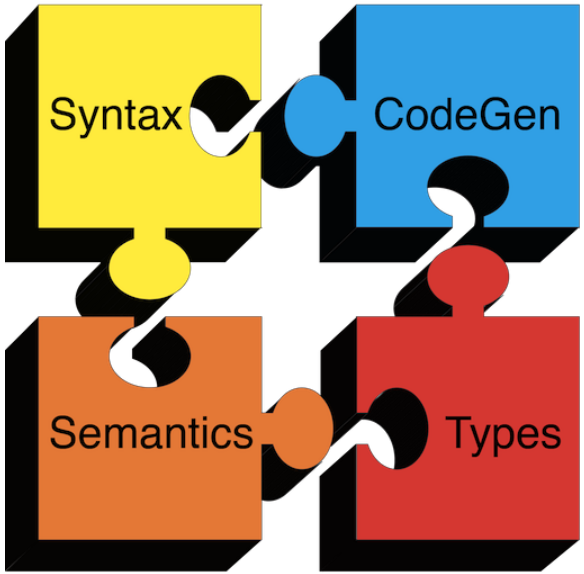

Compilers

Recursive Descent Algorithm

RD Algorithm

- Let TOKEN be the type of tokens
  - Special tokens INT, OPEN, CLOSE, PLUS, TIMES

- Let the global next point to the next input token

RD Algorithm

- Define boolean functions that check for a match of:
  - A given token terminal

    bool term(TOKEN tok) { return *next++ == tok; }
  - The nth production of S:

    bool S_n() { … }
  - Try all productions of S:

    bool S() { … }

RD Algorithm

- For production $E \rightarrow T$

  bool E_1() { return T(); }

- For production $E \rightarrow T + E$

  bool E_2() { return T() && term(PLUS) && E(); }

- For all productions of E (with backtracking)

```
bool E() {
  TOKEN *save = next;
  return    (next = save, E1())
         || (next = save,  E2());   }
```

RD Algorithm

- Functions for non-terminal T

bool T_1() { return term(INT); }
bool T_2() { return term(INT) && term(TIMES) && T(); }
bool T_3() { return term(OPEN) && E() && term(CLOSE); }

```
bool T() {
  TOKEN *save = next;
  return    (next = save, T1())
         || (next = save,  T2())
         || (next = save,  T3()); }
```

RD Algorithm

- To start the parser
  - Initialize next to point to first token
  - Invoke E()

- Easy to implement by hand

RD Algorithm

$E \rightarrow T \mid T + E$

$T \rightarrow \text{int} \mid \text{int} * T \mid (E)$

(int)

```
bool term(TOKEN tok) { return *next++ == tok; }

bool E1() { return T(); }
bool E2() { return T() && term(PLUS) && E(); }

bool E()  {TOKEN *save = next; return     (next = save, E1())
                                       || (next = save,  E2());   }
bool T1() { return term(INT); }
bool T2() { return term(INT) && term(TIMES) && T(); }
bool T3() { return term(OPEN) && E() && term(CLOSE); }

bool T() { TOKEN *save = next;  return    (next = save, T1())
                                       || (next = save,  T2())
                                       || (next = save,  T3()); }
```

RD Algorithm

Which lines are incorrect in the recursive descent implementation of this grammar?

$E \rightarrow E' \mid E' + id$

$E' \rightarrow -E' \mid id \mid (E)$

- ☐ Line 3
- ☐ Line 5
- ☐ Line 6
- ☐ Line 12

```
1  bool term(TOKEN tok) { return *next++ == tok; }

2  bool E1() { return E'(); }
3  bool E2() { return E'() && term(PLUS) && term(ID); }
4  bool E() {
5      TOKEN *save = next;
6      return (next = save, E1()) && (next = save,  E2());
7   }

8  bool E'1() { return term(MINUS) && E'(); }
9  bool E'2() { return term(ID); }
10 bool E'3() { return term(OPEN) && E() && term(CLOSE); }
11 bool E'() {
12     TOKEN *next = save;  return (next = save, T1())
13                                 || (next = save,  T2())
14                                 || (next = save,  T3());
15 }
```