CSCC24 – Principles of Programming Languages
Formal Language Theory

Anya Tafliovich[1]

[1]with thanks to S.McIlraigh, G.Penn, P.Ragde

specifying syntax informally

Example:

"Everything between `/*` and `*/` is a comment and should be ignored."

```
/* Do such and such, watching out for problems.
   Store the result in y.  */
x = 3;  */
y = x * 17.2;
```

When syntax is defined informally, incompatible dialects of the language may evolve.

specifying PL syntax

- The state of the art is to define programming language syntax formally.
- There are a number of well-understood formalisms for doing so. We'll talk about this in some detail.
- Lexical rules specify the form of the building blocks of the language:
  - what's a token (keyword, identifier, literal, operator, punctuation, possibly white space)
  - how tokens are delimited
  - where white space can go
  - how comments are specified
- Syntax rules specify how to put the building blocks together.

grammars

Informal idea of grammar: a bunch of rules. For example:

- Don't end a sentence with a preposition.
- Subject and verb must agree in number.

A formal grammar is a somewhat different concept.

A language is a set of strings. A grammar generates a language — it specifies which strings are in the language.

There are many kinds of formal grammar.

Chomsky's Hierarchy

There are several categories of grammar, ordered by expressiveness (the last one is the most expressive):

- Regular Grammars
- Context-Free Grammars
- Context-Sensitive Grammars
- Phrase-Structure Grammars

This hierarchy (circa 1950) is named after linguist (and political activist) Noam Chomsky, who researched grammars for natural languages.

Look him up!

Regular Expressions

Examples:

- $(0+1)^*$
- $1^*(:+;)^*$
- $(a+b)^*aa(a+b)^*$

Notation:

- Kleene star: * superscript denotes 0 or more repetitions.
- alternation: binary "+" denotes choice.
  It is sometimes denoted by $|$, i.e., $(0|1)^*$.
- grouping: "(" and ")" are used for grouping
- empty string: ϵ (epsilon) denotes the empty or "null" string.
- empty language: $\emptyset$ denotes the language with *no* strings.

Regular Expressions

Formally, let Σ be a given finite alphabet. A string is a regular expression (RE) if and only if it can be derived from the primitive regular expressions:

- $\emptyset$
- ϵ
- any a, s.t. $a \in \Sigma$

by a finite number of applications of the following rules:
If r_1 and r_2 are REs, then so are:

- $r_1 + r_2$, or $r_1|r_2$ (union, alternation)
- $r_1 r_2$, or $r_1 \cdot r_2$ (concatenation)
- r_1^* (Kleene closure)
- (r_1) (grouping)

Σ^* denotes the set of all finite strings over the alphabet Σ.

Regular Expressions

Given a RE r over an alphabet Σ, a language associated with r, denoted $L(r)$, is defined by:

- $L(\emptyset)$ is the empty language (language that contains no strings)
- $L(\epsilon)$ is the language $\{\epsilon\}$
- for $a \in \Sigma$, $L(a)$ is the language $\{a\}$

If r_1 and r_2 are regular expressions, then

- $L(r_1 + r_2) = L(r_1) \cup L(r_2)$,
  where $S \cup T = \{s \mid s \in S \text{ or } s \in T\}$
- $L(r_1 r_2) = L(r_1)L(r_2)$,
  where $ST = \{st \mid s \in S,\ t \in T\}$
- $L((r_1)) = L(r_1)$
- $L(r_1^*) = (L(r_1))^*$,
  where S^* denotes the smallest superset of S that contains ϵ and is closed under string concatenation

Regular Expressions

A regular language is a language that can be expressed by a regular expression.

A regular language is a language that is accepted by a finite-state automaton.

Deterministic finite state automata (DFSA) and non-deterministic finite state automata (NFSA) accept the same class of languages, but DFSA can be exponentially larger.

We won't talk about automata in this course: you studied them in cscb36/... .

Regular Expressions — examples

Give regular expressions for these languages:

1. All alphanumeric strings beginning with an upper-case letter.

2. All strings of `a`'s and `b`'s in which the third-last character is `b`.

3. All strings of `0`'s and `1`'s in which every pair of adjacent `0`'s appears before any pairs of adjacent `1`'s.

4. All binary numbers with exactly three `1`'s.

limitations of Regular Expressions

Regular expressions are not powerful enough to describe some languages.

Examples:

- The language consisting of all strings of one or more a's followed by the same number of b's.
- The language consisting of strings containing a's, b's, left brackets, and right brackets, such that the brackets match.

Question: How can we be sure there is no regular expression for these languages?

Context-Free Grammars

CFGs are more powerful than regular expressions.

A CFG has four parts:

- A set of terminals:
  the atomic symbols of the language.
- A set of non-terminals:
  "variables" used in the grammar.
- A special non-terminal chosen as the starting non-terminal or start symbol:
  represents the top-level construct of the language.
- A set of rules (or productions), each specifying one legal way that a non-terminal can be rewritten to a sequence of terminals and non-terminals.

Context-Free Grammars — example

A CFG for real numbers:

- Terminals: `0 1 2 3 4 5 6 7 8 9 .`
- Non-terminals: real-number, part, digit.
- Productions:
  - A digit is any single terminal except ".".
  - A part is a digit.
  - A part is a digit followed by a part.
  - A real-number is a part, followed by ".", followed by a part.
- Start symbol: real-number.

Note that we use recursion to specify repeated occurrences.

We have "defined" this CFG using plain English. Not a very good idea.

Context-Free Grammars

A context-free grammar G is a tuple $\{\Sigma, V, S, P\}$, where

- Σ is the set of terminals
- V is the set of non-terminals
- $\Sigma \cap V = \emptyset$
- S is the start symbol, $S \in V$
- P is the set of production rules,
  where each rule has the form
  $X \to w$ (sometimes written $X \Rightarrow w$ or $X ::= w$), where

$$X \in V \qquad (X \text{ is a single non-terminal}), \text{ and}$$
$$w \in (\Sigma \cup V)^* \qquad (w \text{ is a string of terminals and/or non-terminals})$$

Context-Free Grammars — example

Σ: $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, .\}$

V: {<real-number>,<part>,<digit>}, S: <real-number>

P:

```
<real-number> --> <part> . <part>
<part>        --> <digit>
<part>        --> <digit> <part>
<digit>       --> 0
<digit>       --> 1
<digit>       --> 2
<digit>       --> 3
 ...
<digit>       --> 9
```

Not very convenient. Shorthand: $X \to u|v|w$

```
<real-number> --> <part> . <part>
<part>        --> <digit> | <digit> <part>
<digit>       --> 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

Context-Free Grammars

A production rule $X \to w$ can be applied to a string of the form sXt, $s, t \in (V \bigcup \Sigma)^*$ to produce a string swt.

A grammar G generates a string s if there exists a (finite) sequence of applications of production rules, so that the first production rule in this sequence is applied to a start symbol, and the last production rule in the sequence produces s, $s \in \Sigma^*$. We write $G \Rightarrow^* s$.

The language generated by a grammar G is denoted $L(G)$, that is $L(G) = \{s \mid G \Rightarrow^* s\}$.

Convention: A start symbol is the first non-terminal listed.

Context-Free Grammars

CFGs are more "powerful" than REs. That is, there are languages that cannot be described with a RE but can be described with a CFG.

Example: The language consisting of strings with one or more a's followed by the same number of b's.

There is no regular expression for this language. (Why?)

CFG for the language:

Regular Grammars

- More restricted than CFGs.

- Limited to productions of the form:
  - Left-recursive grammar: at most one non-terminal and it appears as the left-most symbol in the string on the RHS of the production rule.

    ```
    <S> ::= <T> a b
    <T> ::= a | <T> b
    ```

  - Right-recursive grammar: at most one non-terminal and it appears as the right-most symbol in the string on the RHS of the production rule.

    ```
    <S> ::= a <T>
    <T> ::= b <T> | a b
    ```

Extended BNF

CFGs we've seen so far are in Backus-Naur Form (BNF).
There are extensions to BNF that make it more concise, but no more powerful (*i.e.*, there is no language that can be expressed with EBNF but not with BNF). Some examples:

- { *blah* } denotes zero or more repetitions of *blah*.
- [*blah*] denotes that *blah* is optional.
- a + superscript denotes one or more repetitions.
- a numeric superscript denotes a maximum number of repetitions.
- (and) are used for grouping.

There is no one standard EBNF; it just refers to any extension of BNF.
Exercise: Show that the above operators do not add any expressive power.

derivations

A derivation of string s is the sequence of applications of production rules that generates s.

For example, the grammar

```
<real-number> --> <part> . <part>
<part>        --> <digit> | <digit> <part>
<digit>       --> 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

generates the string 97.123.

A string is in the language generated by a grammar iff there is a derivation for it.

Derivation of 97.123:

parse trees

A parse tree is a tree in which

- the root is the start symbol,
- every leaf is a terminal, and
- every internal node is a non-terminal, and its children correspond, in order, to the RHS of one of its productions in the grammar.

Parse trees show the structure within a string of the language.

Parsing is the process of producing a parse tree.

A string is in the language generated by a grammar iff there is a parse tree for the string.

Parse tree for 97.123:

parse trees

Parse tree for 97.123:

syntactic ambiguity — in English

Syntactically ambiguous sentences of English:

- "I saw the dog with the binoculars."
- "The friends you praise sometimes deserve it."
- "They seemed nice to her."

Aside: We can often "disambiguate" ambiguous sentences. How?

syntactic ambiguity — in a programming language

Example (grammar excerpt):

```
<stmt>    --> <assnt-stmt> | <loop-stmt> | <if-stmt>
<if-stmt> --> if <boolean-expr> then <stmt>
            |  if <boolean-expr> then <stmt> else <stmt>
```

Example sentence:

```
if is_odd(x) then
if x == 1 then
print "foo";
else
print "bar";
```

Exercise: Draw the two parse trees.

syntactic ambiguity — definition

A grammar is ambiguous iff it generates a string for which there are two or more distinct parse trees.

A string is ambiguous with respect to a grammar iff that grammar generates two or more distinct parse trees for the string.

Note that having two distinct *derivations* does not make a string ambiguous.

A derivation corresponds to a traversal through a parse tree, and one can traverse a single tree in many orders.

syntactic ambiguity — example

Grammar: if statement two slides ago.

Sentence:

```
if is_odd(x) then
print "foo";
```

One parse tree, two derivations.

Exercise: provide the parse tree and two derivations.

syntactic ambiguity

When specifying a programming language, we want the grammar to be completely unambiguous.

Question: Is there a procedure one can follow to determine whether or not a given grammar is ambiguous?

We have two strategies for dealing with ambiguity:

1. Change the language to include delimiters.
2. Change the grammar to impose associativity and precedence.

dealing with ambiguity — delimiters

```
<stmt>    --> <assnt-stmt> | <loop-stmt> | <if-stmt>
<if-stmt> --> if <boolean-expr> then <stmt> fi
            |  if <boolean-expr> then <stmt>
                                    else <stmt>
               fi
```

Now the ambiguous problem from the previous slide is no longer in the language, and the programmer is forced to disambiguate.

example: a CFG for arithmetic expressions

Grammar $G1$:

```
<expn> --> <expn> + <expn> |
           <expn> - <expn> |
           <expn> * <expn> |
           <expn> / <expn> |
           <expn> ^ <expn> |
           (<expn>)        |
           <identifier>    |
           <literal>
```

The grammar is ambiguous.

Example:

example: a CFG for arithmetic expressions

changing the language to include delimiters

Grammar $G2$:

```
<expn> --> (<expn>) + (<expn>) |
           (<expn>) - (<expn>) |
           (<expn>) * (<expn>) |
           (<expn>) / (<expn>) |
           (<expn>) ^ (<expn>) |
           (<expn>)            |
           <identifier>        |
           <literal>
```

With the new restrictions, we have:

`(8)-((3)*(2))` $\in L(G2)$
`((8)-(3))*(2)` $\in L(G2)$
`8 - 3 * 2` $\notin L(G2)$

changing the grammar to impose precedence

- Low precedence: addition `+` and subtraction `-`.
- Medium precedence: multiplication `*` and division `/`.
- Higher precedence: exponentiation `^`.
- Highest precedence: parenthesized expressions `( <expr> )`.

Approach: introduce a non-terminal for every precedence level.

changing the grammar to impose precedence

Grammar $G3$:

```
<expn> -->
```

Grouping in parse tree now reflects precedence.

changing the grammar to impose precedence

Grouping in parse tree now reflects precedence.

example: a CFG for arithmetic expressions

We still have ambiguity! Example:

associativity

- Deals with operators of same precedence.
- Implicit grouping or parenthesizing.
- Left associative: `*`, `/`, `+`, `-`.
- Right associative: `^`.

Approach: For left-associative operators, put the recursive term before the non-recursive term in a production rule. For right-associative operators, put it after.

changing the grammar to impose associativity

Grammar $G4$:

exercise: dealing with ambiguity

Exercise:

```
<sentence> ::= \empty |
               <course> is <adjective>. |
               <sentence> <sentence>
<course> ::= CSCA08 | CSCA48 | CSCB07 | CSCB09 | CSCC24
<adjective> ::= great | fun | awesome
```

where `\empty` stands for the empty string.

- Demonstrate that the CFG is ambiguous.
- Provide a grammar that generates exactly the same language as above and is not ambiguous.

dealing with ambiguity

1. Can't always remove an ambiguity from a grammar by restructuring productions.
2. An inherently ambiguous language does not possess an unambiguous grammar.

Question. Is there an algorithm that can examine an arbitrary context-free grammar and tell if it is ambiguous?

an inherently ambiguous language

Suppose we want to generate the following language:

$$\mathcal{L} = \{a^i b^j c^k \mid i, j, k \geq 1, i = j \text{ or } j = k\}$$

Grammar:

Two parse trees for $a^i b^i c^i$.

limitations of CFGs

CFGs are not powerful enough to describe some languages.
Examples:

- $\{ a^i b^i c^i \mid i \geq 1 \}$.
- $\{ a^m b^n c^m d^n \mid m, n \geq 1 \}$.

Question: Is there an algorithm that can examine two arbitrary CFGs and determine if they generate the same language?

translation process summary

1. Lexical Analysis:
   Converts source code into sequence of tokens.
   We use regular grammars and finite state automata (recognizers).

2. Syntactic Analysis:
   Structures tokens into initial parse tree.
   We use CFGs and parsing algorithms.

3. Semantic Analysis:
   Annotates parse tree with semantic actions.

4. Code Generation:
   Produces final machine code.

more on this...

Take Compilers & Interpreters!