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 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.

[*]Copyright ©1995-2016 by Alex Aiken. All rights reserved.

2 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.

3 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`.

4 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.

5 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.

5.1 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 $\texttt{e}_1 + \texttt{e}_2$ is represented by a `+` expression object, which has two subtrees: one for the tree representing the expression $\texttt{e}_1$ and one for the tree representing the expression $\texttt{e}_2$.

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 5.2 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 5.5 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.

5.2 AST Lists

List phyla have a distinct set of operations for constructing and accessing lists.

For each phylum named *X* there is a phylum called *X*s (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:

[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 "– –".

```
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) . . . }
```

5.3 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_nodes`. 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_nodes`. 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.

5.4 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.

5.5 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 Cool Reference Manual 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 {...} 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 `let`s with single identifiers, as described in the semantics for `let` in the Cool Reference Manual.

- `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.

5.6 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 5.4).

- 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.