Syntax Analysis: Scanning and Parsing

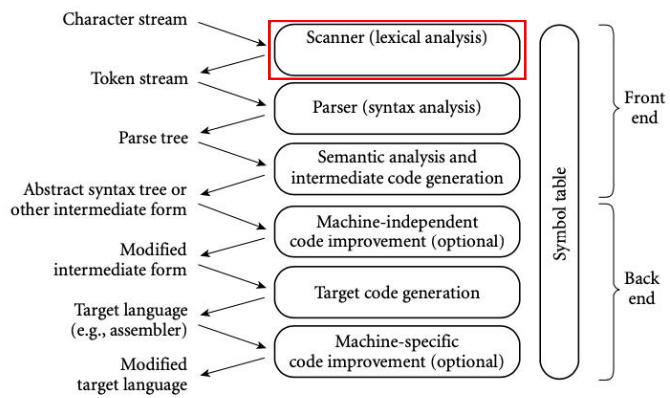
204315: OPL

Outline

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

Scanner

Compilation Process Overview (Recap)



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

Scanner (Recap)

- **Program tokenization**
- Uses **regular expression**
  - **Regular expressions have the capability to express languages.**

```
int main() {
    int i = getint(), j = getint();
    while (i != j) {
        if (i > j) i = i - j;
        else j = j - i;
    }
    putint(i);
}
```

int	main	(	)	{	int	i	=
getint	(	)	,	j	=	getint	(
)	;	while	(	i	!=	j	)
{	if	(	i	>	j	)	i
=	i	-	j	;	else	j	=
j	-	i	;	}	putint	(	i
)	;	}					

Scanner

- responsible for
  - tokenizing source code
  - removing comments
  - (often) dealing with *pragmas* (i.e., compiler directives)
  - saving text of identifiers, numbers, strings
  - saving source locations (file, line, column) for error messages

Scanner/Lexical Analysis

- Unlike natural languages such as English or Thai, **computer languages must be precise**.
  - To provide the precision, **language designers use formal syntactic (syntax) and semantic notation**.
- Different programming languages
  - often provide features with very **similar semantics** but very **different syntax.**
  - It is generally much easier to learn a new language if one is able to identify the common semantic ideas beneath the unfamiliar syntax.

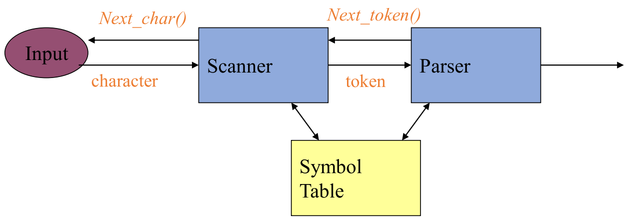
Scanner/Lexical Analysis

- **Lexical analysis** is the first phase of a compilation process.
  - reads/scans character streams from the source code
  - checks for legal tokens
  - passes the data to the syntax analyzer when it demands.
- **Lexical analyzer** needs to scan and identify only a finite set of valid string/token
  - It searches for the pattern defined by the language rules.
- **Regular expressions** have the capability to express finite languages by defining a pattern for finite strings of symbols.
  - The grammar defined by regular expressions is known as **regular grammar**
  - The language defined by regular grammar is known as **regular language**.

Syntax analyzer: checks if the source code follows the grammatical rules of the programming language.

Scanner input/output

- INPUT: sequence of characters
- OUTPUT: sequence of tokens



Some Definitions

- **token** – set of strings that is meaningful in source language
- **pattern** – a rule describing a set of string
- **lexeme** – a sequence of characters that matches the patterns of tokens

Some Examples

Token	Pattern	Sample Lexeme
while	while	while
relation_op	= | != | < | >	<
integer	(0-9)*	42
string	Characters between " "	"hello"

Regular Expression (RE)

- A way to describe pattern that specifies how tokens are generated
- Examples of RE for : *number token, integer token, real token, …*

$$
\begin{aligned}
&number \longrightarrow integer \mid real \\
&integer \longrightarrow digit\ digit^{*} \\
&real \longrightarrow integer\ exponent \mid decimal\ (\, exponent \mid \epsilon \,) \\
&decimal \longrightarrow digit^{*}\ (\ .\ digit \mid digit\ .\ )\ digit^{*} \\
&exponent \longrightarrow (\, \mathtt{e} \mid \mathtt{E} \,)\ (\, + \mid - \mid \epsilon \,)\ integer \\
&digit \longrightarrow \mathtt{0} \mid \mathtt{1} \mid \mathtt{2} \mid \mathtt{3} \mid \mathtt{4} \mid \mathtt{5} \mid \mathtt{6} \mid \mathtt{7} \mid \mathtt{8} \mid \mathtt{9}
\end{aligned}
$$

RE and their operations

- RE is one of the followings
  - A character
  - The empty string, denoted by ε
  - Two regular expressions concatenated
  - Two regular expressions separated by | (i.e., or)
  - A regular expression followed by the Kleene star * (concatenation of zero or more strings)

$$
\begin{aligned}
number &\longrightarrow integer \mid real \\
integer &\longrightarrow digit\ digit\ * \\
real &\longrightarrow integer\ exponent \mid decimal\ (\ exponent \mid \epsilon\ ) \\
decimal &\longrightarrow digit\ *\ (\ .\ digit \mid digit\ .\ )\ digit\ * \\
exponent &\longrightarrow (\ \mathtt{e} \mid \mathtt{E}\ )\ (\ + \mid - \mid \epsilon\ )\ integer \\
digit &\longrightarrow \mathtt{0} \mid \mathtt{1} \mid \mathtt{2} \mid \mathtt{3} \mid \mathtt{4} \mid \mathtt{5} \mid \mathtt{6} \mid \mathtt{7} \mid \mathtt{8} \mid \mathtt{9}
\end{aligned}
$$

Check your understanding

- Are these lexemes generated by these REs?
  - 3.1415
  - 1.1.0
  - 15e-5
  - 3E++
  - .357

$$
\begin{aligned}
number &\longrightarrow integer \mid real \\
integer &\longrightarrow digit\ digit\ * \\
real &\longrightarrow integer\ exponent \mid decimal\ (\ exponent \mid \epsilon\ ) \\
decimal &\longrightarrow digit\ *\ (\ .\ digit \mid digit\ .\ )\ digit\ * \\
exponent &\longrightarrow (\ \mathtt{e} \mid \mathtt{E}\ )\ (\ + \mid - \mid \epsilon\ )\ integer \\
digit &\longrightarrow \mathtt{0} \mid \mathtt{1} \mid \mathtt{2} \mid \mathtt{3} \mid \mathtt{4} \mid \mathtt{5} \mid \mathtt{6} \mid \mathtt{7} \mid \mathtt{8} \mid \mathtt{9}
\end{aligned}
$$

Recognizing Regular Expression

- A mechanical approach to check if a string is generated by a particular RE is by constructing and using a Deterministic Finite Automaton (DFA)
- A DFA is a finite-state machine that accepts or rejects a given string of symbols, by running through a state sequence uniquely determined by the string
- Multiple version of DFAs can be constructed from a single RE

Deterministic Finite Automaton

- A deterministic finite automaton M is a 5-tuple, (Q , Σ , δ , q0 , F) consisting of
  - a finite set of states Q
  - a finite set of input symbols called the alphabet Σ
  - a transition function $\delta : Q \times \Sigma \rightarrow Q$
  - an initial or start state $q0 \in Q$
  - a set of accept states $F \subseteq Q$

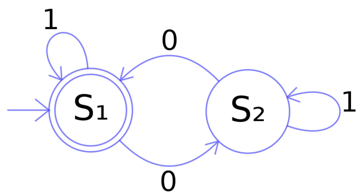
DFA Example

$M = (Q, \Sigma, \delta, q_0, F)$ where

- $Q = \{S_1, S_2\}$
- $\Sigma = \{0, 1\}$
- $q_0 = S_1$
- $F = \{S_1\}$ and
- δ is defined by the following state transition table:

	0	1
S_1	S_2	S_1
S_2	S_1	S_2

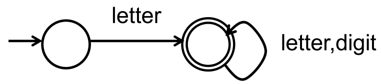
- Given RE = (1*) (0 (1*) 0 (1*))*
  - Possible valid strings
    - 10101
    - 0101
    - 0000
  - Invalid string
    - 000
    - 011
- The corresponding DFA is



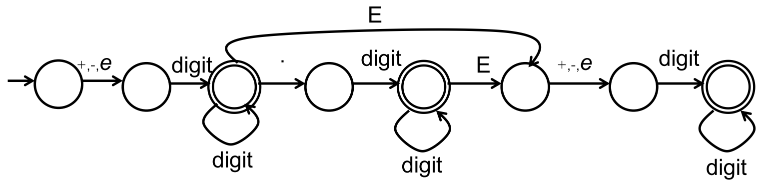
https://en.wikipedia.org/wiki/Deterministic_finite_automaton

More DFA examples

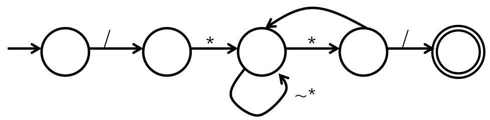
- Identifier



- Numeric



- C comment



Ref: 2301373 Introduction to Compilers

Remarks

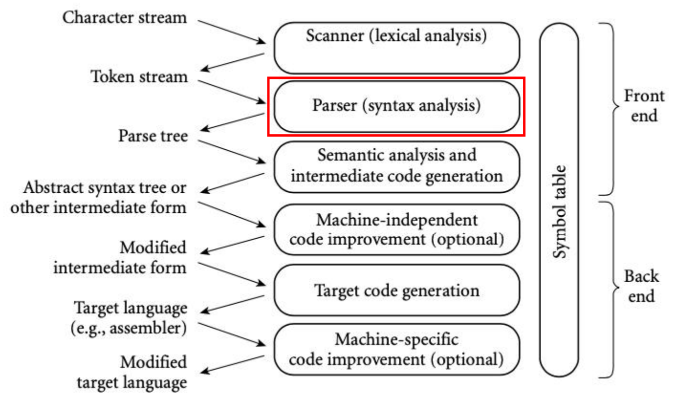
- We run the machine (DFA) over and over to get one token after another
  - always take the longest possible token from the input thus foobar is foobar and never f or foo or foob
  - more to the point, 3.14159 is a real const and never 3, ., and 14159
- Regular expressions "generate" a regular language; DFAs "recognize" it

How to construct a Scanner?

- Manual
  - Write down regular expressions for your language syntax
  - Construct a DFA for the regular expressions
  - Write a program (as a set of nested IFs) which walks the DFA
- Semi-automatic
  - Write down regular expressions for your language syntax
  - Use automatic softwares such as *lex, flex, jflex* that reads REs and source code and produce the tokenization result

Parser

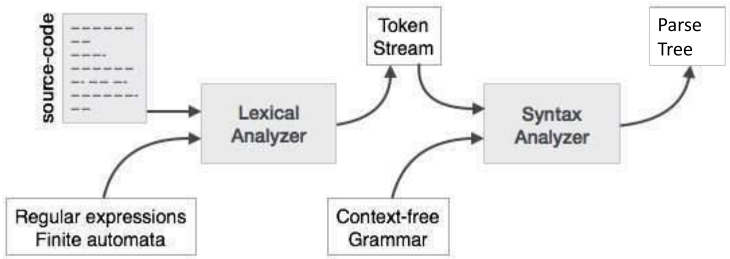
Compilation Process Overview (Recap)



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

Syntax Analyzer

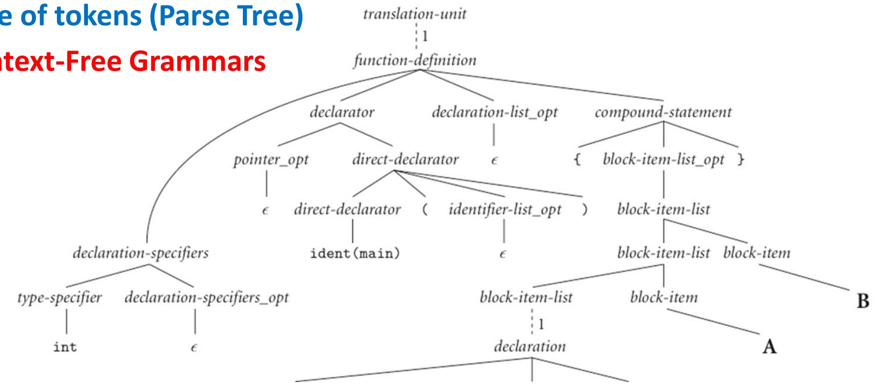
- **Syntax analyzer or parser**
  - takes the input from a lexical analyzer in the form of token streams
  - analyzes the source code (token stream) against the rules to detect any syntax errors in the code.
  - The output of this phase is a parse tree.



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

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

- **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

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

- 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

Rule 1
Rule 2

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

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

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

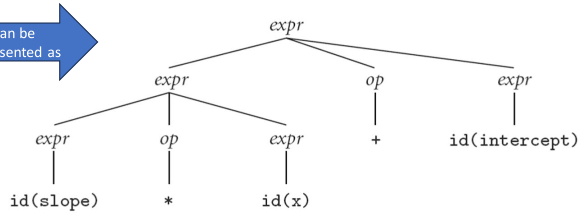
Rule 1
Rule 2
Rule 1
Rule 1
Rule 2
Rule 1

Parse tree

- Parse tree is a graphical representation of the derivations

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

Can be represented as

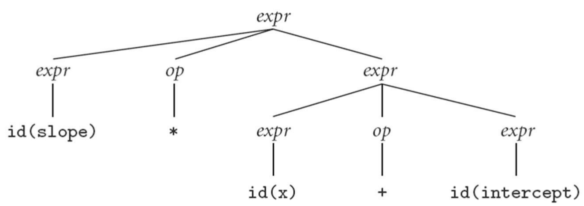


Alternative derivation

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

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

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

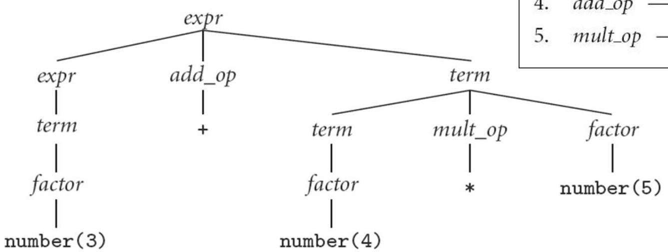


Note: Grammars which produce more than one parse tree is ambiguous – more on this in later slides

Another CFG with precedence

- A better version of grammar can capture **precedence**
- Parse tree for expression grammar (with left associativity) for
  `3 + 4 * 5`

1. *expr* $\longrightarrow$ *term* | *expr* *add_op* *term*
2. *term* $\longrightarrow$ *factor* | *term* *mult_op* *factor*
3. *factor* $\longrightarrow$ `id` | `number` | `-` *factor* | `(` *expr* `)`
4. *add_op* $\longrightarrow$ `+` | `-`
5. *mult_op* $\longrightarrow$ `*` | `/`



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**

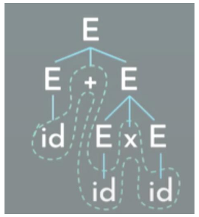
Two approaches to Parsing

- 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

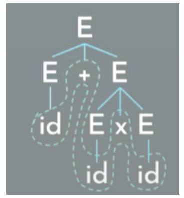
- **The right-most derivation is:**
  - E → E + E
  - E → E + E * E
  - E → E + E * id
  - E → E + id * id
  - E → id + id * id

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

Ambiguity in CFGs **E → E + E | E * E | id**

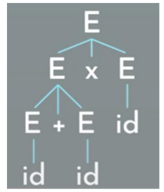
- **The left-most & right-most derivation [1]:**

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

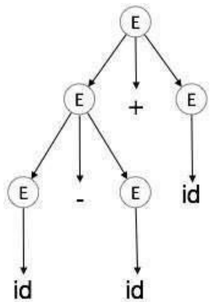


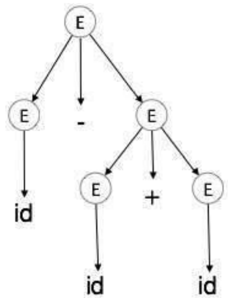
https://www.youtube.com/watch?v=gUaAKAj-rqA

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

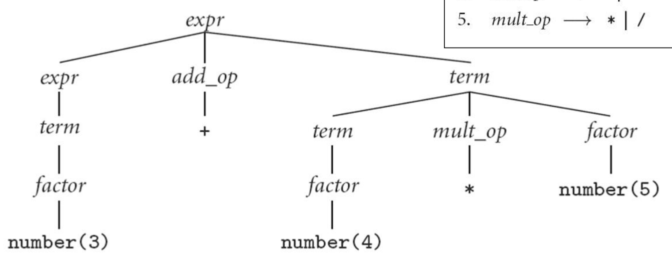
- If two different operators share a common operand, the precedence of operators decides which will take the operand.
  - That is, 2+3*4 can have two different parse trees,
    - one corresponding to (2+3)*4 and
    - another corresponding to 2+(3*4).
  - By setting precedence among operators, this problem 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

CFG with precedence

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

1. *expr* ⟶ *term* | *expr add_op term*
2. *term* ⟶ *factor* | *term mult_op factor*
3. *factor* ⟶ `id` | `number` | - *factor* | (*expr*)
4. *add_op* ⟶ + | -
5. *mult_op* ⟶ * | /

- **Example: "3 + 4 * 5"**



3 + (4 * 5)

3 + 20

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

Table-driven parsing

- We can have a table for selecting which rule to used based on the current non-terminal and the current input token
- **For example:** if the current non-terminal is *expr* and the current input token is *id* we will use production rule #7

$$ → ***end marker token***

```
program   ⟶ stmt_list $$
stmt_list ⟶ stmt stmt_list
stmt_list ⟶ ε
stmt      ⟶ id := expr
stmt      ⟶ read id
stmt      ⟶ write expr
expr      ⟶ term term_tail
term_tail ⟶ add_op term term_tail
term_tail ⟶ ε
term      ⟶ factor factor_tail
factor_tail ⟶ mult_op factor factor_tail
factor_tail ⟶ ε
factor    ⟶ ( expr )
factor    ⟶ id
factor    ⟶ number
add_op    ⟶ +
add_op    ⟶ -
mult_op   ⟶ *
mult_op   ⟶ /
```

Top-of-stack nonterminal	Current input token											
	id	number	read	write	:=	(	)	+	-	*	/	$$
program	1	–	1	1	–	–	–	–	–	–	–	1
stmt_list	2	–	2	2	–	–	–	–	–	–	–	3
stmt	4	–	5	6	–	–	–	–	–	–	–	–
expr	7	7	–	–	–	7	–	–	–	–	–	–
term_tail	9	–	9	9	–	–	9	8	8	–	–	9
term	10	10	–	–	–	10	–	–	–	–	–	–
factor_tail	12	–	12	12	–	–	12	12	12	11	11	12
factor	14	15	–	–	–	13	–	–	–	–	–	–
add_op	–	–	–	–	–	–	–	16	17	–	–	–
mult_op	–	–	–	–	–	–	–	–	–	18	19	–

Figure 2.19 LL(1) parse table for the calculator language. Table entries indicate the production to predict (as numbered in Figure 2.22). A dash indicates an error. When the top-of-stack symbol is a terminal, the appropriate action is always to match it against an incoming token from the scanner. An auxiliary table, not shown here, gives the right-hand-side symbols for each production.

LL(1) grammar
('' is ε):

```
E -> T E'
E' -> + T E'
E' -> ''
T -> F T'
T' -> * F T'
T' -> ''
F -> ( E )
F -> id
```

>>

FIRST	FOLLOW	Nonterminal	+	*	(	)	id	$
{(,id}	{$,)}	E			E -> T E'		E -> T E'	
{+,''}	{$,)}	E'	E' -> + T E'			E' -> ''		E' -> ''
{(,id}	{+,$,)}	T			T -> F T'		T -> F T'	
{*,''}	{+,$,)}	T'	T' -> ''	T' -> * F T'		T' -> ''		T' -> ''
{(,id}	{*,+,$,)}	F			F -> (E)		F -> id	

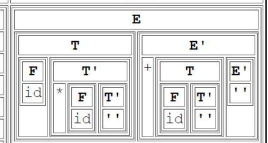
Maximum number of steps: 100

Input (tokens): id * id + id

GO!

Trace		
Stack	Input	Rule
$ E	id * id + id $	
$ E' T	id * id + id $	E -> T E'
$ E' T' F	id * id + id $	T -> F T'
$ E' T' id	id * id + id $	F -> id
$ E' T'	* id + id $	
$ E' T' F *	* id + id $	T' -> * F T'
$ E' T' F	id + id $	
$ E' T' id	id + id $	F -> id
$ E' T'	+ id $	
$ E'	+ id $	T' -> ''
$ E' T +	+ id $	E' -> + T E'
$ E' T	id $	
$ E' T' F	id $	T -> F T'
$ E' T' id	id $	F -> id
$ E' T'	$	
$ E'	$	T' -> ''
$	$	E' -> ''

Tree



LL(1) Parser Visualization

Zak Kincaid and Shaowei Zhu, http://jsmachines.sourceforge.net/machines/ll1.html

LL(1) Parser Visualization

Nullable/First/Follow Table and Transition Table

```
E -> T E'
E' -> + T E'
E' -> ''
T -> F T'
T' -> * F T'
T' -> ''
F -> ( E )
F -> id
```

Nonterminal	Nullable?	First	Follow
S	✕	(, id	
E	✕	(, id	), $
E'	✓	+	), $
T	✕	(, id	+,), $
T'	✓	*	+,), $
F	✕	(, id	+, *,), $

	$	+	*	(	)	id
S				S ::= E $		S ::= E $
E				E ::= T E'		E ::= T E'
E'	E' ::= ε	E' ::= + T E'			E' ::= ε	
T				T ::= F T'		T ::= F T'
T'	T' ::= ε	T' ::= ε	T' ::= * F T'		T' ::= ε	
F				F ::= (E)		F ::= id

Parsing

Token stream separated by spaces: id * id

Start/Reset   Step Forward

www.cs.princeton.edu says

Parsing complete! Press reset to see it again.

OK

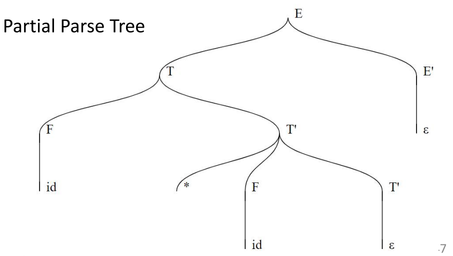
Stack

Remaining Input

Rule

Match $

Partial Parse Tree



https://www.cs.princeton.edu/courses/archive/spring20/cos320/LL1/

References

- PPT of Lecture 2: Lexical Analysis, CS 540, George Mason University
- PPT of 2301373 Introduction to Compilers
- Scott, M. L. (2016). Programming Language Pragmatics.
- Tutorialspoint, https://www.tutorialspoint.com/, accessed on 11/22/2023
- Python, https://docs.python.org/3/reference/grammar.html
- Zak Kincaid and Shaowei Zhu, http://jsmachines.sourceforge.net/machines/ll1.html