Compiler Project - Part 2 Parser

Compilers, Fall 2016

Write a **recursive-descent predictive parser** for Decaf, a small subset of the Java programming language (see the grammar at the end of this document). Left recursion needs to be eliminated and the grammar needs left factoring for some productions.

You should hand in the changed grammar along with your source code.

The parser receives a token stream from the lexical analyzer (which you have already written) and checks that the string of tokens constitute a legal string in the language.
You should use your Lexical Analyzer that you implemented in the first part of the project as a starting point. If there are some issues with your Lexical Analyzer then you are free to take a look at a reference implementation of it that has been posted on the course web under Other materials.

Instructions

Error handling

If no errors are found then the parser prints out "*No errors*". If an error is found, whether or not the lexical analyzer or the parser finds it, you need to print out the respective source line along with a pointer (^) to the location in the line where the error occurs (or where the error is assumed to have occurred). You need to print out an **appropriate error message**. At the end you print out the **number of errors**.

Listing 1: test.dcf

```
class Program {
  int i,j;
  none_such_type o;

  static int myFunc(int n) {

    n = n * 2
    return n;
  }

  static void main() {
    i = myFunc(3);
    $i++;
  }
}
```

Listing 2: parser output

```
  3 : none_such_type o;
      ^ Expected a type.
  6 : n = n * 2
                ^ Expected semicolon.
 12 : $i++;
      ^ Illegal character
Number of errors: 3
```

In order to get the line numbers and the column numbers of tokens, you can use the **%line** and **%column** options of JFlex to supply this information (line number and column of the token). See the JFlex documentation for details on this. You will need to modify your lexical analyser project to supply this information.

Error recovery

When an error has been found, the parser needs to be able to recover, i.e. it needs to get into a state where it can conveniently continue parsing. We don't want our compiler to print out many errors when one particular error is found. Lets assume, for example, that we write the parser without error recovery, i.e. the parser only prints out an error message saying that it expected a specific token instead of the token it received from the lexical analyzer.

Listing 3: Example: missing comma

```
...
  static void myFunction(int i int a) {
...
  }
...
```

Here the COMMA token does not appear between ID(i) token and the INT token. After the parser has received ID(i) token from the scanner, it expects either the token COMMA or the token RPAREN. In our example it receives neither, but instead it gets the INT token. Since the parser does not receive any of the expected tokens, it prints out a message like "Expected a ',' and the next token it assumes to get is the one denoting a type. This is indeed the next token so in this case everything works out.
Now, let's assume the source looks something like this:

Listing 4: Example: missing a type

```
...
  static void myFunction(int i, a) {
...
  }
...
```

Here the programmer forgot to specify the type for parameter a. After the COMMA token the parser expects either an INT or a REAL token. Instead it unexpectedly finds ID(a) as the the next token. In this case the parser should output a message like "Expected a type". In order to recover from this the parser needs to read tokens until it finds a token that signifies an end of the current non-terminal (in this case the *parameters* non-terminal). A safe bet for that is the RPAREN token because that token is in the FOLLOW set for *parameters*. The comma character could also signify that a new parameter is being specified. The set of those "synchronising tokens"must be built for each non-terminal. You should read section 4.4.5 (pg 228 in textbook) for details on error recovery in this kind of parser. It is recommended that you use a combination of method 1) and 2) as describes on page 229.

Testing

As noted before, the goal is that the parser should only report actual errors and have as few as possible extra errors reported. You will need to write your own decaf files for testing, both for good (error-free) programs and programs with errors.To get you started, here is one decaf program that should be parsed without errors.

Listing 5: test.decaf

```
/* A program without parsing errors */
class Program {
  int x;
  real y;

  static real test (real x) {
    real t[10];
    int i;

    t[0] = -2.0 * x;
    for(i=1;i<10;i++) {
      t[i] = -2.0 * t[i-1];
    }
    return t[9];
  }

  static void main() {
    x = 1;
    y = 2.0*(3.14);
    x++;
    if (x>2) {
      y = 123.45E-3;
    }
    else {
      y = test(3.3);
    }
    x = x + 2;
  }
}
```

Recommendations

1. Modify the Lexical Analyzer so that line number and column number of tokens is reported with the token.
2. Start changing the grammar by eliminating left recursion and left factoring the grammar.
3. Write the recursive-descent predictive parser with regard to the changed grammar. You will find that the changed grammar "maps nicely to" the parser functions for each non-terminal. Ignore error handling in this part, just make sure that the parser runs on a legal program in the language.
4. Add error handling to your parser, so that it reports errors in the way specified above.
5. Finally, add error recovery as described above.

Recommendations continued on next page...

6. **Start early!** The time given for this project is well sufficient. Do yourself a favour and start the project early, it can take some time to implement and debug.

Hand-in Instructions

- Due date is Wednesday October 19th
- You can work in a group of up to three students if you want.
- Printouts of code are not required, just hand in electronically.
- Hand in the following electronically in a **.zip** archive named **SourceCode.Zip**
  - Your modified (left-factored, left-recursion-free) grammar:
    - Use the same format for the grammar as specified in this document (non-terminals in *italics*, tokens in **bold**, "::=" symbols in declarations).
    - The file should be named "grammar.pdf" and be a PDF file.
  - All source code
  - A single .jar file named Parser.jar that can be run using the command:
    `java -jar Parser.jar filename.decaf`
  - Keep in mind that your code will be tested by running the above command on a variety of decaf programs. It is in your best interest that this command works flawlessly when the archive is unzipped and the above command runned on the extracted jar file.

Appendix A: Decaf

Decaf is a language that is a very simplified subset of Java. It is defined in full by the BNF below. Here are some notes on the differences between Java and Decaf:

- Decaf is not an object oriented language. Programs in Decaf are written in a single class, that should by convention be named 'Program'. The class is only used as a skeleton around the actual program code. There are no accessibility modifiers (no 'public', 'private' or 'protected') and the keywords 'static', 'abstract', 'const' are not used. There is also expected to be a function named 'main' taking no parameters. This is the main function of the program that will be executed.
- There are two scopes of variables in Decaf, global scope and method scope. Global variables are declared first, inside the 'class' declaration where member variables are declared in Java. These declarations must precede any method declarations. Local variables (method scope) must be declared first in the method declaration before any statements.
- The only types that can be directly used in Decaf are int and real (note that the name of the floating point type is 'real' not 'float' or 'double'). There is also an implicit boolean type used in 'if' statements and 'for' statements for conditional checking. In these places, an integer or real value of zero is interpreted as FALSE, anything else is TRUE. Boolean variable declarations are not supported.
- Functions can have a return type of 'void' which is a separate type. Variable declarations of type 'void' are not supported.
- Since object are not supported, all functions must be declared static and therefore start with the 'static' keyword.
- The for loop has the following constraints on the three parts it takes as argument:
  - The first part must be a variable assignment
  - The second part is interpreted as a boolean condition, that when false at the beginning of the block, breaks out of the loop.
  - The third part must either be an increment or decrement statement.
- Arrays of int or real can be declared either globally or locally. However, they can not be used as parameters to functions. The arrays must be initialised with their size (e.g. 'int a[10]'). This will allocate space for all instances, there is no need to allocate them explicitly with 'new' as is needed in Java.

The BNF in Appendix B defines Decaf in more detail.

Appendix B: BNF for Decaf

The following is a context free grammar for Decaf in BNF form (tokens are in **bold**, nonterminals in *italics*). Note that the terminals **id, num, incdecop, relop, addop, mulop** have already been described above.

program ::= **class** **id** **{** *variable_declarations* *method_declarations* **}**

variable_declarations ::= *variable_declarations* *type* *variable_list* ; | ϵ

type ::= **int** | **real**

variable_list ::= *variable* | *variable_list* , *variable*

variable ::= **id** | **id** [**num**]

method_declarations ::= *method_declaration* *more_method_declarations*

more_method_declarations ::= *more_method_declarations* *method_declaration* | ϵ

method_declaration ::= **static** *method_return_type* **id** (*parameters*)
{ *variable_declarations* *statement_list* **}**

method_return_type ::= *type* | **void**

parameters ::= *parameter_list* | ϵ

parameter_list ::= *type* **id** | *parameter_list* , *type* **id**

statement_list ::= *statement_list* *statement* | ϵ

statement ::= *variable_loc* = *expression* ;
| **id** (*expression_list*) ;
| **if** (*expression*) *statement_block* *optional_else*
| **for** (*variable_loc* = *expression* ; *expression* ; *incr_decr_var*) *statement_block*
| **return** *optional_expression* ;
| **break** ;
| **continue** ;
| *incr_decr_var* ;
| *statement_block*

optional_expression ::= *expression* | ϵ

statement_block ::= { *statement_list* }

incr_decr_var ::= *variable_loc* **incdecop**

optional_else ::= **else** *statement_block* | ϵ

expression_list ::= *expression* *more_expressions* | ϵ

more_expressions ::= , *expression* *more_expressions* | ϵ

expression ::= *simple_expression* | *simple_expression* **relop** *simple_expression*

simple_expression ::= *term* | *sign* *term* | *simple_expression* **addop** *term*

term ::= *factor* | *term* **mulop** *factor*

factor ::= *variable_loc* | **id** (*expression_list*) | **num** | (*expression*) | ! *factor*

variable_loc ::= **id** | **id** [*expression*]

sign ::= + | −