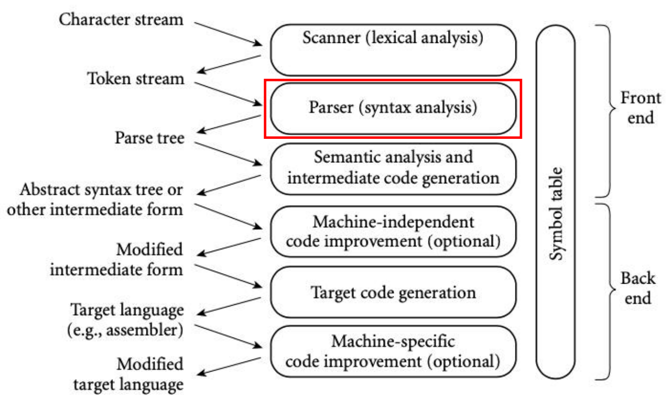
Syntax Analysis: Scanning and Parsing

204315: OPL

Outline

- Scanner
  - Regular Expression
  - Deterministic Finite Automaton
- Parser
  - Context-Free Grammar
  - LL Parsing
  - LR Parsing

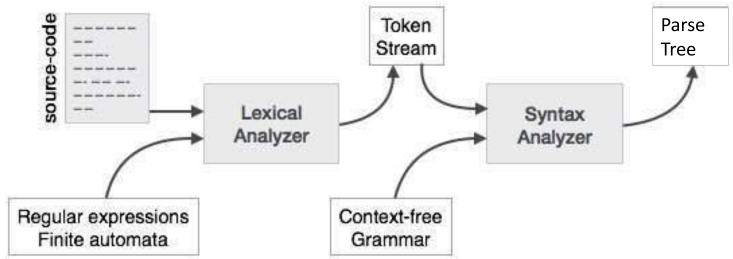
Compilation Process Overview (Recap)



Scott, M. L. (2016). *Programming Language Pragmatics*.

Syntax Analyzer

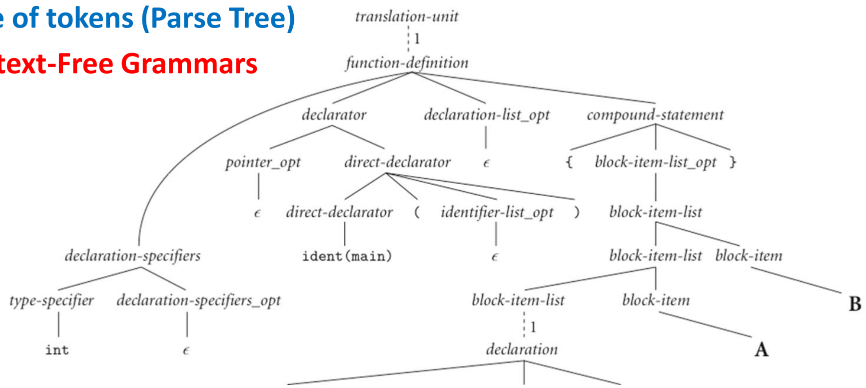
- **Syntax analyzer or parser**
  - takes the input from a lexical analyzer in the form of token streams
  - analyzes the source code (token stream) against the rules to detect any syntax errors in the code.
  - The output of this phase is a parse tree.



https://www.tutorialspoint.com/compiler_design/compiler_design_syntax_analysis.htm

Parsing phase

- **Build tree of tokens (Parse Tree)**
- Uses **Context-Free Grammars**



Scott, M. L. (2016). *Programming Language Pragmatics*.

Limitation of Regular Expression

- REs work well for defining tokens such as identifiers, constant etc..
- But unable to specify nested structure which are central to programming languages
- Therefore, cannot generate
  - balanced parentheses
  - nested chain structures

```
struct student_college_detail
{
    int college_id;
    char college_name[50];
};

struct student_detail
{
    int id;
    char name[20];
    float percentage;
    // structure within structure
    struct student_college_detail clg_data;
}stu_data;
```

https://www.quora.com/What-is-a-nested-structure-and-explain-with-examples-with-syntax

https://www.tutorialspoint.com/compiler_design/compiler_design_syntax_analysis.htm

Context-Free Grammars: Overview

- Context-free grammars (CFGs)
  - a set of recursive rules used to generate patterns of strings
  - used to describe context-free languages
  - CFG is sometimes called Backus-Naur Form (BNF)
- CFSs are studied in fields of
  - theoretical computer science
  - compiler design
  - linguistics
- CFG's are used to
  - describe programming languages and construct parser program in compiler

Karleigh Moore, Alex Chumbley, and Jimin Khim, Context Free Grammars: https://brilliant.org/wiki/context-free-grammars/

Context-Free Grammars: Formal definition

- A context-free grammar G is defined by the 4-tuple G = {V, Σ, R, S}
  - V is a set of non-terminals
    - represents a different type of phrase or clause in the sentence.
  - Σ , is a finite set of terminals, disjoint from V
    - represent actual content of the sentence
  - R, is a set of production rules
    - Member of R is a relation from V -> V x Σ
  - S, is a start symbol S
    - Represents the whole program

Context-Free Grammars: Example

- **V = {*expr, op*}** #non-terminal
- **Σ = {**id, number, +, -, *, /, (,)**}** #terminal
- **R =** #production rules

$$\begin{array}{lcl} expr & \longrightarrow & \texttt{id} \mid \texttt{number} \mid \texttt{-}\ expr \mid \texttt{(}\ expr\ \texttt{)} \\ & & \mid\ expr\ op\ expr \\ op & \longrightarrow & \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{/} \end{array}$$

- **S** = *expr* # start symbol

  - Each of the rules in a CFG is **known as a production**.
  - The **left-hand side symbols** of the productions are known as **variables**, or **non-terminals,** e.g., the language's tokens.
  - **Terminals cannot appear on the left-hand side of any productions**.
  - The **left-hand side** of the first production, is called the **start symbol**. It names the construct defined by the overall grammar.

CFG usage

$$\begin{array}{lcl} expr & \longrightarrow & \texttt{id} \mid \texttt{number} \mid \texttt{-}\ expr \mid \texttt{(}\ expr\ \texttt{)} \\ & & \mid\ expr\ op\ expr \\ op & \longrightarrow & \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{/} \end{array}$$

- CFG can be used to check if a string is grammatical (with respect to the language)
- Can the following statement be derived from the grammars on the top ?

slope * x + intercept

Checking steps

$$\begin{array}{llcl} \text{Rule 1} & expr & \longrightarrow & \texttt{id} \mid \texttt{number} \mid \texttt{-}\ expr \mid \texttt{(}\ expr\ \texttt{)} \\ & & & \mid\ expr\ op\ expr \\ \text{Rule 2} & op & \longrightarrow & \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{/} \end{array}$$

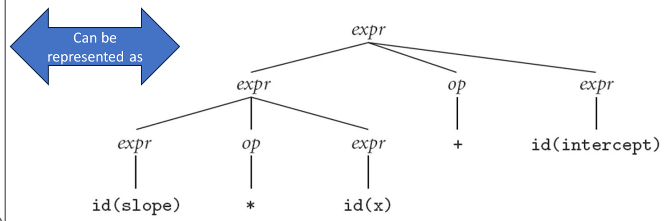
- Repeatedly rewrite the right-most-term until everything is terminal symbol
- Derivation steps for "slope * x + intercept" are

$expr \Longrightarrow expr\ op\ \underline{expr}$	Rule 1
$\Longrightarrow expr\ \underline{op}\ \texttt{id}$	Rule 2
$\Longrightarrow \underline{expr}\ \texttt{+}\ \texttt{id}$	Rule 1
$\Longrightarrow expr\ op\ \underline{expr}\ \texttt{+}\ \texttt{id}$	Rule 1
$\Longrightarrow expr\ \underline{op}\ \texttt{id}\ \texttt{+}\ \texttt{id}$	Rule 2
$\Longrightarrow \underline{expr}\ \texttt{*}\ \texttt{id}\ \texttt{+}\ \texttt{id}$	Rule 1
$\Longrightarrow \underset{\texttt{(slope)}}{\texttt{id}}\ \texttt{*}\ \underset{\texttt{(x)}}{\texttt{id}}\ \texttt{+}\ \underset{\texttt{(intercept)}}{\texttt{id}}$	

Parse tree

- Parse tree is a graphical representation of the derivations

$$\begin{array}{ll} expr & \Longrightarrow expr\ op\ \underline{expr} \\ & \Longrightarrow expr\ \underline{op}\ \texttt{id} \\ & \Longrightarrow \underline{expr}\ \texttt{+}\ \texttt{id} \\ & \Longrightarrow expr\ op\ \underline{expr}\ \texttt{+}\ \texttt{id} \\ & \Longrightarrow expr\ \underline{op}\ \texttt{id}\ \texttt{+}\ \texttt{id} \\ & \Longrightarrow \underline{expr}\ \texttt{*}\ \texttt{id}\ \texttt{+}\ \texttt{id} \\ & \Longrightarrow \underset{\texttt{(slope)}}{\texttt{id}}\ \texttt{*}\ \underset{\texttt{(x)}}{\texttt{id}}\ \texttt{+}\ \underset{\texttt{(intercept)}}{\texttt{id}} \end{array}$$

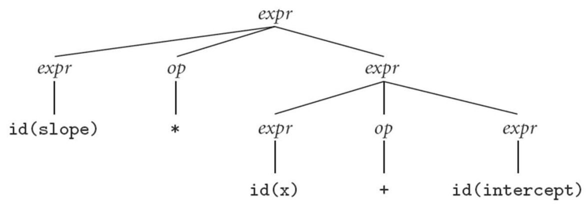


Alternative derivation

$$
\begin{aligned}
expr &\longrightarrow \texttt{id} \mid \texttt{number} \mid \texttt{-}\ expr \mid \texttt{(}\ expr\ \texttt{)} \\
&\quad\mid expr\ op\ expr \\
op &\longrightarrow \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{/}
\end{aligned}
$$

- Alternative parse tree for "slope * x + intercept" if we rewrite the left-most term
- Is this derivation correct ?

expr => expr op expr
=> id op expr
=> id * expr
=> id * expr op expr
=> id * id op expr
=> id * id + expr
=> id * id + id
(slope) (x) (intercept)



Note: Grammars which produce more than one parse tree is ambiguous – more on this in later slides

Another CFG with precedence

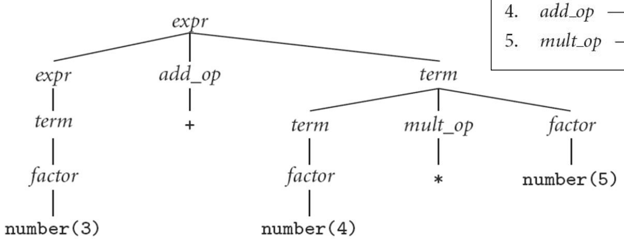
- A better version of grammar can capture **precedence**

1. $expr \longrightarrow term \mid expr\ add\_op\ term$
2. $term \longrightarrow factor \mid term\ mult\_op\ factor$
3. $factor \longrightarrow \texttt{id} \mid \texttt{number} \mid \texttt{-}\ factor \mid \texttt{(}\ expr\ \texttt{)}$
4. $add\_op \longrightarrow \texttt{+} \mid \texttt{-}$
5. $mult\_op \longrightarrow \texttt{*} \mid \texttt{/}$

Derivation using the new grammars

- Parse tree for expression grammar (with left associativity) for
3 + 4 * 5

1. $expr \longrightarrow term \mid expr\ add\_op\ term$
2. $term \longrightarrow factor \mid term\ mult\_op\ factor$
3. $factor \longrightarrow \texttt{id} \mid \texttt{number} \mid \texttt{-}\ factor \mid \texttt{(}\ expr\ \texttt{)}$
4. $add\_op \longrightarrow \texttt{+} \mid \texttt{-}$
5. $mult\_op \longrightarrow \texttt{*} \mid \texttt{/}$



Programming Language Parser

- A parser is to analyze the relationship of each word in a sentence according to the principles of grammar language
  - CFG is a generator for a context-free language (CFL)
- The parser accomplishes two tasks:
  - parsing the code
  - looking for (syntax) errors and generating a parse tree as the output of the phase
- Parsers are expected to parse the whole code even if some (semantic) errors exist in the program

Parsing process

- Derivation
  - a sequence of production rules, in order to get the input string.
- During parsing, we take two decisions for some sentential form of input:
  - deciding the non-terminal which is to be replaced
  - deciding the production rule, by which, the non-terminal will be replaced.
- To decide which non-terminal to be replaced with production rule, we can have two options:
  - **Left-most Derivation**
  - **Right-most Derivation**

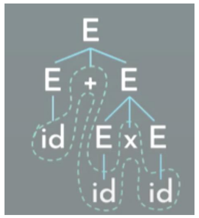
Two approaches to Parsing

- Left-to-right Left-most Derivation (LL Parsing): also called 'top-down', or 'predictive' parsers
  - **scan and replace** the input with **production rules**, **from left to right**.
  - The sentential form derived by the left-most derivation is called the **left-sentential form**.
- Left-to-right Right-most Derivation (LR Parsing): also called 'bottom-up', or 'shift-reduce' parsers
  - **scan and replace** the input with **production rules**, **from right to left**.
  - The sentential form derived from the right-most derivation is called the **right-sentential form**.

LL vs LR

- **Example**
  - **Production rules:**
    - E → E + E
    - E → E * E
    - E → id
  - **Input string:** id + id * id

Parse Tree



- **The left-most derivation is:**
  - E → E + E
  - E → id + E
  - E → id + E * E
  - E → id + id * E
  - E → id + id * id

- **The right-most derivation is:**
  - E → E + E
  - E → E + E * E
  - E → E + E * id
  - E → E + id * id
  - E → id + id * id

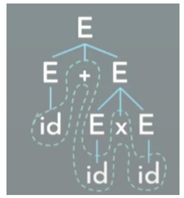
https://www.youtube.com/watch?v=gUaAKAj-rqA

Ambiguity in CFGs

E → E + E | E * E | id

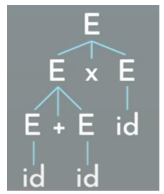
- **The left-most & right-most derivation [1]:**

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



- **The left-most & right-most derivation [2]:**

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

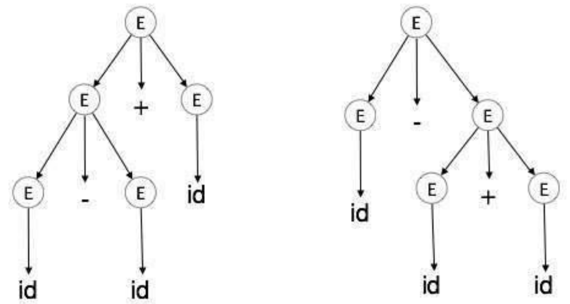


https://www.youtube.com/watch?v=gUaAKAj-rqA

Ambiguity in Grammar

- A grammar G is said to be ambiguous if it has more than one parse tree (left or right derivation) for at least one string.
- **Example**
  - E → E + E
  - E → E – E
  - E → id

For the string **id + id – id**, the above grammar generates two parse trees:



https://www.tutorialspoint.com/compiler_design/compiler_design_syntax_analysis.htm

Is ambiguity bad?

- The language generated by an ambiguous grammar is said to be inherently ambiguous
- Ambiguity in grammar is not good for a compiler construction
- No method can detect and remove ambiguity automatically,
  - but it can be removed by either re-writing the whole grammar without ambiguity, or
  - by setting and following associativity and precedence constraints

Ambiguity in associativity

- Associativity
  - If an operand has operators on both sides, the side on which the operator takes this operand is decided by the associativity of those operators.
  - If the operation is left-associative, then the operand will be taken by the left operator or if the operation is right-associative, the right operator will take the operand.
- Left associative example
  - Operations such as Addition, Multiplication, Subtraction, and Division are left associative.
  - If the expression contains: id op id op id, it will be evaluated as: (id op id) op id
  - For example, (id + id) + id
- Right associative example
  - Operations like Exponentiation are right associative, i.e., the order of evaluation in the same expression will be: id op (id op id)
  - For example, id ^ (id ^ id)

Ambiguity in precedence

- If two different operators share a common operand, the precedence of operators decides which will take the operand.
  - That is, 2+3*4 can have two different parse trees,
    - one corresponding to (2+3)*4 and
    - another corresponding to 2+(3*4).
  - By setting precedence among operators, this problem can be easily removed.
  - As in the previous example,
    - mathematically * (multiplication) has precedence over + (addition),
    - so the expression 2+3*4 will always be interpreted as: 2 + (3 * 4)
- These methods decrease the chances of ambiguity in a language or its grammar

Real-world example: Python’s grammars

```
# If statement
# ----------------

if_stmt:
      | 'if' name_expression ':' block elif_stmt
      | 'if' name_expression ':' block [else_block]

elif_stmt:
      | 'elif' name_expression ':' block elif_stmt
      | 'elif' name_expression ':' block [else_block]
else_block:
      | 'else' ':' block
```

https://docs.python.org/3/reference/grammar.html

References

- Scott, M. L. (2016). Programming Language Pragmatics.
- Tutorialspoint, https://www.tutorialspoint.com/, accessed on 11/22/2023
- https://docs.python.org/3/reference/grammar.html