Error Handling
Syntax-Directed Translation
Recursive Descent Parsing

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

• Idea: specify in the grammar known common mistakes

• Essentially promotes common errors to alternative syntax

• Example:
  - Write $5 \times$ instead of $5 \ast \times$
  - 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
  \[ E \rightarrow \text{int} | (E) | E + E \]

• And the string
  \[ 5 + (2 + 3) \]

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

• 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
CFG: 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
  \[ E \to \text{int} \mid E + E \mid ( E ) \]

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

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

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

### Productions

- \( E \rightarrow E_1 + E_2 \)
- \( E_1 \rightarrow \text{int}_5 \)
- \( E_2 \rightarrow (E_3) \)
- \( E_3 \rightarrow E_4 + E_5 \)
- \( E_4 \rightarrow \text{int}_2 \)
- \( E_5 \rightarrow \text{int}_3 \)

### Equations

- \( E.\text{val} = E_1.\text{val} + E_2.\text{val} \)
- \( E_1.\text{val} = \text{int}_5.\text{val} = 5 \)
- \( E_2.\text{val} = E_3.\text{val} \)
- \( E_3.\text{val} = E_4.\text{val} + E_5.\text{val} \)
- \( E_4.\text{val} = \text{int}_2.\text{val} = 2 \)
- \( E_5.\text{val} = \text{int}_3.\text{val} = 3 \)
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**
  - **Style 1**
  - But note bison has a fixed order of evaluation for actions

• **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 \( E_5.val \)
Dependency Graph

- 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

$E_1 \quad 5 \quad + \quad E_2 \quad 5 \quad + \quad (E_3 \quad 5 \quad ) \quad 10$

$E_4 \quad 2 \quad + \quad E_5 \quad 3$

$int_2 \quad 2 \quad int_3 \quad 3 \quad int_5 \quad 5
Semantic Actions: Notes (Cont.)

- **Synthesized attributes**
  - Calculated from attributes of descendants 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
  \[ P \rightarrow \varepsilon \mid P L \]
Attributes for the Line Calculator

• Each \( E \) has a synthesized attribute \( \text{val} \)
  - Calculated as before
• Each \( L \) has an attribute \( \text{val} \)
  \[
  L \rightarrow E = \{ \text{L.val} = E\text{.val} \}
  \]
  \[
  | + E = \{ \text{L.val} = E\text{.val} + L\text{.prev} \}
  \]
• 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
    
    $P \rightarrow \varepsilon$ \hspace{1cm} \{ $P$.val = 0 \}

    $\mid P_1 L$ \hspace{1cm} \{ $L$.prev = $P_1$.val;
    \hspace{1cm} $P$.val = $L$.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 bottom-up order

\[ P \left\{ \begin{align*} e \rightarrow 0 \rightarrow E_3 \rightarrow + \rightarrow E_4 \rightarrow int_2 \rightarrow 2 \rightarrow + \rightarrow E_5 \rightarrow int_3 \rightarrow 3 \rightarrow + \rightarrow L \rightarrow + \rightarrow P \end{align*} \right\} \]
Example of Inherited Attributes

- **val synthesized**

- **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, 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:

\[
mkleaf(n) = n
\]

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

\[
\begin{array}{c}
\text{PLUS} \\
\downarrow \\
T_1
\end{array}
\]

\[
\begin{array}{c}
\text{PLUS} \\
\downarrow \\
T_2
\end{array}
\]
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

\[
\begin{align*}
E & \rightarrow \text{int} \quad E.\text{ast} = \text{mkleaf}(\text{int.lexval}) \\
| E_1 + E_2 & \quad E.\text{ast} = \text{mkplus}(E_1.\text{ast}, E_2.\text{ast}) \\
| (E_1) & \quad E.\text{ast} = E_1.\text{ast}
\end{align*}
\]
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
  - ... at whose non-terminals we construct 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:
  \[ t_2 \quad t_5 \quad t_6 \quad t_8 \quad t_9 \]
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: ( int_5 )

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

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

( int₅ )
Recursive Descent Parsing

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

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

\[ \text{E} \]
\[ | \text{T} \]
\[ | \text{int} \]

\text{Mismatch: int is not ( !}
\text{Backtrack ...}

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

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

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

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

\[ ( \text{int}_5 ) \rightarrow \text{match! advance input.} \]
Recursive Descent Parsing

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

Diagram:

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

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

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

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

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

Match! Advance input.
Recursive Descent Parsing

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

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 (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; }
    ```
  - The nth production of S:
    ```
    bool S_n() { ... }
    ```
  - Try all productions of S:
    ```
    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

```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
  - 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

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

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(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 $S \rightarrow S \alpha$
  
  ```
  bool $S_1()$ { return $S()$ && term($\alpha$); }
  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
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, fast and simple on modern machines

• In practice, backtracking is eliminated by restricting the grammar