Assignment 3: Parser

COMP3131/9102: Programming Languages and Compilers

Term 1, 2020

Worth: 18/100 marks | **Due: 11:59pm Friday 27 March 2020**

Revision Log

Nothing at this stage.

End Of Revision Log

1. Specification

You are to add code to the parser you built in Assignment 2 to construct an abstract syntax tree (AST) for a VC input program.

If a program is syntactically legal, your parser must build the AST for the program **exactly** as specified below. Otherwise, your parser can print an error message and then stop without having to complete the construction of the AST.

2. Constructing ASTs

The package `VC.ASTs` contains the class definitions for creating the ASTs representing VC programs.

The following template describes all 60 classes. Classes marked with +'s are abstract classes and those marked with ='s are concrete classes. Notice that all classes inherit the instance variable, named position, from AST. All constructors take a position argument (of type SourcePosition) after the arguments (if any) specific to the type of AST node. **This position argument is omitted for clarity in the following template.** How to fill-in the information for the position argument is discussed in Writing Your Parser.

```
+ AST(SourcePosition position)
  = Program(List dlAST)

  + List
    = DeclList(Decl dAST, List dlList)
    = EmptyDeclList()
    = StmtList(Stmt sAST, List slList)
    = EmptyStmtList()
    = ParaList(ParaDecl pAST, List plList)
    = EmptyParaList()
    = ExprList(Expr sAST, List elList)
    = EmptyExprList()
    = ArgList(Arg aAST, List alList)
    = EmptyArgList()

  + Decl
    = FuncDecl(Type tAST, Ident idAST, List fplAST, Stmt cAST)
    = GlobalVarDecl(Type tAST, Ident idAST, Expr eAST)
    = LocalVarDecl(Type tAST, Ident idAST, Expr eAST)
    = ParaDecl(Type tAST, Ident idAST)

  + Expr
    = VarExpr(Var vAST)
    = AssignExpr(Expr e1AST, Expr e2AST)
    = BinaryExpr(Expr e1AST, Operator oAST, Expr e2AST)
    = UnaryExpr(Operator oAST, Expr eAST)
    = CallExpr(Ident idAST, List aplAST)
    = IntExpr(IntLiteral ilAST)
    = BooleanExpr(BooleanLiteral slAST)
    = FloatExpr(FloatLiteral flAST)
    = StringExpr(StringLiteral flAST)
    = ArrayExpr(Var idAST, Expr indexAST)
    = InitExpr(List ilAST)
    = Arg(Expr eAST)
    = EmptyExpr()

  + Stmt
    = CompoundStmt(List dlAST,  List slAST)
    = IfStmt(Expr eAST, Stmt s1AST (, Stmt s2AST)?)
    = ForStmt(Expr e1AST, Expr e2AST, Expr e3AST, Stmt sAST)
    = WhileStmt(Expr eAST, Stmt sAST)
    = ExprStmt(Expr eAST)
    = ContinueStmt()
    = BreakStmt()
    = ReturnStmt(Expr eAST)
    = EmptyCompStmt()
    = EmptyStmt()

  + Type
    = IntType()
    = FloatType()
    = BooleanType()
    = StringType()
    = VoidType()
    = ArrayType(Type tAST, Expr dAST)
    = ErrorType()

  + Var
    = SimpleVar(Ident idAST)

  + Terminal
    = Ident(String value)
    = Operator(String value)
    = IntLiteral(String value)
    = FloatLiteral(String value)
    = BooleanLiteral(String value)
    = StringLiteral(String value)
```

`AST` is an abstract class for all abstract syntax trees. Each concrete subclass contains one constructor (two in the case of IfStmt) for creating a new AST node, and your parser uses these to construct the complete AST for the whole program.

The package `VC.ASTs` defines a design pattern in the interface `Visitor.java`, called `visitor`, for traversing the AST. This design pattern will be used by the last two phases of the VC Compiler, namely, contextual handling and code generation. Its understanding is not necessary for this assignment.

A large number of examples are used below to illustrate how to build the AST nodes for all the VC language constructs. These test cases are part of `Parser.zip` to be downloaded for this assignment. **It should be noted that no instance variable of any AST node can be null.** This is the reason for the existence of all "empty" AST classes. This requirement is not fundamental but represents just a choice made for this particular implementation.

- The empty program: t1.vc (AST)
- List of Global Variable Declarations: t2.vc (AST), t3.vc (AST)
- List of Function Declarations: t4.vc (AST), t5.vc (AST)
- List of Local Variable Declarations: t6.vc (AST), t7.vc (AST), t8.vc (AST)
- List of Statements: t9.vc (AST), t10.vc (AST), t11.vc (AST)
- List of (Formal) Parameters: t12.vc (AST), t13.vc (AST)
- List of Arguments (i.e., Actual Parameters): t14.vc (AST), t15.vc (AST)
- List of Expressions (i.e., expressions in an initialiser): t43.vc (AST), t44.vc (AST)
- Expressions t16.vc (AST), t17.vc (AST), t18.vc (AST), t19.vc (AST), t20.vc (AST), t21.vc (AST), t22.vc (AST), t23.vc (AST) t24.vc (AST), t24a.vc (AST), t24c.vc (AST)
- Assignments t24b.vc (AST)
- If statements t25.vc (AST), t26.vc (AST), t27.vc (AST), t28.vc (AST), t29.vc (AST)
- For Statements t30.vc (AST), t31.vc (AST), t32.vc (AST)
- While Statements t33.vc (AST), t34.vc (AST)
- Return Statements t34r.vc (AST)
- Expression Statements t35.vc (AST)
- Multi-variable declarations t36.vc t37.vc (AST, t37.ast), t38.vc t39.vc (AST, t39.gif)
- Programs t40.vc (AST). t41.vc (AST) t42.vc (AST).
- Arrays
  - Global variable declarations: t43.vc (AST).
  - Local variable declarations: t44.vc (AST).
  - Parameter declarations: t45.vc (AST).
  - Array Expressions: t46.vc (AST).
- SourcePosition t47.vc (AST), t41.vc (AST)

Some specific comments are in order:

- Every declaration statement in which multiple variables are declared is treated as if the variables had been declared in separate statements (with their order of appearance preserved). As a result, for example, the AST for

  ```
  int i, ii, kkk;
  ```

  is **exactly** the same as the AST for:

  ```
  int i;
  int jj;
  int kkk;
  ```

  This treatment applies to both local and global variables. It is up to you to decide how to define the positions of the "new declaration statements" in the input file. The positions for phrases (i.e., language constructs such as declarations and statements) are not marked. For my solution to this problem, see the AST trees (with positions) for t37.vc and t39.vc listed above.
- All lists are displayed in the form of binary subtrees as part of the AST. The head of a list *always* appears the highest in the AST. In other words, the order in which the nodes appear in a list is the same as the order in which the corresponding constructs appearing in the input program. Consider `t5.vc`. The AST nodes for the three functions `f, g` and `h` appear at levels 3, 4 and 5, respectively in the AST (assuming that the root is at level 1). These three functions appear textually in that order in the file `t5.vc`.
- For the current version of the VC language, the abstract class `Var` has only one concrete subclass, `SimpleVar`. More concrete subclasses can be introduced when the VC language evolves.
- `Void` is also considered as a type in C but not in Java.
- The AST class `ErrorType` will not be used in this assignment but will be used in Assignment 4.
- The AST class `StringType` will not be used in this assignment since VC does not permit a variable to be declared to be of this type. In assignment 4, string literals will be assigned this type for type checking purposes.
- `Arg` is defined to be an expression rather than a declaration according to the VC grammar.
- `EmptyExpr` is used in two situations. First, it is used to create an AST node for an empty expression `Expr? ";"` that appears in *for-stmt* , *return-stmt* and *expr-stmt* when Expr is absent. This is illustrated by t31.vc, t32.vc, t34r.vc and t35.vc. Second, `EmptyExpr` is used to create an AST node for an uninitialised variable (t3.vc and t8.vc). Creating these EmptyExpr nodes simplifies the AST construction. All these `EmptyExpr` nodes will be ignored by the later phases of the VC compiler.
- You will not use `EmptyStmt` explicitly in your code. `EmptyStmt` is used *only* in the AST class `IfStmt.java` to create an empty statement for an if statement that does not have a matching else part. Notice that `IfStmt` has two constructors since the else part is optional.
- Global variable, function and local variable declarations are all linked using `DeclList`. We could have defined three separate AST classes for them, say, `GlobalVarDeclList` for global variable declarations, `FuncDeclList` for function declarations, and `LocalVarDeclList` for local variable declarations. But this does not seem to be necessary.

Some AST classes contain a number of helper methods. For example, `Type.java` contains methods such as `isIntType` and `isFloatType`, which allow you to check if the type of a Type node in an AST is an `IntType` or `FloatType`. For this assignment, you will use only the constructors of AST classes to build the AST for a program. You will use these helper methods in Assignments 4 and 5.

Install Package TreeDrawer

Copy ~cs3131/VC/TreeDrawer/TreeDrawer.zip into your VC/TreeDrawer directory. To extract all the files, type:

```
unzip TreeDrawer.zip
```

This package contains the Java classes for drawing the ASTs for VC programs. You will find it extremely helpful to assist you with the development of your parser.

There is no need for you to understand how an AST is actually drawn. However, those who are interested in the Java graphical AWT and Swing are encouraged to study this package. Doing so may enhance your knowledge and programming skills about Java graphical animation.

To avoid clutter, the AST nodes are displayed using abbreviated names of those for the corresponding AST classes. For example, the class `ContinueStmt` is used to construct an AST node representing a continue statement. However, the name used for such a node when displayed is "ConStmt". The interested student can inspect `TreeDrawer/LayoutVisitor.java` to find out the names used for displaying all AST nodes.

Install Package UnParser

Copy ~cs3131/VC/UnParser/UnParser.java into your VC/UnParser directory.

This package, consisting of one Java class, namely `UnParser`, is written in order for this assignment to be automatically marked. Essentially, this package contains an unparser that takes as an AST as input, visit the AST nodes in the depth-first left-to-right, and produces as output an equivalent VC program representation. The reconstructed program is usually different from the original one. The differences, along with how `Printer` will be used for marking, are explained in Marking Criteria.

This class was written in about one hour. It can be easily enhanced to obtain a production quality pretty printer! Once again, mastering a few compiling techniques will enable you to write tools such as pretty printers and converters between high languages of similar programming styles.

Install Package TreePrinter

Copy ~cs3131/VC/TreePrinter/Printer.java into your VC/TreePrinter directory.

This package, consisting of one Java class, namely `Printer`, prints an AST in the text form into a file.

The ASCII ASTs are readable only for tiny programs.

Visitor Design Pattern

Both the AST drawing and code generation entail traversing the AST, visiting the nodes in some suitable order. The modern practice is to organise these tree traversals using the object-oriented visitor design pattern. Given a set of AST node classes, a visitor class is one that implements the corresponding set of visitor methods. In the case of the VC compiler, an AST visitor is one that implements the interface `Visitor` defined in the file `VC.ASTs.Visitor.java`.

The AST drawer, the pretty printer and the unparser are all implemented as visitor classes. *You are not required to understand this pattern to do this assignment*. However, anyone with a good grasp of OO should be able to understand the three packages by studying:

- The interface `Visitor` in the file `VC.ASTs.Visitor.java`,
- The instance method `visit` in each concrete AST class,
- `VC.TreeDrawer.LayoutVisitor.java`,
- `VC.PrettyPrinter.Printer.java`, and
- `VC.UnParser.UnParser.java`.

Assignments 4 and 5 will be implemented around the visitor design pattern.

3. Writing Your Parser

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

Copy ~cs3131/VC/ASTs/ASTs.zip into your VC/ASTs directory. To extract all the files, type:

```
unzip ASTs.zip
```

This package contains the AST classes for creating AST nodes.

You will need to use all and only the AST classes supplied. You are not allowed to modify this package.

Download and install the supporting classes for the parser as follows:

```
1. Copy ~cs3131/VC/Parser/Parser.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 Parser.zip
```

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

```
The Parser package:
=======================

Parser.java:        a skeleton of parser (to be completed by you)
SyntaxError.java:   simple syntax error module
Test Files:         t1.vc -- t23.vc, t24a.vc, t24b.vc, t24c.vc, t25.vc -- t47.vc
Solution Files:     t1.sol -- t23.sol, t24a.sol, t24b.sol, t24c.sol, t25.sol -- t47.sol

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

vc.java:            main compiler module (different from that in Assignment 2)
```

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 solution files are the VC programs obtained by running the unparser on the ASTs of the test case programs.

The parser supplied to you compiles immediately and parses a subset of the VC language. The EBNF grammar for the subset is given at the beginning of the file `Parser.java`. At this stage, the parser parses successfully the following test files:

```
t4.vc
t6.vc
t9.vc
t12.vc
t17.vc
t19.vc
t21.vc
```

The supplied parser is intended to demonstrate how to use the supplied AST classes to construct AST nodes. You will need to to enhance your existing parser to construct an AST for the input program, by adding approximately 500 lines of code.

All the supplied test programs are syntactically legal VC programs. The corresponding solution files are the VC programs produced from the ASTs of the test programs using the unparser supplied. See Marking Criteria for detail.

Each AST node represents a language construct, loosely speaking, a phrase, for the VC language. The position of a language construct can therefore be defined by using an object of the class `SourcePosition`. A `SourcePosition` object contains the four fields:

```
lineStart  --  the line where the construct begins
lineFinish -- the line where the construct ends
CharStart  -- the first char where the construct begins
CharFinish -- the last char where the construct ends
```

The supplied file `Parser.java` demonstrates how to make use of the two helper methods `start` and `finish` to fill-in the position information for an AST node. Sometimes, a "dummy position" is used for an "empty" AST node corresponding to no construct in the input program.

4. The Scanner Class Files

If you want to our scanner, see the instructions in Assignment 2 on how to download the class files.

5. Testing Your Parser

The VC compiler now accepts some switches to set compiler options. To see all the compiler options once you have compiled your compiler files, type:

```
java VC.vc
```

The following information is displayed:

```
Usage: java VC.vc [-options] filename

where options include:
  -ast                display the AST (without SourcePosition)
  -astp               display the AST (with Source Position)
  -t file             print the AST into <file>
  -u file             unparse the AST into <file>
```

The default values for the three options are:

- -ast: off
- -astp: off
- -t: The default file name is filename+t
- -u: The default file name is filename+u

All the test cases supplied for this assignment are intended to demonstrate how to build the ASTs for various language constructs. As always, you may need to use additional test cases for debugging purposes.

Here is a useful way to check the correctness of the AST for a test case, say, `test.vc`:

1. java VC.vc test.vc
   This writes the unparsed VC program in the file test.vcu.
2. java VC.vc -u test.vcuu test.vcu
3. diff test.vcu test.vcuu

If the two files differ, then your AST is wrong. If the two files are identical, your AST could still be wrong (why?).

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 repeats the above three steps on all the files *.vc under the current directory:

```
#! /usr/local/bin/bash
for i in `ls *.vc`
do
        echo $i:
        java VC.vc $i
        java VC.vc -u ${i}uu ${i}u
        diff ${i}u ${i}uu
done
```

You should 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:
        java VC.vc $i
        diff ${i}u `basename $i .vc`.sol
done
```

6. Marking Criteria

Your parser will be assessed only by examining whether it can build the ASTs correctly for syntactically legal VC programs. As before, there are not subjective marks concerning the programming style and documentation of your implementation. Of course, it is in your best interest to improve your programming and software engineering skills.

You will not be marked up or down for however your parser behaves on syntactically illegal VC programs.

In addition, you will not be marked up or down for the precision of the position information recorded for each AST node. However, an accurate position information may make it difficult for you to debug your static semantics component to be developed in Assignment 3.

This assignment will be marked automatically as follows. Suppose the input program being compiled is `test.vc`. Let `test.vcu` be the unparsed VC program produced from your AST for the input program. Let `test.sol` be the VC program reconstructed from our AST for the input program. Then, `test.vcu` and `test.sol` must be identical!

Finally, the unparsed program `test.vcu` differs from the original program `test.vc` in the following aspects:

- The comments in `test.vc`, which have been removed by the scanner, are not reproduced in `test.vcu`.
- The whitespace in `test.vc`, which has been removed by the scanner, is not reproduced in `test.vcu`.
- The parentheses in expressions, which have been removed during the construction of the AST, are not reproduced.

7. Submitting Your Parser

You are not allowed to modify the following packages:

```
ASTs:     Used for constructing the AST nodes
UnParser: Used for marking this assignment
```

There is no point modifying the driver program `vc.java`.

Therefore, you should submit your parser file:

```
Parser.java
```

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

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

The command for submitting your files is:

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

8. Late Penalties

This assignment is worth 18 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.

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.

Jingling Xue
Last updated 03/08/2020 17:48:22