Jim Lambers
CS143 Compilers
Summer Quarter 2000-01
Programming Project 3

This assignment must be submitted electronically by Wednesday, August 8.

1 Overview

In this project, you will create an type checker for the Mocha language. This checker will use the parser from Project 2 to obtain a syntax tree from a given input program and ensure that the program contains semantically valid operations and declarations.

2 Functional Specification

Your task is to implement the semantic actions described in Section 4 of the Mocha Specification. These actions prescribe the computation of synthesized and inherited attributes of grammar symbols in the parse tree, as well as checks that must be performed to determine semantic validity.

In addition to performing the required semantic checks, at the conclusion of parsing, you must display the attribute of the start symbol, *CompilationUnit* (no, I did not choose this hyphenation :). This attribute is a list of the class declarations included in the source file. Each class declaration, in turn, contains a list of each of its member declarations, as well as the name of its base class, if it has one. Each member declaration includes any modifiers attached to the member, as well as the member's type. If the member is a method, the declaration also includes the types of its arguments, and the declarations of any local variables.

3 Implementation

Most of your work will be included in the starter file `parser.y`. In this file, you must supply the code for the semantic actions associated with most productions, using the included comments and the Mocha Specification as a guide.

You will also need to supply the implementation of some functions in the other source files included with the assignment. These functions will be clearly marked by a comment beginning with the string "TODO:". The

necessary information to complete the implementation of these functions can also be found in the Mocha Specification, from the description of the Mocha type system and the rules for determining the type of the result of expressions involving unary and binary operators.

As for producing the output specified above, the actual display is very simple; we have provided `Print` methods for every kind of declaration. The task for you is to ensure that all of this information is correctly gathered and stored so that the output methods will perform as desired.

4 Notes and Suggestions

One key step of this project is assigning types to many of the terminals and non-terminals in the grammar. This is accomplished using the `%token` and `%type` directives, as described in the introduction to `bison` handout and illustrated in the `bison` sample listed there.

To specify a type of a grammar symbol, you should look among the fields in the given `%union` defining `YYSTYPE`. Once you find a field with the appropriate type, use the field's name in conjunction with the `%token` or `%type` directive.

For example, the non-terminals *Expression* and *Expression1*, among others, will use the `type` field, which is of type `MochaType *`:

```
%type<type> Expression Expression1
```

while the terminal `T_identifier` will use the `ident` field, which is of type `const char *`, and the terminal `T_boolliteral` will use the `boolConst` field, which is of type `bool`:

```
%token <ident>T_identifier <boolConst>T_boolliteral
```

You can find the appropriate fields for terminals by examining the lexical analyzer in `scanner.l` for references to `yylval`. For non-terminals, many will use the `type` field since most non-terminals store the type of an expression. The choice of field depends on what the attribute of the non-terminal is supposed to represent.

Some leeway is allowed on how you pass attributes around the parse tree; the key is that you perform all of the checking properly and reconstruct the declarations from the source program. In a number of cases you may find it useful to reach down into the parsing stack for an attribute value; if you do so, you must specify the field of `yylval` explicitly. For example, if you wish to use `$0` as an object of type pointer to `MochaType`, you write `$<type>0`.

You may also find it useful to add semantic actions in the middle of the right side of a production, to set attribute values that may later be inherited. Such actions count as symbols within the right side, affecting the indexing. Also, the symbol `$$` does not refer to the attribute of the left side in such an embedded action, but rather to the attribute of the embedded action itself. For example, consider the scheme

```
P : X  {$$ = $1;}  Y  {$$ = $3 + $2;}
  ;
Y : T_identifier  {$$ = $0;}
```

In the first production, `$2` refers to the attribute of the first action, following `X`. This attribute value is set by using `$$`, not `$2`. However, in a subsequent action, such as the one following `Y`, `$2` may be used, and since this second action is at the end of the right side, `$$` does in fact refer to the attribute of the non-terminal `P` on the left side. Note that the attribute of `Y` is obtained using the notation `$3`, since, counting actions, it is the third "symbol" on the right side. Also, note that in the second production, the attribute set in the embedded action in the first production is inherited using `$0`. A good exercise for you, to make sure you understand how these attributes work, is the following: assume that `X`, `Y`, and `P` above have attributes that are to be interpreted as numeric values x, y and p. What is the value of the synthesized attribute p in terms of x and y? Be careful!

It is recommended that you handle semantically correct input before performing error checking. Many of the test cases we will use are semantically correct Mocha programs, and the work you will do to create the output for these cases plays a key role in Project 4. Once you handle the semantically correct programs, then you should move on to handle the various errors.

The errors listed in the sample outputs are the only types of errors you need to report. The Mocha specification spells out the situations in which an error is to be reported. In some cases, more than one criterion needs to be checked, for different types of errors. The order in which the checks should take place is indicated in the comments in `parser.y`.

5 Testing

Once you are ready to test your type checker, simply type the command `make` at the prompt (assuming your project files are in your current working

directory) to compile your code. If there are no compiler or linker errors, the executable, named `pp3`, will be written to the current directory.

We have provided several sample inputs to help you test your scanner. These may be found in

```
/usr/class/cs143/assignments/pp3/samples
```

With each sample input, there are two corresponding outputs file. The first, with a `.out` extension, contains the output of the re-created declarations of the input program, and the second output file, with a `.err` extension, lists all of the semantic errors detected. This is the actual output generated by our solution, so you should ensure that your output matches ours. We strongly recommend that you create other input files for testing, as we will test your code on a larger collection of inputs.

To run the scanner with any of the samples, or any input files that you may create, simply use this command from the directory containing the executable `pp3`:

`(./pp3 <` *filename* `>` *outputfile*`) >&` *errorfile*

where *filename* is the name of the input file you are using, *outputfile* is the name of an output file of your choosing, and *errorfile* is the name of an error file of your choosing. When testing against the sample input files, you should then use the `diff` command to compare your output against the sample output files. Please make sure that you format your output in the same was as the solution, in order to facilitate grading. To that end, many of the functions used to display output have already been provided in the starter files.

6 Submission

This assignment must be submitted electronically by the given due date. You can find detailed information about the electronic submission process and our policy on late assignments on the course web site.

7 Grading

This assignment is worth 100 points, and counts for 15% of your overall grade in this course. We will grade this assignment by running your program against all of our input files, and comparing the output against the output generated by our solution using `diff`, and points will be deducted based on the differences that are detected.

NOTE: programs that do not compile, or cause segmentation faults when run, will only receive limited partial credit. We will not try to fix your code in these situations. Please do your best to make sure that your code works before you submit it.

8 Portability

While you have a choice as to where you may work on this assignment, you do not have a choice as to where it will be graded. Your program will be run on the AIR Solaris workstations, and must be submitted from there. Before you submit, please test on these workstations to ensure that your code will work just as well for us, where and when we grade it, as it did for you, wherever you wrote it.