1. Overview

Short History of Compiler Construction

Formerly "a mystery", today one of the best-known areas of computing

1957	Fortran	first compilers (arithmetic expressions, statements, procedures)
1960	Algol	first formal language definition (grammars in Backus-Naur form, block structure, recursion, ...)
1970	Pascal	user-defined types, virtual machines (P-code)
1985	C++	object-orientation, exceptions, templates
1995	Java	just-in-time compilation

We only look at imperative languages

Functional languages (e.g. Lisp) and logical languages (e.g. Prolog) require different techniques.

Why should I learn about compilers?

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

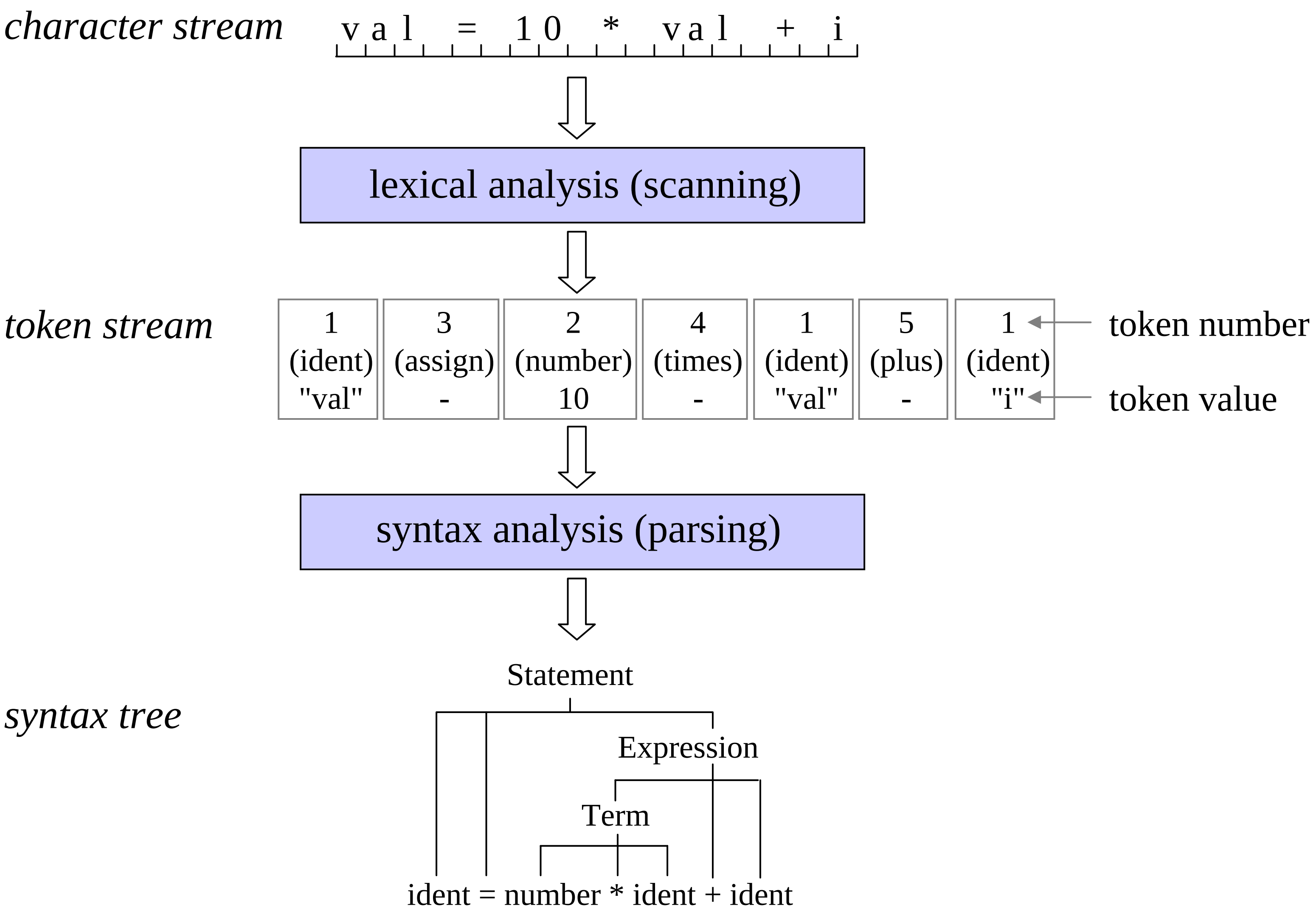
- How do compilers work?
- How do computers work?
  (instruction set, registers, addressing modes, run-time data structures, ...)
- What machine code is generated for certain language constructs?
  (efficiency considerations)
- What is good language design?
- Opportunity for a non-trivial programming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



character stream val = 10 * val + i

lexical analysis (scanning)

token stream

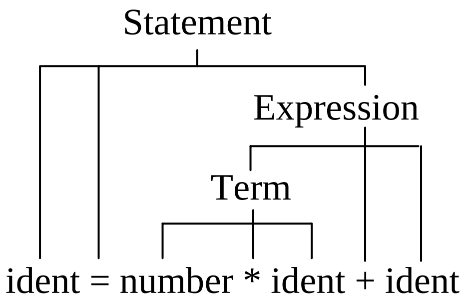
1	3	2	4	1	5	1	token number
(ident)	(assign)	(number)	(times)	(ident)	(plus)	(ident)	
"val"	-	10	-	"val"	-	"i"	token value

syntax analysis (parsing)

syntax tree

Dynamic Structure of a Compiler

syntax tree



Statement

Expression

Term

ident = number * ident + ident

⇩

semantic analysis (type checking, ...)

⇩

intermediate representation

syntax tree, symbol table, ...

⇩

optimization

⇩

code generation

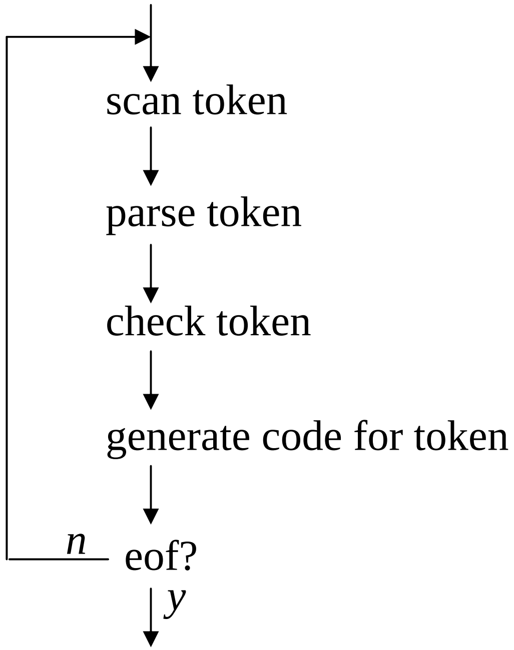
⇩

machine code

ld.i4.s 10
ldloc.1
mul
...

Single-Pass Compilers

Phases work in an interleaved way



scan token

parse token

check token

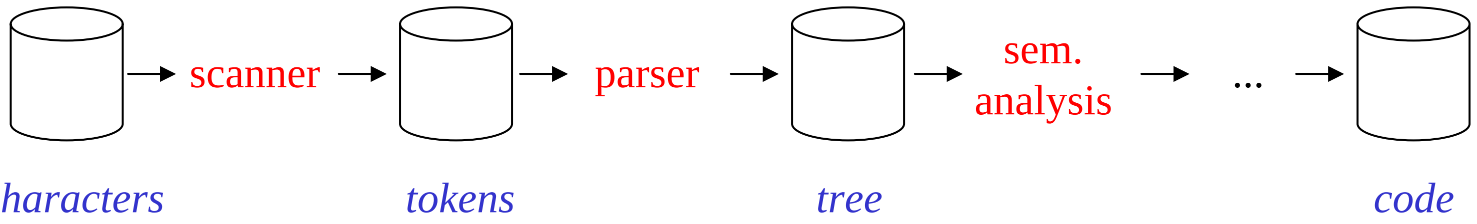
generate code for token

eof?

The target program is already generated while the source program is read.

Multi-Pass Compilers

Phases are separate "programs", which run sequentially

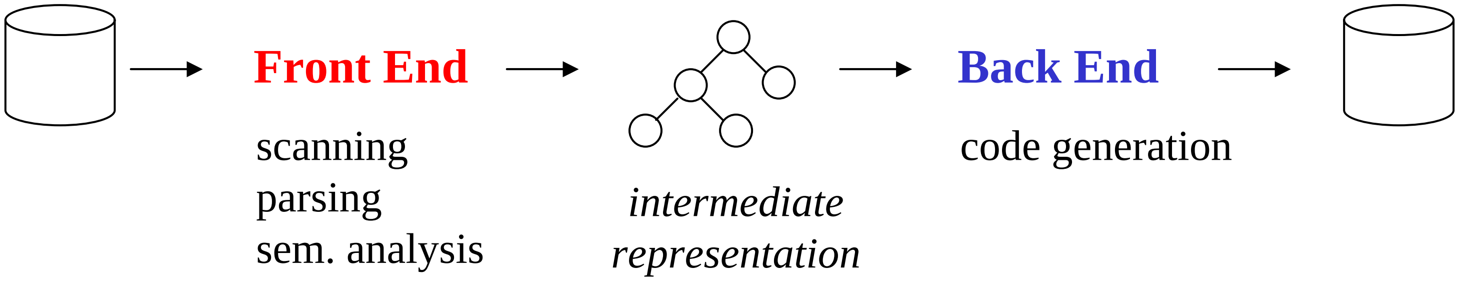


Each phase reads from a file and writes to a new file

Why multi-pass?

- if memory is scarce (irrelevant today)
- if the language is complex
- if portability is important

Today: Often Two-Pass Compilers



Front End

scanning
parsing
sem. analysis

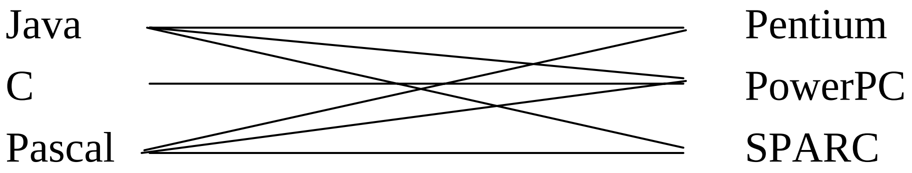
intermediate representation

Back End

code generation

language-dependent

machine-dependent



Java
C
Pascal

Pentium
PowerPC
SPARC

any combination possible

Advantages

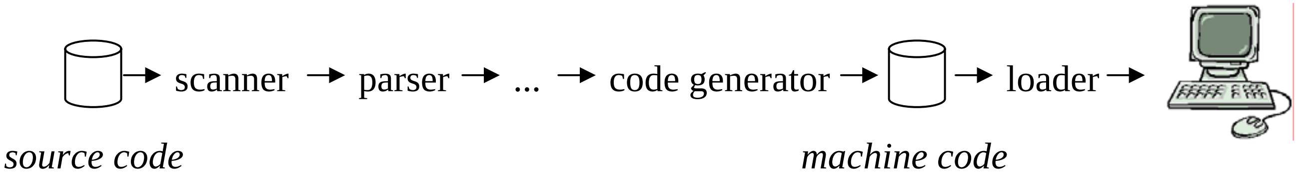
- better portability
- many combinations between front ends and back ends possible
- optimizations are easier on the intermediate representation than on source code

Disadvantages

- slower
- needs more memory

Compiler versus Interpreter

Compiler translates to machine code

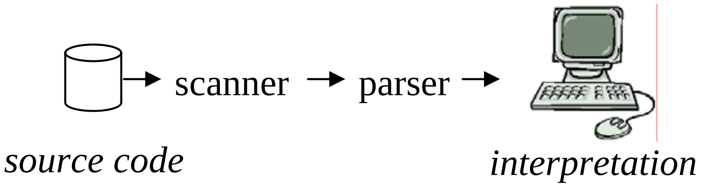


scanner → parser → ... → code generator → loader →

source code

machine code

Interpreter executes source code "directly"



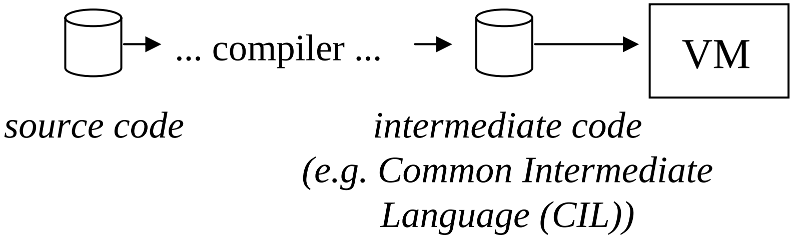
scanner → parser →

source code

interpretation

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

Variant: interpretation of intermediate code



... compiler ... →

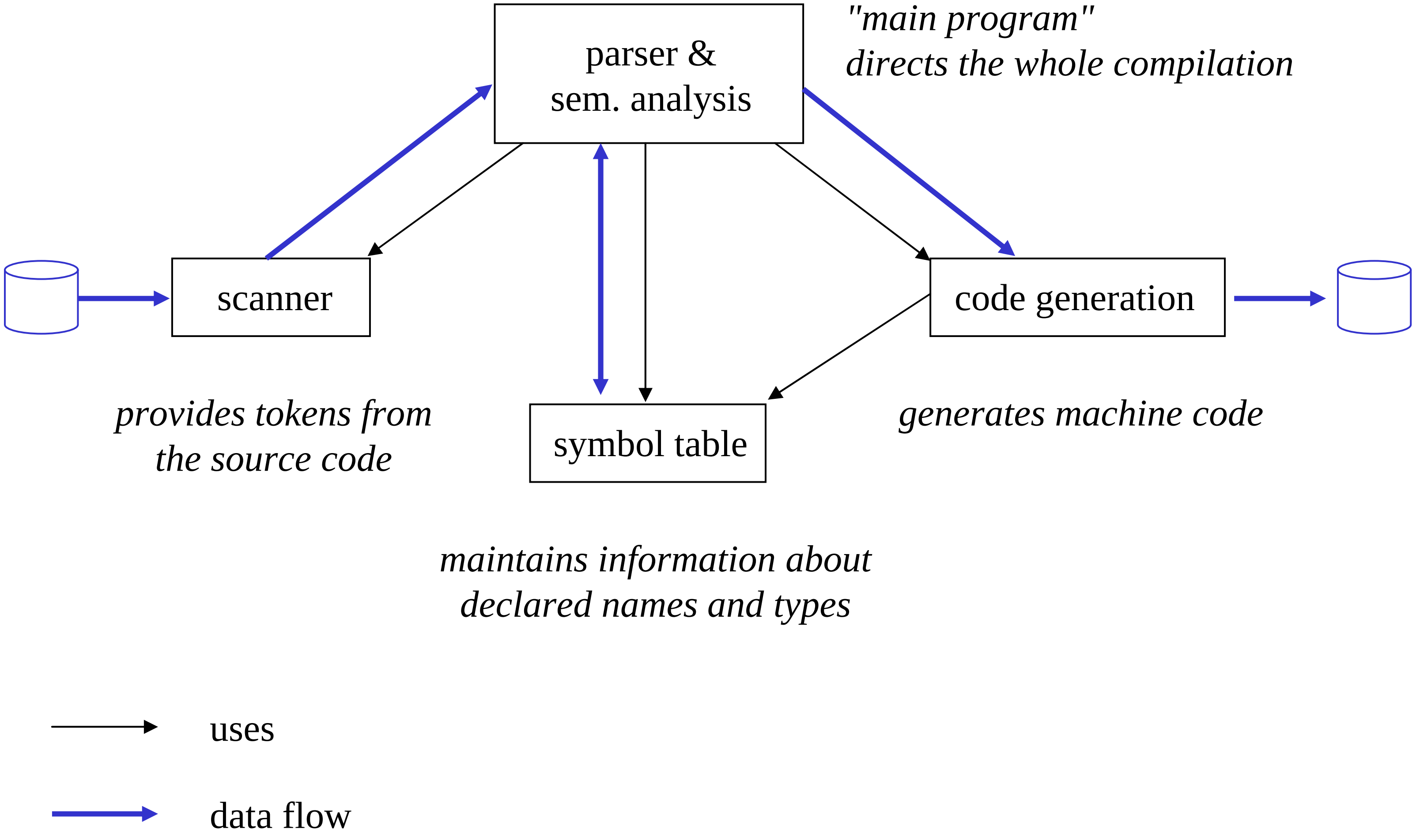
VM

source code

intermediate code (e.g. Common Intermediate Language (CIL))

- 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



parser &
sem. analysis

"main program"
directs the whole compilation

scanner

code generation

symbol table

provides tokens from
the source code

generates machine code

maintains information about
declared names and types

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 derived into smaller units	Statement, Expr, Type, ...
productions	rules how to decompose nonterminals	Statement = Designator "=" Expr ";". Designator = ident ["." ident]. ...
start symbol	topmost nonterminal	CSharp

EBNF Notation

Extended Backus-Naur form

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

symbol	*meaning*	*examples*	
string	denotes itself	"=", "while"	
name	denotes a T or NT symbol	ident, Statement	
=	separates the sides of a production	A = b c d .	
.	terminates a production		
|	separates alternatives	a | b | c	≡ a or b or c
(...)	groups alternatives	a (b | c)	≡ ab | ac
[...]	optional part	[a] b	≡ ab | b
{...}	repetitive part	{ a } b	≡ b | ab | aab | aaab | ...

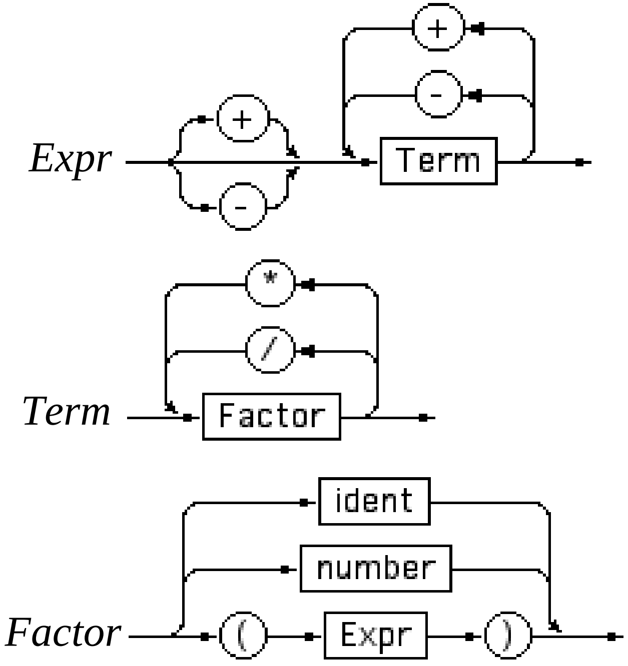
Conventions

- terminal symbols start with lower-case letters (e.g. ident)
- nonterminal symbols start with upper-case letters (e.g. Statement)

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

Operator Priority

Grammars can be used to define the priority of operators

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

input: - a * 3 + b / 4 - c

$$\Rightarrow\quad - \text{ident} * \text{number} + \text{ident} / \text{number} - \text{ident}$$

$$\Rightarrow\quad - \underbrace{\text{Factor} * \text{Factor}} + \underbrace{\text{Factor} / \text{Factor}} - \underbrace{\text{Factor}}$$

$$\Rightarrow\quad \underbrace{-\quad \text{Term} \quad + \quad \text{Term} \quad - \ \text{Term}}$$

$$\Rightarrow\quad \text{Expr}$$

"*" and "/" have higher priority than "+" and "-"

"-" does not refer to *a*, but to *a**3

How must the grammar be transformed, so that "-" refers to *a*?

Terminal Start Symbols of Nonterminals

Which terminal symbols can a nonterminal start with?

```
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 after 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?

Some Terminology

Alphabet

The set of terminal and nonterminal symbols of a grammar

String

A finite sequence of symbols from an alphabet.

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).

Derivations and Reductions

Derivation

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

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

$\alpha \Rightarrow^{*} \beta$ (indirect derivation) $\alpha \Rightarrow \gamma_1 \Rightarrow \gamma_2 \Rightarrow \dots \Rightarrow \gamma_n \Rightarrow \beta$

$\alpha \Rightarrow^{L} \beta$ (left-canonical derivation) the <u>leftmost</u> NTS in α is derived first

$\alpha \Rightarrow^{R} \beta$ (right-canonical derivation) the <u>rightmost</u> NTS in α is derived first

Reduction

The converse of a derivation:
If the right-hand side of a production occurs in β it is replaced with the corresponding NTS

Deletability

A string α is called deletable, if it can be derived to the empty string.
$\alpha \Rightarrow^* \varepsilon$

Example

```
A = B C.
B = [ b ].
C = c | d | .
```

B is deletable: $B \Rightarrow \varepsilon$

C is deletable: $C \Rightarrow \varepsilon$

A is deletable: $A \Rightarrow B\ C \Rightarrow C \Rightarrow \varepsilon$

More Terminology

Phrase

Any string that can be derived from a nonterminal symbol.

Term phrases:	Factor Factor * Factor ident * Factor ...

Sentential form

Any string that can be derived from the start symbol of the grammar.

e.g.: Expr
Term + Term + Term
Term + Factor * ident + Term
...

Sentence

A sentential form that consists of terminal symbols only.

e.g.: ident * number + ident

Language (formal language)

The set of all sentences of a grammar (usually infinitely large).
e.g.: the C# language is the set of all valid C# programs

Recursion

A production is recursive if $\boxed{A \Rightarrow^* \omega_1 \, A \, \omega_2}$

Can be used to represent repetitions and nested structures

Direct recursion $\boxed{A \Rightarrow \omega_1 \, A \, \omega_2}$

Left recursion	A = b | A a.	$A \Rightarrow A\,a \Rightarrow A\,a\,a \Rightarrow A\,a\,a\,a \Rightarrow b\,a\,a\,a\,a\,a \dots$
Right recursion	A = b | a A.	$A \Rightarrow a\,A \Rightarrow a\,a\,A \Rightarrow a\,a\,a\,A \Rightarrow \dots a\,a\,a\,a\,a\,b$
Central recursion	A = b | "(" A ")".	$A \Rightarrow (A) \Rightarrow ((A)) \Rightarrow (((A))) \Rightarrow (((\dots (b)\dots)))$

Indirect recursion $\boxed{A \Rightarrow^* \omega_1 \, A \, \omega_2}$

Example

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

$\text{Expr} \Rightarrow \text{Term} \Rightarrow \text{Factor} \Rightarrow \text{"(" Expr ")"}$

How to Remove Left Recursion

Left recursion cannot be handled in topdown syntax analysis

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

Left recursion can be transformed to iteration

```
E = T | E "+" T.
```

What sentences can be derived?

```
T
T + T
T + T + T
...
```

From this one can deduce the iterative EBNF rule:

```
E = T { "+" T }.
```

1. Overview

Plain BNF Notation

terminal symbols	are written without quotes (ident, +, -)
nonterminal symbols	are written in angle brackets (<Expr>, <Term>)
sides of a production	are separated by ::=

BNF grammar for arithmetic expressions

```
<Expr>    ::=  <Sign> <Term>
<Expr>    ::=  <Expr> <Addop> <Term>

<Sign>    ::=  +
<Sign>    ::=  -
<Sign>    ::=

<Addop>   ::=  +
<Addop>   ::=  -

<Term>    ::=  <Factor>
<Term>    ::=  <Term> <Mulop> <Factor>

<Mulop>   ::=  *
<Mulop>   ::=  /

<Factor>  ::=  ident
<Factor>  ::=  number
<Factor>  ::=  ( <Expr> )
```

- Alternatives are transformed into separate productions
- Repetition must be expressed by recursion

Advantages

- fewer meta symbols (no |, (), [], {})
- it is easier to build a syntax tree

Disadvantages

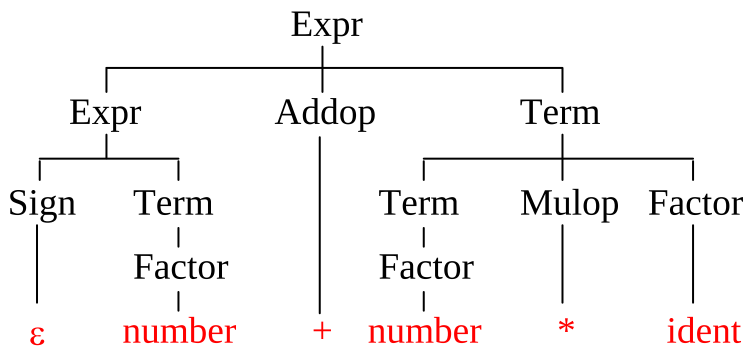
- more clumsy

Syntax Tree

Shows the structure of a particular sentence

e.g. for 10 + 3 * i

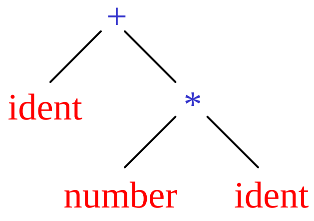
Concrete Syntax Tree (parse tree)



Would not be possible with EBNF because of [...] and {...}, e.g.:
Expr = [Sign] Term { Addop Term }.

Also reflects operator priorities: operators further down in the tree have a higher priority than operators further up in the tree.

Abstract Syntax Tree (leaves = operands, inner nodes = operators)



often used as an internal program representation;
used for optimizations

Ambiguity

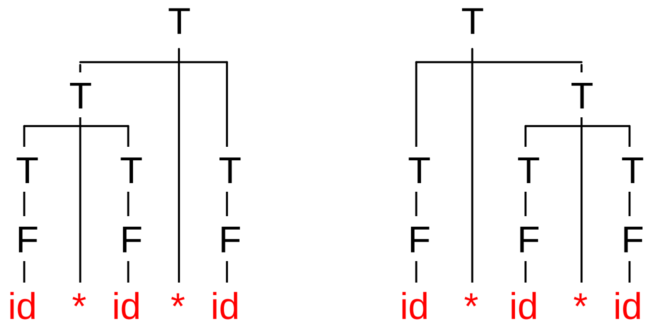
A grammar is ambiguous, if more than one syntax tree can be built for a given sentence.

Example

```
T = F | T "*" T.
F = id.
```

sentence: id * id * id

Two syntax trees can be built for this sentence:



Ambiguous grammars cause problems in syntax analysis!

Removing Ambiguity

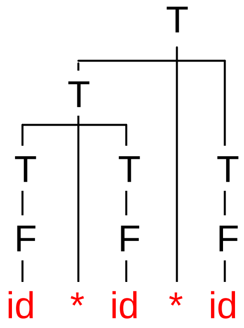
Example

T = F | T "*" T.
F = id.

Only the grammar is ambiguous, not the language.

The grammar can be transformed to

T = F | T "*" F.
F = id.



i.e. T has priority over F

only this syntax tree is possible

Even better: transformation to EBNF

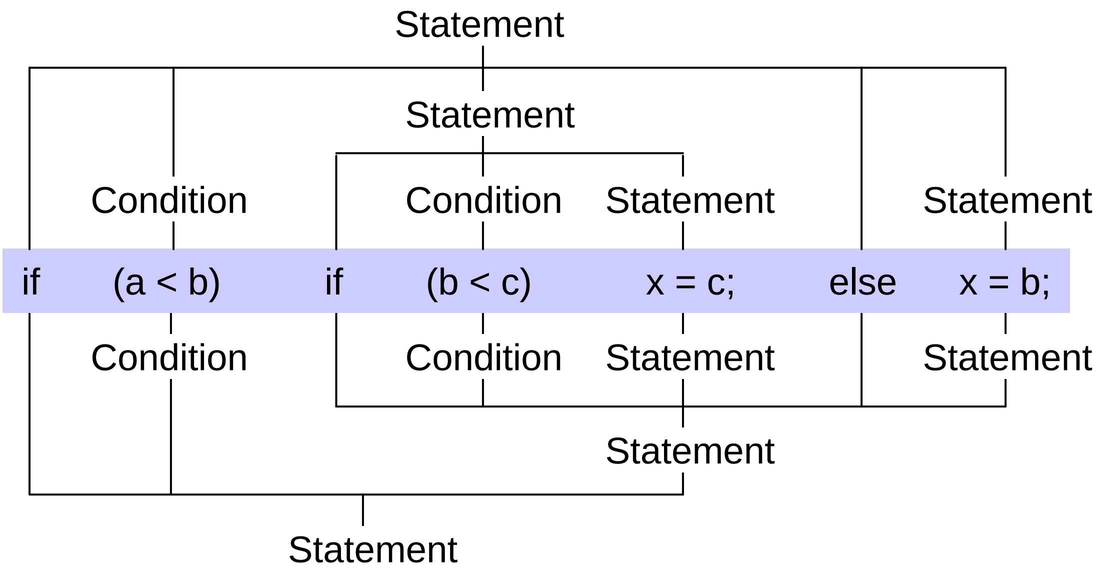
T = F { "*" F }.
F = id.

Inherent Ambiguity

There are languages which are inherently ambiguous

Example: *Dangling Else*

```
Statement =  Assignment
          |  "if" Condition Statement
          |  "if" Condition Statement "else" Statement
          |  ... .
```



There is no unambiguous grammar for this language!

C# solution

Always recognize the longest possible right-hand side of a production

⇒ leads to the lower of the two syntax trees

1. Overview

1.1 Motivation

1.2 Structure of a Compiler

1.3 Grammars

1.4 Syntax Tree and Ambiguity

1.5 Chomsky's Classification of Grammars

1.6 The Z# Language

Classification of Grammars

Due to Noam Chomsky (1956)

Grammars are sets of productions of the form $\alpha = \beta$.

class 0

Unrestricted grammars (α and β arbitrary)

e.g: A = a A b | B c B.
aBc = d.
dB = bb.

$A \Rightarrow aAb \Rightarrow aBcBb \Rightarrow dBb \Rightarrow bbb$

Recognized by Turing machines

class 1

Contex-sensitive grammars ($|\alpha| \leq |\beta|$)

e.g: a A = a b c.

Recognized by linear bounded automata

class 2

Context-free grammars (α = NT, $\beta \neq \varepsilon$)

e.g: A = a b c.

Recognized by push-down automata

class 3

Regular grammars (α = NT, β = T | T NT)

e.g: A = b | b B.

Recognized by finite automata

Only these two classes are relevant in compiler construction

1. Overview

Sample Z# Program

```
class P
    const 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++;
        }
        /*---------- read values ----------*/
        read(x);
        while (-size < x && x < size) {
            if (0 <= x) { val.pos[x]++; }
            else { val.neg[-x]++; }
            read(x);
        }
    }
}
```

main program class; no separate compilation

inner classes (without methods)

global variables

local variables

Lexical Structure of Z#

Names 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 and \n)

<u>no</u> strings

Keywords

class						
if	else	while	read	write	return	break
void	const	new				

Operators

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

Comments /* ... */ may be nested

Types *int* *char* arrays classes

Syntactical Structure of Z# (1)

Programs

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

```
class P
   ... declarations ...
{  ... methods ...
}
```

Declarations

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

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

only one-dimensional arrays

Syntactical Structure of Z# (2)

Statements

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

- input from *System.Console*
- output to *System.Console*

Syntactical Structure of Z# (3)

Expressions

```
Condition    = CondTerm { "||" CondTerm }.
CondTerm     = CondFact { "&&" CondFact }.
CondFact     = Expr Relop Expr.
Relop        = "==" | "!=" | ">" | ">=" | "<" | "<=".
```

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

no constructors