Assignment 3: AST Generation

CSI4104 Compiler Design, Fall 2019
Document Version 4.5

Due date

- Please **do not submit** Assignment 3 before Tuesday Nov. 26, 2019! The grace-period of Assignment 2 ends on the previous day, and any submission before the end of the grace-period will overwrite your Assignment 2 submission!
- The due-date for Assignment 3 is Wednesday, Dec 4 2019, 23:00.

Introduction

We will extend our recursive descent parser from Assignment 2 to produce Abstract Syntax Trees (ASTs) for MiniC. If a program is syntactically legal, then the parser must build the AST for the program exactly as specified below. Otherwise the parser can print any error message and exit without completing the AST for the illegal input program.

Building ASTs

The package MiniC.AstGen contains our class hierarchy of AST nodes. We will use this class hierarchy to create AST nodes for MiniC programs. The following list describes all 53 classes of the class hierarchy. The arguments for the constructors are given also. Classes prefixed with "+" are abstract classes, classes marked with "=" are concrete classes. All classes inherit the instance variable "position" (of type SourcePos) from the base class AST. For space considerations, this argument is omitted in the following list (to be specific, it is the last argument with each constructor).

```
+AST
 =Program(Decl D)

 +Decl
  =EmptyDecl()
  =EmptyFormalParamDecl()
  =FormalParamDecl(Type astType, ID astIdent)
  =FormalParamDeclSequence(Decl lAST, Decl rAST)
  =VarDecl(Type tAST, ID idAST, Expr eAST)
  =DeclSequence(Decl d1AST, Decl d2AST)
  =FunDecl(Type tAST, ID idAST, Decl paramsAST, Stmt stmtAST)

 +Type
  =ArrayType(Type astType, Expr astExpr)
  =IntType()
  =BoolType()
  =FloatType()
  =VoidType()
  =StringType()
  =ErrorType()

 +Expr
  =EmptyExpr()
  =ExprSequence(Expr lAST, Expr rAST)
  =AssignExpr(Expr lAST, Expr rAST)
  =IntExpr(IntLiteral astIL)
  =BoolExpr(BoolLiteral astBL)
  =FloatExpr(FloatLiteral astFL)
  =StringExpr(StringLiteral astSL)
  =ArrayExpr(Expr idAST, Expr indexAST)
  =BinaryExpr(Expr lAST, Operator oAST, Expr rAST)
  =UnaryExpr(Operator oAST, Expr eAST)
  =CallExpr(ID idAST, Expr paramAST)
  =VarExpr(ID Ident)
  =EmptyActualParam()
  =ActualParam(Expr pAST)
  =ActualParamSequence(Expr lAST, Expr rAST)

 +Stmt
  =EmptyStmt()
  =EmptyCompoundStmt()
  =CompoundStmt(Decl astDecl, Stmt astStmt)
  =StmtSequence(Stmt s1AST, Stmt s2AST)
  =ReturnStmt(Expr eAST)
  =AssignStmt(Expr lAST, Expr rAST)
  =IfStmt(Expr eAST, Stmt thenAST)
  =IfStmt(Expr eAST, Stmt thenAST, Stmt elseAST)
  =WhileStmt(Expr eAST, Stmt stmtAST)
  =ForStmt(Expr e1AST, Expr e2AST, Expr e3AST, Stmt stmtAST)
  =CallStmt(Expr eAST)

 +Terminal
  =ID(String Lexeme)
  =Operator(String Lexeme)
  =IntLiteral(String Lexeme)
  =BoolLiteral(String Lexeme)
  =FloatLiteral(String Lexeme)
  =StringLiteral(String Lexeme)
```

AST is the base class for this class hierarchy. Each concrete class provides one constructor (except class IfStmt, which provides 2). Your parser will use these constructors for generating new AST nodes.

Package MiniC.AstGen contains a design pattern in the interface in file visitor.java. This design pattern will be used to traverse the AST (in Assignments 4 and 5). For Assignment 3, understanding of this design pattern is not strictly necessary. But feel free to have a look already now, in case you are curious.

You can draw from a large number of examples to see how to build ASTs for all MiniC language constructs. You find these testcases in Parser/tst/base/AST_testcases. You can also access them at http://elc.yonsei.ac.kr/courses/csi4104/csi4104_assignment_3.html .

- The empty program: c1.mc
- Global variable declarations: c2.mc, c3.mc
- Function declarations: c4.mc, c5.mc
- Local variable declarations: c6.mc, c7.mc, c8.mc
- Statements: c9.mc, c10.mc, c11.mc
- Formal parameters: c12.mc, c13.mc, c14.mc
- Actual parameters: c15.mc, c16.mc, c17.mc
- Assignments: c18.mc, ..., c22.mc
- If statements: c23.mc, ..., c27.mc
- For loops: c28.mc, c29.mc
- While loops: c30.mc, c31.mc
- Multiple variable declarations: c32.mc, c33.mc, c34.mc, c35.mc
- Whole programs: c36.mc, c37.mc
- Arrays:
    - Global variable declarations: c38.mc
    - Local variable declarations: c39.mc
    - Formal array parameters: c40.mc
    - Array expressions: c41.mc

Note that global and local variable declarations use exactly the same class (VarDecl). However, they are displayed differently by the TreeDrawer, depending on whether it is a local declaration or a global declaration. Local declarations are shown as "L.VarDecl", whereas global declarations are shown as "G.VarDecl".

Every declaration in which multiple variables are declared is treated as if the variables had been declared in separate statements (in the order they appear in the source code). The AST for "`int a, b`" is therefore the same as

```
int a;
int b;
```

This holds for local and global variables.

Multiple syntactic constructs of the same kind (e.g., multiple variable declarations as in the above example) are strung together as binary subtrees by *sequence classes*. The sequence classes are `DeclSequence`, `FormalParamDeclSequence`, `ActualParamSequence`, `ExprSequence` and `StmtSequence`. The provided examples show you how the sequence classes are used. With sequence classes, the first construct from the source appears at the top of the tree, and the last construct is always at the bottom of the tree. See e.g., c5.mc, where functions `foo` is the highest in the tree, followed by function `f` and function `g`. Sequences are always terminated by the corresponding Empty* AST node.

The AST class ErrorType will not be used in this assignment. We will use it in Assignment 4.

The AST class StringType will not be used in this assignment, because MiniC does not allow the declaration of variables of type string. In Assignment 4, string literals will be assigned this type during type checking.

Implementing AST Generation in Your Parser

We provide you with an updated version of the skeleton compiler that you should extend to create your MiniC parser. You will find the new skeleton on the server in the archive file /opt/ccugrad/Assignment3/CSI4100_Assignment_3.tgz. Please set up your programming environment as specified in Assignment 1.

In the following, you find a description of the skeleton-files that have been added or changed compared to Assignment 2. All directories are relative to the top-level directory `CSI4100_Assignment_3`:

Package MiniC (directory MiniC)

MiniC.java: the new version of the compiler driver. This version supports now several command line switches (see “Testing your parser” below).

Package Scanner (directory MiniC/Scanner)

Scanner.java: empty scanner implementation which you should **replace** by **your own scanner** from **Assignment 1**. In case your scanner from Assignment 1 does not work properly, you can use scanner classfiles provided with this assignment instead (see Section ``Scanner’’ below).

Package Parser (directory MiniC/Parser)

Parser.java: the skeleton parser for you to complete (you will need to augment this skeleton parser with your parser-code from Assignment2. You should extend your parser-code from Assignment 2 to build ASTs for MiniC input programs). The lecture on Syntax Analysis contains ample examples on AST construction.
tst/base/AST_testcases: the testcases for this assignment. All testcases are syntactically legal MiniC programs.
tst/base/AST_solutions_image: the solutions for this assignment, as png images.
tst/base/AST_solutions_unparsed: the solutions for this assignment as unparsed trees. During marking, we will compare the unparsed solutions produced by your parser to the unparsed solutions provided by our parser.
tst/base/AST_solutions_trees: the solutions for this assignment as ASTs in ASCII format.

The provided parser compiles out-of-the-box. It parses a subset of the MiniC language and handles the following test cases from directory AST_testcases correctly: c1.mc, c4.mc, c5.mc, c6.mc, c9.mc, c12.mc, c13.mc, c14.mc, c40.mc.
After **adding your code from Assignment 2**, you will have to write about 400 lines of additional Java code for building ASTs.

Each AST node represents a language construct, i.e., a phrase, for the MiniC language. The position of a construct can then be defined by using an object of the class `SourcePos` (see also our lecture on Syntax Analysis, in particular, on AST generation). A `SourcePos` object contains four fields:

```
StartLine: the line where the construct begins
EndLine: the line where the construct ends
StartCol: the column where the construct begins
EndCol: the column 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 AST nodes. We use dummy positions for "empty" AST nodes that have no counterpart in the actual input program.

Package TreeDrawer (directory MiniC/TreeDrawer)

The Java classes to draw graphic representations of MiniC ASTs on-screen. It is not necessary to understand how ASTs are actually drawn. However, if you are interested in Java graphics programming, you are encouraged to take a look at this package.
Note that the TreeDrawer abbreviates the names of many AST classes in its output to save space. You can find the exact spellings in file LayoutVisitor.java.

Package AstGen (directory MiniC/AstGen)

This package contains the class hierarchy of AST nodes.

Package Unparser (directory MiniC/Unparser)

This class is written for automated marking and debugging. It contains an unparser that takes an AST as input, visits the AST nodes in depth-first left-to-right order, and produces an equivalent MiniC program. This program is usually different from the original one (see Section “Marking Criteria” below).

Package TreePrinter (directory MiniC/TreePrinter)

The TreePrinter dumps an AST in text format to a file.
The TreeDrawer, TreePrinter and Unparser are implemented using visitor classes. For Assignment 3 knowledge of the Visitor Design Pattern is not needed; we will however use this design pattern for Assignments 4 and 5, based on the discussion in the lecture.

Scanner

In case your scanner from Assignment 1 does not work properly, you can use the scanner classfiles provided in the ``resources'' sub-directory of the CSI4100_Assignment_3.tgz archive. Compile using the "jarNoScanner" build task. See the Assignment 2 specification on further details on the provided scanner classfiles.

Building, Running and Testing Your Parser

Like with the previous assignments, we use the Gradle build automation system to compile our MiniC compiler. The build tasks ``jar'' and ``jarNoScanner'' are the same as with Assignment 2.

The MiniC compiler now accepts a set of command-line switches:

- **-ast** to display the AST on-screen.
- **-astp** to display the AST, including source positions.
- **-t <file>** prints the AST in file <file>, using the TreePrinter.
- **-u <file>** unparses the AST into file <file>.

The provided testcases demonstrate how to build ASTs for MiniC language constructs. You are encouraged to find additional testcases for debugging purposes.

A way to test your parser works as follows:

```
(1.)  java -jar build/libs/MiniC-AstGen.jar –u mytest.mc.u mytest.mc
(2.)  java -jar build/libs/MiniC-AstGen.jar –u mytest.mc.u.u mytest.mc.u
(3.)  diff mytest.mc.u mytest.mc.u.u
```

If the two files differ, then you obviously have a problem in your AST generation. If the two files are the same, there might still be a problem! (Why?)

To make sure that your compiler works correctly for the provided testcases, you are strongly recommended to log in on elc1 and use the Linux **diff** utility to do a byte-by-byte comparison of the ASTs generated by your compiler versus the provided solutions. For example:

```
(1.) java -jar ../build/libs/MiniC-AstGen.jar -t my_c1.mc.ast
     Parser/tst/base/AST_testcases/c1.mc
(2.) diff --brief my_c1.mc.ast
     Parser/tst/base/AST_solutions_trees/c1.mc.ast
(3.) echo $?
     0
```

If diff returns zero (checked by the **echo** command), then the content of your file **`my_c1.mc.ast`** is byte-by-byte identical to the solution in file **`c1.mc.ast`**.

Otherwise, diff will tell you that the files differ. In this case, you can use diff to show the line-by-line differences:

```
diff -u my_c1.mc.ast Parser/tst/base/AST_solutions_trees/c1.mc.ast
```

To find out how to read the output of the diff command, please refer to the GNU diffutils manual.

The most efficient way to test your parser using diff is to adapt the provided test-script of Assignment 1 or Assignment 2. You are allowed to share your **testscript** with our colleagues through our Q&A board.

Please note: ASTs that fail the diff comparison described above will receive zero score.

Marking

Please note that all assignments are *individual* assignments. You are most welcome to discuss your ideas with your colleagues, but ``code-sharing'' and ``code-reuse'' are not allowed. Assignments are designed to facilitate understanding of the course contents. Assignments are also a part of the preparation for the exams.

Your parser will be assessed by examining whether it can build the correct AST for syntactically legal MiniC programs.

You will not be marked up or down for how your parser behaves on syntactically illegal MiniC programs. You will not be marked up or down for the precision of the position information recorded with AST nodes (i.e., position information is optional with this assignment). However, accurate position information may facilitate debugging of your static semantic analysis component that we will develop in Assignment 4.

This assignment will be automatically marked by comparing the AST produced by your compiler (invoked with the – t option) to the solution.

Deliverables and Submission

Please do **not** submit Assignment 3 prematurely! See Section “Due date” at the beginning of this document for the time-period submission is allowed for Assignment 3.

You should extend the parser in file Parser.java so that it constructs ASTs for the complete MiniC specification. Note that for this assignment it is only necessary to modify file Parser.java. Submission will therefore be restricted to this single file. If you need to submit additional files, you must let me know in advance. *You are not allowed to modify other files, e.g., the AST class hierarchy, Token.java aso, otherwise automated testing of your parser might no longer work.*

To submit your file Parser.java, please log in on our server, create a submission directory, e.g., ~/myassignment3, and copy your modified Parser.java to this directory.

After copying relevant files to your submission directory, you must cd to the submission directory and type ‘submit’. This will collect your file(s) and copy it to our repository.

```
[you@elc1 ~] cd ~/myassignment3
[you@elc1 myassignment3] submit
```

Please note: There is no submission unless you issue the ‘submit’ command! Failing to issue the ‘submit’ command will result in 0 points. The timestamp of your submission will be recorded by the ‘submit’ command. Please note also that if you re-submit, then a previous submission will be overwritten and a new timestamp will be recorded.

If you have questions, don’t hesitate to contact us. ☺

Good luck!