Introduction to ANTLR

邓伟信

Overview of Python Interpreter

- python language
  - dynamic type
  - flexible
- possible implementation
  - build an AST or directly work on Parse Tree
  - Symbol Table for scopes

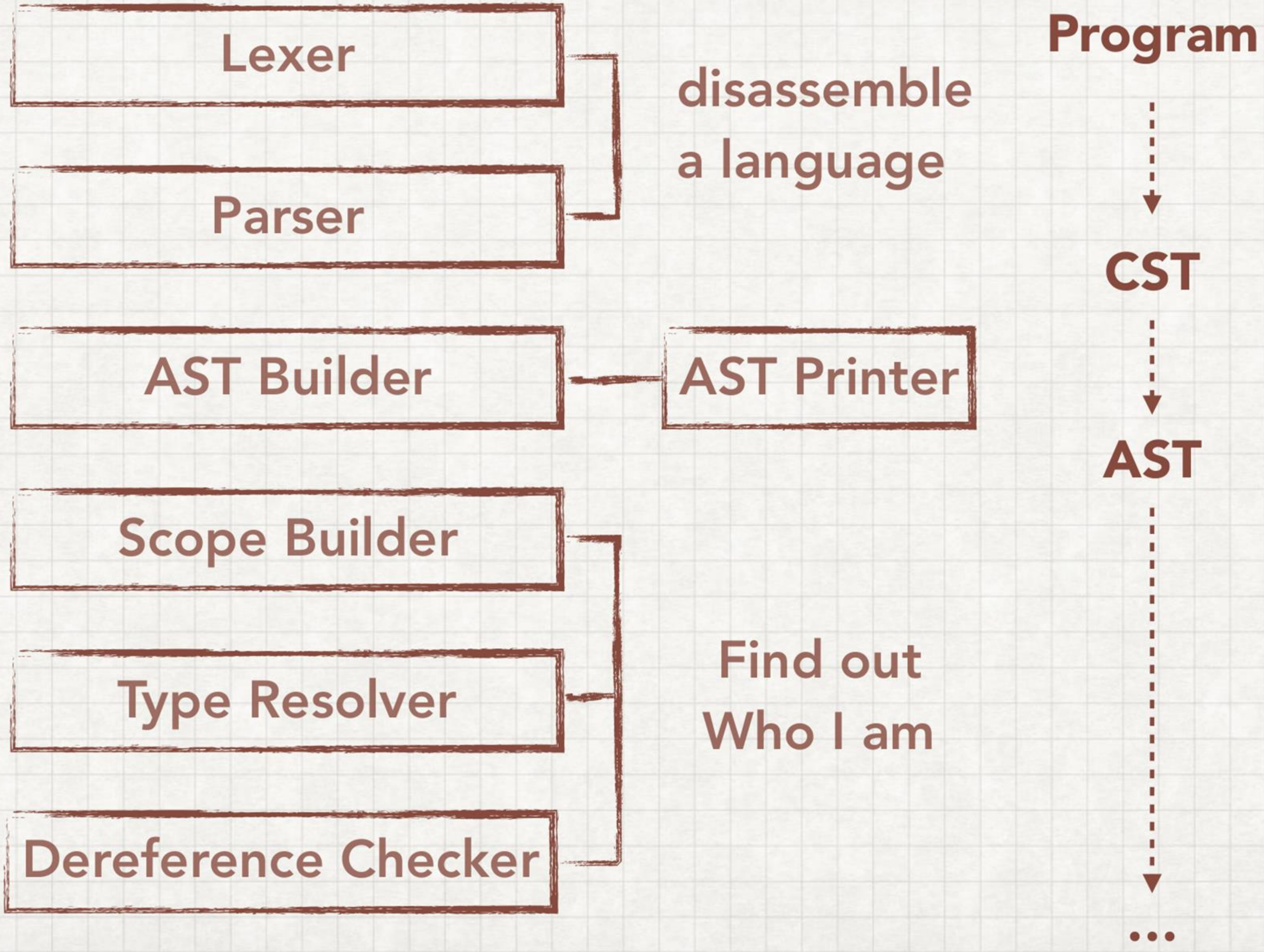
What is ANTLR? Why is ANTLR?

- ANTLR (ANother Tool for Language Recognition), is an ALL(*) parser generator.
- It is possible to hand write a parser, but this process can be complex, error prone, and hard to change.
- There are many parser generators that take a grammar expressed in an domain- specific way, and generates code to parse that language.
  - Popular parser generates include bison and yacc.
- ANTLR has a suite of tools, and GUIs, that makes writing and debugging grammars easy.

Slides from 林虹灏

Semantic Analysis

2018.04.18



Lexer

Parser

disassemble a language

AST Builder

AST Printer

Scope Builder

Type Resolver

Dereference Checker

Find out Who I am

Program

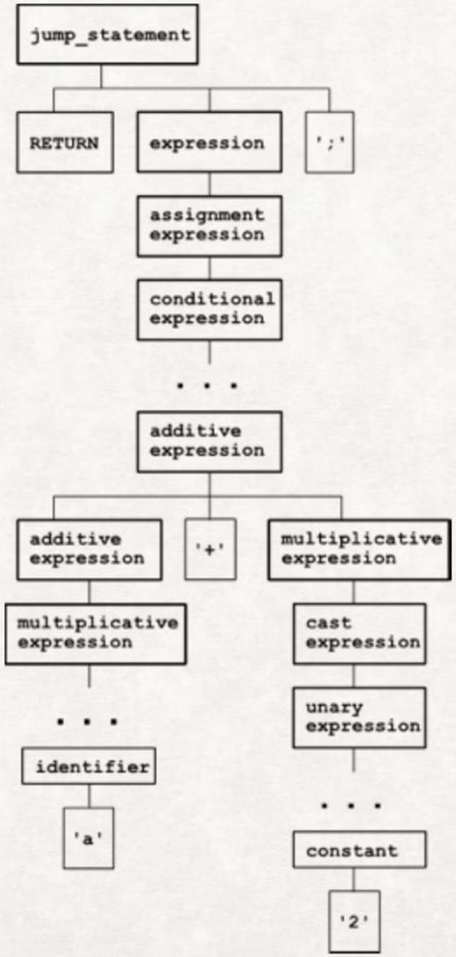
CST

AST

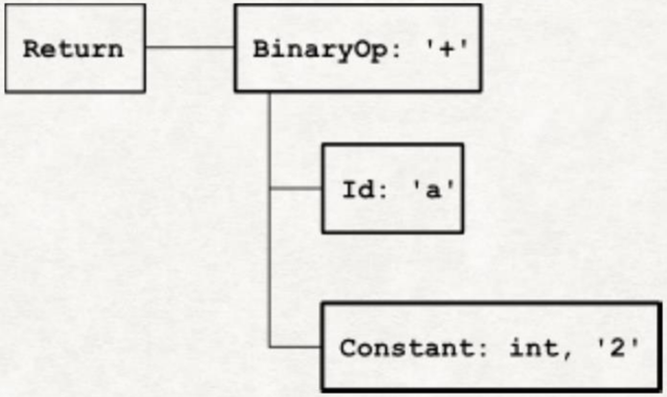
...

Review: From CST to AST

CST



AST

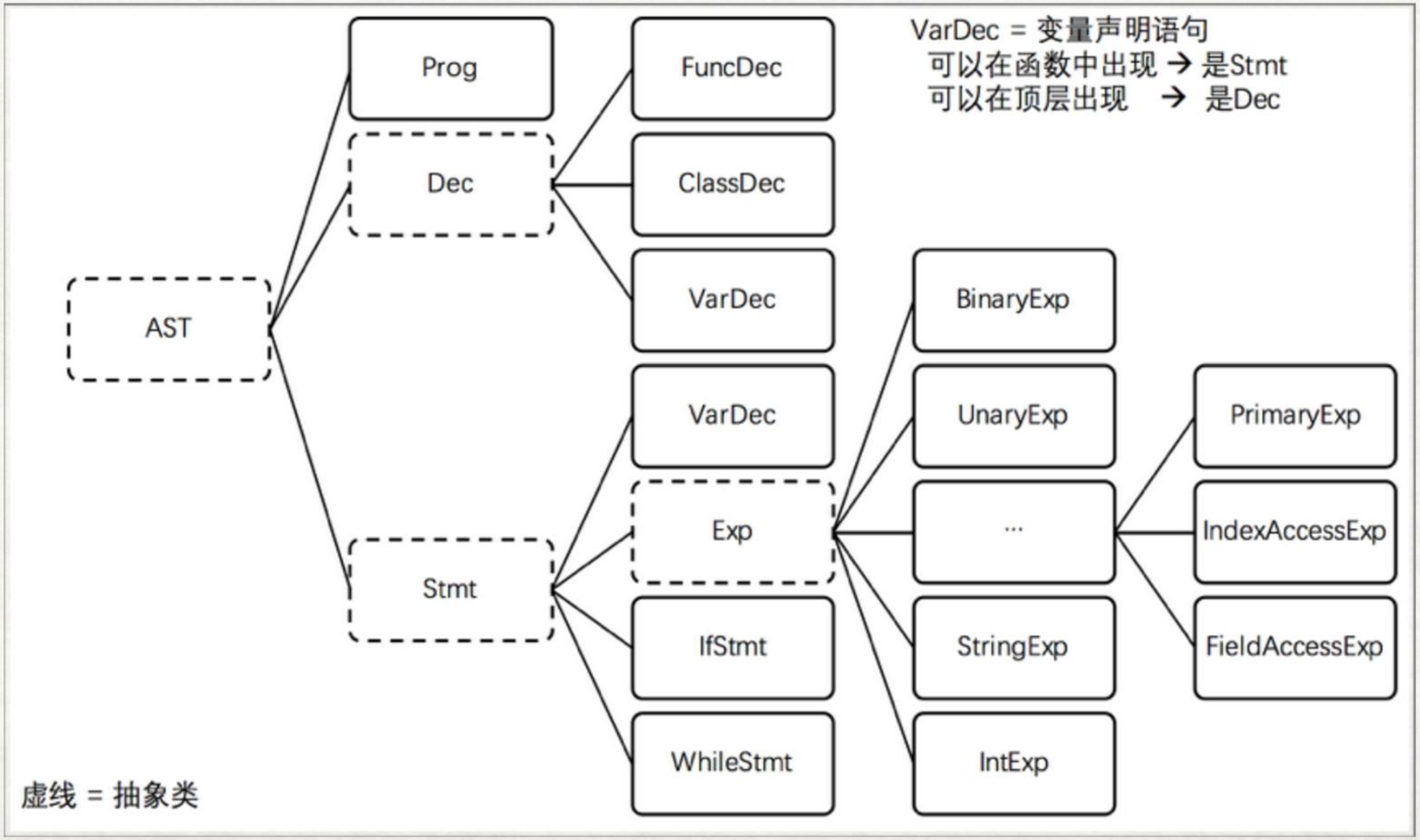


- drop syntactic clutter
- focus on structure

- simple to create
- difficult to analyze

Implement: From CST to AST

AST design：Hierarchy

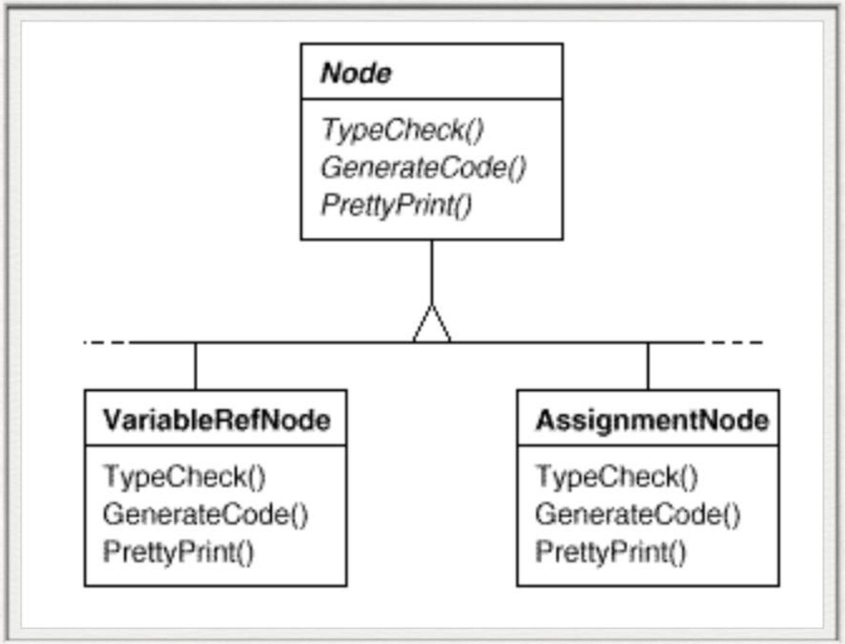


VarDec = 变量声明语句
可以在函数中出现 → 是Stmt
可以在顶层出现 → 是Dec

虚线 = 抽象类

Now We Have An AST…

We want to define operations for type-checking, code optimization, flow analysis



Node

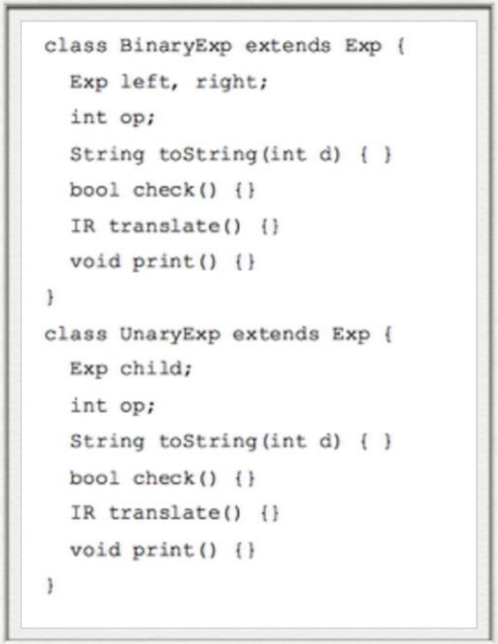
TypeCheck()
GenerateCode()
PrettyPrint()

VariableRefNode

TypeCheck()
GenerateCode()
PrettyPrint()

AssignmentNode

TypeCheck()
GenerateCode()
PrettyPrint()



```
class BinaryExp extends Exp {
  Exp left, right;
  int op;
  String toString(int d) { }
  bool check() {}
  IR translate() {}
  void print() {}
}
class UnaryExp extends Exp {
  Exp child;
  int op;
  String toString(int d) { }
  bool check() {}
  IR translate() {}
  void print() {}
}
```

Hard to understand, maintain, and change

Visitor

Separate an algorithm from an object structure

```
class Visitor {
   void visit(ASTRoot node);
   void visit(ClassDefNode node);
   void visit(BinaryOpNode node);
   ……
}
```

```
class Printer extends Visitor {
   void visit(ASTRoot node);
   void visit(ClassDefNode node);
   void visit(BinaryOpNode node);
   ……
}
```

```
class Node {
   ……
   abstract void accept(Visitor v);
}
```

```
class BinaryExp extends Exp{
   ……
   void accept(Visitor v){
      v.visit(this);
   }
}
```

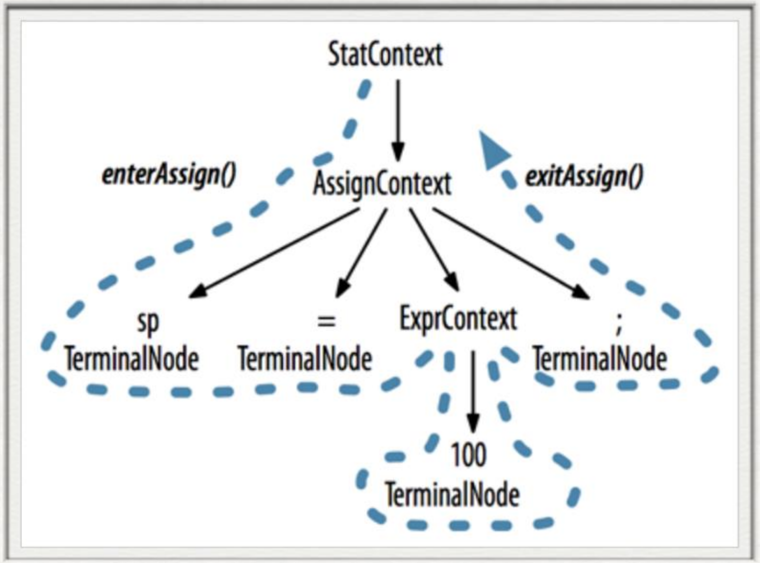
```
class UnaryExp extends Exp{
   ……
   void accept(Visitor v){
      v.visit(this);
   }
}
```

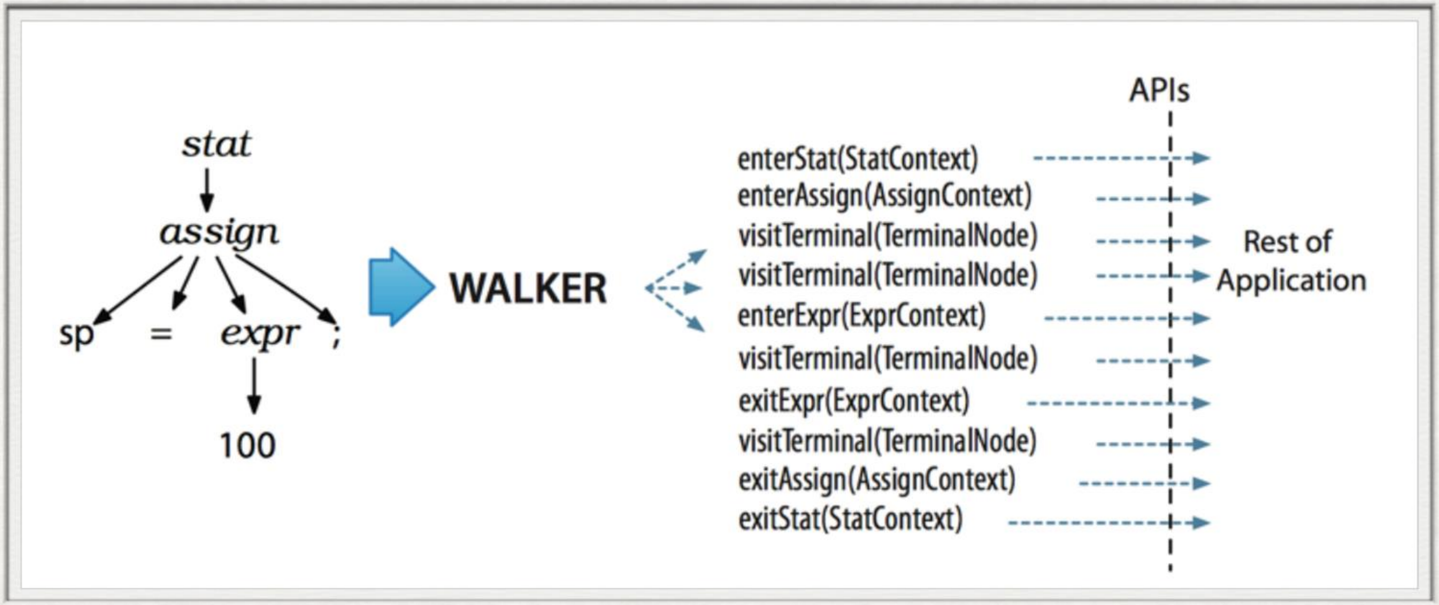
Visitor

- Specify processing methods for different types
- Walk over the tree in the correct order
- Checks the argument for each node
- Control how child nodes are visited during the walk

- Implement:
  - Pretty-Print, Scope-Building, Type-checking…
  - From CST to AST
    - Visitor
    - Listener: walker, enterNode(), exitNode()

Listener





enterStat(StatContext)
enterAssign(AssignContext)
visitTerminal(TerminalNode)
visitTerminal(TerminalNode)
enterExpr(ExprContext)
visitTerminal(TerminalNode)
exitExpr(ExprContext)
visitTerminal(TerminalNode)
exitAssign(AssignContext)
exitStat(StatContext)

Semantic Analysis

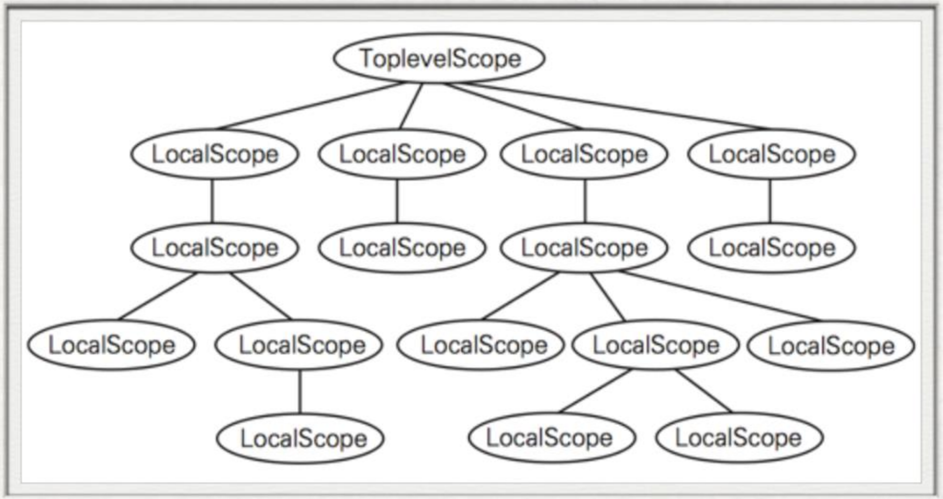
- Who am I?
  - Class? Function? Variables?
  - Static Types
- Where am I from?
  - belong to which scope
  - match identifier declarations with uses
- Can I exist?
  - Type conflict/Invalid operations

```
int x = 1;
bool b(bool x){
    return x;
}
class C{
    C(){
        x();
        {int x;}
    }
    void x(){}
}
int main(){
    string x = "dcba\n";
    int y;
    C c;
    c.x();
    b(true);
    {
        y = x.parseInt();
        int x = y;
    }
    return y;
}
```

Scope

```
int x = 1;
bool b(bool x){
    return x;
}
class C{
    C(){
        x();
        {int x;}
    }
    void x(){}
}
int main(){
    string x = "dcba\n";
    int y;
    C c;
    c.x();
    b(true);
    {
        y = x.parseInt();
        int x = y;
     }
    return y;
}
```

- the portion of a program in which the identifier is accessible
- static scope/dynamic scope



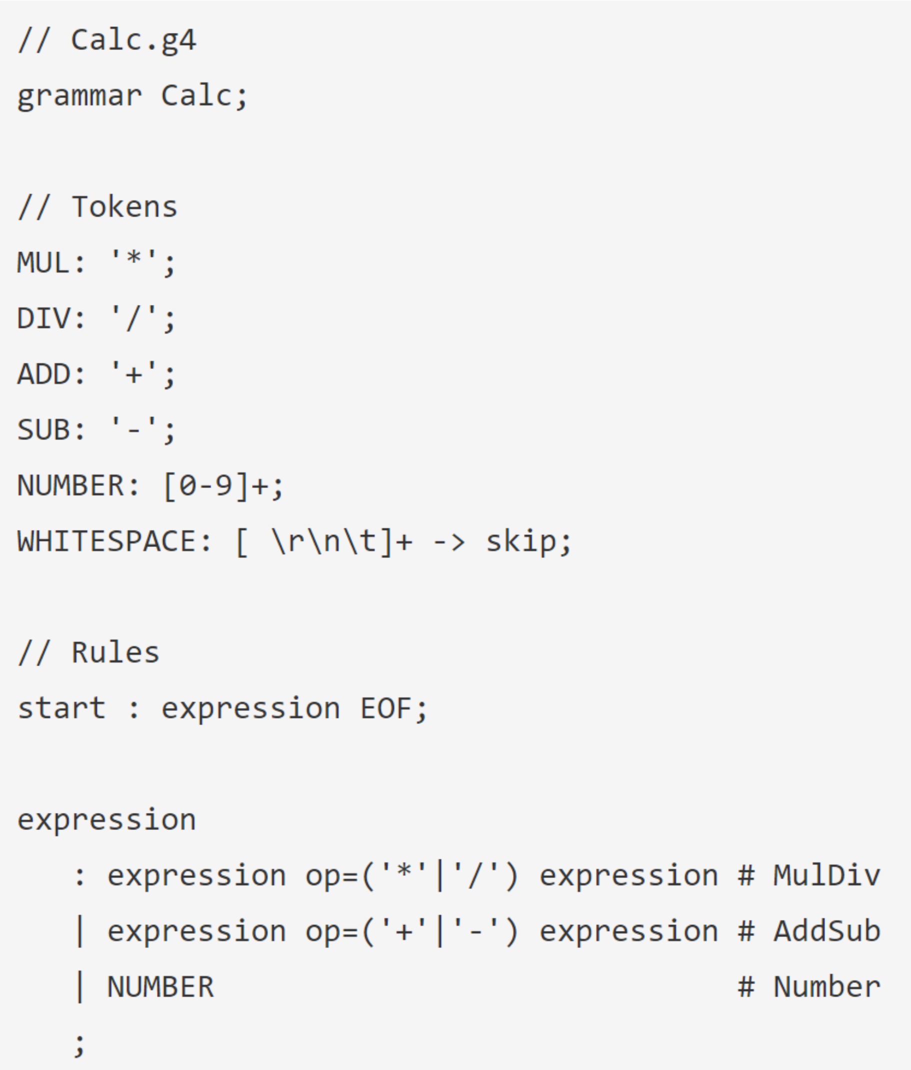
Symbol Table

- a data structure that tracks the current binding of identifiers

- name: variables, functions, types
- type: basic type, array type, class type
- scope: local, global

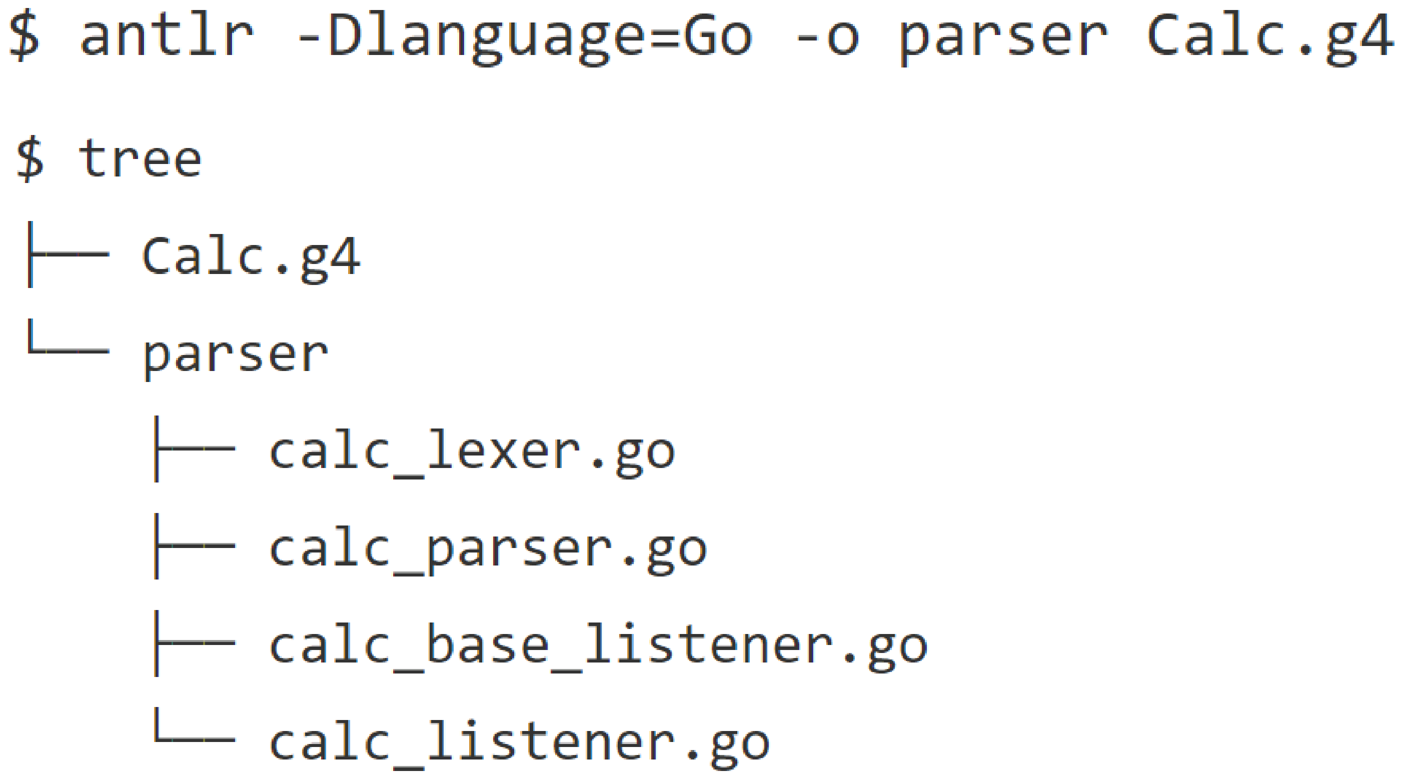
• push_scope()	start a new nested scope
• add_symbol(x)	add a symbol x to table
• find_symbol(x)	find current x
• exit_scope()	exit current scope

Classic Calculator Example



```
// Calc.g4
grammar Calc;

// Tokens
MUL: '*';
DIV: '/';
ADD: '+';
SUB: '-';
NUMBER: [0-9]+;
WHITESPACE: [ \r\n\t]+ -> skip;

// Rules
start : expression EOF;

expression
   : expression op=('*'|'/') expression # MulDiv
   | expression op=('+'|'-') expression # AddSub
   | NUMBER                             # Number
   ;
```

Code Structure



```
$ antlr -Dlanguage=Go -o parser Calc.g4
```

```
$ tree
├── Calc.g4
└── parser
    ├── calc_lexer.go
    ├── calc_parser.go
    ├── calc_base_listener.go
    └── calc_listener.go
```

Classic Calculator Example

- The Lexer takes arbitrary input and returns a stream of tokens.
- For input such as 1 + 2 * 3, the Lexer would return the following tokens: NUMBER (1), ADD (+), NUMBER (2), MUL (*), NUMBER (3), EOF.
- The Parser uses the Lexer’s output and applies the Grammar’s rules. Building higher level constructs, such as expressions that can be used to calculate the result.

Main

```go
func main() {
    // Setup the input
    is := antlr.NewInputStream("1 + 2 * 3")

    // Create the Lexer
    lexer := parser.NewCalcLexer(is)
    stream := antlr.NewCommonTokenStream(lexer, antlr.TokenDefaultChannel)

    // Create the Parser
    p := parser.NewCalcParser(stream)

    // Finally parse the expression
    antlr.ParseTreeWalkerDefault.Walk(&calcListener{}, p.Start())
}
```

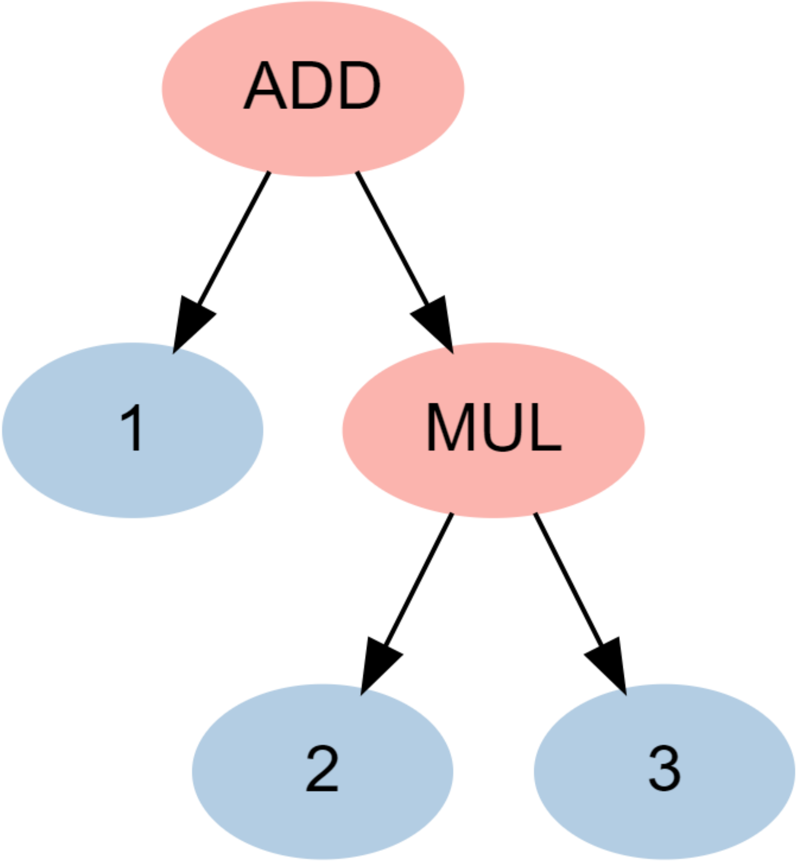
Listener

```
type CalcListener interface {
    antlr.ParseTreeListener

    // EnterStart is called when entering the start production.
    EnterStart(c *StartContext)

    // EnterNumber is called when entering the Number production.
    EnterNumber(c *NumberContext)

    // EnterMulDiv is called when entering the MulDiv production.
    EnterMulDiv(c *MulDivContext)

    // EnterAddSub is called when entering the AddSub production.
    EnterAddSub(c *AddSubContext)
```

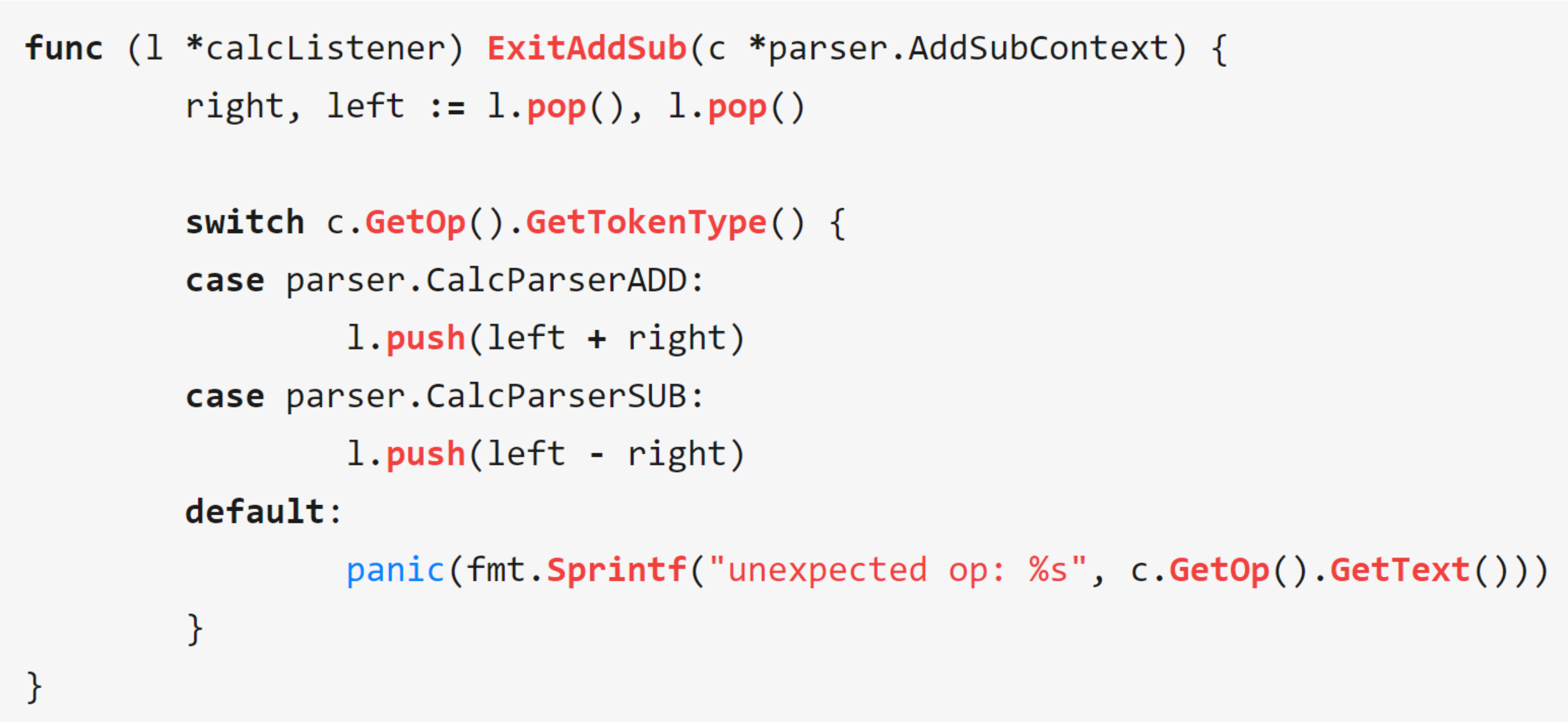
Listener

- There is an Enter and Exit function for each rule found in the grammar.
- As the input is walked, the Parser calls the appropriate function on the listener, to indicate when the rule starts and finishes being evaluated.

Adding the logic



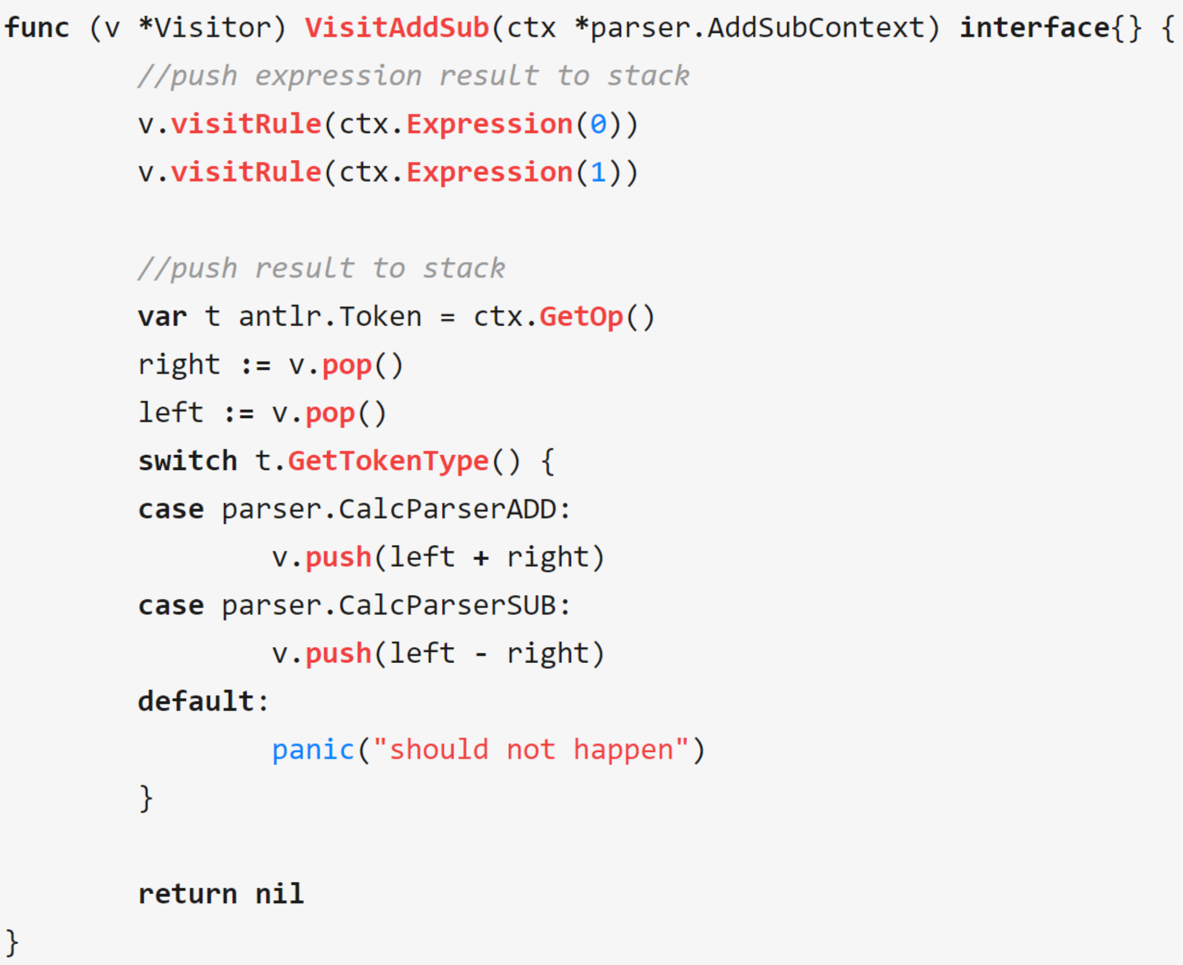
Listener Mode



```
func (l *calcListener) ExitAddSub(c *parser.AddSubContext) {
        right, left := l.pop(), l.pop()

        switch c.GetOp().GetTokenType() {
        case parser.CalcParserADD:
                l.push(left + right)
        case parser.CalcParserSUB:
                l.push(left - right)
        default:
                panic(fmt.Sprintf("unexpected op: %s", c.GetOp().GetText()))
        }
}
```

- ParseTreeProperty can help return nodes

Visitor Mode



```go
func (v *Visitor) VisitAddSub(ctx *parser.AddSubContext) interface{} {
        //push expression result to stack
        v.visitRule(ctx.Expression(0))
        v.visitRule(ctx.Expression(1))

        //push result to stack
        var t antlr.Token = ctx.GetOp()
        right := v.pop()
        left := v.pop()
        switch t.GetTokenType() {
        case parser.CalcParserADD:
                v.push(left + right)
        case parser.CalcParserSUB:
                v.push(left - right)
        default:
                panic("should not happen")
        }

        return nil
}
```

- Visit can return nodes

Reference

- https://www.antlr.org/
- https://blog.gopheracademy.com/advent-2017/parsing-with-antlr4-and-go/
- https://zhuanlan.zhihu.com/p/47179842
- 自制编译器.青木峰郎. Chapter 8, 9. (Can learn how to organize this project)