Describing Syntax

Programming Languages

William Killian

Millersville University

Outline

- Definition of Syntax
- Defining Syntax
  - Context Free Grammars (CFG)
  - Backus-Naur Form (BNF)
  - Extended Backus-Naur Form (EBNF)
- Verifying Syntax
  - Derivations
  - Parse Trees
  - Ambiguity

Syntax

- the form or structure of the expressions, statements, and program units

```
x = 5;
```

is syntactically valid in languages like C, C++, Java

```
[1;2;3] |> List.map pow |> List.fold_left (+)
```

is syntactically valid in languages like Ocaml

```
f +. (0::[])
```

Just looks like nonsense

Defining Syntax

Defining Syntax

- Need some way to define the syntax of a language
  - Should be <u>*extendable*</u>
  - Should be <u>*easy to read*</u>
  - Should be applicable to <u>*all languages*</u>

- Solution:
  - Context-Free Grammars
  - Backus-Naur Form
  - Extended Backus-Naur Form

Context-Free Grammars

- Developed by **Noam Chomsky** in the mid-1950s
- Language generators, meant to describe the syntax of natural languages
- Define a class of languages called context-free languages
- Learn more about it in a Computational Models course (CSCI 340)

Backus-Naur Form

- Created by John Backus (1959) to describe the syntax of Algol 58
- Equivalent to context-free grammars

- Two High-Level Abstractions:
  - Terminals – tokens e.g. `for` `10` `::` `if` `int`
  - Non-Terminals – rules defining part of the language

Backus-Naur Form Rules

LHS → *RHS*

Nonterminal Combinations of terminals/nonterminals

```
<if_stmt> → if <logic_expr> then <stmt>
<ident_list> → identifier
             | identifier, <ident_list>
```

Backus-Naur Form Grammar

- **Grammar:** a finite non-empty set of rules
- A start symbol is a special element of the Nonterminals
  - Defines the “root” rule of the language

```
<program>   → <decl_list>
<decl_list> → <decl> <decl_list>
            | ε
```

- Epsilon (ε) means nothing
- Syntactic lists are described using recursion

Example BNF Grammar

```
<program> → <stmts>
<stmts>   → <stmt>
          | <stmt> ; <stmts>
<stmt>    → <var> = <expr>
<var>     → a | b | c | d
<expr>    → <term> + <term>
          | <term> - <term>
<term>    → <var> | const
```

Extended BNF

- Optional parts are placed in brackets [ ]

```
<proc_call> → ident [(<expr_list>)]
```

- Alternative parts of RHSs are placed inside parentheses and separated via vertical bars

```
<term> → <term> (+|-) const
```

- Repetitions (0 or more) are placed inside braces { }

```
<ident> → letter {letter|digit}
```

Extended BNF Comparison

BNF

```
<expr> → <expr> + <term>
       | <expr> - <term>
       | <term>
<term> → <term> * <factor>
       | <term> / <factor>
       | <factor>
```

EBNF

```
<expr> → <term>   {(+ | -) <term>}
<term> → <factor> {(* | /) <factor>}
```

Verifying Syntax

Verifying Syntax

- How can we show that a sequence of tokens match a grammar defined with BNF?

- **Option 1:** Generate all possible valid "sentences" in the grammar and see if it shows up
  - Good idea?
- **Option 2:** Intelligently expand rules and backtrack when we encounter an error.
  - When we match (and expanded all non-terminals): **Done**
  - If we exhaustively tried all options: **Fail**

Derivation

A derivation is a repeated application of rules, starting with the *start symbol* and ending with all *terminal tokens*

```
<program> => <stmts>
          => <stmt>
          => <var> = <expr>
          => a = <expr>
          => a = <term> + <term>
          => a = <var> + <term>
          => a = b + <term>
          => a = b + const
```

```
<program> → <stmts>
<stmts>   → <stmt>
          | <stmt> ; <stmts>
<stmt>    → <var> = <expr>
<var>     → a | b | c | d
<expr>    → <term> + <term>
          | <term> - <term>
<term>    → <var> | const
```

This is known as a *leftmost* derivation because we expanded the leftmost non-terminal each step

Derivation Example

Perform a left-most derivation

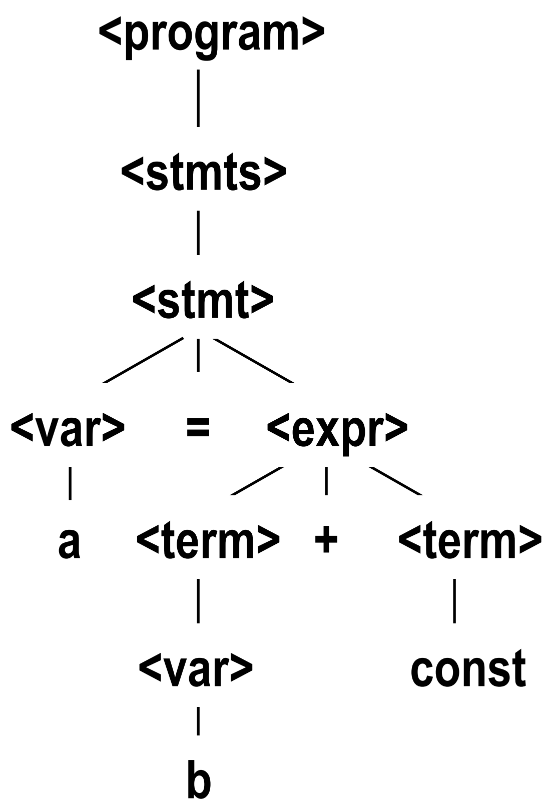
`a = 4; b = a + 1`

`<program>`

```
<program> → <stmts>
<stmts>   → <stmt>
            | <stmt> ; <stmts>
<stmt>    → <var> = <expr>
<var>     → a | b | c | d
<expr>    → <term> + <term>
            | <term> - <term>
<term>    → <var> | const
```

Parse Tree

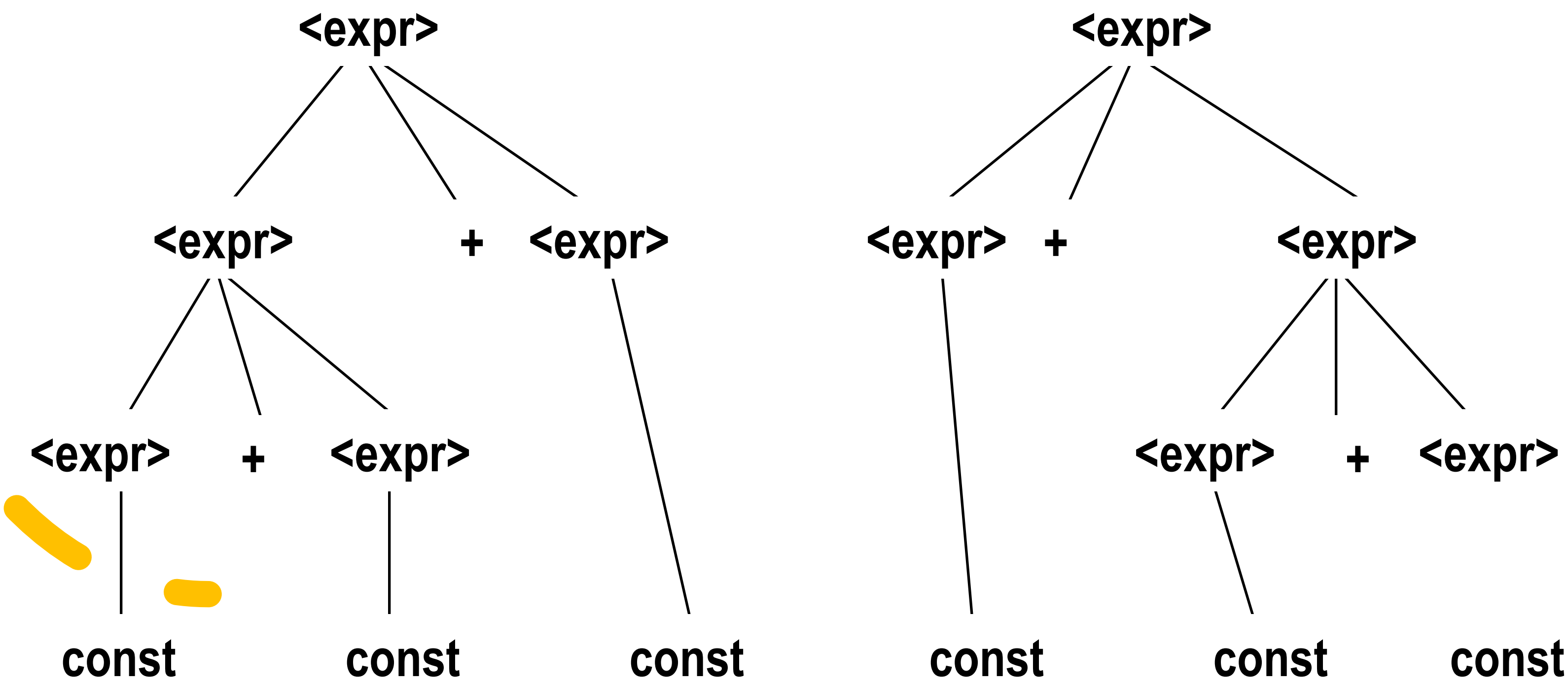
- A Graphical Representation of a derivation



Ambiguity

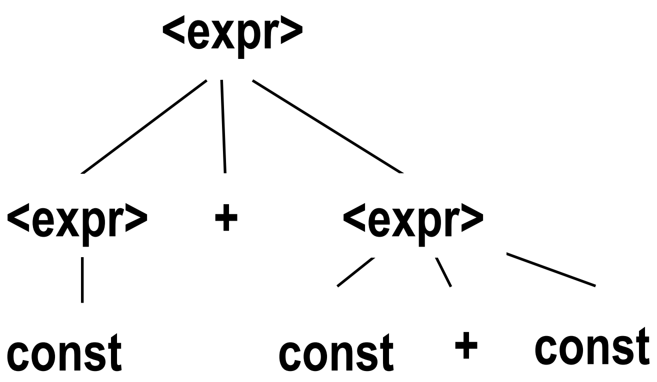
- A grammar is **ambiguous** if and only if it generates a sequence of tokens that has two or more distinct parse trees

```
<expr> → <expr> + <expr>
       | const
```



Ambiguity

- If we use the parse tree to indicate precedence levels and associativity of the operators, we cannot have ambiguity




```
<expr> → const + <expr>
       | const
```

Supporting Precedence

```
<expr> → <expr> - <term>
       | <term>
<term> → <term> / const
       | const
```