The COOL Programming Language

Prof Godfrey C. Muganda
Department of Computer Science
North Central College

CSC 4/565 Fall 2008

This paper defines the syntax and semantics of a small, object-oriented programming language named COOL (Cool Object Oriented Language), a compiler for which will be implemented by students of CSC 4/565. To keep the implementation of the language doable within a 10 week course, COOL lacks many features found in an industry strength programming language. It does contain enough features, however, so that a person who successfully implements a compiler for COOL ++ should be able to see how to implement a compiler for full-featured object-oriented languages.

COOL supports integer and boolean as built-in types, together with the usual operations on those two types. In addition, COOL allows user-defined classes and arrays of any supported type. Arrays are limited to one dimension.

In its object-oriented features, COOL is patterned after modern object-oriented languages such as Java and C#. The language's class structure allows methods and member variables of classes to be designated public, protected, or private. Classes can have constructors, and both methods and constructors can be overloaded. Inheritance is supported, and subclasses may redefine inherited methods (that is, method overriding is supported). The language uses dynamic method binding and supports polymorphism. Finally, the language has features for handling exceptions.

The language has an operator **new** that is used to create new objects. There is no corresponding `delete` operator. Ideally, an implementation of COOL would provide automatic garbage collection (our will not).

The syntax of the language is described using the following EBNF form, a variant of context free grammars. The nonterminal symbols are given in uppercase, whereas the terminal symbols (tokens) are given in lower case. The tokens written in bold face are reserved words. The empty string can appear as part of some syntactic constructs: it is denoted by the Greek letter ϵ.

PROGRAM ::= {CLASS | METHOD}

ACCESS_SPEC ::= **private** | **protected** | **public**
ADDOP ::= + | -
ALLOCATOR ::= **new** TYPE_ID (ARGLIST)
ALLOCATOR ::= **new** TYPE_ID[EXPR]
ARGLIST ::= EXPR {, EXPR } |ϵ
ASSIGNSTMT ::= FACTOR = EXPR
BEXPR ::= SIMPLEEXPR
BEXPR ::= SIMPLEEXPR$_1$ RELOP SIMPLEEXPR$_2$
BLOCK ::= VARDECS **begin** STMTLIST **end**
BODY ::= SUPER_INIT THIS_INIT BLOCK
CALLSTMT ::= **call** FACTOR
CAST_EXPR ::= **cast** (TYPE_ID, EXPR)
CATCH_CLAUSE ::= **catch** (TYPE_ID id) STMTLIST
CEXPR ::= BEXPR { **and** BEXPR }
CLASS ::= **class** id$_1$ SUPER_CLASS **is** { CLASS_MEMBER } **end** id$_2$
CLASS_MEMBER ::= FIELD_DECL
CLASS_MEMBER ::= METHOD_DECL
ELSEPART ::= ϵ | **else** STMTLIST
EXPR ::= CEXPR { **or** CEXPR }
FACTOR$_1$ ::= - FACTOR$_2$
FACTOR$_1$ ::= **not** FACTOR$_2$
FACTOR ::= number
FACTOR ::= **false**
FACTOR ::= **true**
FACTOR ::= **null**
FACTOR ::= ALLOCATOR
FACTOR ::= CAST_EXPR
FACTOR ::= VALUE_OR_REF { MEMBER_PART }
FIELD_DECL ::= ACCESS_SPEC TYPE id { , id };
IFSTMT ::= **if** EXPR **then** STMTLIST
        {**elsif** EXPR **then** STMTLIST}
        ELSEPART
        end if
INPUTSTMT ::= **input** >> FACTOR
LOOPSTMT ::= **loop** STMTLIST **end loop**
MEMBER_PART ::= .id
MEMBER_PART ::= .id(ARGLIST)
MEMBER_PART ::= .id[EXPR]
METHOD ::= **method** M_TYPE METHOD_ID (PARAMETERS) **is** BODY id
METHOD_DECL ::= ACCESS_SPEC **method** M_TYPE id (PARAMETER_DECL);
METHOD_ID ::= id :: id | id
M_TYPE ::= TYPE | **void**
MULTOP ::= $*$ | / | **mod**
OPTIONAL_ID ::= id |ϵ
OPTIONAL_TYPE_ID ::= id |ϵ
OUPUTSTMT ::= **output** << EXPR

OUTPUTSTMT ::= **output** << string
PARAMETER_DECL ::= TYPE OPTIONAL_ID {, TYPE OPTIONAL_ID } | ϵ
PARAMETERS ::= TYPE id {, TYPE id} | ϵ
RELOP ::= == | < | <= | > | >= | #
SIMPLEEXPR ::= TERM { ADDOP TERM }
STMT ::= BLOCK | TRYSTMT
STMT ::= IFSTMT | LOOPSTMT | ASSIGNSTMT
STMT ::= CALLSTMT | OUTPUTSTMT | INPUTSTMT
STMT ::= **continue** | **break**
STMT ::= **return** | **return** EXPR |
STMT ::= **exit**
STMT ::= **throw** EXPR
STMTLIST ::= { STMT; }
SUPER_INIT ::= **super**(ARGLIST); | ϵ
SUPER_CLASS ::= **extends** id | ϵ
TERM ::= FACTOR {MULTOP FACTOR }
THIS_INIT ::= **this**(ARGLIST); | ϵ
TRYSTMT ::= **try** STMTLIST CATCH_CLAUSE { CATCH_CLAUSE } **end try**
TYPE ::= TYPE_ID | TYPE_ID [ ]
TYPE_ID ::= **integer** | **boolean** | id
VALUE_OR_REF ::= **this**
VALUE_OR_REF ::= **super**
VALUE_OR_REF ::= id
VALUE_OR_REF ::= id[EXPR]
VALUE_OR_REF ::= id(ARGLIST)
VALUE_OR_REF ::= (EXPR)
VARDECLIST ::= TYPE id {, id };
VARDECS ::= **declare** VARDECLIST { VARDECLIST } |ϵ

Here is an example of a COOL program. It creates a single object of a class whose constructor prints the message "Hello, World" on the screen.

```
class HelloWorld
  public void HelloWorld();
end HelloWOrld

method void HelloWorld::HelloWorld( ) is
begin
   output << "Hello, World!";
end HelloWorld

method void main() is
begin
  HelloWord helloObj;
  //creating object invokes constructor
  helloObj = new HelloWorld();
end main
```

The language uses C++ style comments: that is, a comment may start with // and end at the end of the line.