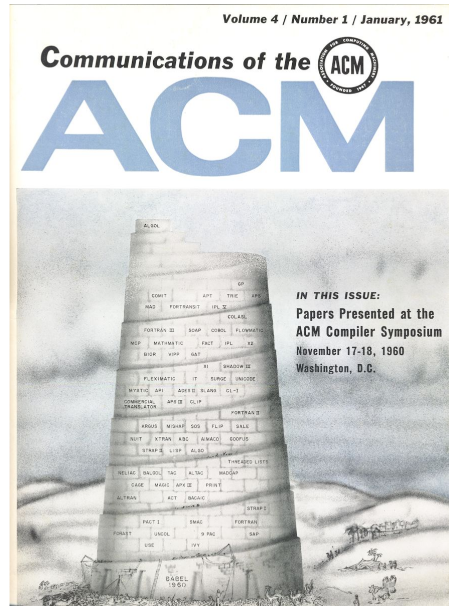
Programming Languages

CSE 3302, Summer 2019
Module 02
Regular Expressions and
Context-Free Grammars



M02

Regular Expressions and Context-Free Grammars

Formality of Programming Languages

- Unlike natural languages, programming languages must be *precise* and without *ambiguity*.
  - Given a source program, the exact steps to execute it must be determinable in a predictable, repeatable fashion.
  - Otherwise how would the computer know what to do and how would the programmer know what the computer will do?
- Formal notation is therefore used to define the language and then to write programs in that language.
  - The notation describes the *syntax* and the *semantics*.

[*"Formal Notation"* …]

- *Notation* is a way to express information, usually in a *written* form.
  - Using the Roman letters (`A` … `Z`) to express the written form of English words is an example of a *notation*.
- *Formal* (here at least) means that the *notation* is sufficiently *precise* and *well defined* that it can be processed *mechanically*.
  - The symbols `+`, `-`, `*`, and `/` along with their accepted meanings are a "formal notation" for expressing basic arithmetic.
  - It's not hard to write a program that accepts formulae in this notation and returns the corresponding values. The formulae are processed *mechanically*. (That is, no *human interpretation* is required.)

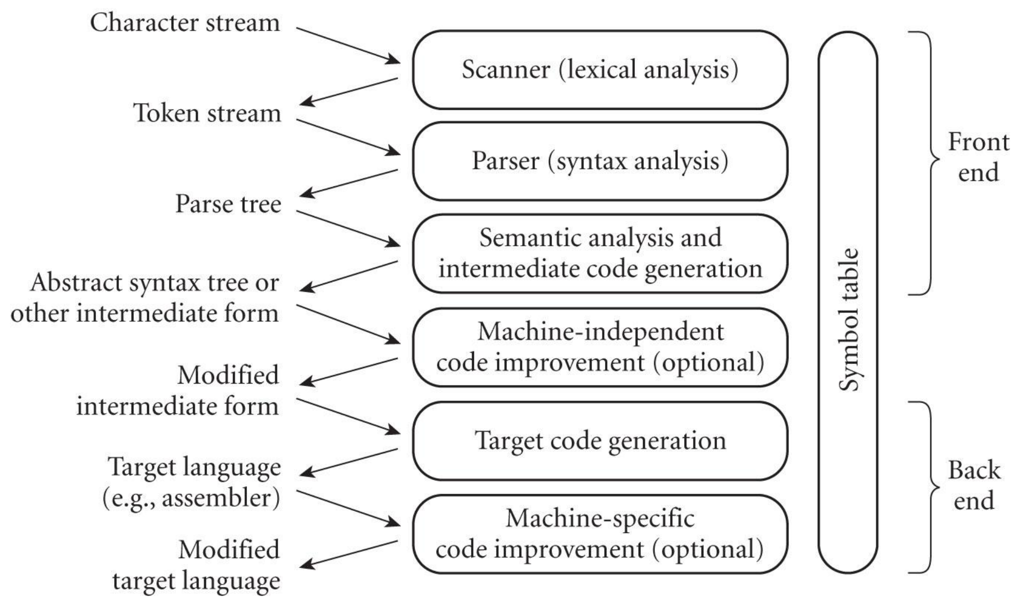
Programming Language Syntax

- *Syntax* is the *form* of the language as opposed to its *meaning* (the *semantics*)
- Why differentiate?
  - Languages tend to differ *greatly* in syntax but often have *very similar* aspects of semantics.
  - Easier to learn new language if the meaning behind the form can be distinguished.
  - Syntactic analysis has been extensively studied. There are many algorithms for determining the form of a program.

Specifying Syntax

- As seen in the compilation overview, the first two phases of compilation are the *lexical analysis* and the *syntactical analysis*.
  - Lexical Analysis converts characters into *tokens*.
  - Syntactical Analysis converts tokens into a *parse tree*.
- *Regular expressions* (RE) are used to specify the *token* formats.
- A *Context-Free Grammar* (CFG) is used to specify the parse structure.

Phases of Compilation



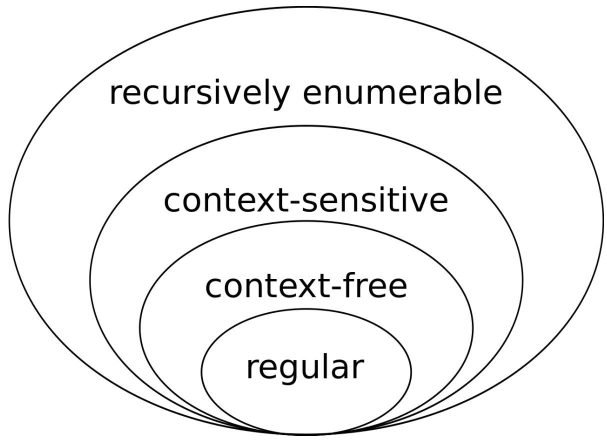
[*Grammars and the Chomsky Language Hierarchy*]

Terminals	a, b
Non-terminals	S
Production rules	$S \to \mathtt{a}S\mathtt{b}$ $S \to \varepsilon$
Start symbol	S

- A *grammar* specification includes
  - A set of *terminal* symbols
  - A set of *non-terminal* symbols
  - A set of *production rules*
  - A *start symbol*
- Restrictions on the form of the production rules can be used to categorize grammars into classes, each of which is more *expressive* than the previous.
- The example expresses the language $\mathtt{a}^n\mathtt{b}^n$, for $n \geq 0$.

[*Grammars and the Chomsky Language Hierarchy*]

- The hierarchy shows four levels of grammar expressivity.
- Each properly includes the previous.
- The levels have increasing expressiveness, but also parsing complexity, resource requirements, etc.



[*Grammars and the Chomsky Language Hierarchy*]

Type	Name	Allowable Productions	Example Language	Example Grammar	Example Use	Recognizing Automaton	Storage Required	Parsing Complexity
0	Recursively Enumerable	Unrestricted				Turing Machine	Infinite Tape	Undecidable
1	Context Sensitive	$\alpha \rightarrow \beta$ where $\lvert\alpha\rvert \leq \lvert\beta\rvert$ $\alpha \in V^*NV^*$ $\beta \in V^+$	$a^n b^n c^n$	$S \rightarrow aSBC$ $S \rightarrow aBC$ $CB \rightarrow BC$ $aB \rightarrow ab$ $bB \rightarrow bb$ $bC \rightarrow bc$ $cC \rightarrow cc$		Linear Bounded Automaton	Tape a linear multiple of input length	NP Complete
2	Context Free	$A \rightarrow \alpha$ $A \in N$ $\alpha \in V^*$	$a^n b^n$	$S \rightarrow aSb$ $S \rightarrow ab$	Arithmetic Expression $x = a + b * c$	Pushdown Automaton	Pushdown Stack	$O(n^3)$
3	Regular Right Linear Finite Automaton Recognizable	$A \rightarrow xB$ $A \rightarrow x$ $A, B \in N$ $x \in T^*$	$a^n b$	aa^*b or $S \rightarrow ab$ $S \rightarrow aS$	Identifier VECTOR7	Finite Automaton	Finite Storage	$O(n)$

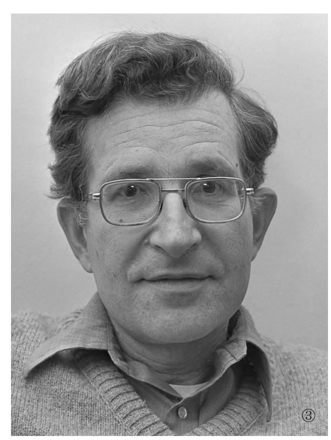
https://www.cs.utexas.edu/~novak/chomsky.gif

[*"Recursive" Production Rules …*]

- Later we'll learn that *recursion* is what separates *Context-Free Grammars* from *Regular Expressions*. No recursion in REs.
- On the previous chart, $S \longrightarrow \mathtt{a}S$ appears as a rule for an RE.
  - Hey, isn't that *recursive*? After all, *S* refers to itself, right?

- ***No!*** It just ***looks*** as if it's recursive. :)
- Because the *S* appears on the far right (there's nothing after *S* in the rule), this is just a way of expressing a Kleene-* operation (`a*`).
- A proof that this isn't true recursion requires going deeper in formal language theory than required for this class, so just accept it as true.

[*John Backus, Peter Naur, and Noam Chomsky*]



① http://www.columbia.edu/cu/computinghistory/backus.html
② http://homepages.cs.ncl.ac.uk/brian.randell/NATO/N1968/NAUR.htm
③ https://commons.wikimedia.org/wiki/File:Noam_Chomsky_(1977).jpg

Tokens

- A *token* is the basic building block of a program.
  - The shortest strings of characters in the source program that have individual meaning.
- Tokens come in various types, according to the programming language's specification.
  - Common types include *numbers*, *identifiers*, *keywords*, *operators*, *punctuation marks*, and so forth.
- The formats of the tokens of a programming language are normally specified using *regular expressions*.

Simple Regular Expression

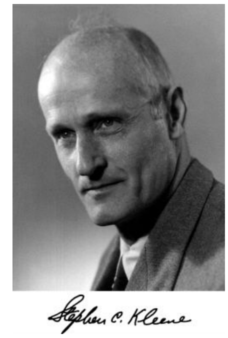
natural_number ⟶ *non_zero_digit* *digit**

non_zero_digit ⟶ `1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9`

digit ⟶ `0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9`

- Describes any string of decimal digits one or more digits in length that does not start with a `0`.

Definition of a Regular Expression (RE)



/ˈkleɪni/ KLAY-nee

1. A *character*
2. The *empty string*, denoted as ε
3. The *concatenation* of two REs

    Meaning one RE after the other RE
4. The *alternation* of two REs, denoted by |

    Meaning one RE *or* the other RE
5. An RE followed by the *Kleene star*, denoted as *

    Meaning zero or more repetitions of the RE

https://www.nap.edu/read/9649/chapter/10

Definition …

- The set of strings defined by an RE is known as its *language*.
    - Not the same sense as in *programming language*!
- An RE is *not* allowed to *recurse*.
    - That is, no part of an RE can refer to itself, even indirectly.
    - If it did, it's no longer an RE, but a CFG instead.
    - (Remember the *Chomsky Language Hierarchy*.)

- There are some notational conveniences that don't change the expressive power of REs, but are very useful.

Notational Conveniences …

- Kleene Plus is an example of a notational convenience.
  - It means *one or more* occurrences of a RE.

- It's easily represented using Kleene Star (*zero or more*).
  - If `A` is an RE, `A+` is defined to be `AA*`.

- By expressing Kleene Plus in terms of Kleene Star, we see that we didn't gain any expressive power, just convenience.

Notational Conveniences …

- There are other conveniences.
  - `A?` ≡ `A|ε` *or* 0 or 1 occurrence of an RE.
  - `[abc]` ≡ `a|b|c` *or* any of a set of characters.
  - (and there are others …)

- Each of these conveniences is representable using the basic five ways of writing a regular expression, so no expressive power is gained by using them.

Extensions …

- The definition of RE is often truly *extended* by string-processing programming languages.
  - Perl, Python, Ruby, `awk`, `sed`, … all add extensions that increase the expressive power beyond the *regular sets*.
    - For example, referring back to an earlier matched string is not possible in a formal RE.

- Though these languages call their representations Regular Expressions, they are *not* true REs.

Numeric Literal Regular Expression

number ⟶ *integer* | *real*

integer ⟶ *digit* *digit**

real ⟶ *integer* *exponent* | *decimal* (*exponent* | ε)

decimal ⟶ *digit** (`.` *digit* | *digit* `.`) *digit**

exponent ⟶ (`e` | `E`) (`+` | `-` | ε) *integer*

digit ⟶ `0` | `1` | `2` | `3` | `4` | `5` | `6` | `7` | `8` | `9`

- Q: Why the weird-looking definition for *decimal*?

Numeric Literal Examples

- *integer*
  - Simple! Just any non-zero length string of decimal digits.
  - `0`, `1`, `2`, `3`, `4567`, `007`, `070`, …
  - But ***not***, e.g., `-1` (there's no sign on a numeric literal).
- *real*
  - Trickier! An *integer* with an *exponent* or a *decimal* with an optional *exponent*.
    - *exponent* is `e` or `E` followed by an optionally signed *integer*.
    - *decimal* is a decimal point with digits on *at least* one side.
  - `0E+1`, `2e-3`, `4E5`, `6.`, `.7`, `8.9E0`, …
  - But ***not***, e.g., `.E1` (need at least one digit next to the `.`)

Regular Expression Limitations

- Regular Expressions are *great*!
  - Simple to compose.
  - Simple to analyze.
  - Simple to decide if a given string matches a given RE.
    - *Linear* in the length of the string.
- *But*, they can't be used to check, e.g., *matching* or *nested* parentheses, brackets, etc.
- Why not?

Regular Expression Limitations

- A regular expression is trivially convertible to a non-deterministic finite automata (NFA) which is trivially convertible to a deterministic finite automata (DFA).

- The important word in those names is *Finite*.
  - The automata have a fixed, finite number of states.
  - No matter how big that number is, it's possible to construct a string that is *just a little bit more complicated* so the finite automata cannot recognize it.

Context-Free Grammar (CFG)

- Regular Expressions are limited in expressiveness.
  - Cannot define strings that are required to have arbitrary …
    - Well-formed and/or nested parentheses, brackets, etc.
      e.g., `([[{(){}[][]}]{[]}])`
    - Matching pairs
      e.g., $a^n b^n \longrightarrow$ ab, aabb, aaabbb, …

- Next step up is a *Context-Free Grammar*
  - Definitions can refer to themselves, i.e., can *recurse*.
  - We generally express CFGs in *Backus-Naur Form*.

[*Backus - Naur Form*]

- A *formal notation* for writing Context-Free Grammars.
- Originally developed to specify Algol and first used in the Algol 60 report.
- Looked a bit different from what we use here, but the expressive power is the same. We use …
    - "⟶" instead of "`::=`"
    - *italics* for *non-terminals* instead of enclosing in < >
    - ε for empty string
    - Won't bother with "" unless required for clarity

BNF

```
<postal-address> ::= <name-part> <street-address> <zip-part>

<name-part> ::= <personal-part> <last-name> <opt-suffix-part> <EOL>
              | <personal-part> <name-part>

<personal-part> ::= <initial> "." | <first-name>

<street-address> ::= <house-num> <street-name> <opt-apt-num> <EOL>

<zip-part> ::= <town-name> "," <state-code> <ZIP-code> <EOL>

<opt-suffix-part> ::= "Sr." | "Jr." | <roman-numeral> | ""

<opt-apt-num> ::= <apt-num> | ""
```

https://en.wikipedia.org/wiki/Backus–Naur_form

Ours

postal-address ⟶ *name-part street-address zip-part*

name-part ⟶ *personal-part last-name opt-suffix-part EOL*
| *personal-part name-part*

personal-part ⟶ *initial* . | *first-name*

street-address ⟶ *house-num street-name opt-apt-num EOL*

zip-part ⟶ *town-name* , *state-code ZIP-code EOL*

opt-suffix-part ⟶ Sr. | Jr. | *roman-numeral* | ε

opt-apt-num ⟶ *apt-num* | ε

CFG Example

expr ⟶ *id* | *number* | - *expr* | (*expr*) | *expr op expr*

id ⟶ (_ | a | b | … | z)(_ | a | b | … | z | 0 | 1 | … | 9)*

op ⟶ + | - | * | /

- Notice that *expr* refers to itself. This definition is *recursive*.
    - And *expr* is *not* on the far right, so *true* recursion.
- Q: What set of strings does *id* define?

CFG Example

- To *derive* (or *generate*) a string from a CFG, begin with the start symbol and replace non-terminals according to the rules until only terminals remain.
    - A *sentential form* is the start symbol or any form derived from it.
    - A *sentence* is a sentential form which has only terminal symbols.
- Example, generate

    `slope * x + intercept`

    using the *expr* CFG.

<u>expr</u>

expr op <u>expr</u>

expr <u>op</u> id

<u>expr</u> + *id*

expr op <u>expr</u> + *id*

expr <u>op</u> id + *id*

<u>expr</u> * *id* + *id*

<u>id</u> * *<u>id</u>* + *<u>id</u>*

`slope * x + intercept`

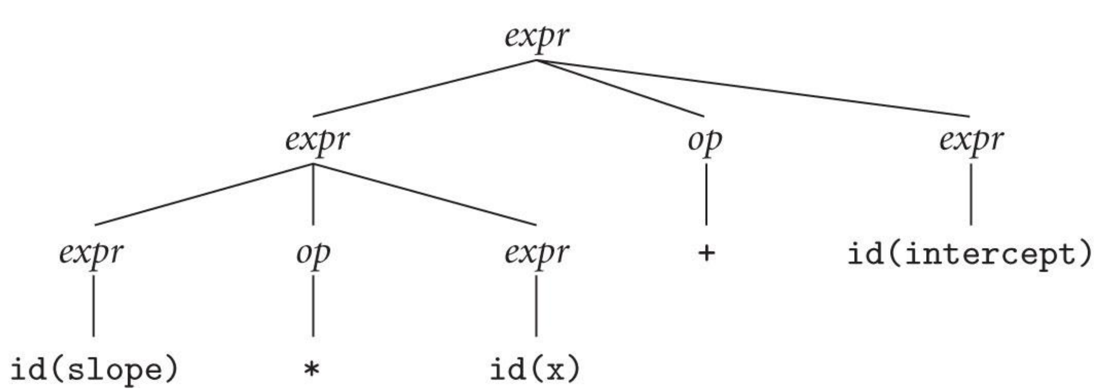
CFG Example

- That derivation was *Right-Most*.
    - We replaced the *right-most* non-terminal each time we took a step in the derivation.
    - *Except* for the final replacement of the *id* non-terminals with their actual words. (This was to make the derivation a little shorter and easier to fit on the slide.)
- A *Left-Most* derivation replaces the *left-most* non-terminal each time a step is taken in the derivation. (Duh.)

CFG Example

- The steps we took in the derivation correspond to the construction of a *parse tree*.
- The *root* of the parse tree is the *start symbol*.
- Every time we use a production rule, it's the same as adding a new (set of) node(s) to the tree.
- All internal nodes of the parse tree are *non-terminals*.
- The leaves of the final parse tree are *terminals*.
  - These terminals are the *tokens* of the original string.

CFG Example Parse Tree (from the Right)



CFG Example

- There's another way to do the generation of

  `slope * x + intercept`

  using the *expr* CFG.
- This derivation goes from the *left* instead of the *right*.

expr

expr op expr

id op expr

*id * expr*

*id * expr op expr*

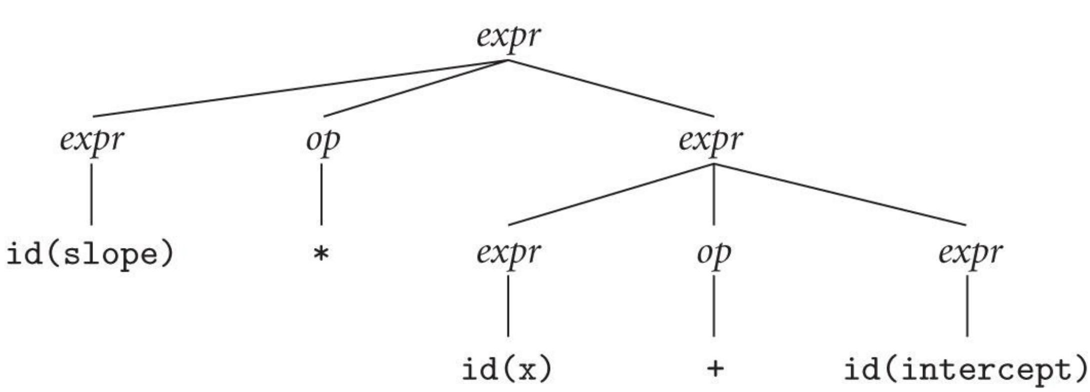
*id * id op expr*

*id * id + expr*

*id * id + id*

`slope * x + intercept`

CFG Example Parse Tree (from the Left)



CFG Example

- The CFG allows two parse trees for the same string because its definition is *ambiguous*.
  - Also, it allows incorrect parsing of operator precedence.
- So is the answer to always derive from the right?
  - After all, that got the correct derivation.
- No!
  - Consider the right-most derivation of `intercept + slope * x`

expr

expr op expr

expr op id

*expr * id*

*expr op expr * id*

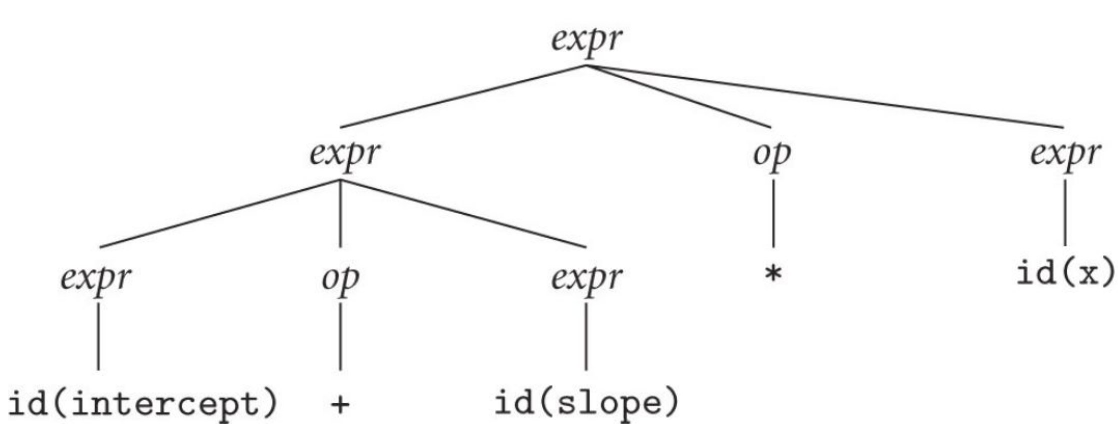
*expr op id * id*

*expr + id * id*

*id + id * id*

`intercept + slope * x`

CFG Example 2 Parse Tree (from the Right)



Improved CFG Example

- We need a CFG that is not ambiguous *and* honors our concept of operator precedence.
- A better (though more complex) CFG would be

  expr ⟶ *term* | *expr add_op term*

  term ⟶ *factor* | *term mult_op factor*

  factor ⟶ *id* | *number* | - *factor* | (*expr*)

  add_op ⟶ + | -

  mul_op ⟶ * | /

expr

expr add_op term

expr add_op factor

expr add_op id

expr + *id*

term + *id*

term mult_op factor + *id*

term mult_op id + *id*

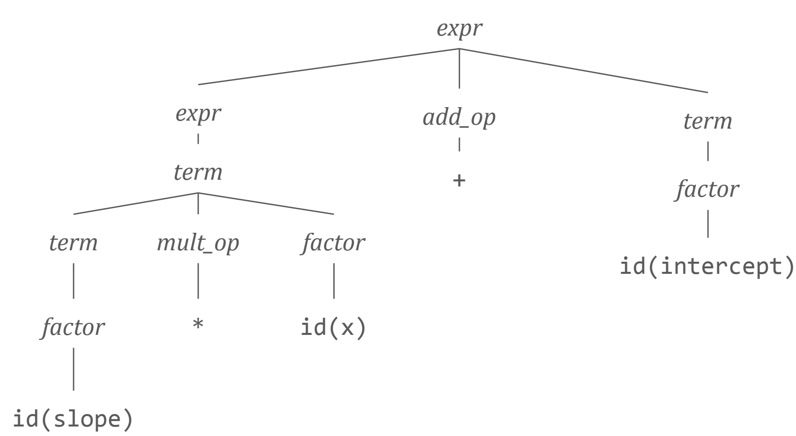
term * *id* + *id*

factor * *id* + *id*

id * *id* + *id*

`slope * x + intercept`

Improved CFG Parse Tree (from the Right)



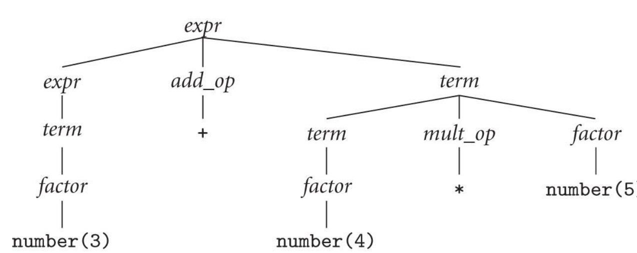
Improved CFG Example

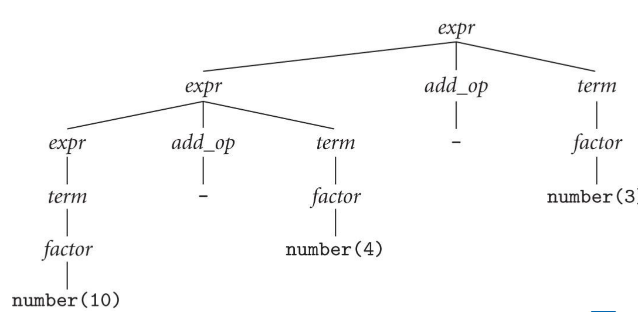
- What about doing the derivation from the left?
- We still get the correct derivation!
  - The grammar is unambiguous and it honors our intuitive understanding of operator precedence.
  - Multiply and divide are "tighter" than addition and subtraction.
- But what about *associativity*?
  - How are "like" operations grouped?

expr
expr add_op term
term add_op term
term mult_op factor add_op term
factor mult_op factor add_op term
id mult_op factor add_op term
*id * factor add_op term*
*id * id add_op term*
*id * id + term*
*id * id + factor*
*id * id + id*
`slope * x + intercept`

Improved CFG Example

- This CFG handles *precedence* and *associativity* according to our intuitive expectations.
  - 3+4*5 means
    - 3+(4*5) = 23
    - *not* (3+4)*5 = 35.
  - 10-4-3 means
    - (10-4)-3 = 3
    - *not* 10-(4-3) = 9.





Improved CFG Example

- OK, so it works. But *why* does it work?
- Operator *precedence* is enforced by having nested rules that permit the repetition of only certain operators at each precedence level.
  - *expr* is "looser" than *term* as it occurs higher in the CFG definition.
  - Therefore the *add_op* operators are "looser" than the *mult_op* operators.
- Operator *associativity* is enforced by having the *expr* and *term* rules recurse on their left side rather than the right.
  - Therefore operations group from the left instead of the right.

expr ⟶ *term* | *expr add_op term*

term ⟶ *factor* | *term mult_op factor*

factor ⟶ *id* | *number* | - *factor* | (*expr*)

add_op ⟶ + | -

mul_op ⟶ * | /

Improved CFG

- In summary, we have seen that operator *precedence* and *associativity* are defined by how the CFG is constructed.
  - They are both *syntactic* properties of the language.
- *Precedence* is in *levels*.
  - Several operators may be at the same precedence level.
  - E.g., In C, + and -, * and /. (The latter are at a higher level.)
- Associativity is *left-to-right*, *right-to-left*, or *does-not-apply*.
  - E.g., in C, +,-,*, and / are L→R but +=, -=, *=, /= are R→L.
  - Any non-associative operators in C? What about Python?

Review Questions (1)

1. What is the difference between *syntax* and *semantics*?
2. What are the three basic operations that can be used to build complex *regular expressions* from simpler regular expressions?
3. What additional operation is provided in *context-free grammars*?
4. What is *Backus-Naur Form*? When and why was it developed?
5. Name a language in which indentation affects program syntax?

Review Questions (2)

6. When discussing context-free languages, what is a *derivation*? What is a *sentential form*?
7. What is the difference between a *right-most* derivation and a *left-most* derivation?
8. What does it mean for a CFG to be *ambiguous*?
9. What are *associativity* and *precedence*? Why are they significant in parse trees?