Assignment 2: Recogniser

COMP3131/9102: Programming Languages and Compilers

Term 1, 2020

Worth: 12/100 marks

Due: 11:59pm Saturday 14 March 2020

Revision Log

Nothing at this stage.

End Of Revision Log

0. FAQs

Please read the FAQs for this assignment.

1. Specification

You are to implement a predictive recursive-descent parser for the VC language. In this assignment, your parser checks only syntactic correctness of the input program. In Assignment 3, you will complete your parser to build an Abstract Syntax Tree (AST) for the input program. The parser you implement for this assignment is also known as a *recogniser* in language theory and computational complexity. Hence the name *Recogniser*.

As in C, C++ and Java and many other languages before them, the **if** statement in the VC language suffers from the dangling-else problem. Your parser must always "match each **else** with the closest previous unmatched **then**."

Your parser will call the scanner you developed in Assignment 1 to obtain a sequence of tokens from the input program. If your scanner does not work properly, you can use the scanner provided for you (see Section 3).

Your compiler, which consists of a scanner and a parser at this stage, is required to work as follows. If the input program is syntactically legal, your compiler should announce success by calling:

```
System.out.println("Compilation was successful.");
```

This must be the last message your compiler prints to the standard output. If the input program is syntactically illegal, your compiler should call:

```
System.out.println("Compilation was unsuccessful.");
```

This must be the last message your compiler prints to the standard output. Before this last message, your parser is expected to print some meaningful error messages when syntax errors are detected.

2. Writing Your Parser

Set up your compiling environment as specified in Assignment 1 spec.

Download and install the supporting classes for this assignment as follows:

```
1. Copy ~cs3131/VC/Recogniser/Recogniser.zip into your VC directory
2. Set your current working directory as VC.
3. Extract the bundled files in the zip file as follows:
   unzip Recogniser.zip
```

The files bundled in this zip file are listed below. If you have trouble in handling Recogniser.zip, you can also download the supporting classes individually **all** from `~cs3131/VC/Recogniser` and install them into the respective directories (i.e., packages) as specified below:

```
The Recogniser package:
=======================

Recogniser.java:   a skeleton of parser (to be completed by you)
SyntaxError.java:  simple syntax error module
Test Files:        t1.vc, t2.vc, ..., t31.vc
Solution Files:    t1.sol, t2.sol, ..., t31.sol

The VC package:
===============

vc.java:           main compiler module (different from that in Assignment 1)
```

Your parser will use `ErrorReporter.java` you installed in your VC directory in Assignment 1. If you have not done so or have lost the file, copy it from ~cs3131/VC.

The parser supplied to you (consisting of about 300 lines of code) compiles immediately and parses a subset of the VC language. The EBNF grammar for the subset is given at the beginning of the file `Recognise.java`. At this stage, the parser parses successfully only the following single test case:

```
t1.vc
```

You are required to extend this parser to obtain a parser for the VC language, which will consist of approximately 550 lines of Java code.

You are, of course, welcome to develop this assignment completely from scratch provided it provides exactly the same interface as required.

The test programs `t19.vc -- t27.vc` are not legal VC programs. The corresponding solutions files contain some error messages. Do not ever to try to make your parser produce the same error messages! I did not incorporate any error handling strategy into my parser. You are free to print whatever message that is desirable in each case. This part of the assignment is not associated with any marks.

In the VC grammar, several nonterminals for expressions are left-recursive. You can replace these left-recursive productions by their equivalent EBNF productions. For example, the productions for *cond-or-expr* can be replaced with:

```
cond-or-expr -> cond-and-expr ( "||" cond-and-expr )*
```

The VC grammar is not LL(1) due to the existence of productions such as:

```
program  ->  ( func-decl | var-decl )*
```

In this case, both *func-decl* and *var-decl* begin with the same nonterminal *type*. In fact, it is not even LL(2)! However, A simple left-factoring transformation can eliminate this LL(1) (or LL(2) if you wish) parsing conflict.

In my implementation, only one token of lookahead is used. This means that the VC grammar can be easily transformed so that the transformed is LL(1).

3. The Scanner Class Files

If your scanner does not work properly, you can download our scanner implementation available only in class files.

- Copy ~cs3131/VC/Recogniser/Scanner-Sol.zip into the **parent directory** of your VC directory.
- Run

  ```
  unzip Scanner-Sol.zip
  ```

  This will extract the following class files from `Scanner-Sol.zip`:

  ```
  Scanner.class  Token.class   SourceFile.class  SourcePosition.class
  ```

  and install them under package `VC.Scanner`

Your parser will use only the single public (instance) method of the class `Scanner`:

```
getToken: returns the next token in the input and advances the
          position of currentChar in the input so that new tokens can be recognised.
```

This is all the help that you can expect from us if you use this scanner implementation.

4. Testing Your Parser

Make sure that your CLASSPATH includes the parent directory that contains the directory `VC`.

Create and run your parser on a UNIX-based machine as follows:

1. Compile the Java files:

   ```
   javac VC.java
   ```

   This will create all required class files for your parser.
2. Run your parser on a test file, `test.vc`, as follows:

   ```
   java VC.vc test.vc
   ```

Although some test files are provided for this assignment, you are responsible for designing additional test cases to make sure your parser works as desired. It is useful to use small test files to test individual aspects of your parser initially. It is important to test for boundary cases. Finally, you should try to design test cases to cover all possibilities. For example, since declarations and statements are both optional inside a block, it is reasonable to have separate test cases as follows:

```
(1) {  }
(2) { int i; }
(3) { i = 2; }
(4) { int i; i = 2;}
```

You should use a shell script to run your parser automatically over a large number of test cases. For example, the following simple shell script compiles every VC program under the current directory and pipes the output to the file with the same name but the suffix *.sol*.

```
#! /usr/local/bin/bash
for i in `ls *.vc`
do
        echo $i:
        b=`basename $i .vc`
        java VC.vc $i > $b.sol
done
```

You can also use a script to compare your solutions with ours on the supplied test cases:

```
#! /usr/local/bin/bash
for i in `ls t*.vc`
do
        echo $i:
        b=`basename $i .vc`
        java VC.vc $i > $b.xxx
        diff $b.xx $b.sol
done
```

5. Syntactic Errors

You are **not** required to recover from a syntax error.

On discovering the first syntax error, your parser is expected to print a meaningful error message indicating roughly the nature of the error and where in the program the error is detected. Your parser can then stop processing the remaining input and return to the caller. **Your error message must contain ERROR somewhere in it to enable automatical marking.**

This naive way of handling syntax errors has already been implemented in the supplied parser template file `Recogniser.java`. Every parsing method except `parseProgram` is required to throw a **SyntaxError** exception. On encountering the first error, a parsing method will call the method `syntacticError` of the class `Parser`, which will

- call the `errorReporter` object to print the supplied error message and increase the error counter, and
- throw a SyntaxError exception.

Eventually, `parseProgram` will catch this exception; it does nothing but simply returns to its caller. The caller (i.e, the `main` method of the class `vc`) examines the error counter variable of the `errorReporter` object to find out if the parsing has been successful or not.

6. Marking Criteria

Your parser at this stage functions as a recogniser, which takes as input a VC program and answers "Compilation was successful" meaning "yes" when the input is syntactically legal and "Compilation was unsuccessful" meaning "no" otherwise. Therefore, your parser will be assessed only by examining whether it can parse various syntactically legal and illegal inputs correctly. You will not be marked up or down for the syntax error handling aspect of your parser.

However, it is in your best interest to practice good programming and software engineering principles in coding.

7. Submitting Your Parser

Submit your parser file:

```
Recogniser.java
```

You should also submit the following Java files if you have modified them:

```
ErrorReporter.java
SyntaxError.java
vc.java
```

It is really unnecessary to modify any files except `Recogniser.java`.

The command for submitting your files is:

```
give cs3131 recogniser your-java-files (not the class files)
```

8. Late Penalties

This assignment is worth 12 marks (out of 100). You are strongly advised to start early and do not wait until the last minute. You will lose **3 marks** for each day the assignment is late.

Extensions will not be granted unless you have legitimate reasons and have let the LIC know ASAP, preferably one week before its due date.

You must complete this assignment before you can do Assignment 3.

9. Plagiarism

As you should be aware, UNSW has a commitment to detecting plagiarism in assignments. In this particular course, we run a special program that detects similarity between assignment submissions of different students, and then manually inspect those with high similarity to guarantee that the suspected plagiarism is apparent.

If you receive a written letter relating to suspected plagiarism, please contact the LIC with the specified deadline. While those students can collect their assignments, their marks will only be finalised after we have reviewed the explanation regarding their suspected plagiarism.

This year, CSE will adopt a uniform set of penalties for the programming assignments in all CSE courses. There will be a range of penalties, ranging from "0 marks for the assessment item", "negative marks for the value of the assessment item" to "failure of course with 0FL."

Here is a statement of UNSW on plagiarism:

Plagiarism is the presentation of the thoughts or work of another as one's own.*

Examples include:

- direct duplication of the thoughts or work of another, including by copying material, ideas or concepts from a book, article, report or other written document (whether published or unpublished), composition, artwork, design, drawing, circuitry, computer program or software, web site, Internet, other electronic resource, or another person's assignment without appropriate acknowledgement;
- paraphrasing another person's work with very minor changes keeping the meaning, form and/or progression of ideas of the original;
- piecing together sections of the work of others into a new whole;
- presenting an assessment item as independent work when it has been produced in whole or part in collusion with other people, for example, another student or a tutor; and,
- claiming credit for a proportion a work contributed to a group assessment item that is greater than that actually contributed.†

Submitting an assessment item that has already been submitted for academic credit elsewhere may also be considered plagiarism. Knowingly permitting your work to be copied by another student may also be considered to be plagiarism. An assessment item produced in oral, not written form, or involving live presentation, may similarly contain plagiarised material.

The inclusion of the thoughts or work of another with attribution appropriate to the academic discipline does not amount to plagiarism.

Students are reminded of their Rights and Responsibilities in respect of plagiarism, as set out in the University Undergraduate and Postgraduate Handbooks, and are encouraged to seek advice from academic staff whenever necessary to ensure they avoid plagiarism in all its forms.

The Learning Centre website is the central University online resource for staff and student information on plagiarism and academic honesty. It can be located at:

www.lc.unsw.edu.au/plagiarism

The Learning Centre also provides substantial educational written materials, workshops, and tutorials to aid students, for example, in:

- correct referencing practices;
- paraphrasing, summarising, essay writing, and time management;
- appropriate use of, and attribution for, a range of materials including text, images, formulae and concepts.

Individual assistance is available on request from The Learning Centre.
Students are also reminded that careful time management is an important part of study and one of the identified causes of plagiarism is poor time management. Students should allow sufficient time for research, drafting, and the proper referencing of sources in preparing all assessment items.

* Based on that proposed to the University of Newcastle by the St James Ethics Centre. Used with kind permission from the University of Newcastle.
† Adapted with kind permission from the University of Melbourne.

Have fun!

Jingling Xue

last updated 02/26/2020 17:17:28