Compiler Construction: Practical Introduction

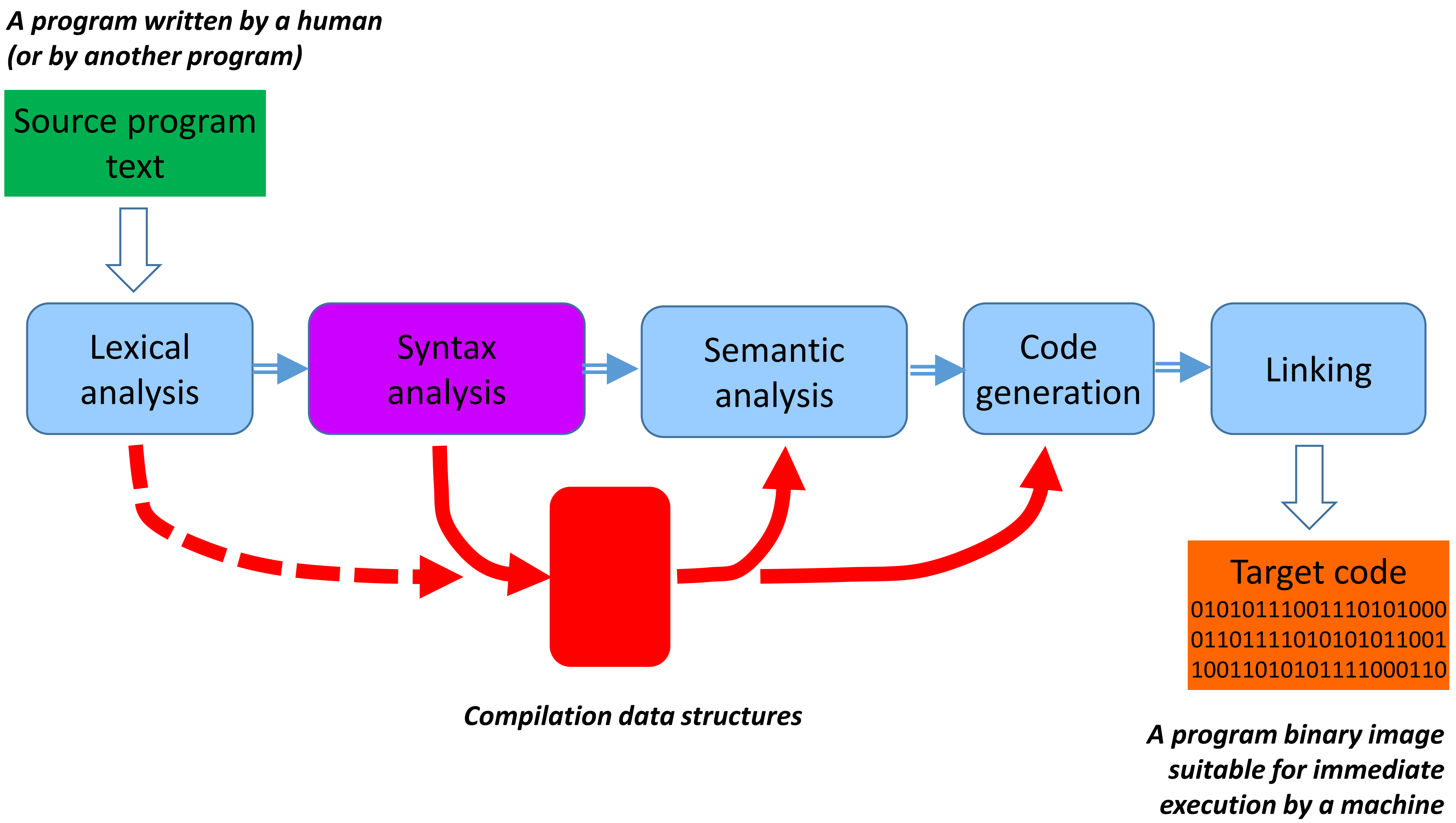
Samsung Compiler Bootcamp

Samsung Research Russia
Moscow 2019

Lecture 2
Syntax Analysis

- Syntax analysis: why and what for?
- Formal grammars
- Implementation approaches
- Top-down vs. Bottom-up
- Development techniques.

Compilation: An Ideal Picture

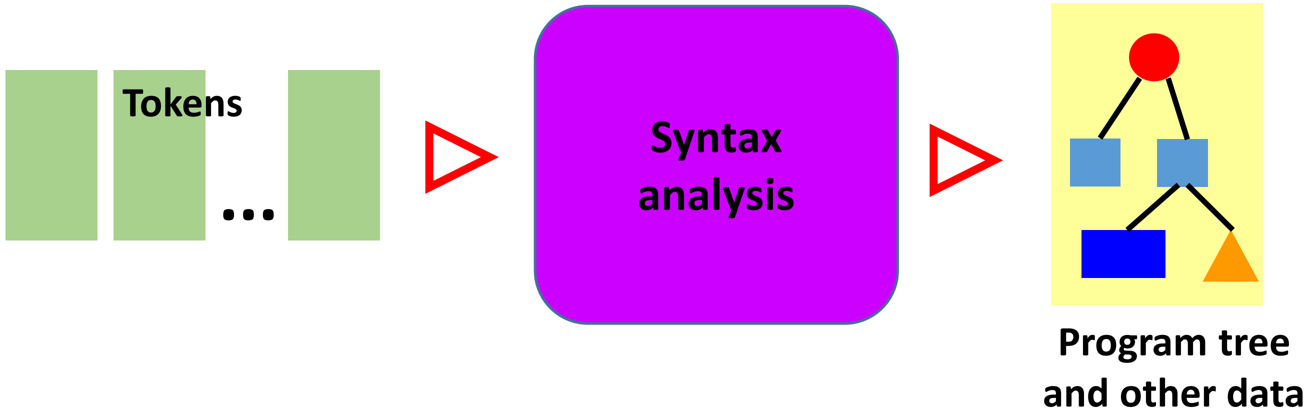


Syntax Analysis

Syntax analysis: why & what for?

- To **check correctness** of the syntactic structure of the source program in accordance with the language grammar.
- To **convert** the source program to an intermediate regular form (representation) which is suitable for subsequent processing (semantic analysis, optimization, code generation).

  The intermediate representation must be **semantically equivalent** to the source program.
- Syntax analysis can be done (completely or partially) *together with semantic analysis* (for simple languages).



Syntax Analysis

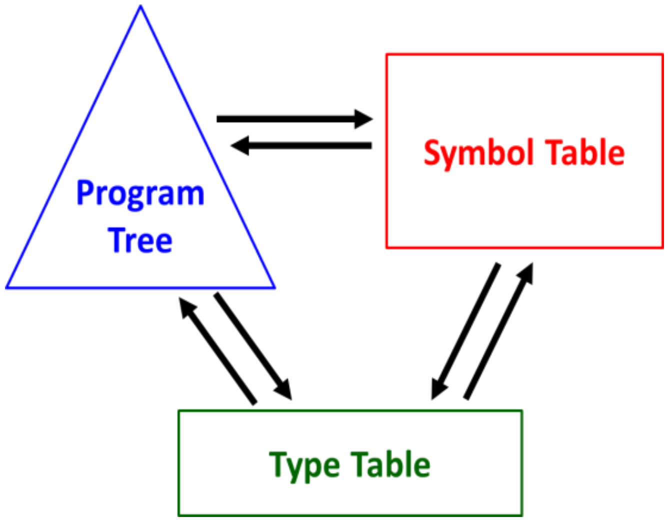
The result of syntax analysis:

an **internal program representation**.

Example:

a **tree structure**, whose nodes and sub-trees correspond to structure elements of the source program.

Tree is not mandatory and not the only possible program representation (*details follow later*).



Even if there is **no tree** while syntax analysis (simplest cases), it exists **implicitly** ("the parser reflects the tree while running").

Syntax Analysis: Theory

- Theoretical basis for syntax analysis:
  formal grammar theory.
  Main results were in 60-80s.
- Powerful and efficient algorithms were designed and implemented based on the theory.
- Any parsing program implements (implicitly or explicitly) a particular algorithm of syntax analysis.

Syntax Analysis: Theory

С. З. Свердлов
Языки программирования и методы трансляции
Издательство ПИТЕР, 2007
ISBN 978-5-469-00378-6

Глава «Теоретические основы трансляции»:

- Carefully selected and small amount of information related to the theory;
- Simple and clearly written explanations.

Syntax Analysis: Implementation Techniques

- Despite of good theoretical basis, each language requires **individual approach**. The common theory rarely works for 100% (almost never ☺).
- There are powerful and convenient **tools** for automated parser generation (YACC/Bison – will be considered in details on the next lecture).
- Anyway, the basis for any parser is a formal (or semi-formal) **language grammar definition**.

Syntax Analysis: Implementation Techniques

- Almost any language definition contains its (more or less) **formal grammar specification**.
- Typically, the grammar is just for information, but not for compiler writers. Therefore, to create a parser using the grammar from the language definition, we have **to transform** it.

  Russian translation of the 1st edition of the “Dragon book”: examples of C++ & C# grammars ready for implementation.

- Some syntactical details just **cannot be represented** in the grammar – we have to take them into account in implementation.

  Example: variable initialization in declarations.

Compiler Development Technologies

(Syntax analysis as an example.)

- Top-down or bottom-up parsing?
- «Hand-made» or automated development?

	By hand	By tools: ***parser generators***
Top-down parsing	Most often: **Recursive descent parser**	ANTLR COCO etc.
Bottom-up parsing	Rarely (too complicated)	Yacc, Bison, Jay, GPPG, etc.

- **The most of real compilers are "hand made" ☺.**

The gcc C++parser was initially implemented using bison but later reimplemented as a recursive descent parser

Syntax Analysis: Implementation Techniques

- **Top-down parsing**

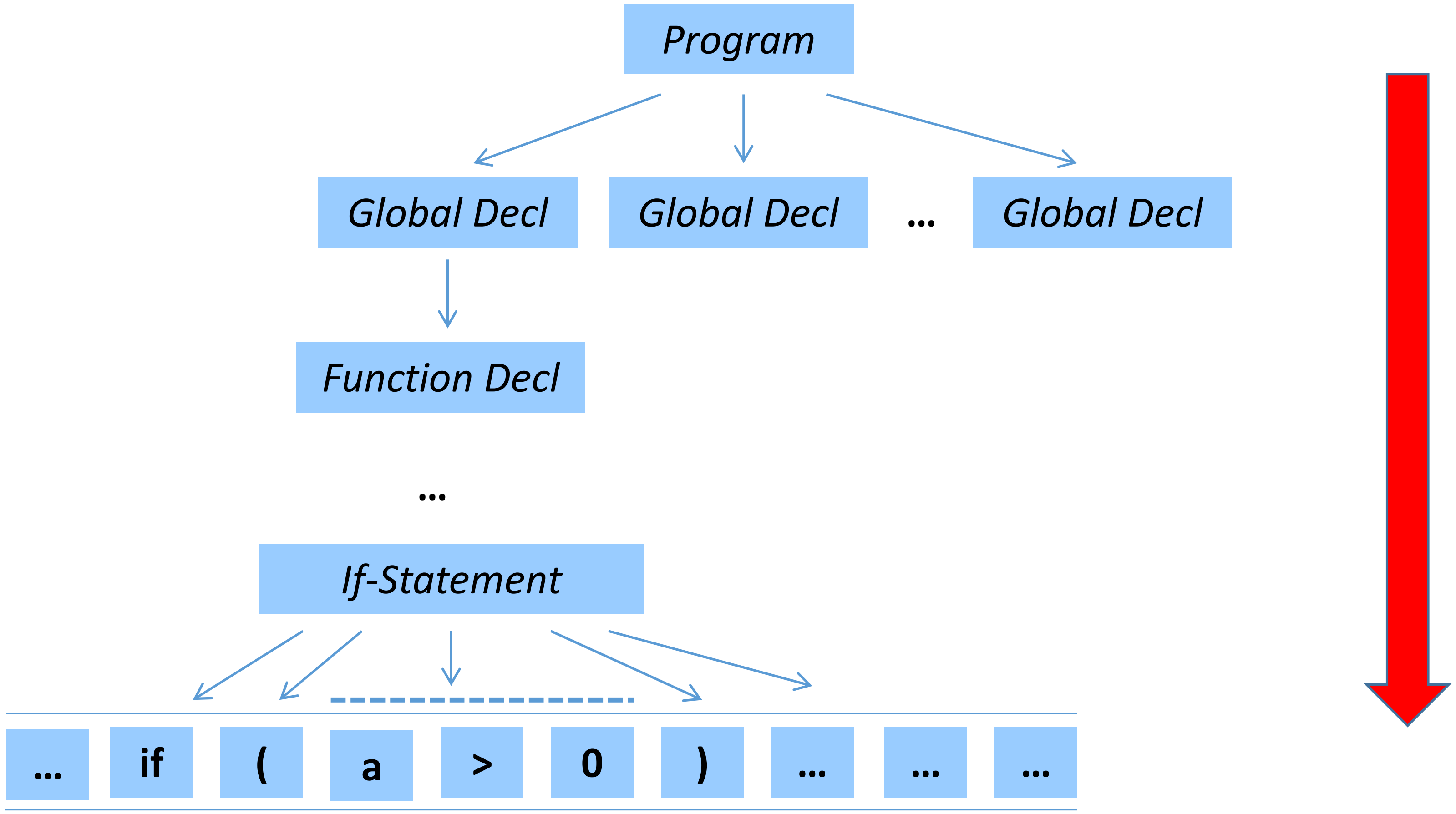
  The common parsing direction – starting from more common language notions (non-terminals) to more concrete, down to tokens.

  Parsing algorithm is organized in accordance with language grammar rules: syntax-directed (syntax-driven) parsing.

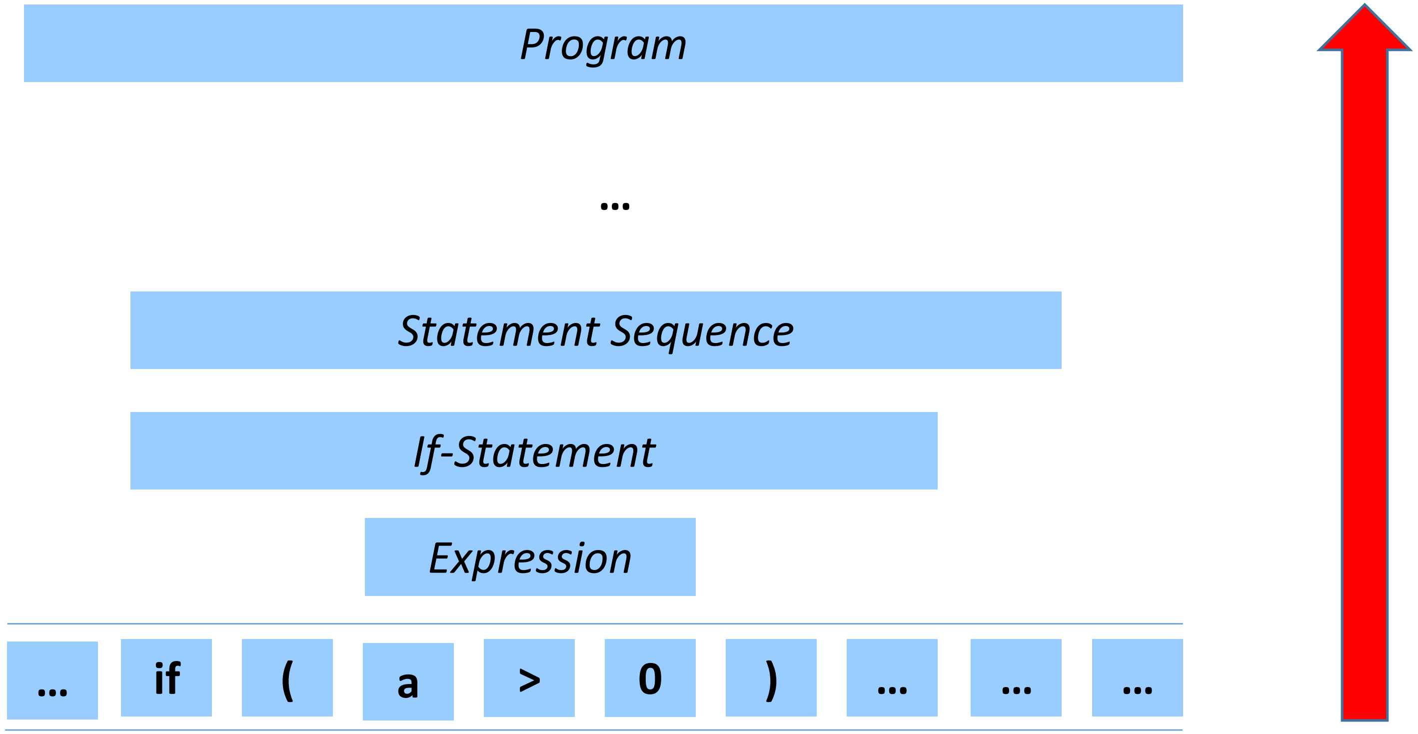
- **Bottom-up parsing**

  The common parsing direction – from source tokens to grammar rules; the algorithm tries to reduce token sequences to more common non-terminal grammar symbols.

Top-down Parsing



Bottom-up Parsing



Syntax Analysis: Implementation Techniques

- **Top-down OR bottom-up?**

T>b Top-down algorithm is much easier to implement. Bottom up requires much more efforts (typically it's table-driven).

t<B Top-down algorithm is less stable in case of syntax errors. Error recovery techniques are much harder to implement than for bottom-up..

t<B Top-down algorithm is less refactored: "syntax part" of the compiler is typically not a separate part of the compiler but spread over its source text. It's much harder to modify & maintain.

T>b Interface between the top-down parser & other compiler components (passes) can be organized in a more clear & convenient way than for the bottom-up parser.

Syntax Analysis: Implementation Techniques

- **«Hand made» development OR automation tools?**

h<A Tools significantly **speed up** parser development (if you know how to use them ☺).

H>a Significant effort are required **to adapt the language grammar** to conform the requirements of a particular tool.

H>a The interface between automatically generated parser & other compiler components is typically **rather restricted**.

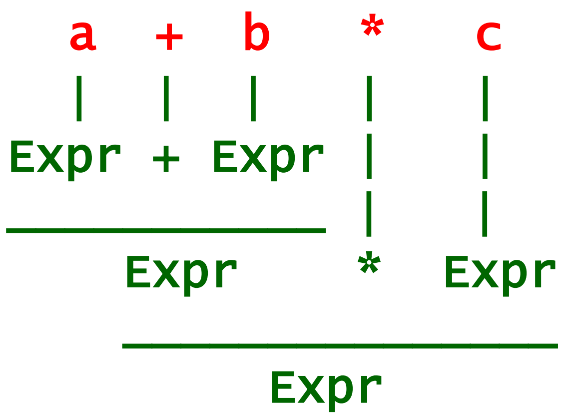
h<A Error recovery mechanism is easier to implement (at least for Yacc/Bison).

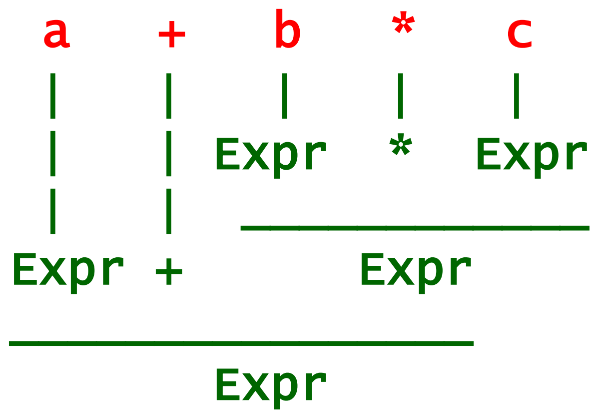
Syntax Analysis: a grammar for expressions

The first approach:

```
Expr -> Expr + Expr
Expr -> Expr – Expr
Expr -> Expr * Expr
Expr -> Expr / Expr
Expr -> Id
Expr -> ( Expr )
```

- The grammar correctly defines expression structure.
- **The problem**: using this grammar we can interpret an expression by **more than one way**.





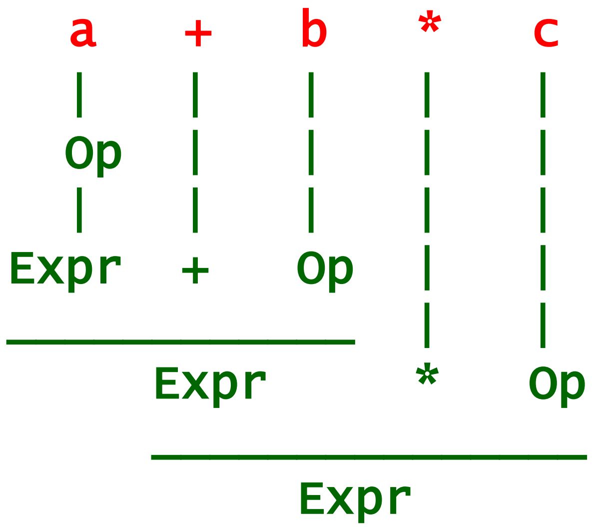
The grammar is ambiguous.

Syntax Analysis: a grammar for expressions

The second approach:

```
Expr -> Expr + Op
Expr -> Expr - Op
Expr -> Expr * Op
Expr -> Expr / Op
Op -> Id
Op -> ( Expr )
```

- The grammar correctly defines expression structure.
- The grammar is **unambiguous**.
- **The problem: operator preferences** are not taken into account.



Syntax Analysis: a grammar for expressions

The third approach:

```
Expr -> Term
Expr -> Expr + Term
Expr -> Expr - Term
Term -> Factor
Term -> Term * Factor
Term -> Term / Factor
Factor -> Id
Factor -> ( Expr )
```

- The grammar correctly defines expression structure.
- The grammar is **unambiguous**.
- **Operator preferences** are taken into account.

```
  a      +     b      *      c
  |      |     |      |      |
Factor   |   Factor   |      |
  |      |     |      |      |
Term     |   Term     *    Factor
  |      |   ___________________
  |      |            |
Expr     +          Term
_____________________________
            Expr
```

So far so good… ☺☹

Recursive Descent Parsing: Common Rules

- For each non-terminal (grammar production) a **corresponding parsing function** is declared.
- Each function sequentially **reads tokens** composing the given non-terminal, or reports an error.
- Each non-terminal in the right part of the rule is treated as **the call to the corresponding function**.

- Why parsing is "recursive"?- because any non-trivial grammar has rules with direct or indirect "self-declarations" which is actually a recursion.
- Why parsing is "descent"?- because the parsing process starts with the very common production down to more concrete ones.

Recursive Descent Parsing: History

To program recursive descent parser is as fast as just write ☺. (David Gries)

David Gries
Compiler Construction for Digital Computers
John Wiley and Sons, New York, **1971**, 281 pages.

Грис, Д.
Конструирование компиляторов
для цифровых вычислительных машин
Издательство: М.: Мир
544 страниц; **1975** г.

Написать рекурсивный нисходящий парсер можно, не отрывая пера от бумаги ☺. (David Gries)

Recursive Descent Parser: Example

```
Expr -> Term
Expr -> Expr + Term
Expr -> Expr – Term
```

```
Term -> Factor
Term -> Term * Factor
Term -> Term / Factor
```

```
Factor -> Id
Factor -> ( Expr )
```

```
void parseExpr()
{
    parseExpr(); // !!!!!!!
    if ( tk=get(), tk==tkPlus || tk==tkMinus )
    {
        parseTerm();
    }
}
void parseTerm()
{
    parseTerm(); // !!!!!!!
    if ( tk=get(), tk==tkStar || tk==tkSlash )
    {
        parseFactor();
    }
}
void parseFactor()
{
    if ( tk=get(), tk==tkLParen )
    {
        parseExpr();
        get(); // skip ‘)’
    }
    else
        parseId();
}
```

LEFT RECURSION!! ☹☹

Recursive Descent Parser: Example

A grammar with the left recursion always can be transformed to an equivalent grammar with the right recursion.

The fourth approach

```
Expr -> Term
Expr -> Expr + Term
Expr -> Expr – Term

Term -> Factor
Term -> Term * Factor
Term -> Term / Factor

Factor -> Id
Factor -> ( Expr )
```

```
Expr -> Term
Expr -> Term + Expr
Expr -> Term - Expr

Term -> Factor
Term -> Factor * Term
Term -> Factor / Term

Factor -> Id
Factor -> ( Expr )
```

Recursive descent parser: example

«Programming» solution:

- Use EBNF format for more clarity;
- Replacing recursion for iteration.

```
Expr -> Term
Expr -> Expr + Term
Expr -> Expr - Term

Term -> Factor
Term -> Term * Factor
Term -> Term / Factor

Factor -> Id
Factor -> ( Expr )
```

The fifth (final) approach:

```
Expr -> Term
          { +|- Term }


Term -> Factor
          { *|/ Factor }


Factor -> Id
Factor -> ( Expr )
```

Recursive descent parser: example

```
Expr ->
  Term { +|- Term }
```

```
Term ->
  Factor { *|/ Factor }
```

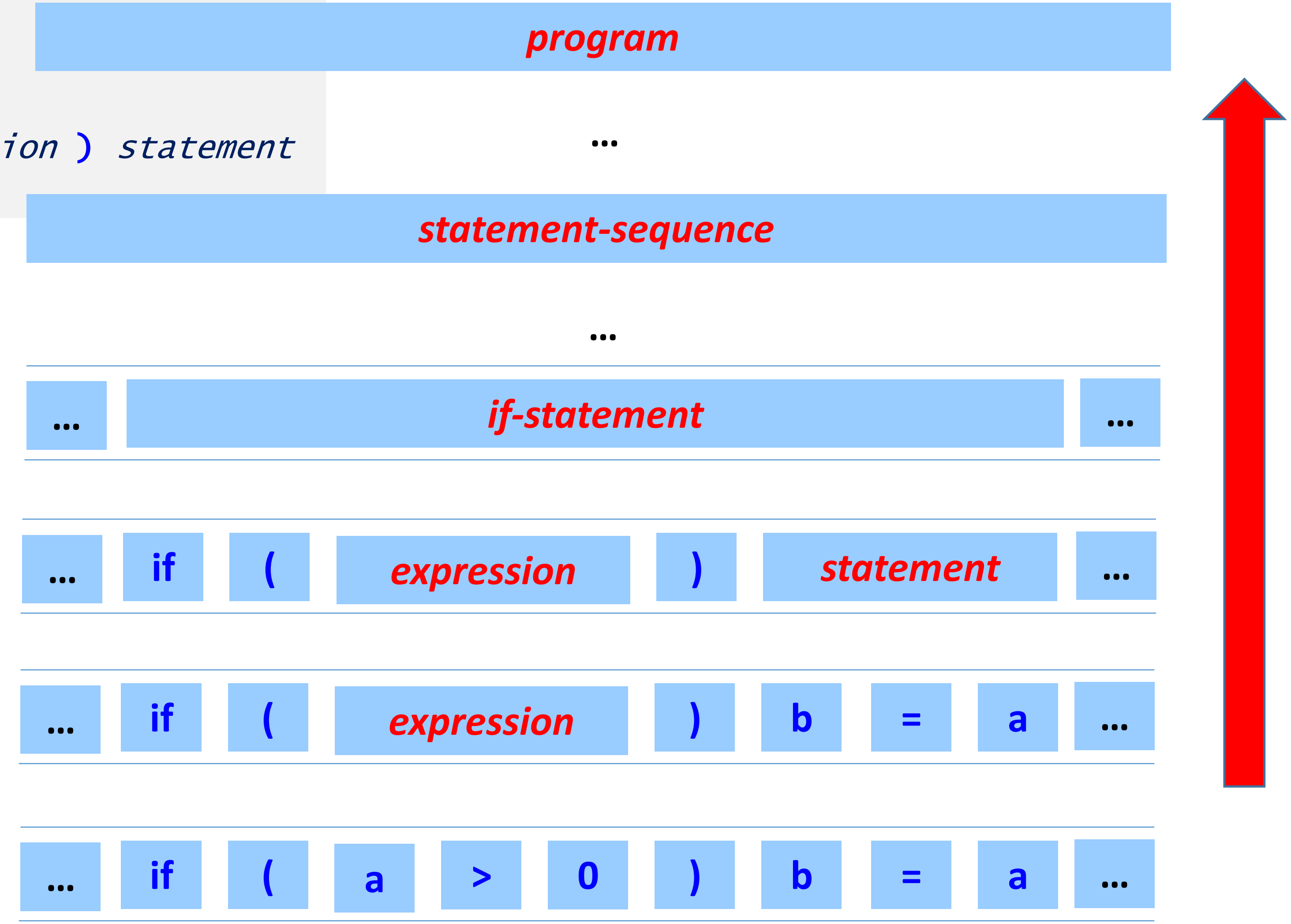
```
Factor -> Id
Factor -> ( Expr )
```

```
Tree parseExpr()
{
    Tree left = parseTerm();
    while ( tk=get(), tk==tkPlus||tk==tkMinus )
        left = mkBinTree(tk,left,parseTerm());
    return left;
}

Tree parseTerm()
{
    Tree left = parseFactor();
    while ( tk=get(), tk==tkStar||tk==tkSlash )
        left = mkBinTree(tk,left,parseFactor());
    return left;
}
Tree parseFactor()
{
    Tree res;
    if ( tk=get(), tk==tkLParen )
    {
        res = parseExpr();
        get(); // skip ')'
    }
    else
        res = mkUnaryTree(parseId());
    return res;
}
```

Bottom-up Parsing: The Idea

```
program -> statement-sequence
statement-sequence -> statement
statement-sequence -> { statement }
...
statement -> ...
statement- > if-statement
...
if-statement -> if ( expression ) statement
...
```



Bottom-up Parsing: an example of a table-driven parser

```
Factor -> Id
Factor -> ( Expr )
```

Finite state automate with the stack memory
Детерминированный конечный автомат с магазинной памятью

Input token

Next state

Read the next token?

State number

Factor

State	Input	Go to	Read next	Comment
f1	Id	0		Return
f2	∀ - (			Error
f3	(	call e1	+	Goto e1 with return
f3	)	0		Return
f4				Error

Bottom-up parsing: an example of a table-driven parser

```
Term ->
 Factor { *|/ Factor }
```

Term

State	Input	Go to	Read next	Comment
t1		call f1		Goto f1 with return
t2	*	call f1	+	Goto f1 with return
t3		t2		
t4	∀ - /			Error
t4	/	call f1	+	Goto f1 with return
t5		t2		

Bottom-up parsing: an example of a table-driven parser

Bottom-up parsing:

- Is controlled by the input token stream
- Uses its own stack memory for keeping return states
- Tables can be generated automatically (by a tool) from the grammar
- The single algorithm can work with tables for the whole grammar category.

Not shown:

- Error processing – some ideas in yacc/bison
- Semantic actions! – will see later for yacc/bison