CS 241 — Spring 2020 — Assignment 8

Assignments for CS 241

← Assignment 7	Assignment 8	Assignment 9 →
~~Friday, July 10th, at 5:00 pm~~ Sunday, July 12th, at 5:00 pm	**~~Friday, July 17th, at 5:00 pm~~ Sunday, July 19th, at 5:00 pm**	Wednesday, July 29th, at 5:00 pm
	P1 • P2 • P3 • P4 • P5 • P6	

In Assignment 8, you will implement the semantic analysis phase of the WLP4 compiler for which you wrote a scanner in Assignment 6 and a parser in Assignment 7. In particular, it is the job of this assignment to catch all remaining compile-time errors in the source program; a program that makes it through this phase of compilation is guaranteed to be free of compile-time errors.

Note: In Assignments 9 and 10, you will build on your solution for this assignment. In order to proceed to those assignments, you must, at minimum, complete the part of this assignment devoted to building the symbol table (i.e., you must at least do Problem 1). As long as your solution does at least this much correctly, you will not be disadvantaged when you start Assignment 9. However, in order to properly complete Assignment 10 (which deals with pointer operations), you will need a working type-checker.

Resources

The specific semantic rules that must be enforced are those outlined in the "Context-sensitive Syntax" section of the WLP4 Specification. A summary of the semantic rules of WLP4 related to types is available for download, and you should use it as a reference. The summary contains formal descriptions of type rules for WLP4; although these should match the English descriptions on the WLP4 specification page, the WLP4 specification should be considered the definitive resource. Please report any discrepancies between the two documents.

Testing

You must test your semantic analyzer yourself in Linux. To test your semantic analyzer, you will need .wlp4i files (WLP4I format) as generated by the parser specified in Assignment 7. In case you do not wish to use your own parser, you can use one that we have provided. Invoke it using the command `wlp4parse`.

Starting from a .wlp4 source file, the complete sequence of commands to test your semantic analyzer on it is:

```
wlp4scan < foo.wlp4 > foo.scanned
wlp4parse < foo.scanned > foo.wlp4i
racket wlp4gen.rkt < foo.wlp4i
```

or

```
wlp4scan < foo.wlp4 > foo.scanned
wlp4parse < foo.scanned > foo.wlp4i
./wlp4gen < foo.wlp4i
```

This can be abbreviated to

```
cat foo.wlp4 | wlp4scan | wlp4parse | racket wlp4gen.rkt
```

or

```
cat foo.wlp4 | wlp4scan | wlp4parse | ./wlp4gen
```

It is strongly recommended that you create a parse tree which can then be traversed to check for various compile-time errors. It may be useful to store the following items at each node in the tree:

- The rule at the node: e.g., "start BOF procedures EOF"
- A list of tokens: e.g., "start", "BOF", "procedures", "EOF"
- A list of child nodes.

Future assignments will build upon the code written in this assignment. In future assignments you may want to add additional capabilities to nodes in the tree, for example keeping track of what the node does (addition, subtraction, etc.).

Remember the leaf nodes of the parse tree are terminals, so you can detect if you reached a leaf in the tree if the LHS of the rule is one of: "BOF", "BECOMES", "COMMA", "ELSE", "EOF", "EQ", "GE", "GT", "ID", "IF", "INT", "LBRACE", "LE", "LPAREN", "LT", "MINUS", "NE", "NUM", "PCT", "PLUS", "PRINTLN", "RBRACE", "RETURN", "RPAREN", "SEMI", "SLASH", "STAR", "WAIN", "WHILE", "AMP", "LBRACK", "RBRACK", "NEW", "DELETE", "NULL".

Recall that in Assignment 3 Problem 1 you had to read a tree as part of your solution. You may find your solution to A3P1 helpful for starting this assignment.

Each solution in this assignment builds upon the previous question: for each *n*, your solution for A8Q*n* should also be a solution for A8Q*n-1*.

For the questions involving building the symbol table, you may assume that during testing, no variable or procedure name contains ERROR.

Marmoset Notes

For each question, Marmoset has public and release tests, and often has secret tests as well. The weights assigned to each question are given on the assignment.

When you release test a submission, Marmoset will only report on public and release tests. No information about secret tests will be visible until after the assignment deadline.

This means the number of marks you see on Marmoset (prior to the assignment deadline) will not necessarily match the number of marks shown on the assignment page.

Problem 1 — 15 marks of 85 (filename: `wlp4gen.rkt` or `wlp4gen.cc`)

For problem 1, assume that the WLP4 file contains only the procedure wain. In other words, assume that the production `procedures → procedure procedures` is not used, so that the non-terminal `procedures` is guaranteed to derive `wain` directly.

Implement reading in a .wlp4i file (WLP4I format) and creating a parse tree. Modify the program so that it collects information about declarations and types. To solve this problem, you must build a symbol table that contains an entry for every declared identifier in the program (including the parameters of wain). If an identifier is declared more than once, or is used without first having been declared, you must output a string containing ERROR to standard error. You are encouraged, but not required, to make your error message more descriptive than this, as long as the string ERROR occurs. These are the only errors you need to check for this problem.

If the input program contains no errors of the kind described above, then you must output a symbol table to standard error containing the name and type of every identifier declared in the program (including the parameters of wain). The symbol table must be formatted as follows: the first line contains the name of the procedure (wain); subsequently, the output contains one line for each declared identifer, and each line consists of the (case-sensitive) name of the identifier, followed by a space, followed by the identifier's type (`int` or `int*`). The order in which the entries in the symbol table occur is not important.

For example, if the input is

```
int wain(int *a, int b) {
  int c = 0;
  return *a + b + c;
}
```

then the output (on stderr) should be

```
wain
a int*
b int
c int
```

Click here to go back to the top of the page.

Problem 2 — 5 marks of 85 (filename: `wlp4gen.rkt` or `wlp4gen.cc`)

Extend your solution for Problem 1 to account for the possible presence of other procedures in the program, but still only process procedure wain. For example, if the input is

```
int id (int x) { return x; }
int wain(int *a, int b) {
  int c = 0;
  return *a + id(b) + c;
}
```

then the output (on stderr) should be

```
wain
a int*
b int
c int
```

Click here to go back to the top of the page.

Problem 3 — 10 marks of 85 (filename: `wlp4gen.rkt` or `wlp4gen.cc`)

Extend your solution for Problem 2 to include signatures for all procedures encountered, but still only show the variables local to procedure *wain*. The signature of a procedure is the sequence of *parameter* types that the procedure was declared to take; it does not include the result type. Between procedures, leave an extra newline, so that we can tell where one procedure ends and the next one begins. For this problem, you must indicate an error (by writing a string containing ERROR to stderr) if a procedure is declared more than once.

For example, when using the same input to the previous problem, the output on stderr should be:

```
int id (int x) { return x; }
int wain(int *a, int b) {
  int c = 0;
  return *a + id(b) + c;
}
```

then the output (on stderr) should be

```
id int

wain int* int
a int*
b int
c int
```

Click here to go back to the top of the page.

Problem 4 — 15 marks of 85 (filename: `wlp4gen.rkt` or `wlp4gen.cc`)

Extend your solution for Problem 3 to include type information for variables in all procedures. Between procedures, leave an extra newline, so that we can tell where one procedure ends and the next one begins. If any variable is declared more than once in the same scope, or used in a scope in which it is not declared, output a string containing ERROR to stderr. If any procedure is declared more than once, or called in a context in which it is not accessible, output a string containing ERROR to stderr. By the end of this problem, all errors related to declarations and scope should be caught.

For example, if the input is:

```
int id (int x) { return x; }
int wain(int *a, int b) {
  int c = 0;
  return *a + id(b) + c;
}
```

then the output (on stderr) should be

```
id int
x int

wain int* int
a int*
b int
c int
```

Click here to go back to the top of the page.

Problem 5 — 20 marks of 85 (filename: `wlp4gen.rkt` or `wlp4gen.cc`)

Extend your solution for Problem 4 to check for type errors in expressions. In particular, for every subtree with an `expr` or `lvalue` node at its root, compute the type associated with that subtree, or output a string containing ERROR if the expression cannot be validly typed. If every expression in the program has a valid type, and there are no declaration errors, then your program should produce no output.

For example, in a node corresponding to the rule

```
expr → expr PLUS term
```

if the `expr` and `term` on the right-hand side both have type `int*`, then your program should output ERROR to standard error. Otherwise, the `expr` on the left-hand side is well-typed, and its type is as given on the semantic rules handout.

Click here to go back to the top of the page.

Problem 6 — 20 marks of 85 (filename: `wlp4gen.rkt` or `wlp4gen.cc`)

Extend your solution for Problem 5 to check for type errors in statements and tests. These are program elements that do not themselves have types, but demand that some of their subelements be given particular types. For example, `println` can only be used with type `int`, and `delete []` can only be used with type `int*`.

By the end of this problem, your program should check and enforce every type rule given on the semantic rules handout. If a source program contains any kind of semantic error, you must output a string containing ERROR to standard error. If the source program is free of semantic errors, your program should produce no output.

Click here to go back to the top of the page.