Compiler Construction

Hanspeter Mössenböck

University of Linz

http://ssw.jku.at/Misc/CC/

Text Book

N.Wirth: Compiler Construction, Addison-Wesley 1996
http://www.ethoberon.ethz.ch/WirthPubl/CBEAll.pdf

1. Overview

Why should I learn about compilers?

It's part of the general background of any software engineer

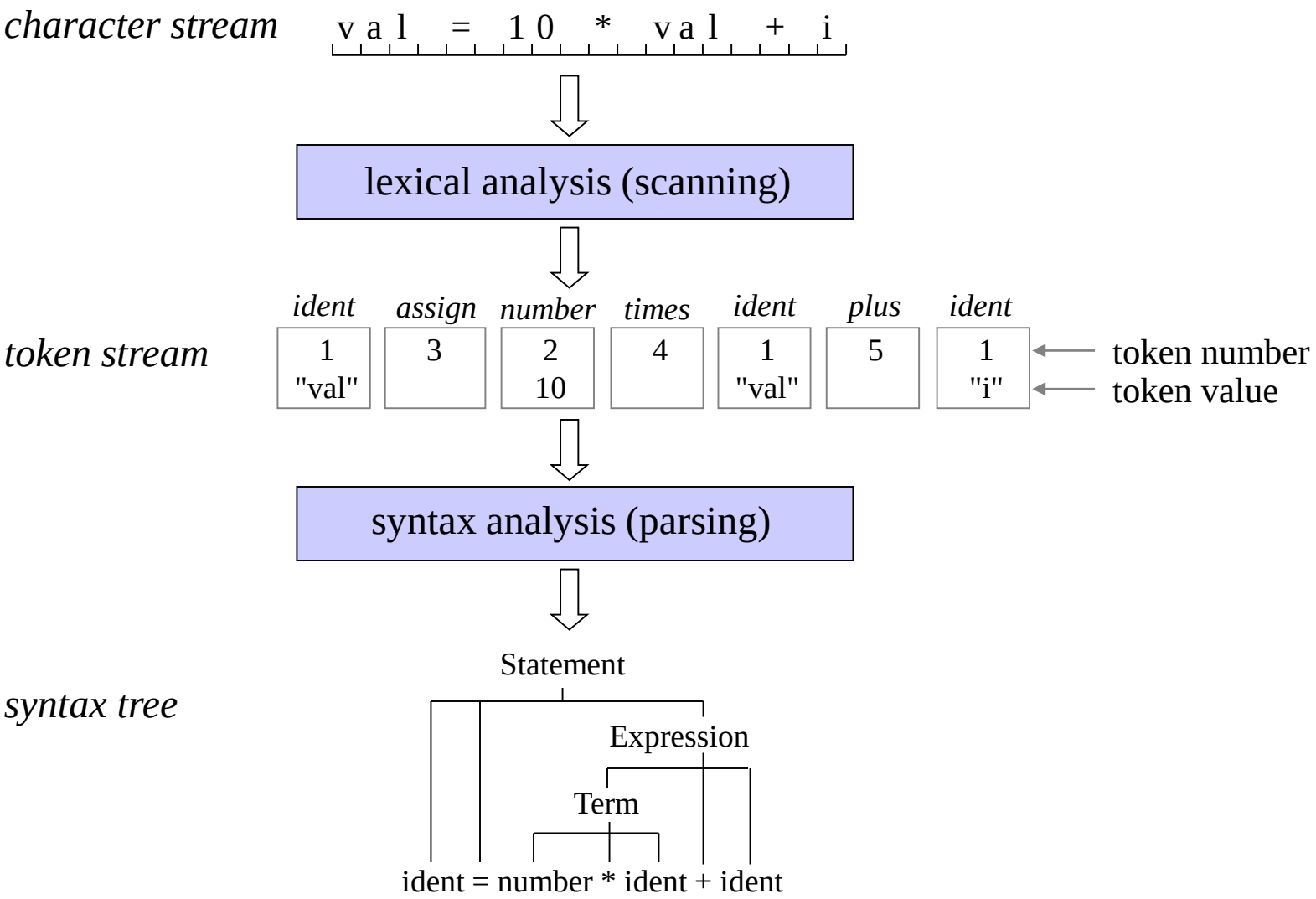
- How do compilers work?
- How do computers work?
  (instruction set, registers, addressing modes, run-time data structures, ...)
- What machine code is generated for certain language constructs?
  (efficiency considerations)
- What is good language design?
- Opportunity for a non-trivial programming project

Also useful for general software development

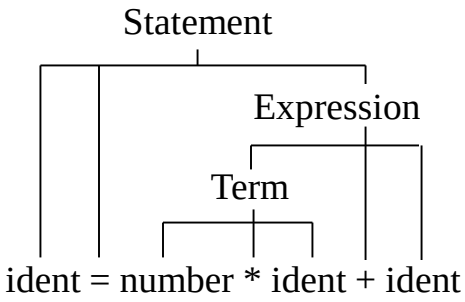
- Reading syntactically structured command-line arguments
- Reading structured data (e.g. XML files, part lists, image files, ...)
- Searching in hierarchical namespaces
- Interpretation of command codes
- ...

1. Overview

Dynamic Structure of a Compiler



Dynamic Structure of a Compiler



syntax tree

semantic analysis (type checking, ...)

intermediate representation

syntax tree, symbol table, ...

optimization

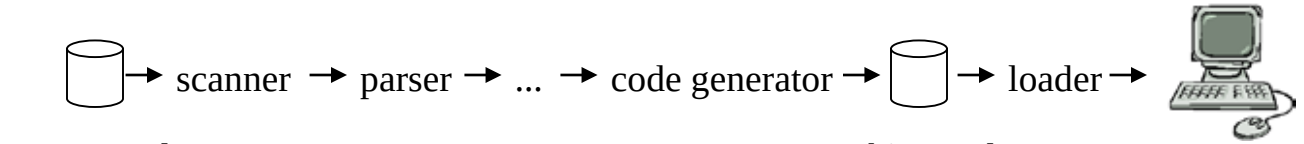
code generation

machine code

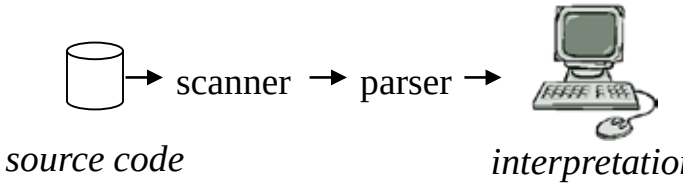
```
const 10
load  1
mul
...
```

Compiler versus Interpreter

Compiler translates to machine code

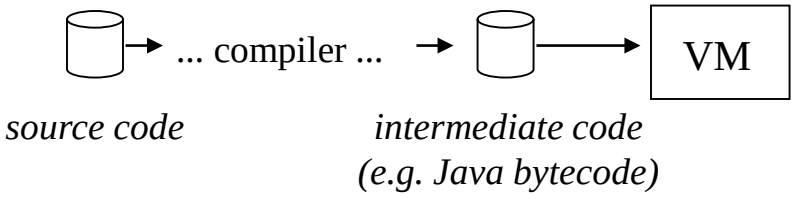


Interpreter executes source code "directly"



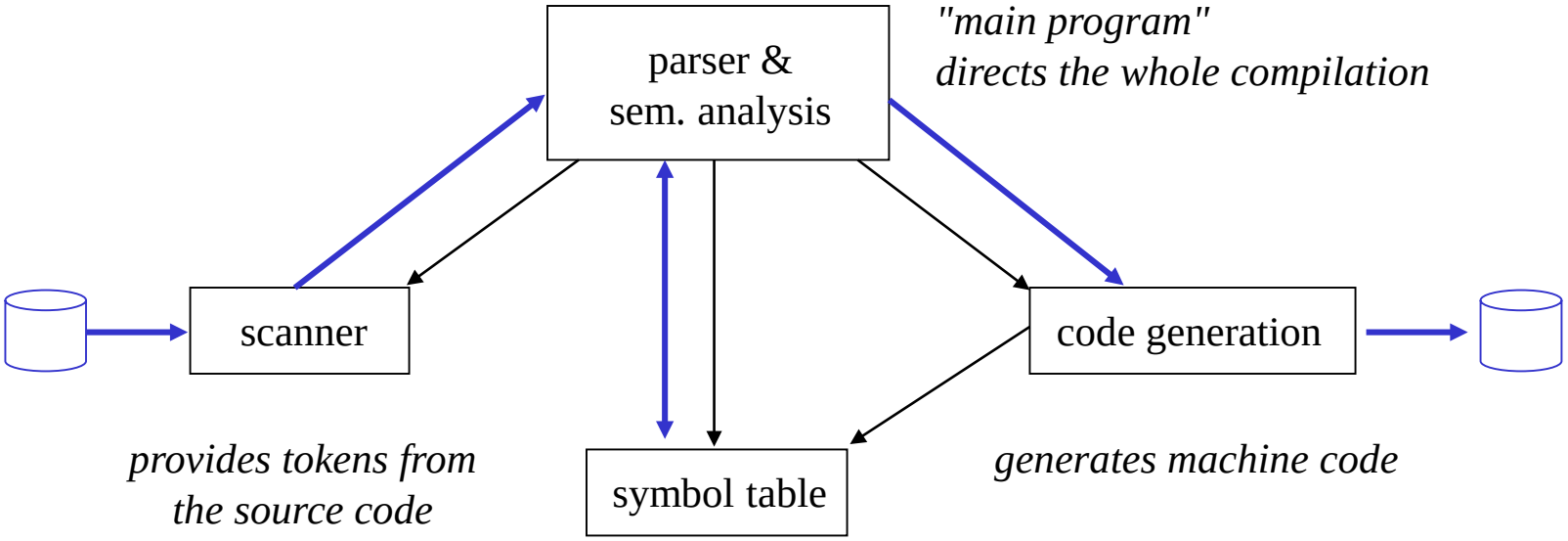
- statements in a loop are scanned and parsed again and again

Variant: interpretation of intermediate code



- source code is translated into the code of a *virtual machine* (VM)
- VM interprets the code simulating the physical machine

Static Structure of a Compiler



uses

data flow

1. Overview

What is a grammar?

Example

```
Statement = "if" "(" Condition ")" Statement ["else" Statement].
```

Four components

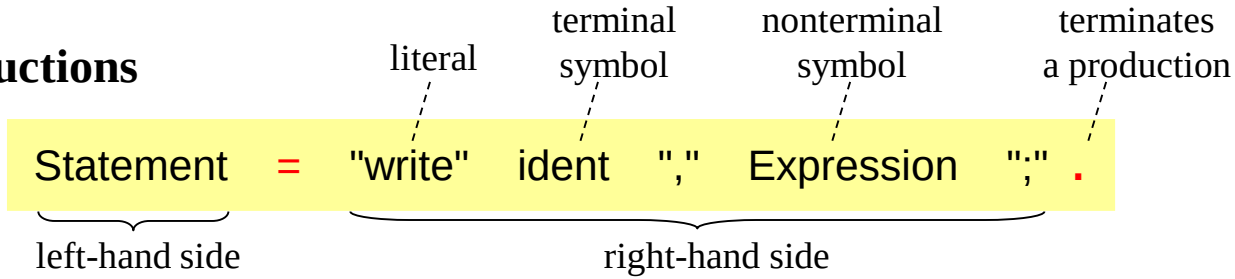
terminal symbols	are atomic	"if", ">=", ident, number, ...
nonterminal symbols	are decomposed into smaller units	Statement, Condition, Type, ...
productions	rules how to decompose nonterminals	Statement = Designator "=" Expr ";". Designator = ident ["." ident]. ...
start symbol	topmost nonterminal	Java

EBNF Notation

Extended Backus-Naur form for writing grammars

John Backus: developed the first Fortran compiler
Peter Naur: edited the Algol60 report

Productions



by convention
- terminal symbols start with lower-case letters
- nonterminal symbols start with upper-case letters

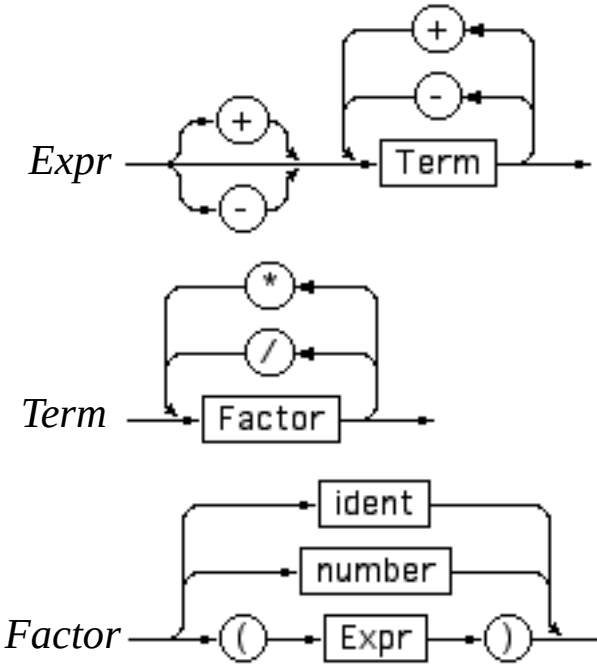
Metasymbols

|	separates alternatives	a | b | c	º a or b or c
(...)	groups alternatives	a (b | c)	º ab | ac
[...]	optional part	[a] b	º ab | b
{...}	iterative part	{a}b	º b | ab | aab | aaab | ...

Example: Grammar for Arithmetic Expressions

Productions

```
Expr   = ["+" | "-"] Term {("+" | "-") Term}.
Term   = Factor {("*" | "/") Factor}.
Factor = ident | number | "(" Expr ")".
```



Terminal symbols

simple TS: "+", "-", "*", "/", "(", ")" (just 1 instance)

terminal classes: ident, number (multiple instances)

Nonterminal symbols

Expr, Term, Factor

Start symbol

Expr

Terminal Start Symbols of Nonterminals

What are the terminal symbols with which a nonterminal can start?

```
Expr   = ["+" | "-"] Term {("+" | "-") Term}.
Term   = Factor {("*" | "/") Factor}.
Factor = ident | number | "(" Expr ")".
```

First(Factor) = ident, number, "("

First(Term) = First(Factor)
= ident, number, "("

First(Expr) = "+", "-", First(Term)
= "+", "-", ident, number, "("

Terminal Successors of Nonterminals

Which terminal symbols can follow a nonterminal in the grammar?

```
Expr   = ["+" | "-"] Term {("+" | "-") Term}.
Term   = Factor {("*" | "/") Factor}.
Factor = ident | number | "(" Expr ")".
```

Follow(Expr) = ")", eof

Follow(Term) = "+", "-", Follow(Expr)
= "+", "-", ")", eof

Follow(Factor) = "*", "/", Follow(Term)
= "*", "/", "+", "-", ")", eof

Where does *Expr* occur on the right-hand side of a production? What terminal symbols can follow there?

Strings and Derivations

String

A finite sequence of symbols from an alphabet.
Alphabet: all terminal and nonterminal symbols of a grammar.

Strings are denoted by greek letters (α, β, γ, ...)
e.g: α = ident + number
β = - Term + Factor * number

Empty String

The string that contains no symbol (denoted by ε).

Derivation

$\alpha \Rightarrow \beta$ (direct derivation)

$$\underbrace{\text{Term} + \underbrace{\text{Factor}}_{\text{NTS}} * \text{Factor}}_{\alpha} \Rightarrow \underbrace{\text{Term} + \underbrace{\text{ident}}_{\text{right-hand side of a production of NTS}} * \text{Factor}}_{\beta}$$

$\alpha \Rightarrow^* \beta$ (indirect derivation) $\quad \alpha \Rightarrow \gamma_1 \Rightarrow \gamma_2 \Rightarrow \ldots \Rightarrow \gamma_n \Rightarrow \beta$

Recursion

A production is recursive if $X \Rightarrow^* w_1 X w_2$

Can be used to express repetitions and nested structures

Direct recursion $X \Rightarrow w_1 X w_2$

Left recursion	X = b | X a.	X ⇒ X a ⇒ X a a ⇒ X a a a ⇒ b a a a a a ...
Right recursion	X = b | a X.	X ⇒ a X ⇒ a a X ⇒ a a a X ⇒ ... a a a a a b
Central recursion	X = b | "(" X ")".	X ⇒ (X) ⇒ ((X)) ⇒ (((X))) ⇒ (((... (b)...)))

Indirect recursion $X \Rightarrow^* w_1 X w_2$

Example

```
Expr   = Term {"+" Term}.
Term   = Factor {"*" Factor}.
Factor = id | "(" Expr ")".
```

Expr ⇒ Term ⇒ Factor ⇒ "(" Expr ")"

How to Remove Left Recursion

Left recursion cannot be handled in topdown parsing

X = b | X a. Both alternatives start with *b*.
The parser cannot decide which one to choose

Left recursion can always be transformed into iteration

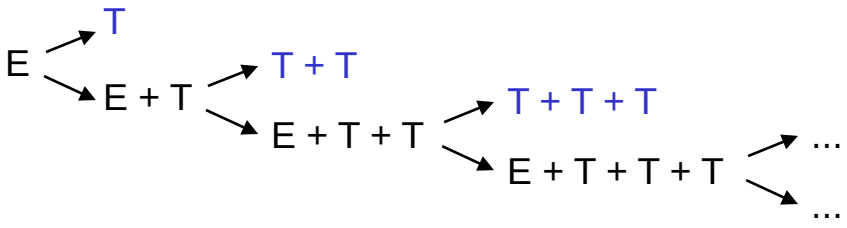
X ⊢ baaaa...a

X = b {a} .

Another example

E = T | E "+" T.

What phrases can be derived?



Thus

E = T {"+" T}.

1. Overview

Classification of Grammars

Due to Noam Chomsky (1956)

Grammars are sets of productions of the form a = b.

class 0 **Unrestricted grammars** (a and b arbitrary)

e.g: X = a X b | Y c Y.
a Y c = d.
d Y = b b.

X ⊢ aXb ⊢ aYcYb ⊢ dYb ⊢ bbb

Recognized by Turing machines

class 1 **Context-sensitive grammars** (|a| ≤ |b|)

e.g: a X = a b c.

Recognized by linear bounded automata

class 2 **Context-free grammars** (a = NT, b ≠ e)

e.g: X = a b c.

Recognized by push-down automata

class 3 **Regular grammars** (a = NT, b = T or T NT)

e.g: X = b | b Y.

Recognized by finite automata

Only these two classes are relevant in compiler construction

1. Overview

Sample MicroJava Program


```
program P
   final int size = 10;
   class Table {
      int[] pos;
      int[] neg;
   }
   Table val;
{
   void main()
      int x, i;
   {  //---------- initialize val ----------
      val = new Table;
      val.pos = new int[size];
      val.neg = new int[size];
      i = 0;
      while (i < size) {
         val.pos[i] = 0; val.neg[i] = 0; i = i + 1;
      }
      //---------- read values ----------
      read(x);
      while (x != 0) {
         if (x >= 0) val.pos[x] = val.pos[x] + 1;
         else if (x < 0) val.neg[-x] = val.neg[-x] + 1;
         read(x);
      }
   }
}
```

main program; no separate compilation

classes (without methods)

global variables

local variables

Lexical Structure of MicroJava

Identifiers	ident = letter {letter | digit | '_'}.	
Numbers	number = digit {digit}.	all numbers are of type *int*
Char constants	charConst = '\'' char '\''.	all character constants are of type *char* (may contain \r, \n, \t)
no strings		

Keywords

program	class					
if	else	while	read	print	return	void
final	new					

Operators

+	-	*	/	%	
==	!=	>	>=	<	<=
(	)	[	]	{	}
=	;	,	.		

Comments	// ... eol			
Types	*int*	*char*	arrays	classes

Syntactical Structure of MicroJava

Programs

```
Program  =  "program" ident
            {ConstDecl | VarDecl | ClassDecl}
            "{" {MethodDecl} "}".
```

```
program P
   ... declarations ...
{  ... methods ...
}
```

Declarations

```
ConstDecl   =  "final" Type ident "=" (number | charConst) ";".
VarDecl     =  Type ident {"," ident} ";".
MethodDecl  =  (Type | "void") ident "(" [FormPars] ")"
               {VarDecl} Block.

Type        =  ident [ "[" "]" ].
FormPars    =  Type ident {"," Type ident}.
```

just one-dimensional arrays

Syntactical Structure of MicroJava

Statements

```
Block      =  "{" {Statement} "}".
Statement  =  Designator  (   "=" Expr ";"
                            |   "(" [ActPars] ")" ";"
                            )
           |  "if" "(" Condition ")" Statement ["else" Statement]
           |  "while" "(" Condition ")" Statement
           |  "return" [Expr] ";"
           |  "read" "(" Designator ")" ";"
           |  "print" "(" Expr ["," number] ")" ";"
           |  Block
           |  ";".
ActPars    =  Expr {"," Expr}.
```

- input from *System.in*
- output to *System.out*

Syntactical Structure of MicroJava

Expressions

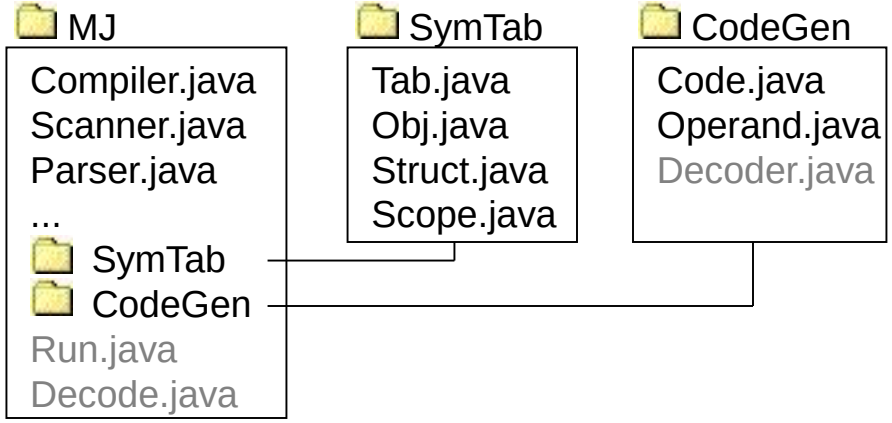
```
Condition   =  Expr Relop Expr.
Relop       =  "==" | "!=" | ">" | ">=" | "<" | "<=".
```

```
Expr        =  ["-"] Term {Addop Term}.
Term        =  Factor {Mulop Factor}.
Factor      =  Designator [ "(" [ActPars] ")" ]
            |  number
            |  charConst
            |  "new" ident [ "[" Expr "]" ]
            |  "(" Expr ")".
Designator  =  ident { "." ident | "[" Expr "]" }.
Addop       =  "+" | "-".
Mulop       =  "*" | "/" | "%".
```

no constructors

The MicroJava Compiler

Package structure



Compilation of a MicroJava program

```
java MJ.Compiler  myProg.mj
```

myProg.mj → compiler → myProg.obj

Execution

```
java MJ.Run  myProg.obj  -debug
```

myProg.obj → interpreter

Decoding

```
java MJ.Decode  myProg.obj
```

myProg.obj → decoder → myProg.code