Programming Assignment 1: Frontend

1 Overview

Programming assignments 1-3 will direct you to design and build a compiler for Cool. Each assignment will cover one component of the compiler: frontend (lexical and syntactic analysis), semantic analysis, and backend (code generation and optimization). Each assignment will ultimately result in a working component, which can interface with other components. You will implement the compiler in Java.

We strongly recommend that you work in teams of 2 or 3 students. You must work in the same team on all three assignments PA1-PA3. Please sign-up for your team on QM+.

Documentation for Cool programming language and all the tools needed for building a compiler will be made available on the QM+ page. This includes a manual for Antlr4, as well as the manual for the spim simulator.

There is a lot of information in this handout, and you need to know most of it to write a working frontend. *Please read the handout thoroughly.*

2 Frontend

For this assignment, you are to write a frontend. It consists of a lexical analyzer (also called a scanner or lexer) and a syntactic analyzer (also called a parser). You will be using an *analyzer generator* called Antlr4.

The output of your frontend will be an abstract syntax tree (AST). You will construct this AST using Listener or Visitor automatically generated by the analyzer generator.

You must make some provision for graceful termination if a fatal error occurs. Uncaught exceptions are unacceptable.

3 Lexical Analysis

For lexical analyzer, you will describe the set of tokens for Cool in Antlr4 input format, and the analyzer generator will generate the actual Java code for recognizing tokens in Cool programs.

You should follow the specification of the lexical structure of Cool given in the Cool Reference Manual. Your scanner should be robust—it should work for any conceivable input. For example, you must handle errors such as an EOF occurring in the middle of a string or comment, as well as string constants that are too long. These are just some of the errors that can occur; see Cool Reference Manual for the rest.

Note that if the lexical specification is incomplete (some input has no regular expression that matches), then the generated scanner does undesirable things. *Make sure your specification is complete.*

For class identifiers, object identifiers, integers, and strings, the semantic value should be of type `Symbol`. For boolean constants, the semantic value is of type `java.lang.Boolean`. When a lexical error is encountered, the semantic value is the error message string. The lexemes for the other tokens do not carry any interesting information.

There is an issue in deciding how to handle the special identifiers for the basic classes (`Object`, `Int`, `Bool`, `String`), `SELF_TYPE`, and `self`. However, this issue doesn't actually come up until later phases of the compiler—the scanner should treat the special identifiers exactly like any other identifier.

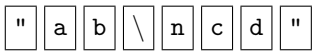
Do *not* test whether integer literals fit within the representation specified in the Cool Reference Manual—simply create a Symbol with the entire literal's text as its contents, regardless of its length.

Your lexer should maintain a variable that indicates which line in the source text is currently being scanned. This feature aids the parser in printing useful error messages. This variable is generated and maintained automatically by Antlr4.

Programs tend to have many occurrences of the same lexeme. For example, an identifier is generally referred to more than once in a program (or else it isn't very useful!). To save space and time, a common compiler practice is to store lexemes in a *string table*. We provide a string table implementation in Java. For the moment, you only need to know that the type of string table entries is `Symbol`.

3.1 Strings

Your scanner should convert escape characters in string constants to their correct values. For example, if the programmer types these eight characters:



your scanner would return the token `STR_CONST` whose semantic value is these 5 characters:

where  represents the literal ASCII character for newline.

Following specification on page 15 of the Cool Reference Manual, you must return an error for a string containing the literal null character. However, the sequence of two characters

is allowed but should be converted to the one character

.

4 Syntactic Analysis

You will need to refer to the syntactic structure of Cool, found in the Cool Reference Manual.

Your frontend should build an AST using the `ast` and `ast.visitor` packages provided. You will need this data structure for this and future assignments. The root (and only the root) of the AST should be of type `ProgramNode`. For programs that parse successfully, the output of `parser` is a listing of the AST.

Your parser need only work for programs contained in a single file—don't worry about compiling multiple files.

For programs that have errors, the output is the error messages of the parser. We have supplied you with an error reporting routine that prints error messages in a standard format; please do not modify it.

Since the mycoolc compiler uses pipes to communicate from one stage to the next, any extraneous characters produced by the parser can cause errors; in particular, the semantic analyzer may not be able to parse the AST your parser produces.

You may use precedence declarations, but only for expressions. Do not use precedence declarations blindly (i.e., do not respond to failure of Antlr to generate an analyzer from your grammar by adding precedence rules until it goes away).

The Cool `let` construct introduces an ambiguity into the language (try to construct an example if you are not convinced). Cool Reference Manual resolves the ambiguity by saying that a `let` expression extends as far to the right as possible.

5 Error Handling

The purpose of error handling is to permit the parser to continue after some anticipated error. It is not a panacea and the parser may become completely confused. You should use the error handling capabilities in Antlr4. See Antlr4 documentation for how best do it.

5.1 Lexical Error Handling

There are several requirements for reporting and recovering from lexical errors:

- When an invalid character (one that can't begin any token) is encountered, a string containing just that character should be returned as the error string. Resume lexing at the following character.
- If a string contains an unescaped newline, report that error as `‘‘Unterminated string constant’’` and resume lexing at the beginning of the next line—we assume the programmer simply forgot the close-quote.
- When a string is too long, report the error as `‘‘String constant too long’’` in the error string in the `ERROR` token. If the string contains invalid characters (i.e., the null character), report this as `‘‘String contains null character’’` or `‘‘String contains escaped null character’’`. In either case, lexing should resume after the end of the string. The end of the string is defined as either
  1. the beginning of the next line if an unescaped newline occurs after these errors are encountered; or
  2. after the closing `"` otherwise.
- If a comment remains open when EOF is encountered, report this error with the message `‘‘EOF in comment’’`. Do *not* tokenize the comment's contents simply because the terminator is missing. Similarly for strings, if an EOF is encountered before the close-quote, report this error as `‘‘EOF in string constant’’`.
- If you see "`*)`" outside a comment, report this error as `‘‘Unmatched *)’’`, rather than tokenzing it as `*` and `)`.

5.2 Syntactic Error Handling

To receive full credit, your parser should recover in at least the following situations:

- If there is an error in a class definition but the class is terminated properly and the next class is syntactically correct, the parser should be able to restart at the next class definition.
- Similarly, the parser should recover from errors in features (going on to the next feature), a `let` binding (going on to the next variable), and an expression inside a `{...}` block.

Do not be overly concerned about the the line numbers that appear in the error messages your parser generates. If your parser is working correctly, the line number will generally be the line where the error occurred. For erroneous constructs broken across multiple lines, the line number will probably be the last line of the construct.

In your README, describe which errors you attempt to catch. Your test file bad.cl should have some instances that illustrate the errors from which your parser can recover.

6 Testing

The first way to test your scanner is to generate sample inputs and run them using myfrontend -x, which prints out the line number and the lexeme of every token recognized by your scanner.

To test the parser, you will need a working scanner. Don't automatically assume that the scanner is bug free—latent bugs in the scanner may cause mysterious problems in the parser.

You will run your frontend using myfrontend. When you think your frontend is working, try running mycoolc to invoke your frontend together with all other compiler components which we provide. This will be a complete Cool compiler that you can try on any test programs.

You should test this compiler on both good and bad inputs to see if everything is working. Remember, bugs in your parser may manifest themselves anywhere. To run the testsuite, use testme lexer parser. Make sure to inspect the logs and detailed outputs generated by the script. Do not rely only on PASS/FAIL counts.

7 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 StringTable class provides methods for inserting and querying string tables in a variety of ways. It is implemented using a HashMap. Each entry in the string table stores a 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. Throughout the rest of the compiler (except parts of the code generator), a pointer to an entry in a string table 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 entries `x` and `y` represent the same string is simple. Note that it does not make sense to compare entries from different string tables, as these are guaranteed to be different even if the strings are the same. TreeConstants class uses string tables.

8 Abstract Syntax Trees

After lexical analysis and parsing, a Cool program is represented internally by the Cool compiler as an abstract syntax tree (AST).

The template code for this assignment includes the definition of AST data type (in package `ast`). The AST data type provides, for each kind of Cool construct, a class for representing constructs 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 template code also provides a convenient way to implement AST traversals using Visitor Design Patterns (in package `ast/visitors`). For example, `DumpVisitor` pretty-prints an AST. You will implement other visitors in PA1-PA2.

8.1 AST Class Hierarchy

All AST classes are derived from the class `TreeNode`. It has one field for the number of the line in which the construct corresponding to the AST node appeared in the source file. The line number is used by a Cool compiler to give good error messages.

Each class definition in `ast` package comes with a number of fields and getter methods for these fields. Fields are only visible to methods of that class or derived classes. All classes define `accept` methods for the Visitor Design Pattern.

Each class corresponds to a portion of the Cool grammar. The fields of each class correspond to non-terminals and terminals that appear in productions in the Cool syntax specification in the manual. This correspondence between AST data type and Cool program syntax should make clear how to use AST Classes.

A sample class definition is

```
public class ClassNode extends TreeNode {
    protected Symbol name;
    protected Symbol parent;
    protected List<FeatureNode> features = new LinkedList<FeatureNode>();
    protected Symbol filename;
    ...
}
```

An object of `ClassNode` is a node with four children: a `Symbol` (a type identifier) for the class name, a `Symbol` (another type identifier) for the parent class, a list of `FeatureNode`, and a `Symbol` for the filename in which the class definition occurs.

8.2 AST Classes

This section briefly describes each class and its role in the compiler. It may be helpful to read this section in conjunction with the code.

- **ProgramNode** This class is for the root of the AST. At the end of parsing, the root holds the final list of classes. The only needed use of this class is already in the template.
- **ClassNode** This class is for AST nodes for Cool classes. See the examples above.
- **MethodNode** This is one of the two classes derived from `FeatureNode`. Use this class to build AST nodes for methods. It holds method name, return type, list of formal parameters, and the body of the method.
- **AttributeNode** This is one of the two classes derived from `FeatureNode`. Use this class to build AST nodes for attributes. The `init` field is for the expression that is the optional initialization.
- **FormalNode** This class is for formal parameters in method definitions. The fields are name and declared type of the formal parameter.
- **BranchNode** Use this class to build an AST node for each branch of a `case` expression. A branch has the form

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

  which corresponds to the field names of `BranchNode` in the obvious way.
- **AssignNode** This is the class for assignment expressions.

- `StaticDispatchNode` and `DispatchNode` There are two different kinds of dispatch in Cool and they have distinct classes. See the Cool Reference Manual for a discussion of static vs. dynamic dispatch. Cool syntax has a shorthand for dispatch that omits the `self` parameter. Don't use `NoExpressionNode` in place of `self`; you need to fill in the symbol for `self` for the rest of the compiler to work correctly.
- **CondNode** This is the class for `if-then-else` expressions.
- **LoopNode** This is the class for `loop-pool` expressions.
- **CaseNode** This class builds an AST for a `case` expression. It holds a list of case branches. See `BranchNode` above.
- **BlockNode** This is the class for {...} block expressions.
- **LetNode** This is the class for `let` expressions. Note that the `LetNode` 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.
- **PlusNode** This is the class for + expressions.
- **SubNode** This is the class for − expressions.
- **MulNode** This is the class for ∗ expressions.
- **DivideNode** This is the class for / expressions.
- **NegNode** This is the class for ~ expressions.
- **LTNode** This is the class for < expressions.
- **EqNode** This is the class for = expressions.
- **LEqNode** This is the class for <= expressions.
- **CompNode** This is the class for `not` expressions.
- **IntConstNode** This is the class for integer constants.
- **BoolConstNode** This is the class for boolean constants.
- **StringConstNode** This is the class for string constants.
- **NewNode** This is the class for `new` expressions.
- **IsVoidNode** This is the class for `isvoid` expressions.
- **ObjectNode** This class is for expressions that are just object identifiers. Object identifiers are used in many places in the syntax, but there is only one production for object identifiers as expressions.
- **NoExpressionNode** Using this class for optional initialization expressions.

Getting and turning in the assignment

1. To set up your team's git repository, one of the team members types the following commands:

   ```
   cd ~
   mkdir cool
   cd cool
   git clone -o distro https://github.research.its.qmul.ac.uk/ecs652/distro.git
   cd  distro
   git remote add origin https://github.research.its.qmul.ac.uk/ecs652/<your-team-name>.git
   git push -u origin master
   git checkout -b frontend
   git push -u origin frontend
   ```

   For this assignment, ⟨`your-team-name`⟩ is the name assigned to your team/group on QM+ (for example, teamX).

   These commands create everything you need to start working on the compiler. They also set up a git repository that you should use to save your work, collaborate with your team members, and finally submit the assignment. All work on this assignment by all team members should be done and submitted in a branch called `frontend`.

   The above commands need to be executed **only once per team** *(not per every team member).*

2. To set up your local git repository from your team's repository, type

   ```
   git clone https://github.research.its.qmul.ac.uk/ecs652/<your-team-name>.git distro
   cd  distro
   git remote add distro https://github.research.its.qmul.ac.uk/ecs652/distro.git
   ```

   These commands are needed whenever you start working in a new setting (e.g., your laptop, your home directory on the school's network, but not on the machine where you performed the initial setup of team repository as explained above).

3. To start or continue working on this assignment, type

   ```
   git checkout frontend
   ```

   Before running this command, make sure you do not have any uncommitted local changes on your current branch, because they will carry over to your frontend branch.

   Use `git status` to check for uncommitted changes on the current branch. If there are any — either `git commit` or `git stash` them. This will become particularly important later on, when your team is working on several branches (for example, fixing PA1 on frontend branch and implementing PA2 on semant branch).

4. To get updates from your team's repo (e.g., changes made by other team members)

   ```
   git pull
   ```

   This may result in merge conflicts, which you need to resolve and eventually push to your team's repository.

5. To commit all your changes (locally):

   ```
   git add .
   git commit -a -m "meaningful commit message"
   ```

6. To push your changes to your team's repository (after one or more local commits):

   ```
   git push
   ```

   This command will fail if your team's repository has changed (e.g., by another team member). You will need to merge these changes using `git pull` as above before you can push your work upstream.

7. Add `bin` to the `PATH`. For example, on Linux with bash shell, type

   ```
   export PATH=~/cool/distro/bin:$PATH
   ```

   You need to execute this command every time you open a new shell. You can do it automatically by adding this command to your bash profile.

8. To compile and run the template code, type

   ```
   cd assignments/pa1
   buildme frontend
   ./myfrontend good.cl
   ```

   Try it now – it should work, and print some error messages about `token recognition error`, because the lexer is not implemented yet.

9. Follow the instructions in the README file.

10. The files that you will need to modify are: `CoolLexer.g4` and `CoolParser.g4`. You may also modify the following files: `ASTBuilder.java`, `TreeConstants.java`, `CoolErrorListener.java`, and `CoolErrorStrategy.java`.

    You should not edit any other files. In fact, if you modify anything in these files, you may find it impossible to complete the assignment.

11. To turn the assignment in, commit all your changes and push them to your team's repository:

    ```
    git add .
    git commit -m "Final version"
    git push origin frontend
    ```

    For marking, we will automatically use the latest commit on the frontend branch of your repository (regardless of the commit message) at the deadline.

12. If we publish changes (e.g., additional tests or bug fixes) on the shared repository called `distro`, the following commands should be used by one of the team members to integrate our updates into their team's repository.

    ```
    git checkout master
    git pull distro master
    git push origin master
    git checkout frontend
    git merge master
    # resolve merge conflicts if any and run "git commit"
    git push
    ```

    Then, all the other team members can get these changes using `git pull` as before.