CS164: Programming Assignment 3
Dpar Parser Generator and *Decaf* Parser

Assigned: Thursday, October 7, 2004
Due: Wednesday, October 20, 2004, at **11:59pm**

October 8, 2004

1 Introduction

Overview

In this assignment, you will develop *Dpar*, a parser generator that will produce an LL(1) parser from a high-level syntactic specification. Using your parser generator, you will develop a parser for *Decaf*, our classroom subset of Java. This parser will create ASTs for input *Decaf* programs, so after connecting this parser with your lexer from PA2 and your interpreter from PA1, you will be able to interpret *Decaf* programs on an environment that's strictly your own.

The purpose of this project is to get hands-on experience with LL(1) parsers, and with automatic generation of parsers from a grammar. This project will also serve as your next case study of how to generate powerful code from a simple high-level specification.

You will complete this programming assignment in six steps (you are not required to follow these steps, as long as your parser generator and your *Decaf* parser behaves as expected):

1. First, you will write a LL(1) parser. This is the code that reads input tokens and manipulates the parse stack as directed by an LL(1) parsing table. After configuring this parser with an LL(1) table provided by us and hooking it up with your scanner from PA2, you will get a working parser for a small subset of *Decaf*. The curious will now be able to modify the LL(1) table by hand to change the grammar recognized by the parser.

2. Next, you'll start working on the *Dpar* parser generator. First, you will implement the computation of FIRST and FOLLOW sets. For your convenience, the grammar will be represented as an AST (not a Java AST this time), which will be obtained from a grammar specification `.dpar` file using a parser provided by us.

3. Having computed FIRST and FOLLOW sets, you will write code that constructs the LL(1) parsing table. You will also write code that outputs the table in Java. This table is essentially all that your *Dpar* parser generator will generate. You will now be able to configure your LL(1) parser from Step 1 by writing a grammar in a `.dpar` file. At this point, you should experiment with several small grammars, to learn how to convert them into LL(1) form.

4. You will then add support for user-specified actions, which will allow you to use your parser as a syntax-directed translator. As a test, you will now be able to hack up a simple calculator by adding simple actions to a grammar of arithmetic expressions.

5. You will then write an LL(1) grammar for full *Decaf*. This step will boil down to some left-factoring and left-recursion elimination. These transformations will be done by hand, but the adventurous types should feel free to implement them as automatic transformations on the grammar AST (again, this is optional). As a reward, their `.dpar` should become cleaner and simpler.

6. Almost done. You will add actions that will build the AST for *Decaf*.

7. Finally, you will link your interpreter from PA1 with the generated *Decaf* parser, yielding a full implementation of the program constructs supported by the interpreter.

On this project, you will work in pairs. You are allowed to form different teams for PA3, but you are not allowed to disband a team in the middle of PA3.

What you are going to learn and why it is useful

First, you are going to learn in practice how LL(1) parsers work, and how to compute the FIRST and FOLLOW sets, and also the parsing table. This experience should solidify the concepts from the lecture.

Second, you will learn about syntax-directed translation, which is a process of translating the parse tree into another form during the parsing process. In our case, you are going to translate the parse tree into Decaf ASTs. You will see that coding the actions for the translation follows a different programming model than you are used to from object-oriented programming, and you will get familiar with debugging this semantic-stack-based programming model.

Third, you will get another practice with generating code from high-level specifications. In this assignment, the specification will describe (i) the grammar of the language that we want to parse; and (ii) the translation actions to be performed during the parsing process. The generated code will be a Java code contaning the parsing table. This table will configure the LL(1) parser code that remains the same for each grammar.

2 The Assignment

Your goal in PA3 is to

1. implement a parser generator called *Dpar*,
2. use *Dpar* to generate a parser for *Decaf*, and
3. develop a sufficient collection of test cases to test your parser generator and your *Decaf* parser.

Your parser generator must turn grammar specifications described in Section 3.3 into LL(1) parsers. Your parser must recognize token sequences that are valid according to the *Decaf* language definition in Section 2.1.

Note: You may want to skip the rest of Section 2 on the first reading. Although this section contains important details, it is sufficient to refer back to these details as you are progressing through the steps of the assignment described in Section 3.3.

2.1 *Decaf*'s Language Definition (Syntactic Specification)

The syntactic specification for *Decaf* appears in Figure 1. The grammar is suitable for reference, but not for use as input to a parser generator, as it is ambiguous and makes liberal use of regular-expression style notation not accepted by *Dpar*. The regular expression notation is defined as follows:

$x+$	1 or more instances of x.
$x*$	0 or more instances of x.
$[x]$	0 or 1 instances of x.
$\{x\}^{+},$	1 or more instances of x separated by commas.

Non-terminals are always written in all-caps. Big curly braces are for grouping, and curly braces enclosed in single-quotes indicate the actual curly brace terminals. Parentheses always correspond to terminals.

To disambiguate this grammar, we need associativity and precedence rules, which are as follows. All of the binary operators (including field dereference) are left associative. The field dereference operator (`.`) has highest precedence, then unary minus and not (`!`), then multiplication and division, then addition and subtraction, then the shift operators, then the relational operators, then the equality operators, then logical and (&&), and finally logical or (||).

3 Implementation notes

3.1 The starter kit

The starter kit is available on the web at:

PROGRAM	→	CLASSDECL+
CLASSDECL	→	**class** ID $\big[$ **extends** ID $\big]$ '{' DECL* '}'
DECL	→	FIELDDECL | METHODDECL
FIELDDECL	→	$\big[$ **static** $\big]$ TYPE $\big\{$ ID $\big\}^{+}$, **;**
METHODDECL	→	$\big[$ **static** $\big]$ $\big\{$ TYPE | **void** $\big\}$ ID **(** $\Big[$ $\big\{$ TYPE ID $\big\}^{+}$, $\Big]$ **)** BLOCK
BLOCK	→	'{' STMT* '}'
STMT	→	VARDECL | STATEMENT
VARDECL	→	TYPE $\big\{$ ID $\big\}^{+}$, **;**
TYPE	→	**int** | **boolean** | ID
STATEMENT	→	LOCATION = EXPR **;**
	|	METHODCALL **;**
	|	**print** **(** EXPR **)** **;**
	|	**if** **(** EXPR **)** BLOCK $\big[$ **else** BLOCK $\big]$
	|	**while** **(** EXPR **)** BLOCK
	|	**return** $\big[$ EXPR $\big]$ **;**
	|	BLOCK
METHODCALL	→	$\big[$ SIMPLEEXPR **.** $\big]$ ID **(** $\big[$ EXPR^{+}, $\big]$ **)**
LOCATION	→	ID
	|	SIMPLEEXPR **.** ID
EXPR	→	SIMPLEEXPR
	|	METHODCALL
	|	**new** ID **( )**
	|	LITERAL
	|	EXPR BINOP EXPR
	|	UNARYOP EXPR
	|	**(** EXPR **)**
SIMPLEEXPR	→	LOCATION | **this**
BINOP	→	ARITHOP | RELOP | EQOP | CONDOP
ARITHOP	→	+ | - | * | / | << | >>
RELOP	→	< | > | <= | >=
EQOP	→	== | !=
CONDOP	→	&& | ||
UNARYOP	→	- | !
LITERAL	→	INTLIT | BOOLLIT | **null** | STRINGLIT
BOOLLIT	→	**true** | **false**

Figure 1: The grammar for *Decaf*.

`http://www-inst.eecs.berkeley.edu/~cs164/starters/PA3.zip`

You can also get the starter kit from your Unix account at:

`~cs164/public_html/starters/PA3.zip`.

Usage of the starter kit is the same as PA1. Please to refer to the PA1 handout for information on how to import the zip file into a new project.

3.2 Relevant Reading

You may wish to read a tutorial on how to write a parser using an off-the-shelf parser generator, such as `yacc`, `bison`, or `CUP`. Understanding the parser specification that is fed into these parser generators will help you develop *Dpar*, our parser generator. When studying the specifications, focus on how grammar productions are combined with executable actions. Feel free to disregard shift-reduce conflicts and precedence declarations; because *Dpar* will produce an LL(1) parser, not an LR parser, *Dpar* won't suffer from the former and won't support the latter.

The yacc parser generator is described in [ASU], on page 257. The [FLC] textbook doesn't describe any parser generator, but you can read about CUP at
`http://www.cs.princeton.edu/~appel/modern/java/CUP/manual.html`.

3.3 Project Steps

We suggest that you build your parser and parser generator one piece at a time, which will help you verify the correctness of your code at each step. This project is divided into 7 steps. The harness code that is used to run the generated parsers is in `Main.java`. The harness code to run the parser generator can be found in `LL1ParserGenerator`.

The harness code given will show you how to set up the different parts of the code for testing, however we will be running your code on many more test cases than what is given in these files.

In general for this project, like in PA2, feel free to add fields or methods to any of the classes that we have provided. However, make sure that the methods that the `Main` class relies upon are not broken, as they will be used for testing. Also, make sure you don't change the interface to the `LL1ParserGenerator` class.

Step 0 – Writing the LL(1) parser and connecting it with your lexer.

In this step, you are going to write the LL(1) parser code. This parser will call the lexer you developed in PA2. You will configure the parser with parsing tables provided by us; the result will be a parser for a subset of *Decaf* restricted to simple arithmetic expressions. No ASTs will be created by this parser because this parser will perform no syntax-directed translation. In other words, this parser will act only as a recognizer: it will only answer whether the input program has a syntax error or not.

The parser table we have provided in the starter kit for this step resides in the `SimpleParserTable` class. It was created by our parser generator from the syntax specification in file `calculator.dpar`. You are going to develop such a parser generator in Steps 2 and 3 of this assignment.

The three main classes that you need from PA2 are `LookaheadLexer`, `TokenMap` and `LookaheadLexer.To`

Instead of your lexer, you can use our PA2 solution, available at `http://www-inst.eecs.berkeley.edu/~cs164/starters/PA2solution.zip`. If you use our solution, to install it, you will need to create a project and import the zip file, as you usually do with a starter kit. The solution we provide contains only class files for the generated parser; no source code is provided. We can give you a hardcopy of the PA2 solution if you are interested.

To reference in PA3 the classes residing in the PA2 solution (whether the solution is your own or ours), you need to edit the properties of your PA3 project as follows:

1. Open your PA2 project if it is not already open. This is the project with the PA2 solution. Let's assume the project name is PA2.
2. Right-click on your PA3 project, and go to Properties/Java Build Path/Projects. Check the box next to PA2. Click OK.
3. Now you should be able to import into PA3 the class `edu.berkeley.cs164.lexer.LookaheadLexer` and any other classes you may need from your solution to PA2.

Error reporting. On syntactically invalid inputs, your parser must throw a `SyntaxErrorException`. This exception must carry the invalid token, so that the parser's caller (`Main()`) can report the location of the syntax error stored in this token.

Classes to Modify:

`LL1Parser`	The parser file; you need to write the `parse` method.

Other Classes Needed:

`Lexer`	A wrapper for your PA2 lexer.
`LL1Table`	Base class for all LL1 parsing tables
`SimpleParserTable`	Generated table to use for parsing. Created from file `calculator.dpar` .
`Token`	PA2 token.
`TokenMap`	PA2 token map generated from `decafTokens.dlex`.
`calculator.dpar`	Dpar specification from which the parser has been generated by our parser generator (the generator is not provided).
`testinput.txt`	Very simple test file. This file should be accepted by the parser once it is linked with the lexer.

Step 1 - Computing First and Follow Sets

In order to generate the table, the first step will be to build the FIRST and FOLLOW sets. We have provided you with the skeletons of two classes, FIRST and FOLLOW, for this purpose. The two classes will receive an AST for the input specification and a TokenMap, and must produce the FIRST and FOLLOW sets to be used to produce the table.

You will construct the first and follow sets from a grammar representation of a language (i.e., a grammar AST) using the algorithms that can be found either in the lecture notes or the Dragon book.

The grammar rules that specify the parser are given in .dpar files. For example, here is the supplied simple.dpar with productions for simple arithmetic expressions:

```
S -> E <EOF> ;
E -> T X ;
X -> <PLUS> E | _ ;
T -> B Y ;
Y -> <TIMES> T | _ ;
B -> <LPAREN> E <RPAREN> | <INTLITERAL> ;
```

The syntax is mostly straightforward. On the right-hand side of productions, non-terminals are referenced directly by name, while terminals are referenced with angle brackets. The | character has its standard meaning, and _ stands for ϵ. We require that there be at most one production for each non-terminal (*ie.* you cannot write two productions with the same non-terminal on the left-hand side; you must use |), and the first non-terminal listed is the start non-terminal (in this case E). There is additional syntax for describing actions, but you can ignore it until Part 3. Note that the above example grammar is right-associative; Parts 3 and 4 will require a different style of left-factoring.

We have written a parser for .dpar files which generates an AST representation similar in structure to the Eclipse JDT AST. Here are brief descriptions of the *Dpar* AST classes (for details, see the source code in the edu.berkeley.cs164.parser.ast package):

DPASTNode The root of the AST node class hierarchy.

Spec Represents the entire *Dpar* spec.

Production A single production in the spec.

RightHandSide One of the possible right-hand sides of a production. For example, if we have A -> B | C, both B and C are right-hand sides.

NameWithAction Abstract class representing a name associated with an action (we deal with actions more in Part 3; ignore them for now).

Terminal A reference to a terminal on the right-hand side of a production.

NonTerminal A reference to a non-terminal on the right-hand side of a production.

Epsilon A reference to ϵ on the right-hand side of a production.

SimpleName Represents a name of a terminal or non-terminal.

DPVisitor Root of the visitor class hierarchy.

DPGenericVisitor A visitor that provides default implementations for the `visit()` methods.

DPASTPrinter An example visitor that prints an AST.

You can experiment by writing a .dpar file, reading it in with `DParParser` (look at the `generate()` method in `ParserCodeGenerator` to see how), and then printing the AST with `DPASTPrinter` to see what it looks like.

Classes to modify:

`FIRST`	You have to write this class so the getFirstSet method will take in either a NonTerminal, a Terminal, a RightHandSide, or a production, and will return a HashSet of Integers with the IDs of all the Terminals in the FIRST Set.
`FOLLOW`	You have to write this class so the getFollowSet method will take in a NonTerminal name and will return a HashSet of Integers with the IDs of all the Terminals in the FOLLOW Set.

Other Classes Needed:

Dpar AST Classes	The AST classes described above.
`LL1Table`	Superclass of the parser you will generate.

Step 2 - Generating the Parsing Table

In this step, you will write your own parsing table code generator. The task is better divided into two subtasks. The first, step is to create the table using the FIRST and FOLLOW sets computed in Step 1, and can be implemented in the `buildTable` method in `LL1TableGenerator`. The actual code generation can be done in the `emitTable` code in the same class.

One annoyance you probably noticed while working on PA2 was that after you ran `LexerCodeGenerator` to generate a lexer, you had to refresh your project in Eclipse to get the new `LexerCode` class to be compiled. To alleviate that annoyance in PA3, we have provided an *Ant* build file that both runs the parser code generator and recompiles the generated parser. Apache Ant is a Java-based build tool that is nicely integrated with Eclipse. For more on Ant, see its home page at `http://ant.apache.org`. Here is how you can use Ant to help with parser generation:

1. Open the `build.xml` file and make sure the `eclipse-home` property is set appropriately (see the comments in the file for details).
2. Right-click on the `build.xml` file and go to Run Ant. Then click the Run button to actually run the build.
3. To run the same build again, you can go to the Run / External Tools top-level menu, and choose the build file (or use the keyboard shortcuts ALT+R, then E, then 1).

Classes to Modify:

`LL1ParserGenerator`	Complete the `emitCode()` method to generate the parser code. Your code should be output to `FullLL1Table.java` in the `edu.berkeley.cs164.parser.runtime` package. The generated class should be a subclass of `LL1Table`.
`LL1TableGenerator`	Write the `createTable()` and `emitTable()` methods.

Other Classes Needed:

Dpar AST Classes	The AST classes described above.
`LL1Table`	Superclass of the parser you will generate.

Step 3 - Adding Actions for Syntax-Directed Translation

You will be adding support for *actions* to `LL1ParserCodeGenerator` in this step, enabling the generated parser to perform syntax-directed translation, as studied in lecture. An action is simply some arbitrary Java code that manipulates the *semantic stack* of the parser (the `stack` field in `LL1Table`) in a certain disciplined way. After finishing this step, you will be ready to complete a fully functional *Decaf* parser that creates Eclipse JDT ASTs.

The syntax for specifying actions in a .dpar file is again fairly simple. Here is the previous grammar for simple expressions extended with actions to construct an Eclipse AST (this grammar is also in `simple-with-actions.dpar`):

```
IMPORTS: [| import org.eclipse.jdt.core.dom.*;
            import java.util.*; |]

PROLOGUE: [| private AST ast = new AST(new HashMap()); |]

S -> E <EOF> [| push(peek(-1)); |]
E -> T X ;
X -> <PLUS> E [|
         InfixExpression e = ast.newInfixExpression();
         e.setLeftOperand((Expression)peek(-2));
         e.setRightOperand((Expression)peek(0));
         e.setOperator(InfixExpression.Operator.PLUS);
         push(e);
     |] | _ ;
T -> B Y ;
Y -> <TIMES> T [|
         InfixExpression e = ast.newInfixExpression();
         e.setLeftOperand((Expression)peek(-2));
         e.setRightOperand((Expression)peek(0));
         e.setOperator(InfixExpression.Operator.TIMES);
         push(e);
     |] | _ ;
// this is a comment
B -> <LPAREN> E <RPAREN> [| push(peek(-1)); |]
   | <INTLITERAL> [|
     push(ast.newNumberLiteral(((Token)peek(0)).getLexeme())); |] ;
```

At the beginning of the file, we can specify classes to be imported in the generated LL1Table in the `IMPORTS:` section, and we can create a prologue with any fields and helper methods we need in the `PROLOGUE:` section. All Java code is enclosed in special square braces (eg. `[|` code `|]`). Actions are enclosed in these braces, and they can appear after any terminal, non-terminal, or ϵ on the right-hand side of a production. The action should execute immediately after the preceding terminal, non-terminal, or epsilon is successfully parsed. Single-line comments are delimited with `//`.

As in lecture, our syntax-directed translation works by manipulating a *semantic stack*. The semantic stack stores information (arbitrary objects) about the terminals and non-terminals on the right-hand side of productions. The following rules govern the use of the semantic stack for our generated parsers:

1. Each successfully parsed terminal pushes the corresponding `Token` object on the stack (using the terminology from lecture, the `Token` object is the *attribute* of the terminal). You can implement this from the `parse` method in `LL1Parser`.

2. When ϵ is successfully parsed, you want to push into the stack a copy of the top of the stack. Can you think of the reason for this?

3. An action must push *exactly one* object on the stack, and it must not pop any objects (although it can look at the objects on the stack using the `peek` method). The actions in our example adhere to this rule. You do not need to do any checking of action code to ensure that this rule is followed; just be sure to follow it in the actions you write.

4. Each successfully parsed production must (1) pop all objects that were pushed on the stack in the parsing of the production; and (2) push back the object corresponding to the last terminal, non-terminal, or action on the right-hand side of the production. For example, if we successfully employ the production

   ```
   B -> <LPAREN> E <RPAREN> [| push(peek(-1)); |]
   ```

   then the parser must pop the objects corresponding to `<LPAREN>`, `E`, `<RPAREN>`, and the action, and then push back the object corresponding to the last object in the production, in this case the action.

In summary, you will need to change `LL1ParserGenerator` to do the following additional things in this step:

- Emit code for imports, the prologue, and the actions in the appropriate places. Actions are represented in the AST by `Action` objects that wrap a `String` representation of the code to be executed. The imports and prologue are represented similarly in the `Spec` class.
- Ensure that the code corresponding to the actions gets executed at the apropriate time. You can do this by treating actions as just another type of `RuntimeSymbol` that can be pushed into your LL1Stack.
- Modify the `parse` method in `LL1Parser` to implement rule 1 above.
- Emit code that handles rule 4 above by properly manipulating the stack when the RHS of a production has been succesfully parsed. The easiest way to do this is by using the `StackHousekeeping` class.
- For ϵ rules, you want to push into the stack a copy of the last item in the stack. The `StackHousekeeping` class already does this if you pass 0 to it.

When you are done, you can test your code using the `simple-with-actions.dpar` file, and with your own tests of course.

Step 4 - *Decaf* Grammar

In Step 3 of this assignment, you will complete the specification of the *Decaf* grammar. We provide a complete grammar for *Decaf* in `decaf.dpar`, but some of the rules are unsuitable for parsing by an LL(1) parser. Once you fix up these rules, you will be able to generate a parser for the full *Decaf* language. This parser contains no actions, so it will only *recognize Decaf* programs; it won't build their ASTs.

In `decaf.dpar`, much of the work of translating the Figure 1 grammar into one suitable for a recursive descent parser has already been done. However, the productions for the `EXPR` nonterminal are still essentially in the same form as in the figure, ambiguous and left-recursive. Your task is to add and rewrite productions to make parsing to `EXPR` unambiguous, respecting the precedence rules described above while allowing for a left-associative AST to be constructed with actions. You should also ensure the resulting rules are left-factored and are in a correct order (see WA3 for the rationale).

For associativity, remember that we want to eventually use actions to construct a left-associative AST for those operators that are left-associative. Your elimination of left-recursion and left-factoring should keep this mind, and be done in a style similar to that in the last couple of slides of the lecture notes on syntax-directed translation. There, a left-recursive production

```
    E -> E + T [| ...  |] | T
```

was transformed to the left-factored form

```
    E -> T E'
    E' -> + T [| ...  |] E' | ϵ
```

preserving the position of the action and thereby maintaining the left-associativity of the constructed AST. The difference in our case is that we have not inserted the actions yet (that will be done in the next step).

The only file you need to modify for this step is `decaf.dpar`. The `Main` class is already set up to run a couple of tests, but you should certainly add more.

Step 5 - Syntax-directed translation to an AST

In this step you will add any remaining actions necessary to `decaf-with-actions.dpar` so that the generated parser will construct a complete Eclipse JDT AST when run on a *Decaf* program. The productions that need actions are marked with TODO comments in `decaf-with-actions.dpar`; they include the production for statements and those productions you wrote in Step 3 for expressions. You will need to copy over the expression productions you created from `decaf.dpar`. Note that as written, the generated parser from `decaf-with-actions.dpar` will *not* work, since the actions for statements and expressions are missing. We have provided many actions for you; studying them may help you in adding actions to these productions.

Step 6 - Connecting the parser with the interpreter

As a final extra step, you will connect your interpreter from PA1 with your *Decaf* parser, yielding a complete implementation of the program constructs handled by your interpreter. All you have to do for this step is modify the `Interpreter` class to run your interpreter. Remember that your interpreter only handles a subset of Decaf, so you'll have to accordingly limit the input programs that you feed to it. We have provided three test cases for you to test your combined lexer / parser / interpreter; these are exactly the test cases that we will use when testing your implementation, so be sure that they work.

Classes to modify:

`Interpreter`	Hook in your interpreter to this class.

Handing in the Assignment

Follow the instructions on the course web site (the process will likely be similar or identical to PA2 submission).

If you are working in a *regularr team*, you will submit only one project per team. You can submit the project from either partner's account. Give your partners in the PARTNERS file.

If you are working in a *coalition team*, submit one solution per team member. See the lecture slides for grading rules. Give your partners in the PARTNERS file. **GOOD LUCK!**