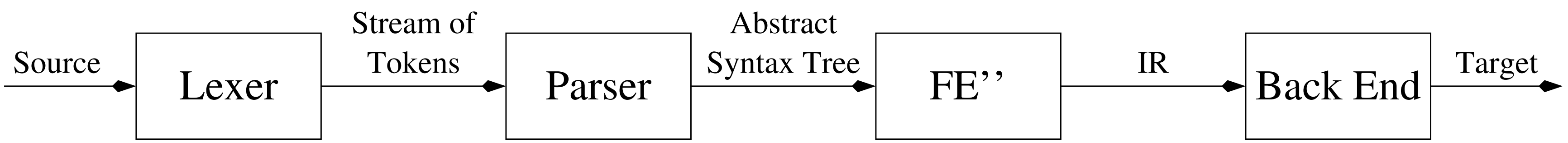
Syntax Analysis

Front End:

- Lexical Analysis - Break source into *tokens*.
- Syntax Analysis - Parse phrase structure.
- Semantic Analysis - Calculate meaning.

Our Compiler:

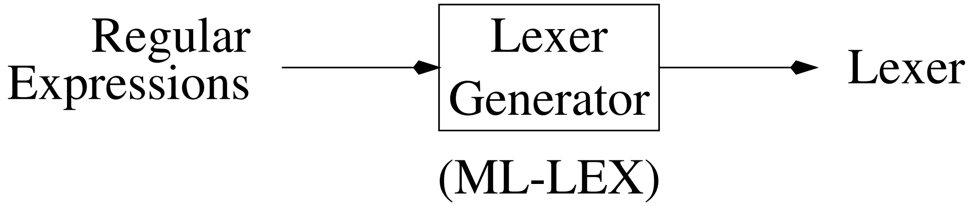


Parser Functions:

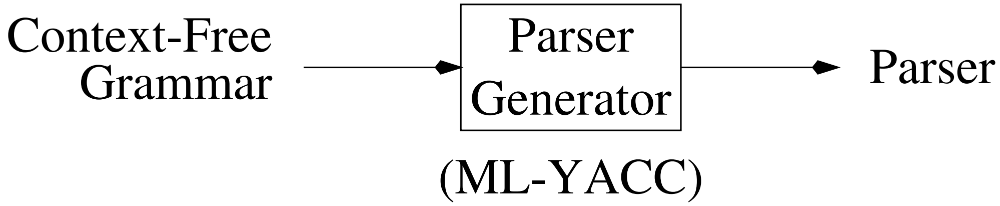
- Verify that token stream is valid.
- If it is not valid, report syntax error and recover.
- Build Abstract Syntax Tree (AST).

Syntactic Analysis

Regular Expressions - describe legal sequences of characters in a token.



Context-Free Grammars - describe legal sequences of tokens in a program.



Definitions

- *Language* - set of strings
- *String* - finite sequence of *symbols* taken from finite *alphabet*
  - Regular Expressions describe a language.
  - Context-Free Grammar also describes a language.

	Lexical Analysis	Syntax Analysis
language	set of tokens	set of source programs
string	token	source program
symbol	ASCII character	token

Context-Free Grammars

- Given by BNF expressions (Backus-Naur Form).
- Context-free grammars are more powerful than regular expressions.
  - Any language that can be generated using regular expressions can be generated by a context-free grammar.
  - There are languages that can be generated by a context-free grammar that cannot be generated by any regular expression.
- Examples:
  - Matching parentheses
  - Nested comments

Context-Free Grammars

- Context-Free Grammars consist of a set of *productions*.

$$symbol \;\rightarrow\; symbol\ symbol\ ...\ symbol$$

- Symbol types:
  - *terminal* that corresponds to a token-type.
  - *non-terminal* that denotes a set of strings.
- Left-Hand Side (LHS) - *non-terminal*.
- Right-Hand Side (RHS) - *terminals* or *non-terminals*
- *Start Symbol* - A special *non-terminal*.
- Each production specifies how terminals and non-terminals may be combined to form a substring in language.
- Easy to specify recursion:

$$stmt \;\rightarrow\; IF\ exp\ THEN\ stmt\ ELSE\ stmt$$

Start Symbol

- String of token-types is in language described by grammar if it can be derived from a given *start symbol*
- Derivations:
  1. begin with start symbol
  2. while non-terminals exist, replace any non-terminal with RHS of production

Example

Non-Terminals:

```
stmt        (Start Symbol)
expr
expr_list
```

Terminals (tokens):

```
SEMI ";"
ID
ASSIGN ":="
LPAREN "("
RPAREN ")"
NUM
PLUS  "+"
PRINT "print"
COMMA ","
```

$$
\begin{aligned}
stmt &\rightarrow stmt; stmt \\
stmt &\rightarrow ID \ := \ expr \\
stmt &\rightarrow PRINT \, (\, expr\_list \,)
\end{aligned}
$$

$$
\begin{aligned}
expr &\rightarrow ID \\
expr &\rightarrow NUM \\
expr &\rightarrow expr + expr \\
expr &\rightarrow (\, stmt, \ expr \,)
\end{aligned}
$$

$$
\begin{aligned}
expr\_list &\rightarrow expr \\
expr\_list &\rightarrow expr\_list, \ expr
\end{aligned}
$$

Example Derivation

Show that the following expression can be derived from start symbol.

```
ID := NUM; PRINT(NUM)
```

Another Derivation

The previous derivation is called *left-most* because the left-most non-terminal is always expanded first. We can also find a right-most derivation.

```
ID := NUM; PRINT(NUM)
```

Left-most derivation:

$stmt$

$stmt; stmt$

$ID := expr; stmt$

$ID := NUM; stmt$

$ID := NUM; PRINT\ (\ expr\_list\ )$

$ID := NUM; PRINT\ (\ expr\ )$

$ID := NUM; PRINT\ (\ NUM\ )$

Right-most derivation:

$stmt$

$stmt; stmt$

$stmt; stmt$

$stmt; PRINT\ (\ expr\_list\ )$

$stmt; PRINT\ (\ expr\ )$

$stmt; PRINT\ (\ NUM\ )$

$ID := expr; PRINT\ (\ NUM\ )$

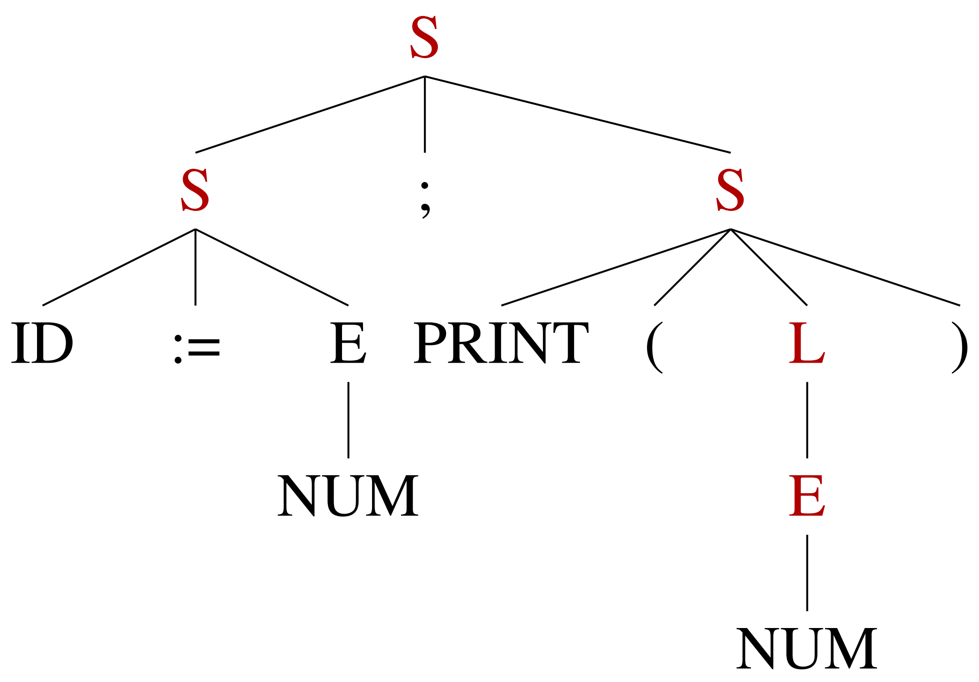
$ID := NUM; PRINT\ (\ NUM\ )$

Parse Trees

- *Parse Trees* - Graphical representation of derivation.
- Each internal node is labeled with a non-terminal.
- Each leaf node is labeled with a terminal.
- Parse Tree of the example:

Parse Trees

- *Parse Trees* - Graphical representation of derivation.
- Each internal node is labeled with a non-terminal.
- Each leaf node is labeled with a terminal.
- Parse Tree of the example:



Ambiguous Grammars

A grammar is ambiguous if it can derive a string of tokens with two or more different parse trees.

Non-Terminals:

```
expr
```

Terminals (tokens):

```
ID
NUM
PLUS   "+"
MULT   "*"
```

$$
\begin{aligned}
expr &\rightarrow ID \\
expr &\rightarrow NUM \\
expr &\rightarrow expr + expr \\
expr &\rightarrow expr * expr
\end{aligned}
$$

Consider: $4 + 5 * 6$

`NUM(4) PLUS NUM(5) MULT NUM(6)`

Ambiguous Grammars

A grammar is ambiguous if it can derive a string of tokens with two or more different parse trees.

Non-Terminals:

```
expr
```

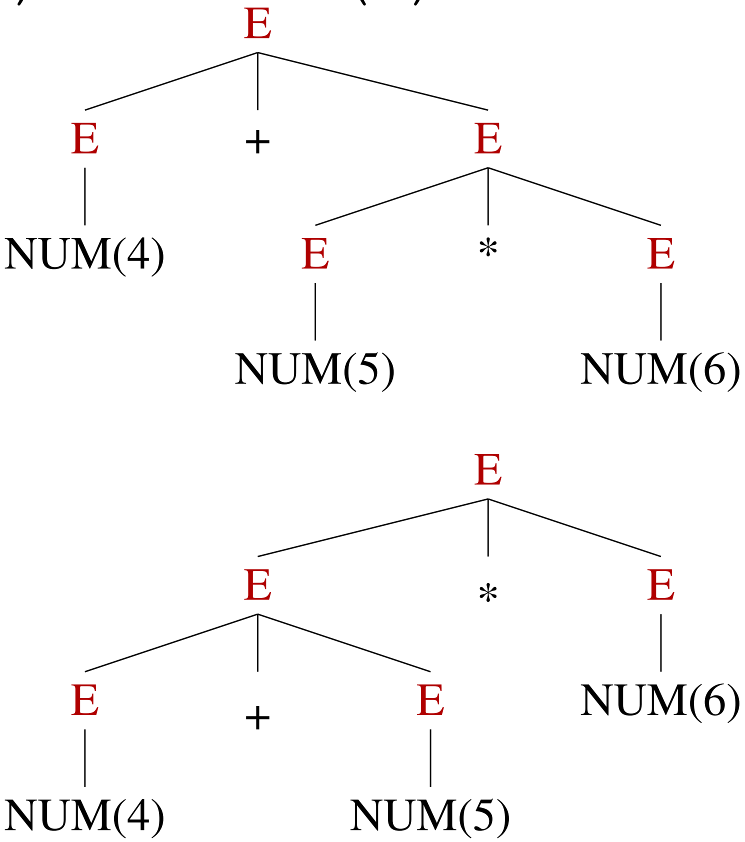
Terminals (tokens):

```
ID
NUM
PLUS   "+"
MULT   "*"
```

$$
\begin{aligned}
expr &\rightarrow ID \\
expr &\rightarrow NUM \\
expr &\rightarrow expr + expr \\
expr &\rightarrow expr * expr
\end{aligned}
$$

Consider: $4 + 5 * 6$

`NUM(4) PLUS NUM(5) MULT NUM(6)`



Ambiguous Grammars

- *Problem*: compilers use parse trees to interpret meaning of parsed expressions.
  - Different parse trees may have different meanings, resulting in different interpreted results.
  - For example, does $4 + 5 * 6$ equal 34 or 54?
  - Languages with ambiguous grammars are DISASTROUS! Programmers can't understand what their programs mean!
- *Solution*: rewrite grammar to eliminate ambiguity.
  - Operators have a relative *precedence*. We say some operands *bind tighter* than others. ("*" binds tighter than "+")
  - Operators with the same precedence must be resolved by *associativity*. Some operators have *left associativity*, others have *right associativity*.

Ambiguous Grammars

Non-Terminals:

```
expr  (an Expression)
term  (a Term
       can be added)
fact  (a Factor
       can be mult'd)
```

Terminals (tokens):

$expr \rightarrow expr + term$

$expr \rightarrow term$

$term \rightarrow term * fact$

$term \rightarrow fact$

$fact \rightarrow ID$

$fact \rightarrow NUM$

$4 + 5 * 6$

`NUM(4) PLUS NUM(5) MULT NUM(6)`

Ambiguous Grammars

Non-Terminals:

```
expr  (an Expression)
term  (a Term
       can be added)
fact  (a Factor
       can be mult'd)
```

Terminals (tokens):

$expr \rightarrow expr + term$

$expr \rightarrow term$

$term \rightarrow term * fact$

$term \rightarrow fact$

$fact \rightarrow ID$

$fact \rightarrow NUM$

$fact \rightarrow (expr)$

$(4 + 5) * 6$

```
LPAREN NUM(4) PLUS NUM(5)
RPAREN MULT NUM(6)
```

End-of-File Marker

- Parse must also recognize the End-of-File (EOF).
- EOF marker in the grammar is “$”
- Introduce new start symbol and the production $E' \rightarrow\ E\$$

Grammars and Lexical Analysis

- Grammars can also describe token structure:

  ```
  ( a | b )* abb
  ```

  $W \rightarrow aW$
  $W \rightarrow bW$
  $W \rightarrow aX$
  $X \rightarrow bY$
  $Y \rightarrow bZ$
  $Z \rightarrow \epsilon$

- Can combine lexical analysis and syntax analysis into one module.
- Disadvantages:
  - Regular expression specification is more concise.
  - Separating phases increases compiler modularity.

Context-Free Grammar and REs

- Context-free grammars are more powerful than regular expressions.
  - Any language that can be generated using regular expressions can be generated by a context-free grammar.
  - There are languages that can be generated by a context-free grammar that cannot be generated by any regular expression.
- Since it turns out that DFAs and NFAs are equivalent to REs, CFGs are strictly more powerful than the whole lot!
- The proof is in two parts:
  - Given a regular expression R , we can generate a CFG G such that L(R) == L(G).
  - We can define a grammar G for which there there is no FA F such that L(F) == L(G).

We will prove the first part rigorously. I'll just give you the intuition as to why the second part is true (take a complexity class if you are interested!)

Context-Free Grammars and REs

We will define a translation from REs to CFGs *by induction on the structure of REs*. That means:

- There will be one case for each kind of RE.
- The CFGs for complex REs are built by first recursively constructing the CFGs for their simpler parts. For example, to build the CFG for the regular expression $r_1\ r_2$, we first build the CFG for r_1 and for r_2 and then use those regular expressions to build the CFG for $r_1|r_2$.

The translation will have the form:
$\mathsf{trans}(r, S) = CFG$
and it builds a CFG that produces the same language (set of strings) when S is your start symbol.

Context-Free Grammar and REs

Base Cases:

- Simple Symbols: $\mathsf{trans}(a, S) =$
  $S \rightarrow \ a$
- Empty string: $\mathsf{trans}(\epsilon, S) =$
  $S \rightarrow \ \epsilon$

Context-Free Grammar and REs

Inductive Cases:

- Alternation: $\mathsf{trans}(r_1|r_2, S) =$
  $S \rightarrow\ S_1$
  $S \rightarrow\ S_2$
  $\mathsf{trans}(r_1, S_1)$
  $\mathsf{trans}(r_2, S_2)$
  (S_1 and S_2 are *fresh* grammar variables – unused anywhere else)

- Concatenation: $\mathsf{trans}(r_1\ r_2, S) =$
  $S \rightarrow\ S_1\ S_2$
  $\mathsf{trans}(r_1, S_1)$
  $\mathsf{trans}(r_2, S_2)$
  (S_1 and S_2 are fresh variables)

- Kleene closure: $\mathsf{trans}(r_1*, S) =$
  $S \rightarrow\ \epsilon$
  $S \rightarrow\ S_1\ S$
  $\mathsf{trans}(r_1, S_1)$
  (S_1 is *fresh*)

Context-Free Grammar and REs

Theorem: **For all regular expressions** r, the language of r is the same as the language of $\mathsf{trans}(r, S)$ (when S is used as the starting symbol).

How do we prove something for ALL regular expressions rather than just for a few examples?

Induction on the Structure of Regular Expressions

Proofs by induction on the structure of regular expressions have 5 cases:

- Base cases (2): Prove the theorem holds for simple symbols a and the empty expression ϵ without any extra assumptions.
- Inductive cases (3): The cases of the proof for complex regular expressions depend upon the theorem holding for its smaller subexpressions:
  - case $r_1|r_2$: we assume that the theorem holds for r_1 and r_2 and on the basis of that assumption we show that the theorem holds for $r_1|r_2$.
  - case $r_1\ r_2$: we assume that the theorem holds for r_1 and r_2 and on the basis of that assumption we show that it holds for $r_1\ r_2$.
  - case $r*$: we assume that the theorem holds for r and on the basis of that assumption we show that it holds for $r*$.

After proving our result for just these general five cases, we know that our theorem holds for ALL regular expressions.

Context-Free Grammar with no RE/FA

Informally, we sketch why CFGs are strictly more expressive than FAs (which are equivalent to REs)

$$\begin{aligned} S &\rightarrow (S) \\ S &\rightarrow \epsilon \end{aligned}$$

- FAs have a FINITE number of states, N
- FA must "remember" number of "("s, to generate ")"s
- At or before $N + 1$ "("s FA will revisit a state.
- That state represents two different counts of ")"s.
- Both counts must now be accepted.
- One count will be invalid.