Domain-Specific Languages

Programming Language Primer

Andrzej Wąsowski

IT UNIVERSITY OF COPENHAGEN

PROCESS AND SYSTEM MODELS GROUP

Why this primer lecture?

- **Before the course:** take compiler/programming language course
- **Nice to have:** functional programming, design patterns, mobile app programming
- **After the course:** "free" projects on designing domain specific languages, gladly with industrial partners
- **Or (language path):** automated software analysis, programming language seminar
- **Or (software engineering path):** software architecture

- We start with refreshing the prerequisite material
- Crash course for those who have not seen it

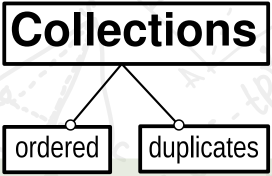
Objective: recall prerequisites for the course

- Some basic math
- Compiler architecture
- Syntax, semantics, types
- Programming language concepts

AGENDA

Common Kinds of Collections

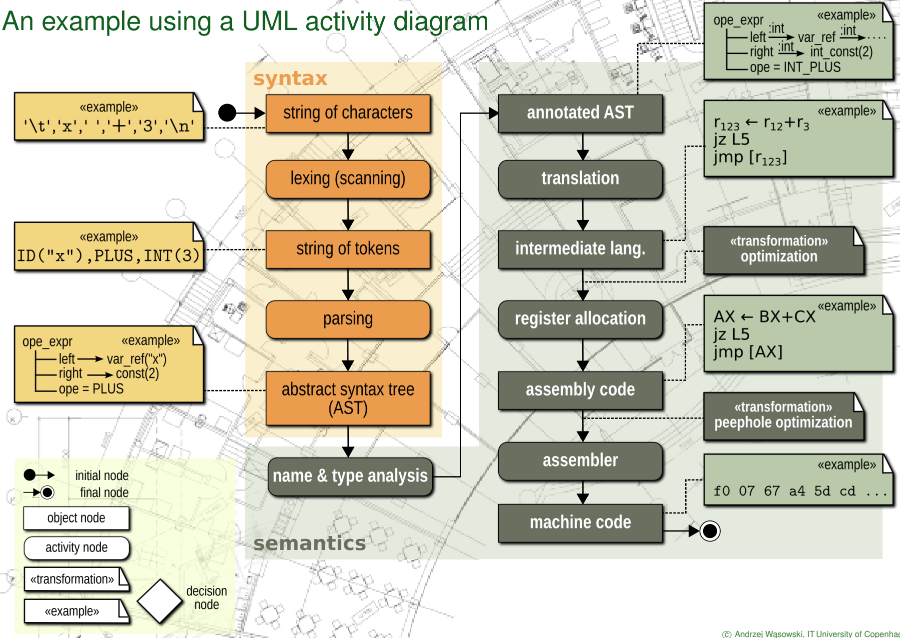
Using a FODA feature diagram



unordered	no dup.	**set** $\{a, b, c\} = \{b, a, c, b\}$
unordered	duplicates	**bag, multiset** $\{a, b, b, c\} = \{b, a, c, b\}$ $\{a, b, b, c\} \neq \{a, b, c\}$ alternative view, a mapping to naturals: $\{a, b, b, c\} \approx [a \mapsto 1, b \mapsto 2, c \mapsto 1]$
ordered	duplicates	**sequence, list, string** $\langle a, b, c, c\rangle \neq \langle a, b, c\rangle$ alternative view, a mapping from naturals: $\langle a, b, c, c\rangle \approx [1 \mapsto a, 2 \mapsto b, 3 \mapsto c, 4 \mapsto c]$
ordered	no dup.	**ordered set, unique list** $\langle a, b, c\rangle \neq \langle a, c, b\rangle$

Architecture of a Compiler

An example using a UML activity diagram



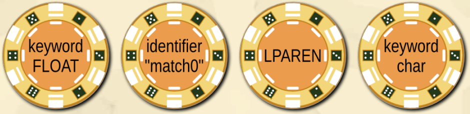
Lexical Syntax

Defines the vocabulary of a language

lexer/scanner/tokenizer input: unstructured, flat, sequence of characters

`f` `l` `o` `a` `t` `␣` `m` `a` `t` `c` `h` `0` `(` `c` `h` `a` `r` `␣` . . .

output: flat sequence of tokens (identified meaningful character groups)



. . .

example input (87 bytes)

```
float match0(char *s)
{
    /* find a zero */
    if (!strncmp(s,"0.0", 3))
        return 0.;
}
```

example output (25 tokens)

kwFLOAT ID("match0") LPAREN
kwCHAR AST ID("s") RPAREN
LBRACE kwIF LPAREN BANG
ID("strncmp") LPAREN ID("s")
COMMA STRING("0.0") COMMA
INT(3) RPAREN RPAREN kwRETURN
INT(0) PERIOD SEMI RBRACE

Regular Expressions

A domain specific language for defining legal tokens

Regular expressions (regexps)

Let Σ be an alphabet of characters,
Let ϵ be an empty character string. Then

- ϵ is a regexp defining language $L = \{\epsilon\}$
- a is a regexp for any $a \in \Sigma$, defining $L = \{a\}$

Let r, s be regexps defining R (resp. S), then:

- $r \mid s$ is a regexp defining language $R \cup S$
- rs is a regexp defining $\{vw \mid v \in R \wedge w \in S\}$
- r^+ generates $\{v_1 ... v_n \mid v_i \in R, 1 \leq i \leq n, n \in \mathbb{N}\}$

Syntactic sugar

- r^* = $r^+ | \epsilon$ (Kleene star)
- $r? = r | \epsilon$ (optional)
- $[a - zA - Z] = a \mid \cdots \mid z \mid A \mid \cdots \mid Z$

Examples

`class`

[0-9]+

[a-z][a-z0-9]*

([0-9]+"."[0-9]*)|
([0-9]*"."[0-9]+)

Token

A scanner recognizes a token as the longest prefix of current input matching any regexp of the language.

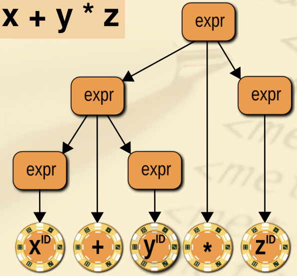
Torben Ægidius Mogensen. *Introduction to Compiler Design.*
© Springer-Verlag London Limited 2011 ← **more details on regexp matching algorithms**

Syntax Trees

How are tokens organized into phrases?

Possible grammatical structure

x + y * z



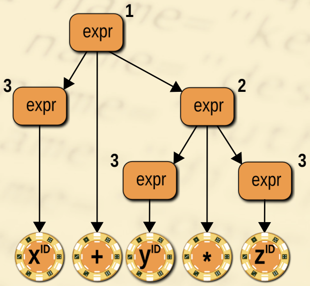
How do we specify **all legal syntax trees** like this one?

Context-free grammar

$$\text{expr} \rightarrow_1 \text{expr} + \text{expr}$$
$$\text{expr} \rightarrow_2 \text{expr} * \text{expr}$$
$$\text{expr} \rightarrow_3 \text{ID}$$

The grammar is ambigous!

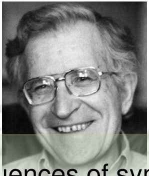
Derivation Tree

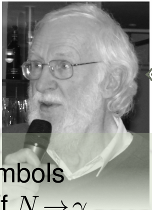


rule 1
rule 3
rule 2
rule 3
rule 3

left-most derivation

Context-free Grammars


Noam Chomsky


Peter Naur

Context-free grammar (CFG)

A set of **productions** over **terminal** symbols (tokens) and **nonterminal** symbols

A production rewrites a nonterminal (left) to a list of terminals and nonterminals (right)

Any nonterminal used on the right of a production appears on the left of some production

One nonterminal is a **start symbol**

Context-free language

Let α, β and γ be sequences of symbols

A *derivation relation*: $\alpha N \beta \Rightarrow \alpha\gamma\beta$ iff $N \to \gamma$

A CFG G over terminals T defines a *context free language* over T $\{w \in T^* \mid S \Rightarrow^* w\}$.

Each regular language is context-free

Extended Backus-Naur form (EBNF) syntactic sugar

alternative: $S \to \alpha \mid \beta \longrightarrow \begin{cases} S \to \alpha \\ S \to \beta \end{cases}$

optional: $S \to \alpha\, T?\, \beta \longrightarrow \begin{cases} S \to \alpha T' \beta \\ T' \to T \mid \epsilon \end{cases}$

Kleene$*$: $S \to \alpha T^* \beta \to \begin{cases} S \to \alpha T' \beta \\ T' \to (TT')? \end{cases}$

grouping: $S \to \alpha(\beta)\gamma \longrightarrow \begin{cases} S \to \alpha T' \gamma \\ T' \to \beta \end{cases}$

N. Chomsky. *Three models for the description of language*
IEEE Trans. on Information Theory 2(3):113–124. 1956

Example

```
op → "+" | "*"
exp → exp op exp | ID | "(" exp ")"
```

LL-Parsing

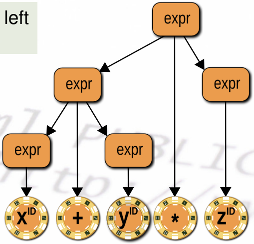
read input from left to right

search for left derivations

Parser

A translator from a sequence of tokens to a data structure representing a parse tree

parsing



Recall our example grammar

expr $\to_1$ expr "+" expr
expr $\to_2$ expr "$*$" expr
expr $\to_3$ ID

LL(k) grammar

A for which always the next production in a left derivation can be selected deterministically (unambiguously) based on reading next k tokens.

Ambiguity (conflict)

Derive from a start symbol expr:

$\boxed{x + y} * z \longrightarrow$ use rule 1?

$x + \boxed{y * z} \longrightarrow$ or rule 2?

Not an ANTLR error!

LL($*$) parser

Selects productions by recognizing if following tokens belong to regular language (no limit on prefix length).

Torben Ægidius Mogensen. *Introduction to Compiler Design*.
© Springer-Verlag London Limited 2011 ← **has a guide on writing and disambiguating grammars**

Concrete and Abstract Syntax

Concrete syntax[†]

What users interact with creating programs

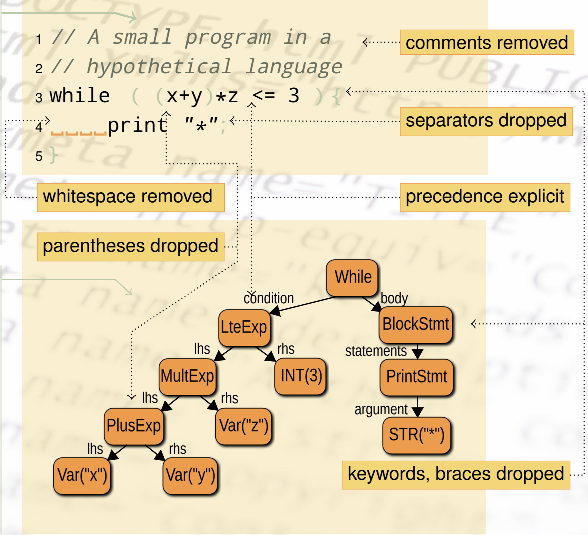
Defined by lexical spec+grammar

Trivially: what enters lexer/parser

Abstract syntax[†]

Data structure (a tree, hence AST) storing the program, cleaned of notational details of parse trees

Trivially: what comes out of parsing stage



[†] Markus Völter. *DSL Engineering. Designing, implementing and using domain specific languages.* `http://www.dslbook.org/`. 2013 ← **Lots of stuff on syntax. We will explore more later**

Type Checking

Abstract execution aiming to find some errors quickly

```
1 // Example in Java
2 int f(int a, int b){
3   int j = a + b;
4   {
5     String a="Hi";
6     print a;
7   }
8   return a;
9 }
```

$\sigma_1 = []$

$\sigma_2 = [a \mapsto \text{int}, b \mapsto \text{int}]$

$\sigma_3 = [a \mapsto \text{int}, b \mapsto \text{int}, j \mapsto \text{int}]$

$\sigma_5 = [a \mapsto \text{String}, b \mapsto \text{int}, j \mapsto \text{int}]$

String "Hi" is printed

$\sigma_7 = [a \mapsto \text{int}, b \mapsto \text{int}, j \mapsto \text{int}]$

Integer value of a is returned

$\sigma_8 = []$

static scoping: scopes are AST nodes

Scoping

Enter scope:
add local symbols

Exit scope:
drop local symbols

Most languages use static scoping. Emacs lisp & bash use dynamic scopes (environments store the last variable def. seen in the call stack)

Example: typing l. 3

... bottom-up AST traversal

$$\dfrac{\dfrac{\sigma(\texttt{a}) = \text{int}}{\sigma \vdash \texttt{a} : \text{int}} \qquad \dfrac{\sigma(\texttt{b}) = \text{int}}{\sigma \vdash \texttt{b} : \text{int}}}{\dfrac{\sigma \vdash (\texttt{a + b}) : \text{int}}{\sigma \vdash \texttt{int j = a + b} : \text{ok}}}$$

Typing rules (simplified)

$$\frac{\sigma \vdash e : T}{\sigma \vdash T\ v := e : \text{ok}}\text{(dcl)}, \qquad \frac{\sigma(v) = T}{\sigma \vdash v : T}\text{(var)}$$

$$\frac{\sigma \vdash e_1 : \text{int} \qquad \sigma \vdash e_2 : \text{int}}{\sigma \vdash (e_1 + e_2) : \text{int}}\text{(plus)}$$

Objectives of Type Checking

and of name/scope analysis

- **Eliminate invalid programs**, more than lexical and syntactic analyses
- Find **undeclared variables**, misspelled names, etc.
- Find **type mismatch errors**, e.g.
  - multiply a number by a string:

    ```
    3.54159 * "Life is beautiful"
    ```

  - call a function with wrong number of parameters:

    ```
    sqrt(2, 1.2, 1)
    ```

- Eliminate need for **dynamic checks** (performance)
- Gather information needed for **memory allocation**
- **Type inference**: finding most general types without type declarations
  - For instance n is integer in the following Haskell program:

```
factorial 0 = 1
factorial n = n * factorial (n - 1)
```

Approaches to Types

in programming language implementations

Inference ≠ dynamic typing

Inference finds **static** types for untyped objects by solving constraints.

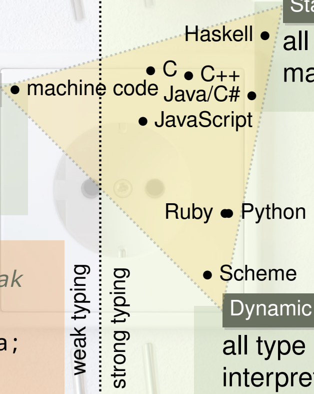
Strong typing

Static typing

all type checks are made before execution

Weak typing

operations are applied without any checking of operand types



strong typing: operations are only applied to properly typed operands

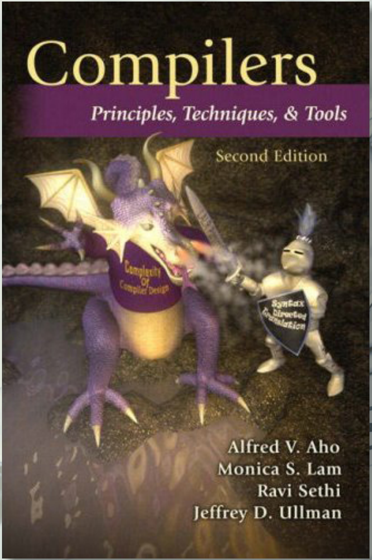
Languages mix approaches

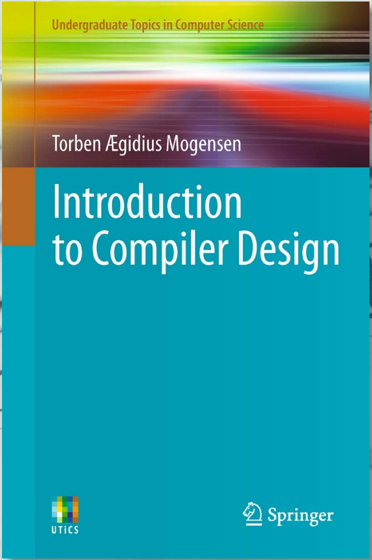
```
// Some C types are weak
union{ int x_int;
       float y_real;} a;
a.x_int = 44;
printf("%f",a.y_real);
```

Dynamic typing

all type checks during interpretation/execution

Torben Ægidius Mogensen. *Introduction to Compiler Design*.
© Springer-Verlag London Limited 2011 ← Slide inspired by Fig. 5.1 therein


Compilers
Principles, Techniques, & Tools
Second Edition
Alfred V. Aho
Monica S. Lam
Ravi Sethi
Jeffrey D. Ullman


Undergraduate Topics in Computer Science
Torben Ægidius Mogensen
Introduction
to Compiler Design
UTiCS
Springer

Syntax and Semantics

concrete/abstract syntax boundary blurred by compilers
included into IDEs; trace back to concrete syntax

Concrete syntax

A representation of a program as text, with whitespace, parentheses, curly braces, and so on, as in “3+(a)”

Abstract syntax

A representation of a program as a tree, either a datatype term or an object structure; whitespace, parentheses and so on abstracted away; simplifies the processing, interpretation and compilation

Static semantics

compile-time correctness of a program: are variables declared? is it well-typed? properties that can be checked without execution

Dynamic semantics

the meaning or effect of a program at run-time; what happens when it is executed? Dynamic semantics is captured by generated code or interpreter

name analysis → type checking → dependent types
→ static flow analysis → operational semantics

The syntax-semantics boundary is a convenience distinction as well
types/declarations can theoretically be tracked by grammars

Definitions after

Peter Sestoft. *Programming Language Concepts* © Springer-Verlag, London 2012

Main Syntactic Categories

Usually defined by sub-languages corresponding to grammar non-terminals

Expression†

A program part whose main purpose is to compute a value

Statement

A program part whose main purpose is to modify the state of the computation (modifying store, producing an output, etc.)

Declaration

specifies type, identifier and other aspects of program elements such as variables and functions

Java examples

```
// expressions
Math.abs(x)*4 - z
x++
(a) ? 100 : -1
// statements
while (true) { ... }
if ( x==0 ) { ... }
else { ... }
// declarations
int a;
protected: bool visit ();
class ModelViewer { ... };
```

† In most imperative programming languages expressions may also produce side-effects, so the change program state, but this is not their main purpose, for example incrementation (x++) in Java

Polymorphism

Subtype polymorphism & virtual Method Calls

```
class SomeCube {
  double capacity () { return 2.0; }
}
class StandardCube extends SomeCube {
  double capacity() { return 1.0; }
}
...
SomeCube c = new StandardCube();
return c.capacity();        // returns 1.0
```

functions take arguments that subtype formal parameter types.

Methods are resolved dynamically to actual classes.

Parametric polymorphism (generics)

```
ArrayList<Person> p=new ArrayList<Person>();
p.add(new Person("Kirsten"));
p.add(new Exception("Bo"));//compile-
time error
Person q = p.get(2);        //no cast needed
```

code works on types "with holes", ignoring concrete types; generic programming

Examples inspired by
Peter Sestoft. *Java Precisely.* 2nd Edition. The MIT Press 2005.

Function Closures

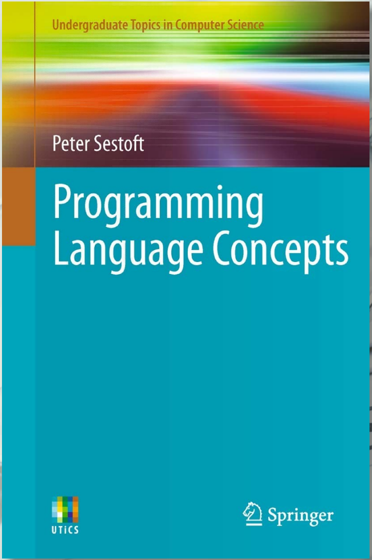
with anonymous functions, aka lambdas, aka delegates

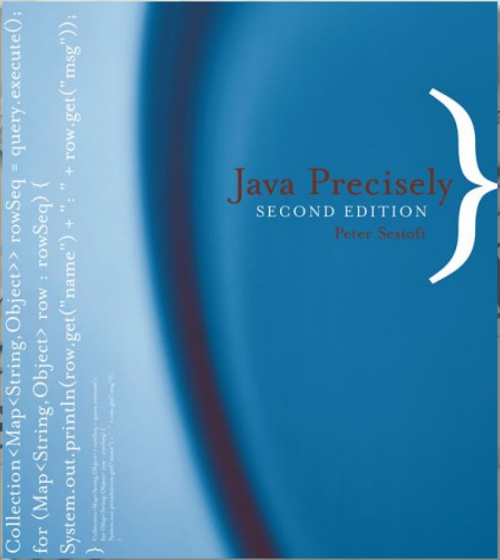
C# Example

```
int c=3;
IEnumerable<int> numbers =
     new List<int>() { 1, 2, 3, 4, 5 };
IEnumerable<int> multiplied =
     numbers.Select(i => c*i);
```

anonymous function returning value of argument multiplied by `c`

variable `c` escapes syntactic scope during execution of `Select`


Undergraduate Topics in Computer Science
Peter Sestoft
Programming Language Concepts
UTiCS
Springer


Collection<Map<String, Object>> rowSeq = query.execute();
for (Map<String, Object> row : rowSeq) {
System.out.println(row.get("name") + " : " + row.get("msg"));
}
Java Precisely
SECOND EDITION
Peter Sestoft

Design by Contract in a Nutshell

A stronger alternative to types

- **pre-condition**: a property that is **assumed** to hold **before** function call
- **post-condition**: a property **guaranteed** to hold **after** the function call
- **invariant**: a property that **always** holds
  - **class invariant**: a property that is guaranteed to hold about class objects all the time (usually in between class method calls).
  - loop invariant: a property that holds for all loop iterations (so before the first iteration, and after each subsequent iteration).

```
class Person {
  // invariant: age >= 0
  int age;
  // post-condition: getAge() >= 0
  int getAge ();
  // pre-condition: n >= 0
  // post-condition: getAge() == n
  int setAge (int n);
}
```