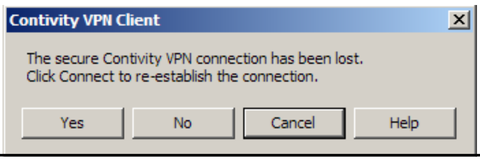
Introduction To Parsing

I'M HUNGRY. CAN I HAVE A SNACK?
SURE. HELP YOURSELF.
YOU CAN HAVE AN APPLE OR AN ORANGE FROM THE FRIDGE.
EVEN THOUGH WE'RE BOTH TALKING ENGLISH, WE'RE NOT SPEAKING THE SAME LANGUAGE.

Reading Quiz

- What does "recursive descent" mean?
- Name a "truth that might hurt".
- Name a "peril of JavaSchools".



Contivity VPN Client

The secure Contivity VPN connection has been lost.
Click Connect to re-establish the connection.

Outline

- Formal languages
- Parser overview
- Context-free grammars (CFGs)
- Derivations
- Ambiguity

In One Slide

- A **parser** takes a sequence of tokens as input. If the input is valid, it produces a *parse tree* (or *derivation*).
- **Context-free grammars** are a notation for specifying formal languages. They contain *terminals*, *non-terminals* and *productions* (aka *rewrite rules*).

#4

Formal Languages

- *Formal languages* are very important in CS
  - Especially in programming languages
- Regular languages
  - The "weakest" formal languages widely used
  - Many applications (e.g., virus scanning)
- Today we study *context-free languages*
  - A "stronger" type of formal language

#5

Limitations of Regular Languages

- Intuition: A finite automaton that runs long enough must repeat states
  - Pigeonhole Principle: imagine 20 states and 300 transitions
- A finite automaton can't remember how often it has visited a particular state
  - Only enough memory to store in which state it is
  - Cannot count, except up to a finite limit
- Language of balanced parentheses is not regular: $\{ (^n\ )^n \mid n > 0\}$

#6

The Functionality of the Parser

- **Input:** sequence of tokens from lexer
  - e.g., the .cl-lex files you make in PA2
- **Output:** *parse tree* of the program
  - Also called an *abstract syntax tree*.
- **Output:** error if the input is not valid.
  - e.g., “parse error on line 3”

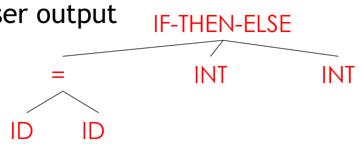
#7

Example

- Cool program text

  if x = y then 1 else 2 fi
- Parser input

  IF ID = ID THEN INT ELSE INT FI
- Parser output



#8

Comparison with Lexical Analysis

Phase	*Input*	*Output*
Lexer	Sequence of characters	Sequence of tokens
Parser	Sequence of tokens	Parse tree

#9

The Role of the Parser

- Not all sequences of tokens are programs
  - then x * / + 3 while x ; y z then
- The parser must **distinguish between valid and invalid sequences of tokens**
- We need
  - A language to describe valid sequences of tokens
  - A method (an algorithm) for distinguishing valid from invalid sequences of tokens

#10

Programming Language Structure

- Programming languages have **recursive** structure
- Consider the language of arithmetic expressions with integers, +, *, and ( )
- An expression is either:
  - an integer
  - an expression followed by "+" followed by expression
  - an expression followed by "*" followed by expression
  - a '(' followed by an expression followed by ')'
- int , int + int , (int + int) * int are expressions

#11

Notation for Programming Languages

- An alternative notation:

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

- We can view these rules as *rewrite rules*
  - We start with E and replace occurrences of E with some right-hand side

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

#12

Observation

- All arithmetic expressions can be obtained by a sequence of replacements
- Any sequence of replacements forms a valid arithmetic expression
- This means that we cannot obtain

  (int)))

  by any sequence of replacements. Why?

- This notation is a *context free grammar*

#13

Context Free Grammars

- A **context-free grammar** consists of
  - A set of ***non-terminals N***
    - Written in uppercase in these notes
  - A set of ***terminals T***
    - Lowercase or punctuation in these notes
  - A ***start symbol S*** (a non-terminal)
  - A set of ***productions*** (rewrite rules)
- Assuming $E \in N$

  $E \to \varepsilon$, or

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

#14

Examples of CFGs

Simple arithmetic expressions:

$E \to \text{int}$

$E \to E + E$

$E \to E * E$

$E \to (E)$

- 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 symbol

#15

The Language of a CFG

Read productions as replacement rules:

$X \to Y_1 \ldots Y_n$

Means X can be replaced by $Y_1 \ldots Y_n$

$X \to \varepsilon$

Means X can be erased

(replaced with empty string)

#16

Key Idea

To construct a valid sequence of terminals:

- Begin with a string consisting of the start symbol "S"
- Replace any <u>non-terminal</u> X in the string by a right-hand side of some production

  $X \to Y_1 \ldots Y_n$

3. Repeat (2) until there are only terminals in the string

#17

The Language of a CFG (Cont.)

More formally, write

$$X_1 \ldots X_{i-1}\, X_i\, X_{i+1} \ldots X_n \\ \to \\ X_1 \ldots X_{i-1}\, Y_1 \ldots Y_m\, X_{i+1} \ldots X_n$$

if there is a production

$$X_i \to Y_1 \ldots Y_m$$

#18

The Language of a CFG (Cont.)

Write

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

if

$X_1 \dots X_n \rightarrow \dots \rightarrow \dots \rightarrow Y_1 \dots Y_m$

in 0 or more steps

#19

The Language of a CFG

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

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

L(G) is a set of strings over the alphabet of terminals.

#20

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 $S \rightarrow 1\ B$
  $B \rightarrow 0 \mid 1\ B$
- What about $S \rightarrow \varepsilon \mid (\ S\ )$

#21

Arithmetic Example

Simple arithmetic expressions:

$$E \rightarrow E{+}E \mid E * E \mid (E) \mid \text{id}$$

Some elements of the language:

id	id + id
(id)	id * id
(id) * id	id * (id)

#22

Cool Example

A fragment of COOL:

```
EXPR  →  if EXPR then EXPR else EXPR fi
      |  while EXPR loop EXPR pool
      |  id
```

#23

Cool Example (Cont.)

Some elements of the language

id

if id then id else id fi

while id loop id pool

if while id loop id pool then id else id

if if id then id else id fi then id else id fi

#24

Notes

The idea of a CFG is a big step. But:

- Membership in a language is “yes” or “no”
  - we also need a **parse tree** of the input
- We must **handle errors** gracefully
- We need an **implementation** of CFGs
  - bison, yacc, ocamlyacc, ply, etc.

#25

More Notes

- Form of the grammar is important
  - Many grammars generate the same languages
    - Give me an example.
  - Automatic tools are sensitive to the grammar
  - Note: Tools for regular languages (e.g., flex) are also sensitive to the form of the regular expression, but this is rarely a problem in practice

#26

Derivations and Parse Trees

A ***derivation*** is a sequence of productions

$S \to \dots \to \dots$

A derivation can be **drawn as a tree**

- Start symbol is the tree’s root
- For a production $X \to Y_1 \dots Y_n$ add children $Y_1, \dots, Y_n$ to node X

#27

Derivation Example

- Grammar

  $E \rightarrow E{+}E \mid E * E \mid (E) \mid id$

- String

  $id * id + id$

- We're going to build a derivation

Derivation Example (Cont.)

$$\left.\begin{array}{ll} & E \\ \rightarrow & E{+}E \\ \rightarrow & E * E{+}E \\ \rightarrow & id * E + E \\ \rightarrow & id * id + E \\ \rightarrow & id * id + id \end{array}\right\}$$

Thus

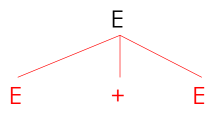
$E \rightarrow^{*}$ id * id + id

Derivation in Detail (1)

E

E

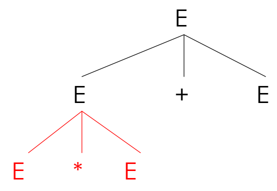
Derivation in Detail (2)



$$
\begin{aligned}
& E \\
\rightarrow\ & E+E
\end{aligned}
$$

#31

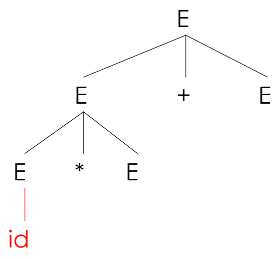
Derivation in Detail (3)



$$
\begin{aligned}
& E \\
\rightarrow\ & E+E \\
\rightarrow\ & E*E+E
\end{aligned}
$$

#32

Derivation in Detail (4)

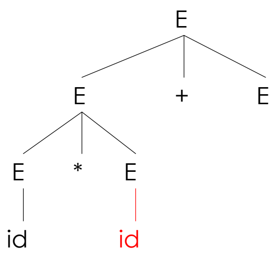


$$
\begin{aligned}
& E \\
\rightarrow\ & E+E \\
\rightarrow\ & E*E+E \\
\rightarrow\ & id*E+E
\end{aligned}
$$

#33

Derivation in Detail (5)

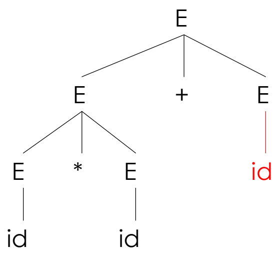
$$
\begin{aligned}
& E \\
\rightarrow\ & E+E \\
\rightarrow\ & E * E+E \\
\rightarrow\ & id * E + E \\
\rightarrow\ & id * id + E
\end{aligned}
$$



#34

Derivation in Detail (6)

$$
\begin{aligned}
& E \\
\rightarrow\ & E+E \\
\rightarrow\ & E * E+E \\
\rightarrow\ & id * E + E \\
\rightarrow\ & id * id + E \\
\rightarrow\ & id * id + id
\end{aligned}
$$



#35

Notes on Derivations

- A ***parse tree*** has
  - Terminals at the leaves
  - Non-terminals at the interior nodes
- A left-to-right traversal of the leaves is the original input
- The parse tree shows the association of operations, the input string does not!

#36

Left-most and Right-most Derivations

- The example is a ***left-most* derivation**
  - At each step, replace the left-most non-terminal
- There is an equivalent notion of a ***right-most* derivation**

$$
\begin{aligned}
& E \\
\rightarrow\ & E+E \\
\rightarrow\ & E+id \\
\rightarrow\ & E * E + id \\
\rightarrow\ & E * id + id \\
\rightarrow\ & id * id + id
\end{aligned}
$$

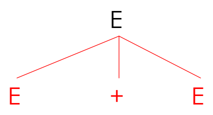
#37

Right-most Derivation in Detail (1)

$$E$$

#38

Right-most Derivation in Detail (2)

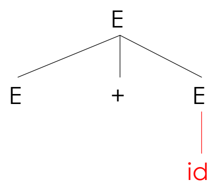


$$
\begin{aligned}
& E \\
\rightarrow\ & E+E
\end{aligned}
$$

#39

Right-most Derivation in Detail (3)

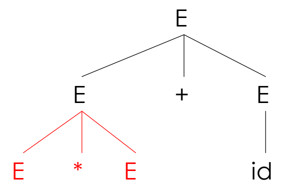
$$\begin{aligned}&E\\ \rightarrow\ &E+E\\ \rightarrow\ &E+id\end{aligned}$$



#40

Right-most Derivation in Detail (4)

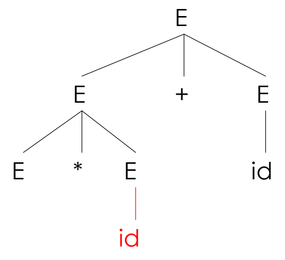
$$\begin{aligned}&E\\ \rightarrow\ &E+E\\ \rightarrow\ &E+id\\ \rightarrow\ &E*E+id\end{aligned}$$



#41

Right-most Derivation in Detail (5)

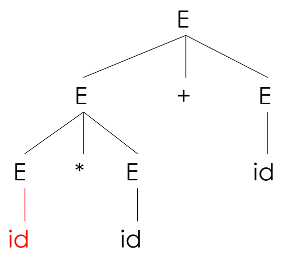
$$\begin{aligned}&E\\ \rightarrow\ &E+E\\ \rightarrow\ &E+id\\ \rightarrow\ &E*E+id\\ \rightarrow\ &E*id+id\end{aligned}$$



#42

Right-most Derivation in Detail (6)

$$
\begin{aligned}
& E \\
\rightarrow\ & E+E \\
\rightarrow\ & E+\mathrm{id} \\
\rightarrow\ & E * E + \mathrm{id} \\
\rightarrow\ & E * \mathrm{id} + \mathrm{id} \\
\rightarrow\ & \mathrm{id} * \mathrm{id} + \mathrm{id}
\end{aligned}
$$



#43

Derivations and Parse Trees

- Note that for each parse tree there is a left-most and a right-most derivation
- The difference is the order in which branches are added
- We will start with a parsing technique that yields left-most derivations
  - Later we'll move on to right-most derivations.

#44

Summary of Derivations

- We are not just interested in whether

  $s \in L(G)$
  - We need a parse tree for s
- A derivation defines a parse tree
  - But one parse tree may have many derivations
- Left-most and right-most derivations are important in parser implementation

#45

Review

- A parser consumes a sequence of tokens s and produces a parse tree
- Issues:
  - How do we **recognize** that s ∈ L(G) ?
  - A **parse tree** of s describes how s ∈ L(G)
  - **Ambiguity**: more than one parse tree (interpretation) for some string s
  - **Error**: no parse tree for some string s
  - How do we **construct the parse tree**?

#46

Ambiguity

- Grammar

  E → E + E | E * E | (E) | int

- Strings

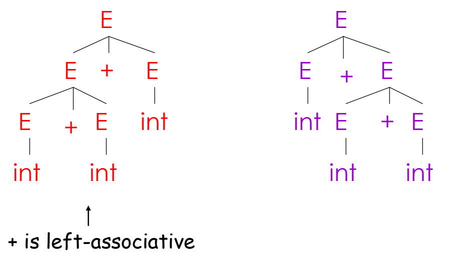
  int + int + int

  int * int + int

#47

Ambiguity. Example

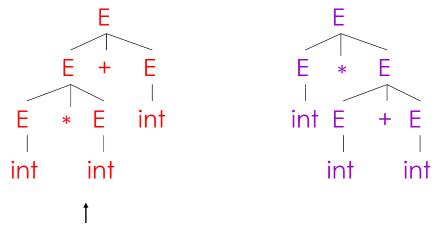
The string int + int + int has two parse trees



#48

Ambiguity. Example

The string int * int + int has two parse trees



* has higher precedence than +

#49

Ambiguity (Cont.)

- A grammar is ***ambiguous*** if it has more than one parse tree for some string
  - Equivalently, there is more than one right-most or left-most 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

#50

Dealing with Ambiguity

- There are several ways to handle ambiguity
- Most direct method is to rewrite the grammar unambiguously

  E → E + T | T

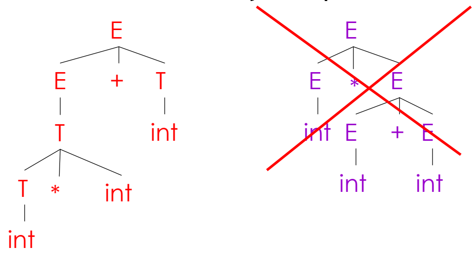
  T → T * int | int | (E)

- Enforces precedence of * over +
- Enforces left-associativity of + and *

#51

Ambiguity. Example

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



Ambiguity: The Dangling Else

- Consider the grammar

  E → if E then E

  | if E then E else E

  | OTHER

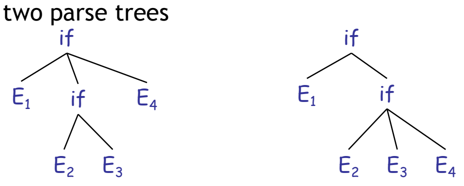
- This grammar is also ambiguous

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 (distinguish between matched and unmatched "then")

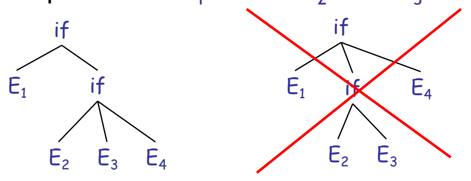
```
E →  MIF                 /* all then are matched */
   | UIF                 /* some then are unmatched */
MIF → if E then MIF else MIF
    |  OTHER
UIF → if E then E
    |  if E then MIF else UIF
```

- Describes the same set of strings

#55

The Dangling Else: Example 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

#56

Ambiguity

- No general techniques for handling 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
  - We need disambiguation mechanisms

#57

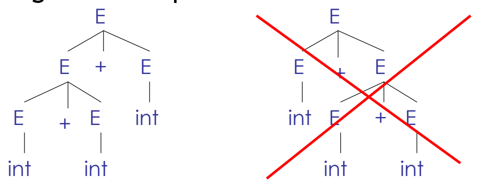
Precedence and Associativity Declarations

- 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 …

#58

Associativity Declarations

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

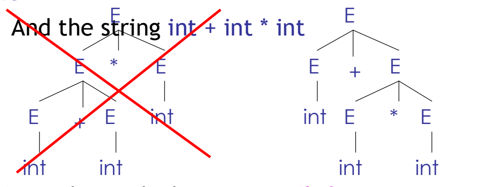


- Left-associativity declaration: %left +

#59

Precedence Declarations

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



- Precedence declarations: %left +
  %left *

#60

Review

- We can specify language syntax using CFG
- A parser will answer whether $s \in L(G)$
- … and will build a parse tree
- … and pass on to the rest of the compiler

- Next episode:
  - How do we answer $s \in L(G)$ and build a parse tree?

#61

Homework

- Thursday: WA1 (written homework) due
  - You must work alone.
  - Write or print out your answers.
  - Turn in before class Thrusday or in drop-box.

- Thursday: Chapters 2.4 - 2.4.1
  - 1 page in book, 3 pages on CD

- Friday: PA2 (Lexer) due
  - You may work in pairs.

#62