Compilers

Error Handling
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)**

<table>
<thead>
<tr>
<th>Error kind</th>
<th>Example</th>
<th>Detected by ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lexical</td>
<td>...$ ...</td>
<td>Lexer</td>
</tr>
<tr>
<td>Syntax</td>
<td>..x *% ...</td>
<td>Parser</td>
</tr>
<tr>
<td>Semantic</td>
<td>...int x; y = x(3); ...</td>
<td>Type checker</td>
</tr>
<tr>
<td>Correctness</td>
<td>your favorite program</td>
<td>Tester/User</td>
</tr>
</tbody>
</table>
• Error handler should
  - Report errors accurately and clearly
  - Recover from an error quickly
  - Not slow down compilation of valid code

• Good error handling is not easy to achieve
• Approaches from simple to complex
  - Panic mode
  - Error productions
  - Automatic local or global correction

• Not all are supported by all parser generators
• 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

• Such tokens are called 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 \texttt{error} to describe how much input to skip

\[E \rightarrow \text{int} \mid E + E \mid (E) \mid \text{error int} \mid (\text{error})\]
Error Handling

- **Error productions**
  - Specify known common mistakes in the grammar

- **Essentially promotes common errors to alternative syntax**

- **Example:**
  - Write $5 \times$ instead of $5 * x$
  - Add the production $E \rightarrow ... \mid EE$

- **Disadvantage**
  - Complicates the grammar
Idea: find a correct “nearby” program
- Try token insertions and deletions (minimum edit distance)
- Exhaustive search

Disadvantages:
- Hard to implement
- Slows down parsing of correct programs
- “Nearby” is not necessarily “the intended” program
• Past
  - Slow recompilation cycle (even once a day)
  - Find as many errors in one cycle as possible
  - Researchers could not let go of the topic

• Present
  - Quick recompilation cycle
  - Users tend to correct one error/cycle
  - Complex error recovery is less compelling
  - Panic-mode seems enough
Compilers

Abstract Syntax Trees

Syntax  CodeGen
Semantics  Types
• A parser traces the derivation of a sequence of tokens

• But the rest of the compiler needs a structural representation of the program

• **Abstract syntax trees**
  – Like parse trees but ignore some details
  – Abbreviated as AST
• Consider the grammar
  \[ E \rightarrow \text{int} \mid (E) \mid E + E \]

• And the string
  \[ 5 + (2 + 3) \]

• After lexical analysis (a list of tokens)
  \[ \text{int}_5 \, '+' \, ('\text{int}_2 \, '+' \, \text{int}_3 \, ')' \]

• During parsing we build a parse tree ...
Abstract Syntax Trees

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

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

Recursive Descent Parsing

Syntax CodeGen
Semantics  Types
• The parse tree is constructed
  - From the top
  - From left to right

• Terminals are seen in order of appearance in the token stream:
  \[ t_2 \ t_5 \ t_6 \ t_8 \ t_9 \]
• Consider the grammar

\[
E \rightarrow T | T + E \\
T \rightarrow \text{int} | \text{int} \ast T | (E)
\]

• Token stream is: \((\text{int}_5)\)

• Start with top-level non-terminal \(E\)
  - Try the rules for \(E\) in order
E → T | T + E
T → int | int * T | (E)

( int₅ )
↑
Recursive Descent

E → T | T + E
T → int | int * T | ( E)

( int₅)
Recursive Descent

E → T | T + E
T → int | int * T | ( E)

Mismatch: int does not match ( Backtrack ...
Recursive Descent

E → T | T + E
T → int | int * T | ( E )

( int₅ )
Recursive Descent

\[ E \rightarrow T \mid T + E \]
\[ T \rightarrow \text{int} \mid \text{int} \ast T \mid (E) \]

Mismatch: int does not match (E)

(E \rightarrow T \mid T + E)

(Mismatch: int does not match (E))
Recursive Descent

\[ E \rightarrow T \mid T + E \]
\[ T \rightarrow \text{int} \mid \text{int} \ast T \mid (E) \]
Recursive Descent

\[
E \rightarrow T \mid T + E \\
T \rightarrow \text{int} \mid \text{int} \ast T \mid (E)
\]

Match! Advance input.
E → T | T + E
T → int | int * T | (E)

Recursive Descent

E
  └── T

  └── (E)

  └── int

  └── 5
Recursive Descent

\[ E \rightarrow T \mid T + E \]
\[ T \rightarrow \text{int} \mid \text{int} \ast T \mid (E) \]

Diagram:

```
     E
    / |
   /  |
  T   E
     /  /
    /   /
   (    )
     /   /
    /     
   T  E  T
```

(int₅)
Recursive Descent

\[ E \rightarrow T \mid T + E \]
\[ T \rightarrow \text{int} \mid \text{int} \ast T \mid (E) \]

Match! Advance input.
Recursive Descent

E → T | T + E
T → int | int * T | (E)

( int₅ )

Match! Advance input.

Alex Aiken
E → T | T + E
T → int | int * T | (E)

Recursive Descent

End of input, accept.
Choose the derivation that is a valid recursive descent parse for the string \textit{id + id} in the given grammar. Moves that are followed by backtracking are given in red.

E  
E'  
E' + E  
id + E  
id + E'  
id + id  

E  
E'  
-E'  
(id)  
E' + E  
-E' + E  
id + E  
id + E'  
id + -E'  
id + id  

E  
E'  
E  
E'  
(id)  
E' + E  
id + E  
id + id  

Recursive Descent

\[
E \rightarrow E' \mid E' + E \\
E' \rightarrow -E' \mid \text{id} \mid (E)
\]
Choose the derivation that is a valid recursive descent parse for the string \texttt{id + id} in the given grammar. Moves that are followed by backtracking are given in red.

\[
\begin{align*}
E &\rightarrow E' | E' + E \\
E' &\rightarrow -E' | id | (E) \\
E &\rightarrow E + E \\
\end{align*}
\]
Compilers

Recursive Descent Algorithm
• Let $\text{TOKEN}$ be the type of tokens
  – Special tokens \text{INT, OPEN, CLOSE, PLUS, TIMES}

• Let the global \text{next} point to the next input token
• Define boolean functions that check the token string for a match of:
  - A given token terminal
    ```
    bool term(TOKEN tok) { return *next++ == tok; }
    ```
  - The nth production of $S$:
    ```
    bool $S_n() \{ \ldots \} $
    ```
  - Try all productions of $S$:
    ```
    bool $S() \{ \ldots \} $
    ```
For production $E \rightarrow T$

```cpp
bool E_1() { return T(); }
```

For production $E \rightarrow T + E$

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

For all productions of $E$ (with backtracking)

```cpp
bool E() {
    TOKEN *save = next;
    return (next = save, E_1()) || (next = save, E_2());
}
```
• Functions for non-terminal T

\[ T \rightarrow \text{int} \mid \text{int} \ast T \mid (E) \]

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, \( T_1() \))
    || (next = save, \( T_2() \))
    || (next = save, \( T_3() \));
}
• To start the parser
  - Initialize next to point to first token
  - Invoke E()

• Easy to implement by hand
\[
E \rightarrow T \mid T + E \\
T \rightarrow \text{int} \mid \text{int} \ast T \mid (E)
\]

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()); }
Which lines are incorrect in the recursive descent implementation of this grammar?

E → E' | E' + id
E' → -E' | id | (E)

Line 3
Line 5
Line 6
Line 12

```c
1 bool term(TOKEN tok) { return *next++ == tok; }
2 bool E1() { return E1(); }
3 bool E2() { return E1() && term(PLUS) && term(ID); }
4 bool E() {
5     TOKEN*save = next;
6     return (next = save, E1()) && (next = save, E2());
7 }
8 bool E1() { return term(MINUS) && E1(); }
9 bool E2() { return term(ID); }
10 bool E3() { return term(OPEN) && E() && term(CLOSE); }
11 bool E() {
12     TOKEN*next = save; return (next = save, E1()) || (next = save, E2()) || (next = save, E3());
13 }
```
Which lines are incorrect in the recursive descent implementation of this grammar?

E → E' | E' + id
E' → -E' | id | (E)

Line 3
Line 5
Line 6
Line 12
Compilers

Recursive Descent Algorithm Limitations
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(OPN) && E() && term(CLOSE); }

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

E → T | T + E
T → int | int * T | ( E )

RDA Limitations

int * int

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

- Completely general recursive-descent
  - algorithms support such “full” backtracking
Presented recursive descent algorithm is not general
  – But is easy to implement by hand

Is sufficient for grammars where for any non-terminal at most one production can match

The example grammar can be rewritten to work with the presented algorithm
  – By *left factoring*
• Which rule to choose when token is `if`:

\[
\text{EXPR} \rightarrow \text{if EXPRESSION then EXPRESSION} \text{ else EXPRESSION} \\
| \text{if EXPRESSION then EXPRESSION}
\]

• What’s the general form of `EXPR`?

\[
A \rightarrow \alpha \beta_1 \mid \alpha \beta_2 \\
\alpha : \text{if EXPRESSION then EXPRESSION} \\
\beta_1 : \text{else EXPRESSION} \\
\beta_2 : \varepsilon
\]

• Can be Transformed to:

\[
A \rightarrow \alpha A' \\
A' \rightarrow \beta_1 \mid \beta_2 \\
\text{EXPR} \rightarrow \text{if EXPRESSION then EXPRESSION REST} \\
\text{REST} \rightarrow \text{else EXPRESSION} \mid \varepsilon
\]

Left Factoring
Compilers

Syntax
CodeGen
Semantics
Types

Left Recursion

Alex Aiken
Left Recursion

- Consider a production $S \rightarrow S$ a
  
  ```c
  bool S_1() { return S() && term(a); }
  bool S() { return S_1(); }
  ```

- $S()$ goes into an infinite loop

- A *left-recursive* grammar has a non-terminal $S$
  
  $S \rightarrow^+ S\alpha$ for some $\alpha$

- Recursive descent does not work in such cases
• Consider the left-recursive grammar
  \[ S \rightarrow S \alpha | \beta \]

• \( S \) generates all strings starting with a \( \beta \) and followed by a number of \( \alpha \)

• Can rewrite using right-recursion
  \[ S \rightarrow \beta S' \]
  \[ S' \rightarrow \alpha S' | \varepsilon \]
Left Recursion

- 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
  
  \[
  S \rightarrow \beta_1 S' \mid \ldots \mid \beta_m S' \\
  S' \rightarrow \alpha_1 S' \mid \ldots \mid \alpha_n S' \mid \varepsilon
  \]
The grammar

\[ S \rightarrow A \alpha | \delta \]
\[ A \rightarrow S \beta \]

is also left-recursive because

\[ S \rightarrow^+ S \beta \alpha \]

This left-recursion can also be eliminated

See Dragon Book for general algorithm
- Section 4.3.3
Choose the grammar that correctly eliminates left recursion from the given grammar:

\[
\begin{align*}
E & \rightarrow E + T \mid T \\
T & \rightarrow \text{id} \mid (E)
\end{align*}
\]

- \( E \rightarrow E + \text{id} \mid E + (E) \mid \text{id} \mid (E) \)
- \( E' \rightarrow + TE' \mid \varepsilon \)
- \( T \rightarrow \text{id} \mid (E) \)
- \( E \rightarrow \text{id} + E \mid E + T \mid T \)
- \( T \rightarrow \text{id} \mid (E) \)
Choose the grammar that correctly eliminates left recursion from the given grammar:

\[
E \rightarrow E + T \mid T
\]

\[
T \rightarrow \text{id} \mid (E)
\]

- Option 1:
  
  \[
  E \rightarrow E + \text{id} \mid E + (E) \\
  \mid \text{id} \mid (E)
  \]

- Option 2:
  
  \[
  E \rightarrow E' + T \mid T
  \]

  \[
  E' \rightarrow \text{id} \mid (E)
  \]

- Option 3:
  
  \[
  E \rightarrow \text{id} + E \mid E + T \mid T
  \]

  \[
  T \rightarrow \text{id} \mid (E)
  \]
• **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