1 Overview of the Programming Project

Programming phase I–IV will result to design and build a compiler for Cool. Each phase will cover one component of the compiler: lexical analysis, parsing, semantic analysis, and code generation. Each phase will ultimately result in a working compiler phase which can interface with other phases.

We are doing our projects in C++.

**For the phase 1**, we are to write a lexical analyzer, also called a scanner, using a lexical analyser generator. (The C++ tool is called flex; the Java tool is called jlex.) we will describe the set of tokens for Cool in an appropriate input format, and the analyzer generator will generate the actual code (C++or Java) for recognizing tokens in Cool programs.

**For the phase 2** we will write a parser for Cool. The assignment makes use of two tools: the parser generator (the C++ tool is called bison; the Java tool is called CUP) and a package for manipulating trees. The output of your parser will be an abstract syntax tree (AST). we will construct this AST using semantic actions of the parser generator. The reader will need to refer to the syntactic structure of Cool, found in Figure 1 of The Cool Reference Manual (as well as other parts). The documentation for bison and CUP is available online.

**//output ( SCREEN SHOTS)**

**attached to the rar**

**//instructions to run different files**

cool.flex is a skeleton file for the specification of the lexical analyzer. You should complete it with your regular expressions, patterns and actions.

test.cl is a COOL program that you can test the lexical analyzer on. It contains some errors, so it won't compile with coolc. However, test.cl does not exercise all lexical constructs of COOL and part of your assignment is to rewrite test.cl with a complete set of tests for your lexical analyzer.

cool-parse.h contains definitions that are used by almost all parts of the compiler. DO NOT MODIFY.

stringtab.{cc|h} and stringtab\_functions.h contains functions to manipulate the string tables. DO NOT MODIFY.

utilities.{cc|h} contains functions used by the main() part of the lextest program. You may want to use the strdup() function defined in here. Remember that you should not print anything from inside cool.flex! DO NOT MODIFY.

lextest.cc contains the main function which will call your lexer and print out the tokens that it returns. DO NOT MODIFY.

mycoolc is a shell script that glues together the phases of the compiler using Unix pipes instead of statically linking code. While inefficient, this architecture makes it easy to mix and match the components you write with those of the course compiler. DO NOT MODIFY.

cool-lexer.cc is the scanner generated by flex from cool.flex. DO NOT MODIFY IT, as your changes will be overritten the next time you run flex.

The \*.d files are automatically generated Makefiles that capture dependencies between source and header files in this directory. These files are updated automatically by Makefile; see the make documentation for a detailed explanation.

Instructions ------------

To compile your lextest program type:

% make lexer

Run your lexer by putting your test input in a file 'foo.cl' and run the lextest program:

% ./lexer foo.cl

To run your lexer on the file test.cl type:

% make dotest

If you think your lexical analyzer is correct and behaves like the one we wrote, you can actually try 'mycoolc' and see whether it runs and produces correct code for any examples. If your lexical analyzer behaves in an unexpected manner, you may get errors anywhere, i.e. during parsing, during semantic analysis, during code generation or only when you run the produced code on spim. So beware.

**Instructions**

**------------**

**To compile your parser program type:**

% make parser

This produces an executable named "parser" which is standalone

phase of the Cool compiler. It requires lexer, semant, and cgen

to do anything useful.

To test your parser on a file 'foo.cl' type

% myparser foo.cl

myparser is a shell script that "glues" together lexer and

parser using pipes.

To run your parser on the files good.cl and bad.cl type:

% make dotest

To run the (provided) lexer and your parser on a file called test.cl type:

% ./lexer test.cl | ./parser

If you think your parser is correct and behaves like the one we wrote, you may want to run a COOL compiler using

your parser:

% mycoolc foo.cl

To overwrite the default lexical analyzer with yours, replace lexer (which is a symbolic link to the "official" lexer) with your lexer .

**FOR PHASE 1 (Lexical analysis code )**

/\*

\* The scanner definition for COOL.

\*/

/\*

\* Stuff enclosed in %{ %} in the first section is copied verbatim to the

\* output, so headers and global definitions are placed here to be visible

\* to the code in the file. Don't remove anything that was here initially

\*/

%{

#include <cool-parse.h>

#include <stringtab.h>

#include <utilities.h>

#define MAX 100

int t\_lines;

int print(){printf("Unrecognized string");return(ERROR);};

int value=0;

char \*msg="comment error";

/\* The compiler assumes these identifiers. \*/

#define yylval cool\_yylval

#define yylex cool\_yylex

/\* Max size of string constants \*/

#define MAX\_STR\_CONST 1025

#define YY\_NO\_UNPUT /\* keep g++ happy \*/

extern FILE \*fin; /\* we read from this file \*/

/\* define YY\_INPUT so we read from the FILE fin:

\* This change makes it possible to use this scanner in

\* the Cool compiler.

\*/

#undef YY\_INPUT

#define YY\_INPUT(buf,result,max\_size) \

if ( (result = fread( (char\*)buf, sizeof(char), max\_size, fin)) < 0) \

YY\_FATAL\_ERROR( "read() in flex scanner failed");

char string\_buf[MAX\_STR\_CONST]; /\* to assemble string constants \*/

char \*string\_buf\_ptr;

extern int curr\_lineno;

extern int verbose\_flag;

extern YYSTYPE cool\_yylval;

/\*

\* Add Your own definitions here

\*/

%}

/\*

\* Define names for regular expressions here.

\*/

identifier [a-zA-Z][a-zA-Z0-9]\*

digits [0-9]+

number digit+

CLASS (class|CLASS)

ELSE (else|ELSE)

FI (fi|FI)

IF (if|IF)

IN (in|IN)

INHERITS (inherits|INHERITS)

LET (let|LET)

LOOP (loop|LOOP)

POOL (pool|POOL)

THEN (then|THEN)

WHILE (while|WHILE)

CASE (case|CASE)

ESAC (esac|ESAC)

OF (of|OF)

DARROW =>

NEW (new|NEW)

ISVOID (isvoid|ISVOID)

STR\_CONST (str\_const|STR\_CONST)

BOOL\_CONST (true|false)

ASSIGN =

NOT ~

x [(]

y [\*]

z [)]

LE <=

line \n

open [(][\*]

close [\*][)]

str ["].\*["]

%%

/\*

\* Nested comments

\*/

{open} { printf("hellovirat");

if(value>=0) value++; else {yylval.error\_msg=msg; print(); }

}

{close} {

if(value-1>=0) value--; else {yylval.error\_msg=msg; print(); }

}

/\*

\* The multiple-character operators.

\*/

{CLASS} { return (CLASS); }

{FI} { return (FI); }

{IF} {return (IF); }

{IN} {return (IN); }

{INHERITS} {return (INHERITS);}

{LET} {return (LET); }

{LOOP} {return (LOOP); }

{POOL} {return (POOL); }

{THEN} {return (THEN); }

{WHILE} {return (WHILE); }

{CASE} {return (CASE); }

{ESAC} {return (ESAC); }

{OF} {return (OF); }

{DARROW} {return (DARROW); }

{NEW} {return (NEW); }

{ISVOID} {return (ISVOID); }

{BOOL\_CONST} {return (BOOL\_CONST);}

{ASSIGN} {return (ASSIGN);}

{NOT} {return (NOT);}

{LE} {return(LE);}

{digits} {

cool\_yylval.symbol = inttable.add\_string(yytext);

return (INT\_CONST);

}

line {t\_lines++;}

/\*

\* Keywords are case-insensitive except for the values true and false,

\* which must begin with a lower-case letter.

\*/

/\*

\* String constants (C syntax)

\* Escape sequence \c is accepted for all characters c. Except for

\* \n \t \b \f, the result is c.

\*

\*/

{str} {printf("string");}

%%

**FOR PHASE 2 (PARSER DEFINITION)**

/\*

\* cool.y

\* Parser definition for the COOL language.

\*

\*/

%{

#include <iostream>

#include "cool-tree.h"

#include "stringtab.h"

#include "utilities.h"

extern char \*curr\_filename;

/\* Locations \*/

#define YYLTYPE int /\* the type of locations \*/

#define cool\_yylloc curr\_lineno /\* use the curr\_lineno from the lexer

for the location of tokens \*/

extern int node\_lineno; /\* set before constructing a tree node

to whatever you want the line number

for the tree node to be \*/

#define YYLLOC\_DEFAULT(Current, Rhs, N) \

Current = Rhs[1]; \

node\_lineno = Current;

#define SET\_NODELOC(Current) \

node\_lineno = Current;

/\* IMPORTANT NOTE ON LINE NUMBERS

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* The above definitions and macros cause every terminal in your grammar to

\* have the line number supplied by the lexer. The only task you have to

\* implement for line numbers to work correctly, is to use SET\_NODELOC()

\* before constructing any constructs from non-terminals in your grammar.

\* Example: Consider you are matching on the following very restrictive

\* (fictional) construct that matches a plus between two integer constants.

\* (SUCH A RULE SHOULD NOT BE PART OF YOUR PARSER):

plus\_consts : INT\_CONST '+' INT\_CONST

\* where INT\_CONST is a terminal for an integer constant. Now, a correct

\* action for this rule that attaches the correct line number to plus\_const

\* would look like the following:

plus\_consts : INT\_CONST '+' INT\_CONST

{

// Set the line number of the current non-terminal:

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// You can access the line numbers of the i'th item with @i, just

// like you acess the value of the i'th exporession with $i.

//

// Here, we choose the line number of the last INT\_CONST (@3) as the

// line number of the resulting expression (@$). You are free to pick

// any reasonable line as the line number of non-terminals. If you

// omit the statement @$=..., bison has default rules for deciding which

// line number to use. Check the manual for details if you are interested.

@$ = @3;

// Observe that we call SET\_NODELOC(@3); this will set the global variable

// node\_lineno to @3. Since the constructor call "plus" uses the value of

// this global, the plus node will now have the correct line number.

SET\_NODELOC(@3);

// construct the result node:

$$ = plus(int\_const($1), int\_const($3));

}

\*/

void yyerror(char \*s); /\* defined below; called for each parse error \*/

extern int yylex(); /\* the entry point to the lexer \*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\* DONT CHANGE ANYTHING IN THIS SECTION \*/

Program ast\_root; /\* the result of the parse \*/

Classes parse\_results; /\* for use in semantic analysis \*/

int omerrs = 0; /\* number of errors in lexing and parsing \*/

%}

/\* A union of all the types that can be the result of parsing actions. \*/

%union {

Boolean boolean;

Symbol symbol;

Program program;

Class\_ class\_;

Classes classes;

Feature feature;

Features features;

Formal formal;

Formals formals;

Case case\_;

Cases cases;

Expression expression;

Expressions expressions;

char \*error\_msg;

}

/\*

Declare the terminals; a few have types for associated lexemes.

The token ERROR is never used in the parser; thus, it is a parse

error when the lexer returns it.

The integer following token declaration is the numeric constant used

to represent that token internally. Typically, Bison generates these

on its own, but we give explicit numbers to prevent version parity

problems (bison 1.25 and earlier start at 258, later versions -- at

257)

\*/

%token CLASS 258 ELSE 259 FI 260 IF 261 IN 262

%token INHERITS 263 LET 264 LOOP 265 POOL 266 THEN 267 WHILE 268

%token CASE 269 ESAC 270 OF 271 DARROW 272 NEW 273 ISVOID 274

%token <symbol> STR\_CONST 275 INT\_CONST 276

%token <boolean> BOOL\_CONST 277

%token <symbol> TYPEID 278 OBJECTID 279

%token ASSIGN 280 NOT 281 LE 282 ERROR 283

/\* DON'T CHANGE ANYTHING ABOVE THIS LINE, OR YOUR PARSER WONT WORK \*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\* Complete the nonterminal list below, giving a type for the semantic

value of each non terminal. (See section 3.6 in the bison

documentation for details). \*/

/\* Declare types for the grammar's non-terminals. \*/

%type <program> program

%type <classes> class\_list

%type <class\_> class

/\* You will want to change the following line. \*/

%type <features> feature\_list

%type <feature> feature

%type <formals> formal\_list

%type <formal> formal

%type <expression> expr

%type <expressions> comma\_expr\_list

%type <expressions> expr\_semi\_list

%type <cases> expr\_case\_list

%type <expression> expr\_let\_part\_1

%type <expression> expr\_let\_part\_2

/\* Precedence declarations go here. \*/

/\* the bottom, the precedence is higher\*/

%nonassoc IN

%right ASSIGN "<-"

%left NOT

%nonassoc "<=" '<' '='

%left '+' '-'

%left '\*' '/'

%left ISVOID

%left '~'

%left '@'

%left '.'

%%

/\*

Save the root of the abstract syntax tree in a global variable.

\*/

program : class\_list

{

@$ = @1;

SET\_NODELOC(@1)

ast\_root = program($1);

}

;

class\_list

: class ';' /\* single class \*/

{

@$ = @1;

SET\_NODELOC(@1);

$$ = single\_Classes($1);

parse\_results = $$;

}

| class\_list class ';' /\* several classes \*/

{

@$ = @2;

SET\_NODELOC(@2);

$$ = append\_Classes($1,single\_Classes($2));

parse\_results = $$;

}

;

/\* If no parent is specified, the class inherits from the Object class. \*/

class : CLASS TYPEID '{' feature\_list '}'

{

@$ = @4;

SET\_NODELOC(@4);

$$ = class\_($2,idtable.add\_string("Object"),$4,

stringtable.add\_string(curr\_filename));

}

| CLASS TYPEID INHERITS TYPEID '{' feature\_list '}'

{

@$ = @6;

SET\_NODELOC(@6);

$$ = class\_($2,$4,$6,stringtable.add\_string(curr\_filename));

}

| error

{}

;

/\* Feature list may be empty, but no empty features in list. \*/

feature\_list: /\* empty \*/

{

$$ = nil\_Features();

}

| feature ';' /\* single feature \*/

{

@$ = @1;

SET\_NODELOC(@1);

$$ = single\_Features($1);

}

| feature\_list feature ';' /\* mutil features \*/

{

@$ = @2;

SET\_NODELOC(@2);

$$ = append\_Features($1, single\_Features($2));

}

;

/\* feature grammar, two kinds of features, one is method, the

\* other is attr

\*/

feature : OBJECTID '(' formal\_list ')' ':' TYPEID '{' expr '}'

{

@$ = @8;

SET\_NODELOC(@8);

$$ = method($1, $3, $6, $8);

}

| OBJECTID ':' TYPEID

{

@$ = @3;

SET\_NODELOC(@3);

// init is null, so to create a null expression

$$ = attr($1, $3, no\_expr());

}

| OBJECTID ':' TYPEID ASSIGN expr

{

@$ = @5;

SET\_NODELOC(@5);

// create ASSIGN node

Expression assign\_node = assign($1, $5);

$$ = attr($1, $3, assign\_node);

}

| error

{}

;

/\* formal list grammar \*/

/\* [formal[, formal]\*] , need to return Formals\*/

formal\_list : /\* empty \*/

{

$$ = nil\_Formals();

}

| formal

{

@$ = @1;

SET\_NODELOC(@1)

$$ = single\_Formals($1);

}

| formal\_list ',' formal

{

@$ = @3;

SET\_NODELOC(@3);

$$ = append\_Formals($1, single\_Formals($3));

}

;

/\* formal grammar \*/

formal : OBJECTID ':' TYPEID

{

@$ = @3;

SET\_NODELOC(@3);

$$ = formal($1, $3);

}

;

/\* comma expr list grammar \*/

/\* [expr[,expr]\*] should return Expresstions \*/

comma\_expr\_list : /\* empty\*/

{

$$ = nil\_Expressions();

}

| expr

{

@$ = @1;

SET\_NODELOC(@1);

$$ = single\_Expressions($1);

}

| comma\_expr\_list ',' expr

{

@$ = @3;

SET\_NODELOC(@3);

$$ = append\_Expressions($1, single\_Expressions($3));

}

;

expr\_semi\_list : expr ';'

{

@$ = @1;

SET\_NODELOC(@1);

// Expressions single\_Expression(Expression e)

$$ = single\_Expressions($1);

}

| expr\_semi\_list expr ';'

{

@$ = @2;

SET\_NODELOC(@2);

$$ = append\_Expressions($1, single\_Expressions($2));

}

| error ';'

{ }

;

expr\_let\_part\_1 : LET expr\_let\_part\_2

{

@$ = @2;

SET\_NODELOC(@2);

$$ = $2;

}

;

expr\_let\_part\_2 : OBJECTID ':' TYPEID IN expr

{

@$ = @5;

SET\_NODELOC(@5);

$$ = let($1, $3, no\_expr(), $5);

}

| OBJECTID ':' TYPEID ASSIGN expr IN expr

{

@$ = @7;

SET\_NODELOC(@7);

$$ = let($1, $3, $5, $7);

}

| OBJECTID ':' TYPEID ',' expr\_let\_part\_2

{

@$ = @5;

SET\_NODELOC(@5);

$$ = let($1, $3, no\_expr(), $5);

}

| OBJECTID ':' TYPEID ASSIGN expr ',' expr\_let\_part\_2

{

@$ = @7;

SET\_NODELOC(@7);

$$ = let($1, $3, $5, $7);

}

| error IN expr

{}

| error ',' expr\_let\_part\_2

{}

;

expr\_case\_list : OBJECTID ':' TYPEID DARROW expr ';'

{//ID:TYPE=>expr;

@$ = @5;

SET\_NODELOC(@5);

// Cases single\_Cases(Case e)

// Case branch(Symbol name, Symbol type\_decl, Expression expr)

$$ = single\_Cases(branch($1, $3, $5));

}

| expr\_case\_list OBJECTID ':' TYPEID DARROW expr ';'

{ //ID:TYPE=>expr;ID:TYPE=>expr;......ID:TYPE=>expr;

@$ = @6;

SET\_NODELOC(@6);

$$ = append\_Cases($1, single\_Cases(branch($2, $4, $6)));

}

;

/\* expr grammar \*/

expr : OBJECTID ASSIGN expr

{

@$ = @3;

SET\_NODELOC(@3);

$$ = assign($1, $3);

}

| expr '.' OBJECTID '(' comma\_expr\_list ')'

{

// dispatch

@$ = @5;

SET\_NODELOC(@5);

$$ = dispatch($1, $3, $5);

}

| expr '@' TYPEID '.' OBJECTID '(' comma\_expr\_list ')'

{

// static dispatch, used to access methods in parent class

@$ = @7;

SET\_NODELOC(@7);

$$ = static\_dispatch($1, $3, $5, $7);

}

| OBJECTID '(' comma\_expr\_list ')'

{ // self dispatch

// self.id(), so need to create a 'self' expression

@$ = @3;

SET\_NODELOC(@3);

$$ = dispatch(object(idtable.add\_string("self")), $1, $3);

}

| IF expr THEN expr ELSE expr FI

{ // if...else.... grammar

@$ = @7;

SET\_NODELOC(@7);

$$ = cond($2, $4, $6);

}

| WHILE expr LOOP expr POOL

{ // while grammar

@$ = @5;

SET\_NODELOC(@5);

$$ = loop($2, $4);

}

| '{' expr\_semi\_list '}'

{ // block grammar

// block(Expressions): Expression

@$ = @2;

SET\_NODELOC(@2);

$$ = block($2);

}

| expr\_let\_part\_1

{

// let grammar is a recursive grammar--TODO

// it is hard, wait to solve

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

@$ = @1;

SET\_NODELOC(@1);

$$ = $1;

}

| CASE expr OF expr\_case\_list ESAC

{

// case grammar

// typcase(expression, cases)

@$ = @5;

SET\_NODELOC(@5);

$$ = typcase($2, $4);

}

| NEW TYPEID

{

@$ = @2;

SET\_NODELOC(@2);

$$ = new\_($2);

}

| ISVOID expr

{

@$ = @2;

SET\_NODELOC(@2);

$$ = isvoid($2);

}

| expr '+' expr

{

@$ = @3;

SET\_NODELOC(@3);

$$ = plus($1, $3);

}

| expr '-' expr

{

@$ = @3;

SET\_NODELOC(@3);

$$ = sub($1, $3);

}

| expr '\*' expr

{

@$ = @3;

SET\_NODELOC(@3);

$$ = mul($1, $3);

}

| expr '/' expr

{

@$ = @3;

SET\_NODELOC(@3);

$$ = divide($1, $3);

}

| '~' expr

{

@$ = @2;

SET\_NODELOC(@2);

$$ = neg($2);

}

| expr '<' expr

{

@$ = @3;

SET\_NODELOC(@3);

$$ = lt($1, $3);

}

| expr "<=" expr

{

@$ = @3;

SET\_NODELOC(@3);

$$ = leq($1, $3);

}

| expr '=' expr

{

@$ = @3;

SET\_NODELOC(@3);

$$ = eq($1, $3);

}

| NOT expr

{

@$ = @2;

$$ = comp($2);

}

| '(' expr ')'

{

@$ = @2;

SET\_NODELOC(@2);

$$ = $2;

}

| OBJECTID

{

@$ = @1;

SET\_NODELOC(@1);

$$ = object($1);

}

| INT\_CONST

{

@$ = @1;

SET\_NODELOC(@1);

$$ = int\_const($1);

}

| STR\_CONST

{

@$ = @1;

SET\_NODELOC(@1);

$$ = string\_const($1);

}

| BOOL\_CONST

{

@$ = @1;

SET\_NODELOC(@1);

$$ = bool\_const($1);

}

;

/\* end of grammar \*/

%%

/\* This function is called automatically when Bison detects a parse error. \*/

void yyerror(char \*s)

{

extern int curr\_lineno;

cerr << "\"" << curr\_filename << "\", line " << curr\_lineno << ": " \

<< s << " at or near ";

print\_cool\_token(yychar);

cerr << endl;

omerrs++;

if(omerrs>50) {fprintf(stdout, "More than 50 errors\n"); exit(1);}

}