COL226 Assignment 3 - Avval Amil - 2020CS10330

Overview

The submission consists of 8 files (including this README.md file) in a folder. There are 6 files used to make the lexer + parser and 1 file (program.while) which contains a program written in the WHILE programming language which can be changed at will to any syntactically correct/incorrect program in the same language for testing purposes. The other 6 files are as follows -
1) datatypes.sml - This contains all the datatypes defined to help in the parsing
2) glue.sml - This is just a piece of code to glue the files so that the lexer and parser work in conjunction properly
3) wast.cm - This file tells what files are to be used together in order for the abstract syntax tree to be created out of a program
4) wast.lex - This contains the ML-lex specifications of the lexer made and defined the tokens, keywords, valid characters and also has regular expressions to denote what kind of strings can be used for the identifiers and numerals. This file also has the implementation of a hash table to map keywords. 5) wast.yacc - This contains the ML-yacc specifications for the parser and defines what terminals are present, what non-terminals are present (along with their type as defined in the datatypes.sml file) and finally the context free grammar used for the parsing purposes.
6) while_ast.sml - This is the main file which contains the function which takes in a program of the WHILE programming language (which is basically a string) and gives an output of the AST type as defined in the datatypes.sml file.

Context Free Grammar

The context free grammar used is as follows -

begin: PROGRAMM IDE CONS block
block : declrlist LBRACE cmds RBRACE
declrlist : declr SCOLON declrlist
|
declr : VARR varlist COLON typ
typ : INTT
| BOOLL
varlist : variable varlistcom
varlistcom : COMMA variable varlistcom
|
cmds : cmdd SCOLON cmds
|
cmdd : variable ASSIGN intexp
| variable ASSIGN boolexp
| READD variable
| READD variable
| WRITEE intexp
| WRITEE boolexp
| IFF boolexp THENN LBRACE cmds RBRACE ELSEE LBRACE cmds RBRACE ENDIFF
| WHILEE boolexp DOO LBRACE cmds RBRACE ENDWHH
intexp : intterm PLUSS intexp
| intterm MINUSS intexp
| intterm
intterm : intfac MUL intterm
| intfac DIVV intterm
| intfac MODD intterm
| intfac
intfac : variable
| PLUSS NUM
| NEG NUM
| NUM
| NEG intfac
| LPAR intexp RPAR
boolexp : boolterm ORR boolexp
| boolterm
boolterm : boolfac ANDD boolterm
| boolfac
boolfac : TT
| FF
| variable
| comp
| LPAR boolexp RPAR
| NOTT boolfac
comp : intexp LTT intexp
| intexp LEQQ intexp
| intexp GTT intexp
| intexp GEQQ intexp
| intexp EQQ intexp
| intexp NEQQ intexp
| boolexp LTT boolexp
| boolexp LEQQ boolexp
| boolexp GTT boolexp
| boolexp GEQQ boolexp
| boolexp EQQ boolexp
| boolexp NEQQ boolexp
variable : IDE

All the symbols in all caps are terminals while are the lowercase symbols are non-terminals. Note that NUM and IDE terminals are not a fixed string like the other terminals, but are instead described by a regular expression each and stand for numeral and identifier respectively. The 'begin' nonterminal is the starting state. Also note that all hte nonterminals are such that their datatypes are of the same name as the non-terminals themselves e.g the non-terminal 'comp' is of type 'Comp', the non-terminal 'variable' is of type 'Variable' and so on.

AST Datatype Definition

The AST datatype along with other auxiliary datatypes are defined in the file 'datatypes.sml'. The datatypes are as follows:

type Variable = string
type Typ = string
datatype Ast = PROG of string * Block
and Block = BLK of Declr list * CmdSeq
and Declr = DEC of Variable list * Typ
and CmdSeq = empty | SEQ of Cmd * CmdSeq
and Cmd = SETINT of Variable * Intexp
| SETBOOL of Variable * Boolexp
| READINT of Variable
| READBOOL of Variable
| WRITEINT of Intexp
| WRITEBOOL of Boolexp
| ITE of Boolexp * CmdSeq * CmdSeq
| WH of Boolexp * CmdSeq
and Intexp = PLUS of Intterm * Intexp
| MINUS of Intterm * Intexp
| ITERM of Intterm
and Intterm = TIMES of Intfac * Intterm
| DIV of Intfac * Intterm
| MOD of Intfac * Intterm
| IFAC of Intfac
and Intfac = IVAR of Variable
| NUMB of string
| NEGVE of Intfac
| IEXP of Intexp
and Boolexp = OR of Boolterm * Boolexp
| BTERM of Boolterm
and Boolterm = AND of Boolfac * Boolterm
| BFAC of Boolfac
and Boolfac = TT
| FF
| BVAR of Variable
| COMP of Comp
| BEXP of Boolexp
| LOGNOT of Boolfac
and Comp = INEQ of Intexp * Intexp
| ILT of Intexp * Intexp
| ILEQ of Intexp * Intexp
| IGT of Intexp * Intexp
| IGEQ of Intexp * Intexp
| IEQ of Intexp * Intexp
| BNEQ of Boolexp * Boolexp
| BLT of Boolexp * Boolexp
| BLEQ of Boolexp * Boolexp
| BGT of Boolexp * Boolexp
| BGEQ of Boolexp * Boolexp
| BEQ of Boolexp * Boolexp

Note that many constructors have been used (with very suggestive names) that were not specified in the assignment but which I felt that added to the ease of interpretation of the production rules and datatypes.

Auxiliary Functions and Data

I have defined a structure KeyWord structure in the 'wast.lex' file which will maintain a hash table to see if a string acting as an identifier in the program is a keyword or not. This is necessary as keywords cannot be used as variable names. This is inspired from the KeyWord structure in ML-Lex and Yacc User Guide.

Other Design Decisions

I have used many constructors functions to help in the parsing. These constructors are present in the datatypes specified above and are suggestive in nature e.g the constructor LOGNOT takes an input a boolfactor and is used when the logical not is taken of a boolean factor, the constructor INEQ is used when the not equal to relational operator is used on two integer expressions and so on.

Syntax Directed Translation

The semantic rules along with their respective productions are present in the implementation-

begin: PROGRAMM IDE CONS block (PROG(IDE,block))
block : declrlist LBRACE cmds RBRACE (BLK(declrlist, cmds))
declrlist : declr SCOLON declrlist (declr::declrlist)
| ([])
declr : VARR varlist COLON typ (DEC(varlist,typ))
typ : INTT ("int")
| BOOLL ("bool")
varlist : variable varlistcom (variable :: varlistcom)
varlistcom : COMMA variable varlistcom (variable :: varlistcom)
| ([])
cmds : cmdd SCOLON cmds (SEQ(cmdd,cmds))
| (empty)

```
cmdd : variable ASSIGN intexp (SETINT(variable, intexp))
| variable ASSIGN boolexp (SETBOOL(variable,boolexp))
| READD variable (READINT(variable))
| READD variable (READBOOL(variable))
| WRITEE intexp (WRITEINT(intexp))
| WRITEE boolexp (WRITEBOOL(boolexp))
| IFF boolexp THENN LBRACE cmds RBRACE ELSEE LBRACE cmds RBRACE ENDIFF (ITE(boolexp, cmds1, cmds2))
| WHILEE boolexp DOO LBRACE cmds RBRACE ENDWHH (WH(boolexp,cmds))
intexp : intterm PLUSS intexp (PLUS(intterm,intexp))
| intterm MINUSS intexp (MINUS(intterm,intexp))
| intterm (ITERM(intterm))
intterm : intfac MUL intterm (TIMES(intfac,intterm))
| intfac DIVV intterm (DIV(intfac,intterm))
| intfac MODD intterm (MOD(intfac,intterm))
| intfac (IFAC(intfac))
intfac : variable (IVAR(variable))
| PLUSS NUM (NUMB("+"^NUM))
| NEG NUM (NUMB("~"^NUM))
| NUM (NUMB(NUM))
| NEG intfac (NEGVE(intfac))
| LPAR intexp RPAR (IEXP(intexp))
boolexp : boolterm ORR boolexp (OR(boolterm,boolexp))
| boolterm (BTERM(boolterm))
boolterm : boolfac ANDD boolterm (AND(boolfac,boolterm))
| boolfac (BFAC(boolfac))
boolfac : TT (TT)
| FF (FF)
| variable (BVAR(variable))
| comp (COMP(comp))
| LPAR boolexp RPAR (BEXP(boolexp))
| NOTT boolfac (LOGNOT(boolfac))
comp : intexp LTT intexp (ILT(intexp1,intexp2))
| intexp LEQQ intexp (ILEQ(intexp1,intexp2))
| intexp GTT intexp (IGT(intexp1,intexp2))
| intexp GEQQ intexp (IGEQ(intexp1,intexp2))
| intexp EQQ intexp (IEQ(intexp1,intexp2))
| intexp NEQQ intexp (INEQ(intexp1,intexp2))
| boolexp LTT boolexp (BLT(boolexp1,boolexp2))
| boolexp LEQQ boolexp (BLEQ(boolexp1,boolexp2))
| boolexp GTT boolexp (BGT(boolexp1,boolexp2))
| boolexp GEQQ boolexp (BGEQ(boolexp1,boolexp2))
| boolexp EQQ boolexp (BEQ(boolexp1,boolexp2))
| boolexp NEQQ boolexp (BNEQ(boolexp1,boolexp2))
variable : IDE (IDE)
```

This results in the input string being converted into an object of the type AST as defined in 'datatypes.sml'. The semantic rules are enclosed in parentheses.

How to Run

Download the submission folder. Then open the directory in the terminal. Type 'sml' to open the sml interpreter. Next, type -

CM.make "wast.cm"

to make the appropriate files for the lexing and parsing. After this, type -

Control.Print.printDepth:=50

so that the AST is visible properly (any sufficiently large number may be used, 50 is just an arbitrary number). Next, type

Wast.compile "program.while"

to compile the WHILE program in the file program.while and create the AST. If your program is in some other file. Then put that file in the same directory as the other files of the submission and the "program.while" can be replaced with the name of that file. I have included a sample program which is free of any syntactic errors and should give a sample AST when the above commands are used.

Acknowledgements

1) http://rogerprice.org/ug/ug.pdf - The syntax for ML-lex, ML-yacc and hash table for keywords was based off this guide.
2) http://www.smlnj.org/doc/ML-Yacc/ - Referred this to understand ML-Yacc
3) https://www.smlnj.org/doc/ML-Lex/manual.html - Reference for ML-Lex