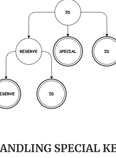
Scanner

Jose Andreas - 119010501

BASIC IDEAS

- There are four token classes (**ID**, **INT_NUM**, **SPECIAL**, and **RESERVED**).
- There are two types of token sets, namely, **INT_NUM** sets and **ID**.
- The set **ID** contains everything, be it **SPECIAL**, **RESERVED**, or **INT_NUM**.
- **ID** instances always starts with an alphabetical character.
- **ID** instances cannot contain any **SPECIAL** characters.
- `Scanner::scan()` only scans for **ID** after it exhausts every other possibility.

SCANNER CODE LOGIC



The picture above explains it all. The code traverses through the whole file character by character (albeit it also peeks at the next character for each iteration per se) and tries to matches it with the set of **SPECIAL** and **RESERVED** keywords. If it doesn't match, it stays in the same state (either **ID** or **INT_NUM**), eats up the current character, and continues to the next character.

TOP-LEVEL MATCHING RULES

HANDLING SPECIAL KEYWORDS

Note that **SPECIAL** entries may contain more that one character. Hence, there need to be rules for dealing with such cases. Fortunately, the following bit of code does just that.

```
#define str(character) std::string(1, character)
if (is_special(str(ch))) {
    // s_char -> s_char_tail.
    if (is_special(str(ch) + tail)) {
        // terminating state -> id.
        save();

        // store the first special char.
        buffer += ch;
        break;
    };

    // buffered_s_char -> s_char.
    if (is_special(buffer + ch)) {
        buffer += ch;
        // terminating state -> two length special.
        save();
        break;
    };

    // terminating state -> id | int_num on nonempty buffer.
    save();

    // terminating state -> single length special.
    buffer = ch;
    save();
    break;
};
```

The code essentially enforces the following rules:

1. IF the next input character is also **SPECIAL**, IF the combination matches a different **SPECIAL** keyword, then `remember` and `continue` to next character.
2. IF the next input character is also **SPECIAL**, yet, the combination doesn't form another **SPECIAL** keyword, `return` **SPECIAL**.
3. IF the next input character is not **SPECIAL**, `return` **SPECIAL**.

Or, in grammar form:

```
//
// WARNING:
//  the following snippet is only for
//  explanatory purposes, not much else.
//

expr:
  SPECIAL tail;

tail:
  SPECIAL {
    if (matches) {
      add_to_buffer(ch);
      return;
    };
    create_token("special", buffer);
    clear_buffer();
    return SPECIAL;
  };
  ID {
    create_token("special", ch);
    clear_buffer();
    return SPECIAL;
  };
  INT_NUM {
    create_token("special", ch);
    clear_buffer();
    return SPECIAL;
  };
```

So, the bottom line is, the scanner checks if the current input and characters that follow can merge into a **SPECIAL** character. If they can't, the scanner will just follow the top-level matching rules (in other words, return `SPECIAL`).

HANDLING RESERVED WORDS

Like its needy token class counterpart, **RESERVED** words also needs to be handled with extra care. It is because an **ID** may or may not contain **RESERVED** words. Fret not, however, as the solution is really simple; Simply check if the merger between the current **RESERVED** word and the next character is yet another **RESERVED** word.

```
//
// context:
// buffer + ch forms the current word.
// tail refers to the next character.
// EOF denotes the end of file.
//

// terminating state -> reserved word.
if (is_reserved(buffer + ch)
    && (tail == ' ' || tail == '\n' || tail == EOF)) {
    buffer += ch;
    save();
    break;
};
```

HANDLING INT_NUM

Because of the way **INT_NUM** behaves, it is the most clear-cut token class to manage. It's always between multiple copies of itself (a chain of **INT_NUM**s) or a single copy of itself. So, when the next character is no longer an **INT_NUM**, the scanner will only `return` **INT_NUM** if, and **ONLY** if, the every character is an **INT_NUM**. Below is a snippet that handles **INT_NUM** situated at the edge.

```
// edge case handling for when int_num is at the edge.
//
// breaking point for integers,
// edge case number -> space or end of line.
if (tail == ' ' || tail == '\n' || tail == EOF) {
    save();
};
//
break;
```

HANDLING ID

EVERYTHING that is not **INT_NUM**, **SPECIAL**, or **RESERVED**, Quite simple, really.

NFA GRAPH

The NFA is made by following the basic ideas mentioned previously.

DFA GRAPH

The DFA is constructed by adding some extra conditions and a terminating state. Do note that the code implementation only categorizes the input types into **ID** and **INT_NUM** (the third, **SPECIAL** is expressed through several if-statements and acts as an indicator for breaking points).