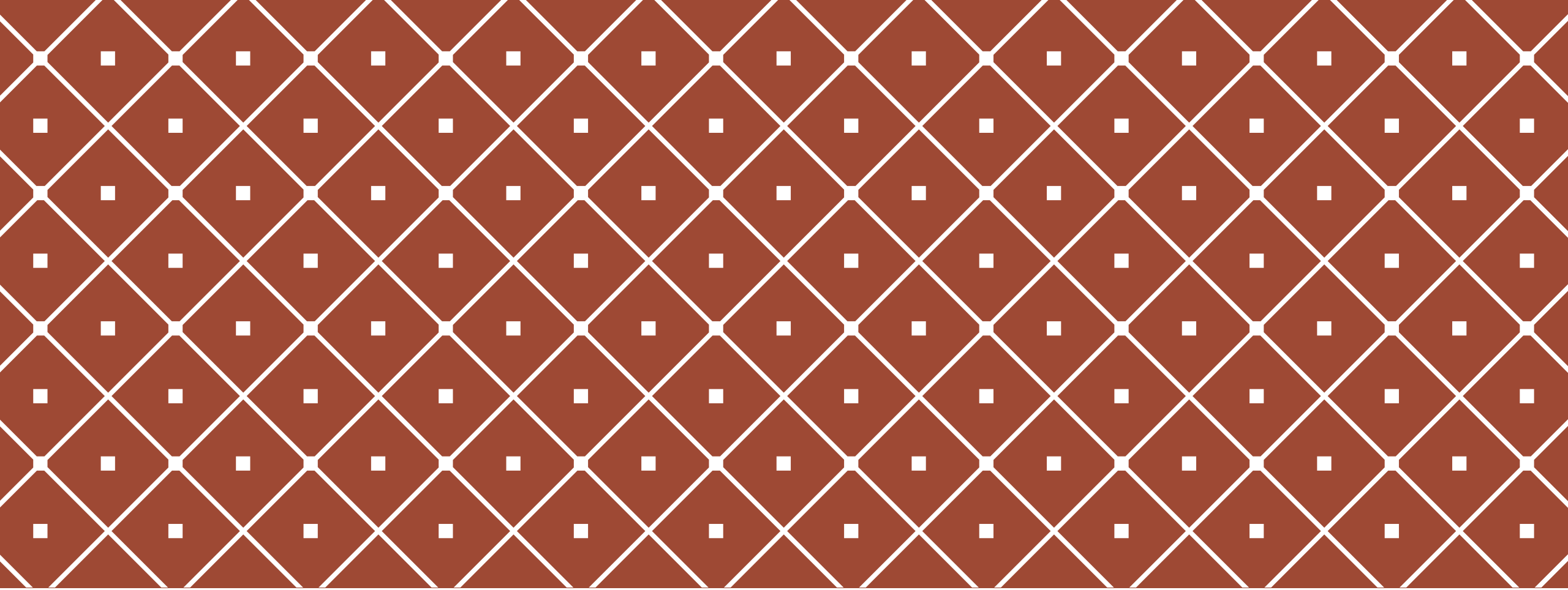


DVA339 HT18 – LECTURE 1

introduction, lexical analysis

PRACTICALITIES

Lectures Tuesday 9:15 – 12:00 and Thursday 13:15 – 16:00

- 3 hours
- to enable more exercises, discussion, and *live coding*

Possibly booked labs – looking into room availability

- Fridays 13-15, and
- if you have questions talk to me in connection with the lecture, or
- send me an email

EXAMINATION

Written exam, 2.5 credits

- more theoretical, examines understanding of concepts used in labs
- no book

Written assignment, 1 credit

- two a4 pages
- suggested topics on homepage – let me know if you select your own

Labs, 4 credits

- Lab 1: two parts, handwritten parser and lexer
- Lab 2: five parts, generated lexer and parser, pretty printer, evaluator, type checker and code generation.
- quite a bit of coding, start in good time!

Work in groups of two!

- discuss in larger groups

COURSE BOOK

Modern Compiler Implementation in Java, Andrew W. Appel

- Object oriented

Course language C#

- examples and code skeletons
- suggested implementation language

Other possibilities (support given on a best effort basis)

- F# (great!)

Recommended development environments

- Visual Studio (Code) on Windows and Mac
- Visual Studio Code on Linux, https://code.visualstudio.com/download

HOMEPAGE

Homepage, http://www.danielhedin.net/dva339/

- news,
- lectures,
- labs, and
- other resources

Canvas

- announcements, and
- handing in labs

WHAT IS EXPECTED OF YOU?

Participate, ask questions, read the course material

- be active! *there are no stupid questions!*

Not all material can be covered by lectures

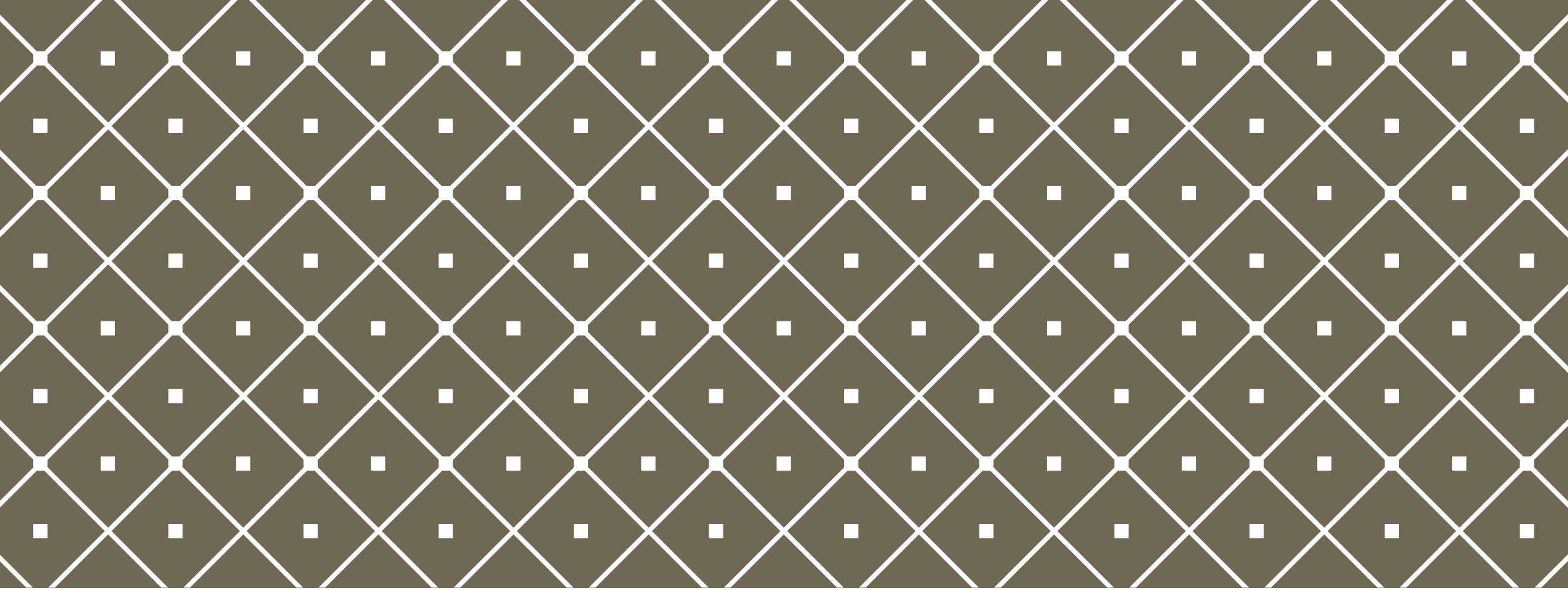
- read the suggested material before the lecture

This is a relatively programming heavy course

- lab 1 - two parts
- lab 2 - five parts
- start early, ask question, discuss with friends, meet the deadlines

The course is running in parallel with Parallel System, DVA336

- also a fairly programming intensive course.
- *fun! but requires planning*



INTRODUCTION TO COMPILERS

WHAT IS A COMPILER

A compiler takes a program in one language, the source language

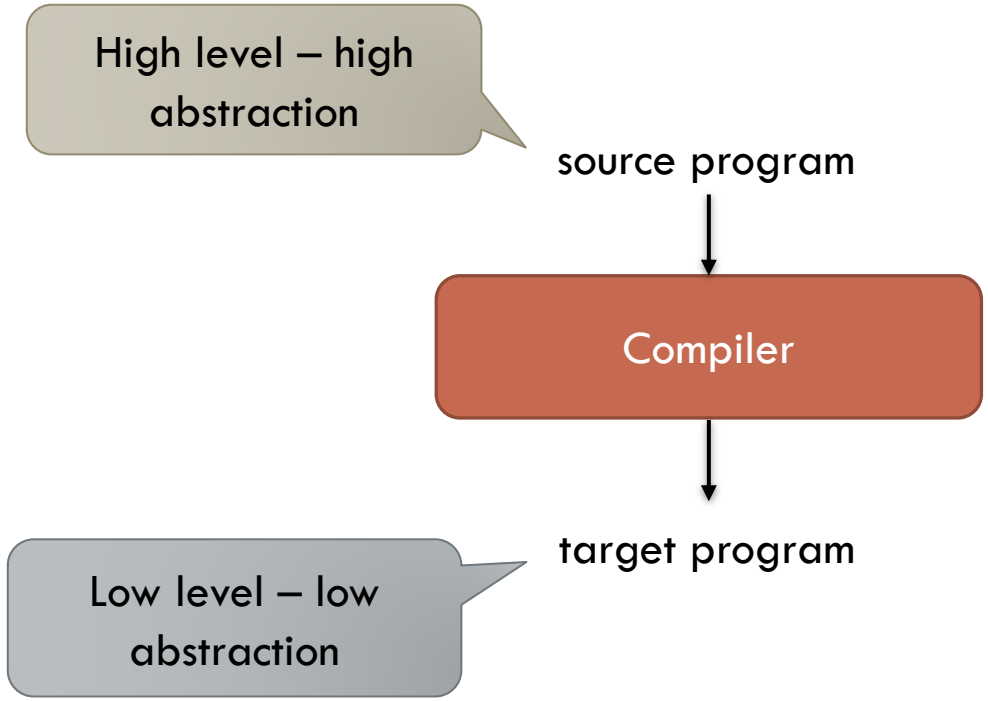
- preferably of high abstraction

and produces an equivalent program in another language, the target language

- sometimes understandable by a computer
- binary, native code
- bytecode, to be run using an interpreter

Examples

- C# to CIL, Java to Java bytecode, C to binary, Haskell to C, Haskell to binary, …



PROGRAMMING LANGUAGES

Study of programming languages relatively recent

- driven by the need for more expressive languages as computers and programs grew more complex

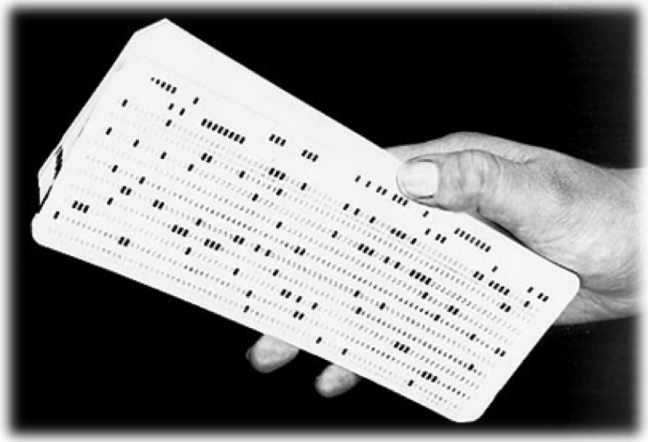
1st generation: programming binary

2nd generation: assembly code

3rd generation: structured languages

- (Fortran, Ada, C, …) Java, C#, Haskell, ML, F#, …
- Increased productivity
- Reduced errors

Domain specific languages



PROGRAMMING LANGUAGES

The study of programming languages borrows from linguistics

- linguistics, scientific study of natural languages

Both ideas and terminology borrowed, but …

- … as is often the case with borrowed terminology the meaning is not always preserved entirely.

We can use previous knowledge of natural languages when approaching the subject of programming languages

- while keeping in mind the possibility of terminological mismatch

Programming languages are *formal languages*

- DVA337 (last period …)
- They are typically a subset of *context-free languages (but not always)*

WHY STUDY COMPILER THEORY?

To be able to create language processors

- Parser for configuration files (e.g. XML)
- Translator from one language to another
- Command line interpreter
- ***Domain specific languages***
- …

Deeper understanding of compilers

- How to write efficient code
- Understand design decisions in a language
- Helps to become a better programmer

But most importantly – it's fun!

When studying compiler theory

relate to your own experience with using compilers

PHASES OF A COMPILER

Typical phases of a real compiler

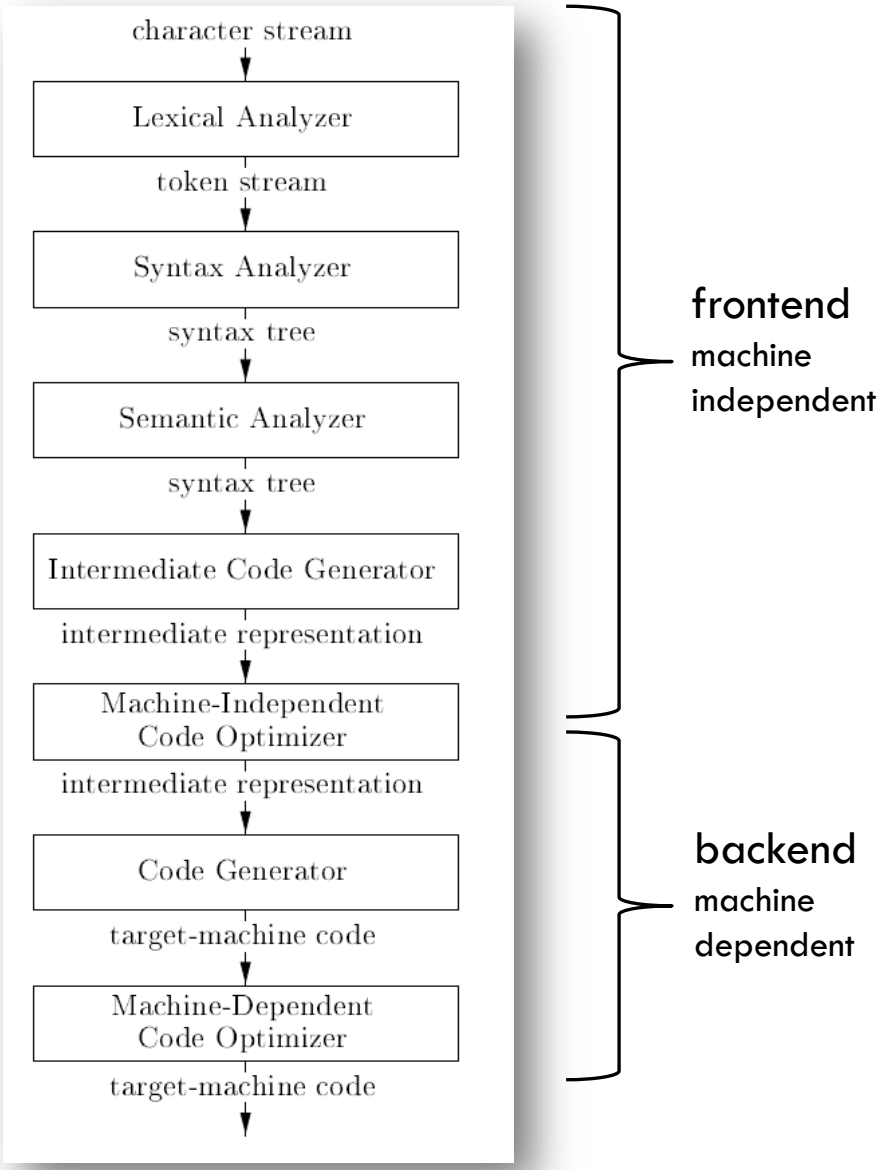
Frontend - machine independent

- Lexical analysis
- Syntax Analysis (Parsing)
- Semantic Analysis (Type checking)
- IR Generation
- Optimization

Backend - machine dependent

- Code generation
- Optimization

Each phase can contain a number of steps



PHASES IN THIS COURSE

We must simplify

- target stack machine instead of 3-operand
- no need for intermediate representation – code can easily be generated from AST

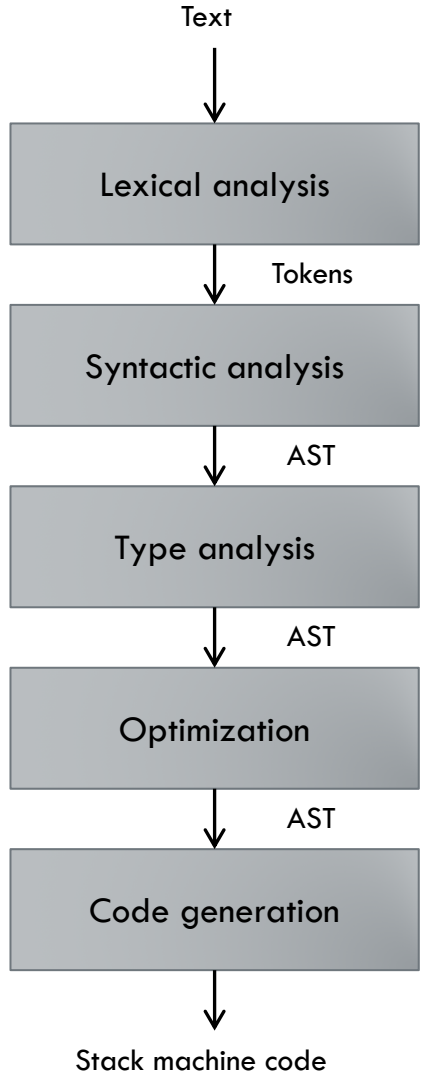
Lexical analysis (Lexing)

Syntactic analysis (Parsing)

Type analysis

Optimization

Code generation (for stack machine)



LEXICAL ANALYSIS

Stream of (ASCII) characters converted to stream of tokens

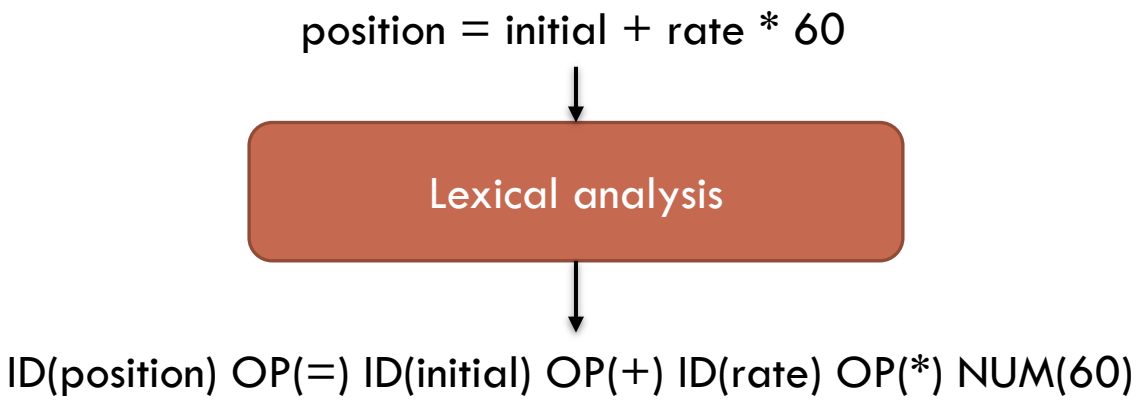
- input to the next phase, the parser

Token types

- identifiers, ID
- numbers, NUM
- operators, OP

Token

- token type, and possibly
- attributes (semantic values)



SYNTACTIC ANALYSIS

Parsing

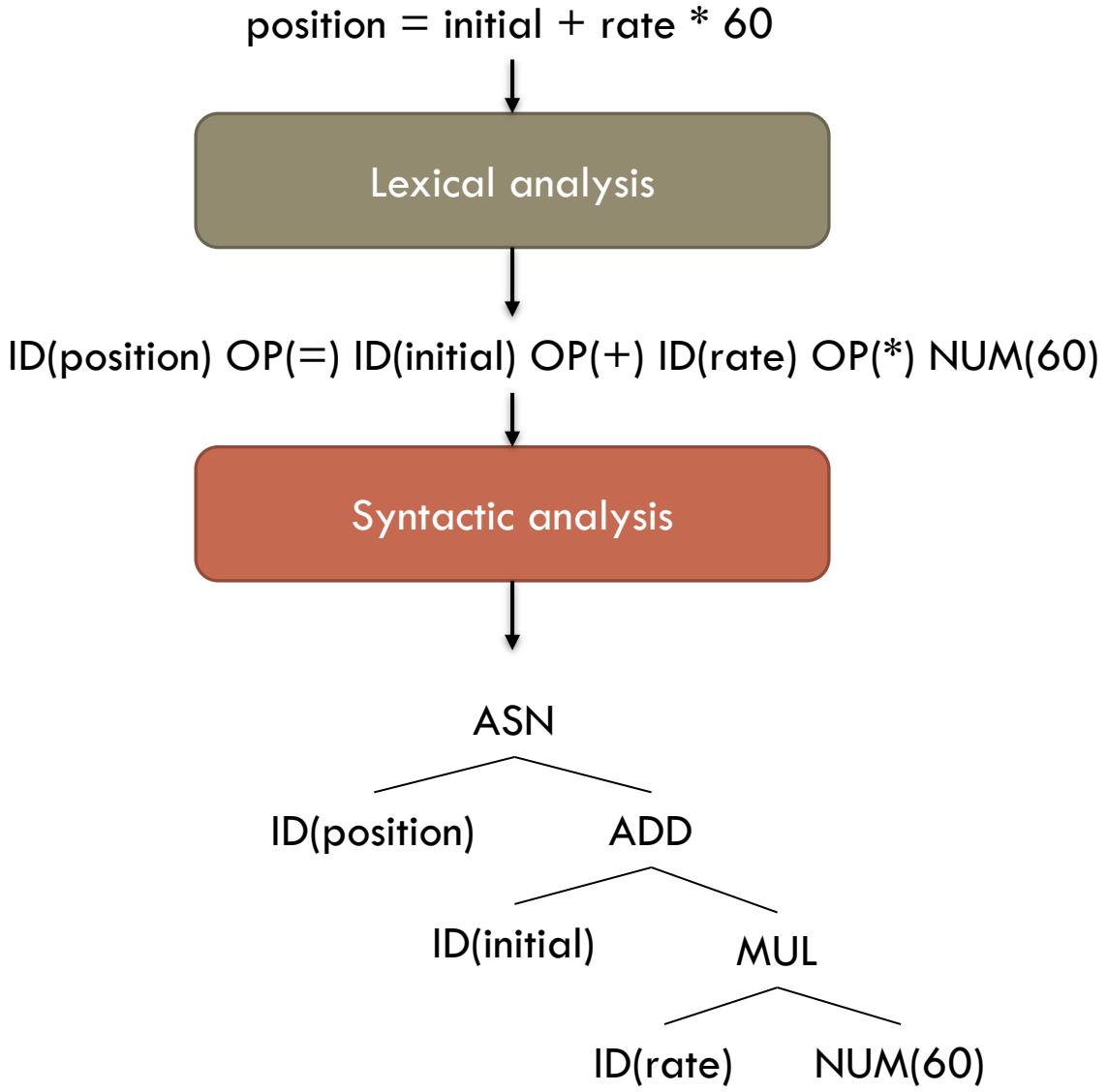
Stream of tokens translated to abstract syntax tree (AST)

- based on *grammar*

Abstract syntax tree

- structure encodes important information about program
- assignment of addition expression to variable
- addition of variable and multiplication expression
- multiplication of a variable and number

Tree encodes evaluation order



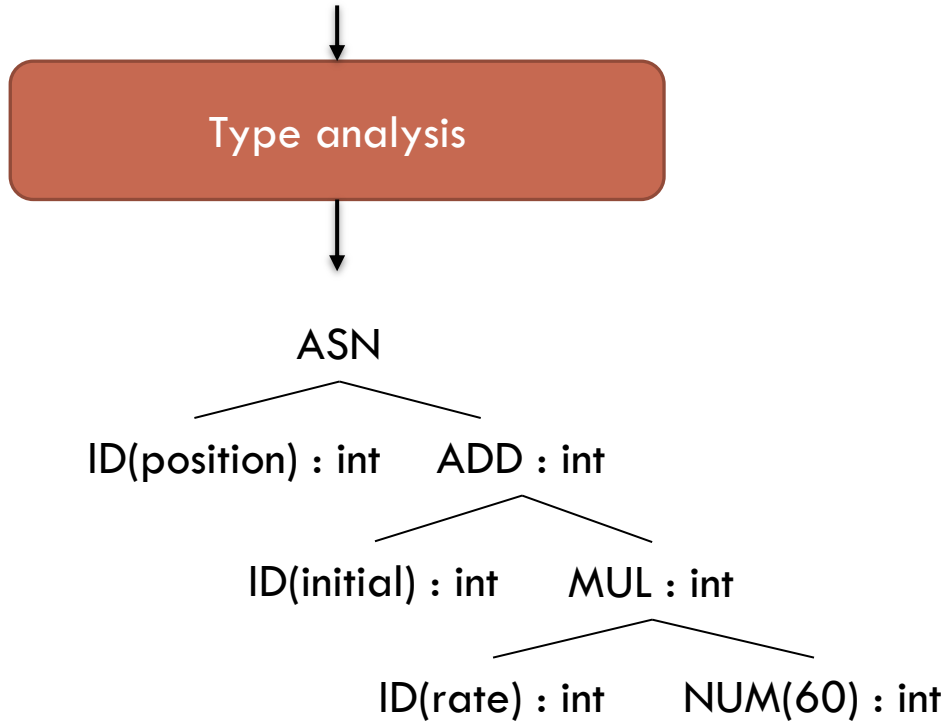
TYPE ANALYSIS

Checks that all parts of the program are type correct

- assignment of an integer addition to an integer variable
- integer addition, since addition of an integer variable and integer multiplication
- integer multiplication, since multiplication of integer variable and integer

May decorate the AST with type information for later stages

- optimization
- code generation



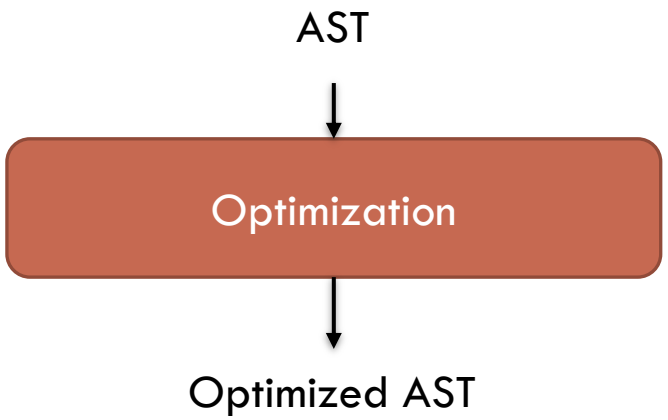
OPTIMIZATION

Hard to do on AST

- most optimizations easier with data flow information or on SSA form

Variants of the following possible

- Constant propagation
- Constant folding
- Common subexpression elimination
- Dead code elimination
- Tail recursion elimination
- Strength reduction
- ….



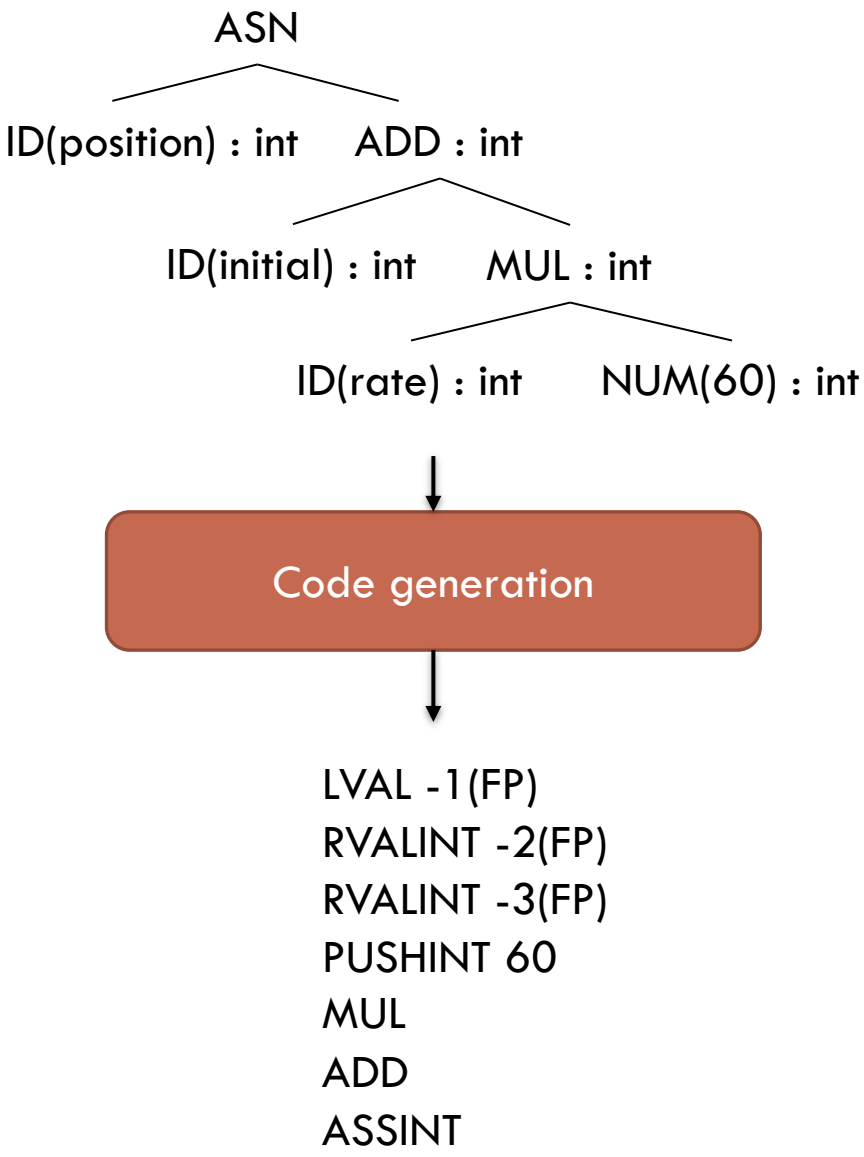
CODE GENERATION

Generates stack code from abstract syntax tree

- stack machine
- type information may be needed to select instructions
- RVALINT
- PUSHINT
- ASSINT

Trac42 code

- local creation, built to illustrate important concepts



COURSE OVERVIEW

Week 1

- lexing and parsing

Week 2

- predictive recursive descent, LR

Week 3

- lexer and parser generators, pretty printing, visitors, and decoration

Week 4

- structural operational semantics, evaluators

Week 5

- type systems, type derivations, type checking

Week 6

- code generation

Week 7

- only Thursday Jan 3
- used as backup if needed

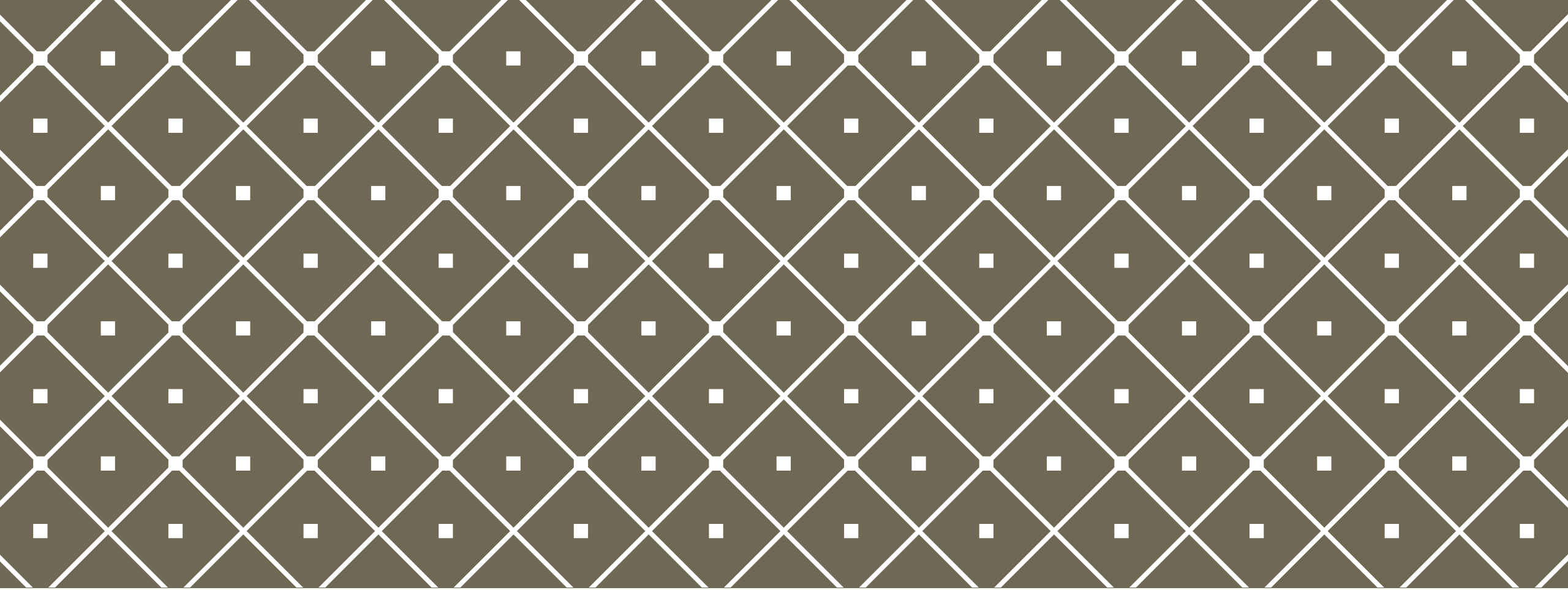
Week 8

- optimization, exam preparations

TODAY

Fundamentals of lexical analysis

Implementation of lexical analysis



LEXICAL ANALYSIS

NATURAL LANGUAGES | IN THE EYES OF A NON-LINGUIST

Study of languages is nothing new – predates computers by a couple of thousand years.

Linguistics, scientific study of natural languages

Separates the *form* of the language from the *meaning* of the language

- syntax – the form
- semantics – the meaning

Syntax is split into *lexical* structure and *grammatical* structure

- lexical structure, the words, word classes, c.f., *lexicon*
- grammatical structure, how to build sentences, c.f., *grammar*

LEXICAL STRUCTURE

Lexical analysis is based on the *lexical structure* of program

- the 'words of the programming language'

Intuitively, can you identify some lexical components in a programming language of choice?

LEXICAL STRUCTURE

Lexical analysis is based on the *lexical structure* of program

Lexical structure

- keywords: if while int …
- operators: + - * / ++ …
- separators: ( ) [ ] { } , . ; …
- identifiers: x y counter stmt1 …
- numbers: 0 1 42

Notice the correspondence to words, word classes, and punctuation

- words: if while int + - * … (but not separators)
- punctuation: ( ) [ ] { } …
- word classes: keywords, operators, separators, …

WHAT ABOUT PUNCTUATION?

Why did we separate the punctuation from the words?

- in linguistics, part of the *orthography*, writing conventions
- not part of the grammar of the natural language (or the *abstract syntax* of a programming language)

Orthography important for understanding written language (parsing!)

- eats shoots and leaves
- eats, shoots, and leaves
- different meanings

No such separation for programming languages

- grammar of English closer to *abstract syntax*

We will come back to this later

AN NOTE ON TERMINOLOGY

Words of programming languages are sometimes called *lexemes*

- not the same meaning as in linguistics!
- (lexical token in Appel, tokens in other work, …)

Word classes also have many different names

- token types, token categories, …

But token also used as the name of what is produced by the lexer

- some risk of confusion!

In this course we will use *lexical token, token type, and token*

- following Appel

LEXICAL ANALYSIS

Programs are written using some character encoding (ASCII or Unicode)

Programming language grammars expressed in terms of

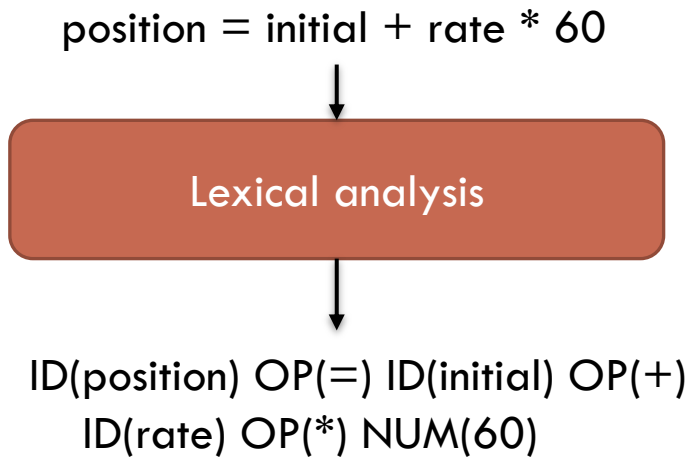
- lexical tokens and token types
- expected input by parser/simpler than writing parser directly on characters

Lexical analysis is the process, where

- a stream of characters is translated to
- a stream of *tokens*

A *token* is a pair consisting of

- the token type
- optional attributes providing additional information on the token
- ID(x), NUM(10), OP(+), SEP([)



LEXICAL ANALYSIS

Lexical analysis is performed by scanning the character stream

- identifying lexical tokens (lexemes), and
- outputting the corresponding token

Whitespace removed

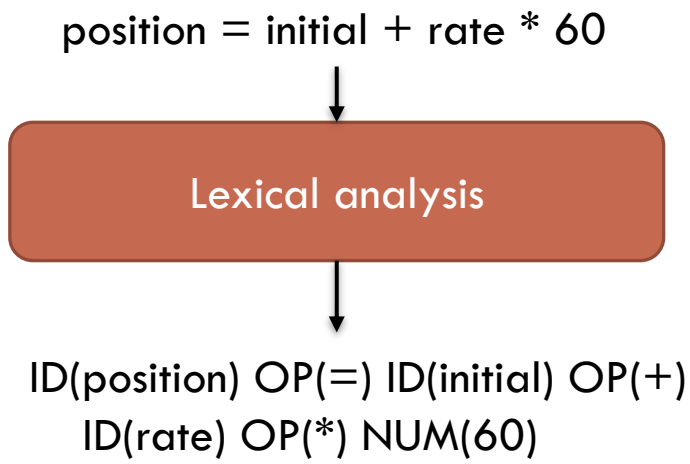
- seen as a separator, but generates no token

Newlines (typically) removed in the same way as whitespace

- in some languages newlines matter

Comments removed

- // line comments
- /* inline comments */



EXERCISE

Given the token types

- keywords, identifiers, numbers, separators and operators

Tokenize “if (x < 3) { y += x++; }”

TOKEN ATTRIBUTES

Token attributes carry information about the input needed later by phases

Consider the expression, 2 + 3

- the parser does not care about the actual numbers, only about the token type
- the parser may care about the operator, to encode precedence
- an evaluator needs information about the actual numbers and the actual operator to be able to compute the result

Such information is stored in the attributes of the token

- also called *semantic values*, values that pertain to the meaning of the program rather than the syntax

Question: can you think of more things to store about tokens that can be useful later on?

ERROR REPORTING

Parsing can fail – if the program is not a syntactically correct program

- parser expects a certain token, but receives another

We want to report

- where in the source code the error is (a simple 'parse error' on thousands of lines of code is not very useful)
- and, maybe, what the parser expected

We need to be able to tie the offending token to the corresponding position in the source code

- line number
- column number

ID(x, 1, 5) NUM(72, 3, 18)

SPECIFYING LEXICAL STRUCTURE

A lexer maps character streams to token streams

- by mapping lexical tokens to tokens

To do this we must know

- what lexical tokens there are, and
- how to create a token we must know the corresponding token type and the position in the source

How can we define which lexical tokens correspond to each token type?

Finite token types, simply enumerate

- keywords, separators, whitespace, newline, …
- operators (for most languages, but some allow user defined operators)

But how do we handle infite token types?

- identifiers, numbers

SPECIFYING LEXICAL STRUCTURE

We use *regular expressions*!

- regular expression define regular languages

keywords: if | while | …

whitespace: ' ' | \t

- space, tab

newline: \n | \r | \r\n

- UNIX, old Mac, Windows

numbers: [0-9]+

- one or more digits

identifiers: [a-zA-Z][a-zA-Z0-9]*

- letter followed by zero or more letters or digits

REGULAR EXPRESSIONS

The language of regular expressions over an alphabet Σ is

$r ::= \varepsilon \mid x \in \Sigma \mid rr \mid r|r \mid r^* \mid (r)$

empty regular expression, ε

character in alphabet, $x \in \Sigma$

sequence, rr

option, $r|r$

repetition, r^*

What is $(0|1|2|3|4|5|6|7|8|9)(0|1|2|3|4|5|6|7|8|9)^*$?

EXERCISE, DERIVED FORMS

Assuming we have regular expressions over ASCII characters

$$r ::= \varepsilon \mid x \in \text{ASCII} \mid rr \mid r|r \mid r^* \mid (r)$$

Can you express

- a range of characters, `[a-z]`
- one or more, `r+`
- optional, `r?`
- not in range, `[^a-z]`

PRACTICAL REGULAR EXPRESSIONS

Assume we want to write regular expressions over ASCII characters

Some characters cannot be written and must be escaped

- `\n, \r, \t, …`

Some characters are used to write the regular expression themselves and must be escaped

- `. " ( ) { } [ ] + * / | ^ $ \`
- space `' '`

Examples

- a+b matches the string *a* or the string *b*
- a\+b matches the string *a+b*
- "a+b" matches the string *a+b*
- \"a+b\" matches the string *"a* or the string *b"*

EXERCISE

What does the following regular expression define?

```
(+|-)?[0-9]+\.[0-9]+((E|e)(+|-)?[0-9]+)?
```

```
\"[^\"\n]*\"
```

SPECIFYING LEXICAL ANALYSIS

For each lexical token in the language define the token type

regexp	type
print	KEYW
;|\(|\)|,	SEP
\+|:=	OP
[a-zA-Z][a-zA-Z0-9]*	ID
[0-9]+	NUM

Notice that some of the token types overlap. Is this a problem?

- KEYW and ID overlap completely
- ID and NUM overlap partially

AMBIGUITIES

Overlapping token types lead to ambiguities, more than one way to tokenize

Should "xyz123" be tokenized as

- ID(xyz123) or
- ID(xyz)NUM(123)

Should “print1” be tokenized as

- ID(print1) or
- KEYW(print)NUM(1)

Principle: longest match

regexp	type
print	KEYW
;|\(|\)|,	SEP
\+|:=	OP
[a-zA-Z][a-zA-Z0-9]*	ID
[0-9]+	NUM

AMBIGUITIES

Overlapping token types lead to ambiguities, more than one way to tokenize

Should "print" be tokenized as

- ID(print)
- KEYW(print)

regexp	type
print	KEYW
;|\(|\)|,	SEP
\+|:=	OP
[a-zA-Z][a-zA-Z0-9]*	ID
[0-9]+	NUM

Principle: order of definition

PRINCIPLES

Order of definition

- the first defined regular expression takes precedence
- `KEYW` over `ID`

Longest match

- match as far as possible
- "xy" becomes ID(xy)
- "xyz123" becomes ID(xyz123)
- "print" becomes KEYW(print)

regexp	type
`print`	`KEYW`
`;|\(|\)|,`	`SEP`
`\+|:=`	`OP`
`[a-zA-Z][a-zA-Z0-9]*`	`ID`
`[0-9]+`	`NUM`

EXERCISE

Tokenize "print+1(;++\nprint1 x y"

- given longest match
- definition order
- add position, line and column, to the token

To start

- first token is KEYW(print, 1,1)

regexp	type
\n|\r\n|\r|	
print	KEYW
;|\(|\)|,	SEP
\+|:=	OP
[a-zA-Z][a-zA-Z0-9]*	ID
[0-9]+	NUM

SO FAR

lexical tokens/lexemes

token types

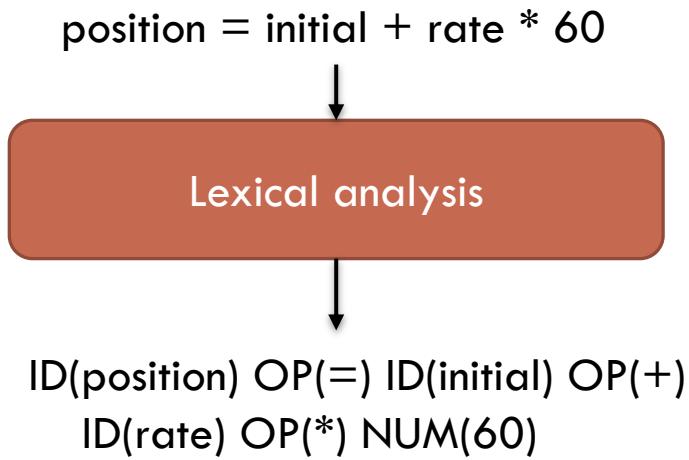
- keywords, operators, separators, identifiers, numbers, …

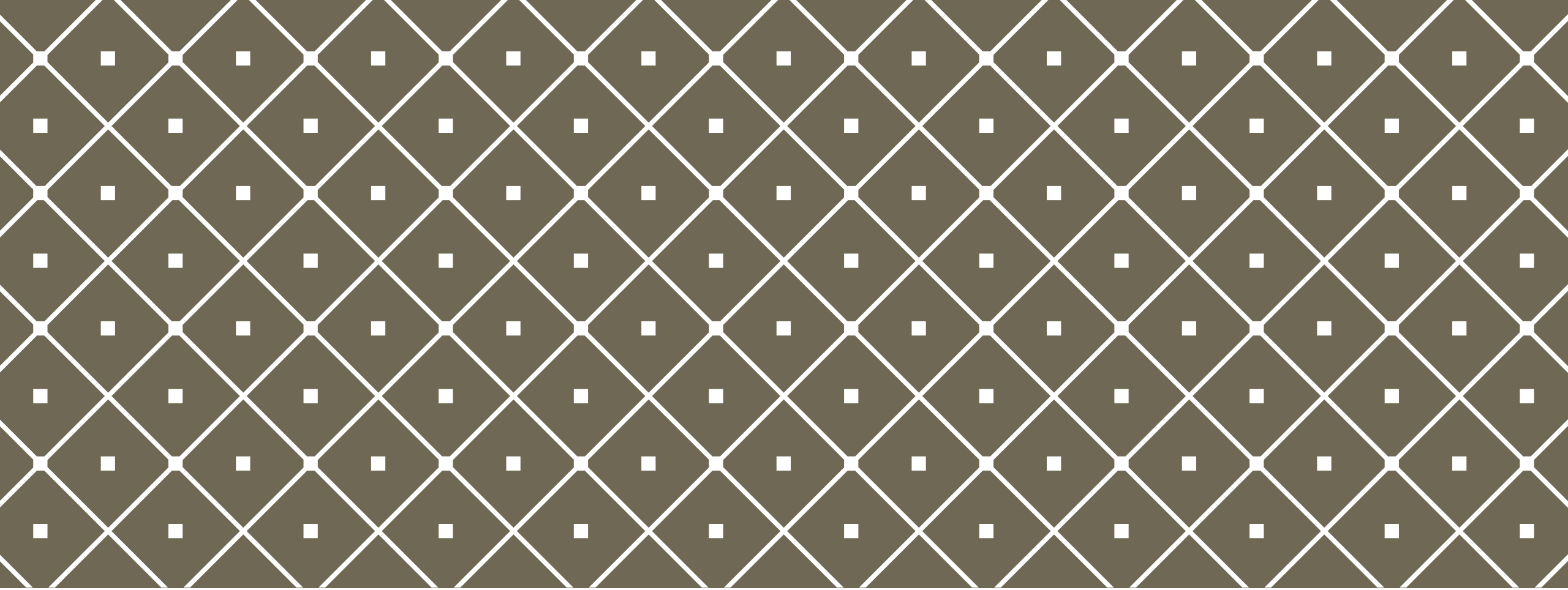
token attributes/semantic values

- the lexical token/lexeme
- the line and column number

regular expressions

- for specifying token types





IMPLEMENTING A LEXER

in an object oriented language

THE LEXER

The lexer should take a string and methods to get all tokens

```
public class Lexer
{
    public Lexer(string text)
    {

        ...

    }

    public Token Next()
    {
        // returns next token

    }

    public Token Peek()
    {
        // looks at the next token without forwarding

    }

}
```

TOKENS IN PRACTICE

Tokens consist of

- a token type
- a lexeme
- the start position of the lexeme in the source code

```
public class Token
{
    public Type type;
    public string lexeme;

    public int line;
    public int column;

    public enum Type { KEYW, SEP, OP, ID, NUM, EOF };

    ...
```

HOW TO PEEK?

Simply buffer!

```
public class Lexer
{

    ...

    internal Token buf;

    ...

    public Token Next()
    {
        Token t = Peek();
        buf = null;
        return t;
    }

    public Token Peek()
    {
        if (buf != null) { return buf; }
        ... compute new token

    }

...
```

HANDLING WHITE SPACE

Cannot simply remove, since we track position in source

- must keep track of current line and column

Whitespace, forwards column

- space, ' ', 1 character wide
- tab, \t, typically 8 characters wide

Newline, resets column, forwards line

- \r\n, only one newline!
- \r,
- \n,

I typically make use of a trim function that removes white-space and takes care of the tracking

```
void Trim() { ... }
```

MATCHING

How should we match the different token types?

Use convenient match functions

- `MatchIdentifier` matches `[a-zA-Z][a-zA-Z0-9]*` (longest match!)
- `MatchNumber` matches `[0-9]+` (longest match!)
- `Match` takes a list of strings and returns the first that matches
- All return `string.Empty` if no match

```
string MatchIdentifier()

string MatchNumber()
```

```
string Match(string[] lexemes)
```

```
static string[] newline = { "\r\n", "\r", "\n" };
static string[] operators = { "+", ":=" };
static string[] separators = { "(", ")", ",", ";" };
```

MATCHING

Using such matching functions it is easy to compute the next token

1. trim the input
2. if end of input generate EOF token
3. in order, generate token if match
    1. match separators
    2. match operators
    3. match identifiers
        1. is keyword?
    4. match numbers
4. profit!

regexp	type
`print`	KEYW
`;|\(|\)|,`	SEP
`\+|:=`	OP
`[a-zA-Z][a-zA-Z0-9]*`	ID
`[0-9]+`	NUM

ERRORS

Error handling in a lexer is hard

- too little structure to be able to correct and continue

Simply throw an exception

```
public class LexerException : Exception
{
    public LexerException(string message) : base(message)
    {
    }
}
```

DONE

That was Lab 1.1!

- well, apart from the missing code :D
- and writing tests
- use the Test11.exe to test your lexer

Use the time to familiarize yourselves with C#

- and Visual Studio or Visual Studio Code

Use Test11.exe to test your lexer

- you'll have to trust me – it doesn't do anything bad
- sorry, can't give you the source (it contains the solution)
- don't hand in labs that do not pass at least a 100 tests