CS 315 – Programming Languages
Syntax

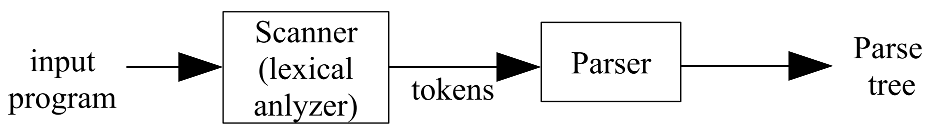
Programming languages must be <u>precise</u>

- Remember *instructions*
- This is unlike natural languages

Precision is required for

- syntax – think of this as the format of the language
- semantics – think of this as the meaning of the language

Recall the first part of the compilation sequence:



<u>Specification</u>	<u>Tool</u>	<u>Result</u>
Regular expressions	Scanner generator (lex) (Alternatively hand-built)	Scanner – Implements a DFA (deterministic finite automata)
Context-free grammar	Parser generator (yacc) (Alternatively hand-built)	Parser – Implements a PDA (push-down automata)

E.g. regular expressions:

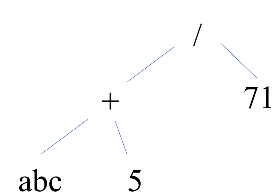
DIGIT:	[0-9]
<u>NUMBER</u>:	DIGIT+
CHARACTER:	[a-zA-Z]
<u>IDENTIFIER</u>:	CHARACTER (CHARACTER | DIGIT) *
<u>OP</u>:	('+' | '*' | '-' | '/')

Red characters are meta-characters. + means 1 or more, * (Kleene star) means 0 or more, | is used for alteration, etc. We will cover these in more detail later.

E.g. context-free grammar:

expression → expression OP expression
| '(' expression ')'
| IDENTIFIER
| NUMBER

(abc + 5) / 71



See code as trees, not as lines.

Specifying Syntax

Tokens (produced as a result of lexing / scanning, based on regular expressions)

- Concatenation ab
- Alternation a | b
- Kleene closure (repetition) a *

These rules are called **regular expressions**.
Set of strings that can be defined in terms of regular expressions are called **regular languages**.

Syntax tree (produced as a result of parsing, based on context-free grammars)

- Add recursion in addition to the above

These rules are called **context-free grammars**.
Set of strings that can be defined in terms of context-free grammars are called **context-free languages**.

Examples:

Ex1) The set of identifiers can be represented via regular expressions:

IDENTIFIER: [a-zA-Z][a-zA-Z0-9]*

Ex2) The set of palindromes cannot be represented via regular expressions. It requires recursion, thus context-free grammars are needed.

```
palindrome → '0' palindrome '0'
        |    '1' palindrome '1'
        |    '1'
        |    '0'
        |    ε
```

* zero or more

\+ one or more

? matches 0 or 1 of the preceding token.
making it an optional

Tokens & Regular Expressions

Token: basic building block
e.g. keywords, identifiers, symbols, constants
(while, if) (variable x) (+, -, *, /) (“abc”)

Regular expressions are used to specify tokens:

- a character
- the empty string, denoted by ε
- two regular expressions next to each other (concatenation)
- two regular expressions separated by | (alternation)
- a regular expression followed by a Kleene star (repetition)
- parentheses used to avoid ambiguity

E.g.:

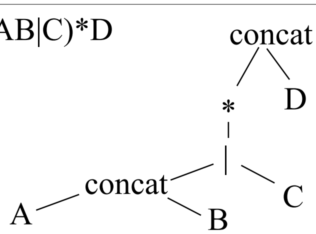
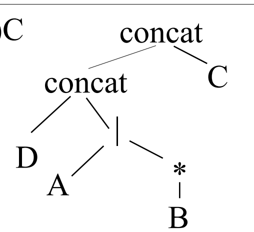
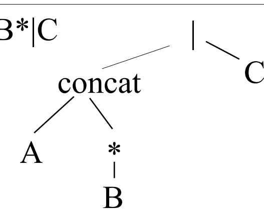
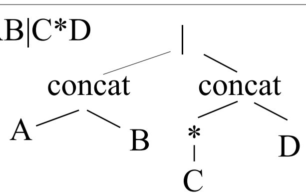
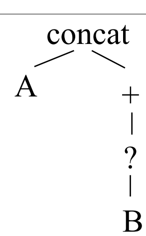
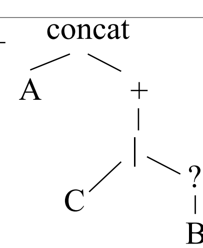
DIGIT: 0 | 1 | 2 | … | 9 – [0-9]
INTEGER: DIGIT DIGIT* – DIGIT+
DECIMAL: DIGIT* ('.' DIGIT | DIGIT '.') DIGIT*
EXPONENT: (e | E) ('+' | '-' | ε) INTEGER – (e | E) ('+' | '-')? INTEGER
REAL: INTEGER EXPONENT
| DECIMAL (EXPONENT | ε) – DECIMAL EXPONENT?
NUMBER: INTEGER | REAL

Precedence of regular expression operators:

Highest	()
	*, +, ?
	concatenation
Lowest	|

Think of these as trees as well. Highest precedence operator is at the bottom, lowest is at the top.

Exercises:

(AB|C)*D 	D(A|B*)C 	AB*|C 
AB|C*D 	AB?+ 	A(C|B?)+ 

<table>
<tr>
<td>(AB|C)*D
Valid:
• ABD
• D
• CABCD
• ABABD
Invalid:
• ACD</td>
<td>AB|C*D
Valid:
• AB
• D
• CD
• CCD
Invalid:
• ABD</td>
</tr>
<tr>
<td>D(A|B*)C
Valid:
• DAC
• DC
• DBC
• DBBC
Invalid:
• DABC</td>
<td>AB*|C
Valid:
• C
• A
• AB
• ABB
Invalid:
• ABC</td>
</tr>
<tr>
<td>AB?+
Valid:
• A
• AB
• ABB</td>
<td>A(C|B?)+
Valid:
• AC
• ACC
• A
• ABC
• ABBCBCC</td>
</tr>
</table>

<u>Context-free Grammars</u>

Regular expressions cannot be used to describe PL structures such expressions. Instead, we use context-free grammars, that is CFGs. They are described using the BNF (Backus Normal Form) notation.

expr → ID
    | NUMBER   *token* also *terminal*
    | '-' expr
    | '(' expr ')'   *terminal*
    | expr op expr

non-terminal also *start symbol*

op → '+' | '-' | '*' | '/'

non-terminal

- only non-terminals can appear as LHS (left-hand side)
- LHS only has a single non-terminal (otherwise it becomes context sensitive)
- non-terminals and terminals can appear on right-hand side (RHS)
- concatenation is allowed on RHS
- parenthesis and Kleene closure operators are not allowed on RHS in BNF

There is also the extended BNF notation, called the EBNF. EBNF removes the last restriction: parenthesis and Kleene closure operators are allowed in EBNF. The grammar

id_list → ID (',' ID)*

is in EBNF and is equivalent to the following BNF grammar:

id_list → ID
    | id_list ',' ID

Also, the following forms are all the same:

op → '+' | '-' | '*' | '/'

op → '+'
op → '-'
op → '*'
op → '/'

op → '+'
   → '-'
   → '*'
   → '/'

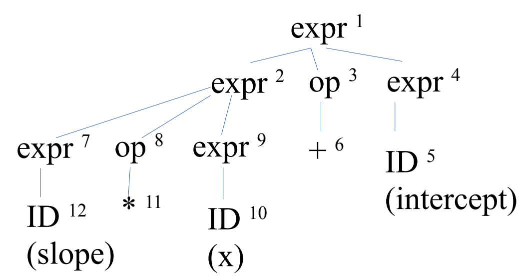
Derivations and parse-trees

Context-free grammars describe how to generate synthetically valid strings. E.g.: Consider the following input string:

slope * x + intercept

And the following grammar:

1. expr → ID
2. | NUMBER
3. | '-' expr
4. | '(' expr ')'
5. | expr op expr
6. op → '+'
7. | '-'
8. | '*'
9. | '/'



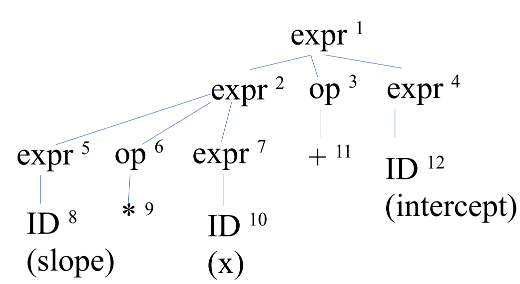
Here is a **rightmost derivation** of the input string using the above grammar:

expr	apply rule 5
=> expr op expr	apply rule 1
=> expr op ID	apply rule 6
=> expr '+' ID	apply rule 5
=> expr op expr '+' ID	apply rule 1
=> expr op ID '+' ID	apply rule 8
=> expr '*' ID '+' ID	apply rule 1
=> ID (slope) '*' ID (x) '+' ID (intercept)	

In a rightmost derivation, we always expand the rightmost non-terminal.

Here is a **leftmost derivation** of the input string using the above grammar:

expr	apply rule 5
=> expr op expr	apply rule 5
=> expr op expr op expr	apply rule 1
=> ID op expr op expr	apply rule 8
=> ID '*' expr op expr	apply rule 1
=> ID '*' ID op expr	apply rule 6
=> ID '*' ID '+' expr	apply rule 1
=> ID (slope) '*' ID (x) '+' ID (intercept)	

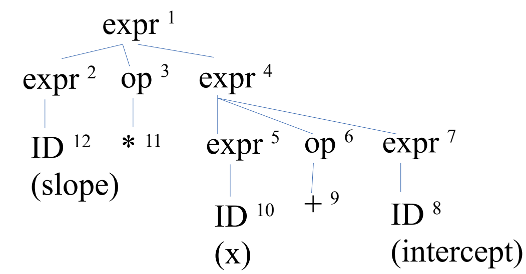


In a leftmost derivation, we always expand the leftmost non-terminal.

Note that these two trees are the same, structurally, but created in different orders. However, for this particular grammar, we can come up with a derivation that results in a different tree.

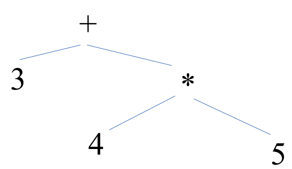
Here is another rightmost derivation that produces a different tree:

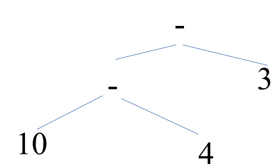
expr	apply rule 5
=> expr op expr	apply rule 5
=> expr op expr op expr	apply rule 1
=> expr op expr op ID	apply rule 6
=> expr op expr '+' ID	apply rule 1
=> expr op ID '+' ID	apply rule 8
=> expr '*' ID '+' ID	apply rule 1
=> ID (slope) '*' ID (x) '+' ID (intercept)	



If a grammar results in derivations with different parse trees, then the grammar is **ambiguous**.
Ambiguous grammars create problems in parsing, since different parsing trees result in different semantics. For instance:

- *precedence*: 3+4*5 means 3+(4*5), since * has higher precedence than +
  - Thus, the multiplication should be **deeper** in the parse tree
- *associativity*: 10-4-3 means (10-4)-3, since – is left-associative
  - Thus, the first subtraction should be **deeper** in the parse tree





An unambiguous grammar for expressions

Note: You should know this by heart.

1. expr → expr add_op term
2. | term

3. term → term mult_op factor
4. | factor

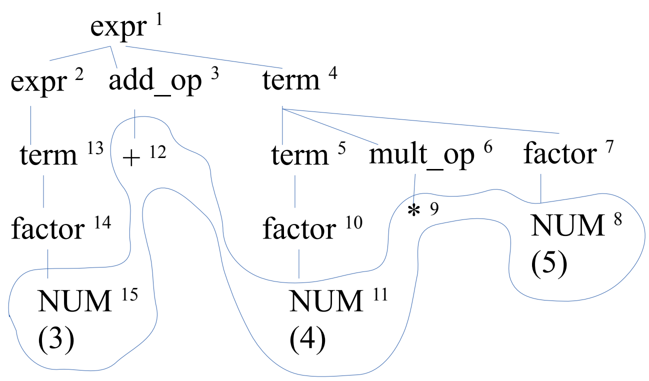
5. factor → '-' factor
6. | '(' expr ')'
7. | ID
8. | NUM

9. add_op → '+'
10. | '-'

11. mul_op → '*'
12. | '/'

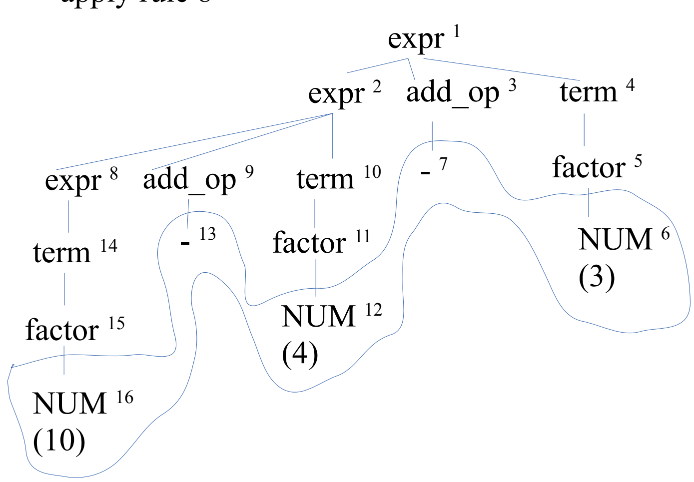
Let's do a RMD: 3 + 4 * 5

expr	apply rule 1
=> expr add_op term	apply rule 3
=> expr add_op term mult_op factor	apply rule 8
=> expr add_op term mult_op NUM	apply rule 11
=> expr add_op term '*' NUM	apply rule 4
=> expr add_op factor '*' NUM	apply rule 8
=> expr add_op NUM '*' NUM	apply rule 9
=> expr '+' NUM '*' NUM	apply rule 2
=> term '+' NUM '*' NUM	apply rule 4
=> factor '+' NUM '*' NUM	apply rule 8
=> NUM (3) '+' NUM (4) '*' NUM (5)	



Let's do a RMD: 10 – 4 – 3

expr	apply rule 1
=> expr add_op term	apply rule 4
=> expr add_op factor	apply rule 8
=> expr add_op NUM	apply rule 10
=> expr '-' NUM	apply rule 1
=> expr add_op term '-' NUM	apply rule 4
=> expr add_op factor '-' NUM	apply rule 8
=> expr add_op NUM '-' NUM	apply rule 10
=> expr '-' NUM '-' NUM	apply rule 2
=> term '-' NUM '-' NUM	apply rule 4
=> factor '-' NUM '-' NUM	apply rule 8
=> NUM (10) '-' NUM (4) '-' NUM (3)	



An unambiguous grammar always results in the same tree, irrespective of the derivation performed. Different derivations may result in different construction orders of the trees, but the structure of the parse tree will always be the same.

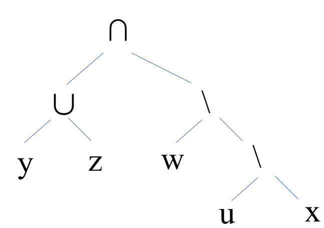
There are few details to note about the grammar we used for arirhmetic expressions:

- Left recursion results in left associativity (if we were to use right recursion, we would have had right associativity)
- Order of the rules correspond to the precedence of the operators. In particular, lower precedence operators appear earlier in the rule list.

Example: Let's say we want to create an expression language that has the following operators:

	∪	∩	\
precedence:	low	medium	high
associativity:	left	left	right

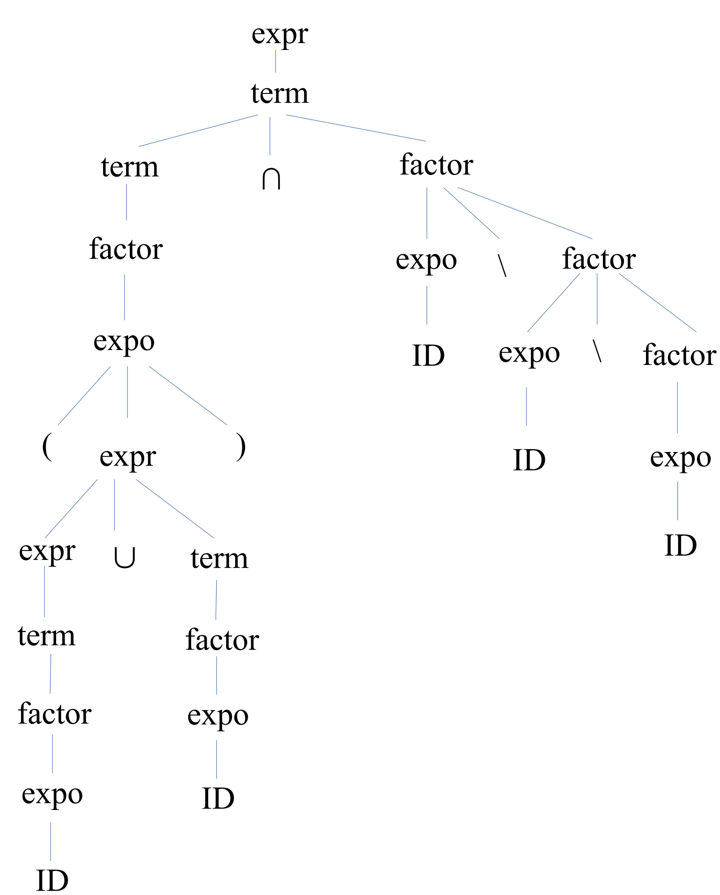
Grammar:



expr → expr '∪' term
        | term
term → term '∩' factor
        | factor
factor → expo '\' factor
        | expo
expo → '(' expr ')'
        | ID
        | NUM

Example input: (y ∪ z) ∩ w \ u \ x

```
expr
  => term
  => term '∩' factor
  => term '∩'  expo '\' factor
  => term '∩'  expo '\' expo '\' factor
  => term '∩'  expo '\' expo '\' expo
  => term '∩'  expo '\' expo '\' ID
  => term '∩'  expo '\' ID '\' ID
  => term '∩'  ID '\' ID '\' ID
  => factor '∩'  ID '\' ID '\' ID
  => expo '∩'  ID '\' ID '\' ID
  => '(' expr ')' '∩'  ID '\' ID '\' ID
  => '(' expr '∪' term ')' '∩'  ID '\' ID '\' ID
  => '(' expr '∪' factor ')' '∩'  ID '\' ID '\' ID
  => '(' expr '∪' expo ')' '∩'  ID '\' ID '\' ID
  => '(' expr '∪' ID ')' '∩'  ID '\' ID '\' ID
  => '(' term '∪' ID ')' '∩'  ID '\' ID '\' ID
  => '(' factor '∪' ID ')' '∩'  ID '\' ID '\' ID
  => '(' expo '∪' ID ')' '∩'  ID '\' ID '\' ID
  => '(' ID (y) '∪' ID (z) ')' '∩'  ID (w) '\' ID (u) '\' ID (x)
```



Consider the following grammar for if/else statements:

stmt → if_stmt | nonif_stmt
if_stmt → IF '(' logical_expr ')' THEN stmt
| IF '(' logical_expr ')' THEN stmt ELSE stmt
nonif_stmt → …

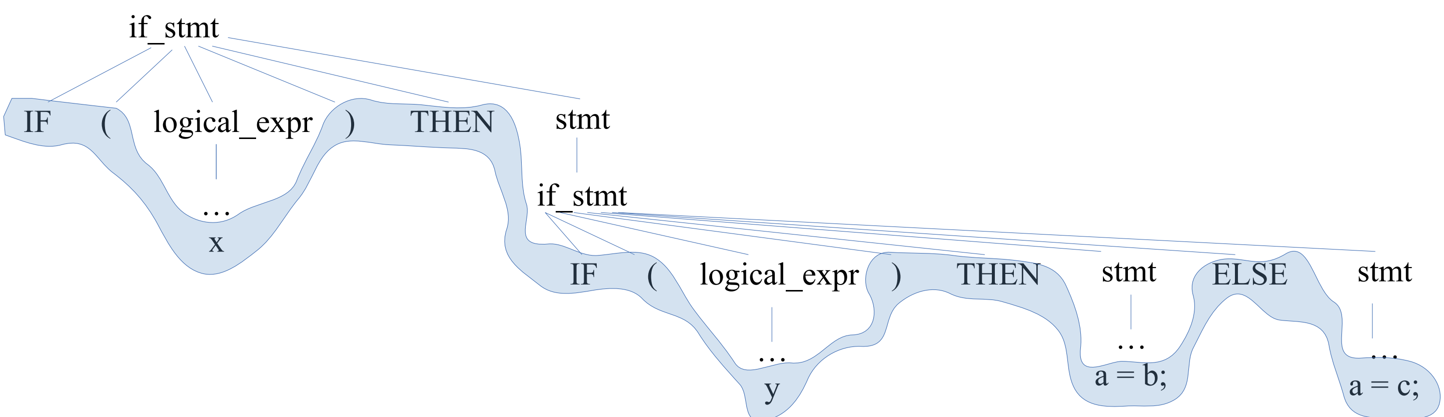
Is this grammar unambiguous? We can prove otherwise via an example:

Consider the following input:

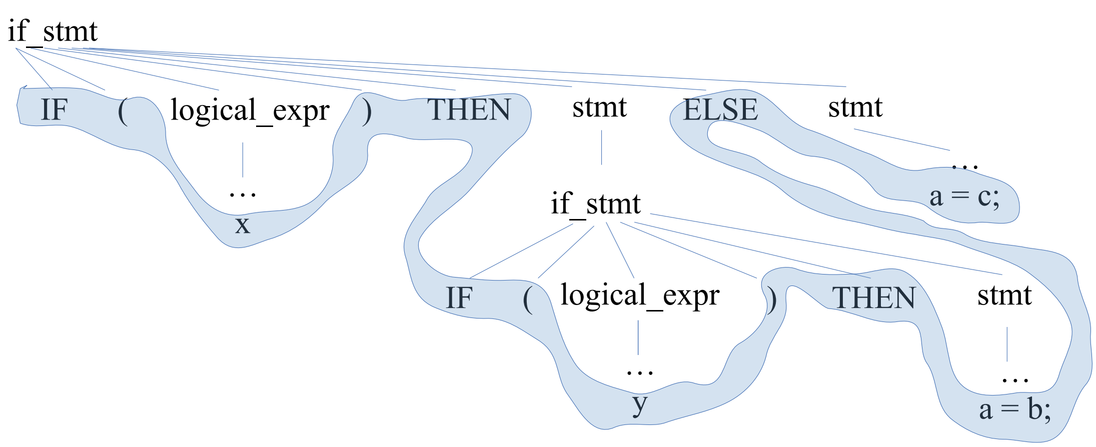
```
if (x) then
    if (y) then
        a = b;
    else
        a = c;
```

We can create two different parse trees:

Alternative 1:



Alternative 2:



The else belongs to the closest dangling if, so the first one is the one we should ideally have.

The following is an **unambiguous** grammar for the same problem:

stmt → if_stmt | nonif_stmt
if_stmt → matched_if | unmatched_if
matched_if → 'if' logical_expr 'then' matched_stmt 'else' matched_stmt
matched_stmt → mathced_if | nonif_stmt
unmatched_if → 'if' logical_expr 'then' stmt
| 'if' logical_expr 'then' matched_stmt 'else' unmatched_if
logical_expr → id '==' lit
nonif_stmt → assgn_stmt
assgn_stmt → id '=' expr
expr → expr '+' term | term
term → '(' expr ')' | id
id → 'A' | 'B' | 'C'
lit → '0' | '1' | '2'

Consider the following input:

```
if A == 0 then
        if  B == 1 then
                C = A + B
        else
                B = C
```

Let us do a leftmost derivation for the input:

stmt
=> if_stmt
=> unmatched_if
=> 'if' logical_expr 'then' stmt
=> 'if' id '==' lit 'then' stmt
=> 'if' 'A' '==' lit 'then' stmt
=> 'if' 'A' '==' '0' 'then' stmt
=> 'if' 'A' '==' '0' 'then' if_stmt
=> 'if' 'A' '==' '0' 'then' matched_if
=> 'if' 'A' '==' '0' 'then' 'if' logical_expr 'then' matched_stmt 'else' matched_stmt
=> 'if' 'A' '==' '0' 'then' 'if' id '==' lit 'then' matched_stmt 'else' matched_stmt
=> 'if' 'A' '==' '0' 'then' 'if' 'B' '==' lit 'then' matched_stmt 'else' matched_stmt
=> 'if' 'A' '==' '0' 'then' 'if' 'B' '==' '1' 'then' matched_stmt 'else' matched_stmt
=> 'if' 'A' '==' '0' 'then' 'if' 'B' '==' '1' 'then' nonif_stmt 'else' matched_stmt
=> 'if' 'A' '==' '0' 'then' 'if' 'B' '==' '1' 'then' assgn_stmt 'else' matched_stmt
=> 'if' 'A' '==' '0' 'then' 'if' 'B' '==' '1' 'then' id '=' expr 'else' matched_stmt
=> 'if' 'A' '==' '0' 'then' 'if' 'B' '==' '1' 'then' 'C' '=' expr '+' term 'else' matched_stmt
=> 'if' 'A' '==' '0' 'then' 'if' 'B' '==' '1' 'then' 'C' '=' term '+' term 'else' matched_stmt
=> 'if' 'A' '==' '0' 'then' 'if' 'B' '==' '1' 'then' 'C' '=' id '+' term 'else' matched_stmt
=> 'if' 'A' '==' '0' 'then' 'if' 'B' '==' '1' 'then' 'C' '=' 'A' + term 'else' matched_stmt
=> 'if' 'A' '==' '0' 'then' 'if' 'B' '==' '1' 'then' 'C' '=' 'A' + 'B' 'else' matched_stmt
=> 'if' 'A' '==' '0' 'then' 'if' 'B' '==' '1' 'then' 'C' '=' 'A' + 'B' 'else' nonif_stmt
=> 'if' 'A' '==' '0' 'then' 'if' 'B' '==' '1' 'then' 'C' '=' 'A' + 'B' 'else' assgn_stmt
=> 'if' 'A' '==' '0' 'then' 'if' 'B' '==' '1' 'then' 'C' '=' 'A' + 'B' 'else' id '=' expr
=> 'if' 'A' '==' '0' 'then' 'if' 'B' '==' '1' 'then' 'C' '=' 'A' + 'B' 'else' 'B' '=' expr
=> 'if' 'A' '==' '0' 'then' 'if' 'B' '==' '1' 'then' 'C' '=' 'A' + 'B' 'else' 'B' '=' term
=> 'if' 'A' '==' '0' 'then' 'if' 'B' '==' '1' 'then' 'C' '=' 'A' + 'B' 'else' 'B' '=' id
=> 'if' 'A' '==' '0' 'then' 'if' 'B' '==' '1' 'then' 'C' '=' 'A' + 'B' 'else' 'B' '=' 'C'