Syntax and Parsing

Aditya Thakur

Production Rules

- Express context-free grammars as list of rules
- Has four parts
  - A set of terminals (tokens)
  - A set of non-terminals (NT)
  - A set of production (or grammar) rules
  - A start non-terminal (start symbol, like “main” in a program)

Production Rules for English

S -> *P V P*

P -> *A N*

V -> loves
V -> hates
V -> eats

A -> a
A -> the

N -> dog
N -> cat
N-> rat

Derivations

- Derivation of a string by a grammar
  - Starting with the starting NT
  - Replace a NT by the RHS of one of its rules
  - Repeat until only terminals remain

- The language L(G) of a grammar G is defined as
  - The set of strings derivable from the starting NT of G

Parse Tree

- A parser of a compiler builds parse trees
- Graphical representation of the syntax structure of a string
- Making explicit how the string is generated from the rules
  - Root: starting NT
  - Interior nodes: NTs
  - Leaves: terminals
  - Edges: node X to nodes $a_1, \ldots, a_n$ for a rule $X \text{ -> } a_1 \ldots a_n$

Infinite languages

S -> S S

S -> (S)

S -> ( )

Backus-Naur Form (BNF)

- Meta-language for describing the syntax of a programming language

- Differences between BNF and production rules:
  - Non-terminals are enclosed in special brackets
  - All alternatives are grouped together and separated by '|'
  - The symbol '::=' is used to separate left from right
  - Full names, indicating the meaning of the strings being defined, are used for non-terminal symbols.

BNF for English

<Sentence> ::= *<NounPhrase>* *<Verb>* *<NounPhrase>*

<NounPhrase> ::= *<Article>* *<Noun>*

<Verb> ::= loves | hates | eats

<Noun> ::= dog | cat | rat

<Article> ::= a | the

BNF (cont.)

- BNF uses the following notations:
  - Non-terminals enclosed in `<` and `>` such as `<NP>`
  - Rules written as: `X ::= RHS`
    - `X` must be a non-terminal
    - `RHS` can be
      - A sequence of terminals and non-terminals, or
      - Sequences of terminals and non-terminals separated by the symbol | (meaning “or”)
    - If `RHS` is empty (i.e., length 0 sequence), we use ϵ or *empty*

A Grammar for Arithmetic Expressions

```
<expr> ::= <expr> + <expr> | <expr> * <expr> | (<expr>) | NUM
```

- A grammar for arithmetic expressions
  - Terminal: `+, *, (, ), NUM` (some number)
  - Non-terminal: `<expr>`
  - Example of production rule:

    ```
    <expr> ::= <expr> + <expr> | <expr> * <expr> | (<expr>) | NUM
    ```
  - Start non-terminal: `<expr>`
- Intuitively it is just a recursive definition:
  - NUM is an expression
  - The addition/multiplication of 2 expressions is also an expression
  - Parenthesized expressions are also expressions

Parse Tree

- Similar to what we defined for production rules
- What is the parse tree for 1+2*3 ?

Ambiguity

- A grammar is ambiguous if
  - A string has two different parse trees
  - Note: not derivations, but parse trees (why?)
- Back to our earlier example
  - Consider the string 1 + 2 * 3
  - It has two parse trees
    - Lack of precedence (* should be higher than +)
- There is also a problem with associativity
  - Consider the string 1 + 2 + 3

Revised Grammar

- Unambiguous grammar that expresses both precedence and associativity

```
  <expr>   ::= <expr> + <term>   |  <term>
  <term>   ::= <term> * <factor>   |  <factor>
<factor>   ::= (<expr>)  |  NUM
```

- New rule “term” to establish a “precedence cascade”
- First two rules left recursive -> left associativity
- Another example: 1 * (2 + 3)

Extended BNF (EBNF)

- The idea: adding short-hands to simplify productions
- Three main short-hands: repetition, optional, grouping

Extended BNF (EBNF)

- `{x}` 0 or more instances of `x` (repetition)
  - Example

```
<number> ::= <digit> | <number> <digit>
<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

becomes

```
<number> ::= <digit> {<digit>}
<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

Extended BNF (cont.)

- `[x]` 0 or 1 instance of `x` (optional)
  - Example

```
<if-stmt> ::= if <cond> then <stmt>
            |  if <cond> then <stmt> else <stmt>
```

becomes

```
<if-stmt> ::= if <cond> then <stmt> [else <stmt>]
```

Extended BNF (cont.)

- `(x)` parentheses used for grouping items together (grouping)
  - Example

```
<expr> ::= <expr> + <expr> | <expr> - <expr> | <num>
```

Becomes

```
<expr> ::= <expr> (+ | -) <expr> | <num>
```

Conversions between BNF and EBNF

- BNF → EBNF
  - Recursion in grammar

```
<A> ::= <A> a | <B>    =>
                                <A> ::= <B> { a }
```

  - Common string to factor out with grouping and options

```
<A> ::= a <B> | a      =>
                                <A> ::= a [<B>]

<A> ::= a <B> | a <C> =>
                                <A> ::= a (<B> | <C>)
```

Conversions between BNF and EBNF

- EBNF → BNF
  - Options: [ ]

```
<A> ::= a [<B>] <C>   =>
                                     <A> ::= a <C> | a <B> <C>
```

  - Repetition: { }

```
<A> ::= a { <B1> <B2> <Bn> } <C> =>
                                    <A> ::= <B> <C>
                                    <B> ::= <B> <B1> <B2> <Bn> | a
```

  - Grouping: ()

```
<A> ::= a (<B> | <C>) <D>   =>
                                    <A> ::= a <B> <D> | a <C> <D>
```

Language Generated by BNF

- Example BNF:

```
<s> ::= 0 0 0 <s> 1 | empty
```

- Language generated?
  - All strings with *3n* 0's followed by *n* 1's, for $n \in \mathbb{N}$

$$\{0^{3n}1^n : n \in \mathbb{N}\}$$

Give a BNF/EBNF for a Language

The set of strings consisting of the keyword **`begin`**, followed by one or more statements with a semicolon after each one, followed by the keyword **`end`**. Use the non-terminal **`<statement>`**, and do not give productions for it

- BNF

```
<s> ::= begin <statements> end
<statements> ::= <statements> <statement> ; | <statement>;
```

- EBNF

```
<s> ::= begin <statements> end
<statements> ::= <statement>; {<statement>;}
```

Non-context free languages?

- Ensure that there exists declarations before each use of a variable
- Ensure that the number of formal parameters match the number of actual parameters for each function

Solution?

- Move these checks to semantic analysis