CSE 3341 Project 2 - CORE Parser

Overview

The goal of this project is to build a parser for the Core language discussed in class. At the end of this handout is the grammar you should follow for this project. This project should be completed using Java or Python.

Your submission should compile and run on stdlinux. It is your responsibility to ensure your code works there.

You need to write a parser for the language described. Every valid input program for this language should be parsed, and every invalid input program for this language should be rejected with an error message.

Main

Your project should have a main procedure in a file called Main.java or Main.py, which does the following in order:

1. Instantiate the scanner with the file name given as a command line argument.
2. Generate a parse tree for the input Core program using recursive descent.
3. Perform semantic checks on the parse tree.
4. Use recursive descent to print the Core program from the parse tree.

Make sure your parser can be ran using one of these commands so that it will interact with my tester.sh and grading scripts:

- java Main Correct/1.code
- python3.7 Main.py Correct/1.code

Input to your project

The input will be identical to the input for project 1. The scanner should processes the sequence of ASCII characters in the file and should produce a sequence of tokens as input to the parser. The parser performs syntax analysis of this token stream.

Parse Tree Representation

You should generate a parse tree for the input program, using the top-down recursive descent approach described in class.

You may represent the parse tree however you like. In the "Suggestions" section I give a recommendation on how to do this, not a requirement.

Output from your project

All output should go to stdout. This includes error messages - do not print to stderr.

The parser functions should only produce output in the case of an error.

The printer functions should produce "pretty" code with the appropriate indentation, i.e. statement sequences for if/while statements should be indented, and nested statements should be further indented. I do not have strict format requirements here, the printer functions are mainly to help you verify you generated a correct parse tree.

To make things somewhat uniform, I suggest using the tab character to create indentations. Also, use as few spaces as possible in your print functions, e.g. instead of printing:

```
input x ;
z = x ;
```

print this:

```
input x;
z=x;
```

NOTE: THE PRINT FUNCTIONS SHOULD ONLY BE CALLED AFTER THE ENTIRE PROGRAM HAS BEEN PARSED! The purpose of the print functions is to verify you have a complete parse tree that has accurately stored all the information we will need to interpret the program in project 3. The print functions should have no interaction with the scanner.

Invalid Input and Semantic Checks

Your parser should recognize and reject invalid input. For any error, you have to catch it and print a message. The message should have the form "ERROR: some description" (ERROR in uppercase). The description should be a few words and should accurately describe the source of the problem. You do not need to have sophisticated error messages that denote exactly where the problem is in the input file. After printing the error message to stdout, just exit back to the OS. But make sure you catch the error conditions! When given invalid input, you parser should not "crash" with a segmentation fault, uncaught exceptions, ect. Up to 20% of your score will depend on the handling of incorrect input and on printing useful error messages.

There are several categories of errors you should catch:

Scanner Errors

First, the scanner should make sure that the input stream of characters represents a valid sequence of tokens. For example, characters such as '_' and '%' are not allowed in the input stream. Your scanner from project 1 should already be catching these. If the scanner finds an error, the parser should stop executing.

Parser Errors

Second, the parser should make sure that the stream of tokens is correct with respect to the context-free grammar. If the parser detects a token in the wrong place it should print a meaningful error message and stop parsing.

Semantic Errors

Third, after building the parse tree there are additional checks that need to be done to ensure that the program "makes sense" (semantic checking). For this project, the following semantic checking needs to occur:

Verify that every ID being used has been declared in that scope and prior to being used. variables declared in the DeclSeq are "global" variables, and can be accessed from any scope. Variables declared in a StmtSeq are "local" variables, and are only in scope for that StmtSeq or any nested scope. For example, there are two semantic errros in this program:

```
program
   int x;
begin
   x=1;
   int y;
   y=x;
   if x==y then
      z=y;
      int z;
   endif
   y=z;
end
```

1. "z=y;" is before "z" has been declared
2. "y=z" is happening after "z" has gone out of scope

Verify that IDs were declared with the appropriate type (int or ref) for how they are being used. There are two places where you need to check that the variables being used are of the correct type:

1. "id = new;", the id here needs to have been declared as ref, not int
2. "id = ref id", both id's being used here need to have been declared as ref, not int

Check for "doubly-declared" variable names. All variables declared within the same scope should have unique names, but variables declared in different scopes can have the same name. For example,

```
program
    int x, y;
    int z, x;
begin
    x=1;
end
```
should result in an error because there are x's declared in the global scope, but
```
program
    int x, y;
begin
    int z, x;
    if y==z then
        int x;
        x=1;
    endif
end
```
should **NOT** result in an error because the three x's are all declared in different scopes.

Testing Your Project

I will provide some test cases. The test cases I will provide are rather weak. You should do additional testing testing with your own cases. For each test case I provide (e.g. t4) there are two files (e.g. t4.code and t4.expected).
I will provide a "tester.sh" script that works similar to how the script from project 1 worked, and it will use the diff command to compare your output to the expected output.

Suggestions

There are many ways to approach this project. Here are some suggestions:

- Spend some time making sure you understand the grammar on the last page.
- Make sure you understand the recursive descent parser described in lecture.
- Plan to spend a significant amount of time on the parser. Once it is working correctly, the semantic checking and printing should be simple.
- Have separate functions for the parsing and semantic checks. Trying to do these simultaneously will increase the complexity of your code and increase the likelihood of errors.
- Represent the parse tree by creating a class for each nonterminal. Instances of these classes will then represent the nodes of your parse tree. Each class should at least include a field for each potential child, a parse method, and a print method.

- The parse tree does not need to store everything, many tokens can be discarded after checking them. For example, the "Program" root node does not need to store the PROGRAM, BEGIN, and END tokens, we just to need to verify those were present in the input.
- Pick a small subset of the language (e.g. only a few of the grammar productions) and implement a fully functioning parser for that subset. Do extensive testing. Add more grammar productions. Repeat.
- Post questions on piazza, and read the questions other students post. You may find details you missed on your own. You are encouraged to share test cases with the class on piazza.

Project Submission

On or before 11:59 pm September 24th, you should submit the following:

- Your complete source code.
- An ASCII text file named README.txt that contains:
  - Your name on top
  - The names of all the other files you are submitting and a brief description of each stating what the file contains
  - Any special features or comments on your project
  - A description of the overall design of the parser, including the parse tree representation.
    - Yes, the graders and I already know the design of the parser and the parse tree representation. You should still describe your understanding of these things here.
  - A brief description of how you tested the parser and a list of known remaining bugs (if any)

Submit your project as a single zipped file to the Carmen dropbox for Project 2.

If the time stamp on your submission is 12:00 am on June 5th or later, you will receive a 10% reduction per day, for up to three days. If your submission is more than 3 days late, it will not be accepted and you will receive zero points for this project. If you resubmit your project, only the latest submission will be considered.

Grading

The project is worth 100 points. Correct functioning of the parser is worth 65 points. The handling of error conditions is worth 20 points. The implementation style and documentation are worth 15 points.

Academic Integrity

The project you submit must be entirely your own work. Minor consultations with others in the class are OK, but they should be at a very high level, without any specific details. The work on the project should be entirely your own; all the design, programming, testing, and debugging should be done only by you, independently and from scratch. Sharing your code or documentation with others is not acceptable. Submissions that show excessive similarities (for code or documentation) will be taken as evidence of cheating and dealt with accordingly; this includes any similarities with projects submitted in previous instances of this course.

Academic misconduct is an extremely serious offense with severe consequences. Additional details on academic integrity are available from the Committee on Academic Misconduct (see http://oaa.osu.edu/coamresources.html). If you have any questions about university policies or what constitutes academic misconduct in this course, please contact me immediately.

Please note this is a language like C or Java where whitespaces have no meaning, and whitespace can be inserted between keywords, identifiers, constants, and specials to accommodate programmer style. This grammar does not include rules about whitespace because that would add immense clutter.

<prog> ::= **program** <decl-seq> **begin** <stmt-seq> **end** | **program begin** <stmt-seq> **end**

<decl-seq> ::= <decl> | <decl><decl-seq>

<stmt-seq> ::= <stmt> | <stmt><stmt-seq>

<decl> ::= <decl-int> | <decl-class>

<decl-int> ::= **int** <id-list> **;**

<decl-class> ::= **ref** <id-list> **;**

<id-list> ::= **id** | **id ,** <id-list>

<stmt> ::= <assign> | <if> | <loop> | <in> | <out> | <decl>

<assign> ::= **id =** <expr> **;** | **id = new ;** | **id = ref id ;**

<in> ::= **input id ;**

<out> ::= **output** <expr> **;**

<if> ::= **if** <cond> **then** <stmt-seq> **endif**

| **if** <cond> **then** <stmt-seq> **else** <stmt-seq> **endif**

<loop> ::= **while** <cond> **begin** <stmt-seq> **endwhile**

<cond> ::= <cmpr> | **! (** <cond> **)**

| <cmpr> **or** <cond>

<cmpr> ::= <expr> **==** <expr> | <expr> **<** <expr>

| <expr> **<=** <expr>

<expr> ::= <term> | <term> **+** <expr> | <term> **–** <expr>

<term> ::= <factor> | <factor> ***** <term>

<factor> ::= **id** | **const** | **(** <expr> **)**