CMPE 152: Compiler Design

October 17 Lab

Department of Computer Engineering
San Jose State University

Fall 2017
Instructor: Ron Mak

www.cs.sjsu.edu/~mak

Backus Naur Form (BNF)

- A text-based way to describe source language syntax.
  - Named after John Backus and Peter Naur.
- Text-based means it can be read by a program.
- Example: A *compiler-compiler* that can automatically generate a parser for a source language after reading (and parsing) the language's syntax rules written in BNF.

Backus Naur Form (BNF), *cont'd*

- Uses certain meta-symbols.
  - Symbols that are part of BNF itself but are not necessarily part of the syntax of the source language.

: :=	"is defined as"
|	"or"
< >	Surround names of nonterminal (not literal) items

BNF Example: U.S. Postal Address

```
<postal-address> ::= <name-part> <street-part> <city-state-part>
<name-part> ::= <first-part> <last-name>
              | <first-part> <last-name> <suffix>
<first-part> ::= <first-name> | <capital-letter> .
<suffix> ::= Sr. | Jr. | <roman-numeral>
<street-part> ::= <house-number> <street-name>
                | <house-number> <street-name> <apartment-number>
<city-state-part > ::= <city-name> , <state-code> <ZIP-code>
<first-name> ::= <name>
<last-name> ::= <name>
<street-name> ::= <name>
<city-name> ::= <name>
<house-number> ::= <number>
<apartment-number> ::= <number>
<state-code> ::= <capital-letter> <capital-letter>
<capital-letter> ::= A|B|C|D|E|F|G|H|I|J|K|L|M
                   |N|O|P|Q|R|S|T|U|V|W|X|Y|Z
<name> ::= …
<number> ::= …
```

etc.

Mary Jane
123 Easy Street
San Jose, CA 95192

BNF: Optional and Repeated Items

- To show optional items in BNF, use the vertical bar |.
- Example: "An expression is a simple expression optionally followed by a relational operator and another simple expression."

```
<expression> ::=
    <simple expression>
        | <simple expression> <rel op> <simple expression>
```

BNF: Optional and Repeated Items

- BNF uses recursion for repeated items.
- Example: "A digit sequence is a digit followed by zero or more digits."

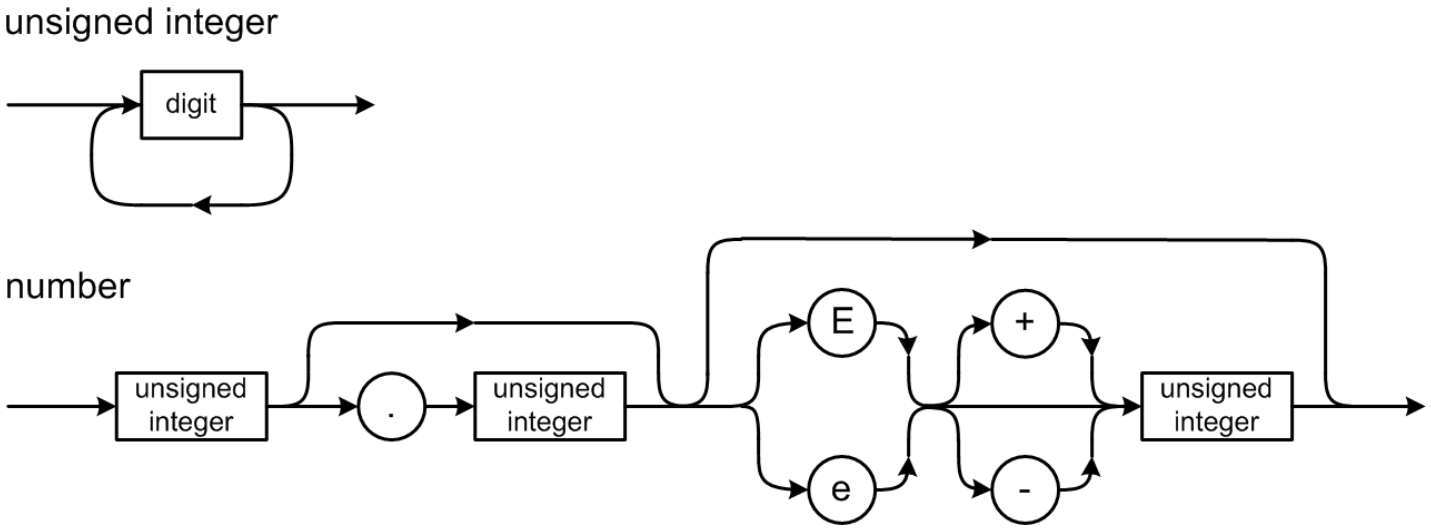
```
<digit sequence> ::= <digit>
                   | <digit> <digit sequence>
```

Right recursive

```
<digit sequence> ::= <digit>
                   | <digit sequence> <digit>
```

Left recursive

BNF Example: Pascal Number

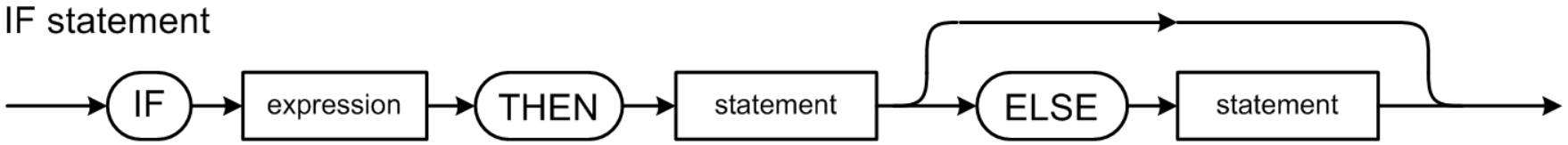



```
<digit sequence> ::= <digit>
                   | <digit> <digit sequence>
<unsigned integer> ::= <digit sequence>
<unsigned real> ::= <unsigned integer>.<digit sequence>
                  | <unsigned integer>.<digit sequence> <e> <scale factor>
                  | <unsigned integer > <e> <scale factor>
<scale factor> ::= <unsigned integer> | <sign> <unsigned integer>
<e> ::= E | e
<sign> ::= + | -
<number> ::= <unsigned integer> | <unsigned real>
```

Repetition via recursion.

The sign is optional.

BNF Example: Pascal **`IF`** Statement




```
<if statement> ::= IF <expression> THEN <statement>
                 | IF <expression> THEN <statement> ELSE <statement>
```

- It should be straightforward to write a parsing method from either the syntax diagram or the BNF.

Grammars and Languages

- A grammar defines a language.
- Grammar = the set of all the BNF rules (or syntax diagrams)
- Language = the set of all the legal strings of tokens according to the grammar
- Legal string of tokens = a syntactically correct statement

Grammars and Languages

- A statement is in the language (it's syntactically correct) if it can be derived by the grammar.
- Each grammar rule "produces" a token string in the language.
- The sequence of productions required to arrive at a syntactically correct token string is the derivation of the string.

Grammars and Languages, *cont'd*

- Example: A very simplified expression grammar:

```
<expr> ::= <digit>
         | <expr> <op> <expr>
         | ( <expr> )
<digit> ::= 0|1|2|3|4|5|6|7|8|9
<op> ::= + | *
```

- What strings (expressions) can we derive from this grammar?

Derivations and Productions

- Is **(1 + 2)*3** valid in our expression language?

PRODUCTION	GRAMMAR RULE
<expr> ➔ <expr> <op> <expr>	<expr> ::= <expr> <op> <expr>
➔ <expr> <op> <digit>	<expr> ::= <digit>
➔ <expr> <op> 3	<digit> ::= 3
➔ <expr>*3	<op> ::= *
➔ (<expr>)*3	<expr> ::= (<expr>)
➔ (<expr> <op> <expr>)*3	<expr> ::= <expr> <op> <expr>
➔ (<expr> <op> <digit>)*3	<expr> ::= <digit>
➔ (<exp> <op> 2)*3	<digit> ::= 2
➔ (<expr> + 2)*3	<op> ::= +
➔ (<digit> + 2)*3	<expr> ::= <digit>
➔ (1 + 2)*3	<digit> ::= 1

- Yes! The expression is valid.

```
<expr> ::= <digit>
         | <expr> <op> <expr>
         | ( <expr> )
<digit> ::= 0|1|2|3|4|5|6|7|8|9
<op> ::= + | *
```

Extended BNF (EBNF)

- Extended BNF (EBNF) adds meta-symbols **{ }** and **[ ]**

{ }	Surround items to be repeated zero or more times.
[ ]	Surround optional items.

- Originally developed by Niklaus Wirth.
  - Inventor of Pascal.
  - Early user of syntax diagrams.

Extended BNF (EBNF)

- Repetition (one or more):
  - BNF:

```
<digit sequence> ::= <digit>
                   | <digit> <digit sequence>
```

  - EBNF:

```
<digit sequence> ::= <digit> { <digit> }
```

Extended BNF, *cont'd*

- Optional items.
  - BNF:

```
<if statement> ::= IF <expression> THEN <statement>
                 | IF <expression> THEN <statement>
                                  ELSE <statement>
```

  - EBNF:

```
<if statement> ::= IF <expression> THEN <statement>
                                  [ ELSE <statement> ]
```

Extended BNF, *cont'd*

- Optional items.
  - BNF:

```
<expression> ::= <simple expression>
                 | <simple expression>
                        <rel op> <simple expression>
```

  - EBNF:

```
<expression> ::= <simple expression>
                        [ <rel op> <simple expression> ]
```