The Parser

- Scanner vs. parser
  - Why regular expressions are not enough
- Grammars (context-free grammars)
  - grammar rules
  - derivations
  - parse trees
  - ambiguous grammars
  - useful examples
- Reading:
  - Sections 4.1 and 4.2

The Functionality of the Parser

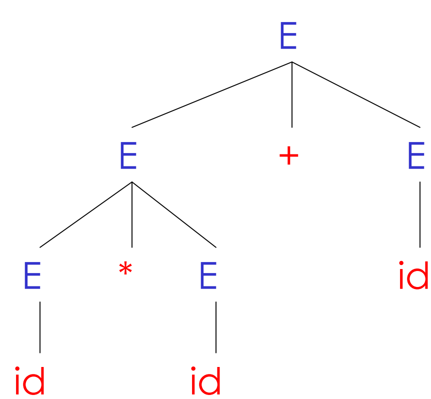
- **Input:** sequence of tokens from scanner
- **Output:** parse tree of the program
  - parse tree is generated if the input is a legal program
  - if input is an illegal program, syntax errors are issued
- Note:
  - Instead of parse tree, some parsers produce directly:
    - abstract syntax tree (AST) + symbol table (as in P3), or
    - intermediate code, or
    - object code
  - For the moment, we'll assume that parse tree is generated

Comparison with Lexical Analysis

Phase	*Input*	*Output*
Scanner	String of characters	String of tokens
Parser	String of tokens	Parse tree

Example

- **The program:**
  x * y + z
- **Input to parser:**
  ID TIMES ID PLUS ID
  we'll write tokens as follows:
  id * id + id
- **Output of parser:**
  the parse tree →



Why are regular expressions not enough?

- Write an automaton that accepts strings
  - "a", "(a)", "((a))", and "(((a)))"
  - How about: "a", "(a)", "((a))", "(((a)))", … "$(^k a)^k$"

What must parser do?

1. Recognizer: not all strings of tokens are programs
   - must distinguish valid and invalid strings of tokens
2. Translator: must expose program structure
   - e.g., associativity and precedence
   - hence must return the parse tree

We need:

- A language for describing valid strings of tokens
  - context-free grammars
  - (analogous to regular expressions in the scanner)
- A method for distinguishing valid from invalid strings of tokens (and for building the parse tree)
  - the parser
  - (analogous to the state machine in the scanner)

Context-Free Grammars (CFGs)

- Example: Simple Arithmetic Expressions
  - In English:
    - An integer is an arithmetic expression.
    - If exp_1 and exp_2 are arithmetic expressions, then so are the following:

      exp_1 - exp_2
      exp_1 / exp_2
      (exp_1)

- the corresponding CFG: we'll write tokens as follows:

exp → INTLITERAL	E → intlit
exp → exp MINUS exp	E → E - E
exp → exp DIVIDE exp	E → E / E
exp → LPAREN exp RPAREN	E → (E)

Reading the CFG

- The grammar has five terminal symbols:
  - **intlit, -, /, (,)**
  - terminals of a grammar = tokens returned by the scanner.
- The grammar has one non-terminal symbol:
  - **E**
  - non-terminals describe valid sequences of tokens
- The grammar has four productions or rules,
  - each of the form: E → α
    - left-hand side = a single non-terminal.
    - right-hand side = either
      - sequence of 1 or more terminals and/or non-terminals, or
      - ε (an empty production); again, the book uses symbol λ

Example, revisited

- Note:
  - a more compact way to write previous grammar:

    E → intlit | E - E | E / E | (E)

    or

    E → intlit
      | E - E
      | E / E
      | (E)

A formal definition of CFGs

- A CFG consists of
  - A set of *terminals T*
  - A set of *non-terminals N*
  - A *start symbol S* (a non-terminal)
  - A set of *productions:*

    $X \rightarrow Y_1\ Y_2 \ldots Y_n$

    where $X \in N$ and $Y_i \in T \cup N \cup \{\varepsilon\}$

Notational Conventions

- In these notes
  - Non-terminals are written upper-case
  - Terminals are written lower-case
  - Start symbol is left-hand side of first production

The Language of a CFG

The language defined by CFG is set of strings that can be derived from the start symbol of grammar

Derivation: Read productions as rules:

$$X \to Y_1 \ldots Y_n$$

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

Derivation: key idea

1. Begin with string of start symbol "S"
2. Replace any non-terminal *X* in string by rhs of some production

$$X \rightarrow Y_1 \ldots Y_n$$

3. Repeat (2) until no non-terminals in string

Derivation: an example

CFG:

$E \rightarrow id$

$E \rightarrow E + E$

$E \rightarrow E * E$

$E \rightarrow (E)$

Is string id * id + id in language defined by grammar?

derivation:

E

→ E+E

→ E*E+E

→ id*E+E

→ id*id+E

→ id*id+id

Terminals

- "Terminals" because there are no rules for replacing them
- Once generated, terminals are permanent
- Therefore, terminals are the tokens of the language

The Language of a CFG

More formally, write

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

if there is a production

$$X_i \rightarrow Y_1 \ldots Y_m$$

The Language of a CFG

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

The Language of a CFG

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

$$\{a_1 \dots a_n \mid S \rightarrow^* a_1 \dots a_n \text{ and every } a_i \text{ is a terminal}\}$$

Examples

Strings of balanced parentheses

$$\{(^i)^i \mid i \geq 0\}$$

The grammar:

$$S \rightarrow (S)$$

$$S \rightarrow \varepsilon$$

or

$$S \rightarrow (S) \mid \varepsilon$$

Arithmetic Example

Simple arithmetic expressions:

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

Some examples of strings in the language:

id

(id)

(id) * id

id + id

id * id

id * (id)

Notes

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

- Membership in a language is "yes" or "no"
  - we also need parse tree of the input!
  - furthermore, we must handle errors gracefully
- Need an "implementation" of CFG's,
  - i.e., the parser
  - we will create the parser using a parser generator
    - available generators: CUP, bison, yacc

More Notes

- Form of the grammar is important
  - Many grammars generate the same language
  - Parsers are sensitive to the form of the grammar
- Example:

  E → E + E
  | E – E
  | intlit

  is not suitable for LL(1) parser (common parser)
  Stay tuned, you will soon understand why

Derivations and Parse Trees

A *derivation* is a sequence of productions

$$S \to \ldots \to \ldots \to \ldots$$

A derivation can be drawn as a tree

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

Derivation Example

- Grammar

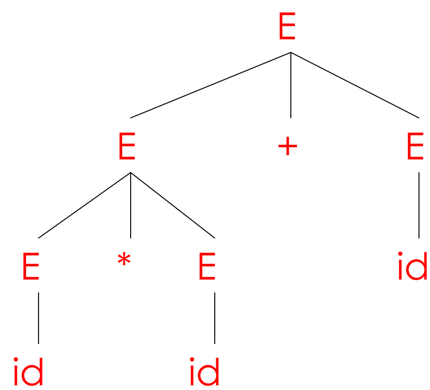
  $E \to E+E \mid E*E \mid (E) \mid id$

- String

  id * id + id

Derivation Example (Cont.)

E
→ E+E
→ E*E+E
→ id*E+E
→ id*id+E
→ id*id+id



Derivation in Detail (1)

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

E

E

Notes on Derivations

- A parse tree has
  - Terminals at the leaves
  - Non-terminals at the interior nodes
- An in-order traversal of the leaves is the original input
- The parse tree shows the association of operations, the input string does not

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

E
→ E+E
→ E*E+E
→ id*E+E
→ id*id+E
→ id*id+id

Right-most Derivation in Detail

id * id + id

E

E

Derivations and Parse Trees

- Note that right-most and left-most derivations have the same parse tree
- The difference is the order in which branches are added

Summary of Derivations

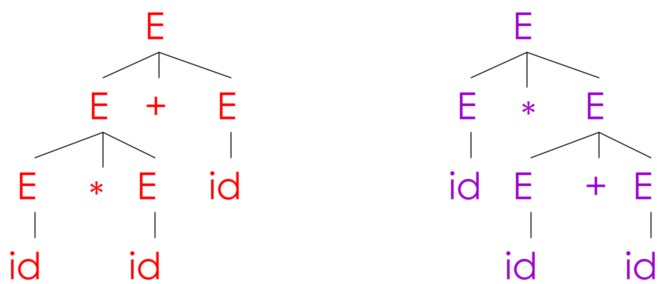
- Not just interested in whether $s \in L(G)$
  - We need a parse tree for *s*,
    (because we need to build the AST)
- A derivation defines a parse tree
  - But one parse tree may have many derivations
- Left-most and right-most derivations are important in parser implementation

Ambiguity

- Grammar
  $E \rightarrow E+E \mid E*E \mid (E) \mid id$
- String
  id * id + id

Ambiguity (Cont.)

This string has two parse trees



Example Parse Trees

- for each of parse trees, find the corresponding left-most derivation
- for each of parse trees, find the corresponding right-most derivation