MAHARISHI UNIVERSITY OF MANAGEMENT

COMPUTER SCIENCE DEPARTMENT

COMP 440 – Compilers (DE)

Programming Assignment 3

CP Parser

In this programming assignment you are to write a parser specification for the CP programming language. As in lab 2, you will be using the SableCC lexer and parser generator. A grammar that defines CP's syntax appears below. You should examine the grammar carefully to learn the structure of CP constructs. In most cases, structures are very similar to those of Java and C++. Note that at this stage you need not understand exactly what each construct does, but rather just the syntactic structure of each construct.

(1)	*program*	→	*class_decls*
(2)	*class_decls*	→	*class_decls class_decl*
(3)		|	*class_decl*
(4)	*class_decl*	→	*class_hdr* { }
(5)		|	*class_hdr* { *member_decls* }
(6)	*class_hdr*	→	`class` *id*
(7)	*member_decls*	→	*member_decls member*
(8)		|	*member*
(9)	*member*	→	*field*
(10)		|	*method*
(11)		|	;
(12)	*field*	→	*type id* ;
(13)		|	*type id* = expr ;
(14)		|	*type id* [*int_lit*] ;
(15)	*type*	→	`int`
(16)		|	`char`
(17)		|	`bool`
(18)		|	`string`
(19)		|	`float`
(20)	*method*	→	*method_hdr* (*formals*) *block*
(21)		|	*method_hdr* ( ) *block*
(22)	*method_hdr*	→	`void` *id*
(23)		|	*type id*
(24)	*formals*	→	*formal*
(25)		|	*formals* , *formal*
(26)	*formal*	→	*type id*
(27)		|	*type id* [ ]
(28)	*block*	→	{ *stmts* }
(29)		|	{ }
(30)	*stmts*	→	*stmts stmt*
(31)		|	*stmt*
(32)	*stmt*	→	*simple_stmt*
(33)		|	`if` *condition stmt* `else` *stmt*
(34)			
(35)	*condition*	→	(*expr*)
(36)	*local_decl*	→	*type id* ;
(37)		|	*type id* = *expr* ;
(38)		|	*type id* [*int_lit*] ;

(39)	*simple_stmt*	→	*local_decl*
(40)		|	*field_access* = *expr* ;
(41)		|	*method_call* ;
(42)		|	`return` ;
(43)		|	`return` *expr* ;
(44)		|	*block*
(45)		|	`while` *condition block*
(46)		|	`do` *stmts* `while` *condition* ;
(47)		|	;
(48)	*field_access*	→	*id*
(49)		|	*id* . *id*
(50)		|	*array_ref*
(51)	*array_ref*	→	*id* [*expr*]
(52)		|	*id* . *id* [*expr*]
(53)	*method_call*	→	*id* ( )
(54)		|	*id* (*args*)
(55)		|	*id* . *id* ( )
(56)		|	*id* . *id* (*args*)
(57)	*args*	→	*expr*
(58)		|	*args* , *expr*
(59)	*expr*	→	*expr* || *E2*
(60)		|	*expr* && *E2*
(61)		|	*E2*
(62)	*E2*	→	*E3* < *E3*
(63)		|	*E3* > *E3*
(64)		|	*E3* <= *E3*
(65)		|	*E3* >= *E3*
(66)		|	*E3* == *E3*
(67)		|	*E3* != *E3*
(68)		|	*E3*
(69)	*E3*	→	*E3* + *E4*
(70)		|	*E3* - *E4*
(71)		|	*E4*
(72)	*E4*	→	*E4* * *E5*
(73)		|	*E4* / *E5*
(74)		|	*E4* % *E5*
(75)		|	*E5*
(76)	*E5*	→	! *E5*
(77)		|	+ *E5*
(78)		|	- *E5*
(79)		|	*primary*
(80)	*primary*	→	*field_access*
(81)		|	*method_call*
(82)		|	*int_lit*
(83)		|	*char_lit*
(84)		|	*str_lit*
(85)		|	*float_lit*
(86)		|	`true`
(87)		|	`false`
(88)		|	(*expr*)

CP Grammar

The CP grammar listed above encodes the following precedence and associativity. The unary **!, +,** and **-** operators have the highest precedence. The *****, /, and % binary operators have the next highest precedence. The **+** and **–** binary operators have the third highest precedence. The relational operators (**==**, **!=**, <, <=, >= and >) have the fourth highest precedence. The boolean operators (**&&** and ||) have the lowest precedence. All binary operators are left-associative, except the relational operators which do not associate at all (i.e., A==B==C is illegal). The unary operators are (of course) right-associative. The following table shows the precedence and associativity of the CP operators as specified in the above CP grammar.

Precedence	Operator	Associativity
1	**!, +, -** (unary ops)	right-associative
2	*****, /, %	left-associative
3	+, -	left-associative
4	==, !=, <, >, <=, >=	non-associative
5	&&, ||	left-associative

Thus !A+B*C==3 || D!=F is equivalent to the following fully-parenthesized expression: ((((!A)+(B*C))==3) || (D!=F)).

In addition to generating the lexer and parser, the SableCC tool also automatically generates all the node classes used to represent each construct. Furthermore, when the parser is run, it automatically parses and generates a syntax tree representing the input program. If you have ever used other tools, you will greatly appreciate the framework of classes generated by this tool.

SableCC Considerations/Requirements

- Copy your **CP_lexer.scc** file from lab assignment 2 to a new file **CP_parser.scc**. Your main task will be to add the CP production rules to **CP_parser.scc**. Look at the examples of parser specifications given in the SableCC documentation. Don't forget that you need to include the **Ignored Tokens** section in your **CP_parser.scc** file to specify which tokens, such as whitespace and comments, are not part of the CP grammar (see Chapter 3 of the SableCC documentation for an example). As shown in the examples, the **Productions** keyword must appear immediately prior to the CP grammar rules, otherwise, SableCC cannot tell where the tokens end and the productions begin.
- A common error when starting out is to use uppercase letters in token or non-terminal names. This is NOT allowed in SableCC. Only lowercase letters and '_' are allowed in these names. Apparently this is to minimize the number of naming conflicts when generating the classes used to represent a syntax tree. The name of a generated class starts with a **T** if it represents a token and a **P** if it represents a non terminal. The rest of the name is derived from the token or non terminal name; the first letter is always capitalized as is the first letter following a '_'. For example, a non terminal named **method_hdr** would generate a class named **PMethodHdr**. Similarly, a token named **white_space** would cause a class named **TWhiteSpace** to be generated as seen in lab 2.
- Another common error is to forget that each alternative rule defining a non terminal must have its own unique name. Consider the following productions:

```
term = id_ref
     | {plus} term + factor
     | {minus} term - factor ;
```

Observe that the second and third alternatives are preceded by a name inside { }. These names are used as part of the name of the parse tree node classes representing these alternatives; the class names start with '**A**', followed by the name inside {..}, and ending with the non-terminal that appears on the left-hand side of the rule. In this example, the class representing the second alternative will be **APlusTerm**, and the third alternative will be represented by **AMinusTerm**. The first alternative will be represented by class **ATerm** since a name inside { } has not been given. Clearly, to avoid naming conflicts, only one alternative can omit this name. If more than one alternative omits the name inside { }, then you will get the error:

java.lang.RuntimeException: [0, 0] Redefinition of ATerm

- Consider the following SableCC rule.

  expr = ident dot ident

  The token ident occurs twice on the right hand side of this rule. Therefore, you would receive the following error message:

  [1, 15] Redefinition of AExpr.Ident

  SableCC error messages always indicate the line number and column number where the error occurs. This error message says that the error occurs in column 15 of line 1. **AExpr** is the class representing the rule, and the name of each field in this class is derived from the token names and non terminal names on the right hand side of the rule. In this case **Ident** comes from **ident** on the right-hand side (and there are two of them which causes the redefinition error). To correct this problem you will have to name the conflicting fields yourself. The following will correct the error.

  expr = [id1]:ident dot [id2]:ident

  Thus the two fields in **AExpr** will now be named **Id1** and **Id2** eliminating the naming conflict. The other field is named **Dot**.

- The following error message means that "dot" occurring on line 5 and column 10 has not been defined as either a token or non terminal. The "Dot" in the error message indicates that the token/non-terminal "dot" has not been defined in the parser specification.

  [5, 10] PDot and TDot undefined

  If it is defined as a non terminal, a class named **PDot** will be generated. If it is defined as a token, then a class named **TDot** will be generated. The way to correct this particular error would be to define 'dot' as a token (i.e., "dot='.';" in the Tokens section of the parser specification).

- **Important:** When you generate the parser for CP, if you have shift-reduce conflicts or reduce-reduce conflicts, then you have made a mistake in specifying the grammar. Check your grammar carefully and make sure it matches exactly; such errors are usually because there is a missing comma (,), semicolon (;), or parenthesis () in your grammar specification.

- **You must also add the following production rule to the grammar (becomes rule 33):**

  stmt → `if` *condition stmt*

  The original grammar is LALR(1). However, when this rule is added, the grammar will have conflicts. That is, it has the well-known "dangling else" problem. Therefore, SableCC will complain that there are shift/reduce errors since the changed grammar is ambiguous. The parser, for example, does not know how to parse statements like

  if (E1) if (E2) S1 else S2

  The grammar doesn't specify whether the else should go with the outer or the inner if. Once you have tested your grammar without this rule, you should insert it and modify the grammar to eliminate the shift/reduce errors. The correct association is to match the else part with the nearest unmatched if (the second if). You must modify the grammar provided to enforce this ``nearest match'' rule.

  The easiest way to correct the grammar is to restructure the grammar for the if-statement as shown on page 69 (1st ed.) or 67 (2nd ed.) of the text (the grammar is already structured to make this easy). There is also a

special FAQ on the web page with additional hints.

- **You must modify the precedence and associativity of some operators in the above grammar:** The above grammar does not have the correct operator precedence. The equality operators, == and !=, should have a lower precedence than the comparison operators, <, >, <=, and >=. In addition, the Boolean-or operator (||) should have a lower precedence than the Boolean-and operator (&&). Finally, == and != should be left associative rather than non-associative. You must modify the grammar provided to enforce this change in operator precedence and associativity. Correcting the precedence and associativity of the grammar will be important for this lab project because you will be tested in the exams to make sure you know how to do this. Before you begin this part of the assignment, read and make sure you understand the handout describing how to explicitly specify operator precedence and associativity in a grammar; it is posted to the course web site near the Lab 3 assignment specification. Be sure that your parser for CP properly reflects these precedence and associativity rules. The correct precedence and associativity of all operators is shown in the table below and your grammar specification must reflect this.

Precedence	Operator	Associativity
1	**!, +, -** (unary ops)	right-associative
2	***, /, %**	left-associative
3	+, -	left-associative
4	<, >, <=, >=	non-associative
5	==, !=	left-associative
6	&&	left-associative
7	||	left-associative

Evaluation Criteria

Your grammar MUST parse exactly those programs allowed by the CP grammar given in this lab, nothing more and nothing less. The only changes to the above CP grammar should be the changes in precedence and associativity and the addition of the if-then statement described above. Furthermore, if the grammar does not match, you may have conflicts or problems with labs 4 and 5 that use this parser.

One of the most important aspects of good object-oriented software design and development is the naming of classes, methods, and variables; names should make programs easy to read, understand, and modify. Thus names should be meaningful and helpful to other programmers who look at your code. Since a primary goal of this course is to help develop professionalism, a major part of the grade for this lab will be how well and how clearly you name your tokens, non terminals, and alternative production rules, i.e., how well the syntax tree node classes are named. These classes are named (as described above) based on the lexer and parser specification you define in labs 2 and 3. Good naming conventions will save you a lot of trouble and confusion on future lab assignments (Lab5 in particular).

The production rules in the CP Grammar table above, except for expressions, give a minimum standard for the naming of non-terminals. However, the syntax rules for expressions are intentionally named E2, E3, E4, etc. Therefore, it is up to you to name these non-terminals in a meaningful way; E2 or expr2 is unacceptable and will significantly reduce the grade on the overall lab project (one full grade, i.e., A to a B). It should be clear FROM THE NAME which production rule is represented by the class. Therefore, the name of the non-terminal and the name in braces are both significant. Consider for example, the following specification of rule (68):

additive_expr = {plus} additive_expr plus some_other_expr_name

This naming of the non-terminal (left-hand side) is acceptable because the class representing the node would be **APlusAdditiveExpr** ; from this name it is clear that the operator is plus and the expression is an additive expression. However, **some_other_expr_name** or **expr3** are not acceptable because they are not meaningful. The following is also another acceptable possibility for the non-terminal name:

term = {plus} term plus some_other_expr_name

However, the following would not be acceptable because it is unclear which operator is involved in the expression, i.e., **op1** is not meaningful.

additive_expr = {op1} additive_expr plus some_other_expr_name

Using numbers in names is usually not a good idea since the order or precedence of an operator or rule could change. If one has to look at the grammar and SableCC parser specification to determine which operator is involved, then the naming is NOT adequate.

We will look at each parser specification and let you know whether or not you need to improve the naming of expressions. This criterion will also be used when evaluating exam questions in which SableCC production rules have to be specified. However, do NOT ask us to evaluate every rule as you are doing this lab (that will be too time consuming for everyone). We will evaluate your specification when we evaluate lab 3; if renaming is necessary you will be notified of the rules that need to be improved at that time. The final evaluation will be made when you hand in your project at the end of the course. The grade will only be lowered if adequate corrections to the parser specification have not been made by the end of the course or if the naming conventions have been **copied from another student**.

Using SableCC to Build a Parser

You should copy **CP_lexer.scc** from lab assignment 2 to a new file **CP_parser.scc**. You may change the names used in the above CP grammar in any way you wish, although the names should be meaningful and helpful because they determine the names of node classes in the syntax tree representing the program. You can't change the CP language itself (i.e., the sequences of tokens considered valid) and you must use the token names you defined and used in lab 2. Some of the production rules like those for *class_hdr* and *method_hdr* are there to simplify later lab assignments (you will see why when you start doing lab 4). However, you are free to change the names of these or any other non-terminals in the grammar as long as the names are meaningful.

Once your grammar is in the right format and generates no error messages, SableCC will generate the **analysis**, **lexer**, **node**, and **parser** packages. Download **lab3.zip** from the course web site. In it you will find, **ParserDriver.java**, a program you can use to test your parser. Test your parser using the set of test files also included. Your parser will take the input file named on the command line and pass it to the scanner; the scanner reads and builds tokens for the parser. The parser will then process tokens and match the right hand side of production rules as described in the book and in class. After compiling **ParserDriver**, you can test your parser by typing the following at the DOS prompt:

```
java  ParserDriver   your_input_file
```

Whenever a change is made to **CP_parser.scc**, SableCC must be rerun to generate the new parser; then you must also recompile **ParserDriver.java** to make sure the new changes to the parser have been compiled so these changes will be used when you run the parser the next time. **Sometimes it becomes necessary to delete all of the node files after changes to your parser specification.**

What to Hand In

You should test your parser on syntactically valid and invalid programs. For valid programs, you should use the tests provided and create a file listing the sequence of tokens that has been parsed. For invalid programs, you should show the error message (each syntax error will have to be in a separate file – since SableCC only gives one error before stopping). Give three or four example files containing syntax errors.

Hand in a ZIP file containing (1) your **CP_parser.scc** grammar specification, (2) your set of invalid test files, and (3) the output files created when using our test files. You must also include (4) a README file stating the status of your project and describing how your parser behaves on the various test files. Missing or unnecessary files will reduce your grade (do not include any of the files from the **node**, **lexer**, **parser**, or **analysis** directories since they can be generated from your **CP_parser.scc** file).