Compiler Construction: Practical Introduction

Samsung Compiler Bootcamp

Samsung Research Russia
Moscow 2019

Lecture 4
Flex & YACC

- Backus-Naur Form
- Bottom-up translation
- Yacc & clones; references
- Yacc-based technology
- Toy language grammar
- Conflicts in grammars
- Backup: Flex & Bison cooperation

Backus-Naur Form

- It is a formal, mathematical way to specify context-free grammars (CFG).
- It is precise and unambiguous.
- Grammars of the most programming languages are described by BNF.
- Left parts of productions in CFG are a single non-terminals. All productions are context-independent replacements for an every non-terminal.
- John Backus presented his notation about Algol 58 (1959). Peter Naur developed more precise form of this notation for Algol 60 (1963)

BNF Notation

The grammar is composed by a series of rules ("productions").
Each rule looks like as follows:

NonTerminal ::= Sequence-of-terminals-and nonterminals

Example:

<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

"::=" (sometimes ":=" or "->" is used) means "**is defined as**"
"|" means "**or**"

- Symbols enclosed by *angle brackets* are **nonterminals** (grammar concepts)
- Symbols *without angle brackets* are **terminals** (elements from language alphabet)

Examples of BNF productions

```
<while loop> ::= while ( <condition> ) <statement>

<assignment statement> ::= <variable> = <expression>

<statement list> ::= <statement>
                   | <statement list> <statement>

<unsigned integer> ::= <digit>
                     | <unsigned integer> <digit>
```

EBNF Notation

EBNF (**Extended BNF**) was suggested by Niklaus Wirth in 1977.

EBNF uses the following conventions:

- Non-terminals begin with uppercase letters (discard '<>')
- Repetitions (zero or more) are enclosed by '{}'
- Optional parts (zero or one) are enclosed by '[]'
- Use '()' to group items together
- Terminals are enclosed in quotes ("")

BNF versus EBNF

```
<while loop> ::= while ( <condition> ) <statement>

<assignment statement> ::= <variable> = <expression>

<statement list> ::= <statement>
                   | <statement list> <statement>

<unsigned integer> ::= <digit>
                     | <unsigned integer> <digit>
```

```
While-loop ::= 'while' '(' condition ')' Statement

Assignment-statement ::= Variable '=' Expression

Statement-list ::= Statement { Statement }

Unsigned-integer ::= Digit { Digit }
```

Automatic parser generation

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

	By hand	By tools	
Top-down parsing	Most often: **Recursive descent parser**	ANTLR COCO etc.	**LL(1)**
Bottom-up parsing	Rarely (too complicated)	Yacc, Bison, Jay, GPPG, etc.	**LALR(1)**

- **L(eft to right input) L(eftmost derivation) with 1 parsed symbol.**
- **LA(look ahead) L(eft to right input) R(ightmost defivation) with 1 parsed symbol**

Yacc/Bison & clones

- **YACC** – Yet another compiler compiler 1970: based on C.
- **Bison** – Yacc version for GNU: based on C.
- **GPPG** – Gardens Point Parser Generator: Yacc version for C# and .NET.
- **Jay** – Yacc version for Java.
- …A lot of YACC clones for almost all popular languages.

All YACCs have **identical** parsing algorithm.

Yacc/Bison: references

YACC - Yet Another Compiler Compiler
http://yacc.solotony.com/yacc_rus/index.html
Перевод оригинальной статьи (так себе, но понятно)

Компилятор компиляторов Bison – первое знакомство
http://trpl.narod.ru/CC_Bison.htm

Bison - Генератор синтаксических анализаторов, совместимый с YACC
http://www.opennet.ru/docs/RUS/bison_yacc/bison_1.html
Перевод официального руководства GNU

Lex и YACC в примерах
http://rus-linux.net/lib.php?name=/MyLDP/algol/lex-yacc-howto.html

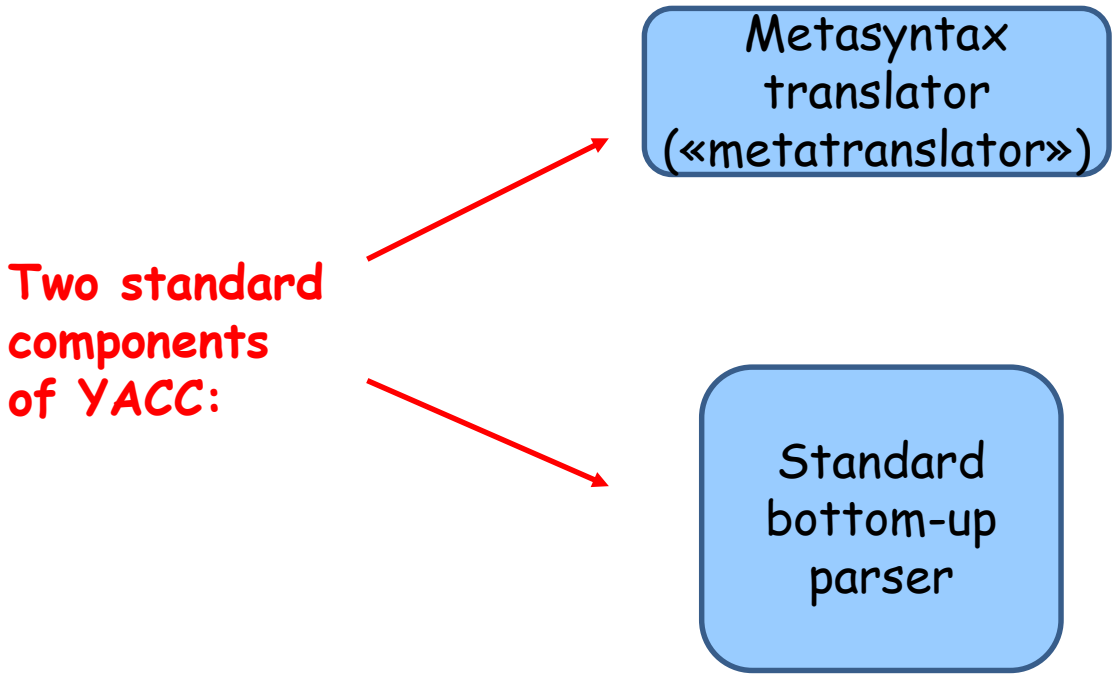
Gardens Point Parser Generator
YACC-совместимый генератор для C#; http://gppg.codeplex.com/

Yacc/Bison & clones: features

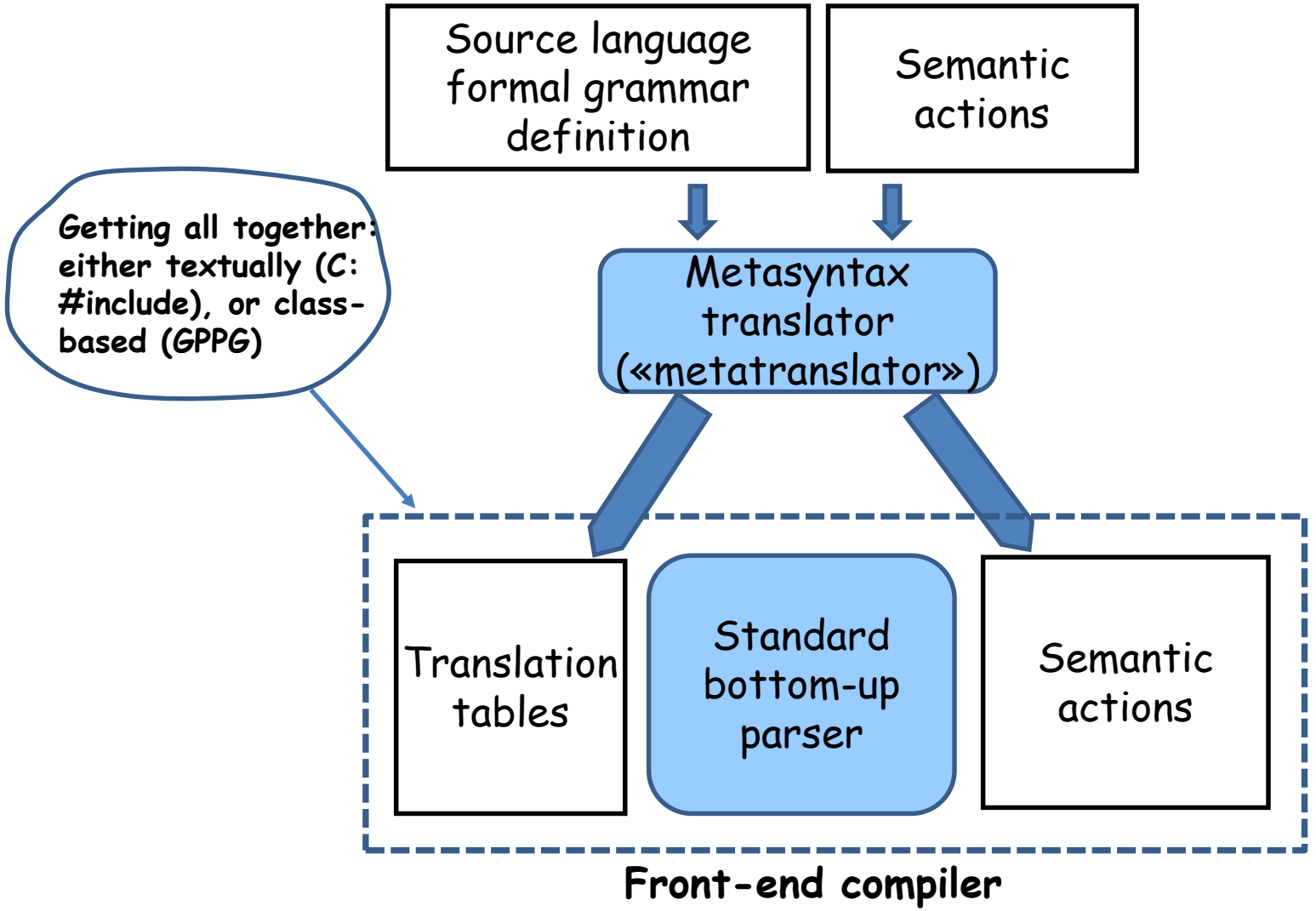
- Generates **bottom-up** syntax parsers.
- Has its **own notation** (formalism) for grammar specification.
- Internally, the grammar is represented in a **table form**; the generated parser is table-driven.
- Source tokens should be generated by a separate lexical analyzer: either by a hand made analyzer or by Lex/Flex or compatible (Yacc uses integer token codes).

- Very good grammar readability.
- Separation the grammar from semantic actions.
- Rules with left recursion are allowed.
- Good standard support for error recovery.
- Hard to debug the grammar and to find ambiguities.

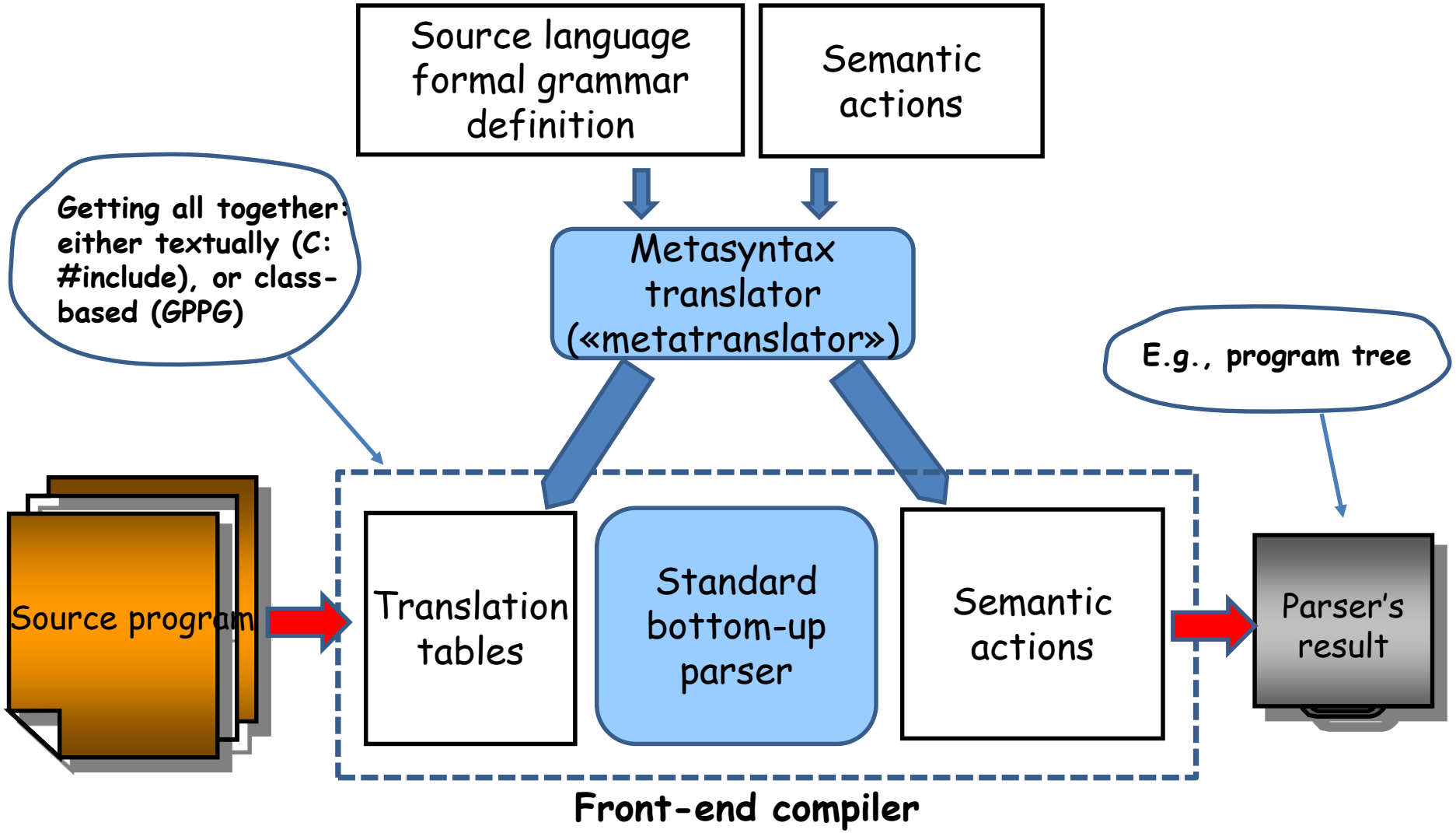
Yacc based technology



Yacc based technology



Yacc based technology



Yacc: Grammar structure

Common declarations (implementation language)
%%
Declarations of token, types, associativity,...
Declaration of the **main rule**
%%
Grammar rules (together with semantic actions)
%%
Common declarations (implementation language)

Yacc: Grammar Conventions

The grammar is composed by a series of rules ("productions").
Each rule looks like as follows:

```
NonTerminal : Sequence-of-terminals-and nonterminals
```

See examples on the following slides.

Conventions:

- "`:`" is used instead of "`::=`"
- "|" still means "**or**"
- Nonterminals are usual identifiers
- Terminals ("tokens") are sequences of characters enclosed by single quotes (`''`) or identifiers defined as "tokens"
- Metasymbols as `[]`, `()`, `{}` are not in use
  => No grouping
  => Use recursion instead of repetition

Example: Toy language grammar

```
// Identifiers & numbers
%token IDENTIFIER
%token NUMBER


// Keywords
%token IMPORT CLASS EXTENDS PRIVATE PUBLIC STATIC VOID IF ELSE
%token WHILE LOOP RETURN PRINT NULL NEW INT REAL


// Delimiters
%token LBRACE       //  {
%token RBRACE       //  }
%token LPAREN       //  (
%token RPAREN       //  )
%token LBRACKET     //  [
%token RBRACKET     //  ]
%token COMMA        //  ,
%token DOT          //  .
%token SEMICOLON    //  ;


// Operator signs
%token ASSIGN       //  =
%token LESS         //  <
%token GREATER      //  >
%token EQUAL        //  ==
%token NOT_EQUAL    //  !=
%token PLUS         //  +
%token MINUS        //  -
%token MULTIPLY     //  *
%token DIVIDE       //  /

%start CompilationUnit
```

Language alphabet

Grammar main rule

Gets converted to

```
enum Tokens
{
    IDENTIFIER,
    NUMBER,
    ...
};
```

Example: Toy language grammar

```
CompilationUnit
        : Imports ClassDeclarations
        ;
Imports
        :  /* empty */
        | Import Imports
        ;
Import
        : IMPORT IDENTIFIER SEMICOLON
        ;
ClassDeclarations
        : /* empty */
        | ClassDeclaration ClassDeclarations
        ;
ClassDeclaration
        :        CLASS IDENTIFIER SEMICOLON Extension ClassBody
        | PUBLIC CLASS IDENTIFIER SEMICOLON Extension ClassBody
        ;
Extension
        : /* empty */
        | EXTENDS Identifier
        ;
ClassBody
        : LBRACE              RBRACE
        | LBRACE ClassMembers RBRACE
        ;
 ClassMembers
        :              ClassMember
        | ClassMembers ClassMember
        ;
```

Grammar: program & classes

Example: Toy language grammar

```
ClassMember
        : FieldDeclaration
        | MethodDeclaration
        ;
FieldDeclaration
        : Visibility Staticness Type IDENTIFIER SEMICOLON
        ;
Visibility
        : /* empty */
        | PRIVATE
        | PUBLIC
        ;
Staticness
        : /* empty */
        | STATIC
        ;
MethodDeclaration
        : Visibility Staticness MethodType IDENTIFIER Parameters Body
        ;
Parameters
        : LPAREN                RPAREN
        | LPAREN ParameterList RPAREN
        ;
ParameterList
        :                       Parameter
        | ParameterList COMMA Parameter
        ;
Parameter
        : Type IDENTIFIER ;
```

Grammar: declarations

Example: Toy language grammar

```
MethodType
        : Type
        | VOID
        ;
Body
        : LBRACE LocalDeclarations Statements RBRACE
        ;
LocalDeclarations
        :                       LocalDeclaration
        | LocalDeclarations LocalDeclaration
        ;
LocalDeclaration
        : Type IDENTIFIER SEMICOLON
        ;
```

Example: Toy language grammar

```
Statements
        :             Statement
        | Statements Statement
        ;
Statement
        : Assignment | IfStatement | WhileStatement | ReturnStatement
        | CallStatement | PrintStatement | Block
        ;
Assignment
        : LeftPart ASSIGN Expression SEMICOLON
        ;
LeftPart
        : CompoundName
        | CompoundName LBRACKET Expression RBRACKET
        ;
CompoundName
        :                   IDENTIFIER
        | CompoundName DOT IDENTIFIER
        ;
IfStatement
        : IF LPAREN Relation RPAREN Statement
        | IF LPAREN Relation RPAREN Statement ELSE Statement
        ;
WhileStatement
        : WHILE Relation LOOP Statement SEMICOLON
        ;
ReturnStatement
        : RETURN             SEMICOLON
        | RETURN Expression SEMICOLON
        ;
```

Grammar: statements

Example: Toy language grammar

```
CallStatement
        : CompoundName LPAREN                RPAREN SEMICOLON
        | CompoundName LPAREN ArgumentList RPAREN SEMICOLON
        ;
ArgumentList
        :                      Expression
        | ArgumentList COMMA Expression
        ;
PrintStatement
        : PRINT Expression SEMICOLON
        ;
Block
        : LBRACE              RBRACE
        | LBRACE Statements RBRACE
        ;
```

Grammar: statements

Example: Toy language grammar

```
Relation
        : Expression
        | Expression RelationalOperator Expression
        ;
RelationalOperator
        : LESS | GREATER | EQUAL | NOT_EQUAL
        ;
Expression
        :          Term Terms
        | AddSign Term Terms
        ;
AddSign
        : PLUS | MINUS
        ;
Terms
        : /* empty */
        | AddSign Term Terms
        ;
Term
        : Factor Factors
        ;
Factors
        : /* empty */
        | MultSign Factor Factors
        ;
MultSign
        : MULTIPLY | DIVIDE
        ;
```

Grammar: expressions

Example: Toy language grammar

```
Factor
        : NUMBER
        | LeftPart
        | NULL
        | NEW NewType
        | NEW NewType LBRACKET Expression RBRACKET
        ;
NewType
        : INT
        | REAL
        | IDENTIFIER
        ;
Type
        : INT        ArrayTail
        | REAL       ArrayTail
        | IDENTIFIER ArrayTail
        ;
ArrayTail
        : /* empty */
        | LBRACKET RBRACKET
        ;
```

Grammar: types

Toy grammar: comments

1. No means for expression repetitions (like in BNF format) in YACC notation; we have to use recursion instead.

```
ParameterList
        :                          Parameter
        | ParameterList COMMA Parameter
        ;
...
ArgumentList
        :                        Expression
        | ArgumentList COMMA Expression
        ;
...
Statements
        :              Statement
        | Statements Statement
        ;
```

Toy grammar: comments

2. Both right and left recursions are allowed and supported.

```
Expression
        :           Term Terms
        | AddSign Term Terms
        ;
AddSign
        : PLUS | MINUS
        ;
Terms
        : /* empty */
        | AddSign Term Terms
        ;
Term
        : Factor Factors
        ;
Factors
        : /* empty */
        | MultSign Factor Factors
        ;
MultSign
        : MULTIPLY | DIVIDE
        ;
```

Toy grammar: comments

3. Grouping is not supported; we have to add extra rules for grouping

```
AddSign
        : PLUS | MINUS
        ;
Terms
        : /* empty */
        | AddSign Term Terms
        ;
Term
        : Factor Factors
        ;
Factors
        : /* empty */
        | MultSign Factor Factors
        ;
MultSign
        : MULTIPLY | DIVIDE
        ;
```


```
Terms
        : /* empty */
        | (PLUS|MINUS) Term Terms
        ;

Factors
        : /* empty */
        | (MULTIPLY | DIVIDE)
                  Factor Factors
        ;
```

Toy grammar translation

```
C:\Lectures\GPG 1.5.0\binaries>
                gppg /conflicts "C:\Lectures\Lecture 8\Toy.yacc

Shift/Reduce conflict
 Shift "IDENTIFIER":    State-20 -> State-21
 Reduce 30:     MethodType -> Type

Shift/Reduce conflict
 Shift "ELSE":  State-87 -> State-88
 Reduce 50:     IfStatement -> IF, LPAREN, Relation, RPAREN, Statement

Shift/Reduce conflict
 Shift "LBRACKET":      State-120 -> State-122
 Reduce 48:     CompoundName -> IDENTIFIER
```

Toy grammar translation

```
C:\Lectures\GPG 1.5.0\binaries>
              gppg /conflicts "C:\Lectures\Lecture 8\Toy.yacc

Shift/Reduce conflict
 Shift "IDENTIFIER":    State-20 -> State-21
 Reduce 30:     MethodType -> Type

FieldDeclaration: Visibility Staticness Type . IDENTIFIER SEMICOLON
MethodType: Type .
```


```
FieldDeclaration
       : Visibility Staticness Type IDENTIFIER SEMICOLON
       ;
...
MethodDeclaration
       : Visibility Staticness MethodType IDENTIFIER Parameters Body
       ;
...
Type
       : ...
       | IDENTIFIER ArrayTail
       ;
MethodType
       : Type
       | ...
       ;
```

Shift/Reduce conflicts are resolved in favor of Shift

Toy grammar translation

```
C:\Lectures\GPG 1.5.0\binaries>
                gppg /conflicts "C:\Lectures\Lecture 8\Toy.yacc


Shift/Reduce conflict
 Shift "ELSE":  State-87 -> State-88
 Reduce 50:     IfStatement -> IF, LPAREN, Relation, RPAREN, Statement

IfStatement: IF LPAREN Relation RPAREN Statement .
IfStatement: IF LPAREN Relation RPAREN Statement . ELSE Statement
```

2

```
IfStatement
        : IF LPAREN Relation RPAREN Statement
        | IF LPAREN Relation RPAREN Statement ELSE Statement
        ;
```

```
IfStatement
        : IF LPAREN Relation RPAREN Statement ElseTail
        ;
ElseTail
        : /* empty */
        | ELSE Statement
        ;
```

Toy grammar translation

```
Shift/Reduce conflict
 Shift "LBRACKET":      State-120 -> State-122
 Reduce 48:     CompoundName -> IDENTIFIER

CompoundName: IDENTIFIER .
Type: IDENTIFIER . ArrayTail
```

3

```
C [ 10 ] = 7 ;   // assignment
C [ ] a ;        // declaration
```

```
Assignment
        : LeftPart ASSIGN Expression SEMICOLON
        ;
LeftPart
        : CompoundName
        | CompoundName LBRACKET Expression RBRACKET
        ;
Body
        : LBRACE LocalDeclarations Statements RBRACE
        ;
LocalDeclaration
        : Type IDENTIFIER SEMICOLON
        ;
```

```
Type
        : ...
        | IDENTIFIER ArrayTail
        ;
ArrayTail
        : ...
        | LBRACKET RBRACKET
        ;
```

Toy grammar translation

Let's introduce an error to the grammar:

```
Body
        : LBRACE LocalDeclarations Statements RBRACE
        ;
...
Statement
        : LocalDeclaration
        | Assignment
        | IfStatement
        | WhileStatement
        | ReturnStatement
        | CallStatement
        | PrintStatement
        | Block
        ;
```

```
Reduce/Reduce conflict in state 131 on symbol INT
 Reduce 34:     LocalDeclarations -> LocalDeclarations, LocalDeclaration
 Reduce 38:     Statement -> LocalDeclaration
```

Reduce/Reduce should be resolved by developer (by transforming the grammar)

Toy grammar: semantic actions

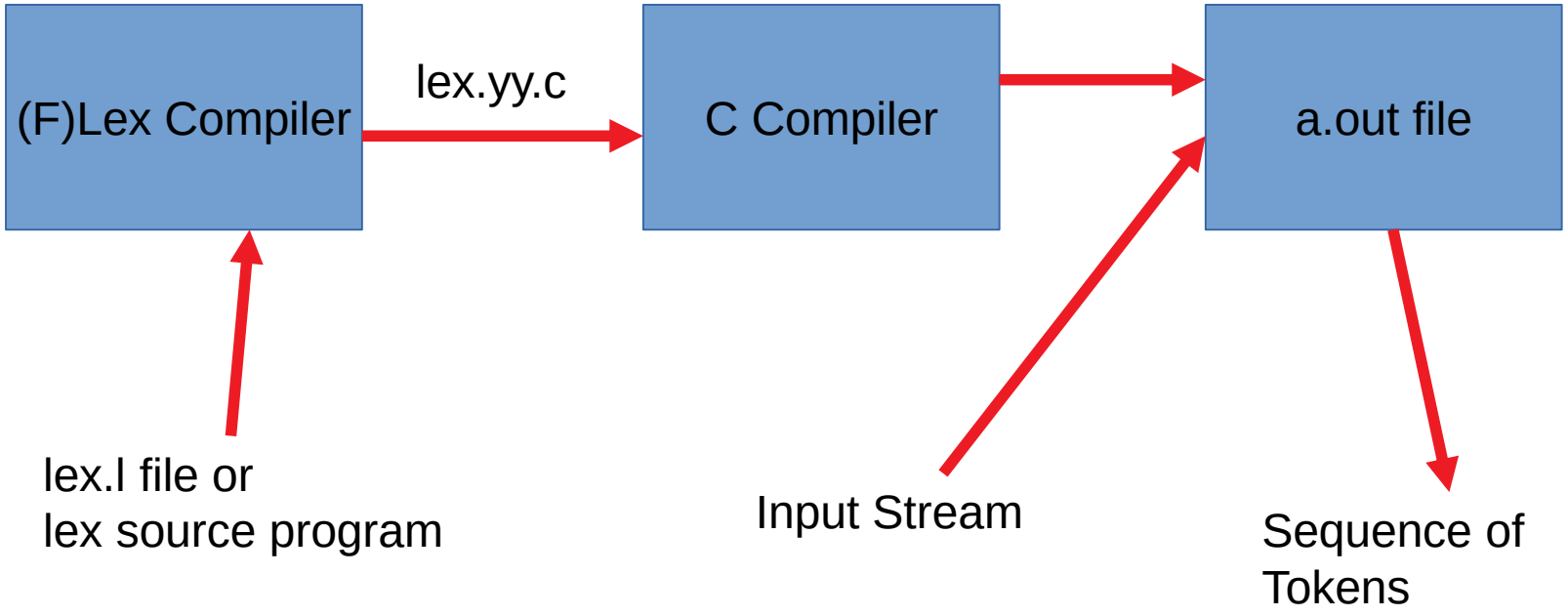
```
Statements
        :            Statement { $$ = createStmtList($1); }
        | Statements Statement { $$ = addStmtToList($1,$2); }
        ;
Statement
        : Assignment | IfStatement | WhileStatement | ReturnStatement
        | ...
        ;
Assignment
        : LeftPart ASSIGN Expression SEMICOLON  { $$ = createAssign($1,$3); }
        ;
IfStatement
        : IF LPAREN Relation RPAREN Statement
                                        { $$ = createIf($3,$5,NULL); }
        | IF LPAREN Relation RPAREN Statement ELSE Statement
                                        { $$ = createIf($3,$5,$7); }
        ;
WhileStatement
        : WHILE Relation LOOP Statement SEMICOLON { $$ = createWhile($2,$4); }
        ;
ReturnStatement
        : RETURN            SEMICOLON { $$ = createReturn(NULL); }
        | RETURN Expression SEMICOLON { $$ = createReturn($2); }
        ;
```

Backup
Flex & Bison

Flex – Fast Lexical Analyzer Generator

- FLEX is a tool for generating lexical analyzers (scanners or lexers). Is often used together with YACC or BISON.
- Written by Vern Paxson in C in 1987.
- The function **yylex()** is automatically generated by the flex when it is provided with a **.l file** and this **yylex()** function is expected by parser to call to retrieve tokens from a token stream.

Flex – Fast Lexical Analyzer Generator



Flex: Program Structure

Definition Section: declarations of vars,

```
%{
  // C-defintions
%}
```

Rules section:

```
%%
Pattern   Action
%%
```

User Code Section

Flex : Program Example

```
%option noyywrap
%{
int no_of_lines = 0;
Int no_of_chars = 0;
%}

%%
\n          { ++no_of_lines; }
.           { ++no_of_chars: }
end         { return; }
%%

/*User code */
int main()
{
   yylex();
   printf("lines = %d, chars = %d\n", no_of_lines, no_of_chars);
   return 0;
}
```

Flex: Pattern Examples

`[0-9]`	all the digits between *'0'* and *'9'*
`[0,9]`	either *'0'*, *','*, *'9'*
`[0-9]+`	one or more digit between *'0'* and *'9'*
`[^a]`	all the other characters except *'a'*
`[^A-Z]`	all the other characters except the upper case letters
`a{2,4}`	either *'aa'*, *'aaa'* or *'aaaa'*
`a{2,}`	two or more occurences of *'a'*
`a*`	0 or more occurences of *'a'*
`a+`	1 or more occurences of *'a'*
`w(x|y)z`	'wxz' or 'wyz'

Flex & Bison Cooperation

parser.y

```
%{
int yylex();
%}
…
%token SOME_KEYWORD
...
```

lex.l

```
%{
#include “parser.tab.h” /* import keywords */
…
%}
%%
some_input      { return SOME_KEYWORD; }
```

```
$ bison -d parser.y  # produces parser.tab.h with tokens for lexer.l
                     # and parser.tab.c
$ flex lex.l         # produces lex.yy.c
$ gcc lex.yy.c parser.tab.c
```

Flex & Bison Cooperation

parser.c

```
int yyparse()
{
  while ((token = yylex()) != 0) {
    switch (token) {
       case rule1: …
       case rule2: …
       ...
    }
    if (error)
       yyerror(msg);
  }
  return …;
}
```

main_source.c

```
int main()
{
  yyparse();
}

void yyerror(const char* msg)
{
}
```

lexer.c

```
int yylex()
{
   switch (input_type) {
   case 1: return token1;
   case 2: return token2;
   ...
   }
}
```