A Tour of the Cool Support Code*

1 Introduction

The Cool compiler project provides a number of basic data types to make the task of writing a Cool compiler tractable in the timespan of the course. This document provides an overview of the Cool compiler support code, which includes:

- a linked list data type;
- a string table package;
- a symbol table package;
- miscellaneous utilities for the parser and lexer;
- a package for managing Cool abstract syntax trees;
- routines for printing abstract syntax trees;
- miscellaneous routines for handling multiple input files and command-line flags;
- the runtime system.

This document should be read in conjunction with the source code. With the exception of the abstract syntax tree package (which is automatically generated), there are also extensive comments in the source.

The purpose of the Cool support code is to be easy to understand and use. There are much more efficient implementations of most of the data structures used in the system. It is recommended that students implementing a Cool compiler also stick to simple, obviously correct, and perhaps inefficient implementations—writing a compiler that works correctly is usually difficult enough.

The Cool system is written in C++ and it is assumed that the reader has some familiarity with that language. The base system deliberately uses a simple subset of the language. The heavily used features are: classes, single inheritance, virtual functions, and templates. Overloading is used very sparingly; with one exception, operator overloading is avoided completely (the exception is the operator `<<`). Destructors are not used and in fact memory management is ignored completely. Memory management is a very important part of software development as it is currently practiced in industry, but memory management is also tricky, error-prone, and very time consuming to get completely right. It is suggested that students also not worry about memory management in writing their own Cool compilers.

2 Lists

The file `list.h` implements a simple linked list datatype. The operations are similar to those provided in Lisp-family languages: a constructor `List` adds a new element to the front of a list, `hd` returns the first element of a list, and `tl` returns the tail of a list. Functions are also provided for computing the length of a list, applying a function to every element of a list, and printing a list. Example uses of lists can be found in the implementation of both string tables and symbol tables.

3 String Tables

All compilers manage large numbers of strings such as program identifiers, numerical constants, and string constants. Often, many of these strings are the same. For example, each identifier typically occurs many times in a program. To ensure that string constants are stored compactly and manipulated efficiently, a specialized data structure, the *string table*, is employed.

A string table is a lookup table that maintains a single copy of each string. The Cool string table class provides methods for inserting and querying string tables in a variety of ways (see the file `stringtab.h`). While production compilers use hashed data structures to implement string tables, the Cool string tables are implemented as lists (see `stringtab_functions.h`). The components of Cool string tables are of type `Entry`. Each `Entry` stores a string, the length of the string, and an integer index unique to the string.

An important point about the structure of the Cool compiler is that there are actually three distinct string tables: one for string constants (`stringtable`), one for integer constants (`inttable`), and one for identifiers (`idtable`). The code generator must distinguish integer constants and string constants from each other and from identifiers, because special code is produced for each string constant and each integer constant in the program. Having three distinct string tables makes this distinction easy. Note that each of the three tables has a different element type (`StrEntry`, `IntEntry`, and `IdEntry`), each of which is a derived class of `Entry`. Throughout the rest of the compiler (except parts of the code generator), a pointer to an `Entry` is called a `Symbol`, irrespective of whether the symbol represents an integer, string, or identifier.

Because string tables store only one copy of each string, comparing whether two `IntEntry`s, `StrEntry`s, or `IdEntry`s `x` and `y` represent the same string can be done simply by comparing the two pointers `x == y`. Note that it does not make sense to compare entries from different string tables (e.g., `IntEntry`s with `StrEntry`s) as these are guaranteed to be different even if the strings are the same.

Three methods are provided to add elements to a table: **add_string(char *s,int m)**, which adds a string **s** of at most **m** characters; **add_string(char *s)**, which adds a string **s** to the table; and **add_int(int i)**, which converts integer **i** to a string and adds the string to the table. Each of these methods returns a type derived from **Entry** to describe the symbol table entry, on which the method **get_string** is defined for extracting the entry's string. Before using these functions you should read the documentation in stringtab.cc. If you don't use the interface with the string table manager correctly, your program may crash.

4 Symbol Tables

In addition to strings, compilers must also determine and manage the scope of program names. A symbol table is a data structure for managing scope. Conceptually, a symbol table is just another lookup table. The key is the symbol (the name) and the result is whatever information has been associated with that symbol (e.g., the symbol's type).

In addition to adding and removing symbols, symbol tables also support operations for entering and exiting scopes and for checking whether an identifier is already defined in the current scope. The lookup operation must also observe the scoping rules of the language; if there are multiple definitions of identifier `x`, the scoping rules determine which definition a lookup of `x` returns. In most languages, including Cool, inner definitions hide outer definitions. Thus, a lookup on `x` returns the definition of `x` from the innermost scope with a definition of `x`.

Cool symbol tables are implemented as lists of scopes, where each scope is a list of $\langle \text{identifier}, \text{data} \rangle$ pairs. The "data" is whatever data the programmer wishes to associate with each identifier. The symbol table operations are very straightforward to define on this structure and are documented in `symtab.h`. An example illustrating the use of symbol tables is in the file `symtab_example.cc`.

5 Utilities

The files `utilities.h` and `utilities.cc` define a few functions useful in writing and debugging a Cool parser and lexical analyzer. See the source code for documentation.

6 Abstract Syntax Trees

After lexical analysis and parsing, a Cool program is represented internally by the Cool compiler as an abstract syntax tree. The project comes with a definition of Cool abstract syntax trees (ASTs) built in. The AST package is by far the largest piece of code in the base system and requires the most time to learn. The learning process is made more complex because the AST code is generated automatically from a specification in the file `cool-tree.aps`. While the generated code is quite simple and regular in structure, it is also devoid of comments. This section serves as the documentation for the AST package.

6.1 Phyla and Constructors

The AST data type provides, for each kind of Cool construct, a class for representing expressions of that kind. There is a class for `let` expressions, another class of `+` expressions, and so on. Objects of these classes are nodes in Cool abstract syntax trees. For example, an expression $\mathtt{e}_1 + \mathtt{e}_2$ is represented by a `+` expression object, which has two subtrees: one for the tree representing the expression $\mathtt{e}_1$ and one for the tree representing the expression $\mathtt{e}_2$.

The Cool abstract syntax is specified in a language called APS. In APS terminology, the various kinds of abstract syntax tree nodes (`let`, `+`, etc.) are called *constructors*. (Don't confuse this use of the term "constructor" with C++ constructors; while similar, this is a slightly different meaning taken from functional languages that predates C++.) The form of the AST is described by a set of *phyla*. Each phylum has one or more constructors.

Phyla are really just types. That is, instead of having one large group of undifferentiated constructors, the constructors are grouped together according to function, so that, for example, the constructors for expression ASTs are distinguished from the constructors for class ASTs. The phyla are defined at the beginning of `cool-tree.aps`:

```
module COOL begin
  phylum Program;

  phylum Class_;
```

```
phylum Classes = LIST[Class_];

phylum Feature;
phylum Features = LIST[Feature];

phylum Formal;
phylum Formals = LIST[Formal];

phylum Expression;
phylum Expressions = LIST[Expression];

phylum Case;
phylum Cases = LIST[Case];
```

From the definition it can be seen that there are two distinct kinds of phyla: "normal" phyla and list phyla. "Normal" phyla each have associated constructors; list phyla have a fixed set of list operations.

Each constructor takes typed arguments and returns a typed result. The types may either be phyla or any ordinary C++ type. In fact, the phyla declarations are themselves compiled into C++ class declarations by an APS compiler. A sample constructor definition is

```
constructor class_(name : Symbol; parent: Symbol; features : Features;
                   filename : Symbol) : Class_;
```

This declaration specifies that the `class_` constructor[1] takes four arguments: a `Symbol` (a type identifier) for the class name, a `Symbol` (another type identifier) for the parent class, a `Features`, and a `Symbol` for the filename in which the class definition occurs. The phylum `Features` is defined to be a list of `Feature`'s by the declaration

```
phylum Features = LIST[Feature];
```

See Section **??** for a description of the operations defined on AST lists.

The `class_` constructor returns an AST of type (or phylum) `Class_`. In `cool.y` there is the following example of a use of the `class_` constructor:

```
class   :  CLASS TYPEID INHERITS TYPEID IS optional_feature_list END ';'
           { $$ = class_($2,$4,$6,stringtable.add_string(curr_filename)); }
```

The `class_` constructor builds a `Class_` tree node with the four arguments as children. Because the phyla (types) of the arguments are declared, the C++ type checker enforces that the `class_` constructor is applied only to arguments of the appropriate type. See Section **??** and `cool-tree.aps` to learn the definitions of the other constructors.[2]

There is a real danger of getting confused because the same names are used repeatedly for different entities in different contexts. In the example just above, small variations of the name `class` are used for a terminal (`CLASS`), a non-terminal (`class`), a constructor (`class_`), and a phylum (`Class_`). These uses are all distinct and mean different things. There is also a `class_` member of the `union` declaration in `cool.y`, which means yet something else. Most uses are distinguished consistently by capitalization, but a few are not. When reading the code it is important to keep in mind the role of each symbol.

[1]The name `class_` is chosen to avoid a conflict with the C++ keyword `class`.

[2]Comments in `cool-tree.aps` begin with two hyphens "– –".

6.2 AST Lists

List phyla have a distinct set of operations for constructing and accessing lists. Note that the AST package uses its own definition of lists, which is distinct from and has different operations than the list type defined in `list.h`.

For each phylum named X there is a phylum called Xs (except for `Classes`, which is a list of `Class_` nodes) of type `List[X]`. List functions are defined automatically for each list. For the `Class_` phylum some of the list functions and methods are:

```
Classes  nil_Classes();
Classes  single_Classes(Class_);
Classes  append_Classes(Classes,Classes);
Class_   nth(int index);
int      len();
```

These functions will be familiar to anyone with a passing knowledge of Lisp or Scheme. The function `nil_`*`phylums`*`()` returns an empty list of type *phylum*. The function `single_`*`phylums`* makes a list of length 1 out of its *phylum* argument. The function `append_`*`phylums`* appends two lists of *phylums*. The method `nth` selects the `index`'th element of its list argument. The method `len` returns the length of the list.

AST lists also have a simple list iterator. There is a method `first` that returns the index of the first element of the list, a predicate `more` that is false if its index argument is the last element of the list, and a method `next` that returns the next index of the list. This iterator is quite naive and inefficient; to find the nth element of the list, up to n elements of the list must be examined. However, it is simple to understand and use. The list functions are defined in `cool-X-tree.h` and `tree.h`. A typical use of the iterator functions to walk through a list `l` is:

```
for(int i = l->first(); l->more(i); i = l->next(i))
     { . . . do something with l->nth(i) . . . }
```

6.3 The AST Class Hierarchy

With the exception of lists, all AST classes are derived from the class `tree_node`. All of the lists are lists of `tree_node`s. The `tree_node` class and the AST list template are defined in `tree.h`.

The `tree_node` class definition contains everything needed in an abstract syntax tree node except information specific to particular constructors. There is a protected data member `line_number`, the line number where the expression corresponding to the AST node appeared in the source file. The line number is used by a Cool compiler to give good error messages.

Several functions are defined on all `tree_node`s. The important functions are: `dump`, which pretty prints an AST and `get_line_number`, which is a selector for the corresponding data member.

Each of the phyla is a class derived directly from `tree_node`. As stated previously, the phyla exist primarily to group related constructors together and as such do not add much new functionality.

Each of the constructors is a class derived from the appropriate phyla. Each of the constructor classes defines a function of the same name that can be used to build AST nodes. The `dump` function is also defined automatically for each constructor.

6.4 Class Members

Each class definition of the tree package comes with a number of members. Some of the member functions are discussed above. This section describes the data members and some more (but not all) of the rest of the functions, as well as how to add new members to the classes.

Each constructor has data members defined for each component of that constructor. The name of the member is the name of the field in the constructor, and it is only visible to member functions of the constructor's class or derived classes. For example, the `class_` constructor has four data members:

```
Symbol name;
Symbol parent;
Features features;
Symbol filename;
```

Here is a complete use of one member:

```
class__class c;
Symbol p;

Symbol class__class::get_parent() { return parent; }

c = class(idtable.add_string("Foo",3),idtable.add_string("Bar"),nil_Features(),
          stringtable.add_string("filename"));
p = c->get_parent();  // Sets p to the symbol for "Bar"
```

It will be useful in writing a Cool compiler to extend the AST with new functions such as `get_parent`. Simply modify the `cool-tree.h` file to add functions to the class of the appropriate phylum or constructor.

6.5 The Constructors

This section briefly describes each constructor and its role in the compiler. Each constructor corresponds to a portion of the Cool grammar. The order of arguments to a constructor follows the order in which symbols appear in productions in the Cool syntax specification in the manual. This correspondence between constructors and program syntax should make clear how to use the arguments of constructors. It may be helpful to read this section in conjunction with `cool-tree.aps`.

- `program`
  This constructor is applied at the end of parsing to the final list of classes. The only needed use of this constructor is already in the skeleton `cool.y`.

- `class_`
  This constructor builds a class node from two types and a list of features. See the examples above.

- `method`
  This is one of the two constructors in the `Feature` phylum. Use this constructor to build AST nodes for methods. Note that the second argument is a list of `Formals`.

- `attr`
  This is the constructor for attributes. The `init` field is for the expression that is the optional initialization.

- `formal`
  This is the constructor for formal parameters in method definitions. The field names are self-explanatory.

- `branch`
  This is the single constructor in the `Case` phylum. A branch of a `case` expression has the form

  ```
  name : typeid => expr;
  ```

  which corresponds to the field names in the obvious way. Use this constructor to build an AST for each branch of a `case` expression.

- `assign`
  This is the constructor for assignment expressions.

- `static_dispatch` and `dispatch`
  There are two different kinds of dispatch in Cool and they have distinct constructors. See the CoolAid for a discussion of static vs. normal dispatch. Note there is a shorthand for dispatch that omits the `self` parameter. Don't use the `no_expr` constructor in place of `self`; you need to fill in the symbol for `self` for the rest of the compiler to work correctly.

- `cond`
  This is the constructor for `if-then-else` expressions.

- `loop`
  This is the constructor for `loop-pool` expressions.

- `typcase`
  This constructor builds an AST for a `case` expression. Note that the second argument is a list of case branches (see the `branch` constructor above).

- `block`
  This is the constructor for $\{\ldots\}$ block expressions.

- `let`
  This is the constructor for `let` expressions. Note that the `let` constructor only allows one identifier. When parsing a `let` expression with multiple identifiers, it should be transformed into nested `lets` with single identifiers, as described in the semantics for `let` in the CoolAid.

- `plus`
  This is the constructor for $+$ expressions.

- `sub`
  This is the constructor for $-$ expressions.

- `mul`
  This is the constructor for $*$ expressions.

- `divide`
  This is the constructor for $/$ expressions.

- `neg`
  This is the constructor for ~ expressions.

- `lt`
  This is the constructor for < expressions.

- `eq`
  This is the constructor for = expressions.

- `leq`
  This is the constructor for <= expressions.

- `comp`
  This is the constructor for `not` expressions.

- `int_const`
  This is the constructor for integer constants.

- `bool_const`
  This is the constructor for boolean constants.

- `string_const`
  This is the constructor for string constants.

- `new_`
  This is the constructor for `new` expressions.

- `isvoid`
  This is the constructor for `isvoid` expressions.

- `no_expr`
  This constructor takes no arguments. Use `no_expr` where constructor arguments are missing because an optional expression is omitted, except for a missing `self` in a dispatch expression (see the discussion of dispatch above).

- `object`
  This constructor is for expressions that are just object identifiers. Note that object identifiers are used in many places in the syntax, but there is only one production for object identifiers as expressions.

6.6 Tips on Using the Tree Package

There are a few common errors people make using a tree package.

- The tree package implements an abstract data type. Violating the interface (e.g., by casting, pointer arithmetic, etc.) invites disaster. Stick to the interface as it is defined. When adding new members to the class declarations, be careful that those members do not perturb the interface for the existing functions.

- The value `NULL` is not a valid component of any AST. Never use `NULL` as an argument to a constructor. Use `nil_`*`phylum`* to create empty lists.

- All tree nodes and lists are distinct. A test such as

  ```
  if (x == nil_Expression()) { ... }
  ```

  is always false, because `nil_Expression()` creates a new empty list each time it is used. To check whether a list is empty, use the `len` method (see `tree.h`).

- It is also pointless to compare with childless tree nodes. For example,

  ```
  if (x == no_expr()) { ... }
  ```

  is always false, because `no_expr` creates a new AST each time it is called. Define a virtual method to determine the constructor of `x` (see Section **??**).

- The tree package functions perform checks to ensure that trees are used correctly. If something bad is detected, the function `fatal_error` is invoked to terminate execution. To determine where the problem lies it is best to use the debugger `dbx` or `gdb`. Set a breakpoint on `fatal_error` and use the `where` command to find out what routine caused the error.

7 The Runtime System

The runtime system consists of a set of hand-coded assembly language functions that are used as subroutines by Cool programs. Under `spim` the runtime system is in the file `trap.handler`. The use of the runtime system is a concern only for code generation, which must adhere to the interface provided by the runtime system.

The runtime system contains four classes of routines:

1. startup code, which invokes the main method of the main program;
2. the code for methods of predefined classes (**Object**, **IO**, **String**);
3. a few special procedures needed by Cool programs to test objects for equality and handle runtime errors;
4. the garbage collector.

The Cool runtime system is in the file ~cs164/lib/trap.handler; it is loaded automatically whenever spim/xspim is invoked. Comments in the file explain how the predefined functions are called.

The following sections describe what the Cool runtime system assumes about the generated code, and what the runtime system provides to the generated code. Read Section 13 of the CoolAid manual for a formal description of the execution semantics of Cool programs.

7.1 Object Header

The first three 32-bit words of each object are assumed to contain a class tag, the object size, and a pointer for dispatch information. In addition, the garbage collector requires that the word immediately before an object contain -1; this word is not part of the object.

Figure **??** shows the layout of a Cool object; the offsets are given in numbers of bytes. The garbage collection tag is -1. The class tag is a 32-bit integer identifying the class of the object. The runtime

offset -4	Garbage Collector Tag
offset 0	Class tag
offset 4	Object size (in 32-bit words)
offset 8	Dispatch pointer
offset 12...	Attributes

Figure 1: Object layout.

system uses the class tag in equality comparisons between objects of the basic classes and in the abort functions to index a table containing the name of each class.

The object size field and garbage collector tag are maintained by the runtime system; only the runtime system should create new objects. However, *prototype objects* (see below) must be coded directly by the code generator in the static data area, so the code generator should initialize the object size field and garbage collector tag of prototypes properly. Any statically generated objects must also initialize these fields.

The dispatch pointer is never actually used by the runtime system. Thus, the structure of dispatch information is not fixed. You should design the structure and use of the dispatch information for your code generator. In particular, the dispatch information should be used to invoke the correct method implementation on dynamic dispatches.

For **Int** objects, the only attribute is the 32-bit value of the integer. For Bool objects, the only attribute is the 32-bit value 1 or 0, representing either true or false. The first attribute of String objects is an object pointer to an Int object representing the size of the string. The actual sequence of ASCII characters of the string starts at the second attribute (offset 16), terminates with a 0, and is then padded with 0's to a word boundary.

The value *void* is a null pointer and is represented by the 32-bit value 0. All uninitialized variables (except variables of type Int, Bool, and String; see the *CoolAid*) are set to *void* by default.

7.2 Prototype Objects

The only way to allocate a new object in the heap is to use the `Object.copy` method. Thus, there must be an object of every class that can be copied. For each class X in the Cool program, the code generator should produce a skeleton X object in the data area; this object is the prototype of class X.

For each prototype object the garbage collection tag, class tag, object size, and dispatch information must be set correctly. For the basic classes `Int`, `Bool`, and `String`, the attributes should be set to the defaults specified in the *CoolAid*. For the other classes the attributes of the prototypes may be whatever you find convenient for your implementation.

7.3 Stack and Register Conventions

The primitive methods in the runtime system expect arguments in register **$a0** and on the stack. Usually **$a0** contains the **self** object of the dispatch. Additional arguments should be on top of the stack, first argument pushed first (an issue only for **String.substr**, which takes two arguments). Some of the primitive runtime procedures expect arguments in particular registers.

Figure **??** shows which registers are used by the runtime system. The runtime system may modify any of the scratch registers without restoring them (unless otherwise specified for a particular routine). The heap pointer is used to keep track of the next free word on the heap, and the limit pointer is used to

Scratch registers	**\$v0**,**\$v1**,**\$a0**–**\$a2**,**\$t0**–**\$t4**
Heap pointer	**\$gp**
Limit pointer	**\$s7**

Figure 2: Usage of registers by the runtime system.

Main_protObj	The prototype object of class **Main**	Data
Main_init	Code that initializes an object of class **Main** passed in **\$a0**	Code
Main.main	The main method for class **Main** **\$a0** contains the initial **Main** object	Code
Int_protObj	the prototype object of class **Int**	Data
Int_init	code that initializes an object of class **Int** passed in **\$a0**	Code
String_protObj	the prototype object of class **String**	Data
String_init	code initializing an object of class **String** passed in **\$a0**	Code
_int_tag	a single word containing the class tag for the **Int** class	Data
_bool_tag	a single word containing the class tag for the **Bool** class	Data
_string_tag	a single word containing the class tag for the **String** class	Data
class_nameTab	a table, which at index (class tag) $*$ 4 contains a pointer to a **String** object containing the name of the class associated with the class tag	Data
bool_const0	the **Bool** object representing the boolean value false	Data

Figure 3: Fixed labels.

keep track of where the heap ends. These two registers should not be modified or used by the generated code—they are maintained entirely in the runtime system. All other registers, apart from **\$at**,**\$sp**,**\$ra**, remain unmodified.

7.4 Labels Expected

The Cool runtime system refers to the fixed labels listed in Figure **??**. Each entry describes what the runtime system expects to find at a particular label and where (code/data segment) the label should appear.

There is no need for code that initializes an object of class **Bool** if the generated code contains definitions of both **Bool** objects in the static data area. The easiest way to ensure these labels are correct in the generated code is to adopt the following naming conventions:

`<class>_init`	for init code of class `<class>`
`<class>.<method>`	for method `<method>` code of class `<class>`
`<class>_protObj`	for the prototype object of class `<class>`

Finally, Figure **??** lists labels defined in the runtime system that are of interest to the generated code.

Object.copy	A procedure returning a fresh copy of the object passed in **$a0** Result will be in **$a0**
Object.abort	A procedure that prints out the class name of the object in **$a0** Terminates program execution
Object.type_name	Returns the name of the class of object passed in **$a0** as a string object Uses the class tag and the table class_nameTab
IO.out_string	The value of the string object on top of the stack is printed to the terminal. Does not modify **$a0**.
IO.out_int	The integer value of the Int object on top of the stack is printed to the terminal. Does not modify **$a0**.
IO.in_string	Reads a string from the terminal and returns the read string object in **$a0**. (The newline that terminates the input is not part of the string)
IO.in_int	Reads an integer from the terminal and returns the read int object in **$a0**.
String.length	Returns the integer object which is the length of the string object passed in **$a0**. Result in **$a0**.
String.concat	Returns a new string, made from concatenating the string object on top of the stack to the string object in **$a0**. Return value in **$a0**
String.substr	Returns the substring of the string object passed in **$a0**, from index i with length l. The length is defined by the integer object on top of the stack, and the index by the integer object on the stack below l. Result in **$a0**.
equality_test	Tests whether the objects passed in $t1 and $t2 have the same primitive type {Int,String,Bool} and the same value. If they do, the value in **$a0** is returned, otherwise $a1 is returned.
_dispatch_abort	Called when a dispatch is attempted on a void object. Prints the line number, from **$t1**, and filename, from **$a0**, at which the dispatch occurred, and aborts.
_case_abort	Should be called when a case statement has no match. The class name of the object in **$a0** is printed, and execution halts.
_case_abort2	Called when a case is attempted on a void object. Prints the line number, from **$t1**, and filename, from **$a0**, at which the dispatch occurred, and aborts.

Figure 4: Labels defined in the runtime system.

7.5 Execution Startup

On startup, the following things happen:

1. A fresh copy of the **Main** prototype object is made on the heap and then initialized by a call to **Main_init**.

   The code generator must define **Main_init**. **Main_init** should execute all initialization code of **Main**'s parent classes and finally execute the initializations of attributes in **Main** (if there are any).

2. Control is transferred to **Main.main**, passing a pointer to the newly created **Main** object in register **$a0**. Register **$ra** contains the return address.

3. If control returns from **Main.main**, execution halts with the message "COOL program successfully executed".

8 The Garbage Collector

The Cool runtime environment includes two different garbage collectors, a generational garbage collector and a stop-and-copy collector. The generational collector is the one used for programming assignments; the stop-and-copy collector is not currently used. The generational collector automatically scans memory for objects that may still be in use by the program and copies them into a new (and hopefully much smaller) area of memory.

Generated code must contain definitions specifying which of several possible configurations of the garbage collector the runtime system should use. The location **_MemMgr_INITIALIZER** should contain a pointer to an initialization routine for the garbage collector and **_MemMgr_COLLECTOR** should contain a pointer to code for a collector. The options aire no collection, generational collection, or stop-and-copy collection; see comments in the trap.handler for the names of the **INITIALIZER** and **COLLECTOR** routines for each case. If the location **_MemMgr_TEST** is non-zero and a garbage collector is enabled, the garbage collector is called on every memory allocation, which is useful for testing that garbage collection and code generation are working properly together.

The collectors assume every even value on the stack that is a valid heap address is a pointer to an object. Thus, a code generator must ensure that even heap addresses on the stack are in fact pointers to objects. Similarly, the collector assumes that any value in an object that is a valid heap address is a pointer to an object (the exceptions are objects of the basic classes, which are handled specially).

The collector updates registers automatically as part of a garbage collection. Which registers are updated is determined by a register mask that can be reset. The mask should have bits set for whichever registers hold heap addresses at the time a garbage collection is invoked; see the file trap.handler for details.

Generated code must notify the collector of every assignment to an attribute. The function `_GenGC_Assign` takes an updated address a in register `$a1` and records information about a that the garbage collector needs. If, for example, the attribute at offset 12 from the `$self` register is updated, then a correct code sequence is

```
sw      $x 12($self)
addiu   $a1 $self 12
jal     _GenGC_Assign
```

Calling `_GenGC_Assign` may cause a garbage collection. Note that if garbage collector is *not* being used it is equally important *not* to call `_GenGC_Assign`.