# Lecture 15: Project 1 Related

# Bison/Flex interface

- Lexer communicates syntactic categories of tokens as integers.
- These may be defined in the Bison file as symbolic constants (in `%token`, `%left`, `%right` declarations).
- In addition, single-character tokens, by convention, "represent themselves": *e.g.*, to indicate a token written as `';'` in the parser, the lexer returns the character `';'`, which is an integer in C/C++, as it is in Java.
- Lexer communicates semantic values to attach to tokens in the variable `yylval`, a C++ union defined by the `%union` declaration in the Bison file.
- The `%token<...>` and `%type<...>` declarations in the Bison file tell which branch of the union should contain the semantic value of the symbols they declare.
- The Bison parser calls the function `yylex` to get the next token's syntactic and semantic values.

# Lexer Features

- In lexical actions, `yytext` is a *C* string (type `char*`) containing the matched token, and `yyleng` contains its length.
- Actions that execute **return** cause the lexer to deliver a token (the returned token is the syntactic category).
- Actions that don't return indicate tokens that are skipped.
- It's always the action of the longest match that gets chosen (or the first in case of ties). As a result,

```
for   { return FOR; }
[A-Za-z][A-Za-z0-9]*  { return ID; }
```

will return `FOR` for the input "`for`" and `ID` for the input "`forage`," just as is usually intended.xs

# Lexer Features (II)

- You can define abbreviations above the first %% in the lexer file for use in patterns, as in

```
ALPHA    [a-zA-Z_]
ALNUM    [a-zA-Z_0-9]
```

  which allows you to write

```
{ALPHA}{ALNUM}* { rule for ID; }
```

- The converted Flex program is a C/C++ program. The actions are general C/C++ statements, as is code after the second %%, and indented lines in the rest of the parser.
- Use this for "special effects", such as keeping track of how many open and close brackets (`'()'`, `'[]'`, `'{}'`) you have seen so that you know when to ignore newlines.

# Lexer States

- The lexer is essentially a DFSA. You can define alternative starting states in this DFSA with `%s` and `%x` declarations above the first `%%`, as in

```
%x SPECIAL
```

- This says that patterns that start with `<SPECIAL>` match only when `yylex` starts the machine in state `SPECIAL`, and in that state, other patterns do not match.
- In actions, one can change the start state for subsequent calls with

```
BEGIN SPECIAL;
```

  to make `SPECIAL` the start state, or `BEGIN INITIAL` to go back to the default start state.
- I found this feature useful when implementing `INDENT` and `DEDENT`.

# Some Tricks

1. We've set up the skeleton so that `yylex` actually *gives* you a chance to do things before and after the FSA is called (with the function `yylex_raw`),
   - For example, when a certain change of indentation requires that you deliver three `DEDENT` tokens, you can arrange to have `yylex` return `DEDENT` three times before again calling `yylex_raw`.

- 2. A pattern cannot match 0 characters. However, you can get a similar effect by matching one (arbitrary) character and then having your lexical action put it back using `yyless`.

# Parser Points

- Keep semantic actions simple. For the most part, you don't need much other than, e.g.,

```
statement: "return" expr     { $$ = NODE(RETURN, $2); $$->setLoc (@1); }
```

- The `@1` notation means "the source location of `$1`" (`return`), and the `setLoc` method on nodes sets the node's location based on the argument.
- In the absence of `setLoc`, the `NODE` function will set the location of its result to that of the first child that has a location.
- Feel free to introduce new supporting functions after the second %%. You may need to forward-declare them before the first %%, as illustrated in the parser skeleton (e.g., `yyerror`).

# ASTs

- Each type of AST node contains an integer value indicating what type of node it is.
- Can use these to distinguish node types, and for Project 1, the default method declarations given in `AST` suffice for most of the project.
- For later projects, I suggest using a more OOP style, allowing different nodes to react in different ways without a specific test for node type.
- To this end, we've set up a mechanism that allows the `NODE` and `LEAF` choose subtypes of `AST` for specific node types. See the examples in `stmts.cc` and `exprs.cc`.
- Although you don't need to do tree-processing in this project, aside from building them, you may want to handle checks for improper placement of `break`, `continue`, `return`, `def`, and `class` by doing a recursive post-pass over the tree. Alternative is using some global variables in the parser.

# General Advice

- *Read the Project Documentation:* there actually is useful information there!
- *Read the Skeleton:* it gives some clues and contains work you need not do.
- *Read the Tool Documentation:* The manuals for Flex and Bison are online.
- *Read the C++ Documentation:* Especially concerning C++ library types and functions (`vector`, `string`, `set`, `map`, `algorithms`, `iostream`, and the C library.
- *Write Test Cases:* See also HW 4.
- *Use GIT:* Commit often (I have 37 commits just to change the previous year's solution to this year's). Learn how to coordinate with your partners.
- *Meet Regularly With Your Team.* Have a clear idea of what everyone's job is.