

# Compiler Construction: Practical Introduction

## Lecture 5 Automatic Parser Generators: Yacc/Bison

Eugene Zouev  
Fall Semester 2024  
Innopolis University

# Automatic parser generation

## *YACC/Bison*

# Automatic parser generation

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

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

# Yacc/Bison & clones

- **YACC** - Yet another compiler compiler 1970: based on C.
- **Bison** - Yacc version for GNU: based on C. Now, C++ and Java are also supported.
- **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 including ML.

All YACCs have **identical** parsing algorithm.

# Yacc/Bison: references (Russian)

**YACC - Yet Another Compiler Compiler**

[http://yacc.solotony.com/yacc\\_rus/index.html](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](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](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**

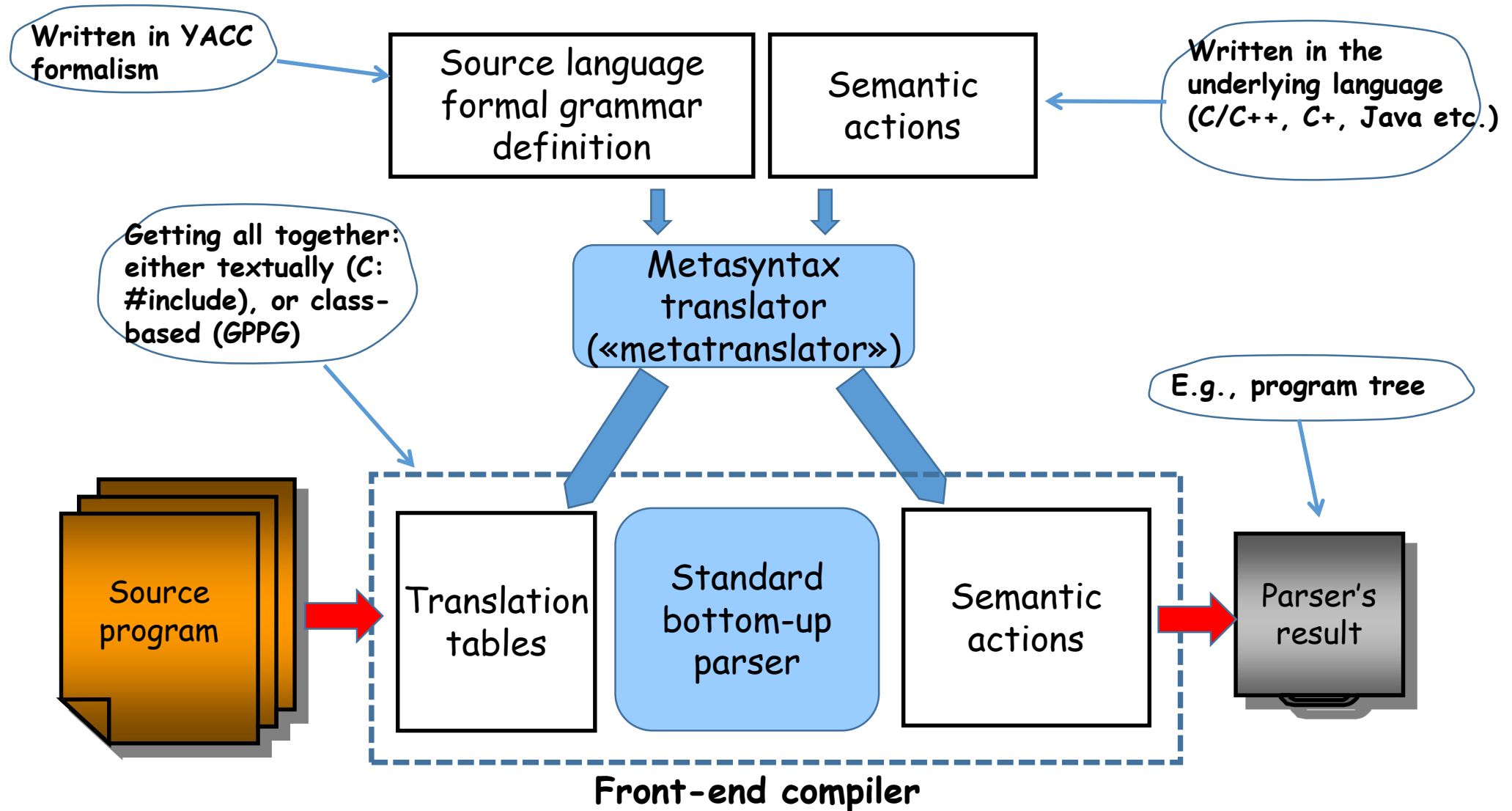
YACC-compatible parser generator for C#;

<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: The 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)



# 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
```

Tokens & Initial  
production

Language  
alphabet


Grammar  
main rule



# Lexics & Syntax

Token declarations get converted by YACC to the enum-declaration

```
...
%token LBRACE      // {
%token RBRACE      // }
%token LPAREN       // (
%token RPAREN       // )
%token LBRACKET     // [
%token RBRACKET     // ]
%token COMMA        // ,
...
```




```
enum Tokens
{
    ...
    LBRACE,
    RBRACE,
    LPAREN,
    ...
};
```

How to connect parser generated by YACC with (an external) scanner?

Standard bottom-up parser has the (preliminary) function declaration:

```
int yylex();
```

In “common declarations” section you should provide (your own) implementation of it:



```
int yylex() { ... }
```

# Toy Language Grammar

```
CompilationUnit
: /* empty */
|      ClassDeclarations
| Imports ClassDeclarations
| Imports
;
Imports
: Import Imports
;
ClassDeclarations
: ClassDeclaration ClassDeclarations
```

```
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
;
```

Right recursion  
OK!

**Grammar:  
program &  
classes**

Left recursion  
OK!

# 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**

# Toy Language Grammar

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

**Grammar:  
declarations**

# Toy Language Grammar

```
Statements
:
| 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**

# 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**

# 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**



# Toy Language Grammar

## Grammar: types

```
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
;
```

# Toy Language Grammar

## Grammar: types

```
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
;
```

## Alternative rules for Type

```
Type
: BasicType
| BasicType ArrayTail
;

BasicType
: INT
| REAL
| IDENTIFIER
;

ArrayTail
: LBRACKET RBRACKET
;
```

# Toy Grammar: Comments

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

Recursion is just for representing lists/sequences

Parameter , Parameter , ..., Parameter  
Expression , Expression , ..., Expression  
Statement ... Statement

```
ParameterList
:
| ParameterList COMMA Parameter
;

...
ArgumentList
:
| ArgumentList COMMA Expression
;

...
Statements
:
| Statements Statement
;
```

# Toy grammar: comments

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

```
Expression
:      Term Terms
|      AddSign Term Terms
;

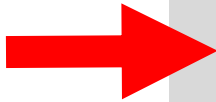
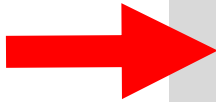
AddSign
:      PLUS | MINUS
;

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
;
```



### EBNF

```
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
```

1

## Shift/Reduce conflict

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

2

## Shift/Reduce conflict

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

3

# 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

1

FieldDeclaration: Visibility Staticness Type . IDENTIFIER SEMICOLON

MethodType: Type .

public static T m ...

What is *T*: a Type  
or a MethodType?

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

Shift/Reduce conflicts are resolved  
by default 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

2

IfStatement: IF LPAREN Relation RPAREN Statement .

IfStatement: IF LPAREN Relation RPAREN Statement . ELSE Statement

This is the if statement (of course)...

*if ( relation ) stmt else stmt*

...But this is the if statement as well! ☺

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

3

CompoundName: IDENTIFIER .

Type: IDENTIFIER . ArrayTail

The key problem here is **what does identifier mean** - either an existing type OR a new name of a declaration!

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

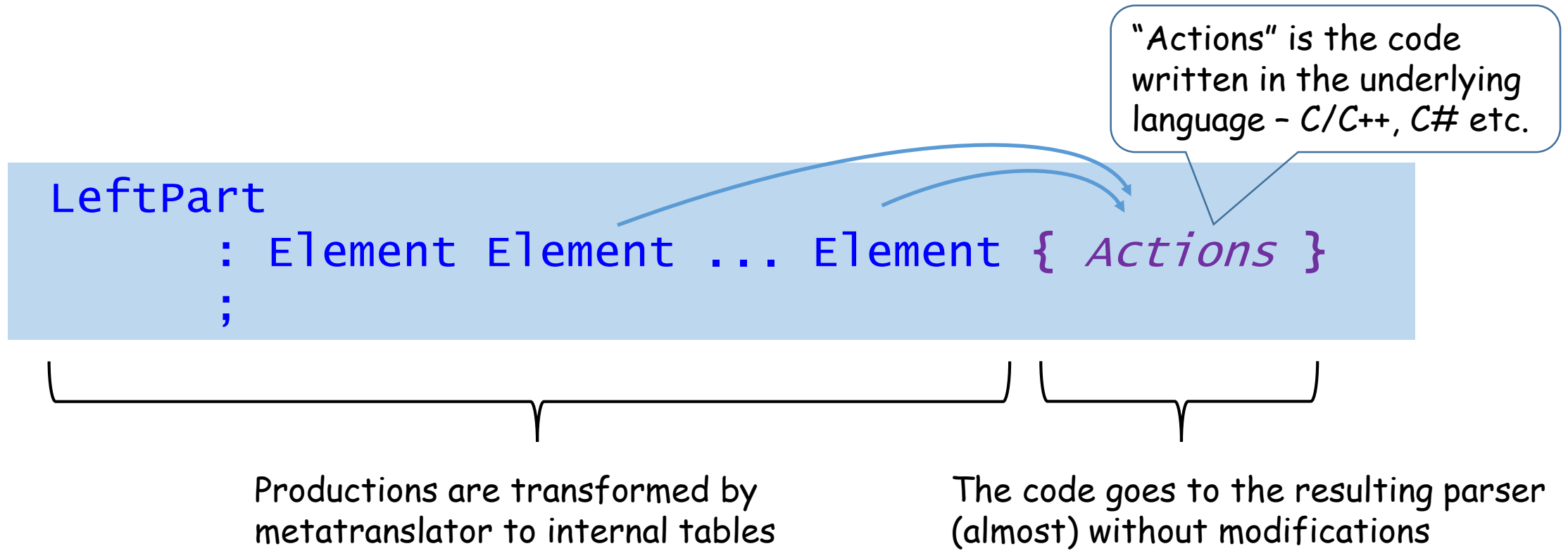
Reduce/Reduce conflict in state 131 on symbol INT

Reduce 26: LocalDeclarations -> LocalDeclarations, LocalDeclaration

Reduce 38: Statement -> LocalDeclaration

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

# Toy Grammar: Semantic Actions



The question: how "actions" gets information from the right part of the production?

# Toy Grammar: Semantic Actions

IfStatement

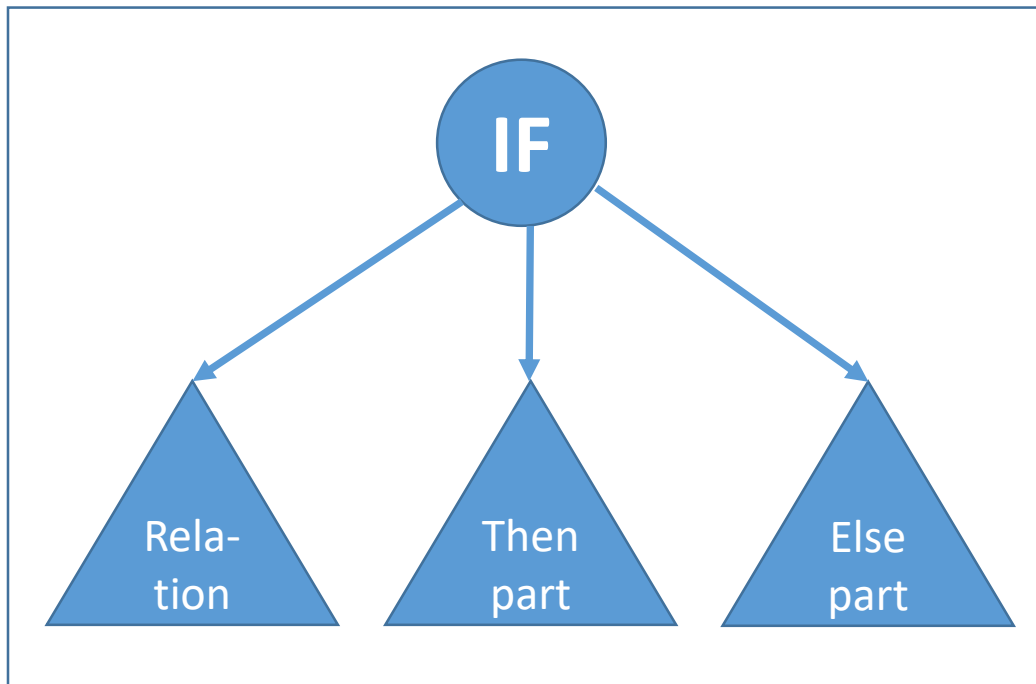
```
: IF LPAREN Relation RPAREN Statement ElseTail { ... }
```

```
; ① ② ③ ④ ⑤ ⑥
```

ElseTail

```
: /* empty */ { $$ = null; }  
| ELSE Statement { $$ = $2; }  
;
```

Our aim is to build a sub-tree out of **if** elements



```
{ $$ = createIfTree($3,$5,$6); }
```

The result of the whole production

Results of the corresponding non-terminals

# 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); }
;
```