CONCEPTS OF PROGRAMMING LANGUAGES

Lab2

Spring Semester 2025

Outline

- Syntax and Semantic of Programming Languages
- Lexical Analyzer
- Syntax Analyzer

SYNTAX AND SEMANTIC OF A PROGRAMMING LANGUAGES

Syntax and Semantic of a Programming Languages

Syntax - The form of its expressions, statements, and program units.

Semantics - The meaning of the expressions, statements, and program units.

Syntax Error Example1

```
#include <iostream>
int main()
{
    // invalid operator (<), extraneous semicolon, undeclared variable (x)
    std::cout < "Hi there"; << x;
    return 0;
}
```

Syntax Error Example2

```
#include <iostream>
 int main()
{
    cout << "Geeks for geeks!"
    // missing semicolon at end of  statement
    return 0;
}
```

```
#include <iostream>
int main()
{
    int a = 10 ;
    int b = 0 ;
    std::cout << a << " / " << b << "
    = " << a / b;
    // division by 0 is undefined
    return 0;
}
```

Semantic Error Example1

Semantic Error Example2

```
#include <iostream>
int main()
{
	int a;
	a=10.5;
	cout << a << endl;
	return 0;
}
```

LEXICAL ANALYZER

Lexemes & Tokens - Example1

- Consider the following Java statement –

  index = 2 * count + 17 ;

Lexeme	Token
index	identifier
=	assign_op
2	int_literal
*	mult_op
count	identifier
+	plus_op
17	int_literal
;	semicolon

Lexemes & Tokens - Example2

- Consider the following C++ statement –

  E = M * C ** 2 ;

Lexeme	Token
E	identifier
=	assign_op
M	identifier
*	mult_op
C	identifier
**	exp_op
2	int_literal
;	semicolon

Lexemes & Tokens – Example3

- Consider the following C++ statement –

for (int count = 1; count <= 100; count++)

Lexeme	Token
for	for
(	LPAREN
int	reserved word
count	identifier
=	assign_op
1	int_literal
;	semicolon
count	identifier
<=	LE_comparison
100	int_literal
;	semicolon
count	identifier
++	inc_op
)	RPAREN

Lexemes & Tokens – Example4

- Consider the following C++ statement –

```
while ( i >= 0 ) {
    sum += i;
    i++;    }
```

Lexeme	Token
while	while
(	LPAREN
i	identifier
>=	GE_comparison
0	int_literal
)	RPAREN
{	LBRACE
sum	identifier
+	plus_op
=	assign_op
i	identifier
;	semicolon
i	identifier
++	inc_op
;	semicolon
}	RBRACE

Lexemes & Tokens – Example5

Lexeme	Token
if	if
(	LPAREN
y	identifier
<=	LE_comparison
t	identifier
)	RPAREN
y	identifier
=	assign_op
y	identifier
-	sub_op
3	int_literal
;	semicolon

- Consider the following C++ statement –
  - if (y <= t)
    - y = y - 3;

Lexemes & Tokens – Example6

- Consider the following C++ assignment statement –
  - result = oldsum - value / 100;

Lexeme	Token
result	identifier
=	assign_op
oldsum	identifier
-	sub_op
value	identifier
/	div_op
100	int_literal
;	semicolon

Lexemes & Tokens – Example7

- Consider the following C++ statement –
  - int a = 10;

Lexeme	Token
int	reserved word
a	identifier
=	assign_op
10	int_literal
;	semicolon

SYNTAX ANALYZER

Parser

- Goals of the parser, given an input program -
  - Find all syntax errors; for each, produce an appropriate diagnostic message, and recover.

- Example -
  - if (i > j // Error, unbalanced parentheses
  - max = i // Error, missing semicolon
  - Syntax error, insert ") Statement" to complete If Statement
  - Syntax error, insert ";" to complete Statement

Grammars

- In computer science and linguistics, a formal grammar, or sometimes simply grammar, is a precise description of a formal language — that is, of a set of strings.
- Commonly used to describe the syntax of programming languages.

Formal Grammars

- A formal grammar, or sometimes simply grammar, consists of -
  - A finite set of production rules with a left-hand side and a right-hand side consisting of a sequence of the above symbols.
  - A grammar is a finite nonempty set of rules.
  - A start symbol;
  - A finite set of terminal symbols;
  - A finite set of nonterminal symbols;
  - An abstraction (or nonterminal symbol) can have more than one R.H.S.

<stmt> → <single_stmt> | begin <stmt_list> end

Formal Grammars (Cont.)

CFG

- Grammar - a collection of rules
- Non-terminals – uppercase letters
- Terminals – lowercase letters
- Examples of CFG rules -
  - ID_LIST → identifier | identifier, ID_LIST
  - IF_STMT → **if** LOGIC_EXPR **then** STMT

BNF

- Grammar - a collection of rules
- Non-terminals - BNF abstractions
- Terminals - lexemes or tokens
- Examples of BNF rules -
  - <ident_list> → identifier | identifier, <ident_list>
  - <if_stmt> → **if** <logic_expr> **then** <stmt>

Describing Lists

- Syntactic lists are described using recursion

  <ident_list> → ident | ident, <ident_list>

- A derivation is a repeated application of rules, starting with the start symbol and ending with a sentence (all terminal symbols).

Example1

A Grammar for Simple Assignment Statements

<assign> → <id> = <expr>
<id> → A | B | C
<expr> → <id> + <expr>
| <id> * <expr>
| (<expr>)
| <id>

Example2

- Start symbol S,
- Terminals {a, b, ε},
- Non-terminals {S, A, B}, and
- No. of rules = 3

S → ABS

S → ε (where ε is the empty string)

A → AB

A → ab

A → aa

B → b

B → bb

Derivation

- To check whether a sequence of tokens is legal or not:
  - We start with a nonterminal called the **start** symbol
  - We apply productions, rewriting nonterminals, until only terminals remain
  - A **derivation** replaces a nonterminal on LHS of a production with RHS
  - The $\Rightarrow$ symbol denotes a derivation step

Derivation (Cont.)

Leftmost and Rightmost Derivations

- When deriving a sequence of tokens ...
  - More than one nonterminal may be present and can be expanded
  - A **leftmost derivation** chooses the leftmost nonterminal to expand
  - A leftmost derivation is denoted by $\Rightarrow_{lm}$
  - A **rightmost derivation** chooses the rightmost nonterminal to expand
  - A rightmost derivation is denoted by $\Rightarrow_{rm}$

Example1

- A leftmost derivation for `(id + num)*id`

$$
\begin{aligned}
expr &\Rightarrow_{lm} expr\ op\ expr \Rightarrow_{lm} (\ expr\ )\ op\ expr \Rightarrow_{lm} (\ expr\ op\ expr\ )\ op\ expr \\
&\Rightarrow_{lm} (\ \mathbf{id}\ op\ expr\ )\ op\ expr \Rightarrow_{lm} (\ \mathbf{id} + expr\ )\ op\ expr \\
&\Rightarrow_{lm} (\ \mathbf{id} + \mathbf{num}\ )\ op\ expr \Rightarrow_{lm} (\ \mathbf{id} + \mathbf{num}\ ) * expr \Rightarrow_{lm} (\ \mathbf{id} + \mathbf{num}\ ) * \mathbf{id}
\end{aligned}
$$

- A rightmost derivation for `(id + num)*id`

$$
\begin{aligned}
expr &\Rightarrow_{rm} expr\ op\ expr \Rightarrow_{rm} expr\ op\ \mathbf{id} \Rightarrow_{rm} expr * \mathbf{id} \Rightarrow_{rm} (\ expr\ ) * \mathbf{id} \\
&\Rightarrow_{rm} (\ expr\ op\ expr\ ) * \mathbf{id} \Rightarrow_{rm} (\ expr\ op\ \mathbf{num}\ ) * \mathbf{id} \\
&\Rightarrow_{rm} (\ expr + \mathbf{num}\ ) * \mathbf{id} \Rightarrow_{rm} (\ \mathbf{id} + \mathbf{num}\ ) * \mathbf{id}
\end{aligned}
$$

- Consider the following simplified grammar for expressions

$$
expr \to expr\ op\ expr \mid (\ expr\ ) \mid \mathbf{id} \mid \mathbf{num}
$$

$$
op \to + \mid - \mid * \mid /
$$

Example2

```
A = A * (B + (C * A))
```

```
<assign> → <id> = <expr>
<id> → A | B | C
<expr> → <id> + <expr>
       | <id> * <expr>
       | ( <expr> )
       | <id>
```

Example2 (Cont.)

A = A * (B + (C * A))

<assign> =>$_{lm}$ <id> = <expr>

=>$_{lm}$ A = <expr>

=>$_{lm}$ A = <id> * <expr>

=>$_{lm}$ A = A * <expr>

=>$_{lm}$ A = A * (< expr >)

=>$_{lm}$ A = A * (<id> + <expr>)

=>$_{lm}$ A = A * (B + <expr>)

=>$_{lm}$ A = A * (B + (<expr>))

=>$_{lm}$ A = A * (B + (<id> * <expr>))

=>$_{lm}$ A = A * (B + (C * <expr>))

=>$_{lm}$ A = A * (B + (C * <id>))

=>$_{lm}$ A = A * (B + (C * A))

<assign> → <id> = <expr>
<id> → A | B | C
<expr> → <id> + <expr>
| <id> * <expr>
| (<expr>)
| <id>

Example3

```
B = C * (A * C + B)
```

```
<assign> → <id> = <expr>
<id> → A | B | C
<expr> → <id> + <expr>
       | <id> * <expr>
       | ( <expr> )
       | <id>
```

Example3(Cont.)

B = C * (A * C + B)

<assign> =>$_{lm}$ <id> = <expr>

=>$_{lm}$ B = <expr>

=>$_{lm}$ B = <id> * <expr>

=>$_{lm}$ B = C * <expr>

=>$_{lm}$ B = C * (< expr >)

=>$_{lm}$ B = C * (<id> * <expr>)

=>$_{lm}$ B = C * (A * <expr>)

=>$_{lm}$ B = C * (A * <id> + <expr>)

=>$_{lm}$ B = C * (A * C + <expr>)

=>$_{lm}$ B = C * (A * C + <id>)

=>$_{lm}$ B = C * (A * C + B)

<assign> → <id> = <expr>
<id> → A | B | C
<expr> → <id> + <expr>
| <id> * <expr>
| (<expr>)
| <id>

Example4

```
A = A * (B + (C))
```

```
<assign> → <id> = <expr>
<id> → A | B | C
<expr> → <id> + <expr>
       | <id> * <expr>
       | ( <expr> )
       | <id>
```

Example4 (Cont.)

A = A * (B + (C))

<assign> $=>_{lm}$ <id> = <expr>
$=>_{lm}$ A = <expr>
$=>_{lm}$ A = <id> * <expr>
$=>_{lm}$ A = A * <expr>
$=>_{lm}$ A = A * (< expr >)
$=>_{lm}$ A = A * (<id> + <expr>)
$=>_{lm}$ A = A * (B + <expr>)
$=>_{lm}$ A = A * (B + (<expr>))
$=>_{lm}$ A = A * (B + (<id>))
$=>_{lm}$ A = A * (B + (C))

<assign> → <id> = <expr>
<id> → A | B | C
<expr> → <id> + <expr>
| <id> * <expr>
| (<expr>)
| <id>

Exercises

Problem1

Consider the following grammar:

<S> → <A> a <B> b

<A> → <A> b | b

<B> → a <B> | a

Which of the following sentences are in the language generated by this grammar?

a. baab
b. bbbab
c. bbaaaaa
d. bbaab

a and d

Exercises

Problem2

Consider the following grammar:

<S> → a <S> c <B> | <A> | b

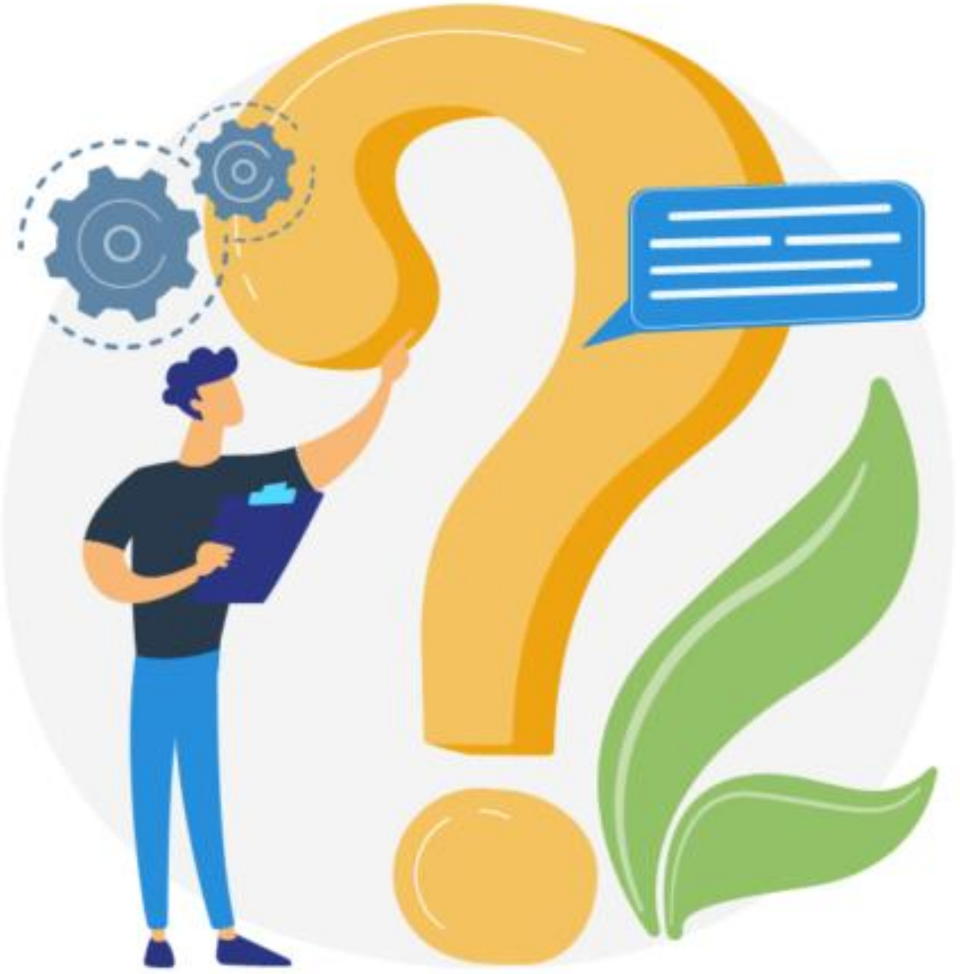
<A> → c <A> | c

<B> → d | <A>

Which of the following sentences are in the language generated by this grammar?

a. abcd
b. acccbd
c. acccbcc
d. acd
e. accc

a and e



Any question slide for ppt

Adapt it with your needs and it will capture all the audience attention.


THANK
YOU!