Error Handling
Syntax-Directed Translation
Recursive Descent Parsing

Lecture 6

Announcements

• PA1
  - Due today at midnight
  - README, test case
  - Your name(s)
• WA1
  - Due today at 5pm
• PA2
  - Assigned today
• WA2
  - Assigned Tuesday

Outline

• Extensions of CFG for parsing
  - Precedence declarations
  - Error handling
  - Semantic actions
• Constructing a parse tree
• Recursive descent

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>

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

Approaches to Syntax Error Recovery

• 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

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

Syntax Error Recovery: Error Productions

- Idea: specify in the grammar known common mistakes
- Essentially promotes common errors to alternative syntax
- Example:
  - Write `5 x` instead of `5 * x`
  - Add the production \[E \rightarrow \ldots \mid E + E\]
- Disadvantage:
  - Complicates the grammar

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 Tree. (Cont.)

- Consider the grammar
  \[ E \rightarrow \text{int} | (E) | E + E \]
- And the string
  \[ 5 + (2 + 3) \]
- After lexical analysis (a list of tokens)
  \[ \text{int}, '+', '(' \text{int}, '+', \text{int}, ')' \]
- 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

- 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: An Example

- Consider the grammar
  \[ E \rightarrow \text{int} | E + E | (E) \]
- 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:
  \[ E \rightarrow \text{int} \{ E\text{.val} = \text{int}\text{.val} \} \]
  \[ | E + E \{ E\text{.val} = E_1\text{.val} + E_2\text{.val} \} \]
  \[ | (E) \{ E\text{.val} = E\text{.val} \} \]

String: \[ 5 + (2 + 3) \]
Tokens: \[ \text{int}, '+', '(' \text{int}, '+', \text{int}, ')' \]

Semantic Actions: An Example (Cont.)

- Productions
  \[ E \rightarrow E_1 + E_2 \]
  \[ E_1 \rightarrow \text{int} \]
  \[ E_2 \rightarrow (E) \]
  \[ E_3 \rightarrow E_4 + E_5 \]
  \[ E_4 \rightarrow \text{int} \]
  \[ E_5 \rightarrow \text{int} \]

- Equations
  \[ E\text{.val} = E_1\text{.val} + E_2\text{.val} \]
  \[ E_1\text{.val} = \text{int}\text{.val} = 5 \]
  \[ E_2\text{.val} = E_3\text{.val} \]
  \[ E_3\text{.val} = E_4\text{.val} + E_5\text{.val} \]
  \[ E_4\text{.val} = \text{int}\text{.val} = 2 \]
  \[ E_5\text{.val} = \text{int}\text{.val} = 3 \]
Semantic Actions: Notes

- Semantic actions specify a system of equations
  - Order of resolution is not specified
- Example:
  \[ E_3.val = E_4.val + E_5.val \]
  - Must compute \( E_4.val \) and \( E_5.val \) before \( E_3.val \)
  - We say that \( E_3.val \) depends on \( E_4.val \) and \( E_5.val \)
- The parser must find the order of evaluation

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

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 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
  \[ P \rightarrow \epsilon \mid P \ L \]

Attributes for the Line Calculator

- Each E has a synthesized attribute \( \text{val} \)
  - Calculated as before
- Each L has an attribute \( \text{val} \)
  \[ L \rightarrow E = \quad ( L\text{val} = E\text{val}) \]
  \[ L \rightarrow + E = \quad ( L\text{val} = E\text{val} + L\text{prev}) \]
- 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
  \[ P \rightarrow \epsilon \quad ( P\text{val} = 0) \]
  \[ P \rightarrow P_1 L \quad ( P\text{val} = L\text{val}; L\text{prev} = P_1\text{val}) \]
  - Each L has an inherited attribute \( \text{prev} \)
  - \( L\text{prev} \) is inherited from sibling \( P_1\text{val} \)
- Example ...

Example of Inherited Attributes

- \( P \) synthesized
- \( L\text{prev} \) inherited
- All can be computed in depth-first order

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

Constructing a Parse Tree

• We define a synthesized attribute ast
  - Values of ast values are ASTs
  - We assume that int.lexval is the value of the integer lexeme
  - Computed using semantic actions

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

Parse Tree Example

• Consider the string \text{int}5\text{+}(\text{int}2\text{+int}3)\
• A bottom-up evaluation of the ast attribute:

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

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

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

\[
\begin{align*}
E & \rightarrow T \ | T + E \\
T & \rightarrow \text{int} \ | \text{int} \ast T \ | (E)
\end{align*}
\]
• Token stream is: (int5)
• Start with top-level non-terminal E
  - Try the rules for E in order
Recursive Descent Parsing

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

E

(\text{int}_5)

Prof. Aiken CS 143 Lecture 6

37

Recursive Descent Parsing

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

E \mid T

int

Mismatch: int is not (!
Backtrack ...

(\text{int}_5)

Prof. Aiken CS 143 Lecture 6

39

Recursive Descent Parsing

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

E \mid T

int

Mismatch: int is not (!
Backtrack ...

(\text{int}_5)

Prof. Aiken CS 143 Lecture 6

40

Recursive Descent Parsing

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

E \mid T

int

Mismatch: int is not (!
Backtrack ...

(\text{int}_5)

Prof. Aiken CS 143 Lecture 6

41

Recursive Descent Parsing

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

E \mid T

int

Mismatch: int is not (!
Backtrack ...

(\text{int}_5)

Prof. Aiken CS 143 Lecture 6

42
Recursive Descent Parsing

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

E
  | 
  T
( int )

Match! Advance input.

( int )

Prof. Aiken CS 143 Lecture 6 43

Recursive Descent Parsing

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

E
  | 
  T
( int )

Match! Advance input.

( int )

Prof. Aiken CS 143 Lecture 6 44

Recursive Descent Parsing

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

E
  | 
  T
( int )

( int )

Prof. Aiken CS 143 Lecture 6 45

Recursive Descent Parsing

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

E
  | 
  T
( int )

Match! Advance input.

( int )

Prof. Aiken CS 143 Lecture 6 46

Recursive Descent Parsing

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

E
  | 
  T
( int )

Match! Advance input.

( int )

Prof. Aiken CS 143 Lecture 6 47

Recursive Descent Parsing

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

E
  | 
  T
( int )

End of input, accept.

( int )

Prof. Aiken CS 143 Lecture 6 48
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 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; }
  - The nth production of S:
    bool S_n() { … }
  - Try all productions of S:
    bool S() { … }

A Recursive Descent Parser (3)

• 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, E_1())
             || (next = save, E_2());   }

A 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()

• Notice how this simulates the example parse

• Easy to implement by hand

Example

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

bool term(TOKEN tok) { return *next++ == tok; }
bool E_1() { return T(); }
bool E_2() { return T() && term(TIMES) && T(); }
bool E_3() { return term(OPEN) && E() && term(CLOSE); }
bool E() { return next = save, E_1();
           || (next = save, E_2());
           || (next = save, E_3());

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

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

- $S()$ goes into an infinite loop

- A left-recursive grammar has a non-terminal $S$ 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
  
  $S \rightarrow \beta S'$
  $S' \rightarrow \alpha S' \mid \epsilon$

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
  
  $S \rightarrow S \beta_1 \mid \ldots \mid S \beta_m$
  $S' \rightarrow S \alpha_1 \mid \ldots \mid S \alpha_n \mid \epsilon$

General Left Recursion

- The grammar
  
  $S \rightarrow A \alpha \mid \delta$
  $A \rightarrow S \beta$
  
  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

- Unpopular because of backtracking
  
  - Thought to be too inefficient

  - In practice, backtracking is eliminated by restricting the grammar