Ambiguity and Errors
Syntax-Directed Translation

Outline
• Ambiguity (revisited)
• Extensions of CFG for parsing
  ➢ Precedence declarations
  ➢ Error handling
  ➢ Semantic actions
• Constructing a parse tree

Ambiguity Revisited
• Ambiguity common in programming languages.
  ➢ Arithmetic expressions
  ➢ IF-THEN-ELSE (The Dangling Else Problem)
• Consider the grammar
  \[ E \rightarrow \text{if} \ E \text{ then } E \]
  \[ | \text{if} \ E \text{ then } \text{else} \ E \]
  | OTHER
  ➢ This grammar is ambiguous.

The Dangling Else Example
• The expression
  \[ \text{if} \ E_1 \text{ then if } E_2 \text{ then } E_3 \text{ else } E_4 \]
  has two parse trees.
• Typically we want the second form.
The Dangling Else: A Fix

• else matches the closest unmatched then.
• We can describe this in the grammar for the same language as follows:

\[
E \rightarrow \text{MIF} /* \text{all then are matched} */ \\
| \text{UIF} /* \text{some then are unmatched} */ \\
\text{MIF} \rightarrow \text{if } E \text{ then MIF else MIF} \\
| \text{OTHER} \\
\text{UIF} \rightarrow \text{if } E \text{ then } E \\
| \text{if } E \text{ then MIF else UIF}
\]

The Dangling Else: Revisited

• The expression if \( E_1 \) then if \( E_2 \) then \( E_3 \) else \( E_4 \)

\[
\text{if } E_1 \text{ if } E_2 \ E_3 \ E_4
\]

• Not valid because the then expression is not a MIF

• A valid parse tree (for a UIF)

Handling Ambiguity

• No general techniques for dealing with ambiguity.
  ➢ Impossible to convert automatically an ambiguous grammar to an unambiguous one.
    ❑ Ambiguity checking is undecidable.
    ❑ Inherently ambiguous context-free languages exist.

• Instead of rewriting the grammar, use the more natural (ambiguous) grammar along with disambiguating declarations.
  ➢ Most parser generator tools allow precedence and associativity declarations to disambiguate grammars.

Associativity Declarations

• Consider the grammar

\[
E \rightarrow E + E | \text{int}
\]

• Ambiguous: two parse trees of \( \text{int} + \text{int} + \text{int} \)

\[
E + E + E \\
\]

• Left associativity declaration: %left +
Precedence Declarations

- Consider the grammar \( E \rightarrow E + E \mid E \ast E \mid \text{int} \)
- And the string \( \text{int + int} \ast \text{int} \)
- Precedence declarations: `%left +`
- Precedence declarations: `%left *`

Error Handling

Types of errors

- **Lexical**: An error that produces a wrong token
  - E.g., misspelling of an identifier, keyword or operator
- **Syntactic**: A program that does not satisfy the CFG of the language
  - E.g., expression with unbalanced parentheses, missing semicolon
- **Semantic**: An error that needs context sensitive information to identify
  - E.g., Operator applied to an incompatible operand, Accessing an undeclared variable
- **Logical**: Errors in the execution model
  - E.g., Infinitely recursive call, Accessing an array out of bounds, Dereferencing a null pointer

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 (from simple to complex)
  - Panic mode (most popular!)
  - Error productions
  - Automatic local or global correction
  - Not all are supported by all parser generators.
Error Recovery: Panic Mode

- **Idea:** When an error is detected:
  - Discard tokens until one with a clear role is found. Then continue.
  - Such tokens are called *synchronizing* tokens.
    - Typically the statement or expression terminators.
- Consider the erroneous expression
  \[(1 + + 2) + 3\]
- Panic-mode recovery:
  - Skip ahead to next integer and then continue.
- **Bison:** uses the special terminal *error* to describe how much input to skip.
  \[E \rightarrow \text{int} | \text{E + E} | (\text{E}) | \text{error int} | (\text{error})\]

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 \[E \rightarrow \ldots \text{E E}\]
- **Disadvantage**
  - Complicates the grammar

Error Recovery: Local and Global Correction

- **Idea:** Find a correct “nearby” program
  - Try token insertions and deletions.
  - Exhaustive search.
    - Cf. Use of FIRST and FOLLOW sets in recursive descent parsers.
- **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.
- **Present**
  - Quick recompilation cycle.
  - Users tend to correct one error/cycle.
  - Complex error recovery not needed.
  - Panic-mode seems enough.
Abstract Syntax Trees

ASTs

- So far a parser traces the derivation of a sequence of tokens. But the rest of the compiler needs a structural representation of the program.
- Abstract syntax tree (AST) is like parse tree but ignores some details.
- Consider the grammar: \( E \rightarrow \text{int} \mid (\ E\ ) \mid E + E \)
- and the string: \( 5 + (2 + 3) \)
- After lexical analysis, we have a list of tokens
  - \( \text{int}_5 \), \( '+' \), \( '(' \), \( \text{int}_2 \), \( '+' \), \( \text{int}_3 \), \( ')' \)
- which, during parsing is turned into 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 (no redundant delimiters => more compact and easier to use).
- AST is an important data structure in a compiler.
Semantic Attributes and Actions

- Will be used 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 numeric value.
  - For non-terminals, \( \text{val} \) is the expression’s value (which is computed from the values of the sub-expressions).
- We annotate the grammar with actions:
  \[
  E \rightarrow \text{int} \{\ E.\text{val} = \text{int} \} \\
  | E_1 + E_2 \{ E.\text{val} = E_1.\text{val} + E_2.\text{val} \} \\
  | ( E_1 ) \{ E.\text{val} = E_1.\text{val} \}
  \]

Semantic Actions: Notes

- Semantic actions specify a system of equations.
  - 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 \( E_3.\text{val} \).
    - That is, \( E_3.\text{val} \) depends on \( E_4.\text{val} \) and \( E_5.\text{val} \).
- The parser must find the order of evaluation for the attributes.
  - An attribute must be computed after all its successors in the dependency graph have been computed.
  - In the example, attributes can be computed bottom-up.
  - Cyclically defined attributes are not legal here.
  - However, in the compiler generator context, we can show how it can be generalized with iterative computation.
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 the starting value.
- A program is a sequence of lines:
  \[ P \rightarrow \epsilon \mid P \ L \]

Classification of Attributes

- **Synthesized attributes**
  ➢ Calculated from attributes of descendents in the parse tree. E.g., \( E.\text{val} \).
  ➢ Can always be calculated in a bottom-up order.
  ➢ A grammar with only synthesized attributes is called \( S \)-attributed grammar.

- **Inherited Attributes**
  ➢ Calculated from attributes of parent and/or siblings in the parse tree. E.g., a line calculator.

Attributes for the Line Calculator

- Each \( E \) has a synthesized attribute \( \text{val} \), as before.
- Each \( L \) has a synthesized attribute \( \text{val} \)
  \[ L \rightarrow E = \{ L.\text{val} = E.\text{val} \} \]
  \[ \mid + E = \{ L.\text{val} = E.\text{val} + L.\text{prev} \} \]
  ➢ Furthermore, the value of the previous line is obtained from the inherited attribute \( L.\text{prev} \).
- Each \( P \) has a synthesized attribute \( \text{val} \), the value of its last line.
  \[ P \rightarrow \epsilon \{ P.\text{val} = 0 \} \]
  \[ \mid P \ L \{ P.\text{val} = L.\text{val}; L.\text{prev} = P.\text{val} \} \]
  ➢ Each \( L \) has an inherited attribute \( \text{prev} \).
  ➢ \( L.\text{prev} \) is inherited from sibling \( P.\text{val} \).
Example of Inherited Attributes

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

Constructing an AST

- Semantic actions can be used to build ASTs (do type checking, code generation, etc).
- The process is called **syntax-directed translation**.
- We first define the AST data type.
- Consider an abstract tree type with two constructors:
  
  \[
  \text{mkleaf}(n) = \begin{cases} 
  n & \text{if } n \text{ is an integer} \\
  \text{PLUS} & \text{otherwise}
  \end{cases}
  \]

  \[
  \text{mkplus}(E1, E2) = \begin{cases} 
  \text{PLUS} & \text{if } E1, E2 \text{ are ASTs} \\
  \text{PLUS}(E1, E2) & \text{otherwise}
  \end{cases}
  \]

Constructing a Parse Tree

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

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

  \[
  | E1 + E2 \quad E.\ast = \text{mkplus}(E1, E2, E.\ast)
  \]

  \[
  | (E1) \quad E.\ast = E1.\ast
  \]

Parse Tree Example

- Consider the string \texttt{int5 ‘+’ ‘( ‘int2 ‘+’ int3 ‘) ’}’
- A bottom-up evaluation of the **ast** attribute:

  \[
  E.\ast = \text{mkplus(mkleaf(5), mkplus(mkleaf(2), mkleaf(3)))}
  \]
• Attributes \( A(X) \)
  ➢ Synthesized \( S(X) \)
  ➢ Inherited \( I(X) \)

• Attribute computation rules (Semantic functions)

\[
X_0 \rightarrow X_1 \ X_2 \ldots \ X_n
\]

\[
S(X_0) = f(I(X_0), A(X_1), A(X_2), \ldots, A(X_n))
\]

\[
I(X_j) = g_j(I(X_0), A(X_1), A(X_2), \ldots, A(X_{j-1}))
\]

for all \( j \) in \( 1..n \)

\[
P(A(X_0), A(X_1), A(X_2), \ldots, A(X_n))
\]

---

Information Flow

- Inherited
- Synthesized
- Computed
- Available

---

An Extended Example

Distinct identifiers in a straight-line program.

BNF

\[
<\text{exp}> ::= \text{<var>} \mid <\text{exp}> + <\text{exp}>
\]

\[
<\text{stm}> ::= \text{<var>} := <\text{exp}> \mid <\text{stm}> ; <\text{stm}>
\]

Attributes

\[
<\text{var}> \uparrow \text{id}
\]

\[
<\text{exp}> \uparrow \text{ids}
\]

\[
<\text{stm}> \uparrow \text{ids} \uparrow \text{num}
\]

• Semantics specified in terms of \textit{sets} (of identifiers).
Alternate approach: Using flex

Attributes

- $\downarrow \text{envi}$: list of vars in preceding context
- $\uparrow \text{envo}$: list of vars for following context
- $\uparrow \text{dnum}$: number of new variables

\[
\text{<exp> ::= <var>}
\]

\[
\text{<exp> ::= <exp1> + <exp2>}
\]

\[
\text{<exp>.ids = { <var>.id } U <exp>.ids}
\]

\[
\text{<stm> ::= <var> ::= <exp>}
\]

\[
\text{<stm>.ids = { <var>.id } U <exp>.ids}
\]

\[
\text{<stm>.num = | <stm>.ids } |
\]

\[
\text{<exp>.envo = if member(<var>.id,<exp>.envi) then <exp>.envi else cons(<var>.id,<exp>.envi)}
\]

Review

- We can specify language syntax using CFG.
- A parser will answer whether $s \in L(G)$
- … and will build a parse tree,
- … which is converted into an AST,
- … and passed on to the rest of the compiler.

- Next lectures:
  - How do we answer $s \in L(G)$ and build a parse tree?
  - After that: from AST to assembly language.