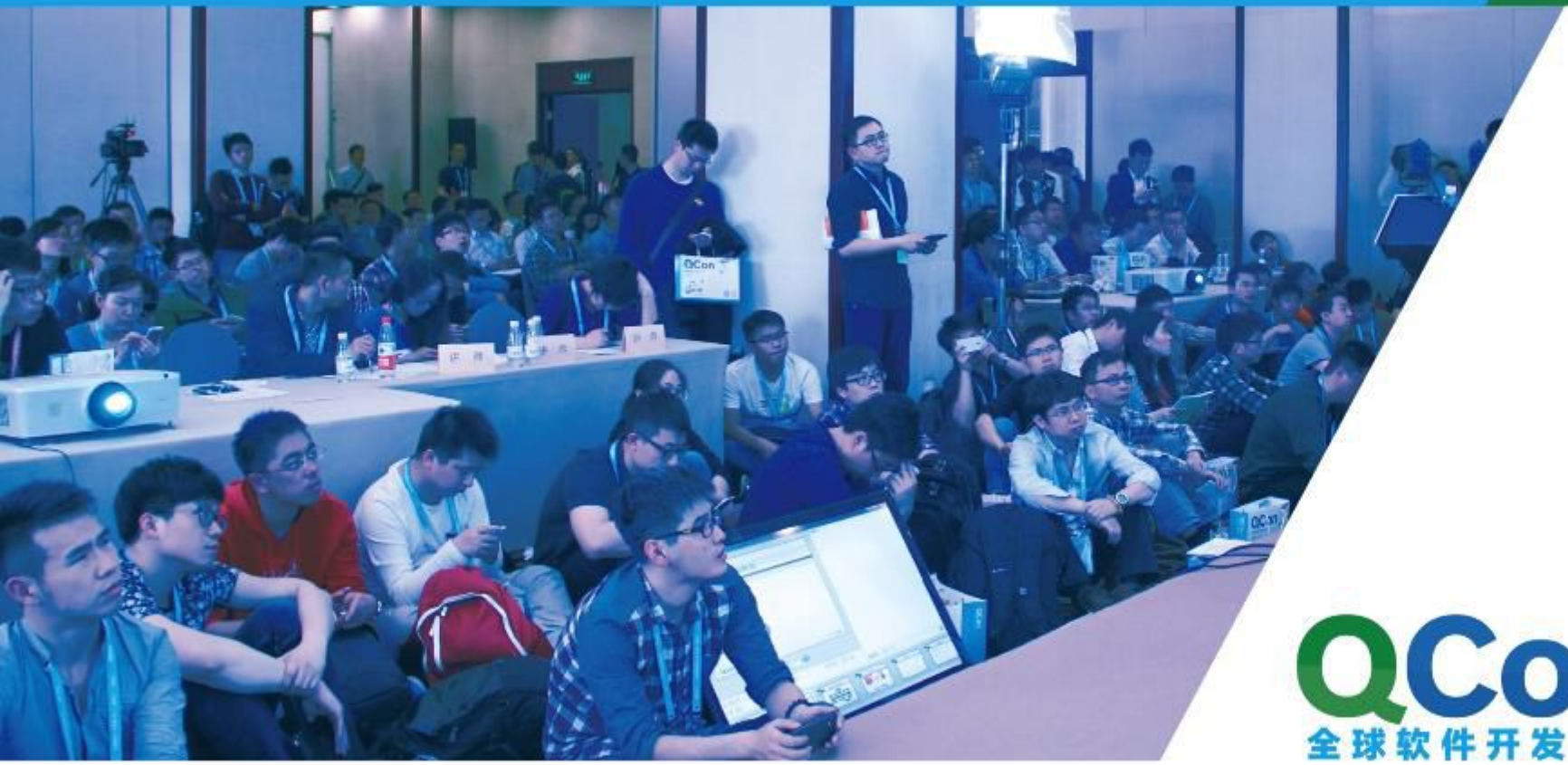
QCon全球软件开发大会

International Software Development Conference



QCon
全球软件开发大会

Parser组合子

从玩具到专业工具

程序猿的编译器情结

为什么我们仍然需要写Parser

- 编译器的前端
- DSL和解释器模式
- 大量文本表示的数据
  - JSON，Mark Down，模板生成器
- 某些人机交互的语言

Parser是已经解决的问题吗？

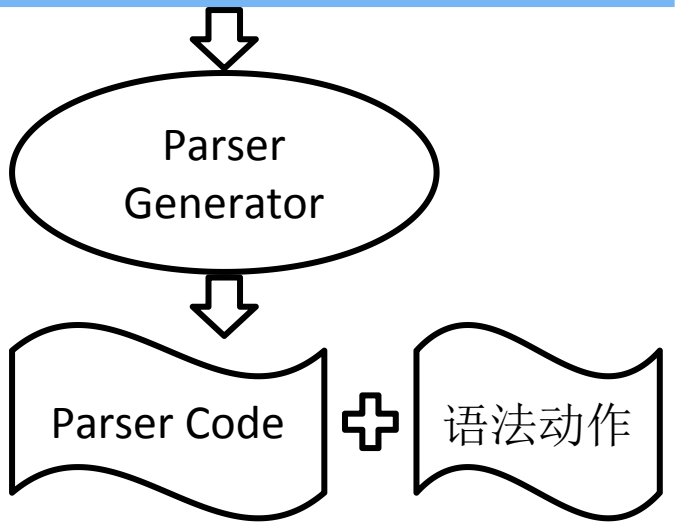
- 理论上是的，实际是……
- 写一个非常好的Parser非常耗时间
- 能生成好的Parser的工具都难用
- 好用的Parser生成工具生成的Parser都不好
- 唯一的方法是重新发明轮子……

Parser组合子的简介-大纲

- Parser生成器 vs. Parser组合子
- 适合组合子的文法
- 寻找适合组合文法的编程结构

Parser生成器 vs. Parser组合子

```
N ::= '0'..'9' | '(' E ')'
F ::= N | F '*' N
T ::= F | T '+' F
E ::= T
```




```
Production<Expression> N, F, T;
F.Rule = N |
         from left in F
         from op in Asterisk
         from right in N
         select new BinaryExp(
             left, op, right);
```

文法与语法动作直接融汇在代码中

适合组合子的文法

```
X ::= ‘a’
X ::= A B ‘c’ D
X ::= [A]
X ::= {B}
X ::= A | B
X ::= A?
…
```

EBNF扩展巴科斯范式
表达能力丰富
适合Parser生成器

```
X → ‘a’
X → ε
X → A B

X → A | B
X → Y
```

ε-乔姆斯基范式+2种扩展
其余的操作利用组合的力量来表达
适合Parser组合子

寻找适合组合文法的编程结构

- X → ‘a’

```
X = Grammar.AsTerminal(‘a’);
```

- X → ε

```
X = Grammar.Empty(parseResult);
```

- X → A | B

```
X = Grammar.Union(A, B);
```

- X → A B

```
X = Grammar.Concat(A, B); //为什么不行呢
```

```
X = from a in A
    from b in B
    select new AstNode(a, b);

X = A.SelectMany(a => B, (a, b) =>
    new AstNode(a, b)));
```

- X → Y

```
X = from y in Y select Transform(y);

X = Y.Select(y => Transform(y));
```

第一个Parser组合子及其改进-大纲

- 第一个Parser组合子
- 问题与改进
- 引进CPS风格
- 自动错误恢复
- 小结

第一个Parser组合子的基础设施

```
public delegate Result<T> Parse<T>(ForkableScanner s);

public class Result<T>
{
    public T Value { get; }
    public ForkableScanner Rest { get; }

    public Result(T value, ForkableScanner rest)
    {
        Value = value;
        Rest = rest;
    }
}
```

组合子函数原型

```
//X → ‘a’
public static Parse<string> AsTerminal(this Token token)

//X → ε
public static Parse<T> Empty<T>(T value)

//X → A | B
public static Parse<T> Union<T>(
    this Parse<T> parse1,
    Parse<T> parse2)
```

组合子函数原型

```
//X → A B
public static Parse<TR> SelectMany<T1, T2, TR>(
    this Parse<T1> parse1,
    Func<T1, Parse<T2>> parse2Selector,
    Func<T1, T2, TR> resultSelector)

//X → Y
public static Parse<TR> Select<T, TR>(
    this Parse<T> parse1,
    Func<T, TR> resultSelector)
```

- Demo文法：混合加乘运算表达式

```
U → '[0..9]+'
U → '(' T ')'

F → U
F → F '*' U

T → F
T → T '+' F

E → T$
```

左递归！

```
U → '[0..9]+'
U → '(' T ')'

F → U F1
F1 → '*' U F1
F1 → ε

T → F T1
T1 → '+' F T1
T1 → ε

E → T$
```

消除左递归的文法

缺点总结

- 无限制的回溯
- 左公因式极大地影响性能
- 产生式的顺序对结果有影响
- 语法错误无法准确表达，且只能报告第一个错误
- 不支持左递归，写法冗长丑陋

第一步改进：CPS风格的Parse函数

- 目标1：分支语法由深度优先的方式改为广度优先
- 目标2：单步执行Parse函数，从而能在每一步做出决策
- 解决方案：CPS (Continuation Pass-in Style)

```
public delegate Result<T> Parse<T>(ForkableScanner s);

// Continuation函数原型
public delegate Parse<TFuture> Future<in T, TFuture>(T value);
```

基础设施的改变

```
// 用来构建Parse<T>的构建器
public abstract class ProductionBase<T>
{
    // CPS风格的构建函数
    public abstract Parse<TFuture> BuildParse<TFuture>(
        Future<T, TFuture> future // Continuation
    );
}

// 结果类的基类
public abstract class Result<T>
{
    public abstract T GetResult();
}
```

```
// 单步解析的结果
public class StepResult<T> : Result<T>

// 解析结束时的结果
public class StopResult<T> : Result<T>
```

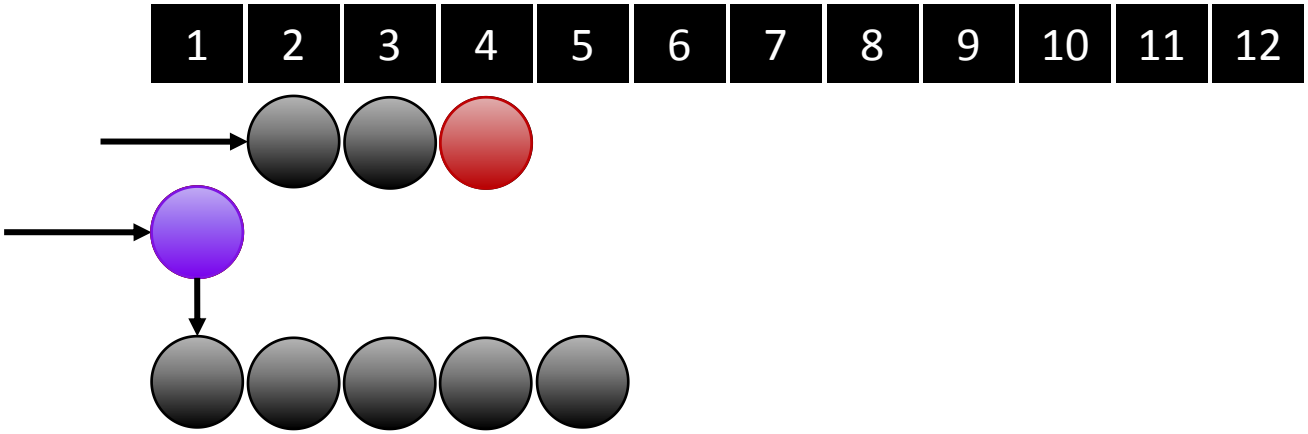
组合子的改变

- 现在组合子不是在Parse<T>上进行组合，而是在ProductionBase<T>上进行组合
- 每一步Parse都将自己的Parse结果传给自己的Future
- StepResult令解析动作单步执行
- Result.GetResult方法驱动Parse向前进行
- 请通过Demo研究上述变化

自动错误恢复

- 错误恢复的意义
  - 报告输入中全部语法错误，而不仅仅是第一个
- 错误恢复的传统做法
  - 给定错误恢复产生式——额外工作
  - 在自顶向下解析器中手动恢复错误——很多代码
- 设计目标：全自动的错误恢复

自动错误恢复策略



三个改变

- Result<T>增加记录错误等级，以及汇总解析错误的能力
- Terminal产生式进行错误恢复
- Best函数选取错误等级较低的分支继续解析
- 请通过Demo研究上述变化

小结

- 至此我们得到了一套自动错误恢复的自顶向下解析器组合子
- 支持LL(∞)文法
- 无限回溯得到了一定程度的缓解
- 仍然不支持左递归文法
- 出现连续语法错误时将会产生爆炸式增长的错误恢复分支
- 左公因式仍然被计算多次

引入LR自动机-大纲

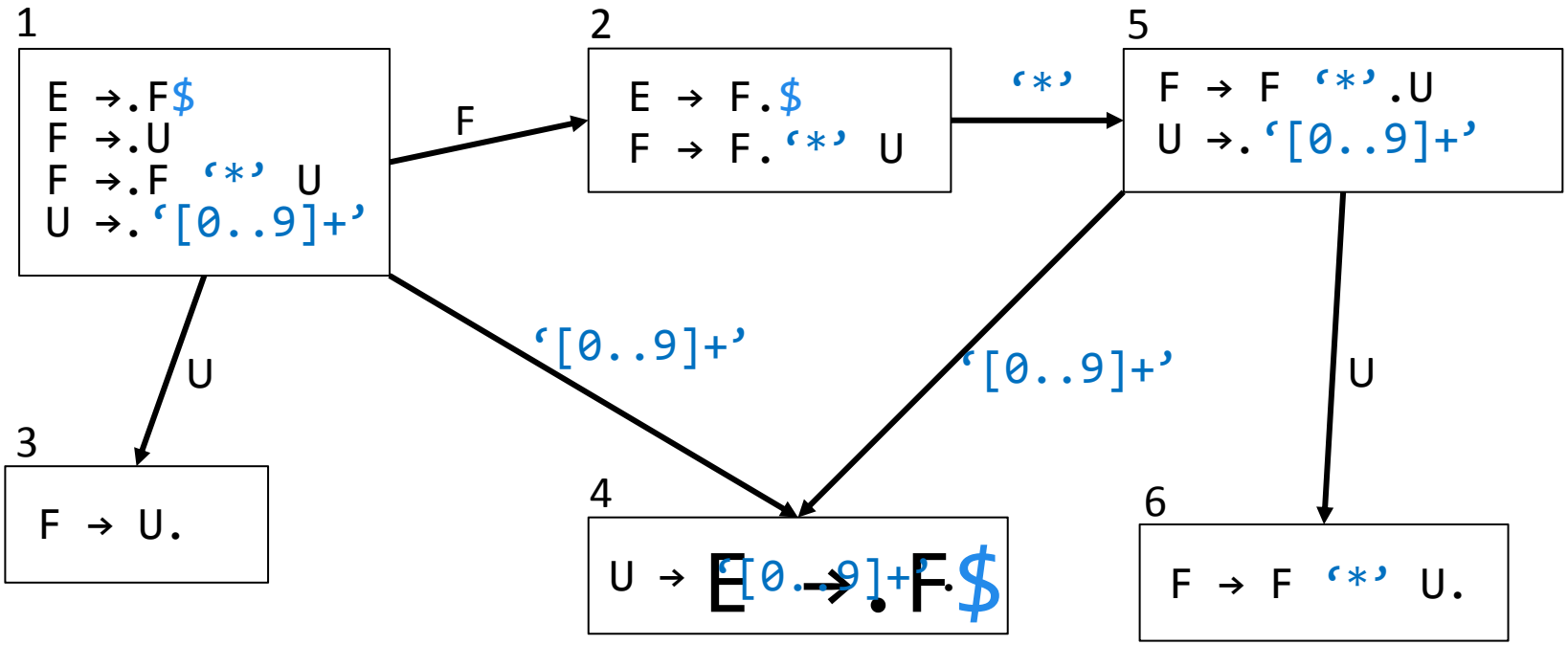
- 自底向上解析——LR(0)自动机的生成
- 多头栈的引入——从LR(0)进化为GLR
- 恐慌模式的错误恢复策略
- 歧义合并器
- 最终成品

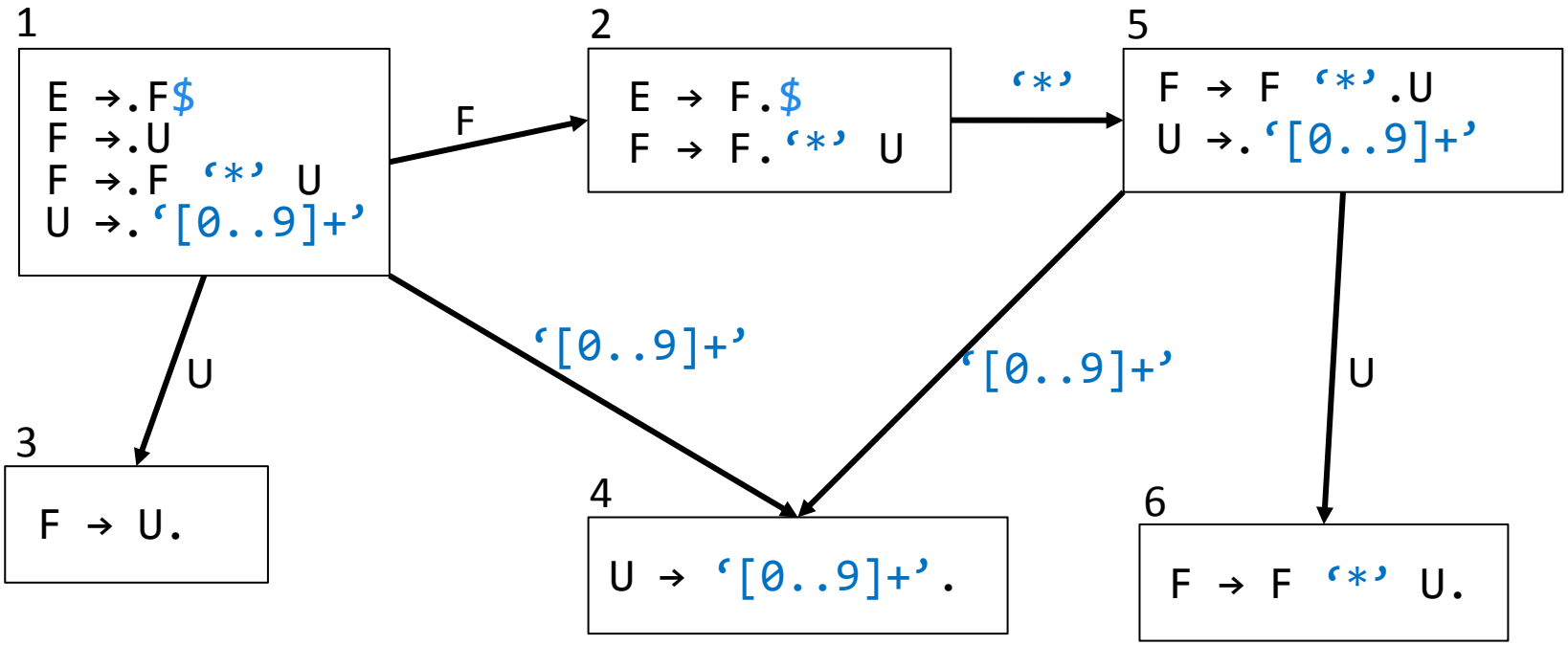
自底向上解析

- 第一步：从文法生成自动机

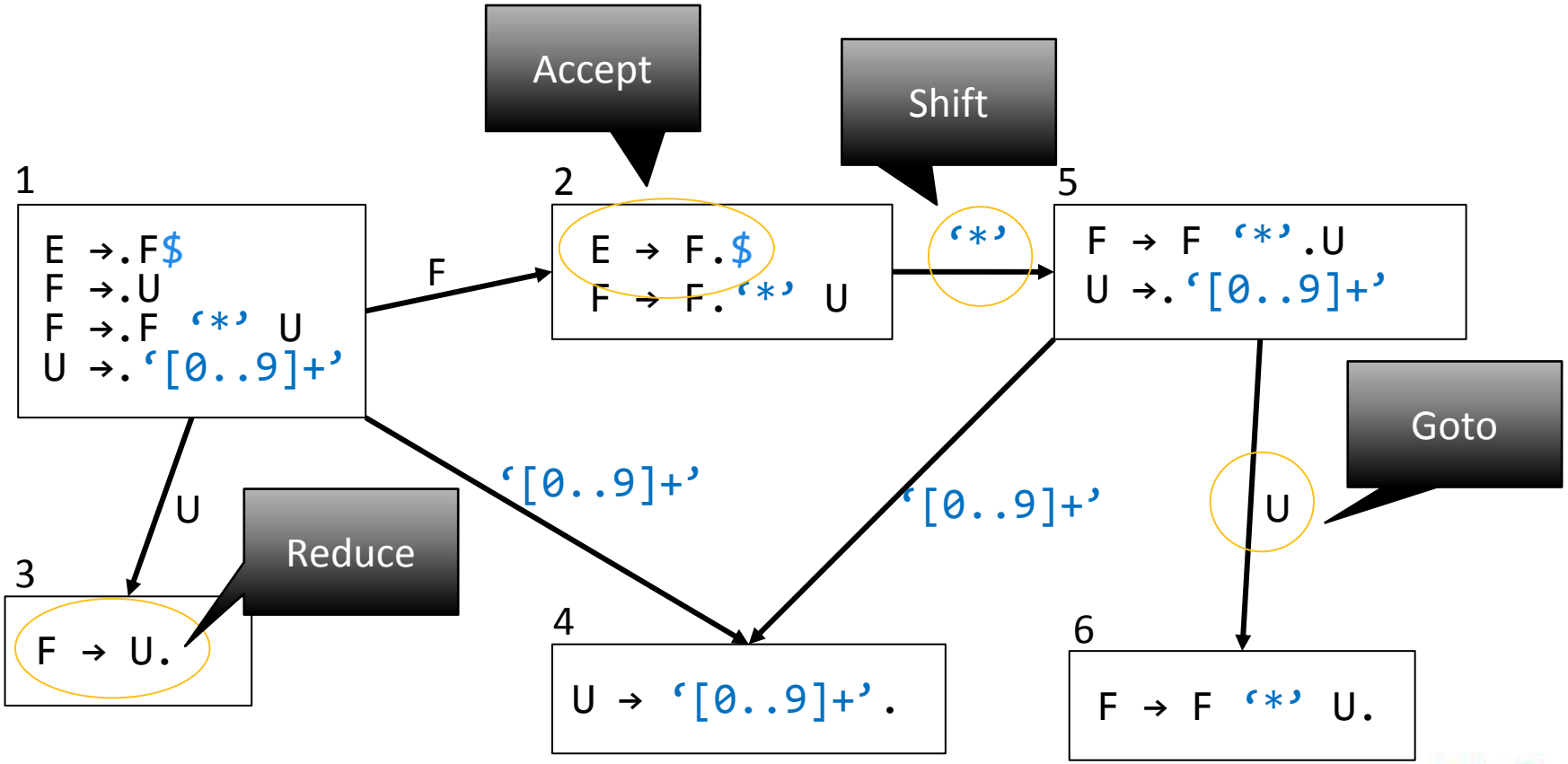
```
U → '[0..9]+'
U → '(' T ')'

F → U
F → F '*' U

T → F
T → T '+' F

E → F$
```

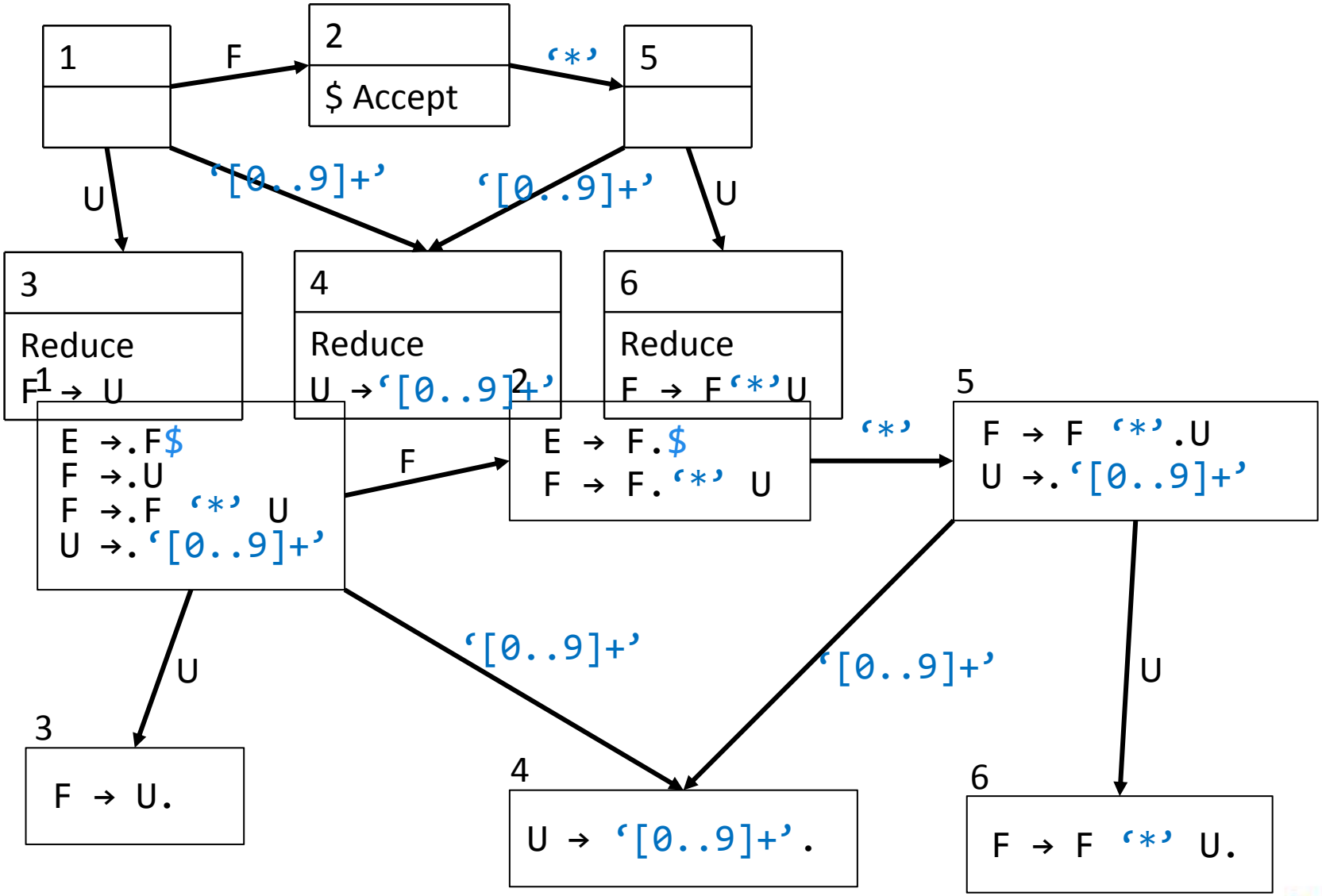
```
E →.F$
```

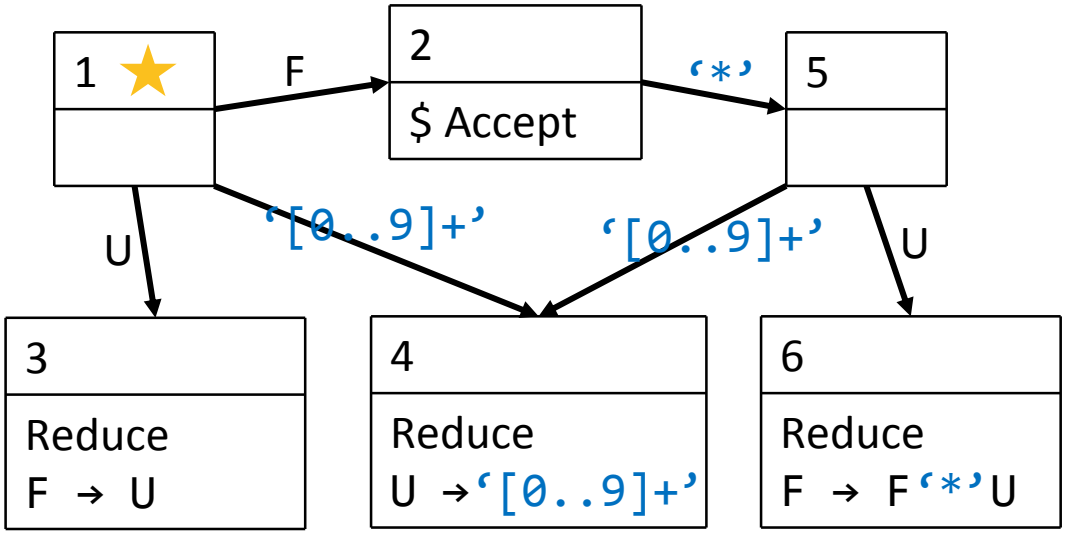

1
E →.F$
F →.U
F →.F '*' U
U →.'[0..9]+'
2
E → F.$
F → F.'*' U
5
F → F '*'.U
U →.'[0..9]+'
3
F → U.
4
U → '[0..9]+'.
6
F → F '*' U.
F
U
'*'
'[0..9]+'
'[0..9]+'
U


1
E →.F$
F →.U
F →.F '*' U
U →.'[0..9]+'
2
E → F.$
F → F.'*' U
5
F → F '*'.U
U →.'[0..9]+'
3
F → U.
4
U → '[0..9]+'.
6
F → F '*' U.
F
'*'
U
'[0..9]+'
'[0..9]+'
U

LR(0)自动机的动作


Accept
Shift
1
E →.F$
F →.U
F →.F '*' U
U →.'[0..9]+'
2
E → F.$
F → F.'*' U
'*'
5
F → F '*'.U
U →.'[0..9]+'
F
U
'[0..9]+'
'[0..9]+'
U
Goto
Reduce
3
F → U.
4
U → '[0..9]+'.
6
F → F '*' U.


1
F
2
$ Accept
'*'
5
U
'[0..9]+'
'[0..9]+'
U
3
Reduce
F → U
4
Reduce
U → '[0..9]+'
6
Reduce
F → F '*' U
1
E →.F$
F →.U
F →.F '*' U
U →.'[0..9]+'
F
2
E → F.$
F → F.'*' U
'*'
5
F → F '*'.U
U →.'[0..9]+'
U
'[0..9]+'
'[0..9]+'
U
3
F → U.
4
U → '[0..9]+'.
6
F → F '*' U.

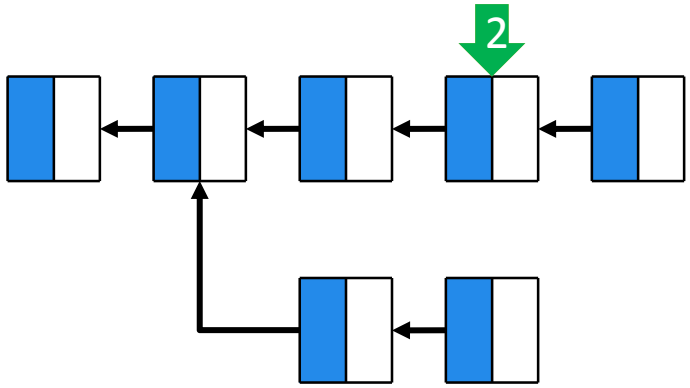

1
F
2
$ Accept
'*'
5
U
'[0..9]+'
'[0..9]+'
U
3
Reduce
F → U
4
Reduce
U →'[0..9]+'
6
Reduce
F → F'*'U


1
2
5
*
6


3 * 7

从LR(0)到GLR

- 自动机中的冲突项
  - Shift-Shift冲突
  - Shift-Reduce冲突
  - Reduce-Reduce冲突
- 多头栈的引入



- 利用Best函数选出能够继续解析的栈结点
- 利用GC清除过期的栈结点

错误恢复策略

- 当一个输入单词没有任何合法的Shift或Reduce动作，进入错误恢复
- 复原栈的位置到上一次成功的Shift后（取消其后的Reduce效果）
  - 因为LR(0)自动机的Reduce动作可能是多余的
- 利用多头栈尝试以下选项
  - 删掉当前输入单词，直接读入下一个输入单词
  - 在自动机的当前状态，尝试插入所有Shift动作的单词
  - 删掉当前输入单词后，尝试插入所有Shift动作的单词（相当于替换）
- 利用Best函数选取错误最少的路线

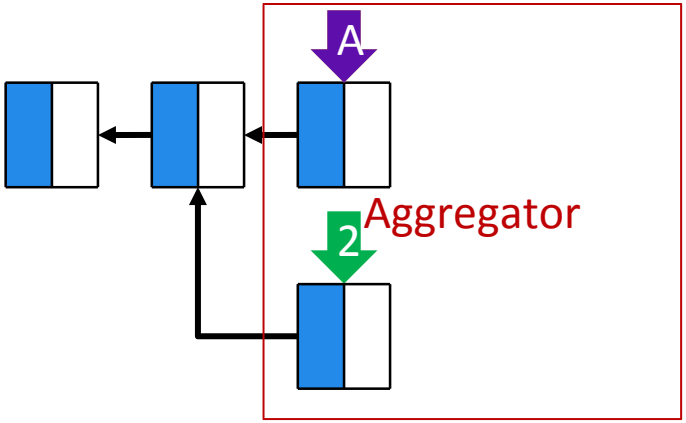
恐慌模式的错误处理

- 普通的错误处理仍然很难处理连续大量错误的情况
- 当多头栈中并行处理的错误恢复结点大于阈值时触发
- 处理步骤
  - 从多头栈中任意选取一个栈顶结点，抛弃其他并存的栈顶结点
  - 持续弹栈，直到栈顶状态S存在Goto动作，取得Goto动作的非终结符A
  - 持续读取和抛弃输入单词，直到有一个单词是在A的Follow集合当中
  - 模拟执行Goto(S,A)动作，将目标结点压栈
  - 回到标准模式运转

歧义合并器

- 当文法存在歧义时，多头栈的多个栈顶结点将持续并存

```
Statements.Rule = Statement.Many();
Statement.Rule = VarDeclStatement | ExpressionStatement;

Statement.AmbiguityAggregator = (s1, s2) => new
AmbiguityStatementNode(s1, s2);
```



最终成品

- 完美支持左递归文法，左公因式不会有任何额外开销
- 支持任何CFG，有歧义的文法也可以使用
- 大大减少错误处理的分支（需要用LR或LALR自动机代替LR(0)自动机）
- 用法上完全没有变复杂

专业工具级

下一步

- 本课程的代码：

https://github.com/ninputer/qcon_demo

- Github项目VBF.Compilers

https://github.com/ninputer/vbf

THANKS