Error Handling
Syntax-Directed Translation
Recursive Descent Parsing

CS143
Lecture 6

Instructor: Fredrik Kjolstad
Slide design by Prof. Alex Aiken, with modifications
Announcements

- PA1 & WA1
  - Due today at midnight

- PA2 & WA2
  - Assigned today
Outline

• Extensions of CFG for parsing
  – Precedence declarations
  – Error handling
  – Semantic actions

• Constructing an abstract syntax tree (AST)

• Recursive descent parsing
Error Handling

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

• Many kinds of possible errors

<table>
<thead>
<tr>
<th>Error kind</th>
<th>Example (C)</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>
Syntax Error Handling

• 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
Syntax Error Recovery

• Approaches from simple to complex
  – Panic mode
  – Error productions
  – Automatic local or global correction

• 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
  – Typically the statement or expression terminators
Error Recovery: Panic Mode (Cont.)

• 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

\[
E \rightarrow \text{int} \mid E + E \mid ( E ) \mid \text{error} \mid \text{int} \mid ( \text{error} )
\]
Error Recovery: Error Productions

• Idea: specify in the grammar known common mistakes

• Essentially promotes common errors to alternative syntax

• Example:
  – Write $5 \times x$ instead of $5 \times x$
  – Add the production $E \rightarrow \ldots \mid E \ E$

• Disadvantage
  – Complicates the grammar
Error Recovery: Local and Global 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
  – Not supported by most tools
Syntax Error Recovery: Past and Present

• 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
Abstract Syntax Trees

• So far a parser traces the derivation of a sequence of tokens

• 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
Abstract Syntax Trees (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)
  \[ \text{int}_5 \text{+} \text{int}_2 \text{+} \text{int}_3 \]

• 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 Abstract Syntax Tree

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

• This is what we’ll use to construct ASTs

• 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
Semantic Actions: Example

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

• For each symbol \( X \) define an attribute \( X.val \)
  – 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:
  \[
  \begin{align*}
  E &\rightarrow \text{int} \quad \{ \text{E.val} = \text{int.val} \} \\
  &\mid E_1 + E_2 \quad \{ \text{E.val} = E_1.val + E_2.val \} \\
  &\mid ( E_1 ) \quad \{ \text{E.val} = E_1.val \}
  \end{align*}
  \]
Semantic Actions: Example (Cont.)

• String: $5 + (2 + 3)$
• Tokens: $\text{int}_5 \ ‘+’ \ (‘ \text{int}_2 \ ‘+’ \text{int}_3 \ ‘)$

<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

- **Declarative Style**
  - Order of resolution is not specified
  - The parser figures it out

- **Imperative Style**
  - The order of evaluation is fixed
  - Important if the actions manipulate global state
Semantic Actions: Notes

• We’ll explore actions as pure equations
  – But note bison has a fixed order of evaluation for actions

• 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 \( E_5.\text{val} \)
Dependency Graph

- Each node labeled E has one slot for the \textit{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

E

int_5

E_1

E_2

E_3

E_4

E_5

int_2

int_3

int_5

E

5

+ 10

5

E

(5)

E

5

2

E

2

+ 3

3

int

int

int

int

int

int

int

int

int
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
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, computation, …

• 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:

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

\[
\text{mkplus}(T_1, T_2) = \begin{array}{c}
\text{PLUS} \\
\end{array}
\]

\[
\begin{array}{c}
T_1 \\
\end{array} \quad \begin{array}{c}
T_2 \\
\end{array}
\]
Constructing an AST

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

$E \rightarrow \text{int}$  \quad E.\text{ast} = \text{mkleaf(}\text{int.lexval})$

$\mid E_1 + E_2$ \quad E.\text{ast} = \text{mkplus(}E_1.\text{ast}, \, E_2.\text{ast})$

$\mid ( \, E_1 \, )$ \quad E.\text{ast} = E_1.\text{ast}$
Abstract Syntax Tree Example

- Consider the string `int_5 ' + ' (' int_2 ' + ' int_3 ')'`
- A bottom-up evaluation of the `ast` attribute:
  
  \[
  E.ast = \text{mkplus}(\text{mkleaf}(5), \text{mkplus}(\text{mkleaf}(2), \text{mkleaf}(3)))
  \]
Summary

• We can specify language syntax using CFG

• A parser will answer whether $s \in L(G)$
  – … and will trace a parse tree
  – … in whose productions we build an AST
  – … that we pass on to the rest of the compiler
Intro to Top-Down Parsing: The Idea

- 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
```
Recursive Descent Parsing

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

• Token stream is: \((\text{int}_5)\)

• Start with top-level non-terminal \(E\)
  
  – Try the rules for \(E\) in order
Recursive Descent Parsing

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

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

\[(\text{int}_5)\]
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 …

( int₅ )
Recursive Descent Parsing

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

\[
E \\
| \\
T
\]

( \text{int}_5 )

↑
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

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

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

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

Match! Advance input.
Recursive Descent Parsing

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

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

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

E
   /\   /
T E
  /\ /
 T
 /\   /
 int

( int\(_5\) )

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

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

( int<sub>5</sub> )

[Diagram of syntax tree]

Match! Advance input.
Recursive Descent Parsing

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

End of input, accept.

( int\(_5\) )
A Recursive Descent Parser: Preliminaries

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

- Let the global next point to the next token
A (Limited) Recursive Descent Parser (2)

• Define boolean functions that check the token string for a match of
  – A given token terminal
    ```cpp
    bool term(TOKEN tok) { return *next++ == tok; }
    ```
  – The nth production of S:
    ```cpp
    bool S_n() { … }
    ```
  – Try all productions of S:
    ```cpp
    bool S() { … }
    ```
A (Limited) Recursive Descent Parser (3)

• 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, E_1())
            || (next = save, E_2());
  }`
A (Limited) Recursive Descent Parser (4)

• 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, $T_1()$
              || (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()`

- Easy to implement by hand
  - 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
Example

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

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(OPEN) && E() && term(CLOSE); }

bool T() { TOKEN *save = next; return (next = save, T_1()) || (next = save, T_2()) || (next = save, T_3()); }
When Recursive Descent Does Not Work

• Consider a production \( S \rightarrow S \ a \)
  
  \[
  \begin{align*}
  &\text{bool } S_1() \{ \text{return } S() && \text{term}(a); \} \\
  &\text{bool } S() \{ \text{return } S_1(); \}
  \end{align*}
  \]

• \( S() \) goes into an infinite loop

• A left-recursive grammar has a non-terminal \( S \)
  
  \[
  S \rightarrow^+ S\alpha \text{ for some } \alpha
  \]

• Recursive descent does not work in such cases
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 \( \alpha \)

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

• 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

• Historically unpopular because of backtracking
  – Was thought to be too inefficient
  – In practice, with some tweaks, fast and simple on modern machines

• Backtracking can be controlled by restricting the grammar