Introduction to Parsing
(adapted from CS 164 at Berkeley)

Outline

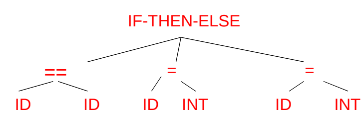
- Parser overview
- *Context-free grammars (CFG's)*
- Derivations
- *Syntax-Directed Translation*

The Functionality of the Parser

- **Input:** *sequence of tokens from lexer*
- **Output:** abstract syntax tree of the program
- **One-pass compiler**: directly generate assembly code
  - This is what you will do in the first assignment
  - Bali → SaM code

Example

- Pyth: if x == y: z =1
  else: z = 2
- Parser input: IF ID == ID : ID = INT ELSE : ID = INT
- Parser output (*abstract syntax tree):*



Why A Tree?

- Each stage of the compiler has two purposes:
  - Detect and filter out some class of errors
  - Compute some new information or translate the representation of the program to make things easier for later stages
- Recursive structure of tree suits recursive structure of language definition
- With tree, later stages can easily find "the else clause", e.g., rather than having to scan through tokens to find it.

Notation for Programming Languages

- Grammars:

  $E \to int$
  $E \to E + E$
  $E \to E * E$
  $E \to (E)$
- We can view these rules as rewrite rules
  - We start with E and replace occurrences of E with some right-hand side
- $E \to E * E \to (E) * E \to (E + E) * E \to \ldots$
  $\to (int + int) * int$

Context-Free Grammars

- A CFG consists of
  - A set of *non-terminals* N
    - By convention, written with capital letter in these notes
  - A set of *terminals* T
    - By convention, either lower case names or punctuation
  - A *start symbol* S (a non-terminal)
  - A set of *productions*
- Assuming $E \in N$

  $E \to \varepsilon$, or
  $E \to Y_1 Y_2 \ldots Y_n$ where $Y_i \in N \cup T$

Examples of CFGs

Simple arithmetic expressions:

$E \to int$
$E \to E + E$
$E \to E * E$
$E \to (E)$

- One non-terminal: E
- Several terminals: int, +, *, (,)
  - Called terminals because they are never replaced
- By convention the non-terminal for the first production is the start one

Key Idea

1. Begin with a string consisting of the start symbol
2. Replace any *non-terminal* X in the string by a right-hand side of some production
   $X \rightarrow Y_1 \ldots Y_n$
3. Repeat (2) until there are only terminals in the string
4. The successive strings created in this way are called *sentential forms.*

The Language of a CFG (Cont.)

Write

$X_1 \ldots X_n \rightarrow^* Y_1 \ldots Y_m$

if

$X_1 \ldots X_n \rightarrow \ldots \rightarrow \ldots \rightarrow Y_1 \ldots Y_m$

in 0 or more steps

The Language of a CFG

Let *G* be a context-free grammar with start symbol S. Then the language of *G* is:

$L(G) = \{ a_1 \ldots a_n \mid S \rightarrow^* a_1 \ldots a_n$ and every a_i is a terminal $\}$

Examples:

- $S \rightarrow 0$ also written as $S \rightarrow 0 \mid 1$
  $S \rightarrow 1$
  Generates the language { "0", "1" }
- What about $S \rightarrow 1\ A$
  $A \rightarrow 0 \mid 1$
- What about $S \rightarrow 1\ A$
  $A \rightarrow 0 \mid 1\ A$
- What about $S \rightarrow \varepsilon \mid (\ S\ )$

Derivations and Parse Trees

- A *derivation* is a sequence of sentential forms resulting from the application of a sequence of productions

  S → … → …

- *Parse tree: summary of derivation w/o specifying completely the order in which rules were applied*
  - Start symbol is the tree's root
  - For a production $X \rightarrow Y_1 \ldots Y_n$ add children $Y_1, \ldots, Y_n$ to node X

Derivation Example

- Grammar

  E → E + E | E * E | (E) | int

- String

  int * int + int

Derivation in Detail (1)

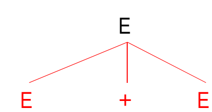
E

E

Derivation in Detail (2)

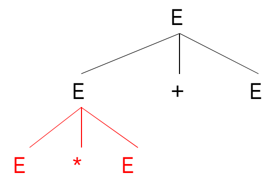
E

→ E + E



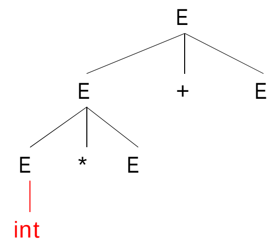
Derivation in Detail (3)

E
→ E + E
→ E * E + E



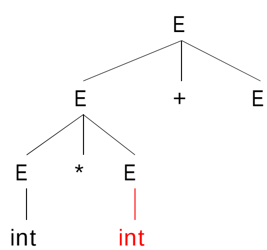
Derivation in Detail (4)

E
→ E + E
→ E * E + E
→ int * E + E



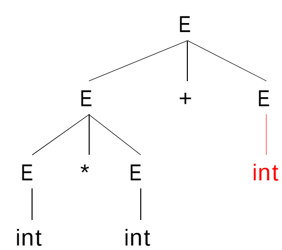
Derivation in Detail (5)

E
→ E + E
→ E * E + E
→ int * E + E
→ int * int + E



Derivation in Detail (6)

E
→ E + E
→ E * E + E
→ int * E + E
→ int * int + E
→ int * int + int



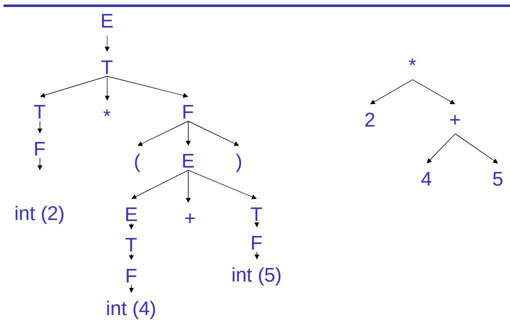
Notes on Derivations

- A parse tree has
  - Terminals at the leaves
  - Non-terminals at the interior nodes
- A left-right traversal of the leaves is the original input
- The parse tree shows the association of operations, the input string does not !
  - There may be multiple ways to match the input
  - Derivations (and parse trees) choose one

AST vs. Parse Tree

- AST is condensed form of a parse tree
  - operators appear at *internal* nodes, not at leaves.
  - "Chains" of single productions are collapsed.
  - Lists are "flattened".
  - Syntactic details are omitted
    - e.g., parentheses, commas, semi-colons
- AST is a better structure for later compiler stages
  - omits details having to do with the source language,
  - only contains information about the *essential* structure of the program.

Example: 2 * (4 + 5)   Parse tree *vs.* AST



Summary of Derivations

- We are not just interested in whether
  $s \in L(G)$
- Also need derivation (or parse tree) and AST.
- Parse trees slavishly reflect the grammar.
- Abstract syntax trees abstract from the grammar, cutting out detail that interferes with later stages.
- A derivation defines a parse tree
  - But one parse tree may have many derivations
- Derivations drive translation (to ASTs, etc.)
- Leftmost and rightmost derivations most important in parser implementation

Ambiguity

- Grammar

  E → E + E | E * E | (E) | int

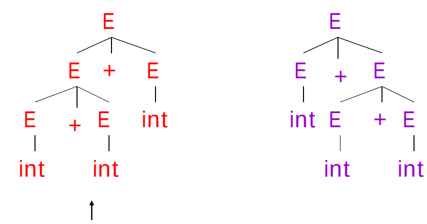
- Strings

  int + int + int

  int * int + int

Ambiguity. Example

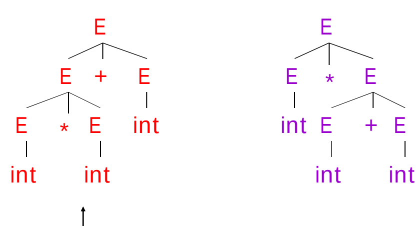
The string int + int + int has two parse trees



+ is left-associative

Ambiguity. Example

The string int * int + int has two parse trees



* has higher precedence than +

Ambiguity (Cont.)

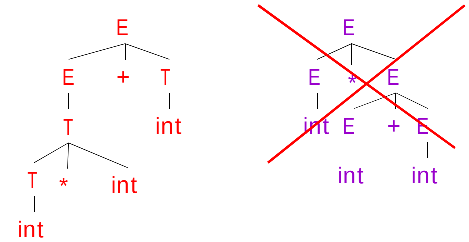
- A grammar is *ambiguous* if it has more than one parse tree for some string
  - Equivalently, there is more than one rightmost or leftmost derivation for some string
- Ambiguity is *bad*
  - Leaves meaning of some programs ill-defined
- Ambiguity is *common* in programming languages
  - Arithmetic expressions
  - IF-THEN-ELSE

Dealing with Ambiguity

- There are several ways to handle ambiguity
- Most direct method is to rewrite the grammar unambiguously
  - E → E + T | T
  - T → T * int | int | (E)
- Enforces precedence of * over +
- Enforces left-associativity of + and *

Ambiguity. Example

The int * int + int has only one parse tree now

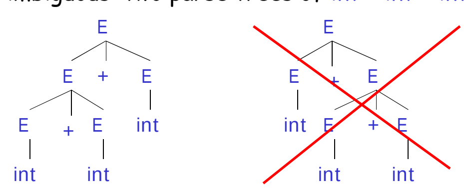


Ambiguity

- Impossible to convert automatically an ambiguous grammar to an unambiguous one
- Used with care, ambiguity can simplify the grammar
  - Sometimes allows more natural definitions
  - But we need disambiguation mechanisms
- Instead of rewriting the grammar
  - Use the more natural (ambiguous) grammar
  - Along with disambiguating declarations
- Most tools allow *precedence and associativity declarations* to disambiguate grammars
- Examples …

Associativity Declarations

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



- Left-associativity declaration: %left '+'

Summary

- Grammar is specified using a context-free language (CFL)
- Derivation: starting from start symbol, use grammar rules as rewrite rules to derive input string
  - Leftmost and rightmost derivations
- Parse trees and abstract syntax trees
- Ambiguous grammars
  - Ambiguity should be eliminated by modifying grammar, by specifying precedence rules etc. depending on how ambiguity arises in the grammar
- Remaining question: how do we find the derivation for a given input string?