

# Software Tools Compilers

## Part II - Lecture 4

# Today's Outline

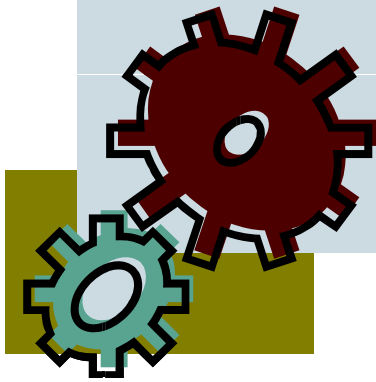
- Introduction to Compilers
- Syntax
- LL(k) Parsers

2009  
YEAR

COMPSCI 732

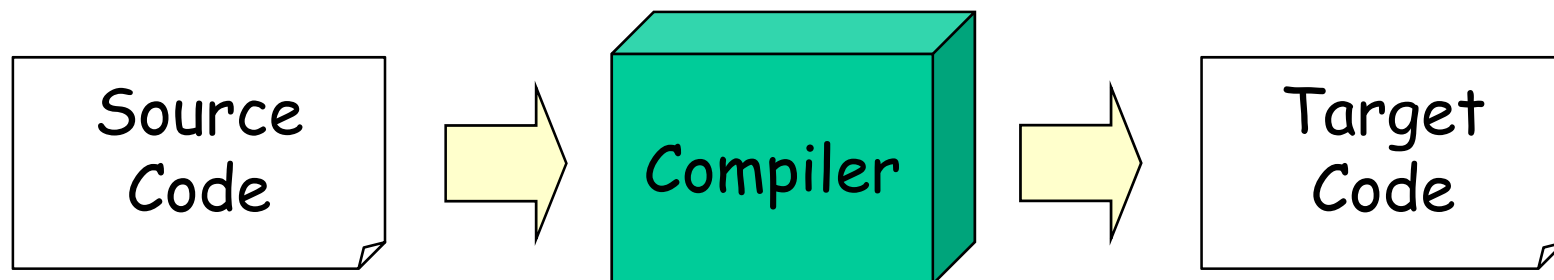
The University of Auckland | New Zealand

# Introduction to Compilers



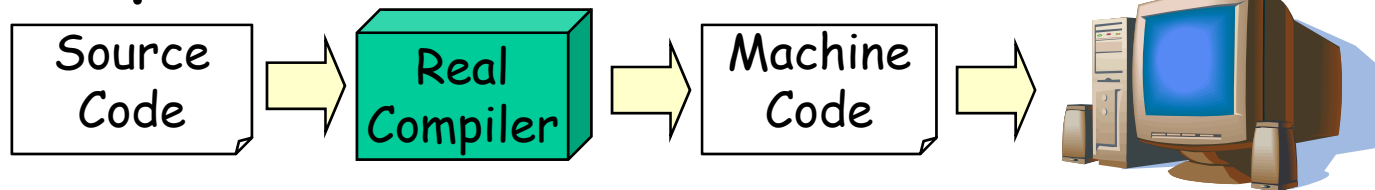
# Compiler Overview

- Compilers are programs that generate target code from source code
  - Source code is typically higher-level, e.g. textual
  - Target code is typically lower-level, e.g. binary machine code (the target is a machine architecture)
- Compilation involves translation between two computer languages
- Compilation may involve other steps such as error checking and optimization

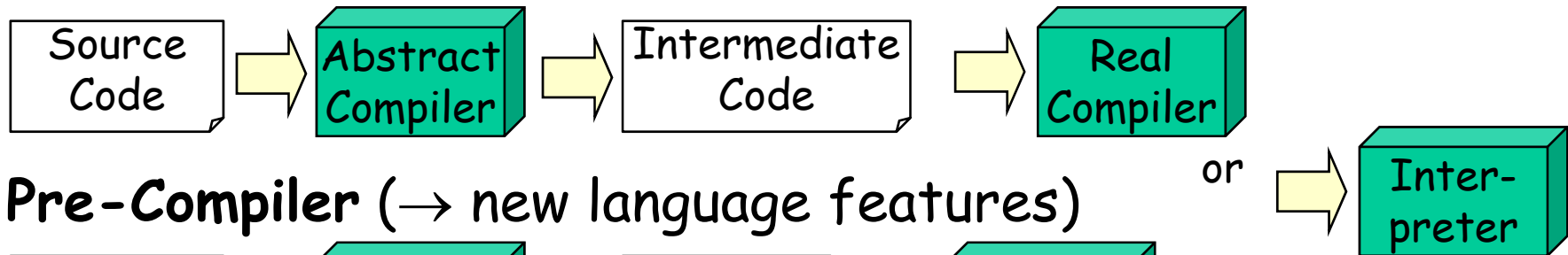


# Types of Compilers

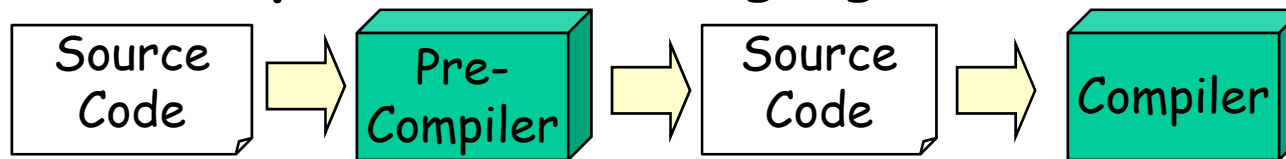
## Compiler for Real Machine



## Compiler for Abstract Machine (→ platform independence)



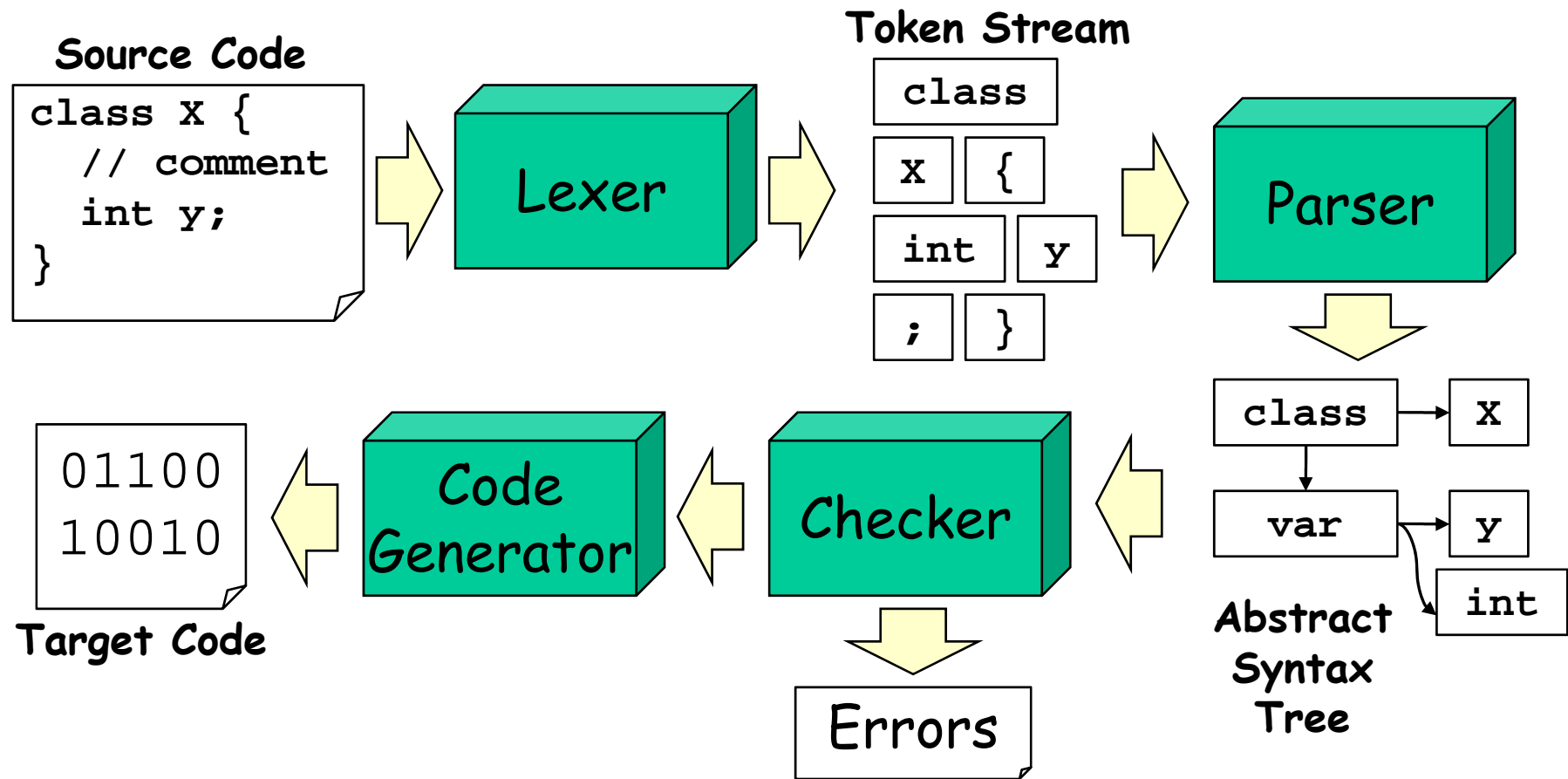
## Pre-Compiler (→ new language features)



## De-Compiler (→ reverse engineering)



# Stages of a Compiler



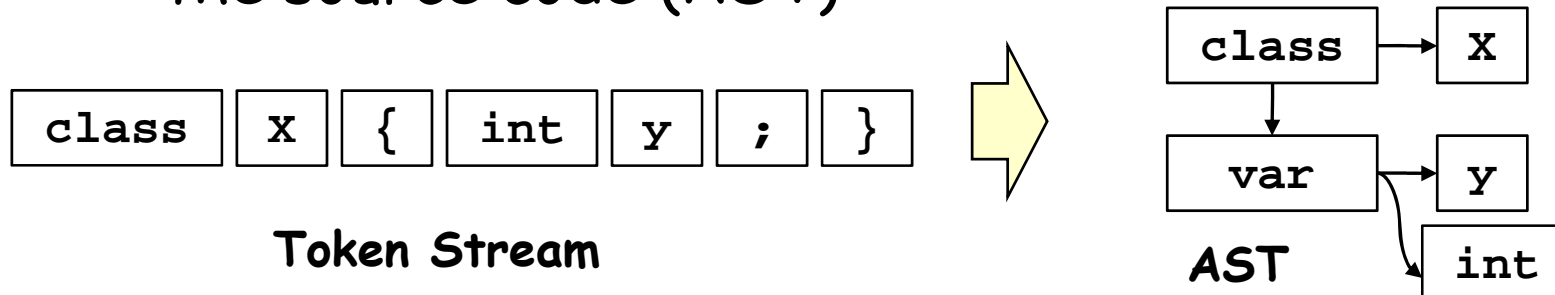
- Lexer chops the source code into tokens
- Parser constructs the syntactic relations between the tokens (abstract syntax tree, AST)



# Syntax

# Syntax of a Language

- Syntax means structure (within and between words)
  - How do the words ("tokens", "lexemes") look like?  
→ specified by regular expressions
  - How can the words be combined into sentences?  
→ specified by a context-free grammar
- Analyzing source code in two steps:
  - Lexer groups characters into tokens (token stream) and removes whitespace (e.g. space, tab, return)
  - Parser groups tokens into a tree-representation of the source code (AST)





# Regular Expressions

- Regular expressions are used to describe the tokens of a language: `EXPR_NAME: expr ;`
- *expr* can consist of the following parts:
  - Literals: `'string'` (e.g. `CLASS: 'class' ;` )
  - Character range: `'char1' .. 'char2'`  
E.g. `DIGIT: '0' .. '9' ;`
  - Alternatives: `expr1 | expr2 | ... | exprn`
  - Optional occurrence: `(expr)?`
  - Multiple occurrence (0..\*): `(expr)*`
  - At least one occurrence (1..\*): `(expr)+`
  - Brackets ( ) around expressions if necessary

# Regular Expression Examples

2009  
YEAR

COMPSCI 732

The University of Auckland | New Zealand

```
// keywords
CLASS: 'class';
IF: 'if';
LPAREN: '(';
PLUSPLUS: '++';
```

```
class Example {
    boolean x = true;
    int y = -99;
    float z = 0.99999;
}
```

```
// literals (i.e. values for types)
BOOL_LITERAL: 'true'|'false';
INT_LITERAL: ('+'|'-')? ('0'..'9')+
FLOAT_LITERAL:
    ('+'|'-')? ('0'..'9')+ '.' ('0'..'9')*;

// identifier with letter and alphanumeric postfix
// e.g. for variable names, method names, ...
IDENTIFIER: ('a'..'z' | 'A'..'Z')
    ('a'..'z' | 'A'..'Z' | '0'..'9')* ;
```

# Context-free Grammars

- Grammar rules are used to describe the relations between tokens: `ruleName: expr ;`
- Grammar rules are similar to regular expressions, but can also use other grammar rules (recursion possible!)
- *expr* can consist of the following parts:
  - Token ("terminal" symbol): `TOKEN_NAME`
  - Rule reference ("non-terminal" symbol): `ruleName`
  - Alternatives: `expr1 | expr2 | ... | exprn`
  - Optional occurrence: `(expr)?`
  - Multiple occurrence (0..\*): `(expr)*`
  - At least one occurrence (1..\*): `(expr)+`
  - Brackets ( ) around expressions if necessary

# Example: Java Grammar

**compilationUnit:**  
**(packageDeclaration)?**  
**(importDeclaration)\***  
**(typeDeclaration)\* ;**

**packageDeclaration:**

**PACKAGE IDENTIFIER (DOT IDENTIFIER)\* SEMI;**

**importDeclaration:**

**IMPORT IDENTIFIER (DOT IDENTIFIER)\* (DOT STAR)? SEMI;**

**typeDeclaration: classDeclaration | interfaceDeclaration;**

**classDeclaration: modifiers CLASS IDENTIFIER**  
**(EXTENDS type)? (IMPLEMENTS typeList)? classBody;**

```
package my.package;  
import java.util.*;  
import javax.swing.JButton;  
class HelloWorld extends Hello  
implements MyInterface { ... }
```

# More Java Grammar

```
// Java expressions (parts that can be evaluated)
```

```
expr: INT_LITERAL | BOOL_LITERAL | ...  
    | IDENTIFIER  
    | PLUSPLUS expr           // e.g. ++x  
    | expr PLUS expr         // e.g. x+1  
    | expr STAR expr         // e.g. x*y  
    | LPAREN expr RPAREN     // e.g. (2+y)*x  
    | LPAREN expr RPAREN QUESTION  
    | expr COLON expr ;      // e.g. (x)? 1 : 2
```

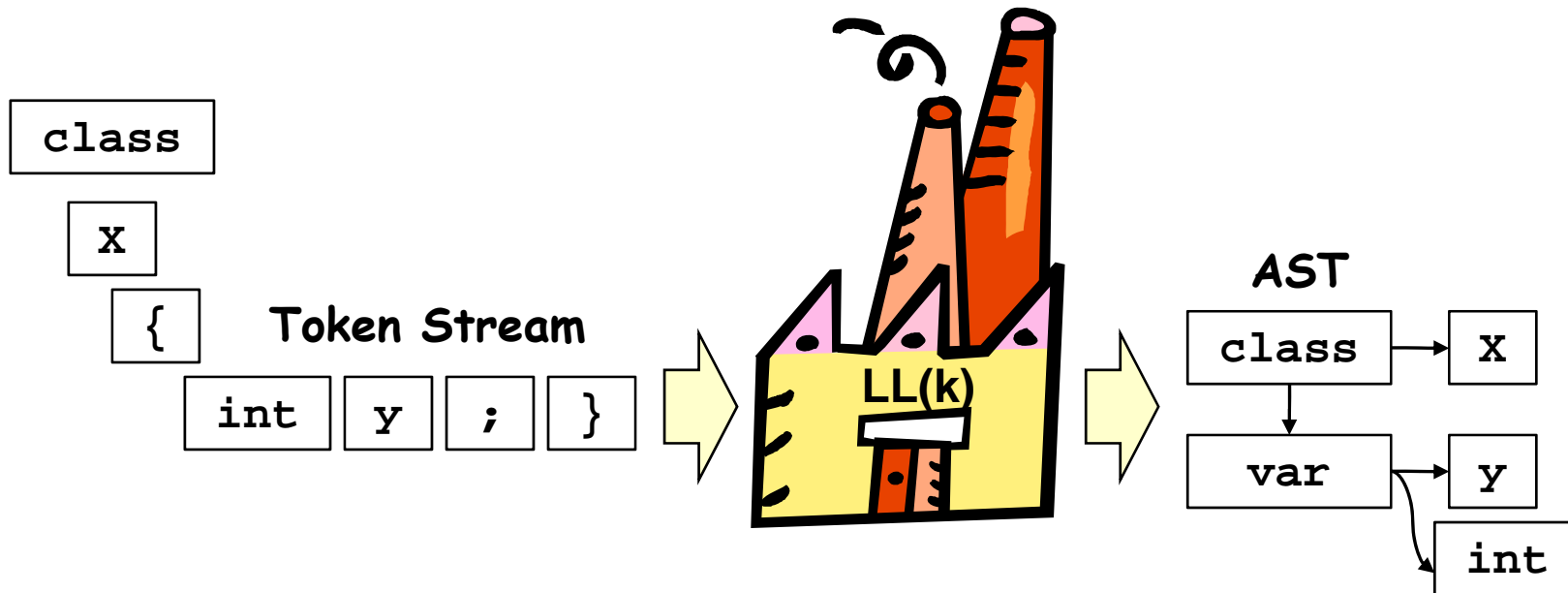
```
// Java statements (parts that can be executed)
```

```
statement: if | for | while | assign | ...  
          | expr SEMI           // e.g. m();  
          | LCURLY (statement)* RCURLY ; // e.g. { a(); b(); }
```

```
// if statement with optional else
```

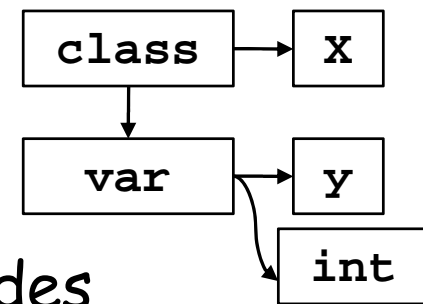
```
if: IF LPAR expr RPAR statement  
   (ELSE statement)?           // e.g. if (true) a(); else b();
```

# LL(k) Parsers



# LL(k) and LR(k) Parsers

- There are different parser algorithms; popular ones are LL(k) and LR(k)
- The first L means the token stream is processed from Left to right
- LL means top-down parsing:
  - First create root of AST
  - Decompose parent nodes into child nodes
- LR means bottom-up parsing:
  - First look at the tokens and group them using the grammar rules (starting at lowest level of AST)
  - Group child nodes into parent nodes
- Lookahead (k): the parser looks at the next k tokens in the stream to decide how to proceed



# How LL(k) Works

1. Begin with the start rule (usually first grammar rule)
2. Look at the alternatives on the right side; try to figure out which alternative to choose by looking at the next k tokens
3. Go through the symbols of the chosen alternative:
  - If terminal: try to match it with the next token
  - If non-terminal: consider the rule of the non-terminal and recurse to step 2.

```
type: CLASS IDENTIFIER classBody
      | INTERFACE IDENTIFIER intfBody;
classBody: LCURLY (member)* RCURLY;
member: IDENTIFIER IDENTIFIER SEMI
        | IDENTIFIER IDENTIFIER
          LPAREN params RPAREN methodBody;
```

Input to parse

```
class Foo {
  int x;
  String y;
  void m() { }
}
```



# Parsing Example

1. Start rule `type`, which alternative?  
Next token is `CLASS` therefore choose first alternative  
Go through symbols: `CLASS IDENTIFIER classBody`
  - `CLASS` is matched with input
  - `IDENTIFIER (Foo)` is matched with input
2. Apply rule `classBody`, only one alternative
  - `LCURLY` is matched with input
3. Apply rule `member`, which alternative? Look ahead 3 tokens.
  - Match `IDENTIFIER (int)`, `IDENTIFIER (x)` and `SEMI`
  - Jump back where we left in rule `classBody`

```
type: CLASS IDENTIFIER classBody
    | INTERFACE IDENTIFIER intfBody;
classBody: LCURLY (member)* RCURLY;
member: IDENTIFIER IDENTIFIER SEMI
    | IDENTIFIER IDENTIFIER
    LPAREN params RPAREN methodBody;
```

## Input to parse

```
class Foo {
    int x;
    String y;
    void m() { }
}
```

# Parsing Example Cont.

4. Back in rule `classBody` we see another member (no `LCURLY` yet)  
So we apply rule `member` again. Lookahead 3 tokens.
  - Match `IDENTIFIER (String)`, `IDENTIFIER (y)` and `SEMI`
  - Jump back where we left in rule `classBody`
5. Back in rule `classBody` we see another member (no `LCURLY` yet)  
So we apply rule `member` again. Lookahead 3 tokens.
  - Match `IDENTIFIER (void)`, `IDENTIFIER (m)`, `LPAREN`
  - Process parameters in `params` and body in `methodBody`
  - Jump back where we left in rule `classBody`
6. Found `RCURLY`, so no more members. Done!

```
type: CLASS IDENTIFIER classBody
      | INTERFACE IDENTIFIER intfBody;
classBody: LCURLY (member)* RCURLY;
member: IDENTIFIER IDENTIFIER SEMI
        | IDENTIFIER IDENTIFIER
          LPAREN params RPAREN methodBody;
```

Input to parse

```
class Foo {
  int x;
  String y;
  void m() { }
}
```

# Parser Implementation

1. Write a method for every rule
2. Check for alternatives with lookahead (ifs / whiles)
3. Match tokens (removes tokens from token stream)
4. Call methods of rules to apply the rules

```
type:  
  CLASS  
  IDENTIFIER  
  classBody  
|  INTERFACE  
  IDENTIFIER  
  intfBody;
```

```
classBody:  
  LCURLY  
  (member)*  
  RCURLY;
```

```
void type() {  
  if(tokens[0] == CLASS) {  
    match(CLASS); match(IDENTIFIER);  
    classBody();  
  } else if(tokens[0] == INTERFACE) {  
    match(INTERFACE); match(IDENTIFIER);  
    intfBody();  
  } else System.out.printf("Error!"); }  
}
```

```
void classBody() {  
  match(LCURLY);  
  while (tokens[0] != RCURLY) member();  
  match(RCURLY); }  
}
```



# Summary

# Today's Summary

- Compilers generate target code from source code
  - Lexer groups characters into tokens
  - Parser groups tokens into AST
- Regular expressions are used to describe tokens
- Context-free grammars are used to describe relations between tokens (as captured in the AST)
- LL(k) parsers can be implemented by making a method out of every grammar rule

## References:

- Hursh Jain. Grammars and Parsers.  
<http://www.mollypages.org/page/grammar/>
- Dick Grune, Criel J.H. Jacobs. Parsing Techniques.  
<http://www.cs.vu.nl/~dick/PTAPG.html>

# Quiz

1. Explain the stages of a typical compiler.
2. Create regular expressions and a grammar for simple arithmetic expressions (numbers, +, \*, brackets)  
(Tip: start each rule with a token so you can use it for lookahead in question 3)
3. Give pseudo code for a LL(k) parser for 2.