Design
Lexer && parser

Grammar

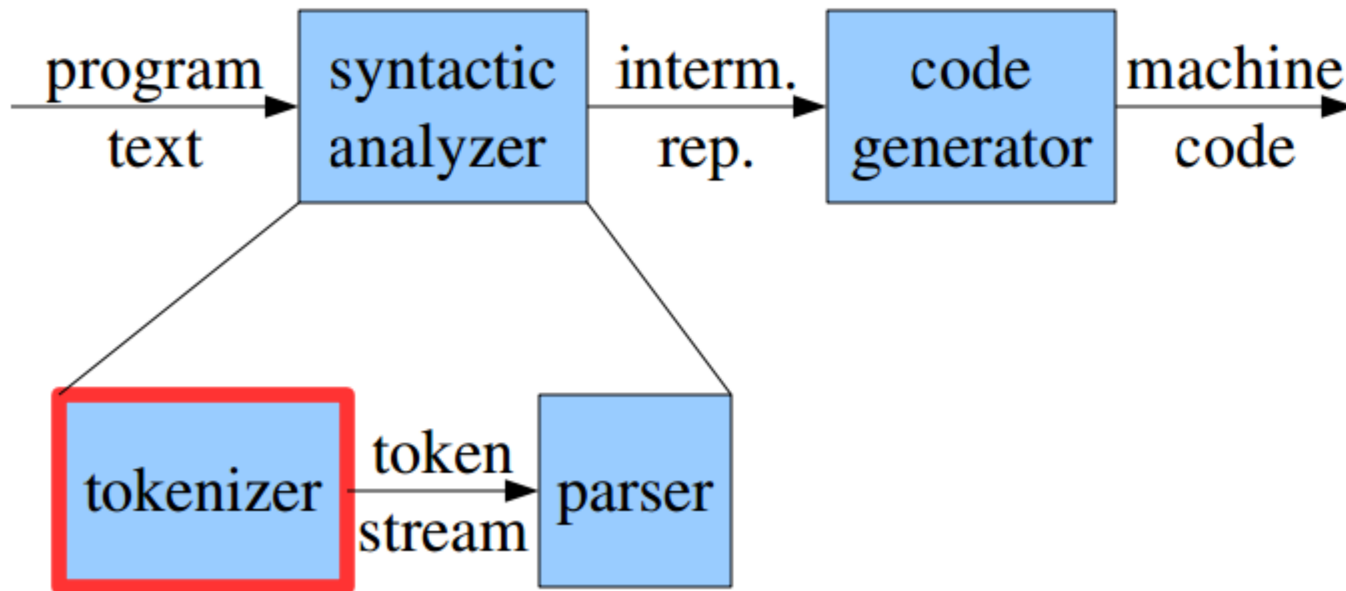
```
program -> decls stmts
block -> {decls stmts}
decls -> decls decl | ϵ
decl -> type *id*;
type -> type [*num*] | *basic* | *record*
{decls}
stmts -> stmts stmt | ϵ

stmt -> assign;
    | *if*(assign) stmt
    | *if*( assign ) stmt *else* stmt
    | *while*( assign ) stmt
    | *do* stmt *while* ( assign );
    | *break*;
    | *return*;
    | *return* loc;
    | *print* loc;
    | block
loc -> loc [assign] | *id* | loc.*id*
```

```
assign -> loc=assign | bool
bool -> bool||join | join
join -> join&&equality | equality
equality -> equality==rel | equality!=rel | rel
rel -> expr<expr | expr<=expr | expr>=expr |
expr>expr | expr
expr -> expr+term | expr-term | term
term -> term*unary | term/unary | unary
unary -> !unary | -unary | factor
factor -> (assign) | loc | *num* | *real* | *true* |
*false* | *string*
```

Grammar Terminals

```
digit -> [0-9]
digits -> digit+
integralDigits -> -? digits
exponent -> ((e | E) integralDigits)
num -> digits exponent?
real -> digits . digits exponent?
basic -> (long | double | string | bool)
alpha -> [a-zA-Z]
alphaNumeric -> [a-zA-Z0-9]
id -> alpha alphaNumeric*
```

The structure of a compiler



Lexer

- Process of converting a sequence of characters into a sequence of tokens
- Called by a parser
- Usually based on a finite-state machine
- May involve backtracking

Lexer and parser
Some interface discussions

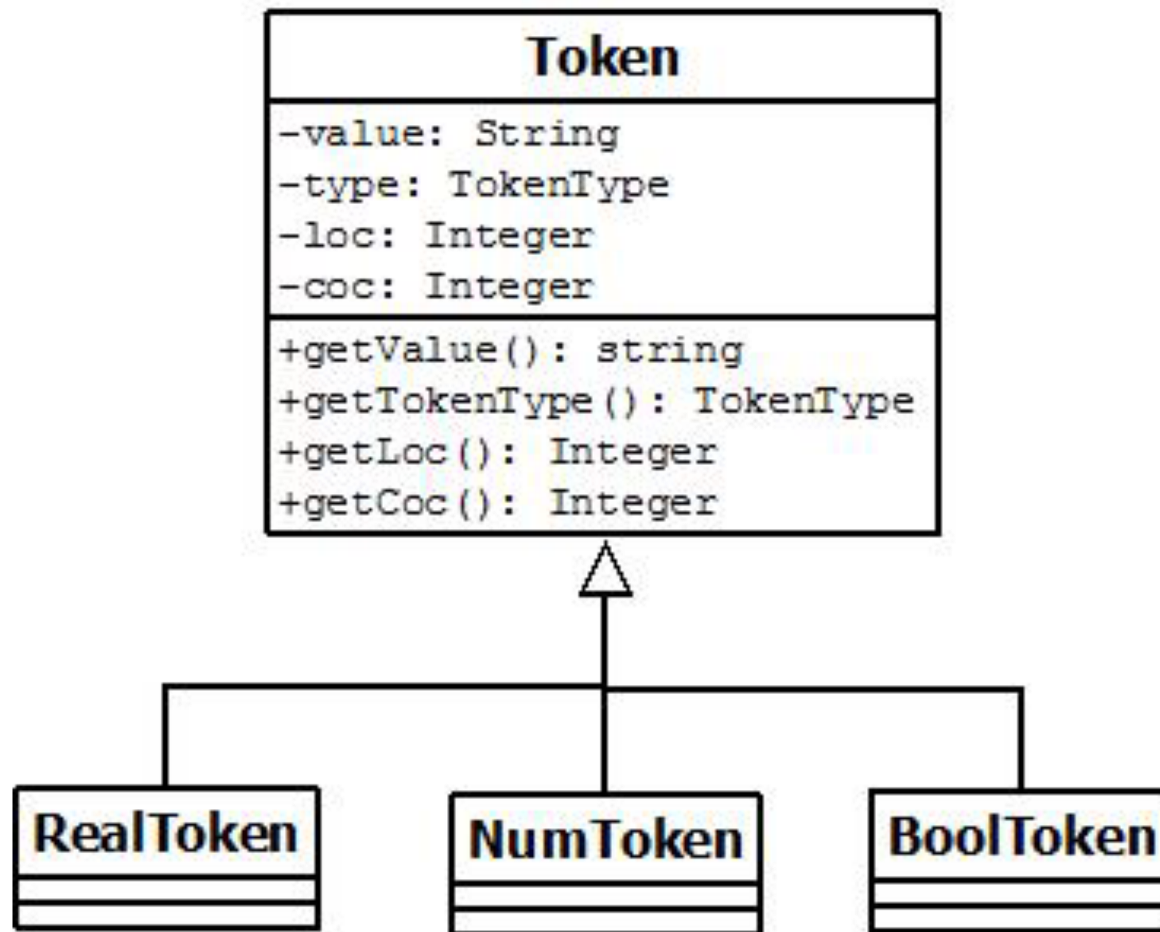
There are various options with pros and cons

e.g. Fortran 90

```
DO 5 I = 1.25
<variable, "DO5I"> <assign> <number,"
1.25">
```

```
DO 5 I = 1,25
<do> <number, "5"> <variable, "I">
<assign> <number, "1"> <comma>
<number, "25">
```

Lexer Interface



Lexer Interface

Lexer

+getNextToken(): Token

+getTokenList(): List<Token>

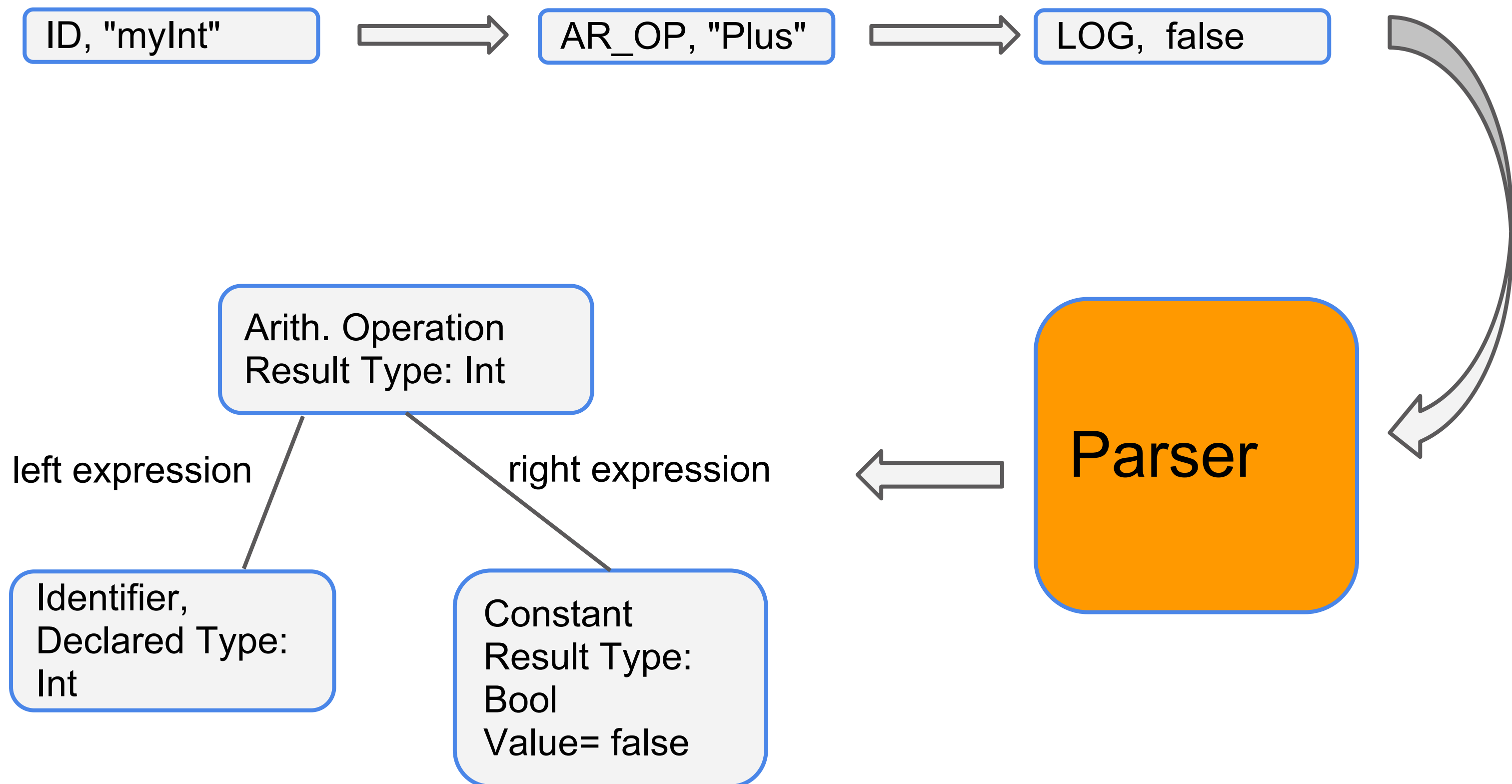
+setSymbolTable(st:SymbolTable): SymbolTable

Lexer Interface

<<enumeration>>
TokenType

NUM
REAL
TRUE
FALSE
STRING
ID
IF
ELSE
WHILE
DO
BREAK
RETURN
PRINT
ASSIGNOP
AND
OR
EQUALS
NOT_EQUALS
LESS
LESS_OR_EQUAL
GREATER
GREATER_EQUAL
PLUS
MINUS
TIMES
DIVIDE
NOT
NEGATE
LEFT_PARAN
RIGHT_PARAN
LEFT_BRACKET
RIGHT_BRACKET
LEFT_BRACE
RIGHT_BRACE
SIMICOLON

Parser Overview



Parser Tasks

Parse

- convert sequence of tokens to a sequence of derivations
- deal with ambiguousness
  - dangling else

Parser Tasks

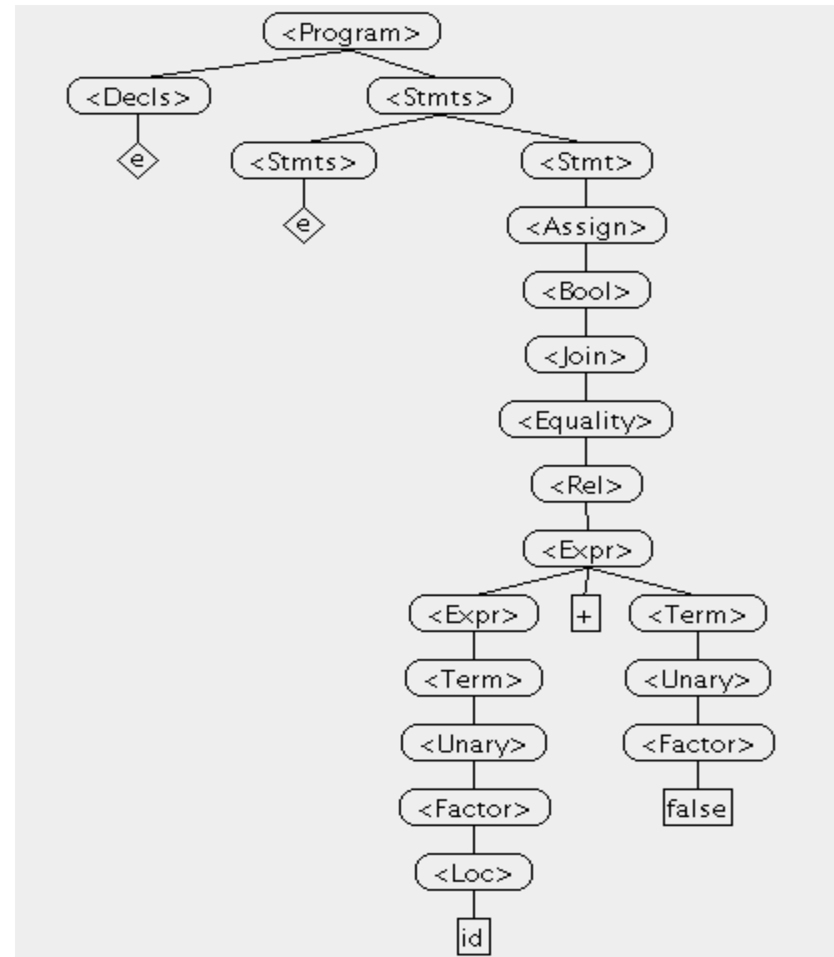
Analyze

- semantic analysis
    - Convert parse tree to AST
    - assign attributes
    - deal with circular dependencies

Parser Implementation

```
myInt + false
```

- parse

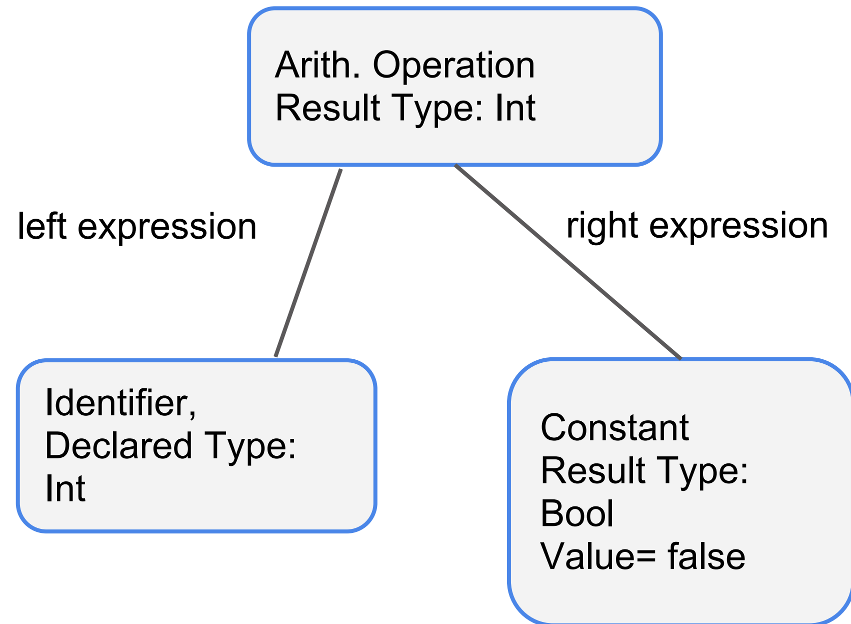


CFGExperimenter (http://www.curtclifton.net/cfg-experimenter)

Parser Implementation

```
myInt + false
```

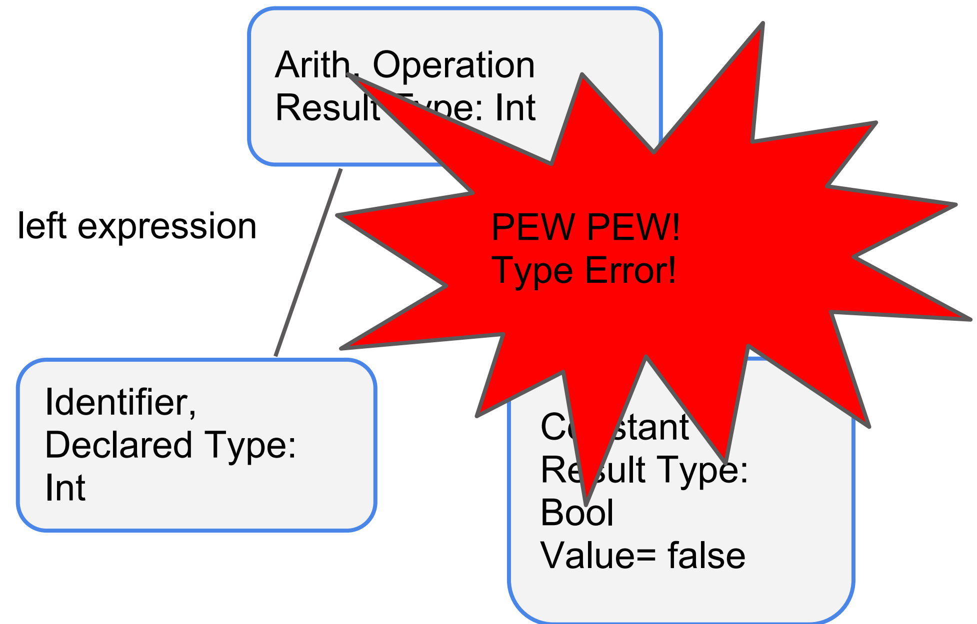
- parse
- to AST



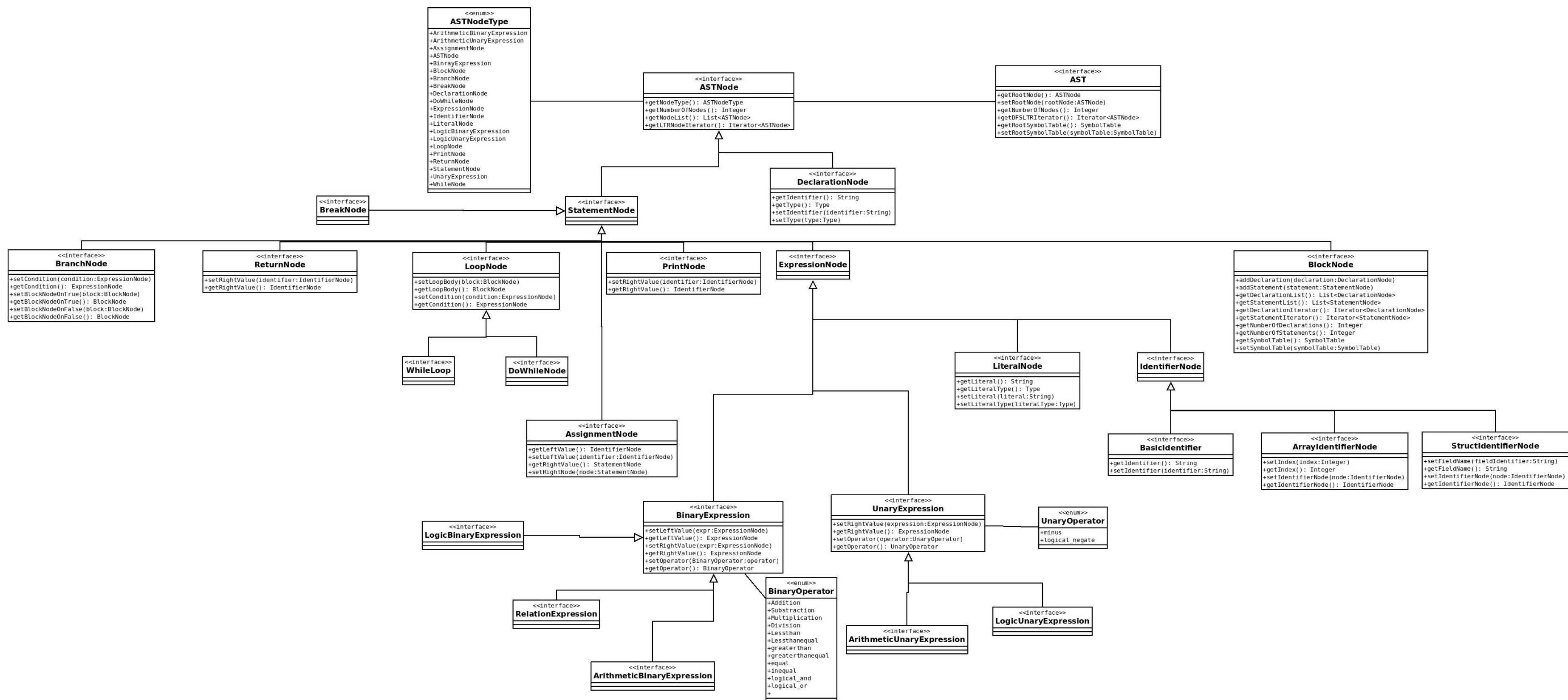
Parser Implementation

```
myInt + false
```

- parse
- to AST
- analyze semantic



Parser
To be more precise...

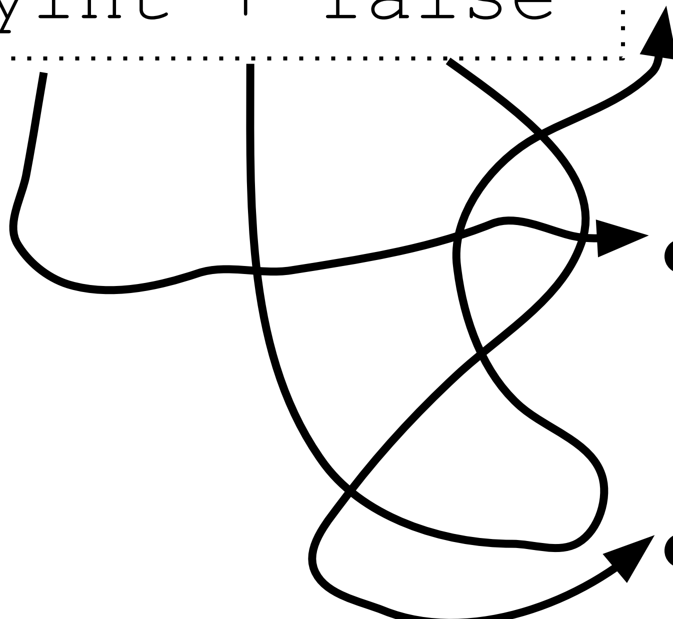


https://github.com/swp-uebersetzerbau-ss13/common/blob/master/doc/ast.jpeg

Parser
To be more precise...

- two design approaches
  - tree structure
    - ASTNode::GetNodeList()
    - reflects grammar
    - supports visualisation
  - interface only design
    - ASTNode::GetNodeType()/SetNodeType(n)
    - implementation can use an arbitrary inheritance hierarchy
    - maximum degree of freedom

Parser
Recap the situation...

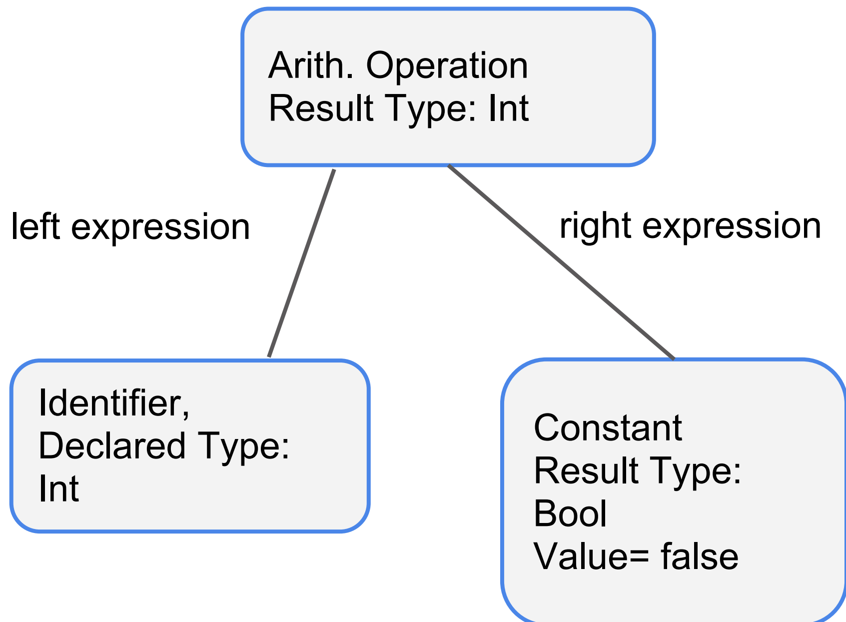


- ArithmeticBinaryExpression
  - extends ExpressionNode
    - extends StatementNode
      - extends ASTNode
- BasicIdentifier
  - extends IdentifierNode
    - extends ExpressionNode
      - ....
- LiteralNode
  - extends ExpressionNode
    - ...

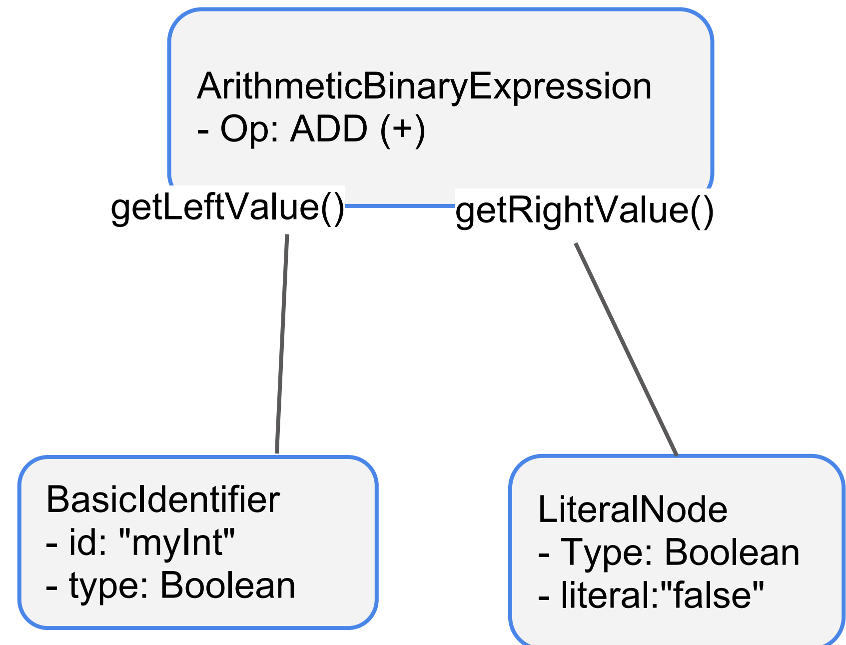
Parser
Recap the situation...

```
myInt + false
```

Idea



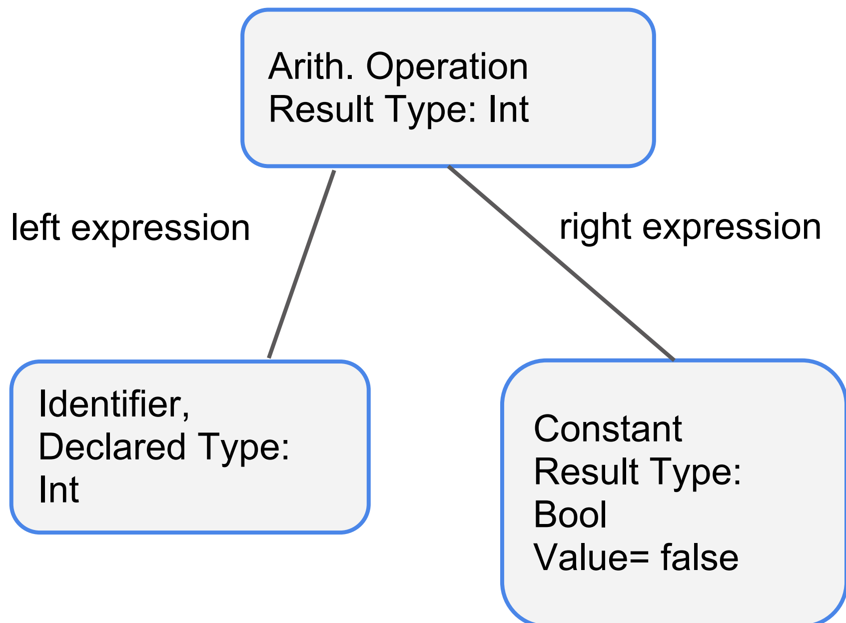
Realisation



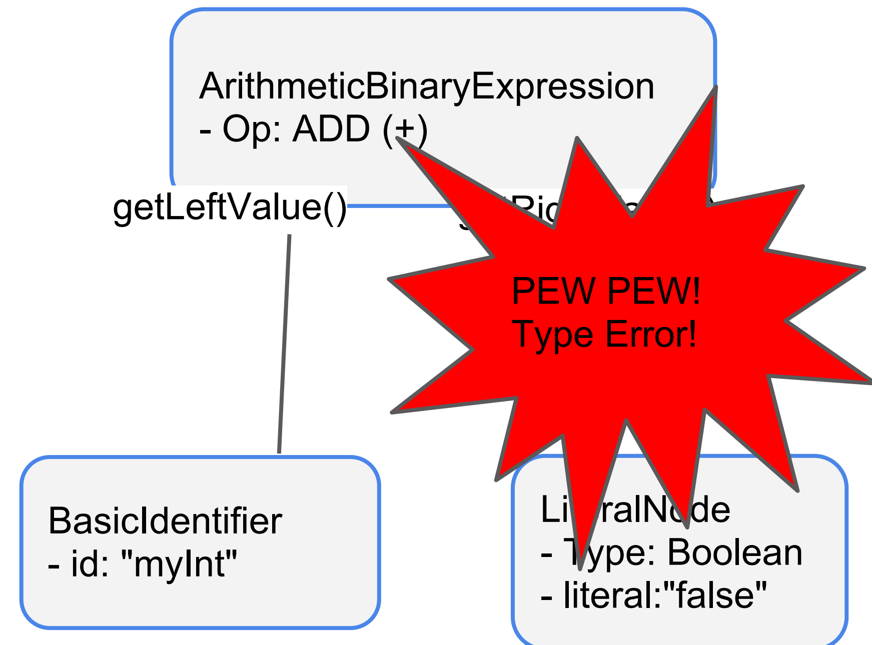
Parser
Recap the situation...

```
myInt + false
```

Idea



Realisation



Sources

- http://cg.scs.carleton.ca/~morin/teaching/3002/notes/lex.pdf
- Frau Prof. Fehr
- Aaaand special thanks go to ...
    - Florian
    - Frank