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 → if E then E
  | if E then E else E
  | OTHER

  - ➢This grammar is ambiguous.

The Dangling Else Example

- The expression

  if E_1 then if E_2 then E_3 else E_4

  has two parse trees.

- Typically we want the second form.

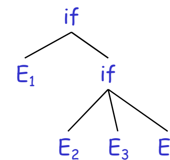
The Dangling Else: *A Fix*

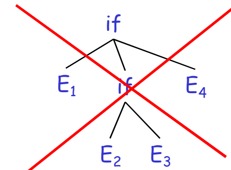
- else matches the closest unmatched then.
- We can describe this in the grammar for the same language as follows:

E → MIF /* all then are matched */
| UIF /* some then are unmatched */
MIF → if E then MIF else MIF
| OTHER
UIF → if E then E
| if E then MIF else UIF

The Dangling Else: Revisited

- The expression if E_1 then if E_2 then E_3 else E_4





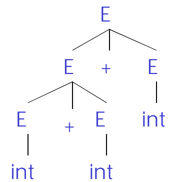
- A valid parse tree (for a UIF)
- Not valid because the then expression is not a MIF

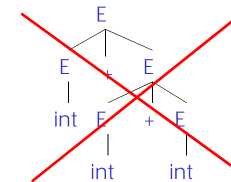
Handling Ambiguity

- 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 → E + E | int
- Ambiguous: two parse trees of int + int + int

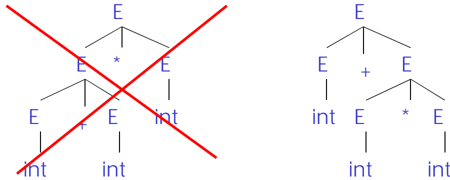




- Left associativity declaration: %left +

Precedence Declarations

- Consider the grammar E → E + E | E * E | int
  - ➢ And the string int + int * int



E
E + E
int E * E
int int

- Precedence declarations: %left +
  %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 → int | E + E | (E) | error int | (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 → … | 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

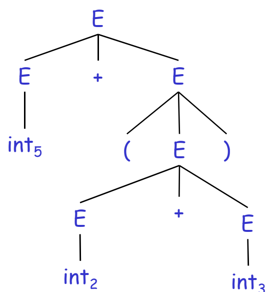
- So far a parser traces the derivation of a sequence of tokens. But the rest of the compiler needs a structural representation of the program.
- *Abstract syntax tree (AST)* is like parse tree but ignores some details.
- Consider the grammar: $E \rightarrow \text{int} \mid (E) \mid E + E$

  and the string: $5 + (2 + 3)$
- After lexical analysis, we have a list of tokens

  int_5 '+' '(' int_2 '+' int_3 ')'

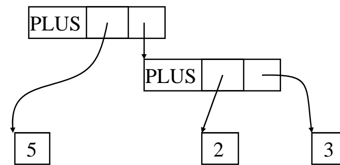
  which, during parsing is turned into a parse tree …

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: $X \rightarrow Y_1 \ldots Y_n$ { action }
  - That can refer to or compute symbol attributes.

Semantic Actions: An Example

- Consider the grammar
  $E \rightarrow int \mid E + E \mid (E)$
- For each symbol X, define an attribute X.val
  - For terminals, val is the associated numeric value.
  - For non-terminals, val is the expression's value (which is computed from the values of the sub-expressions).
- We annotate the grammar with actions:

$E \rightarrow int$ { E.val = int.val }
$\mid E_1 + E_2$ { $E.val = E_1.val + E_2.val$ }
$\mid (E_1)$ { $E.val = E_1.val$ }

(cont'd)

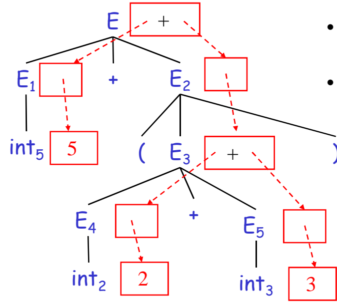
- String: 5 + (2 + 3)
- Tokens: int_5 '+' '(' int_2 '+' int_3 ')'

Productions	Equations
$E \rightarrow E_1 + E_2$	$E.val = E_1.val + E_2.val$
$E_1 \rightarrow int_5$	$E_1.val = int_5.val = 5$
$E_2 \rightarrow (E_3)$	$E_2.val = E_3.val$
$E_3 \rightarrow E_4 + E_5$	$E_3.val = E_4.val + E_5.val$
$E_4 \rightarrow int_2$	$E_4.val = int_2.val = 2$
$E_5 \rightarrow int_3$	$E_5.val = int_3.val = 3$

Semantic Actions: Notes

- Semantic actions specify a system of equations.
  - Example: $E_3.val = E_4.val + E_5.val$
    - Must compute $E_4.val$ and $E_5.val$ before $E_3.val$.
    - That is, $E_3.val$ depends on $E_4.val$ and $E_5.val$.
- 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.

Dependency Graph



- Each node labeled E has one slot for val attribute
- Note the dependencies.

Classification of Attributes

- *Synthesized* attributes
  - Calculated from attributes of descendents in the parse tree. E.g., E.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.

A Line Calculator

- Each line contains an expression:

  $E \rightarrow$ int | E + E
- Each line is terminated with the = sign:

  $L \rightarrow$ E = | + E =
  - In second form, the value of previous line is used as the starting value.
- A program is a sequence of lines:

  $P \rightarrow \varepsilon$ | P L

Attributes for the Line Calculator

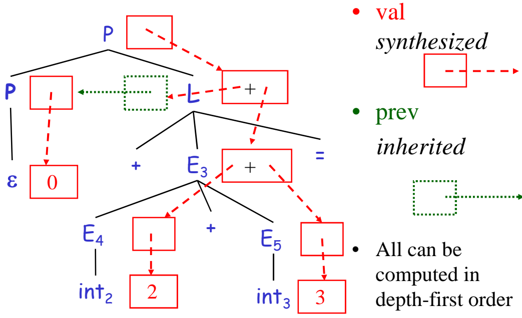
- Each E has a synthesized attribute val, as before.
- Each L has a synthesized attribute val

  $L \rightarrow$ E = { L.val = E.val }

  | + E = { L.val = E.val + L.prev }
  - Furthermore, the value of the previous *line* is obtained from the inherited attribute L.prev.
- Each P has a synthesized attribute val, the value of its *last line*. $P \rightarrow \varepsilon$ { P.val = 0 }

  | P_1 L { P.val = L.val; L.prev = P_1.val }
  - Each L has an inherited attribute prev.
  - L.prev is inherited from sibling P_1.val.

Example of Inherited Attributes

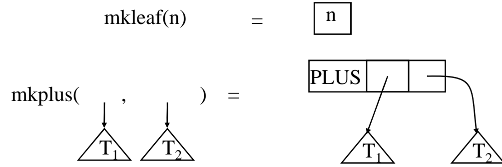


- val
  synthesized
- prev
  inherited
- All can be computed in depth-first order

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:

mkleaf(n) =



Constructing a Parse Tree

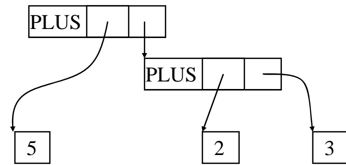
- We define a synthesized attribute ast
  - ➢ Values of ast are ASTs.
  - ➢ We assume that int.lexval is the value of the integer lexeme.
  - ➢ AST is computed using semantic actions.

$E \rightarrow int$	E.ast = mkleaf(int.lexval)
$\mid E_1 + E_2$	E.ast = mkplus(E_1.ast, E_2.ast)
$\mid (E_1)$	E.ast = E_1.ast

Parse Tree Example

- Consider the string int_5 '+' '(' int_2 '+' int_3 ')'
- A bottom-up evaluation of the ast attribute:

  E.ast = mkplus(mkleaf(5),
  mkplus(mkleaf(2), mkleaf(3))



- Attributes A(X)
  - Synthesized S(X)
  - Inherited I(X)
- Attribute computation rules (Semantic functions)

X0 -> X1 X2 … Xn

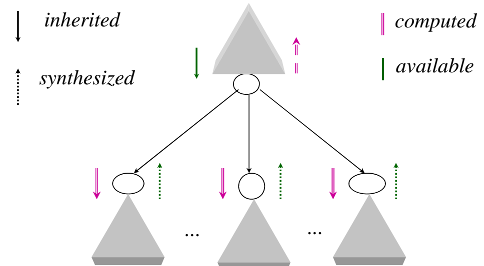
S(X0) = *f*(I(X0), A(X1), A(X2), …, A(Xn))

I(X*j*) = G*j*(I(X0), A(X1), A(X2), …, A(X*j-1*))

for all *j* *in* 1..n

P(A(X0), A(X1), A(X2), …, A(Xn))

Information Flow



- Synthesized Attributes

  Pass information *up* the parse tree
- Inherited Attributes

  Pass information *down* the parse tree *or sideways* from left siblings to right siblings
- Attribute values assumed to be available from the context.
- Attribute values computed using the semantic rules provided.

These constraints on the attribute evaluation rules permit top-down left-to-right (one-pass) traversal *of the parse tree to compute the meaning.*

An Extended Example

Distinct identifiers in a straight-line program.

BNF

```
<exp> ::=       <var>       | <exp> + <exp>
<stm> ::= <var> :=  <exp> | <stm> ; <stm>
```

Attributes

```
<var>        ↑ id
<exp>        ↑ ids
<stm>        ↑ ids   ↑ num
```

- Semantics specified in terms of *sets* (of identifiers).

```
   <exp>    ::=   <var>
<exp>.ids = { <var>.id }
   <exp>    ::=   <exp1> + <exp2>
<exp>.ids = <exp>.ids U <exp>.ids
   <stm>    ::=   <var> := <exp>
<stm>.ids ={ <var>.id } U <exp>.ids
<stm>.num = | <stm>.ids |
   <stm>    ::=   <stm1> ; <stm2>
<stm>.ids =  <stm1>.ids  U <stm2>.ids
<stm>.num = | <stm>.ids |
```


Alternate approach : Using *lists*

- Attributes
  - ↓ **envi** : list of vars in preceding context
  - ↑ **envo** : list of vars for following context
  - ↑ **dnum** : number of *new* variables

```
   <exp>    ::=   <var>
<exp>.envo =
     if member(<var>.id,<exp>.envi)
     then   <exp>.envi
     else   cons(<var>.id,<exp>.envi)
```


Attribute Computation Rules (pre-post conditions)

```
 <exp>    ::=    <exp1> + <exp2>
↓ envi           ↓ envi   ↓ envi
↑ envo           ↑ envo   ↑ envo
↑ dnum           ↑ dnum   ↑ dnum

  <exp1>.envi   =    <exp>.envi
  <exp2>.envi   =    <exp1>.envo
  <exp>.envo    =    <exp2>.envo
  <exp>.dnum    =  length(<exp>.envo)
```


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.