Programming Assignment 2
Due October 9, 2017 at 11:59pm

1 Overview of the Programming Project

Programming assignments 1-4 will direct you to design and build an interpreter for L. Each assignment will cover one component of the interpreter: lexical analysis, parsing, interpreting L and performing type inference on L.

For this assignment, you will build a parser for the L programming language. The tool you will use for this is called `bison`. You will describe the grammar of L in an appropriate input format, and the parser generator will generate the actual C++ code for parsing tokens into an L abstract syntax tree. On-line documentation for all the tools needed for the project will be made available on the CS345H web site. This includes the manual for for bison. **You will need to refer to the bison manual to complete this assignment**. You may work either individually or in pairs for this assignment.

2 Abstract Syntax Tree

The data structures for the abstract syntax tree you will build for L ca be found in the `/ast/` sub-directory. You will need to understand the interface of the AST nodes to complete this assignment. This section sketches the high-level deign of the AST library. Please refer to the AST source code for any further details.

The provided AST library contains the following AST nodes:

```
AstBinOp
AstBranch
AstExpressionList
AstIdentifier
AstIdentifierList
AstInt
AstLambda
AstLet
AstList
AstNil
AstRead
AstString
AstUnOp
Expression
```

Observe that not every construct in L has an AST node. Notably, function definitions do not have any type. Therefore you need represent function definitions as a combination of an AstLet and an AstLambda as discussed in lecture.

All AST nodes inherit from the abstract base class `Expression`. This means the following code is legal and will always succeed:

```
AstInt* i = ...
Expression* e = i;
```

On the other hand, the following code snipped will not compile because the expression `e` may not be of type `AstInt`.

```
Expression* e = ...
AstInt* i = e;
```

However, often you know that an value of type `Expression` must really be a specific subtype. In this case, you can cast the expression to the desired subtype in the following way:

```
Expression* e = ...
AstInt* i = static_cast<AstInt*>(e);
```

Observe that this will always compile, even if the type of `e` may not be `AstInt`. In this case, your program will most likely override random memory values and crash later at an completely unrelated point! To avoid this, you can also check the dynamic type of any expression before casting. You can do this in the following way:

```
Expression* e = ...
assert(e->get_type() == AST_INT);
AstInt* i = static_cast<AstInt*>(e);
```

The types for all expressions are defined in `enum expression_type` in the file `Expression.h`. With this extra assertion, your code will fail with an assertion failure notifying you of the problem at the source. **You should never cast without asserting the correct type of expressions first! You will save many hours of painful debugging if you do.**

For debugging, all AST nodes also have a `to_string()` method you can call that will print a human-readable string representation of the node. For example, to print an expression, you would write:

```
Expression* e = ...
cout << "Expression e: " << endl << e->to_string() << endl;
```

All AST classes use *factory make methods*. This means you cannot allocate any AST node using the `new` operator. For example, the following code will *not* compile:

```
Expression* e = new AstInt(2);
```

Instead, you write:

```
Expression* e = AstInt::make(2);
```

`make` methods are static members of the classes that allocate new instances for you. They also manage all memory centrally, so you never need to delete any memory. **Never call `delete` on any AST nodes. Doing so will result in horrible crashes!**

The Ast types `AstExpressionList` and `AstIdentifierList` also have a method `append(e)` that returns a new list with the element appended. For instance, to obtain an expression list with expression `e` added at the end, you would write:

```
AstExpressionList* l = ...
AstExpressionList* new_l = l->append(e);
```

Observe that all AST nodes are immutable; the method `append` returns a new expression list with the desired element added.

3 Introduction to Bison

Bison allows you to implement a parser and associate semantic actions with each grammar rule. In general, a bison file is structured as follows:

```
%{
Declarations
%}
Definitions
%%
Rules
```

In the declarations section, all token types are defined. These are all terminals in the L grammar.

After the token, you will need to specify associativity and precedence both for operators and some non-terminals. For example, this section may contain:

```
%left TOKEN_PLUS
%left TOKEN_TIMES
```

This first of all states that plus and times are left-associative. Furthermore, it also encodes the precedence of operators. Specifically, the associativity listed *last* has the highest precedence. Sometimes you need to declare the precedence for operators that have no associativity, you can do this by placing `%noassoc X` at the appropriate place in this list.

The rules section contains rules such as the two rules below for expressions:

```
expression: TOKEN_INT
{
   string lexeme = GET_LEXEME($1);
   long int val = string_to_int(lexeme);
   Expression* e = AstInt::make(val);
   $$ = e;
}
| expression TOKEN_PLUS expression
{
   $$ = AstBinOp::make(PLUS, $1, $3);
}
```

Observe that the grammar encoded by these rules is

$$\textit{expression} \rightarrow \texttt{TOKEN\_INT} \mid \textit{expression}\ \texttt{TOKEN\_PLUS}\ \textit{expression}$$

The semantic actions to build an AST as specified as C++ code inside the braces. Bison allows you to access the attribute (or value) of each symbol in your grammar using special variables that start with a `$`. Specifically, the variable `$$` refers to the attribute of the non-terminal on the left-hand side and the variables `$1` through `$n` refer to the attribute of terminals and non-terminals on the right-hand side.

For example, in the rule

```
expression: expression TOKEN_PLUS expression
{
   $$ = AstBinOp::make(PLUS, $1, $3);
}
```

the variable `$$` is the value of the left-hand side (expression). This value is computed as an AST node whose children are the value of the first (`$1`) and third (`$3`) symbol (both expressions) on the right-hand side. Similarly, the action for `TOKEN_INT` gets the lexeme from the token using variable `$1` and assigns the resulting AST node for integers to the variable `$$`. All your actions should be of this general form.

Observe that the C++ type of all `$` variables is `Expression`. Sometimes you know that the type of a particular `$i` is more specific; in this case you must cast the value as explained earlier.

3.1 Precedence of Rules

In addition to allowing precedence declarations on terminals, `bison` also supports precedence declaration on rules. Specifically, the L manual specifies at the end of section 5 that all L expressions expand as far to the right as possible. You can encode this in `bison` by specifying a dummy terminal `EXPR` with the lowest associativity by placing the line

```
%nonassoc EXPR
```

at the top of your associativity declarations and adding `%prec` to the end of any rule that is ambiguous to force expressions to extend as far right as possible. For example, you can disambiguate the rule for the if statement as follows:

```
TOKEN_IF expression TOKEN_THEN expression TOKEN_ELSE expression %prec EXPR
```

Read the `bison` manual for an in-depth explanation of the `%prec` declaration.

3.2 Using Bison

Recall from lecture that `bison` does not work with any arbitrary grammar. You must read the `bison` manual to understand how to write your grammar so it works with `bison`. If there is a problem with your grammar, `bison` will report either a *shift-reduce* error or a *reduce-reduce* error. Refer to the manual to understand how to remove these errors.

It is imperative that you add **one rule at a time**, compile, check the output for bison errors such as shift/reduce and reduce/reduce conflicts, understand their source and fix them. Then, test the rule you just wrote to ensure it actually creates the AST node you expect it to create. **Typing the entire grammar first and trying to fix these errors later is impossible. No one has ever successfully completed a `bison` assignment this way and the course staff will also not be able to help you debug your errors**.

Important: **You must check the output on the console after each compilation for errors. Bison will still build a parser even if errors are present; it will just not work as expected.**

4 Files and Directories

To get started, create a directory where you want to do the assignment on any William & Mary computer science computer and execute the following command in that directory:

```
/projects/cs345.tdillig/PA2/get-assignment
```

This command will copy a number of files to your directory. The only file you will need to modify for this assignment is `parser.y` This file contains a skeleton for a parser for L. Except for the sections indicated, you are welcome to make modifications to our skeleton. You can actually build a parser with the skeleton description, but it only parses only very few constructs. You should understand all the provided constructs before adding your own and read the `bison` manual.

4.1 Building, Running and Testing your Parser

To compile your parser, simply type

```
make
```

in your directory. This will build a binary called `parser` that you can run. For example, to run your parser on the file `test.L`, you type:

```
./parser test.L
```

Once your parser is finished, this should print the AST of your program to the screen. For your reference, you can also run a reference parser whose binary we provide. To run the reference parser, type:

```
/projects/cs345.tdillig/parser test.L
```

A big component of this assignment is to test your parser thoroughly. It is your responsibility to ensure your parser accepts all legal constructs in L as specified in the reference manual, rejects all invalid ones and never crashes. You will want to feed many different inputs to your parser to test it.

4.2 Turning in and Grading

You must hand in the following for this assignment:

- The file `parser.y` containing your parser
- Five interesting test cases to test your parser in a file called `tests.L`

For this assignment, you will receive 20% credit for your test cases and 80% for your parser. We will test your parser automatically on the best selection of submitted test cases, so it is very much in your interest to have your tests included.

Important: Since we grade your parser automatically, do not print anything in your final parser since this will confuse our grading script and potentially result in a bad grade for you.