Project 2

CS4110

Fall 2018

JavaCC

Arden Prado

Francisco Rodriguez

How to run:

The program can be compiled from command line or terminal using the command "javac *.java". This will compile all java files in the source. This step can also be skipped if using the .class files provided. To run the program, type:

"java -cp [PATH] compilers_project_2.MyNewGrammar [FILENAME]"

Where [PATH] specifies the install path to the compilers_project_1 package, and [FILENAME] specifies the file to be parsed. Test files are included in "./test/"

CFG

SimpleNode Start() : Program()

void Program() : Decl() [Program()]

void Decl() : LOOKAHEAD(3) VariableDecl()

| FunctionDecl()

| ClassDecl()

| InterfaceDecl()

void VariableDecl() : Variable() < _semicolon >

void Variable() : Type() < _id >

void Type() : < _int > (< _leftbracket > < _rightbracket >)*

| < _double > (< _leftbracket > < _rightbracket >)*

| < _boolean > (< _leftbracket > < _rightbracket >)*

| < _string > (< _leftbracket > < _rightbracket >)*

| < _id > (< _leftbracket > < _rightbracket >)*

void FunctionDecl() : Type() < _id > <_leftparen > Formals() <_rightparen > StmtBlock()

| < _void > < _id > < _leftparen > Formals() < _rightparen > StmtBlock()

void Formals() : [VariableList()]

void VariableList() : Variable() [< _comma > VariableList()]

void ClassDecl() : < _class > < _id > [< _extends > < _id >] [ImplementsList()] < _leftbrace > [FieldList()] < _rightbrace >

void ImplementsList() : < _implements > < _id > (< _comma > < _implements > < _id >)*

void FieldList() : Field() [FieldList()]

void Field() : LOOKAHEAD(3) VariableDecl()

| FunctionDecl()

void InterfaceDecl() : < _interface > < _id > < _leftbrace > [PrototypeList()] < _rightbrace >

void PrototypeList() : Prototype() [PrototypeList()]

void Prototype() : Type() < _id > < _leftparen > Formals() < _rightparen > < _semicolon >

| < _void > < _id > < _leftparen > Formals() < _rightparen > < _semicolon >

void StmtBlock() : < _leftbrace > StmtBlockPrime()

void StmtBlockPrime() : LOOKAHEAD(3)

[StmtList()] < _rightbrace >

| VariableDeclList() [StmtList()] < _rightbrace >

void VariableDeclList() : VariableDecl() VariableDeclListPrime()

void VariableDeclListPrime() : LOOKAHEAD(2)

VariableDeclList()

| {}

void StmtList() : Stmt() [StmtList()]

void Stmt() : [Expr()] < _semicolon >

| IfStmt()

| WhileStmt()

| ForStmt()

| BreakStmt()

| ReturnStmt()

| PrintStmt()

| StmtBlock()

void IfStmt() : < _if > < _leftparen > Expr() < _rightparen > Stmt() [< _else > Stmt()]

void WhileStmt() : < _while > < _leftparen > Expr() < _rightparen > Stmt()

void ForStmt() : < _for > < _leftparen > [Expr()] < _semicolon > Expr() < _semicolon > [Expr()] < _rightparen > Stmt()

void BreakStmt() : < _break > < _semicolon >

void ReturnStmt() : < _return > [Expr()] < _semicolon >

void PrintStmt() : < _println > < _leftparen > ExprList() < _rightparen > < _semicolon >

void ExprList() : Expr() [< _comma > ExprList()]

void Expr() : LOOKAHEAD(< _id > ((< _leftparen >) | (< _period > < _id > < _leftparen >)))

Call() ExprPrime()

|

LValue() [< _assignop > Expr()] ExprPrime()

|

Constant() ExprPrime()

|

< _leftparen > Expr() < _rightparen > ExprPrime()

|

< _minus > Expr() ExprPrime()

|

< _not > Expr() ExprPrime()

|

< _readln > < _leftparen > < _rightparen > ExprPrime()

|

< _new > < _leftparen > < _id > < _rightparen > ExprPrime()

|

< _newarray > < _leftparen > < _intconstant > < _comma > Type() < _rightparen > ExprPrime()

void ExprPrime() : < _plus > Expr()

|

< _minus > Expr()

|

< _multiplication > Expr()

|

< _division > Expr()

|

< _mod > Expr()

|

< _less > Expr()

|

< _lessequal > Expr()

|

< _greater > Expr()

|

< _greaterequal > Expr()

|

< _equal > Expr()

|

< _notequal > Expr()

|

< _and > Expr()

|

< _or > Expr()

|

{}

void LValue() : < _id > ((< _leftbracket > Expr() < _rightbracket >) | (< _period > < _id >)
)*

void Call() : < _id > [< _period > < _id >] < _leftparen > Actuals() < _rightparen >

void Actuals() : [ExprList()]

void Constant() : < _intconstant >

| < _doubleconstant >

| < _stringconstant >

| < _booleanconstant >

| < _null >

Optimization Methods

As a top parser (or LL) our project requirements differed greatly from those of our peers. For starters, our output didn't require indicating shifting and reductions as those aren't present in LL parsers. Instead we output a list of tokens as well as the parse tree created using JJTree. To optimize our parser we first modified the CFG given in the project. We added new productions in order to avoid ambiguity.

Localized Lookahead

There were a couple of productions were we couldn't find a way to optimize the CFG. Instead we relied on a localized lookahead to determine which production rules to use at a given time.

Removing Left Recursion

LL parsing cannot have left recursion so for productions such as Type and Expr required getting rid of the recursion. By using the formula we learned in class we removed the left recursion by augmenting the productions with a production prime.

Test Design

	Input	Expected Outcome	Program Result
1	void f (double x, double y) { }	Pass	Pass
2	int i = 1;	Fail	Fail *Cannot assign in declaration*

3	// in function
	result = 5.times(4);
		Fail	Fail
			5.times is not valid

4	// in function
	front = in.nextLine();
		Pass	Pass

5	// in function
	front = in.nextLine;
		Pass	Pass

6	int[][][] super;	Pass	Pass
7	a[3][4.5][b] = result = x + y + z;	Fail	Fail *No values allowed inside brackets*
8	// in function for (; ;) x = 1;	Fail	Fail *Expression required between brackets inside parentheses*

9	// in function
	if (h>w) g=h;
	else h=g;
	double a;
		Fail	Fail
			No new variable declarations after statement declarations

10	// in class boolean b; userDefinedClassType f(){} double d; string g(){}	Pass	Pass
11	the sample Toy program given in project #1	Fail	Fail *Variables cannot be assigned to a value in the declaration*
12	Revised sample Toy program #2 created for project #1	Pass	Pass

13	// in function
	while (true)
	{
	  if (x == 2)
	  break;
	  x = x + 1;
	}
		Pass	Pass

14	// in function
	String s;
	s = readln();
	println(s + "this is a string");
		Pass	Pass

15	// in class
	type t;
	t = 2;
		Fail	Fail
			Cannot assign variables outside of functions

Strength and Constraints

JavaCC generates all the necessary files outside of the main .jjt file we worked on. However, that also made it difficult to see the changes made by modifying the file. We relied heavily on our CFG which was clearly defined. Because JavaCC generates top-down parsers, translating our CFG to a useable parser was possible without compromising readability of the grammar. As stated earlier however, top-down parsers cannot be successfully created using left-recursive grammars, so a large part of the project revolved around resolving left-recursiveness in the language. Another core strength of JavaCC is fast parsing capabilities when the grammar is LL(1). This also meant that resolving choice conflicts with minimal reliance on local lookahead modifications was a priority of ours.