Error Handling

- Purpose of the compiler is
  - To detect non-valid programs
  - To translate the valid ones

- Many kinds of possible errors (e.g. in C)

Error kind	Example	Detected by …
Lexical	… $ …	Lexer
Syntax	… x *% …	Parser
Semantic	… int x; y = x(3); …	Type checker
Correctness	your favorite program	Tester/User

Error handler should:

① Report errors accurately and clearly.
② Recover from an error quickly.
③ Not slow down compilation of valid code.

Three methods:

① Panic mode
② Error productions → present
③ Automatic local or global correction → past

Panic mode is simplest, most popular method

When an error is detected:
- Discard tokens until one with a clear role is found
- Continue from there

Looking for synchronizing tokens
- Typically the statement or expression terminators

- Consider the erroneous expression

  (1 + + 2) + 3

- Panic-mode recovery:
  - Skip ahead to next integer and then continue

- Bison: use the special terminal error to describe how much input to skip

  E → int | E + E | (E) | error int | (error)

  Normal

Error productions specify known common mistakes in the grammar.

Example:

Write 5 x instead of 5*x

Add the production E → ... | E E

Disadvantage

Complicates the grammar.

(warning but not error)

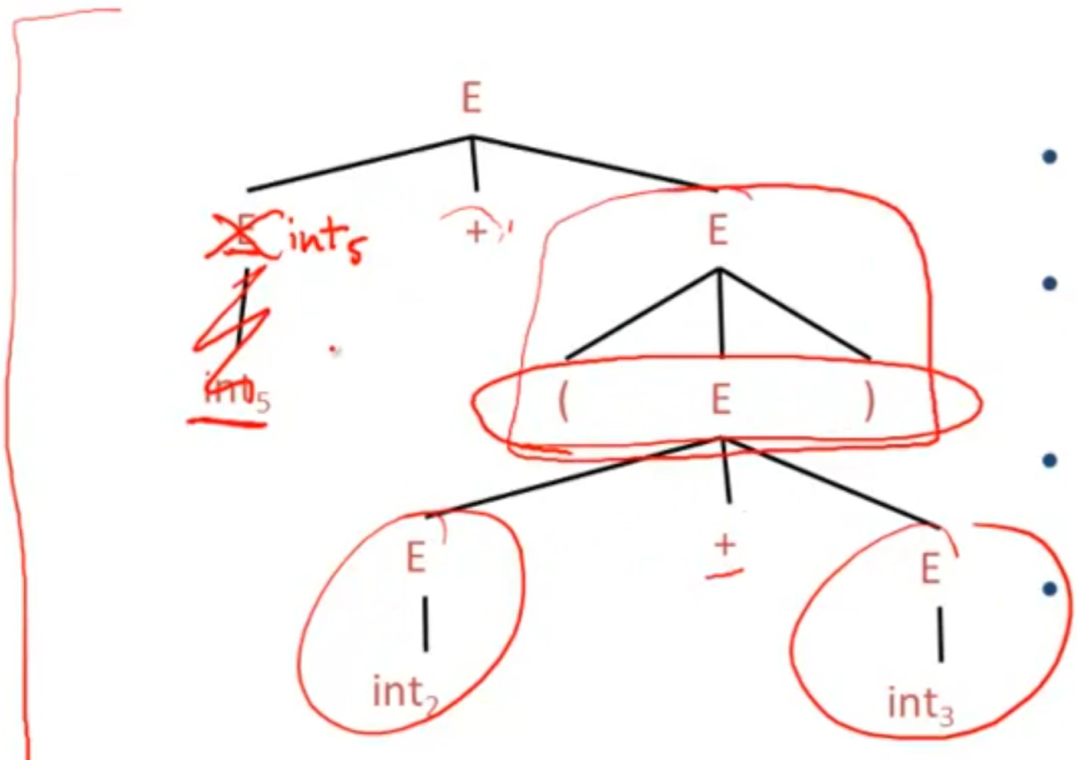
Error Correction

Idea: find a correct "nearby" program
- Try token insertions and deletions
- Exhaustive search

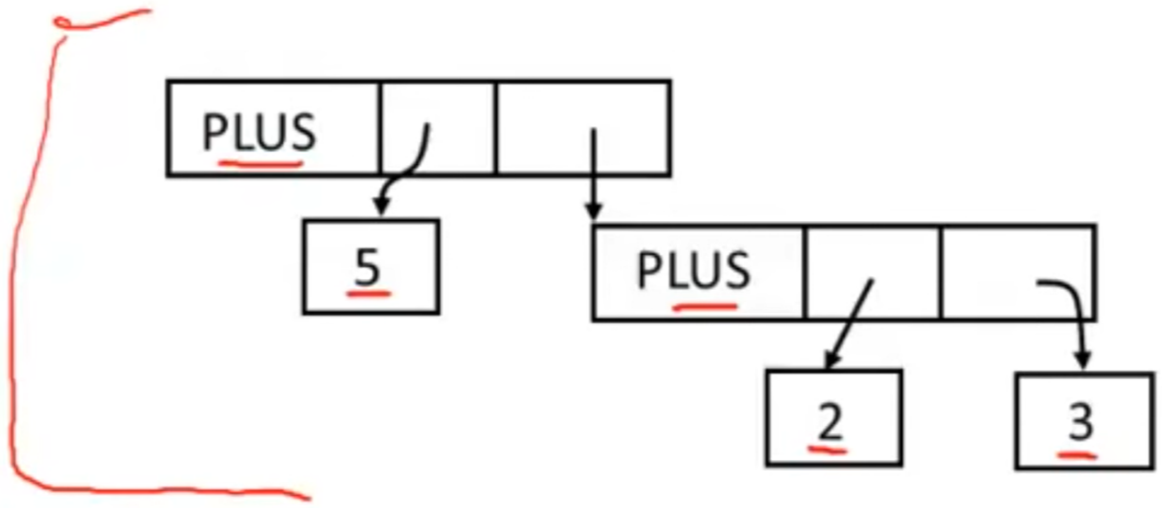
Disadvantages:
- Hard to implement
- Slows down parsing of correct programs
- "Nearby" is not necessarily "the intended" program

Abstract Syntax Trees

AST like parse trees but ignore some details.



- A parse tree:
- Traces the operation of the parser
- Captures nesting structure
- But too much information
  - Parentheses
  - Single-successor nodes



- Also captures the nesting structure
- But abstracts from the concrete syntax
  => more compact and easier to use
- An important data structure in a compiler

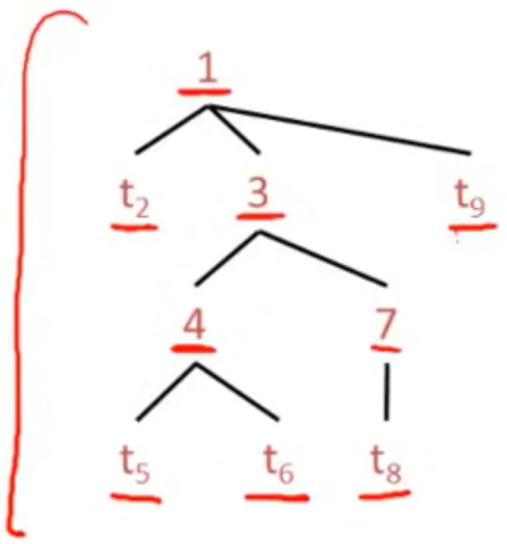
Recursive Descent Parsing

top-down

Recursive Descent

- The parse tree is constructed
  - From the top
  - From left to right
- Terminals are seen in order of appearance in the token stream:

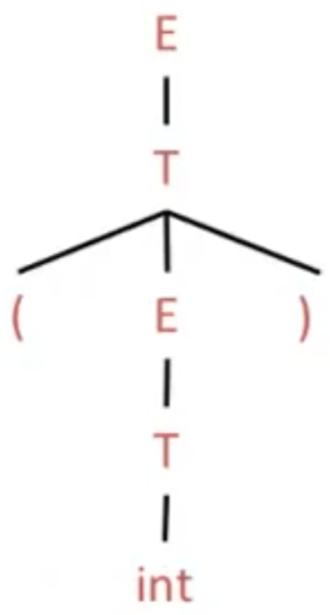
  t_2 t_5 t_6 t_8 t_9



$E \rightarrow T \mid T + E$

$T \rightarrow int \mid int * T \mid (E)$

(int_5)



End of input, accept.

Recursive Descent 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

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

    true/false

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

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

    → bool S() { ... }
- For production E → T

  bool E_1() { return T(); }
- For production E → 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()); }
```

- Functions for non-terminal T

```
bool T1() { return term(INT); }
bool T2() { return term(INT) && term(TIMES) && T(); }
bool T3() { return term(OPEN) && E() && term(CLOSE); }
```

T → int

T → int * T

T → (E)

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

To start the parser:

- Initialize next to point to first token
- Invoke E()

Easy to implement by hand.

$E \rightarrow T \mid T + E$

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


```
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()); }
```

Alex Aiken

$E \rightarrow T \mid T + E$
$T \rightarrow int \mid int * T \mid (E)$

```
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()); }
```

int * int rejected

What happened?!

If a production for non-terminal X succeeds
- Cannot backtrack to try a different production for X later.

General recursive-descent algorithms support such "full" backtracking
- Can implement any grammar.

Presented recursive-descent algorithm is not general
- But is easy to implement by hand

Sufficient for grammars where for any non-terminal at most one production can succeed

The example grammar can be written to work with the presented algorithm: By left factoring

Left Recursion

- Consider a production $S \rightarrow S\ a$

```
bool S1() { return S() && term(a); }
bool S() { return S1(); }
```

$S \rightarrow Sa \rightarrow Saa \rightarrow Saaa \rightarrow \cdots \rightarrow Sa\cdots a \rightarrow$

- S() goes into an infinite loop

- A left-recursive grammar has a non-terminal S

  $S \rightarrow^{+} S\alpha$ for some α

- Recursive descent does not work in such cases

- Consider the left-recursive grammar

  $S \rightarrow S\alpha \mid \beta$

  $S \rightarrow S\alpha \rightarrow S\alpha\alpha \rightarrow S\alpha\alpha\alpha \rightarrow \cdots \rightarrow S\alpha\cdots\alpha \rightarrow \beta\alpha\cdots\alpha$

- S generates all strings starting with a β and followed by any number of α's

- Can rewrite using right-recursion

  $S \rightarrow \beta S'$

  $S' \rightarrow \alpha S' \mid \varepsilon$

  $S \rightarrow \beta S' \rightarrow \beta\alpha S' \rightarrow \beta\alpha\alpha S' \rightarrow \cdots \rightarrow \beta\alpha\cdots\alpha S' \rightarrow \beta\alpha\cdots\alpha$

- In general

$$S \rightarrow S\,\alpha_1 \mid \ldots \mid S\,\alpha_n \mid \beta_1 \mid \ldots \mid \beta_m$$

- All strings derived from S start with one of $\beta_1,\ldots,\beta_m$ and continue with several instances of $\alpha_1,\ldots,\alpha_n$
- Rewrite as

$$\begin{aligned} S &\rightarrow \beta_1\,S' \mid \ldots \mid \beta_m\,S' \\ S' &\rightarrow \alpha_1\,S' \mid \ldots \mid \alpha_n\,S' \mid \varepsilon \end{aligned}$$

- The grammar

$$S \rightarrow A\,\alpha \mid \delta$$
$$A \rightarrow S\,\beta$$

$S \rightarrow A\alpha \rightarrow S\beta\alpha$

is also left-recursive because

$$S \rightarrow^{+} S\,\beta\,\alpha$$

- This left-recursion can also be eliminated
- See Dragon Book for general algorithm
- Recursive descent
  - Simple and general parsing strategy
  - Left-recursion must be eliminated first
  - … but that can be done automatically
- Used in production compilers
  - E.g., gcc