Defining Program Syntax
Syntax And Semantics

- **Programming language syntax**: how programs look, their form and structure
  - Syntax is defined using a formal grammar
- **Programming language semantics**: what programs do, their behavior and meaning
  - Semantics is harder to define
Outline

- Grammar and parse tree examples
- BNF and parse tree definitions
- Constructing grammars
- Phrase structure and lexical structure
- Other grammar forms
An English Grammar

A sentence $S$ is a noun phrase $NP$, a verb $V$, and a noun phrase $NP$.

$S ::= NP \ V \ NP$

A noun phrase $NP$ is an article $A$ and a noun $N$.

$NP ::= A \ N$

A verb $V$ is... $V ::= loves | hates | eats$

An article $A$ is... $A ::= a | the$

A noun $N$ is... $N ::= dog | cat | rat$
How The Grammar Works

• The grammar is a set of rules that say how to build a tree—a *parse tree*
• `<S>` at the root of the tree
• The grammar’s rules define how children can be added at any point in the tree
• For instance,  

```
<S> ::= <NP> <V> <NP>
```

defines nodes `<NP>`, `<V>`, and `<NP>`, in that order, as children of `<S>`
Parse
Derivation

One derivation that $\langle S \rangle = \textbf{the dog loves the cat}$ is produced by the grammar rules:

$$
\begin{align*}
\langle S \rangle &= \langle NP \rangle \langle V \rangle \langle NP \rangle \\
&= \langle A \rangle \langle N \rangle \langle V \rangle \langle NP \rangle \\
&= \langle A \rangle \langle N \rangle \langle V \rangle \langle A \rangle \langle N \rangle \\
&= \langle A \rangle \langle N \rangle \text{loves} \langle A \rangle \langle N \rangle \\
&= \text{the} \langle N \rangle \text{loves} \langle A \rangle \langle N \rangle \\
&= \text{the dog loves} \langle A \rangle \langle N \rangle \\
&= \text{the dog loves the} \langle N \rangle \\
&= \text{the dog loves the cat}
\end{align*}
$$
Parse Tree: the dog loves the cat

\[
\begin{align*}
\text{<S> ::= <NP> <V> <NP>} \\
\text{<NP> ::= <A> <N>} \\
\text{<V> ::= loves | hates | eats} \\
\text{<A> ::= a | the} \\
\text{<N> ::= dog | cat | rat}
\end{align*}
\]

\[
\begin{align*}
\text{<S> = <NP> <V> <NP>} \\
\text{= <A> <N> loves <A> <N>} \\
\text{= the dog loves the cat}
\end{align*}
\]
Exercise 1

1. Which of the following are valid <S>?
   - the dog hates the dog
   - dog loves the cat
   - loves the dog the cat

2. Parse:
   - a cat eats the rat
   - the dog loves cat
Outline

- Grammar and parse tree examples
- BNF and parse tree definitions
- Constructing grammars
- Phrase structure and lexical structure
- Other grammar forms
BNF Grammar Definition

- *Backus Naur Form* grammar consists of four parts:
  - The set of *tokens*
  - The set of *non-terminal symbols*
  - The *start symbol*
  - The set of *productions*
BNF Grammar Definitions Explained

- **Start symbol**: `<S>` ::= `<NP>` `<V>` `<NP>`
- **A production**: `<NP>` ::= `<A>` `<N>`
- **Non-terminal symbols**: `<V>` ::= `loves` | `hates` | `eats`
- `<A>` ::= `a` | `the`
- `<N>` ::= `dog` | `cat` | `rat`
- **Tokens**
Definition, Continued

- The *tokens* are the smallest units of syntax
  - Strings of one or more characters of program text
  - They are atomic: not treated as being composed from smaller parts
- The *non-terminal symbols* stand for larger pieces of syntax
  - They are strings enclosed in angle brackets, as in `<NP>`
  - They are not strings that occur literally in program text
  - The grammar says how they can be expanded into strings of tokens
- The *start symbol* is the particular non-terminal that forms the root of any parse tree for the grammar
Definition, Continued

• The *productions* are the tree-building rules

• Each one has a left-hand side, the separator `::=`, and a right-hand side
  
  - The left-hand side is a single non-terminal
  
  - The right-hand side is a sequence of one or more things, each of which can be either a token or a non-terminal

• A production gives one possible way of building a parse tree: it permits the non-terminal symbol on the left-hand side to have the symbols on the right-hand side, in order, as its children in a parse tree
Alternatives (OR)

- When there is more than one production with the same left-hand side, an abbreviated form can be used.
- In BNF grammar:
  - Gives the left-hand side (symbol),
  - the separator ::=,
  - and then a list of possible right-hand sides separated by the special symbol |
Example

\[ <exp> ::= <exp> + <exp> | <exp> * <exp> | ( <exp> ) | a | b | c \]

Note that there are six productions in this grammar. It is equivalent to this one:

\[
\begin{align*}
<exp> & ::= <exp> + <exp> \\
<exp> & ::= <exp> * <exp> \\
<exp> & ::= ( <exp> ) \\
<exp> & ::= a \\
<exp> & ::= b \\
<exp> & ::= c
\end{align*}
\]
Empty

- The special non-terminal `<empty>` is for places where you want the grammar to generate nothing.
- For example, this grammar defines a typical if-then construct with an optional else part:

\[
\begin{align*}
<\text{if-stmt}> & \ ::= \text{if } <\text{expr}> \text{ then } <\text{stmt}> <\text{else-part}> \\
<\text{else-part}> & \ ::= \text{else } <\text{stmt}> | <\text{empty}>
\end{align*}
\]
Grammar Parse Derivation

- Begin with a start symbol
- Choose a production with start symbol on left-hand side
- Replace start symbol with the right-hand side of that production

1. Choose a non-terminal $S$ in resulting string
2. Choose a production $P$ with non-terminal $S$ on its left-hand side
3. Replace $S$ with the right-hand side of $P$
4. Repeat process until no non-terminals remain.

$\langle S \rangle ::= \langle NP \rangle \ \langle V \rangle \ \langle NP \rangle$
$\langle NP \rangle ::= \langle A \rangle \ \langle N \rangle$
$\langle V \rangle ::= \text{loves} \ | \ \text{hates} \ | \ \text{eats}$
$\langle A \rangle ::= \text{a} \ | \ \text{the}$
$\langle N \rangle ::= \text{dog} \ | \ \text{cat} \ | \ \text{rat}$

a cat eats the rat

$\langle S \rangle = \langle NP \rangle \ \langle V \rangle \ \langle NP \rangle$
$\quad = \langle A \rangle \ \langle N \rangle \ \langle V \rangle \ \langle NP \rangle$
$\quad = \langle A \rangle \ \langle N \rangle \ \langle V \rangle \ \langle A \rangle \ \langle N \rangle$
$\quad = \langle A \rangle \ \langle N \rangle \ \text{eats} \ \langle A \rangle \ \langle N \rangle$
$\quad = \ \text{a} \ \langle N \rangle \ \text{eats} \ \langle A \rangle \ \langle N \rangle$
$\quad = \ \text{a} \ \text{cat} \ \text{eats} \ \langle A \rangle \ \langle N \rangle$
$\quad = \ \text{a} \ \text{cat} \ \text{eats} \ \text{the} \ \langle N \rangle$
$\quad = \ \text{a} \ \text{cat} \ \text{eats} \ \text{the} \ \text{rat}$
Parse Trees

- To build a parse tree, put the start symbol at the root.
- Add children to every non-terminal, following any one of the productions for that non-terminal in the grammar.
- Done when all the leaves are tokens.
- Read off leaves from left to right—that is the string derived by the tree.

\[
\begin{align*}
\text{\(<S> \ ::= \ <NP> \ <V> \ <NP>\)} \\
\text{\(<NP> \ ::= \ <A> \ <N>\)} \\
\text{\(<V> \ ::= \ loves | hates | eats\)} \\
\text{\(<A> \ ::= \ a | the\)} \\
\text{\(<N> \ ::= \ dog | cat | rat\)}
\end{align*}
\]

\[
\begin{align*}
\text{\(<S> \ = \ a \ cat \ eats \ the \ rat\)}
\end{align*}
\]
A Programming Language Grammar

\[ <exp> ::= <exp> + <exp> | <exp> * <exp> | ( <exp> ) | a | b | c \]

- An expression can be:
  - the sum of two expressions,
  - or the product of two expressions,
  - or a parenthesized subexpression,
  - or \( a \),
  - or \( b \),
  - or \( c \)
Parse and Parse Tree: \( a + b \times c \)

\[
\begin{align*}
\langle \text{exp} \rangle & = \langle \text{exp} \rangle + \langle \text{exp} \rangle \\
& = a + \langle \text{exp} \rangle \\
& = a + \langle \text{exp} \rangle \times \langle \text{exp} \rangle \\
& = a + b \times c
\end{align*}
\]

\[
\text{exp} ::= \text{exp} + \text{exp} \mid \text{exp} \times \text{exp} \mid (\text{exp}) \mid a \mid b \mid c
\]
Parse and Parse Tree: \(((a+b)\times c)\)

\[
<\text{exp}> = ( ( <\text{exp}> ) * <\text{exp}> ) \\
= ( ( ( <\text{exp}> ) * <\text{exp}> ) ) \\
= ( ( ( <\text{exp}> ) * c ) ) \\
= ( ( ( <\text{exp}> + <\text{exp}> ) * c ) ) \\
= ( ( a + b ) * c )
\]

\[
<\text{exp}> ::= <\text{exp}> + <\text{exp}> | <\text{exp}> * <\text{exp}> | ( <\text{exp}> ) | a | b | c
\]
Exercise 2

1. Parse each of these strings:
   a. $a+b$
   b. $a*b+c$
   c. $(a+b)*c$

2. Give the parse tree for each of these strings:
   a. $a+b$
   b. $a*b+c$
   c. $(a+b)*c$
Compiler Note

- What we just did is parsing: trying to find a parse tree for a given string
- That’s what compilers do for every program you try to compile: try to build a parse tree for your program, using the grammar for whatever language you used
- Take a course in compiler construction to learn about algorithms for doing this efficiently
Language Definition

- We use grammars to define the syntax of programming languages.
- The language defined by a grammar is the set of all strings that can be derived by some parse tree for the grammar.
- As in the previous example, that set is often infinite (though grammars are finite).
- Constructing grammars is a little like programming...
Outline

• Grammar and parse tree examples
• BNF and parse tree definitions
• Constructing grammars
• Phrase structure and lexical structure
• Other grammar forms
Constructing Grammars

- Most important trick: divide and conquer
- Example: the language of Java declarations:
  - a type name,
  - a list of variables separated by commas,
  - and a semicolon
- Each variable can optionally be followed by an initializer:

```java
float a;
boolean a, b, c;
int a = 1, b, c = 1 + 2;
```
Example, Continued

• Easy if we postpone defining the comma-separated list of variables with initializers:

\[
\text{int } a=1, \ b, \ c=1+2;
\]

• Primitive type names are easy enough too:

\[
\text{<var-dec> } ::= \text{<type-name> <declarator-list> ;}
\]

• (Note: skipping constructed types: class
names, interface names, and array types)
Example, Continued

- That leaves the comma-separated list of variables with initializers
- Again, postpone defining variables with initializers, and just do the comma-separated list part:

```c
int a=1, b, c=1+2;
```

```latex
\textit{var-dec} ::= \textit{type-name} \textit{declarator-list} ;
\textit{declarator-list} ::= \textit{declarator} |
                                 \textit{declarator} \textit{declarator-list}
```
Example, Continued

```
int a=1, b, c=1+2;
```

- That leaves the variables with initializers:

```
<var-dec> ::= <type-name> <declarator-list> ;
<declarator-list> ::= <declarator>
| <declarator>, <declarator-list>
<declarator> ::= <variable-name>
| <variable-name> = <expr>
```

- For full Java, we would need to allow pairs of square brackets after the variable name
- There is also a syntax for array initializers
- And definitions for `<variable-name>` and `<expr>`
Grammar Construction Example

Construct a grammar in BNF for each language:

1. \(<\text{digit}\>>\) as a character 0-9.
   
   \[
   \text{<digit>} ::= 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9
   \]

2. \(<\text{unsigned}\>>\) as the set of all strings with one or more \(<\text{digit}\>>\). Note the left-recursion.
   
   \[
   \text{<unsigned>} ::= \text{<digit>} \mid \text{<unsigned> <digit>}
   \]

3. \(<\text{signed}\>>\) as the set of all strings starting with – or + and followed by an \(<\text{unsigned}\>>\).
   
   \[
   \text{<signed>} ::= +\text{<unsigned>} \mid -\text{<unsigned>}
   \]
Exercise 3

Construct a grammar in BNF for each language:

1. `<integer>` as the set of all strings of `<signed>` or `<unsigned>`.
2. `<decimal>` as the set of all strings of `<integer>` followed by a ‘.’ and optionally followed by an `<unsigned>`.
3. `<2or3digits>` as the set of all strings of two or three `<digit>`.
4. `<AdigitB>` as the set of all strings beginning with ‘A’ and followed by a `<digit>` or a ‘B’.
5. `<1+2’s>` as the set of all strings beginning with ‘1’ and followed by any number of 2’s.
6. `<2’s+1>` as the set of all strings beginning with any number of 2’s and followed by a ‘1’.
7. `<AdigitBs>` as the set of all strings beginning with ‘A’ and optionally followed by any number of `<digit>` or ‘B’.

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

<unsigned> ::= <digit> | <unsigned> <digit>

<signed> ::= +<unsigned> | -<unsigned>
```
Outline

• Grammar and parse tree examples
• BNF and parse tree definitions
• Constructing grammars
• Phrase structure and lexical structure
• Other grammar forms
Where Do Tokens Come From?

Tokens are pieces of program text that we choose not to think of as being built from smaller pieces

- Identifiers (\texttt{count}), keywords (\texttt{if}), operators (\texttt{==}), constants (\texttt{123.4}), etc.

- Programs stored in files are just sequences of characters

- How is such a file divided into a sequence of tokens?

\[
\texttt{digit} ::= 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9
\]

\[
\texttt{unsigned} ::= \texttt{digit} \mid \texttt{unsigned} \texttt{digit}
\]
Lexical Structure And Phrase Structure

- **Phrase structure**: how a program is built from a sequence of tokens

\[
\text{<if-stmt> ::= if <expr> then <stmt> <else-part>}
\]
\[
\text{<else-part> ::= else <stmt> | <empty>}
\]

- **Lexical structure**: how tokens are built from a sequence of characters

\[
\text{<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9}
\]
\[
\text{<unsigned> ::= <digit> | <unsigned> <digit>}
\]
One Grammar For Both

- You could do it all with one grammar by using characters as the only tokens
- Not done in practice: things like white space and comments would make the grammar too messy to be readable

\[
\text{<if-stmt> ::= if <white-space> <expr> <white-space> then <white-space> <stmt> <white-space> <else-part>}
\]

\[
\text{<else-part> ::= else <white-space> <stmt> | <empty>}
\]
Separate Grammars

- Usually there are two separate grammars
  - One says how to construct a sequence of tokens from a file of characters
  - One says how to construct a parse tree from a sequence of tokens

\[
\text{<program-file>} ::= \text{<end-of-file>} | \text{<element>} \text{<program-file>}
\]

\[
\text{<element>} ::= \text{<token>} | \text{<one-white-space>} | \text{<comment>}
\]

\[
\text{<one-white-space>} ::= \text{<space>} | \text{<tab>} | \text{<end-of-line>}
\]

\[
\text{<token>} ::= \text{<identifier>} | \text{<operator>} | \text{<constant>} | \ldots
\]
Separate Compiler Passes

- The *scanner* reads the input file and divides it into tokens according to the first grammar.
- The scanner discards white space and comments.
- The *parser* constructs a parse tree (or at least goes through the motions—more about this later) from the token stream according to the second grammar.
Exercise 4

List the scanner output from the following:

if \ x == 5 \ then \ y = x + y \ endif
Historical Note #1

- Early languages sometimes did not separate lexical structure from phrase structure
  - Early Fortran and Algol dialects allowed spaces anywhere, even in the middle of a keyword
    - Do 10 I = 1.25; → Do10I=1.25; /* Assignment */
    - Do 10 I = 1,25; → Do10I=1,25; /* Loop */
  - Other languages like PL/I allow keywords to be used as identifiers
    - IF THEN THEN THEN = ELSE; ELSE ELSE = THEN;
- This makes them harder to scan and parse
- It also reduces readability
Historical Note #2

- Some languages have a *fixed-format* lexical structure—column positions are significant
  - One statement per line (i.e. per card)
  - First few columns for statement label
  - Etc.
- Early dialects of Fortran, Cobol, and Basic
- Almost all modern languages are *free-format*: column positions are ignored
Outline

- Grammar and parse tree examples
- BNF and parse tree definitions
- Constructing grammars
- Phrase structure and lexical structure
- Other grammar forms
Other Grammar Forms

• BNF variations
• EBNF variations
• Syntax diagrams
BNF Variations

- Some use $\rightarrow$ or $=$ instead of ::= 
- Some leave out the angle brackets and use a distinct typeface for tokens 
- Some allow single quotes around tokens, for example to distinguish ‘|’ as a token from | as a meta-symbol
EBNF Variations

• Additional syntax to simplify some grammar chores:
  - \{x\} or x* to mean zero or more repetitions of x
  - x+ to mean one or more repetitions of x
  - [x] to mean x is optional (i.e. x | <empty>)
  - ( ) for grouping
  - | anywhere to mean a choice among alternatives
  - Quotes around tokens, if necessary, to distinguish from all these meta-symbols
EBNF Examples

\[ \text{<if-stmt>} ::= \text{if} \ <\text{expr}> \ \text{then} \ <\text{stmt}> \ [\text{else} \ <\text{stmt}>] \]

\[ \text{<stmt-list>} ::= \{ <\text{stmt}> ; \} \]

\[ \text{<thing-list>} ::= \{ ( <\text{stmt}> | <\text{declaration}> ) ; \} \]

\[ \text{<digit>} ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 \]

\[ \text{<unsigned>} ::= <\text{digit}>+ \]

\[ \text{<signed>} ::= (+ | -) <\text{unsigned}> \]

- Anything that extends BNF this way is called an Extended BNF: EBNF
- There are many variations
Exercise 5

Construct a grammar in EBNF for each language:

1. `<unsigned>` as the set of all strings with one or more `<digit>`.  
2. `<signed>` as the set of all strings starting with – or + and followed by an `<unsigned>`.  
3. `<integer>` as the set of all strings of `<signed>` or `<unsigned>`.  
4. `<decimal>` as the set of all strings of `<integer>` followed by a ‘.’ and optionally followed by an `<unsigned>`.  
5. `<identifier>` as the set of all strings starting with `<alpha>` and followed by zero or more `<alpha>` or `<digit>`.  

EBNF Extensions

- `{x}` or `x*` to mean zero or more repetitions of x  
- `x+` to mean one or more repetitions of x  
- `[x]` to mean x is optional (i.e. x | `<empty>`)  
- ( ) for grouping  
- | anywhere to mean a choice among alternatives
Exercise 5 continued

Construct a grammar in EBNF for each language:

1. `<1+2’s>` as the set of all strings beginning with ‘1’ and followed by any number of 2’s.
2. `<2’s+1>` as the set of all strings beginning with any number of 2’s and followed by a ‘1’.
3. `<AdigitBs>` as the set of all strings beginning with ‘A’ and optionally followed by any number of `<digit>` or ‘B’.
4. Indiana non-vanity license plates, such as: 22Z1.
5. Scientific notation (e.g. 1.2E-13)
Syntax Diagrams

- Syntax diagrams ("railroad diagrams")
- Start with an EBNF grammar
- A simple production is just a chain of boxes (for nonterminals) and ovals (for terminals):

\[
<\text{if-stmt}> ::= \text{if} \ <\text{expr}> \ \text{then} \ <\text{stmt}> \ \text{else} \ <\text{stmt}>
\]

```
if-stmt  
  if   expr   then   stmt   else   stmt
```

if-stmt

\[
<\text{if-stmt}> ::= \text{if} \ <\text{expr}> \ \text{then} \ <\text{stmt}> \ \text{else} \ <\text{stmt}>
\]
Bypasses

- Square-bracket pieces from the EBNF get paths that bypass them

\[
<\text{if-stmt}> ::= \text{if} \ <\text{expr}> \ \text{then} \ <\text{stmt}> \ [\text{else} \ <\text{stmt}>]
\]

Diagram:

```
if-stmt
  ↓
if  expr  then  stmt  else  stmt
```

Chapter Two
Branching

- Use branching for multiple productions

\[ \text{exp} ::= \text{exp} + \text{exp} | \text{exp} \ast \text{exp} | ( \text{exp} ) | a | b | c \]
Loops

- Use loops for EBNF curly brackets

\[
\text{<exp>} ::= \text{<addend>} \{ + \text{<addend>} \}
\]
Syntax Diagrams, Pro and Con

- Easier for people to read casually
- Harder to read precisely: what will the parse tree look like?
- Harder to make machine readable (for automatic parser-generators)
Formal Context-Free Grammars

- In the study of formal languages and automata, grammars are expressed in yet another notation:

  \[ S \to aSb \mid X \]
  \[ X \to cX \mid \varepsilon \]

  S is a string of symbols \( aSb \) or \( X \).
  X is a string of symbols \( cX \) or empty.

- These are called *context-free grammars* because children of a node only depend on that node’s non-terminal symbol; not on the context of neighboring nodes in the tree. Simpler to define and compile.

- Context sensitive language elements include scope but is not generally part of a grammar.

- Other kinds of grammars are also studied: *regular grammars* (weaker), *context-sensitive grammars* (stronger), etc.
Many Other Variations

- BNF and EBNF ideas are widely used
- Exact notation differs, in spite of occasional efforts to get uniformity
- But as long as you understand the ideas, differences in notation are easy to pick up
Example

WhileStatement:
while ( Expression ) Statement

DoStatement:
do Statement while ( Expression ) ;

ForStatement:
for ( ForInit_opt ; Expression_opt ; ForUpdate_opt )
Statement

[from The Java™ Language Specification, James Gosling et. al.]
Scanner and Parser Generators

- Formal language theory has led to many tools that automate the generation of scanners and parsers from grammar specifications
- Generally called compiler compilers
- Sample tools
  - Accent, ALE, Anagram, Bison, BYACC, Cogencee, Coco, Depot4, LEX, FLEX, Happy, Holub, LLGEN, PRECC, QUEX, RDP, STYX, VisualParse++, YACC++
- Java tools
  - ANTLR, Beaver, Coco/R, CUP, JavaCC, JFLex, JParsec, OpenL, SableCC, SJPT
Scanner or Lexer Generators

- Scannar (also called lexer) generators produce lexical analysers
- A scannar or lexer is used to perform lexical analysis, or the breaking up of an input stream into meaningful units, or tokens
- Sample Lexers
  - Lex, FLex, JLex, Quex, OOLex, re2c, tclex
- FLEX (Fast LEXical analyser generator): a tool for automatically generating a lexer or scanner (lex.yy.c) given a lex specification (*.l)
  - Input file: *.l
  - Output file: lex.yy.c
FLex Input File

- The general format of FLex input file (*.l)
  ... definitions ...
  %%
  ... rules ...
  %%
  ... subroutines ...
- Definitions: macros and header files
- Rules: patterns and associated C statements
- Subroutines: C statements and functions
Sample Input File

/* int.l: input file for the lexer recognizing strings of integers in the input */
%
#include <stdio.h>
%
%option noyywrap  /* Tell flex to read only one input file */
%
%
[0-9]+ {  
    printf(“Found an integer: %s\n”, yytext);
 }

.  {  }  /* Ígnore all other characters */
%
%
int main(void) {  /* Call the lexer, then quit */
    yylex();
    return 0;
}

Lexer Production and Usage

• Production
  - `flex int.l` → `lex.yy.c`
  - `gcc –o int lex.yy.c` → `int`

• Usage
  - For the input: `ab*^c123t+$5!&/6yz`
  - The **int** lexer produces
    - Found an integer: 123
    - Found an integer: 5
    - Found an integer: 6
Parser Generators

- Parser generators produce syntax analysers
- A parser performs syntactic analysis based on a formal grammar written in a notation similar to BNF
- Sample Parsers
  - LLGEN, PRECC, JavaCC, SableCC, YACC, STYX
- YACC (Yet Another Compiler Compiler): a tool for automatically generating a parser (y.tab.c) given a grammar written in a yacc specification (*.y); A grammar specifies a set of production rules, which define a language, and corresponding actions to perform the semantics.
  - Input file: *.y
  - Output file: y.tab.c
YACC Input File

- The same format as FLEX
  ... definitions ...
  %%
  ... rules ...
  %%
  ... subroutines ...
- Rule format:
  name    : names and 'single character's's
  | alternatives
  ;
Sample Input File (calc.y)

%{
#include <ctype.h>
#include <stdio.h>
#define YYSTYPE double /* Used for Yacc stack */
%

%token NUMBER
%left '+' '-'
%left '*' '/'
%left UMINUS

%%
production rules | semantic actions
lines : lines expr '\n' {}, printf("%g\n", $2); }
| lines '\n'
| /* this is a production of the empty word */
;

expr : expr '+' expr {}, $$ = $1 + $3; }
| expr '-' expr {}, $$ = $1 - $3; }
| expr '*' expr {}, $$ = $1 * $3; }
| expr '/' expr {}, $$ = $1 / $3; }
| '(' expr ')' {}, $$ = $2; }
| '-' expr %prec UMINUS {}, $$ = - $2; }
| NUMBER \\

%%
yylex() {
  int c;
  while (((c = getchar()) == ' '));
  if (((c == '.') || (isdigit(c))) {
    ungetc(c, stdin);
    scanf("%lf", &yylval);
    return NUMBER
  }
  return c;
}
Parser Production and Usage

- **Production**
  - `yacc calc.y` → `y.tab.c`
  - `gcc –o calc y.tab.c` → `calc`

- **Usage**
  - For the input: `2+3*5–12/3`
  - The `calc` parser produces
    - `13`
Conclusion

• We use grammars to define programming language syntax, both lexical structure and phrase structure

• Connection between theory and practice
  - Two grammars, two compiler passes
  - Parser-generators can write code for those two passes automatically from grammars
Conclusion, Continued

• Multiple audiences for a grammar
  - Novices want to find out what legal programs look like
  - Experts—advanced users and language system implementers—want an exact, detailed definition
  - Tools—parser and scanner generators—want an exact, detailed definition in a particular, machine-readable form