

# Compiler Construction

## LR(0) + Visitor

Ran Shaham and Ohad Shacham  
School of Computer Science  
Tel-Aviv University

---

---

---

---

---

---

---

---

## PA2

- Submission extended to Dec 25 due to complexity quiz
- Please submit according to the requested submission structure
- Make sure that you assign the value to symbol's value in class Token
- PA1 grades + comments will be available soon

2

---

---

---

---

---

---

---

---

## TA1

- Theoretical assignment regarding parsing
- Read carefully
- Submit to Paz

3

---

---

---

---

---

---

---

---

## LR(0) parsing

1. Construct transition relation between states
  - Use algorithms **Initial item set** and **Next item set**
  - States are set of LR(0) items
    - Shift items of the form  $P \rightarrow \alpha \bullet S \beta$
    - Reduce items of the form  $P \rightarrow \alpha \bullet$
2. Construct parsing table
  - If every state contains no conflicts use LR(0) parsing algorithm
  - If states contain conflict
    - Rewrite grammar or
    - Use stronger parsing technique

4

---

---

---

---

---

---

---

---

## LR(0) Example

$S \rightarrow ES$   
 $E \rightarrow T$   
 $E \rightarrow E + T$   
 $T \rightarrow i$   
 $T \rightarrow ( E )$

- non-terminals denoted by upper-case letters
- terminals denoted by lower-case letters

5

---

---

---

---

---

---

---

---

## LR(0) Example

Precomputed LR(0) items:

- 1:  $S \rightarrow \bullet ES$
- 2:  $S \rightarrow E \bullet S$
- 3:  $S \rightarrow E S \bullet$
- 4:  $E \rightarrow \bullet T$
- 5:  $E \rightarrow T \bullet$
- 6:  $E \rightarrow \bullet E + T$
- 7:  $E \rightarrow E \bullet + T$
- 8:  $E \rightarrow E + \bullet T$
- 9:  $E \rightarrow E + T \bullet$
- 10:  $T \rightarrow \bullet i$
- 11:  $T \rightarrow i \bullet$
- 12:  $T \rightarrow \bullet ( E )$
- 13:  $T \rightarrow ( \bullet E )$
- 14:  $T \rightarrow ( E \bullet )$
- 15:  $T \rightarrow ( E ) \bullet$

6

---

---

---

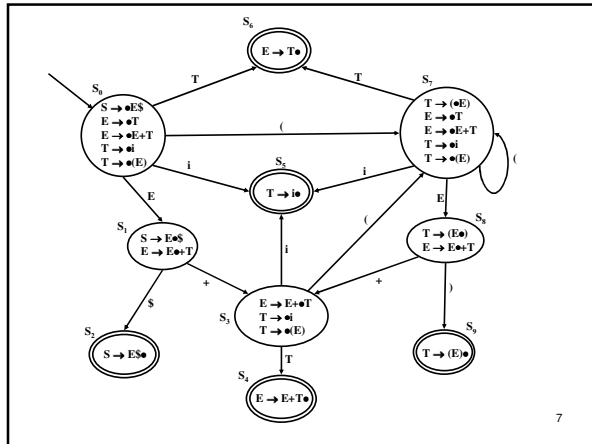
---

---

---

---

---




---

---

---

---

---

---

---

---

---

---

GOTO table symbol								ACTION table
state	i	+	(	)	\$	E	T	
0	5		7			1	6	shift
1		3			2			shift
2	S→E\$							reduce
3	5		7				4	shift
4	E→E+T							reduce
5	T→i							reduce
6	E→T							reduce
7	5		7			8	6	shift
8		3			9			shift
9	T→(E)							reduce

---

---

---

---

---

---

---

---

---

---

**LR(0) example:  $i + (i + i)$**

state	i	+	(	)	\$	E	T	
0	5		7			1	6	shift
1		3			2			shift
2	S→E\$							reduce
3	5		7				4	shift
4	E→E+T							reduce
5	T→i							reduce
6	E→T							reduce
7	5		7			8	6	shift
8		3			9			shift
9	T→(E)							reduce

Stack	Input	Action
$S_0$	$i + (i + i) \$$	shift
$S_0 i S_5$	$+ (i + i) \$$	reduce by $T \rightarrow i$
$S_0 T S_6$	$+ (i + i) \$$	reduce by $E \rightarrow T$
$S_0 E S_1$	$+ (i + i) \$$	shift
$S_0 E S_1 + S_3$	$(i + i) \$$	shift

---

---

---

---

---

---

---

---

---

---



## Example

$E \rightarrow E + E$   
 $E \rightarrow i$   
 $E \rightarrow ( E )$

Is the grammar LR(0) ?

Ambiguous  
Shift - Reduce  
conflict

1 + 2 + 3



13

---

---

---

---

---

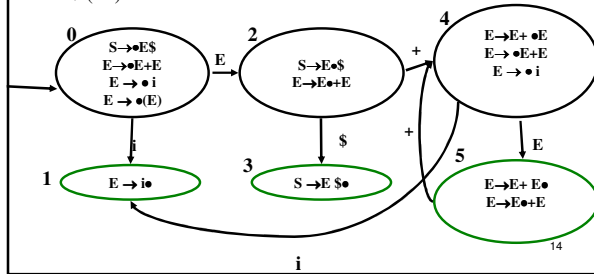
---

---

---

## Example

$S \rightarrow ES$   
 $E \rightarrow E + E$   
 $E \rightarrow i$   
 $E \rightarrow ( E )$



14

---

---

---

---

---

---

---

---

## Example

$E \rightarrow E + T$   
 $E \rightarrow ( E )$   
 $E \rightarrow T$   
 $T \rightarrow i$   
 $E \rightarrow V = E$   
 $V \rightarrow i$

Is the grammar LR(0) ?

Ambiguous  
Reduce - Reduce  
conflict

$T \rightarrow i$   
 $V \rightarrow i$

15

---

---

---

---

---

---

---

---

## Example

$E \rightarrow E + T$   
 $E \rightarrow ( E )$   
 $T \rightarrow i[E]$   
 $T \rightarrow i$

Is the grammar LR(0) ?

Ambiguous  
Shift - Reduce  
conflict

$T \rightarrow i[E]$   
 $T \rightarrow i$

16

---

---

---

---

---

---

---

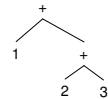
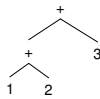
---

## Example

$E \rightarrow E + E$   
 $| E - E$   
 $| E * E$   
 $| E / E$   
 $| \text{num}$   
 $| \text{id}$

Is the grammar LR(0) ?

1 + 2 + 3



17

---

---

---

---

---

---

---

---

## Example

$E \rightarrow E + E$   
 $| E - E$   
 $| E * E$   
 $| E / E$   
 $| \text{num}$   
 $| \text{id}$

$E \rightarrow E + T$   
 $| E - T$   
 $| T$

$T \rightarrow T * F$   
 $| T / F$   
 $| F$

$F \rightarrow \text{num}$   
 $| \text{id}$

18

---

---

---

---

---

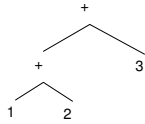
---

---

---

# Example

1 + 2 + 3  
num + num + num  
F + num + num  
T + num + num  
E + num + num  
E + F + num  
E + T + num  
E + num  
E + F  
E + T  
E



$E \rightarrow E + T$   
|  $E - T$   
|  $T$   
 $T \rightarrow T * F$   
|  $T / F$   
|  $F$   
 $F \rightarrow \text{num}$   
|  $\text{id}$

---

---

---

---

---

---

---

---

# Visitor

---

---

---

---

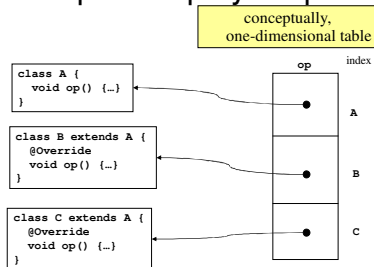
---

---

---

---

# Single dispatch - polymorphism



---

---

---

---

---

---

---

---

## What if we need more operations?

```
class A {  
  void op1() {...}  
  void op2() {...}  
  void op3() {...}  
}
```

```
class B extends A {  
  @Override  
  void op1() {...}  
  @Override  
  void op2() {...}  
  @Override  
  void op3() {...}  
}
```

```
class C extends A {  
  @Override  
  void op1() {...}  
  @Override  
  void op2() {...}  
  @Override  
  void op3() {...}  
}
```

Want to separate complicated operations from data structures

22

---

---

---

---

---

---

---

---

## What if we need more operations?

```
class A {  
}
```

```
class B extends A {  
}
```

```
class C extends A {  
}
```

×

```
class op1 extends op {  
  // lots of code  
}
```

```
class op2 extends op {  
  // lots of code  
}
```

```
class op3 extends op {  
  // lots of code  
}
```

Problem: OO languages support only single-polymorphism. We seem to need double-polymorphism

23

---

---

---

---

---

---

---

---

## What if we need more operations?

```
class A {  
}
```

```
class B extends A {  
}
```

```
class C extends A {  
}
```

```
class op1 extends op {  
  doOp(A a) {  
    // lots of code  
  }  
  doOp(B b) {  
    // lots of code  
  }  
}
```

```
class op2 extends op {  
  doOp(A a) {  
    // lots of code  
  }  
  doOp(B b) {  
    // lots of code  
  }  
}
```

```
class op3 extends op {  
  doOp(A a) {  
    // lots of code  
  }  
  doOp(B b) {  
    // lots of code  
  }  
}
```

Overloading is static

24

---

---

---

---

---

---

---

---

## Visitor Pattern

- Separate operations on objects of a data structure from object representation
- Each operation (pass) may be implemented as separate visitor
- Use double-dispatch to find right method for object
- Instance of a [design pattern](#)

25

---

---

---

---

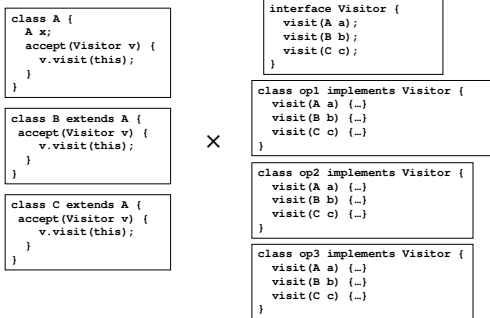
---

---

---

---

## Visitor pattern in Java



26

---

---

---

---

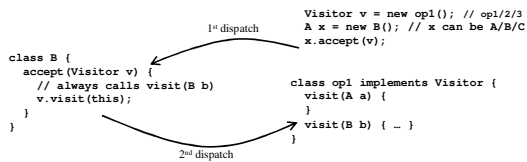
---

---

---

---

## Double dispatch example



27

---

---

---

---

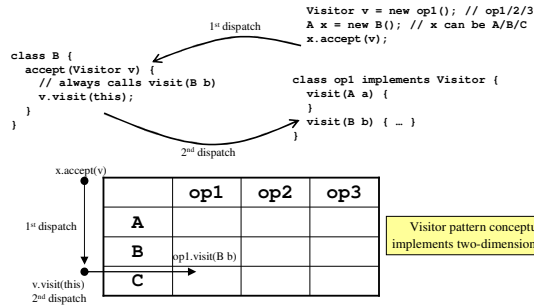
---

---

---

---

## Double dispatch example



## Visitor variations

```

interface PropagatingVisitor {
  /** Visits a statement with a given
   * context object (book-keeping)
   * and returns the result
   * of the computation on this node.
   */
  Object visit(Stmt st, Object context);
  Object visit(Expr e, Object context);
  Object visit(BinaryOpExpr e, Object context);
  ...
}
  
```

- Propagate values down the AST (and back)

29

## Evaluating visitor example

```

public class SLEvaluator implements PropagatingVisitor {
  public void evaluate(ASTNode root) {
    root.accept(this);
  }

  /** x = 2*7
   */
  public Object visit(AssignStmt stmt, Object env) {
    Expr rhs = stmt.rhs;
    Integer expressionValue = (Integer) rhs.accept(this, env);
    VarExpr var = stmt.varExpr;
    ((Environment)env).update(var, expressionValue);
    return null;
  }

  /** expressions like 2*7 and 2*y
   */
  public Object visit(BinaryOpExpr expr, Object env) {
    Integer lhsValue = (Integer) expr.lhs.accept(this, env);
    Integer rhsValue = (Integer) expr.rhs.accept(this, env);
    int result;
    switch (expr.op) {
    case PLUS:
      result = lhsValue.intValue() + rhsValue.intValue();
      ...
    }
    return new Integer(result);
  }
  ...
}
  
```

```

class Environment {
  Integer get(VarExpr ve) [-]
  void update(VarExpr ve, int value) [-]
}
  
```

30

## AST traversal

```

class BinaryOpExpr extends Expression {
    Object accept(Visitor v) {
        return v.visit(this);
    }
    Expression lhs, rhs;
}
class NumberExpr extends Expression {
    Object accept(Visitor v) {
        return v.visit(this);
    }
    int val;
}

public class SLEvaluator ... {
    public Object visit(BinaryOpExpr e, Object env) {
        Integer lhsValue=(Integer)e.lhs.accept(this,env);
        Integer rhsValue=(Integer)e.rhs.accept(this,env);
        int result;
        switch (expr.op) {
            case PLUS:
                result=lhsValue.intValue()+rhsValue.intValue();
                ...
        }
        return new Integer(result);
    }
    public Object visit(NumberExpr e, Object env) {
        return e.val;
    }
    public Object visit(VarExpr e, Object env) {
        return ((Environment)env).get(e);
    }
}
    
```

`SLEvaluator ev = new SLEvaluator();`  
`Integer result = (Integer)root.accept(ev);`

---

---

---

---

---

---

---

---

## Error recovery

- How to catch errors
  - `public void syntax_error(Symbol cur_token)`
- Optional error recovery
  - Use error token in your grammar
  - `stmt ::= expr SEMI`

```

| while_stmt SEMI
| if_stmt SEMI
| ...
| error SEMI ;
                
```

---

---

---

---

---

---

---

---