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 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 comments in the source.

The purpose of the Cool support code is to be easy to understand and use; the code is also written in Cool (where possible, and in Extended Cool elsewhere) which is limiting at times. 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.

2 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.aps`, with handcoded parts added. While the generated code is quite simple and regular in structure, it has few comments. This section serves as the documentation for the AST package.

2.1 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 `if` expressions, another class of `+` expressions, and so on. Objects of these classes are nodes in Cool abstract syntax trees. For example, an expression e_1 `+` e_2 is represented by a `+` expression object, which has two subtrees: one for the tree representing the expression e_1 and one for the tree representing the expression 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`, `plus`, etc.) are called *constructors*. (Don't confuse this use of the term "constructor" with C++/Java 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.

A phylum is a special kind of type; in APS a *phylum* is a type that is used to build syntax trees. Other types are simply values used for computation. Each constructor (node specification) is associated with a particular phylum. For example, the constructors for expression ASTs are distinguished from the constructors for class ASTs. The phyla are defined at the beginning of `Cool.aps`:

```
module COOL[] begin
  ...

  phylum Program;

  phylum Class;
  phylum Classes := SEQUENCE[Class];

  phylum Feature;
  phylum Features := SEQUENCE[Feature];

  phylum Formal;
  phylum Formals := SEQUENCE[Formal];

  phylum Expression;
  phylum Expressions := SEQUENCE[Expression];

  phylum Case;
  phylum Cases := SEQUENCE[Case];

  ...
```

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

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

```
  constructor class_decl(name : Symbol; parent: Symbol;
                         features : Features; filename : Symbol) : Class;
```

This declaration specifies that the `class_decl` constructor 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 sequence of `Feature` instances by the declaration

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

See Section 2.4 for a description of the operations defined on AST sequences.

2.2 Cool Realization of APS

The APS description of the AST is rendered as a set of Cool classes that are used by the compiler (written in Cool or Extended Cool) to manipulate the Cool program being compiled.

Each APS phylum is represented by a Cool class with the same name. This class extends `CoolNode`. Each attribute declared in APS is converted into a Cool attribute of the corresponding Cool class, with a getter and a setter (`get_...`, `set_...`).

Each APS constructor is represented by a Cool class whose name is the constructor name preceded by `C`, because the names of all Cool classes must start with capital letters. The Cool class extends the phylum's class and takes parameters corresponding exactly to parameters of the APS constructors. It has getters and setters for each of the parameters as well. (Warning: calling the setters is a mistake in CompSci 654/754.) Each Cool class corresponding to an APS constructor also implements the "accept" method to implement the VISITOR design pattern (q.v.).

Furthermore,the class `CoolNodeFactory` has a method for every constructor which construct an object of the corresponding class. This class is extended (indirectly) by the parse, and thus one can write code such as

```
$$ = class_decl($2,superclass_name,
                make_constructor($2,nil_Formals()),symbol(filename));
```

See Section 2.6 and `Cool.aps` to learn the definitions of the other constructors.[1]

There is a real danger of getting confused because the same names are used repeatedly for different entities in different contexts. In the grammar itself, we also have nonterminal `class` and terminal `CLASS`. 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.

2.3 Type case and Visitor design pattern

A compiler needs to examine values of ASTs. The AST package used in the Cool compiler has two main alternatives: the `match` constructor and the VISITOR design pattern.

For example, suppose we have a value `f` of type `Feature`, and we wish to treat it differently depending on whether it is a method or an attribute. Using "`match`" we can write:

```
f match {
  case m:Cmethod => ...
  case a:Cattr => ...
}
```

Inside the first case, we can use `m.get_formals()` to get the formals, etc. The advantage of the "match" technique is that everything is done in one method in the current context—we can refer to variables from outside the match through the magic of lexical scoping. We can have different methods both using "match" without any overlap or confusion.

If `f` is null, this code will raise an exception. If we omitted the `Cattr` case and `f` happened to be an attribute declaration, again an exception is raised. These exceptions can serve as important sanity checks. Don't put in a case unless that case is possible, and if you put in a case, make sure you do something sensible.

The alternate technique is to use the VISITOR design pattern. To do this, the current class must extend `CoolVisitor` directly or indirectly. Then, we simply write `f.accept(this)`. Then `f` will immediately all us back, either using `visit_method` or `visit_attr`. The current class should override these methods to do the action.

```
f.accept(this);
```

[1] Comments in `Cool.aps` begin with two hyphens "`--`".

```
    ...
  };

override
def visit_method(the_node:Cmethod,overridep:Boolean,name:Symbol,
                 formals:Formals,return_type:Symbol,expr:Expression) : Any = ...

override
def visit_attr(the_node:Cattr,name:Symbol,of_type:Symbol) : Any = ...
```

The benefit of the VISITOR design pattern is that each constructor's case gets its own method. The "visit" methods take the node and all the constructor arguments as parameters. Although this makes the method headers wordy, it avoids the need to call getters explicitly as was needed when using the "match" technique. A common mistake is to call the visit methods directly. Do not do this: call the `accept` methods, not the `visit_X` methods.

If you do not override the visit methods for the constructor, the original code will call the visit method for the phylum, which if not overridden will cause an error. Occasionally, one may wish to override the visit method for the phylum if there are a lot of cases that are handled the same way.

However do not override the visit method for the phylum and then do a "match" case analysis. This gets the worst of both "match" and the VISITOR design pattern. Code doing this indicates that the programmer doesn't understand how to use either technique.

2.4 AST Sequences

Sequence phyla have a distinct set of operations for constructing and accessing sequences. In Cool, for the APS phylum `Xs := SEQUENCE[X]` we generate four classes: the parent class `Xs` and three child classes: `Xs_nil`, `Xs_one` and `Xs_append`. The `Xs` class has the following features:

```
class Xs() extends CoolNode() {
  def elements() : XsEnumeration = ...;
  def size() : Int = ...;
  def nth(i : Int) : X = ...;
  def concat(l : Xs) : Xs = ...;
  def addcopy(e : X) : Xs = concat(new Xs_one(e));
}
```

These members are overridden as appropriate by the child classes. The meaning of the features is as follows:

elements() Return an enumeration of the elements;

size() Return the number of elements in the sequence;

nth(i) Return the i'th element (starting at 0) of the sequence;

concat(l) Return a new sequence appending l to the end of this sequence.

addcopy(e) A shorthand to add a new element to the end.

Using the enumeration is preferable to using an indexed loop since the `nth` method is inefficient. It also is cleaner when writing in Cool since Cool does not have "for" loops. One writes:

```
var enum : ExpressionsEnumeration = es.elements();
while (enum.has_next()) {
  var e : Expression = enum.next();
  ...
}
```

One may also use the VISITOR design pattern with sequences: calling `s.accept(this)` causes the sequence to call us back with `visit_Xs_one(.)` with each element in the sequence. If you wish to perform a further case analysis on the value with the VISITOR design pattern, the overriding of `visit_Xs_one(X x)` should simply call accept: `x.accept(this)`. Make sure you avoid the common mistake of calling a visit function directly!

There are no special shortcuts in `CoolNodeFactory` to create sequences, so one must use `new Xs_nil()` to create an empty sequence (or one of the other classes mentioned above) or `concat` or `addcopy` to build longer sequences.

2.5 The AST Class Hierarchy

All AST classes are derived from the class `CoolNode` which has the identity (used internally) and the line number (used for error messages). There is also an `accept` method for the VISITOR design pattern (q.v.).

Each of the phyla is a class derived directly from `CoolNode`. As stated previously, the phyla exist primarily to group related constructors together and as such do not add much new functionality. Phyla are the locus of attributes (q.v.) used in PA4 and on.

Each of the constructors is converted into a Cool class (with prefix `C`) derived from the appropriate phylum. The Cool class defines getters for each of the constructor parameters. The class `CoolNodeFactory` defines a method for each APS constructor that constructors a node and sets its ID and line number attributes.

2.6 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.aps`.

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

- `class_decl`
  This constructor builds a class node from two types and a sequence of features. If the superclass is "native" the superclass name is set to `'native` and the constructor must be native. 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 third argument is a sequence of `Formal`s. A native method (or constructor) has `no_expr()` as its entire body so that the code generator knows to omit generating code for it. If the code generator generated code for it, there would be

two definitions: the correct one in `cool-runtime.s` and the incorrect one generated by the code generator.

- `attr`
  This is the constructor for attributes. Native attributes are assigned the (impossible) type `native`. A class that has native attributes must perforce have a native constructor and cannot be inherited by other classes. There is no space for the initializer here; instead the implicit constructor will contain an assignment.

- `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

  ```
  case 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. A `null` branch has the name "null" and the type "Null."

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

- `static_dispatch` and `dispatch`
  Most method calls in Cool are normal dispatches. Note there is a shorthand for dispatch that omits the `this` parameter. Don't use the `no_expr` constructor in place of `this`; you need to fill in the symbol for `this` for the rest of the compiler to work correctly. The `static_dispatch` node is used for

  1. `super` calls. The node requires an object (use `this`) and a type name (use the name of the superclass of the current class).
  2. (optionally) super constructor calls.

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

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

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

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

- `let`
  This is the constructor for local variables. The "body" is the rest of the current block.

- `add`
  This is the constructor for `+` expressions.
- `sub`
  This is the constructor for `-` expressions.
- `mul`
  This is the constructor for `*` expressions.
- `div`
  This is the constructor for `/` expressions.
- `neg`
  This is the constructor for unary `-` expressions.
- `lt`
  This is the constructor for `<` expressions.
- `leq`
  This is the constructor for `<=` expressions.
- `comp`
  This is the constructor for `!` expressions.
- `int_lit`
  This is the constructor for integer literals.
- `bool_lit`
  This is the constructor for boolean literals.
- `string_lit`
  This is the constructor for string literals.
- `unit`
  This is the constructor for `()` expressions.
- `nil`
  This is the constructor for `null` expressions.
- `alloc`
  This is the constructor for the allocation part of `new` expressions; the dispatch part should be implemented using `dispatch`. Since the name of the constructor is the same as its class, `new Foo(null)` is parsed as:

  ```
  dispatch(alloc('Foo),'Foo,Expressions_one(nil()))
  ```

  (where `'` means that the following identifier should be seen as a symbol.)
- `no_expr`
  This constructor takes no arguments. Use `no_expr` for a native body.
- `variable`
  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.

2.7 Attributes

The file `Cool.aps` file not only declares all the phyla and constructors of the language, but also defines "attributes." These are values stored on nodes which are set during static semantic analysis and used in the code generation phase. For example:

```
attribute Program.boolean_class : remote Class := null;
```

This says that the program node has an attribute names `boolean_class` of type `Class` (`remote` is an APS artifact; it basically means that the class node is existing, not a newly created node).

Attributes can be accessed using `get_`*a* and `set_`*a* methods in the standard way.

2.8 Using the Visitor Framework

The tree package includes support for the VISITOR design pattern. The visitor design pattern is used to enable implicit case analysis on tree nodes. Any AST node can *accept* a visitor by calling the `accept` method with an object of a *visitor* class (`CoolVisitor` or a descendant). Frequently the visitor is the same object. The visitor overrides methods such as

```
override def visit_add(node : Cadd, e1 : Expression, e2 : Expression) : Any
```

This method will be called if the node accepting the visitor is indeed an "add" node. The `accept` call will return whatever the visitor method returns. The `Any` type is used because Cool does not have parametric polymorphism.

For example, consider the following code to check whether an expression is a literal:

```
class IsLiteral() extends CoolVisitor()
{
  def test(e : Expression) : Boolean =
    e.accept(this) match {
      case b : Boolean => b
    };

  // default:
  override def visit_Expression(n : Expression) : Any = false;

  override def visit_int_lit(n : Cint_lit, token : Symbol) : Any = true;
  override def visit_bool_lit(n : Cbool_lit, token : Boolean) : Any = true;
  override def visit_string_lit(n : Cstring_lit, token : Symbol) : Any = true;
  override def visit_nil(n : Cnil) : Any = true;
  override def visit_unit(n : Cunit) : Any = true;
}
```

The `test` method asks the node to accept itself. The node then identifies itself by calling the appropriate visitor. If (say) the node was an "add" node, the non-overridden version of the method in class `Visitor` would be called, which by default calls `visit_Expression`, which by default (but not here, since it is overridden) calls `visit_Node`, which by default aborts the program.

A simple visitor pattern thus is similar to (and somewhat clumsier than) a `match` block. The advantage is that the different branches are their own methods. If the branches need to do a large amount of work, then using a `match` expression will cause the surrounding method to get too big.

When a sequence node is asked to accept a visitor; it causes the visitor to be called with every element in the sequence; the method called is `visit_`*Xs*`_one(node:`*X*`)`. Here is code to count the number of attributes in a class declaration node:

```
class CountAttrs() extends CoolVisitor() {
  var count : Int = 0;

  def getNumAttrs(n : Cclass_decl) : Int = {
    count = 0;
    n.get_features().accept(this);
    count
  };

  override def visit_Features_one(n : Feature) : Any = n.accept(this);

  override def visit_Feature(f : Feature) : Any = ();

  override def visit_attr(n : Cattr, name : Symbol, ty : Symbol) : Any =
    count = count + 1;
}
```

Notice how the `visit_Features_one` method simply requests a further visit. We need to override `visit_Feature` (or `visit_method`, but this takes a lot of parameters) so that our code doesn't crash if the class has some methods.

Even this code is not terribly convincing for the utility of `Visitor` now that (as of Cool 2013), there are enumerations available:

```
  def getNumAttrs(n : Cclass_decl) : Int = {
    var count : Int = 0;
    var fe : FeaturesEnumeration = n.get_features().elements();
    while (fe.has_next()) fe.next() match {
      case a:Cattr => count = count + 1
      case f:Feature => ()
    };
    count
  };
```

The one place where visitors really show their benefit is when using `CoolTreeVisitor`. The latter class overrides all the visit methods to ask each of the children to accept it. As a result, by default, the tree visitor traverses an entire AST. Counting all the local variables *anywhere* in a tree is very easy:

```
class CountLocals() extends CoolTreeVisitor() {
  var count : Int = 0;

  def getNumLocals(n : CoolNode) : Int = {
    count = 0;
    n.accept(this);
    count
  };
```

```
  override def visit_let(n : Clet, name : Symbol, ty : Symbol,
                         init : Expression, body : Expression) : Any = {
    super.visit_let(n,name,ty,init,body);
    count = count + 1
  }
}
```

The visitor calls the superclass implementation so as to continue the counting in subtrees (especially `init` and `body`). The result value of the visit methods in unimportant in this case; as with sequence visiting, side-effects are used to collect information.

2.9 Tips on Using the Tree Package

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

- The value `null` is not a valid component of any AST. Never use `null` as an argument to a constructor. Use `new` *`phylum`*`_nil()` to create empty sequences.
- All tree nodes and sequences are distinct; the equality operator is true only for identical nodes. A test such as

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

  is always false, because `no_expr` creates a new AST each time it is called. Use pattern matching.
- The `nil` constructor for Cool's `null` expression never returns a null pointer.
- It is also pointless to compare with empty sequences: For example,

  ```
  if (x == new Expressions_nil()) { ... }
  ```

  To check whether a sequence is empty, don't use pattern matching (since the append of two empty sequences is empty itself). Use the `size()` method instead.

3 (Attributed) Abstract Syntax Trees

The Cool compiler can be run in stages, with the parser/scanner stage outputting an abstract syntax tree which is then read in by the semantic analysis stage which then prints the attributed tree onto output where it can be read by the code generator. This is done by using the `-P` options of the Cool compiler:

`$CLASSHOME/cmd/coolc -P` *phases filename*`.cool <` *input* `>` *output*

The phases is a "word" comprised of the following letters:

l run scanner (alone!)

p run scanner and parser together (no input required)

s run semantic analysis

c run code generation (no output generated)

a run the machine-independent optimizer

Unless one of the letters is "`p`" it will be necessary to give a possible attributed tree on input. Unless one of the letters is "`c`" the output will consist of a list of tokens (if the "`l`" letter is used) or an attributed tree (unless a syntax or semantic error is discovered). You can do multiple phases (`-P psc` is the same as regular cool compilation), but do not mix "`l`" with other phases. The letter "`a`" invokes the machine-independent optimizer which is one of the homework assignments for CompSci 854.

This section describes the format used to print abstract trees with or without attributes. The code used to print and read trees is in class `CoolTreeDumper` (and its immediate parent class `CoolAbstractTreeDumper`) and class `CoolTreeParser` (and its parent class `CoolAbstractTreeParser`). The class `CoolTreeDumper` is confusingly defined in `CoolTreeParser.cool` rather than having its own file.

When a node is printed, we follow the following format:

1. An indented line giving basic information and non-child arguments to the APS constructor:

   `@`*id* `=` *constructor-name*`:`*line-number argument ...*

   Each argument is followed by a space to make parsing easier. Symbols are printed with a prefixed tick (`'`), but spaces in the symbol are converted into `\s` so that spaces can be seen consistently as terminating values.

2. Then any non-default attribute is written: one per line with one greater level of indentation than the main node using the form

   `>`*name* `=` *value*

   For values which are node references, we use the form `@`*id* instead of printing the node (which could get recursive!)

3. Then each of the children nodes is printed (recursively) with one greater indentation. A sequence node is not explicitly printed: instead all the children of the sequence are printed with one greater indentation. The greater indentation is used by the parser to determine when the sequence is done.

Indentation is done in multiples of two. So indentation level of zero has no spaces, level 1 means two spaces, level 2 means four spaces, etc.

Here is an example of the start of an attributed tree:

```
@0 = program:0
  >any_class = @9
  >unit_class = @62
  >int_class = @77
  >boolean_class = @95
  >string_class = @168
    @9 = class_decl:24 'Any 'native 'basic.cool
      >inheritablep = true
```

```
@8 = method:24 false 'Any 'Any
  >owner = @9
  >feature_of_class = @9
  @7 = no_expr:24
    >of_type = 'native
```

The class declaration is indented *four* spaces because it is a member of a sequence. The same holds for the method node, but you can see that the `no_expr` node is indented only one level (two spaces) more than its enclosing method.

4 Optimization

The Cool compiler has some optimizations possible, controlled by the following flags:

-O Turn on machine-dependent optimization

-A ***options*** Pass options to the machine-independent optimizer.

5 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 `cool-runtime.s`. 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 native methods of predefined classes (`Any`, `IO`, `String`, `ArrayAny`, `Int`, `Boolean`, `Unit`, `Symbol`);
3. a few special procedures needed by Cool programs to handle runtime errors;
4. the garbage collector.

The Cool runtime system is in the file $CLASSHOME/lib/cool-runtime.s; 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.

5.1 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 part of the object although it precedes the location addressed by the pointer.

Figure 1 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 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.

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.

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 `Boolean` 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. For `ArrayAny` objects, the first attribute is a pointer to an `Int` object giving the number of entries in the array, then the actual entries follow afterward.

The value *null* is a null pointer and is represented by the 32-bit value 0. All attributes (except those of type `Int`, `Boolean`, and `Unit`; see the *CoolAid*) are set to *null* by default.

5.2 Prototype Objects

The only way to allocate a new object in the heap is to use the "secret" `Any.clone` 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. The attributes should be set to the default values for the specific type: for the basic classes `Int`, `Boolean`, and `Unit`, the attributes should be set to the defaults specified in the *CoolAid*. For the other classes, the default value is null (zero).

5.3 Stack and Register Conventions

The primitive methods in the runtime system expect arguments in register **$a0** and on the stack. Usually **$a0** contains the **this** object of the dispatch. Additional arguments should be on top of the stack, first argument pushed first . Some of the primitive runtime procedures expect arguments in particular registers.

Figure 2 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

Scratch registers	**$v0**,**$v1**,**$a0**–**$a2**,**$t0**–**$t2**
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.Main	The constructor for class `Main` **$a0** contains the initial `Main` object	Code
Int_protObj	the prototype object of class `Integer`	Data
Boolean_protObj	the prototype object of class `Boolean`	Data
String_protObj	the prototype object of class `String`	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
boolean_lit0	the `Boolean` object representing the boolean value false	Data
boolean_lit1	the `Boolean` object representing the boolean value true	Data

Figure 3: Fixed labels.

pointer is used to 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.

5.4 Labels Expected

The Cool runtime system refers to the fixed labels listed in Figure 3. Each entry describes what the runtime system expects to find at a particular label and where (code/data segment) the label should appear.

Use the following naming conventions for label generation:

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

Finally, Figure 4 lists labels defined in the runtime system that are of interest to the generated code.

5.5 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.Main**.

   The code generator must define **Main.Main**. **Main.Main** should execute the parent constructor and then execute its own body.

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

Any.Any	Immediately returns.
Any.clone	A procedure returning a fresh copy of the object passed in **$a0**. The result will be in **$a0**.
_dispatch_abort	Called when a dispatch is attempted on a null object. Prints the filename (`$a0`), line number (**$t1**), and aborts.
_divide_abort	Called when the program is about to do a bad divide: a division by zero, or MININT/-1. Prints the filename (`$a0`), line number (**$t1**), and aborts.
_case_abort	Called when a case statement has no match. First prints the filename (**$a1**), line number (**$t1**). Then it prints a message including the class name of the object (or `null`) in **$a0**, and execution halts.

Figure 4: Selected labels defined in the runtime system.

6 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 a initialization routine for the garbage collector and **_MemMgr_COLLECTOR** should contain a pointer to code for a collector. The options either no collection, generational collection, or stop-and-copy collection; see comments in the cool-runtime.s 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 assumes 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 cool-runtime.s 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 `$a1` and records information about a that the garbage collector needs. If, for example, the attribute at offset 12 from the `$s0` register is updated, then a correct code sequence is

```
sw      $x 12($s0)
addiu   $a1 $s0 12
```

```
jal     _GenGC_Assign
```

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