COSC 312 – Homework #8b

Turing Machine Design: C++ Pre/Post-Increment Syntax Parser

Group 9: Brandan Roachell and Rob Bray

April 3, 2021

Collaborators

- Brandan Roachell: Designed and implemented recognition of the increment operator before and after the variable, parenthesis pair matching, and helped with test cases. Created final writeup in LaTeX.
- Rob Bray: Designed and implemented the compiler-like error message system that points to the first instance of improper syntax and assigns it an error code if it has one. Helped with the design of the increment operator recognition and did most of the legality testing with obscure syntax cases. Wrote rough draft of writeup.

The Idea

Our idea for this Turing machine originally started as an idea Dr. Berry threw out after class. We wanted to create a machine that could recognize syntax for the increment operator "++" and parentheses in the C++ programming language. It decides whether the input is a valid line of code that increments a variable `a` in C++ and displays an error code if not. We liked the idea because it sounded pretty feasible and would provide some hands-on experience with how a compiler might behave while parsing code. It is also interesting because C++ has more different rules for valid incrementing than one may expect, leading to some interesting cases and discovery of interactions between "++" and parentheses.

How Our Machine Works

Valid input:

- This machine *only* recognizes the increment operator (i.e., "++"). It does not handle general addition expressions such as `a+a` and will reject any string of `+`'s that is not even in number.
- The variable being incremented MUST be `a`, and there cannot be more than one instance of `a`.
- The tape input must end with a semicolon to be accepted, as it could not even be considered a valid C++ line otherwise. Anything following the semicolon is also ignored.
- No whitespace is permitted anywhere in the input.
- Other than the above clarifications, our TM is intended to follow the grammar rules of C++ as they relate to pre/post-incrementing and parenthesis placement.

The alphabet is {`a,+,(,),;`}. See Test Cases section for concrete examples.

We have learned that the C++ increment operator grammar rules can be briefly summarized as follows:

- There can be as many pre-increments as you want, as long as they are valid (even number of `+`'s).
- There can be no more than one post-increment anywhere.

- The post-increment can *only* happen on a higher "level" than any pre-increment. For example, `++a++;` has both a pre and post-increment on level 0, so this is invalid. `++(a++);` has a pre-increment on level 1 but a post-increment on level 0, so it is invalid. `++(++a)++;` has a pre and post-increment on level 1, so it is invalid. `(++a)++;` has a pre-increment on level 0 and a post-increment on level 1, so it is valid.

It is the last two parts of the grammar that our machine has trouble with, and we further detail this in the Limitations section. For now, we will describe all of the major parts and features of this TM.

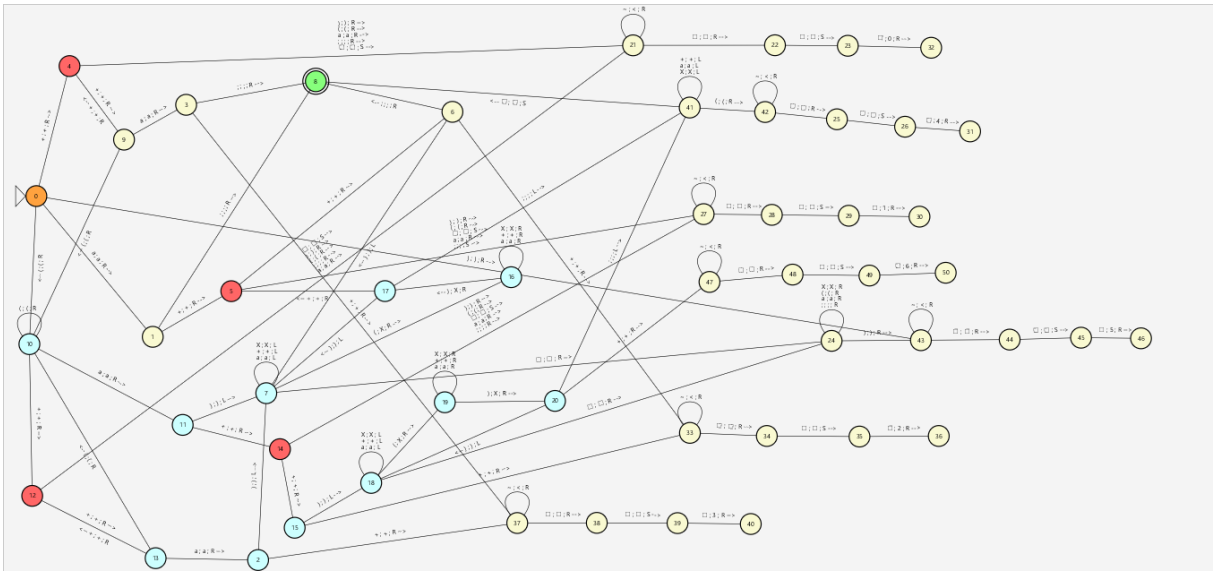

Figure 1: An overview of our Turing machine. Orange highlights the start state. Red states indicate places where a `+` MUST be read to be a valid increment statement, blue states indicate the set of states that handles input with parentheses, and the green state is the accept state. Error code handling can be seen on the right.

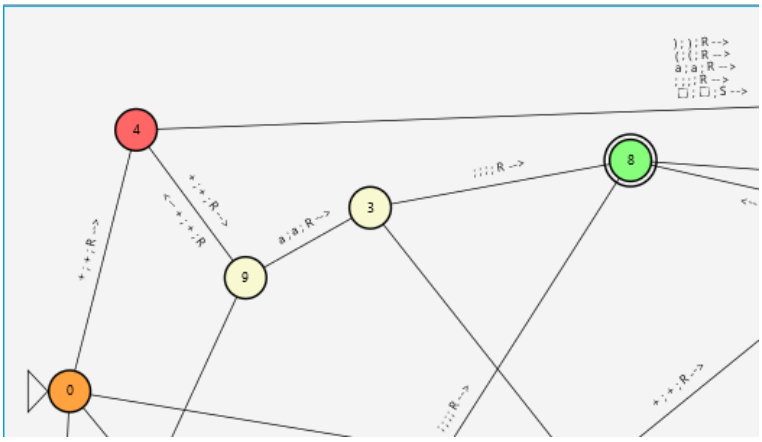

Figure 2: States 4, 9, and 3 recognize simple pre-increment statements like `++++a;`, and 4 and 9 ensure that an even number of `+`'s is being read. If there is an odd number (state 4) and anything other than another `+` is read, error code 0 will be returned (not pictured).

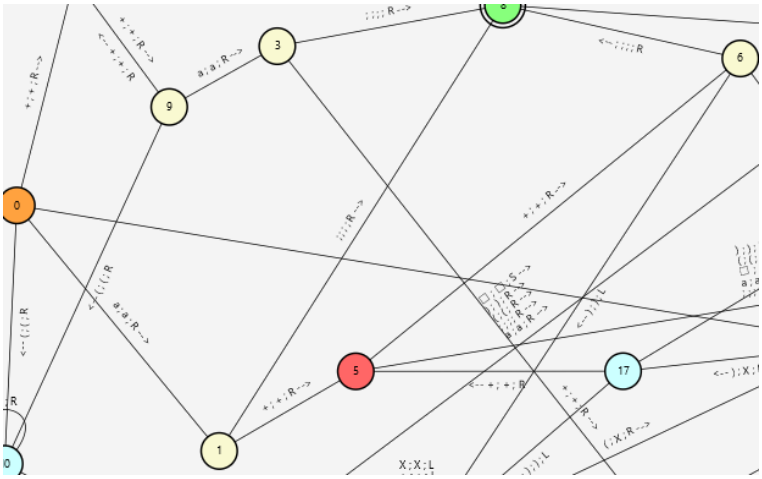

Figure 3: Similarly, states 1, 5, and 6 recognize simple post-increment statements like `a++;`, and 5/6 ensure that exactly two `+`'s are being read. If there is only one, error code 1 will be returned (from state 5, not pictured), and if there are more than two `+`'s, error code 2 will be returned (transition from state 6, not pictured). Note that `a;` is also a valid statement: while nothing incremented, it is valid because there are zero attempted increments.

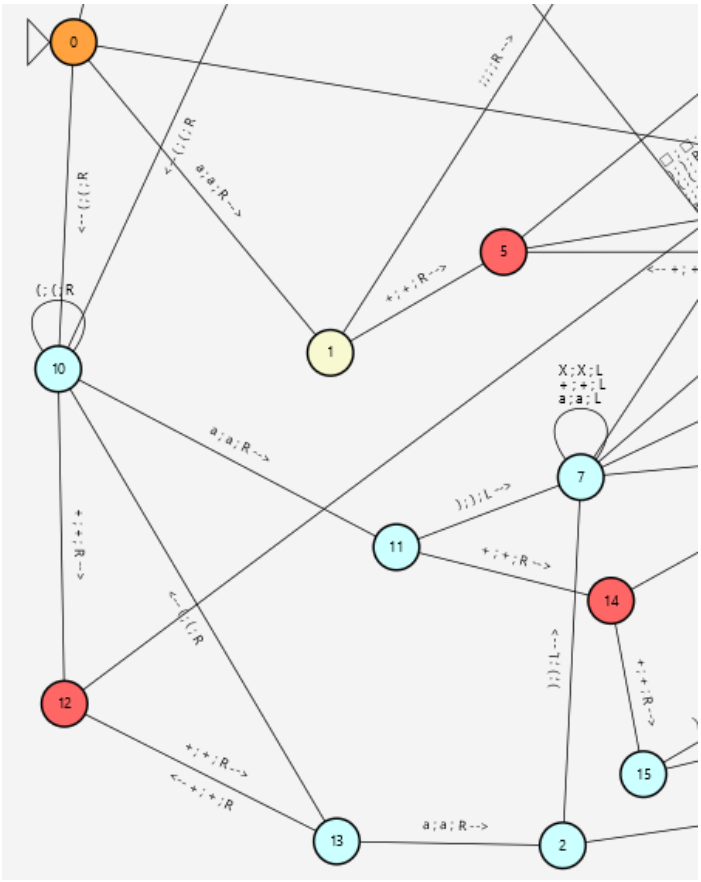

Figure 4: Using similar logic, states 10, 11, 12, 13, 14, and 15 handle pre/post-increments following a left parenthesis. However, difficulty comes with trying to handle parentheses alongside incrementing. Also note that you may read any number of `(`'s at state 10 or repeat 10→12→13→10.

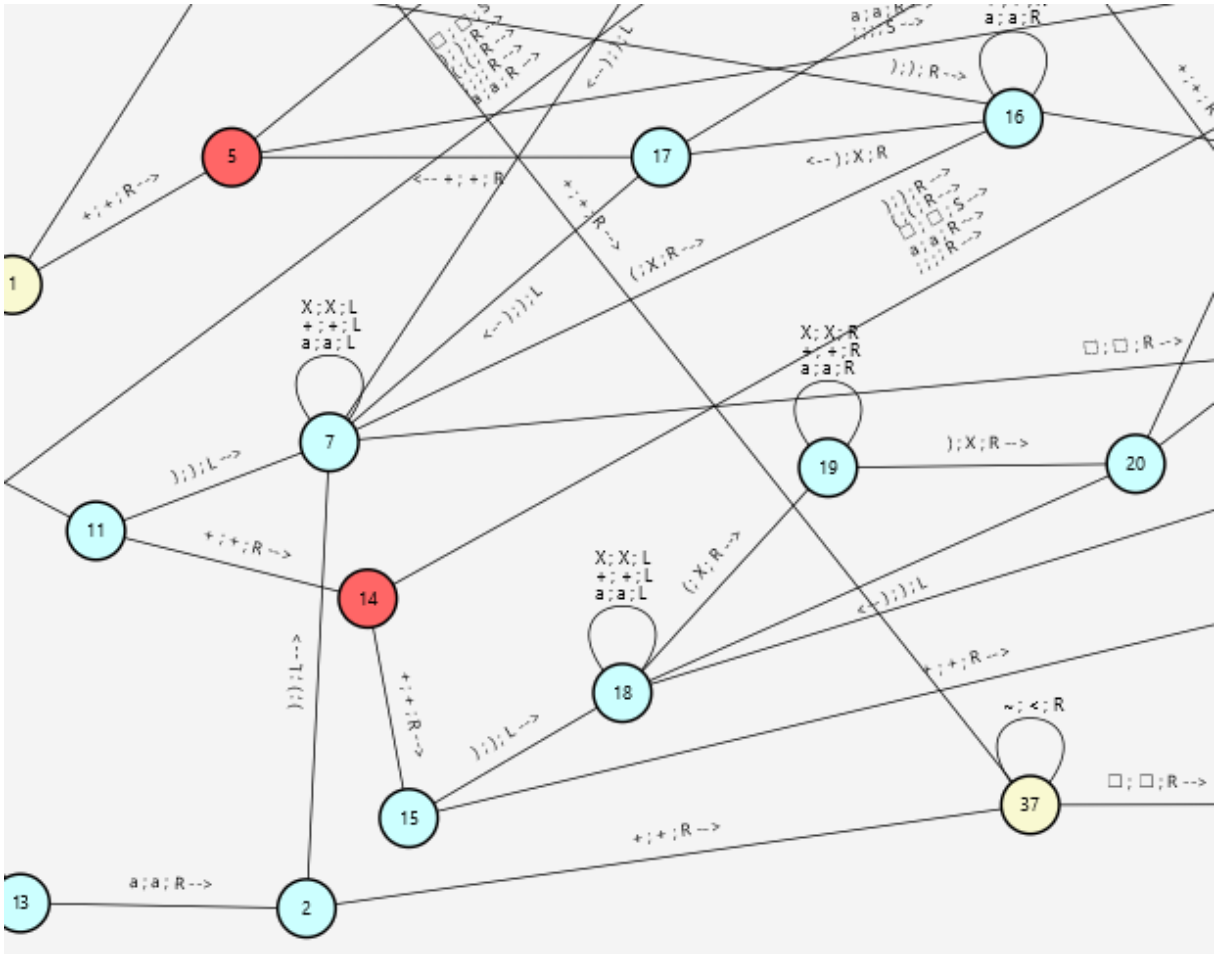

Figure 5: Now here is a more sophisticated feature of the machine—parenthesis matching. Before describing it, note there are two branches here that are identical (11→7→16→17 and 15→18→19→20), and the purpose is to remember if a post-increment has already been seen or not. Since no more than one post-increment is allowed anywhere, it is necessary for the machine to keep track of what is no longer allowed, despite the syntax being valid. We do this by branching to what is essentially another copy of the machine that excludes the post-increment transition option (states 15-20, and 20→47 produces error code 6). As for the description of how parenthesis matching works here, when the machine reads `)`, it will perform the following actions:

1. Read and write `)`, but go left.
2. Continue left until a `(` is found. If `(` is not found, produce an error.
3. Upon reading `(`, write `X`.
4. Now go right until `)` is found again. Write `X` for `)`.
5. It has basically removed this set of parentheses from being read by the counter again, and before the machine can accept, this process repeats until no right parentheses remain.

There is a similar method for checking for a missing `)` or extra `(` around transition 17→41 and 20→41, where it searches backwards for any remaining `(`'s before accepting. If it finds a leftover `(`, it produces an error instead.

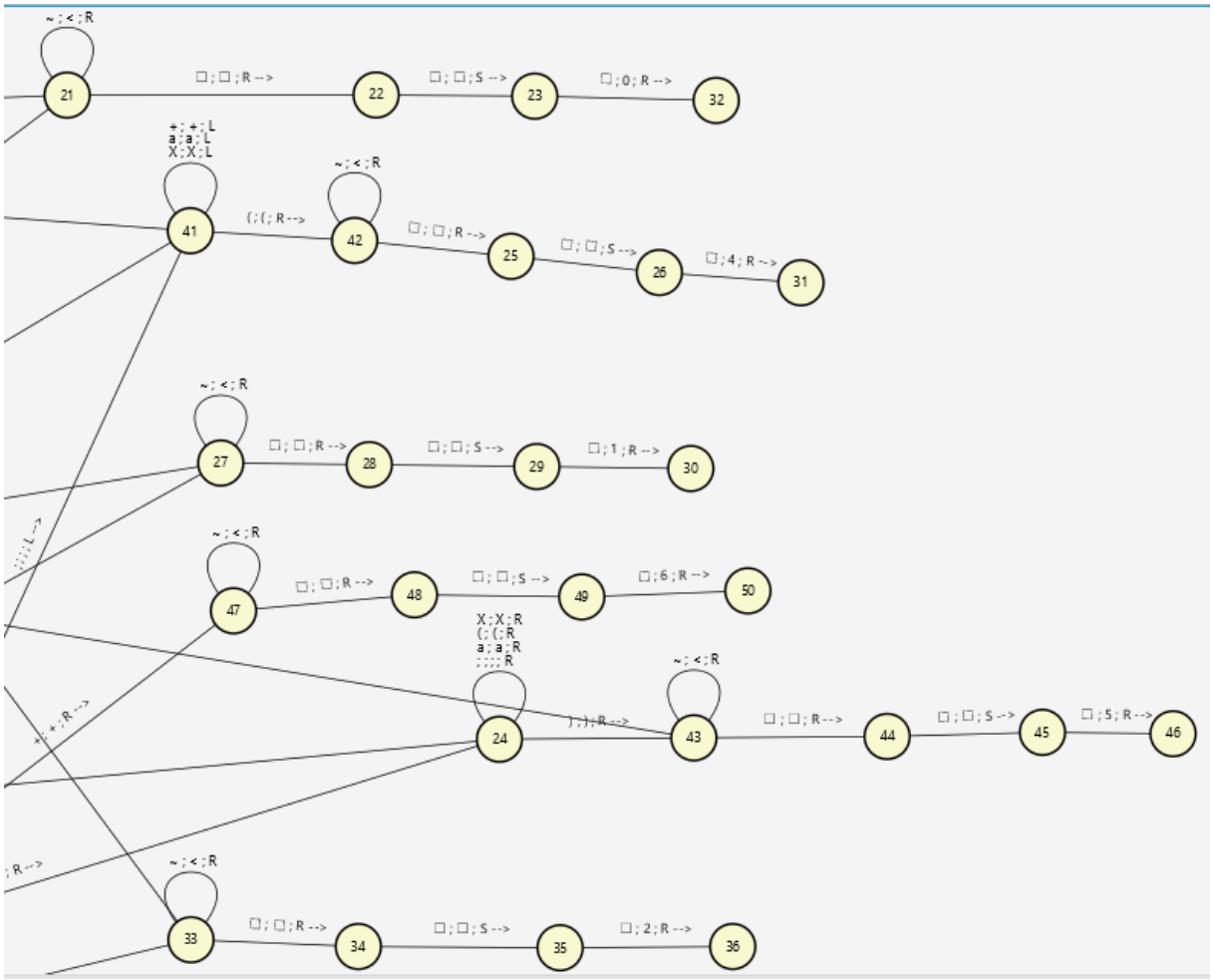

Figure 6: Error message generation. More about error codes below, but the TM produces error messages by printing <'s after the point of rejection until the end of the input, a space, and then the corresponding error code number. There is a hard-coded transition (0→43) for) being the first character.

Error Codes

If input is found to be invalid and the error is related to parsing, one of the following numbers will be returned, along with a pointer to the location containing the error (other issues will cause the machine to halt but will not have an error code):

0. odd number of +'s in pre-increment
1. only 1 + in post-increment
2. more than 2 +'s in post-increment
3. tried to pre-increment on the same or higher "level" as a post-increment
4. no matching right parenthesis found for this left parenthesis
5. no matching left parenthesis found for this right parenthesis
6. more than one post-increment

Test Cases

We have come up with the following test cases and noted what our machine does with them and whether it agrees with C++'s grammar. These test cases do not include improper input (e.g., missing semicolon, multiple `a`'s) or any parenthesis counting cases, since we believe they are guaranteed to halt and reject since they are separate from the complex logic of the parsing. However, our machine does not correctly handle all of the cases (see How Our Machine Works and end of Test Cases), and we explain more in the following section.

Note: "accept/reject" is the behavior of the TM, and "legal/illegal" is C++'s verdict. The number following rejects is our error code.

- `++++++++a;` - accepts and is legal
- `a+;` - rejects (1) and is illegal
- `a++++;` - rejects (2) and is illegal
- `++a++;` - rejects (3) and is illegal
- `(++a)++;` - accepts and is legal
- `(++(++a))++;` - accepts and is legal
- `(a)++;` - accepts and is legal
- `((a)++);` - accepts and is legal
- `((a++)++);` - rejects (6) and is illegal
- `(++(a));` - accepts and is legal
- `++(++(++(++a)));` - accepts and is legal
- `++(++(+(++a)));` - rejects (0) and is illegal
- `++(++(++(+a)));` - rejects (0) and is illegal
- `+(++(++(++a)));` - rejects (0) but compiles in C++; however, we do not consider it a legal post-increment, so this is intended
- `+a;` - rejects (0) but compiles; same reasoning as above case
- `(++++++(++a));` - accepts and is legal
- `++++++(++++(++a));` - accepts and is legal
- `++++a();` - rejects and is illegal
- `a+(a);` - rejects (1) and is illegal
- `((((a++)))++);` - rejects (6) and is illegal
- `(++(a++));` - accepts but is illegal (should be 3) (see limitations)

- `(++(++a)++);` - accepts but is illegal (should be 3) (see limitations)
- `(++(a)++);` - accepts but is illegal (should be 3) (see limitations)
- `((a)++)++;` - accepts but is illegal (should be 6) (see limitations)

Limitations

After constructing the parentheses recognizer, we ran into some more complicated test cases that our machine does not correctly decide, and because of the time constraint, we were not able to implement all of the necessary fixes. However, we will describe in detail how we planned to fix the remaining known problems. The problems stem from how the parentheses interact with the increment operators. Our machine does parenthesis matching completely separately from parsing, which may be a flaw in the design itself. We do have some potential fixes for the machine in its current state.

The issue with error code 3 (pre-incrementing on the same or a higher level than a post-increment) is that input where the levels are separated by parentheses (i.e., any case other than `++a++;` or `(++a++);` with as many pairs of parentheses) requires the machine to recognize what level it is on and check for `+`'s before it like the case `(++(a)++);`, where both increments are on level 1, but the machine does not remember whether it can post-increment here and has no sense of level. Our proposed solution is to do the following when it reads a post-increment:

1. Read the second `+` and go left twice. Read `X`.
2. Continue left until `a` is found. If a `+` is found before `a`, then there is more than one post-increment—return error 6. If `X` is found again before `a`, then this is a higher level. Somehow "remember" this.
3. Continue left. It must read as many `X`'s as it "remembered," but `+`'s are fine here. This is to get to the opposite matching parenthesis.
4. Read the matching `X`. Now it must continue left until it finds a blank space (the beginning of the input). It cannot read a `+`, or else this is a pre-increment occurring on the same or higher level—return error code 3 if this happens.

The parsing of the input `++(++a)++;` when it reaches the post-increment is as follows:
`++X++aX+;` (1 - read the second plus, go left twice)
`++X++aX++;` (2 - read `X` and going left)
`++X++aX++;` (3 - found `a` and continuing left)
`++X++aX++;` (4 - skipping valid pre-increment, found left `X` and going left—no more `+`'s allowed past this point)
`++X++aX++;` (5 - read a `+`, return error 3)

The issue with error code 6 (more than one post-increment) could also be solved with this fix, since it is caught in step 2 while going left.