CS 241 Week 9 Tutorial
Semantic Analysis and Code Generation
Fall 2019

1 Symbol Table Building and Type Checking

1.1 Symbol Tables
In the MIPS assembler you wrote for Assignments 3 and 4, you had to check for duplicate labels in one pass and check for missing labels in a second pass. Why is this not necessary in the WLP4 compiler?

1.2 Type Checking
Determine if each of the following WLP4 code fragments is well-typed.

```wlp4
int foo(int x, int y){
    return x + 7 * y + 1;
}
...
int a = 0;
int b = 0;
int* c = NULL;
int* d = NULL;
(1) *(d+(((c-&b)+d)-(c+(a*b))))=(c-d+*new int[d+b-c]);
(2) if(*(c+a%b)<(&a-&b)){println(&*&*c-(&b));}else{delete[]*d+&a-c;}
```

2 Code Generation Conventions and Tips
These are a collection of non-mandatory tips which will most likely make it easier to write your code generator.

- All expressions should store their result in $3. This will make it much larger to combine small expressions into larger ones.
- When evaluating expressions, store any intermediate results on the stack rather than in registers. This is easier to reason about and makes it much harder to make mistakes.
- Define some helper functions that return segments of code that you will need to generate often. In particular, you may want functions which generate the MIPS code for push and pop.
- Store the value 4 in $4 and 1 in $11, or some other suitable registers. You’ll use these constants on a regular basis and it helps to have them consistently available.
Output comments as part of your assembly code. It is extremely difficult to debug generated assembly, and even comments such as “addition starts/ends” or similar will help with debugging.

3 Pre- and Post-Increment Code Generation

Recall that C and C++ have the pre- and post-increment operators ++i and i++. Suppose we added the following rules to the WLP4 grammar:

\[
\text{factor} \rightarrow \text{PLUS PLUS lvalue} \\
\text{factor} \rightarrow \text{lvalue PLUS PLUS}
\]

We will assume that scanning and parsing all work out, and that these operators can only be used on INT types. Write type rules for these operators and pseudocode to generate the correct MIPS output for each of these grammar rules.