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
  \[(1 \times + 2) + 3\]
- 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 \(5 \times x\) 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 all tools support it

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

- And the string
  \[ 5 \times (2 \times 3) \]

- After lexical analysis (a list of tokens)
  \[ \text{int}_5 \times (\ \text{int}_2 \times \text{int}_3 ) \]

- During parsing we build a parse tree ...

Example of Parse Tree

- Traces the operation of the parser

  [Diagram of a parse tree with nodes labeled E, int, +, and other tokens]

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

- 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:
  \[ E \rightarrow \text{int} \{ E.\text{val} = \text{int}.\text{val} \} \]
  \[ \mid E \ E \{ E.\text{val} = E_1.\text{val} + E_2.\text{val} \} \]
  \[ \mid (\ E \ ) \{ E.\text{val} = E.\text{val} \} \]

Semantic Actions: An Example (Cont.)

- String: \[ 5 \times (2 \times 3) \]

- Tokens: \[ \text{int}_5 \times (\ \text{int}_2 \times \text{int}_3 ) \]

- Productions

  \[
  \begin{align*}
  E &\rightarrow E_1 + E_2 \\
  E_1 &\rightarrow \text{int}_5 \\
  E_2 &\rightarrow ( E_3 ) \\
  E_3 &\rightarrow E_4 \times E_5 \\
  E_4 &\rightarrow \text{int}_2 \\
  E_5 &\rightarrow \text{int}_3
  \end{align*}
  \]

- Equations

  \[
  \begin{align*}
  E.\text{val} &= E_1.\text{val} + E_2.\text{val} \\
  E_1.\text{val} &= \text{int}_5.\text{val} = 5 \\
  E_2.\text{val} &= E_3.\text{val} \\
  E_3.\text{val} &= E_4.\text{val} + E_5.\text{val} \\
  E_4.\text{val} &= \text{int}_2.\text{val} = 2 \\
  E_5.\text{val} &= \text{int}_3.\text{val} = 3
  \end{align*}
  \]
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 \( L.\text{prev} \)

Example of Inherited Attributes

- \( \text{val} \) synthesized
- \( \text{prev} \) 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:

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

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

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

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

\[
E \rightarrow E_1 + E_2 \quad \text{E.ast} = \text{mkplus(E_1.ast, E_2.ast)}
\]

\[
E \rightarrow (E_1) \quad \text{E.ast} = E_1.ast
\]

Parse Tree Example

• Consider the string \text{int5} \cdot \text{+} \cdot (\text{int2} \cdot \text{+} \cdot \text{int3})
• 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 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

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

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

• Token stream is: \( (\text{int}_5) \)

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

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

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

Mismatch: int is not (!
Backtrack …

Recursive Descent Parsing

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

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

Mismatch: int is not (!
Backtrack …

Recursive Descent Parsing

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

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

Mismatch: int is not (!
Backtrack …

Recursive Descent Parsing

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

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

Mismatch: int is not (!
Backtrack …
Recursive Descent Parsing

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

E

T

( int )

Match! Advance input.

Recursive Descent Parsing

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

E

T

( int )

Match! Advance input.

Recursive Descent Parsing

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

E

T

( int )

Recursive Descent Parsing

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

E

T

( int )

End of input, accept.
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 \( E \rightarrow T \)
  
  ```
  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 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

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

```
bool term(TOKEN tok) { return *next++ == tok; }
``` 
```
bool E() { return T(); }
``` 
```
bool E() { return T() && term(PLUS) && E(); }
``` 
```
bool E() { return T() && term(PLUS) && E(); }
```
**When Recursive Descent Does Not Work**

- Consider a production \( S \rightarrow S a \)
  ```
  bool S() { return S() && term(a); } 
  bool S() { return S1(); } 
  ```
- \( S() \) 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

**Elimination of Left Recursion**

- 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
  \( S \rightarrow \beta S' \)
  \( S' \rightarrow \alpha S' | \epsilon \)

**More Elimination of Left-Recursion**

- In general
  \( S \rightarrow S \alpha_1 | \ldots | S \alpha_n | \beta_1 | \ldots | \beta_m \)
- 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
  \( S \rightarrow \beta_1 S' | \ldots | \beta_m S' \)
  \( S' \rightarrow \alpha_1 S' | \ldots | \alpha_n S' | \epsilon \)

**General Left Recursion**

- The grammar
  \( S \rightarrow A \alpha | \delta \)
  \( A \rightarrow S \beta \)
  is also left-recursive because
  \( S \rightarrow \alpha S \beta \alpha \)
- 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