Describing Syntax and Semantics

Syntax and Semantics

syntax is the form (structure, grammar) of a language

semantics is the meaning of a language

```
if (a > b) a = a + 1;
else b = b + 1;
```

syntax: `if-else` is an operator that takes three operands - a condition and two statements

semantics: if the value of a is greater than the value of b, then increment a. Otherwise, increment b.

Both the syntax and semantics of a programming language must be carefully defined so that:

- Those that write compilers can implement the language (correctly), so that programs developed with one implementation run correctly under another (portability)
- Programmers can use the language (correctly)

Describing syntax

A **language** is a set of strings of characters from some alphabet.

English (using the standard alphabet)

binary numbers (using the alphabet {0, 1})

The syntax rules of a language determine whether or not arbitrary strings belong to the language.

A languages syntax must define the basic units or “words” of the language, called **lexemes**. A few Java lexemes include:

- `if`
- `++`
- `+`
- `3.27`
- `Count`

Lexemes and tokens

Lexemes are grouped into categories called **tokens**. Each token has one or more lexemes

Tokens are specified using regular expressions or finite automata

The scanner/lexical analyzer of a compiler processes the characters in the source code program and identifies the appropriate tokens

Once the tokens are identified the sequence of the tokens is analyzed to see if the order exists within the language

Sample tokens

token	sample lexemes
`identifier`	`count, i, x2, ...`
`if`	`if`
`add_op`	`+, -`
`mult_op`	`*, /, %`
`semi colon`	`;`
`int_literal`	`0, 10, -2.4`

Describing syntax with BNF

BNF is Backus Naur Form or Backup Normal Form

BNF is:

- a **metalanguage** - a language used to describe other languages
- the standard way to describe programming language syntax
- often used in language reference manuals

The class (set) of languages that can be described using BNF is called the **context-free languages**

BNF descriptions are also called **context-free grammars** or just grammars

Grammars

A **rule** has a left-hand side (LHS) and a right-hand side (RHS), and consists of **terminal** and **nonterminal** symbols

A grammar is a finite nonempty set of rules

An **abstraction** (or nonterminal symbol) *can* have more than one RHS

A derivation is a repeated application of rules, starting with the start symbol and ending with a sentence (all terminal symbols)

Eventually ALL nonterminals will be derived into terminals

Grammar Notation

→	is defined as
|	or (alternatives)
<something>	a nonterminal - replaced by the definition of <something> All nonterminals will eventually be replaced by a terminal
something (with no < >)	a token or terminal - a “word” in the language being defined
ε	the empty string (nothing)

Sample C grammar

```
<if-stmt> → if (<expr>) <stmt>
<if-stmt> → if (<expr>) <stmt> else <stmt>
```

Each definition above is called a rule or production. A full BNF definition would include definitions for all the nonterminals used - <expr> and <stmt> are not defined above

The two rules above can be abbreviated using | as follows:

```
   <if-stmt> → if (<expr>) <stmt>
             | if (<expr>) <stmt> else <stmt>
```

Example (expressions in C):

```
<expr> → id | num | (<expr>) | <expr> <op> <expr>
<op> → + | - | * | / | == | < | <= | > | >=
```

Note that BNF definitions can be recursive

Defintions

The symbol $\rightarrow$ is called **derives**

Each string of symbols derived from the start symbol (including the start symbol itself) is called a **sentential form**

A **leftmost derivation** is a derivation in which the leftmost nonterminal is always chosen for replacement

A **rightmost derivation** is a derivation in which the rightmost nonterminal is always chosen for replacement

Derivation order has no affect on the set of strings that can be derived

Derivations

A **derivation** is a sequence of replacements using the rules of a grammar.

- Always begin with the **start** symbol for the grammar
  - By *tradition*, the nonterminal on the LHS of the first rule is the start symbol **or** has an identifiable name such as `<start>` or `<program>`
- The derivation will show that a particular sequence of tokens belongs to the language
- The process ends when all nonterminals have been derived into terminals **AND** all input tokens have been used

to show that `id + num` is a valid expression:

```
<expr> → <expr> <op> <expr>
       → id <op> <expr>
       → id + <expr>
       → id + num
```

Sample derivation

To show that `id < (num / num)` is a valid expression:

```
<expr> → <expr> <op> <expr>
       → id <op> <expr>
       → id < <expr>
       → id < (<expr>)
       → id < (<expr> <op> <expr>)
       → id < (num <op> <expr>)
       → id < (num / <expr>)
       → id < (num / num)
```

Another sample grammar

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

A variation so that all individual statements <stmt> end with a semicolon

```
<stmts> -> <stmt> <stmts> | ε
<stmt> -> <var> = <expr> ;
```

What would be the derivation for a `= b + 42;`

Sample derivation

Sample leftmost derivation for

`a = b + 42`

```
                                    Token queue a = b + 42
<program> => <stmts>                            a = b + 42
          => <stmt>                             a = b + 42
          => <var> = <expr>                     a = b + 42
          => a = <expr>                         = b + 42
          => a = <term> + <term>                b + 42
          => a = <var> + <term>                 b + 42
          => a = b + <term>                     + 42
          => a = b + const (42)                 42
```

All nonterminals have been derived and the queue is empty

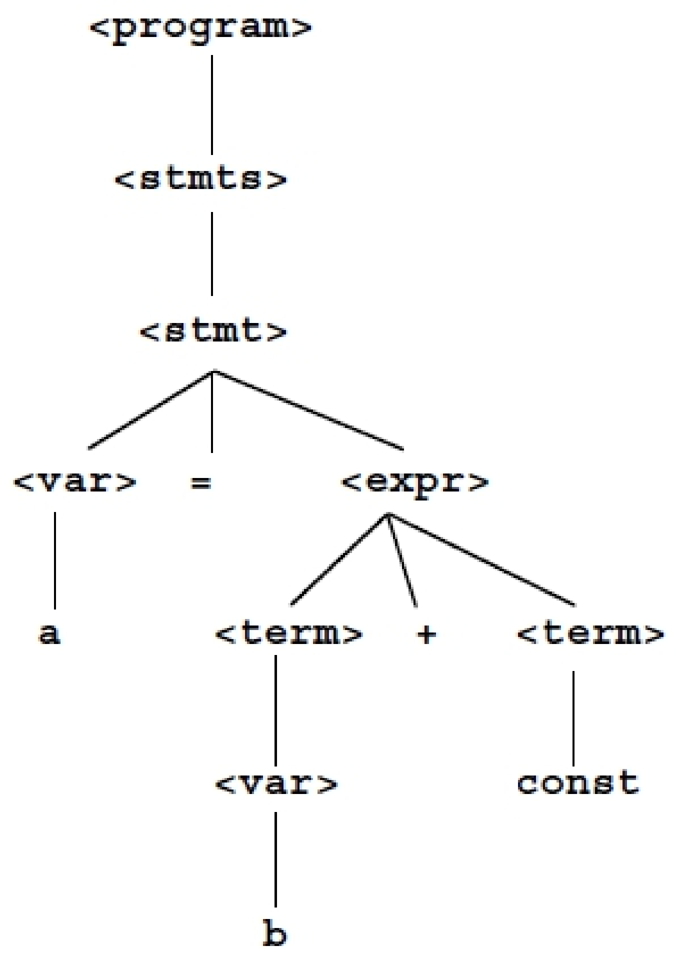
Sample derivation 2

```
Using the variation requiring a semicolon
<program> => <stmts>
          => <stmt> <stmts>
          => <var> = <expr> ;
          => a = <expr> ; <stmts>
          => a = <term> + <term> ; <stmts>
          => a = <var> + <term> ; <stmts>
          => a = b + <term> ; <stmts>
          => a = b + const (42) ; <stmts>
          => a = b + const (42) ; ε
```

Parse trees

A **parse tree** is a hierarchical representation of a derivation

A grammar is **ambiguous** if and only if it generates a sentential form that has two or more distinct parse trees

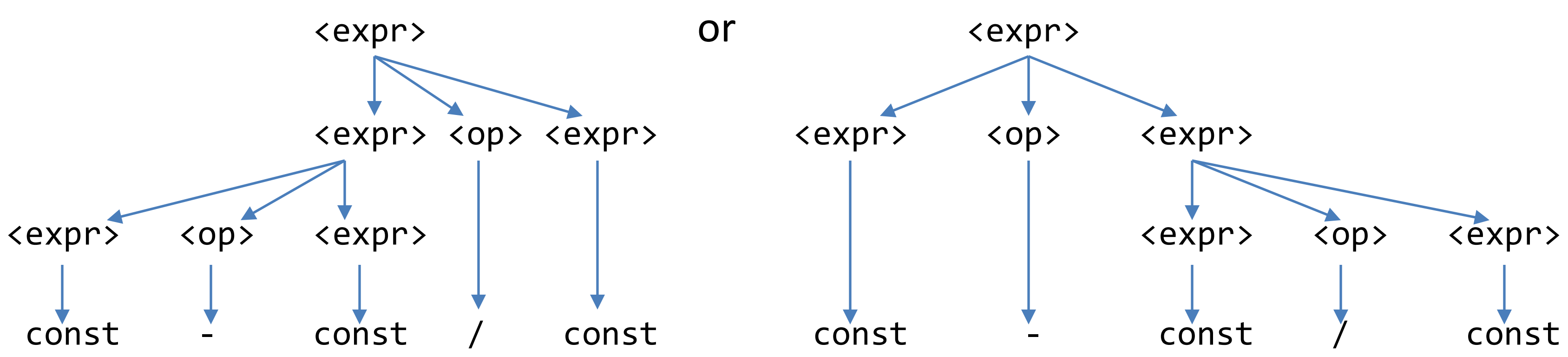


Ambiguous grammars

An ambiguous expression grammar:

```
<expr> -> <expr> <op> <expr> | const
<op> -> / | -
```

Parse *trees* for `const – const / const`



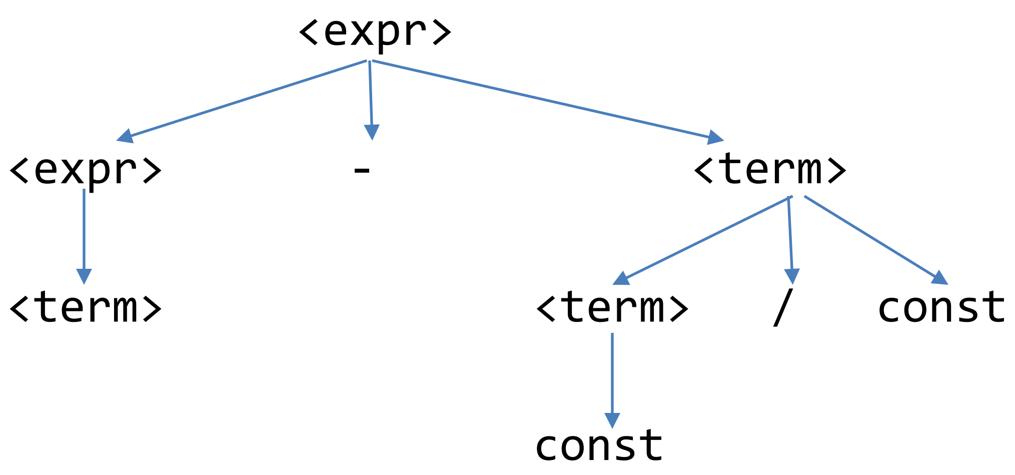
If we want the parse trees to indicate precedence levels of the operators we cannot have ambiguity

Unambiguous grammar

An unambiguous expression grammar:

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

Parse *tree* for `const – const / const`


$$\begin{tikzcd}
 & \text{<expr>} \arrow[dl] \arrow[d] \arrow[dr] & & & \\
\text{<expr>} \arrow[d] & \text{-} & \text{<term>} \arrow[dl] \arrow[d] \arrow[dr] & & \\
\text{<term>} & \text{<term>} \arrow[d] & \text{/} & \text{const} & \\
 & \text{const} & & &
\end{tikzcd}$$

Unambiguous expression grammar

```
expr    -> term | expr add_op term
term    -> factor | term mult_op factor
factor  -> id | number | -factor | ( expr )
add_op  -> + | -
mult_op -> * | / | %
```

This grammar captures associativity *and* precedence

Recursive Descent Parsing

Parsing is the process of tracing or constructing a parse tree for a given input string

Parsers usually do not analyze lexemes; that is done by a lexical analyzer, which is called by the parser

A recursive descent parser traces out a parse tree in top-down order; it is a top-down parser

Each nonterminal in the grammar has a subprogram associated with it; the subprogram parses all sentential forms that the nonterminal can generate

The recursive descent parsing subprograms are built directly from the grammar rules

Recursive descent parsers, like other top-down parsers, cannot be built from left-recursive grammars