Parsing

Prof Rajeev Barua
CS 3101 -- Slide Set 5

Are regular expressions enough to recognize full programming languages?

Regular expressions are powerful enough to recognize tokens.

- But are they powerful enough to recognize full programming languages?
  - No, but they can recognize parts of programming languages.

Example of a language subset recognized by regular expressions:

Simple expressions of numbers and operators:

Consider the following regular expressions:

```
digits = [0-9]+
sum = (digits "+")* digits
```

The second rule above is not a regular expression, but it can be expanded into one by substitution. (trivially):

```
sum = ([0-9]+ "+")* [0-9]+
```

Example when regular expressions are not enough

Example of a language subset ***not*** recognized by regular expressions:

Parenthesized expressions:

> **E.g:** ((5 + (62 + 7)) + 83). These are called balanced parenthesized expressions if each open parenthesis is balanced by a later closed parenthesis.

Unfortunately, balanced parenthesized expressions cannot be recognized with regular expressions. At first a simple recursive definition seems like it might help:

```
digits = [0-9]+
sum = expr "+" expr
expr = "(" sum ")" |digits
```

Unfortunately, **`expr`** cannot be expanded into a regular expression. If we try, we end up with a recursive definition for **expr** (similar problem for **`sum`** as well.)

Why regular expressions are not enough

Intuition for why no regular expression can recognize parenthesized expressions:

Intuitively, the problem is that a finite automaton with N states cannot remember more than N parenthesis.

- Recall that FAs have a 1-to-1 correspondence with regular expressions.
- Since we need to recognize an unbounded number of parenthesis, we cannot use any finite regular expression.

Instead we need a theoretical formulation that allows recursion! (The above wished language gives us a hint).

This is called a *context-free grammar* (CFG). To see the simplest formulation of CFGs, first we will eliminate | and *:

How to know if a regular expression is possible

In general, for a given language, a regular expression is possible, if it can be recognized by remembering which one of a finite (bounded) number of states we are currently in.

- This is because every regular expression can be recognized by at least one Deterministic Finite Automaton (DFA). A DFA has a finite number of states.

Example when regular expression is possible:
We want to derive a regular expression that accepts binary numbers, composed of 0s and 1s, where the number of 1s is a multiple of three. This is possible. Although the number of 1s can be infinite, we don't need to remember that. Instead, we only need to remember what the remainder of the number is, when divided
Example when regular expression is NOT possible:
We want to derive a regular expression that accepts binary numbers, composed of 0s and 1s, where the number of 111 patterns exceeds the number of 000 patterns. This is NOT possible. To recognize such a language, we need to remember the following quantity = (number of 111 patterns seen so far - number of 000 patterns seen so far.) Since this number can be unboundedly large, no DFA exists that can remember this quantity.

Converting regular expressions to CFGs

Eliminating alternation:

```
    expr = ab(c|d)e
```

can be rewritten as:

```
    aux = c
    aux = d
    expr = a b aux e
```

Eliminating closure: (*)

```
    expr = (a b c)*
```

can be rewritten as:

```
    expr = a b c expr
    expr = ϵ
```

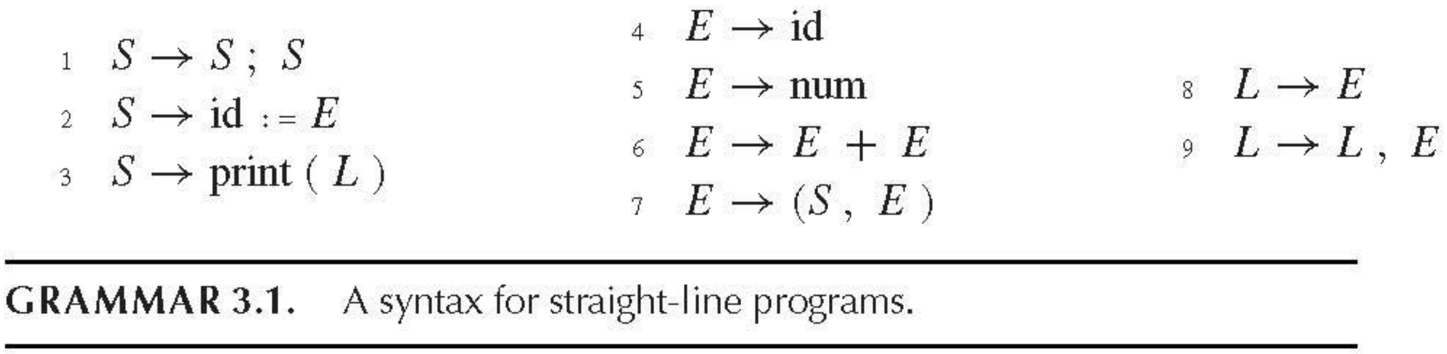
What we are left with is a CFG form.

Context-Free Grammars (CFGs)

We can formalize the above notion into a mathematical formulation called CFGs:

- Intermediate symbols like **`expr, sum`** are called NON-TERMINALS.
- All symbols that actually appear in the language strings are called TERMINALS.
- PRODUCTIONS are rules that define non-terminals in terms of terminals and non-terminals (recursion is allowed.)

Example grammar:



1 $S \rightarrow S \,;\, S$
2 $S \rightarrow \text{id} := E$
3 $S \rightarrow \text{print} \,(\, L \,)$
4 $E \rightarrow \text{id}$
5 $E \rightarrow \text{num}$
6 $E \rightarrow E + E$
7 $E \rightarrow (S \,,\, E\,)$
8 $L \rightarrow E$
9 $L \rightarrow L \,,\, E$

GRAMMAR 3.1. A syntax for straight-line programs.

Above is a grammar for straight-line programs of expressions.

- **`S, E, L`** are non-terminals for Statement, Expressions, Lists of Expressions.
- **`id, print, num , +, ( , ), ,:=, ;`** are terminal symbols (recognized as tokens in lex)

Example of a program's CFG

Consider the following program:

```
a := 7;
b := c + (d := 5 + 6, d)
```

This can be written in terms of tokens (after lex) as:

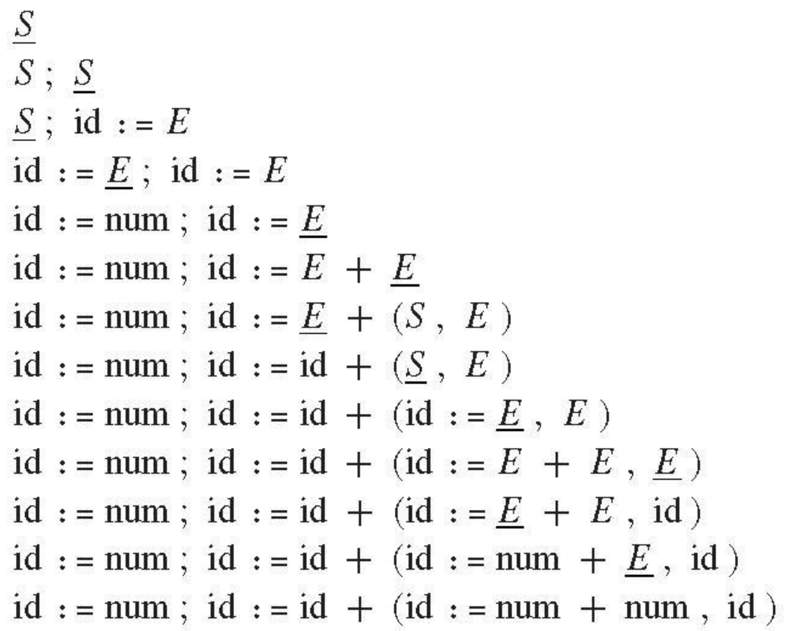
```
id := num; id := id + (id := num + num, id)
```

We can derive that the above string is a member of presented grammar by the following expansion, called a DERIVATION. Derivation of above recognition is on the right.

There can be many derivations based on what we choose to expand first. Examples:

- Leftmost derivation: when leftmost non-terminal is always expanded first.
- Similarly rightmost derivation can be defined.
- Many other derivations.

Example on right is neither a leftmost nor a rightmost derivation.



$$\begin{array}{l}
\underline{S} \\
S \;;\; \underline{S} \\
\underline{S} \;;\; \text{id} := E \\
\text{id} := \underline{E} \;;\; \text{id} := E \\
\text{id} := \text{num} \;;\; \text{id} := \underline{E} \\
\text{id} := \text{num} \;;\; \text{id} := E + \underline{E} \\
\text{id} := \text{num} \;;\; \text{id} := \underline{E} + (S \,,\, E) \\
\text{id} := \text{num} \;;\; \text{id} := \text{id} + (\underline{S} \,,\, E) \\
\text{id} := \text{num} \;;\; \text{id} := \text{id} + (\text{id} := \underline{E} \,,\, E) \\
\text{id} := \text{num} \;;\; \text{id} := \text{id} + (\text{id} := E + E \,,\, \underline{E}) \\
\text{id} := \text{num} \;;\; \text{id} := \text{id} + (\text{id} := \underline{E} + E \,,\, \text{id}) \\
\text{id} := \text{num} \;;\; \text{id} := \text{id} + (\text{id} := \text{num} + \underline{E} \,,\, \text{id}) \\
\text{id} := \text{num} \;;\; \text{id} := \text{id} + (\text{id} := \text{num} + \text{num} \,,\, \text{id})
\end{array}$$

DERIVATION 3.2.

Parse trees

Derivation information can be presented in a graph structure called a parse tree:

The parse tree for the previous example program is on the right.

Since the parse tree does not specify an order of expansion, more than one derivation can exist for the same parse tree.

- This is a good thing since ideally we want a single parse tree for a program, not multiple derivations for it!

But are parse trees really unique? We look at that next.

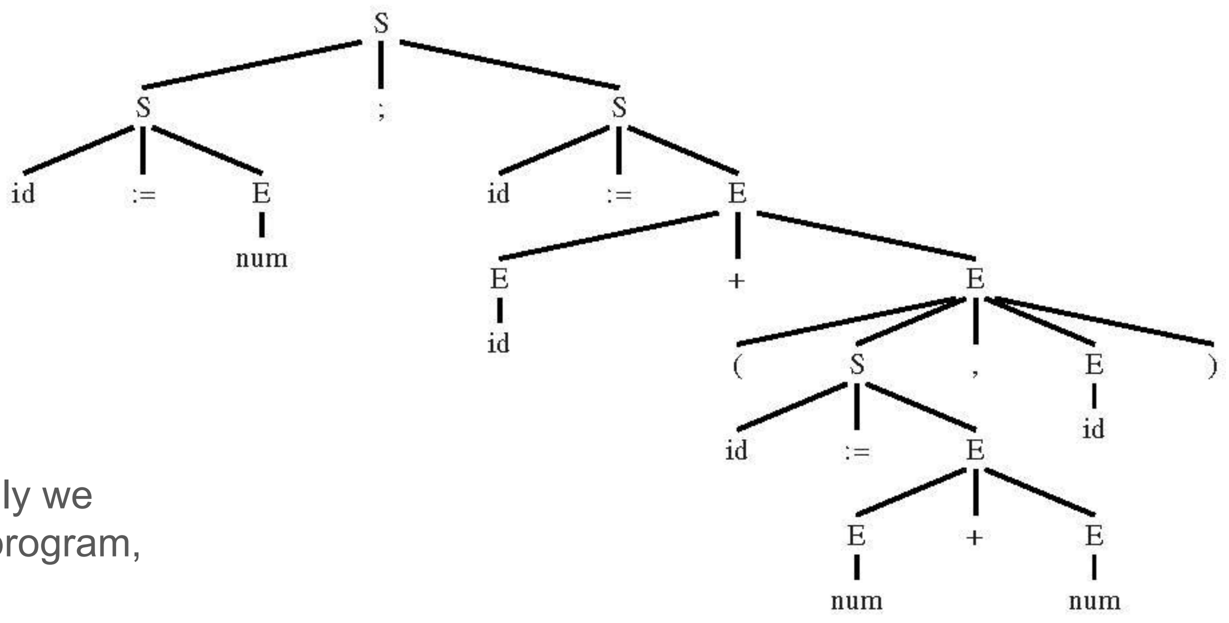


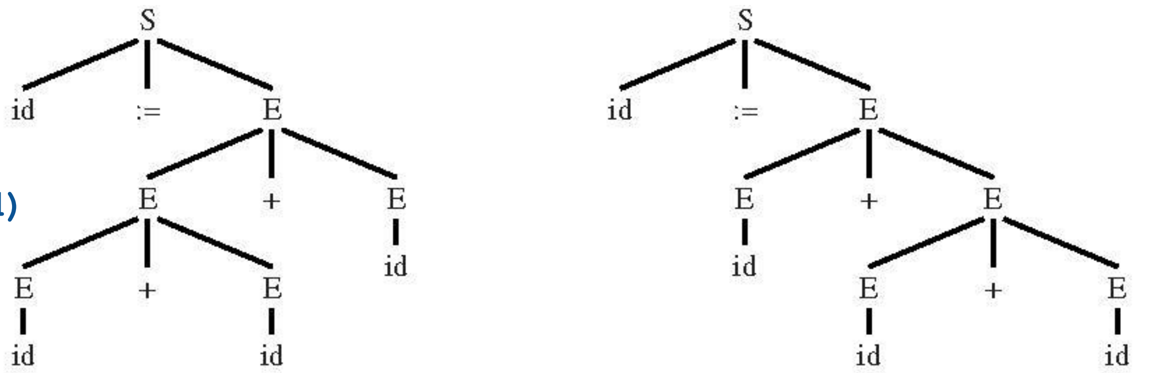
FIGURE 3.3. Parse tree.

Ambiguous grammars

A grammar is ambiguous if it can derive a sentence with two different parse trees.

- The above grammar (in fig 3.1) is ambiguous since id := id + id + id has more than one parse tree, depending on whether the first or second addition is done first:

`((id + id) + id)`



`(id + (id + id))`

FIGURE 3.4. Two parse trees for the same sentence using Grammar 3.1.

In the above, both parse trees have nearly the same meaning since theoretically addition is associative, but sometimes different meanings are possible, as we see on the next slide for subtraction.

- Actually, **addition is not associative on computers** because of overflows and finite precision.
- C specifies associativity from the left `((id + id) + id)`. So ambiguity is not acceptable even here.

Why ambiguous grammars are unacceptable

For grammar 3.5 on the top right:

String **`1-2-3`** has two different parse trees shown in figure 3.6.

- They mean different things!
- **`1-(2-3) = 2`** is not the same as **`(1-2)-3 = -4`**

The C language requires associativity from the left for operators of equal precedence, but this grammar does not capture it.

Similarly, **`1 + 2 * 3`** is also ambiguous. In this case, grammar should capture PEMDAS (aka BODMAS) rule, but it does not.

Hence ambiguous grammars are unacceptable. We need to disambiguate them.

$$E \to \text{id}$$
$$E \to \text{num}$$
$$E \to E * E$$
$$E \to E \,/\, E$$
$$E \to E + E$$
$$E \to E - E$$
$$E \to (\, E \,)$$

GRAMMAR 3.5.

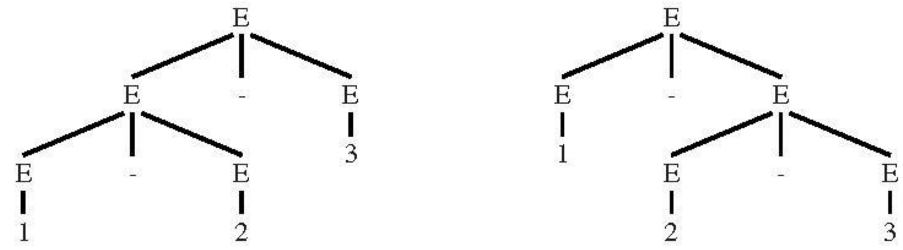


FIGURE 3.6. Two parse trees for the sentence `1-2-3` in Grammar 3.5.

Removing ambiguity: mathematical rules

Using precedence as defined by PEMDAS:

- **Associate respecting precedence**, from highest to lowest.
  - E.g., multiplication (X) has higher precedence than addition (+).
    - So `1 + 2 * 3` means `(1 + (2 X 3))`, not `(1 + 2) X 3`

- **Associate from one direction** (e.g., the left for C) for operators of equal precedence:
  - Multiplication and division have equal precedence. So resolve ambiguity via associativity from one direction. (E.g., associate from the left)
    - So `1 X 2 / 3` means `(1 X 2) / 3`, not `1 X (⅔)`
    - Addition and subtraction also have equal precedence.
  - Also associate from one direction for the same operator:
    - Eg. `(1 - 2) - 3` is correct, not `1 - (2 - 3)`.

Removing ambiguity in CFGs

In CFGs, we can enforce the second property (association from the left) by adding a MINUSEXP non-terminal in addition to minus:

MINUSEXP $\longrightarrow$ id

MINUSEXP$\longrightarrow$ MINUSEXP **-** id

But adding the first property (precedence) to CFGs is less obvious.

Removing ambiguity in CFGs

We can solve both problems for grammar in fig 3.5 as follows:

$$
\begin{array}{lll}
E \rightarrow E + T & T \rightarrow T * F & F \rightarrow \mathrm{id} \\
E \rightarrow E - T & T \rightarrow T \,/\, F & F \rightarrow \mathrm{num} \\
E \rightarrow T & T \rightarrow F & F \rightarrow (\, E \,)
\end{array}
$$

GRAMMAR 3.8.

Definitions: E, T, F are Expression, Term, and Factor.

- Expression: things containing + or -
- Terms: things containing * or /
- Factor: things containing indivisible items (ID, NUM or parenthesized expressions).

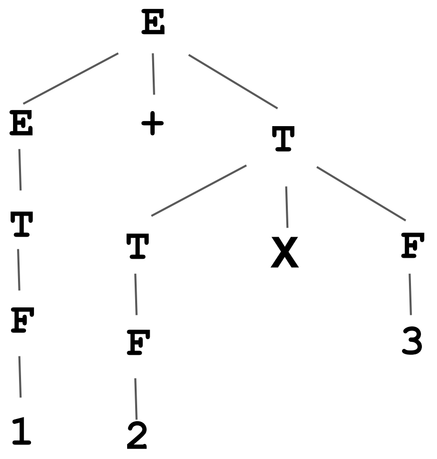
So intuitively, E, T, F have increasing order of precedence.
Each of these three non-terminals are defined only in terms of higher precedence non-terminals or operators, so the higher precedence operators have to be applied first, and only if they are on the right of the RHS (if on left of RHS, no need).

- E.g., parenthesis is applied first, since only F expands to it directly, so F has to be immediate parent of parenthesis in the parse tree.
- Also `*` and `/` are applied next, since only T expands to them (and to F), so T has to be immediate parent of F. Last, E is expanded to + and -. As a result, parenthesis are prioritized, then * and /, and then + and -.

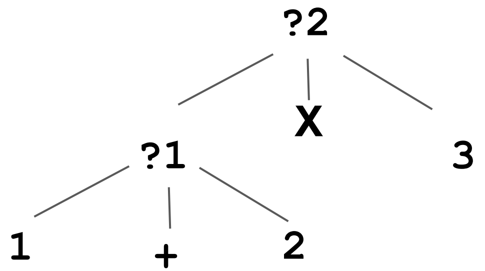
Example of removing ambiguity

Consider `1 + 2 X 3` parsed by above grammar in Figure 3.8.

Here is the only possible parse tree:



The following parse tree is not possible:



The parse tree on the right is not possible because **`?2`** cannot be any non-terminal. Only **`T`** expands to **`X`**, so if **`?2`** is **`T`**, then it expands to **`T * F`**, in which case **`?1`** is **`T`**, which is impossible, since **`T`** cannot expand to addition without going through a parenthesis first, which is absent.

Knowing when to recognize the whole program in CFG

We can use the End-of-file marker as follows:

Suppose token `$` is the end-of-file marker, and `S` is the top-level non-terminal. Then the following rule now recognizes the entire program in `S`’ where `S` is a statement list.

```
S' → S $
```

This is called the augmented grammar.

Recognizing CFGs automatically

To recognize programming languages as CFGs, we need a mathematical executable machine model that can ‘run’ and do this recognition.

This is similar to the fact that we needed DFAs to recognize regular expressions. Regular expressions are suitable for formal language specification, but cannot be run (i.e., executed). In contrast, DFAs can be executed.

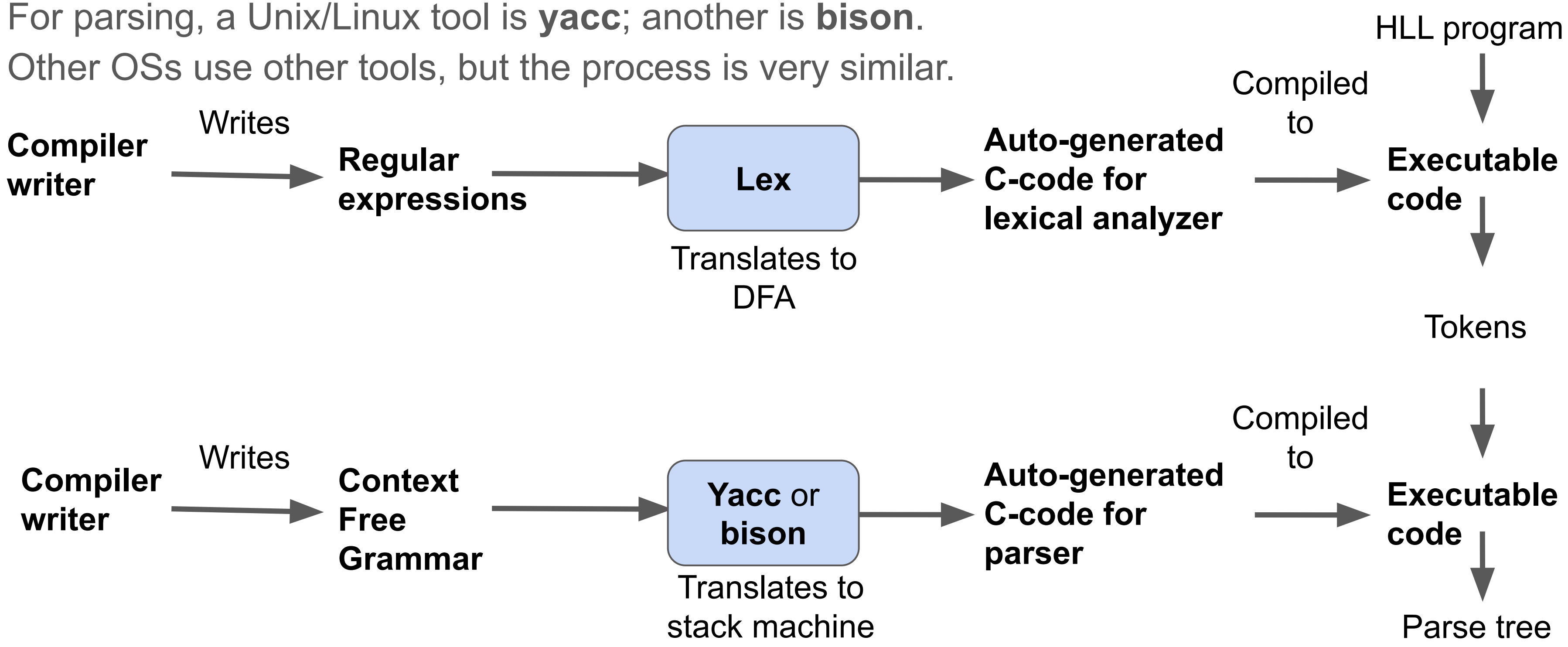
Similarly for parsing, we do the following steps:

- Compiler writer writes language specifications formally by writing CFGs.
- Then the compiler’s parsing code automatically translates CFG into an executable translation called a Stack Machine. (Translation mechanism is not in syllabus. How stack machines look like and how they run are also not in syllabus.)
- Then parsing code automatically converts the stack machine into an output C program, which when compiled and run, is the parser for the language. (similar process to lex)

Commonly used open-source lex and parse tools

Commonly used open-source tools are often used for building lex and parse stages of the compiler.

- For lexical analysis, a Unix/Linux tool is **lex**.
- For parsing, a Unix/Linux tool is **yacc**; another is **bison**.
- Other OSs use other tools, but the process is very similar.


HLL program
Compiler writer
Writes
Regular expressions
Lex
Translates to DFA
Auto-generated C-code for lexical analyzer
Compiled to
Executable code
Tokens
Compiler writer
Writes
Context Free Grammar
Yacc or bison
Translates to stack machine
Auto-generated C-code for parser
Compiled to
Executable code
Parse tree