CS3276 Compilers

Project #4 Grade Sheet

Name: \_Arturo Perez\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Score: \_94/100\_\_

1. **Proper submission of the project** (10 points)
   1. Submission of tig\_parse.y plus supporting files for executable (5 points): \_\_\_\_\_\_\_\_\_\_\_
   2. All files compile without warning messages (5 points): \_\_\_\_\_\_\_\_\_\_\_
2. **Proper execution** (75 points)

Compiler builds a correct AST for all syntactically correct tiger programs (75 points):\_\_\_-6\_\_\_\_

* When decs is epsilon, incorrectly creates a DecList with two nullptrs.
* Reclist incorrectly places the EFieldList generated by reclist2 in another EFieldList
* In the lvalue case for arrays, you don’t need to worry about checking if lvalue is a SimpleVar

1. **Documentation** (5 points)
   1. README file with program description and instructions (5 points): \_\_\_\_\_\_\_\_\_\_\_
2. **Programming style** (10 points): \_\_\_\_\_\_\_\_\_\_\_

/\*\*\*

Name: Arturo Perez

Userid: perezae

Email: arturo.e.perez@vanderbilt.edu

Descrption: This program will construct an AST for the Tiger langauge.

Last modified: 26 March 2018

\*\*\*/

/\* this is the definition section of the grammar file. \*/

/\* here is where you place your includes, function prototypes, \*/

/\* and Yacc/Bison special operators. \*/

%{

#include <cstdio>

#include <string>

#include "util.h"

#include "errormsg.h"

#include "symbol.h"

#include "absyn.h"

using namespace std;

absyn::Exp\* absyn\_root;

int yylex(void); /\* function prototype \*/

void yyerror(char \*s);

/\* test that an exp is really an lvalue \*/

const absyn::Var\* really\_lvalue(absyn::Exp\* really\_an\_lvalue);

/\* test that an lvalue is really a simple ID that could be a typename \*/

S\_symbol really\_typename(absyn::Var\* really\_a\_typename);

%}

%union {

int pos;

int ival;

string \*sval;

absyn::Var\* var;

absyn::Exp\* exp;

absyn::Dec\* dec;

absyn::Ty\* ty;

absyn::Field\* field;

absyn::FieldList\* fieldList;

absyn::ExpList\* expList;

absyn::FunDec\* fundec;

absyn::FunDecList\* fundecList;

absyn::DecList\* decList;

absyn::Namety\* namety;

absyn::NametyList\* nametyList;

absyn::EFieldList\* efieldList;

}

%token <sval> ID STRING

%token <ival> INT

%token

COMMA COLON SEMICOLON LPAREN RPAREN LBRACK RBRACK

LBRACE RBRACE DOT

PLUS MINUS TIMES DIVIDE EQ NEQ LT LE GT GE

AND OR ASSIGN

ARRAY IF THEN ELSE WHILE FOR TO DO LET IN END OF

BREAK NIL

FUNCTION VAR TYPE

/\* THEN/ELSE stuff taken from page 234 of O'Reilly book \*/

%nonassoc THEN

%nonassoc ELSE STMT

%nonassoc ASSIGN

%nonassoc ARRAY\_LITERAL

%left OR

%left AND

%nonassoc EQ NEQ GT LT GE LE

%left PLUS MINUS

%left TIMES DIVIDE

%left UMINUS

%nonassoc LBRACK LPAREN LBRACE DOT

%type <exp> exp program expseq seq

%type <expList> list arglist arglist2

%type <var> lvalue

%type <efieldList> reclist reclist2

%type <dec> dec vardec

%type <decList> decs

%type <nametyList> tydeclist

%type <namety> tydec

%type <fieldList> tyfields fieldseq

%type <field> tyfield

%type <fundecList> fundeclist

%type <fundec> fundec

%type <ty> ty

%start program

/\* The next section is the rule section of the grammar file. \*/

/\* That is where you will place your grammar for the Tiger \*/

/\* language. \*/

/\* For project #4, you will need to add semantic actions that \*/

/\* build an abstract syntax tree. \*/

%%

program: exp { absyn\_root = $1 ; }

exp: INT { $$ = new absyn::IntExp(EM\_tokPos, $1) ; }

| STRING { $$ = new absyn::StringExp(EM\_tokPos, Appel\_String($1)) ; }

| BREAK { $$ = new absyn::BreakExp(EM\_tokPos) ; }

| lvalue { $$ = new absyn::VarExp(EM\_tokPos,$1) ; }

| NIL { $$ = new absyn::NilExp(EM\_tokPos) ; }

| exp PLUS exp { $$ = new absyn::OpExp(EM\_tokPos, absyn::OpExp::EXP\_PLUS, $1, $3) ; }

| exp MINUS exp { $$ = new absyn::OpExp(EM\_tokPos, absyn::OpExp::EXP\_MINUS, $1, $3) ; }

| exp TIMES exp { $$ = new absyn::OpExp(EM\_tokPos, absyn::OpExp::EXP\_MUL, $1, $3) ; }

| exp DIVIDE exp { $$ = new absyn::OpExp(EM\_tokPos, absyn::OpExp::EXP\_DIV, $1, $3) ; }

| MINUS exp %prec UMINUS { $$ = new absyn::OpExp(EM\_tokPos, absyn::OpExp::EXP\_MINUS, new absyn::IntExp(EM\_tokPos,0), $2) ; }

| exp EQ exp { $$ = new absyn::OpExp(EM\_tokPos, absyn::OpExp::EXP\_EQ, $1, $3) ; }

| exp NEQ exp { $$ = new absyn::OpExp(EM\_tokPos, absyn::OpExp::EXP\_NE, $1, $3) ; }

| exp GT exp { $$ = new absyn::OpExp(EM\_tokPos, absyn::OpExp::EXP\_GT, $1, $3) ; }

| exp LT exp { $$ = new absyn::OpExp(EM\_tokPos, absyn::OpExp::EXP\_LT, $1, $3) ; }

| exp GE exp { $$ = new absyn::OpExp(EM\_tokPos, absyn::OpExp::EXP\_GE, $1, $3) ; }

| exp LE exp { $$ = new absyn::OpExp(EM\_tokPos, absyn::OpExp::EXP\_LE, $1, $3) ; }

| exp AND exp { $$ = new absyn::IfExp(EM\_tokPos, $1, $3, new absyn::IntExp(EM\_tokPos,0)) ; }

| exp OR exp { $$ = new absyn::IfExp(EM\_tokPos, $1, new absyn::IntExp(EM\_tokPos,1), $3) ; }

| lvalue ASSIGN exp { $$ = new absyn::AssignExp(EM\_tokPos, $1, $3) ; }

| IF exp THEN exp { $$ = new absyn::IfExp(EM\_tokPos, $2, $4, nullptr) ; }

| IF exp THEN exp ELSE exp { $$ = new absyn::IfExp(EM\_tokPos, $2, $4, $6) ; }

| WHILE exp DO exp %prec STMT { $$ = new absyn::WhileExp(EM\_tokPos, $2, $4) ; }

| FOR ID ASSIGN exp TO exp DO exp %prec STMT { $$ = new absyn::ForExp(EM\_tokPos, S\_Symbol($2), $4, $6, $8) ; }

| LPAREN RPAREN { $$ = new absyn::SeqExp(EM\_tokPos, nullptr) ; }

| LPAREN exp RPAREN { $$ = $2 ; }

| LPAREN seq RPAREN { $$ = $2 ; }

| lvalue LBRACK exp RBRACK OF exp %prec ARRAY\_LITERAL { $$ = new absyn::ArrayExp(EM\_tokPos, really\_typename($1), $3, $6) ; }

// NOTE: used 'lvalue' rather than 'ID' to avoid a conflict

// (see hint #6 from project #3).

// Must insure that the lvalue is really just an ID

| ID LPAREN arglist RPAREN { $$ = new absyn::CallExp(EM\_tokPos,S\_Symbol($1), $3) ; }

| ID LBRACE reclist RBRACE { $$ = new absyn::RecordExp(EM\_tokPos, S\_Symbol($1), $3) ; }

| LET decs IN expseq END { $$ = new absyn::LetExp(EM\_tokPos, $2, $4 ) ; }

decs: /\* epsilon \*/ { $$ = new absyn::DecList(nullptr, nullptr) ; }

| dec decs { $$ = new absyn::DecList($1, $2); }

dec: tydeclist { $$ = new absyn::TypeDec(EM\_tokPos, $1) ; }

| vardec { $$ = $1 ; }

| fundeclist { $$ = new absyn::FunctionDec(EM\_tokPos, $1); }

// The following two nonterminals each introduce a shift/reduce conflict,

// since we can parse two types as a single list (by shifting) or reduce

// each to a separate list and the parse the two lists as two decs.

// We want to do the former, which corresponds to a shift, so its ok.

tydeclist: tydec tydeclist { $$ = new absyn::NametyList($1, $2) ; }

| tydec { $$ = new absyn::NametyList($1, nullptr) ; }

fundeclist: fundec fundeclist { $$ = new absyn::FunDecList($1, $2) ; }

| fundec { $$ = new absyn::FunDecList($1, nullptr) ; }

tydec: TYPE ID EQ ty { $$ = new absyn::Namety(S\_Symbol($2), $4) ; }

ty: ID { $$ = new absyn::NameTy(EM\_tokPos, S\_Symbol($1)) ; }

| LBRACE tyfields RBRACE { $$ = new absyn::RecordTy(EM\_tokPos, $2) ; }

| ARRAY OF ID { $$ = new absyn::ArrayTy(EM\_tokPos, S\_Symbol($3)) ; }

tyfields: /\*epsilon\*/ { $$ = nullptr ; }

| fieldseq { $$ = $1 ; }

fieldseq: tyfield { $$ = new absyn::FieldList($1, nullptr) ; }

| tyfield COMMA fieldseq { $$ = new absyn::FieldList($1, $3) ; }

tyfield: ID COLON ID { $$ = new absyn::Field(EM\_tokPos, S\_Symbol($1), S\_Symbol($3)) ; }

vardec: VAR ID ASSIGN exp { $$ = new absyn::VarDec(EM\_tokPos, S\_Symbol($2), nullptr, $4) ; }

| VAR ID COLON ID ASSIGN exp { $$ = new absyn::VarDec(EM\_tokPos, S\_Symbol($2), S\_Symbol($4), $6) ; }

fundec: FUNCTION ID LPAREN tyfields RPAREN EQ exp { $$ = new absyn::FunDec(EM\_tokPos, S\_Symbol($2), $4, nullptr, $7) ; }

| FUNCTION ID LPAREN tyfields RPAREN COLON ID EQ exp { $$ = new absyn::FunDec(EM\_tokPos, S\_Symbol($2), $4, S\_Symbol($7), $9) ; }

expseq: /\*epsilon\*/ { $$ = new absyn::SeqExp(EM\_tokPos, nullptr) ; }

| list { $$ = new absyn::SeqExp(EM\_tokPos, $1) ; }

list: exp { $$ = new absyn::ExpList($1, nullptr) ; }

| exp SEMICOLON list { $$ = new absyn::ExpList($1, $3) ; }

seq: exp SEMICOLON list { $$ = new absyn::SeqExp(EM\_tokPos, new absyn::ExpList($1, $3)) ; }

reclist: /\*epsilon\*/ { $$ = nullptr ; }

| reclist2 { $$ = new absyn::EFieldList(nullptr, $1) ; }

reclist2: ID EQ exp { $$ = new absyn::EFieldList(new absyn::EField(S\_Symbol($1), $3), nullptr) ; }

| ID EQ exp COMMA reclist2 { $$ = new absyn::EFieldList(new absyn::EField(S\_Symbol($1), $3), $5) ; }

arglist: /\*epsilon\*/ { $$ = nullptr ; }

| arglist2 { $$ = $1 ; }

arglist2: exp { $$ = new absyn::ExpList($1, nullptr) ; }

| exp COMMA arglist2 { $$ = new absyn::ExpList($1, $3) ; }

lvalue: ID {$$ = new absyn::SimpleVar(EM\_tokPos,S\_Symbol($1)) ; }

| lvalue DOT ID { $$ = new absyn::FieldVar(EM\_tokPos, $1, S\_Symbol($3)) ; }

| lvalue LBRACK exp RBRACK { $$ = new absyn::SubscriptVar(EM\_tokPos, new absyn::SimpleVar(EM\_tokPos, really\_typename($1)), $3) ; }

%%

/\* this section is for user defined subroutines \*/

void yyerror(char \*s)

{

EM\_error(EM\_tokPos, "%s", s);

}

/\* test that an exp is really an lvalue \*/

const absyn::Var\* really\_lvalue(absyn::Exp\* really\_an\_lvalue)

{

if (dynamic\_cast<absyn::VarExp\*>(really\_an\_lvalue) == NULL)

{

EM\_error(really\_an\_lvalue->getPos(), "%s", "illegal lvalue");

return nullptr;

}

else

{

return dynamic\_cast<absyn::VarExp\*>(really\_an\_lvalue)->getVar();

}

}

/\* test that an lvalue is really a simple ID that could be a typename \*/

S\_symbol really\_typename(absyn::Var\* really\_a\_typename)

{

if (dynamic\_cast<absyn::SimpleVar\*>(really\_a\_typename) == NULL) {

EM\_error(really\_a\_typename->getPos(), "%s", "Illegal type name");

return S\_Symbol("<ERROR--Bad Type Name>");

} else {

return dynamic\_cast<absyn::SimpleVar\*>(really\_a\_typename)->getSymbol();

}

}

Arturo Perez

Dr. Roth

Compilers 3276

I did not receive any unauthorized help on this assignment.

26 March 2018

In this project we incorporated semantic action into our parser so we can create Abstract Syntax Trees

for the Tiger language. Every Tiger rule now has a corresponding semantic action. The assignment was

tough at first to understand, but once you got the hand of matching constructors and types, it became

more like a matching game.