Error Handling
Syntax-Directed Translation
Recursive Descent Parsing

Lecture 6
Outline

- Extensions of CFG for parsing
  - Precedence declarations (previous slide set)
  - Error handling (slight digression)
    - I.e., what kind of error handling is available in parsers
  - Semantic actions

- Constructing a parse tree

- Recursive descent
Error Handling

• Purpose of the compiler is
  - To detect non-valid programs
    • And provide good feedback
  - 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>... $ ... (not used in C)</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 Handling

- **Purpose of the compiler is**
  - To detect non-valid programs
    - And provide good feedback
  - 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>... $ ... (not used in C)</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>
Syntax Error Handling

• Error handler should
  - Report errors accurately and clearly
    • Want to identify problem quickly and fix it
  - Recover from an error quickly
    • Compiler shouldn’t take a long time to figure out what to do when it hits an error
  - Not slow down compilation of valid code
    • I.e., don’t force good programs to pay the price for error handling

• Good error handling is not easy to achieve
Approaches to Syntax Error Recovery

• Three approaches, from simple to complex
  - Panic mode (used today)
  - Error productions (used today)
  - Automatic local or global correction
    • Idea that was pursued quite a bit in past
    • historically interesting contrast to what is done now

• Not all are supported by all parser generators
Error Recovery: Panic Mode

• 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**
  - Just tokens that have a well-known role in the language
  - Typically the statement or expression terminators
A Typical Strategy

Skip to the end of a statement or the end of a function if an error is found in a statement or function and then begin parsing either the next statement or the next function.
Syntax Error Recovery: Panic Mode (Cont.)

• Consider the erroneous expression
  
  
  \((1 \ \text{++} \ 2) + 3\)

• Panic-mode recovery:
  
  - Policy (for this particular kind of error) might be:
    Skip ahead to next integer and then continue
• Bison: use the special terminal error to describe how much input to skip

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

Blue are “normal” options.
Red are error options

Parser is attempting to parse something (haven’t seen how that works yet) and reaches a state where it expects an int, or a + or a parenthesized expression but if that isn’t working out and it gets stuck, it hits panic button: throw out everything up to the next int error matches all input up to next integer
• Bison: use the special terminal `error` to describe how much input to skip

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

Blue are “normal” options.
Red are error options

Similarly, if it encounters an error somewhere inside a pair of matched parentheses, just throw away the whole thing and continue parsing after the closing parenthesis.

Can have these productions that involve the `error` token for as many different kinds of errors as you like.
Syntax Error Recovery: Error Productions

- Idea: specify in the grammar known common mistakes

- Example:
  - Writing a compiler for language used by lots of mathematicians
  - They often write $5 \times$ instead of $5 * x$ and complain that this generates parse errors
    - Which state that the former is not a well-formed expression
  - Solution: Add the production $E \rightarrow \ldots | E \ E$
    - This makes the expression well formed
Syntax Error Recovery: Error Productions

• Essentially promotes common errors to alternative syntax

• Disadvantage
  - Complicates the grammar
  - If it’s used a lot grammar is going to be a lot harder to understand

• But it is used in practice!
  - E.g. gcc and other production C compilers will often warn you about things you’re not supposed to do but they’ll accept them anyway
  - Error productions is usually the mechanism by which this is done
• Previous mechanisms are primarily for detection. Following method actually tries to do correction!
Error Recovery: Local and Global Correction

- Idea: find a correct “nearby” program
  - I.e., programs that aren't too “different” from the original program
  - Try token insertions and deletions
    - E.g., Minimize the edit distance from bad token to newly inserted token
  - Exhaustive search (within some specified bounds)
Error Recovery: Local and Global Correction

• Disadvantages:
  - Hard to implement
    • It’s actually quite complex
  - Slows down parsing of correct programs
    • Because you need to keep enough state around to manage the search or the editing
  - “Nearby” is not necessarily “the intended” program
    • Not really all that clear what “nearby” means
    • “Nearby” is not necessarily “the intended” program
    • Not all tools support it
Error Recovery: Local and Global Correction

• Best example: The PL/C compiler
  - PL: Because it’s a PL1 compiler
  - C: Either “correction” or “Cornell” (where the compiler was built)
  - Well known for being willing to compile absolutely anything
    • Phone book
    • Speech from a Hamlet soliloquy
    • It would give lots of error messages
      - Many quite funny
    • But in the end it always produced a valid working PL1 program

• But, why bother?
Syntax Error Recovery: Past and Present

• When this was done (in the 1970s)
  - Slow recompilation cycle (even once a day)
    • Submit program in morning, get compiler output in the afternoon
    • With that kind of turnaround, even a single syntax error could be devastating: could lose a whole day just because of typo in a keyword
    • So having a compiler that can correct the program for you if it’s a small error could save you a whole day
  - So want to find as many errors in one cycle as possible
    • And then check whether the corrections were right
    • Allow even more debugging before next round
  - Researchers could not let go of the topic
Syntax Error Recovery: Past and Present

• Present
  - Quick recompilation cycle
  - Users tend to correct one error/cycle
    • Usually the first error, since that tends to be the most reliable report from the compiler (and it needs to be fixed before others can be fixed)
  - Complex error recovery is less compelling than it was a few decades ago
  - Panic-mode seems enough
Abstract Syntax Trees

• So far a parser traces the derivation of a sequence of tokens
  - Not all that useful to the compiler because...

• The rest of the compiler needs a structural representation of the program
  - Data structure that tells it what the operations are in the program and how they’re put together

• So for various reasons a parse tree isn’t what we want to work on
Abstract Syntax Trees

- **Abstract syntax trees**
  - Like parse trees but ignore some details
    - Abstracted away some of the details
  - Abbreviated as AST
  - The core data structure used in compilers

- To be clear: We could do compilation perfectly well using a parse tree.
  - Since it is a faithful representation of program structure
  - Just inconvenient to use because of unnecessary details
Abstract Syntax Tree. (Cont.)

• 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 ...
Example of Parse Tree

- Traces the operation of the parser
- Does capture the nesting structure
- But too much info
  - Parentheses
  - Single-successor nodes
Example of Parse Tree

- Traces the operation of the parser
- Does capture the nesting structure
- But too much info
  - Parentheses
  - Single-successor nodes

AST compresses out all the “junk” in the parse tree
Example of Abstract Syntax Tree

- Also captures the nesting structure
  - via which plus node nested inside the other
- But **abstracts** from the concrete syntax
  => more compact and easier to use
- An important data structure in a compiler
Example of Abstract Syntax Tree

• Consider designing algorithms to traverse this as opposed to the parse tree
  - Quite a bit easier
Semantic Actions

• Semantic actions are program fragments embedded within production bodies
  - This is what we’ll use to construct ASTs
  - These actions become part of the AST

• Ex: \[ E \rightarrow E + T \ {\text{print}}('+')) \]
Semantic Actions

- **Ex:** \[ E \rightarrow E + T \{\text{print('+')}\} \]

  - Interpretation of this particular production: When building the parse tree, any time you use this production, you end up with \( E \) having 4 children: \( E \), +, T, and \{\text{print('+')}\}

  - Once AST built, perform left to right depth-first traversal and execute each action when we encounter it’s leaf node
    - See example on p. 59 of text
Semantic Actions

• This is what we’ll use to construct ASTs
  - Semantic actions are program fragments embedded within production bodies

• Each grammar symbol may have attributes
  - For terminal symbols (lexical tokens) attributes can be calculated by the lexer

• Each production may have an action
  - Written as: $X \rightarrow Y_1 \ldots Y_n \{ \text{action} \}$
  - That can refer to or compute symbol attributes
  - Basically, you are associating rules or program fragments to productions in a grammar
Semantic Actions: An Example

• Consider the grammar
  \[ E \rightarrow \text{int} \mid E + E \mid (E) \]

• For each symbol \( X \) define an attribute \( X.\text{val} \)
  - For terminals, \( \text{val} \) is the associated lexeme
  - For non-terminals, \( \text{val} \) is the expression’s value (and is computed from values of subexpressions)

• We annotate the grammar with actions:
  \[
  E \rightarrow \text{int} \quad \{ \text{E.val} = \text{int.val} \} \\
  \mid E_1 + E_2 \quad \{ \text{E.val} = E_1.\text{val} + E_2.\text{val} \} \\
  \mid (E_1) \quad \{ \text{E.val} = E_1.\text{val} \}
  \]
Semantic Actions: An Example (Cont.)

- String: 5 + (2 + 3)
- Tokens: \texttt{int}_5 \texttt{ '+'} \texttt{ '(' \texttt{int}_2 \texttt{ '+'} \texttt{int}_3 \texttt{ ')}\texttt{')}

<table>
<thead>
<tr>
<th>Productions</th>
<th>Equations</th>
</tr>
</thead>
<tbody>
<tr>
<td>( E \rightarrow E_1 + E_2 )</td>
<td>( E\text{.val} = E_1\text{.val} + E_2\text{.val} )</td>
</tr>
<tr>
<td>( E_1 \rightarrow \text{int}_5 )</td>
<td>( E_1\text{.val} = \text{int}_5\text{.val} = 5 )</td>
</tr>
<tr>
<td>( E_2 \rightarrow (E_3) )</td>
<td>( E_2\text{.val} = E_3\text{.val} )</td>
</tr>
<tr>
<td>( E_3 \rightarrow E_4 + E_5 )</td>
<td>( E_3\text{.val} = E_4\text{.val} + E_5\text{.val} )</td>
</tr>
<tr>
<td>( E_4 \rightarrow \text{int}_2 )</td>
<td>( E_4\text{.val} = \text{int}_2\text{.val} = 2 )</td>
</tr>
<tr>
<td>( E_5 \rightarrow \text{int}_3 )</td>
<td>( E_5\text{.val} = \text{int}_3\text{.val} = 3 )</td>
</tr>
</tbody>
</table>
Semantic Actions: Notes

• Semantic actions specify a system of equations
  - Order of resolution is not specified

• Example:
  \[ E_3.\text{val} = E_4.\text{val} + E_5.\text{val} \]
  - Must compute \( E_4.\text{val} \) and \( E_5.\text{val} \) before \( E_3.\text{val} \)
  - We say that \( E_3.\text{val} \) depends on \( E_4.\text{val} \) and \( E_5.\text{val} \)

• The parser must find the order of evaluation
Each node labeled $E$ has one slot for the val attribute
Note the dependencies
Evaluating Attributes

• An attribute must be computed after all its successors in the dependency graph have been computed
  - In previous example attributes can be computed bottom-up

• Such an order exists when there are no cycles
  - Cyclically defined attributes are not legal
Dependency Graph

\[
E = (E_3 + E_4) + E_2 + E_1 + int_5 + int_2 + int_3
\]
Semantic Actions: Notes (Cont.)

- **Synthesized attributes**
  - Calculated from attributes of descendents in the parse tree
  - `E.val` is a synthesized attribute
  - Can always be calculated in a bottom-up order

- **Grammars with only synthesized attributes are called S-attributed grammars**
  - Most common case
Inherited Attributes

• Another kind of attribute

• Calculated from attributes of parent, and/or siblings, and/or self in the parse tree

• Example: a line calculator
A Line Calculator

• Each line contains an expression
  \[ E \rightarrow \text{int} \mid E + E \]

• Each line is terminated with the = sign
  \[ L \rightarrow E = \mid + E = \]

• In second form the value of previous line is used as starting value

• A program is a sequence of lines
  \[ P \rightarrow \varepsilon \mid P \mid L \]
Attributes for the Line Calculator

• Each $E$ has a synthesized attribute $\text{val}$
  - Calculated as before

• Each $L$ has an attribute $\text{val}$
  
  $L \rightarrow E = \{ L.\text{val} = E.\text{val} \}$
  
  | $+ E = \{ L.\text{val} = E.\text{val} + L.\text{prev} \}$

• We need the value of the previous line
• We use an inherited attribute $L.\text{prev}$
Attributes for the Line Calculator (Cont.)

- Each $P$ has a synthesized attribute $\text{val}$
  - The value of its last line
    $$P \rightarrow \varepsilon \quad \{ P\.val = 0 \}$$
    $$| \quad P_1 L \quad \{ P\.val = L\.val; \}$$
    $$\quad L\.prev = P_1\.val$$
  - Each $L$ has an inherited attribute $\text{prev}$
  - $L\.prev$ is inherited from sibling $P_1\.val$

- Example ...
Example of Inherited Attributes

- **val** synthesized
- **prev** inherited
- All can be computed in depth-first order
Example of Inherited Attributes

- val synthesized
- prev inherited
- All can be computed in depth-first order

\[
\begin{align*}
P &\quad : \quad 5 \\
0 &\quad + \quad E_3 &\quad = \quad -5 \\
0 &\quad + \quad E_4 &\quad + \quad E_5 &\quad = \quad 5 \\
\epsilon &\quad : \quad 0 \\
2 &\quad + \quad 3 &\quad = \quad 5 \\
\text{int}_2 &\quad 2 &\quad \text{int}_3 &\quad 3 \\
\end{align*}
\]
Semantic Actions: Notes (Cont.)

• Semantic actions can be used to build ASTs

• And many other things as well
  - Also used for type checking, code generation, ...

• Process is called syntax-directed translation
  - Substantial generalization over CFGs
Constructing An AST

• We first define the AST data type
  - Supplied by us for the project
• Consider an abstract tree type with two constructors:

\[
mkleaf(n) = \begin{array}{c}
\hline
n \\
\hline
\end{array}
\]

\[
mkplus(T_1, T_2) =
\]

\[
\begin{array}{c}
\hline
PLUS \\
\hline
\end{array}
\]

\[
\begin{array}{c}
\hline
T_1 \\
\hline
\end{array}
\quad
\begin{array}{c}
\hline
T_2 \\
\hline
\end{array}
\]

\[
\begin{array}{c}
\hline
\end{array}
\]

\[
\begin{array}{c}
\hline
\end{array}
\]

\[
\begin{array}{c}
\hline
\end{array}
\]

\[
\begin{array}{c}
\hline
\end{array}
\]

\[
\begin{array}{c}
\hline
\end{array}
\]
Constructing a Parse Tree

- We define a synthesized attribute $\text{ast}$
  - Values of $\text{ast}$ attribute are ASTs
  - We assume that $\text{int.lexval}$ is the value of the integer lexeme
  - Computed using semantic actions

\[
\begin{align*}
E & \rightarrow \text{int} \quad E.\text{ast} = \text{mkleaf}(\text{int.lexval}) \\
| \quad E_1 + E_2 & \quad E.\text{ast} = \text{mkplus}(E_1.\text{ast}, E_2.\text{ast}) \\
| \quad (E_1) & \quad E.\text{ast} = E_1.\text{ast}
\end{align*}
\]
Parse Tree Example

- Consider the string \texttt{int}_5 + (\texttt{int}_2 + \texttt{int}_3)
- A bottom-up evaluation of the \texttt{ast} attribute:

\[
\text{E.ast} = \text{mkplus} (\text{mkleaf}(5), \text{mkplus}(\text{mkleaf}(2), \text{mkleaf}(3)))
\]
Summary

- We can specify language syntax using CFG

- A parser will answer whether $s \in L(G)$
  - ... and will build a parse tree
  - ... which we convert to an AST
  - ... and pass on to the rest of the compiler
• Now, on to the actual parsing algorithms (there are a few)
Intro to Top-Down Parsing: The Idea

- The parse tree is constructed
  - From the top (i.e., starting with root node)
  - From left to right
- Terminals are seen in order of appearance in the token stream:
  \[ t_2 \quad t_5 \quad t_6 \quad t_8 \quad t_9 \]

Numbers correspond to order in which nodes constructed
Intro to Top-Down Parsing: The Idea

- The parse tree is constructed
  - From the top (i.e., starting with root node)
  - From left to right
- Terminals are seen in order of appearance in the token stream:
  \[ t_2 \ t_5 \ t_6 \ t_8 \ t_9 \]

And yes, there are also bottom-up parsing algorithms
Recursive Descent Parsing

• Consider the grammar (for int expressions)
  \[ E \rightarrow T \mid T + E \]
  \[ T \rightarrow \text{int} \mid \text{int} \ast T \mid (E) \]

• Token stream is: \((\text{int}_5)\)

• Start with top-level non-terminal \(E\)
  - Try the rules for \(E\) in order
  - There is trial-and-error involved

  And note I’m not giving pseudocode, but instead walk through the algorithm for this particular grammar
Recursive Descent: Three Components

• Grammar that we’re using

• Parse tree that we’re building
  - Initially just the root of the parse tree

• Input that we’re processing
  - Indicated by the red arrow
  - Always points to the next terminal symbol to be read
To Repeat

• Top to bottom
• Left to Right
• As we’re building the parse tree, whenever we get to a leaf (i.e., a terminal), we check whether it matches the current terminal in the input
  - If yes, then continue
  - If no, then backtrack
• But: while we are at a non-terminal, we have no way of knowing whether current track will succeed!
Recursive Descent Parsing

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

Highlighting indicates which production we're going to try:

\[ (\text{int}_5) \]
Recursive Descent Parsing

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

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

\[ ( \text{int}_5 ) \]
Recursive Descent Parsing

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

Mismatch: int is not "(")
Backtrack ...
Recursive Descent Parsing

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

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

Mismatch: int is not "(
Backtrack ...
Recursive Descent Parsing

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

( int\(_5\) )

\[ \uparrow \]
Recursive Descent Parsing

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

\[
( \text{int}_5 ) \\
\uparrow
\]

Match! Advance input.
Recursive Descent Parsing

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

Diagram:

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

\((\text{int}_5)\)
Recursive Descent Parsing

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

\[
(\text{int}_5)
\]
Recursive Descent Parsing

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

\[ (\text{int}_5) \]

\[ \text{Match! Advance input.} \]
Recursive Descent Parsing

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

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

(int 5)
```

Match! Advance input.
Recursive Descent Parsing

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

End of input, accept.
Now for the general algorithm
But first...
A Recursive Descent Parser. Preliminaries

• Let TOKEN be the type of tokens
  - Special tokens INT, OPEN, CLOSE, PLUS, TIMES
  - E.g., TOKEN is a type, and the list above are examples of instances of that type

• Let the global next point to the next token
  - I.e., next plays the same role as the red cursor arrow
A (Limited) Recursive Descent Parser (2)

• Define boolean functions that check the token string for a match of
  - A given token terminal
    
    bool term(TOKEN tok) { return *next++ == tok; } 
  - Read: Is what next is pointing to equal to tok?
    (That is, is tok equal to the thing that next is currently pointing to in the input stream?)
    • If so, return true
    • Note: as a side effect, next is incremented regardless of whether the match succeeded or failed!
A (Limited) Recursive Descent Parser (2)

- Define boolean functions that check the token string for a match of
  - The nth production of S: (note: a particular production of a particular nonterminal S)
    ```
    bool S_n() { ... }
    ```

- Again, returns a bool, and only checks for the success of only one production of S
- We’ll look at “code” for that in a minute
A (Limited) Recursive Descent Parser (2)

- Define boolean functions that check the token string for a match of
  - Try all productions of S:
    ```
    bool S() { ... }
    ```

  Succeeds if there is ANY production of S can match the input
A (Limited) Recursive Descent Parser (2)

• So, there are two classes of functions for each nonterminal

  - One class where there is one function per production, each function able to check whether the corresponding production matches the input

  - One class that combines all the productions for a particular nonterminal and checks whether any of them can match the input
Embedded *term()* calls

• Note that in both of the previously mentioned classes, there are embedded *term()* calls.

• Why?
  - The input attempting to be matched consists of terminals
  - So only way to know if a production for a non-terminal *CAN* match a portion of the input is to call the *term()* function (on some terminal(s) in the input)
  - Thus code of both $S_n$ and $S$ must call *term()*
Who Cares?

• You should: As mentioned term() has a side effect!
  - Moves the input pointer (whether or not there is a match)

• Bottom line: When either $S_n$ or $S$ are called, the input pointer has been incremented (via embedded term() calls) so that it is pointing at the first terminal that was NOT matched.

Now let's see this work
A (Limited) Recursive Descent Parser (3)

• For production $E \rightarrow T$
  
  ```
  bool $E_1()$ { return $T()$; }
  ```

• Why?
  - $E_1()$ is the function that deals with the first production for nonterminal $E$
  - It is supposed to return true if this first production can matches a given input
  - How can this production match an input?
    • Only if some production of $T$ matches the input
    • And we have a name for the function that tries all the productions of $T$. It’s called $T()$.
  - So, $E_1()$ succeeds exactly when $T()$ succeeds
A (Limited) Recursive Descent Parser (3)

- For production $E \rightarrow T + E$
  
  ```cpp
  bool E_2() { return T() && term(PLUS) && E(); }
  ```

- Little more work here: Succeeds if $T + E$ can match some input. How does this happen?
  - Some production of $T$ has to match a portion of the input AND
  - We have to find a $+$ in the input following whatever $T$ matched AND
  - If $+$ has been matched, some production of $E$ needs to match a portion of the input
• For production $E \rightarrow T + E$
  
  ```
  bool E_2() { return T() && term(PLUS) && E(); }
  ```

• Little more work here: Succeeds if $T + E$ can match some input. How does this happen?
  - Some production of $T$ has to match a portion of the input AND
  - We have to find a $+$ in the input following whatever $T$ matched AND
  - If $+$ has been matched, some production of $E$ needs to match a portion of the input

Note use of short circuiting && here
A (Limited) Recursive Descent Parser (3)

• For production $E \rightarrow T + E$

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

• Little more work here: Succeeds if $T + E$ can match some input. How does this happen?
  - Some production of $T$ has to match a portion of the input AND
  - We have to find a $+$ in the input following whatever $T$ matched AND
  - If $+$ has been matched, some production of $E$ needs to match a portion of the input

Note also how the side-effecting moves pointer
A (Limited) Recursive Descent Parser (3)

- For all productions of E (with backtracking)
  - Only state that we have to worry about is the `next` pointer,
  - Needs to be restored if we have to “undo” decisions

```c
bool E() {
    TOKEN *save = next;
    return (next = save, E_1()) || (next = save, E_2());
}
```

Note that if $E_1()$ matches, then the `next` pointer will have been advanced to point to the token following the portion matched by $E_1()$. 
A (Limited) Recursive Descent Parser (3)

- For all productions of $E$ (with backtracking)
  - Only state that we have to worry about is the $next$ pointer,
  - Needs to be restored if we have to “undo” decisions

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

Note saved $next$ ptr before any other “code”

Note restoring $next$ ptr before trying $E_2()$
A (Limited) Recursive Descent Parser (3)

• For all productions of E (with backtracking)
  - Only state that we have to worry about is the `next` pointer,
  - Needs to be restored if we have to “undo” decisions

```cpp
bool E() {
    TOKEN *save = next;
    return    (next = save, E_1())
    || (next = save, E_2());   }
```

But what about this saved ptr here? Not needed, but done for uniformity
Recall Our Grammar

\[ E \rightarrow T \mid T + E \]
\[ T \rightarrow \text{int} \mid \text{int} \times T \mid (E) \]
A (Limited) Recursive Descent Parser (4)

- Functions for non-terminal T

```c
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()); }
```
Recursive Descent Parsing. Notes.

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

• Notice how this simulates the example parse

• Easy to implement by hand (and people often do this)
  - But not completely general
  - Cannot backtrack once a production is successful
  - Works for grammars where at most one production can succeed for a non-terminal
Complete Example

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

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

bool E_{1}() { return T(); }
bool E_{2}() { return T() && term(PLUS) && E(); }

bool E() {TOKEN *save = next; return (next = save, E_{1}())
          || (next = save, E_{2}()); }

bool T_{1}() { return term(INT); }
bool T_{2}() { return term(INT) && term(TIMES) && T(); }
bool T_{3}() { return term(OP) && E() && term(CLOSE); }

bool T() { TOKEN *save = next; return (next = save, T_{1}())
          || (next = save, T_{2}())
          || (next = save, T_{3}()); }

85
Another Example (Same Grammar)

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

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

bool E_1() { return T(); }
bool E_2() { return T() && term(PLUS) && E(); }

bool E() { TOKEN *save = next; return (next = save, E_1()) || (next = save, E_2()); }

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()); }
```
Still Another Example (Same Grammar)

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

```cpp
int * int

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

bool E_1() { return T(); }
bool E_2() { return T() && term(PLUS) && E(); }

bool E() { TOKEN *save = next; return (next = save, E_1())
           || (next = save, E_2()); }

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()); }
```
Still Another Example (Same Grammar)

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

```cpp
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()); }
```

rejected! So what happened?
So What Happened?

- When \texttt{int} matched the first production for \texttt{T}, we said that \texttt{T()} was done, had matched the input.
- This caused the parse to fail because the rest of the string was not consumed.
- Once the call to \texttt{T()} had succeeded, there was no way to backtrack and try alternative production for \texttt{T}.
- To succeed, would have to be able to say that even though we found a part that matched, since the overall parse failed, that must not have been the right production to choose for \texttt{T}
  - Note here trying \texttt{T_2()} would have resulted in success
So, the Problem

- The problem is that while there is backtracking while trying to find a production that works for a given non-terminal, there is no backtracking once we have found a production that succeeds for that non-terminal.
- So once a non-terminal function commits and returns and says “I have found a way to parse part of the input using one of my productions”, there is no way in this algorithm to go back and revisit that decision.
Bottom Line

• The Recursive Descent algorithm we’ve seen thus far is not completely general
  - It fails on some inputs on which it should succeed
• BUT, recursive descent IS a general technique
  - There are algorithms for Recursive Descent parsing that can parse any grammar (implement full language of any grammar)
  - Such algorithms have more sophisticated backtracking mechanisms than does the algorithm I’ve presented thus far.
So Why Show This Algorithm?

- Well, as we’ve seen, it’s easy to implement by hand.
- Also, it works on a large class of grammars
  - Any grammar where for any non-terminal at most one production can succeed.
- So, if you know by the way that you’ve built your grammar that the RD algorithm is only in situations where for any terminal at most one production can succeed, you’re good!
- The example grammar can be rewritten to work with this algorithm via *left-factoring*
Recursive Descent: Another Issue

• Consider a simple grammar with a single production $S \rightarrow S \ a$
  
  bool $S_1()$ { return $S()$ && term(a); } 
  bool $S()$ { return $S_1()$; }

• $S()$ always 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
A Major Problem?

• Well, though it might seem so at first, the answer is that no, it’s not.
Elimination of Left Recursion

• Consider the left-recursive grammar

\[ S \rightarrow S \alpha \mid \beta \]

• \( S \) generates all strings starting with a \( \beta \) and followed by a number of \( \alpha \)
  - Note that it produces strings right to left
  - Very last thing it produces is first thing in input

• This is why it causes issues for recursive descent parsing
  - Which wants to process first part of the string first (left to right)
Elimination of Left Recursion

• Consider the left-recursive grammar
  \[ S \rightarrow S \alpha | \beta \]

• This gives us the idea on how to fix it: replace left-recursion with right-recursion
  - Create exactly same language

• Can rewrite using right-recursion
  \[ S \rightarrow \beta S' \]
  \[ S' \rightarrow \alpha S' | \varepsilon \]
More Elimination of 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

\[ S \rightarrow \beta_1 \ S' \mid \ldots \mid \beta_m \ S' \]
\[ S' \rightarrow \alpha_1 \ S' \mid \ldots \mid \alpha_n \ S' \mid \varepsilon \]
General Left Recursion

• 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
  - In fact, automatically - does not require human intervention

• See Dragon Book for general algorithm
  - Section 4.3
Summary of Recursive Descent

• Simple and general parsing strategy
  - Left-recursion must be eliminated first
  - ... but that can be done automatically

• Unpopular because of backtracking
  - Thought to be too inefficient

• In practice, backtracking is eliminated by restricting the grammar