CS 153: Concepts of Compiler Design

October 19 Class Meeting

Department of Computer Science
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."

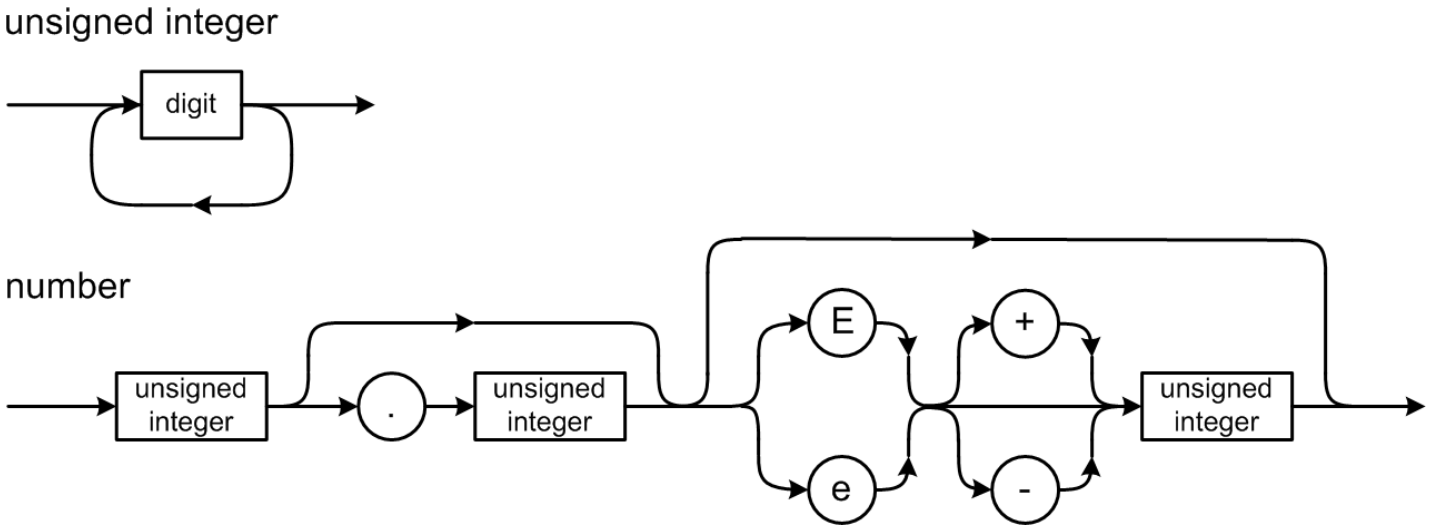
```
<digit sequence> ::= <digit>
                   | <digit> <digit sequence>
```

Right recursive

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

Left recursive

BNF Example: Pascal Number

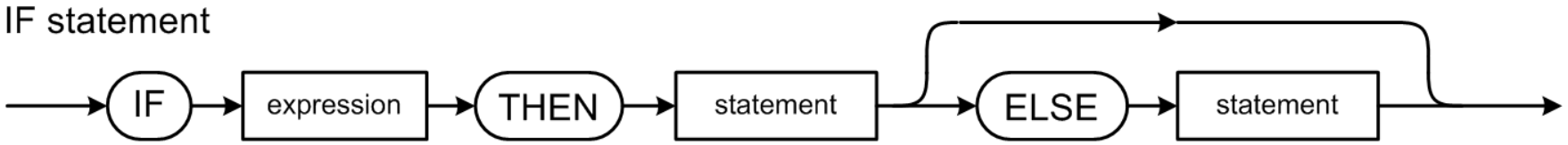



```
<digit sequence> ::= <digit>
                   | <digit> <digit sequence>
<unsigned integer> ::= <digit sequence>
<unsigned real> ::= <unsigned integer>.<digit sequence>
                  | <unsigned integer>.<digit sequence> <e> <scale factor>
                  | <unsigned integer > <e> <scale 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> ]
```

Compiler-Compilers

- Professional compiler writers generally do not write scanners and parsers from scratch.
- A compiler-compiler is a tool for writing compilers.
- It can include:
  - A scanner generator
  - A parser generator
  - Parse tree utilities

Scanners and Parsers

- The scanner is the part of the compiler that reads the source program and breaks it apart into tokens.
- The parser figures out the structure of each source program statements and determines which statements (assignment, if, while, etc.).
- The parser repeatedly calls the scanner to read and return the next token.

Compiler-Compilers, *cont'd*

- Feed a compiler-compiler a grammar written in a textual form such as BNF or EBNF.
- The compiler-compiler generates a scanner, parser, and a parse tree utilities implemented in a high-level language.
  - Such as Java and C++

Popular Compiler-Compilers

- Yacc
  - "Yet another compiler-compiler"
  - Generates a bottom-up parser written in C.
  - GNU version: Bison
- Lex
  - Generates a scanner written in C.
  - GNU version: Flex
- JavaCC
  - Generates a scanner and a top-down parser written in Java.
- ANTLR4
  - Generates a scanner, parser, and parse tree routines written in Java or C++.

The code generated by a compiler-compiler can be in a high level language such as Java or C++. However, you may find the code to be ugly and hard to read.

ANTLR 4 Compiler-Compiler

- Feed ANTLR 4 the grammar for a source language and it will automatically generate a scanner and a parser.
  - Define the source language's tokens with regular expressions
    - ➔ ANTLR 4 generates a scanner for the source language.
  - Specify the grammar's productions with BNF
    - ➔ ANTLR 4 generates a parser for the source language.

ANTLR 4 Compiler-Compiler, *cont'd*

- The generated scanner and parser are written in Java or C++.
  - ANTLR calls the scanner a "lexer".
- A command-line option specifies C++, otherwise the default implementation language is Java.
- Some of the tools only work with the Java code, so even if you want C++ code, you may want also to generate the Java code.

Download and Install

- Download and install ANTLR 4:
  http://www.antlr.org/

- ANTLR 4 book:
  The Definitive ANTLR 4 Reference
  by Terence Parr
  https://www.amazon.com/Definitive-ANTLR-4-Reference/dp/1934356999/ref=sr_1_1?s=books&ie=UTF8&qid=1508372536&sr=1-1&keywords=antlr4

ANTLR 4 Plug-ins for IDEs

- http://www.antlr.org/tools.html
- Eclipse plugin: https://github.com/antlr4ide/antlr4ide
  - Be sure to follow all the instructions for creating a Java-based ANTLR 4 project in Eclipse.

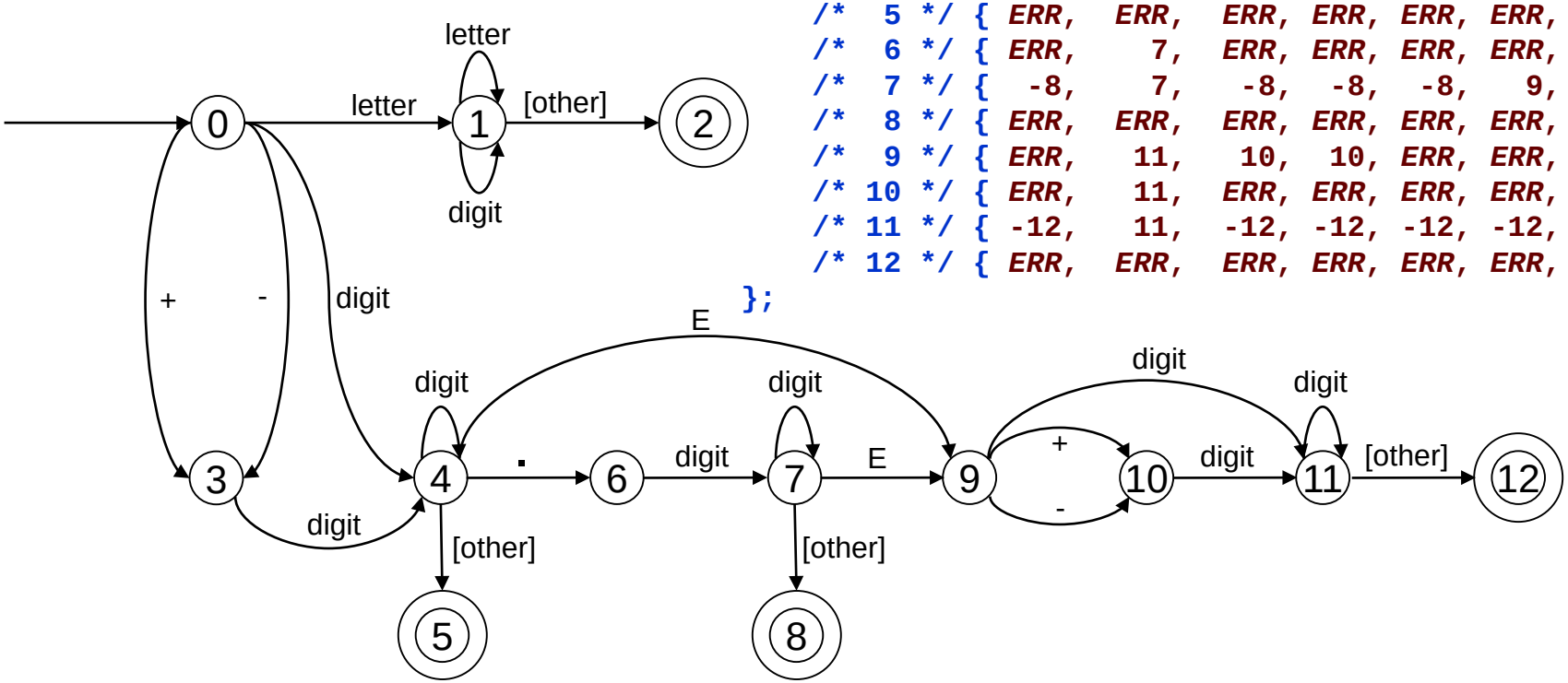
Review: DFA for a Pascal Identifier or Number

Negative numbers in the matrix are **accepting states**.

```
private static final int matrix[][] = {

/*         letter digit   +    -    .    E other */
/*  0 */ {   1,    4,    3,   3, ERR,   1, ERR },
/*  1 */ {   1,    1,   -2,  -2,  -2,   1,  -2 },
/*  2 */ { ERR,  ERR,  ERR, ERR, ERR, ERR, ERR },
/*  3 */ { ERR,    4,  ERR, ERR, ERR, ERR, ERR },
/*  4 */ {  -5,    4,   -5,  -5,   6,   9,  -5 },
/*  5 */ { ERR,  ERR,  ERR, ERR, ERR, ERR, ERR },
/*  6 */ { ERR,    7,  ERR, ERR, ERR, ERR, ERR },
/*  7 */ {  -8,    7,   -8,  -8,  -8,   9,  -8 },
/*  8 */ { ERR,  ERR,  ERR, ERR, ERR, ERR, ERR },
/*  9 */ { ERR,   11,   10,  10, ERR, ERR, ERR },
/* 10 */ { ERR,   11,  ERR, ERR, ERR, ERR, ERR },
/* 11 */ { -12,   11,  -12, -12, -12, -12, -12 },
/* 12 */ { ERR,  ERR,  ERR, ERR, ERR, ERR, ERR },

};
```



The ANTLR Lexer

- The ANTLR-generated lexer (scanner) is a DFA created from the regular expressions in the grammar file that describe the tokens.

Example ANTLR Grammar File

Expr.g6

```
grammar Expr;

/** The start rule; begin parsing here. */
prog:   stat+ ;

stat:   expr NEWLINE
    |   ID '=' expr NEWLINE
    |   NEWLINE
    ;

expr:   expr ('*'|'/') expr
    |   expr ('+'|'-') expr
    |   INT
    |   ID
    |   '(' expr ')'
    ;

ID  :   [a-zA-Z]+ ;      // match identifiers <label id="code.tour.expr.3"/>
INT :   [0-9]+ ;         // match integers
NEWLINE:'\r'? '\n' ;     // return newlines to parser (is end-statement signal)
WS  :   [ \t]+ -> skip ; // toss out whitespace
```

t.expr

```
193
a = 5
b = 6
a+b*2
(1+2)*3
```

Java Command Line

```
Expr-Java$ ls
Expr.g4 t.expr
Expr-Java$ antlr4 Expr.g4
Expr-Java$ ls
Expr.g4 ExprBaseListener.java
ExprLexer.tokens ExprParser.java
Expr.tokens ExprLexer.java ExprListener.java t.expr
Expr-Java$ javac Expr*.java
Expr-Java$ grun Expr prog -gui t.expr
```

- **grun** runs a test harness for the grammar.

A Java Main Program for ANTLR

ExprJoyRide.java

```
public class ExprJoyRide
{
    public static void main(String[] args) throws Exception
    {
        String inputFile = null;
        if (args.length > 0) inputFile = args[0];

        InputStream is = System.in;
        if (inputFile != null) is = new FileInputStream(inputFile);

        ANTLRInputStream input = new ANTLRInputStream(is);
        ExprLexer lexer = new ExprLexer(input);

        CommonTokenStream tokens = new CommonTokenStream(lexer);

        System.out.println("Tokens:");
        tokens.fill();
        for (Token token : tokens.getTokens())
        {
            System.out.println(token.toString());
        }

        ExprParser parser = new ExprParser(tokens);
        ParseTree tree = parser.prog();

        System.out.println("\nParse tree (Lisp format):");
        System.out.println(tree.toStringTree(parser));
    }
}
```

Print the list of tokens.

Print the parse tree in Lisp format.

Demo

C++ Command Line

```
Expr-Cpp$ ls
Expr.g4 t.expr
Expr-Cpp$ antlr4 -Dlanguage="Cpp" Expr.g4
Expr-Cpp$ ls
Expr.g4 ExprBaseListener.h ExprLexer.tokens ExprParser.cpp
Expr.tokens ExprLexer.cpp ExprListener.cpp ExprParser.h
ExprBaseListener.cpp ExprLexer.h ExprListener.h t.expr
```

- Unfortunately, **`grun`** only works with Java code.

A C++ Main Program for ANTLR

ExprMain.cpp

```
int main(int, const char **)
{
    ifstream ins;
    ins.open("t.expr");

    ANTLRInputStream input(ins);
    ExprLexer lexer(&input);
    CommonTokenStream tokens(&lexer);

    cout << "Tokens:" << endl;
    tokens.fill();
    for (Token *token : tokens.getTokens())
    {
        std::cout << token->toString() << std::endl;
    }

    ExprParser parser(&tokens);
    tree::ParseTree *tree = parser.prog();

    cout << endl << "Parse tree (Lisp format):" << endl;
    std::cout << tree->toStringTree(&parser) << endl;

    return 0;
}
```

Print the list of tokens.

Print the parse tree in Lisp format.

Demo

Compiler Team Project

- Write a compiler for that will generate code for the Java virtual machine (JVM).
- The source language should be a procedural, non-object-oriented language.
  - A language that the team invents (highly recommended option!)
  - A subset of an existing language.
    - Example: Small C (https://en.wikipedia.org/wiki/Small-C)
  - Tip: Start with a simple language!
  - No Scheme, Lisp, or Lisp-like languages.

Compiler Team Project

- The object language must be Jasmin, the assembly language for the Java virtual machine.
  - You will be provided an assembler that translates Jasmin assembly language programs into .class files for the JVM.
- You must use the ANTLR 4 compiler-compiler.
- You can also use any Java code from the **Writing Compilers and Interpreters** book.
- Compile and run source programs written in your language.

Compiler Team Project Deliverables

- The source files of a working compiler.
  - Java or C++ source files
  - The ANTLR **`.g4`** grammar files.
  - Do not include the Java or C++ files that ANTLR generated.

Compiler Team Project Deliverables, *cont'd*

- Written report (5-10 pp. single spaced)
  - Include: Syntax diagrams
    for key source language constructs.
  - Include: Code templates
    for key source language constructs.

Compiler Team Project Deliverables, *cont'd*

- Instructions on how to build your compiler.
  - If it's not standard or not obvious.
- Instructions on how to run your compiler (scripts OK).
- Sample source programs written in your language to compile and execute.
- Sample output from executing your source programs.

Post Mortem Report

- Private individual post mortem report (up to 1 page from each student)
  - What did you learn from this course?
  - An assessment of your accomplishments for your project.
  - An assessment of each of your project team members.
- Private: To be read only by the instructor.