

Faculté des Sciences,
de la Technologie
et de la Communication

Coala2Java

**A Front-End Compiler to translate
Coala programs to Java customized
for the CAA-DRIP framework**

Federico Wiecko

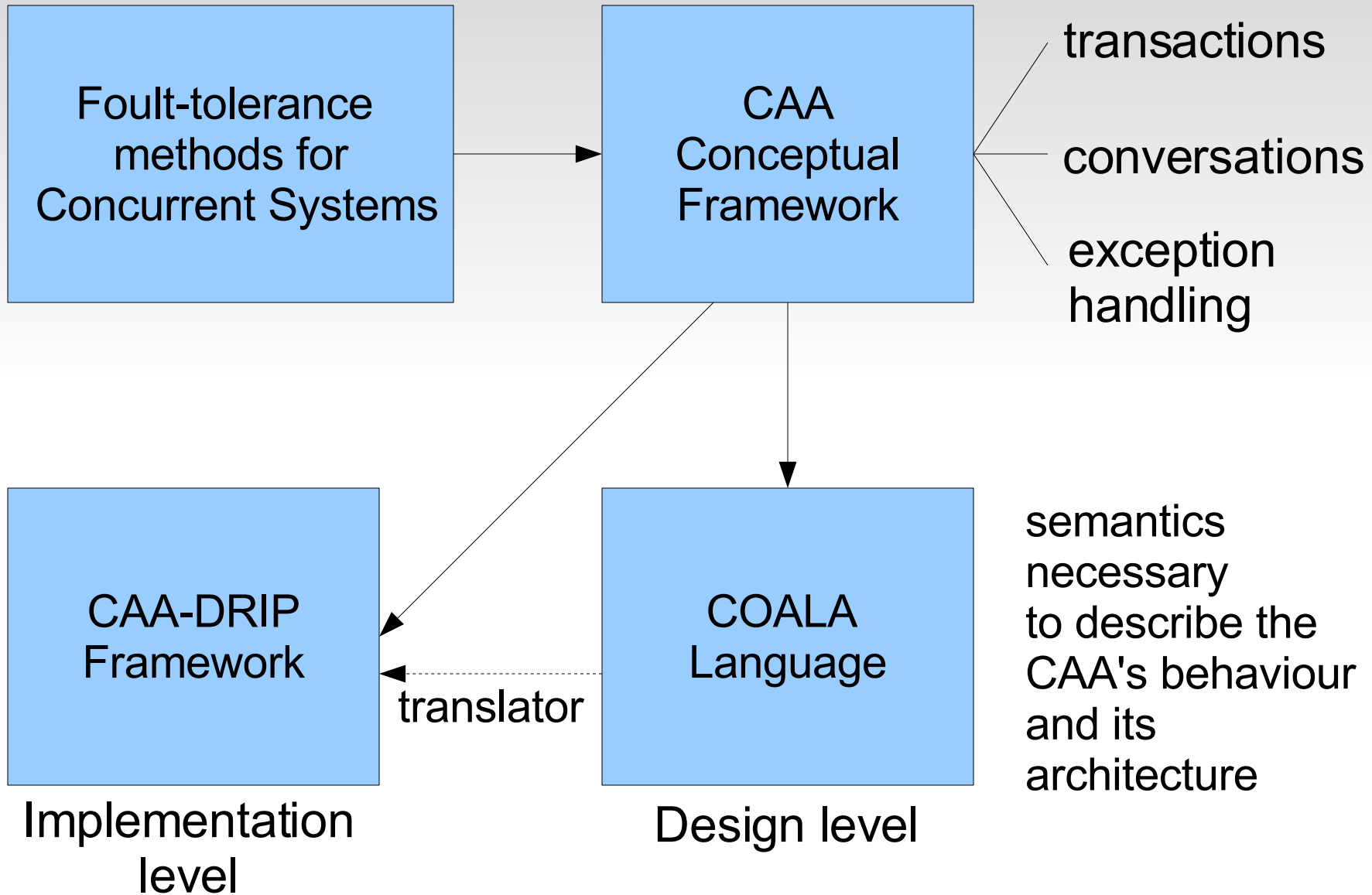
University of Luxembourg
federico.wiecko@uni.lu



Table of Contents

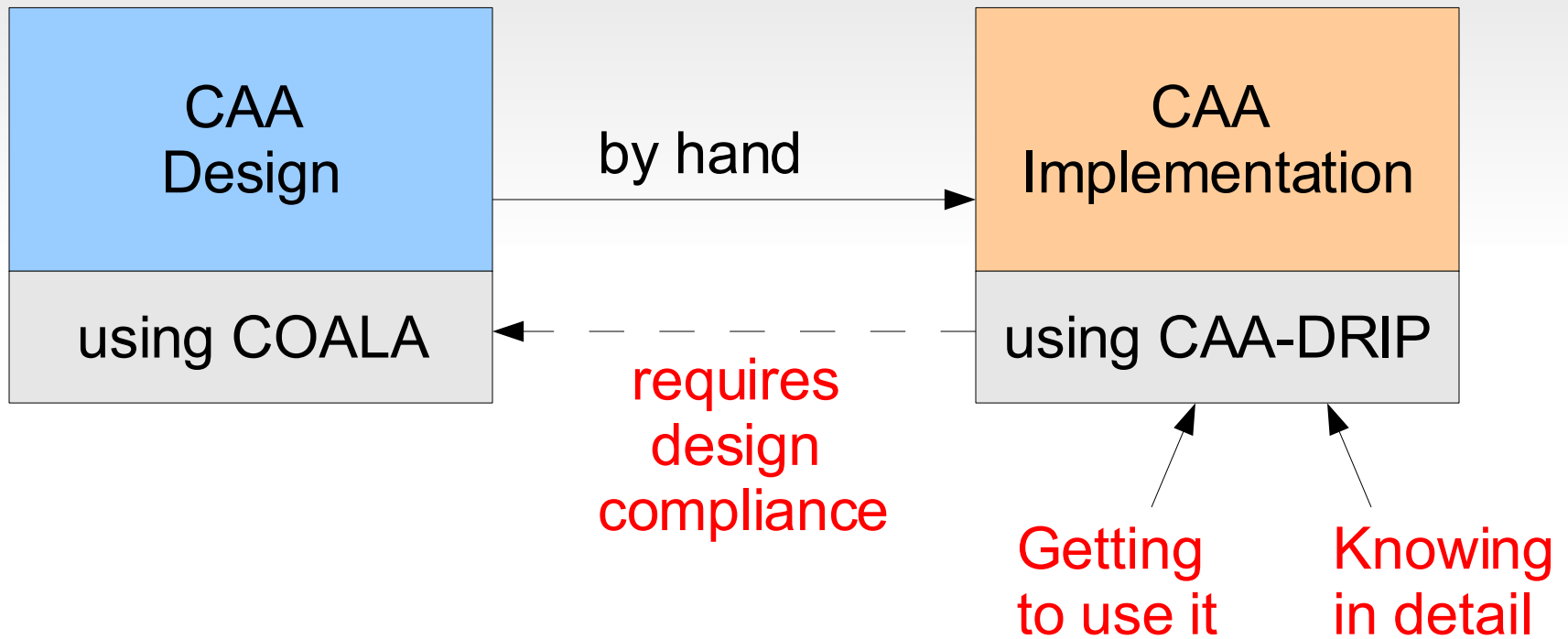
- Motivation
 - What, How and Why
- Background
 - Coordinated Atomic Action (CAA)
 - The COALA Language
 - The CAA-DRIP framework
- The compiler
 - A typical compiler
 - Evaluation of different tools
 - The SableCC framework
- Problems found
 - Ambiguous grammars, Shift/Reduce conflicts
 - CST – AST transformations
- Future work

Motivation

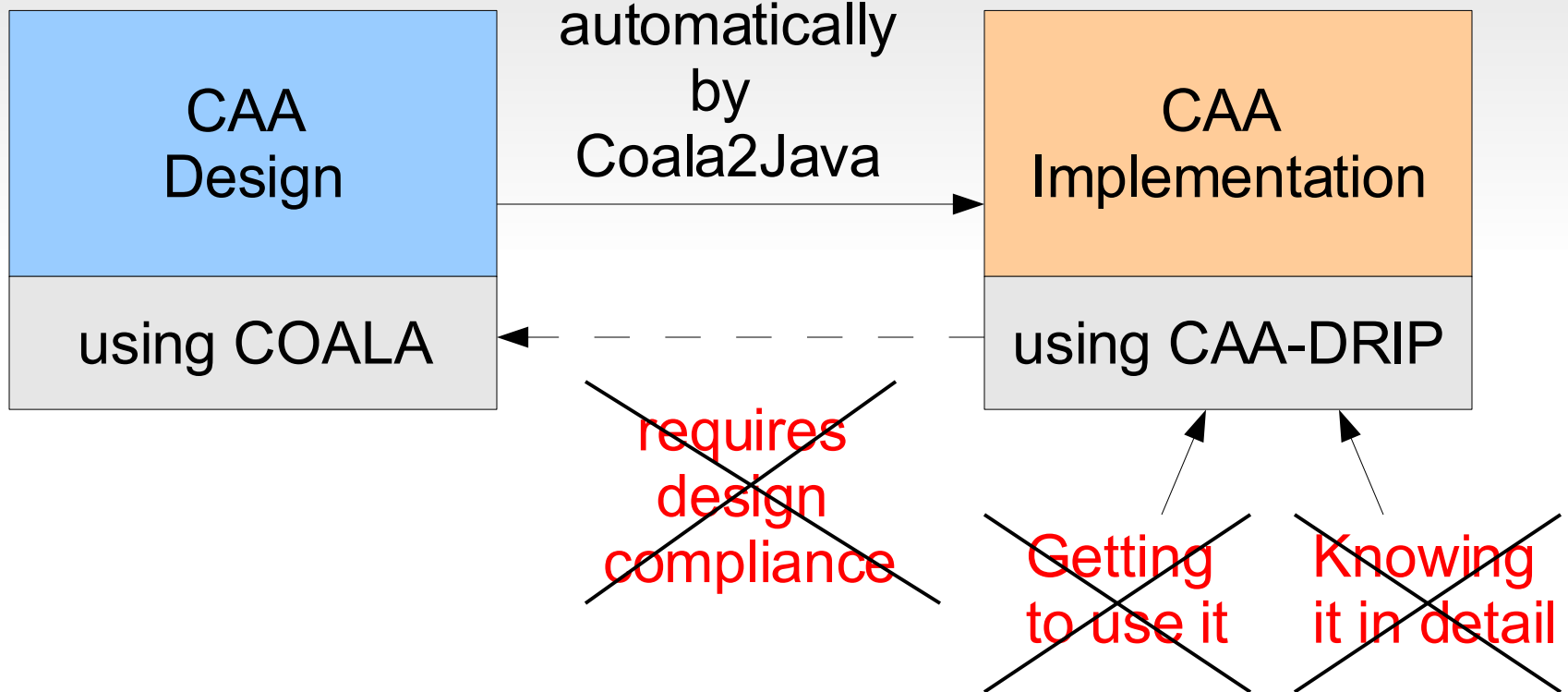


Motivation

TODAY



SOON (ongoing work)



How to achieve the goal?

The answer is

to build a **specific-purpose compiler**

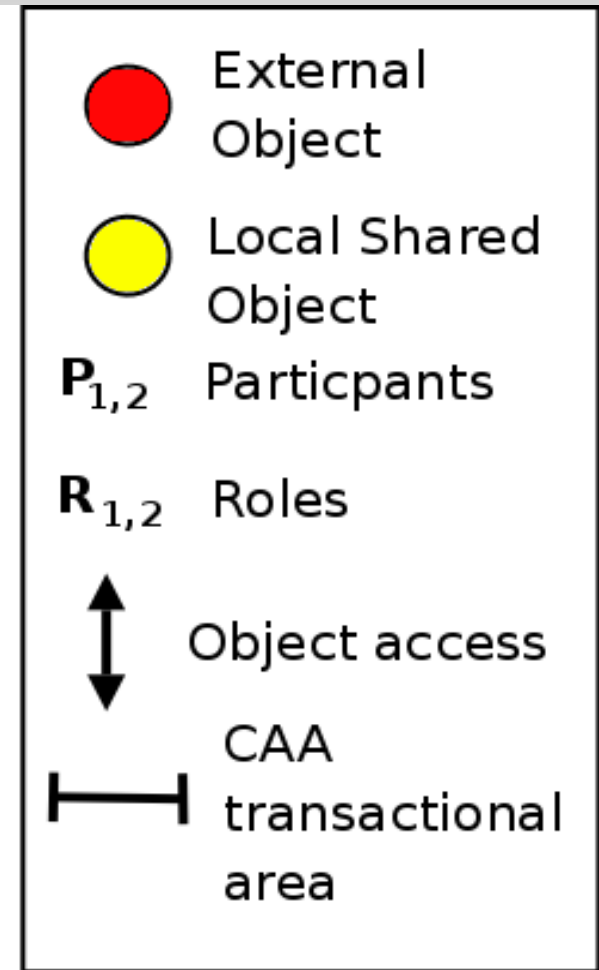
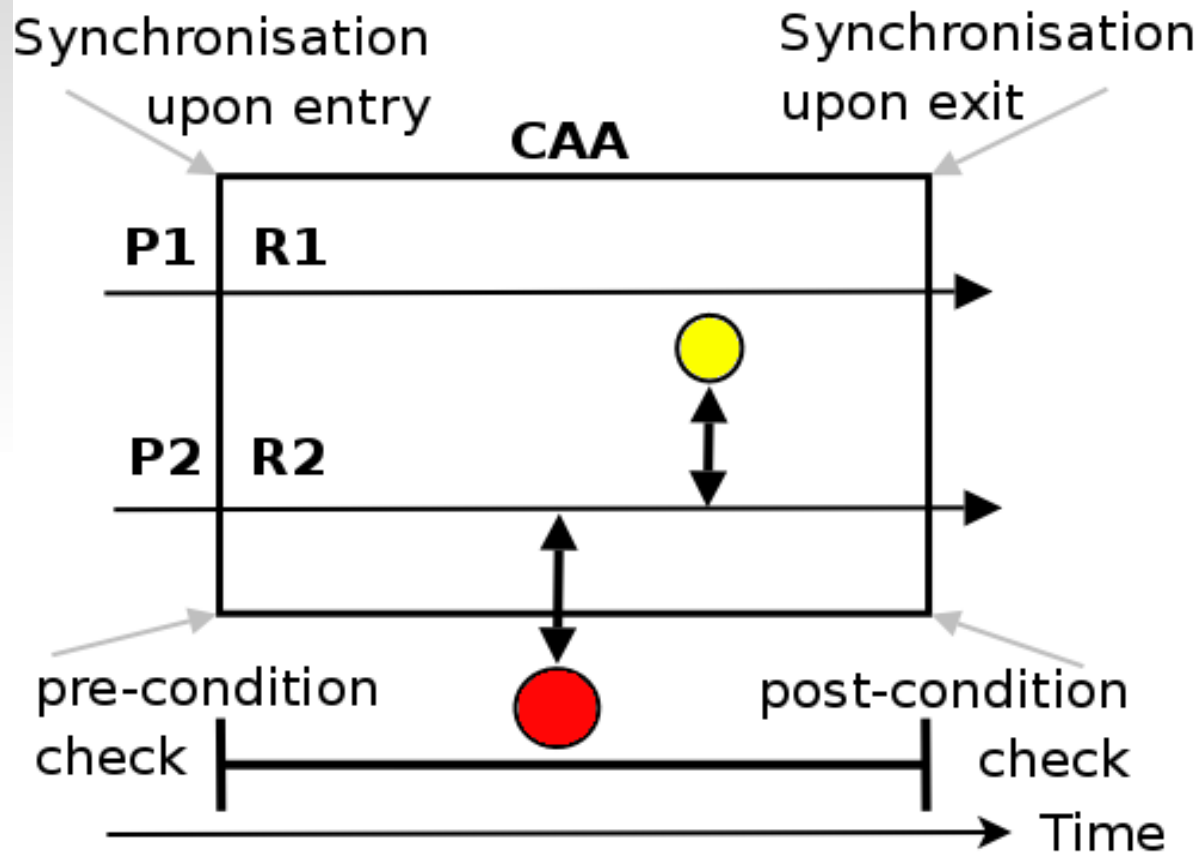
Why a compiler?

- The techniques are widely known.
- Each step in the design is absolutely determined
- An automatic way to achieve the semantic control can be added

Table of Contents

- Motivation
 - What, How and Why
- Background
 - Coordinated Atomic Action (CAA)
 - The COALA Language
 - The CAA-DRIP framework
- The compiler
 - A typical compiler
 - Evaluation of different tools
 - The SableCC framework
- Problems found
 - Ambiguous grammars, Shift/Reduce conflicts
 - CST – AST transformations
- Future work

Coordinated Atomic Action (CAA)



The possible outcomes of CAA are:

- ▶ with a **normal outcome** – no error has been detected
- ▶ **after successful recovery**
 - with a normal outcome
 - with a degraded exceptional outcome
 - with an abort outcome
- ▶ **with a failure outcome** – when recovery fails.

The COALA Language

- ▶ Provides the semantics necessary to describe the CAA's behaviour and its structure
- ▶ Very well-suited language for the design of reliable distributed systems
- ▶ It provides a formalization of the CA action concept.

A simple example of a COALA program: (part1)

```
;; *****
Caa SimpleTest;
;; *****
```

Interface

Roles

```
Role1;
Role2;
```

Body

```
Exception e1,e2:String;
Resolution e1,e2 -> e1
           e1 -> e1
```

```
Role Role1;
Begin
```

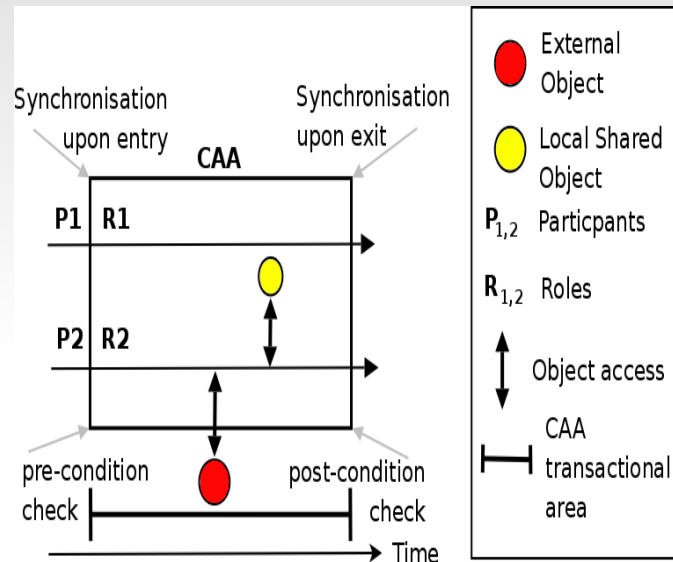
```
;; role actions
```

```
End
```

```
Handler H1
Begin
```

```
;; handler1 actions
```

```
End
```



A simple example of a COALA program: (part2)

```
    Handling
        e1 -> H1;
        e2 -> H1 ;; ups, this exception will not
                    ;; be handled, the semantic
End Role1;                    ;; checking should detect it

Role Role2;
    Begin
        ;; role2 actions
    End
    Handler H2
        Begin
            ;; handler2 actions
        End

        Handling
            e1-> H2;

    End Role2;

End SimpleTest;
```

The COALA's grammar:

The Coala's grammar defines (among other things)

- The Lexical Tokens
 - ▶ a set of reserved word (i.e Begin, Caa, Raise, etc)
 - ▶ a set of reserved symbols (i.e ';' , '_' , '=' , etc).
- The Grammar Rules (or Syntactic Rules)

program \longrightarrow ▶ caaModule+

caaModule \longrightarrow ▶ **Caa** name ';' [caaInterface] [caaBody]
End name ';'

caaInterface \longrightarrow ▶ **Interface** [useField] [roleField]
[exceptionField]

Note: The EBNF-like grammar of COALA is provided in [1]

Semantic Rules (part1)

- 1) Every CAA included by using the “**Use Caa**” clause must be defined
- 2) Every CAA that is used to create a **new instance of a nested CAA** (e.g Caa_1["newInstanceName"]) must be included in the “*Use Caa*” clause in the CAA's body.
- 3) Every exception that appear in the **left part of the arrow** in the “**Resolution**” field must be previously declared in
 - the “*Exceptions*” clause in the CAA's Body, or
 - in the interface of one of the CAAs included in as NestedCAA, or
 - is one of the predefined default exception (fail or abort).

Besides, the list of parameters of the exception must coincide with the declaration of the exception in the “*Exception*” field.

Semantic Rules (part2)

4) Every exception that appear in the **right part of the arrow** in the “**Resolution**” field must be previously declared in the “*Exceptions*” clause in the Caa's Body

Besides, the list of parameters of the exception must coincide with the declaration of the exception in the “*Exception*” field.

5) Every **variable** that is used inside of an instruction block (e.g MethodCall, Assign) that belongs to a role or handler, must be declared in one of the following places:

- into the “**Objects**” clause, its scope is limited to the CAA's body
- by using the “**Where**” clause, its scope is limited to the grammar rule associated with the where (i.e Role, Handler, Handling, Resolution)
- into the **heading** of the definition of a **role/handler**, its scope is restricted to that role or handler.

Semantic Rules (part3)

6) The **list of parameters of a role/handler** must coincide with their respective declaration. For a role that is given in the interface of the CAA by using the “*Roles*” clause and for a handler that is provided inside of a role definition by using the “*Handler*” clause.

7) For every “**Call**” clause (i.e. Caa roleName(rpm1, ..., prm_n) Of caaName["caainstanceName"]) the following rules apply:

- the instance name(i.e "caainstanceName") must be previously declared inside the “*NestedCaa*” clause
- the role (i.e roleName) must be declared inside the CAA included in the “*NestedCaa*” clause
- the list of parameters for that role must coincide with their respective declaration inside the CAA included in the “*NestedCaa*” clause

Semantic Rules (part4)

8) Every **exception** that is used in a “**Signal**” clause must be previously declared in the CAA's interface or is the predefined abort exception. Besides, the list of parameters of the exception declared inside the “*Resolution*” field must coincide in number and type with the list provided in the “*Exception*” clause.

9) Every **exception** that is used in a “**Raise**” clause must be previously declared in the CAA's interface or is the predefined abort exception. Besides, the list of parameters of the exception declared inside the “*Resolution*” field must coincide in number and type with the list provided in the “*Exception*” clause.

10) Every **exception** that is used in a “**Catch**” clause must be previously declared in the “*Exception*” clause inside the CAA's body or in the interface of one of the CAA included in the “*NestedCaa*” clause or is one of the predefined exceptions (i.e abort or fail).

Semantic Rules (part5)

11) For every **exception** that appear in the right part of the arrow in the “*Resolution*” clause must exist a unique handler in the right part of the arrow in the “*Handling*” clause

12) The **list of parameters of a handler** that is used in the right part of the arrow in the “*Handling*” clause must coincide with its declaration in the “*Handler*” clause

CAA-DRIP: a simple example

```
//managers  
Manager mgrRole1;  
Manager mgrRole2;
```

```
//roles  
Role role1;  
Role role2;
```

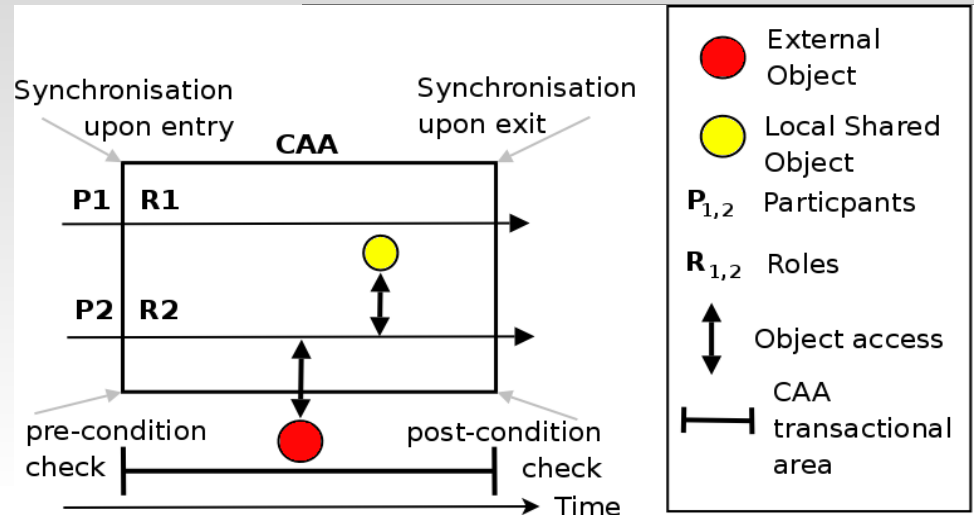
```
//handlers  
Handler hnd1;  
Handler hnd2;
```

```
//participants  
Thread t1;  
Thread t2;
```

```
//*****
```

```
mgrRole1 = new ManagerImpl("mgrRole1", "CAA SimpleTest");  
role1 = new fw.caa.simpleTest.Role1("role1", mgrRole1,  
mgrRole1);
```

```
mgrRole2 = new ManagerImpl("mgrRole2", "CAA SimpleTest");  
role2 = new fw.caa.simpleTest.Role2("role2", mgrRole2,  
mgrRole1);
```



CAA-DRIP: a simple example (cont)

```
/**
//*****
//Handlers
hnd1 = new fw.caa.simpleTest.handlers.Handler1("hnd1",
mgrRole1);

hnd2 = new fw.caa.simpleTest.handlers.Handler2("hnd2",
mgrRole2);

Hashtable ehRole1 = new Hashtable();
Hashtable ehRole2 = new Hashtable();

ehRole1.put(fw.exceptions.E1.class, hnd1);
ehRole2.put(fw.exceptions.E1.class, hnd2);

/* This code would allow E2 to be handled
hRole1.put(fw.exceptions.E2.class, hnd1);
hRole2.put(fw.exceptions.E2.class, hnd2);
*/

mgrRole1.setExceptionAndHandlerList(ehRole1);
mgrRole2.setExceptionAndHandlerList(ehRole2);

//*****

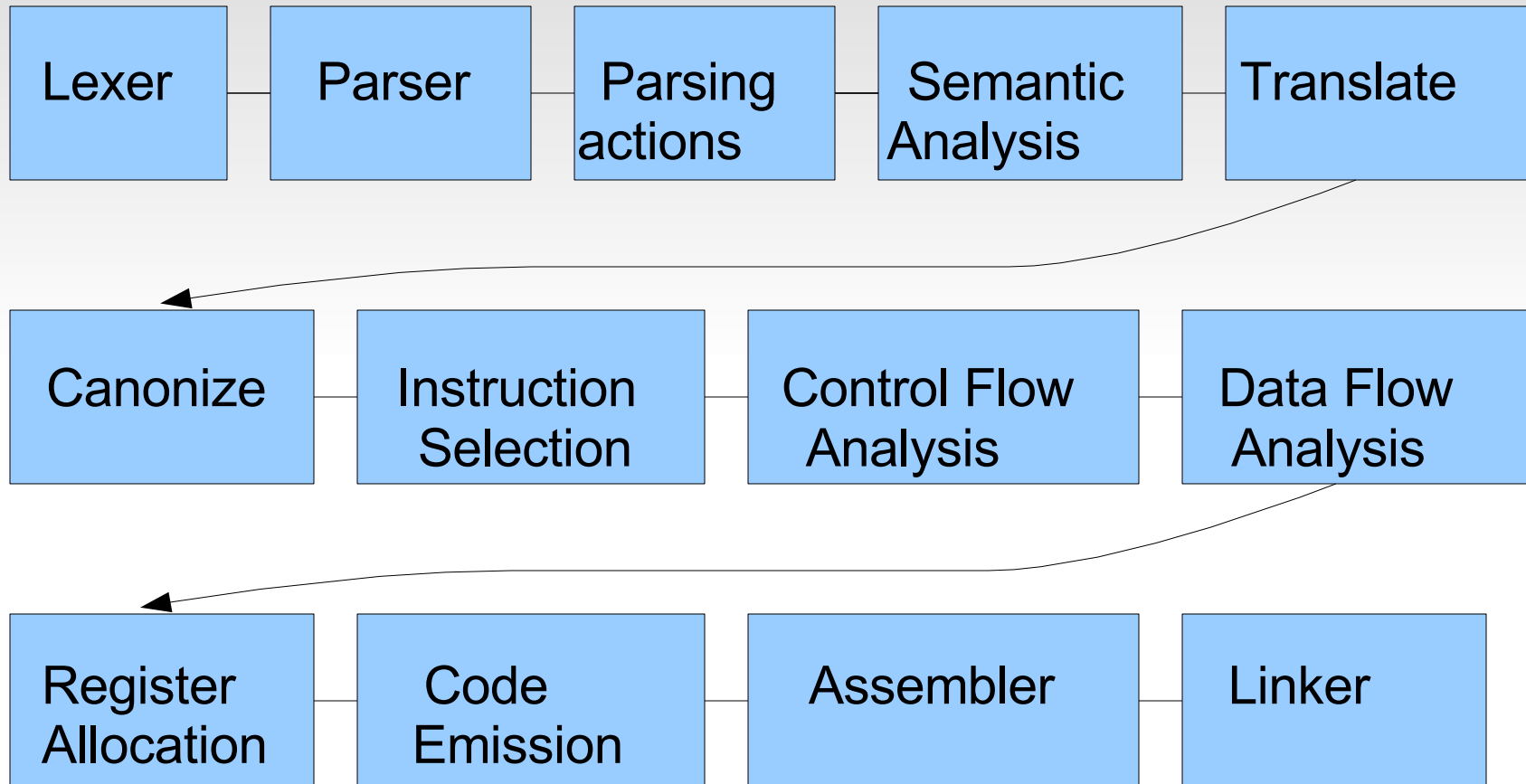
```

Table of Contents

- Motivation
 - What, How and Why
- Background
 - Coordinated Atomic Action (CAA)
 - The COALA Language
 - The CAA-DRIP framework
- The compiler
 - A typical compiler
 - Evaluation of different tools
 - The SableCC framework
- Problems found
 - Ambiguous grammars, Shift/Reduce conflicts
 - CST – AST transformations
- Future work

A typical compiler

The phases are



Steps involved to build our compiler

- **Lex:** Break the source file into individual words or tokens.
- **Parse:** Analyze the phrase structure of the program
- **Parsing Actions:** Build a piece of Abstract Syntax Tree (AST)
- **Semantic Analysis:** Determine what each phrase means, relate uses of variables to their definitions, check types of expressions, request translation of each phrase.
- **Frame Layout:** Place variables, function parameters, etc. into activation records (stack frames) in a machine-dependent way.
- **Translate:** Produce intermediate representation trees, a notation that is not tied to any particular source language or target machine architecture

Note: More information about compilers are provided in [2] and [3]

Evaluation of different tools

Parser Generators

- **JLEX/CUP**
- **PCCTS/JavaCC**
- **SableCC**

Evaluation criteria:

- ◆ **AST generated automatically**
- ◆ **Support for tree walkers**
- ◆ **EBNF support**
- ◆ **Easy to use (documentation)**
- ◆ **Easy to maintain**

Parser Generator

Properties

	JLex/CUP	PCCTS/ JavaCC	SableCC
<i>Sort of Parser</i>	LARL(1)	LL(k)	LARL(1)
Macros Support	yes		yes
Lexer States	yes		yes
EBNF	no	yes	yes
Lexer and Parser tightly integrated	no	yes	yes
Unicode Charset Support	no	JavaCC only	yes
Automatic AST Generation	no	yes	yes
Support for tree walkers	no	yes	yes

The SableCC Framework

Using SableCC requires the following steps

- ▶ Creating a SableCC specification file containing the lexical definitions and the grammar of the language to be compiled
- ▶ Launching SableCC on the specification file to generate a framework
- ▶ Creating one or more working classes, possibly inheriting from classes generated by SableCC
- ▶ Creating a main compiler class that activates lexer, parser and working classes.
- ▶ Compiling the compiler with a Java compiler

Table of Contents

- Motivation
 - What, How and Why
- Background
 - Coordinated Atomic Action (CAA)
 - The COALA Language
 - The CAA-DRIP framework
- The compiler
 - A typical compiler
 - Evaluation of different tools
 - The SableCC framework
- Problems found
 - Ambiguous grammars, Shift/Reduce conflicts
 - CST – AST transformations
- Future work

Problems Found

- Ambiguous grammars
- Shift/Reduce conflicts
- Reduce/Reduce conflicts
- CST – AST transformations

Ambiguous grammars and ambiguous languages

An Example

given the grammar:

$$A \rightarrow A + A \mid A - A \mid a$$

This grammar is ambiguous since there are two parse trees for the string “ $a + a - a$ ”

However, the language that it generates is not ambiguous because exists a non-ambiguous grammar generating the same language:

$$A \rightarrow A + a \mid A - a \mid a$$

A simple example of shift/reduce conflicts

The grammar

expr \rightarrow expr + expr

the string to parse is: "*expr+ expr +expr*"

two interpretations are possible:

(1) (expr+expr) + expr

(2) expr + (expr+expr)

When the parse is in this position "*expr+ expr (.) +expr*"

- to reduce expr + expr as expr (case 1)
- to shift the next expr (case 2)

In other words, the parser **don't have enough information** to make a decision !

A simple example of reduce/reduce conflicts:

The grammar

$$E \rightarrow A \mid B$$
$$A \rightarrow a \mid b$$
$$B \rightarrow a$$

will produce a reduce/reduce conflict
after parsing an “a”.

and the parser will be in this position

$$E \rightarrow A \mid B$$
$$A \rightarrow a (.) \mid b$$
$$B \rightarrow a (.)$$

Solution: to eliminate the rules that are in conflict by making some transformations in the grammar

Example, E can be rewritten as $E \rightarrow a \mid b$

CST – AST transformations: A simple example

Suppose that we have the following grammar implemented in SableCC

Tokens

```
add = '+';
mul = '*';
left_paren = '(';
right_paren = ')';
number = [0-9]+;
whitespace = (' ')+;
```

Ignored Tokens

```
whitespace;
```

Productions

```
expr = {add} [left]:expr add [right]:expr
      | {mul} [left]:expr mul [right]:expr
      | {paren} left_paren expr right_paren
      | {number} number;
```

CST – AST transformations: A simple example

The grammar rewritten is

Productions

```
expr = {add} expr add factor  
      | {factor} factor;
```

```
factor= {mul} factor mul value  
        | {value} value;
```

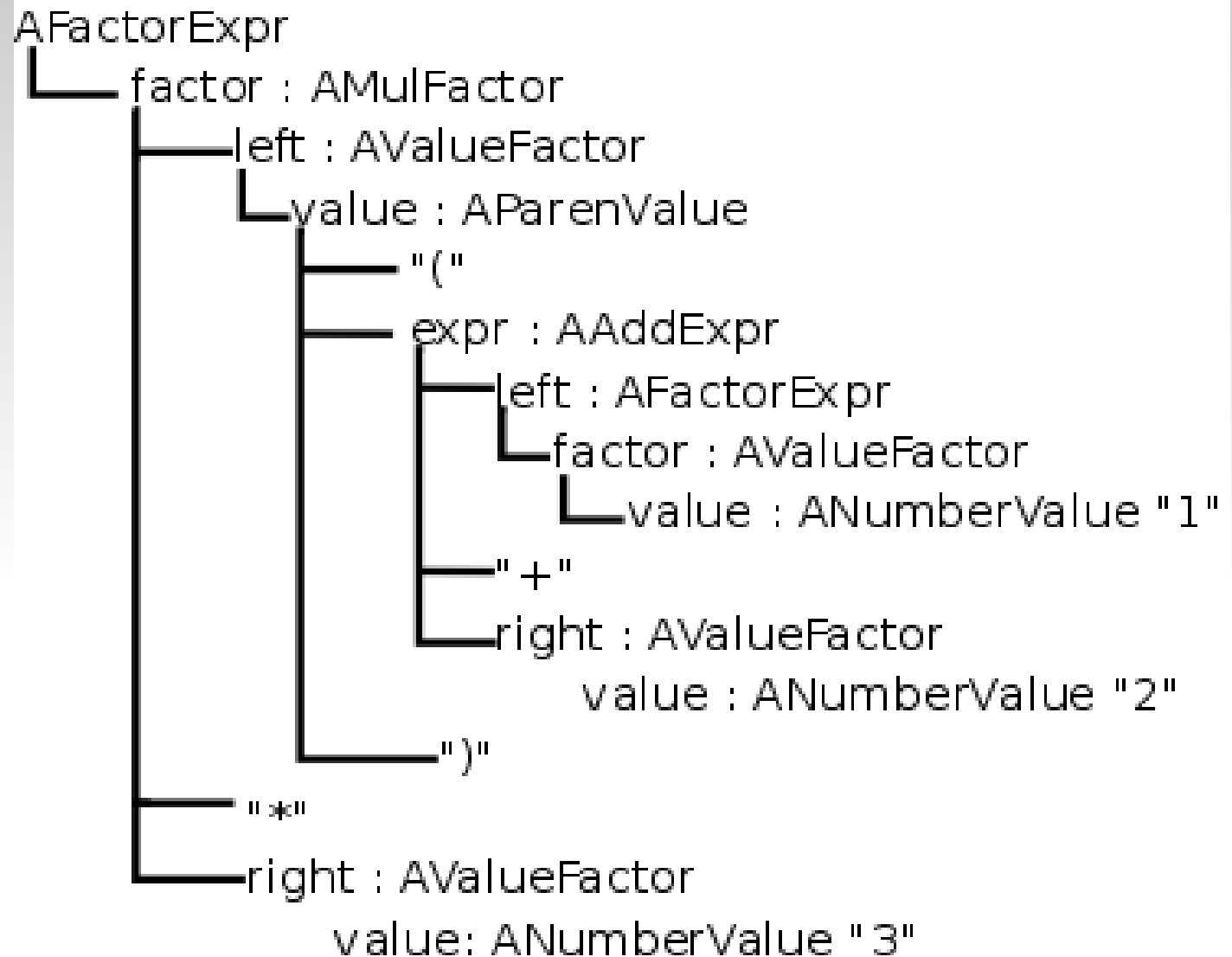
```
value = {number} number  
        | {paren} left_paren expr right_paren;
```

However, now is no so easy to deal with the CST

Parsing the expression

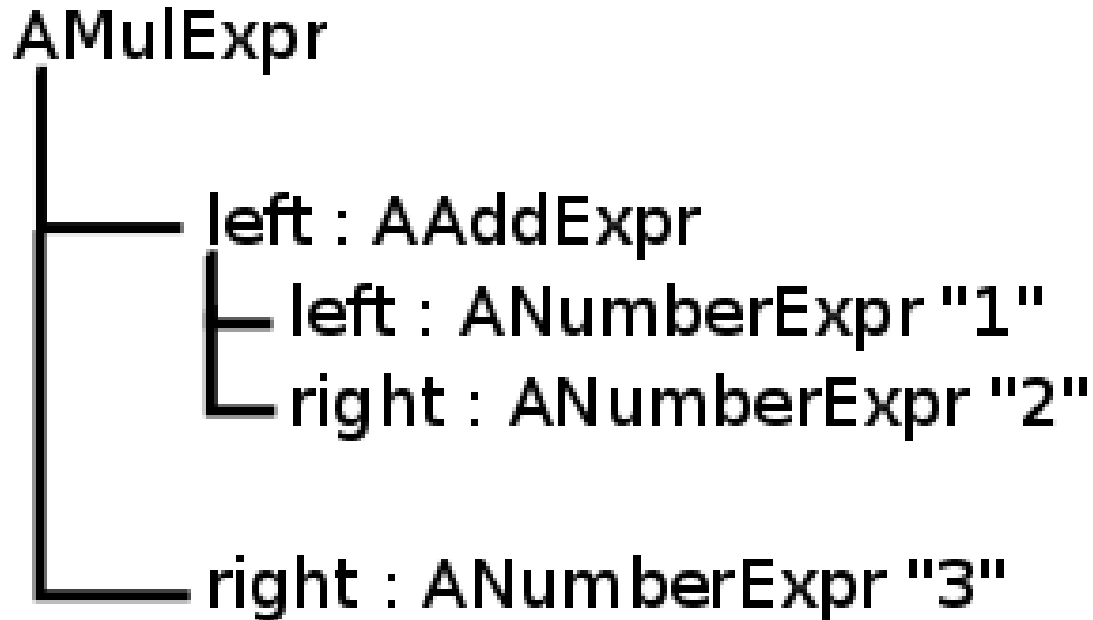
(1 + 2) * 3

produces the concrete syntax tree (CST) that is shown in the picture



Abstract Syntax Tree (AST)

Here is the equivalent AST for the expression: $(1+2) * 3$



Thus, is necessary to make a transformation !

CST -> AST Transformation

Productions

```
expr {-> expr} =  
  {add} expr add factor {->New expr.add(expr, factor)}  
  | {factor} factor      {-> factor.expr};
```

```
factor {-> expr} =  
  {mul} factor mul value {->New expr.mul(factor, value)}  
  | {value} value        {-> value.expr};
```

```
value {-> expr} =  
  {number} number        {-> New expr.number(number)}  
  | {parens} left_paren expr right_paren {->expr.expr};
```

Abstract Syntax Tree

```
expr =  
  | {add} [left]:expr [right]:expr  
  | {mul} [left]:expr [right]:expr  
  | {number} number;
```

Table of Contents

- Motivation
 - What, How and Why
- Background
 - Coordinated Atomic Action (CAA)
 - The COALA Language
 - The CAA-DRIP framework
- The compiler
 - A typical compiler
 - Evaluation of different tools
 - The SableCC framework
- Problems found
 - Ambiguous grammars, Shift/Reduce conflicts
 - CST – AST transformations
- Future work

Future Work

About the compiler

The main steps to be completed are:

- To implement the semantic rules previously mentioned
- To translate to intermediate code
- To generate of Java code customized for CAA-DRIP

Future Work

About the Language

- Inclusion of **types** in Coala
 - ▶ to define **operators over types**
 - ▶ to provide the necessary **semantic controls over types**
- Enrich the grammar with **loop statements** like the classic “**for**” or “**while**”
- Enhance Coala by adding new characteristics, for example
 - ▶ by adding the **finally clause to the try/catch sentence** to manage exceptions that have not been associated with a handler
 - ▶ by providing the **support to manage composite CAA**
- To change the grammar to make it **easier to use**, for example by replacing the **begin/end** reserved words by braces **'{ / }'**

Bibliography

- [1] Julie VACHON. “*COALA: A Design Language for Reliable Distributed Systems*”. *Phd Thesis, École Polytechnique Fédérale de Lausanne, Switzerland, #2302, 2000*
- [2] Andrew W. Appel. “*Modern Compiler Implementation in Java.*”
Book published by Cambridge University Press, 1998.
- [3] Aho, Sethi and Ullman. “*Compilers: Principles, Techniques and Tools.*”
Book published by Addison-Wesley, 1986.

Bibliography

- [4] Étienne CAGNON. *“SableCC, an object oriented compiler framework”*
School of Computer Science, McGill University, Montreal, March 1998
- [5] A. Capozucca, N. Guelfi, P. Pelliccione, A. Romanovsky, A. Zorzo. *CAA-DRIP: a framework for implementing Coordinated Atomic Actions*
The 17th International Symposium on Software Reliability, Raleigh, North Carolina, USA, 2006

**Thank you very much for
your interest in this work**

Questions ?

A real (and common) example of a shift/reduce conflict

We have now the ambiguous grammar

$S \rightarrow \text{if } E \text{ then } S$

$S \rightarrow \text{if } E \text{ then } S \text{ else } S$

$S \rightarrow \text{other}$

and the string to parse is: "if a then if b then s1 then s2"

Again, two interpretations are possible:

- if a then { if b then s1 else s2 }
- if a then { if b then s1 } else s2

In most programming languages the usual interpretation is the first one. Thus, when the parser will be in this situation

$S \rightarrow \text{if } E \text{ then } S (.)$

$S \rightarrow \text{if } E \text{ then } S (.) \text{ else } S$

it has to decide between a reduce or a shift

The solution: to rewrite the grammar

Then the grammar

$S \rightarrow \text{if } E \text{ then } S$

$S \rightarrow \text{if } E \text{ then } S \text{ else } S$

$S \rightarrow \text{other}$

can be changed by introducing auxiliary nonterminals M and U such that

$S \rightarrow M$

$S \rightarrow U$

$M \rightarrow \text{if } E \text{ then } M \text{ else } M$

$M \rightarrow \text{other}$

$U \rightarrow \text{if } E \text{ then } S$

$U \rightarrow \text{if } E \text{ then } M \text{ else } U$

Evaluation of different tools

1) JLEX/CUP

- **Java equivalent of the popular lex/yacc (Yet Another Compiler Compiler)**
- **CUP generates a LALR(1) parsers**
- **CUP generates a parser that will execute the action code associated with each alternative**

Evaluation of different tools

2) PCCTS/JavaCC

- **PCCTS/JavaCC generates a LL(k) parsers**
- **Extended Backus-Naur Form (EBNF) syntax**
- **AST generated automatically**
- **Support for tree-walkers**

Evaluation of different tools

3) SableCC

- **SableCC specification files do not contain any action code.**
- **AST generated automatically**
- **Shorter development cycle**
- **Support for tree walkers**

Rightmost Derivation: An Example

Given the grammar:

$$\begin{array}{ll} S \rightarrow \text{id} := E & E \rightarrow \text{num} \\ E \rightarrow E + E & E \rightarrow \text{id} \\ E \rightarrow (S, E) & \end{array}$$

If we want to parse the expression

$$\text{id} := \text{id} + (\text{id} := \text{num} + \text{num}, \text{id})$$

the rightmost derivation would be

$$\begin{array}{l} \underline{S} \\ \text{id} := \underline{E} \\ \text{id} := E + \underline{E} \\ \text{id} := E + (S, \underline{E}) \\ \text{id} := E + (\underline{S}, \text{id}) \\ \text{id} := E + (\text{id} := \underline{E}, \text{id}) \\ \text{id} := E + (\text{id} := E + E) \end{array}$$

Hierarchy of grammar classes

