CS 540 Fall 2022 Programming Assignment #3

Due date on blackboard

MinLisp and Type Checking

For this assignment, you will be using YACC and flex to create a parser and typechecker for MinLisp. Once you can parse MinLisp programs correctly, you will augment your parser with a symbol table and actions so that you are checking the type system of MinLisp, as described in this document and in the associated MinLisp specification. Work on each element of the task in sequence – don't move to the next part until you know the previous part works! If you are working in C or C++, all input will come from stdin; if in Java, input will come from a file (name on command line). In all cases, output must go to the screen (stdout).

Part I: Parsing MinLisp

Information about the lexical elements are given in the spec. The first step is to use flex/jflex to have it find and return the given tokens. You will also want to write a rule that allows you to count the line in the input file. Once you are confident that this is working, the next step is to use bison/byacc to write the parser itself.

The grammar for the language is given in the MinLisp specification. You shouldn't have to make changes from the grammar for this assignment but is okay if you do (as long as the changes you make do not change the langauge recognized).

If a syntax error is detected during parsing, YACC will terminate by default. This is fine; however, you need to create a yyerror() procedure so that at termination time, the line number where the error occurred can be printed. **Be sure that you can parse all of the given examples (and MinLisp code that you write before moving to the next step!**

Part II: Symbol Table for MinLisp

The first part of the typechecking is to create a symbol table structure to keep track of the symbols that are defined in the program. The goal is to have a way to be sure that the variables used in an input program have been declared (and are visible) and that variable names aren't duplicated in a single namespace. Named variables include function names (which are visible globally), array names (which are visible globally), parameters (which are only visible in the declared function) and identifiers in let statements (which are only visible in the expression associated with the **let** statement – which may in turn contain a let statement). These are the only places where new identifiers are created. Since we are only using a single pass on the input program, a function cannot be (legally) called until it has been declared. However, recursive calls (such as in the recursive add function below) are legal.

To accomplish this task, you will add actions to your YACC grammar from part I. You may be tempted to add actions to your lex file to put identifiers into your symbol table but this is not a good idea since you don't know in what context (definition or use) you are seeing the given token. So, send the lexeme to the parser (as described in class) and deal with the symbols appropriately there. To keep track of symbols encountered, you will have to implement a data structure that keeps track of the visible names for all currently active scopes during the parsing process. As the parsing process encounters a new scope, this information needs to be added and any identifiers declared are recorded in this scope (and checked to make sure they are not duplicates). The use of identifiers within a scope must be checked against the current symbol table to see if the name is valid. As a scope is exited, all information about that scope is discarded.

If, during the parse of a MinLisp program, you encountered a duplicate declaration or a variable that is not declared, a message about this should be printed out, along with the associated line number. Processing should continue (you don't have to do anything special for this to happen). Be sure your error messages are informative. For example, for this simple syntatically correct MinLisp program:

```
(define add (x y x)
  ( if (= y 0)
     x
     (incr (add x (- y 1)))
   )
)

(define incr(x) (+ x y))

( define main() (add x 2))
```

example error messages could be:
Line 1: Duplicate name x in this scope
Line 4: Undeclared function incr
Line 8: Undeclared variable y
Line 10: Undeclared variable x
Note: Generating incorrect messages is also an error. For example, in line 4, incr is unknown but add should be known and in your symbol table. Be sure that this part works before moving to part III.

Part III: Typechecking MinLisp –the type rules

Once you have a working symbol table, you will implement the type system given in the MinLisp document. There are lots of rules that have to be checked. Read the document carefully to locate these. Yes, there are other things you could check, but you don't have to.

When you find violations of the type rules, print out the line number and the problem (just as in part II) and continue processing. Try to avoid 'cascading errors' – the situation where one error creates a stream of errors for the same line; however, you will not be penalized for these errors since they are impossible to remove entirely. One approach to avoiding them is to give an incorrect expression the correct type (once you have printed out the error message), so that further processing will assume everything is fine. For example, if x + y is a type error because x is a function, still give the overall expression type integer.

There will be minlisp programs available but with and without type errors. However, there is no guarentee that these programs exercise all possible errors.

Submitting

Submit the assignment electronically. Be sure to submit all files needed to build your assignment.