Programming Assignment 1 (P1)

Edit: grading rubric

This assignment has two parts. Part 1 is due at **11 PM on Sep 12 2016. Part 1 will not be accepted past this time** (the regular late policy does not apply to Part 1). Part 2 is due at **11 PM on Sep 15 2016**. For Part 2, see the late policy for information on late submissions. Make sure to follow the submission instructions.

*Each student must do this assignment **alone**. You may work in pairs from Programming Assignment 2 (not Part 2) onwards. You may discuss this assignment with the instructor, TA, and other students, but you may not share code.*

Overview

In this assignment, you will write several Java classes to be used later in the semester to represent a symbol table. This is a simple assignment to get you up to speed with your computing environment, Java, and our programming and testing conventions.

Make sure you read through everything carefully. The assignment appears long but it is less work than it might initially seem. If it seems difficult, ask for help; you may be misunderstanding something.

Specifications

For this assignment you will implement four Java classes: `SymTable`, `Sym`, `DuplicateSymException`, and `EmptySymTableException`. You will also write a program called `P1.java` to test your implementations.

The `SymTable` class will be used by the compiler you write later in the semester to represent a *symbol table*: a data structure that stores the identifiers declared in the program being compiled (e.g., function and variable names) and information about each identifier (e.g., its type, where it will be stored at runtime). The symbol table will be implemented as a `List` of `HashMaps`. Eventually, each `HashMap` will store the identifiers declared in one scope in the program being compiled.

The `HashMap` keys will be `Strings` (the declared identifier names) and the associated information will be `Syms` (you will also implement the `Sym` class). For now, the only information in a `Sym` will be the type of the identifier, represented using a `String` (e.g., `"int"`, `"double"`, etc.).

The `DuplicateSymException` and `EmptySymTableException` classes will define exceptions that can be thrown by methods of the `SymTable` class.

In addition to defining the four classes, you will write a main program to test your implementation. You will be graded on the correctness of your `Sym` and `SymTable` classes, on how thoroughly you test the classes that you implement, on the efficiency of your code, and on your programming style.

The `Sym` Class

The `Sym` class must be in a file named `Sym.java`. You must implement the following `Sym` constructor and public methods (and no other public or protected methods):

`Sym(String type)`	This is the constructor; it should initialize the `Sym` to have the given `type`.
`String getType()`	Return this `Sym`'s type.

`String toString()`	Return this `Sym`'s type. (This method will be changed later in a future project when more information is stored in a `Sym`.)

The `SymTable` Class

The `SymTable class` must be in a file named `SymTable.java`. It must be implemented using a `List` of `HashMaps`. (Think about the operations that will be done on a `SymTable` to decide whether to use an `ArrayList` or a `LinkedList`.) The `HashMaps` must map a `String` to a `Sym`. This means that the `SymTable` class will have a (private) field of type `List<HashMap<String,Sym>>`.

`List` and `HashMap` are defined in the `java.util` package. This means that you will need to have the line

```
import java.util.*;
```

at the top of `SymTable.java`.

You must implement the following `SymTable` constructor and public methods (and no other public or protected methods):

`SymTable()`	This is the constructor; it should initialize the `SymTable`'s List field to contain a single, empty `HashMap`.
`void addDecl(String name, Sym sym) throws DuplicateSymException, EmptySymTableException`	If this `SymTable`'s list is empty, throw an `EmptySymTableException`. If either `name` or `sym` (or both) is `null`, throw a `NullPointerException`. If the first `HashMap` in the list already contains the given name as a key, throw a `DuplicateSymException`. Otherwise, add the given `name` and `sym` to the first `HashMap` in the list.
`void addScope()`	Add a new, empty `HashMap` to the front of the list.
`Sym lookupLocal(String name)`	If this `SymTable`'s list is empty, throw an `EmptySymTableException`. Otherwise, if the first `HashMap` in the list contains name as a key, return the associated `Sym`; otherwise, return `null`.
`Sym lookupGlobal(String name)`	If this `SymTable`'s list is empty, throw an `EmptySymTableException`. If **any** `HashMap` in the list contains `name` as a key, return the first associated `Sym` (i.e., the one from the `HashMap` that is closest to the front of the list); otherwise, return `null`.
`void removeScope() throws EmptySymTableException`	If this `SymTable`'s list is empty, throw an `EmptySymTableException`; otherwise, remove the `HashMap` from the front of the list. To clarify, throw an exception only if *before attempting to remove*, the list is empty (i.e. there are no `HashMaps` to remove).
`void print()`	This method is for debugging. First, print `"\nSym Table\n"`. Then, for each `HashMap M` in the list, print `M.toString()` followed by a newline. Finally, print one more newline. All output should go to `System.out`.

The `DuplicateSymException` and `EmptySymTableException` Classes

These two classes (which must be in files named `DuplicateSymException.java` and `EmptySymTableException.java`) will simply define the two checked exceptions that can be thrown by the `SymTable` class. Each exception must be able to be created using a constructor that takes no arguments.

To define a checked exception named `XXX`, you can use code like this:

```
public class XXX extends Exception {
}
```

Note that the class has an empty body (it will have a no-argument constructor by default).

The `main` program

To test your `SymTable` implementation, you will write a main program in a file named `P1.java`. The program must not expect any command-line arguments or user input. It can read from one or more files; if you set it up to do that, be sure to hand in the file(s) along with P1.java.

Be sure that your `P1.java` tests all of the `Sym` and `SymTable` operations and all situations under which exceptions are thrown. Also think about testing both "boundary" and "non-boundary" cases.

It is up to you how your program works. A suggested approach is to write your program so that output is only produced if one of the methods that it is testing does ***not*** work as expected (e.g., if the `lookupLocal` method of the `SymTable` class returns `null` when you expect it to return a non-null value). This will make it much easier to determine whether your test succeeds or fails. The one exception to this approach is that `P1.java` will need to test the `print` method of the `SymTable` class and that will cause output to be produced.

To help you understand better the kind of code you would write using this suggested approach, look at TestList.java. This file contains a main program designed to test a (fictional) `List` class whose methods are documented in `TestList.java`. You are being asked to write something similar (in a file called `P1.java`) to test the `Sym` and `SymTable` classes. You should be able to write `P1.java` *before* you write the classes that it's designed to test.

Test Code

After the Part 1 deadline, download our P1.java file and test it against the expected output. Make sure that your actual output matches this.

On a Linux machine you can see whether two files match by using the `diff` utility. For example, typing `diff file1 file2` compares the two files `file1` and `file2`. Typing `diff -b -B file1 file2` does the same comparison, but ignores differences in whitespace.

If you send the output of `P1.java` to a file, you can use `diff` to make sure that it matches the expected output. To send the output of `P1.java` to a file named `out.txt` (on a Linux machine) type `java P1 >| out.txt`.

Handing In

Deadlines are at the top of the page. See instructions for submitting assignments.

By the Part 1 deadline, submit your `P1.java` file (and the files that it reads, if any).
By the Part 2 deadline, submit the rest of your `.java` files. This should include your `Sym.java`, `SymTable.java`, `DuplicateSymException.java`, and `EmptySymTable.java`.

You may work in the environment of your choice, but be aware that your submitted code must run on the department lab Linux machines.

Do not turn in any .class files and do not create any subdirectories in your submission. If you accidentally turn in (or create) extra files or subdirectories, make a new submission that does not include them.

Remember, your `P1.java` is worth 15% of the grade for this assignment and will not be accepted past the deadline.

Grading Criteria

For this program, extra emphasis will be placed on ***style***. In particular,

- Every class, method, and field must have a comment that describes its purpose. Comments should also be used to explain anything that would not be obvious to an experienced Java programmer who has read this assignment.

- Identifiers must conform to standard Java conventions. `UPPER_CASE` with underscores for named constants, `CamelCase` starting with a capital letter for classes, and `camelCase` starting with a lower-case letter for other identifiers. Names should help a reader to understand the code.

- Indentation must be consistent and clear. Use either one tab character or four spaces for each level of indentation. Do not mix spaces and tabs for indentation; either always use tabs or never use them.

- Avoid lines that are longer than 80 characters (including indentation).

- Each field or method must be declared public, protected, or private. If you have good reason to give a field or method "package" (default) access – which is highly unlikely – you must include a comment explaining why.

The goal is to make your code readable to an experienced Java programmer who is used to the conventions. The goal is not to develop your own personal style, even if it's "better" than the standard. For more advice on Java programming style, see Code Conventions for the Java Programming Language. See also the style and commenting standards used in CS 302 and CS 367.

Also be very sure that you use the specified file names (being careful about the upper- and lower-case letters in those names). And be sure that the output that is produced when we run our `P1.java` using your implementations of the `Sym`, `SymTable`, `DuplicateSymException`, and `EmptySymTableException` classes matches the expected output that we provide. We will test that output by automatically comparing it to the expected output and you will lose points for even minor differences.

Program 2

Overview

Due by Sep 30 at 11pm.

For this assignment you will use JLex to write a scanner for our language, called `harambe`, a small subset of the C++ language. Features of `harambe` that are relevant to this assignment are described below. You will also write a main program (`P2.java`) to test your scanner. You will be graded both on the correctness of your scanner and on how thoroughly your main program tests the scanner.

Specifications

Getting Started

Dowload the skeleton code for this project here. Once you unzip it, you will see two folders *deps* and *files*. To compile your project, run the makefile inside the *files* folder.
Note that you need to have the *deps* folder to make this project without errors.

The *files* folder contains the following.

- `cats.jlex`: An example JLex specification. *You will need to add to this file*.
- `sym.java`: Token definitions (this file will eventually be generated by the parser generator). Do not change this file.
- `ErrMsg.java`: The `ErrMsg` class will be used to print error and warning messages. Do not change this file.
- `P2.java`: Contains the main program that tests the scanner. *You will need to add to this file*.
- `Makefile`: A Makefile that uses JLex to create a scanner, and also makes `P2.class`. *You may want to change this file*.

JLex

Use the on-line JLex reference manual, and/or the on-line JLex notes for information about writing a JLex specification.

If you work on a CS Dept. Linux machine, you should have no problem running JLex. You will not be able to work on the CS Dept. Windows machines.

The Language

This section defines the lexical level of the `cats` language. At this level, we have the following language issues:

Tokens

The tokens of the `cats` language are defined as follows:

- Any of the following reserved words (remember that you will need to give the JLex patterns for reserved words *before* the pattern for identifier):

```
bool  int   void  true  false  struct
cin   cout  if    else  while  return
```

- Any identifier (a sequence of one or more letters and/or digits, and/or underscores, starting with a letter or underscore, excluding reserved words).
- Any integer literal (a sequence of one or more digits).
- Any string literal (a sequence of zero or more *string* characters surrounded by double quotes). A *string* character is either
  - an escaped character: a backslash followed by any one of the following six characters:
    1. `n`
    2. `t`
    3. a single quote
    4. a double quote
    5. a question mark
    6. another backslash

    or
  - a single character other than new line or double quote or backslash.

  Examples of legal string literals:

```
""
"&!88"
"use \n to denote a newline character"
"include a quote like this \" and a backslash like this \\"
```

  Examples of things that are ***not*** legal string literals:

```
"unterminated
"also unterminated \"
"backslash followed by space: \ is not allowed"
"bad escaped character: \a AND not terminated
```

- Any of the following one- or two-character symbols:

```
{       }       (       )       ;
,       .       <<      >>      ++
--      +       -       *       /
!       &&      ||      ==      !=
<       >       <=      >=      =
```

Token "names" (i.e., values to be returned by the scanner) are defined in the file [sym.java](). For example, the name for the token to be returned when an integer literal is recognized is `INTLITERAL` and the token to be returned when the reserved word `int` is recognized is `INT`.

Note that code telling JLex to return the special `EOF` token on end-of-file has already been included in the file `cats.jlex` -- you don't have to include a specification for that token. Note also that the `READ` token is for the 2-character symbol >> and the `WRITE` token is for the 2-character symbol <<

If you are not sure which token name matches which token, ask!

Comments

Text starting with a double slash (`//`) or a sharp sign (`#`) up to the end of the line is a comment (except of course if those characters are inside a string literal). For example:

```
// this is a comment
# and so is this
```

The scanner should recognize and ignore comments (but there is no `COMMENT` token).

Whitespace

Spaces, tabs, and newline characters are whitespace. Whitespace separates tokens and changes the character counter, but should otherwise be ignored (except inside a string literal).

Illegal Characters

Any character that is not whitespace and is not part of a token or comment is illegal.

Length Limits

You may not assume any limits on the lengths of identifiers, string literals, integer literals, comments, etc.

What the Scanner Should Do

The main job of the scanner is to identify and return the next token. The value to be returned includes:

- The token "name" (e.g., `INTLITERAL`). Token names are defined in the file `sym.java`.
- The line number in the input file on which the token starts.
- The number of the character on that line at which the token starts.
- For identifiers, integer literals, and string literals: the actual value (a `String`, an `int`, or a `String`, respectively). For a string literal, the value should include the double quotes that surround the string, as well as any backslashes used inside the string as part of an "escaped" character.

Your scanner will return this information by creating a new `Symbol` object in the action associated with each regular expression that defines a token (the `Symbol` type is defined in `java_cup.runtime`; you don't need to look at that definition). A `Symbol` includes a field of type `int` for the token name, and a field of type `Object` (named `value`), which will be used for the line and character numbers and for the token value (for identifiers and literals). See `cats.jlex` for examples of how to call the `Symbol` constructor. See `P2.java` for code that accesses the fields of a `Symbol`.

In your compiler, the `value` field of a `Symbol` will actually be of type `TokenVal`; that type is defined in `cats.jlex`. Every `TokenVal` includes a `linenum` field, and a `charnum` field (line and character numbers start counting from 1, not 0). Subtypes of `TokenVal` with more fields will be used for the values associated with identifier, integer literal, and string literal tokens. These subtypes, `IntLitTokenVal`, `IdLitTokenVal`, and `StrLitTokenVal` are also defined in `cats.jlex`.

Line counting is done by the scanner generated by JLex (the variable `yyline` holds the current line number, counting from 0), but you will have to include code to keep track of the current character number on that line. The code in `cats.jlex` does this for the patterns that it defines, and you should be able to figure out how to do the same thing for the new patterns that you add.

The JLex scanner also provides a method `yytext` that returns the actual text that matches a regular expression. You will find it useful to use this method in the actions you write in your JLex specification.

Note that, for the integer literal token, you will need to convert a `String` (the value scanned) to an `int` (the value to be returned). You should use code like the following:

```
double d = (new Double(yytext())).doubleValue(); // convert String to double
// INSERT CODE HERE TO CHECK FOR BAD VALUE -- SEE ERRORS AND WARNINGS BELOW
int k =  (new Integer(yytext())).intValue();    // convert to int
```

Errors and Warnings

The scanner should handle the following errors as indicated:

Illegal characters

Issue the error message: `illegal character ignored: ch` (where `ch` is the illegal character) and ignore the character.

Unterminated string literals

A string literal is considered to be unterminated if there is a newline or end-of-file before the closing quote. Issue the error message: `unterminated string literal ignored` and ignore the unterminated string literal (start looking for the next token after the newline).

Bad string literals

A string literal is "bad" if it includes a bad "escaped" character; i.e., a backslash followed by something other than an `n`, a `t`, a single quote, a double quote, another backslash, or a question mark. Issue the error message: `string literal with bad escaped character ignored` and ignore the string literal (start looking for the next token after the closing quote). If the string literal has a bad escaped character *and* is unterminated, issue the error message `unterminated string literal with bad escaped character ignored`, and ignore the bad string literal (start looking for the next token after the newline). Note that a string literal that has a newline immediately after a backslash should be treated as having a bad escaped character and being unterminated. For example, given:

```
"very bad string \
abc
```

the scanner should report an unterminated string literal with a bad escaped character on line 1, and an identifier on line 2.

Bad integer literals **(integer literals larger than `Integer.MAX_VALUE`)**

Issue the warning message: `integer literal too large; using max value` and return `Integer.MAX_VALUE` as the value for that token.

For unterminated string literals, bad string literals, and bad integer literals, the line and column numbers used in the error message should correspond to the position of the *first* character in the string/integer literal.

Use the `fatal` and `warn` methods of the `ErrMsg` class to print error and warning messages. Be sure to use *exactly* the wording given above for each message so that the output of your scanner will match the output that we expect when we test your code.

The Main Program

In addition to specifying a scanner, you should extend the main program in `P2.java`. The program opens a file called `allTokens.in` for reading; then the program loops, calling the scanner's `next_token` method until the special end-of-file token is returned. For each token, it writes the corresponding lexeme to a file called `allTokens.out`. You can use `diff` to compare the input and output files (`diff allTokens.in allTokens.out`). If they differ, you've found an error in the scanner. Note that you will need to write the `allTokens.in` file.

Testing

Part of your task will be to figure out a strategy for testing your implementation. As mentioned in the Overview, part of your grade will be determined by how thoroughly your main program tests your scanner.

You will probably want to change `P2.java` to read multiple input files so that you can test other features of the scanner. You will need to create a new scanner each time and you will need to set `CharNum.num` back to one each time (to get correct character numbers for the first line of input). Note that the input files do *not* have to be legal `cats` or C++ programs, just sequences of characters that correspond to `cats` tokens. **Don't forget to include code that tests whether the correct character number (as well as line number) is returned for every token!**

Your `P2.java` should exercise all of the code in your scanner, including the code that reports errors. Add to the provided `Makefile` (as necessary) so that running `make test` runs your `P2` and does any needed file comparisons (e.g., using `diff`) and running `make cleantest` removes any files that got created by your program when `P2` was run. It should be clear from what is printed to the console when `make test` is run what errors have been found.

To test that your scanner correctly handles an unterminated string literal with end-of-file before the closing quote, you may use the file `files/eof.txt`. On a Linux machine, you can tell that there is no final newline by typing: `cat eof.txt` You should see your command-line prompt at the *end* of the last line of the output instead of at the beginning of the following line.

Working in Pairs

Computer Sciences and Computer Engineering graduate students must work alone on this assignment. Undergraduates, special students, and graduate students from other departments may work alone or in pairs.

If you plan to work with a partner, you must let us know ***no later than September 22nd***. To let us know, each partner should hand in a README.txt file, through learn@UW into the p2 submission, filled in with the name and CS login of *both* partners.

If you want to work with a partner, but don't have one, check out the "Search for Teammates!" note in Piazza.

If you are working with a partner and you decide to split up, you must let the TAs know by email so that we can arrange how to divide up any code that has already been written.

Below is some advice on how to work in pairs.

This assignment involves two main tasks:

1. Writing the scanner specification (`cats.jlex`).
2. Writing the main program (`P2.java`).

An excellent way to work together is to do *pair programming*: Meet frequently and work closely together on both tasks. Sit down together in front of a computer. Take turns "driving" (controlling the keyboard and mouse) and "verifying" (watching what the driver does and spotting mistakes). Work together on all aspects of the project: design, coding, debugging, and testing. Often the main advantage of having a partner is not having somebody to write half the code, but having somebody to bounce ideas off of and to help spot your mistakes.

If you decide to divide up the work, you are *strongly* encouraged to work together on task (1) since both partners are responsible for learning how to use JLex. You should also work together on testing; in particular, you should each test the other's work.

Here is one reasonable way to divide up the project:

- Divide up the tokens into two parts, one part for each person.
- Each person extends their own copy of `cats.jlex` by adding rules for their half of the tokens, and extends their own copy of the main program to handle those same tokens.
- Decide together how your `P2.java` should work, and write that code.
- Write test input files for your own tokens, and for the other person's tokens, too.
- After each person makes sure that their scanner and main program work on their own tokens, combine the two (it should be pretty easy to cut and paste one person's JLex rules into the other person's `cats.jlex`, and similarly for the main program).
- Do *not* try to implement all of your half of the tokens at once. Instead, implement just a few to start with to make sure that you both know what you're doing and that you're able to combine your work easily.

The most challenging JLex rules are for the `STRINGLITERAL` token (for which you will need several rules: for a correct string literal, for an unterminated string literal, for a string literal that contains a bad escaped character, and for a string literal that contains a bad escaped character *and* is unterminated). Be sure to divide these up so that each person gets to work on some of them.

It is **very** important to set deadlines and to stick to them. I suggest that you choose one person to be the "project leader" (plan to switch off on future assignments). The project leader should propose a division of tokens, as well as deadlines for completing phases of the program, and should be responsible for keeping the most recent version of the combined code (be sure to keep back-up versions, too, perhaps in another directory or using a version-control system like Mercurial or Git).

To share your code, you can either use e-mail or the project leader can create a directory for the combined code (*not* the directory in which that person develops the code). I suggest that you create a new top-level directory (i.e., at the same level as your `public` and `private` directories), named something like `cs536-P2`. To set the permissions of the directory for the combined code to allow your partner to write into it, change to that directory and type:

```
fs setacl . <login> write
```

using your partner's CS login in place of <login>. You should also prevent any other access by typing:

```
fs setacl . system:anyuser none
```

in the new directory that you create (*not* in your top-level directory). To see what the permissions are in your current directory, type:

```
fs listacl
```

Do *not* try to share by letting your partner log in to your account. Departmental and University policy prohibits your revealing your password to anybody else, including your partner.

Handing in

You will be needed to submit the the entire working folder (all the java files, jlex file, Makefile and pdf) as a compressed file. Please look into Handing Instructions here

Grading criteria

General information on program grading criteria can be found on the Assignments page).

For more advice on Java programming style, see Code Conventions for the Java Programming Language. See also the style and commenting standards used in CS 302 and CS 367.

CS536 Programming Assignment 3
Due on Oct 17th, 11pm

Overview

For this assignment you will use the parser-generator **Java Cup** to write a parser for the `harambe` language. The parser will find syntax errors and, for syntactically correct programs, it will build an abstract-syntax tree (AST) representation of the program. You will also write methods to **unparse** the AST built by your parser and an input file to test your parser. A main program, `P3.java`, that calls the parser and then the unparser is provided for you to use. You will be graded on the correctness of your parser and your unparse methods and on how thoroughly your input file tests the parser. In particular, you should write an input file that causes the action associated with every grammar rule in your Java CUP specification to be executed at least once.

Specifications

Getting Started

Skeleton files on which you should build are in the following tar file:

p3.tar.gz contains all files below.

- `harambe.jlex`: A JLex specification for the harambe language (a solution to program 2). Use this if there were problems with your JLex specification.
- `harambe.cup`: A Java CUP specification for a very small subset of the `harambe` language (you will need to add to this file).
- `harambe.grammar`: A CFG for the `harambe` language. Use this to guide the enhancements you make to `harambe.cup`.
- `ast.java`: Contains class definitions for the AST structure that the parser will build (you will need to add unparsing code to this file, but you should *not* add any new classes, fields, or methods).
- `P3.java`: The main program that calls the parser, then, for a successful parse, calls the unparser (no changes needed).
  Use `make test` to run P3 using `test.cf` as the input, and sending the unparsed output to file `test.out`. Alternatively run it as follows:

  ```
  java P3 test.cf test.out
  ```

- `Makefile`: A Makefile for program 3 (no changes needed).
- `test.cf`: Input for the current version of the parser (you will need to change this file).
- `ErrMsg.java`: Same as for program 2 (no changes needed).
- **`configure.sh`: This script configures your environment variable CLASSPATH for CS computers. Run it as follows:**

```
$. configure.sh
```

That is, type "." followed by a space followed by "`configure.sh`" in the command line.
To set up the environment on your own computer, please see THIS LINK in the resources section

Here is a link to the Java CUP reference manual. There is also a link in the "Tools" section of the "Quick Links" menu on the course website.

Operator Precedences and Associativities

The harambe grammar in the file `harambe.grammar` is ambiguous; it does not uniquely define the precedences and associativities of the arithmetic, relational, equality, and logical operators. You will need to add appropriate precedence and associativity declarations to your Java CUP specification.

- Assignment is right associative.
- The dot operator is left associative.
- The relational and equality operators (<, >, <=, >=, ==, and !=) are non-associative (i.e., expressions like `a < b < c` are not allowed and should cause a syntax error).
- All of the other binary operators are left associative.
- The unary minus and not (!) operators have the highest precedence, then multiplication and division, then addition and subtraction, then the relational and equality operators, then the logical *and* operator (&&), then the logical *or* operator (||), and finally the assignment operator (=).

Note that the same token (MINUS) is used for both the unary and binary minus operator, and that they have different precedences; however, the harambe grammar has been written so that the unary minus operator has the correct (highest) precedence; therefore, you can declare MINUS to have the precedence appropriate for the binary minus operator.

Java Cup will print a message telling you how many *conflicts* it found in your grammar. If the number is not zero, it means that your grammar is still ambiguous and the parser is unlikely to work correctly. **Do not ignore this!** Go back and fix your specification so that your grammar is not ambiguous.

Building an Abstract-Syntax Tree

To make your parser build an abstract-syntax tree, you must add new productions, declarations, and actions to `harambe.cup`. You will need to decide, for each nonterminal that you add, what type its associated value should have. Then you must add the appropriate nonterminal declaration to the specification. For most nonterminals, the value will either be some kind of tree node (a subclass of `ASTnode`) or a `LinkedList` of some kind of node (use the information in `ast.java` to guide your decision). Note that you cannot use parameterized types for the types of nonterminals; so if the translation of a nonterminal is a `LinkedList` of some kind of node, you will have to declare its type as just plain `LinkedList`.

You must also add actions to each new grammar production that you add to `harambe.cup`. Make sure that each action ends by assigning an appropriate value to `RESULT`. Note that the parser will return a `Symbol` whose `value` field contains the value assigned to `RESULT` in the production for the root nonterminal (nonterminal `program`).

Unparsing

To test your parser, you must write the `unparse` methods for the subclasses of `ASTnode` (in the file `ast.java`). When the `unparse` method of the root node of the program's abstract-syntax tree is called, it should print a nicely

formatted version of the program (this is called *unparsing* the abstract-syntax tree). The output produced by calling `unparse` should be the same as the input to the parser except that:

1. There will be no comments in the output.
2. The output will be "pretty printed" (newlines and indentation will be used to make the program readable); and
3. Expressions will be fully parenthesized to reflect the order of evaluation.

For example, if the input program includes:

```
if (b == -1) { x = 4+3*5-y; while (c) { y = y*2+x; } } else { x = 0; }
```

the output of `unparse` should be something like the following:

```
if ((b == (-1))) {
    x = ((4 + (3 * 5)) - y);
    while (c) {
        y = ((y * 2) + x);
    }
}
else {
    x = 0;
}
```

To make grading easier, put open curly braces on the *same* line as the preceding code and put closing curly braces on a line with no other code (as in the example above). Put the first statement in the body of an `if` or `while` on the line following the open curly brace. Whitespace within a line is up to you (as long as it looks reasonable).

Note: Trying to unparse a tree will help you determine whether you have built the tree correctly in the first place. Besides looking at the output of your unparser, you should try using it as the input to your parser; if it doesn't parse, you've made a mistake either in how you built your abstract-syntax tree or in how you've written your unparser.

Another good way to test your code is to try compiling the output of your unparser using the C++ compiler (g++). If your input program uses I/O (`cin` or `cout`), you will first need to add: `#include <iostream>` at the beginning of the file.

It is a good idea to work incrementally (see Suggestions for How to Work on This Assignment below for more detailed suggestions):

- Add a few grammar productions to `harambe.cup`.
- Write the corresponding `unparse` operations.
- Write a test program that uses the new language constructs.
- Create a parser (using `make`) and run it on your test program.

Modifying `ast.java`

We will test your program by using our `unparse` methods on your abstract-syntax trees and by using your `unparse` methods on our abstract-syntax trees. To make this work, you will need to:

1. Modify `ast.java` **only** by filling in the bodies of the `unparse` methods (and you must fill in all of the method bodies).
2. Make sure that no `LinkedList` field is null (i.e., when you call the constructor of a class with a `LinkedList` argument, that argument should never be null). Note that it is OK to make the `ExpNode` field of a `ReturnStmtNode` null (when no value is returned), likewise for the `ExpListNode` field of a `CallExpNode` (when the call has no arguments).

3. Follow the convention that the `mySize` field of a `VarDeclNode` has the value `VarDeclNode.NOT_STRUCT` if the type of the declared variable is a non-struct type.

Testing

Part of your task will be to write an input file called `test.cf` that thoroughly tests your parser and your unparser. You should be sure to include code that corresponds to every grammar rule in the file `harambe.grammar`.

Note that since you are to provide only *one* input file, `test.cf` should contain no syntax errors (you should also test your parser on some bad inputs, but don't hand those in).

You will probably find it helpful to use comments in `test.cf` to explain what aspects of the parser are being tested, but your testing grade will depend only on how thoroughly the file tests the parser.

Suggestions for How to Work on This Assignment

This assignment involves three main tasks:

1. Writing the parser specification (`harambe.cup`).
2. Writing the unparse methods for the AST nodes (in `ast.java`).
3. Writing an input file (`test.cf`) to test your implementation.

If you work with a partner, it is a good idea to share responsibility for all tasks to ensure that both partners understand all aspects of the assignment.

I suggest that you proceed as follows, testing your parser after each change (if you are working alone, I still suggest that you follow the basic steps outlined below, just do them all yourself):

- Working together, start by making a very small change to `harambe.cup`. For example, add the rules and actions for:

  ```
  type ::= BOOL
  type ::= VOID
  ```

  Also update the appropriate `unparse` method in `ast.java`. Make sure that you can create and run the parser after making this small change. (To create the parser, just type `make` in the directory where you are working.)

- Next, add the rules needed to allow struct declarations.

- Next, add the rules needed to allow programs to include functions with no formal parameters and with empty statement lists only, and update the corresponding `unparse` methods.

- Still working together, add the rules (and `unparse` methods) for the simplest kind of expressions -- just plain identifiers.

- Now divide up the statement nonterminals into two parts, one part for each person.

- Each person should extend their own copy of `harambe.cup` by adding rules for their half of the statements, and should extend their own copy of `ast.java` to define the unparse methods needed for those statements.

- Write test inputs for your statements and your partner's statements.

- After each person makes sure that their parser and unparser work on their own statements, combine the two by cutting and pasting one person's grammar rules into the other person's `harambe.cup` (and similarly

for `ast.java`).

- Now divide up the expression nonterminals into two parts and implement those using a similar approach. Note that you will also need to give the operators the right precedences and associativities during this step (see above).
- Divide up any remaining productions that need to be added, and add them.
- Talk about what needs to be tested and decide together what your final version of `test.cf` should include.
- When working on your own, do *not* try to implement all of your nonterminals at once. Instead, add one new rule at a time to the Java CUP specification, make the corresponding changes to the `unparse` methods in `ast.java`, and test your work by augmenting your `test.cf` or by writing a `harambe` program that includes the new construct you added, and make sure that it is parsed and unparsed correctly.

If you worked alone on the previous program and are now working with a partner, see programming assignment 2 for more suggestions on how to work in pairs.

Handing in

Submit all of the files that are needed to create and run your parser and your main program (including your `test.cf`) as well as your `Makefile`.

Do not turn in any `.class` files and do not create any subdirectories in your submission.

CS536 Programming assignment 4

Due: November 4 at 11PM

Overview

For this assignment you will write a name analyzer for harambe programs represented as abstract-syntax trees. Your main task will be to write *name analysis* methods for the nodes of the AST. In addition you will need to:

1. Modify the `Sym` class from program 1 (by including some new fields and methods and/or by defining some subclasses).
2. Modify the `IdNode` class in `ast.java` (by including a new `Sym` field and by modifying its `unparse` method).
3. Write a new main program, `P4.java` (an extension of `P3.java`).
4. Modify the `ErrMsg` class.
5. Update the `Makefile` used for program 3 to include any new rules needed for program 4.
6. Write two test inputs: `nameErrors.cf` and `test.cf` to test your new code.

Specifications

Tar file here: P4.tar.gz. Extract using `tar -xvzf P4.tar.gz`

The files included are:

- `SemSym.java`: Use this code if there were problems with your own version from program 1.
- `SymTable.java`: Use this code if there were problems with your own version from program 1.
- `DuplicateSymException.java`: Use this code if there were problems with your own version from program 1.
- `EmptySymTableException.java`: Use this code if there were problems with your own version from program 1.
- `harambe.cup`: Use this code if there were problems with your own version from program 3.
- `ast.java`: Use this code if there were problems with your own version from program 3. You will need to add to this file or to your own version.

You will also need a JLex file, ErrMsg.java, and Makefile. As detailed below, you can begin by copying these over from previous assignments.

NOTE: some environments seem to be having trouble with the fact that Sym.java and sym.java differ only in case. To address this, in this assignment, Sym.java is instead called SemSym.java (for semantic symbol).

Name Analysis

The name analyzer will perform the following tasks:

1. **Build symbol tables.** You will use the "list of hashtables" approach (using the `SymTable` class from program 1).

2. **Find multiply declared names, uses of undeclared names, bad `struct` accesses, and bad declarations.** Like C, the harambe language allows the same name to be declared in non-overlapping or nested scopes. The formal parameters of a function are considered to be in the same scope as the function body. All names must be declared before they are used. A *bad* `struct` access happens when either the left-hand side of the dot-access is not a name already declared to be of a `struct` type or the right-hand side of the dot-access is not the name of a field for the appropriate type of `struct`. A *bad* declaration is a declaration of anything other than a function to be of type `void` as well as the declaration of a variable to be of a *bad* `struct` type (the name of the `struct` type doesn't exist or is not a `struct` type).

3. **Add `IdNode` links**: For each `IdNode` in the abstract-syntax tree that represents a *use* of a name (not a declaration) add a "link" to the corresponding symbol-table entry. (As stated above, you will need to modify the `IdNode` class in `ast.java` to have a new field of type `Sym`. That is the field that your name analyzer will fill in with a link to the `Sym` returned by the symbol table's `globalLookup` method.)

You must implement your name analyzer by writing appropriate methods for the different subclasses of `ASTnode`. Exactly what methods you write is up to you (as long as they do name analysis as specified). For your reference, a partially complete name analysis method is defined in ProgramNode.

It may help to start by writing the name analysis method for `ProgramNode`, then work "top down", adding a method for `DeclListNode` (the child of a `ProgramNode`), then for each kind of `DeclNode` (except `StructDeclNode`), and so on (and then handle `StructDeclNode` and perhaps other `struct` related nodes at the end). Be sure to think about which nodes' methods need to add a new hashtable to the symbol table (i.e., when is a new scope being entered) and which methods need to remove a hashtable from the symbol table (i.e., when is a scope being exited).

Some of the methods will process the declarations in the program (checking for bad declarations and checking whether the names are multiply declared, and if not, adding appropriate symbol-table entries) and some will process the statements in the program (checking that every name used in a statement has been declared and adding links). Note that you should *not* add a link for an `IdNode` that represents a use of an undeclared name.

`struct` Handling Issues

Name analysis issues surrounding `struct`s come up in several situations:

- **Defining a `struct` type**: for example

  ```
  struct Point {
      int x;
      int y;
  };
  ```

  When defining a `struct`, the name of the `struct` type can't be a name that has already been declared. The fields of a `struct` must be unique to that particular `struct`; however, they can be a name that has been declared outside of the `struct` definition. For this reason, a recommended approach is to have a separate symbol table associated with each `struct` definition and to store this symbol table in the symbol for the name of the `struct` type.

- **Declaring a variable to be of a `struct` type**: for example

  ```
  struct Point pt;
  ```

  When declaring a variable of a `struct` type, in addition to determining if the variable name has been previously declared (and issuing a "multiply declared" error if it is), you should also check that the name of the `struct` type has been previously declared and is actually the name of a `struct` type.

- **Accessing the fields of a `struct`**: for example

  ```
  pt.x = 7;
  ```

When doing name analysis on something like *LHS.RHS*, you will need to check that *LHS* can be used as a struct (for example, a variable that declared as struct or a nested struct field and that *RHS* is the name of a field in the `struct` type associated with *LHS*. You should also add a field of type `Sym` to the DotAccessExpNode to link the DotAccessExpNode to the symbol being accessed.

Error Reporting

Your name analyzer should find all of the errors described in the table given below; it should report the specified position of the error, and it should give *exactly* the specified error message (each message should appear on a single line, rather than how it is formatted in the following table). Error messages should have the same format as in the scanner and parser (i.e., they should be issued using a call to `ErrMsg.fatal`).

If a declaration is both "bad" (e.g., a non-function declared `void`) and is a declaration of a name that has already been declared in the same scope, you should give *two* error messages (first the "bad" declaration error, then the "multiply declared" error).

Type of Error	Error Message	Position to Report
More than one declaration of an identifier in a given scope (note: includes identifier associated with a `struct` definition)	`Multiply declared identifier`	The first character of the ID in the duplicate declaration
Use of an undeclared identifier	`Undeclared identifier`	The first character of the undeclared identifier
Bad `struct` access (LHS of dot-access is not of a `struct` type)	`Dot-access of non-struct type`	The first character of the ID corresponding to the LHS of the dot-access.
Bad `struct` access (RHS of dot-access is not a field of the appropriate a `struct`)	`Invalid struct field name`	The first character of the ID corresponding to the RHS of the dot-access.
Bad declaration (variable or parameter of type `void`)	`Non-function declared void`	The first character of the ID in the bad declaration.
Bad declaration (attempt to declare variable of a bad `struct` type)	`Invalid name of struct type`	The first character of the ID corresponding to the `struct` type in the bad declaration.

Note that the names themselves should *not* be printed as part of the error messages.

During name analysis, if a function name is multiply declared you *should* still process the formals and the body of the function; don't add a new entry to the current symbol table for the function, but do add a new hashtable to the front of the `SymTable`'s list for the names declared in the body (i.e., the parameters and other local variables of the function).

If you find a bad variable declaration (a variable of type `void` or of a bad `struct` type), give an error message and add nothing to the symbol table.

Other Tasks

Extending the `Sym` Class

It is up to you how you store information in each symbol-table entry (each `Sym`). To implement the changes to the unparser described below you will need to know each name's type. For function names, this includes the return type and the number of parameters and their types. You can modify the `Sym` class by adding some new fields (e.g., a `kind` field) and/or by declaring some subclasses (e.g., a subclass for functions that has extra fields for the return type and the list of parameter types). You will probably also want to add new methods that return the values of the new fields and it may be helpful to change the `toString` method so that you can print the contents of a `Sym` for debugging purposes.

Modifying the `IdNode` Class

Two changes to the `IdNode` class are needed:

1. Adding a new field of type `Sym` (to link the node with the corresponding symbol-table entry), and
2. Changing the unparse method so that every use of an ID has its type (in parentheses) after its name. (The point of this is to help you to see whether your name analyzer is working correctly; i.e., does it correctly match each use of a name to the corresponding declaration, and does it correctly set the link from the `IdNode` to the information in the symbol table.) For names of functions, the information should be of the form: `param1Type, param2Type, ..., paramNType -> returnType`. For names of global variables, parameters, and local variables of a non-`struct` type , the information should be `int` or `bool`. For a global or local variable that is of a `struct` type, the information should be the name of the `struct` type. For example, given a program that contains this code:

```
struct Point {
    int x;
    int y;
};
int f(int x, bool b) { }
void g() {
    int a;
    bool b;
    struct Point p;
    p.x = a;
    b = a == 3;
    f(a + p.y*2, b);
    g();
}
```

The unparser should print:

```
struct Point {
    int x;
    int y;
};
int f(int x, bool b) {
}
void g() {
    int a;
    bool b;
    struct Point p;
    p(Point).x(int) = a(int);
    b(bool) = (a(int) == 3);
    f(int,bool->int)((a(int) + (p(Point).y(int) * 2)), b(bool));
    g(->void)();
}
```

P4.java

The main program, `P4.java`, will be similar to `P3.java`, except that

- After parsing, if there are no syntax errors, it will call the name analyzer.
- After that, if there are no errors so far (either scanning, parsing, or name-analysis errors), it will call the unparser.

Calling the name analyzer means calling the appropriate method of the `ASTnode` that is the root of the tree built by the parser.

Modifying the `ErrMsg` Class

Your compiler should quit after the name analyzer has finished if any errors have been detected so far (either by the scanner/parser or the name analyzer). To accomplish this, you can add a static boolean field to the `ErrMsg` class that is initialized to `false` and is set to `true` if the `fatal` method is ever called (warnings should not change the value of this field). Your `main` program can check the value of this field and only call the unparser if it is `false`.

Updating the Makefile

You will need to update the `Makefile` you used for program 3 so that typing "`make`" creates `P4.class`.

Writing Test Inputs

You will need to write two input files to test your code:

1. `nameErrors.cf` should contain code with errors detected by the name analyzer. This means that it should include bad and multiply declared names for all of the different kinds of names, and in all of the different places that declarations can appear. It should also include uses of undeclared names in all kinds of statements and expressions as well as bad `struct` accesses.

2. `test.cf` should contain code with no errors that exercises all of the name-analysis methods that you wrote for the different AST nodes. This means that it should include (good) declarations of all of the different kinds of names in all of the places that names can be declared and it should include (good) uses of names in all kinds of statements and expressions.

Note that your `nameErrors.cf` should cause error messages to be output, so to know whether your name analyzer behaves correctly, you will need to know what output to expect.

As usual, you will be graded in part on how thoroughly your input files test your code.

Some Advice

Here are few words of advice about various issues that come up in the assignment:

- For this assignment you are free to make any changes you want to the code in `ast.java`.

- The tree-traversal code you wrote to perform unparsing provides a good model for the traversal that you need to write to handle name analysis. However, you might not want to declare the name-analysis methods to be abstract methods of class `ASTnode` (as we did for `unparse`). This is because you will not need those methods for all nodes; e.g., you probably won't want a name-analysis method for all of the sub-classes of the `TypeNode` class.

  However, you will need to declare the name-analysis methods to be abstract methods of some of the classes that are lower down in the inheritance hierarchy; for example, you will need to declare an abstract name-analysis method for the `DeclNode` class, because the method for the `DeclListNode` class will call that method for each node in the list.

- If you are working with a partner, you will have to decide how to divide up the work. You might want to divide up some of the "incidental tasks" (like modifying the `ErrMsg`, `Sym`, and `IdNode` classes), then work together to get a small part of the name-analysis phase working (e.g., finding multiply declared global variables). Then you could split up the `ASTnode` subclasses and each implement the name-analysis methods for your subset of those

classes (you might want to start by choosing just a few each, until you have a better idea which ones will require the most work).

Don't forget to test your work as you go along, rather than waiting until everything is finished!

Submission

Please submit **all** the files (including the JLex file and the Makefile). **Do not** include the `deps` folder or the `deps_src` folder. Also do not create any sub-directories when creating your tar file. Create a tar file of your work as follows.

`cd` into your working directory and run `tar -cvzf lastname.firstname.P4.tar.gz *`

Submit this tar file to the P4 folder on Learn@UW.

Programming Assignment 5

Due 11pm, Nov 21

Overview

For this assignment you will write a type checker for `harambe` programs represented as abstract-syntax trees. Your main task will be to write *type checking* methods for the nodes of the AST. In addition you will need to:

1. Write a new main program, `P5.java` (an extension of `P4.java`).
2. Update the `Makefile` used for program 4 to include any new rules needed for program 5.
3. Write two test inputs: `typeErrors.ha` and `test.ha` to test your new code.

Getting Started

You have a couple of options for completing p5:

Using Your Own Code

If you'd like to use your own code, you are free to do so. Copy everything over from your P4, change the name of your driver class to P5.java, and update your Makefile

Correct implementations of `ast.java` and `Sym.java` (and a helper class, `Type.java`) for program 4 are available at the links above, or you may use your own implementation.

Starting Fresh (Recommended)

If you don't want to use eclipse, you can use a fresh version of the code by downloading the tarball here. The Makefile assumes that you already have the CLASSPATH environment variable set. If you do not have it set then use configure.sh file from the previous assignments to set the CLASSPATH.

Specifications

Type Checking

The type checker will determine the type of every expression represented in the abstract-syntax tree and will use that information to identify type errors. In the language we have the following types:

> `int`, `bool`, `void` (as function return types only), `struct` *types*, and *function types*.

A `struct` *type* includes the name of the struct (i.e., when it was declared/defined). A *function type* includes the types of the parameters and the return type.

The operators in the language are divided into the following categories:

- **logical**: not, and, or
- **arithmetic**: plus, minus, times, divide, unary minus
- **equality**: equals, not equals
- **relational**: less than (<), greater than (>), less then or equals (<=), greater than or equals (>=)
- **assignment**: assign

The type rules of the language are as follows:

- **logical operators and conditions**: Only boolean expressions can be used as operands of logical operators or in the condition of an `if` or `while` statement. The result of applying a logical operator to `bool` operands is `bool`.

- **arithmetic and relational operators**: Only integer expressions can be used as operands of these operators. The result of applying an arithmetic operator to `int` operand(s) is `int`. The result of applying a relational operator to `int` operands is `bool`.

- **equality operators**: Only integer or boolean expressions can be used as operands of these operators. Furthermore, the types of both operands must be the same. The result of applying an equality operator is `bool`.
  Note: You don't need to worry about equality operators between string literals. Either accepting it or declining it will be accepted in this assignment.

- **assignment operator**: Only integer or boolean expressions can be used as operands of an assignment operator. Furthermore, the types of the left-hand side and right-hand side must be the same. The type of the result of applying the assignment operator is the type of the right-hand side.

- **`cout` and `cin`**:
  Only an `int` or `bool` expression or a string literal can be printed by `cout`. Only an `int` or `bool` identifer can be read by `cin`. Note that the identifier can be a field of a `struct` type (accessed using `.`) as long as the field is an `int` or a `bool`.

- **function calls**: A function call can be made only using an identifier with function type (i.e., an identifier that is the name of a function). The number of actuals must match the number of formals. The type of each actual must match the type of the corresponding formal.

- **function returns**:
  A `void` function may not return a value.
  A non-`void` function may not have a `return` statement without a value.
  A function whose return type is `int` may only return an `int`; a function whose return type is `bool` may only return a `bool`.

  Note: some compilers give error messages for non-`void` functions that have paths from function start to function end with no `return` statement. For example, this code would cause such an error:

  ```
  int f() {
      cout << "hello";
  }
  ```

  However, finding such paths is beyond the capabilities of our compiler, so don't worry about this kind of error.

You must implement your type checker by writing appropriate member methods for the different subclasses of `ASTnode`. Your type checker should find all of the type errors described in the following table; it must report the specified position of the error, and it must give *exactly* the specified error message. (Each message should appear on a single line, rather than how it is formatted in the following table.)

Type of Error	Error Message	Position to Report
Writing a function; e.g., "`cout << f`", where `f` is a function name.	`Attempt to write a function`	1st character of the function name.
Writing a `struct` name; e.g., "`cout << P`", where `P` is the name of a `struct` type.	`Attempt to write a struct name`	1st character of the `struct` name.
Writing a `struct` variable; e.g., "`cout << p`", where `p` is a variable declared to be of a `struct` type.	`Attempt to write a struct variable`	1st character of the `struct` variable.
Writing a `void` value (note: this can only happen if there is an attempt to write the return value from a `void` function); e.g., "`cout << f()`", where `f` is a `void` function.	`Attempt to write void`	1st character of the function name.
Reading a function: e.g., "`cin >> f`", where `f` is a function name.	`Attempt to read a function`	1st character of the function name.
Reading a `struct` name; e.g., "`cin >> P`", where `P` is the name of a `struct` type.	`Attempt to read a struct name`	1st character of the `struct` name.
Reading a `struct` variable; e.g., "`cin >> p`", where `p` is a variable declared to be of a `struct` type.	`Attempt to read a struct variable`	1st character of the `struct` variable.
Calling something other than a function; e.g., "`x();`", where `x` is not a function name. Note: In this case, you should *not* type-check the actual parameters.	`Attempt to call a non-function`	1st character of the variable name.
Calling a function with the wrong number of arguments. Note: In this case, you should *not* type-check the actual parameters.	`Function call with wrong number of args`	1st character of the function name.
Calling a function with an argument of the wrong type. Note: you should only check for this error if the number of arguments is correct. If there are several arguments with the wrong type, you must give an error message for each such argument.	`Type of actual does not match type of formal`	1st character of the first identifier or literal in the actual parameter.
Returning from a non-void function with a plain `return` statement (i.e., one that does not return a value).	`Missing return value`	0,0
Returning a value from a `void` function.	`Return with a value in a void function`	~~1st character of the returned expression.~~ 1st character of the first identifier or literal in the returned expression.
Returning a value of the wrong type from a non-`void` function.	`Bad return value`	~~1st character of the returned expression.~~ 1st character of the first identifier or literal in the returned expression.
Applying an arithmetic operator (`+`, `-`, `*`, `/`) to an operand with type other than `int`. Note: this includes the `++` and `--` operators.	`Arithmetic operator applied to non-numeric operand`	1st character of the first identifier or literal in an operand that is an expression of the wrong type.
	`Relational`	1st character of the first

Applying a relational operator (<, >, <=, >=) to an operand with type other than `int`.	`operator applied to non-numeric operand`	identifier or literal in an operand that is an expression of the wrong type.
Applying a logical operator (`!`, `&&`, `||`) to an operand with type other than `bool`.	`Logical operator applied to non-bool operand`	1st character of the first identifier or literal in an operand that is an expression of the wrong type.
Using a non-`bool` expression as the condition of an `if`.	`Non-bool expression used as an if condition`	1st character of the first identifier or literal in the condition.
Using a non-`bool` expression as the condition of a `while`.	`Non-bool expression used as a while condition`	1st character of the first identifier or literal in the condition.
Applying an equality operator (`==`, `!=`) to operands of two different types (e.g., "`j == true`", where `j` is of type `int`), or assigning a value of one type to a variable of another type (e.g., "`j = true`", where `j` is of type `int`).	`Type mismatch`	1st character of the first identifier or literal in the left-hand operand.
Applying an equality operator (`==`, `!=`) to `void` function operands (e.g., "`f() == g()`", where `f` and `g` are functions whose return type is `void`).	`Equality operator applied to void functions`	1st character of the first function name.
Comparing two functions for equality, e.g., "`f == g`" or "`f != g`", where `f` and `g` are function names.	`Equality operator applied to functions`	1st character of the first function name.
Comparing two `struct` names for equality, e.g., "`A == B`" or "`A != B`", where `A` and `B` are the names of `struct` types.	`Equality operator applied to struct names`	1st character of the first `struct` name.
Comparing two `struct` variables for equality, e.g., "`a == b`" or "`a != b`", where `a` and `a` are variables declared to be of `struct` types.	`Equality operator applied to struct variables`	1st character of the first `struct` variable.
Assigning a function to a function; e.g., "`f = g;`", where `f` and `g` are function names.	`Function assignment`	1st character of the first function name.
Assigning a `struct` name to a `struct` name; e.g., "`A = B;`", where `A` and `B` are the names of `struct` types.	`Struct name assignment`	1st character of the first `struct` name.
Assigning a `struct` variable to a `struct` variable; e.g., "`a = b;`", where `a` and `b` are variables declared to be of `struct` types.	`Struct variable assignment`	1st character of the first `struct` variable.

Preventing Cascading Errors

A single type error in an expression or statement should not trigger multiple error messages. For example, assume that `P` is the name of a `struct` type, `p` is a variable declared to be of `struct` type `P`, and `f` is a function that has one integer parameter and returns a `bool`. Each of the following should cause only one error message:

```
cout << P + 1          // P + 1 is an error; the write is OK
(true + 3) * 4         // true + 3 is an error; the * is OK
true && (false || 3)   // false || 3 is an error; the && is OK
f("a" * 4);            // "a" * 4 is an error; the call is OK
1 + p();               // p() is an error; the + is OK
(true + 3) == x        // true + 3 is an error; the == is OK
                       // regardless of the type of x
```

One way to accomplish this is to use a special `ErrorType` for expressions that contain type errors. In the first example above, the type given to `(true + 3)` should be `ErrorType`, and the type-check method for the multiplication node should *not* report "`Arithmetic operator applied to non-numeric operand`" for the first operand. But note that the following should each cause *two* error messages (assuming the same declarations of `f` as above):

```
true + "hello"    // one error for each of the non-int operands of the +
1 + f(true)       // one for the bad arg type and one for the 2nd operand of the +
1 + f(1, 2)       // one for the wrong number of args and one for the 2nd operand of the +
return 3+true;    // in a void function: one error for the 2nd operand to +
                  // and one for returning a value
```

To provide some help with this issue, here is an example input file, along with the corresponding error messages. (Note: This is not meant to a complete test of the type checker; it is provided merely to help you understand some of the messages you need to report, and to help you find small typos in your error messages. If you run your program on the example file and put the output into a new file, you can use the Linux utility `diff` to compare your file of error messages with the one supplied here. This will help both to make sure that your code finds the errors it is supposed to find, and to uncover small typos you may have made in the error messages.)

Other Tasks

P5.java

The main program, `P5.java`, will be similar to `P4.java`, except that if it calls the name analyzer and there are no errors, it will then call the type checker.

Updating the Makefile

You will need to update the `Makefile` you used for program 4 so that typing "`make`" creates `P5.class`.

Writing Test Inputs

You will need to write two input files to test your code:

1. `typeErrors.ha` should contain code with errors detected by the type checker. For every type error listed in the table above, you should include an instance of that error for each of the relevant operators, and in each part of a program where the error can occur (e.g., in a top-level statement, in a statement inside a while loop, etc).

2. `test.ha` should contain code with no errors that exercises all of the type-check methods that you wrote for the different AST nodes. This means that it should include (good) examples of every kind of statement and expression.

Note that your `typeErrors.ha` should cause error messages to be output, so to know whether your type checker behaves correctly, you will need to know what output to expect.

Part of the grade depends on how thoroughly the input files you used, test the program. Make sure that you submit the input files you used to test your program.

Some Advice

Here are few words of advice about various issues that come up in the assignment:

- For this assignment you are free to make any changes you want to the code in `ast.java`. For example, you may find it helpful to make small changes to the class hierarchy, or to add new fields and/or methods to some classes.

- As for name analysis, think about which AST nodes need to have type-check methods. For example, for type checking, you do not need to visit nodes that represent declarations, only those that represent statements.

Programming Assignment 6

Due December 12, 11pm

For this assignment you will write a code generator that generates MIPS assembly code (suitable as input to the Spim interpreter) for `harambe` programs represented as abstract-syntax trees.

Specifications

General information

Similar to the fourth and fifth assignments, the code generator will be implemented by writing `codeGen` member functions for the various kinds of AST nodes. *See the on-line Code Generation notes (as well as lecture notes) for lots of useful details*.

In addition to implementing the code generator, you will also update the Makefile and the main program (and call it `P6.java`) so that, if there are no errors (including type errors), the code generator is called **after** the type checker. The code generator should write code to the file named by the second command-line argument.

Note

Your main program should no longer call the unparser, nor should it report that the program was parsed successfully.

Also note that you are *not* required to implement code generation for `structs` or anything `struct`-related (like dot-accesses).

Getting started

Implementation of `ast.java` is made available for you

Some useful code-generation methods can be found in the file `Codegen.java`. Note that to use the methods and constants defined in that file you will need to prefix the names with `Codegen.`; for example, you would write: `Codegen.genPop(Codegen.T0)` rather than `genPop(T0)`. (Alternatively, you could put the declarations of the methods and constants in your `ASTnode` class; then you would not need the `Codegen` prefix.) Also note that a `PrintWriter p` is declared as a static public field in the `Codegen` class. The code-generation methods in `Codegen.java` all write to `PrintWriter p`, so you should use it when you open the output file in your main program (in `P6.java`); i.e., you should include:

```
Codegen.p = new PrintWriter(args[1]);
```

in your main program (or `ASTnode.p` if you put the declarations in the `ASTnode` class). You should also close that `PrintWriter` at the end of the program:

```
Codegen.p.close();
```

Spim

The best way to test your MIPS code is using the simulator SPIM (written by at-the-time UW-Madison Computer Science Professor Jim Larus). The class supports two versions of spim:

1. A command line program, called **spim**
   Accessing spim:
   - Installed on the lab computers at `~cs536-1/public/tools/bin/spim`
   - Available as source as part of the svn repository:
     `svn://svn.code.sf.net/p/spimsimulator/code/`
2. A GUI-driven program, called **QtSpim**
   Accessing QtSpim:
   - Installed on the lab computers at `~cs536-1/public/tools/bin/QtSpim`
   - Avaiable as a binary package here
   - Also avaiable as source as part of the svn repository
     `svn://svn.code.sf.net/p/spimsimulator/code/`, but building it is somewhat painful (trust me on this).

Both of these tools use the same backend, but I recommend using QtSpim since it is much more of a modern interface. Generally, it should be enough to run

```
~cs536-1/public/tools/bin/QtSpim -file <mips_code.s>
```

And use the interactive help or menus from there. However, if you want more guidance on using spim, you can check out this (fairly old) Reference Manual (pdf). Also, check the tutorials page for a screencast on MIPS and SPIM.

where *`src`* is the name of your source file (i.e., the one containing your MIPS code).

Remark

Althouth QtSpim is much more useable, you need to be on one of the CSL machines to ensure that it runs smoothly. If you are planning to use QtSpim, make sure you use the lab machines. If you necessarily have to use QtSpim remotely, make sure you enable X-forwarding when you SSH. (i.e.) use ssh -X user@host.cs.wisc.edu when you login.

To get the Spim simulator to correctly recognize your `main` function and to exit the program gracefully, there are two things you need to do:

1. When generating the function preamble for `main`, add the label "`__start:`" on the line after the label "`main:`" (note that `__start:` contains two underscore characters).

2. When generating the function exit for `main`, instead of returning using "`jr $ra`", issue a `syscall` to exit by doing:

   ```
   li $v0, 10
   syscall
   ```

   (Note that this means that a program that contains a function which calls `main` won't work correctly, which will be ok for the purposes of this project.)

Here is a link to an example `harambe` program and the corresponding MIPS code.

Changes to old code

Required changes:

1. Add to the name analyzer or type checker (your choice), a check whether the program contains a function named `main`. If there is no such function, print the error message: "No main function". Use 0,0 as the line and character numbers.

2. Add a new "`offset`" field to the `Sym` class (or to the appropriate subclass(es) of `Sym`). Change the name analyzer to compute offsets for each function's parameters and local variables (i.e., where in the function's Activation Record they will be stored at runtime) and to fill in the new offset field. Note that each scalar variable requires 4 bytes of storage. You may find it helpful to verify that you have made this change correctly by modifying your unparser to print each local variable's offset.

Suggested changes:

1. Modify the name analyzer to compute and save the total size of the local variables declared in each function (e.g., in a new field of the function name's symbol-table entry). This will be useful when you do code generation for function entry (to set the SP correctly).

2. Either write a method to compute the total size of the formal parameters declared in a function, or modify the name analyzer to compute and store that value (in the function name's symbol-table entry). This will also be useful for code generation for function entry.

3. Change the definition of class `WriteStmtNode` to include a (private) field to hold the type of the expression being written, and change your typecheck method for the `WriteStmtNode` to fill in that field. This will be useful for code generation for the *write* statement (since you will need to generate different code depending on the type of the expression being output).

Non-obvious semantic issues

1. All parameters should be passed by value.

2. The *and* and *or* operators (`&&` and `||`) are *short circuited*, just as they are in Java. That means that their right operands are only evaluated if necessary (for all of the other binary operators, both operands are always evaluated). If the left operand of "`&&`" evaluates to *false*, then the right operand is not evaluated (and the value of the whole expression is *false*); similarly, if the left operand of "`||`" evaluates to *true*, then the right operand is not evaluated (and the value of the whole expression is *true*).

3. In `harambe` (as in C++ and Java), two string literals are considered equal if they contain the same sequence of characters. So for example, the first two of the following expressions should evaluate to *false* and the last two should evaluate to *true*:

```
"a" == "abc"
"a" == "A"
"a" == "a"
"abc" == "abc"
```

4. Boolean values should be output as 1 for *true* and 0 for *false* (and that is probably how you should represent them internally as well).

5. Boolean values should also be input using 1 for *true* and 0 for *false*.

Structs

Work on structs last for this assignment. Based on how the class is going as the deadline approaches, I may decide to either drop structs from the assignment or make it extra credit.

Suggestions for how to work on this assignment

1. Modify name analysis or type checking to ensure that a main function is declared.

2. Modify name analysis so that the code generator can answer the following questions:

    - Is an Id local or global?
    - If local, what is its offset in its function's AR?
    - For each function, how many bytes of storage are needed for its params, and how many are needed for its locals?

3. Implement code generation for each of the following features; be sure to test each feature as it is implemented!

    - global variable declarations, function entry, and function exit (write a test program that just declares some global variables and a main function that does nothing)
    - `int` and `bool` literals (just push the value onto the stack), string literals, and `WriteStmtNode`
    - `IdNode` (code that pushes the value of the id onto the stack, and code that pushes the address of the id onto the stack) and assignments of the form `id=literal` and `id=id` (test by assigning then writing)
    - expressions other than calls
    - statements other than calls and returns
    - call statements and expressions, return statements (to implement a function call, you will need a third code-generation method for the `IdNode` class: one that is called only for a function name and that generates a jump-and-link instruction)