EECS665
Compiler Construction

Drew Davidson
Andrew Riachi
Koyel Pramanick

Lecture: LEEP2 2300
M/W/F 3:00 PM - 3:50 PM

Project 3

Due on October 4th 11:59 PM (Accepted with no penalty).

Accepted late by October 5th 11:59 PM for 1 penalty

Accepted late by October 6th 11:59 PM for 2 penalties

Not accepted on or after October 7th 12:00 AM

Each penalty incurs a 15% stacking reduction on the project grade or uses 1 "penalty token".

Penalty tokens are applied by course personnel to your maximum benefit.

Resources

- Inheritance diagram - *This diagram suggests the is-a relationship between classes that you can use to capture the abstract-syntax tree.*
- Starter files - *Files to get you started on the project, such as a skeleton bison spec to implement the parser. If you don't want to use your completed code from the previous project.*

- The oracle - *Submit an input file (i.e. a source file in the A language, potentially with errors), and the oracle will say what the correct output should be for this project.*

Updates

None yet!

Overview

For this assignment you will extend your **Bison** spec to perform a syntax-directed translation, producing an AST for the input A 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, `main.cpp`, that calls the parser and then the unparser is already provided.

This project is intended to be an extension of the previous project code, but will be invoked via a new argument. Your code should be built and invoked via the command sequence

```
make all
./ac <file.a> -u <outfile.txt>
```

Where `file.a` is an input file to have it's syntax checked and `<outfile.txt>` is a file containing the output of unparsing. The only **required** behavior of your project is that the output file should correspond to pretty-printed A code. Thus, although AST classes and relationships are suggested, you can create whatever types of AST nodes make sense for you.

How We'll Grade

The goal of this project is to build an AST of the input program, and that your AST is accurate. To ensure this goal, we'll test that your compiler can "pretty-print" or "unparse" the input program in a canonical form.

Your output will need to adhere to a specific newline and indentation behavior:

- Open brackets should be followed by a newline.

- Lines of code should be indented by 4 spaces for each block level of nesting.

Other than newlines and indentation, your output will be considered correct if it is *semantically equivalent* to the input (i.e. the output program has the same meaning as the original). This means that (for one), you can add or remove parentheses/spaces to expressions as long as they don't change the meaning of the expression. Consider the input program

```
v : () -> int { return 1 + (2 * 3); }
```

It would be fine to print

```
v : () -> int {
    return 1 + 2 * 3;
}
```

or even

```
v : () -> int {
    return (1 + 2 * 3); }
```

but NOT

```
v : () -> int {
    return (1 + 2) * 3;
}
```

Using The Starter Code

Getting Started

You are encouraged to build upon your P2 files to complete this project. However, "starter files" are provided here as a starting point: p3.tgz

If you choose to use the starter code (or closely mirror the starter code's suggested implementation), the below advice will get you on the right track. Of course, this advice is optional. You don't need to use the same node classes provided, nor use the same filenames or whatever. You just need to build some form of AST and make sure it pretty-prints.

To check on an acceptable format for unparsing output, you may consult the P3 oracle

Building an Abstract-Syntax Tree

To make your parser build an abstract-syntax tree, you must make sure that each production has a syntax-directed definition in the a.yy Bison spec. You will need to decide, for each nonterminal, what type its associated translation attribute 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 `std::list` of some kind of node. Use the information in the inheritance diagram for a suggested list of AST node types and the collaboration diagram above to guide your decision (both of these diagrams are linked above).

Make sure that each action includes an assignment to `$$`. Note that the parser will set the `ASTNode` pointer to the value assigned to the production for the root nonterminal (nonterminal `program`).

Unparsing

To test your Syntax-directed definition, we need to see the AST be output. As such, any `ASTNode` subclass should have an `unparse` function that recursively creates a textual representation of that node. Since the AST was derived from a textual representation, the unparsed output should look a lot like the original input file.

The recommended way to implement the `unparse` functions is to add them to `unparse.cpp` (some examples are already given). 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. 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" (a uniform style of newlines and indentation will be used to make the program readable);
3. Expressions will be fully parenthesized to reflect the order of evaluation.

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.

Modifying `unparse.cpp`

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

1. Modify `unparse.cpp` by implementing the `unparse` methods and creating new node classes in