value 97 stored as an integer type
```

After a backslash, certain single-character escapes represent special values:

```
\a   ASCII 7:   alert or bell
\b   ASCII 8:   backspace
\t   ASCII 9:   horizontal tab
\n   ASCII 10:  line feed or newline
\v   ASCII 11:  vertical tab
\f   ASCII 12:  form feed
\r   ASCII 13:  carriage return
\\   ASCII 92:  backslash
\'   single quote
\"   double quote
```

The following is a list of escaped character codes that refer to the equivalent ASCII codes above.

```
char_lit     = "'" ( char_lit_chars | escaped_char ) "'" .
escaped_char = "\" ( "n" | "r" | "t" | "v" | "f" | "a" | "b" | `\` | "'" | `"` ) .
```

The following are some legal and illegal examples of character literals:

```
'a'
'\t'
'"'
'\"'
'aa'         // illegal: too many characters
''           // illegal: no characters
'\'          // illegal: invalid closing delimiter
```

Unterminated character literals must be reported as errors.

String literals

A string literal represents a string constant obtained from concatenating a sequence of characters (also see the Constants section below).

```
string_lit = `"` { char | escaped_char } `"` .
```

A string literal must start and end on a single line, it cannot be split over multiple lines. It can include escape sequences like \n and this is distinct from a newline character inside the string constant.

For example, the following is legal:

```
"\n"
```

But the following is not legal:

```
"
"
```

Escaped characters have to used correctly. The following string has an invalid escaped character:

```
"\"
```

You can have an single quote or escaped single quote in a string. So both of the following are valid.

```
"'"
"\'"
```

Empty strings are allowed.

```
""
```

Unterminated string literals must be reported as errors.

Type literals

The following are the keywords used to specify Decaf types.

```
int bool void string
```

Boolean constant literals

The following keywords are used as constants for boolean types.

```
true false
```

List of Tokens

The following is an alphabetically sorted list of tokens for Decaf with the token names used in the homework.

```
T_AND            &&
T_ASSIGN         =
T_BOOLTYPE       bool
T_BREAK          break
T_CHARCONSTANT   char_lit (see section on Character literals)
T_COMMA          ,
T_COMMENT        comment
T_CONTINUE       continue
T_DIV            /
T_DOT            .
T_ELSE           else
T_EQ             ==
T_EXTERN         extern
T_FALSE          false
T_FOR            for
T_FUNC           func
T_GEQ            >=
T_GT             >
T_ID             identifier (see section on Identifiers)
T_IF             if
T_INTCONSTANT    int_lit (see section on Integer literals)
T_INTTYPE        int
T_LCB            {
T_LEFTSHIFT      <<
T_LEQ            <=
T_LPAREN         (
T_LSB            [
T_LT             <
T_MINUS          -
T_MOD            %
T_MULT           *
T_NEQ            !=
T_NOT            !
T_NULL           null
T_OR             ||
T_PACKAGE        package
T_PLUS           +
T_RCB            }
T_RETURN         return
T_RIGHTSHIFT     >>
T_RPAREN         )
T_RSB            ]
T_SEMICOLON      ;
T_STRINGCONSTANT string_lit (see section on String literals)
T_STRINGTYPE     string
T_TRUE           true
T_VAR            var
T_VOID           void
```

```
T_WHILE          while
T_WHITESPACE     whitespace (see section on Whitespace)
```

Types

Decaf has four types: void, booleans, integers and strings. String types, however, can only be used with extern functions. Void types are for return types of functions only (called MethodType below) and not used in variable declarations.

```
ExternType = ( string | Type ) .
Type = ( int | bool ) .
MethodType = ( void | Type ) .
```

Decaf also has a limited *array type* for arrays of integers and booleans.

Boolean types

A boolean type represents the set of Boolean truth values denoted by the predeclared constants `true` and `false`. The predeclared boolean type is `bool`. This is represented as the LLVM type `Int1`.

```
BoolConstant = ( true | false ) .
```

Integer types

A integer type refers to the set of all signed 32 bit integers (-2147483648 to 2147483647) corresponding to the LLVM type `Int32`. The predeclared integer type is `int`.

String types

A string type represents the set of string values. A string value is a (possibly empty) sequence of bytes. Strings are immutable: once created, it is impossible to change the contents of a string. The LLVM type that corresponds to a Decaf string type is `Int8Ptr`. The predeclared string type is `string`.

Array types

Decaf has integer and boolean arrays. However, arrays are declared only in the global (package declaration) scope as part of the field declarations (see FieldDecl).

```
ArrayType = "[" int_lit "]" Type .
```

All arrays are one-dimensional and have a size that is fixed at compile-time. Arrays are indexed from 0 to n – 1, where n > 0 is the size of the array. The usual bracket notation is used to index arrays. Since arrays have a compile-time fixed size and cannot be declared as method parameters (or local variables), there is no facility to query the length of an array variable in Decaf. Arrays must be initialized to all zeroes at declaration time.

Constants

Decaf has *boolean constants*, *integer constants*, and *string constants*. Integer constants can be created using character literals and integer literals. `BoolConstant` is defined (in the Types section) as either `true` or `false`.

```
Constant = ( int_lit | char_lit | BoolConstant ) .
```

String constants can only be used with extern functions. See the Types section for more details.

Decaf program structure

Program

A Decaf program starts with optional external function declarations followed by the package definition (a Decaf package is like a module or namespace). A package has optional global variables (called field variables) followed by method (function) definitions.

```
Program = Externs package identifier "{" FieldDecls MethodDecls "}" .
```

External Functions

A Decaf program can access external function that are linked, such as the Decaf standard library functions which are implemented in C, and accessed from within the Decaf program as external functions. For now, only external functions are allowed. External data cannot be declared.

```
Externs    = { ExternDefn } .
ExternDefn = extern func identifier "(" [ { ExternType }+, ] ")" MethodType ";" .
```

Global variables

Decaf has global variables with scope limited to their package that appear before any method declarations. Global variables in Decaf are called *field declarations*. They can be simple declarations without initialization (assumed to be zero initialized by the compiler) or non-array variables can be declared with an assignment to a constant (see Constants section). Variables are always defined using the `var` reserved word.

```
FieldDecls = { FieldDecl } .
FieldDecl  = var { identifier }+, Type ";" .
FieldDecl  = var { identifier }+, ArrayType ";" .
FieldDecl  = var identifier Type "=" Constant ";" .
```

The assignment to an identifier has to be a constant:

```
package foo { var a int; var b int = a; } // Invalid!
```

The following is an example of an array field declaration. Notice the array type has the length before the type of the elements of the array.

```
package foo { var list [100]int; } // Array declaration
```

Method declarations

Functions or methods in Decaf start with the reserved word `func`, then the name of the method and in parentheses is the argument list followed by the return type of the method.

```
MethodDecls = { MethodDecl } .
MethodDecl  = func identifier "(" [ { identifier Type }+, ] ")" MethodType Block .
```

The program must contain a declaration for a method called `main` that has no parameters. The return type of the method `main` can be either type `int` or `void`. Execution of a Decaf program starts at this method `main`. Methods defined as part of a package can have zero or more parameters and must have a return statement of type `MethodType` explicitly or implicitly defined, e.g. if your `main` function which does not have an explicit return statement can either have a `ret i32 0` or `ret void` in LLVM assembly depending on the return type of the `main` function, either `int` or `void` respectively.

Blocks

Decaf blocks have a section for local variable definitions first followed by statements.

```
Block = "{" VarDecls Statements "}" .
```

Variable Declarations

Local variables are declared using the reserved word `var` followed by a comma separated list of variables for each type and followed by the type of the variable(s). They cannot be assigned a value when they are defined.

```
VarDecls = { VarDecl } .
VarDecl  = var { identifier }+, Type ";" .
```

There is no assignment allowed for local variables:

```
func foo() int { var a int = 10; } // Invalid!
```

Statements

Statements in Decaf consist of variable assignment, method calls, syntax for various kinds of control flow, special statements for breaking out of or continuing to the top of the block.

```
Statements = { Statement } .
```

Blocks statement

Statements can also be Blocks (see section on Blocks).

```
Statement = Block .
```

Assign statement

Assignment to an `Lvalue` is a statement in `Decaf`. The location for the `Lvalue` can be either a scalar variable or an array location.

```
Statement = Assign ";" .
Assign    = Lvalue "=" Expr .
Lvalue    = identifier | identifier "[" Expr "]" .
```

Method calls

```
Statement  = MethodCall ";" .
MethodCall = identifier "(" [ { MethodArg }+, ] ")" .
MethodArg  = Expr | string_lit .
```

External functions are declared using the extern keyword. These functions are provided at using a separate library which is linked with your Decaf program at runtime. Some minimal type checking is done using the declaration. The most useful library functions that you will use are the `print_string`, `print_int` and `read_int` functions.

Unless it is `void`, the return value is a type that can be assigned to an `Lvalue`:

```
z = read_int();
```

In this case, the integer variable `z` receives the result of calling the `read_int` library function. The return value can also be declared to be `void` in which case assigning the output of a library function to an `Lvalue` will result in a semantic error.

If statement

```
Statement = if "(" Expr ")" Block [ else Block ] .
```

While statement

```
Statement =  while "(" Expr ")" Block .
```

For statement

The `for` loop in Decaf has the usual structure `for ( init ; check ; post )` followed by the `Block` of the for loop.

```
Statement = for "(" { Assign }+, ";" Expr ";" { Assign }+, ")" Block .
```

The init, check and post parts of the `for` loop cannot be empty:

```
for(; a < b; ) // Invalid!
```

Return statement

```
Statement = return [ "(" [ Expr ] ")" ] ";" .
```

The following are all valid return statements:

```
return(3);
return(b); // where b was declared as "var b bool;"
return();
return;
```

Break statement

A `break` statement terminates execution of the innermost `for` or `while` loop (branches to end of loop).

```
Statement = break ";" .
```

Continue statement

A `continue` statement begins the next iteration of the innermost `for` or `while` loop at its post statement (see For statement).

```
Statement = continue ";" .
```

Expressions

Operands

Operands are the elementary values in an expression.

```
Expr = identifier .
Expr = MethodCall .
Expr = Constant .
```

Unary Operators

There are only two unary operators in Decaf. One for logical negation and the other for unary minus of arithmetic expressions. The result of `UnaryNot` is of type `bool` and the result of `UnaryMinus` is of type `int` .

```
UnaryOperator = ( UnaryNot | UnaryMinus ) .
UnaryNot = "!" .
UnaryMinus = "-" .
```

Binary Operators

Binary operators are split into boolean binary operators and arithmetic binary operators. The result of using a boolean operator is the type `bool` and the result of using an arithmetic operator is the type `int`.

```
BinaryOperator = ( ArithmeticOperator | BooleanOperator ) .
ArithmeticOperator = ( "+" | "-" | "*" | "/" | "<<" | ">>" | "%" ) .
BooleanOperator = ( "==" | "!=" | "<" | "<=" | ">" | ">=" | "&&" | "||" ) .
```

The boolean connectives `&&` and `||` are interpreted using short circuit evaluation (http://en.wikipedia.org/wiki/Short-circuit_evaluation). This means: the second operand is not evaluated if the result of the first operand determines the value of the whole expression. For example, if the result is `false` for `&&` or `true` for `||`.

Binary `%` computes the modulus of two numbers. Given two operands of type `int`, `a` and `b`: If `b` is positive, then `a % b` is a minus the largest multiple of `b` that is not greater than `a`. If `b` is negative, then `a % b` is `a` minus the smallest multiple of `b` that is not less than `a` (in this case the result will be less than or equal to zero).

Operators and Precedence

Unary operators have the highest precedence. For the other binary operators the precedence is defined as follows. All operators at the same precedence level get equal precedence. All operators with equal precedence associate left. The `UnaryMinus` operator associates to the right.

Precedence	Operator
7	`UnaryMinus`
6	`UnaryNot`
5	`* / % << >>`
4	`+ -`
3	`== != < <= > >=`
2	`&&`
1	`||`

Primary expressions

Primary expressions build larger expressions from operands, operators and parentheses. The parentheses are used to group expressions to obtain different orders of evaluation. Parentheses can be omitted if the desired evaluation is consistent with the precedence rules (see the Operators and Precedence section). The type of the `Expr` on the left hand side is determined by the `Expr` on the right hand side and the operator used.

```
Expr = Expr BinaryOperator Expr .
Expr = UnaryOperator Expr .
Expr = "(" Expr ")" .
```

Index expression

In this expression, the `identifier` must be an Array Type (see section on Array Types). The `Expr` is evaluated to give an array index and the result of the evaluation must be of type `int` . The integer value is then used to find the element of the array which is of type `int` or `bool` depending on the Array Type.

```
Expr = identifier "[" Expr "]" .
```

Decaf grammar

The entire set of rules that describe the `Decaf` grammar specification is collected in one place below. For explanation of each of the rules read the descriptions provided in the above sections.

```
Program = Externs package identifier "{" FieldDecls MethodDecls "}" .
Externs    = { ExternDefn } .
ExternDefn = extern func identifier "(" [ { ExternType }+, ] ")" MethodType ";" .
FieldDecls = { FieldDecl } .
FieldDecl  = var { identifier }+, Type ";" .
FieldDecl  = var { identifier }+, ArrayType ";" .
FieldDecl  = var identifier Type "=" Constant ";" .
MethodDecls = { MethodDecl } .
MethodDecl  = func identifier "(" [ { identifier Type }+, ] ")" MethodType Block .
Block = "{" VarDecls Statements "}" .
VarDecls = { VarDecl } .
VarDecl  = var { identifier }+, Type ";" .
Statements = { Statement } .
Statement = Block .
Statement = Assign ";" .
Assign    = Lvalue "=" Expr .
Lvalue    = identifier | identifier "[" Expr "]" .
Statement  = MethodCall ";" .
MethodCall = identifier "(" [ { MethodArg }+, ] ")" .
MethodArg  = Expr | string_lit .
Statement = if "(" Expr ")" Block [ else Block ] .
Statement =  while "(" Expr ")" Block .
Statement = for "(" { Assign }+, ";" Expr ";" { Assign }+, ")" Block .
Statement = return [ "(" [ Expr ] ")" ] ";" .
Statement = break ";" .
Statement = continue ";" .
Expr = identifier .
Expr = MethodCall .
Expr = Constant .
UnaryOperator = ( UnaryNot | UnaryMinus ) .
UnaryNot = "!" .
UnaryMinus = "-" .
BinaryOperator = ( ArithmeticOperator | BooleanOperator ) .
ArithmeticOperator = ( "+" | "-" | "*" | "/" | "<<" | ">>" | "%" ) .
BooleanOperator = ( "==" | "!=" | "<" | "<=" | ">" | ">=" | "&&" | "||" ) .
Expr = Expr BinaryOperator Expr .
Expr = UnaryOperator Expr .
Expr = "(" Expr ")" .
Expr = identifier "[" Expr "]" .
ExternType = ( string | Type ) .
Type = ( int | bool ) .
MethodType = ( void | Type ) .
BoolConstant = ( true | false ) .
ArrayType = "[" int_lit "]" Type .
Constant = ( int_lit | char_lit | BoolConstant ) .
```

Decaf Semantics

Program Structure

- A method called `main` has to exist in the Decaf program.

Type Checking

Make sure the following type checks are implemented in the compiler.

- Binary `+ - * / % >> << < > <= >=` and unary `-` only work on integer expressions.
- Binary `&& ||` and unary `!` only work on boolean expressions.
- Binary `== !=` work on any type, but both operands have to have the same type.
- Assignment to a function parameter is valid and should change the value as for a local variable
- The `&&` and `||` operators are short-circuiting (this is already specified in the spec)
- If you have multiple return statements in one block then only the first is used, but the others should still be type checked.
- Indexing a scalar is a semantic error. `{ var x int; x[0] = 1; }` is a semantic error, and `{ var x,y int; y = x[0]; }` is a semantic error, and the same if `x` is a field variable.
- Indexing with a bool is a semantic error. `{ var xs[10] int; func main() int { var x int; x = xs[true]; }` is a semantic error.
- Using a non-bool expression for a loop condition is a semantic error. `{ while (1) {} }` and `{ var x int; for (x = 0; 1; x = x + 1) {} }` are semantic errors.
- Using a non-bool expression in an if statement condition is a semantic error. `{ if (0) {} }` is a semantic error.
- A return statement with an expression is not allowed in function with void return type. `{ func foo() void { return (1); }` and `{ func bar() void {} func foo() void { return (bar()); }` are both semantic errors.
- A return statement with no expression in a non-void function produces an default return value (see “Default values” section below).
- Cannot use a void function in an expression. `func foo() void {} func main() int { if (foo()) {} }` is invalid.
- Cannot call a method with the wrong number of arguments.
- Find all cases where there is a type mismatch between the definition of the type of a variable and a value assigned to that variable. e.g. `bool x; x = 10;` is an example of a type mismatch.
- Find all cases where an expression is well-formed, where binary and unary operators are distinguished from relational and equality operators. e.g. `true + false` is an example of a mismatch but `true != true` is not a mismatch.
- Check that all variables are defined in the proper scope before they are used as an lvalue or rvalue in a Decaf program.
- Check that the return statement in a method matches the return type in the method definition. e.g. `func foo() bool { return(10); }` is an example of a mismatch.

Default values

- Integer variables default to zero when initialized.
- Boolean variables default to False when initialized.
- For function return type: integer return type defaults to zero.
- For function return type: boolean return type defaults to True.

For function return values, here is an example:

```
func panama() bool {
  print_string("Panama");
}
if (flag && panama()) { // panama() will return True
 ...
}
```

Scoping Rules

This section clarifies the behaviour with scoping.

- Having two fields with the same name is a semantic error.
- Having two methods with the same name is a semantic error.
- Having a field and a method with the same name is a semantic error.
- externs count as methods for scoping.
- However, having an extern function with the same name as a function inside a package is allowed (the package should defined a new scope in which the local function is defined). See the example below.
- Having two local variables with the same name declared at the same block is a semantic error. `{ var x int; var x int; }` is an error, but `{ var x int; { var x int; } }` is ok.
- Having a local variable in the outer block of a method that has a parameter with the same name is a semantic error. `func foo(x int) void { var x int; }` is an error, but `func foo(x int) void { { var x int; } }` is ok.
- A function can be referred to anywhere in the program, including before its definition. `package C { func foo() void { bar() }; func bar() void {}; }` is ok.
- Functions, fields, arguments, and local variables all share the same namespace (symbol table) and can shadow each other except for the above rules. e.g. in `package C { func foo() void {}; func bar() void { var foo int; foo(); } }` the `foo` in `foo();` refers to the local int variable, not the function resulting in an error.
- `break` and `continue` only apply to the innermost containing loop. Using `break` or `continue` outside of a loop results in a semantic error.

The following code is acceptable because of the scope defined by the `package` for the functions in the package. The `foo` call in `main` uses the locally scoped `foo` (defined inside the package).

```
extern func foo() int;

package Scoping {
    func foo(x int) void { { var x int; } }
    func main() int { foo(1);  }
}
```

Statements

These are semantic errors that can occur when using statements in Decaf.

- The return type of `main` can be either `int` or `void` (not `bool`) and the return statement inside main must match the return type of `main`.
- Assigning a scalar to an array is considered a type mismatch.
- The following produce undefined behaviour, but must not produce compile time semantic errors:
    - Using the value of any uninitialized scalar variable or array element (this is allowed in the reference implementation)
    - A function with no return statement is equivalent to ending the function with a return statement that has no expression: `return;` (this is allowed in the reference implementation)
- Assigning to an array cell at an invalid index can either produce a compile-time or runtime error.
- Any bool argument to a integer parameter must be converted while keeping its value, not just for `print_int`.
- Passing a argument to a function parameter with a different type is a semantic error except for the special case of passing a `bool` as an `int`.
- Declaring an array of size less than or equal to zero is a semantic error.
- Assignment to a function parameter is valid and should change the value as for a local variable.

Last updated December 07, 2020.
Forked from the JHU MT class code on github (https://github.com/mt-class/jhu) by Matt Post (https://github.com/mjpost) and Adam Lopez (https://github.com/alopez).